Oracle® Java SE Embedded

Developer's Guide

Release 8

E28300-05

July 2014

Documentation that describes essential concepts and common tasks for Oracle Java SE Embedded technology, for platform and application developers.

ORACLE®

Oracle Java SE Embedded Developer's Guide, Release 8

E28300-05

Contents

13 Launch Embedded Applications

14 Develop Headful Applications

15 Codecache Tuning

Part IV Appendixes

Preface

Oracle Java SE Embedded is a customizable Java Runtime Environment (JRE) plus tools. It can be used as the foundation of a wide range of embedded applications.

About This Guide

This guide is a consolidation of three guides that were available for Oracle Java SE Embedded Release 8: a Concepts Guide, a Platform Developer's Guide, and an Application Developer's Guide. Rather than having separate guides, the information is now available in a single guide. This should enhance the ability to find information related to any aspect of Oracle Java SE Embedded:

- Part I, "Quick Start"

  Shows both platform and application developers how to get up and running quickly, plus a basic overview of Oracle Java SE Embedded technology.

- Part II, "Platform Development"

  Describes how to choose the minimum Java Virtual Machine (JVM) and compact profile for the embedded device and how to create the JVM with the jrecreate tool.

- Part III, "Embedded Application Development"

  Presents information to assist with both headless and headful application development.

Audience

Refer to the portions of the guide that pertain to your role as a platform developer or application developer.

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.

Related Documents

- *Oracle Java SE Embedded Release Notes*

Conventions

The following text conventions are used in this document:

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

Part I

Quick Start

Choose the best quick start guide for you, depending on whether you are a platform developer or an application developer. If your plan to install Oracle Java SE Embedded on your embedded device and develop applications for it, look at both quick start guides.

- Introducing Oracle Java SE Embedded
- Quick Start for Platform Developers
- Quick Start for Application Developers

1

Introducing Oracle Java SE Embedded

Learn the advantages of using Oracle Java SE Embedded technology on your devices, and which of the Java SE Embedded documents will be most relevant to accomplishing your goals.

This page contains the following topics:

Embedded Systems: The Wave of the Future

Embedded systems are computer-based but unlike desktop computers and their applications. An embedded system's computer is embedded in a device. The variety of devices is expanding daily.

ATMs
- Parking Meters
- POS Systems
- Lottery/Gaming Systems
- Multi Function Printers
- Intelligent Power Module
- Netbooks

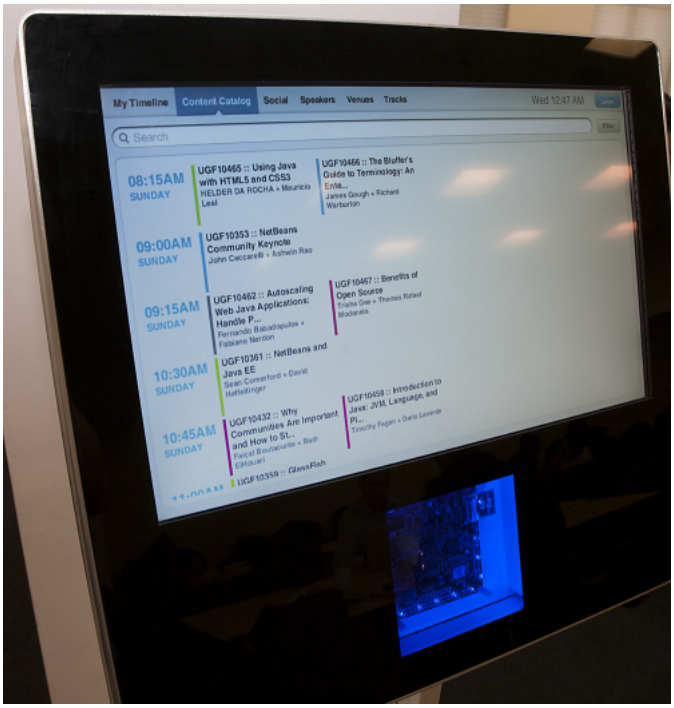
Routers & Switches
- Storage Appliances
- Network Management Systems
- Medical Imaging Systems
- Radar Systems
- Industrial PCs
- Factory Automation Systems
- Geo-Imaging Devices

Smart Meters
- RFID Readers
- Video Conferencing Systems
- In-Flight Entertainment Systems
- Video Streaming Systems
- Electronic Voting Systems
- Voice Messaging Systems
- Security Systems

Many embedded systems are headless, meaning they have no conventional user interface: no keyboard, no mouse, and no display. These systems respond to sensor inputs, such as thermometers and accelerometers. They respond by adjusting actuators, such as indicating alarms or sending messages. A headless system that is connected to a network can run a web server and provide a web-based user interface.

As hardware costs decrease, headful embedded systems are increasingly common. Although users interact with such systems, typically by means of buttons or touch-sensitive displays, the devices do not look like, or behave like, a desktop computer. Figure 1–1 shows an example of a headful embedded system. In this example, for demonstration purposes, the embedded computer is visible through a window in the case.

Figure 1–1 Headful Embedded System Example: A Kiosk



Embedded systems typically have attributes that make them quite different from desktop applications:

- Their functions are fixed at the factory; they are dedicated, not general purpose. Their workload is predictable.
- They use custom hardware (sensors and actuators).
- They are carefully optimized in multiple dimensions to use the least computational resources that will meet requirements.
- They do not tolerate malfunction. Usually, an embedded system cannot be fixed after it is deployed.
- With few resources, special purpose hardware, and high demand for correct operation on day one, they are hard to develop software for.

Why Use Java for Your Embedded System?

Embedded system applications were once written in assembly language. But as embedded computer hardware resources grew, pressures such as time-to-market and development cost drove a shift to the more portable and less error-prone C and C++ languages. These languages required more memory and more CPU cycles, but declining hardware costs and rising software complexity made the trade worthwhile.

Continuing this progression, Moore's Law and other factors have more recently made Java increasingly attractive for embedded system development. These other factors include a worldwide population of Java developers who have trained in enterprise and other non-embedded domains. This talent and knowledge can be extended to embedded systems by developing their application code in the Java programming language.

Java is:

- A modern, object-oriented, language without the error-inducing complexity of C++. Objects are natural representations for sensors and actuators.
- Less prone to errors than C. For example, there are no pointers, and memory management is automatic.
- Highly portable; Java classes do not need to be recompiled to run on a different CPU or operating system.
- Security-oriented. Java libraries support encrypting sensitive data sent to or from embedded devices, and validating digitally signed code downloaded to update or extend embedded applications in the field.
- Multi-threaded, enabling the natural expression of parallel activities and their simultaneous execution on platforms whose operating system thread model takes advantage of multiple CPU cores.
- Equipped with a large collection of OS-independent libraries including database access and graphical user interfaces.
- Tunable to match hardware resources and application needs. For example, there are multiple options for runtime compilation to native instructions, and for how and when unused (garbage) memory is reclaimed.
- Extensible with native methods written in C that interact with special purpose embedded system hardware.
- Debuggable on the desktop and remotely. A desktop Java Runtime Environment (see Which Java Embedded is Right for Your Device?) has the same APIs as one that runs on an embedded computer, except for hardware-specific interfaces and behaviors. Most functional debugging can be done on a desktop computer. An embedded system that has a network connection can be debugged and profiled remotely.

Which Java Embedded is Right for Your Device?

Oracle offers three Java Runtime Environment product families:

- Java Platform, Standard Edition (Java SE): For Macintosh, Windows, Linux, and Solaris desktop and server class computers. Hardware resources for JREs and Java applications are rarely an issue on these computers. For more information, see `http://www.oracle.com/technetwork/java/javase/overview/`
- Oracle Java SE Embedded: For embedded systems with tens of megabytes of memory for a JRE and Java applications, with or without graphical user interface hardware. Oracle Java SE Embedded is described in this document.
- Oracle Java Micro Edition Embedded (Oracle Java ME Embedded): For headless embedded systems on devices that have a megabyte or less of memory for a JRE and applications. For more information, see `http://www.oracle.com/technetwork/java/embedded/overview/javame/`

A Java Runtime Environment (JRE) enables and supports the safe execution of portable Java program instructions (bytecodes) on a particular CPU and operating system. A typical JRE has several components, including:

- A JRE's Java Virtual Machine (JVM) verifies and translates bytecodes into CPU instructions and arranges for them to be executed.
- A JRE's memory manager interacts with the computer's operating system to allocate dynamic (heap) memory and automatically reclaim it (collect garbage).
- A JRE's application programming interfaces (APIs) provide services, such as file systems, database access, and graphical user interfaces. These services are sometimes implemented on corresponding operating system services, but insulate Java applications from operating system dependencies.
- A JRE's resource files store data such as time zones, fonts, and locales.
- A JRE's launcher is an operating system command that starts a JRE running a Java application.

Although by definition, JREs must be similar to make Java application code portable, they can differ in composition. For example, different virtual machine designs can take advantage of resources available on some computers but not on others.

The Heart of Java SE Embedded: Customize Your Runtime Environment

The Java Runtime Environment (JRE) consists of a Java Virtual Machine (JVM) plus Libraries and Toolkits.

Oracle Java SE Embedded is a set of components from which you can build a custom JRE that meets your application's functional requirements without sacrificing memory to unneeded JRE features. All custom JREs are functional subsets of the Oracle Java SE JRE, which means that you can develop and functionally test embedded application code on desktop computers, except for classes that depend on the presence of special-purpose embedded hardware. You can also reuse application classes in desktop, server, and embedded applications.

You use the `jrecreate` tool to select the size of the JVM to be installed and the profile to be included as components of the custom JRE. The JVM you choose depends on performance characteristics, and the profile depends on the libraries used by the applications that will run on the device. As you develop applications, you can use the `jdeps` tool to determine the minimum profile required for your application.

2

Quick Start for Platform Developers

Learn how to get your embedded devices running with the minimal JRE required for your device, plus the embedded applications you plan to run.

This chapter contains the following topics:

Introduction

Java SE Embedded has the tools to enable you to install custom JREs on the target device. The job of a platform developer is to determine which JRE components are required for the device and for the applications that will run on the device. See About Custom JREs.

Quick Start Example

This section will walk you through a simple example of how to set up a JRE on your embedded device and deploy your Java applications.

Let's suppose that you have a typical setup:

- A host computer running Linux
- An embedded device with Linux installed
- An Oracle Java SE Embedded bundle that matches the embedded device's hardware and operating system

You want to put a JRE on your embedded device that will have the correct compact profile and JVM to run the two example headless applications shown here.

Here are the basic tasks that a platform developer would be likely to face in deciding which JRE to install on their embedded device.

- Task 1, "Install Oracle Java SE Embedded on your host computer"
- Task 2, "Determine which profile to use for your applications"
- Task 3, "Determine which JVM to use"
- Task 4, "Use jrecreate to create the JRE"
- Task 5, "Deploy the JRE to the embedded device"
- Task 6, "Deploy your embedded applications to the device"

The next section shows a simple example of how to accomplish these tasks.

Note: In most cases, embedded devices run headless applications (no user interface). The Hello World and Hello RMI applications used as examples in this guide are headless applications.

Headful applications (with a UI) are developed using the JavaFX APIs in the Java Development Kit (JDK). If you are working with headful applications, see the appendix Preparing a Device for JavaFX.

This scenario is very simple. For more information and more options, see the More Information links at the end of each section.

Task 1 Install Oracle Java SE Embedded on your host computer

Download and set up Oracle Java SE Embedded on a host computer.

1. Download the Oracle Java SE Embedded bundle from http://www.oracle.com/technetwork/java/embedded/embedded-se/downloads/index.html
2. Extract the bundle from the tar file:

```
$ cd download
$ gunzip *.gz
$ tar -xvf *.tar

List of unpacked files ...
```

3. Set the following environment variables:
   - Set `EJDK_HOME` to `download/ejdk1.8.0_06`
   - If not already set, define value of `JAVA_HOME` as the path of the local Java SE installation.
4. Verify the installation:

```
$ cd $EJDK_HOME
$ bin/jrecreate.sh --help
Usage: jrecreate --help

Summary of jrecreate syntax ...
```

More Information:

- Install Oracle Java SE Embedded

Task 2 Determine which profile to use for your applications

Use the Java `jdeps` tool across all of your platform applications to determine the minimal compact profile to install on your device: compact1, compact2, or compact3.

You can use the `jdeps` command to determine the minimal compact profile you can use. To illustrate the use of the `jdeps` command, let's take two simple headless application examples: Hello World and Hello RMI.

In the Hello World example, shown in Example 2–1, the application merely prints "Hello World" to the console.

Example 2–1 Hello World Application

```
public class HelloWorld {
  public static void main (String args[]) {
```

```
    System.out.println("Hello world!");
  }
}
```

In the Hello RMI example, shown in Example 2–2, the application uses the RMI `LocateRegistry` class to access a remote object and print a stack trace if there is an error.

Example 2–2 Hello RMI Application

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloRMI {
  public static void main (String args[]) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new SecurityManager());
    }
    try {
      Registry registry = LocateRegistry.getRegistry("testRMI");
      System.out.println("Hello RMI!");
      } catch (Exception e) {
        System.err.println("RMI exception:");
          e.printStackTrace();
      }
  }
}
```

Let's also assume that both of your applications have been compiled. If you want to compile either of the example applications, the compile commands are shown in Example 2–3.

Example 2–3 Compile Commands for Hello World and Hello RMI

```
% javac HelloWorld.java
% javac HelloRMI.java
```

Now you can use the `jdeps` tool to determine the minimum compact profile required to run each application. Example 2–4 show the command and results for the Hello World application.

Example 2–4 jdeps Command and Results for the Hello World Application

```
% jdeps -P HelloWorld.class

HelloWorld.class ->
/net/test11.us.example.com/export/java-re/jdk/8/ea/b124/binaries/linux-i586/jre/li
b/rt.jar
   <unnamed> (HelloWorld.class)
      -> java.io                                            compact1
      -> java.lang                                          compact1
```

The results in Example 2–4 show that the compact1 profile is the minimal profile required to run the Hello World application.

Example 2–5 shows the command results for the Hello RMI application.

Example 2–5 jdeps Command and Results for the Hello RMI Application

```
% jdeps -P HelloRMI.class
```

```
HelloRMI.class ->
/net/test11.us.example.com/export/java-re/jdk/8/ea/b124/binaries/linux-i586/jre/li
b/rt.jar
   <unnamed> (HelloRMI.class)
      -> java.io                                          compact1
      -> java.lang                                        compact1
      -> java.rmi.registry                                compact2
```

The results in Example 2–5 show that the `java.rmi.registry` class in the Hello RMI application minimally require the compact2 profile.

More Information:

- About Compact Profiles

Task 3 Determine which JVM to use

Choose which version of the JVM you want to install, based on which performance characteristics are most important: minimal, client, or server.

Choose the JVM based on which performance characteristics are most important for running your applications. In a nutshell, the three Java SE Embedded JVMs have the following performance characteristics:

Table 2–1 JVM Versions and Their Primary Performance Characteristics

JVM Version	Primary Performance Characteristics
Minimal	Minimal memory footprint
Client	Responsiveness
Server	Tuned for long-running applications

In this example, let's try out the minimal JVM, which has the smallest footprint, and upgrade to a higher JVM if we are not happy with performance.

More Information:

- About Oracle Java SE Embedded JVMs

Task 4 Use jrecreate to create the JRE

Since we have chosen the profile and the VM to install, we can now install the JRE to a temporary directory on the host computer using the `jrecreate` command, which is included in the `bin` folder of the extracted Java SE Embedded bundle.

Run the following command on the host computer:

```
% download/ejdk1.8.0_06/bin/jrecreate.sh \
     --profile compact1 \
     --dest /tmp/defaultJRE/
```

More Information:

- Create Your JRE with jrecreate

Task 5 Deploy the JRE to the embedded device

To deploy the JRE, simple copy the JRE files that you created with the jrecreate command over to the embedded device. How you copy depends on how whether you use the network or an SD card, but here is an example of a copy command:

```
scp -r /tmp/defaultJRE/* root@target:/opt/local/ejdk1.8.0_06/
```

More Information:

- Deploy Your JRE to the Embedded Device

Task 6 Deploy your embedded applications to the device

If you have compiled the Hello World and Hello RMI applications, you can try copying them over and running them on your embedded device. Or, you can try copying over an application of your own.

More Information:

- About Connecting to the Target Device

3

Quick Start for Application Developers

Learn how to test your applications so they will run with the compact profile that matches the JRE installed on the embedded device.

This page contains the following topics:

Typical Tasks for Embedded Application Developers

Here are the basic tasks that embedded application developers use to develop an application for an embedded device.

> **Note:** If you have a device or your own that you need to set up with Java SE Embedded, see Quick Start for Platform Developers.

Quick Start Examples

The tasks in the following sections will walk you through a simple example of how to develop applications for an embedded device. These instructions assume the following conditions:

- You will work on a host computer for application development. You can work with a supported, standard JDK installation, using your regular development tools. Download the standard JDK from. `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- You have a basic knowledge of the Java programming language and Java SE technology.
- You know which compact profile is installed, or will be installed, on the embedded device as part of the JRE.

For basic information about compact profiles, see About Compact Profiles.

This quick start scenario is very simple. For more information and more options, see the More Information links at the end of each section.

Tip: NetBeans IDE provides support for profiling, running, and debugging applications on both the host computer and target device. See Developing Embedded Applications in NetBeans IDE.

Task 1 Develop Your application

Develop your application using your regular Java development tools and the standard JDK. The difference from standard development is that you will frequently test your application to ensure the libraries in your application meet the requirements of the compact profile installed on the device.

To illustrate the process, let's take two simple headless application examples: Hello World and Hello RMI.

In the Hello World example, shown in Example 3–1, the application merely prints "Hello World" to the console.

Example 3–1 Hello World Application

```
public class HelloWorld {
  public static void main (String args[]) {
    System.out.println("Hello world!");
  }
}
```

In the Hello RMI example, shown in Example 3–2, the application uses an RMI `LocateRegistry` class to access a remote object and print a stack trace if there is an error.

Example 3–2 Hello RMI Application

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloRMI {
  public static void main (String args[]) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new SecurityManager());
    }
    try {
      Registry registry = LocateRegistry.getRegistry("testRMI");
      System.out.println("Hello RMI!");
      } catch (Exception e) {
        System.err.println("RMI exception:");
          e.printStackTrace();
      }
  }
}
```

More Information:

- Host Development for a Target Device
- About Compact Profiles
- How compact profile information is displayed in the APIs: APIs and Javadocs
- Talk to the platform developer if you plan to use more than the basic set of character encodings.

Note: In most cases, embedded devices run headless applications (no user interface). If you are working with headful applications, see Develop Headful Applications.

Task 2 Test your application frequently

As you develop your application, test it frequently to make sure the libraries use fit the compact profile that is planned for the device. See Your Choices When Creating a Custom JRE.

To test, first compile the application, then run the `jdeps` command. For more complex applications, you can additionally test the application on the target device.

In order to use the `jdeps` tool, you must compile your application. As shown in Example 3–3, use `javac` without the `-profile` option, because the purpose is to determine the profile.

Example 3–3 Compile Commands for Hello World and Hello RMI

```
% javac HelloWorld.java
% javac HelloRMI.java
```

Now you can use the `jdeps` tool to determine the minimum compact profile required to run each application. Example 3–4 show the command and results for the Hello World application.

Example 3–4 jdeps Command and Results for the Hello World Application

```
% jdeps -P HelloWorld.class

HelloWorld.class ->
/net/test11.us.example.com/export/java-re/jdk/8/ea/b124/binaries/linux-i586/jre/li
b/rt.jar
   <unnamed> (HelloWorld.class)
      -> java.io                                            compact1
      -> java.lang                                          compact1
```

The results in Example 3–4 show that the compact1 profile is the minimum profile required to run the HelloWorld application.

Example 3–5 shows the `jdeps` command results for the Hello RMI application.

Example 3–5 jdeps Command and Results for the Hello RMI Application

```
% jdeps -P HelloRMI.class
HelloRMI.class ->
/net/test11.us.example.com/export/java-re/jdk/8/ea/b124/binaries/linux-i586/jre/li
b/rt.jar
   <unnamed> (HelloRMI.class)
      -> java.io                                            compact1
      -> java.lang                                          compact1
      -> java.rmi.registry                                  compact2
```

The result in Example 3–5 show that a `java.rmi.registry` class in the Hello RMI application requires the compact2 profile, so compact2 would be the minimum profile that would need to be installed.

More Information:

- Use the jdeps Tool to Test Minimum Required Compact Profile

Task 3 Package and deploy your application

Compile the final application using the `-profile` option, as in the example in Specifying the Profile in the javac CommandExample 3–6.

Example 3–6 Specifying the Profile in the javac Command

```
$ javac -profile compact1 HelloWorld.java
```

Package the application by creating a JAR file in the normal fashion.

To deploy the application, simply copy the JAR file and any resource files to the embedded device.How you copy to the device depends on how whether you use the network or an SD card.

> **Tip:** For larger applications, it is a best practice to test the application on the embedded device in iterative steps to see how the application performs. In this way you can make performance improvement changes to the application in the early stages of the development of teach component.

More Information:

- Compile with the javac Tool and the -profile Option
- Testing Your Application on the Target Device
- Launch the Application with the java Launcher Tool

Part II

Platform Development

This part of the guide provides information for anyone who wants to create a custom JRE with Oracle Java SE Embedded tools and install it on an embedded device.

Part I contains the following chapters:

- Install Oracle Java SE Embedded
- About Custom JREs
- About Oracle Java SE Embedded JVMs
- About Compact Profiles
- Create Your JRE with jrecreate
- Deploy Your JRE to the Embedded Device

See also the Quick Start for Platform Developers to get an overview of the process.

4

Install Oracle Java SE Embedded

This chapter describes how to unpack and set up an Oracle Java SE Embedded installation bundle and the contents of the unpacked directories on a host computer.

> **Tip:** For a quick overview of the process of installing Java SE Embedded on a host, creating a custom JRE with jrecreate, and moving the JRE to the target device, see Quick Start for Platform Developers.

The instructions in this chapter are for Linux hosts. Users of Windows hosts must make adjustments as is common when working with products that have a Unix background. Some examples:

- File path and environment variable names must be adjusted.
- The `jrecreate` command is a `.bat` file rather than the `.sh` file shown in examples.
- 7-zip or an equivalent utility can be substituted for `gunzip` to unpack an Oracle Java SE Embedded bundle.

> **Note:** Oracle Java SE Embedded is a modular system that must be configured before launching. You must use the `jrecreate` command to select the runtime components (such as APIs and virtual machines) that are appropriate for the functional and performance needs of your devices and applications. See Your Choices When Creating a Custom JRE. For details about how to create an embedded JRE, see Create Your JRE with jrecreate.

This chapter contains the following topics:

Why Install on a Host Computer?

Oracle Java SE Embedded does not have a JRE that you can install on an embedded device out of the box. You must build a custom JRE with the `jrecreate` tool on a host computer, then copy the JRE to the embedded device.

> **Tip:** If you are an application developer and do not plan to create a custom JRE and install it on an embedded device yourself, you do not need to install Oracle Java SE Embedded on a host computer. Use a supported version of the Java Development Kit (JDK) to develop your applications on a host computer. See the chapters in *Embedded Application Development*.

This chapter contains information about installing Oracle Java SE Embedded on your host computer.

Once you have finished installing Oracle Java SE Embedded on your host computer, the next steps are:

- *Create Your JRE with jrecreate*
- *Deploy Your JRE to the Embedded Device*

Prerequisites for the Host Computer

Before installing Java SE Embedded on your host computer, it must meet the following conditions.

- A supported version of the Java SE Development Kit (JDK) or the Java Runtime Environment (JRE) is installed.
- The JDK or JRE directory is searched before any other Java installation on the host. You can use the `PATH` environment variable or another method, such as a symbolic link.
- The `JAVA_HOME` environment variable names the same JDK or JRE directory.
- You have downloaded an Oracle Java SE Embedded bundle that matches the target platform's hardware and operating system. Download the Oracle Java SE Embedded bundle from `http://www.oracle.com/technetwork/java/embedded/embedded-se/downloads/index.html`

  If you plan to develop for multiple target platform types, you need a corresponding bundle for each.

Install Java SE Embedded on the Host Computer

Oracle Java SE Embedded is bundled in a compressed tar file that, when uncompressed, creates a directory called `ejdk<version>`. Subdirectories contain the `jrecreate` command and the Oracle Java SE Embedded components.

Copy the bundle to a host directory of your choice (*`installDir`*), and set it up as follows:

1. Extract:

```
$ cd installDir
$ gunzip *.gz
$ tar -xvf *.tar

List of unpacked files ...
```

2. Remove the tar file.
3. Set the `EJDK_HOME` environment variable to *`installDir`*`/ejdk<version>`.

4. Verify the installation by displaying the `jrecreate` command help:

```
$ cd $EJDK_HOME
$ bin/jrecreate.sh --help
Usage: jrecreate --help

Summary of jrecreate syntax ...
```

Java SE Embedded Installed Directories

Extracting creates the directory *installDir*`/ejdk<`*version*`>/`, which contains the following subdirectories:

- `bin/`

  Contains the `jrecreate` script that you use to create a custom JRE for the target.

- `doc/`

  Reserved for possible future use.

- `lib/`

  Contains the `jrecreate` command implementation.

- *target*:

  The name of this *target* directory represents the operating system and CPU architecture of your embedded device. The directory contains the components that the `jrecreate` command uses to compose a custom JRE, as described in Create Your JRE with jrecreate.

5

About Custom JREs

Learn about the basic concepts that underlie the custom JREs that you create with Oracle Java SE Embedded technology.

This page contains the following topics:

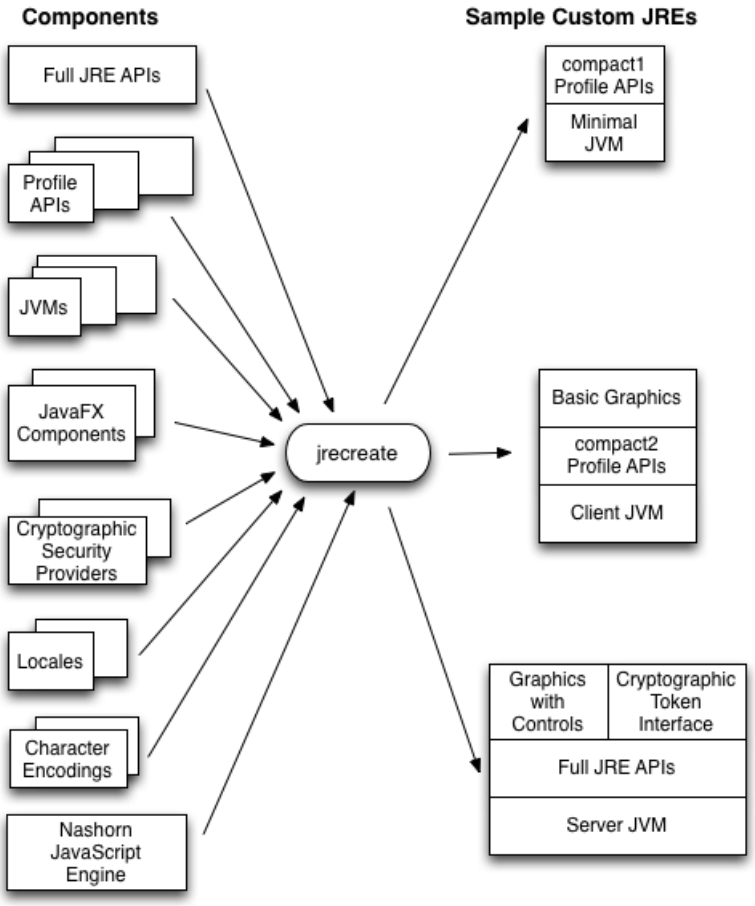
Your Choices When Creating a Custom JRE

Space optimizations make a full JRE created with Oracle Java SE Embedded smaller than the functionally equivalent out-of-the-box Java SE JRE.

In addition, you can create even smaller JREs in steps down to about 12 MB of static footprint. Using the `jrecreate` command, you can choose among components to build a custom JRE that matches expected workloads and response goals.

Figure 5–1 shows the components that go into a custom JRE, and how JREs of varying function and size can be created from the Oracle Java SE Embedded components.

Figure 5–1 jrecreate Composes Custom JREs from Components



You can find out more about the Oracle Java SE Embedded JRE components shown on the left in Figure 5–1 by following these links:

- Full JRE APIs
- Profile APIs: see About Compact Profiles
- JVMs: see About Custom JREs
- JavaFX Components: JavaFX Components for Oracle Java SE Embedded
- Cryptographic Service Providers
- Locales
- Character Encodings
- Nashorn JavaScript Engine

The right side of Figure 5–1 shows examples of custom JREs, created with the `jrecreate` tool included with Oracle Java SE Embedded. In this diagram, the simplest JRE consists of a minimal JVM and the compact1 profile APIs. More complex is a client JVM, compact2 profile APIs, and basic graphics. Largest and most capable is a server JVM, full JRE APIs, graphics with controls, and a cryptographic token interface.

Cryptographic Service Providers

For cryptographic operations, the Java language defines standard APIs for application developers, which are implemented by replaceable service providers.

Oracle Java SE Embedded includes two optional provider packages. Choose either or both, as your application requires. See Create Your JRE with jrecreate for how to use the `jrecreate` tool to add provider packages. The packages are:

- Cryptographic Token Interface Standard PKCS#11 package, by RSA Security. This package is for devices that have smart cards or hardware security modules. This extension increases the static size of a JRE by about 234 KB.
- Elliptic Curve Cryptography (ECC) package, an alternative to the RSA public key standard. This extension increases the static size of a JRE by about 36 KB.

Locales

By default, the `jrecreate` command creates the US English (en_us) locale. Optionally, it creates all Java SE locales. For information about Java SE locales, see the list of supported locales for JRE. To add optional locales to your JRE, see the `--extension` option in jrecreate Options.

Character Encodings

By default, the `jrecreate` command creates the basic set of character encodings. Optionally, it creates the extended set defined at the same URL. To add optional character encodings, see the `--extension` option in jrecreate Options.

Nashorn JavaScript Engine

All Oracle Java SE profiles and the full JRE include the `javax.scripting` API which supports scripting language statements in Java source files. At run time, the scripting statements are executed by a user-supplied scripting engine.

To include the Nashorn JavaScript engine in a JRE, see the `--extension` option in jrecreate Options.

For more information on the Nashorn JavaScript engine, see `https://blogs.oracle.com/nashorn/entry/welcome_to_the_nashorn_blog`.

JSR 197 Specification JAR

JSR 197 is the specification of the Generic Connection Framework (GCF) Optional Package for the Java 2 Platform, Standard Edition, version 1.0. The API is equivalent to CLDC 1.0 GCF. The MIDP 2.0 and FP 1.1 have a more complete GCF API (for example HTTPS).

The embedded package supports:

- `file`

  Local file read, write, and read/write

- `datagram`

  UDP send and receive

- `socket`

TCP client and server

- `http`

  HTTP client

Note that the JSR 197 JAR provided with Oracle Java SE Embedded has no functional implementation. It delegates all functions to the `java.io` and `java.net` packages.

The JSR 197 specification APIs are provided as a JAR file in the Oracle Java SE Embedded download bundle. It is not supported by the `jrecreate` command, but you can add it manually. The JAR file is located in the following directory of an Oracle Java SE Embedded installation:

`ejdk<`*`version`*`>/<`*`platform`*`>/options/gcf/lib`

This location needs to be in the classpath for any application that needs it, or you can add it as a standard extension into the `lib/ext` directory of the JRE.

6

About Oracle Java SE Embedded JVMs

Learn about the JVMs that you can choose from when building a custom JRE.

A Java virtual machine (JVM) is the basis of a Java Runtime Environment. A JVM loads, verifies, and executes application and library code. A JVM can execute code by interpreting the bytecodes directly, or it can compile frequently used bytecode blocks into machine instructions, cache the compiled blocks, and then run them faster without interpretation. A JVM also manages dynamic memory allocation and reclamation, known as garbage collection.

Three Java Virtual Machines are included with the Oracle Java SE Embedded software. Each design balances performance-resource tradeoffs in a different way.

You can experiment with all three JVMs, and you can use one for debugging and another for production. To include particular Java Virtual Machines in a custom JRE, see Create Your JRE with jrecreate.

Minimal JVM

The minimal JVM emphasizes a minimal memory footprint over nonessential features. For example, it has a single serial garbage collector and no serviceability features. The minimal JVM does not support all `java` launcher options. See Launch the Application with the java Launcher Tool.

Client JVM

The client JVM is a full-featured Java Virtual Machine optimized for responsiveness. It starts and compiles applications more quickly and uses less memory than the server JVM. It has garbage collection options and supports debugging and profiling.

Server JVM

The server JVM is functionally identical to the client JVM, but it is tuned for long-running applications. It uses more memory than the client JVM. It launches and compiles applications more slowly but runs them faster because it creates more optimized code.

The server JVM is not available on all targets. See *Oracle Java SE Embedded Release Notes* for specific information about which targets support the server JVM.

Note: On platforms or JREs with no server VM, `-server` is aliased to the available VM, - either `-client` or `-minimal`. Similarly in a JRE produced by `jrecreate` that only has the minimal VM, `-client` is aliased to `-minimal`. Note that `-minimal` is never aliased; if you request it and it is not present, you will get an error.

7

About Compact Profiles

This chapter discusses the concept of compact profiles in the Oracle Java SE Embedded platform.

This chapter contains the following sections:

Compact Profiles for Subsets of the Java SE API

You can minimize space requirements on the embedded device by limiting the static footprint to Java API packages that the application uses. You can build a custom JRE that contains the full set of Java SE APIs or one of three subsets, called profiles.

Compact1 Profile APIs

Similar to the legacy Connected Device Configuration (CDC) with the Foundation Profile, secure sockets layer (SSL), logging, and scripting language support, including Javascript. When configured with the minimal JVM, the compact1 profile APIs have a static footprint of about 12MB.

Compact2 Profile APIs

Adds these packages to compact1:

- Remote Method Invocation (RMI, JSR 66))
- Java API for XML Processing (JAXP, JSR 280)
- Java Database Connectivity (JDBC, JSR 169)

When configured withe the minimal JVM, the compact2 profile APIs have a static footprint of about 17MB.

Compact3 Profile APIs

The compact2 profile adds serviceability, naming, the compiler API, and more security.

For a description of serviceability, see
http://openjdk.java.net/groups/hotspot/docs/Serviceability.html

Note that the compact3 profiles cannot be configured with the minimal JVM.

Full JRE APIs

The full JRE adds desktop, web services, and CORBA APIs. It also supports Java Flight Recorder (JFR) command line options.

The full JRE APIs include all the Java SE classes, yet space-saving optimizations make a full Oracle Java SE Embedded JRE substantially smaller than a Java SE JRE. When configured with the client JVM, the full JRE APIs have a static footprint of about 50MB.

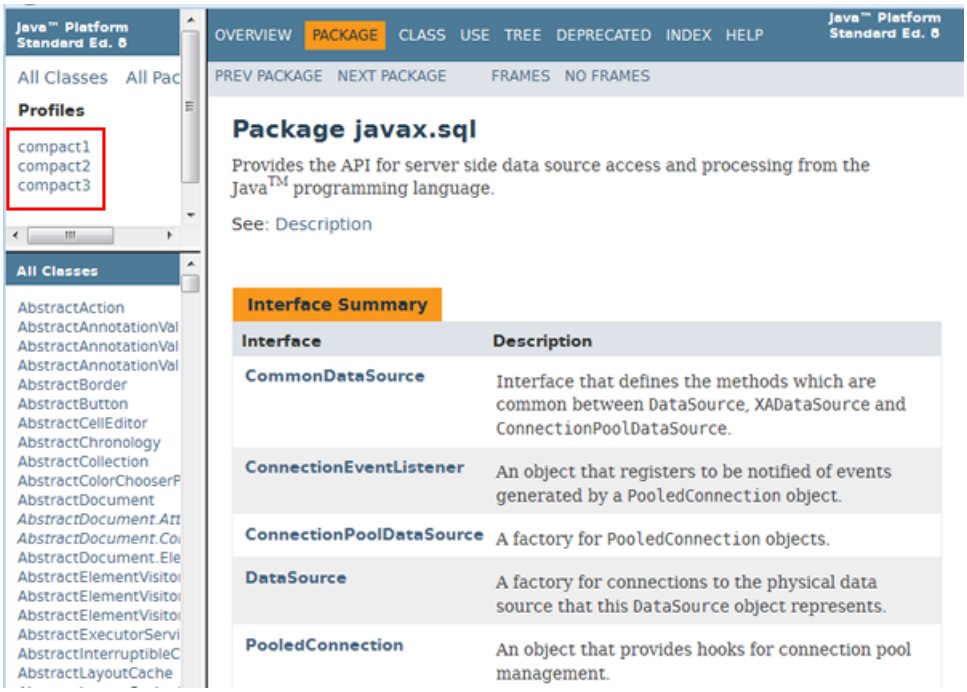
Note: For headless configurations, the full JRE APIs include the `java.awt` and `java.swing` classes. These classes are provided for printing and drawing bitmaps in memory. An attempt to create a GUI window in this mode raises the `HeadlessException`.

Configuring the JRE for Swing/AWT Headful Applications describes a configuration that supports AWT/Swing.

Determining Compact Profiles for API Objects

The Java SE API documentation shows which compact profiles each API object belongs to. For example, Figure 7–1 is a screenshot of an API documentation page showing that the `javax.sql` package belongs to all three compact profiles.

Figure 7–1 Example Javadoc with Information about Compact Profiles



8

Create Your JRE with jrecreate

This chapter describes how to create your Java Runtime Environments with the `jrecreate` tool.

This chapter contains the following topics:

Running jrecreate

You must run `jrecreate` on the host computer, then copy over the resulting JRE to the embedded device.

> **Note:** `jrecreate` needs approximately 1 GB of free memory to run.

jrecreate Command Syntax

```
jrecreate --dest host-destination-directory [options]
```

jrecreate Options

Options can be specified in any order. Only one option is required, the directory into which `jrecreate` writes the JRE.

Note that double hyphens precede the long form of all options.

> **Note:** The ARM Swing/AWT configuration of Oracle Java SE Embedded does not support all `jrecreate` options. See Configuring the JRE for Swing/AWT Headful Applications.

-h
--help
Optional. Use as the only option when you need a summary of the command-line options.

-d *path*
--dest *path*
Required. Directory in which to write the JRE image and related files. The directory must not exist.

-p *profile*
--profile *profile*
Optional. Default: full JRE APIs.

Creates a JRE whose APIs are based on a specified profile (see About Compact Profiles). The valid values are one of the following:

- `compact1`
- `compact2`
- `compact3`

--vm *jvm*
Optional. If neither `--vm` nor `--profile` is specified, all available JVMs are included in the JRE.

Creates a JRE that has a particular Java Virtual Machine (JVM) or all JVMs. See About Oracle Java SE Embedded JVMs. The valid values are one of the following:

- `minimal`
- `client`
- `server` (if available for the target computer)
- `all` (all available for the target computer)

To learn how the `java` program launcher chooses a JVM when multiple JVMs are available, see Launch the Application with the java Launcher Tool.

If you do not specify the `--vm` option, then the `jrecreate` command uses the value of the `--profile` option to select a JVM as follows:

- `compact1`: `minimal`
- `compact2`: `minimal`
- `compact3`: `client`

-x *extension* [,*extension* ...]
--extension *extension* [,*extension* ...]
Optional. Default: no extensions

Adds optional components to a JRE. With the exceptions noted below, you can use the components in any combination and with all profiles and JVMs.

You can specify multiple extensions in either of the following ways:

- Multiple option instances:
```
--extension sunec \
--extension sunpkcs11
```
- Comma separated list of extensions:
```
--extension sunec,sunpkcs11
```

Valid --extension Values

- `fx:graphics`: Installs the JavaFX Graphics component and Base component for headful applications, described in JavaFX Components for Oracle Java SE Embedded.
- `fx:controls`: Installs the JavaFX Controls component, the Graphics component, the Base component, and the FXML component for headful applications, described in JavaFX Components for Oracle Java SE Embedded. If you specify `fx:controls`, you do not need to add `fx:graphics` as a value.
- `sunec`: Security provider for Elliptic Curve Cryptography (ECC). See Cryptographic Service Providers.
- `sunpkcs11`: Security provider for Cryptographic Token Interface Standard PKCS#11. See Cryptographic Service Providers.
- `locales`: Include all Java SE locales instead of the default US-English. See Locales.
- `charsets`: Include extended character encoding set. See Character Encodings.
- `nashorn`: Include the Nashorn JavaScript engine. See Nashorn JavaScript Engine.

-g
--debug
Optional. Default: no debug support.

Creates a JRE that contains debugging support, including Java Virtual Machine Tool Interface (JVMTI) support for inspecting and controlling an application.

Notes on the `--debug` component:

- This option is not valid for the minimal JVM. If that combination is specified, then the `jrecreate` command selects the client JVM.
- Some debugging functionality requires that the `java` command be run with the `-XX:+UsePerfData` flag. By default in Oracle Java SE Embedded, this flag is turned off.

-k
--keep-debug-info
Optional. Default: no debugging information.

Does not strip debugging information from class and unsigned JAR files. This option increases the size of some JRE files for easier debugging.

--no-compression
Optional. Default: compressed JAR files.

Creates a JRE whose unsigned JAR files are created in an uncompressed format. This option trades increased footprint for potentially improved startup time. Experiment to see if the option is beneficial in your environment.

-n
--dry-run
Optional. Default: create the JRE.

Generates a descriptive report without creating a JRE image.

-v
--verbose
Optional. Default: terse output.

Displays verbose output, including the commands that the `jrecreate` command invokes. This option is helpful for debugging.

--ejdk-home *path*
Optional. Default: value of `EJDK_HOME` environment variable if set; otherwise `/ejdk<version>`

Directory that contains the Oracle Java SE Embedded runtime components to use in the custom JRE. Use this option when you have multiple Oracle Java SE Embedded installations and you want to run the `jrecreate` command in *`installation1`* but build a JRE with components from *`installation2`*. For example, *`installation1`* might be contain ARM components, and *`installation2`* might contain x86 components.

jrecreate Command Examples

The following example show how the `jrecreate` options are used to create some common custom JREs.

The *`installDir`* variable represents the host directory where Oracle Java SE Embedded is installed.

Example 1

Smallest JRE: headless, `compact1` profile, minimal JVM (default for `compact1`).

```
% installDir/ejdk<version>/bin/jrecreate.sh \
--profile compact1 \
--dest /tmp/defaultJRE/
```

Example 2

`compact2` profile, client JVM with debugging support.

```
% installDir/ejdk<version>/bin/jrecreate.sh \
    --dest /tmp/exampleJRE1/ \
    --profile compact2 \
    --vm client \
    --keep-debug-info \
    --debug
```

Example 3

JavaFX controls and Java SE locales added to server JVM and `compact3` profile.

```
% installDir/ejdk<version>/bin/jrecreate.sh \
    --dest /tmp/exampleJRE2 \
    --profile compact3 \
    --vm server \
    --extension fx:controls \
    --extension locales
```

Example 4

Full JRE APIs, all JVMs (default).

```
% installDir/ejdk<version>/bin/jrecreate.sh \
    --dest /tmp/exampleJRE3
```

jrecreate Command Output

When it begins to run, the `jrecreate` command displays a summary of the options used to build the JRE, for example:

```
Building JRE using options Options {
    ejdk-home: /home/xxxx/ejdk/ejdk<version>
    dest: /tmp/testjre
    target: linux_i586
    vm: minimal
    runtime: compact1 profile
    debug: false
    keep-debug-info: false
    no-compression: false
    dry-run: false
    verbose: false
    extension: []
}
```

JRE Directories

After the command completes, the destination (`--dest`) directory on the host may contain the following directories, depending on which components are included in the JRE:

- `bin/`: Target-native commands, minimally including the `java` JRE application launcher. The complement of tools varies according to the value of the `--profile` option.
- `lib/`: The files that make up the core of the JRE, including classes, JVMs, time zone information, and other resources.
- `release`: A text file that tools can read to obtain attributes of the generated JRE, such as the Java version, profile name (if applicable), operating system name, and CPU architecture.
- `bom`: A text file that documents how the JRE was created, including the `jrecreate` command options and the files that the command used.
- `COPYRIGHT`, `LICENSE`, `README`, `THIRDPARTYLICENSEREADME.txt`: Legal and other documentation. Present only if `--profile` is not specified (full JRE APIs).

Configuring the JRE for Swing/AWT Headful Applications

The ARM AWT/Swing configuration of Oracle Java SE Embedded supports the Swing and AWT graphics APIs if you need to run Swing/AWT applications. Consult Oracle Java SE Embedded System Requirements for the platforms that support headful Swing/AWT applications.

To install the Swing/AWT APIs, you must create a full JRE by using the `jrecreate` command without the --profile option. With a full JRE, you can install the client VM (the `--vm client` option), the server VM (the `--vm server` option) or both (the `--vm all` option, or leave the `--vm` option unspecified). The minimal VM is not supported.

If you plan to run both AWT/Swing and JavaFX applications on a target that supports both AWT/Swing and JavaFX, you can add JavaFX APIs by adding the `--extension` option with `fx:graphics` or `fx:controls`. To keep the footprint small, only install the JavaFX APIs when you know that you will be running JavaFX applications.

Note: An application can only make calls to the Swing/AWT API or the JavaFX API, not both.

9

Deploy Your JRE to the Embedded Device

This chapter describes how to deploy a custom JRE to a target device.

It contains the following sections.

Moving the Custom JRE to the Target Device

To make a JRE image run on a target device, copy its files from the `jrecreate` destination (`--dest`) directory (see jrecreate Command Output) to the file system of the target device.

How you copy depends on the device and your environment. For example, you can use an SD card or a network connection.

Adapt the following steps to your environment (*`destDir`* is the `jrecreate` destination directory):

1. Recursively copy *`destDir`* from the host to the device directory where you want the JRE installed. For example:

   ```
   $ scp -r /tmp/SmallJRE/* root@target:/opt/local/ejdk<version>/
   ```

2. If necessary, update the device's `PATH` environment variable to include the `bin/` directory of the JRE. For instance:

   ```
   $ PATH=$PATH:/opt/local/ejdk<version>/bin/
   $ export PATH
   ```

To launch applications on your deployed JRE, see Launch Embedded Applications.

Verifying Your Deployed JRE

To verify that the JRE is correctly deployed on the target device, run the `java` command with the `-version` option on the target. For example:

```
$ ssh root@target java -version
```

Your command output will display details such as build numbers and components.

If you need to debug your running application, you must use the `-XX:+UsePerfData` flag of the `java` command. In Oracle Java SE Embedded, this flag is turned off by default.

For more information about deploying and launching embedded applications, see Deploy Embedded Applications and Launch Embedded Applications.

Part III

Embedded Application Development

This part of the guide contains information for developers who plan to develop applications for embedded devices with Oracle Java SE Embedded JREs installed. It contains the following chapters:

10

Essentials for Developing Embedded Applications

This chapter describes the fundamentals of developing Java applications that run on an Oracle Java SE Embedded custom JRE.

This chapter contains the following topics:

Host Development for a Target Device

The fundamental difference between conventional and embedded Java programming is the presence of a second computer, called the target. The target is embedded in a device — a printer, a medical instrument, an industrial tool, or whatever houses the embedded system.

If the target has sufficient resources and is supported by the Java SE Development Kit (JDK), you can develop an embedded application on it, as you would develop a Java desktop application or a Java server application. Resources include CPU cycles, memory, file system, display, pointing device, and keyboard. Click the **Download** button on the Java SE download page to see a list of the processor-operating system combinations for which JDKs are available:
`http://www.oracle.com/technetwork/java/javafx/downloads/`

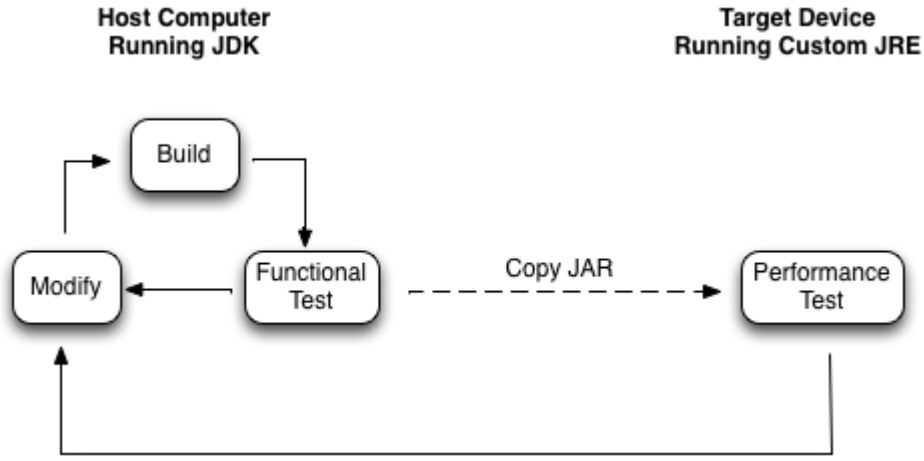
Resource-rich targets are rare. By contrast, desktop and laptop computers are common, inexpensive, familiar, and have resources in abundance. An array of software development tools, such as integrated development environments and code repositories, is also available for desktop and laptop computers. Accordingly, embedded application development is usually divided between a desktop or laptop computer, called a host, and the target.

You use the host computer to write and compile source code, to perform functional testing, to package the compiled application into an executable Java archive (JAR) file, and, possibly, to remotely start and stop applications on the target. See Develop and Test Your Embedded Application and Deploy Embedded Applications for details.

Host-Target Development Cycle

Figure 10–1 shows that embedded application development is the same as conventional Java application development, with an additional activity: performance testing on the target. Performance testing, as the term is used here, means running the application on the actual deployment hardware to discover hardware-specific issues. These might include excessive memory consumption, insufficient speed, race conditions, and problems with peripherals, including displays, sensors, and actuators. Functional testing on the host computer cannot reveal these conditions, but it can expose problems in core application behavior.

Figure 10–1 Fundamental Embedded Development Cycle

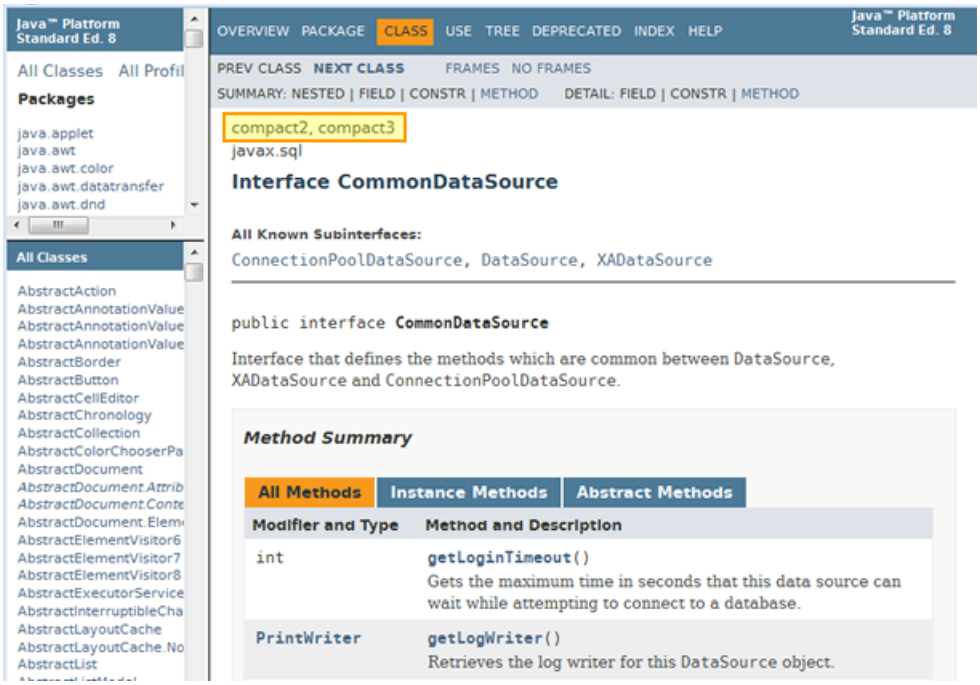


Because custom JREs support networking and can support remote Java debugging and profiling, you can copy JARs to, and run performance tests on, a target that is located in another room, building, or continent.

APIs and Javadocs

The Java SE API documentation displays profile information. Figure 10–2 shows an example of an interface that is not in the compact1 profile APIs. Its definition explicitly names the compact2 and compact3 profiles. All interfaces are implicitly in the full JRE APIs.

Figure 10–2 Example Javadoc Showing an Interface in Two Profiles



Native Methods

Embedded applications sometimes need access to target device hardware for which there is no Java API. Sensors and actuators are typical examples. Typically, device access functions are available in C. You can call these functions from Java by creating intermediating native methods. For a description of the Java Native Interface (JNI), see the JNI specification in the *Java SE Developer Guides*.

For a simple example of JNI, see the Oracle MoonOcean blog at
`https://blogs.oracle.com/moonocean/entry/a_simple_example_of_jni`

JDK 7 Limitations

If you want to use JDK 7, observe this caution:

Note: When using JDK 7, ensure that you use only APIs included in the custom JRE created for your device by the platform developer (See About Compact Profiles). If your application uses an API that is not present in the target device's custom JRE, the host's JDK 7 compiler and runtime mask your error. When you test the application on the target device, its custom JRE throws a `ClassNotFoundException`. Test early and periodically on the target to detect this error promptly.

Also, the `jdeps` tool is not available in JDK 7.

11

Develop and Test Your Embedded Application

This chapter describes techniques for developing, testing, and compiling your embedded application. It applies to both headless and headful applications.

This chapter contains the following topics:

- Develop Your Application
- Test Your Application on the Host Computer
- Testing Your Application on the Target Device

For information that is unique to headful applications, see Develop Headful Applications.

> **Tip:** For an overview of the development process, see Quick Start for Application Developers.

Develop Your Application

Develop applications on a host computer with a full JDK. You can use the Java application development tools of your choice. See, for example, Developing Embedded Applications in NetBeans IDE.

As you refer to the standard JDK documentation, use the information in the API documentation about compact profiles to ensure that you are working within the constraints of the JRE that will be installed on the embedded device. See Determining Compact Profiles for API Objects.

Test Your Application on the Host Computer

Test your application frequently to ensure that your application will run under the compact profile that is included in your custom JRE. There are two standard JDK tools that check the profile: `javac` and `jdeps`.

Compile with the javac Tool and the -profile Option

When you compile your application with the javac tool, specify the profile installed on the target device with the -profile option to test whether all the APIs in your application are included in the profile that be included in your custom JRE.

For example, the following command compiles the application and at the same time tests to make sure that all of your APIs are included in the `compact1` profile.

```
$ javac -profile compact1 Hello.java
```

If `Hello.java` uses an API that is not present in the `compact1` profile, the compiler detects the error.

Use the jdeps Tool to Test Minimum Required Compact Profile

After compiling, you can use the `jdeps` dependency analyzer tool with the `-P` option to determine the minimum compact profile required for each class in your application. For example:

```
% jdeps -P HelloWorld.class
```

Here is an example of the output:

```
HelloWorld.class ->
/net/test11.us.example.com/export/java-re/jdk/8/ea/b124/binaries/linux-i586/jre/li
b/rt.jar
   <unnamed> (HelloWorld.class)
      -> java.io                                            compact1
      -> java.lang                                          compact1
```

In this example, both classes used in the application minimally require the `compact1` profile.

For more information about this tool, see the `jdeps` documentation for UNIX or Windows.

Testing Your Application on the Target Device

Occasionally you should test your application on the target device, and when the application is final, deploy it. See Deploy Embedded Applications.

12

Deploy Embedded Applications

This chapter shows you how to package your application and copy it to the target device.

It contains the following sections.

Package Your Application on the Host with the jar Tool

Use the standard jar command to create a .jar file of the compiled application. See the Java SE `jar` tool documentation for UNIX and Windows

About Connecting to the Target Device

As noted in Host Development for a Target Device, there are many options for controlling the target and copying compiled applications to it. The setup details are target-specific, and cannot be described in this guide.

Compiling on the host and running on the target requires a way to copy compiled application files from host to target. There are many possibilities, including:

- Writing to a storage device supported by both computers, such as an SD card or flash drive. Format the drive as FAT32 to eliminate file ownership/permission issues.
- Remotely mounting the target's file system on the host with the Network File System (NFS) or Samba.
- Committing compiled code from the host to a shared repository and updating the target from the repository.
- Using remote commands from a host over a network, such as `scp` (secure copy).

The target must provide a way to initiate operating system commands, notably the `java` launcher. To accomplish this, you can do one of the following:

- Connect a keyboard and display directly to the target.
- Use `ssh` from a network-connected host to run commands on the target remotely.

  Using a network for communication enables the host and target to be geographically separated.

To provide a uniform environment for examples, this chapter assumes that your host can communicate with the target with the standard network commands `scp` and `ssh`. You need the IP address, user name, and password to connect the host to the target.

Copy the Application to the Target Device

Copy the `.jar` file from the host to the target with the `scp` command.

Next Steps

Launch the application on the target device. See Launch Embedded Applications.

13

Launch Embedded Applications

This chapter shows you how to use the `java` launcher with various options and flags to launch your embedded applications on the target device. It also shows how to exit an application running on an embedded device and where to go for troubleshooting help.

It contains the following sections.

Launch the Application with the java Launcher Tool

Start an application for performance testing on the target device with the `java` launcher. You can use the `-cp` or `-jar` option.

The following example shows the commands for remotely launching the application with `ssh` and `java -cp`. In this example, the user name on the target device is `pi`

```
$ ssh pi@192.0.2.0
pi@192.0.2.0's password:
...
$ java -cp Hello.jar helloworldapp.HelloWorldApp
```

The final argument is the *`package.class`* that contains the application's `main()` method.

Besides this generic example, you can Launch Your Application with a Specific JVM or Launch Your Application in Debug Mode. See also Unsupported java Launcher Options in the Minimal JVM

Launch Your Application with a Specific JVM

An Oracle Java SE Embedded JRE can contain multiple JVMs. You can specify a particular JVM with these `java` launcher options:

- `-minimal`
- `-client`
- `-server`

If you use the `-client` and `-server` option and that JVM is not present or available, then it will be aliased to the available JVM. Using the `-minimal` option causes an error if the minimal JVM is not present.

Enable Client Compiler (C1) Profiled Inlining

The client compiler inlining policy uses profile information to improve performance in the minimal and client JVMs. You can enable C1 profiled inlining by using the `java` launcher flag `-XX:+C1ProfileInlining`.

Improving JVM Startup Times with Class Data Sharing

Class Data Sharing (CDS) is used with the HotSpot VM to reduce JVM startup times. For general information about CDS, see the Java SE 8 documentation.

Starting with Oracle Java SE Embedded 8u6, you can customize the set of classes included in the classlist and the name of the shared-archive file, and you can specify a custom location for the classlist. These are special features for Oracle Java SE Embedded applications.

Note: The shared-archive file content is considered trusted by default, meaning the contents are not checked, but it contains executable native code that runs outside the Java sandbox. Ensure that the integrity of any shared-archive file you generate cannot be subject to potentially malicious modifications.

There are three `java` launcher flags that are related to custom CDS classlists:

-XX:DumpLoadedClassList=*<classlist_file>*
Creates a custom classlist by writing out the set of all classes loaded by the boot loader to the named file in *<classlist_file>*. This custom classlist can be used by CDS when creating a shared archive.

-XX:SharedClassListFile=*<classlist_file>*
Specifies a user-defined classlist to be used when creating a shared archive, such as the one created by the `-XX:DumpLoadedClassList` flag.

-XX:SharedArchiveFile=*<archive_file>*
Specifies the name and location of the shared-archive file to be written to during a CDS dump, or read from during JVM execution with CDS enabled.

Here is an example of how you build and implement a custom classlist for your application.

Task 1 Generate the custom list of core library (boot) classes used by your application

```
java -XX:DumpLoadedClassList=./MyApp.classlist MyApp
```

Task 2 Generate a custom shared archive containing those classes

```
java -Xshare:dump -XX:SharedClassListFile=./MyApp.classlist
    -XX:SharedArchiveFile=./MyApp.jsa
```

Task 3 Run your application using the custom shared archive

```
java -Xshare:on -XX:SharedArchiveFile=./MyApp.jsa MyApp
```

Launch Your Application in Debug Mode

If you have a JRE on the target that was built with debugging support, you can start the application in debug mode, then attach a JVMTI-compliant remote debug client from any host on the same network. One client is `jdb`, described in the Java SE tools documentation for UNIX and Windows. Example 13–1 shows launching with `java -jar` in debug mode.

Example 13–1 Launching with -jar in Debug Mode

```
$ java -jar Hello.jar \
-agentlib:jdwp=transport=dt_socket,server=y,address=8000
```

Note: Some debugging functionality requires that the `java` tool be run with the `-XX:+UsePerfData` flag. By default in Oracle Java SE Embedded, this flag is turned off, whereas it is turned on by default in the Java SE JDK.

Unsupported java Launcher Options in the Minimal JVM

To reduce memory use, the minimal JVM does not support some virtual machine options passed by the `java` launcher. When an unsupported option is specified, depending on its seriousness, the launcher prints a warning message, ignores the option, and launches the JVM, or prints an error message and terminates without launching the JVM.

Table 13–1 Options not Supported by the Minimal JVM

Option(s)	java Launcher Response
`-agentpath:jdwp`, `-Xrunjdwp`	Error
`-javaagent:jarpath=[options]`	Error
`-Xagent:hprof`, `-Xrunhprof`, `-agentlib:hprof`, `-agentpath:hprof`	Error
`-Xcheck:jni`	Warning
`-Xincgc`, `-XX:+UseGarbageCollector`	Warning
`-Xshare:auto`	Warning
`-Xshare:dump`, `-Xshare:on`	Error
`-XX:+ManagementServer`, `-Dcom.sun.management`	Error
`-XX:NativeMemoryTracking`	Error
`-XX:*Flight*`	Error

The minimal JVM does not support remote debugging, profiling, monitoring, and serviceability tools. These include `jcmd`, `jdb`, `jinfo`, `jmap`, `jstack`, and others, including integrated development environments. The minimal JVM ignores requests from these tools.

Exit an Application Running on an Embedded Device

Java applications cannot be terminated from the device's console. However, applications can be terminated using a remote shell:

```
$ pkill java
```

Troubleshooting

If you have difficulty with an application, you can find diagnostic help in the *Java Platform, Standard Edition Troubleshooting Guide*.

Also refer to *Oracle Java SE Embedded Release Notes* for known issues.

14

Develop Headful Applications

We recommend that you use the JavaFX API to create headful applications. If you prefer, Oracle Java SE Embedded provides an AWT/Swing configuration for ARM.

This chapter contains the following topics:

Headful Applications Using JavaFX

JavaFX is based on the scene graph construct, takes advantage of hardware acceleration, and supports animation. You can learn about JavaFX technology by visiting the Java SE Client Technologies page.

Figure 14–1 shows an example of a JavaFX headful application running on an embedded device.

Figure 14–1 City Explorer Example

Figure 14–2 shows a photo of a typical embedded touch device. Oracle Java SE Embedded supports multitouch events on supported devices. See *Oracle Java SE Embedded Release Notes* for devices that are supported for multitouch events.

Figure 14–2 Example Touch Device

A JavaFX GUI application can reference APIs from any Java library. For example, a JavaFX application can call Java API libraries to access native system capabilities or to communicate with other embedded applications.

The look and feel of JavaFX applications can be customized. Cascading Style Sheets (CSS) separate appearance and style from implementation so that developers can

concentrate on coding. Graphic designers can easily customize the appearance and style of the application through the CSS.

Because the desktop and embedded JavaFX APIs are compatible, you can do most GUI design and functional testing on a host computer, then copy the application to the target for final testing and tuning. For a quick view of the development process, see Quick Start for Application Developers.

Areas where behavior differs between embedded and desktop JavaFX platforms include the following:

- Embedded graphics processing units (GPUs) are typically much less powerful than desktop GPUs. A less powerful GPU can lead to decreased performance in some applications.
- UI controls can have a different appearance and behavior on embedded platforms depending on whether or not there is a touch interface.
- On embedded platforms, JavaFX provides a virtual keyboard for text input by users and for testing.

JavaFX Components for Oracle Java SE Embedded

In order to minimize the JRE, the JavaFX APIs are divided into two components:

- JavaFX Graphics Component
- JavaFX Controls Component

You must know which JavaFX component is included in the custom JRE for your application to run successfully on the target device.

JavaFX Graphics Component

JavaFX applications with effect and animation run on a JRE that is built with the Graphics component. The Graphics component supports windows, scene graph, animation, timelines, and property binding. The Graphics component does not support events or UI controls. A typical application running on a JRE that contains only the Graphics component can display effects and animation but cannot handle user input.

The Graphics component that is installed into a custom JRE with Oracle Java SE Embedded includes the Base component, as described on the Projects and Components page of the OpenJFX website.

The Graphics component's static footprint is about 6 MB.

Table 14–1 shows the packages included in the Graphics component.

Table 14–1 JavaFX Graphics Component Packages

Package	Description
`javafx.animation`	Animation
`javafx.application`	Lifecycle
`javafx.beans`	JavaFX beans
`javafx.beans.binding`	JavaFX bean property binding
`javafx.beans.property`	Read-only and read-write bean properties
`javafx.beans.value`	Observable and writable values

Table 14–1 (Cont.) JavaFX Graphics Component Packages

Package	Description
javafx.collections	JavaFX observable collections
javafx.concurrent	Threading classes
javafx.css	Styleable properties
javafx.event	Event handling
javafx.geometry	2D and 3D
javafx.scene	Scene graph core
javafx.scene.canvas	Immediate mode rendering
javafx.scene.effect	Visual effects
javafx.scene.image	Loading and displaying images
javafx.scene.input	Input events
javafx.scene.layout	Scene layout
javafx.scene.paint	Colors and gradients
javafx.scene.shape	2D and 3D shapes
javafx.scene.text	Text rendering and metrics
javafx.scene.transform	2D and 3D transformations
javafx.stage	Windowing
javafx.util	Utilities
javafx.util.converter	String converters

JavaFX Controls Component

The Controls component adds support for UI controls, event handling, and charts. The Controls component also includes the FXML component, described on the Projects and Components page of the OpenJFX website. In addition, installing the Controls component in the custom JRE automatically includes the Graphics component.

The total footprint of the Controls component plus the other components installed with it is about 9 MB.

Table 14–2 shows the JavaFX packages included in the Controls component.

Table 14–2 JavaFX Controls Component Packages

Package	Description
javafx.fxml	FXML
javafx.scene.chart	Charts
javafx.scene.control	UI controls
javafx.scene.control.cell	Cells for UI controls

Configuring Fonts

See https://wiki.openjdk.java.net/display/OpenJFX/Font+Setup for information about configuring fonts for JavaFX applications.

Unsupported JavaFX Features

The Graphics and Controls components are a large subset of JavaFX. Compared to the desktop version, the JavaFX components included in Oracle Java SE Embedded have the following restrictions:

- They do not support the `WebView` node (a scene graph node for displaying and interacting with web content).
- They do not support applets, Swing integration, or Standard Widget Toolkit (SWT) integration.
- They do not have the `MediaView` class or any support for media playback.
- They do not support APIs related to integration with a desktop UI (for example, access to a global clipboard, window decorations or iconization, file and directory choosers).
- The embedded version of the JavaFX virtual keyboard supports only the US English locale.
- When running with the compact1 profile (see Compact1 Profile APIs), JavaFX components do not support user interfaces defined with FXML, including those generated by JavaFX Scene Builder.

Table 14–3 lists JavaFX desktop packages that are not present in either of the two Oracle Java SE Embedded JavaFX components.

Table 14–3 JavaFX Packages not Available in Oracle Java SE Embedded

Package	Description
`javafx.beans.property.adapter`	Integration with JavaBeans
`javafx.embed.swing`	Integration with the Swing API
`javafx.embed.swt`	Integration with the SWT API
`javafx.print`	Printing
`javafx.scene.media`	Media playback
`javafx.scene.web`	Web content
`netscape.javascript`	JavaScript integration

Using FXML Markup Instead of JavaFX APIs

If you have a web design background, or if you would like to separate the user interface (UI) and the back-end logic, then you can use FXML to develop the presentation aspects of the UI.

FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code. This separation of the presentation and application logic is attractive to web developers because they can assemble a user interface that takes advantage of Java components without mastering the code for fetching and filling in the data.

> **Note:** All but the `compact1` profile can use FXML See About Compact Profiles.

To see how to use FXML and integrate it into a JavaFX application, see the Java SE Client Technologies page.

Using JavaFX Scene Builder to Design the UI and Export to FXML

If you prefer to design UIs without writing code or FXML directly, then use JavaFX Scene Builder. As you build the layout of your UI, Scene Builder generates the FXML code for the layout, which is then integrated into a JavaFX application that contains the application logic.

For links to JavaFX Scene Builder documentation, see the Java SE Client Technologies page.

Note: The `compact1` profile cannot use JavaFX Scene Builder because the underlying FXML output is not supported.

JavaFX Sample Applications

The JavaFX sample applications download zip file contains Ensemble 8, a sample that was tested on embedded platforms that support JavaFX and have JREs that were built with the graphics component. Note that the 3DViewer and Modena samples are known not to run on embedded devices.

To access the Ensemble 8 sample, download the JavaFX samples by following the link from the Java SE downloads page at
http://www.oracle.com/technetwork/java/javase/downloads/

Once you have extracted the sample JAR file from the sample zip file, you can double-click it to run it on your host machine. Use the following procedure to run the sample on a target device that is supported for JavaFX and is running a JRE that was built with the graphics component.

1. Transfer the JAR file to the target embedded device.
2. On the target device, run the `java` launcher to invoke the sample's main class, for example:

   ```
   $ deployDir/bin/java -classpath /tmp/JavaFXSamples/Ensemble8.jar
   ensemble.EnsembleApp
   ```

   Note the following about this example:

   - On the target, *`deployDir`* contains an Oracle Java SE Embedded JRE that includes the controls component or the graphics component.
   - On the target, the `PATH` environment variable is set so that the `Java` launcher is found in *`deployDir`*`/bin/`.
   - On the target, the extracted sample applications are in the `/tmp/JavaFXSamples/` directory.
   - If the target device uses software rendering, add the following option to the command line:

     ```
     -Djavafx.platform=directfb
     ```

Java applications cannot be terminated from the device's console. Instead, terminate them using a remote shell:

```
$ pkill java
```

Font Setup in Headful Applications

See https://wiki.openjdk.java.net/display/OpenJFX/Font+Setup for information about configuring fonts.

Swing and AWT APIs

JavaFX components included in an Oracle Java SE Embedded JRE have much smaller static footprints than the Swing and AWT libraries, but if you prefer you can use the ARM AWT/Swing configuration of Oracle Java SE Embedded. This configuration supports the Swing and AWT graphics APIs.

Note: Note that an application can only make calls to the Swing/AWT API or the JavaFX API, not both.

Make sure that the JRE for the target device includes the AWT/Swing configuration. See Configuring the JRE for Swing/AWT Headful Applications.

15

Codecache Tuning

This chapter describes techniques for reducing the just-in-time (JIT) compiler's consumption of memory in the codecache, where it stores compiled methods.

This chapter contains the following topics:

Introduction

The Java Virtual Machine (JVM) generates native code and stores it in a memory area called the codecache. The JVM generates native code for a variety of reasons, including for the dynamically generated interpreter loop, Java Native Interface (JNI) stubs, and for Java methods that are compiled into native code by the just-in-time (JIT) compiler. The JIT is by far the biggest user of the codecache. This appendix describes techniques for reducing the JIT compiler's codecache usage while still maintaining good performance.

This chapter describes three ways to reduce the JIT's use of the codecache:

- Constrain the amount of codecache available to the JIT.
- Tune the JIT to compile fewer methods.
- Tune the JIT to generate less code per method.

java Launcher Codecache Option Summary

The JVM options passed by the `java` launcher listed in the tables in this section can be used to reduce the amount of codecache used by the JIT. The table descriptions are summaries. Most of the options are described in more detail in the sections that follow.

How to Use the Codecache Options of the java Command

The options listed in the following sections share the following characteristics.

- All options are `-XX` options, for example, `-XX:InitialCodeCacheSize=32m`. Options that have true/false values are specified using `+` for true and `-` for false. For example, `-XX:+PrintCodeCache` sets this option to true.
- For any option that has "varies" listed as the default value, run the launcher with `XX:+PrintFlagsFinal` to see your platform's default value.
- If the default value for an option differs depending on which JVM is being used (client or server), then both defaults are listed, separated by a '/'. The client JVM default is listed first. The minimal JVM uses the same JIT as the client JVM, and therefore has the same defaults.

Codecache Size Options

Table 15–1 summarizes the codecache size options. See also Constraining the Codecache Size.

Table 15–1 Codecache Size Options

Option	Default	Description
`InitialCodeCacheSize`	160K (varies)	Initial code cache size (in bytes)
`ReservedCodeCacheSize`	32M/48M	Reserved code cache size (in bytes) - maximum code cache size
`CodeCacheExpansionSize`	32K/64K	Code cache expansion size (in bytes)

Codecache Flush Options

Table 15–2 summarizes the codecache flush options.

Table 15–2 Codecache Flush Options

Option	Default	Description
`ExitOnFullCodeCache`	false	Exit the JVM if the codecache fills
`UseCodeCacheFlushing`	false	Attempt to sweep the codecache before shutting off compiler
`MinCodeCacheFlushingInterval`	30	Minimum number of seconds between codecache sweeping sessions
`CodeCacheMinimumFreeSpace`	500K	When less than the specified amount of space remains, stop compiling. This space is reserved for code that is not compiled methods, for example, native adapters.

Compilation Policy Options

Table 15–3 summarizes the compilation policy (when to compile) options.

Table 15–3 Compilation Policy Options

Option	Default	Description
`CompileThreshold`	1000 or 1500/10000	Number of interpreted method invocations before (re-)compiling
`OnStackReplacePercentage`	140 to 933	NON_TIERED number of method invocations/branches (expressed as a percentage of `CompileThreshold`) before (re-)compiling OSR code

Compilation Limit Options

Table 15–4 summarizes the compilation limit options, which determine how much code is compiled).

Table 15–4 Compilation Limit Options

Option	Default	Description
`MaxInlineLevel`	9	Maximum number of nested calls that are inlined
`MaxInlineSize`	35	Maximum bytecode size of a method to be inlined
`MinInliningThreshold`	250	Minimum invocation count a method needs to have to be inlined
`InlineSynchronizedMethods`	true	Inline synchronized methods

Diagnostic Options

Table 15–5 summarizes the diagnostic options.

Table 15–5 Diagnostic Options

Option	Default	Description
`PrintFlagsFinal`	false	Print all JVM options after argument and ergonomic processing
`PrintCodeCache`	false	Print the code cache memory usage when exiting
`PrintCodeCacheOnCompilation`	false	Print the code cache memory usage each time a method is compiled

Measuring Codecache Usage

To measure the success of a codecache usage reduction effort, you must measure the codecache usage and the effect on performance. This section explains how to measure the codecache usage. It is up to you to decide the best way to measure performance for your application.

Start with a baseline (the amount of codecache used when no codecache reduction techniques are applied), and then monitor the effect of your codecache reduction techniques on both codecache size and performance relative to the baseline.

Keep in mind that the codecache starts relatively small and then grows as needed as new methods are compiled. Sometimes compiled methods are freed from the codecache, especially when the maximum size of the codecache is constrained. The memory used by free methods can be reused for newly compiled methods, allowing additional methods to be compiled without growing the codecache further.

You can get information on codecache usage by specifying `-XX:+PrintCodeCache` on the `java` launcher command line. When your application exits, you will see output similar to the following:

```
CodeCache: size=32768Kb used=542Kb max_used=542Kb free=32226Kb
 bounds [0xb414a000, 0xb41d2000, 0xb614a000]
 total_blobs=131 nmethods=5 adapters=63
 compilation: enabled
```

The most useful part of the output for codecache reduction efforts is the first line. The following describes each of the values printed:

- `size`: The maximum size of the codecache. It should be equivalent to what was specified by `-XX:ReservedCodeCacheSize`. Note that this is not the actual amount

of physical memory (RAM) used by the codecache. This is just the amount of virtual address space set aside for it.

- `used`: The amount of codecache memory actually in use. This is usually the amount of RAM the codecache occupies. However, due to fragmentation and the intermixing of free and allocated blocks of memory within the codecache, it is possible that the codecache occupies more RAM than is indicated by this value, because blocks that were used then freed are likely still in RAM.
- `max_used`: This is the high water mark for codecache usage; the maximum size that the codecache has grown to and used. This generally is considered to be the amount of RAM occupied by the codecache, and will include any free memory in the codecache that was at some point in use. For this reason, it is the number you will likely want to use when determining how much codecache your application is using.
- `free`: This is `size` minus `used`.

The `-XX:+PrintCodeCacheOnCompilation` option also produces the same output as the first line above produced by `-XX:+PrintCodeCache`, but does so each time a method is compiled. It can be useful for measuring applications that do not terminate. It can also be useful if you are interested in the codecache usage at a certain point in the application's execution, such as after application startup has completed.

Because `max_used` generally represents the amount of RAM used by the codecache, this is the value you will want note when you take your baseline measurement. The sections that follow describe how you can reduce `max_used`.

Constraining the Codecache Size

Constraining the codecache size means the codecache is limited to a size that is less than what would an unconstrained codecache would use. The `ReservedCodeCacheSize` option determines the maximum size of the codecache. It defaults to a minimum of 32MB for the client JVM and 48MB for the server VM. For most Java applications, this size is so large that the application will never fill the entire codecache. Thus the codecache is viewed as being unconstrained, meaning the JIT will continue to compile any code that it thinks should be compiled.

When is Constraining the Codecache Size Useful?

Applications that make state changes that result in a new set of methods being "hot" can benefit greatly from a constrained codecache.

A common state change is from startup to regular execution. The application might trigger a lot of compilation during startup, but very little of this compiled code is needed after startup. By constraining the codecache, you will trigger codecache flushing to throw away the code compiled during startup to make room for the code needed during application execution.

Some applications make state changes during execution, and tend to stay in the new state for an extended period of time. For these applications, the codecache only needs to be big enough to hold the compiled code needed during any given state. Thus if your application has five distinct states, each needing about 1MB of codecache to perform well, then you can constrain the codecache to 1MB, which will be an 80% reduction over the normal 5MB codecache usage for the application. Note, however, that each time the application makes a state change, there will be some performance degradation while the JIT compiles the methods needed for the new state.

How to Constrain the Codecache Size

When the codecache is constrained (its usage approaches or reaches the `ReservedCodeCacheSize`), to compile more methods, the JIT must first throw out some already compiled methods. Discarding compiled methods is known as codecache flushing. The `UseCodeCacheFlushing` option turns codecache flushing on and off. By default it is on. You can disable this feature by specifying `XX:-UseCodeCacheFlushing`. When enabled, the codecache flushing is triggered when the memory available in the codecache is low. It is critical to enable codecache flushing if you constrain the codecache. If flushing is disabled, the JIT does not compile methods after the codecache fills up.

To determine an appropriate `ReservedCodeCacheSize` value for your application, you must first see how much codecache the application uses when the codecache is unconstrained. Use the `XX:+PrintCodeCache` option described in Measuring Codecache Usage, and examine the `max_used` value, which is how much codecache your application uses. You can then try setting `ReservedCodeCacheSize` to smaller values and see how well your application performs.

If you are trying to use a small (less than 5MB) codecache, you must consider `CodeCacheMinimumFreeSpace`. For larger codecaches, leave the default value alone. Generally, the JIT keeps enough space free in the codecache to honor this option. For a small code cache, add CodeCacheMinimumFreeSpace to your new `ReservedCodeCacheSize`. As an example, suppose:

```
max_used = 3M
CodeCacheMinimumFreeSpace = 500k
```

To reduce the codecache size from 3MB to 2MB, increase `ReservedCodeCacheSize` to 2500k (2M+500K) After making the change, verify that `max_used` changes to 2M.

When constraining the codecache, usually `CodeCacheMinimumFreeSpace` can be set to a lower value. However, `CodeCacheMinimumFreeSpace` should be at least 100KB. If free space is exhausted, the JVM throws `VirtualMachineError` and exits, or in rare cases, crashes. For the 3MB to 2MB example, the following settings are appropriate:

```
-XX:ReservedCodeCacheSize=2100k
-XX:CodeCacheMinimumFreeSpace=100k
```

Finding the optimal `ReservedCodeCacheSize` for your needs is an iterative process. You can repeatedly use smaller and smaller values for `ReservedCodeCacheSize` until your application's performance degrades unacceptably, and then increase until you get acceptable performance again. You should also gauge the incremental return you are achieving. You might find that you can decrease `max_used` by 50% with only a 5% performance degradation, and decrease `max_used` by 60% with a 10% performance degradation. In this example, the second 10% codecache reduction cost as much performance as the initial 50% codecache reduction. You might conclude in this case that the 50% reduction a good balance between codecache usage and performance.

Reducing Compilations

Reducing the number of compiled methods, or the rate at which they are compiled, is another effective way of reducing the amount of codecache that is used. Two main command line options that affect how aggressively methods are compiled: `CompileThreshold` and `OnStackReplacePercentage`. `CompileThreshold` relates to the number of method invocations needed before the method is compiled. `OnStackReplacePercentage` relates to the number of backwards branches taken in a method before it gets compiled, and is specified as a percentage of `CompileThreshold`.

When a method's combined number of backwards branches and invocations reaches or exceeds `CompileThreshold` * `OnStackReplacePercentage` / 100, the method is compiled. Note that there is also an option called `BackEdgeThreshold`, but it currently does nothing. Use `OnStackReplacePercentage` instead.

Larger values for these options decreases compilations. Setting the options larger than their defaults defers when a method gets compiled (or recompiled), possibly even preventing a method from ever getting compiled. Usually, setting these options to larger values also reduces performance (methods are interpreted), so it is important to monitor both performance and codecache usage when you adjust them. For the client JVM, tripling the default values of these options is a good starting point. For the server JVM, `CompileThreshold` is already set fairly high, so probably does not need to be adjusted further.

Reducing Compiled Method Sizes

There are a number of command-line options that reduce the size of compiled methods, but generally at some performance cost. Like the codecache reduction methods described in other sections, the key is finding a setting that gives good code cache usage reduction, without much performance loss.

The options described in this section all relate to reducing the amount of inlining the compiler does. Inlining is when the compiler includes the code for a called method into the compiled code for the method being compiled. Inlining can be done many levels deep, so if method `a()` calls `b()` which in turn calls `c()`, then when compiling `a()`, `b()` can be inlined in `a()`, which in turn can trigger the inlining of `c()` into `a()`.

The JIT compiler uses multiple heuristics to determine if a method should be inlined. In general, the heuristics are tuned for optimal performance. However, you can adjust some of them to sacrifice some performance for less codecache usage. The more useful options to tune are described below:

InlineSmallCode

The value of this option determines how small an already compiled method must be for it to be inlined when called from a method being compiled. If the compiled version of this method is bigger than the setting for `InlineSmallCode`, then it is not inlined. Instead, a call to the compiled version of the method is generated.

MaxInlineLevel

This option represents the maximum nesting level of the inlining call chain. Inlining becomes much less useful at deeper levels, and can eventually be harmful to performance due to code bloat. The default value is 9. Setting `MaxInlineLevel` as low as 1 or 2 ensures that trivial methods, such as getters and setters, are inlined.

MaxInlineSize

This option represents the maximum bytecode size of an inlined method. It defaults to 35, but is automatically reduced at each inlining level. Setting it very small (around 6) ensures that only trivial methods are inlined.

MinInliningThreshold

The interpreter tracks invocation counts at method call sites. This count is used by the JIT to help determine if a called method should be inlined. If a method's number of invocations is less than `MinInliningThreshold`, the method is not inlined. Raising this threshold reduces the number of method call sites that are inlined. Trivial methods are always inlined and are not subject to the setting of `MinInliningThreshold`.

InlineSynchronizedMethods

This option can be used to disable the inlining of methods that are declared as `synchronized`. Because synchronizing is a fairly expensive operation, especially on multi-core devices, the benefits of inlining even small synchronized methods is greatly diminished. You might find you can disable the inlining of synchronized methods with little or no perceived performance degradation, but with a noticeable reduction in the codecache usage.

Part IV

Appendixes

The following appendixes are included in this guide

For Platform Developers:

- Preparing a Device for JavaFX

For Application Developers

- Developing Embedded Applications in NetBeans IDE

A

Preparing a Device for JavaFX

If you want to run headful applications on your target device using the JavaFX API, there is some extra configuration required. This appendix shows platform developers how to prepare the BeagleBoard-xM as an example device.

This chapter contains the following sections:

Preparing the BeagleBoard-xM for JavaFX Applications

To run JavaFX applications on the BeagleBoard-xM, you must configure its software as described in this section. Configuration includes building an operating system, which can take a few hours.

Use a Suitable Build Machine

Use a 32-bit Ubuntu build machine with at least 8 GBytes of free space for a kernel build or about 30 GBytes free for a full system build with GNOME.

> **Note:** Do not build on a VirtualBox shared folder. Vboxfs does not support the creation of hard links, which the build system needs.

Install the Tools and Configure the System

1. Type the following command to install the build tools:

```
$ sudo apt-get install \
build-essential chrpath coreutils corkscrew cvs desktop-file-utils \
diffstat docbook-utils git-core help2man libncurses5-dev \
subversion texi2html texinfo
```

2. Set the default shell to `bash`:

```
$ sudo dpkg-reconfigure dash
```

3. When asked if `dash` should be the default shell, answer **No**.

Obtain the Ångström/Open Embedded Scripts

Clone the Ångström setup repository:

```
$ git clone http://github.com/Angstrom-distribution/setup-scripts.git
```

```
$ cd setup-scripts/
```

Set Up the BeagleBoard

In the `setup-scripts` directory, run the following commands:

```
$ MACHINE=beagleboard ./oebb.sh config beagleboard
$ MACHINE=beagleboard ./oebb.sh update
```

Build the Distribution

In the `setup-scripts` directory, run the following commands. The setup scripts take several hours to run.

```
$ . ~/.oe/environment-angstromv2012.12
$ bitbake console-image
```

Prepare the SD Card and Write File Systems

1. In the `setup-scripts` directory, run the following commands, but replace */dev/sdX* with the correct device for an SD card connected to your Linux PC.

   Note: Use `uboot.img` from the Ångström website instead of the one you built. If you use the one you built, then the USB input devices do not work.

```
$ mkdir -p /tmp/boot /tmp/rootfs
$ sudo -s /
sources/meta-angstrom/contrib/omap3-mkcard.sh /dev/sdX

$ mount /dev/sdX1 /tmp/boot ; mount /dev/sdX2 /tmp/rootfs
$ cd build/tmp-angstrom_v2012_12-eglibc/deploy/images/beagleboard/
$ tar jxf console-image-beagleboard.tar.bz2 -C /tmp/rootfs
$ tar zxf modules-3.2.28-r122a-beagleboard.tgz -C /tmp/rootfs
$ cp MLO /tmp/boot

$ wget http://www.beagleboard.org/angstrom-mirror/ \
www.angstrom-distribution.org/demo/beagleboard/u-boot.img \
-O /tmp/boot/u-boot.img

$ cat > /tmp/boot/uEnv.txt << EOF
$ vram=24M

$ dvimode="1280x800MR - 32@60 mem=99M@0x80000000 mem=384M@0x88000000 \
omapfb.vram=0:12M,1:8M,2:4M"

$ optargs="consoleblank=0"
$ console="console=ttyO2,115200n8"
$ mmcroot="/dev/mmcblk0p2"
$ EOF
```

2. Unmount the file system:

```
$ sync
$ umount /tmp/boot /tmp/rootfs
```

Boot a BeagleBoard xM with the Card

The first boot takes a long time. Slower SD cards take a very long time for the first boot. If you have a serial console attached, you will see it pause for 30 to 90 minutes after the following output. On the display, you see only the lines that start with time stamps.

```
Starting Recreate Volatile Files and Directories...
Starting Load Random Seed...
Started Machine ID first boot configure                              [  OK  ]
Starting Run pending postinsts...
Started Load Random Seed                                             [  OK  ]
Started Recreate Volatile Files and Directories                      [  OK  ]
[    5.725891] usb 1-2.2.1.3: New USB device found, idVendor=413c, idProduct=2106
[    5.735839] usb 1-2.2.1.3: New USB device strings: Mfr=1, Product=2,
SerialNumber=0
[    5.746093] usb 1-2.2.1.3: Product: Dell QuietKey Keyboard
[    5.753997] usb 1-2.2.1.3: Manufacturer: Dell
[    6.739074] twl_rtc twl_rtc: Power up reset detected.
[    6.753692] twl_rtc twl_rtc: Enabling TWL-RTC.
[    6.779571] twl_rtc twl_rtc: rtc core: registered twl_rtc as rtc0
[    7.904846] smsc95xx v1.0.4
[    8.042388] smsc95xx 1-2.1:1.0: eth0: register 'smsc95xx' at
usb-ehci-omap.0-2.1, smsc95xx USB 2.0 Ethernet, 9e:34:b3:e9:1c:9d
[    8.058837] usbcore: registered new interface driver smsc95xx
```

When the device finishes the initial setup, log in as `root` with an empty password.

Install the Required Packages

Execute the following commands to install the required packages:

```
$ opkg update

$ opkg install libgles-omap3 omap3-sgx-modules gdm liberation-fonts udev \
gcc gcc-symlinks libc6-dev binutils make g++ g++-symlinks \
libstdc++-dev libstdc++6

$ opkg install ntpdate
$ ln -sf /lib/libudev.so.1 /lib/libudev.so.0
$ depmod -a
$ reboot
```

Build and Install DirectFB

1. Download and unzip the `DirectFB` source files from:

   http://www.directfb.org/downloads/Core/DirectFB-1.4/DirectFB-1.4.7.tar.gz.

2. Go to the created `directfb` folder and run:

```
./configure --with-inputdrivers=none --with-gfxdrivers=none \
  --without-tools --disable-static \
  --prefix=/usr \
  AR=/usr/arm-angstrom-linux-gnueabi/bin/ar
make
make install
```

Note: Software rendering with DirectFB for JavaFX applications requires the following option on the `java` command line:

```
-Djavafx.platform=directfb
```

Disable Cursor Blinking

By default, the BeagleBoard is configured for the console cursor to blink, which can cause flickering and visual defects, especially on a touchscreen device. Disable the blinking with these commands:

```
$ cat > /etc/init.d/configure_cursorblink << EOF
#!/bin/sh
/bin/echo 0 > /sys/devices/virtual/graphics/fbcon/cursor_blink
EOF
$ chmod 755 /etc/init.d/configure_cursorblink
$ ln -sf /etc/init.d/configure_cursorblink etc/rc5.d/S99configure_cursorblink
$ /etc/init.d/configure_cursorblink
```

Update the Graphics Drivers (Recommended)

Updating the graphics drivers is optional, but recommended because the update resolves some rendering issues. Perform the update on the 32-bit x86 Linux machine on which you created the image.

1. Download the TI graphics SDK version 4.09.00.01 from http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/gfxsdk/latest/index_FDS.html.

2. Make sure you have an ARM cross-compiler on your system.

   You need a cross-compiler for ARM installed on your system. The recommended compiler is Sourcery CodeBench Lite Edition for ARM GNU/Linux, which is available at: http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/.

   The following instructions assume that the 2009q1 version of this compiler is installed in `/opt`.

3. Execute the following commands to install the graphics drivers:

   ```
   cd ~/setup-scripts/build/tmp-angstrom_v2012_12-eglibc/sysrootfs \
     /beagleboard/usr/src/kernel
   sudo make ARCH=arm BUILD_BUILDHOST=x86
   CROSS_COMPILE=/opt/arm-2009q1/bin/arm-none-linux-gnueabi- scripts
   ```

   (Point `CROSS_COMPILE` to the prefix to add to all cross-compilation commands. For example, `/opt/arm-2009q1/bin/arm-none-linux-gnueabi- `).

   Note: There is no space between `/opt/arm-2009q1` and `/bin/arm`, but there **is** a space between `arm-none-linux-gnueabi-` and `scripts`.

4. Edit the `Rules.make` file in the TI graphics SDK directory.

   Set `CSTOOL_DIR` to point to the root of your BeagleBoard tools. For example, `/opt/arm-2009q1`.

Set HOME to point to the directory above the <InstallationOfGraphicsSDK>. For example, if the SDK is installed in /home/someguy/Graphics_SDK_4_09_00_01, then set HOME=/home/someguy. For example:

```
set CSTOOL_DIR /opt/arm-2009q1
set CSTOOL_PREFIX=arm-none-linux-gnueabi-
set HOME DirectoryAboveGRAPHICS_INSTALL_DIR
```

5. Go to the TI graphics SDK (GRAPHICS_INSTALL_DIR) directory and execute the following commands to build the drivers:

```
sudo make all_km OMAPES=5.x KERNEL_INSTALL_DIR= \
  setup-scripts/build/tmp-angstrom_v2012_12-eglibc/ \
  sysrootfs/beagleboard/usr/src/kernel modules
```

6. Shut down your BeagleBoard xM and attach the SD card to your x86 Linux machine.

   Folders should be automatically mounted to the /media/$USER/Angstrom and /media/$USER/boot folders. If they are not, then mount them manually.

   Note: On older Ubuntu systems, file systems are mounted on /media/Angstrom and /media/boot.

7. Execute the following commands in the TI graphics SDK directory:

```
cd gfx_rel_es5.x
sudo cp *.so /media/$USER/Angstrom/usr/lib
sudo cp *.ko /media/$USER/Angstrom/lib/modules/3.2.28/kernel/drivers/gpu/pvr
sudo mkdir -p /media/$USER/Angstrom/usr/local/bin
sudo cp pvrsrvctl /media/$USER/Angstrom/usr/local/bin
sudo rm /media/$USER/Angstrom/etc/init.d/pvr-init
sudo cp rc.pvr /media/$USER/Angstrom/etc/init.d/pvr-init
sudo cp sgx* /media/$USER/Angstrom/usr/bin
sudo sync
sudo umount /media/$USER/Angstrom /media/$USER/boot
```

8. Boot the BeagleBoard xM with the updated SD card.

Disable the GDM (X Login Manager)

To disable the GDM, type the following command:

```
systemctl disable gdm.service
```

Configure the SGX Driver (Optional)

The default configuration of the SGX driver is to write directly to the frame buffer. This is fast, but can result in graphic artifacts, such as tearing or partial updates during animation. If you encounter problems with this setting, you can modify the file /etc/powervr.ini on the BeagleBoard, changing the line:

```
WindowSystem=libpvrPVR2D_FRONTWSEGL.so
```

to:

```
WindowSystem=libpvrPVR2D_FLIPWSEGL.so
```

B

Developing Embedded Applications in NetBeans IDE

Learn how to use NetBeans IDE to profile, run, and debug your embedded applications on both host and target.

This appendix contains the following topics:

Oracle Java SE Embedded Support in NetBeans IDE

NetBeans IDE offers support for Oracle Java SE Embedded in the following ways.

- You can set compact profiles so that the IDE will display violations in the source code. See `http://wiki.netbeans.org/CompactProfiles`
- You can run and debug your application on either the host or target within the IDE. See `http://wiki.netbeans.org/JavaSEEmbeddedHowTo`
- If you need to create a custom JRE for your own device, you can use the `jrecreate` support in NetBeans. See `http://wiki.netbeans.org/JavaSEEmbeddedHowTo#Creating_.26_Uploading_new_Embedded_JRE`
- As with all Java applications, you have instant access to Java SE API documentation, which includes information about compact profiles.

Related Links

- NetBeans IDE website: `http://netbeans.org/`
- JDK8 Support in NetBeans: `http://wiki.netbeans.org/JDK8`
- Ten-minute Hello World tutorial: `http://netbeans.org/kb/docs/java/quickstart.html`

Remote Debugging

You can remotely debug an application with NetBeans IDE, provided that the target's JRE was created with the `--debug` option and the client or server JVM. Examine the target JRE's `bom` file to see the JRE's configuration. For information about the `--debug` option of `jrecreate` and the `bom` file, see jrecreate Options.

Start the application in debug mode on the target (see Launch the Application with the java Launcher Tool for an example). With the application waiting for a remote

connection, in the NetBeans IDE on your host, choose **Debug** and then **Attach Debugger**. Fill in the **Attach Debugger** dialog fields similar to those in Figure B–1, substituting the target's IP address and the port you specified when you launched the application.

Figure B–1 Example Attach Debugger Dialog