# Oracle® Java Micro Edition Embedded Client

Developer's Guide

Release 1.1.1

**E20632-03**

May 2013

This documentation is for application developers. It describes compiling, debugging, and profiling.

ORACLE®

Oracle Java Micro Edition Embedded Client Developer's Guide, Release 1.1.1

E20632-03

# Contents

## 4 Profiling With NetBeans and jvmtihprof



## 5 Diagnosing Memory Leaks

# Preface

This manual explains how to use the Oracle Java Micro Edition Embedded Client (Oracle Java ME Embedded Client) to create and test applications.

## Audience

This document is intended for application developers for the Oracle Java ME Embedded Client. It is also useful for implementers of Oracle Java ME Embedded Client platforms who wish to test their implementation.

## Documentation Accessibility

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

### Access to Oracle Support

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

## Operating System Commands

This document might not contain information about basic Linux commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following resource for this information:

- Ubuntu operating system documentation, which is found at:

  https://help.ubuntu.com

## Related Documents

For more information, see the following documents in the Oracle Java Micro Edition Embedded Client documentation set:

- *Oracle Java Micro Edition Embedded Client Architecture Guide*
- *Oracle Java Micro Edition Embedded Client Customization Guide*

**Note:** The *Oracle Java Micro Edition Embedded Client Architecture Guide* is a prerequisite for all Oracle Java Micro Edition Embedded Client guides. It defines concepts that are mentioned in the other guides.

## 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

This chapter describes the Oracle Java ME Embedded Client components, developer tools, and application programming interface (API) documentation.

This chapter includes the following topics:

- Section 1.1, "Overview"
- Section 1.2, "Developer Tools"
- Section 1.3, "Finding API Documentation"

## 1.1 Overview

The Oracle Java ME Embedded Client, version 1.1.1 is based on the Connected Device Configuration of Java Micro Edition (JavaME), also known as CDC. The product also includes the Foundation Profile (FP) and optional packages based on Java Service Requests (JSRs) in the Java Community Process.

Refer to the *Oracle Java Micro Edition Embedded Client Architecture Guide* for a more detailed overview of this release.

## 1.2 Developer Tools

The Oracle Java ME Embedded Client allows developers to develop, build and test embedded applications. Limited support is provided for IDE environments such as NetBeans. Developers can work with command line tools directly, rather than using an IDE interface.

For an overview of command line options, refer to Chapter 2, "Compiling Your Application."

The *Oracle Java Micro Edition Embedded Client Installation Guide* describes how to configure these IDEs to work with the SDK.

## 1.3 Finding API Documentation

The API documentation for CDC, FP, and for JSRs supported in this release, are found online at:

`http://docs.oracle.com/javame/embedded.html`

If you prefer to download the API documentation for a configuration, profile or optional package and install it locally, visit the Java Community Process (JCP) program web site. For example, for JSR 218 (CDC 1.1):

`http://jcp.org/en/jsr/detail?id=218`

# 2 Compiling Your Application

This chapter describes compiling an application from the command line or integrated development environment (IDE).

This chapter includes these topics:

- Section 2.1, "Compiling from the Command Line"
- Section 2.2, "Compiling in an IDE"

## 2.1 Compiling from the Command Line

You need to compile your application on the development workstation (Linux) to create class files that can then be deployed to the target device. Simple shell scripts and command lines are presented in this section to illustrate simple compilation and compiling for debugging or profiling.

### 2.1.1 Prerequisites

Before compiling, JDK 6.0 must be installed on your development workstation. If you do not yet have it, follow the instructions in Section 2.1.1.1, "Installing the JDK."

#### 2.1.1.1 Installing the JDK

Follow these steps to install the JDK.

1. Download JDK 6 update 45 from `http://www.oracle.com/technetwork/java/javase/downloads/jdk6downloads-1902814.html`.
   If you download a later JDK version, modify the next two steps accordingly.
2. Open a Linux terminal and cd into the directory where JDK 6 has been downloaded. After installation, the environment variable `JDK_INSTALL_DIR` will be set to this directory.

   Set file permissions to enable execution with the following command:

   ```
   sudo chmod a+x jdk-6u45-linux-i586.bin
   ```

3. Run the installer with the following command:

   ```
   ./jdk-6u45-linux-i586.bin
   ```

### 2.1.2 Bytecode Requirements for CDC Applications

An Oracle Java ME Embedded Client program must comply with the requirements of the Connected Device Configuration (CDC), version 1.1.2. The correct bytecode will be

generated for CDC provided that you specify JDK 1.4 at the command line with both the -source and the -target options. Refer to Section 2.1.3, "Compiling an Application" for an example.

## 2.1.3 Compiling an Application

1. On the workstation, cd into the base directory where the source files are located.
2. Compile with the command

   ```
   javac -source 1.4 -target 1.4 helloworld/HelloWorld.java
   ```

   The class file `HelloWorld.class` will be created in the `helloworld` directory

> **Note:** The value 1.4 must be assigned to the options -source and -target to ensure that the compiled byte codes are compatible with CDC.

## 2.1.4 Setting Environment Variables

You should make sure that these environment variables are set correctly before running a compiled program on either your development workstation or on the target device.

> **Note:** This section uses the bash shell. Make adjustments if you use a different shell.

### 2.1.4.1 `CVM_HOME`

You need to set the environment variable `CVM_HOME` to:

*InstallDir*`/Oracle_JavaME_Embedded_Client/binaries`

Where *InstallDir* is the directory on your Linux workstation where you installed the Oracle Java ME Embedded Client.

### 2.1.4.2 `PATH`

Your `PATH` environment variable should be set to include the location of the `javac` command. In these instructions, *`Version`* stands for the JDK version you have installed, for example, 1.6.0_45.

1. Edit either of the files `~/.bashrc` or `~/.bash_profile`.
2. Append the following lines:

   ```
   export JAVA_HOME=/home/myname/tools/jdkVersion
   ```

   ```
   export PATH=/home/myname/bin:$PATH/tools/jdkVersion
   ```

3. Log out of your Linux account and log back in.
4. Use the following command to verify that javac can be found:

   ```
   which java
   ```

> **Note:** You can avoid setting the `PATH` on the target device by entering the relative path or full path to the `cvm` executable on the command line.

### 2.1.5 Running an Application on the Target Device

1. Make sure the class file is accessible by the target host.

   This is typically accomplished by mounting the base directory (where the source and class files are located) as an nfs file system on the device. Or, on devices with less capability, the class file can be copied to a suitable directory on the device with a command such as `ftp` or `scp`.

2. From the workstation, open a terminal window that connects to the target device using a protocol such as ssh or telnet.

3. Change directory to that containing the `cvm` executable with the command

   `cd` *cvm_install_dir*`/bin`

   where cvm_install_dir is the location where Oracle Java ME Embedded Client was installed on the device.

4. Run the program on the device with the command

   ```
   ./cvm -cp /home/myname/working/HelloWorld  helloworld.HelloWorld
   ```
   assuming the first argument is the working directory on the device containing the class file.

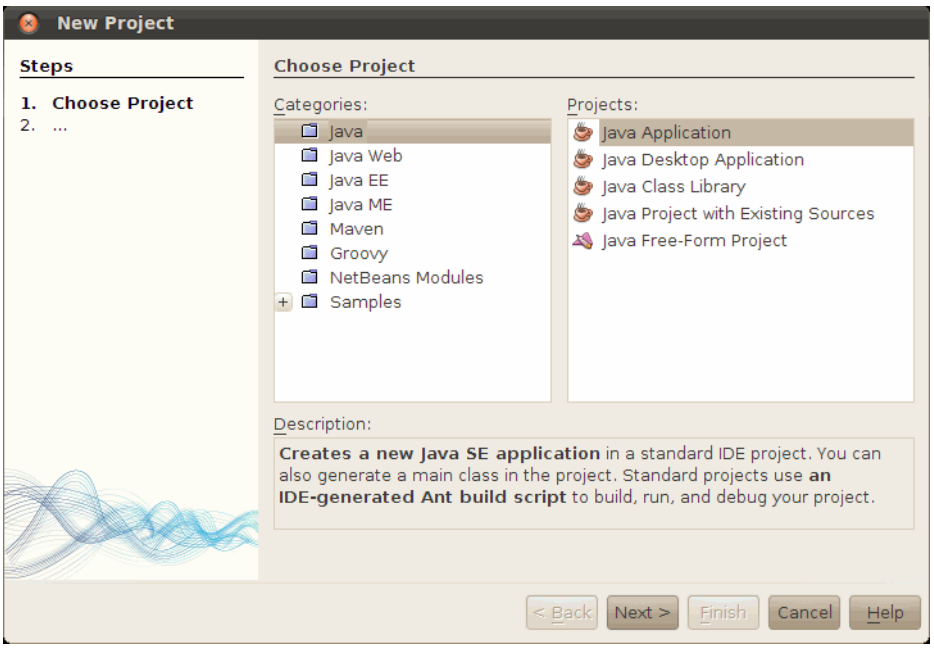
5. Check the terminal output to validate the result.

## 2.2 Compiling in an IDE

If you prefer to work within an IDE such as NetBeans or Eclipse, you can compile an Oracle Java ME Embedded Client application by setting the project type to Java application, and specifying options to make the compiler generate bytecodes compatible with Java 1.4. For NetBeans, this is illustrated in the following steps.

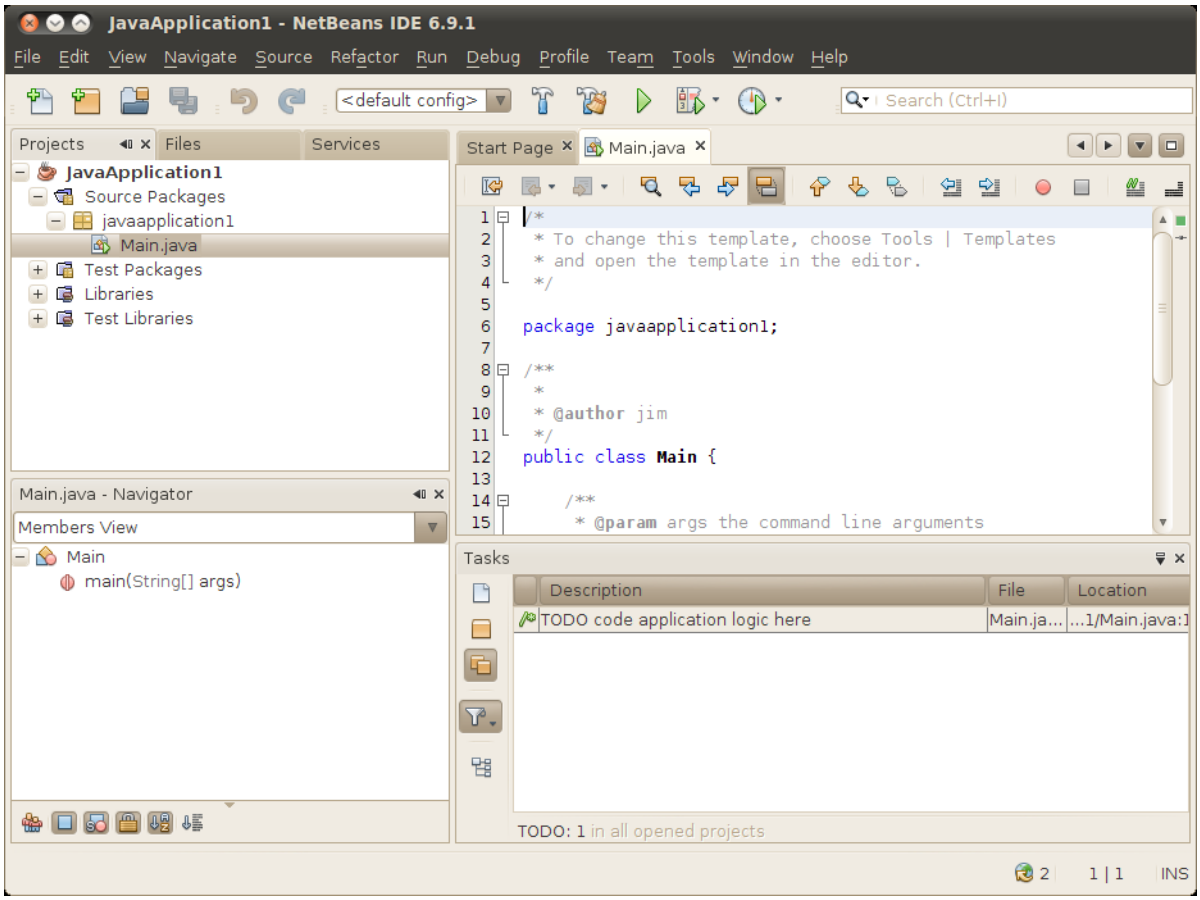
1. Create a new project in NetBeans. Choose the category Java and project type Java Application, as shown:

***Figure 2–1 Create New java Application Project***



The newly created project creates a default Main class program.

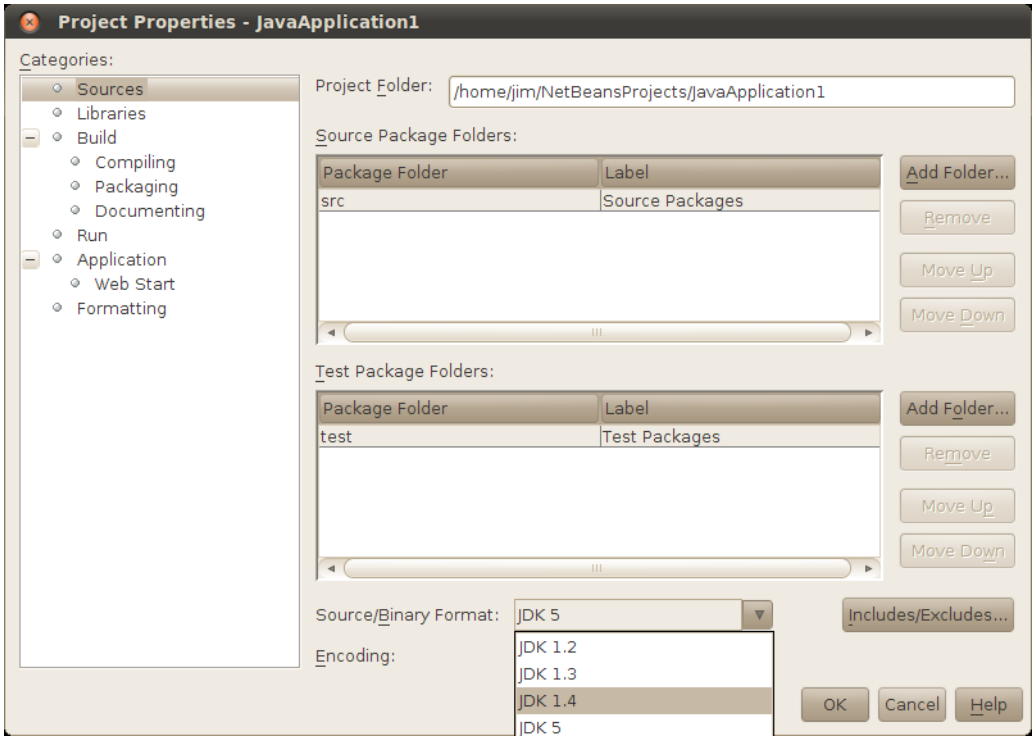
***Figure 2–2 New Java Project in NetBeans***




Since this is a CDC program, you must change the compiler options to generate JDK 1.4 bytecodes. Select your project in the Projects page and right-click to get the Project Properties dialog box.

Under Categories, choose the first item, "Sources." Near the bottom of the dialog, pull down the Source/Binary Format menu, and select JDK 1.4.

***Figure 2–3 Selecting Java 1.4 Encoding***




## 2.2.1 Setting `bootclasspath` in an IDE

It is advisable to set `bootclasspath` when working in an IDE.

# 3 Debugging With NetBeans

This chapter describes how to debug an application with the NetBeans integrated development environment (IDE).

This chapter includes these topics:



## 3.1 Introduction

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.

## 3.2 Launching `cvm` in Debug Mode

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

> **Note:** With the Oracle Java ME Embedded Client, cvm is installed on your system in the following location:
>
> *InstallDir*`/Oracle_JavaME_Embedded_Client/binaries/bin/cvm`

### 3.2.1 `cvm` Debug Mode Syntax

Example 3–1 and Example 3–2 show how to launch `cvm` in debug mode on a target device. These examples assume that the target device runs a Unix-style operating

system and that socket networking is operational. Make adjustments as necessary for your target device. Use nfs networking if available to mount the workstation directory containing the compiled classfiles on the target device. Or, copy the classfiles from the workstation to the device using ftp or scp.

Be sure to set up the environment variables correctly before running cvm. For example,

```
CVM_HOME=/mnt/sda1/work/cvm
CLASSPATH=/mnt/sda1/work
CLASSNAME=helloworld.Helloworld
```

Chose the appropriate command syntax and sub-options to launch cvm in debug mode.

For the appropriate command syntax and a list of debug sub-options, run

```
cvm -agentlib:jdwp=help
```

or refer to
http://download.oracle.com/javase/1.5.0/docs/guide/jpda/conninv.html.

***Example 3–1 cvm Listens for Connection from Debugger***

```
% cvm -agentlib:jdwp=transport=dt_socket,server=y,address=port -Xdebug \
-classpath $CLASSPATH $CLASSNAME
```

***Example 3–2 cvm Connects to Debugger***

```
% cvm -agentlib:jdwp=transport=dt_socket,server=n,address=host:port -Xdebug \
-classpath $CLASSPATH $CLASSNAME
```

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

- -agentlib:jdwp and the transport and address sub-options 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.

## 3.2.2 cvm Debug Mode Example

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

***Example 3–3 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
```

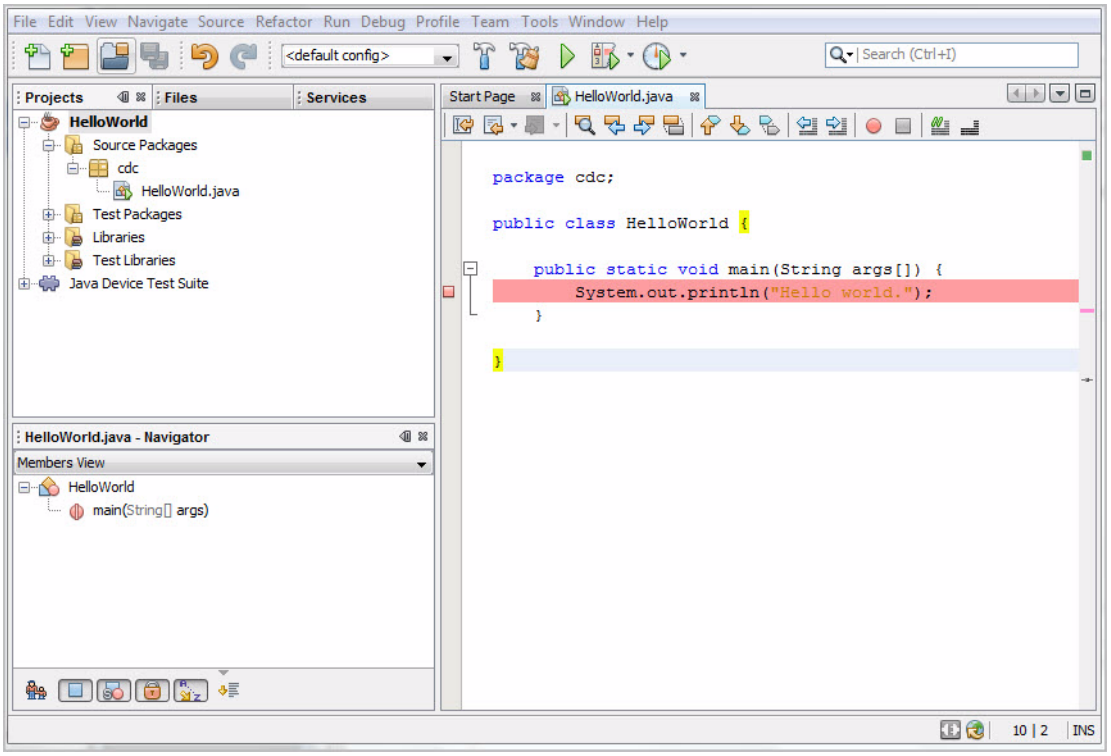
## 3.3 Attaching the NetBeans IDE Debugger to cvm

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 3–1 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.

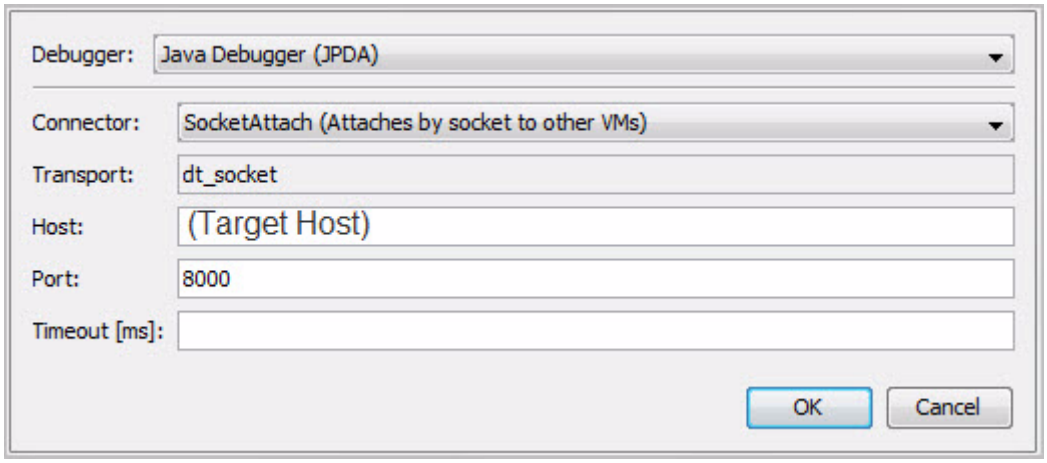
***Figure 3–1 Breakpoint in*** `HelloWorld.java`




2. On the target host, launch cvm with `server=y` and, for this example, `address=8000`, similar to the example in Section 3.2.2, "cvm Debug Mode Example".
3. In NetBeans, choose **Debug** > **Attach Debugger**.
4. Set up the Attach Debugger dialog as shown in Figure 3–2.

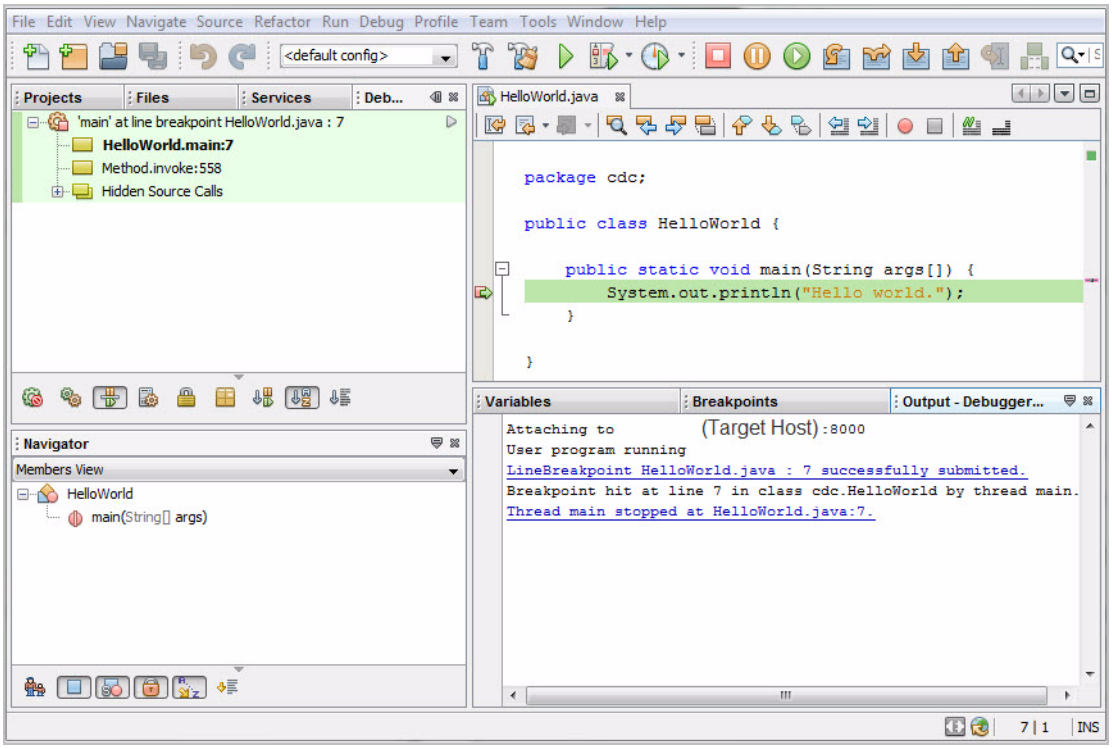
   Substitute the target host name for (Target Host).

***Figure 3–2 Attach Debugger Dialog (Debugger as Client)***



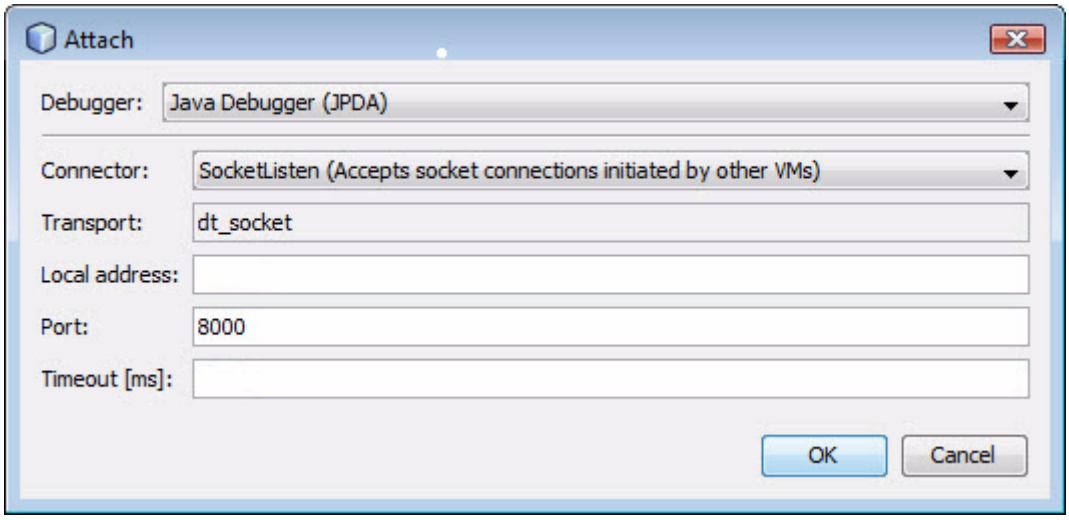

The debugger connects and indicates that execution has stopped at the breakpoint as similar to Figure 3–3.

***Figure 3–3 Debugger Connected and Stopped at Breakpoint***




If you want the debugger to be the server, complete the Attach Debugger dialog similar to Figure 3–4. After clicking **OK**, launch `cvm` on the target host with `server=n` and `address=` the debuggers' host and port.

***Figure 3–4 Debugger as Server Attach Debugger Setup***




## 3.4 Attaching to cvm with jdb

After launching cvm as a debug server (see Section 3.2.2, "cvm Debug Mode Example") on the target host, you can connect to it with the jdb command line debugger using syntax similar to Example 3–4. The jdb command is in *JavaSEinstall*/bin/.

***Example 3–4 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]
```

# 4

# Profiling With NetBeans and jvmtihprof

This chapter describes two options for profiling application performance and memory usage.

This chapter includes these topics:



## 4.1 Introduction

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. For more information, see `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`

## 4.2 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.

---

### 4.2.1 Calibrate the Profiler Agent

1. On the target host, calibrate the profiler agent by issuing commands equivalent to those shown in Example 4–1.

   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.

***Example 4–1 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
```

### 4.2.2 Start `cvm` with the Profiler Agent

1. Launch `cvm` with the profiler agent using a command equivalent to that shown in Example 4–2 for a Linux target host.

***Example 4–2 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.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**.

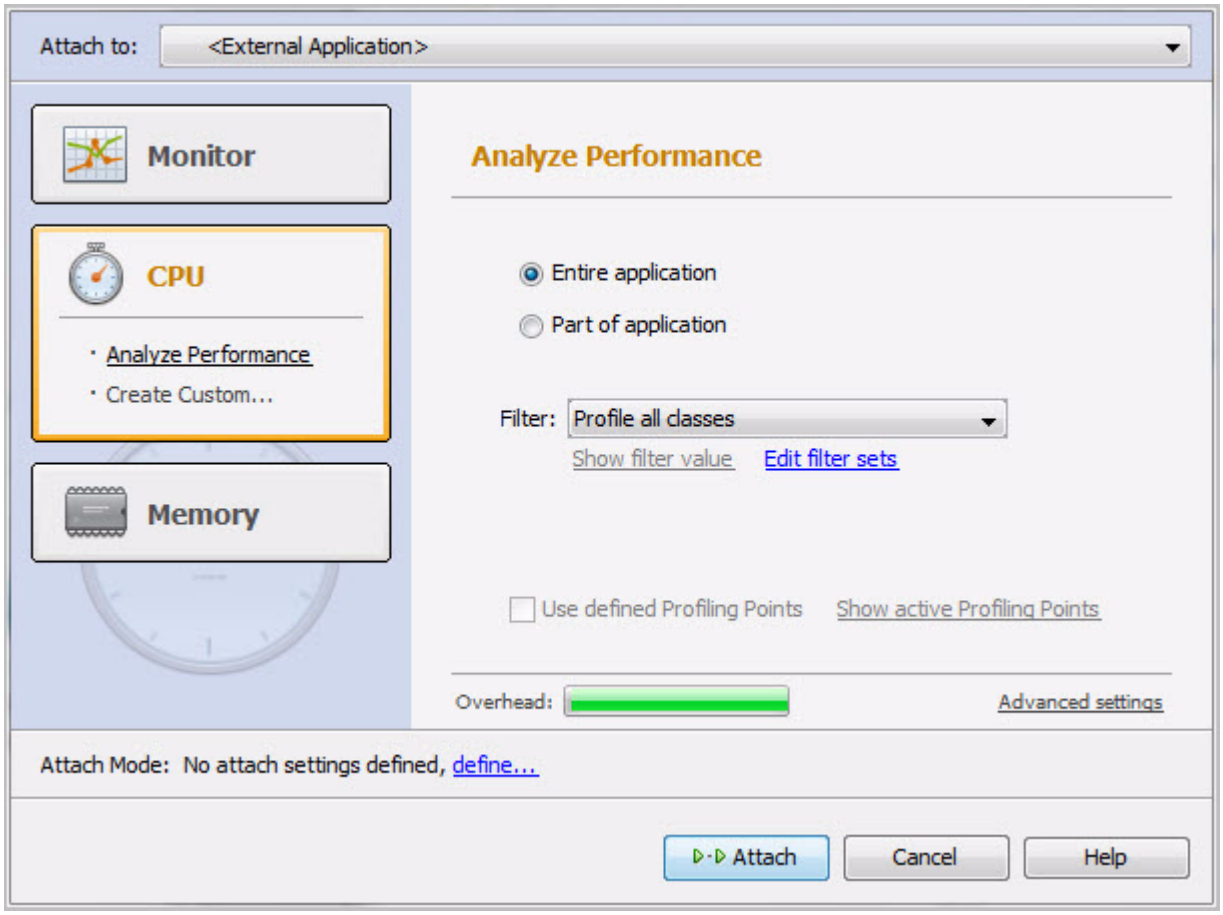
`libprofilerinterface.so` is a shared native code library. Building CDC creates this file.

### 4.2.3 Attach the NetBeans Profiler

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

   The Attach Profiler dialog appears, similar to Figure 4–1.

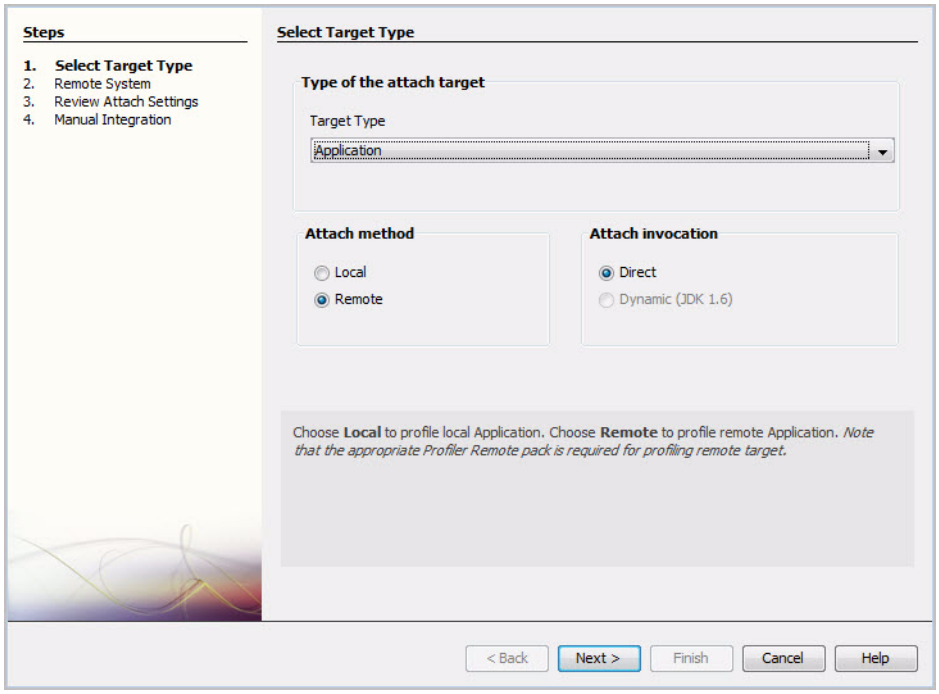
***Figure 4–1 Attach Profiler Dialog***




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

   The Select Target Type screen appears, similar to Figure 4–2.

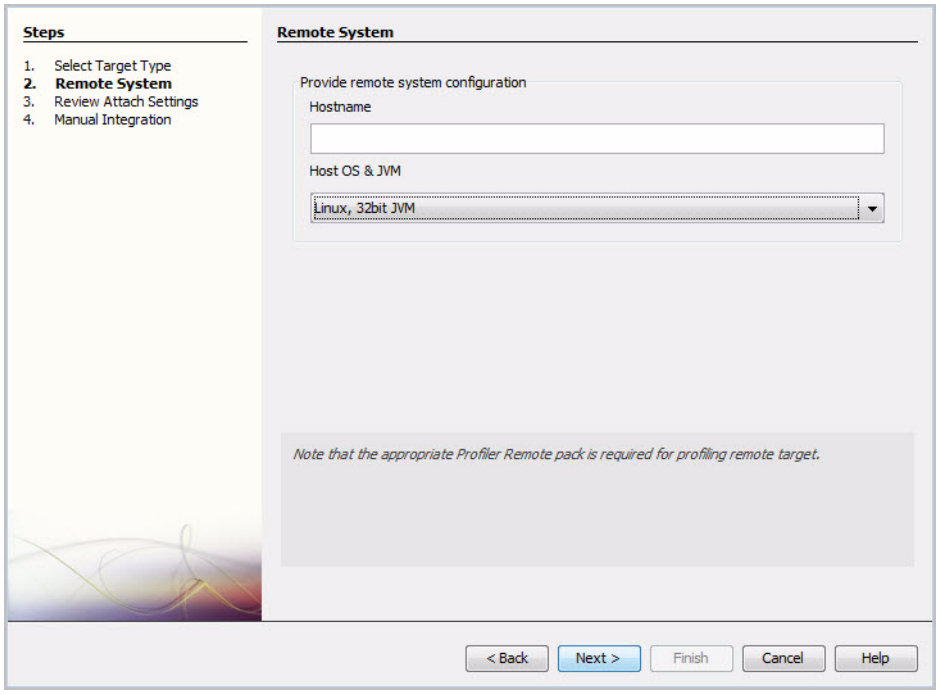
*Figure 4–2 Select Target Type Screen*




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 4–3.

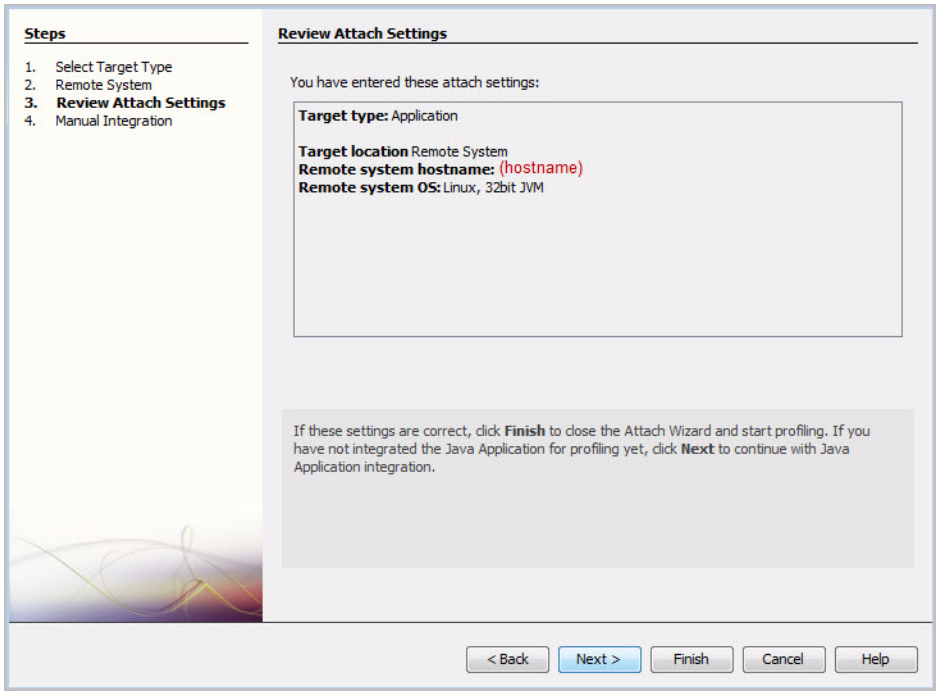
*Figure 4–3 Remote System Screen*




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 4–4.

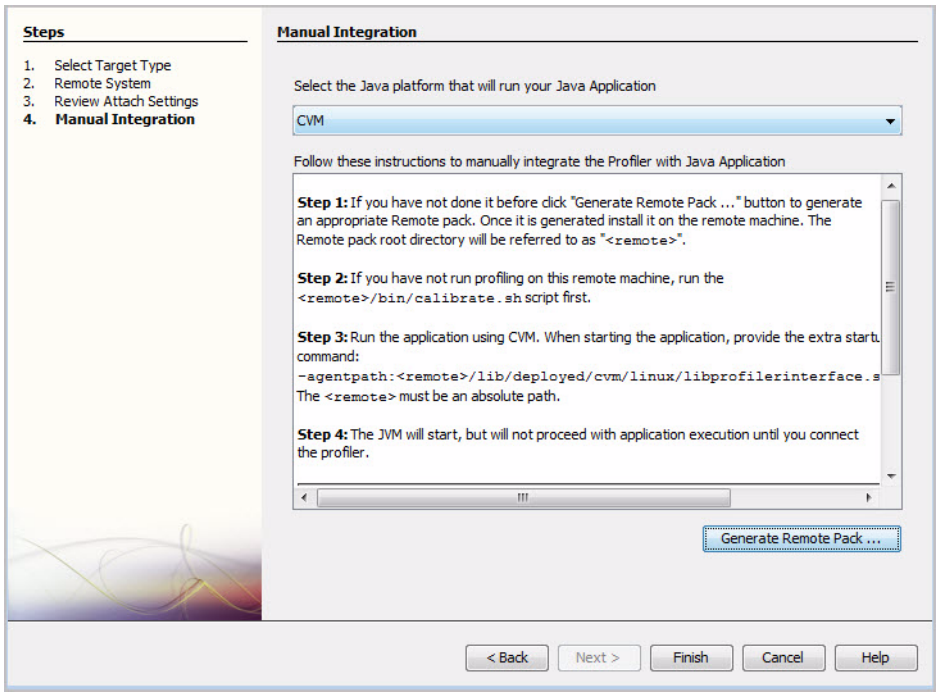
*Figure 4–4 Review Attach Settings Screen*




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

The Manual Integration screen appears, similar to Figure 4–5.

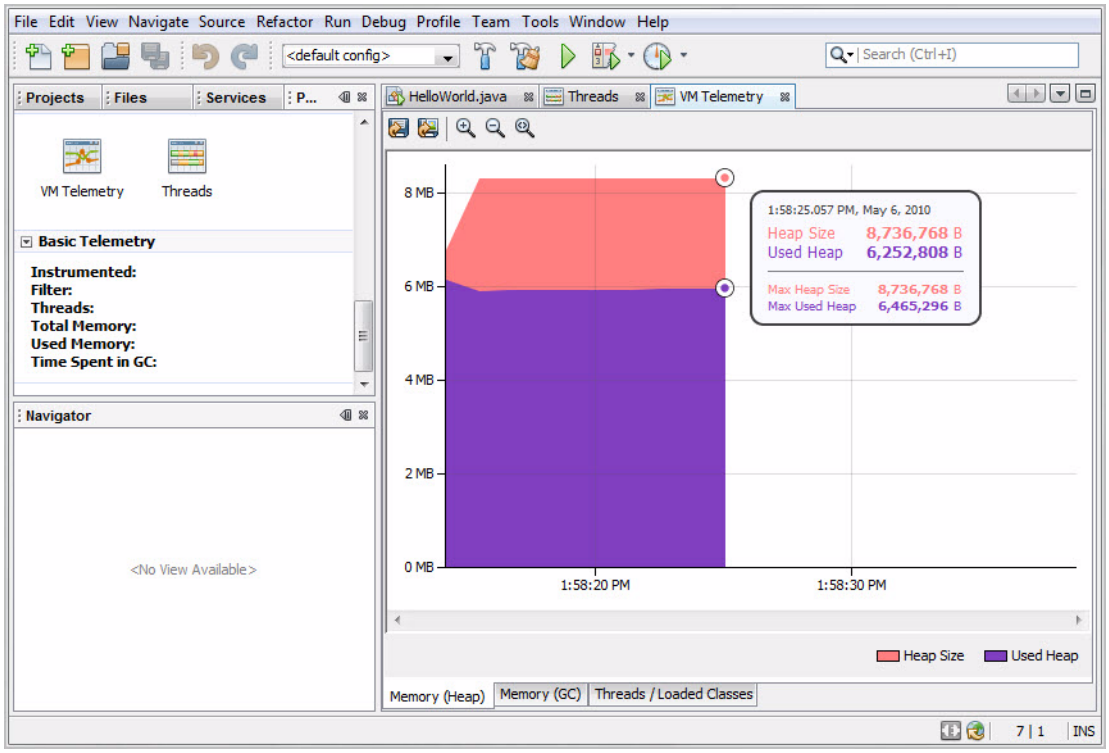
***Figure 4–5 Manual Integration Screen***




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 4–6.

***Figure 4–6 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 4–2.
2. In the NetBeans IDE, choose Profile > Attach Profiler.
3. In the Attach Profile dialog, click Attach.

## 4.3 Simple Local Profiling with `jvmtihprof`

Example 4–3 is a simple profiling example that creates a file of profiling data for a `HelloWorld` application.

***Example 4–3 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 4–1 lists the profiling options.

***Table 4–1 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 |

# 5 Diagnosing Memory Leaks

This chapter describes options for finding and diagnosing application memory leaks.

This chapter includes these topics:



## 5.1 VM Inspector and cvmsh

The VM inspector is a collection of utilities that report virtual machine state, including memory-related state, during execution. To use the VM inspector, you must run an Oracle Java Micro Edition Embedded Client binary that was built with `CVM_INSPECTOR=true`. To verify your build, run `cvm` with this option: `-XbuildOptions`.

`cvmsh` is a command-line front end that sends VM inspector commands to a running `cvm` process. Launch `cvmsh` and `cvm` as illustrated by the following simple example:

```
% cvm -cp testclasses.zip cvmsh
```

For a description of the VM inspector and `cvmsh`, refer to:

`https://weblogs.java.net/blog/2007/07/31/cvms-vm-inspector`.

## 5.2 jvmtihprof and jhat

`jhat` is a web-based Java heap analysis tool, which is described at `http://download.oracle.com/javase/6/docs/technotes/tools/share/jhat.html`. To create the data for `jhat` to display, run a command like this on the target device:

```
% bin/cvm -Xbootclasspath/a:lib/java_crw_demo.jar  \
-agentlib:jvmtihprof=heap=dump,format=b -cp yourApp.jar yourMainMethod
```

Your application must be in the Jar file designated by *yourApp*`.jar`. The command creates a heap dump in `java.hprof.txt`. To display the heap with `jhat`, copy `java.hprof.txt` to your host, then run this command:

```
% jhat java.hprof
```

The `jhat` command starts a web server on port 7000. View the data in a web browser with this URL: `http://localhost:7000`, then drill down into the objects that were dumped.