

Chapter 6

Platform Tools

With every software platform, tools are required to produce and manipulate executable code. Java and .NET are no exception. These platforms ship with a wealth of command-line utilities that perform common tasks like compiling, executing, documenting, and debugging programs. This chapter is intended to serve as a reference to the most common tools available for the Java and .NET platforms. Specifically, tools from the following categories will be presented for each platform:

- Compiler
- Interpreter
- Documentation Generator
- Packager
- Debugger
- Disassembler
- Code Signing Tool

Compiler

In Java and .NET, *compiling* is the process of converting source files into bytecode or intermediate language, respectively. After compilation, files can be executed by the Java Virtual Machine or .NET Common Language Runtime.

Java

Java compilation is accomplished using the *javac* program located in the Java SDK's *bin* directory.

javac – Java Compiler

The *javac* program compiles *.java* source files into *.class* bytecode files. Any number of source files can be declared (separated by spaces) like this:

```
javac MyFirstProgram.java MySecondProgram.java
```

The compiler produces a *.class* file for each class defined within a source file. Each *.class* file is named after the class that it represents. Unless the *-classpath* option is set, *javac* searches for referenced classes in the path declared by the *CLASSPATH* environment variable. Similarly, unless otherwise specified with the *-d* option, *javac* places compiled files in the same directory as their corresponding source files.

In addition to normal source files, the *javac* compiler can process special “project” files that contain a list of compiler options and source filenames. When specifying them on the *javac* command-line, project files must be prefixed with an *@* symbol. Any number of space-delimited

project files may be specified and each project file can contain any number of source filenames and/or compiler options separated by white space. To illustrate, consider a file named `myproject.txt` that consists of the following text (separated by spaces, tabs, or linefeeds):

```
-verbose HelloWorld.java Helper.java Utility.java
```

Given this project file, the following `javac` command would compile the `HelloWorld.java`, `Helper.java`, and `Utility.java` files in verbose mode:

```
javac @myproject.txt
```

Usage

```
javac [options] files
```

The *options* for `javac` are listed below. The *files* argument represents the source files to be compiled or project files to be processed (separated by spaces).

Options

```
-bootclasspath path
```

Specifies the location of system classes to be used by the compiler. If not specified, the compiler uses the classes located under the Java SDK's `/jre/lib` directory. The *path* argument consists of a semi-colon (on Windows) or colon-delimited (on UNIX) list of directories or JAR/ZIP files.

```
-classpath path
```

Specifies the path that the Java compiler uses to locate classes referenced in the source files. This option overrides the `CLASSPATH` environment setting. The *path* argument consists of an ordered list of directories or JAR/ZIP files delimited by semi-colons on Windows or colons on UNIX. Unless specified by the `-sourcepath` option, this option also indicates the location of source files.

```
-d directory
```

Specifies the target directory into which or under which compiled files should be placed. If the `-d` option is not set, `javac` places compiled files in the same directory as their corresponding source files. If a class is defined within a package, `javac` automatically creates an equivalent directory structure under the target directory in which to store the compiled file.

```
-deprecation
```

Indicates that a warning should be generated each time a deprecated API is accessed. By default, regardless of the number of times a deprecated API is used, only one warning is generated for each source file.

```
-encoding encoding
```

Specifies the character encoding used by the source files. If not specified, the default platform encoding is used.

```
-extdirs directoryList
```

Specifies the location of Java extension classes. The *directoryList* argument is a semi-colon-delimited (on Windows) or colon-delimited (on UNIX) list of directories that contain extension JAR files.

`-g`

Indicates that all debugging information should be included in compiled files. If not specified, only line numbers and source file information are included.

`-g:none`

Indicates that no debugging information should be included in the compiled file.

`-g:{lines,var,source}`

Indicates the type of debug information that should be included in compiled files. The valid values are “lines” (line numbers), “vars” (local variables), and “source” (source file information). One or more of these values may be specified in a comma-delimited list.

`-help`

Prints usage and options information to standard out.

`-JinterpreterOption`

Compiles instructions for the *java* interpreter into a `.class` file. The Java interpreter follows these instructions at runtime. The *interpreterOption* argument may contain any option that is recognized by the *java* interpreter. Only one java option can be declared per `-J` option. Use multiple `-J` options to specify multiple *java* options. For example, the following compilation instruction indicates that the interpreter should have an initial heap size of 32 MB and a maximum heap size of 128 MB:

```
javac -J-Xms32m -J-Xmx128m MyProgram.java
```

`-nowarn`

Suppresses all warning messages.

`-O`

Indicates that the compiled file should be optimized for speed. This option may enlarge class files and hinder debugging.

`-source version`

Indicates the version of Java in which the source code is written. To illustrate, the `assert` keyword was added to Java starting with version 1.4. Therefore, to compile code that uses the `assert` keyword, the `-source 1.4` option must be used. If not specified, this option sets the `-target` option as well.

`-sourcepath path`

Specifies the location of source files required to perform a compilation. The *path* argument is a sem-colon-delimited (on Windows) or colon-delimited (on UNIX) list of directories and JAR/ZIP files. Source files will be compiled whenever an associated class file cannot be found or the class file is not current.

`-target version`

Specifies the Java version under which the compiled program is intended to run. Though the class file format has not changed between versions, this setting is useful when a program requires features that are only available under a newer version of Java. This option prevents an application from running under an older version of Java.

`-verbose`

Indicates that compilation activities, such as all source files being compiled, should be displayed on the console. This option provides the user with better feedback regarding the work being performed by the compiler.

`-X`

All non-standard compiler options are declared in the form `-XoptionName`. Without specifying an option name, the `-X` option displays a list of the compiler's non-standard options.

`-Xstdout file`

Instructs the compiler to write its output to the given file rather than to the console.

`-Xswitchcheck`

Issues a warning if the source code includes a `switch` statement in which a `case` block “falls through” to the `case` below (i.e., the `case` does not end with a `break` statement).

.NET/C#

C# programs are compiled using the `csc.exe` program that resides in the .NET Framework SDK's `bin` directory.

csc – C# Compiler

The `csc` program compiles `.cs` source files into `.exe` intermediate language files. Any number of source files can be declared (separated by spaces) like this:

```
csc MyFirstProgram.cs MySecondProgram.cs
```

By default, the `csc` compiler produces a single executable assembly file (`.exe` extension) that runs as a console application (i.e., from the command-line). All of the classes defined in the given source files are compiled into one assembly file. When generating an executable, at least one of the classes defined in any source file must include a static `Main()` method. The executable will use this `Main()` method as its entry point. If more than one class defines a static `Main()` method, the `/main` option must be used to indicate which class's `Main()` method should be used as the assembly's entry point. If not specified using the `/out` option, the assembly is named after the source file that includes the class containing the static `Main()` method used as the assembly's entry point.

In addition to console executables, using the `/target` option, the `csc` compiler can produce library (`.dll` extension), module (`.netmodule` extension), or Windows executable (`.exe` extension) assemblies. Unlike executables, library and module assemblies do not require a static `Main()` method as an entry point.

In addition to normal source files, the `csc` compiler can process special “response” files that contain a list of compiler options and source filenames. When specifying them on the `csc` command-line, project files must be prefixed with an `@` symbol. Any number of space-delimited project files may be specified and each project file can contain any number of source filenames

and/or compiler options separated by white space. To illustrate, consider a file named `myresponse.rsp` that consists of the following text (separated by spaces, tabs, or linefeeds):

```
/target:winexe HelloWorld.cs Helper.cs Utility.cs
```

Given this response file, the following `csc` command would compile the `HelloWorld.cs`, `Helper.cs`, and `Utility.cs` files into a Windows executable:

```
csc @myproject.txt
```

In addition to compiling C# source code into intermediate language, `csc` can combine resource files and compiled modules into a single assembly, extract bug report information, define conditional compilation symbols, and generate XML documentation.

Usage

```
csc [options] files
```

The *options* for `csc` are listed below. The *files* argument represents the source files to be compiled or response files to be processed (separated by spaces).

Options

```
/addmodule:files
```

Adds one or more modules to the assembly. The *files* parameter represents a semi-colon delimited list of files.

```
/baseaddress:address
```

The base address from which to load a DLL.

```
/bugreport:file
```

Generates a bug report file that contains information such as source code, compiler options, and version information.

```
/checked[+|-]
```

Indicates whether or not an exception should be thrown when an integer arithmetic statement results in a value outside the range of its data type. This option has no effect on statements that reside within the scope of a `checked` or `unchecked` keyword. The `/checked` and `/checked+` options indicate that an exception should be thrown in the case of an arithmetic overflow while the absence of a `/checked` option or a `/checked-` option indicate that no exception should be thrown.

```
/codepage:identifier
```

Specifies the code page to use for all source files.

```
/debug[+|-]
```

```
/debug:full
```

```
/debug:pdbonly
```

Indicates whether or not debug information should be generated and written to a `.pdb` debug file. The absence of the `/debug` option or `/debug-` indicate that no debug information should be generated (this is the default). Specifying `/debug` or `/debug+` indicate that debug information should be generated. In addition, it is possible to indicate whether a debugger should be able to attach to a running program within this compilation

and perform source code debugging. Specifying `/debug` or `/debug:full` indicates that a debugger can attach to the running program and perform source code debugging (this is the default). The `/debug:pdbonly` option indicates that a debugger cannot perform source code debugging by attaching to a running program (this will only produce assembly language). To perform source code debugging with the `/debug:pdbonly` option set, the program must be started in the debugger.

```
/define:symbolNames  
/d:symbolNames
```

Defines one or more conditional compilation symbols (equivalent to the `#define` command in source code). The *symbolNames* parameter represents a semi-colon delimited list of symbol names.

```
/doc:file
```

Generates a documentation file in XML that contains the documentation comments embedded in the source code. The *file* parameter indicates the name of the generated XML documentation file.

```
/filealign:number
```

Specifies the size of sections in the output file. Valid values for the *number* parameter are 512, 1024, 2048, 4096, 8192, and 16384.

```
/fullpaths
```

Indicates that the compiler should display the full path to files that produce compilation errors or warnings.

```
/help  
/?
```

Prints a list of compiler options to standard out.

```
/incremental[+|-]  
/incr[+|-]
```

Enables or disables incremental compilation. The `/incremental` and `/incremental+` options are equivalent while the absence of the `/incremental` option and `/incremental-` option are equivalent.

```
/lib:directories
```

Specifies a list of directories in which to search for assemblies that are referenced using the `/reference` option.

```
/linkresource:file[,identifier]  
/linkres:file[,identifier]
```

Embeds a link to a .NET resource file (not the resource itself) into the compiled assembly. The resource can be specified by filename or filename and logical name (i.e., unique identifier).

```
/main:class  
/m:class
```

Specifies the class that contains the `Main()` method that should be used as the assembly's entry point.

`/noconfig`

Indicates that the compiler settings specified in the *csc.rsp* file should be not be used.

`/nologo`

Suppresses output of the startup banner and copyright information.

`/nostdlib[+|-]`

Indicates whether or not the standard system library (*mscorlib.dll*) should be imported. The `/nostdlib` and `/nostdlib+` options are equivalent and both indicate that the standard system library should not be imported. The `/nostdlib-` or absence of the `/nostdlib` option are equivalent and both indicate that the standard system library should be imported.

`/nowarn[:numbers]`

Suppresses compiler warnings. The *numbers* parameter represents a comma-delimited list of warning numbers that should be suppressed. If no warning numbers are specified, all warnings are suppressed.

`/optimize[+|-]`

`/o[+|-]`

Indicates whether or not the compilation should be optimized for speed, size, and efficiency. The `/optimize`, `/optimize+`, and absence of the `/optimize` option are all equivalent and all indicate that compile-time and run-time optimizations should be performed. The `/optimize-` option suppresses optimization.

`/out:file`

Specifies the name of the output file created by the compiler.

`/recurse:[directory]\wildcard`

Indicates that the compiler should compile all files in the current directory (or specified *directory*) and subdirectories that satisfy the *wildcard* parameter (e.g., **.cs*).

`/reference:files`

`/r:files`

Specifies assembly files that contain classes required for compilation (i.e., that are referenced by the source code). These classes may reside in library (**.dll*), module (**.netmodule*), and/or executable (**.exe*) files.

`/resource:file[,identifier]`

`/res:file[,identifier]`

Embeds a .NET resource file into the output file created by the compiler. The resource can be specified by filename or filename and logical name (i.e., unique identifier).

`/target:format`

`/t:format`

Specifies the format of the output file created by the compiler. Valid formats include *exe* (command-line application), *library* (dynamic link library), *module* (code library without an assembly manifest), and *winexe* (Windows application).

`/unsafe`

Indicates that compilation should be allowed for source files containing unsafe code (i.e., code within an `unsafe` block).

```
/utf8output
```

Indicates that the compiler should use UTF-8 encoding when printing output to the console.

```
/warn:level  
/w:level
```

Specifies the level at which the compiler should generate warning messages. Valid levels range from 0 to 4 (i.e., 0, 1, 2, 3, 4) where 0 means all warnings should be suppressed and 4, which is the default, indicates that all warnings should be output to the console.

```
/warnaserror[+|-]
```

Indicates that warnings should be promoted to errors so that compilation is stopped. The `/warnaserror` and `/warnaserror+` options are equivalent and both cause warnings to be treated like errors. The `/warnaserror-` or the absence of the `/warnaserror` option indicates that warnings should not be promoted to errors.

```
/win32icon:file
```

Specifies the name of an icon file to embed in the compiled assembly. This icon is often displayed by a file explorer utility or on a desktop shortcut.

```
/win32res:file
```

Embeds the specified Win32 resource file into the output file created by the compiler.

Interpreter (Java)

Since Java programs are not compiled to native code, they require an interpreter for execution. .NET does not require an interpreter because all .NET code is compiled to native code at runtime by the just-in-time (JIT) compiler built into the Common Language Runtime (CLR). Additionally, when a .NET assembly is executed, the CLR is automatically invoked to perform the execution. In contrast, the Java interpreter is not invoked automatically but must be explicitly called whenever a Java program is executed.

java – Java Interpreter

Located in the Java SDK's *bin* directory, the Java interpreter (also known as the Java Virtual Machine) is used to run Java programs that have been compiled to platform-neutral byte-codes. A Java class is executed by specifying its fully qualified class name (including package information) like this:

```
java com.sourcestream.util.NetworkMonitor
```

Since the *java* interpreter uses class names rather than filenames, the `.class` file extension is not specified.

In order for a class to be executed by the interpreter, the class must implement a `main()` method having this signature:


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

This `main()` method is the entry point at which the interpreter begins execution. Similar to classes, the java interpreter can execute properly constructed JAR files. To create an executable JAR file, the *META-INF/manifest.mf* file within the JAR must specify a `Main-Class` attribute like this:

```
Manifest-Version: 1.0
Main-Class: MyMainClass
```

The `Main-Class` attribute indicates which class file within the JAR should be invoked when the JAR is executed. Of course, this class file must contain a `main()` method. The `-jar` option is used to execute an executable JAR like this:

```
java -jar MyProgram.jar
```

Command-line arguments prior to the class/JAR name are passed as options to the *java* interpreter. Command-line arguments following the class/JAR name are passed as a `String` array to the executable class's `main()` method like this:

```
java -Xdebug MyPrintNamesProgram Larry Curly Moe
```

In this example, the `-XDebug` option is processed by the *java* interpreter since it is declared prior to the program name (`MyPrintNamesProgram`) while the three names (`Larry`, `Curly`, and `Moe`) are command-line arguments passed as a `String` array to the `main()` method of the `MyPrintNamesProgram` class.

In addition to the standard *java* interpreter, Windows versions of Java include a program called *javaw*. The *javaw* program executes a Java application without opening a console window. Using *javaw*, graphical Java applications can be launched similarly to native Windows applications (without opening a command-line window). Regarding usage rules and options, the *javaw* program is identical to *java*.

Usage

```
java [interpreterOptions] className [programArguments]
java [interpreterOptions] -jar jarFile [programArguments]
```

The *interpreterOptions* argument represents options that are processed by the interpreter. In contrast, *programArguments* represents arguments that are processed by the application being executed. The *className* and *jarFile* arguments are the names of the class or JAR file to be executed, respectively.

Options

```
-classpath path
-cp path
```

Indicates the path that the Java interpreter uses to locate classes referenced in the compiled files. This option overrides the `CLASSPATH` environment setting. The *path* consists of an ordered list of directories or JAR/ZIP files delimited by semi-colons on Windows or colons on UNIX.

```
-client
```

Tunes interpreter for optimal execution of client-side programs.

`-Dname=value`

Sets properties in the systems properties list according to the given *name* and *value*.
Java programs can read system properties using the
`java.lang.System.getProperty()` static method.

`-disableassertions[:className|:packageName...]`

`-da[:className|:packageName...]`

Disables assertions within the specified class or package. If no class or package is specified, all assertions (except those in system classes) are disabled. When disabling assertions for a particular package, the fully qualified package name must be followed by three periods.

`-disablesystemassertions`

`-dsa`

Disables assertions within all system classes.

`-enableassertions[:className|:packageName...]`

`-ea[:className|:packageName...]`

Enables assertions within the specified class or package. If no class or package is specified, all assertions (except those in system classes) are enabled. When enabling assertions for a particular package, the fully qualified package name must be followed by three periods.

`-enablesystemassertions`

`-esa`

Enables assertions within all system classes.

`-help`

`-?`

Prints usage and options information to standard out.

`-jar jarFile`

Executes the specified jar file. To be executable, the jar's manifest file must contain a `Main-Class` attribute that indicates the name of the class whose `main()` method should be called.

`-server`

Tunes interpreter for optimal execution of server-side programs.

`-showversion`

Prints the version of interpreter to standard out and continues executing.

`-verbose`

`-verbose:class`

Prints a message to standard out each time a class is loaded.

`-verbose:gc`

Prints a message to standard out each time the garbage collector is invoked.

`-verbose:jni`

Prints a message to standard out each time a native library is invoked.

`-version`
Prints the version of the interpreter to standard out and terminates execution.

`-X`
Prints a message to standard out that presents non-standard options.

`-Xbatch`
Instructs the JIT compiler to perform compilation using a high-priority thread. By default, compilation is performed using a low-priority thread while high-priority threads are reserved for interpretation.

`-Xbootclasspath: path`
Specifies the location of system classes to be used by the interpreter. The *path* argument consists of a semi-colon- (on Windows) or colon-delimited (on UNIX) list of directories or JAR/ZIP files.

`-Xbootclasspath/a: path`
Appends the specified path to the boot classpath. The *path* argument consists of a semi-colon- (on Windows) or colon-delimited (on UNIX) list of directories or JAR/ZIP files.

`-Xbootclasspath/p: path`
Prepends the specified path to the boot classpath. The *path* argument consists of a semi-colon- (on Windows) or colon-delimited (on UNIX) list of directories or JAR/ZIP files.

`-Xdebug`
Instructs the interpreter to listen for requests from debuggers and allow them to connect to it.

`-Xfuture`
Provides additional bytecode verification checks.

`-Xincgc`
Indicates that the interpreter should use incremental garbage collection. This strategy requires that the garbage collector runs continually using a low priority thread. Incremental garbage collection may reduce the length of delays caused by garbage collection but results in lower performance overall.

`-Xint`
Instructs the interpreter that no JIT compilation should be performed.

`-Xloggc: filename`
Prints information regarding garbage collector activity to the specified file.

`-Xmixed`
Indicates that the most frequently used blocks of code should be JIT compiled while the lesser used code should be interpreted.

`-XmsinitialHeapSize [k|m]`

Specifies the initial amount of memory allocated to the Java interpreter's heap. By default, this amount is specified in bytes. Append a `k` or `m` to indicate kilobytes or megabytes, respectively. The default initial heap size is 1MB.

`-XmxmaxHeapSize[k|m]`

Specifies the maximum amount of memory to be allocated to the Java interpreter's heap. By default, this amount is specified in bytes. Append a `k` or `m` to indicate kilobytes or megabytes, respectively. The default maximum heap size is 64MB.

`-Xnoclassgc`

Indicates that classes should not be garbage collected.

`-Xnoagent`

When debugging with the `-Xdebug` option, indicates that JDWP (Java Debug Wire Protocol) should be used rather than the original `sun.tools.debug` agent.

`-Xprof`

Prints code profiling information to standard out.

`-Xrunhprof[:help|:optionName=value,...]`

Prints code profiling information to standard out according to the specified options. Use `-Xrunhprof:help` for a list of available option names and values.

`-Xrunjdpw[:optionName=value,...]`

Used in conjunction with the `-Xdebug` option, indicates that JDWP (Java Debug Wire Protocol) should be used to attach to and conduct debug sessions on the current virtual machine. Among other things, options can be specified to indicate the type of connection and the port on which to listen for debug connection requests like this:

`-Xrunjdpw:transport=dt_socket,server=y,suspend=n,address=5000`

`-XssstackSize[k|m]`

Specifies the stack size used by the interpreter. By default, this amount is specified in bytes. Append a `k` or `m` to indicate kilobytes or megabytes, respectively. The default stack size is 400K.

Documentation Generator

The Java and .NET platforms provide facilities for creating API documentation directly from source code. As presented in Chapter 3, both platforms can quickly and easily produce HTML formatted documentation (or, through custom coding or XML transformation, any other format).

Java

Documentation for the Java platform is produced using a program called *javadoc*. This program is located in the Java SDK's *bin* directory.

***javadoc* – Java Documentation Generator**

The *javadoc* program generates HTML documentation for one or more classes and packages. Though it produces HTML by default, by writing a custom *doclet*, it is possible to generate output in any format. The *javadoc* program uses command-line arguments to determine the classes and packages for which documentation should be generated. For example, documentation can be generated for individual source files like this:

```
javadoc c:\source\HelloWorld.java c:\source\Example.java
```

By default, *javadoc* writes documentation to the current directory. To specify a different directory, use the `-d` option like this:

```
javadoc -d c:\docs c:\source\HelloWorld.java
```

Wildcards may be used to document multiple classes like this:

```
javadoc -d c:\docs c:\source\*.java c:\backup\source\Hello*.java
```

The previous command documents all Java source files located in the `c:\source` directory and source files whose names begin with `Hello` in the `c:\source\backup` directory.

By default, *javadoc* searches for source files starting from the current directory. The `-sourcepath` option can be used to explicitly specify the location of source files. Using the `-sourcepath` option, documentation can be generated for entire packages like this:

```
javadoc -sourcepath c:\source java.util java.util.zip
```

The previous command generates documentation for the `java.util` and `java.util.zip` packages located at `c:\source\java\util` and `c:\source\java\util\zip`, respectively. Additionally, multiple source locations can be specified like this on Windows:

```
javadoc -sourcepath c:\source;c:\backup\source java.util
```

Or like this on UNIX:

```
javadoc -sourcepath /source:/backup/source java.util
```

Notice that *javadoc* does not automatically recurse through subpackages. If package recursion is desired, the `-subpackages` option must be specified like this:

```
javadoc -sourcepath c:\source -subpackages java.util
```

Though we have demonstrated them separately, packages and individual source files can be documented simultaneously like this:

```
javadoc -sourcepath c:\source java.util c:\source\HelloWorld.java
```

The previous command documents all source files located in the `java.util` package as well as the `HelloWorld.java` file. Finally, complex *javadoc* commands can be simplified by storing all options and lists of packages and source files in a text file and referencing the file on the command-line like this:

```
javadoc @mydocs.txt
```

Notice that the filename is prefaced with a @ symbol to indicate that the file contains *javadoc* instructions. In addition, multiple files can be specified on the command-line like this:

```
javadoc @mydocs.txt @mydocs2.txt @mydocs3.txt
```

Usage

```
javadoc [options] [packages] [sourceFiles] [@files]
```

The *options* argument may include any of the following *javadoc* options. The *packages* argument consists of one or more fully qualified package names that reside in the current directory or whose location is specified by the `-sourcepath` option. The *sourceFiles* argument may include one or more files containing Java source code. If located in the current directory, source files can be declared simply by their filename; otherwise, source files must be fully qualified. Lastly, the *@files* argument represents one or more text files (prefaced with the @ symbol) containing any number of *javadoc* options, package names, and/or source files.

Options

`-author`

Indicates that information declared using the `@author` tag in the source code should be included in the generated documentation. This option is for the standard doclet only.

`-bootclasspath path`

Specifies the location of system classes. If not specified, the compiler uses the classes located under the Java SDK's `/jre/lib` directory. The *path* argument consists of a semi-colon (on Windows) or colon-delimited (on UNIX) list of directories or JAR/ZIP files.

`-bottom text`

Indicates that the specified *text* should be included at the bottom of each documentation file. This option is for the standard doclet only.

`-breakiterator`

Indicates that the `java.text.BreakIterator` class should be used to determine the first sentence of class member descriptions. This sentence is used to display summary information. This option is for the standard doclet only.

`-charset encoding`

Specifies the character encoding used by the documentation comments within the source code. The specified encoding is added to each documentation HTML file. For example, given an encoding value of `ISO-8859-1`, the following META tag would be added to the top of each documentation file:

```
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

This option is for the standard doclet only.

`-classpath path`

Specifies the location of classes referenced by the source files being documented. This option overrides the `CLASSPATH` environment setting. The *path* consists of an ordered list of directories or JAR/ZIP files delimited by semi-colons on Windows or colons on

UNIX. Unless specified by the `-sourcepath` option, this option also indicates the location of source files.

`-d directory`

Specifies the target directory into and under which documentation files should be placed. If the `-d` option is not set, *javadoc* writes documentation files into and under the current directory. If a source file is defined within a package, *javadoc* automatically creates an equivalent directory structure under the target directory in which to store the documentation file. This option is for the standard doclet only.

`-docencoding encoding`

Specifies the character set encoding to use when generating the documentation files. This option is for the standard doclet only.

`-docfilessubdirs`

Indicates that, in addition to files in the document directory, all subdirectories should be recursively copied. This option is for the standard doclet only.

`-doclet class`

Specifies the custom doclet class to use when generating documentation. The *class* argument represents a fully qualified class name. If this option is not specified, the standard HTML doclet is used.

`-docletpath classpath`

Specifies the location of the doclet class if not specified by the `CLASSPATH` environment variable or the `-classpath` option.

`-doctitle text`

Specifies the title to display at the top of the documentation overview page.

`-encoding encoding`

Specifies the character set encoding used by the source files.

`-exclude packages`

When using the `-subpackages` option, specifies the packages that should be excluded when documentation is generated. The *packages* argument is a colon-delimited list of fully qualified package names.

`-excludedocfilessubdir directories`

When using the `-docfilessubdirs` option, specifies the directories to exclude from the copy operation. The *directories* argument consists of a colon-delimited list of directories relative to the document directory.

`-extdirs directoryList`

Specifies the location of Java extension classes. The *directoryList* argument is a semi-colon-delimited (on Windows) or colon-delimited (on UNIX) list of directories that contain extension JAR files.

`-footer text`

Indicates that the specified *text* should be included as a footer at the bottom of each documentation file. This option is for the standard doclet only.

`-group title packages`

Specifies how packages should be grouped on the package list page. Any number of `-group` options may be specified depending upon the number of groups desired. The *title* argument specifies the title for each group. The *packages* argument is a colon-delimited list of packages that should be included in each group. The `*` wildcard can be used to indicate that subpackages should be included within a group. For example, the following command indicates that documentation should be generated for the `java.util` package and subpackages as well as the `java.awt`, `java.awt.event`, and `java.awt.image` packages:

```
javadoc -group "Utility Packages" java.util* -group "GUI Packages"
java.awt:java.awt.event:java.awt.image
```

This option is for the standard doclet only.

`-header text`

Indicates that the specified *text* should be included as a header at the top of each documentation file. This option is for the standard doclet only.

`-help`

Prints usage and options information.

`-helpfile file`

Specifies a file that contains help information for the documented classes. The generated documentation includes a link to this file.

`-JjavaOption`

Passes *java* options directly to the Java interpreter. The *javadoc* program uses the Java interpreter when generating documentation.

`-link url`

Specifies the base URL to use when linking to previously generated documentation. The directory indicated by the *url* argument must contain a file called *package-list* that contains a list of documented packages. The *package-list* file is automatically generated by *javadoc* each time it is run. This option is for the standard doclet only.

`-linkoffline url packageListUrl`

Same as the `-link` option except that the *package-list* file is located at *packageListUrl* rather than *url*. This option is for the standard doclet only.

`-linksource`

Indicates that an HTML version of each source file should be generated. This option is for the standard doclet only.

`-locale language_country_variant`

Specifies the locale to use when generating documentation. Used to locate locale-specific resource files.

`-nocomment`

- Indicates that documentation should include only generic API information (e.g., method names and signatures), ignoring all Javadoc tags within the source code. This option is for the standard doclet only.
- `-nodeprecated`
- Indicates that documentation should not be generated for deprecated class members. Suppresses generation of the deprecated member list HTML file and all links to it. This option is for the standard doclet only.
- `-nodeprecatedlist`
- Suppresses generation of the deprecated member list HTML file. Unlike the `-nodeprecated` option, does not prevent documentation of deprecated class members. This option is for the standard doclet only.
- `-nohelp`
- Indicates that a help file should not be generated. This option is for the standard doclet only.
- `-noindex`
- Indicates that the documentation index should not be generated. This option is for the standard doclet only.
- `-nonavbar`
- Indicates that navigation bars should be suppressed. Also suppresses headers and footers specified by the `-header` and `-footer` options, respectively. This option is for the standard doclet only.
- `-noqualifier packages`
`-noqualifier all`
- Indicates that, for the specified packages, class names should not include package information (i.e., class names should not be fully qualified). The *packages* argument is a colon-delimited list of package names. The `*` wildcard can be used to indicate subpackages. The `all` argument indicates that package information should be suppressed for all classes, regardless of package. This option is for the standard doclet only.
- `-nosince`
- Indicates that all `@since` tags in the source code should be ignored. This option is for the standard doclet only.
- `-notree`
- Indicates that the class hierarchy page should not be generated. This option is for the standard doclet only.
- `-overview file`
- Inserts a comment contained in the specified *file* into the documentation overview page.
- `-package`
- Document all `public`, `protected`, and package visible classes and class members (excluding `private`).

- `-private`
Document all `public`, `protected`, `package`, and `private` classes and class members.
- `-protected`
Document all `public` and `protected` classes and class members (excluding `package` and `private`).
- `-public`
Document only `public` classes and class members (excluding `protected`, `package`, and `private`).
- `-quiet`
Indicates that all output should be suppressed except for warning and error messages.
- `-serialwarn`
Indicates that a warning message should be displayed for any class that implements `Serializable` but does not include a `@serial` tag.
- `-source 1.4`
Indicates that the source code being documented contains the new `asserts` keyword defined by JDK 1.4.
- `-sourcepath path`
Specifies the location of source files to document. The *path* argument is a semicolon-delimited (on Windows) or colon-delimited (on UNIX) list of directories and JAR/ZIP files.
- `-splitindex`
Creates a separate index for classes and class members that begin with the same letter rather than one large index. This option is for the standard doclet only.
- `-stylesheetfile file`
Specifies a cascading stylesheet document to use when generating the HTML documentation. This option is for the standard doclet only.
- `-subpackages packages`
Indicates that the specified *packages* and their subpackages should be documented. The *packages* argument is a colon-delimited list of fully qualified package names. This option is for the standard doclet only.
- `-tag tagName:locations:headerText`
Specifies information regarding a custom documentation tag within the source code. The *tagName* argument specifies the name of the custom tag and the *headerText* argument specifies text that should immediately precede the text specified by the custom tag in the source code. Finally, the *locations* argument indicates where this custom tag is permitted. Valid values for the *locations* argument include any combination of the following: `a` (all locations permitted), `o` (overview), `p` (packages), `t` (types, i.e. classes and interfaces), `m` (methods), `c` (constructors), and `f` (fields). In addition, including an `x` value in the *locations* argument indicates that the custom tag may be disabled. The order

in which `-tag` options are specified indicates the order in which the tags are shown in the generated documentation. This option is for the standard doclet only.

`-taglet className`

Specifies the name of a taglet class that should be used to process a custom documentation tag. This option is for the standard doclet only.

`-tagletpath classpath`

Indicates the location of taglet classes. The *classpath* argument consists of a colon-delimited list of directories and JAR/ZIP files. This option is for the standard doclet only.

`-use`

Indicates that a page that documents the uses of a package or class should be generated. This option is for the standard doclet only.

`-verbose`

Outputs detailed messages regarding the *javadoc*'s activity.

`-version`

Indicates that version information specified by the `@version` tag should be included in generated documentation. This option is for the standard doclet only.

`-windowtitle text`

Specifies the text that should be displayed in the browser's title bar as the documentation is viewed. This text is placed within the generated HTML page's `<title>` tag. This option is for the standard doclet only.

.NET/C#

Unlike Java, C# does not require a stand-alone application to produce source code documentation. Rather, this functionality is built into the C# compiler (*csc*) that resides in the .NET Framework SDK's *bin* directory. And unlike *javadoc*, *csc* produces documentation in XML format rather than HTML. Once generated, the XML can then be transformed into any desired format, including HTML. Of course, knowledge of XSL stylesheets is required to transform the XML documentation into a suitable format for viewing. Alternatively, Microsoft's Visual Studio.NET product can produce "comment pages" in HTML format with just a few clicks.

csc – C# Compiler and Documentation Generator

In addition to compiling C# source code into intermediate language assemblies, *csc* can also generate API documentation. Documenting source code with *csc* is very simple--simply use the `/doc` option along with the names of the source code files to document like this:

```
csc /doc:mydocs.xml HelloWorld.cs CodeLibrary.cs
```

In addition to compiling the source files into an assembly, this command generates a documentation file named *mydocs.xml* that includes the XML comments embedded in the *HelloWorld.cs* and *CodeLibrary.cs* files.

Usage

```
csc /doc:documentationFile sourceFiles
```

The `/doc` option instructs the `csc` compiler to create an XML documentation file containing the source code comments embedded in the given source files. The `sourceFiles` argument represents a space-delimited list of source file names.

Options

See the “Compiler” section for a complete list of `csc` options.

Packager

For execution and distribution, the C# compiler automatically packages compiled classes into assemblies. An assembly is a single file that contains any number of compiled classes. In contrast, the Java compiler creates a separate file for each compiled class. Unlike C# classes that must reside within an assembly, individual Java class files can be executed on their own. However, it is possible to combine multiple class files into a single executable file or non-executable object library. Though not required for execution, packaging Java classes into a single file is often useful for organization or distribution purposes. For this reason, Java includes a tool that can be used to combine any number of compiled classes into a single file.

Java

The tool that packages compiled classes into a single file is called *jar* (Java Archiver). This program is located in the Java SDK’s *bin* directory.

jar – Java Archiver

The Java archive tool can be used to create an archive file (a.k.a., JAR file), list the contents of an archive file, or extract classes from an archive file. JAR files use the standard ZIP format for packaging and compression. Therefore, it is possible to examine and manipulate JAR files with just about any ZIP utility. JAR files typically consist of compiled class files, resource files, and meta-information regarding individual classes or the archive file itself. JAR meta-information is stored in the *META-INF/manifest.mf* file. Among other things, this meta-information may contain the name of the JAR’s startup class (in the case of an executable JAR), versioning information, library dependencies, or digital signature data (for signed JARs). For example, in order to make a JAR executable, the class file that serves as the application’s entry point must be specified in the manifest file like this:

```
Main-Class: com.sourcestream.apps.MyProgram
```

Usage

```
jar {ctxu}[vfm0Mi] [jar-file] [manifest-file] [-C dir] [files]
```

The `jar` tool accepts numerous and varied command-line arguments. The first argument consists of a single letter that indicates one of four possible operation options (e.g., `c`, `t`, `x`, `u`). These options indicate the operation that the `jar` program should perform. Following the operation option, one or more modifier options may be specified (e.g., `v`, `f`, `m`, `0`, `M`, `i`). Modifier options alter the manner in which the `jar` tool performs the specified operation. Following the modifier options, a `jar` file, manifest file, directory, or

files list may be specified (depending on the given command and modifier options). The *files* argument is a space-delimited list of files or directory names and may include wildcards like this:

```
jar cvf my.jar *.class
```

Using the `*` wildcard, this command creates an archive file containing all of the class files in the current directory. Optionally, the operation and modifier options group may be prefaced with a hyphen (`-`) like this:

```
jar -cvf my.jar *.class
```

Though the hyphen does not serve any specific purpose, it is a common convention among UNIX utilities and may slightly improve readability (serving as a reminder as to which argument represents the operation and modifier options).

Operation Options

`c`

Creates a new archive file.

`t`

Lists contents of the specified archive file.

`x`

Extracts the specified files or directories from an archive. If no files or directories are specified, all files in the archive are extracted.

`u`

Updates the specified archive. Adds files to an archive or, using the `m` option, adds a manifest file.

Modifier Options

`v`

Generates verbose output.

`f`

Indicates that the archive file name is specified on the command-line following the modifier options.

`m`

When using the `c` or `u` operation options, indicates that the contents of the specified manifest file should be added to the JAR's manifest.

`0`

When using the `c` or `u` operation options, creates a JAR file without compression. Note that this option is the number zero, not the letter O.

`M`

When using the `c` or `u` operation options, indicates that a manifest file should not be created.

i

Generates an index of the JAR files that are referenced by the specified JAR. Produces a file called *META-INF/index.list*.

Other Option

-C

Indicates that the JAR should change to the given directory and include the following files and directories. Multiple -C options can be specified and are all interpreted relative to the current directory.

.NET/C#

A separate packager is not required for .NET applications since the C# compiler automatically combines compiled classes into a single assembly file. For example, consider the following compiler command:

```
csc /out:MyLibrary.dll /target:library Source1.cs Source2.cs
```

This command compiles all of the class files defined in the *Source1.cs* and *Source2.cs* files and packages them together in a single library file named *MyLibrary.dll*.

Debugger

The Java and .NET platforms both provide simple command-line-based debugging tools. Among other things, these tools provide a means of stepping through code one line at a time, evaluating local variables, and examining objects on the heap. For these tools to be most effective, the source code must be compiled using the appropriate debug option: “`javac -g`” in Java and “`csc /debug`” for C#.

Java

The Java command-line debug tool is named *jdb* (Java Debugger). This program is located in the Java SDK’s *bin* directory.

jdb – Java Debugger

The Java debugger, *jdb*, is a simple command-line-based tool that assists the development process by allowing the programmer to step through Java source code while evaluating the state of the application. Since *jdb* supports a large number of options and commands, a simple example may be the best way to introduce this tool. A basic debugging session using *jdb* is demonstrated as follows:

1. Launch a console window and move to the directory containing the compiled Java file that you wish to debug. (This file must contain a `public static void main()` method and should be compiled using the `-g` compiler option so that variable information will be included in the compiled file.)
2. Run the *jdb* tool from the command-line like this on Windows:

```
JDK_home\bin\jdb
```

Or like this on UNIX:

JDK_home/bin/jdb

The *jdb* tool will prompt you as follows:

```
Initializing jdb ...
>
```

3. Before running a program from the *jdb* prompt, set a breakpoint so that execution will stop and allow you to step through the program one line at a time. Breakpoints can be set using the “stop at *class:line*” or “stop in *class.method*” syntax. To illustrate, the following command sets a breakpoint at the start of the `main()` method in the `CircleArea` class:

```
> stop in CircleArea.main
```

Another way to get the `CircleArea` class to stop execution at the first line (without explicitly setting a breakpoint) is to run *jdb* like this:

```
jdb -launch CircleArea
```

The `-launch` option indicates that the specified class should be run and execution should stop on the first executable line. When the `-launch` option is used, the `run` command is not necessary (program execution begins automatically).

4. After setting a breakpoint, run the program using the “run `CircleArea`” command and examine the output as follows:

```
> run CircleArea
run CircleArea
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint CircleArea.main

Breakpoint hit: "thread=main", CircleArea.main(), line=5 bci=0
5         float pi = 3.14159f;

main[1]
```

Notice that the debugger notifies you of uncaught exceptions and breakpoints that have been set. The output also indicates that a breakpoint was hit in the `CircleArea.main()` method at line 5 in the source code. Also notice that while debugging, the *jdb* prompt changes to the name of the current thread (`main[1]`).

5. To see the line of source code that is about to be executed as well as a few lines before and after it, use the `list` command as follows:

```
main[1] list
1     class CircleArea
2     {
3         public static void main(String[] args)
4         {
5 =>     float pi = 3.14159f;
6         float radius = 121.9f;
7         float area = pi * radius * radius;
8
9         System.out.print("Area of circle with radius of " + radius);
```

```
10      System.out.println(" is " + area);
main[1]
```

The => symbol represents the instruction pointer indicating that line 5 is the next line to be executed.

6. To execute the next line in the program, issue a `step` command like this:

```
main[1] step
>
Step completed: "thread=main", CircleArea.main(), line=6 bci=3
6      float radius = 121.9f;
main[1]
```

As evidenced by the following `list` command, the `step` command executed line 5 and stopped at line 6:

```
main[1] list
2      {
3      public static void main(String[] args)
4      {
5          float pi = 3.14159f;
6 =>      float radius = 121.9f;
7          float area = pi * radius * radius;
8
9          System.out.print("Area of circle with radius of " + radius);
10         System.out.println(" is " + area);
11     }
main[1]
```

7. Issue two additional `step` commands in order to execute lines 6 and 7.

8. Evaluate the value of the current method's local variables using the `locals` command like this:

```
main[1] locals
Method arguments:
args = instance of java.lang.String[0] (id=296)
Local variables:
pi = 3.14159
radius = 121.9
area = 46682.805
main[1]
```

9. After examining the state of the local variables, continue program execution and exit the debugger (since no other breakpoints were set) using the `cont` command like this:

```
main[1] cont
Area of circle with radius of 121.9 is 46682.805
>
The application exited
```

As you can see, *jdb* can be a very useful tool for debugging simple Java applications. However, a comprehensive integrated development environment (IDE) is usually better suited for debugging large, complex applications.

Usage

```
jdb [options] [class [classArguments]]
```


The *jdb* program can be executed with or without specifying any options or class. The `class` argument (used in conjunction with the `-launch` option) indicates the name of the class to begin debugging. Options can be passed to the program that is to be debugged by specifying a class name and arguments on the *jdb* command-line.

Options

`-attach [host:]port`

Instructs the debugger to attach to a running VM on the specified `host` using the specified `port`. In order to attach to a running VM, the virtual machine must have been started with the appropriate debug settings like this:

```
java -Xdebug -Xnoagent
      -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5000
```

`-connect [connectorName:optionName=value,...]`

Connects to the specified connector and passes it the specified option names and values.

`-dbgtrace [flags]`

Prints *jdb* debug information.

`-help`

Displays usage and options information.

`-launch`

Indicates the name of the program to begin debugging. Automatically stops execution at the first line.

`-listen [host:port]`

Listens for a running VM to connect at the given address.

`-listenany`

Listens for a running VM to connect at any available address.

`-sourcepath directories`

Indicates the directories in which to search for source files. The *directories* argument is a comma-delimited list of directories.

`-tclient`

Indicates that *jdb* should invoke the client-optimized HotSpot VM.

`-tserver`

Indicates that *jdb* should invoke the server-optimized HotSpot VM.

`-version`

Displays *jdb* version information.

Commands

`!!`

Repeat previous command.

numberOfTimes command

Repeats the given command the specified number of times.

catch [uncaught|caught|all] [exceptionClass]

Stops execution when the specified *exceptionClass* is thrown.

class class

Display information about the specified class.

classes

Lists currently loaded classes.

classpath

Displays the classpath for the target VM.

clear

Lists all breakpoints.

clear class.method[(parameterType, ...)]

Clears all breakpoints in the specified method.

clear class:line

Clears the breakpoint at the specified line.

cont

Continue execution from the current breakpoint.

disablegc class

Disables garbage collection for the specified class.

down

Moves down the stack one frame.

down numberOfFrames

Moves down the stack the specified number of frames.

dump object

Prints information for specified object such as methods and variables.

enablegc class

Enables garbage collection for the specified class.

eval expression

Prints the result of the given expression. Same as the `print` command.

exclude class...

Do not display step or method events for the specified class or classes.

exclude none

Clear any previous exclude commands.

`exit`
Exits the debugger. Same as the `quit` command.

`fields class`
Displays the fields of the specified class.

`ignore exceptionClass`
Cancels the `catch` instruction for the specified exception. That is, *jdb* should not stop execution when the specified exception occurs.

`interrupt threadID`
Interrupt the specified thread.

`kill threaded exceptionObject`
Kill the specified thread with the specified exception.

`list`
Displays the source code immediately before and after the next line to be executed.

`list lineNumber`
Displays the source code on and around the specified line number.

`list method`
Displays the source code for the specified method.

`locals`
Display all local variables currently on the stack. To show local variables, the Java class must have been compiled using the `-g` debug option.

`lock object`
Displays lock information for the specified object.

`methods class`
Displays the methods exposed by the specified class.

`monitor`
Displays all monitors.

`monitor command`
Executes the specified command each time program execution stops.

`pop`
Pops the stack through the current frame.

`print expression`
Prints the result of the given expression. Same as the `eval` command.

`next`

Executes the current line, stepping over any method calls. Use the `step` command to step into methods.

`quit`
Exits the debugger. Same as the `exit` command.

`read file`
Read and execute the specified command file.

`redefine classID className`

`reenter`
Pops the stack and reenters the current frame.

`resume threadID...`
Resumes executing the specified thread or threads. If no threads are specified, all are resumed.

`run`
Starts execution of the class specified on the debugger's command-line.

`run class [classArguments]`
Starts execution of the specified class, passing any given arguments to the class.

`set variableName=value`
Assign a new value to the specified variable.

`step`
Executes the current line, stepping into any method calls. Use the `next` command to step over methods.

`stepi`
Executes a single VM instruction.

`step up`
Execute until the current method returns to its caller.

`stop`
Displays all breakpoints.

`stop at class:line`
Sets a breakpoint in the specified class at the specified line.

`stop in class.method[(parameterType,...)]`
Sets a breakpoint in the specified class at the first line of the specified method.

`suspend [threadID...]`
Suspends the specified thread or threads. If no threads are specified, all are suspended.

`thread threadID`
Sets the current thread.

`threadgroup name`
Sets the current thread group.

`threadgroups`
Displays all thread groups.

`threadlocks`
Display lock information for all threads.

`threadlocks threadID`
Displays lock information for the specified thread.

`threads`
List all threads.

`threads threadgroup`
List all threads in given thread group.

`trace methods`
Display trace information for method entry and exit.

`trace methods threadID`
Display trace information for method entry and exits for the specified thread.

`unmonitor monitorNumber`
Delete the specified monitor.

`untrace methods`
Turn off trace for method entry and exit.

`untrace methods threadID`
Turn off trace for method entry and exits for the specified thread.

`unwatch access class.field`
Turns off the watch that indicates when the specified field is accessed.

`unwatch all class.field`
Turns off the watch that indicates when the specified field is accessed or modified.

`up`
Moves up the stack on frame.

`up numberOfFrames`
Moves up the stack the specified number of frames.

`use`
Displays the source path.

`use sourcePath`

Sets the source path.

`watch access class.field`

Indicates each time the specified field is accessed.

`watch all class.field`

Indicates each time the specified field is accessed or modified.

`where`

Displays a stack trace for the current thread.

`where all`

Displays a stack trace for all threads.

`where threadID`

Displays a stack trace for the specified thread.

`wherei`

Displays a stack trace with program counter information for the current thread.

`wherei all`

Displays a stack trace with program counter information for all threads.

`wherei threadID`

Displays a stack trace with program counter information for the specified thread.

`version`

Displays version information for *jdb*.

.NET/C#

The .NET command-line debug tool is called *cordbg.exe* (Common Language Runtime debugger). This program is located in the .NET Framework SDK's *bin* directory.

***cordbg* – .NET Common Language Runtime Debugger**

The .NET debugger, *cordbg*, is a simple command-line-based tool that assists the development process by allowing the programmer to step through C# source code while evaluating the state of the application. Since *cordbg* supports a large number of commands, a simple example may be the best way to introduce this tool. A basic debugging session using *cordbg* is demonstrated as follows:

1. Launch a console window and move to the directory containing the assembly that you wish to debug. (For source-level debugging, this file must have been compiled using the `-debug` compiler option. Only command-line executables, specified with the `/target:exe` option, can be debugged using *cordbg*.)
2. Run the *cordbg* tool from the command-line like this:

```
.NET\Framework\SDK\home\bin\cordbg
```

The *cordbg* program will prompt you as follows:

```
Microsoft (R) Common Language Runtime Test Debugger Shell Version
1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

(cordbg)
```

3. At the (cordbg) prompt, use the `run` command to start the debug session for the desired assembly (i.e., *CircleArea.exe*) like this:

```
(cordbg) run CircleArea.exe
Process 2600/0xa28 created.
Warning: couldn't load symbols for
c:\windows\microsoft.net\framework\v1.1.4322\mscorlib.dll
[thread 0x814] Thread created.

007:      float pi = 3.14159f;
(cordbg)
```

After invoking the `run` command, execution automatically stops at the first line of the `Main()` method without executing it (shown as line number 7).

4. To see the line of source code that is about to be executed as well as a few lines before and after it, use the `show` command as follows:

```
(cordbg) show
002:
003: class CircleArea
004: {
005:     public static void Main()
006:     {
007:*    float pi = 3.14159f;
008:      float radius = 121.9f;
009:      float area = pi * radius * radius;
010:
011:      Console.Write("Area of circle with radius of " + radius);
012:      Console.WriteLine(" is " + area);
(cordbg)
```

The `*` symbol represents the instruction pointer indicating that line 7 is the next line to be executed.

5. To execute the next line in the program, issue a `step` command like this:

```
(cordbg) step

008:      float radius = 121.9f;
(cordbg)
```

The `step` command executes a single line and advances to the next (outputting the next line to be executed). As demonstrated by the following `show` command, the `step` command executed line 7 and stopped at line 8:

```
(cordbg) show
003: class CircleArea
004: {
005:     public static void Main()
006:     {
007:      float pi = 3.14159f;
008:*    float radius = 121.9f;
```

```

009:      float area = pi * radius * radius;
010:
011:      Console.WriteLine("Area of circle with radius of " + radius);
012:      Console.WriteLine(" is " + area);
013:  }
(cordbg)

```

6. Use the `break` and `go` commands to set a breakpoint at line 11 and advance to it (executing lines 8 and 9) like this:

```

(cordbg) break 11
Breakpoint #1 has bound to C:\download\CircleArea.exe.
#1      c:\download\CircleArea.cs:11      Main+0x12(il) [active]
(cordbg) go
break at #1      c:\download\CircleArea.cs:11      Main+0x12(il) [active]

011:      Console.WriteLine("Area of circle with radius of " + radius);
(cordbg)

```

7. Evaluate the value of the `Main()` method's local variables using the `print` command like this:

```

(cordbg) print
pi=3.141590118408203
radius=121.9000015258789
area=46682.8046875
$thread=(0x04a44f88) <System.Threading.Thread>
(cordbg)

```

8. After examining the state of the local variables, complete program execution (since no other breakpoints were set) and exit the debugger using the `go` and `quit` commands like this:

```

(cordbg) go
Area of circle with radius of 121.9 is 46682.8
[thread 0x920] Thread created.
[thread 0xa34] Thread exited.
Process exited.
(cordbg) quit

C:\>

```

As you can see, *cordbg* can be a very useful tool for debugging simple .NET applications. However, a comprehensive integrated development environment (IDE) is usually better suited for debugging large, complex applications.

Usage

```
cordbg [program [programArguments]] [!commands]
```

The *cordbg* program can be executed with or without specifying a program to debug. The *program* argument indicates the name of the assembly to begin debugging. The *commands* argument represents a space-delimited list of commands. Each command must be prefaced by an exclamation point (!). For example, the following line starts the debugger, loads the *CircleArea.exe* assembly, and executes two `step` commands and a `show` command:

```
C:\> cordbg CircleArea.exe !step !step !show
```



```
Microsoft (R) Common Language Runtime Test Debugger Shell Version
1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

```
(corDBG) run CircleArea.exe
Process 3684/0xe64 created.
Warning: couldn't load symbols for
c:\windows\microsoft.net\framework\v1.1.4322\
mscorlib.dll
[thread 0xe68] Thread created.

007:      float pi = 3.14159f;
(corDBG) step

008:      float radius = 121.9f;
(corDBG) step

009:      float area = pi * radius * radius;
(corDBG) show
004: {
005:   public static void Main()
006:   {
007:     float pi = 3.14159f;
008:     float radius = 121.9f;
009:*   float area = pi * radius * radius;
010:
011:     Console.Write("Area of circle with radius of " + radius);
012:     Console.WriteLine(" is " + area);
013:   }
014: }
(corDBG)
```

Of course, the *corDBG* program can be launched without any command-line arguments using the *run* command as demonstrated previously.

Commands

> *[file]*

Writes each command to the specified file. If no file is specified, the debugger stops writing commands to a file.

< *file*

Reads and executes linefeed-delimited commands contained in the specified file.

appdomainenum *[option]*

ap *[option]*

Displays all application domains, assemblies, and modules in the current process. Valid options are *attach* (list application domains and prompt user for the domain to attach to), *detach* (list application domains and prompt user for the domain to detach from), *0* (list application domains), and *1* (list application domains and assemblies). The default option is *1*.

associatesource {*s*|*b breakpointID*} *file*

as {*s*|*b breakpointID*} *file*

Associates the specified source file with the current stack frame (*s* option) or the specified breakpoint (*b* option).

```
attach pid  
a pid
```

Attaches the debugger to a running process specified by the given process identification number (*pid*).

```
break [[[file]:]lineNumber] | [[class::]function[:offset]]]  
b [[[file]:]lineNumber] | [[class::]function[:offset]]]
```

Sets or displays breakpoints. Sets a breakpoint according to the specified file and line number or the given class and function. If no options are set, the `break` command displays all currently set breakpoints. Same as the `stop` command.

```
catch [event]  
ca [event]
```

Displays or sets the events that cause the debugger to break. Valid event values are `e[xceptions]` (all exceptions), `u[nhandled]` (unhandled exceptions), `c[lass]` (class load events), `m[odule]` (module load events), and `t[hread]` (thread start events). If no event is specified, the `catch` command displays a list of event types and whether or not they will be caught. Specific exceptions can be caught by specifying an exception type following the `e[xceptions]` event like this: `catch exceptions System.DivideByZeroException`

```
cont [count]  
con [count]
```

Continues from a breakpoint or caught exception. If the `count` option is specified, execution will continue for the specified number of times (i.e., skip multiple breakpoints or exceptions). Same as the `go` command.

```
delete [breakpointIDs]  
del [breakpointIDs]
```

Deletes some or all breakpoints. The `breakpointIDs` argument is a comma-delimited list of breakpoint IDs to delete. If the `breakpointIDs` argument is not specified, the `delete` command deletes all breakpoints. Same as the `remove` command.

```
detach  
de
```

Detaches the debugger from the current process. After detaching the debugger, the current process continues normal execution.

```
disassemble [0xaddress][{+|-}delta][lineCount]  
dis [0xaddress][{+|-}delta][lineCount]
```

Displays native instructions for the specified `address`. The `lineCount` option specifies the number of instructions to display before and after the specified address (the default is five). The `delta` option indicates a number to add to or subtract from the current instruction pointer.

```
down [numberOfFrames]  
d [numberOfFrames]
```

Moves the stack pointer down the stack the specified number of frames. If not specified, the default number of frames is one.

```
dump address [numberOfBytes]
```

```
du address [numberOfBytes]
```

Dumps a block of memory at the specified *address*. The *numberOfBytes* option indicates the number of bytes to display.

```
exit  
ex
```

Exits the debugger (same as `quit`).

```
funceval [class::]function [args]  
f [class::]function [args]
```

Executes the specified function and stores the return value in the `$result` variable. The `print` command can be used to evaluate the `$result` variable. The *args* option is a space-delimited list of variable names, integers, or the constants `Null`, `True`, and `False`.

```
go [count]  
g [count]
```

Continues from a breakpoint or caught exception. If the *count* option is specified, execution will continue for the specified number of times (i.e., skip multiple breakpoints or exceptions). Same as the `cont` command.

```
help [commands]  
h [commands]  
? [commands]
```

Displays usage instructions for the specified commands. The *commands* option is a space-delimited list of commands for which usage information should be displayed. If no commands are specified, `help` displays a complete list of valid commands.

```
ignore [event]  
ig [event]
```

Displays or clears the events that cause the debugger to break. Valid event values are `e[xceptions]` (all exceptions), `u[nhandled]` (unhandled exceptions), `c[lass]` (class load events), `m[odule]` (module load events), and `t[hread]` (thread start events). If no event is specified, the `ignore` command displays a list of event types and whether or not they will be cause execution to stop. Specific exceptions can be ignored by specifying an exception type following the `e[xceptions]` event like this: `ignore exceptions System.DivideByZeroException`

```
in [numberOfLines]  
i [numberOfLines]
```

Steps to the next line of execution, stepping into method calls. The *numberOfLines* argument specifies the number of lines to advance (the default is one). Same as the `si`, `ssingle`, and `step` commands.

```
kill  
k
```

Stops the current process but the debugger continues to run.

```
list option  
l option
```

Displays a list of loaded modules, classes, or global functions. Valid options are `mod` (modules), `cl` (classes), and `fu` (global functions).

```
mode [[modeName] {0|1}]  
m [[modeName] {0|1}]
```

Displays or sets the debugger mode. Use the `help mode` command to display a list of debugger mode names. Use the `1` argument to turn the mode on and `0` to turn it off.

```
newobj class  
newo class
```

Creates a new object. Stores the object in the `$result` variable.

```
newobjnc class
```

Creates a new object without calling its constructor. Stores the object in the `$result` variable.

```
newstr string  
news string
```

Creates a new string and stores it in the `$result` variable.

```
next [numberOfLines]  
n [numberOfLines]
```

Steps to the next line of execution, stepping over method calls. The `numberOfLines` argument specifies the number of lines to advance (the default is one). Same as the `so` and `nsingle` commands.

```
nsingle [numberOfLines]  
ns [numberOfLines]
```

Steps to the next line of execution, stepping over method calls. The `numberOfLines` argument specifies the number of lines to advance (the default is one). Same as the `next` and `so` commands.

```
out [count]  
o [count]
```

Steps out of the current function. The `count` argument specifies the number of times to step out (the default is one).

```
path [newPath]  
pa [newPath]
```

Displays or sets the path used to search for source files. If `newPath` is specified, the `path` command sets the search path; otherwise, it displays the search path.

```
print [variableName]  
p [variableName]
```

Prints the value of the specified variable. If `variableName` is not specified, the `print` command displays all local variables.

```
processenum  
pro
```

Displays a list of managed processes and application domains.

```
quit  
q
```

Quits the debugger (same as `exit`).

```
refreshsource [sourceFile]  
ref [sourceFile]
```

Reloads the specified source file. Displays usage information if no source file is specified.

```
regdefault [force]  
regd [force]
```

If no other debugger is registered, sets the default debugger to *cordbg.exe*. The *force* argument indicates that the default debugger should be changed to *cordbg.exe* whether or not another debugger has been registered.

```
registers  
reg
```

Displays the CPU register values for the current thread.

```
remove [breakpointIDs]  
rem [breakpointIDs]
```

Deletes some or all breakpoints. The *breakpointIDs* argument is a comma-delimited list of breakpoint IDs to delete. If the *breakpointIDs* argument is not specified, the `remove` command deletes all breakpoints. Same as the `delete` command.

```
resume [~]threadID  
re [~]threadID
```

Resumes execution of the specified thread. The `~` symbol negates the operation by resuming execution of all threads except the specified thread.

```
run [executableFile[args]]  
r [executableFile[args]]
```

Loads and runs the specified executable file, passing it the given arguments (*args*). If a process is running when the `run` command is executed, the process is terminated. If no file is specified, this command terminates the current process and restarts it.

```
set variable value
```

Sets a specified variable to a specified value. The value can be a literal or the name of a variable.

```
setip lineNumber
```

Sets the next line to execute (i.e., instruction pointer) to the specified line number.

```
show [numberOfLines]  
sh [numberOfLines]
```

Displays the source code at, before, and after the current instruction pointer. The *numberOfLines* argument specifies the number of lines to display before and after the current instruction pointer (the default is five).

```
si [numberOfLines]
```

Steps to the next line of execution, stepping into method calls. The *numberOfLines* argument specifies the number of lines to advance (the default is one). Same as the *in*, *ssingle*, and *step* commands.

```
so [numberOfLines]
```

Steps to the next line of execution, stepping over method calls. The *numberOfLines* argument specifies the number of lines to advance (the default is one). Same as the *next* and *nsingle* commands.

```
ssingle [numberOfLines]  
ss [numberOfLines]
```

Steps to the next line of execution, stepping into method calls. The *numberOfLines* argument specifies the number of lines to advance (the default is one). Same as the *in*, *si*, and *step* commands.

```
step [numberOfLines]  
s [numberOfLines]
```

Steps to the next line of execution, stepping into method calls. The *numberOfLines* argument specifies the number of lines to advance (the default is one). Same as the *in*, *si*, and *ssingle* commands.

```
stop [[file:]lineNumber] | [[class:]function[:offset]] | [=0xaddress]
```

Sets or displays breakpoints. Sets a breakpoint according to the specified file and line number or the given class and function. If no options are set, the *stop* command displays all currently set breakpoints. Same as the *break* command.

```
suspend [~]threadID  
su [~]threadID
```

Suspends execution of the specified thread. The *~* symbol negates the operation by suspending execution of all threads except the specified thread.

```
threads [threadID]  
t [threadID]
```

Sets the current thread to the specified thread. If no thread ID is specified, displays a list of active threads.

```
uclear threadID  
uc threadID
```

Clears the current unmanaged exception for the specified thread.

```
up [numberOfFrames]
```

Moves the stack pointer up the stack the specified number of frames. If not specified, the default number of frames is one.

```
uthreads [threadID]  
ut [threadID]
```

Sets the current unmanaged thread to the specified thread. If no thread is specified, lists all unmanaged threads.

```
uwhere  
uw
```

Displays a stack trace for unmanaged code.

```
where [numberOfFrames]  
w [numberOfFrames]
```

If the *numberOfStackFrames* argument is not specified, displays a complete stack trace for the current thread. Otherwise, a stack trace containing the specified number of stack frames is displayed.

```
writememory address numberOfBytes bytes  
wr address numberOfBytes bytes
```

Using the *bytes* argument, writes the specified number of bytes to the specified memory *address*. Multiple values can be written using a single command by comma-delimiting the *address numberOfBytes bytes* options.

```
wt
```

Starting from the current instruction pointer, steps the application one native instruction at a time. Prints a call tree as it steps.

```
x moduleName!searchString
```

Displays symbols in the specified module that match the given search string. The * character can be used as a wildcard to match any string.

Disassembler

As previously discussed, Java and .NET programs are compiled into an intermediate format that can be quickly compiled to native code or interpreted. This intermediate format is known as *bytecode* in Java and *intermediate language* in .NET. At times, it may be useful to view the bytecode or intermediate language for a compiled class or assembly. Fortunately, Java and .NET both provide programs for this purpose. Known as *disassemblers*, these programs read compiled files and output human-readable bytecode or intermediate language. Examining the bytecode or intermediate language may help debug an application or offer insight into how it was optimized by the compiler. See Chapter 2 for a demonstration of the Java and .NET disassemblers.

Java

The Java disassembler is named *javap* (Java Debugger). This program is located in the Java SDK's *bin* directory.

javap – Java Disassembler

In addition to disassembling compiled Java code, *javap* also displays the fields and methods exposed by a class. For example, consider the following program:

```
class HelloWorld  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World");  
    }  
}
```

After compilation, the API for this class can be viewed using the following command:

```
javap HelloWorld
```

This command produces the following output:

```
Compiled from HelloWorld.java
class HelloWorld extends java.lang.Object {
    HelloWorld();
    public static void main(java.lang.String[]);
}
```

As you can see, *javap* indicates that the `HelloWorld` class extends from the default `java.lang.Object` class. In addition, a default no-parameter constructor is provided and a `public main()` method is exposed. In addition to a class's API, the bytecode instructions for the compiled program can be displayed using the `-c` option like this:

```
javap -c HelloWorld
```

This command displays the following bytecode:

```
Method HelloWorld()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 return

Method void main(java.lang.String[])
  0 getstatic #2 <Field java.io.PrintStream out>
  3 ldc #3 <String "Hello World">
  5 invokevirtual #4 <Method void println(java.lang.String)>
  8 return
```

Of course, an understanding of the Java bytecode format would be required to effectively read this output. See Chapter 2 for a more thorough demonstration of the disassembler and an explanation of the generated bytecode.

Usage

```
javap [options] classes...
```

The *classes* argument represents a space-delimited list of classes. The optional *options* argument consists of one or more of the following options.

Options

`-b`

Alters the output format to match that produced by the *javap* program that shipped with JDK 1.1 (enables backward compatibility).

`-bootclasspath path`

Specifies the location of system classes. If not specified, *javap* uses the classes located under the Java SDK's */jre/lib* directory. The *path* argument consists of a semi-colon (on Windows) or colon-delimited (on UNIX) list of directories or JAR/ZIP files.

`-c`

Disassemble the specified class or classes and print the human-readable bytecode to standard out.

`-classpath path`

Sets the CLASSPATH in which to find the specified classes or class dependencies. The *path* argument consists of an ordered list of directories or JAR/ZIP files delimited by semi-colons on Windows or colons on UNIX.

`-extdirs directoryList`

Specifies the location of Java extension classes. The *directoryList* argument is a semi-colon-delimited (on Windows) or colon-delimited (on UNIX) list of directories that contain extension JAR files.

`-help`

Prints usage and options information to standard out.

`-JjavaOption`

Passes the specified option directly to the JVM.

`-l`

Displays a list of line numbers and local variables relative to the disassembled class. Class must be compiled using the `-g` debug option.

`-package`

Displays `package`, `protected`, and `public` class members. This is the default.

`-private`

Displays all class members.

`-protected`

Displays all `protected` and `public` class members.

`-public`

Displays all `public` class members.

`-s`

Displays variables in the internal VM format rather than source code format.

`-verbose`

Displays additional information about class members.

.NET/C#

The .NET disassembler is named *ildasm.exe* (Intermediate Language Disassembler). The *ildasm* program can be displayed as a Windows GUI application (which is the default) or run completely from the command line (using the `/out` or `/text` options). In addition, .NET also includes an assembler called *ilasm* (Intermediate Language Assembler) that can be used to assemble valid intermediate language (IL) files. These programs are located in the .NET Framework SDK's *bin* directory.

ildasm – Intermediate Language Disassembler

In addition to disassembling compiled IL code, *ildasm* also displays graphically the fields and methods exposed by a class. For example, consider the following program:

```
public class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello World");
    }
}
```

After compilation, details about this class can be viewed using the following command:

```
.NET_Framework_SDK\bin\ildasm HelloWorld.exe
```

This command launches the *ildasm* program which, in turn, displays information about the contents of the *HelloWorld.exe* file as shown here:

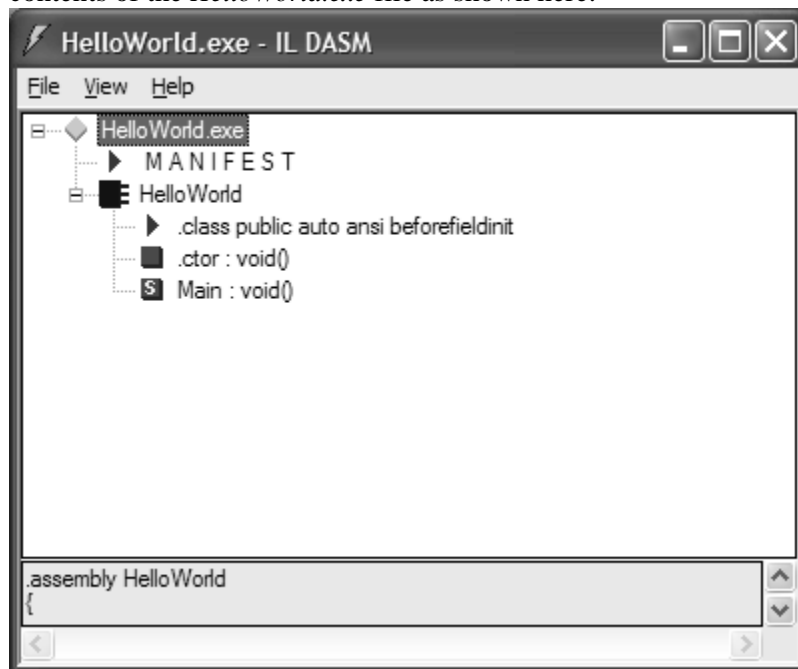
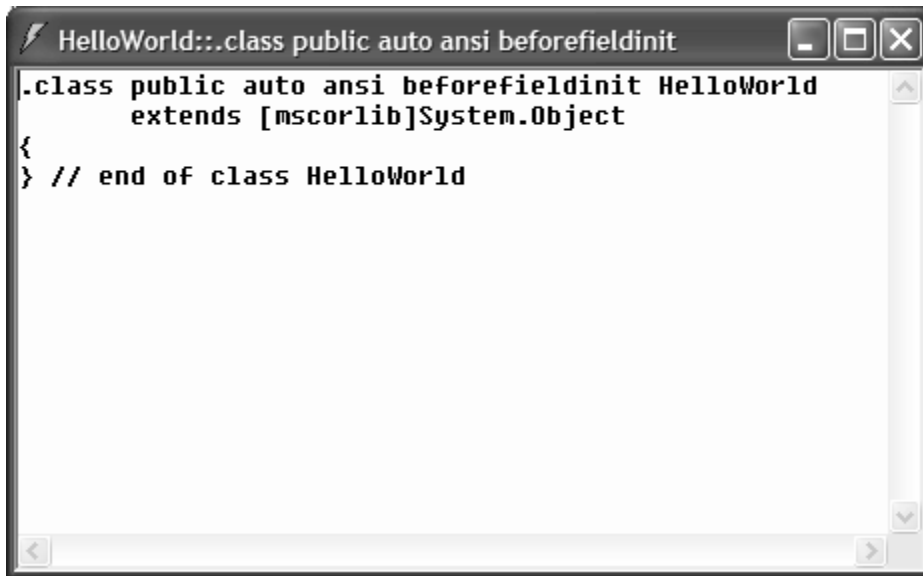


Figure 6.1 Examining *HelloWorld.exe* file using *IL DASM*

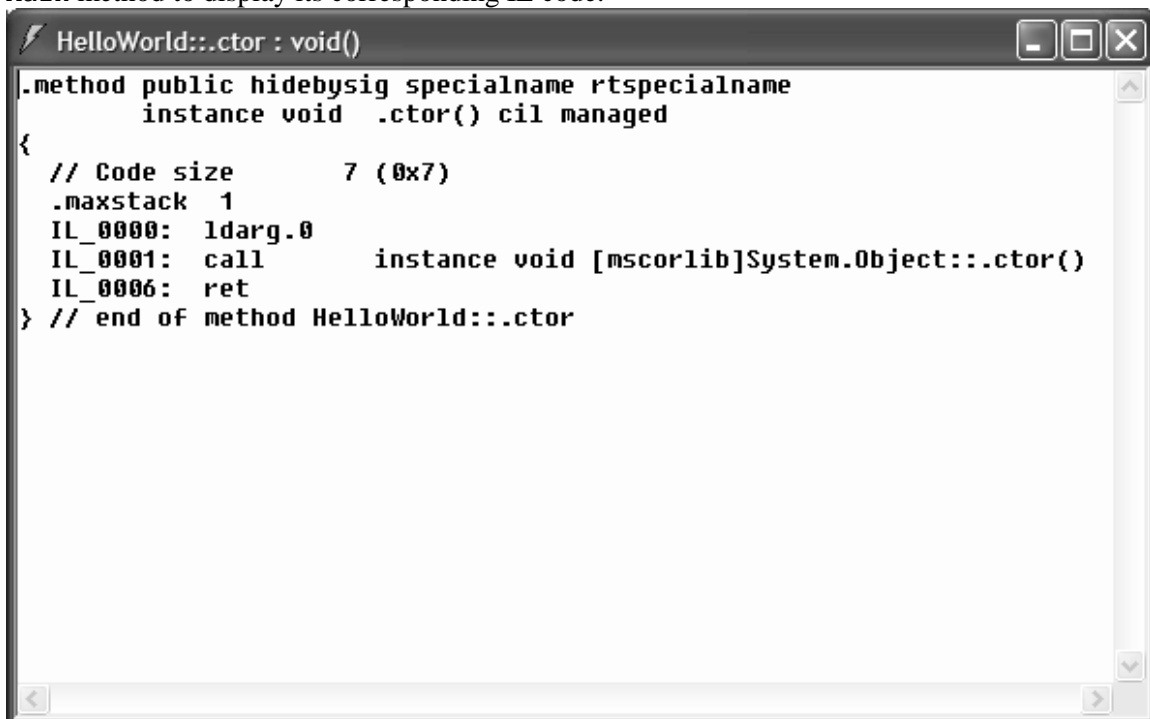
As illustrated in Figure 6.1, the *HelloWorld.exe* file includes a manifest (containing information about the assembly) and a single class named *HelloWorld*. Double-clicking on the *.class* item listed under the *HelloWorld* class displays further class information as shown in Figure 6.2.

A screenshot of the IL DASM application window. The title bar reads 'HelloWorld::.class public auto ansi beforefieldinit'. The main text area contains the following IL code:

```
.class public auto ansi beforefieldinit HelloWorld
    extends [mscorlib]System.Object
{
} // end of class HelloWorld
```

Figure 6.2 Information produced by IL DASM regarding the HelloWorld class

As you can see from Figure 6.2, the HelloWorld class is declared public and extends the System.Object class contained in the mscorlib DLL. Figure 6.1 also indicates that the HelloWorld class contains a single constructor (.ctor) and a Main method, both of which return nothing (void) and accept no parameters. Double-click on the constructor (.ctor) or Main method to display its corresponding IL code.

A screenshot of the IL DASM application window showing the IL code for the HelloWorld constructor. The title bar reads 'HelloWorld::.ctor : void()'. The main text area contains the following IL code:

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size          7 (0x7)
    .maxstack 1
    IL_0000: ldarg.0
    IL_0001: call        instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
} // end of method HelloWorld::.ctor
```

Figure 6.3 IL code for the HelloWorld constructor

Figure 6.3 displays the IL code for the HelloWorld class's default constructor. Remember that when no constructor is explicitly defined, the C# compiler will generate a default no-parameter constructor. As shown in Figure 6.3, the default constructor simply calls its parent class's constructor (System.Object's constructor).

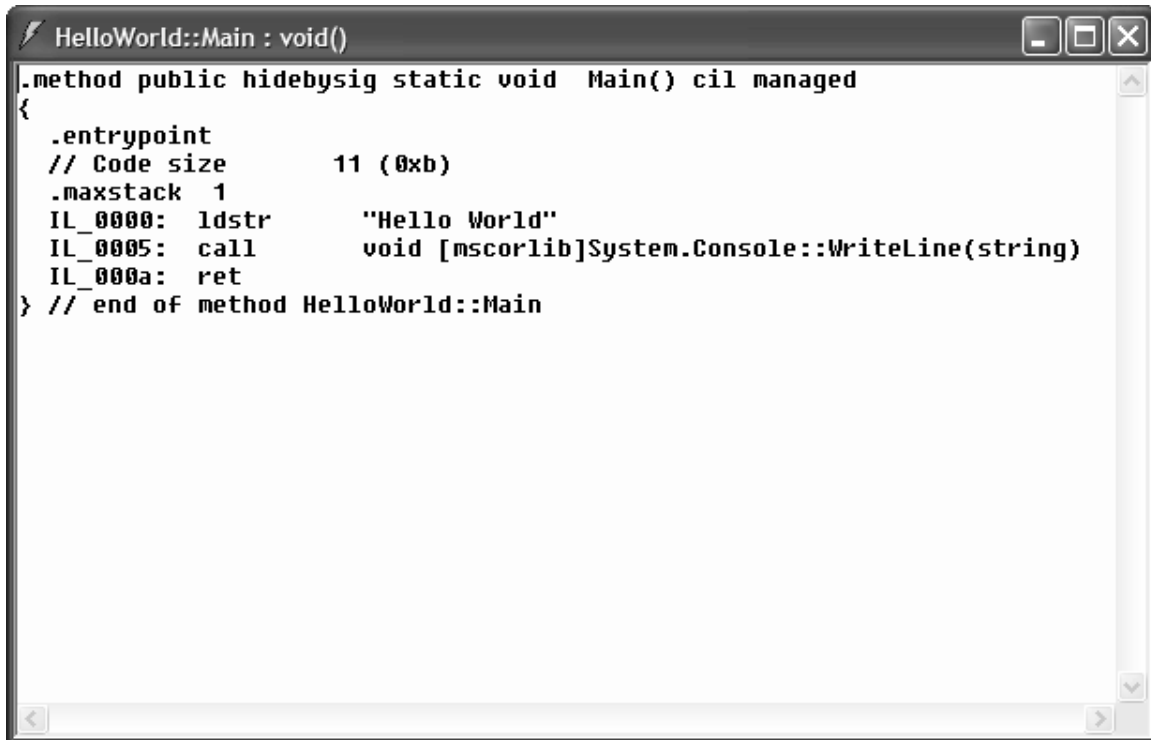


Figure 6.4 IL code for the *HelloWorld* class's *Main()* method

As shown in Figure 6.4, the `Main()` method is marked as the assembly's entry point (`.entrypoint`) and consumes 11 bytes of memory. The IL code indicates that this method simply loads the string "Hello World" onto the stack and then calls the `System.Console` class's `WriteLine()` method in order to write the string to the console. The method exits with a return call (`ret`).

The *ildasm* program can be a valuable tool when debugging .NET classes (especially when the source code is not available). Of course, an understanding of the IL format is necessary to fully appreciate the information provided by the *ildasm* program. See Chapter 2 for a more thorough demonstration of the disassembler and an explanation of the generated IL code.

Usage

```
ildasm [options] [compiledFile] [options]
```

The *ildasm* program loads the specified *compiledFile* (i.e., assembly or module) using the given *options*. If the *options* and *assembly* are not specified, the *ildasm* program launches without loading a file and allows the user to choose a file using a graphical file dialog box. Options may precede or follow the *compiledFile* argument.

Options

```
/?
```

Displays usage and options information.

```
/all
```

Combines the `/bytes`, `/header`, and `/token` options. Valid only in conjunction with `/out` or `/text` options.

```
/bytes
/byt
```

Shows actual bytes in hexadecimal as comments in the IL.

```
/header  
/hea
```

Indicates that file header information should be included in the output. Valid only in conjunction with `/out` or `/text` options.

```
/item=class[:method[(signature)]]  
/ite=class[:method[(signature)]]
```

Disassembles only the specified class or method. Optionally, a method signature may be specified. Valid only in conjunction with `/out` or `/text` options.

```
/linenum  
/lin
```

Displays the original source code line numbers in the output. Requires the file to have been compiled with the `/debug` option. Valid only in conjunction with `/out` or `/text` options.

```
/nobar  
/nob
```

Suppresses the progress meter that is displayed by default during disassembly.

```
/output:file  
/out:file
```

Writes output to the specified *file* rather than displaying a GUI.

```
/noil  
/noi
```

Suppresses the output of IL code. Valid only in conjunction with `/out` or `/text` options.

```
/pubonly  
/pub
```

Disassembles public members only. Same as `/visibility:PUB` option.

```
/quoteallnames  
/quo
```

Indicates that all names should be surrounded by single quotes.

```
/raweh  
/raw
```

Indicates that exception handling clauses should be displayed in raw form.

```
/source  
/sou
```

Displays line numbers from original source code as comments.

```
/text  
/tex
```

Writes output to the console rather than displaying a GUI.

```
/tokens
```

```
/tok
```

Displays metadata tokens for classes and members.

```
/unicode  
/uni
```

Encodes the output in Unicode. Valid only in conjunction with `/out` or `/text` options.

```
/utf8  
/utf
```

Encodes the output in UTF-8. The default is ANSI. Valid only in conjunction with `/out` or `/text` options.

```
/visibility=values  
/vis=values
```

Displays only the types or members specified by *values*. Valid *values* are ASM (assembly), FAA (family and assembly), FAM (family), FOA (family or assembly), PRI (private), PSC (private scope), and PUB (public).

ildasm – Intermediate Language Assembler

In addition to the *ildasm* disassembler, .NET also includes an assembler called *ilasm* (Intermediate Language Assembler). The *ildasm* and *ilasm* utilities are often used in conjunction. That is, an assembly may be disassembled using *ildasm*, altered in some manner (e.g., metadata attributes added), and then reassembled using *ilasm*. To illustrate, the following *ildasm* command disassembles the *HelloWorld.exe* file and creates an IL file called *HelloWorld.il*:

```
ildasm /out:HelloWorld.il HelloWorld.exe
```

In turn, the following *ilasm* command reassembles the *HelloWorld.il* file into a *HelloWorld.exe* file:

```
ilasm HelloWorld
```

Usage

```
ilasm [options] file1 [[options] file2 ...]
```

With *ilasm*, you can specify one or more files (each with their own set of options) to assemble. All files and options are space-delimited.

Options

```
/?
```

Displays usage and options information.

```
/alignment=integer  
/ali=integer
```

Overrides the `.alignment` IL directive and sets the `FileAlignment` property in the NT Optional Header to the specified integer value.

```
/base=integer  
/bas=integer
```

Overrides the `.imagebase` IL directive and sets the `ImageBase` property in the NT Optional Header to the specified integer value.

```
/clock  
/clo
```

Displays compilation times in milliseconds for the following operations: total run, startup, parsing, emitting metadata, resolving references to definitions, fixup and linking, file image generation, and writing the image to a portable executable.

```
/debug  
/deb
```

Includes debug information such as local variable names and line numbers in the assembly.

```
/dll
```

Generates a .dll file as output.

```
/exe
```

Generates a .exe file as output. This is the default.

```
/flags=integer  
/fla=integer
```

Overrides the .corflags IL directive and sets the ImageFlags property in the CLR header to the specified integer value.

```
/key:file
```

Specifies the name of the *file* that contains the key pair used to sign the assembly.

```
/key:@keyStore
```

Specifies a key store containing the key pair used to sign the assembly.

```
/listing  
/lis
```

Outputs a listing of IL instructions for the file being assembled.

```
/nologo  
/nol
```

Suppresses output of the startup banner and copyright information.

```
/output:file  
/out:file
```

Specifies the name of the output file. By default, the output file is given the same name as the input file plus a .exe or .dll extension (defaults to .exe unless the /dll option is specified).

```
/quiet  
/qui
```

Suppresses output of assembly progress information.

```
/resource:file  
/res:file
```

Includes the specified resource file (.res extension) in the generated assembly.

```
/subsystem=integer
```

```
/sub=integer
```

Overrides the `.subsystem IL` directive and sets the subsystem value in the NT Optional Header to the specified integer.

Code Signing Tool

The Java and .NET platforms provide tools for digitally signing applications and managing cryptographic keys and digital certificates. Signed applications allow the recipient to verify a program's integrity (i.e., it has not been altered in transit) and authenticity (i.e., the creator's credentials can be confirmed). Cryptographic keys (both public and private) are used to encrypt and decrypt files and, in conjunction with digital certificates, create secure signatures. Both Java JAR files and .NET assemblies can be digitally signed.

Java

The Java utility for managing cryptographic keys is named *keytool*. The program used to sign JAR files is called *jarsigner*. These programs are located in the Java SDK's *bin* directory.

keytool - Key and Digital Certificate Management Tool

The *keytool* utility generates public/private key pairs for use with public key encryption. Once generated, *keytool* assigns a user-defined alias to the key pair and stores it in an encrypted file known as a *keystore*. A password is required to access the private keys in the keystore. In addition to creating the public key, *keytool* creates a digital certificate in which to store the public key. This certificate can be retrieved from the keystore and stored in a signed JAR file as a means of validating the integrity and authenticity of the file.

A typical key pair can be created using the *keytool* like this:

```
C:\>keytool -genkey -keystore mykeystore -alias mykeypair
Enter keystore password: mypassword
What is your first and last name?
[Unknown]: Dustin Callaway
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]:
What is the name of your City or Locality?
[Unknown]: Pleasant Grove
What is the name of your State or Province?
[Unknown]: Utah
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Dustin Callaway, OU=Unknown, O=Unknown, L=Pleasant Grove, ST=Utah,
C=US correct?
[no]: y

Enter key password for <mykeypair>
(RETURN if same as keystore password):

C:\>
```


The previous instructions create a new public/private key pair named `mykeypair` and stores it in the newly created keystore called `mykeystore`. This key pair can then be used by the `jarsigner` tool to sign a JAR file.

Usage

`keytool command options`

The `keytool` utility requires a *command* followed by zero or more *options*. If no options are specified or required options are missing, `keytool` will prompt for their values.

Commands

`-certreq`

Generates a Certificate Signing Request (CSR) in PKCS#10 format. A CSR can be sent to a Certificate Authority (CA) for authentication. After verifying the requestor's identity, the CA will digitally sign and return a certificate that authenticates the requestor's public key. By signing the certificate, the CA vouches for the identity of the certificate holder (i.e., the public key contained in the certificate does indeed belong to the requestor). The signed certificate returned from the CA can be imported into a keystore using the `keytool`'s `-import` command. This command supports the following options: `-alias`, `-file`, `-keypass`, `-keystore`, `-provider`, `-sigalg`, `-storepass`, `-storetype`, `-v`.

`-delete`

Deletes from a specified keystore the key pair associated with the given alias. This command supports the following options: `-alias`, `-keystore`, `-provider`, `-storepass`, `-storetype`, `-v`.

`-export`

Exports a self-signed digital certificate for the public key associated with the specified alias (if created using `-genkey`) or extracts a previously imported digital certificate. This command supports the following options: `-alias`, `-file`, `-keystore`, `-provider`, `-rfc`, `-storepass`, `-storetype`, `-v`.

`-genkey`

Generates a public/private key pair and stores it in the given keystore under the specified alias. The public key is wrapped in an X.509 self-signed certificate. Use the `-export` option to extract the self-signed certificate. This command supports the following options: `-alias`, `-dname`, `-keyalg`, `-keypass`, `-keysize`, `-keystore`, `-provider`, `-sigalg`, `-storepass`, `-storetype`, `-v`, `-validity`.

`-help`

Prints usage and options information to standard out.

`-identitydb`

Imports the trusted keys from a given JDK 1.1.x identity database to the specified keystore. This command supports the following options: `-file`, `-keystore`, `-provider`, `-storepass`, `-storetype`, `-v`.

`-import`

Reads the certificate from the specified certificate file and stores it in the given keystore under the specified alias. This command is often used in conjunction with `-genkey` (to create a key pair) and `-certreq` (to request authentication from a Certificate Authority). This command can be used to import the digital certificate returned from a Certificate Authority in response to a certification request created by `-certreq`. This command supports the following options: `-alias`, `-file`, `-keypass`, `-keystore`, `-noprompt`, `-provider`, `-storepass`, `-storetype`, `-trustcacerts`, `-v`.

`-keyclone`

Copies the key pair associated with the given alias (specified by the `-alias` option) into a new alias (specified by the `-dest` option). This command supports the following options: `-alias`, `-dest`, `-keypass`, `-keystore`, `-new`, `-provider`, `-storepass`, `-storetype`, `-v`.

`-keypasswd`

Changes the password that protects the private key associated with the given alias. This command supports the following options: `-alias`, `-keypass`, `-keystore`, `-new`, `-provider`, `-storepass`, `-storetype`, `-v`.

`-list`

Prints the contents of the keystore entry associated with the given alias to standard out. By default, this command prints the MD5 fingerprint for the digital certificate. The `-v` option converts the output to a human readable format that includes additional information like owner, issuer, and serial number. Use the `-rfc` option to display the certificate in printable encoding format as specified by Internet RFC 1421. The `-rfc` and `-v` options are mutually exclusive. This command supports the following options: `-alias`, `-keystore`, `-provider`, `-rfc`, `-storepass`, `-storetype`, `-v`.

`-printcert`

Reads a certificate from a specified certificate file and prints its contents to standard out in human readable format. This command supports the following options: `-file`, `-v`.

`-selfcert`

Generates an X.509 self-signed certificate for the public key of the specified alias and keystore. This command supports the following options: `-alias`, `-dname`, `-keypass`, `-keystore`, `-provider`, `-sigalg`, `-storepass`, `-storetype`, `-v`, `-validity`.

`-storepasswd`

Sets the password that protects an entire keystore. Also serves as the default password for any private keys that do not specify their own password. The password must be at least 6 characters long. This command supports the following options: `-keystore`, `-new`, `-provider`, `-storepass`, `-storetype`, `-v`, `-validity`.

Options

`-alias` *name*

The alias name under which a key pair or imported certificate should be stored. If not specified, the default is “mykey”.

`-dest` *newAlias*

Used with the `-keyclone` command, specifies the name of the new alias to which the specified key pair (specified by the `-alias` option) should be copied.

`-dname distinguishedName`

Specifies the X.500 distinguished name to be used by the certificate generated by the `-genkey` or `-selfcert` commands.

`-file file`

Specifies either the input or output file depending on the *keytool* command in use. If not specified, defaults to either standard in or standard out.

`-keyalg algorithm`

Used by the `-genkey` command, specifies the type of keys to generate. If not specified, the default is “DSA”.

`-keypass password`

Specifies the password that protects a private key. If not specified, defaults to the password that protects the entire keystore.

`-keysize size`

Used by the `-genkey` command, specifies the length in bits of generated keys. If not specified, defaults to 1024.

`-keystore file`

Specifies the fully-qualified path to the keystore file. If not specified, defaults to a file name *.keystore* in the user’s home directory.

`-new newPassword`

Used with the `-keyclone`, `-keypasswd`, and `-storepasswd` options, specifies the name of the new password.

`-noprompt`

Used with the `-import` command, turns off interactive prompting.

`-provider`

If not listed in the security properties file (i.e., */jre/lib/security/java.security*), specifies the name of the cryptographic service provider’s main class.

`-rfc`

Used with the `-export` and `-list` commands, indicates that output should be generated in the printable encoding format specified by Internet RFC 1421.

`-sigalg algorithm`

Used by the `-certreq`, `-genkey`, and `-selfcert` commands, specifies the algorithm used to sign the digital certificate. If the public key is encrypted using DSA, the default is “SHA1withDSA”. If encrypted with RSA, the default is “MD5withRSA”.

`-storepass password`

Specifies the password used to protect the entire keystore. Also serves as the default password for any private keys that do not specify their own passwords. The password must be at least 6 characters long.

`-storetype type`

Specifies the type of the keystore to be used. Defaults to storing in a file in the JKS (Java Keystore) format.

`-trustcacerts`

Used with the `-import` command, indicates that the Certificate Authority public keys stored in the `jre/lib/security/cacerts` file should be considered trusted.

`-v`

Indicates that all relevant *keytool* commands should produce additional output (verbose mode).

`-validity time`

Used with the `-genkey` and `-selfcert` commands, specifies in days the amount of time the generated key is valid. If not specified, defaults to 90 days.

jarsigner - JAR Signing and Verification Tool

The *jarsigner* tool is used to sign JAR files and verify their authenticity. When signing a JAR file with *jarsigner*, the user must specify the alias associated with the public/private key pair that is to be used to sign the file as well as the name of the keystore in which the key pair resides. The *jarsigner* tool generates a hash (or digest) corresponding to each class file in the JAR and stores the hash values in a signature file (*.SF* file extension) under the JAR's *META-INF* directory. Additionally, the *jarsigner* tool creates a signature block file (*.DSA* file extension) that contains a signed hash of the signature file plus the digital certificate containing the public key that corresponds to the private key that was used to sign the signature file.

In addition to signing JAR files, *jarsigner* can be used to verify that a JAR file has not been altered since it was signed. Using the digital certificate stored in the signature block file, the verification process begins by validating the signed hash of the JAR's signature file. Only a valid public key can verify (i.e., decrypt) a file that was signed by its corresponding private key. Once the signature file is verified, a hash is computed for each class file in the JAR and compared against the hash that is stored in the signature file. If the hash values do not match, verification fails and the user is notified.

To illustrate, a JAR file named *HelloWorld.jar* can be signed like this:

```
C:\>jarsigner -keystore mykeystore HelloWorld.jar mykeypair
Enter Passphrase for keystore: mypassword
C:\>
```

Once signed, the integrity of the JAR file can be verified like this:

```
C:\>jarsigner -verify HelloWorld.jar
jar verified.
C:\>
```

A verified JAR file indicates that the file has not changed since it was signed.

Usage

```
jarsigner [options] jarFile keystoreAlias  
jarsigner -verify [options] jarFile
```

By default, the *jarsigner* tool signs the given JAR file using the private key associated with the specified alias. Command-line options can be specified to convey required information such as the name and password for the keystore that contains the key pair that should be used to sign the JAR. If not specified on the command-line, *jarsigner* interactively prompts for all required information. Using the `-verify` option, the *jarsigner* tool can be used to verify the authenticity of a signed JAR file.

Options

`-certs`

If specified in conjunction with the `-verbose` or `-verify` options, displays information about the digital certificate(s) included in the signed JAR file.

`-internalsf`

Indicates that a complete encoded copy of the signature file (*.SF* file) should be stored in the signature block file (*.DSA* file). This was the default behavior in previous Java versions but, in order to reduce the size of signed files, the signature file is no longer stored in the signature block file by default. This option is rarely used.

`-JjvmOption`

Passes the specified JVM option through to the Java interpreter.

`-keypass password`

When signing a JAR file, specifies the password for the private key if different from that of the keystore. If different from that of the keystore, *jarsigner* interactively prompts for this password.

`-keystore url`

Specifies the name and location of the keystore file. If not specified, defaults to a file named *.keystore* stored in the user's home directory.

`-provider`

If not listed in the security properties file (i.e., */jre/lib/security/java.security*), specifies the name of the cryptographic service provider's main class.

`-sectiononly`

Indicates that a hash (or digest) for the entire manifest file should not be computed and stored in the signature file (*.SF* file).

`-sigfile filename`

Specifies the name to be used for the signature and signature block files. For instance, if the *filename* is declared as *secure*, the signature and signature block files will be named *secure.SF* and *secure.DSA*, respectively. If not specified, these files will be given the same name as the keystore alias (up to 8 characters).

`-signedjar filename`

Specifies the name of the signed JAR file that is created by the *jarsigner* tool. If the `-signedjar` option is not specified, the original JAR file will be overwritten by the signed JAR.

`-storepass password`

Specifies the password that protects the given keystore. If not specified, *jarsigner* will interactively prompt for the password.

`-storetype type`

Specifies the type of keystore to be used. The default is the Java Keystore (JKS) type that stores key pairs in a password-protected file.

`-verbose`

Indicates that all *jarsigner* commands should produce additional output.

`-verify`

Indicates that the authenticity of the specified signed JAR file should be verified.

.NET/C#

The .NET utility for signing and verifying assemblies as well as managing cryptographic keys is named *sn.exe*. Additionally, assemblies that are to be constructed from existing modules can be built and signed using the *al.exe* program. Both of these programs are located in the .NET Framework SDK's *bin* directory.

sn - Key Creation and Verification Tool

The *sn.exe* program is known as the *Strong Name Tool*. A *strong name* is created from a combination of the assembly's simple text name, version number, culture information, and public key. Every signed assembly can be uniquely identified by its strong name.

The *sn.exe* tool is used to create cryptographic key pairs, sign assemblies, and verify the integrity of signed assemblies. In addition, *sn.exe* manages Cryptographic Service Providers (CSP). A CSP provides concrete implementations of the cryptographic algorithms required for signing and verifying assemblies.

Assembly signing works as follows. First, a hash is computed for the entire assembly. A *hash* is a very large number that represents the current state of a file. The hash can be used to determine if a file has changed because the slightest modification to a file will result in a significant change in its hash value. If two files produce the same hash value, you can be confident that the files are exactly the same. After the hash is created, it is signed (i.e., encrypted) using your private key and stored (along with your public key) in the assembly manifest. A signed assembly is more secure because before loading the assembly, the CLR verifies that it has not changed from the time it was created. The CLR verifies the assembly by decrypting the original hash value using the attached public key and comparing it with a hash value that it computes itself. If the hash values do not match, the assembly has been altered and will not be loaded.

Typically, signing an assembly involves a simple three step process. First, create a key file named *mykeyfile.snk* containing a public/private key pair using the *sn.exe* tool like this:

```
sn -k mykeyfile.snk
```

Second, create an assembly attribute file named *AssemblyInfo.cs* that contains information about the assembly including the name of the key file like this:

```
using System.Reflection;

[assembly: AssemblyTitle("HelloWorld")]
[assembly: AssemblyDescription("A signed HelloWorld assembly")]
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyKeyFile("mykeyfile.snk")]
```

And third, compile all relevant files (including the *AssemblyInfo.cs* file) into a signed assembly like this:

```
csc AssemblyInfo.cs HelloWorld.cs
```

Lastly, you can verify that the assembly is signed by loading it into the *ildasm* disassembler. Once loaded, double-click on the MANIFEST item to display the assembly manifest. Notice that the manifest of a signed assembly includes a public key as shown in Figure 6.5.

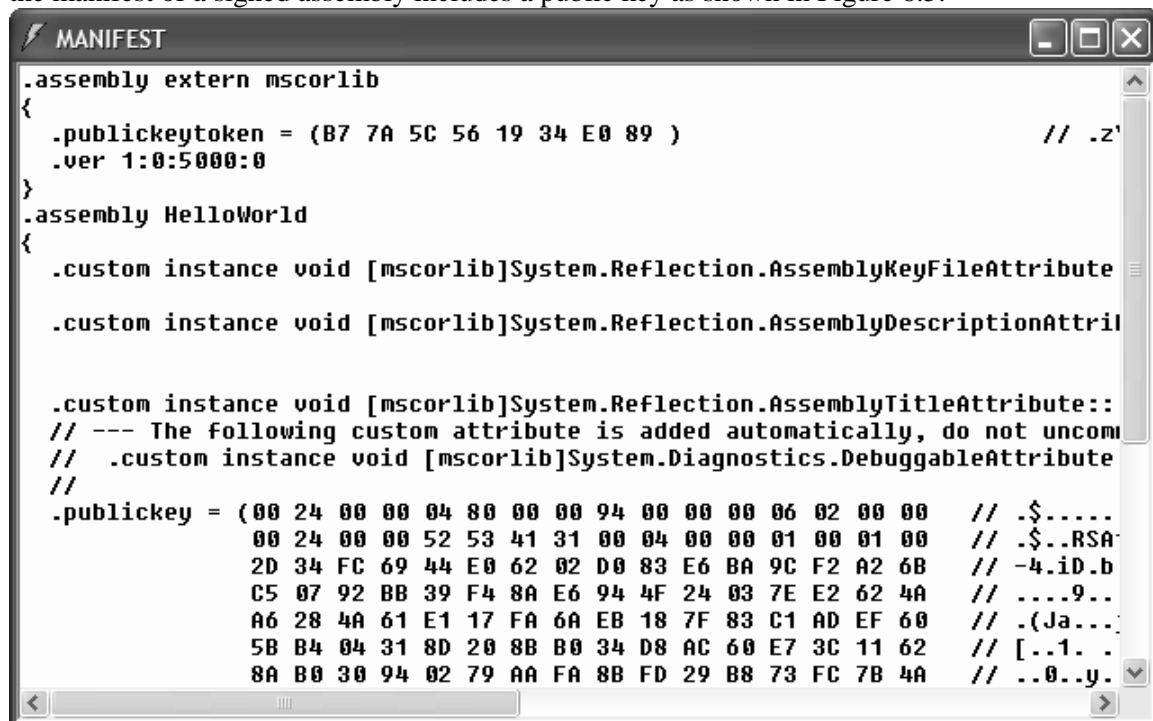


Figure 6.5 Manifest for the signed *HelloWorld* assembly

You can use the *sn.exe* tool's -p or -pc options to extract the public key from your key file. You can give this key to other parties as verification that an assembly was created by you. Keep in mind that the original key file should be kept secret since it contains your private key. If someone were to obtain your private key, they would be able to sign assemblies as if they were you.

Usage

```
sn [options]
```

If no *options* are specified, the *sn* tool displays usage and options information.

Options

`-c [csp]`

Sets the default Cryptographic Service Provider (CSP). If the *csp* argument is not specified, clears default CSP setting.

`-d keyContainer`

Deletes the specified key container from the default CSP.

`-D assembly1 assembly2`

Compares two assemblies to determine if they differ only by digital signature.

`-e assembly outputFile`

Extracts the public key from the signed *assembly* and stores it in the *outputFile*.

`-help`

`-h`

`-?`

Displays usage and options information.

`-i inputFile keyContainer`

Extracts the key pair from *inputFile* and stores it in *keyContainer*.

`-k outputFile`

Generates a new RSA public/private key pair and writes it to the specified output file.

`-m [y|n]`

Specifies whether key containers are machine- or user-specific. The *y* argument indicates machine-specific key containers while *n* indicates that containers are user-specific. If neither *y* nor *n* is specified, the current setting is displayed.

`-o inputFile [outputFile]`

Extracts the public key from *inputFile* and writes it in *.csv* (comma-separated values) format to *outputFile*. If *outputFile* is not specified, copies the public key to the clipboard.

`-p inputFile outputFile`

Extracts the public key from *inputFile* and writes it to *outputFile* in a format that is compatible with the *al.exe* (assembly linker) application's */keyfile* option.

`-pc keyContainer outputFile`

Extracts the public key from the specified key container and stores it in *outputFile* in a format that is compatible with the *al.exe* (assembly linker) application's */keyfile* option.

`-quiet`

`-q`

Suppresses output of success messages.

`-R assembly inputFile`

Re-signs a previously signed *assembly* using the key pair in *inputFile*.

`-Rc assembly keyContainer`

Re-signs a previously signed *assembly* using the key pair in *keyContainer*.

`-t inputFile`

Displays the token for the public key stored in the specified file.

`-tp inputFile`

Displays the token for the public key and the public key itself that is stored in the specified file.

`-T assembly`

Displays the token (or hash) for the public key that was used to sign the specified assembly.

`-Tp assembly`

Displays the token for the public key and the public key itself that was used to sign the specified assembly.

`-v assembly`

Verifies the integrity of a signed assembly.

`-vf assembly`

Forces verification of the integrity of a signed assembly even if verification is disabled with the `-Vr` option.

`-Vl`

Displays the current settings regarding the verification of signed assemblies.

`-Vr assembly [userList] [inputFile]`

Registers verification skipping for the specified *assembly*. A comma-delimited list of users can be specified. Optionally, *inputFile* contains the public key to use for verification.

`-Vu assembly`

Unregisters verification skipping for the specified *assembly*.

`-Vx`

Unregisters verification skipping for all assemblies.

al - Assembly Linker

The *al.exe* program is used to build assemblies (including the assembly manifest) from modules and/or resource files. It is also used to digitally sign the assemblies that it creates.

A module is created using the compiler's `/target:module` option like this:

```
csc /target:module MyCode.cs
```

The previous command produces a module file named *MyCode.netmodule*. The *MyCode* module can be incorporated into a signed assembly like this:

```
al /keyfile:mykeyfile.snk /out:MyCode.dll MyCode.netmodule
```

Once it is created and signed, the CLR will not load the new assembly if its integrity is compromised (i.e., the file is altered in any way).

Usage

```
al /out:outputFile [options] source
```

The assembly linker produces *outputFile* based on the specified *source* file and *options*. Valid *source* options are documented below.

Sources

```
module[,targetAssembly]
```

Specifies the name of a .NET *module* (created using the C# compiler's */t:module* option) and, optionally, the name of a new assembly file into which the *module* should be copied.

```
/embedresource:resourceFile[,name[,private]]  
/embed:resourceFile[,name[,private]]
```

Embeds the specified resource file into the generated assembly file. Optionally, an internal *name* can be assigned to the embedded resource. By default, embedded resources are public. The *private* argument indicates that the embedded resource should not be available to other assemblies.

```
/linkresource:resourceFile[,name[,targetAssembly[,private]]]  
/link:resourceFile[,name[,targetAssembly[,private]]]
```

Adds a link to the specified resource file to the generated assembly file. Optionally, an internal *name* can be assigned to the linked resource. The *targetAssembly* argument indicates the name of an assembly into which the linked resource should be copied. By default, linked resources are public. The *private* argument indicates that the linked resource should not be available to other assemblies.

Options

```
@file
```

Specifies a file containing commands for the *al.exe* program.

```
/algid:algorithmId
```

Specifies the algorithm used to produce a hash of each file in an assembly (except the assembly manifest). The algorithm ID is expressed in hexadecimal. The valid values are 0x0000 (none), 0x8003 (MD5), and 0x8004 (SHA1). The default algorithm is SHA1.

```
/baseaddress:address  
/base:address
```

Specifies the memory address at which to load a DLL at run time.

```
/bugreport:file
```

Generates a text file containing information necessary for bug reporting.

```
/company:companyName  
/comp:companyName
```

Specifies the company name to store in the assembly's `Company` field. Enclose in double quotations if company name contains spaces.

```
/configuration:text  
/config:text
```

Specifies the *text* to store in the assembly's `Configuration` field. Enclose in double quotations if *text* contains spaces.

```
/copyright:text  
/copy:text
```

Specifies the *text* to store in the assembly's `Copyright` field. Enclose in double quotations if *text* contains spaces.

```
/culture:text  
/c:text
```

Specifies the *text* to store in the assembly's `Culture` field. Enclose in double quotations if *text* contains spaces.

```
/delaysign[+|-]  
/delay[+|-]
```

Indicates whether or an assembly is fully or partially signed. `/delay+` or `/delay` indicates that the assembly should be fully signed. `/delay-` (the default) indicates that the assembly should be partially signed. With a fully signed assembly, *al.exe* creates a hash of the assembly manifest (which, in turn, contains a hash of each file in the assembly) and signs the hash with the private key specified by the `/keyfile` or `/keyname` options. With a partially signed assembly, *al.exe* does not generate a digital signature for the assembly. Rather, it simply reserves space for the assembly to be signed later. This option must be used in conjunction with either `/keyfile` or `/keyname` options.

```
/description:text  
/descr:text
```

Specifies the *text* to store in the assembly's `Description` field. Enclose in double quotations if *text* contains spaces.

```
/evidence:file  
/e:file
```

Embeds the *file* containing security evidence in the assembly.

```
/fileversion:version
```

Specifies the *version* to store in the assembly's `FileVersion` field. Enclose in double quotations if *version* contains spaces.

```
/flags:flags
```

Specifies the *flags* to store in the assembly's `Flags` field. Flags are specified in hexadecimal format and are used to indicate the assembly's ability to run concurrently with other different versions of the assembly. Valid values are `0x000` (side-by-side compatible), `0x0010` (cannot execute with other versions running in the same

application domain), 0x0020 (cannot execute with other versions running in the same process), and 0x0030 (cannot execute with other versions running on the same computer).

```
/fullpaths
```

Indicates that error messages should use absolute paths.

```
/help  
/?
```

Displays usage and options information.

```
/keyfile:file  
/keyf:file
```

Specifies a *file* containing the key pair used to sign the assembly. The public key is stored in the assembly manifest and then the private key is used to sign the assembly.

```
/keyname:keyContainer  
/keyn:keyContainer
```

Specifies a key container containing the key pair used to sign the assembly. The public key is stored in the assembly manifest and then the private key is used to sign the assembly.

```
/main:class.method
```

When a module is converted to an executable, specifies the *class* and *method* to use as the entry point.

```
/nologo
```

Suppresses output of the startup banner and copyright information.

```
/out:outputFile
```

Specifies the name of the file created by *al.exe*.

```
/product:text  
/prod:text
```

Specifies the *text* to store in the assembly's `Product` field. Enclose in double quotations if *text* contains spaces.

```
/productversion:text  
/productv:text
```

Specifies the *text* to store in the assembly's `ProductVersion` field. Enclose in double quotations if *text* contains spaces.

```
/target:exe  
/target:lib  
/target:win
```

Specifies the file format of the output file. Valid values are `exe` (console executable), `lib` (dynamic link library), and `win` (Windows executable).

```
/template:assembly
```

Copies all assembly metadata (except `Culture`) from the specified *assembly*. The specified *assembly* must have a strong name.

`/title: text`

Specifies the *text* to store in the assembly's Title field. Enclose in double quotations if *text* contains spaces.

`/trademark: text`

Specifies the *text* to store in the assembly's Trademark field. Enclose in double quotations if *text* contains spaces.

`/version: assemblyVersion`

Specifies the assembly version in *major.minor.build.revision* format. The default is 0.

`/win32icon: file`

Embeds the specified icon *file* in the assembly for display by various file management utilities (e.g., Windows Explorer).

`/win32res: file`

Embeds the specified Win32 resource *file* in the assembly.

Summary

This chapter introduced many of the most commonly used tools for developing Java and .NET applications. The functionality demonstrated by these tools includes compiling, running, documenting, packaging, debugging, disassembling, and securing programs for the Java and .NET platforms. In addition to a brief introduction, a comprehensive reference documenting the proper usage and options available for each tool was presented.