Oracle® Java Micro Edition Connected Device Configuration

Runtime Guide

Release 1.1.2 for Oracle Java Micro Edition Embedded Client

A12345-01

May 2011

This runtime guide describes how to use a Java runtime environment for the Oracle Java Micro Edition Embedded Client that is based on Java Micro Edition Connected Device Configuration with its related profiles and optional packages. It focuses on runtime issues such as deployment, configuration and running application software based on Java technology, in addition to developer issues such as compiling, debugging, and profiling.

Oracle Java Micro Edition Connected Device Configuration Runtime Guide, Release 1.1.2 for Oracle Java Micro Edition Embedded Client.

A12345-01

Contents

List of Examples

List of Figures

List of Tables

Preface

This runtime guide describes how to use a Java runtime environment based on Java Micro Edition Connected Device Configuration (CDC) version 1.1.2 with its related profiles and optional packages. It focuses on runtime issues like deployment, configuration and running application software based on Java technology, in addition to developer issues such as compiling, debugging, and profiling.

Audience

This runtime guide is intended for use within a product development context, including both runtime and application development. From a developer's perspective, runtime issues generally exercise configuration, testing or debugging features of the CDC Java runtime environment.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/support/contact.html` or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Related Documents

For more information, see the following documents:

- *CDC Build System Guide*
- *CDC Porting Guide*
- *CDC HotSpot Implementation Dynamic Compiler Architecture Guide*
- *CDC Technology Compatibility Kit User's Guide*
- *Security Optional Package Technology Compatibility Kit User's Guide*
- *Java Language Specification*. See http://java.sun.com/docs/books/jls.
- *Java Virtual Machine Specification*. See http://java.sun.com/docs/books/vmspec.
- *Java Native Interface (JNI)*. See http://java.sun.com/docs/books/jni.
- *Java Virtual Machine Tools Interface (JVMTI)*. See http://download.oracle.com/javase/1.5.0/docs/guide/jvmti.
- *Inside Java 2 Platform Security*. See http://java.sun.com/docs/books/security.
- *Java ME Unified Emulator Interface Specification* (UEI). See http://java.sun.com/j2me/docs/uei_specs.pdf.

Conventions

The following text conventions are used in this document:

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

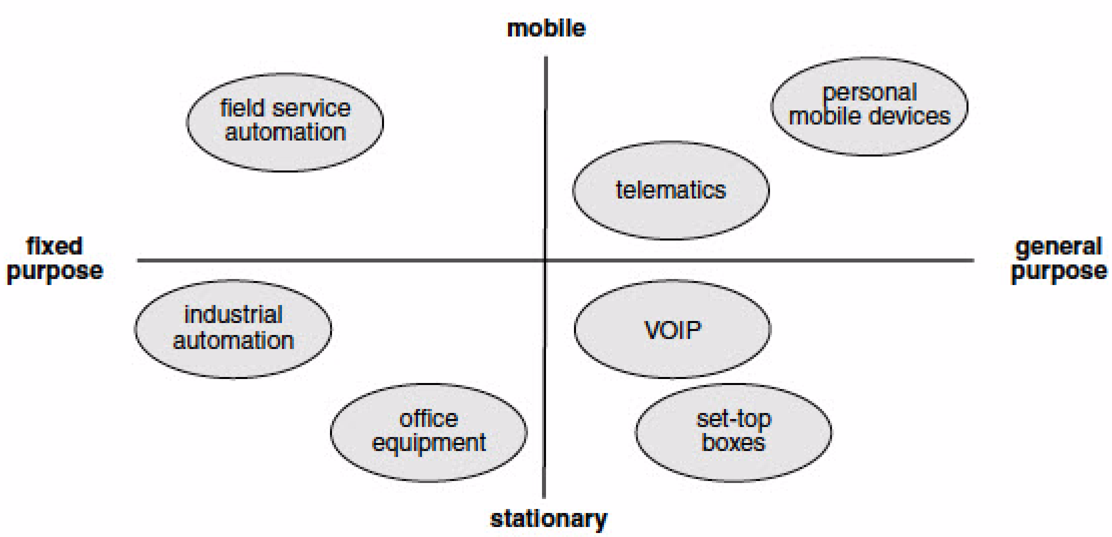
1 Introduction

A Java runtime environment is an implementation of Java technology for a specific target platform. It performs a middleware function with features common to a native application: it is installed, launched and run like a native application. But its real purpose is to launch, run and manage Java application software on the target platform.

The CDC Java runtime environment is an implementation of Java technology for connected devices. These include mobile devices like PDAs and smart phones in addition to attached devices like set-top boxes, printers and kiosks.

CDC target devices can vary widely based on their features and purpose. Figure 1–1 describes some CDC target device categories and organizes them by their two most important characteristics: purpose (fixed or general) and mobility (mobile or attached).

Figure 1–1 CDC Target Device Categories



This runtime guide describes how to use the CDC Java runtime environment for different purposes including application development, runtime development and solution deployment.

This chapter briefly introduces the CDC Java runtime environment through the following:

- Goals

- Usage Contexts
- CDC Technology Implementations
- CDC Target Device Requirements
- Java Micro Edition Technology Standards
- Java ME API Choices
- CDC Application Features
- Developer Tools

1.1 Goals

It is difficult to describe CDC technology without reference to the Java Standard Edition (Java SE) platform because Java SE represents the core of Java technology. In fact, the principal goal of CDC is to adapt Java SE technology from desktop systems to connected devices. Most of CDC's modifications to Java SE APIs are based on identifying features that are either too large or inappropriate for CDC target devices and then either removing or making them optional.

Other related goals of CDC include the following:

- Broaden the number of target devices for Java application software.
- Take advantage of target device features while fitting within their resource limitations.
- Provide a runtime implementation optimized for connected devices.
- Leverage Java SE developer tools, skills and technology.

1.2 Usage Contexts

The CDC Java runtime environment described in this runtime guide can operate in several different usage contexts:

- During *product development*, the CDC Java runtime environment has testing features that can help isolate problems while porting CDC technology to a new target platform. For example, the trace features provide details about opcode and method execution in addition to garbage collection (GC) state.
- One of the final stages of product development is *TCK verification*. A TCK is a test suite that verifies the behavior of an implementation of Java technology. The TCK includes a test harness that runs a candidate Java runtime environment and launches a series of test Java applications. TCK verification is described in the TCK user guides listed in the *CDC Technology Compatibility Kit User's Guide*.
- *Application development* for the CDC platform requires a target Java class library for compiling Java source code and a CDC Java runtime environment for testing and debugging. Chapter 6, "Developer Tools" provides more information about application development with the CDC Java runtime environment.
- When an application is complete and tested, it's ready for *deployment*. CDC provides a number of deployment mechanisms including preloading with `JavaCodeCompact`, managed application models like xlets and network-based provisioning systems.

1.3 CDC Technology Implementations

CDC technology is delivered by Oracle through different kinds of software releases:

- A *Reference Implementation* (RI) demonstrates Java technology that is described in a *Java Specification Request* (JSR) and verified by a corresponding *Technology Compatibility Kit* (TCK). Because it serves a demonstration purpose, an RI does not provide the best available performance features.

  The sample RI described by this guide is based on the Linux/x86/Qt platform.
- An *Optimized Implementation* (OI) is also a TCK-compliant implementation of Java technology. An OI provides the following benefits:
  - Undergoes more quality assurance (QA) testing
  - Provides superior performance
  - Supports a strategic platform or can be used as a starting point for porting Java technology to a different target platform

1.4 CDC Target Device Requirements

CDC is an adaptable technology that can support a range of connected target devices that exist today and in the future. The following are the baseline system requirements of these connected devices:

- Network connectivity
- 32-bit RISC-based or x86 microprocessor

The memory requirements for a CDC Java runtime environment vary based on the native platform, the profile and optional packages and the application. See Section 3.4, "Memory Management" for memory usage guidelines.

Other features of the CDC target device can include:

- A display for a graphical user interface (GUI)
- Unicode font support
- An open or proprietary native platform that provides operating system services

1.5 Java Micro Edition Technology Standards

CDC is part of the family of Java Micro Edition (Java ME) technology standards that support application software for connected devices. From an application developer's perspective, CDC is a standards-based framework for creating and deploying application software on a broad range of consumer and embedded devices. The CDC APIs are largely based on well-known Java SE APIs, which makes the job of migrating skills, tools and source code easier. From a product designer's perspective, CDC provides a standards-based Java runtime environment that supports a variety of target devices. This allows product designers to provide an application platform that fits within their device's resource limitations while supporting a large number of applications and developers.

Java ME standards are developed in collaboration with industry leaders through the Java Community Process (`http://jcp.org`). JCP standards allow Java technology to adapt to the needs of evolving products in an open way by defining APIs that address common needs in application development. Furthermore, these standards allow product designers to choose which API features fit their product needs.

Java ME technology uses three kinds of API standards described in Table 1–1 as building blocks that can be combined in a specific product solution.

Table 1–1 Java ME API Standards

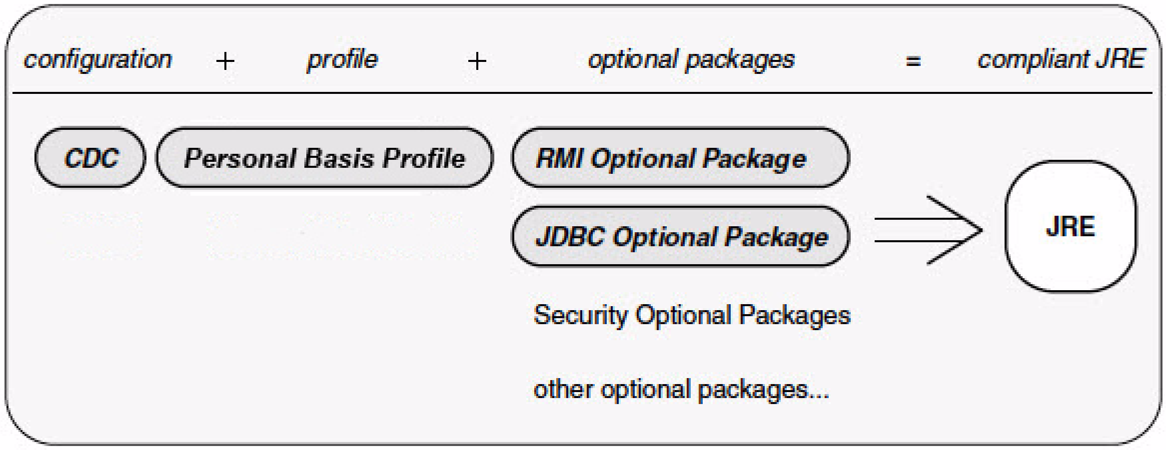
Category	Description	Options
Configuration	Defines the most basic Java class library and Java virtual machine capabilities for a broad range of devices.	*Connected Device Configuration* (CDC, JSR-218) supports connected devices like smart phones, set-top boxes and office equipment.
Profile	Defines additional APIs that support a narrower range of devices. A profile is built on a specific configuration.	*Personal Basis Profile* (JSR-217) provides a standards-based GUI framework for supporting lightweight components. Personal Basis Profile adds support for the xlet application model.
Optional Package	Defines a set of technology-specific APIs.	■ The *Remote Method Invocation (RMI) Optional Package* (JSR-66) provides a subset of the Java SE RMI API for networked devices based on Java technology. It exposes distributed application protocols through Java interfaces, classes and method invocations and shields the developer from the details of network communications. ■ The *Java Database Connectivity (JDBC) Optional Package* (JSR-169) provides a subset of the JDBC 3.0 API that can be used by Java application software to access tabular data sources including spreadsheets, flat files and cross-DBMS connectivity to a wide range of SQL databases. ■ The Security Optional Packages (part of JSR-219) include *Java Secure Socket Extension (JSSE) Optional Package*, the *Java Cryptography Extension (JCE) Optional Package* and the *Java Authentication and Authorization Service (JAAS) Optional Package*. These provide Java SE APIs for extending CDC's security architecture. ■ The *Web Services Optional Package* (JSR-172) provides standard access from Java ME clients to web services.

1.6 Java ME API Choices

Each Java ME licensee can create a Java runtime environment by choosing from a menu of standard APIs. The designer's choice must contain a configuration, and optionally, a profile and any number of optional packages. These choices can vary from product to product. The critical point to understand is that the application developer must separately learn about which API combinations are available for a specific CDC product implementation.

For example, Figure 1–2 describes a Java runtime environment where a product designer selects CDC, RMI Optional Package, and JDBC Optional Package to represent a conforming CDC Java runtime environment.

Figure 1–2 An Example CDC Java Runtime Environment



Note: See the companion document *CDC Build System Guide* for information on how to build a target development version of the CDC Java class library for application development that reflects the APIs chosen for a specific target product. Chapter 6, "Developer Tools" describes how to compile Java application software with such a library.

1.7 CDC Application Features

The applications targeted by CDC technology have certain characteristics that distinguish them from the productivity tools and utilities common to desktop platforms.

- *Network connectivity.* The dominant trends in application development, like web browsers, XML-based web services and RSS, are based on network connectivity. Examples include the evolution of PDAs and cell phones into connected devices and the evolution of office printers into multi-function peripherals that can generate campus-specific reports.
- *Security.* Application developers and users are becoming increasingly aware of the need for security for their mobile and distributed applications. The Java SE security framework in CDC allows applications to use fine-grained security policies for application and enterprise security needs.
- *Application deployment.* Java technology has traditionally provided flexible application models. CDC profiles support managed application models like xlets that allow developers to easily deploy applications over the network, either directly or through a provisioning server.
- *Standard data access*. Mobile clients need access to central databases to view and modify information. The JDBC and web Services optional packages provide standard data access for client-side applications.
- *Portable GUIs.* With the broad range of CDC target devices, applications need a GUI system that is flexible enough different user experiences and workflows while being portable enough to support different target devices. Personal Basis Profile supports conventional AWT-based GUIs in addition to providing a hosting layer

for building and supporting GUIs based on industry-standards and vendor-specific interfaces.

1.8 Developer Tools

Because CDC APIs are derived from Java SE APIs, application developers can migrate both their software and their skills to the CDC platform with little effort. Java SE developers can easily learn CDC APIs by focusing on their small differences with Java SE APIs. It is therefore easy to modify Java SE software for CDC devices. The ability to use Java SE developer tools like compilers, debuggers and profilers makes this transition easier.

The CDC Java runtime environment uses several developer tool-oriented specifications, including the following:

- Because CDC is based on the *Java Virtual Machine Specification* (see `http://java.sun.com/docs/books/vmspec`), application developers can use conventional Java SE compilers like `javac`.
- The *Java Virtual Machine Tools Interface* (JVMTI, see `http://download.oracle.com/javase/1.5.0/docs/guide/jvmti`) defines an interface that allows developer tools like debuggers and profilers to control and measure runtime data for a specific application or benchmark.
- The *Java ME Unified Emulator Interface Specification* (UEI, see `http://java.sun.com/j2me/docs/uei_specs.pdf`) defines an interface that allows an external developer tool to control a Java ME emulator.
- `cvm`, the CDC application launcher, uses many command-line options that are available with `java`, the Java SE application launcher. Many of these options can be used for application testing and development.

Java SE tools like `jar` and `keytool` can also be used in CDC application development and deployment.

2 Software Layout

A CDC Java runtime environment contains the software necessary to run Java applications on a target platform. The software contents of a CDC Java runtime environment can vary, especially during product development when different testing options may be selected at build-time. This chapter describes the organization of a CDC Java runtime environment, including standard files in addition to optional security, developer and test files.

2.1 Standard Files

After installation of CDC 1.1.2, the CDC Java runtime environment is located in its installation directory. Because the location of this installation directory can be anywhere in the local file system, the CDC Java runtime environment specifies this location with the `java.home` system property. Table 2–1 describes the standard files located in the installation directory based on the default build options.

Table 2–1 Standard Files

File	Description
`bin/cvm`	The CDC Java application launcher loads and executes Java applications. *Note*: For the Oracle Java Micro Edition (Java ME) Embedded Client, `cvm` is located in *InstallDir*/*Oracle_JavaME_Embedded_Client*/`binaries/bin`. See the *Oracle Java Micro Edition Embedded Client Reference Guide* for detailed information about `cvm` for the Oracle Java ME Embedded Client.
`lib/`*`class-lib`*`.jar`	*Optional.* The CDC Java class library is used by the CDC Java runtime environment to locate and load core Java classes. The actual name of the archive file indicates the supported CDC specifications, e.g. `cdc.jar`, `foundation-rmi.jar`. *Note:* `lib/`*class-lib*`.jar` is only present for non-preloaded builds.
`lib/content-types.properties`	The MIME content type system property table used by the `sun.net.www` package. Each entry maps a MIME content type to a native application that can handle it. Files are associated with a MIME content type by either the MIME content type returned by an HTTP header or their file name extension.

Table 2–1 (Cont.) Standard Files

File	Description
`lib/security/java.policy`	System-wide security policies.[1]
`lib/security/java.security`	Master security properties.[2]

`lib/zi/America/Los_Angeles`
`lib/zi/Asia/Calcutta`
`lib/zi/Asia/Novosibirsk`
`lib/zi/GMT`
`lib/zi/ZoneInfoMappings`
	Time zone data files used by `sun.util.calendar.ZoneInfoFile`.

1 See *Inside Java 2 Platform Security, Second Edition: Architecture, API Design, and Implementation* by Li Gong (Addison-Wesley, 2003) for more information about Java SE security features.

2 See *Inside Java 2 Platform Security, Second Edition: Architecture, API Design, and Implementation* by Li Gong (Addison-Wesley, 2003) for more information about Java SE security features.

2.2 Security Files

Table 2–2 describes optional security files in versions of the CDC Java runtime environment that include the security optional packages. See *Inside Java 2 Platform Security: Architecture, API Design, and Implementation* by Li Gong (second edition, Addison-Wesley, 2003) for more information about Java SE security features.

Table 2–2 Security Files

File	Description
`lib/jaas.jar`	*Java Authentication and Authorization Service (JAAS) Optional Package* is a part of JSR-219 which is a framework for enforcing access control to resources using a CodeSource-based and Subject-based security model. `jaas.jar` contains the JAAS Optional Package implementation and the `KeyStoreLoginModule` authentication module, which is a subset of what is available in J2SE version 1.4.2.

`lib/jce.jar`
`lib/ext/sunjce_provider.jar`
`lib/sunrsasign.jar`
	Java Cryptography Extension (JCE) Optional Package is a part of JSR-219 which extends the Java Cryptography Architecture (JCA) to include key generation and agreement, encryption and message authentication code (MAC) generation services. `jce.jar` contains the JCE Optional Package implementation which is fully compatible with J2SE version 1.4.2.
	
	`sunjce_provider.jar` contains the default provider implementation of the JCE service provider interface (SPI) and is fully compatible with J2SE version 1.4.2. Note that `lib/ext` is part of the extension class search path, but not part of the system class search path. See Section 3.3, "Class Search Path Basics" for more information about class search paths.
	
	`sunrsasign.jar` contains the default provider implementation of the RSA signature SPI and is fully compatible with the default provider implementation in J2SE version 1.4.2. See "How to Implement a Provider for the Java Cryptography Architecture" in JSR-219.

`lib/jsse-cdc.jar`	*Java Secure Socket Extension (JSSE) Optional Package* is a part of JSR-219 which provides support for secure communication. `jsse.jar` contains both the JSSE Optional Package implementation and the default provider implementation, which is fully compatible with the default provider implementation in J2SE version 1.4.2.
`lib/security/cacerts`	Certificate authority (CA) keystore file. The default keystore password is "`changeit`". See `keytool`(1) for more information about how to use the Java SE SDK key and certificate management tool to change the keystore password.
`lib/security/local_policy.jar` `lib/security/US_export_policy.jar`	Security jurisdiction policy files.

2.3 Development Files

Table 2–3 describes files that can be used with developer tools like compilers and debuggers. These files are further described in Chapter 6.

Table 2–3 Development Files

File	Description
lib/btclasses.zip	The CDC Java class library can be used for compiling application source code. **Note**: Because the contents of these archive files can vary depending on the selected build options, application development must be based on a target development version of the CDC Java class library. See the companion document *CDC Build System Guide* for information about how to build a target development version of the CDC Java class library.
lib/libdt_socket[_g].so lib/libjdwp[_g].so	The Java Debug Wire Protocol (JDWP) shared libraries are necessary for remote debugging.

3 Running Applications

The CDC Java runtime environment includes cvm, the CDC application launcher, for loading and executing Java applications. This chapter describes basic use of the cvm command to launch different kinds of Java applications, in addition to more advanced topics like memory management and dynamic compiler policies.

Note: For the Oracle Java Micro Edition (Java ME) Embedded Client, cvm is located in *InstallDir*/*Oracle_JavaME_Embedded_Client*/binaries/bin. See the *Oracle Java Micro Edition Embedded Client Reference Guide* for detailed information about using cvm for the Oracle Java ME Embedded Client.

3.1 Launching a Java Application

cvm, the CDC application launcher is similar to java, the Java SE application launcher. For the Oracle (Java ME) Embedded Client, see the *Oracle Java Micro Edition Embedded Client Reference Guide* for detailed information about using cvm to launch Java applications for the Oracle Java ME Embedded Client.

Many of cvm's command-line options are borrowed from java. The basic method of launching a Java application is to specify the top-level application class containing the main() method on the cvm command-line. For example,

```
% cvm HelloWorld
```

By default, cvm looks for the top-level application class in the current directory. Alternatively, the synonymous -cp and -classpath command-line options specify a list of locations where cvm searches for application classes instead of the current directory. For example,

```
% cvm -cp /mylib/archive.zip HelloWorld
```

Here cvm searches for HelloWorld in an archive file */mylib/archive*/.zip. See Section 3.3, "Class Search Path Basics" for more information about class search paths.

The -help option displays a brief description of the available command-line options. Appendix A provides a complete description of the command-line options available for cvm.

3.2 Running Managed Applications (*Personal Basis Profile only*)

Managed application models allow developers to offload the tasks of deployment and resource management to a separate application manager. The CDC Java runtime environment includes sample application managers for an xlet application model.

The xlet application model doesn't require an explicit dependency on AWT. These features make xlets appropriate for embedded device scenarios like set-top boxes and PDAs.

3.2.1 Running an Xlet (*Personal Basis Profile*)

The CDC Java runtime environment includes a simple xlet manager named `com.sun.xlet.XletRunner`. Xlets can be graphical, in which case the xlet manager displays each xlet in its own frame, or they can be non-graphical. The basic command syntax to launch `XletRunner` is:

```
% cvm com.sun.xlet.XletRunner { \
   -name xletName \
   (-path xletPath | -codebase urlPath) \
   -args arg1 arg2 arg3 ...} \
   ...
% cvm com.sun.xlet.XletRunner -filename optionsFile
% cvm com.sun.xlet.XletRunner -usage
```

Table 3–1 describes `XletRunner`'s command-line options:

Table 3–1 `XletRunner` ***Command-Line Options***

Option	Description
`-name` *`xletName`*	*Required.* Identifies the top-level Java class that implements the `javax.microedition.xlet.Xlet` interface.
`-path` *`xletPath`*	*Required* (or substituted with the `-codebase` option described below). Specifies the location of the target xlet with a local pathname. The path can be absolute or relative to the current directory. If the xlet is in a `jar` or Zip archive file, then use the archive file name. **Note:** The xlet *must not* be found in the system class path, especially when running more than one xlet, because xlets must be loaded by their own class loader.
`-codebase` *`urlPath`*	*Optional.* Specifies the location of the target xlet with a URL. The `-codebase` option can be substituted for `-path` to provide a URL-formatted path instead of a local pathname.
`-args` *`arg1 [arg2] [arg3]`* `...`	*Optional.* Passes additional runtime arguments to the xlet. Multiple arguments are separated by spaces.
`-filename` *`optionsFile`*	*Optional.* Reads options from an ASCII file rather than from the command line. The `-filename` option must be the first option provided to `XletRunner`.
`-usage`	Display a usage string describing `XletRunner`'s command-line options.

Here are some command-line examples for launching xlets with `com.sun.xlet.XletRunner`:

- To run *MyXlet* in *Myclasses*`.jar`:

  ```
  % cvm com.sun.xlet.XletRunner \
    -name basis.MyXlet \
    -path Myclasses.jar
  ```

- To run an xlet with multiple command-line arguments:

  ```
  % cvm com.sun.xlet.XletRunner \
    -name MyXlet \
    -path . \
  ```

```
-args top bottom right left
```

- To run more than one xlet, repeat the `XletRunner` options:

```
% cvm com.sun.xlet.XletRunner \
  -name ServerXlet -path ./server \
  -name ClientXlet -path ./client
```

- To run an xlet whose compiled code is at the URL http://*myurl.com*/*xlets*/*MyXlet*.class:

```
% cvm com.sun.xlet.XletRunner \
  -name MyXlet \
  -codebase http://myurl.com/xlets/
```

- To run an xlet in a `jar` file named `xlet.jar` with the arguments *colorMap* and *blue*, use the following command line:

```
% cvm com.sun.xlet.XletRunner \
  -name StockTickerXlet \
  -path xlet.jar \
  -args colorMap blue
```

- To run an xlet with command-line options in an argument file:

```
% cvm com.sun.xlet.XletRunner -filename myArgsFile
```

myArgsFile contains a text line with valid `XletRunner` options:

```
-name StockTickerXlet -path Myxlet.jar -args colorMap blue
```

3.3 Class Search Path Basics

The Java runtime environment uses various search paths to locate classes, resources and native objects at runtime. This section describes the two most important search paths: the Java class search path and the native method search path.

3.3.1 Java Class Search Path

Java applications are collections of Java classes and application resources that are built on one system and then potentially deployed on many different target platforms. Because the file systems on these target platforms can vary greatly from the development system, Java runtime environments use the *Java class search path* as a flexible mechanism for balancing the needs of platform-independence against the realities of different target platforms.

The Java class search path mechanism allows the Java virtual machine to locate and load classes from different locations that are defined at runtime on a target platform. For example, the same application could be organized in one way on a MacOS system and another on a Linux system. Preparing an application's classes for deployment on different target systems is part of the development process. Arranging them for a specific target system i s part of the deployment process.

The Java class search path defines a list of locations that the Java virtual machine uses to find Java classes and application resources. A location can be either a file system directory or a `jar` or Zip archive file. Locations in the Java class search path are delimited by a platform-dependent path separator defined by the `path.separator` system property. The Linux default is the colon ":" character.

The Java SE documentation[1] describes three related Java class search paths:

- The *system* or *bootstrap classes* comprise the Java platform. The *system class search path* is a mechanism for locating these system classes. The default system search path is based on a set of `jar` files located in *JRE*`/lib`.
- The *extension classes* extend the Java platform with optional packages like the JDBC Optional Package. The *extension class search path* is a mechanism for locating these optional packages. `cvm` uses the `-Xbootclasspath` command-line option to statically specify an extension class search path at launch time and the `sun.boot.class.path` system property to dynamically specify an extension class search path. The CDC default extension class search path is *CVM*`/lib`, except for some of the provider implementations for the security optional packages described in Table 2–2 which are stored in *CVM*`/lib/ext`. The Java SE default extension class search path is *JRE*`/lib/ext`.
- The *user classes* are defined and implemented by developers to provide application functionality. The *user class search path* is a mechanism for locating these application classes. Java virtual machine implementations like the CDC Java runtime environment can provide different mechanisms for specifying an Java class search path. `cvm` uses the `-classpath` command-line option to statically specify an Java class search path at launch time and the `java.class.path` system property to dynamically specify an user class search path. The Java SE application launcher also uses the `CLASSPATH` environment variable, which is *not* supported by the CDC Java runtime environment.

3.3.2 Native Method Search Path

The CDC HotSpot Implementation virtual machine uses the Java Native Interface[1] (JNI) as its native method support framework. The JNI specification leaves the platform-level implementation of native methods up to the designers of a Java virtual machine implementation. For the Linux-based CDC Java runtime environment described in this runtime guide, a JNI native method is implemented as a Linux shared library that can be found in the native library search path defined by the `java.library.path` system property.

> **Note:** The standard mechanism for specifying the native library search path is the `java.library.path` system property. However, the Linux dynamic linking loader may cause other shared libraries to be loaded implicitly. In this case, the directories in the `LD_LIBRRARY_PATH` environment variable are searched *without* using the `java.library.path` system property. One example of this issue is the location of the Qt shared library. If the target Linux platform has one version of the Qt. shared library in `/usr/lib` and the CDC Java runtime environment uses another version located elsewhere, this directory must be specified in the `LD_LIBRRARY_PATH` environment variable.

Here is a simple example of how to build and use an application with a native method. The mechanism described below is very similar to the Java SE mechanism.

1. Compile a Java application containing a native method.

[1]See the tools documentation at http://download.oracle.com/javase/1.4.2/docs/tooldocs/tools.html for a description of the J2SDK tools and how they use Java class search paths.

[1] See the *Java Native Interface: Programmer's Guide and Specification*.

```
% javac -boot class path lib/btclasses.zip HelloJNI.java
```

2. Generate the JNI stub file for the native method.

```
% Java -bootclasspath lib/btclasses.zip HelloJNI
```

3. Compile the native method library.

```
% gcc HelloJNI.c -shared -I${CDC_SRC}/src/share/javavm/export \
    -I${CDC_SRC}/src/linux/javavm/include -o libHelloJNI.so
```

   This step requires the CDC-based JNI header files in the CDC source release.

4. Relocate the native method library in the `test` directory.

```
% mkdir test
% mv libHelloJNI.so test
```

5. Launch the application.

```
% cvm -Djava.library.path=test HelloJNI
```

   If the native method implementation is not found in the native method search path, the CDC Java runtime environment throws an `UnsatisfiedLinkError`.

3.4 Memory Management

The CDC Java runtime environment uses memory to operate the Java virtual machine and to create, store, and use objects and resources. This section provides an overview of how memory is used by the Java virtual machine. Of course, the actual memory requirements of a specific Java application running on a specific Java runtime environment hosted on a specific target platform can only be determined by application profiling. This section, however, provides useful guidelines.

3.4.1 The Java Heap

When it launches, the CDC Java runtime environment uses the native platform's memory allocation mechanism to allocate memory for native objects and reserve a pool of memory, called the *Java heap*, for Java objects and resources.

- The size of the Java heap can grow and shrink within the boundaries specified by the `-Xmx`*size*, `-Xms`*size* and `-Xmn`*size* command-line options described in Table A–1.
- If the requested Java heap size is larger than the available memory on the device, the Java runtime environment exits with an error message:

```
% cvm -Xmx23000M MyApp
Cannot start VM (error parsing heap size command line option -Xmx)
Cannot start VM (out of memory while initializing)
Cannot start VM (CVMinitVMGlobalState failed)
Could not create JVM.
```

- If there isn't enough memory to create a Java heap of the requested size, the Java runtime environment exits with an error message:

```
% cvm -Xmx2300M MyApp
Cannot start VM (unable to reserve GC heap memory from OS)
Cannot start VM (out of memory while initializing)
Cannot start VM (CVMinitVMGlobalState failed)
Could not create JVM.
```

- If the application launches and later needs more memory than is available in the Java heap, the CDC Java runtime environment throws an `OutOfMemoryError`.
- The heap grows and shrinks between the -Xmn and -Xmx values based on heap utilization. This is true for Linux ports, but not all ports.

  For example,

  ```
  % cvm -Xms10M -Xmn5M -Xmx15M MyApp
  ```

  launches the application *MyApp* and sets the initial Java heap size to 10 MB, with a low water mark of 5 MB and a high water mark of 15 MB.

3.4.2 Garbage Collection

When a Java application creates an object, the Java runtime environment allocates memory out of the Java heap. And when the object is no longer needed, the memory should be recycled for later use by other objects and resources. Conventional application platforms require a developer to track memory usage. Java technology uses an automatic memory management system that transfers the burden of managing memory from the developer to the Java runtime environment.

The Java runtime environment detects when an object or resource is no longer being used by a Java application, labels it as "garbage" and later recycles its memory for other objects and resources. This *garbage collection* (GC) system frees the developer from the responsibility of manually allocating and freeing memory, which is a major source of bugs with conventional application platforms.

GC has some additional costs, including runtime overhead and memory footprint overhead. However, these costs are small in comparison to the benefits of application reliability and developer productivity.

3.4.2.1 Garbage Collection in the CDC HotSpot Implementation

The Java Virtual Machine Specification does not specify any particular GC behavior and early Java virtual machine implementations used simple and slow GC algorithms. More recent implementations like the Java HotSpot Implementation virtual machine provide GC algorithms tuned to the needs of desktop and server Java applications. And now the CDC HotSpot Implementation includes a GC framework that has been optimized for the needs of connected devices.

The major features of the GC framework in the CDC HotSpot Implementation are:

- *Exactness.* Exact GC is based on the ability to track all pointers to objects in the Java heap. Doing so removes the need for object handles, reduces object overhead, increases the completeness of object compaction and improves reliability and performance.
- *Default Generational Collector.* The CDC HotSpot Implementation Java virtual machine includes a generational collector that supports most application scenarios, including the following:
  - general-purpose
  - excellent performance
  - robustness
  - reduced GC pause time
  - reduced total time spent in GC

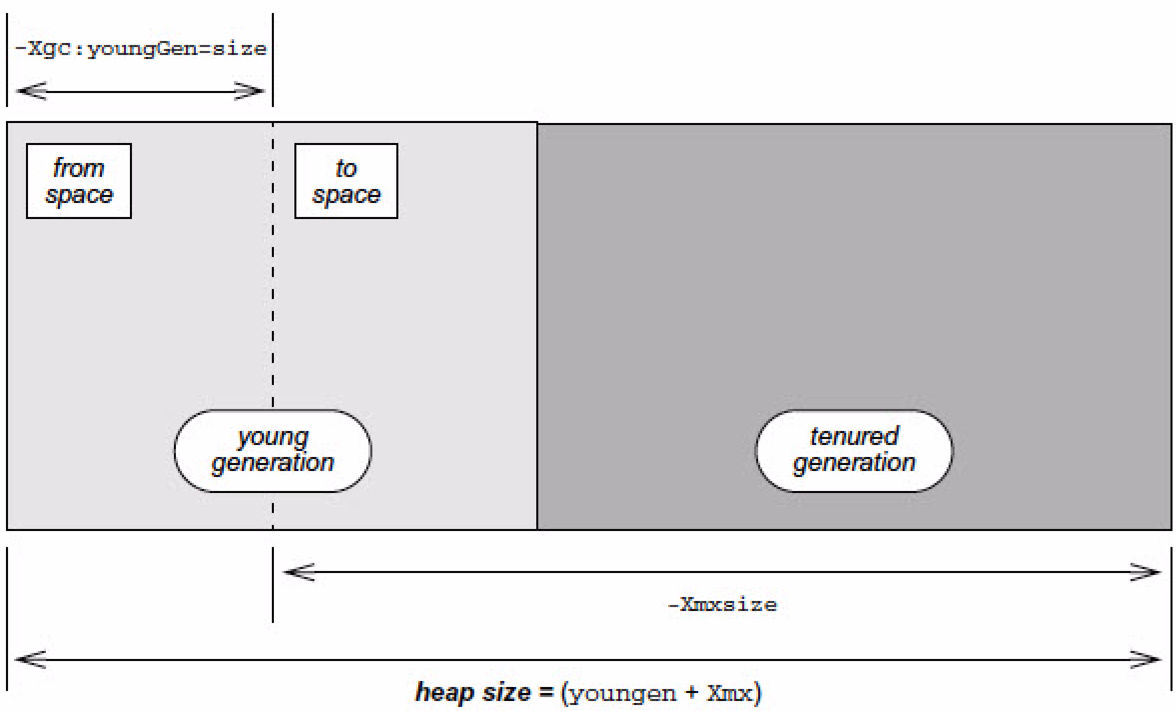
- *Pluggability.* While the default generational collector serves as a general-purpose garbage collector, the GC plug-in interface allows support for device-specific needs. Runtime developers can use the GC plug-in interface to add new garbage collectors *at build-time* without modifying the internals of the Java virtual machine. In addition, starter garbage collector plug-ins are available from Java Partner Engineering.

> **Note:** Needing an alternate GC plug-in is rare. If an application has an object allocation and longevity profile that differs significantly from typical applications (to the extent that the application profile cannot be catered to by setting the GC arguments), and this difference turns out to be a performance bottleneck for the application, then an alternate GC implementation may be appropriate.

3.4.2.2 Default Generational Collector

The default generational collector manages memory in the Java heap. Table 3–1 shows how the Java heap is organized into two heap generations, a *young* generation and a *tenured* generation. The generational collector is really a hybrid collector in that each generation has its own collector. This is based on the observation that most Java objects are short-lived. The generational collector is designed to collect these short-lived objects as rapidly as possible while promoting more stable objects to the tenured generation where objects are collected less frequently.

Figure 3–1 GC Generations



The young generation is based on a technique called *copying semispace*. The young generation is divided into two equivalent memory pools, the *from-space* and the *to-space*. Initially, objects are allocated out of the from-space. When the from-space becomes full, the system pauses and the young generation begins a collection cycle where only the live objects in the from-space are copied to the to-space. The two memory pools then reverse roles and objects are allocated from the "new" from-space. Only surviving objects are copied. If they survive a certain number of collection cycles (the default is 2), then they are promoted to the tenured generation.

The benefit of the copying semispace technique is that copying live objects across semispaces is faster than relocating them within the same semispace. This requires more memory, so there is a trade-off between the size of the young generation and GC performance.

The tenured generation is based on a technique called *mark compact*. The tenured generation contains all objects that have survived several copying cycles in the young generation. When the tenured generation reaches a certain threshold, the system pauses and it begins a full collection cycle where both generations go through a collection cycle. The young generation goes through the stages outlined above. Objects in the tenured generation are scanned from their "roots" and determined to be live or dead. Next, the memory for dead objects is released and the tenured generation goes through a compacting phase where objects are relocated within the tenured generation.

The default generational garbage collector reduces performance overhead and helps collect short-lived objects rapidly, which increases heap efficiency.

3.4.2.3 Tuning Options

The relative sizes of generations can affect GC performance. So the `-Xgc:youngGen` command-line option controls the size of the young object portion of the heap. See Table A–3 for more information about GC command-line options.

- `youngGen` should not be too small. If it is too small, partial GCs may happen too frequently. This causes unnecessary pauses and retain more objects in the tenured generation than is necessary because they don't have time to age and die out between GC cycles.

  The default size of `youngGen` is about 1/8 of the overall Java heap size.

- `youngGen` should not be too large. If it is too large, even partial GCs may result in lengthy pauses because of the number of live objects to be copied between semispaces or generations will be larger.

  By default, the CDC Java runtime environment caps `youngGen` size to 1 MB unless it is explicitly specified on the command line.

- The total heap size needs to be large enough to cater for the needs of the application. This is very application-dependent and can only be estimated.

3.4.3 Class Preloading

The CDC HotSpot Implementation virtual machine includes a mechanism called *class preloading* that streamlines VM launch and reduces runtime memory requirements. The CDC build system includes a special build tool called `JavaCodeCompact` that performs many of the steps at build-time that the VM would normally perform at runtime. This saves runtime overhead because class loading is done only once at build-time instead of multiple times at runtime. And because the resulting class data can be stored in a format that allows the VM to execute in place from a read-only file system (for example, Flash memory), it saves memory.

Note: It is important to understand that decisions about class preloading are generally made at build-time. See the companion document *CDC Build Guide* for information about how to use `JavaCodeCompact` to include Java class files with the list of files preloaded by `JavaCodeCompact` with the CDC Java runtime environment's executable image.

3.4.3.1 Class Preloading and Verification

Java class verification is usually performed at class loading time to insure that a class is well-behaved. This has both performance and security benefits. This section describes a performance optimization that avoids the overhead of Java class verification for some application classes.

One way to avoid the overhead of Java class verification is to turn it off completely:

```
% cvm -Xverify:none -cp MyApp.jar MyApp
```

This approach has the benefit of more quickly loading the application's classes. But it also turns off important security mechanisms that may be needed by applications that perform remote class loading.

Another approach is based on using `JavaCodeCompact` to *preload* an application's

Java classes at build time. The application's classes load faster at runtime and other classes can be loaded remotely with the security benefits of class verification.

Note: `JavaCodeCompact` assumes the classes it processes are valid and secure. Other means of determining class integrity should be used at build-time.

The companion document *CDC Build Guide* describes how to use `JavaCodeCompact` to preload an application's classes so that they are included with the CDC Java runtime environment's binary executable image. Once built, the mechanism for running a preloaded application is very simple. Just identify the application without using `-cp` to specify the user Java class search path.

```
% cvm -Xverify:remote MyApp
```

The `remote` option indicates that preloaded and system classes will not be verified. Because this is the default value for the `-Xverify` option, it can be safely omitted. It is shown here to fully describe the process of preloading an application's classes.

3.4.4 Setting the Maximum Working Memory for the Dynamic Compiler

The `-Xjit:maxWorkingMemorySize` command-line option sets the maximum working memory size for the dynamic compiler. Note that the 512 KB default can be misleading. Under most circumstances the working memory for the dynamic compiler is substantially less and is furthermore temporary. For example, when a method is identified for compiling, the dynamic compiler allocates a temporary amount of working memory that is proportional to the size of the target method. After compiling and storing the method in the code buffer, the dynamic compiler releases this temporary working memory.

The average method needs less than 30 KB but large methods with lots of inlining can require much more. However since 95% of all methods use 30 KB or less, this is rarely an issue. Setting the maximum working memory size to a lower threshold should not adversely affect performance for the majority of applications.

3.5 Tuning Dynamic Compiler Performance

This section shows how to use `cvm` command-line options that control the behavior of the CDC HotSpot Implementation Java virtual machine's dynamic compiler for different purposes:

- Optimizing a specific application's performance.
- Configuring the dynamic compiler's performance for a target device.
- Exercising runtime behavior to aid the porting process.

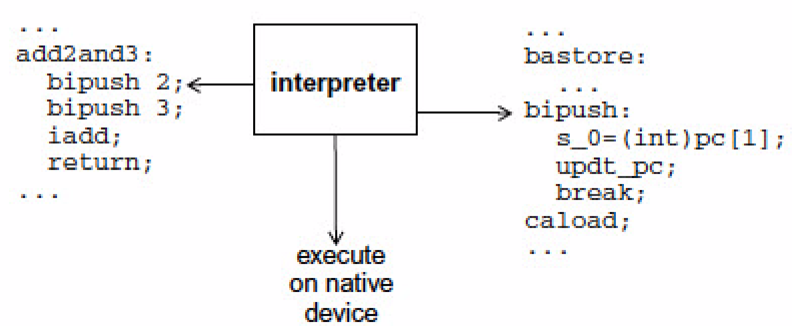
Using these options effectively requires an understanding of how a dynamic compiler operates and the kind of situations it can exploit. During its operation the CDC HotSpot Implementation virtual machine instruments the code it executes to look for popular methods. Improving the performance of these popular methods accelerates overall application performance.

The following subsections describe how the dynamic compiler operates and provides some examples of performance tuning. For a complete description of the dynamic compiler-specific command-line options, see Appendix A.

3.5.1 Dynamic Compiler Overview

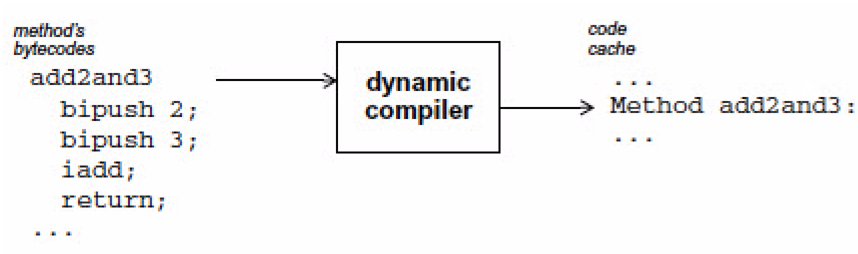
The CDC HotSpot Implementation virtual machine offers two mechanisms for method execution: the *interpreter* and the *dynamic compiler*. The interpreter is a straightforward mechanism for executing a method's bytecodes. For each bytecode, the interpreter looks in a table for the equivalent native instructions, executes them and advances to the next bytecode. Shown in Figure 3–2, this technique is predictable and compact, yet slow.

Figure 3–2 Interpreter-Based Method Execution



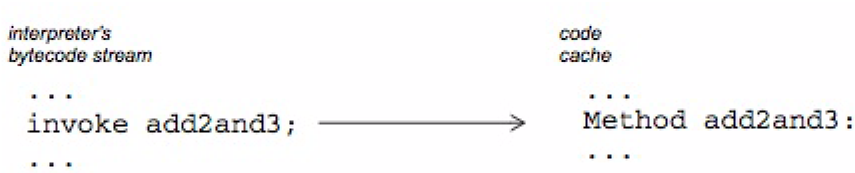
The dynamic compiler is an alternate mechanism that offers significantly faster runtime execution. Because the compiler operates on a larger block of instructions, it can use more aggressive optimizations and the resulting compiled methods run much faster than the bytecode-at-a-time technique used by the interpreter. This process occurs in two stages. First, the dynamic compiler takes the entire method's bytecodes, compiles them as a group into native code and stores the resulting native code in an area of memory called the *code cache* as shown in Figure 3–3.

Figure 3–3 Compiling a Method



Then the next time the method is called, the runtime system executes the compiled method's native instructions from the code cache as shown in Figure 3–4.

Figure 3–4 Executing a Compiled Method



The dynamic compiler cannot compile every method because the overhead would be too great and the start-up time for launching an application would be too noticeable. Therefore, a mechanism is needed to determine which methods get compiled and for how long they remain in the code cache.

Because compiling every method is too expensive, the dynamic compiler identifies important methods that can benefit from compilation. The CDC HotSpot Implementation Java virtual machine has a runtime instrumentation system that measures statistics about methods as they are executed. `cvm` combines these statistics into a single popularity index for each method. When the popularity index for a given method reaches a certain threshold, the method is compiled and stored in the code cache.

- The runtime statistics kept by `cvm` can be used in different ways to handle various application scenarios. To do this, `cvm` exposes certain weighting factors as command-line options. By changing the weighting factors, `cvm` can change the way it performs in different application scenarios. A specific combination of these options express a *dynamic compiler policy* for a target application. An example of these options and their use is provided in Section 3.5.2.1, "Managing the Popularity Threshold".
- The dynamic compiler has options for specifying code quality based on various forms of inlining. These provide space-time trade-offs: aggressive inlining provides faster compiled methods, but consume more space in the code cache. An example of the inlining options is provided in Section 3.5.2.2, "Managing Compiled Code Quality".
- Compiled methods are not kept in the code cache indefinitely. If the code cache becomes full or nearly full, the dynamic compiler decompiles the method by releasing its memory and allowing the interpreter to execute the method. An example of how to manage the code cache is provided in Section 3.5.2.3, "Managing the Code Cache".

3.5.2 Dynamic Compiler Policies

The `cvm` application launcher has a group of command-line options that control how the dynamic compiler behaves. Taken together, these options form *dynamic compiler policies* that target application or device specific needs. The most common are space-time trade-offs. For example, one policy might cause the dynamic compiler to compile early and often while another might set a higher threshold because memory is limited or the application is short-lived.

Table A–7 describes the dynamic compiler-specific command-line options and their defaults. These defaults provide the best overall performance based on experience with a large set of applications and benchmarks and should be useful for most application scenarios. They might not provide the best performance for a specific application or benchmark. Finding alternate values requires experimentation, a knowledge of the target application's runtime behavior and requirements in addition to an understanding of the dynamic compiler's resource limitations and how it operates.

The following examples show how to experiment with these options to tune the dynamic compiler's performance.

3.5.2.1 Managing the Popularity Threshold

When the popularity index for a given method reaches a certain threshold, it becomes a candidate for compiling. `cvm` provides four command-line options that influence when a given method is compiled: the popularity threshold and three weighting factors that are combined into a single popularity index:

- `climit`, the popularity threshold. The default is 20000.
- `bcost`, the weight of a backwards branch. The default is 4.
- `icost`, the weight of an interpreted to interpreted method call. The default is 20.
- `mcost`, the weight of transitioning between a compiled method and an interpreted method and vice versa. The default is 50.

Each time a method is called, its popularity index is incremented by an amount based on the `icost` and `mcost` weighting factors. The default value for `climit` is 20000. By setting `climit` at different levels between 0 and 65535, you can find a popularity threshold that produces good results for a specific application.

The following example uses the `-Xjit:`*option* command-line option syntax to set an alternate `climit` value:

```
% cvm -Xjit:climit=10000 MyTest
```

Setting the popularity threshold lower than the default causes the dynamic compiler to more eagerly compile methods. Since this usually causes the code cache to fill up faster than necessary, this approach is often combined with a larger code cache size to avoid compilation/decompilation thrashing.

3.5.2.2 Managing Compiled Code Quality

The dynamic compiler can choose to inline methods for providing better code quality and improving the speed of a compiled method. Usually this involves a space-time trade-off. Method inlining consumes more space in the code cache but improves performance. For example, suppose a method to be compiled includes an instruction that invokes an accessor method returning the value of a single variable.

```
public void popularMethod() {
...
```

```
  int i = getX();
...
}
public int getX() {
  return X;
}
```

`getX()` has overhead like creating a stack frame. By copying the method's instructions directly into the calling method's instruction stream, the dynamic compiler can avoid that overhead.

`cvm` has several options for controlling method inlining, including the following:

- `maxInliningCodeLength` sets a limit on the bytecode size of methods to inline. This value is used as a threshold that proportionally decreases with the depth of inlining. Therefore, shorter methods are inlined at deeper depths. In addition, if the inlined method is less than *value*`/2`, the dynamic compiler allows unquick opcodes in the inlined method.
- `minInliningCodeLength` sets the floor value for `maxInliningCodeLength` when its size is proportionally decreased at greater inlining depths.
- `maxInliningDepth` limits the number of levels that methods can be inlined.

  For example, the following command-line specifies a larger maximum method size.

  ```
  % cvm -Xjit:inline=all,maxInliningCodeLength=80 MyTest
  ```

3.5.2.3 Managing the Code Cache

On some systems, the benefits of compiled methods must be balanced against the limited memory available for the code cache. `cvm` offers several command-line options for managing code cache behavior. The most important is the size of the code cache, which is specified with the `codeCacheSize` option.

For example, the following command-line specifies a code cache that is half the default size.

```
% cvm -Xjit:codeCacheSize=256k MyTest
```

A smaller code cache causes the dynamic compiler to decompile methods more frequently. Therefore, you might also want to use a higher compilation threshold in combination with a lower code cache size.

The build option `CVM_TRACE_JIT=true` allows the dynamic compiler to generate a status report for when methods are compiled and decompiled. The command-line option `-Xjit:trace=status` enables this reporting, which can be useful for tuning the `codeCacheSize` option.

3.6 Ahead-of-Time Compilation

Ahead-of-time compilation (AOTC) refers to compiling Java bytecode into native machine code beforehand, for example during VM build time or install time. In CDC-HI, AOTC happens when the VM is being executed for the first time on the target platform. A set of Java methods is compiled during VM startup and the compiled code is saved into a file. During subsequent executions of CVM the saved AOTC code is found and executed like dynamically compiled code.

3.6.1 Using AOTC

AOTC is run in two basic stages: an initial run to compile a method list specified in a text file and subsequent runs that use that precompiled method list.

- *Initial run.* AOTC is enabled with the `-aot=true` command-line option. The first time `cvm` is executed, it must also include the `aotMethodList=`*file* to specify the location of the method list file. These methods are compiled and stored in the `cvm.aot` file. The `aotFile=`*file* command-line option can be used to specify an alternate location for the precompiled methods.
- *Subsequent runs.* When `cvm` is run again, it must also use `-aot=true` command-line option and `aotFile=`*file* if it was used.

If it becomes necessary to recompile the method list, this can be done with the `recompileAOT=`*boolean* command-line option.

See Table A–7 for a description of the AOTC command-line options.

3.6.2 How to Create `methodsList.txt`

A good way to produce a method list is to start by building a VM with `CVM_TRACE_JIT=true` and running with `-Xjit:trace=status`. This shows all the methods being compiled while running a particular application. Note that non-romized methods should not be included in the method list.

Adding or removing methods in `methodsList.txt` does not cause AOTC code being regenerated. To regenerate the precompiled AOTC code, use the `recompileAOT=`*boolean* command-line option to delete the `bin/cvm.aot` file.

4 Security

Security is a principal feature of Java technology and an important requirement for mobile and enterprise applications. CDC includes the same security features that are in the Java SE platform. These include built-in security features of the Java programming language and virtual machine in addition to a flexible security framework for more advanced application scenarios.

This chapter provides an overview of the security framework in addition to an outline of the kinds of security procedures that might be performed at runtime. It is not meant to replace the security documentation available for the Java SE platform, but rather to supplement it and show how CDC and the JAAS, JCE and JSSE security optional packages are related to their counterparts in the Java SE platform.

Table 4–1 describes the security documentation for the Java SE platform.

Table 4–1 Security Documentation for the Java SE Platform

Document	Description
Inside Java 2 Platform Security	Describes the Java security framework, including security architecture, deployment and customization. Chapter 12 describes deployment and runtime procedures. See documentation at `http://java.sun.com/docs/books/security`.
Security and the Java Platform	Main web page for Java security issues. See documentation at `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html`.
Java Tutorial, Security Trail	A tutorial section that describes many of the security procedures for the Java platform. Because these are identical between CDC and the Java SE platform, they are not duplicated in this chapter. See documentation at `http://download.oracle.com/javase/tutorial/security/index.html`.
Security	Java SE platform security documentation. See documentation at `http://download.oracle.com/javase/1.4.2/docs/guide/security/spec/security-spec.doc.html`.

4.1 Overview

The security framework shared by the Java SE platform and CDC is based on three key components:

- Built-in Security Features

- Security Policy Framework
- Security Provider Architecture

These provide a solid base for application and runtime security, a flexible mechanism for defining deployment-based security needs and a plug-in mechanism for supplying alternate security implementations.

4.1.1 Built-in Security Features

Java security is based on built-in language and VM security features that have been part of Java technology from its beginning:

- Strongly typed language (*runtime/compile-time/link-time*)
- Bytecode verification (*classloading-time*)
- Safety checks (*runtime*)
- Dynamic class loaders (*classloading-time*)

4.1.2 Security Policy Framework

A security policy controls how system resources are accessed by applications at runtime. The Java security framework includes both a default security policy and a mechanism for describing alternate security policies for application and deployment-specific needs. The main benefits of this security policy framework are:

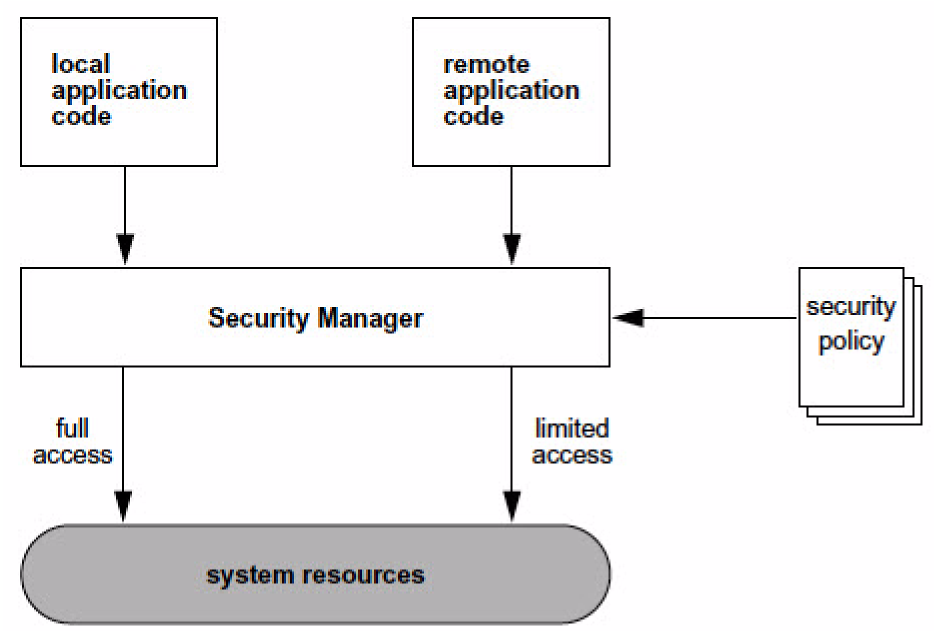
- Code-centric, not identity-centric architecture
- Security policies are described separately from both the applications they control and the Java runtime environment.
- Fine-grained access control at the package, class or field level
- Flexible permission mechanism
- Protection domains provide a layer of abstraction between permissions and code.

The main elements of a security policy are the following:

- `permission` set, a list of permissions granted to the code
- `codeBase`, the location from where the code is loaded
- `signedBy`, the author of the code
- `principal`, the identity of the entity running the code

Figure 4–1 illustrates the Java security model by showing how application code can be loaded from different sources: local and remote. The security manager controls access to system resources by comparing properties of the application code with the current security policy. The default security policy allows full access to local application code and limited access to remote application code. But other security policies are possible. For example, application code from a trusted yet remote source may be given greater access than untrusted code from a local source.

Figure 4–1 Java Security Policy Model



4.1.3 Security Provider Architecture

Beginning with version 1.2, the Java SE platform added some security optional packages that allow Java technology to adapt to more specific requirements of applications and deployments. These security optional packages include a security provider architecture that is *interoperable* because it is based on publicly available security standards, and ex*tensible* because alternate *security provider implementations* can be supplied without requiring modifications to application code.

For example, the JAAS, JCE and JSSE security optional packages include several *service provider interfaces* (SPIs) that describe the requirements of a security provider implementation. Table 2–2 describes the default implementations for these security components.

4.1.4 Custom JSSE Provider Plug-ins

JSSE supports custom Provider plug-ins which can be implemented as extensions of `SSLSocketFactory`.

4.1.5 Oracle JSSE Cipher Suite Support

Many of the standard JSSE algorithm names are prefixed with `SSL_`. JSSE now supports the `TLS_` prefix to be used as an alias to a standard algorithm name.

4.1.6 Self-Integrity Checks

In general, a JCE Provider implementation should include self-integrity checks. For example, Oracle's current JCE provider (Oracle JCE) includes self-integrity checks. However, this is not a requirement of the JCE or Oracle for a third-party JCE provider. A third-party JCE provider should make its own choice regarding whether including self-integrity checks or not.

4.2 Security Procedures

This section outlines the security procedures surrounding the Java security framework described in the previous section. Because these procedures are identical to the procedures used for the Java SE platform, this section just describes the procedure and indicates where to find the appropriate Java SE platform documentation.

4.2.1 Using Alternate Security Providers

From an administrator's perspective, the first step is to choose whether to install and use any alternate security providers. In most cases, the default security providers described in Table 2–2 are sufficient.

For a description of how to install alternate security providers, see *Inside Java 2 Platform Security, Second Edition*. Section 12.5, *Installing Provider Packages*, describes how to install alternate security providers.

4.2.2 Public Key Management

The JAAS optional package includes an extensible authentication framework that can use different forms of authentication. The default `LoginModule` is the `KeyStoreLoginModule`, which uses a protected database (Oracle's JKS keystore file) to store public key data. Other forms of authentication are possible like smart card or Kerberos.

The main tool for managing keystore files is `keytool`(1), which is included in the Java SE platform toolset. `keytool` can be used for

- importing a key
- listing available keys
- replacing a key
- deleting a key

The default keystore file is in `lib/security/cacerts`, described in Table 2–2.

For a description of how to use `keytool` to add and modify keystore entries, see Section 12.8, *Security Tools*, in *Inside Java 2 Platform Security, Second Edition*. The security trail in the *Java Tutorial* also covers how to use `keytool`.

4.2.3 Security Policy Management

Security policies are stored in security policy files. `policytool` is a convenient GUI-based tool for managing security policies. With it, a system administrator can

- identify a keystore
- specify permissions
- specify a codebase

The location of the default security policy file is `lib/security.policy`, described in Table 2–2. Alternate locations can be defined with the `-Djava.security.policy` command-line option.

For a description of how to use the `policytool` to manage security policies, see Section 12.8, *Security Tools*, in *Inside Java 2 Platform Security, Second Edition*. The security trail in the *Java Tutorial* also covers how to use `keytool`.

4.2.4 Seed Generation for Random Number Generation

The CDC Java runtime environment uses a native platform-provided source as an entropy gathering device for seed generation indicated by the `securerandom.source` system property. The Linux default for this system property is `file:/dev/random`.

On some Linux systems, `/dev/random` can block if it hasn't generated sufficient entropy before a random seed is needed and this can cause applications using `java.security.SecureRandom` to hang while waiting for the entropy pool to fill. To avoid this hang problem, the CDC Java runtime environment has a fallback mechanism to read from the `/dev/urandom` device when it determines that there isn't enough entropy for `/dev/random` to work promptly.

Note that `/dev/urandom` is not generally considered strong enough to support applications like keypair generation. If the strongest possible seed generation is required, this fallback mechanism can be disabled by setting the `microedition.securerandom.nofallback` property to `true`. Doing so may run the risk of application hangs on certain devices where the entropy pool is subject to early exhaustion.

5 Localization

The CDC Java runtime environment can be localized to support different languages and cultures. The following sections provide CDC-specific information for localization procedures:

- Setting Locale System Properties
- Timezone Information Files
- Font Management (Personal Basis Profile only)

5.1 Setting Locale System Properties

In the CDC Java runtime environment, the locale system properties described in Table 5–1 are set before `cvm` can parse its command-line arguments. Thus, it is not possible to change the locale by specifying these system properties on the `cvm` command-line with the `-D`*property*=*value* option.

Table 5–1 Locale System Properties

System Property	Description
`user.language`	Two-letter language name code based on ISO 639.
`user.region`	Two-letter region name code based on ISO 3166.
`file.encoding`	Default character-encoding name based on the IANA Charset MIB.

On Linux, these properties are extracted from the `LANG` locale environment variable using the format *language_region.encoding*. The `user.language` property is obtained from the *language* code. The `user.region` property is obtained from the *region* code. The `file.encoding` property is obtained from the *encoding* suffix. For example, to change the locale behavior of `cvm` on Linux, simply change the `LANG` locale environment variable to set the locale system properties.

```
% setenv LANG en_US.ISO8859_1
```

Therefore,

```
user.language = en
user.region = US
file.encoding = ISO8859_1
```

5.2 Timezone Information Files

The `lib/zi` directory contains a small set of example timezone information files. Additional files can be generated and placed in this directory. See the `javadoc` comments for the `sun.util.calendar.ZoneInfoFile` class for information about generating alternate timezone information files.

5.3 Font Management (*Personal Basis Profile only*)

In the CDC Java runtime environment, font management is a subset of the functionality provided by Java SE technology and is described below in Table 5–2.

Table 5–2 Font Management Comparison

Feature	Java SE	CDC
Default font mapping between Java logical fonts and platform logical fonts is specified at build-time.	yes	yes
Logical font mapping in `lib/font.properties` file.	yes	no
Bundled Lucida fonts in `lib/fonts`.	yes	no
Application-specific fonts in an application 's `jar` file.	yes	no

The six logical fonts available to a Java application are described in Table 5–3.

In practice, the only way to specify alternate fonts is to remap the platform logical fonts. TrueType fonts are mapped to logical platform fonts used by the CDC Java runtime environment for the Java logical fonts described in Table 5–3.

Table 5–3 Logical Font Names

Java Logical Font	Qt Logical Font	Example	Description
`default`	`Sans Serif`	Courier	The default font is used when no other font is specified or if an attempt to match a font fails.
`dialog`	`Sans Serif`	Lucida Sans	A font for displaying fixed information within a dialog box or form.
`dialoginput`	`Courier`	Lucida Sans Typewriter	A font that is used for text fields within dialog boxes and forms that represent user input.
`monospaced`	`Courier`	Lucida Sans Typewriter	A non-proportional font where each character has the same width. This simplifies string width calculations for dialog boxes and forms.
`sanserif`	`Sans Serif`	Helvetica	A streamlined font that is simpler to render on low-resolution devices like computer monitors and faxes.
`serif`	`Serif`	Times Roman	A font with short lines at the end of the main strokes of a character to ease visual character recognition.

6 Developer Tools

One of the principal goals of CDC is to leverage conventional Java SE developer tools for use with CDC applications and devices. This chapter shows how to integrate the CDC Java runtime environment with Java SE developer tools like `javac`, `jdb`, and `jvmtihprof`, in addition to integrated development environments such as NetBeans.

6.1 Compiling With `javac`

Compiling Java source code is a separate process from execution. All that is needed is application source code, a Java compiler like `javac` and an appropriate Java class library to compile against. In this way, a developer can compile a Java application on a desktop system and later download it onto a target device for testing or deployment.

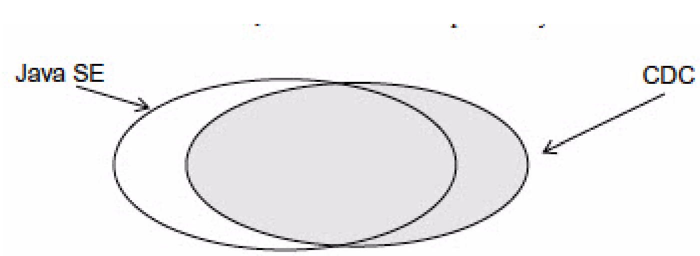
This chapter first reviews the API relationship between the CDC and Java SE platforms. Then it shows how `javac` compiles a Java class for the Java SE platform and how this process changes for CDC. Finally, it shows how to compile an example CDC program.

6.1.1 CDC and Java SE

It is possible to take unmodified application software that was compiled for the Java SE platform and run it on a CDC Java runtime environment because the CDC Java virtual machine can load and execute Java classes that are compliant with the class specification for the Java SE platform.

Figure 6–1 describes the API relationship between the CDC and Java SE platforms. The two platforms have much in common, including most of the core Java class library. Differences between the CDC and Java SE APIs can cause discrepancies at runtime. These differences are based on the need to remove or change certain classes for memory, functionality or performance reasons.

Figure 6–1 CDC and Java SE API Compatibility



There are four major differences between the CDC and Java SE platforms:

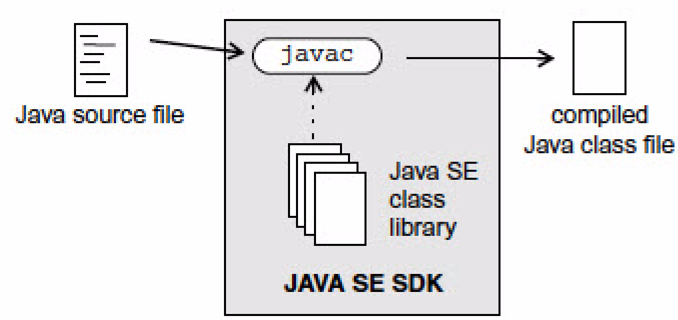
- Some Java SE packages, classes and methods have been removed because they are not appropriate for smaller devices. Compiling application source code against the Java SE class library may work, but the compiled classes may fail to run on a CDC Java runtime environment because the classes are not available at runtime.
- Some packages like `java.sql` are present in the Java SE platform but not in CDC, though they may be added as an optional package. In this case, compiling application source code against the Java SE class library may work but running the compiled classes against the CDC Java runtime environment may not.
- Most Java SE deprecated methods have been removed from CDC. For example, `java.awt.List.clear()` is deprecated in JDK version 1.1 and replaced with `java.awt.List.removeAll()`. In this case, compiling a Java SE application that uses this deprecated method against the CDC Java class library cause `javac` to fail to compile because it cannot find the deprecated method.
- CDC includes CLDC compatibility classes that are not included in the Java SE class library. In this case, compiling CDC source code against the Java SE class library might cause `javac` to fail to compile because these compatibility classes are not present in the Java SE class library.

Therefore, in practice, it is best to recompile Java source code for a Java SE application against a CDC Java class library. Finally, the CDC Java class library is modular and can change based on the needs of a product design. Most of this modularity is based on profiles and optional packages. See Section 1.6, "Java ME API Choices" for an explanation of how CDC APIs can vary.

6.1.2 Compiling Java Source Code for the Java SE Platform

Figure 6–2 shows how `javac` compiles Java source code for the Java SE platform. When `javac` processes Java source code, it uses a Java class library to discover type information about the classes used in the source code. By default, this is the Java SE class library located in `jre/lib/rt.jar`.

Figure 6–2 Compiling Java Source Code for the Java SE Platform

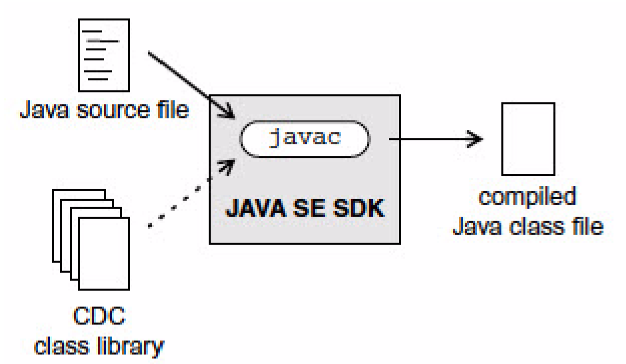


For example, when `javac` encounters a Java type reference like `java.util.BitSet`, it gets the type information from the Java SE class library at compile time. Later, at runtime, when the Java virtual machine creates an object of type `java.util.BitSet`, it also gets the type information from the Java SE class library.

6.1.3 Compiling Java Source Code for CDC

The same `javac` compiler used for developing Java SE applications can be used to compile Java source code for the CDC Java runtime system. The key is to use a different target Java class library to compile against. Figure 6–3 shows how the `javac` compiler uses the `-bootclasspath` command-line option to specify an alternate target Java class library as a cross-compilation target.

Figure 6–3 Compiling Java Source Code for CDC



The mechanics of using `javac` to compile Java source code for CDC differ slightly from those used for the Java SE platform.

6.1.4 Determining the Target Class Library

Section 1.5, "Java Micro Edition Technology Standards", Section 1.6, "Java ME API Choices", and Figure 1–2 show how the API functionality of a specific CDC product implementation can vary based on choices made at design time. Therefore, it is important to use a target development version of the CDC Java class library that represents the APIs available in the configuration, profile and optional packages on the target device.

Note: See the companion document *CDC Build System Guide* for information on how to build a target development version of the CDC Java class library which represents the combination of configuration, profile and optional packages for the target device

---

6.1.5 Useful `javac` Command-Line Options

The page `http://download.oracle.com/javase/6/docs/technotes/tools/windows/javac.html` describes the `javac` cross-compilation options. These are summarized in the following subsections.

6.1.5.1 `-classpath` *classpath*

Sets the user class search path, which is useful for compiling against third-party class libraries.

6.1.5.2 `-bootclasspath` *classpath*

Sets the system class search path. With `javac`, this option overrides the Java SE class library and specifies an alternate target Java class library for cross-compilation like the target development version of the CDC Java class library.

6.1.5.3 `-extdirs` *classpath*

Sets the extensions class search path for optional packages. The CDC default location is the `lib` directory, except for some security optional packages which are found in the `lib/ext` directory.

6.1.5.4 `-source` *release*

Specifies the version of Java source code accepted. In practice, this controls the use of Java programming language syntax that conflicts with identifier names. For example, J2SE 1.4 includes support for the `assert` keyword and J2SE 1.5 includes support for `generics`. Therefore, an `int named assert` is legal in 1.3 and illegal in 1.4. The *release* argument can be set to 1.3, 1.4, 1.5 (or synonym 5), or 1.6 (or synonym 6) for CDC application development.

6.1.5.5 `-target` *version*

This option directs `javac` to generate Java class files for a specific version of the Java virtual machine. It is preferable to set the *version* value to 1.6, though values of 1.2 through 1.5 can also be used for CDC development.

6.1.5.6 `-deprecation`

Show a description of each use or override of a deprecated member or class. Without `-deprecation`, `javac` shows the names of source files that use or override deprecated members or classes.

6.1.6 Compiling an Example CDC Program

The example below demonstrates how to compile an application using the command-line option `-bootclasspath` argument to specify an alternate target Java class library:

```
% javac -target 1.4 -source 1.4 -bootclasspath \
    /home/mydir/myclasses.zip MyApp.java
```

6.2 Application Debugging

You can remotely debug a CDC application with most debuggers that support the Java Virtual Machine Tool Interface (JVMTI) described in `http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html`. The most likely choices are the NetBeans, Oracle JDeveloper, and Eclipse integrated development environments, but you can also use the Java SE `jdb` command line debugger or another compatible debugger. You run the debugger on a development host, and the application plus CDC on the target device. CDC and the debugger communicate over a network.

CDC debugging has the following limitations:

- Only interpreted code can be debugged.
- CDC-debugger connections must use sockets. Shared memory connections are not supported.

6.2.1 Launching `cvm` in Debug Mode

Regardless of the debugger you choose, you launch `cvm` running the application in the same way.

Note: For the Oracle Java ME Embedded Client, cvm is installed on your system in the following location:

InstallDir/*Oracle_JavaME_Embedded_Client*/binaries/bin/cvm

See the *Oracle Java Micro Edition Embedded Client Reference Guide* for detailed information about launching `cvm` for the Oracle Java ME Embedded Client.

6.2.1.1 `cvm` Debug Mode Syntax

Example 6–1 and Example 6–2 show how to launch `cvm` in debug mode on a target host. These examples assume that the target host runs a Unix-style operating system and that socket networking is operational. Make adjustments as necessary for your target host. For the Oracle Java ME Embedded Client, see the *Oracle Java Micro Edition Embedded Client Reference Guide* for the appropriate command syntax and suboptions used to launch `cvm` in debug mode.

For a list of debug suboptions, run `cvm -agentlib:jdwp=help` or refer to `http://download.oracle.com/javase/1.5.0/docs/guide/jpda/conninv.html`.

Example 6–1 `cvm` ***Listens for Connection from Debugger***

```
% cvm -agentlib:jdwp=transport=dt_socket,server=y,address=port -Xdebug ...
```

Example 6–2 `cvm` ***Connects to Debugger***

```
% cvm -agentlib:jdwp=transport=dt_socket,server=n,address=host:port -Xdebug ...
```

When launching `cvm` in debug mode, observe the following requirements:

- `-agentlib:jdwp` and the `transport` and `address` suboptions must be specified.
- The `transport` value must be `dt_socket`.
- Set `server` to `y` to direct `cvm` to listen for a connection from the debugger (the most likely case). Set `server` to `n` to direct `cvm` to attach to a listening debugger.
- If `server=y`, set *port* to the socket port on the target host at which `cvm` listens for a connection. If `server-n`, set *host:port* to the host and socket port at which the debugger waits for a connection from `cvm`.
- `-Xdebug` disables the compiler so the virtual machine interprets the application's bytecodes.

6.2.1.2 `cvm` Debug Mode Example

Example 6–3 shows a simple example of launching `cvm` as a server to debug a `HelloWorld` application.

Example 6–3 Launching `cvm` ***as a Server***

```
% cvm -agentlib:jdwp=transport=dt_socket,server=y,address=8000 -Xdebug -classpath
/home/mydir/myclasses.zip HelloWorld
Listening for transport dt_socket at address: 8000
```

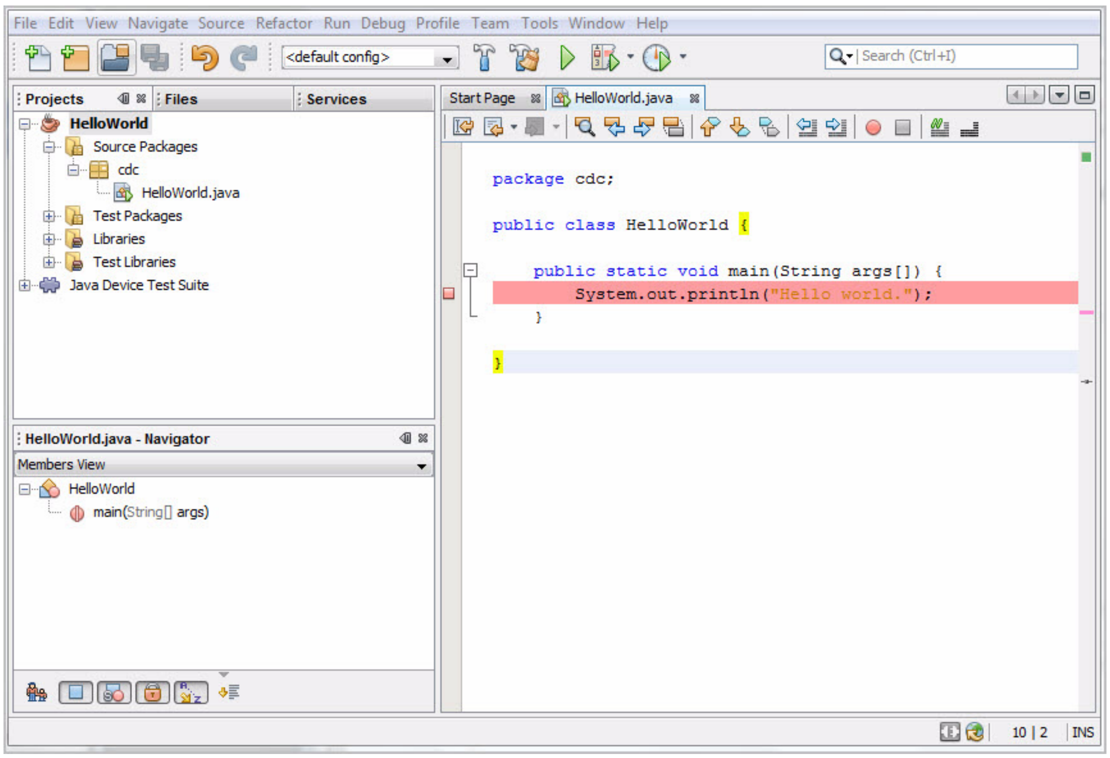
6.2.2 Attaching the NetBeans IDE Debugger to `cvm`

Although this section describes the NetBeans debugger, other IDE debuggers that are compatible with the Java Virtual Machine Tool Interface (JVMTI) can be used similarly. This section first describes the most common arrangement in which `cvm` acts as a server for the debugger, then the converse case.

1. Load the NetBeans project you want to debug and create a debugger operation, such as a breakpoint. Figure 6–4 shows an example.

   Ensure that the project's compiled class files are accessible to the target host and that the class files correspond to the source files loaded in the IDE.

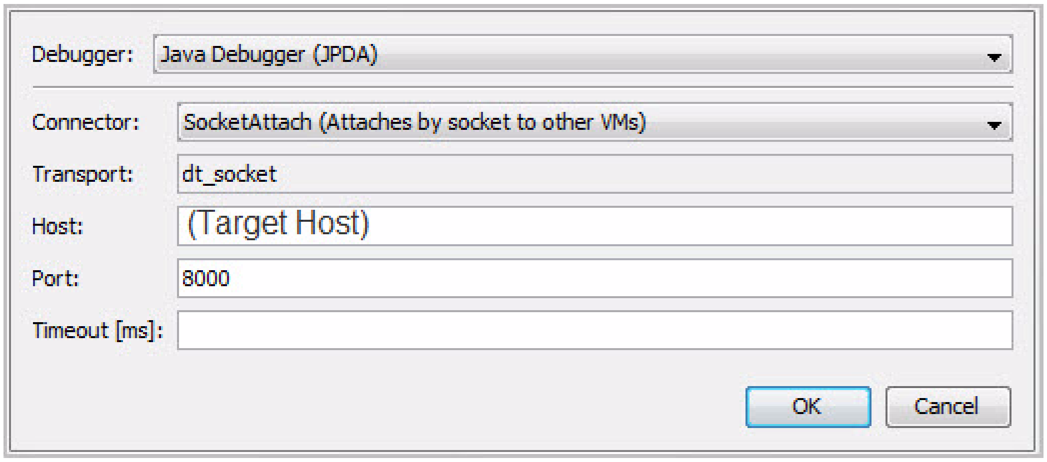
Figure 6–4 Breakpoint in `HelloWorld.java`



2. On the target host, launch `cvm` with `server=y` and, for this example, `address=8000`, similar to the example in Section 6.2.1.2, "`cvm` Debug Mode Example".
3. In NetBeans, choose **Debug** > **Attach Debugger**.
4. Set up the Attach Debugger dialog as shown in Figure 6–5.

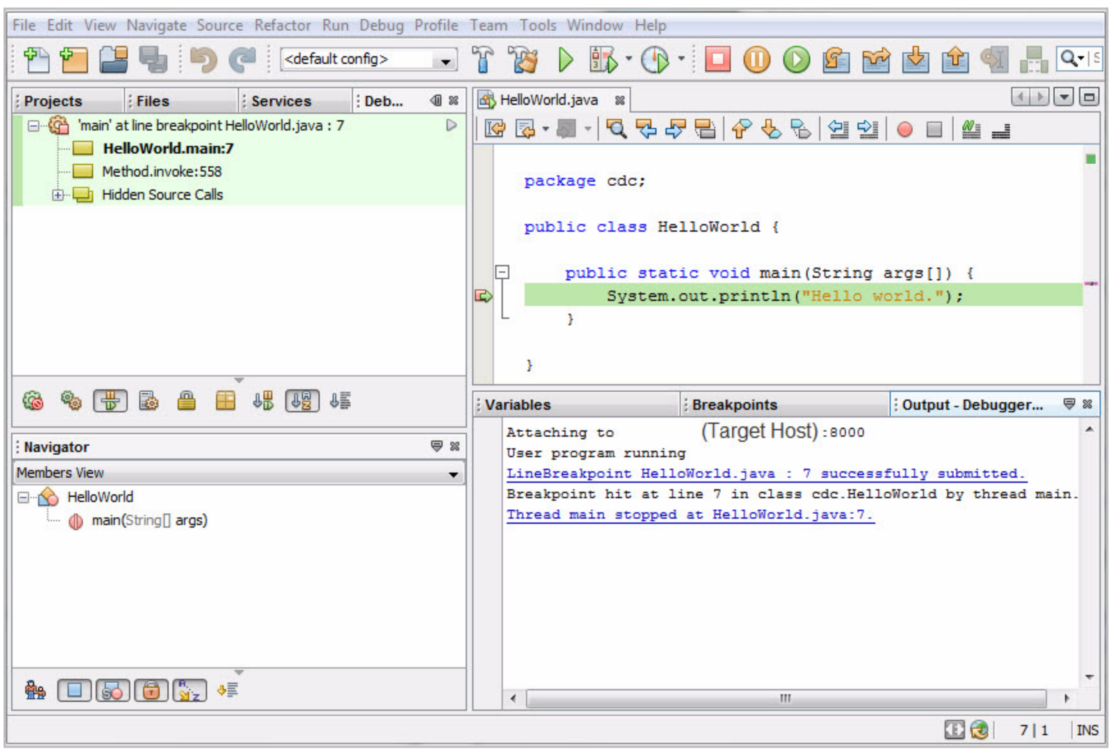
   Substitute the target host name for (Target Host).

Figure 6–5 Attach Debugger Dialog (Debugger as Client)



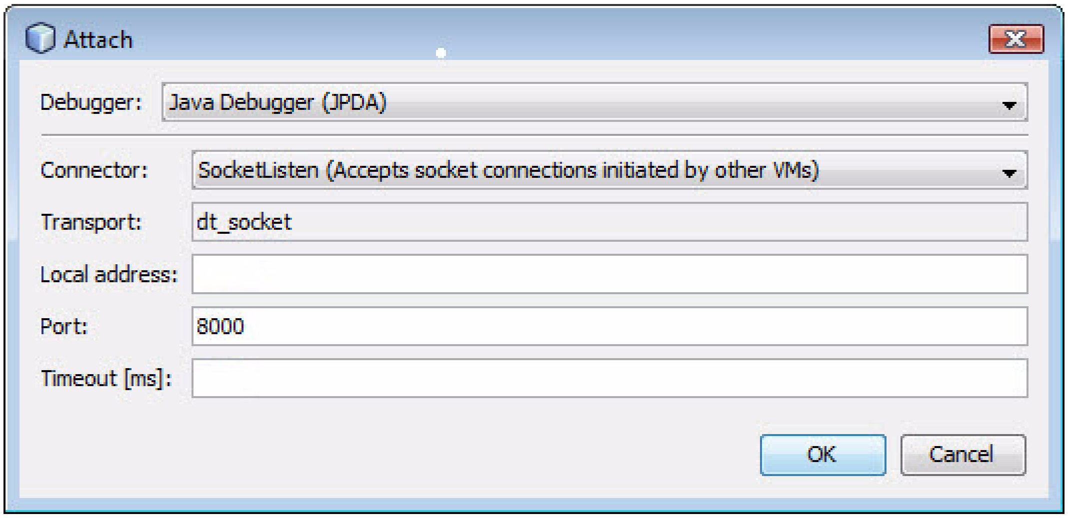
The debugger connects and indicates that execution has stopped at the breakpoint as similar to Figure 6–6.

Figure 6–6 Debugger Connected and Stopped at Breakpoint



If you want the debugger to be the server, complete the Attach Debugger dialog similar to Figure 6–7. After clicking **OK**, launch `cvm` on the target host with `server=n` and `address=` the debuggers' host and port.

Figure 6–7 Debugger as Server Attach Debugger Setup



6.2.3 Attaching to `cvm` with `jdb`

After launching `cvm` as a debug server (see Section 6.2.1.2, "`cvm` Debug Mode Example") on the target host, you can connect to it with the `jdb` command line debugger using syntax similar to Example 6–4. The `jdb` command is in *JavaSEinstall*`/bin/`.

Example 6–4 Attaching to `cvm` ***with*** `jdb`

```
% jdb -connect com.sun.jdi.SocketAttach:hostname=hostname,port=8000
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
VM Started: No frames on the current call stack
main[1]
```

6.3 Application Profiling

Profiling is the acquisition of runtime performance data for an application on a target runtime system. Understanding the runtime behavior of an application allows the developer to identify performance-sensitive components when tuning an application's implementation or selecting runtime features. `cvm` profiling provides reports that include CPU usage, heap allocation statistics, and monitor contention profiles. See `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html` for more information.

This section describes two profiling options using a `HelloWorld` example application:

- Section 6.3.1, "Remote Profiling with the NetBeans IDE"
- Section 6.3.5, "Simple Local Profiling with `jvmtihprof`"

6.3.1 Remote Profiling with the NetBeans IDE

This section describes how to profile remotely with the NetBeans IDE. The steps are:

- Calibrate the Profiler Agent (a one-time operation)
- Start `cvm` with the Profiler Agent
- Attach the NetBeans Profiler

Before you begin, ensure that the project's compiled class files are accessible to the target host and correspond to source files loaded in the IDE. Also be sure that the profiler agent native classes are accessible on the target host. For platforms directly supported by CDC (see the *Build Guide*), the build creates the profiler agent as a `.so` or `.dll` library called `profiler interface`. For platforms that use a CDC port, the details of the profiler agent are platform-specific.

> **Note:** This section covers only the basics of remote profiling. Read the NetBeans online help if you need more information on the subject. This section was created with the NetBeans version 6.7.1 IDE.

6.3.2 Calibrate the Profiler Agent

1. On the target host, calibrate the profiler agent by issuing commands equivalent to those shown in Example 6–5 or, for the Oracle Java ME Embedded Client, see the *Oracle Java Micro Edition Embedded Client Reference Guide* for commands and procedures used on both Linux and Windows platforms.

   Calibration measures the profiler agent overhead so it can be subtracted out of measurements obtained in a profiler run. To run the NetBeans calibrator, the target host must have access to the files `jfluid-server.jar` and `jfluid-server-cvm.jar`. These are NetBeans libraries modified for CDC so they consume less target device file system space. The location of these files is target host-dependent.

   In the following example, use `set CVM_HOME=`*yourCVM* for the Windows operating system. Use `export CVM_HOME=`*yourCVM* for Linux operating system.

Example 6–5 Calibrating the Profiler

```
% set CVM_HOME=yourCVM
% $CVM_HOME/bin/cvm \
-classpath $CVM_HOME/lib/profiler/lib/jfluid-server.jar:\
$CVM_HOME/lib/profiler/lib/jfluid-server-cvm.jar \
-Djava.library.path=$CVM_HOME/bin \
org.netbeans.lib.profiler.server.ProfilerCalibrator
Profiler Agent: JNI On Load Initializing...
Profiler Agent: JNI OnLoad Initialized successfully
Starting calibration...
Calibration performed successfully
For your reference, obtained results are as follows:
Approximate time in one methodEntry()/methodExit() call pair:
When getting absolute timestamp only: 3.085 microseconds
When getting thread CPU timestamp only: 3.1022 microseconds
When getting both timestamps: 5.2254 microseconds

Approximate time in one methodEntry()/methodExit() call pair
in sampled instrumentation mode: 0.7299 microseconds
```

6.3.3 Start `cvm` with the Profiler Agent

1. Launch `cvm` with the profiler agent using a command equivalent to that shown in Example 6–6 for a Linux target host or, for the Oracle Java ME Embedded Client,

see the *Oracle Java Micro Edition Embedded Client Reference Guide* for the appropriate commands used to launch `cvm` with the profiler agent.

Example 6–6 Launching cvm with the Profiler Agent

```
% set CVM_HOME=yourCVM
% $CVM_HOME/bin/cvm -Xmx32M
-agentpath:profilerInstallDir/lib/deployed/cvm/linux/libprofilerinterface.so=profi
lerInstallDir/lib,5140 -cp /home/mydir/myclasses.zip HelloWorld
Profiler Agent: Initializing...
Profiler Agent: Options: >profilerInstallDir/lib,5140<
Profiler Agent: Initialized successfully
Profiler Agent: Waiting for connection on port 5140 (Protocol version: 9)
```

5140 is the default NetBeans profiler port, which you can change in **NetBeans Tools** > **Options** > **Miscellaneous** > **Profiler**.

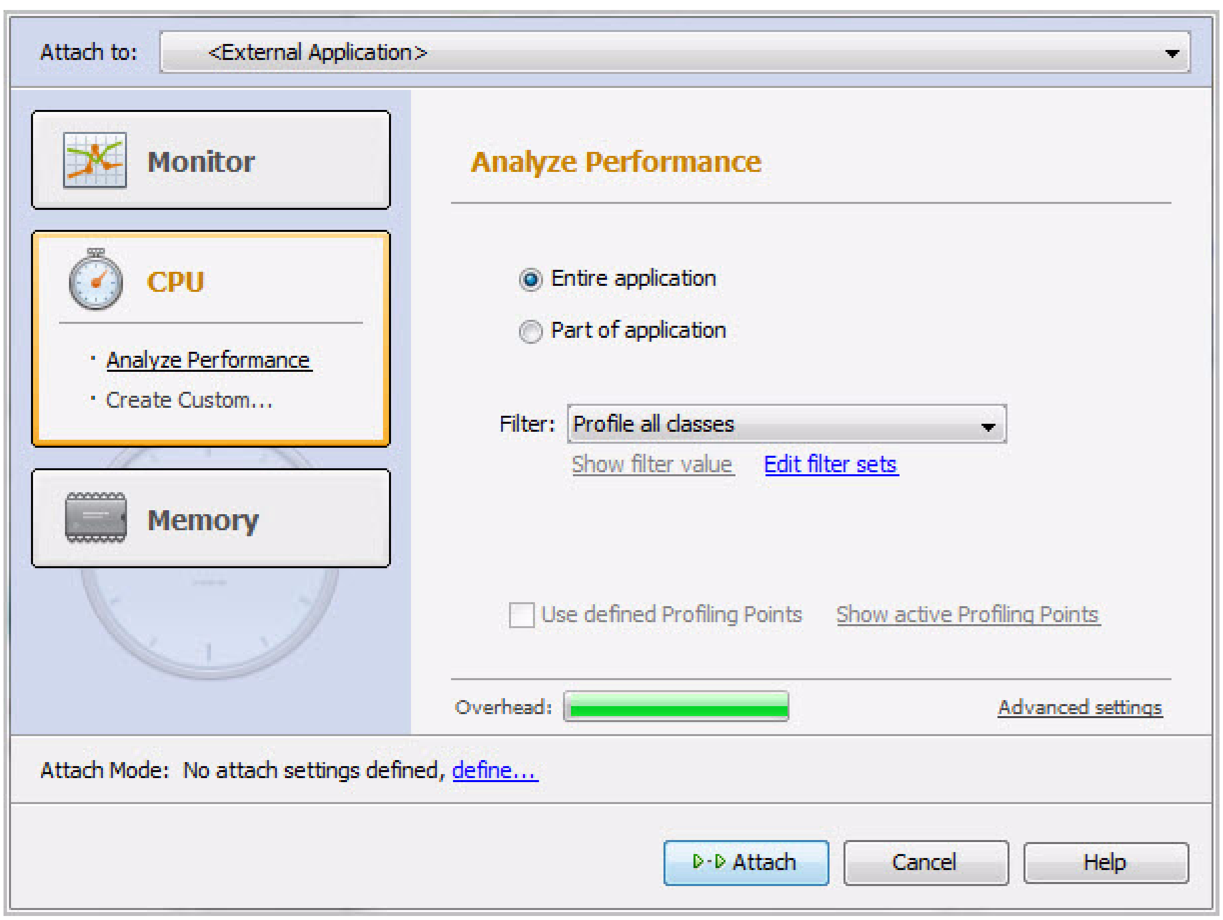
`libprofilerinterface.so` is a shared native code library. For Windows hosts, it is `libprofilerinterface.dll`. Building CDC creates the file.

6.3.4 Attach the NetBeans Profiler

1. Load the project to be profiled, and choose **Profile** > **Attach Profiler.**

   The Attach Profiler dialog appears, similar to Figure 6–8.

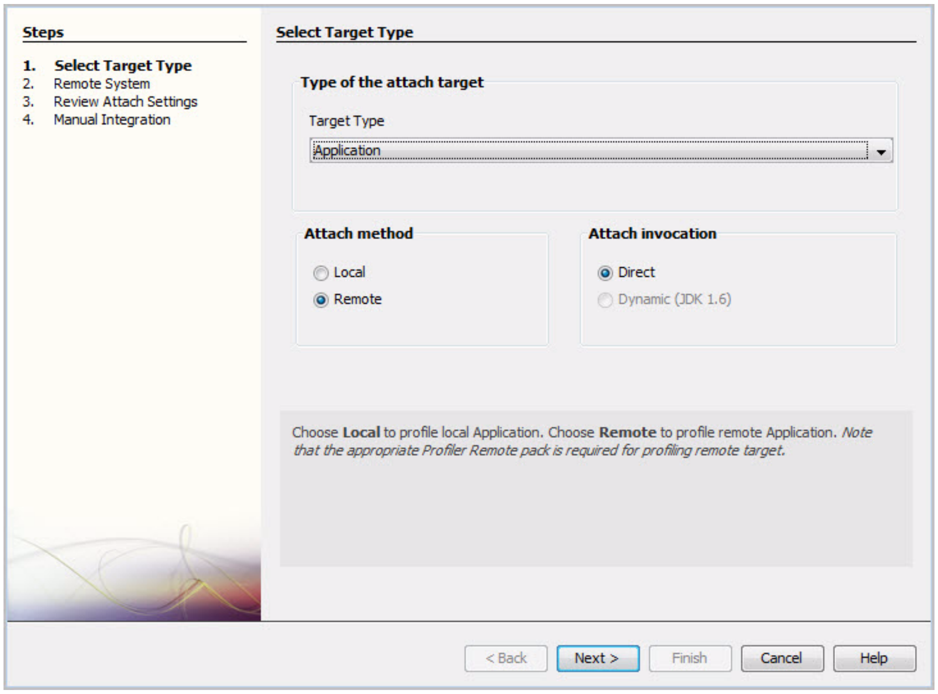
Figure 6–8 Attach Profiler Dialog



2. Near the bottom of the dialog, click **define**.

   The Select Target Type screen appears, similar to Figure 6–9.

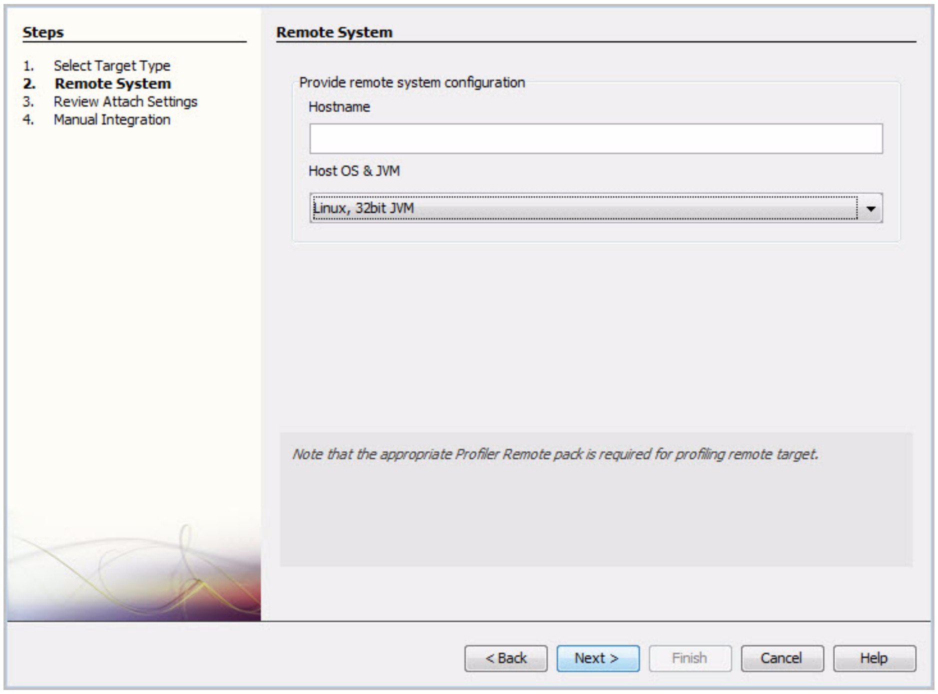
Figure 6–9 Select Target Type Screen



3. Set the values as follows, then click **Next**>:
   - Target Type: Application
   - Attach method: Remote
   - Attach invocation: Direct

   The Remote System screen appears, similar to Figure 6–10.

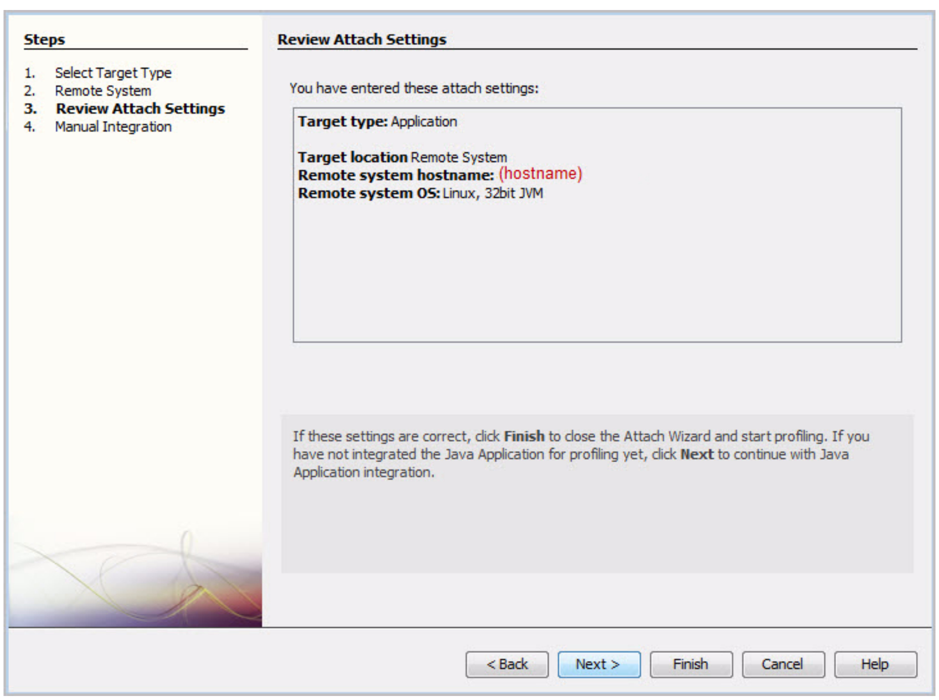
Figure 6–10 Remote System Screen



4. Enter the target host's name or IP address, select its operating system and Java virtual machine from the drop-down, then click **Next>**.

   The Review Attach Settings screen appears, similar to Figure 6–11.

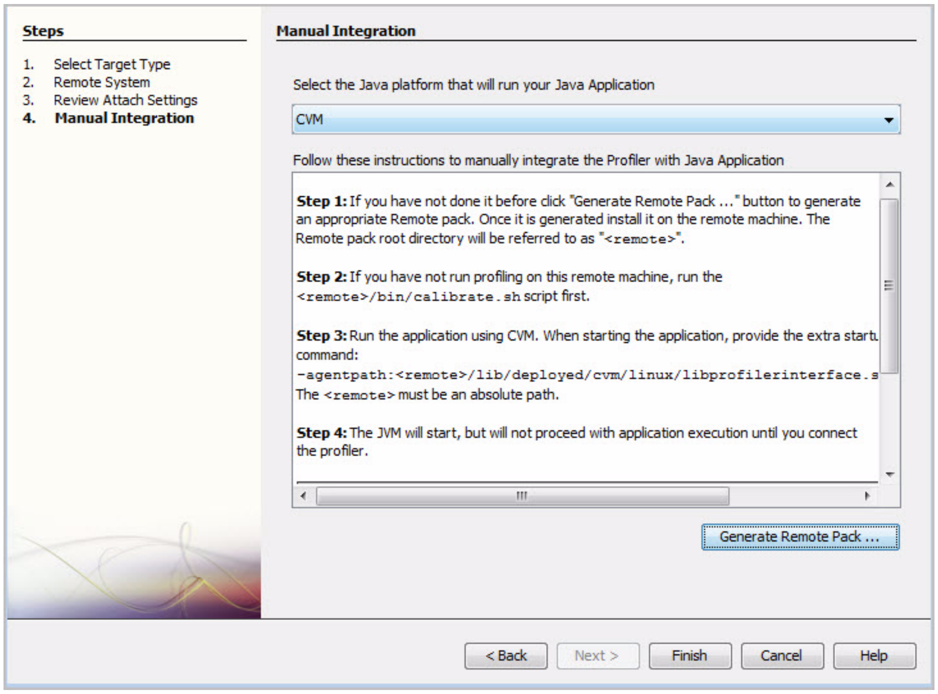
Figure 6–11 Review Attach Settings Screen



5. Verify that the settings are correct, then click **Next**>.

The Manual Integration screen appears, similar to Figure 6–12.

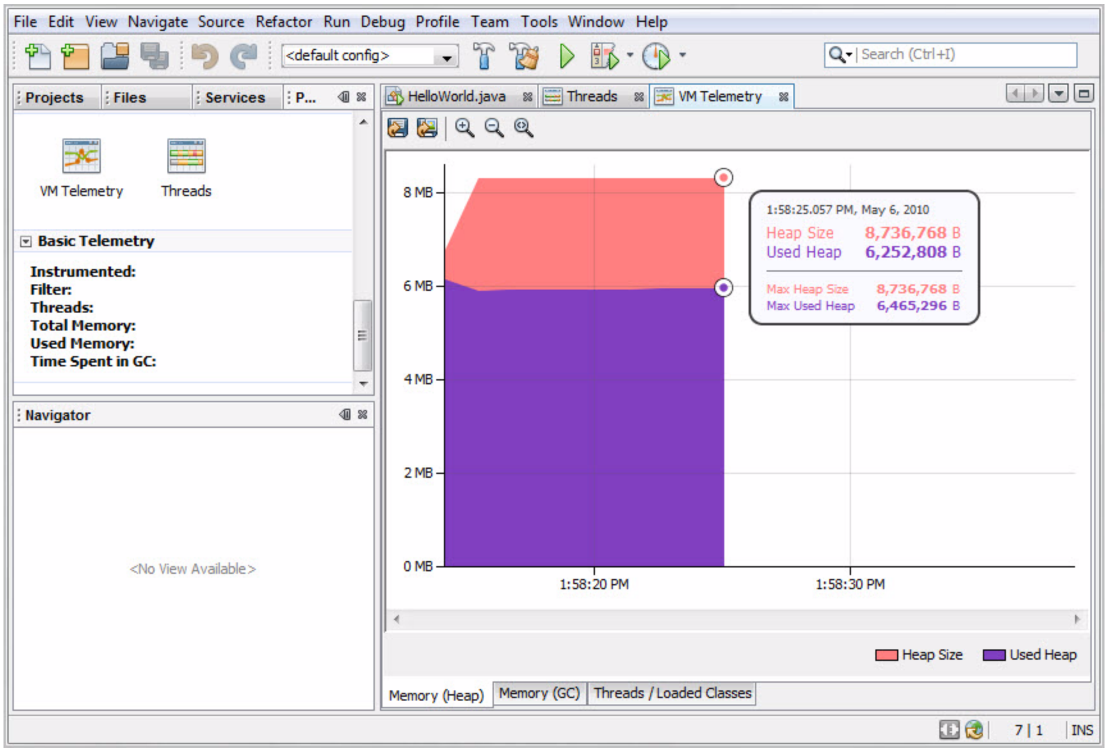
Figure 6–12 Manual Integration Screen



6. In the Manual Integration screen, click **Finish**.
7. In the Attach Profiler dialog click **Attach**.

   Profiling results begin to appear, for example, the heap profile shown in Figure 6–13.

Figure 6–13 Sample Profile Results



Subsequent profiling runs are simpler because the NetBeans IDE remembers settings:

1. On the target host, start the application with the `-agentpath` option shown in Example 6–6.
2. In the NetBeans IDE, choose Profile > Attach Profiler.
3. In the Attach Profile dialog, click Attach.

6.3.5 Simple Local Profiling with `jvmtihprof`

Example 6–7 is a simple profiling example that creates a file of profiling data for a `HelloWorld` application.

Example 6–7 Using `jvmtihprof`

```
% cvm -agentlib:jvmtihprof -Xbootclasspath/a:./lib/mysamples.jar -classpath
/home/mydir/myclasses.zip HelloWorld
Hello world.
Dumping Java heap ... allocation sites ... done.
```

The `-Xbootclasspath` option specifies the location of *mysamples*`.jar`, which is required for profiling. In this example, no output file name is given, so the profile data is in the default file `java.hprof.txt`.

The `-agentlib:jvmtihprof` option controls profiling features. For example:

```
% cvm -agentlib:jvmtihprof=heap=all,cpu=samples,file=profile.txt ...
```

Table 6–1 lists the profiling options.

Table 6–1 Profiling Command-Line Options

Option	Default	Description
`-agentlib:jmvtihprof[=option=value, ...]`		Run the VM with profiling enabled using options specified
`heap=dump|sites|all`	`all`	Heap profiling
`cpu=samples|times|old`	`off`	CPU usage
`monitor=y|n`	`n`	Monitor contention
`format=a|b`	`a`	ASCII or binary output
`file=`*`name`*	`java.hprof.txt`	Write data to file *name* and append `.txt` for ASCII format
`net=`*`host`*`:`*`port`*	`(off)`	Send data over a socket
`depth=`*`size`*	`4`	Stack trace depth
`cutoff=`*`value`*	`0.0001`	Output cutoff point
`lineno=y|n`	`y`	Display line numbers in traces
`thread=y|n`	`n`	Thread in trace
`doe=y|n`	`y`	Dump on exit

A

cvm Reference

This appendix describes the `cvm` command in detail. For the Oracle Java ME Embedded Client, see the *Oracle Java Micro Edition Embedded Client Reference Guide* for detailed information about `cvm` for the Oracle Java ME Embedded Client.

A.1 Synopsis

```
cvm [-options] class [options ...]
cvm [-options] -jar jarfile [options ...]
```

A.2 Description

`cvm` launches a Java application. It does this by starting a Java virtual machine, loading its system classes, loading a specified application class, and then invoking that class's `main` method, which must have the following signature:

```
public static void main(String args[])
```

The first non-option argument to `cvm` is the name of the top-level application class with a fully-qualified class name that contains the `main` method. The Java virtual machine searches for the main application class, and other classes used, in three locations: the *system class path*, the *extension class path* and the *user class path*. See Section 3.3, "Class Search Path Basics," for more information about Java class paths. Non-option arguments after the main application class name are passed to the `main` method.

If the `-jar` *jarfile* command-line option is used, `cvm` launches the application in the `jar` file. The manifest of the `jar` file must contain a line of the form `MainClass:`*classname*. The *classname* string identifies the class having the `main` method which serves as the application's starting point.

Section 3.1, "Launching a Java Application," has more information about launching Java applications with `cvm`.

A.3 Options

`cvm` borrows some of its command-line options from java, the Java SE application launcher. Other options are unique to `cvm` and may require certain build options to enable the necessary runtime features. For command-line options that take a *size* parameter, the default units for size are bytes. Append the letter `k` or `K` to indicate kilobytes, or `m` or `M` to indicate megabytes.

Table A–1 describes the command-line options that are shared with the Java SE application launcher.

Table A–1 Java SE Command-Line Options

Option	Description
`-help`	Display usage information and exit.
`-showversion`	Display product version information and continue.
`-version`	Display product version information and exit.
`-fullversion`	Display build version information and exit.
`-D`*`property`*`=`*`value`*	Set a system property value. See Appendix C, "Java ME System Properties" for a description of security properties for CDC.
`-classpath` *`classpath`* `-cp` *`classpath`*	Specify an alternate user class path.[1] The default user class path is the current directory.
`-Xbootclasspath[/a | /p]:`*classpath*	Specify the extension class path.[2] `/a` appends *classpath* list to the default path. `/p` prepends *classpath* list to the default path.
`-Xms`*size*	Set the start size of the memory allocation pool (heap). This value must be greater than 1000 bytes. The default value is 2M. NOTE: This option is ignored by the generational garbage collector, though it could be used by other garbage collectors.
-Xmx*size*	Set the maximum heap size (high water mark). The default value is 7M.
`-Xmn`*size*	Set the minimum heap size (low water mark). The default value is 1M.
`-Xss`*size*	Each Java thread has two stacks: one for Java code and one for native code. The maximum native stack size of the main thread is determined by the native application launcher (e.g. shell, OS, etc.). For subsequent threads, the maximum native stack size is set by the `-Xss` option, although this can be ignored by the underlying OS. See Table A–4 for a description of the command-line options for controlling the size of the Java stack. The default value is `0` which indicates that the value is actually set by the native environment.

Table A–1 (Cont.) Java SE Command-Line Options

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>-enableassertions [:<<i>package</i>>… | :<<i>class</i>>]
-ea [:<<i>package</i>>... | :<<i>class</i>>]</td><td>Enable Java assertions. These are disabled by default. With no arguments, this switch enables assertions for all user classes. With one argument ending in ..., the switch enables assertions in the specified package and any subpackages. If the argument is simply ..., the switch enables assertions in the unnamed package in the current working directory. With one argument not ending in ..., the switch enables assertions in the specified class.

If a single command line contains multiple instances of these switches, they are processed in order before loading any classes. So, for example, to run a program with assertions enabled only in the package com.wombat.fruitbat (and any subpackages), the following command could be used:

% cvm -ea:com.wombat.fruitbat … <<i>MainClass</i>>

The -enableassertions and -ea switches apply to all class loaders and to system classes (which do not have a class loader). There is one exception to this rule: in their no-argument form, the switches do not apply to system. This makes it easy to turn on assertions in all classes except for system classes. The -enablesystemassertions option enables asserts in all system classes (that is, it sets the default assertion status for system classes to true). To run a program with assertions enabled in the package com.wombat.fruitbat but disabled in class com.wombat.fruitbat.Brickbat, the following command could be used:

% cvm -ea:com.wombat.fruitbat… \
    -da:com.wombat.fruitbat.Brickbat <<i>MainClass</i>></td></tr>
<tr><td>-disableassertions [:<<i>package</i>>… | :<<i>class</i>>]
-da [:<<i>package</i>>... | :<<i>class</i>>]</td><td>Disable Java assertions. This is the default behavior.

With no arguments, -disableassertions or -da disables assertions. With one argument ending in ..., the option disables assertions in the specified package and any subpackages. If the argument is simply ..., the switch disables assertions in the unnamed package in the current working directory. With one argument not ending in ..., the switch disables assertions in the specified class.

The -disableassertions and -da switches apply to all class loaders and to system classes that do not have a class loader. There is one exception to this rule: in their no-argument form, the switches do not apply to system. This makes it easy to turn on assertions in all classes except for system classes. A separate switch is provided to enable assertions in all system classes. See the description of the -disablesystemassertions option.</td></tr>
<tr><td>-enablesystemassertions
-esa</td><td>Enable assertions in all system classes (sets the default assertion status for system classes to true).</td></tr>
<tr><td>-disablesystemassertions
-dsa</td><td>Disable assertions in all system classes.</td></tr>
</table>

[1]See Section 3.3, "Class Search Path Basics" and http://download.oracle.com/javase/1.4.2/docs/tooldocs/tools.html for more information about class search paths.

[2]See Section 3.3, "Class Search Path Basics" and http://download.oracle.com/javase/1.4.2/docs/tooldocs/tools.html for more information about class search paths.

Table A–2 describes the CDC-specific command-line options.

Table A–2 CDC-Specific Command-Line Options

Option	Description
`-XbuildOptions`	Display build options and exit.
`-XshowBuildOptions`	Display build options and continue.
-XappName=*value*	Specify the application name for QPE. This is used to identify the `cvm` process for native application management and control.
`-Xverify:[all | remote | none]`	Perform class verification. ■ `all` verify all classes. ■ `remote` verify all but preloaded and system classes. ■ `none` don ■ 't perform class verification. The default value is `remote`. If `-Xverify` is used without any arguments, the value is `all`.
`-XfullShutdown`	Make sure all resources are freed and the VM destroyed upon exit. This is the default for non-process-model operating systems, but is not needed for process-model operating systems, such as Linux.
`-Xgc:`*suboption*	Specify GC-specific options. The default GC is the generational garbage collector described in Chapter 3, "Running Applications." See Table A–3 for a description of the suboptions. Other garbage collectors are unsupported.
`-Xopt:`*suboption*	Control the Java stack. See Table A–4 for a description of the suboptions. The different suboptions can be appended into a single argument with name/value pair separated by commas.
-XtimeStamping	Enable timestamping.
`-Xtrace:`*flags*	Turn on trace flags. Table A–5 shows the hexadecimal values to turn on each trace flag. To turn on multiple flags, bitwise-OR the values of all the flags you want to turn on, and use that result as the `-Xtrace` value. Requires the `CVM_TRACE=true` build option. (Unsupported.)

Table A–3 describes the suboptions for the `-Xgc` command-line option.

Table A–3 `-Xgc:`suboption

Option	Description
`maxStackMapsMemorySize=`*size*	Set the size of the stack map cache. The default value is `0xFFFFFFFF`.
stat	Collect and display garbage collection statistics.
`youngGen=`*size*	Set the size of the young object generation. NOTE: this option is specific to the default generational collector. The default value is `1M`.

Table A–4 describes the suboptions for the `-Xopt` command-line option, which controls the size of the Java stack. This option is useful for runtime development purposes only and is unsupported.

Table A–4 `-Xopt:`***suboption***

Suboption	Description
`stackMinSize=`*size*	Set the initial size of the Java stack, from <32…65536>. The default for JIT-based systems is `3K` and the default for non-JIT based systems is `1K`.
`stackMaxSize=`*size*	Set the maximum size of the stack, from `<1024…1048576>`. The default for `128K`.
`stackChunkSize=`*size*	Set the amount the stack grows when it needs to expand `<32…65536>`. The default for JIT-based systems is `2K` and the default for non-JIT based systems is `1K`.

Table A–5 describes the flags used by the `-Xtrace` command-line option. This option is useful for runtime development purposes only and is unsupported.

Table A–5 `-Xtrace:`***flags (unsupported)***

Value	Description
`0x00000001`	Opcode execution.
`0x00000002`	Method execution.
`0x00000004`	Internal state of the interpreter loop on method calls and returns.
`0x00000008`	Fast common-case path of Java synchronization.
`0x00000010`	Slow rare-case path of Java synchronization.
`0x00000020`	Mutex locking and unlocking operations.
`0x00000040`	Consistent state transitions. Garbage Collection (GC)-safety state only.
`0x00000080`	GC start and stop notifications.
`0x00000100`	GC root scans.
`0x00000200`	GC heap object scans.
`0x00000400`	GC object allocation.
`0x00000800`	GC algorithm internals.
`0x00001000`	Transitions between GC-safe and GC-unsafe states.
`0x00002000`	Class static initializers.
`0x00004000`	Java exception handling.
`0x00008000`	Heap initialization and destruction, global state initialization, and the safe exit feature.
`0x00010000`	Read and write barriers for GC.
`0x00020000`	Generation of GC maps for Java stacks.
`0x00040000`	Class loading.
`0x00080000`	Class lookup in VM-internal tables.
`0x00100000`	Type system operations.
`0x00200000`	Java code verifier operations.
`0x00400000`	Weak reference handling.
`0x00800000`	Class unloading.
`0x01000000`	Class linking.

Table A–6 describes the command-line options available with the `CVM_JVMTI` build option. See Chapter 6, "Developer Tools," for an example of how to use these command-line options.

Table A–6 JVMTI Options

Option	Description
`-Xdebug`	Enable VM-level debugging support.
`-Xrun`*lib*`:[help]`|`[`*option*`=`*value*`, ...]`	Enable feature in shared library. For example, `hprof` profiling support.

Table A–7 describes the command-line options available with the `CVM_JIT=true` build option. See Chapter 3, "Running Applications," for an example of how to use these command-line options.

Table A–7 `-Xjit`***:options***

Option	Default	Description
`bcost=`*cost*	4	Cost of a backwards branch, between `<0...32767>`.
`climit=`*cost*	20000	The popularity threshold for a given method, between `<0...65535>`. The VM compares a per-method count based on `bcost`, `icost` and `mcost` against this threshold to determine when to compile a given method.
`codeCacheSize=`*value*	512k	Size of code cache where compiled methods are stored, between `<0...32M>`.
`compile=`*suboption*	policy	When to compile methods. See Table A–9 for descriptions of the suboptions for `compile`. The default policy is based on the suboption defaults listed in this table.
`icost=`*cost*	20	Cost of an interpreted-to-interpreted method call, between `<0...32767>`.
`inline=`*suboption*	all	Perform method inlining when compiling. See Table A–8 for descriptions of the suboptions for `inline`.
`lowerCodeCacheThreshold=`*percentage*	90%	Lower code cache threshold, between `<0%..100%>`. The dynamic compiler decompiles methods until the code cache reaches this threshold.
`maxCompiledMethodSize=`*value*	65535	Maximum size of a compiled method, between `<0...64K>`.
`maxInliningCodeLength=`*value*	68	Maximum size of an inlined method, between `<0...1000>`. This value is used as a threshold that proportionally decreases with the depth of inlining. Therefore, shorter methods are inlined at deeper depths. In addition, if the inlined method is less than *value*`/2`, the dynamic compiler allows unquickened opcodes in the inlined method.
`maxInliningDepth=`*value*	12	Maximum inlining depth of inlined methods/frames, between `<0...1000>`.
`maxWorkingMemorySize=`*value*	512k	Maximum working memory size for the dynamic compiler, between `<0...64M>`. See Section 3.4.4, "Setting the Maximum Working Memory for the Dynamic Compiler."

Table A–7 (Cont.) `-Xjit`***:options***

Option	Default	Description
`mcost=`*cost*	50	Cost for transitioning between a compiled method and an interpreted method, and vice versa. Between `<0..32767>`.
`minInliningCodeLength=`*value*	16	The floor value for `maxInliningCodeLength` when its size is proportionally decreased at greater inlining depths.
`policyTriggeredDecompilations=`*boolean*	true	Policy triggered decompilations. If `false`, then never decompiles a method to make room for more compilations. Methods remain compiled until the class is unloaded, even if the code cache is full.
trace=*suboption*		Set dynamic compiler trace options. See Table A–10.
`upperCodeCacheThreshold=`*percentage*	95	Upper code cache threshold, between `<0%...100%>`. The dynamic compiler starts decompiling methods during a GC when the code cache passes this threshold unless `policyTriggeredDecompilations=false.`
`XregisterPhis=`*boolean*	true	*Unsupported.*
`XcompilingCausesClassLoading=`*boolean*	false	*Unsupported.*
`Xpmi=`*boolean*	true	*Unsupported.*
`XregisterLocals=`*boolean*	true	*Unsupported.*
`aot=`*boolean*	`true`	Enable/disable AOTC.
`aotFile=`*file*		AOTC file path.
`recompileAOT=`*boolean*	`false`	Recompile AOTC code when this option is set to `true`. The existing AOTC code is replaced when this option is used.
`aotCodeCacheSize=`*size*	672K	Size for the code cache used for AOTC.
`aotMethodList=`*file*		File containing a list of methods to be compiled and saved for AOTC.

Table A–8 describes the command-line options for selecting when to inline methods.

Table A–8 `-Xjit:inline=`***suboption***

Suboption	Description
`all`	Enable all the options listed below to perform inlining whenever possible. The default.
`none`	Do not perform inlining.
`virtual`	Perform inlining on virtual methods.
`nonvirtual`	Perform inlining on nonvirtual methods.
`vhints`	Virtual hints. Use hints gathered while interpreting a method to choose a target method to get inlined when an `invokevirtual` opcode is compiled.
`ihints`	Interface hints. Use hints gathered while interpreting a method to choose a target method for inlining when an `invokeinterface` opcode is compiled.

Table A–8 (Cont.) `-Xjit:inline=`***suboption***

Suboption	Description
`Xvsync`	Inline virtual synchronized methods. Off by default. *Unsupported.*
`Xnvsync`	Inline non-virtual synchronized methods. Off by default. *Unsupported.*
`Xdopriv`	Inline privileged methods specified by `java.security.AccessController.doPrivileged()`. On by default. *Unsupported.*

Table A–9 describes the top-level command-line options that control dynamic compiler policies.

Table A–9 `-Xjit:compile`***=suboption***

Suboption	Description
`policy`	Compile according to existing compilation policy parameters such as `icost` and `climit`. The default.
`all`	Compile all methods aggressively. *Note:* this hurts performance and should be used only for testing the dynamic compiler.
`none`	Do not compile any methods.

Table A–10 describes the command-line options for controlling dynamic compiler tracing. These options require a build with `CVM_TRACE_JIT=true`. These options are experimental and unsupported.

Table A–10 `-Xjit:trace`***=option***

Suboption	Description
`bctoir`	Print information regarding the conversion of Java bytecodes to the JIT internal representation (IR), including a complete dump of all IR nodes.
`codegen`	Print the generated code in a format similar to the assembler language of the target processor. If the build option `CVM_JIT_DEBUG=true`, then this also prints the `JavaCodeSelect` rule used to generate the code interspersed with the generated code.
`inlining`	Print method inlining information during the bytecode to IR pass, such as which methods were inlined and which ones were not.
`iropt`	Print information about optimizations done in the bytecode to IR pass.
`osr`	Print a message when compilation of a method is triggered by on stack replacement (OSR).
`stats`	Print statistics gathered during compilation.
`status`	Print a line of status each time a method is compiled. The output includes the name of the method and whether or not it was compiled successfully.

B

Serial Port Configuration Notes

The `javax.microedition.io.CommConnection` interface allows a CDC Java runtime environment to expose an OS-level serial port as a logical serial port connection. This appendix shows how to configure an OS-level serial port on a Linux system so that a Java application can access the corresponding logical serial port connection.

> **Note:** While this example is based on the RS-232 serial interface implementation of `CommConnection` in `com.sun.cdc.io.j2me.comm.Protocol`, an alternate implementation could use the `CommConnection` interface to support other forms of serial communication such as IrDA.

Table B–1 Serial Communications References

Interface	Document
RS-232 serial communications	http://www.tldp.org/HOWTO/Serial-HOWTO-4.html
`minicom` serial communications program	`minicom`
Serial port configuration	`setserialport`
Serial port driver interface	`ttyS`

B.1 Serial Port Setup

1. Setup a serial cable connection between two Linux computers.

   Become super-user.

   ```
   % su
   #
   ```

   This step is necessary to allow non-root users to access the serial port.

2. Configure the serial port to use IRQ 4.

   ```
   # setserial /dev/ttyS0 irq 4
   ```

3. Change the file access permissions for the serial port and the lock file.

   ```
   # chmod 777 /dev/ttyS0 /var/lock
   ```

   This allows other users to access the serial port.

4. Launch the `minicom` serial communications program in setup mode.

```
# minicom -s
```

   a. Select `Serial port setup` from the `[configuration]` menu.
   b. In the setup menu, type `A` to change the `Serial Device` setting.

      If the `Serial Device` setting is `/dev/modem`, then change it to `/dev/ttyS0`.
   c. Press **<ENTER>** to confirm the change.
   d. Press **<ENTER>** again to exit the setup menu.
   e. Select the **Save setup as dfl** menu option.
   f. Select the **Exit** menu option.

      This initializes the serial port.
   g. Type **<CONTROL>-a q** to finally exit `minicom`.
5. Follow a similar configuration procedure with the other computer connected to the serial cable.

B.2 OS-Level Testing

The serial connection between the two computers can be tested with the `minicom` serial communications program.

1. Remotely login to each computer.
2. Launch the `minicom`(1) serial communications program on each computer.
3. Type some text into one of the `minicom` windows.
4. Type **<CONTROL>-a q** to finally exit `minicom`.

This should determine that the serial connection is correct.

C

Java ME System Properties

In addition to the standard Java SE system properties, CDC supports the standard Java ME system properties supported by CLDC 1.1 and MIDP 2.0. These system properties are described in Table C–1.

Table C–1 CDC System Properties

System Property	Default Value	Description
`microedition.commports`	No default	Comma-delimited list of available communications ports
`microedition.configuration`	`cdc`	Java ME configuration
`microedition.encoding`	`ISO_LATIN_1`	Unicode character encoding
`microedition.hostname`	No default	Host platform
`microedition.locale`	`en-US`	System locale
`microedition.platform`	`j2me`	Java platform
`microedition.profiles`	No default	Java ME profile
`microedition.securerandom.nofallback`	`false`	Disable the mechanism that allows the CDC Java runtime environment to fallback to using `/dev/urandom` if `/dev/random` doesn't have enough entropy to work properly. See Section 4.2.4, "Seed Generation for Random Number Generation" for more information.
`cdcams.decorations`	`false`	Display native window decorations.
`cdcams.presentation`	No default	Top-level presentation mode class.
`cdcams.repository`	*CVMHOME*`/repository`	Location of application repository.
`cdcams.verbose`	`false`	Display extra diagnostic information.
`java.ext.dirs`	*CVMHOME*`/lib`	Specifies one or more directories to search for installed optional packages, each separated by `File.pathSeparatorChar`.

For a list of the standard Java SE system properties, see the description of `java.lang.System.getProperties()` in the CDC specification.