



Linux Ubuntu 26.04 Manual Pages on command 'java.1'

\$ man java.1

JAVA(1)

JDK Commands

JAVA(1)

NAME

java - launch a Java application

SYNOPSIS

To launch a class file:

java [options] mainclass [args ...]

To launch the main class in a JAR file:

java [options] -jar jarfile [args ...]

To launch the main class in a module:

java [options] -m module[/mainclass] [args ...]

or

java [options] --module module[/mainclass] [args ...]

To launch a source-file program:

java [options] source-file [args ...]

options

Optional: Specifies command-line options separated by spaces. See Overview of Java

Options for a description of available options.

mainclass

Specifies the name of the class to be launched. Command-line entries following classname are the arguments for the main method.

-jar jarfile

Executes a program encapsulated in a JAR file. The jarfile 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 -jar, the specified JAR file is the source of all user classes, and other class path settings are ignored. If you're using JAR files, then see jar.

-m or --module module[/mainclass]

Executes the main class in a module specified by mainclass if it is given, or, if it is not given, the value in the module. In other words, mainclass can be used when it is not specified by the module, or to override the value when it is specified.

See Standard Options for Java.

source-file

Only used to launch a source-file program. Specifies the source file that contains the main class when using source-file mode. See Using Source-File Mode to Launch Source-Code Programs

args ...

Optional: Arguments following mainclass, source-file, -jar jarfile, and -m or --module module/mainclass are passed as arguments to the main class.

DESCRIPTION

The java command starts a Java application. It does this by starting the Java Virtual Machine (JVM), 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)
```

In source-file mode, the java command can launch a class declared in a source file. See Using Source-File Mode to Launch Source-Code Programs for a description of using the source-file mode.

Note: You can use the JDK_JAVA_OPTIONS launcher environment variable to prepend its content to the actual command line of the java launcher. See Using the JDK_JAVA_OPTIONS Launcher Environment Variable.

By default, the first argument that isn't an option of the java command is the fully qualified name of the class to be called. If -jar is specified, then 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 manifest file.

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

javaw

Windows: The javaw command is identical to java, except that with javaw there's no associated console window. Use javaw when you don't want a command prompt window to appear. The javaw launcher will, however, display a dialog box with error information if a launch fails.

USING SOURCE-FILE MODE TO LAUNCH SOURCE-CODE PROGRAMS

To launch a class declared in a source file, run the java launcher in source-file mode. Enforcing source-file mode is determined by two items on the java command line:

? The first item on the command line that is not an option or part of an option. In other words, the item in the command line that would otherwise be the main class name.

? The --source version option, if present.

If the class identifies an existing file that has a .java extension, or if the --source option is specified, then source-file mode is selected. The source file is then compiled and run. The --source option can be used to specify the source version or N of the source code. This determines the API that can be used. When you set --source N, you can only use the public API that was defined in JDK N.

Note: The valid values of N change for each release, with new values added and old values removed. You'll get an error message if you use a value of N that is no longer supported. The supported values of N are the current Java SE release (25) and a limited number of previous releases, detailed in the command-line help for javac, under the --source and --release options.

If the file does not have the .java extension, the --source option must be used to tell the java command to use the source-file mode. The --source option is used for cases when the source file is a "script" to be executed and the name of the source file does not follow the normal naming conventions for Java source files.

In source-file mode, the effect is as though the source file is compiled into memory, and the first class found in the source file is executed. Any arguments placed after the name of the source file in the original command line are passed to the compiled class when it is executed.

For example, if a file were named HelloWorld.java and contained a class named HelloWorld, then the source-file mode command to launch the class would be:

```
java HelloWorld.java
```

This use of source-file mode is informally equivalent to using the following two commands:

```
javac -d <memory> --source-path <source-root> HelloWorld.java
```

```
java --class-path <memory> HelloWorld
```

where <source-root> is computed

In source-file mode, any additional command-line options are processed as follows:

? The launcher scans the options specified before the source file for any that are relevant in order to compile the source file.

This includes: --class-path, --module-path, --add-exports, --add-modules, --limit-modules, --patch-module, --upgrade-module-path, and any variant forms of those options. It also includes the new --enable-preview option, described in JEP 12.

? No provision is made to pass any additional options to the compiler, such as -processor or -Werror.

? Command-line argument files (@-files) may be used in the standard way. Long lists of arguments for either the VM or the program being invoked may be placed in files specified on the command-line by prefixing the filename with an @ character.

In source-file mode, compilation proceeds as follows:

? Any command-line options that are relevant to the compilation environment are taken into account. These include: --class-path/-classpath/-cp, --module-path/-p, --add-exports, --add-modules, --limit-modules, --patch-module, --upgrade-module-path, --enable-preview.

? The root of the source tree, <source-root> is computed from the package of the class being launched. For example, if HelloWorld.java declared its classes to be in the hello package, then the file HelloWorld.java is expected to reside in the directory somedir/hello/. In this case, somedir is computed to be the root of the source tree.

? The root of the source tree serves as the source-path for compilation, so that other source files found in that tree and are needed by HelloWorld could be compiled.

? Annotation processing is disabled, as if -proc:none is in effect.

? If a version is specified, via the --source option, the value is used as the argument for an implicit --release option for the compilation. This sets both the source version accepted by compiler and the system API that may be used by the code in the source file.

? If --enable-preview is specified, the --source N arguments can be omitted. If the Java runtime version is N, then --release N is implied when compiling source files.

? If a module-info.java file exists in the <source-root> directory, its module declaration is used to define a named module that will contain all the classes compiled from .java files in the source tree. If module-info.java does not exist, all the classes compiled from source files will be compiled in the context of the unnamed module.

? The source file that is launched should contain one or more top-level classes, the first of which is taken as the class to be executed.

? For the source file that is launched, the compiler does not enforce the optional restriction defined at the end of JLS 7.6, that a type in a named package should exist in a file whose name is composed from the type name followed by the .java extension.

? If a source file contains errors, appropriate error messages are written to the standard error stream, and the launcher exits with a non-zero exit code.

In source-file mode, execution proceeds as follows:

? The class to be executed is the first top-level class found in the source file. It must contain a declaration of an entry main method.

? The compiled classes are loaded by a custom class loader, that delegates to the application class loader. This implies that classes appearing on the application class path cannot refer to any classes declared in source files.

? If a module-info.java file exists in the <source-root> directory, then all the classes compiled from .java files in the source tree will be in that module, which will serve as the root module for the execution of the program. If module-info.java does not exist, the compiled classes are executed in the context of an unnamed module, as though --add-modules=ALL-DEFAULT is in effect. This is in addition to any other --add-module options that may have been specified on the command line.

? Any arguments appearing after the name of the file on the command line are passed to the main method in the obvious way.

? It is an error if there is a class on the application class path whose name is the same as that of the class to be executed.

See JEP 458: Launch Multi-File Source-Code Programs [<https://openjdk.org/jeps/458>] for complete details.

USING THE JDK_JAVA_OPTIONS LAUNCHER ENVIRONMENT VARIABLE

JDK_JAVA_OPTIONS prepends its content to the options parsed from the command line. The content of the JDK_JAVA_OPTIONS environment variable is a list of arguments separated by white-space characters (as determined by `isspace()`). These are prepended to the command line arguments passed to java launcher. The encoding requirement for the environment variable is the same as the java command line on the system. JDK_JAVA_OPTIONS environment variable content is treated in the same manner as that specified in the command line.

Single (') or double (") quotes can be used to enclose arguments that contain whitespace

characters. All content between the open quote and the first matching close quote are preserved by simply removing the pair of quotes. In case a matching quote is not found, the launcher will abort with an error message. @-files are supported as they are specified in the command line. Any wildcard literal * in the JDK_JAVA_OPTIONS environment variable content isn't expanded and is passed as-is to the starting VM. In order to mitigate potential misuse of JDK_JAVA_OPTIONS behavior, options that specify the main class (such as -jar) or cause the java launcher to exit without executing the main class (such as -h) are disallowed in the environment variable. If any of these options appear in the environment variable, the launcher will abort with an error message. When JDK_JAVA_OPTIONS is set, the launcher prints a message to stderr as a reminder.

Example:

```
$ export JDK_JAVA_OPTIONS='-g @file1 -Dprop=value @file2 -Dws.prop="white spaces"'
```

```
$ java -Xint @file3
```

is equivalent to the command line:

```
java -g @file1 -Dprop=value @file2 -Dws.prop="white spaces" -Xint @file3
```

OVERVIEW OF JAVA OPTIONS

The java command supports a wide range of options in the following categories:

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

? Extra Options for Java: General purpose options that are specific to the Java HotSpot Virtual Machine. They aren't guaranteed to be supported by all JVM implementations, and are subject to change. These options start with -X.

The advanced options aren't 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. Several examples of performance tuning are provided in Performance Tuning Examples. These options aren't guaranteed to be supported by all JVM implementations and are subject to change. Advanced options start with -XX.

? Advanced Runtime Options for Java: Control the runtime behavior of the Java HotSpot VM.

? Advanced JIT Compiler Options for java: Control the dynamic just-in-time (JIT) compilation performed by the Java HotSpot VM.

? Advanced Serviceability Options for Java: Enable gathering system information and perform?

ing extensive debugging.

? Advanced Garbage Collection Options for Java: Control how garbage collection (GC) is performed by the Java HotSpot

Boolean options are used to either enable a feature that's disabled by default or disable a feature that's enabled by default. Such options don't require a parameter. Boolean `-XX op?` 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're expected to specify the size in bytes, then you can use no suffix, or use the suffix `k` or `K` for kilobytes (KB), `m` or `M` for megabytes (MB), or `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, then use a number from 0 to 1. For example, specify `0.25` for 25%.

The following sections describe the options that are deprecated, obsolete, and removed:

? Deprecated Java Options: Accepted and acted upon --- a warning is issued when they're used.

? Obsolete Java Options: Accepted but ignored --- a warning is issued when they're used.

? Removed Java Options: Removed --- using them results in an error.

STANDARD OPTIONS FOR JAVA

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

Note: To specify an argument for a long option, you can use either `--name=value` or `--name value`.

`-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` using the platform specific naming conventions and locations:

? Linux and other POSIX-like platforms: The JVM attempts to load the library named `libfoo.so` in the location specified by the `LD_LIBRARY_PATH` system variable.

? macOS: The JVM attempts to load the library named `libfoo.dylib` in the location specified by the `DYLD_LIBRARY_PATH` system variable.

? Windows: The JVM attempts to load the library named `foo.dll` in the location speci?

fied by the PATH system variable.

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
```

```
-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.

```
--class-path classpath, -classpath classpath, or -cp classpath
```

Specifies a list of directories, JAR files, and ZIP archives to search for class files.

On Windows, semicolons (;) separate entities in this list; on other platforms it is a colon (:).

Specifying classpath overrides any setting of the CLASSPATH environment variable. If the class path option isn't used and classpath isn't set, then the user class path consists of the current directory (.).

As a special convenience, a class path element that contains a base name of an asterisk

isk (*) is considered equivalent to specifying a list of all the files in the direc?

tory with the extension .jar or .JAR. A Java program can't tell the difference be?

tween the two invocations. For example, if the directory mydir contains a.jar and

b.JAR, then the class path element mydir/* is expanded to A.jar:b.JAR, except that

the order of JAR files is unspecified. All .jar files in the specified directory,

even hidden ones, are included in the list. A class path entry consisting of an as?

terisk (*) expands to a list of all the jar files in the current directory. The

CLASSPATH environment variable, where defined, is similarly expanded. Any class path

wildcard expansion that occurs before the Java VM is started. Java programs never

see wildcards that aren't expanded except by querying the environment, such as by

calling `System.getenv("CLASSPATH")`.

```
--disable-@files
```

Can be used anywhere on the command line, including in an argument file, to prevent further @filename expansion. This option stops expanding @-argfiles after the option.

```
--enable-preview
```


Allows classes to depend on preview features [<https://docs.oracle.com/en/java/javase/12/language/index.html#JS?>

LAN-GUID-5A82FE0E-0CA4-4F1F-B075-564874FE2823] of the release.

`--enable-native-access module[,module...]`

Native access involves access to code or data outside the Java runtime. This is generally unsafe and, if done incorrectly, might crash the JVM or result in memory corruption. Native access can occur as a result of calling a method that is either restricted [<https://openjdk.org/jeps/454#Safety>], or native. This option allows code in the specified modules to perform native access. Native access occurring in a module that has not been explicitly enabled is deemed illegal.

module can be a module name, or ALL-UNNAMED to indicate code on the class path.

`---illegal-native-access=parameter`

This option specifies a mode for how illegal native access is handled:

Note: This option will be removed in a future release.

? allow: This mode allows illegal native access in all modules, without any warnings.

? warn: This mode is identical to allow except that a warning message is issued for the first illegal native access found in a module. This mode is the default for the current JDK but will change in a future release.

? deny: This mode disables illegal native access. That is, any illegal native access causes an `IllegalCallerException`. This mode will become the default in a future release.

To verify that your application is ready for a future version of the JDK, run it with

`--illegal-native-access=deny` along with any necessary `--enable-native-access` options.

`--finalization=value`

Controls whether the JVM performs finalization of objects. Valid values are "enabled" and "disabled". Finalization is enabled by default, so the value "enabled" does nothing. The value "disabled" disables finalization, so that no finalizers are invoked.

`--module-path modulepath... or -p modulepath`

Specifies where to find application modules with a list of path elements. The elements of a module path can be a file path to a module or a directory containing modules. Each module is either a modular JAR or an exploded-module directory.

On Windows, semicolons (;) separate path elements in this list; on other platforms it

is a colon (:).

`--upgrade-module-path modulepath...`

Specifies where to find module replacements of upgradeable modules in the runtime image with a list of path elements. The elements of a module path can be a file path to a module or a directory containing modules. Each module is either a modular JAR or an exploded-module directory.

On Windows, semicolons (;) separate path elements in this list; on other platforms it is a colon (:).

`--add-modules module[,module...]`

Specifies the root modules to resolve in addition to the initial module. `module` can also be `ALL-DEFAULT`, `ALL-SYSTEM`, and `ALL-MODULE-PATH`.

`--list-modules`

Lists the observable modules and then exits.

`-d module_name` or `--describe-module module_name`

Describes a specified module and then exits.

`--dry-run`

Creates the VM but doesn't execute the main method. This `--dry-run` option might be useful for validating the command-line options such as the module system configuration.

`--validate-modules`

Validates all modules and exit. This option is helpful for finding conflicts and other errors with modules on the module path.

`-Dproperty=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]]` or `-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 don't have a class loader). There's one exception to this rule: If the option is provided with no arguments, then it doesn't 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 the package `com.wombat.fruitbat` (and any subpackages) but disabled in the class `com.wombat.fruitbat.Brickbat`, use the following command:

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

`-disablesystemassertions` or `-dsa`

Disables assertions in all system classes.

`-enableassertions[:[packagename]...[:classname]]` or `-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 don't have a class loader). There's one exception to this rule: If the option is provided with no arguments, then it doesn't apply to system classes. This makes it easy to enable assertions in all classes except for system classes. The `-enableassertions` 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're processed in order, before loading any classes. For example, to run the `MyClass` application with assertions enabled only in the package `com.wombat.fruitbat` (and any subpackages) but disabled in the class

com.wombat.fruitbat.Brickbat, use the following command:

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

-enablessystemassertions or -esa

Enables assertions in all system classes.

-help, -h, or -?

Prints the help message to the error stream.

--help Prints the help message to the output stream.

-javaagent:jarpath[=options]

Loads the specified Java programming language agent. See `java.lang.instrument`.

--show-version

Prints the product version to the output stream and continues.

-showversion

Prints the product version to the error stream and continues.

--show-module-resolution

Shows module resolution output during startup.

-splash:imagepath

Shows the splash screen with the image specified by `imagepath`. HiDPI scaled images are automatically supported and used if available. The unscaled image file name, such as `image.ext`, should always be passed as the argument to the `-splash` option.

The most appropriate scaled image provided is picked up automatically.

For example, to show the `splash.gif` file from the `images` directory when starting your application, use the following option:

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

See the `SplashScreen` API documentation for more information.

-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.

-verbose:module

Displays information about the modules in use.

--version

Prints product version to the output stream and exits.

-version

Prints product version to the error stream and exits.

-X Prints the help on extra options to the error stream.

--help-extra

Prints the help on extra options to the output stream.

@argfile

Specifies one or more argument files prefixed by @ used by the java command. It isn't uncommon for the java command line to be very long because of the .jar files needed in the classpath. The @argfile option overcomes command-line length limitations by enabling the launcher to expand the contents of argument files after shell expansion, but before argument processing. Contents in the argument files are expanded because otherwise, they would be specified on the command line until the --disable-@files option was encountered.

The argument files can also contain the main class name and all options. If an argument file contains all of the options required by the java command, then the command line could simply be:

```
java @argfile
```

See java Command-Line Argument Files for a description and examples of using @-argfiles.

EXTRA OPTIONS FOR JAVA

The following java options are general purpose options that are specific to the Java HotSpot Virtual Machine.

-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/a:directories|zip|JAR-files

Specifies a list of directories, JAR files, and ZIP archives to append to the end of the default bootstrap class path.

On Windows, semicolons (;) separate entities in this list; on other platforms it is a colon (:).

-Xcheck:jni

Performs additional checks for Java Native Interface (JNI) functions.

The following checks are considered indicative of significant problems with the native code, and the JVM terminates with an irrecoverable error in such cases:

- ? The thread doing the call is not attached to the JVM.
- ? The thread doing the call is using the JNIEnv belonging to another thread.
- ? A parameter validation check fails:
 - ? A jfieldID, or jmethodID, is detected as being invalid. For example:
 - ? Of the wrong type
 - ? Associated with the wrong class
 - ? A parameter of the wrong type is detected.
 - ? An invalid parameter value is detected. For example:
 - ? NULL where not permitted
 - ? An out-of-bounds array index, or frame capacity
 - ? A non-UTF-8 string
 - ? An invalid JNI reference
 - ? An attempt to use a ReleaseXXX function on a parameter not produced by the corresponding GetXXX function

The following checks only result in warnings being printed:

- ? A JNI call was made without checking for a pending exception from a previous JNI call, and the current call is not safe when an exception may be pending.
- ? A class descriptor is in decorated format (Lname;) when it should not be.
- ? A NULL parameter is allowed, but its use is questionable.
- ? Calling other JNI functions in the scope of Get/ReleasePrimitiveArrayCritical or Get/ReleaseStringCritical

Expect a performance degradation when this option is used.

-Xcomp Testing mode to exercise JIT compilers. This option should not be used in production environments.

-Xdebug

Does nothing; deprecated for removal in a future release.

-Xdiag Shows additional diagnostic messages.

-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 aren't present in this mode.

-Xinternalversion

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

-Xlog:option

Configure or enable logging with the Java Virtual Machine (JVM) unified logging framework. See [Enable Logging with the JVM Unified Logging Framework](#).

-Xmixed

Executes all bytecode by the interpreter except for hot methods, which are compiled to native code. On by default. Use -Xint to switch off.

-Xmn size

Sets the initial and maximum size (in bytes) of the heap for the young generation (nursery) in the generational collectors. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or 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 are performed. If the size is too large, then only full garbage collections are performed, which can take a long time to complete. It is recommended that you do not set the size for the young generation for the G1 collector, and keep the size for the young generation greater than 25% and less than 50% of the overall heap size for other collectors. 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 minimum and 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, or 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.

Note that the `-XX:InitialHeapSize` option can also be used to set the initial heap size. If it appears after `-Xms` on the command line, then the initial heap size gets set to the value specified with `-XX:InitialHeapSize`.

`-Xmx` size

Specifies the maximum size (in bytes) of the heap. 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, or 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. 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 are left untouched during GC and are always be considered live. This can result in more memory being permanently occupied which, if not used carefully, throws an out-of-memory exception.

`-Xrs` Reduces the use of operating system signals by the JVM. Shutdown hooks enable the orderly shutdown of a Java application by running user cleanup code (such as closing database connections) at shutdown, even if the JVM terminates abruptly.

? Non-Windows:

? The JVM catches signals to implement shutdown hooks for unexpected termination.

The JVM uses `SIGHUP`, `SIGINT`, and `SIGTERM` to initiate the running of shutdown hooks.

? Applications embedding the JVM frequently need to trap signals such as `SIGINT` or `SIGTERM`, which can lead to interference with the JVM signal handlers. The `-Xrs` option is available to address this issue. When `-Xrs` is used, the signal masks for `SIGINT`, `SIGTERM`, `SIGHUP`, and `SIGQUIT` aren't changed by the JVM, and signal handlers for these signals aren't installed.

? Windows:

? 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 shouldn't initiate shutdown because the operating system doesn't 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 doesn't install a console control handler, implying that it doesn't watch for or process `CTRL_C_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, or `CTRL_SHUTDOWN_EVENT`.

There are two consequences of specifying `-Xrs`:

? Non-Windows: `SIGQUIT` thread dumps aren't available.

? Windows: `Ctrl + Break` thread dumps aren't 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 shared class data if possible (default).

on Require using shared class data, otherwise fail.

Note: The -Xshare:on option is used for testing purposes only. It may cause the VM to unexpectedly exit during start-up when the CDS archive cannot be used (for example, when certain VM parameters are changed, or when a different JDK is used). This option should not be used in production environments.

off Do not attempt to use shared class data.

-XshowSettings

Shows all settings and then continues.

-XshowSettings:category

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

all Shows all categories of settings in verbose detail.

locale Shows settings related to locale.

properties

Shows settings related to system properties.

security

Shows all settings related to security.

sub-category arguments for security include the following:

? security:all : shows all security settings

? security:properties : shows security properties

? security:providers : shows static security provider settings

? security:tls : shows TLS related security settings

vm Shows the settings of the JVM.

system Linux only: Shows host system or container configuration and continues.

-Xss size

Sets the thread stack size (in bytes). Append the letter k or K to indicate KB, m or M to indicate MB, or g or G to indicate GB. The actual size may be rounded up to a multiple of the system page size as required by the operating system. The default value depends on the platform. For example:

? Linux/x64: 1024 KB

? Linux/Aarch64: 2048 KB

? macOS/x64: 1024 KB

? macOS/Aarch64: 2048 KB

? Windows: 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 similar to -XX:ThreadStackSize.

--add-reads module=target-module(,target-module)*

Updates module to read the target-module, regardless of the module declaration. tar?

get-module can be ALL-UNNAMED to read all unnamed modules.

--add-exports module/package=target-module(,target-module)*

Updates module to export package to target-module, regardless of module declaration.

target-module can be ALL-UNNAMED to export to all unnamed modules.

--add-opens module/package=target-module(,target-module)*

Updates module to open package to target-module, regardless of module declaration.

--limit-modules module[,module...]

Specifies the limit of the universe of observable modules.

--patch-module module=file(;file)*

Overrides or augments a module with classes and resources in JAR files or directories.

ries.

--source version

Sets the version of the source in source-file mode.

--sun-misc-unsafe-memory-access= value

Allow or deny usage of unsupported API sun.misc.Unsafe. value is one of:

allow Allow use of the memory-access methods with no warnings at run time.

warn Allow use of the memory-access methods, but issues a warning on the first occasion

that any memory-access method is used. At most one warning is issued.

debug Allow use of the memory-access methods, but issue a one-line warning and a stack trace when any memory-access method is used.

deny Disallow use of the memory-access methods by throwing an UnsupportedOperationException on every usage.

The default value when the option is not specified is warn.

EXTRA OPTIONS FOR MACOS

The following extra options are macOS specific.

-XstartOnFirstThread

Runs the main() method on the first (AppKit) thread.

-Xdock:name=application_name

Overrides the default application name displayed in dock.

-Xdock:icon=path_to_icon_file

Overrides the default icon displayed in dock.

ADVANCED OPTIONS FOR JAVA

These java options can be used to enable other advanced options.

-XX:+UnlockDiagnosticVMOptions

Unlocks the options intended for diagnosing the JVM. By default, this option is disabled and diagnostic options aren't available.

Command line options that are enabled with the use of this option are not supported.

If you encounter issues while using any of these options, it is very likely that you will be required to reproduce the problem without using any of these unsupported options before Oracle Support can assist with an investigation. It is also possible that any of these options may be removed or their behavior changed without any warning.

-XX:+UnlockExperimentalVMOptions

Unlocks the options that provide experimental features in the JVM. By default, this option is disabled and experimental features aren't available.

ADVANCED RUNTIME OPTIONS FOR JAVA

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

-XX:ActiveProcessorCount=x

Overrides the number of CPUs that the VM will use to calculate the size of thread pools it will use 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:AllocateHeapAt=path

Takes a path to the file system and uses memory mapping to allocate the object heap on the memory device. Using this option enables the HotSpot VM to allocate the Java

object heap on an alternative memory device, such as an NV-DIMM, specified by the user.

Alternative memory devices that have the same semantics as DRAM, including the semantics of atomic operations, can be used instead of DRAM for the object heap without changing the existing application code. All other memory structures (such as the code heap, metaspace, and thread stacks) continue to reside in DRAM.

Some operating systems expose non-DRAM memory through the file system. Memory-mapped files in these file systems bypass the page cache and provide a direct mapping of virtual memory to the physical memory on the device. The existing heap related flags (such as -Xmx and -Xms) and garbage-collection related flags continue to work as before.

-XX:-CompactStrings

Disables the Compact Strings feature. By default, this option is enabled. When this option is enabled, Java Strings containing only single-byte characters are internally represented and stored as single-byte-per-character Strings using ISO-8859-1 / Latin-1 encoding. This reduces, by 50%, the amount of space required for Strings containing only single-byte characters. For Java Strings containing at least one multibyte character: these are represented and stored as 2 bytes per character using UTF-16 encoding. Disabling the Compact Strings feature forces the use of UTF-16 encoding as the internal representation for all Java Strings.

Cases where it may be beneficial to disable Compact Strings include the following:

- ? When it's known that an application overwhelmingly will be allocating multibyte character Strings

- ? In the unexpected event where a performance regression is observed in migrating from Java SE 8 to Java SE 9 and an analysis shows that Compact Strings introduces the regression

In both of these scenarios, disabling Compact Strings makes sense.

-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_pidpid.log` where `pid` is the identifier of the process that encountered 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`

? Non-Windows: The following example shows how to set the error log to `/var/log/java/java_error.log`:

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

? Windows: 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 exists, and is writeable, then it will be overwritten. Otherwise, if the file can't 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:

? Non-Windows: The temporary directory is `/tmp`.

? Windows: The temporary directory is specified by the value of the `TMP` environment variable; if that environment variable isn't defined, then the value of the `TEMP` environment variable is used.

`-XX:+ExtensiveErrorReports`

Enables the reporting of more extensive error information in the `ErrorFile`. This option can be turned on in environments where maximal information is desired - even if the resulting logs may be quite large and/or contain information that might be considered sensitive. The information can vary from release to release, and across different platforms. By default this option is disabled.

`-XX:FlightRecorderOptions=parameter=value (or) -XX:FlightRecorderOptions:parameter=value`

Sets the parameters that control the behavior of JFR. Multiple parameters can be specified by separating them with a comma.

The following list contains the available JFR `parameter=value` entries:

`globalbuffersize=size`

Specifies the total amount of primary memory used for data retention. The default value is based on the value specified for `memorysize`. Change the `memorysize` parameter to alter the size of global buffers.

`maxchunksize=size`

Specifies the maximum size (in bytes) of the data chunks in a recording. Append `m` or `M` to specify the size in megabytes (MB), or `g` or `G` to specify the

size in gigabytes (GB). By default, the maximum size of data chunks is set to 12 MB. The minimum allowed is 1 MB.

`memorysize=size`

Determines how much buffer memory should be used, and sets the `globalbuffer?` size and `numglobalbuffers` parameters based on the size specified. Append `m` or `M` to specify the size in megabytes (MB), or `g` or `G` to specify the size in gigabytes (GB). By default, the memory size is set to 10 MB.

`numglobalbuffers`

Specifies the number of global buffers used. The default value is based on the memory size specified. Change the `memorysize` parameter to alter the number of global buffers.

`old-object-queue-size=number-of-objects`

Maximum number of old objects to track. By default, the number of objects is set to 256.

`preserve-repository={true|false}`

Specifies whether files stored in the disk repository should be kept after the JVM has exited. If `false`, files are deleted. By default, this parameter is disabled.

`repository=path`

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

`retransform={true|false}`

Specifies whether event classes should be retransformed using JVMTI. If `false`, instrumentation is added when event classes are loaded. By default, this parameter is enabled.

`stackdepth=depth`

Stack depth for stack traces. By default, the depth is set to 64 method calls. The maximum is 2048. Values greater than 64 could create significant overhead and reduce performance.

`threadbuffersize=size`

Specifies the per-thread local buffer size (in bytes). By default, the local buffer size is set to 8 kilobytes, with a minimum value of 4 kilobytes. Overriding this parameter could reduce performance and is not recommended.

`-XX:LargePageSizeInBytes=size`

Sets the maximum large page size (in bytes) used by the JVM. The size argument must be a valid page size supported by the environment to have any effect. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or g or G to indicate gigabytes. By default, the size is set to 0, meaning that the JVM will use the default large page size for the environment as the maximum size for large pages. See Large Pages.

The following example describes how to set the large page size to 1 gigabyte (GB):

`-XX:LargePageSizeInBytes=1g`

`-XX:MaxDirectMemorySize=size`

Sets the maximum total size (in bytes) of the java.nio package, direct-buffer allocations. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or g or G to indicate gigabytes. If not set, the flag is ignored and 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:-MaxFDLimit`

Disables the attempt to set the soft limit for the number of open file descriptors to the hard limit. By default, this option is enabled on all platforms, but is ignored on Windows. The only time that you may need to disable this is on macOS, where its use imposes a maximum of 10240, which is lower than the actual system maximum.

`-XX:NativeMemoryTracking=mode`

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

`off` Instructs not to track JVM native memory usage. This is the default behavior if you don't specify the `-XX:NativeMemoryTracking` option.

summary

Tracks memory usage only 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:TrimNativeHeapInterval=millis`

Interval, in ms, at which the JVM will trim the native heap. Lower values will reclaim memory more eagerly at the cost of higher overhead. A value of 0 (default) disables native heap trimming. Native heap trimming is performed in a dedicated thread.

This option is only supported on Linux with GNU C Library (glibc).

`-XX:+NeverActAsServerClassMachine`

Enable the "Client VM emulation" mode which only uses the C1 JIT compiler, a 32Mb CodeCache and the Serial GC. The maximum amount of memory that the JVM may use (controlled by the `-XX:MaxRAM=n` flag) is set to 1GB by default. The string "emulated-client" is added to the JVM version string.

By default the flag is set to true only on Windows in 32-bit mode and false in all other cases.

The "Client VM emulation" mode will not be enabled if any of the following flags are used on the command line:

`-XX:{+|-}TieredCompilation`

`-XX:CompilationMode=mode`

`-XX:TieredStopAtLevel=n`

`-XX:{+|-}EnableJVMCI`

`-XX:{+|-}UseJVMCICompiler`

`-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 2, 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:

$4\text{GB} * \text{ObjectAlignmentInBytes}$

Note: As the alignment value increases, the unused space between objects also increases. 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.

? Non-Windows: The following example shows how the `-XX:OnError` option can be used to run the `gcore` command to create a core image, and start the `gdb` debugger to attach to the process in case of an irrecoverable error (the `%p` designates the current process identifier):

```
-XX:OnError="gcore %p;gdb -p %p"
```

? Windows: 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 identifier). This example assumes that the path to the `userdump.exe` utility is specified in the `PATH` environment variable:

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

`-XX:OnOutOfMemoryError=string`

Sets a custom command or a series of semicolon-separated commands to run when an `OutOfMemoryError` exception is first thrown by the JVM. 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. This applies only to `OutOfMemoryError` exceptions caused by Java Heap exhaustion; it does not apply to `OutOfMemoryError` exceptions thrown directly from Java code, nor by the JVM for other types of resource exhaustion (such as native thread creation errors).

`-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 aren't printed.

`-XX:+PreserveFramePointer`

Selects between using the `RBP` register as a general purpose register (`-XX:-PreserveFramePointer`) and using the `RBP` register to hold the frame pointer of the currently executing method (`-XX:+PreserveFramePointer`). If the frame pointer is available, then external profiling tools (for example, Linux `perf`) can construct more accurate stack traces.

`-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 isn't printed.

`-XX:SharedArchiveFile=path`

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

See Application Class Data Sharing.

`-XX:+VerifySharedSpaces`

If this option is specified, the JVM will load a CDS archive file only if it passes an integrity check based on CRC32 checksums. The purpose of this flag is to check for unintentional damage to CDS archive files in transmission or storage. To guarantee the security and proper operation of CDS, the user must ensure that the CDS archive files used by Java applications cannot be modified without proper authorization.

`-XX:SharedArchiveConfigFile=shared_config_file`

Specifies additional shared data added to the archive file.

`-XX:SharedClassListFile=file_name`

Specifies the text file that contains the names of the classes to store in the class data sharing (CDS) archive. This file contains the full name of one class 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 classes 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.

See Application Class Data Sharing.

`-XX:+ShowCodeDetailsInExceptionMessages`

Enables printing of improved `NullPointerException` messages. When an application throws a `NullPointerException`, the option enables the JVM to analyze the program's bytecode instructions to determine precisely which reference is null, and describes the source with a null-detail message. The null-detail message is calculated and returned by `NullPointerException.getMessage()`, and will be printed as the exception

message along with the method, filename, and line number. By default, this option is enabled.

`-XX:+ShowMessageBoxOnError`

Enables the display 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 option is equivalent to the `JFR.start` diagnostic command that starts a recording during runtime. `-XX:StartFlightRecording:help` prints available options and example command lines. You can set the following `parameter=value` entries when starting a JFR recording:

`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's no delay, and this parameter is set to 0.

`disk={true|false}`

Specifies whether to write data to disk while recording. By default, this parameter is enabled.

`dumponexit={true|false}`

Specifies if the running recording is dumped when the JVM shuts down. If enabled and a filename is not entered, the recording is written to a file in the directory where the process was started. The file name is a system-generated name that contains the process ID, recording ID, and current timestamp, similar to `hotspot-pid-47496-id-1-2018_01_25_19_10_41.jfr`. By default, this parameter is disabled.

`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 isn't limited, and this parameter is set to 0.

`filename=path`

Specifies the path and name of the file to which the recording is written when the recording is stopped, for example:

? recording.jfr

? /home/user/recordings/recording.jfr

? c:\recordings\recording.jfr

If %p and/or %t is specified in the filename, it expands to the JVM's PID and the current timestamp, respectively. The filename may also be a directory in which case, the filename is generated from the PID and the current date in the specified directory.

name=identifier

Takes both the name and the identifier of a recording.

maxage=time

Specifies the maximum age of disk data to keep for the recording. This parameter is valid only when the disk parameter is set to true. 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 isn't limited, and this parameter is set to 0s.

maxsize=size

Specifies the maximum size (in bytes) of disk data to keep for the recording. This parameter is valid only when the disk parameter is set to true. The value must not be less than the value for the maxchunksize parameter set with -XX:FlightRecorderOptions. Append m or M to specify the size in megabytes, or g or G to specify the size in gigabytes. By default, the maximum size of disk data isn't limited, and this parameter is set to 0.

path-to-gc-roots={true|false}

Specifies whether to collect the path to garbage collection (GC) roots at the end of a recording. By default, this parameter is disabled.

The path to GC roots is useful for finding memory leaks, but collecting it is time-consuming. Enable this option only when you start a recording for an application that you suspect has a memory leak. If the settings parameter is set to profile, the stack trace from where the potential leaking object was allocated is included in the information collected.

report-on-exit=identifier

Specifies the name of the view to display when the Java Virtual Machine (JVM) shuts down. To specify more than one view, use the report-on-exit parameter repeatedly. This option is not available if the disk option is set to false. For a list of available views, see `jfr help view`. By default, no report is generated.

`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/lib/jfr`. This default settings file collects a predefined set of information with low overhead, so it has minimal impact on performance and can be used with recordings that run continuously. A second settings file is also provided, `profile.jfc`, which provides more data than the default configuration, but can have more overhead and impact performance. Use this configuration for short periods of time when more information is needed.

You can specify values for multiple parameters by separating them with a comma. Event settings and .jfc options can be specified using the following syntax:

`option=value`

Specifies the option value to modify. To list available options, use the `JAVA_HOME/bin/jfr tool`.

`event-setting=value`

Specifies the event setting value to modify. Use the form: `<event-name>#<setting-name>=<value>`. To add a new event setting, prefix the event name with '+',

You can specify values for multiple event settings and .jfc options by separating them with a comma. In case of a conflict between a parameter and a .jfc option, the parameter will take precedence. The whitespace delimiter can be omitted for timespan values, i.e. `20ms`. For more information about the settings syntax, see Javadoc of the `jdk.jfr` package.

To only see warnings and errors from JFR during startup set `-Xlog:jfr+startup=warn?ing`.

`-XX:ThreadStackSize=size`

Sets the Java thread stack size (in kilobytes). Use of a scaling suffix, such as k,

results in the scaling of the kilobytes value so that `-XX:ThreadStackSize=1k` sets the Java thread stack size to 1024*1024 bytes or 1 megabyte. The default value depends on the platform. For example:

? Linux/x64: 1024 KB

? Linux/Aarch64: 2048 KB

? macOS/x64: 1024 KB

? macOS/Aarch64: 2048 KB

? Windows: The default value depends on virtual memory

The following examples show how to set the thread stack size to 1 megabyte in different units:

```
-XX:ThreadStackSize=1k
```

```
-XX:ThreadStackSize=1024
```

This option is similar to `-Xss`.

`-XX:+UseCompactObjectHeaders`

Enables compact object headers. By default, this option is disabled. Enabling this option reduces memory footprint in the Java heap by 4 bytes per object (on average) and often improves performance.

The feature remains disabled by default while it continues to be evaluated. In a future release it is expected to be enabled by default, and eventually will be the only mode of operation.

`-XX:-UseCompressedOops`

Disables the use of compressed pointers. By default, this option is enabled, and compressed pointers are used. This will automatically limit the maximum ergonomically determined Java heap size to the maximum amount of memory that can be covered by compressed pointers. By default this range is 32 GB.

With compressed oops 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 smaller than the compressed oops pointer range. This option works only for 64-bit JVMs.

It's possible to use compressed pointers with Java heap sizes greater than 32 GB.

See the `-XX:ObjectAlignmentInBytes` option.

`-XX:-UseContainerSupport`

Linux only: The VM now provides automatic container detection support, which allows

the VM to determine the amount of memory and number of processors that are available to a Java process running in docker containers. It uses this information to allocate system resources. The default for this flag is true, and container support is enabled by default. It can be disabled with `-XX:-UseContainerSupport`.

Unified Logging is available to help to diagnose issues related to this support.

Use `-Xlog:os+container=trace` for maximum logging of container information. See [Unified Logging with the JVM Unified Logging Framework](#) for a description of using Unified Logging.

`-XX:+UseLargePages`

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

See [Large Pages](#).

`-XX:+UseTransparentHugePages`

Linux only: Enables the use of large pages that can dynamically grow or shrink. This option is disabled by default. You may encounter performance problems with transparent huge pages as the OS moves other pages around to create huge pages; this option is made available for experimentation.

`-XX:+AllowUserSignalHandlers`

Non-Windows: Enables installation of signal handlers by the application. By default, this option is disabled and the application isn't allowed to install signal handlers.

`-XX:VMOptionsFile=filename`

Allows user to specify VM options in a file, for example, `java -XX:VMOptions?File=/var/my_vm_options HelloWorld`.

`-XX:UseBranchProtection=mode`

Linux AArch64 only: Specifies the branch protection mode. All options other than none require the VM to have been built with branch protection enabled. In addition, for full protection, any native libraries provided by applications should be compiled with the same level of protection.

Possible mode arguments for this option include the following:

`none` Do not use branch protection. This is the default value.

`standard`

Enables all branch protection modes available on the current platform.

Enables protection against ROP based attacks. (AArch64 8.3+ only)

ADVANCED JIT COMPILER OPTIONS FOR JAVA

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

`-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`

`-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, or 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`

`-XX:AllocatePrefetchInstr=instruction`

Sets the prefetch instruction to prefetch ahead of the allocation pointer. 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`

`-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`

`-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`

`-XX:AllocatePrefetchStyle=style`

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

- 0 Don't generate prefetch instructions.
- 1 Execute prefetch instructions after each allocation. This is the default setting.
- 2 Use the thread-local allocation block (TLAB) watermark pointer to determine when prefetch instructions are executed.
- 3 Generate one prefetch instruction per cache line.

`-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 compiler threads is selected automatically depending on the number of CPUs and memory available for compiled code. The following example shows how to set the number of threads to 2:

`-XX:CICompilerCount=2`

`-XX:+UseDynamicNumberOfCompilerThreads`

Dynamically create compiler thread up to the limit specified by `-XX:CICompilerCount`. This option is enabled by default.

`-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's 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, then the command is 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 a 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 new line separator (`\n`). The following commands are available:

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

compileonly

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

dontinline

Prevents inlining of the specified method.

exclude

Excludes the specified method from compilation.

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

inline Attempts to inline the specified method.

log Excludes 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 Passes 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,BlockLayout?
ByFrequency
```

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

print Prints generated assembler code after compilation of the specified method.

quiet Instructs not to 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 is 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
```

If you're using commands for the JIT compiler to perform on methods, then see the `-XX:CompileCommand` option.

`-XX:CompilerDirectivesFile=file`

Adds directives from a file to the directives stack when a program starts. See `CompilerControl` [https://docs.oracle.com/en/java/javase/12/vm/compiler-control1.html#GUID-94AD8194-786A-4F19-BFFF-278F8E237F3A].

The `-XX:CompilerDirectivesFile` option has to be used together with the `-XX:UnlockDi?`

agnosticVMOptions option that unlocks diagnostic JVM options.

-XX:+CompilerDirectivesPrint

Prints the directives stack when the program starts or when a new directive is added.

The -XX:+CompilerDirectivesPrint option has to be used together with the -XX:Unlock?

DiagnosticVMOptions option that unlocks diagnostic JVM options.

-XX:CompileOnly=methods

Sets the list of methods (separated by commas) to which compilation should be restricted. Only the specified methods are compiled.

-XX:CompileOnly=method1,method2,...,methodN is an alias for:

-XX:CompileCommand=compileonly,method1

-XX:CompileCommand=compileonly,method2

...

-XX:CompileCommand=compileonly,methodN

-XX:CompileThresholdScaling=scale

Provides unified control of first compilation. This option controls when methods are first compiled for both the tiered and the nontiered modes of operation. The CompileThresholdScaling option has a floating point value between 0 and +Inf and scales the thresholds corresponding to the current mode of operation (both tiered and nontiered). Setting CompileThresholdScaling to a value less than 1.0 results in earlier compilation while values greater than 1.0 delay compilation. Setting CompileThresholdScaling to 0 is equivalent to disabling compilation.

-XX:+DoEscapeAnalysis

Enables the use of escape analysis. This option is enabled by default. To disable the use of escape analysis, specify -XX:-DoEscapeAnalysis.

-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, or g or G to indicate gigabytes. The default value depends on the platform. The initial code cache size shouldn't be 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 already compiled methods that may be inlined. This flag only applies to the C2 compiler. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or g or G to indicate gigabytes. The default value depends on the platform and on whether tiered compilation is enabled. In the following example it 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 isn't 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:FreqInlineSize=size`

Sets the maximum bytecode size (in bytes) of a hot method to be inlined. This flag only applies to the C2 compiler. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or g or G to indicate gigabytes. The default value depends on the platform. In the following example it is set to 325 bytes:

```
-XX:FreqInlineSize=325
```

`-XX:MaxInlineSize=size`

Sets the maximum bytecode size (in bytes) of a cold method to be inlined. This flag only applies to the C2 compiler. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or g or G to indicate gigabytes. By default, the maximum bytecode size is set to 35 bytes:

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

`-XX:C1MaxInlineSize=size`

Sets the maximum bytecode size (in bytes) of a cold method to be inlined. This flag only applies to the C1 compiler. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or g or G to indicate gigabytes. By default, the maximum

bytecode size is set to 35 bytes:

`-XX:MaxInlineSize=35`

`-XX:MaxTrivialSize=size`

Sets the maximum bytecode size (in bytes) of a trivial method to be inlined. This flag only applies to the C2 compiler. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or 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:C1MaxTrivialSize=size`

Sets the maximum bytecode size (in bytes) of a trivial method to be inlined. This flag only applies to the C1 compiler. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or 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:MaxNodeLimit=nodes`

Sets the maximum number of nodes to be used during single method compilation. By default the value depends on the features enabled. In the following example the maximum number of nodes is set to 100,000:

`-XX:MaxNodeLimit=100000`

`-XX:NonNMethodCodeHeapSize=size`

Sets the size in bytes of the code segment containing nonmethod code.

A nonmethod code segment containing nonmethod code, such as compiler buffers and the bytecode interpreter. This code type stays in the code cache forever. This flag is used only if `-XX:SegmentedCodeCache` is enabled.

`-XX:NonProfiledCodeHeapSize=size`

Sets the size in bytes of the code segment containing nonprofiled methods. This flag is used only if `-XX:SegmentedCodeCache` is enabled.

`-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`.

`-XX:+PrintAssembly`

Enables printing of assembly code for bytecoded and native methods by using the ex?

ternal hsdisk-`<arch>.so` or `.dll` library. For 64-bit VM on Windows, it's `hsdis-amd64.dll`. This lets you to see the generated code, which may help you to diagnose performance issues.

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

`-XX:ProfiledCodeHeapSize=size`

Sets the size in bytes of the code segment containing profiled methods. This flag is used only if `-XX:SegmentedCodeCache` is enabled.

`-XX:+PrintCompilation`

Enables verbose diagnostic output from the JVM by printing a message to the console every time a method is compiled. This lets you to see which methods actually get compiled. By default, this option is disabled and diagnostic output isn't printed. You can also log compilation activity to a file by using the `-XX:+LogCompilation` option.

`-XX:+PrintInlining`

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

By default, this option is disabled and inlining information isn't 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, or `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 shouldn't be less than the initial code cache size; see the option `-XX:InitialCodeCacheSize`.

`-XX:+SegmentedCodeCache`

Enables segmentation of the code cache, without which the code cache consists of one large segment. With `-XX:+SegmentedCodeCache`, separate segments will be used for non-method, profiled method, and non-profiled method code. The segments are not re-

sized at runtime. The advantages are better control of the memory footprint, reduced code fragmentation, and better CPU iTLB (instruction translation lookaside buffer) and instruction cache behavior due to improved locality.

The feature is enabled by default if tiered compilation is enabled (-XX:+TieredCompilation) and the reserved code cache size (-XX:ReservedCodeCacheSize) is at least 240 MB.

-XX:StartAggressiveSweepingAt=percent

Forces stack scanning of active methods to aggressively remove unused code when only the given percentage of the code cache is free. The default value is 10%.

-XX:-TieredCompilation

Disables the use of tiered compilation. By default, this option is enabled.

-XX:UseSSE=version

Enables the use of SSE instruction set of a specified version. Is set by default to the highest supported version available (x86 only).

-XX:UseAVX=version

Enables the use of AVX instruction set of a specified version. Is set by default to the highest supported version available (x86 only).

-XX:+UseAES

Enables hardware-based AES intrinsics for hardware that supports it. This option is on by default on hardware that has the necessary instructions. The -XX:+UseAES is used in conjunction with UseAESIntrinsics. Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UseAESIntrinsics

Enables AES intrinsics. Specifying -XX:+UseAESIntrinsics is equivalent to also enabling -XX:+UseAES. To disable hardware-based AES intrinsics, specify -XX:-UseAESIntrinsics. For example, to enable hardware AES, use the following flags:

-XX:+UseAES -XX:+UseAESIntrinsics

Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UseAESCTRIntrinsics

Analogous to -XX:+UseAESIntrinsics enables AES/CTR intrinsics.

-XX:+UseGHASHIntrinsics

Controls the use of GHASH intrinsics. Enabled by default on platforms that support the corresponding instructions. Flags that control intrinsics now require the option

-XX:+UnlockDiagnosticVMOptions.

-XX:+UseChaCha20Intrinsics

Enable ChaCha20 intrinsics. This option is on by default for supported platforms.

To disable ChaCha20 intrinsics, specify -XX:-UseChaCha20Intrinsics. Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UsePoly1305Intrinsics

Enable Poly1305 intrinsics. This option is on by default for supported platforms.

To disable Poly1305 intrinsics, specify -XX:-UsePoly1305Intrinsics. Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UseBASE64Intrinsics

Controls the use of accelerated BASE64 encoding routines for java.util.Base64. Enabled by default on platforms that support it. Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UseAdler32Intrinsics

Controls the use of Adler32 checksum algorithm intrinsic for java.util.zip.Adler32. Enabled by default on platforms that support it. Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UseCRC32Intrinsics

Controls the use of CRC32 intrinsics for java.util.zip.CRC32. Enabled by default on platforms that support it. Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UseCRC32CIntrinsics

Controls the use of CRC32C intrinsics for java.util.zip.CRC32C. Enabled by default on platforms that support it. Flags that control intrinsics now require the option -XX:+UnlockDiagnosticVMOptions.

-XX:+UseSHA

Enables hardware-based intrinsics for SHA crypto hash functions for some hardware.

The UseSHA option is used in conjunction with the UseSHA1Intrinsics, UseSHA256Intrinsics, and UseSHA512Intrinsics options.

The UseSHA and UseSHA*Intrinsics flags are enabled by default on machines that support the corresponding instructions.

This feature is applicable only when using the sun.security.provider.Sun provider for SHA operations. Flags that control intrinsics now require the option -XX:+UnlockDi?

agnosticVMOptions.

To disable all hardware-based SHA intrinsics, specify the `-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. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

`-XX:+UseSHA256Intrinsics`

Enables intrinsics for SHA-224 and SHA-256 crypto hash functions. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

`-XX:+UseSHA512Intrinsics`

Enables intrinsics for SHA-384 and SHA-512 crypto hash functions. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

`-XX:+UseMathExactIntrinsics`

Enables intrinsification of various `java.lang.Math.*Exact()` functions. Enabled by default. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

`-XX:+UseMultiplyToLenIntrinsic`

Enables intrinsification of `BigInteger.multiplyToLen()`. Enabled by default on platforms that support it. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

`-XX:+UseSquareToLenIntrinsic`

Enables intrinsification of `BigInteger.squareToLen()`. Enabled by default on platforms that support it. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

`-XX:+UseMulAddIntrinsic`

Enables intrinsification of `BigInteger.mulAdd()`. Enabled by default on platforms that support it. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

`-XX:+UseMontgomeryMultiplyIntrinsic`

Enables intrinsification of `BigInteger.montgomeryMultiply()`. Enabled by default on platforms that support it. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

-XX:+UseMontgomerySquareIntrinsic

Enables intrinsification of BigInteger.montgomerySquare(). Enabled by default on platforms that support it. Flags that control intrinsics now require the option

-XX:+UnlockDiagnosticVMOptions.

-XX:+UseCMoveUnconditionally

Generates CMove (scalar and vector) instructions regardless of profitability analysis.

-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 if the card is already marked before updating the card table. This option is disabled by default. It should be used only on machines with multiple sockets, where it increases the performance of Java applications that rely on concurrent operations.

-XX:+UseCountedLoopSafepoints

Keeps safepoints in counted loops. Its default value depends on whether the selected garbage collector requires low latency safepoints.

-XX:LoopStripMiningIter=number_of_iterations

Controls the number of iterations in the inner strip mined loop. Strip mining transforms counted loops into two level nested loops. Safepoints are kept in the outer loop while the inner loop can execute at full speed. This option controls the maximum number of iterations in the inner loop. The default value is 1,000.

-XX:LoopStripMiningIterShortLoop=number_of_iterations

Controls loop strip mining optimization. Loops with the number of iterations less than specified will not have safepoints in them. Default value is 1/10th of

-XX:LoopStripMiningIter.

-XX:+UseFMA

Enables hardware-based FMA intrinsics for hardware where FMA instructions are available (such as, Intel and ARM64). FMA intrinsics are generated for the java.lang.Math.fma(a, b, c) methods that calculate the value of (a * b + c) expressions.

`-XX:+UseSuperWord`

Enables the transformation of scalar operations into superword operations. Superword is a vectorization optimization. This option is enabled by default. To disable the transformation of scalar operations into superword operations, specify `-XX:-UseSuperWord`.

ADVANCED SERVICEABILITY OPTIONS FOR JAVA

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

`-XX:+DisableAttachMechanism`

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 diagnostics and troubleshooting tools such as `jcmd`, `jstack`, `jmap`, and `jinfo`.

Note: The tools such as `jcmd`, `jinfo`, `jmap`, and `jstack` shipped with the JDK aren't supported when using the tools from one JDK version to troubleshoot a different JDK version.

`-XX:+DTraceAllocProbes`

Linux and macOS: Enable `dtrace` tool probes for object allocation.

`-XX:+DTraceMethodProbes`

Linux and macOS: Enable `dtrace` tool probes for method-entry and method-exit.

`-XX:+DTraceMonitorProbes`

Linux and macOS: Enable `dtrace` tool probes for monitor events.

`-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 by the JVM. 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 isn't dumped when an `OutOfMemoryError` exception is thrown. This applies only to `OutOfMemoryError` exceptions caused by Java Heap exhaustion; it does not apply to `OutOfMemoryError` exceptions thrown directly from Java code, nor by the JVM for other types of resource exhaustion (such as native thread creation errors).

`-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's 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 identifier):

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

? Non-Windows: The following example shows how to set the heap dump file to `/var/log/java/java_heapdump.hprof`:

```
-XX:HeapDumpPath=/var/log/java/java_heapdump.hprof
```

? Windows: 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 to where log data is written. By default, the file is created in the current working directory, and it's named `hotspot.log`.

? Non-Windows: The following example shows how to set the log file to `/var/log/java/hotspot.log`:

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

? Windows: 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 one of the following events:

? Non-Windows: `Control+\` (SIGQUIT)

? Windows: `Control+C` (SIGTERM)

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 one of the following events:

? Non-Windows: `Control+\` (SIGQUIT)

? Windows: `Control+C` (SIGTERM)

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:+PrintFlagsRanges

Prints the range specified and allows automatic testing of the values. See Validate Java Virtual Machine Flag Arguments.

-XX:+PerfDataSaveToFile

If enabled, saves jstat binary data when the Java application exits. This binary data is saved in a file named hspcrfdata_pid, where pid is the process identifier of the Java application that you ran. Use the jstat command to display the performance data contained in this file as follows:

```
jstat -class file:///path/hspcrfdata_pid
```

```
jstat -gc file:///path/hspcrfdata_pid
```

-XX:+UsePerfData

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

ADVANCED GARBAGE COLLECTION OPTIONS FOR JAVA

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

-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 sizes are configured less aggressively.

-XX:+AlwaysPreTouch

Requests the VM to touch every page on the Java heap after requesting it from the operating system and before handing memory out to the application. By default, this option is disabled and all pages are committed as the application uses the heap space.

-XX:ConcGCThreads=threads

Sets the number of threads used for concurrent GC. Sets threads to approximately 1/4 of the number of parallel garbage collection threads. 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 the `System.gc()` method. This option is disabled by default, meaning that calls to `System.gc()` are processed. If processing of calls to `System.gc()` is disabled, then 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 with the `-XX:+UseG1GC` option.

`-XX:G1AdaptiveIHOPNumInitialSamples=number`

When `-XX:UseAdaptiveIHOP` is enabled, this option sets the number of completed marking cycles used to gather samples until G1 adaptively determines the optimum value of `-XX:InitiatingHeapOccupancyPercent`. Before, G1 uses the value of `-XX:InitiatingHeapOccupancyPercent` directly for this purpose. The default value is 3.

`-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 is a power of 2 and can range from 1 MB to 32 MB. The default region size is determined ergonomically based on the heap size with a goal of approximately 2048 regions.

The following example sets the size of the subdivisions to 16 MB:

`-XX:G1HeapRegionSize=16m`

`-XX:G1HeapWastePercent=percent`

Sets the percentage of heap that you're willing to waste. The Java HotSpot VM doesn't initiate the mixed garbage collection cycle when the reclaimable percentage is less than the heap waste percentage. The default is 5 percent.

`-XX:G1MaxNewSizePercent=percent`

Sets the percentage of the heap size to use as the maximum for the young generation size. The default value is 60 percent of your Java heap.

This is an experimental flag. This setting replaces the `-XX:DefaultMaxNewGenPercent` setting.

`-XX:G1MixedGCCountTarget=number`

Sets the target number of mixed garbage collections after a marking cycle to collect old regions with at most `G1MixedGCLiveThresholdPercent` live data. The default is 8

mixed garbage collections. The goal for mixed collections is to be within this target number.

`-XX:G1MixedGCLiveThresholdPercent=percent`

Sets the occupancy threshold for an old region to be included in a mixed garbage collection cycle. The default occupancy is 85 percent.

This is an experimental flag. This setting replaces the `-XX:G1OldCSetRegion?LiveThresholdPercent` setting.

`-XX:G1NewSizePercent=percent`

Sets the percentage of the heap to use as the minimum for the young generation size. The default value is 5 percent of your Java heap.

This is an experimental flag. This setting replaces the `-XX:DefaultMinNewGenPercent` setting.

`-XX:G1OldCSetRegionThresholdPercent=percent`

Sets an upper limit on the number of old regions to be collected during a mixed garbage collection cycle. The default is 10 percent of the Java heap.

`-XX:G1ReservePercent=percent`

Sets the percentage of the heap (0 to 50) that's reserved as a false ceiling to reduce the possibility of promotion failure for the G1 collector. When you increase or decrease the percentage, ensure that you adjust the total Java heap by the same amount. By default, this option is set to 10%.

The following example sets the reserved heap to 20%:

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

`-XX:+G1UseAdaptiveIHOP`

Controls adaptive calculation of the old generation occupancy to start background work preparing for an old generation collection. If enabled, G1 uses `-XX:InitiatingHeapOccupancyPercent` for the first few times as specified by the value of `-XX:G1AdaptiveIHOPNumInitialSamples`, and after that adaptively calculates a new optimum value for the initiating occupancy automatically. Otherwise, the old generation collection process always starts at the old generation occupancy determined by `-XX:InitiatingHeapOccupancyPercent`.

The default is enabled.

`-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, or g or G to indicate gigabytes.

The default value is selected at run time based on the system configuration.

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 is set as the sum of the sizes al?

located 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. Note that the -Xms op?

tion sets both the minimum and the initial heap size of the heap. If -Xms appears

after -XX:InitialHeapSize on the command line, then the initial heap size gets set to

the value specified with -Xms.

```
-XX:InitialRAMPercentage=percent
```

Sets the initial amount of memory that the JVM will use for the Java heap before ap?

plying ergonomics heuristics as a percentage of the maximum amount determined as de?

scribed 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 option). Adaptive sizing is enabled by default

with the throughput garbage collector by using the -XX:+UseParallelGC option, and the

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 sur?

vivor 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), then the initial size of the survivor space is 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 old generation occupancy (0 to 100) at which to start the first few concurrent marking cycles for the G1 garbage collector.

By default, the initiating value is set to 45%. A value of 0 implies nonstop concurrent GC cycles from the beginning until G1 adaptively sets this value.

See also the `-XX:G1UseAdaptiveIHOP` and `-XX:G1AdaptiveIHOPNumInitialSamples` options.

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. The specified value doesn't adapt to your heap size. By default, for G1 the maximum pause time target is 200 milliseconds. The other generational collectors do not use a pause time goal by default.

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 bytes) 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, or g or G to indicate gigabytes. The default value is selected at run time based on the system configuration. For server deployments, the options `-XX:InitialHeapSize` and `-XX:MaxHeapSize` are often set to the same value.

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 is shrunk. By default, this value is set to 70%.

Minimize the Java heap size by lowering the values of the parameters `MaxHeapFreeRatio` (default value is 70%) and `MinHeapFreeRatio` (default value is 40%) with the command-line options `-XX:MaxHeapFreeRatio` and `-XX:MinHeapFreeRatio`. Lowering `MaxHeapFreeRatio` to as low as 10% and `MinHeapFreeRatio` to 5% has successfully reduced the heap size without too much performance regression; however, results may vary greatly depending on your application. Try different values for these parameters until they're as low as possible yet still retain acceptable performance.

`-XX:MaxHeapFreeRatio=10 -XX:MinHeapFreeRatio=5`

Customers trying to keep the heap small should also add the option `-XX:-ShrinkHeapInSteps`. See Performance Tuning Examples for a description of using this option to keep the Java heap small by reducing the dynamic footprint for embedded applications.

`-XX:MaxMetaspaceSize=size`

Sets the maximum amount of native memory that can be allocated for class metadata. By default, the size isn't 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:MaxNewSize=size`

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

`-XX:MaxRAM=size`

Sets the maximum amount of memory that the JVM may use for the Java heap before applying ergonomics heuristics. The default value is the minimum amount of available memory to the JVM process or 128 GB, whichever is lower.

The maximum amount of available memory to the JVM process is the minimum of the ma-

chine's physical memory and any constraints set by the environment (e.g. container).

Specifying this option disables automatic use of compressed oops if the combined re?

sult 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 maximum amount of available memory for

sizing the Java heap to 2 GB:

```
-XX:MaxRAM=2G
```

```
-XX:MaxRAMPercentage=percent
```

Sets the maximum amount of memory that the JVM may use for the Java heap before ap?

plying ergonomics heuristics as a percentage of the maximum amount determined as de?

scribed in the -XX:MaxRAM option. The default value is 25 percent.

Specifying this option disables automatic use of compressed oops if the combined re?

sult 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:MinRAMPercentage=percent
```

Sets the maximum amount of memory that the JVM may use for the Java heap before ap?

plying ergonomics heuristics as a percentage of the maximum amount determined as de?

scribed in the -XX:MaxRAM option for small heaps. A small heap is a heap of approxi?

mately 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: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.

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 triggers a garbage collection the first time it's 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 is expanded. By default, this value is set to 40%.

Minimize Java heap size by lowering the values of the parameters `MaxHeapFreeRatio` (default value is 70%) and `MinHeapFreeRatio` (default value is 40%) with the command-line options `-XX:MaxHeapFreeRatio` and `-XX:MinHeapFreeRatio`. Lowering `MaxHeapFreeRatio` to as low as 10% and `MinHeapFreeRatio` to 5% has successfully reduced the heap size without too much performance regression; however, results may vary greatly depending on your application. Try different values for these parameters until they're as low as possible, yet still retain acceptable performance.

`-XX:MaxHeapFreeRatio=10 -XX:MinHeapFreeRatio=5`

Customers trying to keep the heap small should also add the option `-XX:-ShrinkHeapInSteps`. See [Performance Tuning Examples](#) for a description of using this option to keep the Java heap small by reducing the dynamic footprint for embedded applications.

`-XX:MinHeapSize=size`

Sets the minimum 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, or g or G to indicate gigabytes. The default value is selected at run time based on the system configuration.

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

`-XX:MinHeapSize=6291456`

`-XX:MinHeapSize=6144k`

`-XX:MinHeapSize=6m`

If you set this option to 0, then the minimum size is set to the same value as the initial size.

`-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-to-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, or 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 are performed. If the size is too high, then only full GCs are performed, which can take a long time to complete. It is recommended that you keep the size for the young generation greater than 25% and less than 50% of the overall heap size.

The following examples show how to set the initial size of the 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 the stop-the-world (STW) worker threads. The default value depends on the number of CPUs available to the JVM and the garbage collector selected. For example, to set the number of threads for G1 GC to 2, specify the following option:

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

```
-XX:+ParallelRefProcEnabled
```

Enables parallel reference processing. By default, collectors employing multiple threads perform parallel reference processing if the number of parallel threads to use is larger than one. The option is available only when the throughput or G1 garbage collector is used (-XX:+UseParallelGC or -XX:+UseG1GC). Other collectors employing multiple threads always perform reference processing in parallel.

```
-XX:+PrintAdaptiveSizePolicy
```

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

`-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:-ShrinkHeapInSteps`

Incrementally reduces the Java heap to the target size, specified by the option `-XX:MaxHeapFreeRatio`. This option is enabled by default. If disabled, then it immediately reduces the Java heap to the target size instead of requiring multiple garbage collection cycles. Disable this option if you want to minimize the Java heap size. You will likely encounter performance degradation when this option is disabled.

See Performance Tuning Examples for a description of using the `MaxHeapFreeRatio` option to keep the Java heap small by reducing the dynamic footprint for embedded applications.

`-XX:StringDeduplicationAgeThreshold=threshold`

Identifies String objects reaching the specified age that 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.

Note: 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, or `g` or `G` to indicate gigabytes. If this option is set to 0, then the JVM selects 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:+UseG1GC`

Enables the use of the garbage-first (G1) garbage collector. It's a server-style garbage collector, targeted for multiprocessor machines with a large amount of RAM. This option 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 (a stable and predictable pause time below 0.5 seconds). By default, this option is enabled and G1 is used as the default garbage collector.

`-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 the op?

tion `-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.

The default value for this option depends on the garbage collector.

`-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 default collector is used.

`-XX:+UseSerialGC`

Enables the use of the serial garbage collector. This is generally the best choice for small and simple applications that don't require any special functionality from garbage collection. By default, this option is disabled and the default collector is used.

`-XX:+UseStringDeduplication`

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

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 the option `-XX:-UseTLAB`.

`-XX:+UseZGC`

Enables the use of the Z garbage collector (ZGC). This is a low latency garbage collector, providing max pause times of a few milliseconds, at some throughput cost.

Pause times are independent of what heap size is used. Supports heap sizes from 8MB to 16TB.

`-XX:ZAllocationSpikeTolerance=factor`

Sets the allocation spike tolerance for ZGC. By default, this option is set to 2.0.

This factor describes the level of allocation spikes to expect. For example, using a

factor of 3.0 means the current allocation rate can be expected to triple at any time.

`-XX:ZCollectionInterval=seconds`

Sets the maximum interval (in seconds) between two GC cycles when using ZGC. By default, this option is set to 0 (disabled).

`-XX:ZFragmentationLimit=percent`

Sets the maximum acceptable heap fragmentation (in percent) for ZGC. By default, this option is set to 25. Using a lower value will cause the heap to be compacted more aggressively, to reclaim more memory at the cost of using more CPU time.

`-XX:+ZProactive`

Enables proactive GC cycles when using ZGC. By default, this option is enabled. ZGC will start a proactive GC cycle if doing so is expected to have minimal impact on the running application. This is useful if the application is mostly idle or allocates very few objects, but you still want to keep the heap size down and allow reference processing to happen even when there are a lot of free space on the heap.

`-XX:+ZUncommit`

Enables uncommitting of unused heap memory when using ZGC. By default, this option is enabled. Uncommitting unused heap memory will lower the memory footprint of the JVM, and make that memory available for other processes to use.

`-XX:ZUncommitDelay=seconds`

Sets the amount of time (in seconds) that heap memory must have been unused before being uncommitted. By default, this option is set to 300 (5 minutes). Committing and uncommitting memory are relatively expensive operations. Using a lower value will cause heap memory to be uncommitted earlier, at the risk of soon having to commit it again.

`-XX:+UseShenandoahGC`

Enables the use of the Shenandoah garbage collector. This is a low pause time, concurrent garbage collector. Its pause times are not proportional to the size of the heap. Shenandoah garbage collector can work with compressed pointers. See `-XX:UseCompressedOops` for further information about compressed pointers.

`-XX:ShenandoahGCMode=mode`

Sets the GC mode for Shenandoah GC to use. By default, this option is set to `satb`. Among other things, this defines which barriers are in use. Possible mode values in?

clude the following:

`satb` Snapshot-at-the-beginning concurrent GC (three pass mark-evac-update). It is a single generation GC.

`generational`

It is also a snapshot-at-the-beginning and concurrent GC, but it is generational. Please see JEP 404 [<https://openjdk.org/jeps/404>] and JEP 521 [<https://openjdk.org/jeps/521>] for its advantages and risks.

`-XX:ShenandoahGCHeuristics=heuristics`

Sets the heuristics for Shenandoah GC to use. By default, this option is set to `adaptive`. This fine-tunes the GC mode selected, by choosing when to start the GC, how much to process on each cycle, and what other features to automatically enable. When `-XX:ShenandoahGCMode` is `generational`, the only supported option is the default, `adaptive`.

Possible heuristics are the following:

`adaptive`

To maintain the given amount of free heap at all times, even during the GC cycle.

`static` Trigger GC when free heap falls below a specified threshold.

`compact`

Run GC more frequently and with deeper targets to free up more memory.

DEPRECATED JAVA OPTIONS

These java options are deprecated and might be removed in a future JDK release. They're still accepted and acted upon, but a warning is issued when they're used.

`-Xloggc:filename`

Sets the file to which verbose GC events information should be redirected for logging. The `-Xloggc` option overrides `-verbose:gc` if both are given with the same java command. `-Xloggc:filename` is replaced by `-Xlog:gc:filename`. See [Enable Logging with the JVM Unified Logging Framework](#).

Example:

`-Xlog:gc:garbage-collection.log`

`-XX:+FlightRecorder`

Enables the use of Java Flight Recorder (JFR) during the runtime of the application.

Since JDK 8u40 this option has not been required to use JFR.

OBSOLETE JAVA OPTIONS

These java options are still accepted but ignored, and a warning is issued when they're used.

`--illegal-access=parameter`

Controlled relaxed strong encapsulation, as defined in JEP 261 [<https://openjdk.org/jeps/261#Relaxed-strong-encapsulation>]. This option was deprecated in JDK 16 by JEP 396 [<https://openjdk.org/jeps/396>] and made obsolete in JDK 17 by JEP 403 [<https://openjdk.org/jeps/403>].

REMOVED JAVA OPTIONS

These java options have been removed in JDK 25 and using them results in an error of:

Unrecognized VM option option-name

`-XX:RTMAbortRatio=abort_ratio`

Specifies 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 is 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 is deoptimized if 50% of all transactions are aborted.

`-XX:RTMRetryCount=number_of_retries`

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

`-XX:+UseRTMDeopt`

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

`-XX:+UseRTMLocking`

Generates 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 available only 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, then 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 XTEST instruction checks if a set of instructions is 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 didn't 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 doesn't 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.

For the lists and descriptions of options removed in previous releases see the Removed Java Options section in:

? The java Command, Release 24 [https://docs.oracle.com/en/java/javase/24/docs/specs/man/java.html]

? The java Command, Release 23 [https://docs.oracle.com/en/java/javase/23/docs/specs/man/java.html]

? The java Command, Release 22 [https://docs.oracle.com/en/java/javase/22/docs/specs/man/java.html]

? The java Command, Release 21 [<https://docs.oracle.com/en/java/javase/21/docs/specs/man/java.html>]

? The java Command, Release 20 [<https://docs.oracle.com/en/java/javase/20/docs/specs/man/java.html>]

? The java Command, Release 19 [<https://docs.oracle.com/en/java/javase/19/docs/specs/man/java.html>]

? The java Command, Release 18 [<https://docs.oracle.com/en/java/javase/18/docs/specs/man/java.html>]

? The java Command, Release 17 [<https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html>]

? The java Command, Release 16 [<https://docs.oracle.com/en/java/javase/16/docs/specs/man/java.html>]

? The java Command, Release 15 [<https://docs.oracle.com/en/java/javase/15/docs/specs/man/java.html>]

? The java Command, Release 14 [<https://docs.oracle.com/en/java/javase/14/docs/specs/man/java.html>]

? The java Command, Release 13 [<https://docs.oracle.com/en/java/javase/13/docs/specs/man/java.html>]

? Java Platform, Standard Edition Tools Reference, Release 12 [<https://docs.oracle.com/en/java/javase/12/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE>]

? Java Platform, Standard Edition Tools Reference, Release 11 [<https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-741FC470-AA3E-494A-8D2B-1B1FE4A990D1>]

? Java Platform, Standard Edition Tools Reference, Release 10 [<https://docs.oracle.com/javase/10/tools/java.htm#JSWOR624>]

? Java Platform, Standard Edition Tools Reference, Release 9 [<https://docs.oracle.com/javase/9/tools/java.htm#JSWOR624>]

? Java Platform, Standard Edition Tools Reference, Release 8 for Oracle JDK on Windows [<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html#BGBCIEFC>]

? Java Platform, Standard Edition Tools Reference, Release 8 for Oracle JDK on Solaris, Linux, and macOS [<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html#BGBCIEFC>]

JAVA COMMAND-LINE ARGUMENT FILES

You can shorten or simplify the java command by using @ argument files to specify one or

more text files that contain arguments, such as options and class names, which are passed to the java command. This lets you to create java commands of any length on any operating system.

In the command line, use the at sign (@) prefix to identify an argument file that contains java options and class names. When the java command encounters a file beginning with the at sign (@), it expands the contents of that file into an argument list just as they would be specified on the command line.

The java launcher expands the argument file contents until it encounters the --disable-@files option. You can use the --disable-@files option anywhere on the command line, including in an argument file, to stop @ argument files expansion.

The following items describe the syntax of java argument files:

? The argument file must contain only ASCII characters or characters in system default encoding that's ASCII friendly, such as UTF-8.

? The argument file size must not exceed MAXINT (2,147,483,647) bytes.

? The launcher doesn't expand wildcards that are present within an argument file. That means an asterisk (*) is passed on as-is to the starting VM. For example *.java stays *.java and is not expanded to Foo.java Bar.java ..., as would happen with some command line shells.

? Use white space or new line characters to separate arguments included in the file.

? White space includes a white space character, \t, \n, \r, and \f.

For example, it is possible to have a path with a space, such as c:\Program Files that can be specified as either "c:\Program Files" or, to avoid an escape, c:\Program "Files.

? Any option that contains spaces, such as a path component, must be within quotation marks using quotation (") characters in its entirety.

? A string within quotation marks may contain the characters \n, \r, \t, and \f. They are converted to their respective ASCII codes.

? If a file name contains embedded spaces, then put the whole file name in double quotation marks.

? File names in an argument file are relative to the current directory, not to the location of the argument file.

? Use the number sign # in the argument file to identify comments. All characters following the # are ignored until the end of line.

? Additional at sign @ prefixes to @ prefixed options act as an escape, (the first @ is re?

moved and the rest of the arguments are presented to the launcher literally).

? Lines may be continued using the continuation character (\) at the end-of-line. The two lines are concatenated with the leading white spaces trimmed. To prevent trimming the leading white spaces, a continuation character (\) may be placed at the first column.

? Because backslash (\) is an escape character, a backslash character must be escaped with another backslash character.

? Partial quote is allowed and is closed by an end-of-file.

? An open quote stops at end-of-line unless \ is the last character, which then joins the next line by removing all leading white space characters.

? Use of the at sign (@) to recursively interpret files isn't supported.

Example of Open or Partial Quotes in an Argument File

In the argument file,

```
-cp "lib/  
cool/  
app/  
jars
```

this is interpreted as:

```
-cp lib/cool/app/jars
```

Example of a Backslash Character Escaped with Another Backslash Character in an Argument File

To output the following:

```
-cp c:\Program Files (x86)\Java\jre\lib\ext;c:\Program Files\Java\jre9\lib\ext
```

The backslash character must be specified in the argument file as:

```
-cp "c:\\Program Files (x86)\\Java\\jre\\lib\\ext;c:\\Program  
Files\\Java\\jre9\\lib\\ext"
```

Example of an EOL Escape Used to Force Concatenation of Lines in an Argument File

In the argument file,

```
-cp "lib/cool app/jars:\\  
lib/another app/jars"
```

This is interpreted as:

```
-cp /lib/cool app/jars:/lib/another app/jars
```

Example of Line Continuation with Leading Spaces in an Argument File

In the argument file,

```
-cp "lib/cool\\
```

```
\app\jars"
```

This is interpreted as:

```
-cp /lib/cool app\jars
```

Examples of Using Single Argument File

You can use a single argument file, such as myargumentfile in the following example, to hold all required java arguments:

```
java @myargumentfile
```

Examples of Using Argument Files with Paths

You can include relative paths in argument files; however, they're relative to the current working directory and not to the paths of the argument files themselves. In the following example, path1/options and path2/options represent argument files with different paths. Any relative paths that they contain are relative to the current working directory and not to the argument files:

```
java @path1/options @path2/classes
```

CODE HEAP STATE ANALYTICS

Overview

There are occasions when having insight into the current state of the JVM code heap would be helpful to answer questions such as:

? Why was the JIT turned off and then on again and again?

? Where has all the code heap space gone?

? Why is the method sweeper not working effectively?

To provide this insight, a code heap state analytics feature has been implemented that enables on-the-fly analysis of the code heap. The analytics process is divided into two parts. The first part examines the entire code heap and aggregates all information that is believed to be useful or important. The second part consists of several independent steps that print the collected information with an emphasis on different aspects of the data.

Data collection and printing are done on an "on request" basis.

Syntax

Requests for real-time, on-the-fly analysis can be issued with the following command:

```
jcmd pid Compiler.CodeHeap_Analytics [function] [granularity]
```

If you are only interested in how the code heap looks like after running a sample workload, you can use the command line option:

```
-Xlog:codecache=Trace
```

To see the code heap state when a "CodeCache full" condition exists, start the VM with the command line option:

```
-Xlog:codecache=Debug
```

See [CodeHeap State Analytics \(OpenJDK\) \[https://bugs.openjdk.org/secure/attach?ment/75649/JVM_CodeHeap_StateAnalytics_V2.pdf\]](https://bugs.openjdk.org/secure/attach?ment/75649/JVM_CodeHeap_StateAnalytics_V2.pdf) for a detailed description of the code heap state analytics feature, the supported functions, and the granularity options.

ENABLE LOGGING WITH THE JVM UNIFIED LOGGING FRAMEWORK

You use the -Xlog option to configure or enable logging with the Java Virtual Machine (JVM) unified logging framework.

Synopsis

```
-Xlog[:[what]][:[output]][:[decorators]][:output-options[,...]]]
```

```
-Xlog:directive
```

what Specifies a combination of tags and levels of the form

tag1[+tag2...][*][=level][,...]. Unless the wildcard (*) is specified, only log mes?

sages tagged with exactly the tags specified are matched. See -Xlog Tags and Levels.

output Sets the type of output. Omitting the output type defaults to stdout. See -Xlog Output.

decorators

Configures the output to use a custom set of decorators. Omitting decorators de?

faults to uptime, level, and tags. See Decorations.

output-options

Sets the -Xlog logging output options.

directive

A global option or subcommand: help, disable, async

Description

The Java Virtual Machine (JVM) unified logging framework provides a common logging system for all components of the JVM. GC logging for the JVM has been changed to use the new logging framework. The mapping of old GC flags to the corresponding new Xlog configuration is described in [Convert GC Logging Flags to Xlog](#). In addition, runtime logging has also been changed to use the JVM unified logging framework. The mapping of legacy runtime logging flags to the corresponding new Xlog configuration is described in [Convert Runtime Logging Flags to Xlog](#).

The following provides quick reference to the -Xlog command and syntax for options:

-Xlog Enables JVM logging on an info level.

-Xlog:help

Prints -Xlog usage syntax and available tags, levels, and decorators along with example command lines with explanations.

-Xlog:disable

Turns off all logging and clears all configuration of the logging framework including the default configuration for warnings and errors.

-Xlog[:option]

Applies multiple arguments in the order that they appear on the command line. Multiple -Xlog arguments for the same output override each other in their given order.

The option is set as:

```
[tag-selection][:[output][:[decorators][:output-options]]]
```

Omitting the tag-selection defaults to a tag-set of all and a level of info.

```
tag[+...] all
```

The `all` tag is a meta tag consisting of all tag-sets available. The asterisk `*` in a tag set definition denotes a wildcard tag match. Matching with a wildcard selects all tag sets that contain at least the specified tags. Without the wildcard, only exact matches of the specified tag sets are selected.

output-options is

```
filecount=file-count filesize=file size with optional K, M or G suffix fold?
```

```
multilines=<true|false>
```

When `foldmultilines` is `true`, a log event that consists of multiple lines will be folded into a single line by replacing newline characters with the sequence `'\'` and `'n'` in the output. Existing single backslash characters will also be replaced with a sequence of two backslashes so that the conversion can be reversed. This option is safe to use with UTF-8 character encodings, but other encodings may not work. For example, it may incorrectly convert multi-byte sequences in Shift JIS and BIG5.

Default Configuration

When the -Xlog option and nothing else is specified on the command line, the default configuration is used. The default configuration logs all messages with a level that matches either warning or error regardless of what tags the message is associated with. The default configuration is equivalent to entering the following on the command line:

```
-Xlog:all=warning:stdout:uptime,level,tags
```

Controlling Logging at Runtime

Logging can also be controlled at run time through Diagnostic Commands (with the `jcmd` utility). Everything that can be specified on the command line can also be specified dynamically with the `VM.log` command. As the diagnostic commands are automatically exposed as MBeans, you can use JMX to change logging configuration at run time.

-Xlog Tags and Levels

Each log message has a level and a tag set associated with it. The level of the message corresponds to its details, and the tag set corresponds to what the message contains or which JVM component it involves (such as, gc, jit, or os). Mapping GC flags to the Xlog configuration is described in [Convert GC Logging Flags to Xlog](#). Mapping legacy runtime logging flags to the corresponding Xlog configuration is described in [Convert Runtime Logging Flags to Xlog](#).

Available log levels:

? off

? trace

? debug

? info

? warning

? error

Available log tags:

There are literally dozens of log tags, which in the right combinations, will enable a range of logging output. The full set of available log tags can be seen using `-Xlog:help`. Specifying all instead of a tag combination matches all tag combinations.

-Xlog Output

The `-Xlog` option supports the following types of outputs:

? stdout --- Sends output to stdout

? stderr --- Sends output to stderr

? file=filename --- Sends output to text file(s).

When using `file=filename`, specifying `%p`, `%t` and/or `%hn` in the file name expands to the JVM's PID, startup timestamp and host name, respectively. You can also configure text files to handle file rotation based on file size and a number of files to rotate. For example, to rotate the log file every 10 MB and keep 5 files in rotation, specify the options `file?size=10M, filecount=5`. The target size of the files isn't guaranteed to be exact, it's just

-Xlog Output Mode

```
-Xlog:async[:[stall|drop]]
```

Decorations

```
[6.567s][info][gc,old] Old collection complete
```

Logging Messages Decorations

[illegible]

utctime or utc Universal Time Coordinated or Coordinated Universal Time.

timemillis or The same value as generated by System.currentTimeMillis?

um

uptimenanos or Nanoseconds since the JVM started.

un

pid or p The process identifier.

level or l	The level associated with the log message.
------------	--

tags or tg	The tag-set associated with the log message.
------------	--

Legacy GC Logging Flags to Xlog Configuration Mapping

tion (GC) Flag tion

[illegible]

gion=trace

GCLogFileSize No configuration Log rotation is handled by the available framework.

NumberOfGCLogFiles	Not Applicable	Log rotation is handled by the framework.
--------------------	----------------	---

Page 71/87

PrintGC	-Xlog:gc	Not Applicable
PrintGCApplicationCon?	-Xlog:safepoint	Note that PrintGCApplication?
currentTime		ConcurrentTime and PrintGCApp?
		plicationStoppedTime are logged
		on the same tag and aren't sep?
		arated in the new logging.
PrintGCApplication?	-Xlog:safepoint	Note that PrintGCApplication?
StoppedTime		ConcurrentTime and PrintGCApp?
		plicationStoppedTime are logged
		on the same tag and not sepa?
		rated in the new logging.
PrintGCCause	Not Applicable	GC cause is now always logged.
PrintGCDateStamps	Not Applicable	Date stamps are logged by the
		framework.
PrintGCDetails	-Xlog:gc*	Not Applicable
PrintGCID	Not Applicable	GC ID is now always logged.
PrintGCTaskTimeStamps	-Xlog:gc+task*=debug	Not Applicable
PrintGCTimeStamps	Not Applicable	Time stamps are logged by the
		framework.
PrintHeapAtGC	-Xlog:gc+heap=trace	Not Applicable
PrintReferenceGC	-Xlog:gc+ref*=debug	Note that in the old logging,
		PrintReferenceGC had an effect
		only if PrintGCDetails was also
		enabled.
PrintStringDeduplica?	`-Xlog:gc+stringdedup*=de?`	Not Applicable
tionStatistics	bug	
PrintTenuringDistribu?	-Xlog:gc+age*=level	Use a level of debug for the
tion		most relevant information, or a
		level of trace for all of what
		was logged for PrintTenur?
		ingDistribution.
UseGCLogFileRotation	Not Applicable	What was logged for PrintTenur?
		ingDistribution.

Convert Runtime Logging Flags to Xlog

These legacy flags are no longer recognized and will cause an error if used directly. Use their unified logging equivalent instead.

Runtime Logging Flags to Xlog Configuration Mapping

Legacy Flag	Runtime	Xlog Configuration	Comment
-------------	---------	--------------------	---------

TraceExceptions	-Xlog:excep?		Not Applicable
-----------------	--------------	--	----------------

	tions=info		
--	------------	--	--

TraceClassLoad?	-Xlog:class+load=level		Use level=info for regular informa?
-----------------	------------------------	--	-------------------------------------

ing		tion, or level=debug for additional	
-----	--	-------------------------------------	--

	information. In Unified Logging		
--	---------------------------------	--	--

	syntax, -verbose:class equals		
--	-------------------------------	--	--

	-Xlog:class+load=info,class+un?		
--	---------------------------------	--	--

	load=info.		
--	------------	--	--

TraceClassLoad?	-Xlog:class+pre?		Not Applicable
-----------------	------------------	--	----------------

ingPreorder	order=debug		
-------------	-------------	--	--

TraceClassUn?	-Xlog:class+un?		Use level=info for regular informa?
---------------	-----------------	--	-------------------------------------

loading	load=level	tion, or level=trace for additional	
---------	------------	-------------------------------------	--

	information. In Unified Logging		
--	---------------------------------	--	--

	syntax, -verbose:class equals		
--	-------------------------------	--	--

	-Xlog:class+load=info,class+un?		
--	---------------------------------	--	--

	load=info.		
--	------------	--	--

VerboseVerifi?	-Xlog:verifica?		Not Applicable
----------------	-----------------	--	----------------

cation	tion=info		
--------	-----------	--	--

TraceClassPaths	-Xlog:class+path=info		Not Applicable
-----------------	-----------------------	--	----------------

TraceClassReso?	-Xlog:class+re?		Not Applicable
-----------------	-----------------	--	----------------

lution	solve=debug		
--------	-------------	--	--

TraceClassIni?	-Xlog:class+init=info		Not Applicable
----------------	-----------------------	--	----------------

tialization			
-------------	--	--	--

TraceLoaderCon?	-Xlog:class+loader+con?		Not Applicable
-----------------	-------------------------	--	----------------

straints	straints=info		
----------	---------------	--	--

TraceClassLoad?	-Xlog:class+loader+data=level	Use level=debug for regular infor?	
-----------------	-------------------------------	------------------------------------	--

tional information.

TraceSafepoint? -Xlog:safepoint+cleanup=info Not Applicable

CleanupTime

TraceSafepoint	-Xlog:safepoint=debug	Not Applicable
----------------	-----------------------	----------------

TraceMonitorIn? -Xlog:monitorinflation=debug Not Applicable

flation

TraceReDe? -Xlog:redefine+class*=level level=info, debug, and trace pro?

fineClasses vide increasing amounts of informa?

tion.

-Xlog Usage Examples

The following are -Xlog examples.

-Xlog Logs all messages by using the info level to stdout with uptime, levels, and tags decorations. This is equivalent to using:

```
-Xlog:all=info:stdout:uptime,levels,tags
```

-Xlog:gc

Logs messages tagged with the gc tag using info level to stdout. The default configuration for all other messages at level warning is in effect.

-Xlog:gc,safepoint

Logs messages tagged either with the `gc` or `safepoint` tags, both using the `info` level, to `stdout`, with default decorations. Messages tagged with both `gc` and `safepoint` won't be logged.

-Xlog:gc+ref=debug

Logs messages tagged with both gc and ref tags, using the debug level to stdout, with default decorations. Messages tagged only with one of the two tags won't be logged.

```
-Xlog:gc=debug:file=gc.txt:none
```

Logs messages tagged with the gc tag using the debug level to a file called gc.txt with no decorations. The default configuration for all other messages at level warn? ing is still in effect.

```
-Xlog:gc=trace:file=gctrace.txt:uptimemillis,pid:filecount=5,filesize=1024
```

Logs messages tagged with the gc tag using the trace level to a rotating file set with 5 files with size 1 MB with the base name gctrace.txt and uses decorations upti? memillis and pid.

The default configuration for all other messages at level warning is still in effect.

`-Xlog:gc::uptime,tid`

Logs messages tagged with the gc tag using the default 'info' level to default the output stdout and uses decorations uptime and tid. The default configuration for all other messages at level warning is still in effect.

`-Xlog:gc*=info,safepoint*=off`

Logs messages tagged with at least gc using the info level, but turns off logging of messages tagged with safepoint. Messages tagged with both gc and safepoint won't be logged.

`-Xlog:disable -Xlog:safepoint=trace:safepointtrace.txt`

Turns off all logging, including warnings and errors, and then enables messages tagged with safepoint using tracelevel to the file safepointtrace.txt. The default configuration doesn't apply, because the command line started with `-Xlog:disable`.

Complex -Xlog Usage Examples

The following describes a few complex examples of using the -Xlog option.

`-Xlog:gc+class*=debug`

Logs messages tagged with at least gc and class tags using the debug level to stdout.

The default configuration for all other messages at the level warning is still in effect

`-Xlog:gc+meta*=trace,class*=off:file=gcmetatrace.txt`

Logs messages tagged with at least the gc and meta tags using the trace level to the file metatrace.txt but turns off all messages tagged with class. Messages tagged with gc, meta, and class aren't be logged as class* is set to off. The default configuration for all other messages at level warning is in effect except for those that include class.

`-Xlog:gc+meta=trace`

Logs messages tagged with exactly the gc and meta tags using the trace level to stdout. The default configuration for all other messages at level warning is still in effect.

`-Xlog:gc+class+heap*=debug,meta*=warning,threads*=off`

Logs messages tagged with at least gc, class, and heap tags using the trace level to stdout but only log messages tagged with meta with level. The default configuration for all other messages at the level warning is in effect except for those that in?

clude threads.

VALIDATE JAVA VIRTUAL MACHINE FLAG ARGUMENTS

You use values provided to all Java Virtual Machine (JVM) command-line flags for validation and, if the input value is invalid or out-of-range, then an appropriate error message is displayed.

Whether they're set ergonomically, in a command line, by an input tool, or through the APIs (for example, classes contained in the package `java.lang.management`) the values provided to all Java Virtual Machine (JVM) command-line flags are validated. Ergonomics are described in Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide. Range and constraints are validated either when all flags have their values set during JVM initialization or a flag's value is changed during runtime (for example using the `jcmd` tool). The JVM is terminated if a value violates either the range or constraint check and an appropriate error message is printed on the error stream.

For example, if a flag violates a range or a constraint check, then the JVM exits with an error:

```
java -XX:AllocatePrefetchStyle=5 -version
intx AllocatePrefetchStyle=5 is outside the allowed range [ 0 ... 3 ]
Improperly specified VM option 'AllocatePrefetchStyle=5'
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

The flag `-XX:+PrintFlagsRanges` prints the range of all the flags. This flag allows automatic testing of the flags by the values provided by the ranges. For the flags that have the ranges specified, the type, name, and the actual range is printed in the output.

For example,

```
intx ThreadStackSize [ 0 ... 9007199254740987 ] {pd product}
```

For the flags that don't have the range specified, the values aren't displayed in the print out. For example:

```
size_t NewSize [ ... ] {product}
```

This helps to identify the flags that need to be implemented. The automatic testing framework can skip those flags that don't have values and aren't implemented.

LARGE PAGES

You use large pages, also known as huge pages, as 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. A TLB is a scarce system resource. A TLB miss can be costly because 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. This results in less pressure on a TLB, and memory-intensive applications may have better performance.

However, using large pages can negatively affect system performance. For example, when a large amount 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.

Linux and Windows support large pages.

Large Pages Support for Linux

Linux supports large pages since version 2.6. To check if your environment supports large pages, try the following:

```
# cat /proc/meminfo | grep Huge
HugePages_Total: 0
HugePages_Free: 0
...
Hugepagesize: 2048 kB
```

If the output contains items prefixed with "Huge", then your system supports large pages.

The values may vary depending on environment. The Hugepagesize field shows the default large page size in your environment, and the other fields show details for large pages of this size. Newer kernels have support for multiple large page sizes. To list the supported page sizes, run this:

```
# ls /sys/kernel/mm/hugepages/
hugepages-1048576kB hugepages-2048kB
```

The above environment supports 2 MB and 1 GB large pages, but they need to be configured so that the JVM can use them. When using large pages and not enabling transparent huge pages (option `-XX:+UseTransparentHugePages`), the number of large pages must be pre-allocated. For example, to enable 8 GB of memory to be backed by 2 MB large pages, login as root and run:

```
# echo 4096 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

It is always recommended to check the value of `nr_hugepages` after the request to make sure the kernel was able to allocate the requested number of large pages.

Note: The values contained in `/proc` and `/sys` reset after you reboot your system, so may want to set them in an initialization script (for example, `rc.local` or `sysctl.conf`).

If you configure the OS kernel parameters to enable use of large pages, the Java processes may allocate large pages for the Java heap as well as other internal areas, for example:

? Code cache

? Marking bitmaps

Consequently, if you configure the `nr_hugepages` parameter to the size of the Java heap, then the JVM can still fail to allocate the heap using large pages because other areas such as the code cache might already have used some of the configured large pages.

Large Pages Support for Windows

To use large pages support on Windows, the administrator must first assign additional privileges to the user who is running the application:

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

Note that these steps are required even if it's the administrator who's running the application, because administrators by default don't have the privilege to lock pages in memory.

APPLICATION CLASS DATA SHARING

Application Class Data Sharing (AppCDS) stores classes used by your applications in an archive file. Since these classes are stored in a format that can be loaded very quickly (compared to classes stored in a JAR file), AppCDS can improve the start-up time of your applications. In addition, AppCDS can reduce the runtime memory footprint by sharing parts of these classes across multiple processes.

Classes in the CDS archive are stored in an optimized format that's about 2 to 5 times larger than classes stored in JAR files or the JDK runtime image. Therefore, it's a good idea to archive only those classes that are actually used by your application. These usually are just a small portion of all available classes. For example, your application may use only a few APIs provided by a large library.

Using CDS Archives

By default, in most JDK distributions, unless `-Xshare:off` is specified, the JVM starts up with a default CDS archive, which is usually located in `JAVA_HOME/lib/server/classes.jsa` (or `JAVA_HOME\bin\server\classes.jsa` on Windows). This archive contains about 1300 core library classes that are used by most applications.

To use CDS for the exact set of classes used by your application, you can use the `-XX:SharedArchiveFile` option, which has the general form:

```
-XX:SharedArchiveFile=<static_archive>:<dynamic_archive>
```

? The `<static_archive>` overrides the default CDS archive.

? The `<dynamic_archive>` provides additional classes that can be loaded on top of those in the `<static_archive>`.

? On Windows, the above path delimiter `:` should be replaced with `;`

The names "static" and "dynamic" are used for historical reasons. The dynamic archive, while still useful, supports fewer optimizations than available for the static CDS archive.

If the full set of CDS/AOT optimizations are desired, consider using the AOT cache described below.

The JVM can use up to two archives. To use only a single `<static_archive>`, you can omit the `<dynamic_archive>` portion:

```
-XX:SharedArchiveFile=<static_archive>
```

For convenience, the `<dynamic_archive>` records the location of the `<static_archive>`. Therefore, you can omit the `<static_archive>` by saying only:

```
-XX:SharedArchiveFile=<dynamic_archive>
```

Manually Creating CDS Archives

CDS archives can be created manually using several methods:

? `-Xshare:dump`

? `-XX:ArchiveClassesAtExit`

? `jcmd VM.cds`

One common operation in all these methods is a "trial run", where you run the application once to determine the classes that should be stored in the archive.

Creating a Static CDS Archive File with `-Xshare:dump`

The following steps create a static CDS archive file that contains all the classes used by the test.Hello application.

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:DumpLoadedClassList=hello.classlist -cp hello.jar test.Hello
```

The classpath specified by the `-cp` parameter must contain only JAR files.

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

`hello.classlist`:

```
java -Xshare:dump -XX:SharedArchiveFile=hello.jsa -XX:SharedClassList?
File=hello.classlist -cp hello.jar
```

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

```
java -XX:SharedArchiveFile=hello.jsa -cp hello.jar test.Hello
```

4. Optional Verify that the `test.Hello` application is using the class contained in the

`hello.jsa` shared archive:

```
java -XX:SharedArchiveFile=hello.jsa -cp hello.jar -Xlog:class+load test.Hello
```

The output of this command should contain the following text:

```
[info][class,load] test.Hello source: shared objects file
```

By default, when the `-Xshare:dump` option is used, the JVM runs in interpreter-only mode (as if the `-Xint` option were specified). This is required for generating deterministic output in the shared archive file. I.e., the exact same archive will be generated, bit-for-bit, every time you dump it. However, if deterministic output is not needed, and you have a large classlist, you can explicitly add `-Xmixed` to the command-line to enable the JIT compiler. This will speed up the archive creation.

Creating a Dynamic CDS Archive File with `-XX:ArchiveClassesAtExit`

Advantages of dynamic CDS archives are:

- ? They usually use less disk space, since they don't need to store the classes that are already ready in the static archive.
- ? They are created with one fewer step than the comparable static archive.

The following steps create a dynamic CDS archive file that contains the classes that are used by the `test.Hello` application, excluding those that are already in the default CDS archive.

1. Create a dynamic CDS archive, named `hello.jsa`, that contains all the classes in `hello.jar` loaded by the application `test.Hello`:

```
java -XX:ArchiveClassesAtExit=hello.jsa -cp hello.jar Hello
```

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


```
java -XX:SharedArchiveFile=hello.jsa -cp hello.jar test.Hello
```

3. Optional Repeat step 4 of the previous section to verify that the test.Hello application is using the class contained in the hello.jsa shared archive.

It's also possible to create a dynamic CDS archive with a non-default static CDS archive.

E.g.,

```
java -XX:SharedArchiveFile=base.jsa -XX:ArchiveClassesAtExit=hello.jsa -cp hello.jar  
Hello
```

To run the application using this dynamic CDS archive:

```
java -XX:SharedArchiveFile=base.jsa:hello.jsa -cp hello.jar Hello
```

(On Windows, the above path delimiter : should be replaced with ;)

As mention above, the name of the static archive can be skipped:

```
java -XX:SharedArchiveFile=hello.jsa -cp hello.jar Hello
```

Creating CDS Archive Files with jcmd

The previous two sections require you to modify the application's start-up script in order to create a CDS archive. Sometimes this could be difficult, for example, if the application's class path is set up by complex routines.

The jcmd VM.cds command provides a less intrusive way for creating a CDS archive by connecting to a running JVM process. You can create either a static:

```
jcmd <pid> VM.cds static_dump my_static_archive.jsa
```

or a dynamic archive:

```
jcmd <pid> VM.cds dynamic_dump my_dynamic_archive.jsa
```

To use the resulting archive file in a subsequent run of the application without modifying the application's start-up script, you can use the following technique:

```
env JAVA_TOOL_OPTIONS=-XX:SharedArchiveFile=my_static_archive.jsa bash app_start.sh
```

Note: to use jcmd <pid> VM.cds dynamic_dump, the JVM process identified by <pid> must be started with -XX:+RecordDynamicDumpInfo, which can also be passed to the application start-up script with the same technique:

```
env JAVA_TOOL_OPTIONS=-XX:+RecordDynamicDumpInfo bash app_start.sh
```

Creating Dynamic CDS Archive File with -XX:+AutoCreateSharedArchive

-XX:+AutoCreateSharedArchive is a more convenient way of creating/using CDS archives. Unlike the methods of manual CDS archive creation described in the previous section, with -XX:+AutoCreateSharedArchive, it's no longer necessary to have a separate trial run. Instead, you can always run the application with the same command-line and enjoy the benefits

of CDS automatically.

```
java -XX:+AutoCreateSharedArchive -XX:SharedArchiveFile=hello.jsa -cp hello.jar Hello
```

If the specified archive file exists and was created by the same version of the JDK, then it will be loaded as a dynamic archive; otherwise it is ignored at VM startup.

At VM exit, if the specified archive file does not exist, it will be created. If it exists but was created with a different (but post JDK 19) version of the JDK, then it will be replaced. In both cases the archive will be ready to be loaded the next time the JVM is launched with the same command line.

If the specified archive file exists but was created by a JDK version prior to JDK 19, then it will be ignored: neither loaded at startup, nor replaced at exit.

Developers should note that the contents of the CDS archive file are specific to each build of the JDK. Therefore, if you switch to a different JDK build, `-XX:+AutoCreateSharedArchive` will automatically recreate the archive to match the JDK. If you intend to use this feature with an existing archive, you should make sure that the archive is created by at least version 19 of the JDK.

Restrictions on Class Path and Module Path

? Neither the class path (`-classpath` and `-Xbootclasspath/a`) nor the module path (`--module-path`) can contain non-empty directories.

? Only modular JAR files are supported in `--module-path`. Exploded modules are not supported.

? The class path used at archive creation time must be the same as (or a prefix of) the class path used at run time. (There's no such requirement for the module path.)

? The CDS archive cannot be loaded if any JAR files in the class path or module path are modified after the archive is generated.

Module related options

The following module related options are supported by CDS: `--module-path`, `--module`, `--add-modules`, and `--enable-native-access`.

The values for these options (if specified), should be identical when creating and using the CDS archive. Otherwise, if there is a mismatch of any of these options, the CDS archive may be partially or completely disabled, leading to lower performance.

? If the `AOTClassLinking` option (see below) was enabled during CDS archive creation, the CDS archive cannot be used, and the following error message is printed:

CDS archive has aot-linked classes. It cannot be used when archived full module graph is

not used

? If the AOTClassLinking option was not enabled during CDS archive creation, the CDS archive can be used, but the "archived module graph" feature will be disabled. This can lead to increased start-up time.

To diagnose problems with the AOT options, you can add -Xlog:aot to the application's VM arguments. For example, if --add-modules jdk.jconsole was specified during archive creation and --add-modules jdk.incubator.vector is specified during runtime, the following messages will be logged:

```
Mismatched values for property jdk.module.addmods
runtime jdk.incubator.vector dump time jdk.jconsole
subgraph jdk.internal.module.ArchivedBootLayer cannot be used because full module graph is
disabled
```

If any of the VM options --upgrade-module-path, --patch-module or --limit-modules are specified, CDS is disabled. This means that the JVM will execute without loading any CDS archives. In addition, if you try to create a CDS archive with any of these 3 options specified, the JVM will report an error.

AHEAD-OF-TIME CACHE

The JDK supports ahead-of-time (AOT) optimizations that can be performed before an application is executed. One example is Class Data Sharing (CDS), as described above, that parses classes ahead of time. AOT optimizations can improve the start-up and warm-up performance of Java applications.

The Ahead-of-Time Cache (AOT cache) is a container introduced in JDK 24 for storing artifacts produced by AOT optimizations. The AOT cache currently contains Java classes and heap objects. In future JDK releases, the AOT cache may contain additional artifacts, such as execution profiles and compiled methods.

An AOT cache is specific to a combination of the following:

- ? A particular application (as expressed by -classpath, -jar, or --module-path.)
- ? A particular JDK release.
- ? A particular OS and CPU architecture.

If any of the above changes, you must recreate the AOT cache.

The deployment of the AOT cache is divided into three phases:

- ? Training: We execute the application with a representative work-load to gather statistical data that tell us what artifacts should be included into the AOT cache. The data are

saved in an AOT Configuration file.

? Assembly: We use the AOT Configuration file to produce an AOT cache.

? Production: We execute the application with the AOT cache for better start-up and warm-up performance.

The AOT cache can be used with the following command-line options:

`-XX:AOTCache=cachefile`

Specifies the location of the AOT cache. The standard extension for cachefile is .aot. This option cannot be used together with `-XX:AOTCacheOutput`.

This option is compatible with AOTMode settings of on, create, or auto (the default).

The cachefile is read in AOT modes on and auto, and is ignored by mode off. The cachefile is written by AOT mode create. In that case, this option is equivalent to `-XX:AOTCacheOutput=cachefile`.

`-XX:AOTCacheOutput=cachefile`

Specifies the location of the AOT cache to write. The standard extension for cachefile is .aot. This option cannot be used together with `-XX:AOTCache`.

This option is compatible with AOTMode settings of record, create, or auto (the default).

`-XX:AOTConfiguration=configfile`

Specifies the AOT Configuration file for the JVM to write to or read from. The standard extension for configfile is .aotconfig.

This option is compatible with AOTMode settings of record, create, or auto (the default). The configfile is read by AOT mode create, and written by the other applicable modes. If the AOT mode is auto, then AOTCacheOutput must also be present.

`-XX:AOTMode=mode`

Specifies the AOT Mode for this run. mode must be one of the following: auto, off, record, create, or on.

? auto: This AOT mode is the default, and takes effect if no `-XX:AOTMode` option is present.

It automatically sets the AOT mode to record, on, or off, as follows:

? If `-XX:AOTCacheOutput=cachefile` is specified, the AOT mode is changed to record (a training run, with a subsequent create operation).

? Otherwise, if an AOT cache can be loaded, the AOT mode is changed to on (a production run).

? Otherwise, the AOT mode is changed to off (a production run with no AOT cache).

? off: No AOT cache is used. Other AOT command line options are ignored.

? record: Execute the application in the training phase. At least one of `-XX:AOTConfiguration=configfile` and/or `-XX:AOTCacheOutput=cachefile` must be specified. If `-XX:AOTConfiguration=configfile` is specified, the JVM gathers statistical data and stores them into `configfile`. If `-XX:AOTConfiguration=configfile` is not specified, the JVM uses a temporary file name, which may be the string `AOTCacheOutput+".config"`, or else a fresh implementation-dependent temporary file name. If `-XX:AOTCacheOutput=cachefile` is specified, a second JVM process is launched to perform the Assembly phase to write the optimization artifacts into `cachefile`.

Extra JVM options can be passed to the second JVM process using the environment variable `JDK_AOT_VM_OPTIONS`, with the same format as the environment variable `JAVA_TOOL_OPTIONS`, which is defined by JVMTI [<https://docs.oracle.com/en/java/javase/24/docs/specs/jvmti.html#tooloptions>].

? create: Perform the Assembly phase. `-XX:AOTConfiguration=configfile` must be specified. The JVM reads history and statistics from `configfile` and writes the optimization artifacts into `cachefile`. Note that the application itself is not executed in this phase.

? on: Execute the application in the Production phase. If `-XX:AOTCache=cachefile` is specified, the JVM tries to load `cachefile` as the AOT cache. Otherwise, the JVM tries to load a default CDS archive from the JDK installation directory as the AOT cache.

The loading of an AOT cache can fail for a number of reasons:

? You are trying to use the AOT cache with an incompatible application, JDK release, or OS/CPU.

? The specified `cachefile` does not exist or is not accessible.

? Incompatible JVM options are used (for example, certain JVMTI options).

Since the AOT cache is an optimization feature, there's no guarantee that it will be compatible with all possible JVM options. See JEP 483 [<https://openjdk.org/jeps/483>], section Consistency of training and subsequent runs for a representative list of scenarios that may be incompatible with the AOT cache.

These scenarios usually involve arbitrary modification of classes for diagnostic purposes and are typically not relevant for production environments.

When the AOT cache fails to load:

? If `AOTMode` was originally `auto`, the JVM will continue execution without using the AOT cache. This is the recommended mode for production environments, especially when you

may not have complete control of the command-line (e.g., your application's launch script may allow users to inject options to the command-line). This allows your application to function correctly, although sometimes it may not benefit from the AOT cache.

If `AOTMode` is on, the JVM will print an error message and exit immediately. This mode should be used only as a "fail-fast" debugging aid to check if your command-line options are compatible with the AOT cache. An alternative is to run your application with `-XX:AOTMode=auto -Xlog:aot` to see if the AOT cache can be used or not.

`-XX:+AOTClassLinking`

If this option is enabled, the JVM will perform more advanced optimizations (such as ahead-of-time resolution of invokedynamic instructions) when creating the AOT cache. As a result, the application will see further improvements in start-up and warm-up performance.

However, an AOT cache created with this option cannot be used when certain command-line parameters are specified in the Production phase. Please see JEP 483 [<https://openjdk.org/jeps/483>] for a detailed discussion of `-XX:+AOTClassLinking` and its restrictions.

When `-XX:AOTMode` is used in the command-line, `AOTClassLinking` is automatically enabled. To disable it, you must explicitly pass the `-XX:-AOTClassLinking` option.

When `-XX:AOTMode` is not used in the command-line, `AOTClassLinking` is disabled by default to provide full compatibility with traditional CDS options such as `-Xshare:dump`.

The first occurrence of the special sequence `%p` in `*configfile*` and `cachefile` is replaced with the process ID of the JVM process launched in the command-line, and likewise the first occurrence of `%t` is replaced by the JVM's startup timestamp. (After replacement there must be no further occurrences of `%p` or `%t`, to prevent problems with sub-processes.) For example:

```
java -XX:AOTConfiguration=foo%p.aotconfig -XX:AOTCacheOutput=foo%p.aot -cp foo.jar
Foo
```

will create two files: `foopid123.aotconfig` and `foopid123.aot`, where 123 is the process ID of the JVM that has executed the application `Foo`.

PERFORMANCE TUNING EXAMPLES

You can use the Java advanced runtime options to optimize the performance of your applications.

Tuning for Higher Throughput

Use the following commands and advanced options to achieve higher throughput performance for your application:

```
java -server -XX:+UseParallelGC -XX:+UseLargePages -Xmn10g -Xms26g -Xmx26g
```

Tuning for Lower Response Time

Use the following commands and advanced options to achieve lower response times for your application:

```
java -XX:+UseG1GC -XX:MaxGCPauseMillis=100
```

Keeping the Java Heap Small and Reducing the Dynamic Footprint of Embedded Applications

Use the following advanced runtime options to keep the Java heap small and reduce the dynamic footprint of embedded applications:

```
-XX:MaxHeapFreeRatio=10 -XX:MinHeapFreeRatio=5
```

Note: The defaults for these two options are 70% and 40% respectively. Because performance sacrifices can occur when using these small settings, you should optimize for a small footprint by reducing these settings as much as possible without introducing unacceptable performance degradation.

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