## NAME

csharp – Interactive C# Shell and Scripting

## SYNOPSIS

**csharp [--attach PID] [-e EXPRESSION] [file1 [file2]]** [compiler-options] [--|-s script-options]

## DESCRIPTION

The *csharp* command is an interactive C# shell and scripting host that allows the user to enter and evaluate C# statements and expressions from the command line or execute C# scripts. The regular *mcs* command line options can be used in this version of the compiler.

Files specified in the command line will be loaded and executed as scripts.

Starting with Mono 2.10, the *csharp* command can be used as an interpreter executed by executables flagged with the Unix execute attribute. To do this, make the first line of your C# source code look like this:

```
#!/usr/bin/csharp
Console.WriteLine ("Hello, World");
```

Starting with Mono 5.0, command line arguments may now be passed to the *csharp* command by specifying either the *-s* or -- (script) options.

The *-s* option is ideal for interpreting executable scripts that utilize shebang syntax (introduced in Mono 2.10). This allows command line arguments to be passed to and consumed cleanly by the script:

```
#!/usr/bin/csharp -s
foreach (var arg in Args)
  Console.WriteLine ($"script argument: {arg}");
```

## OPTIONS

The commands accept all of the commands that are available to the *mcs* command, so you can reference assemblies, specify paths, language level and so on from the command line. In addition, the following command line options are supported:

*−s SCRIPT_FILE*

This option is ideal for authoring executable scripts that utilize the Unix shebang feature. Unix will implicitly append as an argument the path of the script to execute. When the executable is invoked, any arguments then passed to it will be available in the *Args* global. Example: *#!/usr/bin/env csharp -s*

*−−* Any arguments that follow will not be passed to the compiler driver, and instead will be made available in the *Args* global. Example: *csharp -- a b c* will result in Args = { "a", "b", "c" } in the interactive shell.

*−−attach*

This is an advanced option and should only be used if you have a deep understanding of multithreading. This option is availble on the *csharp* command and allows the compiler to be injected into other processes. This is done by injecting the C# shell in a separate thread that runs concurrently with your application. This means that you must take special measures to avoid crashing the target application while using it. For example, you might have to take the proper locks before issuing any commands that might affect the target process state, or sending commands through a method dispatcher.

*−e EXPRESSION*

This will evaluate the specified C# EXPRESSION and exit

## OPERATION

Once you launch the csharp command, you will be greeted with the interactive prompt:

```
$ csharp
Mono C# Shell, type "help;" for help
```

Enter statements below.
csharp>

A number of namespaces are pre-defined with C# these include System, System.Linq, System.Collections and System.Collections.Generic. Unlike the compiled mode, it is possible to add new using statements as you type code, for example:

```
csharp> new XmlDocument ();
<interactive>(1,6): error CS0246: The type or namespace name 'XmlDocument' could not be found. Are you missing a us
csharp> using System.Xml;
csharp> new XmlDocument ();
System.Xml.XmlDocument
```

Every time a command is typed, the scope of that command is one of a class that derives from the class Mono.CSharp.InteractiveBase. This class defines a number of static properties and methods. To display a list of available commands access the 'help' property:

```
csharp> help;
"Static methods:
  LoadPackage (pkg); - Loads the given Package (like -pkg:FILE)
  [...]
  ShowVars ();       - Shows defined local variables.
  ShowUsing ();      - Show active using decltions.
  help;
"
csharp>
```

When expressions are entered, the C# shell will display the result of executing the expression:

```
csharp> Math.Sin (Math.PI/4);
0.707106781186547
csharp> 1+1;
2
csharp> "Hello, world".IndexOf (',');
5
```

The C# shell uses the ToString() method on the returned object to display the object, this sometimes can be limiting since objects that do not override the ToString() method will get the default behavior from System.Object which is merely to display their type name:

```
csharp> var a = new XmlDocument ();
csharp> a;
System.Xml.Document
csharp> csharp> a.Name;
"#document"
csharp>
```

A few datatypes are handled specially by the C# interactive shell like arrays, System.Collections.Hashtable, objects that implement System.Collections.IEnumerable and IDictionary and are rendered specially instead of just using ToString ():

```
csharp> var pages = new Hashtable () {
    >  { "Mono",    "http://www.mono-project.com/" },
    >  { "Linux",   "http://kernel.org" } };
csharp> pages;
{{ "Mono", "http://www.mono-project.com/" }, { "Linux", "http://kernel.org" }}
```

It is possible to use LINQ directly in the C# interactive shell since the System.Linq namespace has been imported at startup. The following sample gets a list of all the files that have not been accessed in a week from /tmp:

```
csharp> using System.IO;
```

```
csharp> var last_week = DateTime.Now - TimeSpan.FromDays (7);
csharp> var old_files = from f in Directory.GetFiles ("/tmp")
    >   let fi = new FileInfo (f)
    >   where fi.LastAccessTime < LastWeek select f;
csharp>
```

You can of course print the results in a single statement as well:

```
csharp> using System.IO;
csharp> var last_week = DateTime.Now - TimeSpan.FromDays (7);
csharp> from f in Directory.GetFiles ("/tmp")
    >   let fi = new FileInfo (f)
    >   where fi.LastAccessTime < last_week select f;
[...]
csharp>
```

LINQ and its functional foundation produce on-demand code for IEnumerable return values. For instance, the return value from a using 'from' is an IEnumerable that is evaluated on demand. The automatic rendering of IEnumerables on the command line will trigger the IEnumerable pipeline to execute at that point instead of having its execution delayed until a later point.

If you want to avoid having the IEnumerable rendered at this point, simply assign the value to a variable.

Unlike compiled C#, the type of a variable can be changed if a new declaration is entered, for example:

```
csharp> var a = 1;
csharp> a.GetType ();
System.Int32
csharp> var a = "Hello";
csharp> a.GetType ();
System.String
csharp> ShowVars ();
string a = "Hello"
```

In the case that an expression or a statement is not completed in a single line, a continuation prompt is displayed, for example:

```
csharp> var protocols = new string [] {
    >   "ftp",
    >   "http",
    >   "gopher"
    > };
csharp> protocols;
{ "ftp", "http", "gopher" }
```

Long running computations can be interrupted by using the Control-C sequence:

```
csharp> var done = false;
csharp> while (!done) { }
Interrupted!
System.Threading.ThreadAbortException: Thread was being aborted
  at Class1.Host (System.Object& $retval) [0x00000]
  at Mono.CSharp.InteractiveShell.ExecuteBlock (Mono.CSharp.Class host, Mono.CSharp.Undo undo) [0x00000]
csharp>
```

## INTERACTIVE EDITING

The C# interactive shell contains a line-editor that provides a more advanced command line editing functionality than the operating system provides. These are available in the command line version, the GUI versions uses the standard Gtk# key bindings.

The command set is similar to many other applications (cursor keys) and incorporates some of the Emacs commands for editing as well as a history mechanism too.

The following keyboard input is supported:

*Home Key, Control-a*
Goes to the beginning of the line.

*End Key, Control-e*
Goes to the end of the line.

*Left Arrow Key, Control-b*
Moves the cursor back one character.

*Right Arrow Key, Control-f*
Moves the cursor forward one character.

*Up Arrow Key, Control-p*
Goes back in the history, replaces the current line with the previous line in the history.

*Down Arrow Key, Control-n*
Moves forward in the history, replaces the current line with the next line in the history.

*Return* Executes the current line if the statement or expression is complete, or waits for further input.

*Control-C*
Cancel the current line being edited. This will kill any currently in-progress edits or partial editing and go back to a toplevel definition.

*Backspace Key*
Deletes the character before the cursor

*Delete Key, Control-d*
Deletes the character at the current cursor position.

*Control-k*
Erases the contents of the line until the end of the line and places the result in the cut and paste buffer.

*Alt-D* Deletes the word starting at the cursor position and appends into the cut and paste buffer. By pressing Alt-d repeatedly, multiple words can be appended into the paste buffer.

*Control-Y*
Pastes the content of the kill buffer at the current cursor position.

*Control-Q*
This is the quote character. It allows the user to enter control-characters that are otherwise taken by the command editing facility. Press Control-Q followed by the character you want to insert, and it will be inserted verbatim into the command line.

*Control-D*
Terminates the program. This terminates the input for the program.

## STATIC PROPERTIES AND METHODS

Since the methods and properties of the base class from where the statements and expressions are executed are static, they can be invoked directly from the shell. These are the available properties and methods:

*Args* An easy to consume array of any arguments specified after either *-s* or -- on the command line. Ideal for self-executing scripts utilizing the *-s* option.

*void LoadAssembly(string assembly)*
Loads the given assembly. This is equivalent to passing the compiler the -r: flag with the specified string.

*void LoadPackage(string package)*
Imports the package specified. This is equivalent to invoking the compiler with the -pkg: flag with the specified string.

*string Prompt { get; set }*
: The prompt used by the shell. It defaults to the value "csharp> ". *string ContinuationPrompt { get; set; }* The prompt used by the shell when further input is required to complete the expression or statement.

*void ShowVars()*
: Displays all the variables that have been defined so far and their types. In the csharp shell declaring new variables will shadow previous variable declarations, this is different than C# when compiled. *void ShowUsing()* Displays all the using statements in effect. *TimeSpan Time (Action a)* Handy routine to time the time that some code takes to execute. The parameter is an Action delegate, and the return value is a TimeSpan. For example:

```
csharp> Time (() => { for (int i = 0; i < 5; i++) Console.WriteLine (i);});
0
1
2
3
4
00:00:00.0043230
csharp>
```

The return value is a TimeSpan, that you can store in a variable for benchmarking purposes.

## GUI METHODS AND PROPERTIES

In addition to the methods and properties available in the console version there are a handful of extra properties available on the GUI version. For example a "PaneContainer" Gtk.Container is exposed that you can use to host Gtk# widgets while prototyping or the "MainWindow" property that gives you access to the current toplevel window.

## STARTUP FILES

The C# shell will load all the Mono assemblies and C# script files located in the ~/.config/csharp directory on Unix. The assemblies are loaded before the source files are loaded.

C# script files are files that have the extension .cs and they should only contain statements and expressions, they can not contain full class definitions (at least not as of Mono 2.0). Full class definitions should be compiled into dlls and stored in that directory.

## AUTHORS

The Mono C# Compiler was written by Miguel de Icaza, Ravi Pratap, Martin Baulig, Marek Safar and Raja Harinath. The development was funded by Ximian, Novell and Marek Safar.

## LICENSE

The Mono Compiler Suite is released under the terms of the GNU GPL or the MIT X11. Please read the accompanying 'COPYING' file for details. Alternative licensing for the compiler is available from Novell.

## SEE ALSO

gmcs(1), mcs(1), mdb(1), mono(1), pkg-config(1)

## BUGS

To report bugs in the compiler, you must file them on our bug tracking system, at: http://www.mono-project.com/community/bugs/

## MAILING LIST

The Mono Mailing lists are listed at http://www.mono-project.com/community/help/mailing-lists/

## MORE INFORMATION

The Mono C# compiler was developed by Novell, Inc (http://www.novell.com, http) and is based on the ECMA C# language standard available here: http://www.ecma.ch/ecma1/STAND/ecma-334.htm

The home page for the Mono C# compiler is at http://www.mono-project.com/docs/about-mono/languages/csharp/ information about the interactive mode for C# is available in http://mono-project.com/docs/tools+libraries/tools/repl/