Java Platform, Standard Edition Tools Reference

Contents Previous Next

java

Launches a Java application.

Synopsis

java [*options*] *classname* [*args*]

java [*options*] **-jar** *filename* [*args*]

javaw [*options*] *classname* [*args*]

javaw [*options*] **-jar** *filename* [*args*]

options

Command-line options separated by spaces. See Options.

classname

The name of the class to be launched.

filename

The name of the Java Archive (JAR) file to be called. Used only with the `-jar` option.

args

The arguments passed to the `main()` method separated by spaces.

Description

The `java` command starts a Java application. It does this by starting the Java Runtime Environment (JRE), loading the specified class, and calling that class's `main()` method. The method must be declared *public* and *static*, it must not return any value, and it must accept a `String` array as a parameter. The method declaration has the following form:

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

The `java` command can be used to launch a JavaFX application by loading a class that either has a `main()` method or that extends `javafx.application.Application`. In the latter case, the launcher constructs an instance of the `Application` class, calls its `init()` method, and then calls the `start(javafx.stage.Stage)` method.

By default, the first argument that is not an option of the `java` command is the fully qualified name of the class to be called. If the `-jar` option is specified, its argument is the name of the JAR file containing class and resource files for the application. The startup class must be indicated by the `Main-Class` manifest header in its source code.

The JRE searches for the startup class (and other classes used by the application) in three sets of locations: the bootstrap class path, the installed extensions, and the user's class path.

Arguments after the class file name or the JAR file name are passed to the `main()` method.

The `javaw` command is identical to `java`, except that with `javaw` there is no associated console window. Use `javaw` when you do not want a command prompt window to appear. The `javaw` launcher will, however, display a dialog box with error information if a launch fails.

Options

The `java` command supports a wide range of options that can be divided into the following categories:

- Standard Options
- Non-Standard Options
- Advanced Runtime Options
- Advanced JIT Compiler Options
- Advanced Serviceability Options
- Advanced Garbage Collection Options

Standard options are guaranteed to be supported by all implementations of the Java Virtual Machine (JVM). They are used for common actions, such as checking the version of the JRE, setting the class path, enabling verbose output, and so on.

Non-standard options are general purpose options that are specific to the Java HotSpot Virtual Machine, so they are not guaranteed to be supported by all JVM implementations, and are subject to change. These options start with `-X`.

Advanced options are not recommended for casual use. These are developer options used for tuning specific areas of the Java HotSpot Virtual Machine operation that often have specific system requirements and may require privileged access to system configuration parameters. They are also not guaranteed to be supported by all JVM implementations, and are subject to change. Advanced options start with `-XX`.

To keep track of the options that were deprecated or removed in the latest release, there is a section named Deprecated and Removed Options at the end of the document.

Boolean options are used to either enable a feature that is disabled by default or disable a feature that is enabled by default. Such options do not require a parameter. Boolean `-XX` options are enabled using the plus sign (`-XX:+`*OptionName*) and disabled using the minus sign (`-XX:-`*OptionName*).

For options that require an argument, the argument may be separated from the option name by a space, a colon (:), or an equal sign (=), or the argument may directly follow the option (the exact syntax differs for each option). If you are expected to specify the size in bytes, you can use no suffix, or use the suffix `k` or `K` for kilobytes (KB), `m` or `M` for megabytes (MB), `g` or `G` for gigabytes (GB). For example, to set the size to 8 GB, you can specify either `8g`, `8192m`, `8388608k`, or `8589934592` as the argument. If you are expected to specify the percentage, use a number from 0 to 1 (for example, specify `0.25` for 25%).

Standard Options

These are the most commonly used options that are supported by all implementations of the JVM.

-agentlib:*libname*[=*options*]

Loads the specified native agent library. After the library name, a comma-separated list of options specific to the library can be used.

If the option `-agentlib:foo` is specified, then the JVM attempts to load the library named `foo.dll` in the location specified by the `PATH` system variable.

The following example shows how to load the heap profiling tool (HPROF) library and get sample CPU information every 20 ms, with a stack depth of 3:

```
-agentlib:hprof=cpu=samples,interval=20,depth=3
```

The following example shows how to load the Java Debug Wire Protocol (JDWP) library and listen for the socket connection on port 8000, suspending the JVM before the main class loads:

```
-agentlib:jdwp=transport=dt_socket,server=y,address=8000
```

For more information about the native agent libraries, refer to the following:

- The `java.lang.instrument` package description at http://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html

- Agent Command Line Options in the JVM Tools Interface guide at http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html#starting

-agentpath:*pathname*[=*options*]

Loads the native agent library specified by the absolute path name. This option is equivalent to `-agentlib` but uses the full path and file name of the library.

-client

Selects the Java HotSpot Client VM. The 64-bit version of the Java SE Development Kit (JDK) currently ignores this option and instead uses the Server JVM.

For default JVM selection, see Server-Class Machine Detection at
http://docs.oracle.com/javase/8/docs/technotes/guides/vm/server-class.html

-D*property*=*value*

Sets a system property value. The *property* variable is a string with no spaces that represents the name of the property. The *value* variable is a string that represents the value of the property. If *value* is a string with spaces, then enclose it in quotation marks (for example `-Dfoo="foo bar"`).

-disableassertions[:[*packagename*]...|:*classname*]

-da[:[*packagename*]...|:*classname*]

Disables assertions. By default, assertions are disabled in all packages and classes.

With no arguments, `-disableassertions` (`-da`) disables assertions in all packages and classes. With the *packagename* argument ending in `...`, the switch disables assertions in the specified package and any subpackages. If the argument is simply `...`, then the switch disables assertions in the unnamed package in the current working directory. With the *classname* argument, the switch disables assertions in the specified class.

The `-disableassertions` (`-da`) option applies to all class loaders and to system classes (which do not have a class loader). There is one exception to this rule: if the option is provided with no arguments, then it does not apply to system classes. This makes it easy to disable assertions in all classes except for system classes. The `-disablesystemassertions` option enables you to disable assertions in all system classes.

To explicitly enable assertions in specific packages or classes, use the `-enableassertions` (`-ea`) option. Both options can be used at the same time. For example, to run the `MyClass` application with assertions enabled in package `com.wombat.fruitbat` (and any subpackages) but disabled in class `com.wombat.fruitbat.Brickbat`, use the following command:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyClass
```

-disablesystemassertions

-dsa

Disables assertions in all system classes.

-enableassertions[:[*packagename*]...|:*classname*]

-ea[:[*packagename*]...|:*classname*]

Enables assertions. By default, assertions are disabled in all packages and classes.

With no arguments, `-enableassertions` (`-ea`) enables assertions in all packages and classes. With the *packagename* argument ending in `...`, the switch enables assertions in the specified package and any subpackages. If the argument is simply `...`, then the switch enables assertions in the unnamed package in the current working directory. With the *classname* argument, the switch enables assertions in the specified class.

The `-enableassertions` (`-ea`) option applies to all class loaders and to system classes (which do not have a class loader). There is one exception to this rule: if the option is provided with no arguments, then it does not apply to system classes. This makes it easy to enable assertions in all classes except for system classes. The `-enablesystemassertions` option provides a separate switch to enable assertions in all system classes.

To explicitly disable assertions in specific packages or classes, use the `-disableassertions` (`-da`) option. If a single command contains multiple instances of these switches, then they are processed in order before loading any classes. For example, to run the `MyClass` application with assertions enabled only in package `com.wombat.fruitbat` (and any subpackages) but disabled in class `com.wombat.fruitbat.Brickbat`, use the following command:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyClass
```

-enablesystemassertions

-esa

Enables assertions in all system classes.

-help

-?

Displays usage information for the `java` command without actually running the JVM.

-jar *filename*

Executes a program encapsulated in a JAR file. The *filename* argument is the name of a JAR file with a manifest that contains a line in the form `Main-Class:classname` that defines the class with the `public static void main(String[] args)` method that serves as your application's starting point.

When you use the `-jar` option, the specified JAR file is the source of all user classes, and other class path settings are ignored.

For more information about JAR files, see the following resources:

- `jar`(1)

- The Java Archive (JAR) Files guide at `http://docs.oracle.com/javase/8/docs/technotes/guides/jar/index.html`
- Lesson: Packaging Programs in JAR Files at `http://docs.oracle.com/javase/tutorial/deployment/jar/index.html`

-javaagent:*jarpath*[=*options*]

Loads the specified Java programming language agent. For more information about instrumenting Java applications, see the `java.lang.instrument` package description in the Java API documentation at `http://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html`

-jre-restrict-search

Includes user-private JREs in the version search.

-no-jre-restrict-search

Excludes user-private JREs from the version search.

-server

Selects the Java HotSpot Server VM. The 64-bit version of the JDK supports only the Server VM, so in that case the option is implicit.

For default JVM selection, see Server-Class Machine Detection at `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/server-class.html`

-showversion

Displays version information and continues execution of the application. This option is equivalent to the `-version` option except that the latter instructs the JVM to exit after displaying version information.

-splash:*imgname*

Shows the splash screen with the image specified by *imgname*. For example, to show the `splash.gif` file from the `images` directory when starting your application, use the following option:

```
-splash:images/splash.gif
```

-verbose:class

Displays information about each loaded class.

-verbose:gc

Displays information about each garbage collection (GC) event.

-verbose:jni

Displays information about the use of native methods and other Java Native Interface (JNI) activity.

-version

Displays version information and then exits. This option is equivalent to the `-showversion` option except that the latter does not instruct the JVM to exit after displaying version information.

-version:*release*

Specifies the release version to be used for running the application. If the version of the `java` command called does not meet this specification and an appropriate implementation is found on the system, then the appropriate implementation will be used.

The *release* argument specifies either the exact version string, or a list of version strings and ranges separated by spaces. A *version string* is the developer designation of the version number in the following form: `1.`*x*`.0_`*u* (where *x* is the major version number, and *u* is the update version number). A *version range* is made up of a version string followed by a plus sign (`+`) to designate this version or later, or a part of a version string followed by an asterisk (`*`) to designate any version string with a matching prefix. Version strings and ranges can be combined using a space for a logical *OR* combination, or an ampersand (`&`) for a logical *AND* combination of two version strings/ranges. For example, if running the class or JAR file requires either JRE 6u13 (1.6.0_13), or any JRE 6 starting from 6u10 (1.6.0_10), specify the following:

```
-version:"1.6.0_13 1.6* & 1.6.0_10+"
```

Quotation marks are necessary only if there are spaces in the *release* parameter.

For JAR files, the preference is to specify version requirements in the JAR file manifest rather than on the command line.

Non-Standard Options

These options are general purpose options that are specific to the Java HotSpot Virtual Machine.

-X

Displays help for all available `-X` options.

-Xbatch

Disables background compilation. By default, the JVM compiles the method as a background task, running the method in interpreter mode until the background compilation is finished. The `-Xbatch` flag disables background compilation so that compilation of all methods proceeds as a foreground task until completed.

This option is equivalent to `-XX:-BackgroundCompilation`.

-Xbootclasspath:*path*

Specifies a list of directories, JAR files, and ZIP archives separated by semicolons (;) to search for boot class files. These are used in place of the boot class files included in the JDK.

Do not deploy applications that use this option to override a class in `rt.jar`, because this violates the JRE binary code license.

-Xbootclasspath/a:*path*

Specifies a list of directories, JAR files, and ZIP archives separated by semicolons (;) to append to the end of the default bootstrap class path.

Do not deploy applications that use this option to override a class in `rt.jar`, because this violates the JRE binary code license.

-Xbootclasspath/p:*path*

Specifies a list of directories, JAR files, and ZIP archives separated by semicolons (;) to prepend to the front of the default bootstrap class path.

Do not deploy applications that use this option to override a class in `rt.jar`, because this violates the JRE binary code license.

-Xcheck:jni

Performs additional checks for Java Native Interface (JNI) functions. Specifically, it validates the parameters passed to the JNI function and the runtime environment data before processing the JNI request. Any invalid data encountered indicates a problem in the native code, and the JVM will terminate with an irrecoverable error in such cases. Expect a performance degradation when this option is used.

-Xcomp

Forces compilation of methods on first invocation. By default, the Client VM (`-client`) performs 1,000 interpreted method invocations and the Server VM (`-server`) performs 10,000 interpreted method invocations to gather information for efficient compilation. Specifying the `-Xcomp` option disables interpreted method invocations to increase compilation performance at the expense of efficiency.

You can also change the number of interpreted method invocations before compilation using the `-XX:CompileThreshold` option.

-Xdebug

Does nothing. Provided for backward compatibility.

-Xdiag

Shows additional diagnostic messages.

-Xfuture

Enables strict class-file format checks that enforce close conformance to the class-file format specification. Developers are encouraged to use this flag when developing new code because the stricter checks will become the default in future releases.

-Xint

Runs the application in interpreted-only mode. Compilation to native code is disabled, and all bytecode is executed by the interpreter. The performance benefits offered by the just in time (JIT) compiler are not present in this mode.

-Xinternalversion

Displays more detailed JVM version information than the `-version` option, and then exits.

-Xloggc:*filename*

Sets the file to which verbose GC events information should be redirected for logging. The information written to this file is similar to the output of `-verbose:gc` with the time elapsed since the first GC event preceding each logged event. The `-Xloggc` option overrides `-verbose:gc` if both are given with the same `java` command.

Example:

```
-Xloggc:garbage-collection.log
```

-Xmaxjitcodesize=*size*

Specifies the maximum code cache size (in bytes) for JIT-compiled code. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default maximum code cache size is 240 MB; if you disable tiered compilation with the option `-XX:-TieredCompilation`, then the default size is 48 MB:

```
-Xmaxjitcodesize=240m
```

This option is equivalent to `-XX:ReservedCodeCacheSize`.

-Xmixed

Executes all bytecode by the interpreter except for hot methods, which are compiled to native code.

-Xmn*size*

Sets the initial and maximum size (in bytes) of the heap for the young generation (nursery). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes.

The young generation region of the heap is used for new objects. GC is performed in this region more often than in other regions. If the size for the young generation is too small, then a lot of minor garbage collections will be performed. If the size is too large, then only full garbage collections will be performed, which can take a long time to complete. Oracle recommends that you keep the size for the young generation between a half and a quarter of the overall heap size.

The following examples show how to set the initial and maximum size of young generation to 256 MB using various units:

```
-Xmn256m
-Xmn262144k
-Xmn268435456
```

Instead of the `-Xmn` option to set both the initial and maximum size of the heap for the young generation, you can use `-XX:NewSize` to set the initial size and `-XX:MaxNewSize` to set the maximum size.

-Xms*size*

Sets the initial size (in bytes) of the heap. This value must be a multiple of 1024 and greater than 1 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes.

The following examples show how to set the size of allocated memory to 6 MB using various units:

```
-Xms6291456
-Xms6144k
-Xms6m
```

If you do not set this option, then the initial size will be set as the sum of the sizes allocated for the old generation and the young generation. The initial size of the heap for the young generation can be set using the `-Xmn` option or the `-XX:NewSize` option.

-Xmx*size*

Specifies the maximum size (in bytes) of the memory allocation pool in bytes. This value must be a multiple of 1024 and greater than 2 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default value is chosen at runtime based on system configuration. For server deployments, `-Xms` and `-Xmx` are often set to the same value. See the section "Ergonomics" in *Java SE HotSpot Virtual Machine Garbage Collection Tuning Guide* at `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/index.html`.

The following examples show how to set the maximum allowed size of allocated memory to 80 MB using various units:

```
-Xmx83886080
-Xmx81920k
-Xmx80m
```

The `-Xmx` option is equivalent to `-XX:MaxHeapSize`.

-Xnoclassgc

Disables garbage collection (GC) of classes. This can save some GC time, which shortens interruptions during the application run.

When you specify `-Xnoclassgc` at startup, the class objects in the application will be left untouched during GC and will always be considered live. This can result in more memory being permanently occupied which, if not used carefully, will throw an out of memory exception.

-Xprof

Profiles the running program and sends profiling data to standard output. This option is provided as a utility that is useful in program development and is not intended to be used in production systems.

-Xrs

Reduces the use of operating system signals by the JVM.

Shutdown hooks enable orderly shutdown of a Java application by running user cleanup code (such as closing database connections) at shutdown, even if the JVM terminates abruptly.

The JVM watches for console control events to implement shutdown hooks for unexpected termination. Specifically, the JVM registers a console control handler that begins shutdown-hook processing and returns `TRUE` for `CTRL_C_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, and `CTRL_SHUTDOWN_EVENT`.

The JVM uses a similar mechanism to implement the feature of dumping thread stacks for debugging purposes. The JVM uses `CTRL_BREAK_EVENT` to perform thread dumps.

If the JVM is run as a service (for example, as a servlet engine for a web server), then it can receive `CTRL_LOGOFF_EVENT` but should not initiate shutdown because the operating system will not actually terminate the process. To avoid possible interference such as this, the `-Xrs` option can be used. When the `-Xrs` option is used, the JVM does not install a console control handler, implying that it does not watch for or process `CTRL_C_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, or `CTRL_SHUTDOWN_EVENT`.

There are two consequences of specifying `-Xrs`:

- Ctrl + Break thread dumps are not available.
- User code is responsible for causing shutdown hooks to run, for example, by calling `System.exit()` when the JVM is to be terminated.

-Xshare:*mode*

Sets the class data sharing (CDS) mode. Possible *mode* arguments for this option include the following:

auto

Use CDS if possible. This is the default value for Java HotSpot 32-Bit Client VM.

on

Require the use of CDS. Print an error message and exit if class data sharing cannot be used.

off

Do not use CDS. This is the default value for Java HotSpot 32-Bit Server VM, Java HotSpot 64-Bit Client VM, and Java HotSpot 64-Bit Server VM.

dump

Manually generate the CDS archive. Specify the application class path as described in "Setting the Class Path".

You should regenerate the CDS archive with each new JDK release.

-XshowSettings:*category*

Shows settings and continues. Possible *category* arguments for this option include the following:

all

Shows all categories of settings. This is the default value.

locale

Shows settings related to locale.

properties

Shows settings related to system properties.

vm

Shows the settings of the JVM.

-Xss*size*

Sets the thread stack size (in bytes). Append the letter `k` or `K` to indicate KB, `m` or `M` to indicate MB, `g` or `G` to indicate GB. The default value depends on virtual memory.

The following examples set the thread stack size to 1024 KB in different units:

```
-Xss1m
-Xss1024k
-Xss1048576
```

This option is equivalent to `-XX:ThreadStackSize`.

-Xverify:*mode*

Sets the mode of the bytecode verifier. Bytecode verification ensures that class files are properly formed and satisfy the constraints listed in section 4.10, Verification of `class` Files in the *The Java Virtual Machine Specification*.

Do not turn off verification as this reduces the protection provided by Java and could cause problems due to ill-formed class files.

Possible *mode* arguments for this option include the following:

remote

Verifies all bytecodes not loaded by the bootstrap class loader. This is the default behavior if you do not specify the `-Xverify` option.

all

Enables verification of all bytecodes.

none

Disables verification of all bytecodes. Use of `-Xverify:none` is unsupported.

Advanced Runtime Options

These options control the runtime behavior of the Java HotSpot VM.

-XX:+CheckEndorsedAndExtDirs

Enables the option to prevent the `java` command from running a Java application if it uses the endorsed-standards override mechanism or the extension mechanism. This option checks if an application is using one of these mechanisms by checking the following:

- The `java.ext.dirs` or `java.endorsed.dirs` system property is set.
- The `lib/endorsed` directory exists and is not empty.
- The `lib/ext` directory contains any JAR files other than those of the JDK.
- The system-wide platform-specific extension directory contains any JAR files.

-XX:+DisableAttachMechanism

Enables the option that disables the mechanism that lets tools attach to the JVM. By default, this option is disabled, meaning that the attach mechanism is enabled and you can use tools such as `jcmd`, `jstack`, `jmap`, and `jinfo`.

-XX:ErrorFile=*filename*

Specifies the path and file name to which error data is written when an irrecoverable error occurs. By default, this file is created in the current working directory and named `hs_err_pid`*pid*`.log` where *pid* is the identifier of the process that caused the error. The following example shows how to set the default log file (note that the identifier of the process is specified as `%p`):

```
-XX:ErrorFile=./hs_err_pid%p.log
```

The following example shows how to set the error log file to `C:/log/java/java_error.log`:

```
-XX:ErrorFile=C:/log/java/java_error.log
```

If the file cannot be created in the specified directory (due to insufficient space, permission problem, or another issue), then the file is created in the temporary directory for the operating system. The temporary directory is specified by the value of the `TMP` environment variable; if that environment variable is not defined, then the value of the `TEMP` environment variable is used.

-XX:+FailOverToOldVerifier

Enables automatic failover to the old verifier when the new type checker fails. By default, this option is disabled and it is ignored (that is, treated as disabled) for classes with a recent bytecode version. You can enable it for classes with older versions of the bytecode.

-XX:+FlightRecorder

Enables the use of the Java Flight Recorder (JFR) during the runtime of the application. This is a commercial feature that works in conjunction with the `-XX:+UnlockCommercialFeatures` option as follows:

```
java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

If this option is not provided, Java Flight Recorder can still be enabled in a running JVM by providing the appropriate `jcmd` diagnostic commands.

-XX:-FlightRecorder

Disables the use of the Java Flight Recorder (JFR) during the runtime of the application. This is a commercial feature that works in conjunction with the `-XX:+UnlockCommercialFeatures` option as follows:

```
java -XX:+UnlockCommercialFeatures -XX:-FlightRecorder
```

If this option is provided, Java Flight Recorder cannot be enabled in a running JVM.

-XX:FlightRecorderOptions=*parameter*=*value*

Sets the parameters that control the behavior of JFR. This is a commercial feature that works in conjunction with the `-XX:+UnlockCommercialFeatures` option. This option can be used only when JFR is enabled (that is, the `-XX:+FlightRecorder` option is specified).

The following list contains all available JFR parameters:

defaultrecording={true|false}

Specifies whether the recording is a continuous background recording or if it runs for a limited time. By default, this parameter is set to `false` (recording runs for a limited time). To make the recording run continuously, set the parameter to `true`.

disk={true|false}

Specifies whether JFR should write a continuous recording to disk. By default, this parameter is set to `false` (continuous recording to disk is disabled). To enable it, set the parameter to `true`, and also set `defaultrecording=true`.

dumponexit={true|false}

Specifies whether a dump file of JFR data should be generated when the JVM terminates in a controlled manner. By default, this parameter is set to `false` (dump file on exit is not generated). To enable it, set the parameter to `true`, and also set `defaultrecording=true`.

The dump file is written to the location defined by the `dumponexitpath` parameter.

dumponexitpath=*path*

Specifies the path and name of the dump file with JFR data that is created when the JVM exits in a controlled manner if you set the `dumponexit=true` parameter. Setting the path makes sense only if you also set `defaultrecording=true`.

If the specified path is a directory, the JVM assigns a file name that shows the creation date and time. If the specified path includes a file name and if that file already exists, the JVM creates a new file by appending the date and time stamp to the specified file name.

globalbuffersize=*size*

Specifies the total amount of primary memory (in bytes) used for data retention. Append `k` or `K`, to specify the size in KB, `m` or `M` to specify the size in MB, `g` or `G` to specify the size in GB. By default, the size is set to 462848 bytes.

loglevel={quiet|error|warning|info|debug|trace}

Specify the amount of data written to the log file by JFR. By default, it is set to `info`.

maxage=*time*

Specifies the maximum age of disk data to keep for the default recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days (for example, specifying `30s` means 30 seconds). By default, the maximum age is set to 15 minutes (`15m`).

This parameter is valid only if you set the `disk=true` parameter.

maxchunksize=*size*

Specifies the maximum size (in bytes) of the data chunks in a recording. Append `k` or `K`, to specify the size in KB, `m` or `M` to specify the size in MB, `g` or `G` to specify the size in GB. By default, the maximum size of data chunks is set to 12 MB.

maxsize=*size*

Specifies the maximum size (in bytes) of disk data to keep for the default recording. Append `k` or `K`, to specify the size in KB, `m` or `M` to specify the size in MB, `g` or `G` to specify the size in GB. By default, the maximum size of disk data is not limited, and this parameter is set to 0.

This parameter is valid only if you set the `disk=true` parameter.

repository=*path*

Specifies the repository (a directory) for temporary disk storage. By default, the system's temporary directory is used.

samplethreads={true|false}

Specifies whether thread sampling is enabled. Thread sampling occurs only if the sampling event is enabled along with this parameter. By default, this parameter is enabled.

settings=*path*

Specifies the path and name of the event settings file (of type JFC). By default, the `default.jfc` file is used, which is located in `JAVA_HOME/jre/lib/jfr`.

stackdepth=*depth*

Stack depth for stack traces by JFR. By default, the depth is set to 64 method calls. The maximum is 2048, minimum is 1.

threadbuffersize=*size*

Specifies the per-thread local buffer size (in bytes). Append `k` or `K`, to specify the size in KB, `m` or `M` to specify the size in MB, `g` or `G` to specify the size in GB. Higher values for this parameter allow more data gathering without contention to flush it to the global storage. It can increase application footprint in a thread-rich environment. By default, the local buffer size is set to 5 KB.

You can specify values for multiple parameters by separating them with a comma. For example, to instruct JFR to write a continuous recording to disk, and set the maximum size of data chunks to 10 MB, specify the following:

```
-XX:FlightRecorderOptions=defaultrecording=true,disk=true,maxchunksize=10M
```

-XX:LargePageSizeInBytes=*size*

On Solaris, sets the maximum size (in bytes) for large pages used for Java heap. The *size* argument must be a power of 2 (2, 4, 8, 16, ...). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. By default, the size is set to 0, meaning that the JVM chooses the size for large pages automatically.

The following example illustrates how to set the large page size to 4 megabytes (MB):

```
-XX:LargePageSizeInBytes=4m
```

-XX:MaxDirectMemorySize=*size*

Sets the maximum total size (in bytes) of the New I/O (the `java.nio` package) direct-buffer allocations. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. By default, the size is set to 0, meaning that the JVM chooses the size for NIO direct-buffer allocations automatically.

The following examples illustrate how to set the NIO size to 1024 KB in different units:

```
-XX:MaxDirectMemorySize=1m
-XX:MaxDirectMemorySize=1024k
-XX:MaxDirectMemorySize=1048576
```

-XX:NativeMemoryTracking=*mode*

Specifies the mode for tracking JVM native memory usage. Possible *mode* arguments for this option include the following:

off

Do not track JVM native memory usage. This is the default behavior if you do not specify the `-XX:NativeMemoryTracking` option.

summary

Only track memory usage by JVM subsystems, such as Java heap, class, code, and thread.

detail

In addition to tracking memory usage by JVM subsystems, track memory usage by individual `CallSite`, individual virtual memory region and its committed regions.

-XX:ObjectAlignmentInBytes=*alignment*

Sets the memory alignment of Java objects (in bytes). By default, the value is set to 8 bytes. The specified value should be a power of two, and must be within the range of 8 and 256 (inclusive). This option makes it possible to use compressed pointers with large Java heap sizes.

The heap size limit in bytes is calculated as:

```
4GB * ObjectAlignmentInBytes
```

Note: As the alignment value increases, the unused space between objects will also increase. As a result, you may not realize any benefits from using compressed pointers with large Java heap sizes.

-XX:OnError=*string*

Sets a custom command or a series of semicolon-separated commands to run when an irrecoverable error occurs. If the string contains spaces, then it must be enclosed in quotation marks.

The following example shows how the `-XX:OnError` option can be used to run the `userdump.exe` utility to obtain a crash dump in case of an irrecoverable error (the `%p` designates the current process):

```
-XX:OnError="userdump.exe %p"
```

The preceding example assumes that the path to the `userdump.exe` utility is specified in the `PATH` environment variable.

-XX:OnOutOfMemoryError=*string*

Sets a custom command or a series of semicolon-separated commands to run when an `OutOfMemoryError` exception is first thrown. If the string contains spaces, then it must be enclosed in quotation marks. For an example of a command string, see the description of the `-XX:OnError` option.

-XX:+PerfDataSaveToFile

If enabled, saves `jstat`(1) binary data when the Java application exits. This binary data is saved in a file named `hsperfdata_<pid>`, where `<pid>` is the process identifier of the Java application you ran. Use `jstat` to display the performance data contained in this file as follows:

```
jstat -class file:///<path>/hsperfdata_<pid>
jstat -gc file:///<path>/hsperfdata_<pid>
```

-XX:+PrintCommandLineFlags

Enables printing of ergonomically selected JVM flags that appeared on the command line. It can be useful to know the ergonomic values set by the JVM, such as the heap space size and the selected garbage collector. By default, this option is disabled and flags are not printed.

-XX:+PrintNMTStatistics

Enables printing of collected native memory tracking data at JVM exit when native memory tracking is enabled (see `-XX:NativeMemoryTracking`). By default, this option is disabled and native memory tracking data is not printed.

-XX:+RelaxAccessControlCheck

Decreases the amount of access control checks in the verifier. By default, this option is disabled, and it is ignored (that is, treated as disabled) for classes with a recent bytecode version. You can enable it for classes with older versions of the bytecode.

-XX:+ResourceManagement

Enables the use of Resource Management during the runtime of the application.

This is a commercial feature that requires you to also specify the `-XX:+UnlockCommercialFeatures` option as follows:

```
java -XX:+UnlockCommercialFeatures -XX:+ResourceManagement
```

-XX:ResourceManagementSampleInterval=*value* (milliseconds)

Sets the parameter that controls the sampling interval for Resource Management measurements, in milliseconds.

This option can be used only when Resource Management is enabled (that is, the `-XX:+ResourceManagement` option is specified).

-XX:SharedArchiveFile=*path*

Specifies the path and name of the class data sharing (CDS) archive file

-XX:SharedClassListFile=*file_name*

Specifies the text file that contains the names of the class files to store in the class data sharing (CDS) archive. This file contains the full name of one class file per line, except slashes (`/`) replace dots (`.`). For example, to specify the classes `java.lang.Object` and `hello.Main`, create a text file that contains the following two lines:

```
java/lang/Object
hello/Main
```

The class files that you specify in this text file should include the classes that are commonly used by the application. They may include any classes from the application, extension, or bootstrap class paths.

-XX:+ShowMessageBoxOnError

Enables displaying of a dialog box when the JVM experiences an irrecoverable error. This prevents the JVM from exiting and keeps the process active so that you can attach a debugger to it to investigate the cause of the error. By default, this option is disabled.

-XX:StartFlightRecording=*parameter*=*value*

Starts a JFR recording for the Java application. This is a commercial feature that works in conjunction with the `-XX:+UnlockCommercialFeatures` option. This option is equivalent to the `JFR.start` diagnostic command that starts a recording during runtime. You can set the following parameters when starting a JFR recording:

compress={true|false}

Specifies whether to compress the JFR recording log file (of type JFR) on the disk using the `gzip` file compression utility. This parameter is valid only if the `filename` parameter is specified. By default it is set to `false` (recording is not compressed). To enable compression, set the parameter to `true`.

defaultrecording={true|false}

Specifies whether the recording is a continuous background recording or if it runs for a limited time. By default, this parameter is set to `false` (recording runs for a limited time). To make the recording run continuously, set the parameter to `true`.

delay=*time*

Specifies the delay between the Java application launch time and the start of the recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days (for example, specifying `10m` means 10 minutes). By default, there is no delay, and this parameter is set to 0.

dumponexit={true|false}

Specifies whether a dump file of JFR data should be generated when the JVM terminates in a controlled manner. By default, this parameter is set to `false` (dump file on exit is not generated). To enable it, set the parameter to `true`.

The dump file is written to the location defined by the `filename` parameter.

Example:

```
-XX:StartFlightRecording=name=test,filename=D:\test.jfr,dumponexit=true
```

duration=*time*

Specifies the duration of the recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days (for example, specifying `5h` means 5 hours). By default, the duration is not limited, and this parameter is set to 0.

filename=*path*

Specifies the path and name of the JFR recording log file.

name=*identifier*

Specifies the identifier for the JFR recording. By default, it is set to `Recording x`.

maxage=*time*

Specifies the maximum age of disk data to keep for the default recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days (for example, specifying `30s` means 30 seconds). By default, the maximum age is set to 15 minutes (`15m`).

maxsize=*size*

Specifies the maximum size (in bytes) of disk data to keep for the default recording. Append `k` or `K`, to specify the size in KB, `m` or `M` to specify the size in MB, `g` or `G` to specify the size in GB. By default, the maximum size of disk data is not limited, and this parameter is set to 0.

settings=*path*

Specifies the path and name of the event settings file (of type JFC). By default, the `default.jfc` file is used, which is located in `JAVA_HOME/jre/lib/jfr`.

You can specify values for multiple parameters by separating them with a comma. For example, to save the recording to test.jfr in the current working directory, and instruct JFR to compress the log file, specify the following:

```
-XX:StartFlightRecording=filename=test.jfr,compress=true
```

-XX:ThreadStackSize=*size*

Sets the thread stack size (in bytes). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default value depends on virtual memory.

The following examples show how to set the thread stack size to 1024 KB in different units:

```
-XX:ThreadStackSize=1m
-XX:ThreadStackSize=1024k
-XX:ThreadStackSize=1048576
```

This option is equivalent to `-Xss`.

-XX:+TraceClassLoading

Enables tracing of classes as they are loaded. By default, this option is disabled and classes are not traced.

-XX:+TraceClassLoadingPreorder

Enables tracing of all loaded classes in the order in which they are referenced. By default, this option is disabled and classes are not traced.

-XX:+TraceClassResolution

Enables tracing of constant pool resolutions. By default, this option is disabled and constant pool resolutions are not traced.

-XX:+TraceClassUnloading

Enables tracing of classes as they are unloaded. By default, this option is disabled and classes are not traced.

-XX:+TraceLoaderConstraints

Enables tracing of the loader constraints recording. By default, this option is disabled and loader constraints recording is not traced.

-XX:+UnlockCommercialFeatures

Enables the use of commercial features. Commercial features are included with Oracle Java SE Advanced or Oracle Java SE Suite packages, as defined on the *Java SE Products* page at `http://www.oracle.com/technetwork/java/javase/terms/products/index.html`

By default, this option is disabled and the JVM runs without the commercial features. Once they were enabled for a JVM process, it is not possible to disable their use for that process.

If this option is not provided, commercial features can still be unlocked in a running JVM by using the appropriate `jcmd` diagnostic commands.

-XX:+UseAppCDS

Enables application class data sharing (AppCDS). To use AppCDS, you must also specify values for the options `-XX:SharedClassListFile` and `-XX:SharedArchiveFile` during both CDS dump time (see the option `-Xshare:dump`) and application run time.

This is a commercial feature that requires you to also specify the `-XX:+UnlockCommercialFeatures` option. This is also an experimental feature; it may change in future releases.

See .

-XX:-UseBiasedLocking

Disables the use of biased locking. Some applications with significant amounts of uncontended synchronization may attain significant speedups with this flag enabled, whereas applications with certain patterns of locking may see slowdowns. For more information about the biased locking technique, see the example in Java Tuning White Paper at `http://www.oracle.com/technetwork/java/tuning-139912.html#section4.2.5`

By default, this option is enabled.

-XX:-UseCompressedOops

Disables the use of compressed pointers. By default, this option is enabled, and compressed pointers are used when Java heap sizes are less than 32 GB. When this option is enabled, object references are represented as 32-bit offsets instead of 64-bit pointers, which typically increases performance when running the application with Java heap sizes less than 32 GB. This option works only for 64-bit JVMs.

It is also possible to use compressed pointers when Java heap sizes are greater than 32GB. See the `-XX:ObjectAlignmentInBytes` option.

-XX:+UseLargePages

Enables the use of large page memory. By default, this option is disabled and large page memory is not used.

For more information, see "Large Pages".

-XX:+UseMembar

Enables issuing of membars on thread state transitions. This option is disabled by default on all platforms except ARM servers, where it is enabled. (It is recommended that you do not disable this option on ARM servers.)

-XX:+UsePerfData

Enables the `perfdata` feature. This option is enabled by default to allow JVM monitoring and performance testing. Disabling it suppresses the creation of the `hsperfdata_userid` directories. To disable the `perfdata` feature, specify `-XX:-UsePerfData`.

-XX:+AllowUserSignalHandlers

Enables installation of signal handlers by the application. By default, this option is disabled and the application is not allowed to install signal handlers.

Advanced JIT Compiler Options

These options control the dynamic just-in-time (JIT) compilation performed by the Java HotSpot VM.

-XX:+AggressiveOpts

Enables the use of aggressive performance optimization features, which are expected to become default in upcoming releases. By default, this option is disabled and experimental performance features are not used.

-XX:AllocateInstancePrefetchLines=*lines*

Sets the number of lines to prefetch ahead of the instance allocation pointer. By default, the number of lines to prefetch is set to 1:

```
-XX:AllocateInstancePrefetchLines=1
```

Only the Java HotSpot Server VM supports this option.

-XX:AllocatePrefetchDistance=*size*

Sets the size (in bytes) of the prefetch distance for object allocation. Memory about to be written with the value of new objects is prefetched up to this distance starting from the address of the last allocated object. Each Java thread has its own allocation point.

Negative values denote that prefetch distance is chosen based on the platform. Positive values are bytes to prefetch. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default value is set to -1.

The following example shows how to set the prefetch distance to 1024 bytes:

```
-XX:AllocatePrefetchDistance=1024
```

Only the Java HotSpot Server VM supports this option.

-XX:AllocatePrefetchInstr=*instruction*

Sets the prefetch instruction to prefetch ahead of the allocation pointer. Only the Java HotSpot Server VM supports this option. Possible values are from 0 to 3. The actual instructions behind the values depend on the platform. By default, the prefetch instruction is set to 0:

```
-XX:AllocatePrefetchInstr=0
```

Only the Java HotSpot Server VM supports this option.

-XX:AllocatePrefetchLines=*lines*

Sets the number of cache lines to load after the last object allocation by using the prefetch instructions generated in compiled code. The default value is 1 if the last allocated object was an instance, and 3 if it was an array.

The following example shows how to set the number of loaded cache lines to 5:

```
-XX:AllocatePrefetchLines=5
```

Only the Java HotSpot Server VM supports this option.

-XX:AllocatePrefetchStepSize=*size*

Sets the step size (in bytes) for sequential prefetch instructions. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. By default, the step size is set to 16 bytes:

```
-XX:AllocatePrefetchStepSize=16
```

Only the Java HotSpot Server VM supports this option.

-XX:AllocatePrefetchStyle=*style*

Sets the generated code style for prefetch instructions. The *style* argument is an integer from 0 to 3:

0

Do not generate prefetch instructions.

1

Execute prefetch instructions after each allocation. This is the default parameter.

2

Use the thread-local allocation block (TLAB) watermark pointer to determine when prefetch instructions are executed.

3

Use BIS instruction on SPARC for allocation prefetch.

Only the Java HotSpot Server VM supports this option.

-XX:+BackgroundCompilation

Enables background compilation. This option is enabled by default. To disable background compilation, specify `-XX:-BackgroundCompilation` (this is equivalent to specifying `-Xbatch`).

-XX:CICompilerCount=*threads*

Sets the number of compiler threads to use for compilation. By default, the number of threads is set to 2 for the server JVM, to 1 for the client JVM, and it scales to the number of cores if tiered compilation is used. The following example shows how to set the number of threads to 2:

```
-XX:CICompilerCount=2
```

-XX:CodeCacheMinimumFreeSpace=*size*

Sets the minimum free space (in bytes) required for compilation. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. When less than the minimum free space remains, compiling stops. By default, this option is set to 500 KB. The following example shows how to set the minimum free space to 1024 MB:

```
-XX:CodeCacheMinimumFreeSpace=1024m
```

-XX:CompileCommand=*command*,*method*[,*option*]

Specifies a command to perform on a method. For example, to exclude the `indexOf()` method of the `String` class from being compiled, use the following:

```
-XX:CompileCommand=exclude,java/lang/String.indexOf
```

Note that the full class name is specified, including all packages and subpackages separated by a slash (`/`). For easier cut and paste operations, it is also possible to use the method name format produced by the `-XX:+PrintCompilation` and `-XX:+LogCompilation` options:

```
-XX:CompileCommand=exclude,java.lang.String::indexOf
```

If the method is specified without the signature, the command will be applied to all methods with the specified name. However, you can also specify the signature of the method in the class file format. In this case, you should enclose the arguments in quotation marks, because otherwise the shell treats the semicolon as command end. For example, if you want to exclude only the `indexOf(String)` method of the `String` class from being compiled, use the following:

```
-XX:CompileCommand="exclude,java/lang/String.indexOf,(Ljava/lang/String;)I"
```

You can also use the asterisk (*) as a wildcard for class and method names. For example, to exclude all `indexOf()` methods in all classes from being compiled, use the following:

```
-XX:CompileCommand=exclude,*.indexOf
```

The commas and periods are aliases for spaces, making it easier to pass compiler commands through a shell. You can pass arguments to `-XX:CompileCommand` using spaces as separators by enclosing the argument in quotation marks:

```
-XX:CompileCommand="exclude java/lang/String indexOf"
```

Note that after parsing the commands passed on the command line using the `-XX:CompileCommand` options, the JIT compiler then reads commands from the `.hotspot_compiler` file. You can add commands to this file or specify a different file using the `-XX:CompileCommandFile` option.

To add several commands, either specify the `-XX:CompileCommand` option multiple times, or separate each argument with the newline separator (`\n`). The following commands are available:

break

Set a breakpoint when debugging the JVM to stop at the beginning of compilation of the specified method.

compileonly

Exclude all methods from compilation except for the specified method. As an alternative, you can use the `-XX:CompileOnly` option, which allows to specify several methods.

dontinline

Prevent inlining of the specified method.

exclude

Exclude the specified method from compilation.

help

Print a help message for the `-XX:CompileCommand` option.

inline

Attempt to inline the specified method.

log

Exclude compilation logging (with the `-XX:+LogCompilation` option) for all methods except for the specified method. By default, logging is performed for all compiled methods.

option

This command can be used to pass a JIT compilation option to the specified method in place of the last argument (*option*). The compilation option is set at the end, after the method name. For example, to enable the `BlockLayoutByFrequency` option for the `append()` method of the `StringBuffer` class, use the following:

```
-XX:CompileCommand=option,java/lang/StringBuffer.append,BlockLayoutByFrequency
```

You can specify multiple compilation options, separated by commas or spaces.

print

Print generated assembler code after compilation of the specified method.

quiet

Do not print the compile commands. By default, the commands that you specify with the `-XX:CompileCommand` option are printed; for example, if you exclude from compilation the `indexOf()` method of the `String` class, then the following will be printed to standard output:

```
CompilerOracle: exclude java/lang/String.indexOf
```

You can suppress this by specifying the `-XX:CompileCommand=quiet` option before other `-XX:CompileCommand` options.

-XX:CompileCommandFile=*filename*

Sets the file from which JIT compiler commands are read. By default, the `.hotspot_compiler` file is used to store commands performed by the JIT compiler.

Each line in the command file represents a command, a class name, and a method name for which the command is used. For example, this line prints assembly code for the `toString()` method of the `String` class:

```
print java/lang/String toString
```

For more information about specifying the commands for the JIT compiler to perform on methods, see the `-XX:CompileCommand` option.

-XX:CompileOnly=*methods*

Sets the list of methods (separated by commas) to which compilation should be restricted. Only the specified methods will be compiled. Specify each method with the full class name (including the packages and subpackages). For example, to compile only the `length()` method of the `String` class and the `size()` method of the `List` class, use the following:

```
-XX:CompileOnly=java/lang/String.length,java/util/List.size
```

Note that the full class name is specified, including all packages and subpackages separated by a slash (`/`). For easier cut and paste operations, it is also possible to use the method name format produced by the `-XX:+PrintCompilation` and `-XX:+LogCompilation` options:

```
-XX:CompileOnly=java.lang.String::length,java.util.List::size
```

Although wildcards are not supported, you can specify only the class or package name to compile all methods in that class or package, as well as specify just the method to compile methods with this name in any class:

```
-XX:CompileOnly=java/lang/String
-XX:CompileOnly=java/lang
-XX:CompileOnly=.length
```

-XX:CompileThreshold=*invocations*

Sets the number of interpreted method invocations before compilation. By default, in the server JVM, the JIT compiler performs 10,000 interpreted method invocations to gather information for efficient compilation. For the client JVM, the default setting is 1,500 invocations. This option is ignored when tiered compilation is enabled; see the option `-XX:+TieredCompilation`. The following example shows how to set the number of interpreted method invocations to 5,000:

```
-XX:CompileThreshold=5000
```

You can completely disable interpretation of Java methods before compilation by specifying the `-Xcomp` option.

-XX:+DoEscapeAnalysis

Enables the use of escape analysis. This option is enabled by default. To disable the use of escape analysis, specify `-XX:-DoEscapeAnalysis`. Only the Java HotSpot Server VM supports this option.

-XX:InitialCodeCacheSize=*size*

Sets the initial code cache size (in bytes). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default value is set to 500 KB. The initial code cache size should be not less than the system's minimal memory page size. The following example shows how to set the initial code cache size to 32 KB:

```
-XX:InitialCodeCacheSize=32k
```

-XX:+Inline

Enables method inlining. This option is enabled by default to increase performance. To disable method inlining, specify `-XX:-Inline`.

-XX:InlineSmallCode=*size*

Sets the maximum code size (in bytes) for compiled methods that should be inlined. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. Only compiled methods with the size smaller than the specified size will be inlined. By default, the maximum code size is set to 1000 bytes:

```
-XX:InlineSmallCode=1000
```

-XX:+LogCompilation

Enables logging of compilation activity to a file named `hotspot.log` in the current working directory. You can specify a different log file path and name using the `-XX:LogFile` option.

By default, this option is disabled and compilation activity is not logged. The `-XX:+LogCompilation` option has to be used together with the `-XX:+UnlockDiagnosticVMOptions` option that unlocks diagnostic JVM options.

You can enable verbose diagnostic output with a message printed to the console every time a method is compiled by using the `-XX:+PrintCompilation` option.

-XX:MaxInlineSize=*size*

Sets the maximum bytecode size (in bytes) of a method to be inlined. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. By default, the maximum bytecode size is set to 35 bytes:

```
-XX:MaxInlineSize=35
```

-XX:MaxNodeLimit=*nodes*

Sets the maximum number of nodes to be used during single method compilation. By default, the maximum number of nodes is set to 65,000:

```
-XX:MaxNodeLimit=65000
```

-XX:MaxTrivialSize=*size*

Sets the maximum bytecode size (in bytes) of a trivial method to be inlined. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. By default, the maximum bytecode size of a trivial method is set to 6 bytes:

```
-XX:MaxTrivialSize=6
```

-XX:+OptimizeStringConcat

Enables the optimization of `String` concatenation operations. This option is enabled by default. To disable the optimization of `String` concatenation operations, specify `-XX:-OptimizeStringConcat`. Only the Java HotSpot Server VM supports this option.

-XX:+PrintAssembly

Enables printing of assembly code for bytecoded and native methods by using the external `disassembler.so` library. This enables you to see the generated code, which may help you to diagnose performance issues.

By default, this option is disabled and assembly code is not printed. The `-XX:+PrintAssembly` option has to be used together with the `-XX:+UnlockDiagnosticVMOptions` option that unlocks diagnostic JVM options.

-XX:+PrintCompilation

Enables verbose diagnostic output from the JVM by printing a message to the console every time a method is compiled. This enables you to see which methods actually get compiled. By default, this option is disabled and diagnostic output is not printed.

You can also log compilation activity to a file by using the `-XX:+LogCompilation` option.

-XX:+PrintInlining

Enables printing of inlining decisions. This enables you to see which methods are getting inlined.

By default, this option is disabled and inlining information is not printed. The `-XX:+PrintInlining` option has to be used together with the `-XX:+UnlockDiagnosticVMOptions` option that unlocks diagnostic JVM options.

-XX:ReservedCodeCacheSize=*size*

Sets the maximum code cache size (in bytes) for JIT-compiled code. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default maximum code cache size is 240 MB; if you disable tiered compilation with the option `-XX:-TieredCompilation`, then the default size is 48 MB. This option has a limit of 2 GB; otherwise, an error is generated. The maximum code cache size should not be less than the initial code cache size; see the option `-XX:InitialCodeCacheSize`. This option is equivalent to `-Xmaxjitcodesize`.

-XX:RTMAbortRatio=*abort_ratio*

The RTM abort ratio is specified as a percentage (%) of all executed RTM transactions. If a number of aborted transactions becomes greater than this ratio, then the compiled code will be deoptimized. This ratio is used when the `-XX:+UseRTMDeopt` option is enabled. The default value of this option is 50. This means that the compiled code will be deoptimized if 50% of all transactions are aborted.

-XX:RTMRetryCount=*number_of_retries*

RTM locking code will be retried, when it is aborted or busy, the number of times specified by this option before falling back to the normal locking mechanism. The default value for this option is 5. The `-XX:UseRTMLocking` option must be enabled.

-XX:-TieredCompilation

Disables the use of tiered compilation. By default, this option is enabled. Only the Java HotSpot Server VM supports this option.

-XX:+UseAES

Enables hardware-based AES intrinsics for Intel, AMD, and SPARC hardware. Intel Westmere (2010 and newer), AMD Bulldozer (2011 and newer), and SPARC (T4 and newer) are the supported hardware. UseAES is used in conjunction with UseAESIntrinsics.

-XX:+UseAESIntrinsics

UseAES and UseAESIntrinsics flags are enabled by default and are supported only for Java HotSpot Server VM 32-bit and 64-bit. To disable hardware-based AES intrinsics, specify `-XX:-UseAES -XX:-UseAESIntrinsics`. For example, to enable hardware AES, use the following flags:

```
-XX:+UseAES -XX:+UseAESIntrinsics
```

To support UseAES and UseAESIntrinsics flags for 32-bit and 64-bit use `-server` option to choose Java HotSpot Server VM. These flags are not supported on Client VM.

-XX:+UseCodeCacheFlushing

Enables flushing of the code cache before shutting down the compiler. This option is enabled by default. To disable flushing of the code cache before shutting down the compiler, specify `-XX:-UseCodeCacheFlushing`.

-XX:+UseCondCardMark

Enables checking of whether the card is already marked before updating the card table. This option is disabled by default and should only be used on machines with multiple sockets, where it will increase performance of Java applications that rely heavily on concurrent operations. Only the Java HotSpot Server VM supports this option.

-XX:+UseRTMDeopt

Auto-tunes RTM locking depending on the abort ratio. This ratio is specified by `-XX:RTMAbortRatio` option. If the number of aborted transactions exceeds the abort ratio, then the method containing the lock will be deoptimized and recompiled with all locks as normal locks. This option is disabled by default. The `-XX:+UseRTMLocking` option must be enabled.

-XX:+UseRTMLocking

Generate Restricted Transactional Memory (RTM) locking code for all inflated locks, with the normal locking mechanism as the fallback handler. This option is disabled by default. Options related to RTM are only available for the Java HotSpot Server VM on x86 CPUs that support Transactional Synchronization Extensions (TSX).

RTM is part of Intel's TSX, which is an x86 instruction set extension and facilitates the creation of multithreaded applications. RTM introduces the new instructions `XBEGIN`, `XABORT`, `XEND`, and `XTEST`. The `XBEGIN` and `XEND` instructions enclose a set of instructions to run as a transaction. If no conflict is found when running the transaction, the memory and register modifications are committed together at the `XEND` instruction. The `XABORT` instruction can be used to explicitly abort a transaction and the `XEND` instruction to check if a set of instructions are being run in a transaction.

A lock on a transaction is inflated when another thread tries to access the same transaction, thereby blocking the thread that did not originally request access to the transaction. RTM requires that a fallback set of operations be specified in case a transaction aborts or fails. An RTM lock is a lock that has been delegated to the TSX's system.

RTM improves performance for highly contended locks with low conflict in a critical region (which is code that must not be accessed by more than one thread concurrently). RTM also improves the performance of coarse-grain locking, which typically does not perform well in multithreaded applications. (Coarse-grain locking is the strategy of holding locks for long periods to minimize the overhead of taking and releasing locks, while fine-grained locking is the strategy of trying to achieve maximum parallelism by locking only when necessary and unlocking as soon as possible.) Also, for lightly contended locks that are used by different threads, RTM can reduce false cache line sharing, also known as cache line ping-pong. This occurs when multiple threads from different processors are accessing different resources, but the resources share the same cache line. As a result, the processors repeatedly invalidate the cache lines of other processors, which forces them to read from main memory instead of their cache.

-XX:+UseSHA

Enables hardware-based intrinsics for SHA crypto hash functions for SPARC hardware. `UseSHA` is used in conjunction with the `UseSHA1Intrinsics`, `UseSHA256Intrinsics`, and `UseSHA512Intrinsics` options.

The `UseSHA` and `UseSHA*Intrinsics` flags are enabled by default, and are supported only for Java HotSpot Server VM 64-bit on SPARC T4 and newer.

This feature is only applicable when using the `sun.security.provider.Sun` provider for SHA operations.

To disable all hardware-based SHA intrinsics, specify `-XX:-UseSHA`. To disable only a particular SHA intrinsic, use the appropriate corresponding option. For example: `-XX:-UseSHA256Intrinsics`.

-XX:+UseSHA1Intrinsics

Enables intrinsics for SHA-1 crypto hash function.

-XX:+UseSHA256Intrinsics

Enables intrinsics for SHA-224 and SHA-256 crypto hash functions.

-XX:+UseSHA512Intrinsics

Enables intrinsics for SHA-384 and SHA-512 crypto hash functions.

-XX:+UseSuperWord

Enables the transformation of scalar operations into superword operations. This option is enabled by default. To disable the transformation of scalar operations into superword operations, specify `-XX:-UseSuperWord`. Only the Java HotSpot Server VM supports this option.

Advanced Serviceability Options

These options provide the ability to gather system information and perform extensive debugging.

-XX:+HeapDumpOnOutOfMemoryError

Enables the dumping of the Java heap to a file in the current directory by using the heap profiler (HPROF) when a `java.lang.OutOfMemoryError` exception is thrown. You can explicitly set the heap dump file path and name using the `-XX:HeapDumpPath` option. By default, this option is disabled and the heap is not dumped when an `OutOfMemoryError` exception is thrown.

-XX:HeapDumpPath=*path*

Sets the path and file name for writing the heap dump provided by the heap profiler (HPROF) when the `-XX:+HeapDumpOnOutOfMemoryError` option is set. By default, the file is created in the current working directory, and it is named `java_pid`*pid*`.hprof` where *pid* is the identifier of the process that caused the error. The following example shows how to set the default file explicitly (`%p` represents the current process identificator):

```
-XX:HeapDumpPath=./java_pid%p.hprof
```

The following example shows how to set the heap dump file to `C:/log/java/java_heapdump.log`:

```
-XX:HeapDumpPath=C:/log/java/java_heapdump.log
```

-XX:LogFile=*path*

Sets the path and file name where log data is written. By default, the file is created in the current working directory, and it is named `hotspot.log`.

The following example shows how to set the log file to `C:/log/java/hotspot.log`:

```
-XX:LogFile=C:/log/java/hotspot.log
```

-XX:+PrintClassHistogram

Enables printing of a class instance histogram after a `Control+Break` event. By default, this option is disabled.

Setting this option is equivalent to running the `jmap -histo` command, or the `jcmd` *pid* `GC.class_histogram` command, where *pid* is the current Java process identifier.

-XX:+PrintConcurrentLocks

Enables printing of `java.util.concurrent` locks after a `Control+Break` event. By default, this option is disabled.

Setting this option is equivalent to running the `jstack -l` command or the `jcmd` *pid* `Thread.print -l` command, where *pid* is the current Java process identifier.

-XX:+UnlockDiagnosticVMOptions

Unlocks the options intended for diagnosing the JVM. By default, this option is disabled and diagnostic options are not available.

Advanced Garbage Collection Options

These options control how garbage collection (GC) is performed by the Java HotSpot VM.

-XX:ActiveProcessorCount=*x*

Overrides the number of CPUs that the VM uses to calculate the size of thread pools it uses for various operations such as Garbage Collection and ForkJoinPool.

The VM normally determines the number of available processors from the operating system. This flag can be useful for partitioning CPU resources when running multiple Java processes in docker containers. This flag is honored even if `UseContainerSupport` is not enabled. See `-XX:-UseContainerSupport` for a description of enabling and disabling container support.

-XX:+AggressiveHeap

Enables Java heap optimization. This sets various parameters to be optimal for long-running jobs with intensive memory allocation, based on the configuration of the computer (RAM and CPU). By default, the option is disabled and the heap is not optimized.

-XX:+AlwaysPreTouch

Enables touching of every page on the Java heap during JVM initialization. This gets all pages into the memory before entering the `main()` method. The option can be used in testing to simulate a long-running system with all virtual memory mapped to physical memory. By default, this option is disabled and all pages are committed as JVM heap space fills.

-XX:+CMSClassUnloadingEnabled

Enables class unloading when using the concurrent mark-sweep (CMS) garbage collector. This option is enabled by default. To disable class unloading for the CMS garbage collector, specify `-XX:-CMSClassUnloadingEnabled`.

-XX:CMSExpAvgFactor=*percent*

Sets the percentage of time (0 to 100) used to weight the current sample when computing exponential averages for the concurrent collection statistics. By default, the exponential averages factor is set to 25%. The following example shows how to set the factor to 15%:

```
-XX:CMSExpAvgFactor=15
```

-XX:CMSInitiatingOccupancyFraction=*percent*

Sets the percentage of the old generation occupancy (0 to 100) at which to start a CMS collection cycle. The default value is set to -1. Any negative value (including the default) implies that `-XX:CMSTriggerRatio` is used to define the value of the initiating occupancy fraction.

The following example shows how to set the occupancy fraction to 20%:

```
-XX:CMSInitiatingOccupancyFraction=20
```

-XX:+CMSScavengeBeforeRemark

Enables scavenging attempts before the CMS remark step. By default, this option is disabled.

-XX:CMSTriggerRatio=*percent*

Sets the percentage (0 to 100) of the value specified by `-XX:MinHeapFreeRatio` that is allocated before a CMS collection cycle commences. The default value is set to 80%.

The following example shows how to set the occupancy fraction to 75%:

```
-XX:CMSTriggerRatio=75
```

-XX:ConcGCThreads=*threads*

Sets the number of threads used for concurrent GC. The default value depends on the number of CPUs available to the JVM.

For example, to set the number of threads for concurrent GC to 2, specify the following option:

```
-XX:ConcGCThreads=2
```

-XX:+DisableExplicitGC

Enables the option that disables processing of calls to `System.gc()`. This option is disabled by default, meaning that calls to `System.gc()` are processed. If processing of calls to `System.gc()` is disabled, the JVM still performs GC when necessary.

-XX:+ExplicitGCInvokesConcurrent

Enables invoking of concurrent GC by using the `System.gc()` request. This option is disabled by default and can be enabled only together with the `-XX:+UseConcMarkSweepGC` option.

-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

Enables invoking of concurrent GC by using the `System.gc()` request and unloading of classes during the concurrent GC cycle. This option is disabled by default and can be enabled only together with the `-XX:+UseConcMarkSweepGC` option.

-XX:G1HeapRegionSize=*size*

Sets the size of the regions into which the Java heap is subdivided when using the garbage-first (G1) collector. The value can be between 1 MB and 32 MB. The default region size is determined ergonomically based on the heap size.

The following example shows how to set the size of the subdivisions to 16 MB:

```
-XX:G1HeapRegionSize=16m
```

-XX:+G1PrintHeapRegions

Enables the printing of information about which regions are allocated and which are reclaimed by the G1 collector. By default, this option is disabled.

-XX:G1ReservePercent=*percent*

Sets the percentage of the heap (0 to 50) that is reserved as a false ceiling to reduce the possibility of promotion failure for the G1 collector. By default, this option is set to 10%.

The following example shows how to set the reserved heap to 20%:

```
-XX:G1ReservePercent=20
```

-XX:InitialHeapSize=*size*

Sets the initial size (in bytes) of the memory allocation pool. This value must be either 0, or a multiple of 1024 and greater than 1 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default value is chosen at runtime based on system configuration. See the section "Ergonomics" in *Java SE HotSpot Virtual Machine Garbage Collection Tuning Guide* at `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/index.html`.

The following examples show how to set the size of allocated memory to 6 MB using various units:

```
-XX:InitialHeapSize=6291456
-XX:InitialHeapSize=6144k
-XX:InitialHeapSize=6m
```

If you set this option to 0, then the initial size will be set as the sum of the sizes allocated for the old generation and the young generation. The size of the heap for the young generation can be set using the `-XX:NewSize` option.

-XX:InitialRAMPercentage=*percent*

Sets the initial amount of memory that the JVM will use for the Java heap before applying ergonomics heuristics as a percentage of the maximum amount determined as described in the `-XX:MaxRAM` option. The default value is 1.5625 percent.

The following example shows how to set the percentage of the initial amount of memory used for the Java heap:

```
-XX:InitialRAMPercentage=5
```

-XX:InitialSurvivorRatio=*ratio*

Sets the initial survivor space ratio used by the throughput garbage collector (which is enabled by the `-XX:+UseParallelGC` and/or `-XX:+UseParallelOldGC` options). Adaptive sizing is enabled by default with the throughput garbage collector by using the `-XX:+UseParallelGC` and `-XX:+UseParallelOldGC` options, and survivor space is resized according to the application behavior, starting with the initial value. If adaptive sizing is disabled (using the `-XX:-UseAdaptiveSizePolicy` option), then the `-XX:SurvivorRatio` option should be used to set the size of the survivor space for the entire execution of the application.

The following formula can be used to calculate the initial size of survivor space (S) based on the size of the young generation (Y), and the initial survivor space ratio (R):

```
S=Y/(R+2)
```

The 2 in the equation denotes two survivor spaces. The larger the value specified as the initial survivor space ratio, the smaller the initial survivor space size.

By default, the initial survivor space ratio is set to 8. If the default value for the young generation space size is used (2 MB), the initial size of the survivor space will be 0.2 MB.

The following example shows how to set the initial survivor space ratio to 4:

```
-XX:InitialSurvivorRatio=4
```

-XX:InitiatingHeapOccupancyPercent=*percent*

Sets the percentage of the heap occupancy (0 to 100) at which to start a concurrent GC cycle. It is used by garbage collectors that trigger a concurrent GC cycle based on the occupancy of the entire heap, not just one of the generations (for example, the G1 garbage collector).

By default, the initiating value is set to 45%. A value of 0 implies nonstop GC cycles. The following example shows how to set the initiating heap occupancy to 75%:

```
-XX:InitiatingHeapOccupancyPercent=75
```

-XX:MaxGCPauseMillis=*time*

Sets a target for the maximum GC pause time (in milliseconds). This is a soft goal, and the JVM will make its best effort to achieve it. By default, there is no maximum pause time value.

The following example shows how to set the maximum target pause time to 500 ms:

```
-XX:MaxGCPauseMillis=500
```

-XX:MaxHeapSize=*size*

Sets the maximum size (in byes) of the memory allocation pool. This value must be a multiple of 1024 and greater than 2 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. The default value is chosen at runtime based on system configuration. For server deployments, `-XX:InitialHeapSize` and `-XX:MaxHeapSize` are often set to the same value. See the section "Ergonomics" in *Java SE HotSpot Virtual Machine Garbage Collection Tuning Guide* at `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/index.html`.

The following examples show how to set the maximum allowed size of allocated memory to 80 MB using various units:

```
-XX:MaxHeapSize=83886080
-XX:MaxHeapSize=81920k
-XX:MaxHeapSize=80m
```

The `-XX:MaxHeapSize` option is equivalent to `-Xmx`.

-XX:MaxHeapFreeRatio=*percent*

Sets the maximum allowed percentage of free heap space (0 to 100) after a GC event. If free heap space expands above this value, then the heap will be shrunk. By default, this value is set to 70%.

The following example shows how to set the maximum free heap ratio to 75%:

```
-XX:MaxHeapFreeRatio=75
```

-XX:MaxMetaspaceSize=*size*

Sets the maximum amount of native memory that can be allocated for class metadata. By default, the size is not limited. The amount of metadata for an application depends on the application itself, other running applications, and the amount of memory available on the system.

The following example shows how to set the maximum class metadata size to 256 MB:

```
-XX:MaxMetaspaceSize=256m
```

-XX:MaxRAMPercentage=*percent*

Sets the maximum amount of memory that the JVM may use for the Java heap before applying ergonomics heuristics as a percentage of the maximum amount determined as described in the `-XX:MaxRAM` option. The default value is 25 percent.

Specifying this option disables automatic use of compressed oops if the combined result of this and other options influencing the maximum amount of memory is larger than the range of memory addressable by compressed oops. See `-XX:UseCompressedOops` for further information about compressed oops.

The following example shows how to set the percentage of the maximum amount of memory used for the Java heap:

```
-XX:MaxRAMPercentage=75
```

-XX:MaxNewSize=*size*

Sets the maximum size (in bytes) of the heap for the young generation (nursery). The default value is set ergonomically.

-XX:MaxTenuringThreshold=*threshold*

Sets the maximum tenuring threshold for use in adaptive GC sizing. The largest value is 15. The default value is 15 for the parallel (throughput) collector, and 6 for the CMS collector.

The following example shows how to set the maximum tenuring threshold to 10:

```
-XX:MaxTenuringThreshold=10
```

-XX:MetaspaceSize=*size*

Sets the size of the allocated class metadata space that will trigger a garbage collection the first time it is exceeded. This threshold for a garbage collection is increased or decreased depending on the amount of metadata used. The default size depends on the platform.

-XX:MinHeapFreeRatio=*percent*

Sets the minimum allowed percentage of free heap space (0 to 100) after a GC event. If free heap space falls below this value, then the heap will be expanded. By default, this value is set to 40%.

The following example shows how to set the minimum free heap ratio to 25%:

```
-XX:MinHeapFreeRatio=25
```

-XX:MinRAMPercentage=*percent*

Sets the maximum amount of memory that the JVM may use for the Java heap before applying ergonomics heuristics as a percentage of the maximum amount determined as described in the `-XX:MaxRAM` option for small heaps. A small heap is a heap of approximately 125 MB. The default value is 50 percent.

The following example shows how to set the percentage of the maximum amount of memory used for the Java heap for small heaps:

```
-XX:MinRAMPercentage=75
```

-XX:NewRatio=*ratio*

Sets the ratio between young and old generation sizes. By default, this option is set to 2. The following example shows how to set the young/old ratio to 1:

```
-XX:NewRatio=1
```

-XX:NewSize=*size*

Sets the initial size (in bytes) of the heap for the young generation (nursery). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes.

The young generation region of the heap is used for new objects. GC is performed in this region more often than in other regions. If the size for the young generation is too low, then a large number of minor GCs will be performed. If the size is too high, then only full GCs will be performed, which can take a long time to complete. Oracle recommends that you keep the size for the young generation between a half and a quarter of the overall heap size.

The following examples show how to set the initial size of young generation to 256 MB using various units:

```
-XX:NewSize=256m
-XX:NewSize=262144k
-XX:NewSize=268435456
```

The `-XX:NewSize` option is equivalent to `-Xmn`.

-XX:ParallelGCThreads=*threads*

Sets the number of threads used for parallel garbage collection in the young and old generations. The default value depends on the number of CPUs available to the JVM.

For example, to set the number of threads for parallel GC to 2, specify the following option:

```
-XX:ParallelGCThreads=2
```

-XX:+ParallelRefProcEnabled

Enables parallel reference processing. By default, this option is disabled.

-XX:+PrintAdaptiveSizePolicy

Enables printing of information about adaptive generation sizing. By default, this option is disabled.

-XX:+PrintGC

Enables printing of messages at every GC. By default, this option is disabled.

-XX:+PrintGCApplicationConcurrentTime

Enables printing of how much time elapsed since the last pause (for example, a GC pause). By default, this option is disabled.

-XX:+PrintGCApplicationStoppedTime

Enables printing of how much time the pause (for example, a GC pause) lasted. By default, this option is disabled.

-XX:+PrintGCDateStamps

Enables printing of a date stamp at every GC. By default, this option is disabled.

-XX:+PrintGCDetails

Enables printing of detailed messages at every GC. By default, this option is disabled.

-XX:+PrintGCTaskTimeStamps

Enables printing of time stamps for every individual GC worker thread task. By default, this option is disabled.

-XX:+PrintGCTimeStamps

Enables printing of time stamps at every GC. By default, this option is disabled.

-XX:+PrintStringDeduplicationStatistics

Prints detailed deduplication statistics. By default, this option is disabled. See the `-XX:+UseStringDeduplication` option.

-XX:+PrintTenuringDistribution

Enables printing of tenuring age information. The following is an example of the output:

```
Desired survivor size 48286924 bytes, new threshold 10 (max 10)
- age 1: 28992024 bytes, 28992024 total
- age 2: 1366864 bytes, 30358888 total
- age 3: 1425912 bytes, 31784800 total
...
```

Age 1 objects are the youngest survivors (they were created after the previous scavenge, survived the latest scavenge, and moved from eden to survivor space). Age 2 objects have survived two scavenges (during the second scavenge they were copied from one survivor space to the next). And so on.

In the preceding example, 28 992 024 bytes survived one scavenge and were copied from eden to survivor space, 1 366 864 bytes are occupied by age 2 objects, etc. The third value in each row is the cumulative size of objects of age n or less.

By default, this option is disabled.

-XX:+ScavengeBeforeFullGC

Enables GC of the young generation before each full GC. This option is enabled by default. Oracle recommends that you *do not* disable it, because scavenging the young generation before a full GC can reduce the number of objects reachable from the old generation space into the young generation space. To disable GC of the young generation before each full GC, specify `-XX:-ScavengeBeforeFullGC`.

-XX:SoftRefLRUPolicyMSPerMB=*time*

Sets the amount of time (in milliseconds) a softly reachable object is kept active on the heap after the last time it was referenced. The default value is one second of lifetime per free megabyte in the heap. The `-XX:SoftRefLRUPolicyMSPerMB` option accepts integer values representing milliseconds per one megabyte of the current heap size (for Java HotSpot Client VM) or the maximum possible heap size (for Java HotSpot Server VM). This difference means that the Client VM tends to flush soft references rather than grow the heap, whereas the Server VM tends to grow the heap rather than flush soft references. In the latter case, the value of the `-Xmx` option has a significant effect on how quickly soft references are garbage collected.

The following example shows how to set the value to 2.5 seconds:

```
-XX:SoftRefLRUPolicyMSPerMB=2500
```

-XX:StringDeduplicationAgeThreshold=*threshold*

`String` objects reaching the specified age are considered candidates for deduplication. An object's age is a measure of how many times it has survived garbage collection. This is sometimes referred to as tenuring; see the `-XX:+PrintTenuringDistribution` option. Note that `String` objects that are promoted to an old heap region before this age has been reached are always considered candidates for deduplication. The default value for this option is `3`. See the `-XX:+UseStringDeduplication` option.

-XX:SurvivorRatio=*ratio*

Sets the ratio between eden space size and survivor space size. By default, this option is set to 8. The following example shows how to set the eden/survivor space ratio to 4:

```
-XX:SurvivorRatio=4
```

-XX:TargetSurvivorRatio=*percent*

Sets the desired percentage of survivor space (0 to 100) used after young garbage collection. By default, this option is set to 50%.

The following example shows how to set the target survivor space ratio to 30%:

```
-XX:TargetSurvivorRatio=30
```

-XX:TLABSize=*size*

Sets the initial size (in bytes) of a thread-local allocation buffer (TLAB). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. If this option is set to 0, then the JVM chooses the initial size automatically.

The following example shows how to set the initial TLAB size to 512 KB:

```
-XX:TLABSize=512k
```

-XX:+UseAdaptiveSizePolicy

Enables the use of adaptive generation sizing. This option is enabled by default. To disable adaptive generation sizing, specify `-XX:-UseAdaptiveSizePolicy` and set the size of the memory allocation pool explicitly (see the `-XX:SurvivorRatio` option).

-XX:+UseCMSInitiatingOccupancyOnly

Enables the use of the occupancy value as the only criterion for initiating the CMS collector. By default, this option is disabled and other criteria may be used.

-XX:+UseConcMarkSweepGC

Enables the use of the CMS garbage collector for the old generation. Oracle recommends that you use the CMS garbage collector when application latency requirements cannot be met by the throughput (`-XX:+UseParallelGC`) garbage collector. The G1 garbage collector (`-XX:+UseG1GC`) is another alternative.

By default, this option is disabled and the collector is chosen automatically based on the configuration of the machine and type of the JVM. When this option is enabled, the `-XX:+UseParNewGC` option is automatically set and you should not disable it, because the following combination of options has been deprecated in JDK 8: `-XX:+UseConcMarkSweepGC -XX:-UseParNewGC`.

-XX:+UseG1GC

Enables the use of the garbage-first (G1) garbage collector. It is a server-style garbage collector, targeted for multiprocessor machines with a large amount of RAM. It meets GC pause time goals with high probability, while maintaining good throughput. The G1 collector is recommended for applications requiring large heaps (sizes of around 6 GB or larger) with limited GC latency requirements (stable and predictable pause time below 0.5 seconds).

By default, this option is disabled and the collector is chosen automatically based on the configuration of the machine and type of the JVM.

-XX:+UseGCOverheadLimit

Enables the use of a policy that limits the proportion of time spent by the JVM on GC before an `OutOfMemoryError` exception is thrown. This option is enabled, by default and the parallel GC will throw an `OutOfMemoryError` if more than 98% of the total time is spent on garbage collection and less than 2% of the heap is recovered. When the heap is small, this feature can be used to prevent applications from running for long periods of time with little or no progress. To disable this option, specify `-XX:-UseGCOverheadLimit`.

-XX:+UseNUMA

Enables performance optimization of an application on a machine with nonuniform memory architecture (NUMA) by increasing the application's use of lower latency memory. By default, this option is disabled and no optimization for NUMA is made. The option is only available when the parallel garbage collector is used (`-XX:+UseParallelGC`).

-XX:+UseParallelGC

Enables the use of the parallel scavenge garbage collector (also known as the throughput collector) to improve the performance of your application by leveraging multiple processors.

By default, this option is disabled and the collector is chosen automatically based on the configuration of the machine and type of the JVM. If it is enabled, then the `-XX:+UseParallelOldGC` option is automatically enabled, unless you explicitly disable it.

-XX:+UseParallelOldGC

Enables the use of the parallel garbage collector for full GCs. By default, this option is disabled. Enabling it automatically enables the `-XX:+UseParallelGC` option.

-XX:+UseParNewGC

Enables the use of parallel threads for collection in the young generation. By default, this option is disabled. It is automatically enabled when you set the `-XX:+UseConcMarkSweepGC` option. Using the `-XX:+UseParNewGC` option without the `-XX:+UseConcMarkSweepGC` option was deprecated in JDK 8.

-XX:+UseSerialGC

Enables the use of the serial garbage collector. This is generally the best choice for small and simple applications that do not require any special functionality from garbage collection. By default, this option is disabled and the collector is chosen automatically based on the configuration of the machine and type of the JVM.

-XX:+UseStringDeduplication

Enables string deduplication. By default, this option is disabled. To use this option, you must enable the garbage-first (G1) garbage collector. See the `-XX:+UseG1GC` option.

String deduplication reduces the memory footprint of `String` objects on the Java heap by taking advantage of the fact that many `String` objects are identical. Instead of each `String` object pointing to its own character array, identical `String` objects can point to and share the same character array.

-XX:+UseTLAB

Enables the use of thread-local allocation blocks (TLABs) in the young generation space. This option is enabled by default. To disable the use of TLABs, specify `-XX:-UseTLAB`.

Deprecated and Removed Options

These options were included in the previous release, but have since been considered unnecessary.

-Xincgc

Enables incremental garbage collection. This option was deprecated in JDK 8 with no replacement.

-Xrun*libname*

Loads the specified debugging/profiling library. This option was superseded by the `-agentlib` option.

-XX:CMSIncrementalDutyCycle=*percent*

Sets the percentage of time (0 to 100) between minor collections that the concurrent collector is allowed to run. This option was deprecated in JDK 8 with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option.

-XX:CMSIncrementalDutyCycleMin=*percent*

Sets the percentage of time (0 to 100) between minor collections that is the lower bound for the duty cycle when `-XX:+CMSIncrementalPacing` is enabled. This option was deprecated in JDK 8 with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option.

-XX:+CMSIncrementalMode

Enables the incremental mode for the CMS collector. This option was deprecated in JDK 8 with no replacement, along with other options that start with `CMSIncremental`.

-XX:CMSIncrementalOffset=*percent*

Sets the percentage of time (0 to 100) by which the incremental mode duty cycle is shifted to the right within the period between minor collections. This option was deprecated in JDK 8 with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option.

-XX:+CMSIncrementalPacing

Enables automatic adjustment of the incremental mode duty cycle based on statistics collected while the JVM is running. This option was deprecated in JDK 8 with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option.

-XX:CMSIncrementalSafetyFactor=*percent*

Sets the percentage of time (0 to 100) used to add conservatism when computing the duty cycle. This option was deprecated in JDK 8 with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option.

-XX:CMSInitiatingPermOccupancyFraction=*percent*

Sets the percentage of the permanent generation occupancy (0 to 100) at which to start a GC. This option was deprecated in JDK 8 with no replacement.

-XX:MaxPermSize=*size*

Sets the maximum permanent generation space size (in bytes). This option was deprecated in JDK 8, and superseded by the `-XX:MaxMetaspaceSize` option.

-XX:PermSize=*size*

Sets the space (in bytes) allocated to the permanent generation that triggers a garbage collection if it is exceeded. This option was deprecated un JDK 8, and superseded by the `-XX:MetaspaceSize` option.

-XX:+UseSplitVerifier

Enables splitting of the verification process. By default, this option was enabled in the previous releases, and verification was split into two phases: type referencing (performed by the compiler) and type checking (performed by the JVM runtime). This option was deprecated in JDK 8, and verification is now split by default without a way to disable it.

-XX:+UseStringCache

Enables caching of commonly allocated strings. This option was removed from JDK 8 with no replacement.

Performance Tuning Examples

The following examples show how to use experimental tuning flags to either optimize throughput or to provide lower response time.

Example 1 - Tuning for Higher Throughput

```
java -d64 -server -XX:+AggressiveOpts -XX:+UseLargePages -Xmn10g  -Xms26g -Xmx26g
```

Example 2 - Tuning for Lower Response Time

```
java -d64 -XX:+UseG1GC -Xms26g Xmx26g -XX:MaxGCPauseMillis=500 -XX:+PrintGCTimeStamp
```

Large Pages

Also known as huge pages, large pages are memory pages that are significantly larger than the standard memory page size (which varies depending on the processor and operating system). Large pages optimize processor Translation-Lookaside Buffers.

A Translation-Lookaside Buffer (TLB) is a page translation cache that holds the most-recently used virtual-to-physical address translations. TLB is a scarce system resource. A TLB miss can be costly as the processor must then read from the hierarchical page table, which may require multiple memory accesses. By using a larger memory page size, a single TLB entry can represent a larger memory range. There will be less pressure on TLB, and memory-intensive applications may have better performance.

However, large pages page memory can negatively affect system performance. For example, when a large mount of memory is pinned by an application, it may create a shortage of regular memory and cause excessive paging in other applications and slow down the entire system. Also, a system that has been up for a long time could produce excessive fragmentation, which could make it impossible to reserve enough large page memory. When this happens, either the OS or JVM reverts to using regular pages.

Large Pages Support

Only Windows Server 2003 supports large pages. To use this feature, the administrator must first assign additional privilege to the user who will be running the application:

1. Select **Control Panel** -> **Administrative Tools** -> **Local Security Policy**.
2. Select **Local Policies** -> **User Rights Assignment**.
3. Double click **Lock pages in memory**; add users and/or groups.
4. Reboot your system.

Note that these steps are required even if it is the administrator who will be running the application, as administrators by default do not have the privilege to lock pages in memory.

Application Class Data Sharing

Application Class Data Sharing (AppCDS) extends CDS (see `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/class-data-sharing.html`) to enable classes from the standard extensions directories (specified by the system property `java.ext.dirs`; see `https://docs.oracle.com/javase/8/docs/technotes/guides/extensions/spec.html`) and the application class path (see "Setting the Class Path") to be placed in the shared archive. AppCDS reduces the footprint and decreases start-up time of your applications provided that a substantial number of classes are loaded from the application class path.

This is a commercial feature that requires you to also specify the `-XX:+UnlockCommercialFeatures` option. This is also an experimental feature; it may change in future releases.

Creating a Shared Archive File, and Running an Application with It

The following steps create a shared archive file that contains all the classes used by the `test.Hello` application. The last step runs the application with the shared archive file.

1. Create a list of all classes used by the `test.Hello` application. The following command creates a file named `hello.classlist` that contains a list of all classes used by this application:

   ```
   java -Xshare:off -XX:+UnlockCommercialFeatures -XX:DumpLoadedClassList=hello.classlist -XX:+UseAppCDS -cp hello.jar test.Hello
   ```

   Note that the `-cp` parameter must contain only JAR files; the `-XX:+UseAppCDS` option does not support class paths that contain directory names.

2. Create a shared archive, named `hello.jsa`, that contains all the classes in `hello.classlist`:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:dump -XX:+UseAppCDS -XX:SharedArchiveFile=hello.jsa
   -XX:SharedClassListFile=hello.classlist -cp hello.jar
   ```

   Note that the `-cp` parameter used at archive creation time must be the same as (or a prefix of) the `-cp` used at run time.

3. Run the application `test.Hello` with the shared archive `hello.jsa`:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=hello.jsa -cp hello.jar test.Hello
   ```

   Ensure that you have specified the option `-Xshare:on` or `-Xshare:auto`.

4. Verify that the `test.Hello` application is using the class contained in the `hello.jsa` shared archive:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=hello.jsa -cp hello.jar -verbose:class
   test.Hello
   ```

   The output of this command should contain the following text:

   ```
   Loaded test.Hello from shared objects file by sun/misc/Launcher$AppClassLoader
   ```

Sharing a Shared Archive across Multiple Application Processes

You can share the same archive file across multiple applications processes that have the exact same class path or share a common class path prefix. This reduces memory usage as the archive is memory-mapped into the address space of the processes. The operating system automatically shares the read-only pages across these processes.

The following steps create a shared archive that both applications `Hello` and `Hi` can use.

1. Create a list of all classes used by the `Hello` application and another list for the `Hi` application:

   ```
   java -XX:+UnlockCommercialFeatures -XX:DumpLoadedClassList=hello.classlist -XX:+UseAppCDS -cp common.jar:hello.jar Hello
   ```

   ```
   java -XX:+UnlockCommercialFeatures -XX:DumpLoadedClassList=hi.classlist -XX:+UseAppCDS -cp common.jar:hi.jar Hi
   ```

   Note that because the `Hello` and `Hi` applications share a common class path prefix (both of their class paths start with `common.jar`), these two applications can share a shared archive file.

2. Create a single list of classes used by all the applications that will share the shared archive file.

   The following commands combine the files `hello.classlist` and `hi.classlist` to one file, `common.classlist`:

   ```
   type hello.classlist hi.classlist > common.classlist
   ```

3. Create a shared archive, named `common.jsa`, that contains all the classes in `common.classlist`:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:dump -XX:SharedArchiveFile=common.jsa -XX:+UseAppCDS
   -XX:SharedClassListFile=common.classlist -cp common.jar
   ```

   The value of the `-cp` parameter is the common class path prefix shared by the `Hello` and `Hi` applications.

4. Run the `Hello` and `Hi` applications with the same shared archive:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:SharedArchiveFile=common.jsa -XX:+UseAppCDS -cp common.jar:hello.jar Hello
   ```

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:SharedArchiveFile=common.jsa -XX:+UseAppCDS -cp common.jar:hi.jar Hi
   ```

Exit Status

The following exit values are typically returned by the launcher when the launcher is called with the wrong arguments, serious errors, or exceptions thrown by the JVM. However, a Java application may choose to return any value by using the API call `System.exit(exitValue)`. The values are:

- `0`: Successful completion
- `>0`: An error occurred

See Also

- javac(1)
- jdb(1)
- javah(1)
- jar(1)
- jstat(1)

Contents Previous Next