Oracle® Java Micro Edition Embedded Client

Customization Guide

Release 1.1.1

E23815-02

May 2013

This documentation is for customizers who want to configure or extend the Oracle Java Micro Edition Embedded Client. Topics include changing locales, timezones, and character sets; installing and removing optional components; and tuning performance.

ORACLE

Oracle Java Micro Edition Embedded Client Customization Guide, Release 1.1.1

E23815-02

Contents

List of Tables

List of Figures

Preface

This guide describes how to customize Oracle Java Micro Edition Embedded Client. Examples of customization include removing unneeded components, replacing selected components with your code, and tuning performance.

Audience

This document is intended for engineers who want to customize Oracle Java Micro Edition Embedded Client to prepare it for application development and deployment.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

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 Developer's 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

Footprints and Optional Components

This chapter describes static and dynamic footprints, and how the static footprint is affected by the presence of optional components.

This chapter includes these topics:

- Section 1.1, "Static and Dynamic Memory Footprints"
- Section 1.2, "Installing and Removing Optional Components"

1.1 Static and Dynamic Memory Footprints

The static footprint is the amount of nonvolatile memory required to hold the Oracle Java Micro Edition Embedded Client code. The dynamic memory footprint is the maximum RAM consumed during execution, which is highly dependent on application and device user behavior. Profiling with real applications under real work loads is best way to estimate RAM requirements and observe the effects of memory/performance tradeoffs. The Oracle Java Micro Edition Embedded Client *Developer's Guide* describes profiling.

You can adjust the amount of dynamic memory used by the compiler, as described in Section 4.1.2. You can uninstall optional components to reduce static memory footprint, or add optional components, which has the side effect of increasing the static footprint as described in Section 1.2.

1.2 Installing and Removing Optional Components

The directory `ojec1.1.1/lib` contains optional components that are installed. The virtual machine loads components in this directory when it starts. The directory `ojec1.1.1/extras/lib` contains optional components that are not installed. If you move the files corresponding to a component from one directory to the other, you install or uninstall the component.

The `cvm -version` command lists the installed components.

Table 1–1 lists the optional components, if they are installed by default, their sizes, and the files that must be moved to install or uninstall them.

Note: Do not delete or move any other file in `ojec1.1.1/lib`.

When you install or uninstall a component, be sure to move *all* the corresponding files to or from `ojec1.1.1/lib`.

Table 1–1 Optional Components and Files

Component	Default	Size (KB)	Files
Java Database Connectivity (JDBC)	Installed	20	`jdbc.jar`
Remote Method Invocation (RMI)	Installed	111	`rmi.jar`

XML API (JSR 280)	Installed	179	`jsr280.jar`
			`jsr280_xmlparser.jar`
			`abstractions.jar`
			`abstractions_agent.jar`

Security Optional Package	Installed	977	`ext/sunjce_provider.jar`
			`jaas.jar`
			`jce.jar`
			`jsse_unsigned.jar`
			`jsse-cdc.jar`
			`secop_cmn.jar`
			`sunrsasign.jar`

Additional character sets	Not installed	2,413	`converters.jar`

Instead of installing or uninstalling the entire Security optional package, you can change subcomponents as shown in Table 1–2.

Table 1–2 Removable Security Optional Package Subcomponents

Subcomponent	Size (KB)	Remove from `lib/`
Java Authentication and Authorization Service (JAAS)	91	`jaas.jar`
Java Cryptography Extension (JCE)	109	`ext/sunjce_provider.jar`
Java Secure Socket Extensions (JSSE)	489	`jsse_unsigned.jar` `jsse-cdc.jar`
RSA-signed Jar files	88	`sunrsasign.jar`

2 Globalization

This chapter describes how to customize globalization features: character sets, timesones, and locales.

This chapter includes these topics:

- Section 2.1, "Adding Character Sets"
- Section 2.2, "Timezones"
- Section 2.3, "Adding Locales"

2.1 Adding Character Sets

Oracle Java Micro Edition Embedded Client supports the character sets mandated by the CDC 1.1.2 specification.

You can add character sets in any combination of the following ways:

- Copy `ojec1.1.1/extras/lib/converters.jar` to `ojec1.1.1/lib/`. `converters.jar` contains additional character sets.
- Add any charset that is compatible with the `sun.io.converters` API to the `-bootclasspath` option.

You cannot delete character sets that are mandated by the CDC specification.

2.2 Timezones

Oracle Java Micro Edition Embedded Client supports all Olson time zones. You cannot delete time zones.

2.3 Adding Locales

Oracle Java Micro Edition Embedded Client supports the en_US locale. You can add the locales defined in the Java SE Runtime Environment (JRE) version 1.4.2.

To add the JRE locales, copy *jre*`/lib/ext/localedata.jar` to `ojec1.1.1/lib/ext/`.

3 Security

This chapter describes security features of Oracle Java Micro Edition Embedded Client in comparison to Java Standard Edition (SE).

This chapter includes these topics:

- Section 3.1, "Overview"
- Section 3.2, "Security Procedures"
- Section 3.3, "Security Files"

3.1 Overview

Security is a principal feature of Java technology and an important requirement for mobile and enterprise applications. Oracle Java Micro Edition Embedded Client 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 run time. It is not meant to replace the security documentation available for the Java SE platform, but rather to supplement it and show how Oracle Java Micro Edition Embedded Client and the JAAS, JCE and JSSE security optional packages are related to their counterparts in the Java SE platform.

Table 3–1 describes the security documentation for the Java SE platform.

Table 3–1 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 run-time procedures. See documentation at `http://www.oracle.com/technetwork/java/javaee/gong-135902.html`
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`.

Table 3–1 (Cont.) Security Documentation for the Java SE Platform

Document	Description
Java Tutorial, Security Trail	A tutorial section that describes many of the security procedures for the Java platform. Because these are identical between Oracle Java Micro Edition Embedded Client 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`.

The security framework shared by the Java SE platform and Oracle Java Micro Edition Embedded Client is based on three key components:

- Built-in Security Features
- Security Policy Framework
- Security Provider Architecture

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

3.1.1 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 (*run-time/compile-time/link-time*)
- Bytecode verification (*classloading-time*)
- Safety checks (*run time*)
- Dynamic class loaders (*classloading-time*)

3.1.2 Security Policy Framework

A security policy controls how system resources are accessed by applications at run time. 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 run-time 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 3–1 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 3–1 Java Security Policy Model

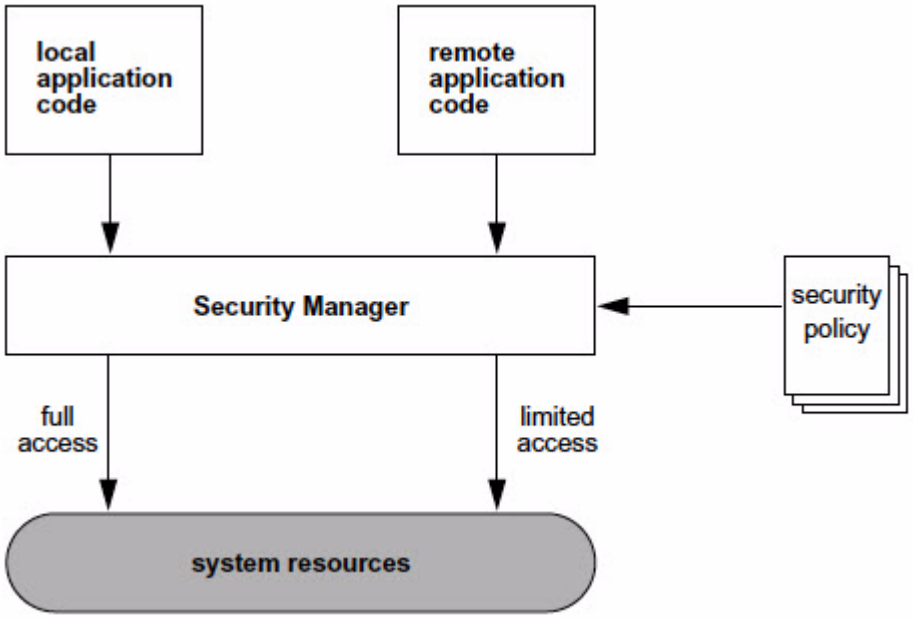

Diagram show Java security policy model with local and remote application code, and the Security Manager

****

3.1.3 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 3–2 describes the default implementations for these security components.

3.1.4 Custom JSSE Provider Plug-ins

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

3.1.5 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.

3.1.6 Self-Integrity Checks

In general, a JCE Provider implementation should include self-integrity checks. For example, Oracle's current JCE provider (called SunJCE Provider) 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.

3.2 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.

3.2.1 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 3–2 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.

3.2.2 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 3–2.

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`.

3.2.3 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 3–2. 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`.

3.2.4 Seed Generation for Random Number Generation

The Oracle Java Micro Edition Embedded Client 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 Oracle Java Micro Edition Embedded Client has a fallback mechanism to read from the `/dev/urandom` device when it determines that there is not enough entropy for `/dev/random` to work promptly.

`/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.

3.3 Security Files

Table 3–2 describes the Oracle Java Micro Edition Embedded Client security files. 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 3–2 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. `lib/ext` is part of the extension class search path, but not part of the system class search path. See the *Architecture Guide* 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.

4 Tuning

This chapter describes how to use runtime options to adjust the performance of the compiler, heap, and class verification to fit your deployment's characteristics and requirements.

This chapter includes the following topics:

4.1 Dynamic Compiler Tuning

This section shows how to use `cvm` command-line options that control the behavior of the Oracle Java Micro Edition Embedded Client 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 run-time 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 Oracle Java Micro Edition Embedded Client 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 the Oracle Java Micro Edition Embedded Client *Architecture Guide*.

4.1.1 Dynamic Compiler Overview

The Oracle Java Micro Edition Embedded Client 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 4–1, this technique is predictable and compact, yet slow.

Figure 4–1 Interpreter-Based Method Execution

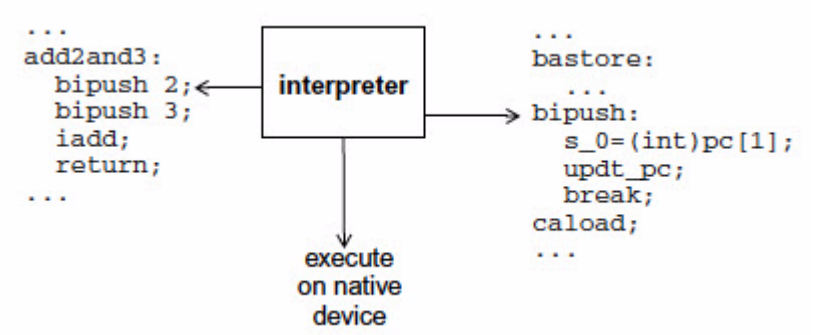

Diagram shows interpreter-based method execution

****

The dynamic compiler is an alternate mechanism that offers significantly faster run-time 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 4–2.

Figure 4–2 Compiling a Method

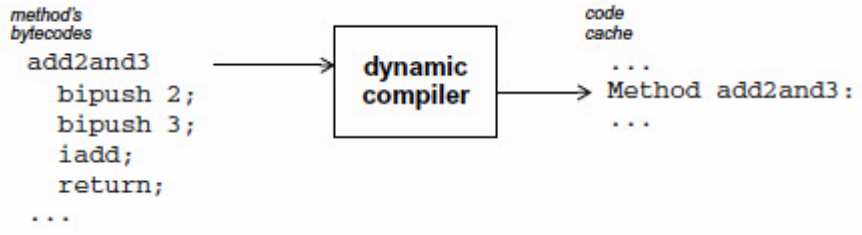

Diagram shows how a method is compiled

****

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

Figure 4–3 Executing a Compiled Method

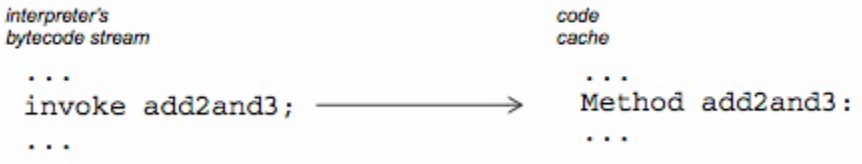

Diagram shows how a compiled method is executed

****

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 Oracle Java Micro Edition Embedded Client Java virtual machine has a run-time 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 run-time statistics kept by `cvm` can be used in different ways to handle various application scenarios. `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 4.1.2.1, "Managing the Popularity Threshold".
- The dynamic compiler has options for specifying code quality based on various forms of inlining. These provide space-time tradeoffs: aggressive inlining provides faster compiled methods, but consume more space in the code cache. An example of the inlining options is provided in Section 4.1.2.2, "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 discards compiled code to obtain memory for a new compilation. A method whose compiled code has been released is interpreted when next invoked. An example of how to manage the code cache is provided in Section 4.1.2.3, "Managing the Code Cache".

4.1.2 Dynamic Compiler Policies

The `cvm` application launcher has a group of command-line options that control how the dynamic compiler behaves. These options form *dynamic compiler policies* that target application or device-specific needs. The most common are space-time tradeoffs. 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.

The cvm Reference appendix in the Architecture Guide lists the dynamic compiler command-line options (`-Xjit`) 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 run-time 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.

4.1.2.1 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 thrashing between compiling and discarding compiled methods.

4.1.2.2 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
  ```

In one experiment, reducing code quality reduced the usage of code cache memory by about 40% while reducing performance by about 5%. The values used were:

- `maxInliningDepth`=3 (default 12)
- `maxInliningCodeLength`=26 (default 64)
- `climit`=60000 (default 20000)

4.1.2.3 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.

Increasing the default code cache size from the default 512KB is beneficial to many applications. On the other hand, when minimizing dynamic memory usage is paramount, you can reduce the code caches size. 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 discard compiled method code more frequently. Therefore, you might also want to use a higher compilation threshold in combination with a lower code cache size.

4.1.3 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. 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.

4.2 Memory Management Tuning

This section provides an overview of how the Java virtual machine manages memory, and the options that you can set to improve performance.

4.2.1 The Java Heap

The virtual machine uses the native platform's memory allocation mechanism to create the *Java heap*, which is where it stores Java objects. The heap is divided into two areas called the young and old generations as shown in Figure 4–4. The names refer to the relative age of objects in the heap. Classifying objects by age is an optimzation technique for garbage collection, which is described in Section 4.2.1.1. Functionally, the heap is a single storage area for objects.

Figure 4–4 Young and Old Java Heap Generations at Startup

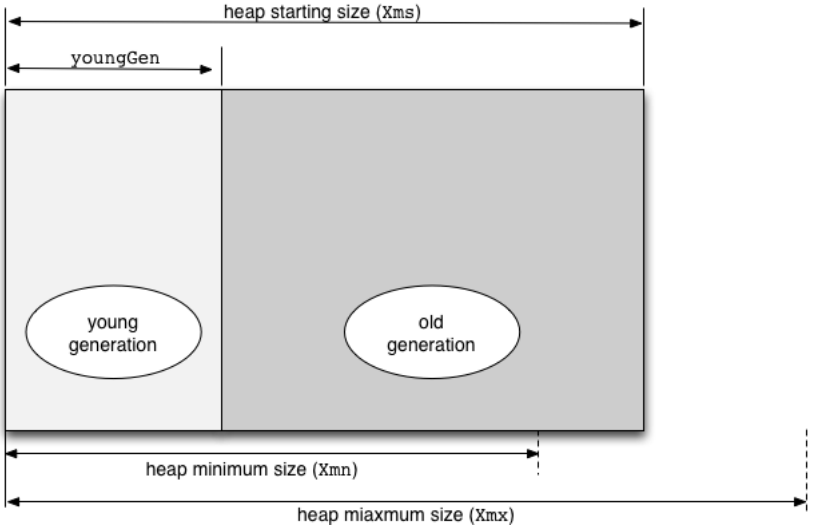

Diagram shows the Java heap divided into two parts, the old generation and the young generation. It also shows the effect of the run-time options for setting the starting, maxmimum, and minimum size of the heap, and the size of the young generation.

****

When you launch the virtual machine with the `cvm` command (described in the *Architecture Guide*), you can specify heap size options that trade memory consumption for performance.

Heap options:

- `-Xms`*size*: The starting size of the heap, which is the space available for object storage, default `2M` (megabytes).
- `-Xgc:youngGen=`*size*: The size of the young generation region within the heap, default 1M.
- `-Xmx`*size*: The maximum size of the heap, default 7M. During execution, as necessary, the virtual machine expands the old generation region to this maximum. The young generation region does not expand. If the heap is at its maximum size and there is insufficient memory for a new object, the result is an `OutOfMemoryError` exception.
- `-Xmn`*size*: The minimum size of the heap, default 1M. During execution, when demand for heap space is low, the virtual machine returns old generation memory to the operating system, down to this minimum size.

Heap usage is highly application-specific. Use profiling (described in the *Developer's Guide*) to monitor heap usage. You can also use the statistics produced by the `cvm` command's `-Xgc:stat` option.

4.2.1.1 Garbage Collection

When a Java application creates an object, the virtual machine allocates memory for the object in the Java heap. After the object is no longer needed, the VM reclaims the object's memory so it can be allocated to new objects. The VM's automatic *garbage collection* (GC) system frees the application 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 run-time

pauses and memory footprint overhead. However, these costs are usually small in comparison to the benefits of application reliability and developer productivity.

Because garbage collection suspends application execution, it is important to set heap-related options so garbage collection does not violate application performance or resource consumption requirements.

4.2.1.1.1 Young Generation Collection The division of the heap into young and old generations of objects is a performance optimization based on the observation that most objects "die young". Young generation garbage collection uses a technique called copy semispace. It is fast and productive and can be run frequently with little noticeable application slowdown. The cost of fast collection is doubling the memory space required to store the young generation (see Figure 4–5).

Figure 4–5 Young Generation From-space and To-space

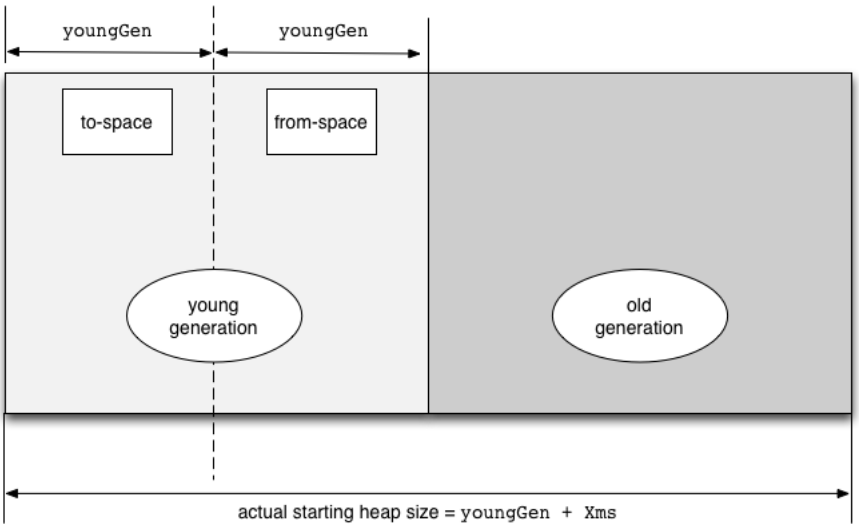

Diagram shows the young generation requiring twice the memory for garbage collection, which increases the starting size of the heap.

****

As shown in Figure 4–5, when the the virtual machine creates the heap, it prepends a region that is equal in size to `youngGen`. The expanded young generation region consists of a from-space and a to-space. During execution, only the from-space is active; new objects are allocated there. The to-space is only used during garbage collection.

When the from-space fills, the young generation garbage collector copies live objects to the to-space, updates references to the live objects, and switches its to-space and from-space pointers, making the old from-space into the to-space. Under typical "most objects die young" conditions, few objects are copied during garbage collection and much of the old from-space is reclaimed. Young objects that survive a small number of garbage collections are assumed to be long-lived, and are promoted to the old generation to prevent further fruitless copying.

For the young generation to work effectively, `youngGen` must be large enough for a substantial number of recently allocated objects to die before the from-space fills. If `youngGen` is too small, the from-space will fill frequently, and the young generation garbage collector will run frequently and reclaim a small percentage of the space. In addition, some objects that will soon die will be promoted to the slower-to-collect old

generation. As a starting point, you can make `youngGen` about 1/8 of `-Xms`*size*, but understanding and profiling your application is the best practice for optimizing heap options.

4.2.1.1.2 Old Generation Collection The old generation does not have the space overhead of a to-space, but its garbage collector runs more slowly and reclaims fewer objects. The trigger for running the old generation collector is a young generation collection that does not reclaim sufficient space. For the best user experience, the old generation collector should run infrequently, which can be realized by making `youngGen` large enough. The old generation collector marks live objects, compacts them starting at the young generation boundary, and updates references to refer to the new object locations.

The virtual machine expands the old generation when, following an old generation garbage collection, the region occupied by old objects is larger than the "high water mark". The high water mark indicates how much heap space was previously used by live objects. Old generation expansion stops at the bound set by `-Xmx`*size*.

4.2.2 Java and Native Stacks

Each Java thread has two stacks, one for Java code and one for native code. The size of native stacks is affected by the following options, whose defaults should be changed only if memory is an extremely scarce resource:

- `-Xopt:stackMinSize`
- `-Xopt:stackMaxSize`
- `-Xopt:stackChunkSize`

The *Architecture Guide* gives the default and permitted values of these options.

By default, the native stack size is set by the target operating system, typically to 1MB. For Linux operating systems, the `limit` command can change the default. A 1MB stack is much larger than most threads need, but neither is it as wasteful as might appear. On most target devices, unused stack space consumes only page table entries. A 64KB native stack size is reasonable for applications whose native code does not recursively enter Java code.

Depending on the target platform's operating system, you can also change the native default stack size with the `-Xss`*size* option. On some operating systems, this option has no effect. The default value of `-Xss` is 0, which means the operating system sets the native stack size.

4.3 Class Verification

By default, Java class verification is performed at class loading time to insure that a class is well-behaved. For large, trusted applications, you can disable Java class verification as follows:

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

Disabling verification reduces both load time and security. Use it with caution.

5

Connecting a Database to JDBC

This chapter describes broadly how to set up a database so it can be used with the Java Database Connectivity optional package (JSR 169) that is included with Oracle Java Micro Edition Embedded Client.

JDBC for CDC (JSR 169) can provide a uniform interface to many databases. The details of configuring a particular database to work with JDBC and Oracle Java Micro Edition Embedded Client vary considerably from database to database and platform to platform. For an overview and example of connecting and using a database with Oracle Java Micro Edition Embedded Client, see the JDBC whitepaper at http://www.oracle.com/technetwork/java/embedded/resources/me-embeddocs/jdbcwhitepaperupdatefinal-1356689.pdf.