CSharpatron

smart, thorough
file & project
conversion
(.js to C#)

spoonsized
www.spoonsized.com

Contents

Overview

CSharpatron is a Unity Editor Extension that converts UnityScript (.js) files to C#. It uses .Net type reflection and fully language aware parsing to evaluate and convert script in an *extremely* comprehensive manner. It is able to not only make the basic language formatting adjustments you might expect, but can also handle a huge number of more subtle fixes such as -

- Inference of all variable types.
- Addition of required type-casts.
- Adding default assignments for local variables.
- Re-formatting literals.
- Making (only) necessary namespace inclusions.
- Making logical comparisons explicit for non boolean types.
- Automatically accounting for different local variable scoping rules between UnityScript and C#.

In the majority of cases a converted file will compile, error-free in C# without need for a single manual edit.

The converter takes an approach that aims to fix-up everything it can, whilst pointing you directly at the few things that it cannot, via clear, thorough error logging. The (few) fixes you may need to make are broken down into 'pre conversion' fixes to be made in the original .js file and 'post conversion' fixes to be made in the final .cs file.

CSharpatron is respectful of your file formatting preferences offering automatic sensing or manual selection to define things like brace placement and spacing. Additional options allow you to express various conversion preferences.

To compliment its unique conversion capabilities, CSharpatron also includes a powerful and elegant **Workspace View**. This window is designed to help you in every step of converting your project to C#. It captures all converter and compiler error information, shows external dependency information for a file, and provides various tools to automate work stages. At any time a single button click can toggle between your original .js and converted .cs versions of a file.

Making many new C# scripts active in one go can be a *significant* challenge due to the many interdependencies and circular dependencies that often exist between files. To solve this problem, CSharpatron implements 'file shadowing'. This is a mechanism that allows you to expose converted files to the C# compiler one or two at a time and in any order (while your .js files remain active). You can use this to work through potential compile errors and then make a file fully active once externally referenced types are also compiling happily in their C# form.

CSharpatron's conversion process does not break any mapped game-object or components.

Whether you need to convert just a few files or an entire project - CSharpatron makes it possible to achieve in minutes what would have been days of tedious, error-prone work by hand.

To see examples of CSharpatron's output, see Appendix A.

You can read about my own experience converting a large (107,000 line) project to C# in the Case Study appendix of this manual.

Why convert to C#?

Without wanting to enter into the holy wars of a 'which language is best' debate, I'll set out my personal motivations for moving my Unity codebase from Unityscript to C# - basically the reasons I created CSharpatron…

- I wanted to be able to utilize the more in depth language features of C#, e.g. tighter type control (especially control over use of references), extension methods, namespaces, nullable types, const variables, easier #ifdef capabilities, to name just a few.

- I grew to dislike the many ways that Unityscript almost encourages sloppy programming practices (and thus bugs). As you can see in my Case Study, there are a horrifying number of ways in which Unityscript allows you to shoot yourself in the foot!

- I grew irritated with the fact that Unityscript is an 'unofficial', hard to pin-down language - often confusingly called Javascript but actually unlike it in so many ways. If you encounter problems or confusion with the language there is no official language spec to lookup, no chance of assistance from beyond the Unity community.

- Most people offering Unity contract work are looking for C# programmers. It pained me to feel like I was creating an ever larger pool of potentially reusable project code that I couldn't easily access from C#.

- I found myself utilizing external libraries that were invariably created in C#. Any work to extend these classes kept throwing me against the world of ugliness and hurt that is Unity's language divide (folder based compilation ordering, etc); working with a mix of languages is clumsy and error prone.

- And most of all: the lengthy compile time for my .js project was starting to drive me *insane*. ….In C# form my project now compiles in about 30% of the time it did in Unityscript!

If you're in a similar position and are ready to trade the good-natured but dangerously permissive Unityscript compiler for the harsh but ever watchful mistress that is the C# compiler, then read on!

Usage

To use CSharpatron you should load up a project containing UnityScript files that you wish to convert to C#. Because CSharpatron parses files much like a compiler, **your starting point must be UnityScript files that are in a *fully compilable state***, i.e. free from syntactic errors.

You will also need to ensure that your project's active platform is the one for which you wish to perform conversion, and that the project has been successfully compiled. The reason for these stipulations is that CSharpatron will only convert script that it 'sees', i.e. code that isn't within comments or non-active #if blocks. To better understand the reasons behind this, see 'How Does CSharpatron Understand Variable Types?'. You may also wish to check out CSharpatron's Limitations.

> *Before doing anything else, back up your source! ...Better yet, if you don't do so already, now would be a great time to start using source control!*
>
> CSharpatron *does* makes its own back-ups of your original .js files as it works, but nevertheless, I wouldn't trust someone else's tool with the only copy of my source and neither should you!

Under Unity's Windows menu you should see **'CSharpatron (.js to C#)'.** Select this in order to open up CSharpatron's Control Center panel. This is the place to look for configuration options and your starting point for any file conversions. You'll probably want to dock this window and/or keep it clearly visible during file conversion.

The next section will discuss the Conversion Process. Before attempting your first real file conversions you may prefer to read about Configuration first.

> *You can watch a video tutorial demonstrating a basic project conversion here.*

Conversion Process

The basic steps of the conversion process are:

> - *Convert files*
> - *Review error log ⟺ fix conversion errors and re-convert*
> - *Activate C# files ⟺ fix C# compile errors*

In more detail, this is the process for using CSharpatron:

1. Select file(s).
- Highlight files or folders in your Project Window. You can use multi-select or click on one or more folders to include their contents.
- If you intend to convert an entire project, it's a good idea to select them all in one go.

2. Hit the 'Convert' button.
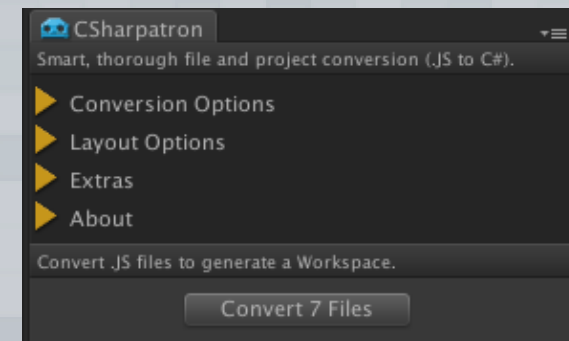
Once conversion completes, review errors using the Workspace View.
- Alternatively you can look at the conversion log: CSharpatronWork/Log.txt

At this point your files have all been converted to C#, but none of them are 'live' yet. Instead, in the same location as each original .js file, you will find a .csTest file of the same name.

If you chose to convert only a subset of your files initially, you can add more at any time.

With a simple project you may see no conversion errors at all (e.g. Unity's Penelope example project), in which case you can jump straight to step 5.

If you do see errors, especially if these are of 'Fix .JS' types, then there's no way those files will compile just yet, so the next thing to do is to fix them...

4. Go through all your workspace files and fix any 'Fix .JS' errors that may be flagged.
- Use the workspace view to see a list of warnings and errors. Click on a line number to jump to that error (assuming you have a compliant External Script Editor configured).
- After making a fix you'll need to wait for the file to be rebuilt by the Unityscript compiler.
- **ReConvert** each affected file.
- Repeat until there are no such errors.
- NOTE: you may still have a few regular errors but most likely these are all things that will fall away or are better addressed in the next phase.

Now you just need to get your freshly prepared C# files *compiling...*

5. Start by finding files that are error free and have zero X-Refs (external references). Individually, or via group-selection (holding shift), make these files 'live'.
- Click on the Workspace View's 'sort by X-Refs' option to order files by X-Refs.
- Each time you take a file live, it will be seen by the C# compiler and will auto-compile. CSharpatron will capture any error output.
- To understand what happens when a file is made live, see 'How A File Goes Live'.
- For files that CSharpatron thinks were converted without error, those files will usually compile without error too; any issues are likely to be very minor. Fix any problems directly in the .cs files.
- You can actually edit either the .cs or .csTest versions of a file interchangeably according to whether the file is 'live' or nor. Remember: use the Workspace error list or context menu to jump to a file.
- Building one (or a few) files at a time makes working through any errors simpler.
- Each time you activate (or deactivate) files you will need to wait for the compiler to finish; CSharpatron deliberately blocks further file-state changes during compilation to reduce the risk of you encountering confusing errors or leaving your project in a broken state.

6. With your X-Ref free files now compiling under C# you will likely see more files whose own X-Refs are now 'fulfilled'. You can take these files live too. Work through as many zero X-Ref files as you can.
- Click again on 'sort by X-Refs' re-sort following changes in file activity.

Even with a simple project, you may reach a point where there are no more 'low hanging fruit' and your files start to have unfulfilled X-Refs. Ordinarily you'd now have to start hacking files around to try and circumvent dependency problems. However, CSharpatron's 'shadowing' mode will - in most cases - totally bypass such headaches and allow you to test compilation of a file right away. For an explanation of this feature see 'How File Shadowing Works'.

Before file shadowing is possible a 'Stubs' file must be built.

7. Use the 'Build Stubs File' command to generate a 'header' file containing all of your classes.
8. Make all files 'live' in 'shadow' mode.
- As before, enabling a few files at a time is a sensible approach.
- You should see few errors. Any that crop up should be fixed in their .cs files.
- You may encounter files that have dependency issues even in 'shadowed' mode (e.g. files reliant on delegate types declared in another file - stubs for delegates *cheat* by just treating them as 'objects' but this doesn't cut it when client files compile against them!). In these cases there are two options:
  - Figure out which files are needed to provide accurate type information and activate those first.
  - Use dependency hacks to comment out problematic elements in your file. See 'Dependency Hacks Tag'
9. Review X-Refs for 'shadowed' files.
- You can use the File Dependencies mode to see *all* files that a given file is dependent upon - most likely a *lot*. Until all of these files are also 'live' a file needs to remain in the shadowed namespace.
10. Use the 'DeShadow' command to scan for files that are ready to be taken out of the shadow namespace and be made fully active.
- Any files that are ready will be automatically made active.

Once all files have been made 'live' and 'de-shadowed', you're almost done. Just two steps remain...

11. If you used Dependency Hacks to make it easier to activate files, now is the time to remove them.
- Look for cyan colored 'live' icons in your Workspace View - these are files with one or more hacks.
- After removing a hack and letting a file compile, its 'live' icon should turn green.
12. With all files compiling successfully you should now see a completion message in the Workspace View's status area. A new command: 'Finalize' should now be available. ...Click it!
- The Finalize button will appear when your C# conversion percentage is 100%.

And that should be it - your files are all converted and have been moved back to the original source file locations. Your project should now be executable in its C# form!

Files cannot be added to a 'Finalized' workspace. If you wish to do more conversion work you can use 'DestroyWorkspace' to delete the finalized workspace and initiate new conversions as for Step 1.

So did everything work?

See my own Case Study to read about how I fared with my own project. TL;DR: the conversion process highlighted the need for a few 'thing != null' checks in a couple of systems, but otherwise ...SUCCESS!

Configuration

Before getting too far into the conversion process, you may wish to review the configuration options available to you. The control center panel will initially look like this:
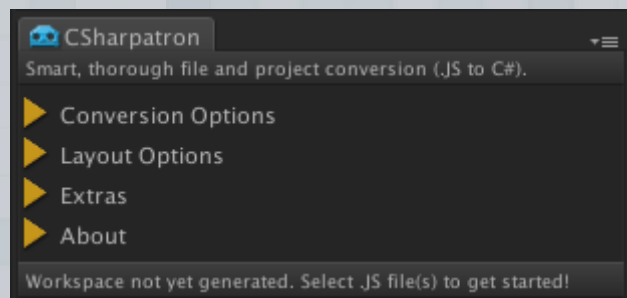
- Click on a yellow arrow to open up a list of related options.
- NOTE: the state of all options is automatically written to your EditorPrefs file.

Conversion Options

These are options relating to how CSharpatron will convert your source.

- **Custom defines.** Allows you to specify any custom defines that will control how CSharpatron parses your scripts. *Only script outside of any #if block or within a specified define can be converted.*

- **Platform defines.** Allows you to choose the core platform defines that you wish to be active for conversion. Because of the way CSharpatron looks up type information, you should only select platform options that make sense for the platform of your currently active project. See 'How Does CSharpatron Understand Variable Types?' for more information on this. *Only script outside of any #if block or within a specified define can be converted.*

- **Local var conversion**. You can choose how you want CSharpatron to reformat local (function) vars. Options are:
  - **Prefer inferred.** Declarations should use an exact type as inferred during conversion. If a type cannot be inferred CSharpatron may fallback to the 'var' keyword. *This is the default option.*
  - **Only Inferred.** As above, except that if a type cannot be inferred (rare), then an error is logged; the var keyword is never used.
  - **Use var keyword.** Where possible the 'var' keyword is used; type is inferred where the 'var' type would not be legal.
  NOTE: all options apply to local variable only since in C# the 'var' keyword is only valid in function scope; class vars must always be converted to an exact type.

- **Infer unqualified decimal literals as 'float'**. This controls what CSharpatron will do when it encounters a decimal constant without a float suffix, e.g. 1.0, 28.5, etc. Select this option and all such constants will be assumed to be floats and given an 'f' suffix, e.g. 1.0f, 28.5f, etc. Leave this unchecked and such constants will be converted as 'doubles' (i.e. no 'f' suffix).

- **Convert all-caps variable declaration to 'const'**. A common practice in professional development is for all constants defined in a program to be named using upper-case. Unityscript has no 'const' keyword, but by ticking this option CSharpatron will automatically declare any suitably named C# variables (that qualify) to be 'const'.

- **Filter libs list**. Each time you convert files with CSharpatron it must generate a large database of type information to supplement information already available via Reflection. This process can take a few seconds. With this option ticked the time is *reduced* by ignoring all symbols contained within a list of libraries that I don't believe many developers - at least those making games - are likely to be using. *By default the option is ticked but you may want to un-tick it if your project is not a game or you encounter conversion errors relating to 'unknown types'.*

- **Write to 'stubs' file during conversion. '**File shadowing' (a feature to help you work through dependency bottle-necks) relies on having type information available via an auto-generated 'stubs' file. With this option ticked CSharpatron will automatically write to the 'stubs' files each time a file is converted.
  By default this option is off. This is because the stubs file must itself have no external dependencies; if you are converting just a few files and/or expect to fix up dependencies by hand, the shadowing concept can't help you much, and you may just see errors from having only a partially complete 'stubs' file.
  If you are converting an entire project my recommendation would be to leave this un-ticked until the first conversion pass is complete. After that you may want to use the 'Rebuild Stubs File' command to build the 'stubs' file, and then tick this option to have each subsequent re-convert update 'stubs' file content as required.

Layout Options

These are options relating to how CSharpatron will format changes and additions to your source files.

- **Source formatting**. This option lets you select how CSharpatron will determine how to format any fix-ups and modifications to your source during conversion. Options are:
  - **Infer from file.** When this setting is active, CSharpatron will look at the formatting within each input file and will generate source modifications in a consistent style. *This is the default option and should typically 'do the right thing'.*
  - **User defined**. If you select this option then sub-options appear which allow you to directly choose how you want script changes to be formatted:
    - **Prefer spaced out source**. If ticked, you will see spaces between operators and vars, e.g.

```
        if ( thisVar < 5 ) otherVar *= 2;
```
If not ticked you would see:
```
        if (thisVar<5) otherVar*=2;
```
- **Prefer line-saver brace style.** If ticked you will see opening braces on the same line, e.g.
```
        void MyFunc(void) {

            ...

        }
```
As opposed to:
```
        void MyFunc(void)

        {

            ...

        }
```
- **Lines between functions.** Some conversion fixes can cause functions and type definitions to be moved within a file. This value expresses a preference for how many empty lines should exist between adjacent functions or types.
- **Tab size.** How many spaces does a tab character represent? *Default value is 4.*

- **Func-decl line-wrap.** When reformatting function declarations, if they exceed this (single line) length then parameters are split over multiple lines, e.g.
```
        void myFunc(int param1, float param2, string param3)
```
becomes...
```
        void myFunc(int param1,

                    float param2,

                    string param3)
```
Of course if you don't like this syntax you can just set a very large number for the wrap value.

- **Tabs as spaces.** If ticked then any inserted tabs will actually be implemented as a block of spaces (of the size given by 'Tab size'). *Default is off.*

- **Tab in mono class body.** In many cases CSharpatron will need to build a class declaration around any file-class methods and variables. This option specifies whether all content within the containing braces of that declaration should be tabbed in or not. *Default is on.*

Extras

- **Dependency hacks tag.** Even with file 'shadowing' you may still encounter situations where getting a file to compile is tricky without having other files already compiling under C# (e.g. when fixing up delegate types). A solution may be to temporarily comment out problem lines in your .cs file. CSharpatron can help you to track such 'hacks' provided you prefix them with a special tag, e.g.
```
    // !CSDEP! callDelegate( param1, param2 )

    or

    /* !CSDEP!

        callDelegate( param1, param2 )

        ...

    */
```
The default tag is the one shown here, but you can use this option to adjust it as you see fit. The presence of one or more instances of this tag will cause the 'live' icon for a file to show *cyan* instead of *green*.
*CAUTION: if you use this feature, I strongly suggest you add the tag for **every single commented out block**. Really. Fail to do this just once and its so easy to forget what was a dependency hack and what was just a commented out line in the original file!*

- **Shadowing namespace.** This allows you to set the name for the special namespace CSharpatron will use when 'shadowing' a file.

- **Show reflected types, Show members of type, Show members with name, Extension methods.** These are kind of a bonus feature in CSharpatron. …Using these fields you can query the .Net reflection database within Unity to learn lots of interesting things about the (*immense* number of) libraries and methods that are potentially available for you to use. You can also 'see' all elements of your own application as part of the same database. *See Appendix D for discussion on how to use these tools.*

- **Suppress log callbacks.** Unity has a simple mechanism that allows an Editor Extension to capture log file output. Unfortunately there is no arbitration of which clients might be using this capability: all you can do is set it, and if another extension was already using it, then tough - that extension will now be broken! CSharpatron makes extensive use of log capturing to collect C# compiler errors. If you're actively using CSharpatron then you really don't want to be suppressing the log callback and missing out on that error capture, however this option exists just in case you want to temporarily ensure that another Editor Extension can function correctly.

About
Version and contact information.

Workspace View

Sort alphabetically by file-name or path

File status line.
- *Left click to select*
- *Shift-click to multi-select*
- *Right click for file-commands*

Sort by error count (various)
- *Click to sort files*
- *Double click to invert ordering*

Sort by number of not-yet-live external references

Sort by file 'live' status

Workspace file-list

Selected file

File error status

Status line

Tiny spoon-bot working tirelessly behind the scenes

Project commands

File commands.
Use multi-select to act on a file-group

Quick help information / Status messages

CSharpatron v1.0 – Workspace

Filename	File-path		Total	Fatal	Error	Fix-JS	Fix-CS	Warn	C#	X-Refs	Live
Source/**StarSystem.js**							Errors: 22. Warnings: 3. X-Refs: 3/14			
Source/Objects/**StoreDepot.js**							Warnings: 5. X-Refs: 0/4			
Source/Objects/**StoreDepotFunnel.js**							Warnings: 2. X-Refs: 0/14			
Source/Rendering/**Sun.js**							Warnings: 5. X-Refs: 0/5			
Source/Support/**SystemSupport.js**							Warnings: 1. X-Refs: 1/5			
Source/Mission/**Task.js**							Errors: 2. Warnings: 19. X-Refs: 0/6			
Source/Mission/**TaskBase.js**							Warnings: 18. X-Refs: 1/3			
Source/MissionData/**TaskDef.js**							Errors: 1. X-Refs: 0/5			
Source/Mission/**TaskManager.js**							Errors: 2. Warnings: 5. X-Refs: 1/6			
Source/Mission/**TaskPreReq.js**							Warnings: 11. X-Refs: 0/6			

TOTAL .JS: 270 files / 107644 lines.   LIVE .CS: 0 files, (0 shadowed) / 0 lines = 0%   SELECTED: 1 file /164 lines.   Errors: 431. Warnings: 2272

Remove [1]   Re-Convert [1]   Toggle 'live' [1]

The Workspace dialog shows all files that CSharpatron has converted. For each file a .csTest version was created alongside the .js original.

You can use this dialog to review errors, plan fix-ups and, when ready, swap .js files for their C# replacements.

Fatal error
.JS (pre) fix-up req.
.CS (post) fix-up req.
Error
Warning
No errors

View Documentation.   Destroy Workspace.   Rebuild Error Log.   View Error Log.   Rebuild 'Stubs' File.   DeShadow File   ◀ 1/5 ▶   Close

Live status for a file

Not live

Live, no errors

Live, C# errors

'Shadowed'

Live, with hacks

Click to toggle file status

File status summary

- Conversion errors
- C# errors
- External type refs.

Left-click element to show list view

Resize window as required

Errors & X-Refs

E

Error line
Left-click to jump to source file

Error information

Click to show/hide error list

Error source

▭ Converter

▦ C# Compile

Source/FlowHandler.js

Errors: 2. Warnings: 7. X-Refs: 1/27

	1	W00: Class 'FlowState' will be relocated outside of Mono class scope to maintain global name scope in C#.
	156	CS00: Delegate type 'callback' must be fixed by user. MANUAL FIX-UP REQUIRED.
	164	INT: Field not a delegate – unrecognized symbol: 'callback'.

Error/warning status

398 W22: Access to static field must be updated for C#. Auto fix-up replaced 'instance.loadedGadget' with class-name 'GameObject'.

416 W06: Function 'GetGadgetCameraParent' declares no return type but returns non-void. Reflection derived type 'GameObject' will be used.

500 W18: Can't perform type-cast for JS 'Function' param 1

528 W22: Access to static field must be updated for C#. Auto fix-up replaced 'SaveGame.instance' with class-name 'SaveGame'.

Click to show file dependencies

Type Dependencies for selected file

Engine.Random'.

Source/ActionPoints/ActionPointType.js [shadowed]

Warnings: 39. Clean. X-Refs: 25/26

External Type References:

ActionPointBase	ActionPointBuriedTreasure	ActionPointRewardRock	ActionPointScannable	APSpawnMethod
BuriedTreasureState	Collectible	CollectibleType	Constants	DetonationRockState
EventManager	GameSupport	InventoryManager	json	LazyUpdateClient
PlayerAnim	PlayerControl	RewardResult	RewardRockState	RewardSource
SaveGame	SSAssert	SSPlayerPrefs	SSTime	SystemSupport
WallDetonationState				

Click to show/hide X-Refs

Type currently not available in C#

File Dependencies for selected file

Type available in C# but **'shadowed'**

Type fully active in C#

Click to show type dependencies

Source/Support/SSPlayerPrefs.js [shadowed]

Warnings: 37. Clean. X-Refs: 4/4

File Dependencies. 3 direct, 156 inherited:

Source/Objects/BuildingBase.js	Source/HUD/UIButtonSheet.js	Source/Support/Util.js
Source/Mission/BuildingProxyManager.js[1]	Source/Player/Cheats.js[1]	Source/Constants.js[1]
Source/Support/DebugPositionHelper.js[1]	Source/Rendering/DynamicProjector.js[1]	Source/Mission/EventManager.js[1]
Source/Support/FakeTouch.js[1]	Source/Moon.js[1]	Source/Support/ObjectStore.js[1]
Source/PoweredDevice.js[1]	Source/HUD/Scanner.js[1]	Source/UILayout/SharedButtons.js[1]
Source/Support/SpringObject.js[1]	Source/Rendering/SSColor.js[1]	Source/Support/SSDebug.js[1]
Source/Support/SSInput.js[1]	Source/Support/SSMath.js[1]	Source/Support/SSTime.js[1]
Source/StarSystem.js[1]	Source/Support/SystemSupport.js[1]	Source/HUD/TypeDefs.js[1]
Source/HUD/UIButton.js[1]	Source/HUD/UIButtonComponent.js[1]	Source/HUD/UIManager.js[1]

Direct dependency

Inherited dependency

Circular dependency

Interface

Please see the pages 'Workspace View' and 'Errors & X-Refs' for a visual overview of the Workspace View. Text in red denotes **interactive** components of the interface.

The *file list* shows all files in your workspace. You can use various options in the upper 'sort-bar' to control sorting of this list.

The *lightning-bolt* icon to the right of each line show the 'live' status for a file. You can click these icons to toggle the state (i.e. switch whether the file's .js or .cs file is currently active). Be aware that each time you click a 'live' icon you will be initiating a compile (which will in turn disable further file operations until complete to avoid potential error states).

You can select files in the file-list by *left clicking* them. Holding down *shift* will multi-select - you will see buttons in the lower panel reflect your selection count (each of these represents a possible group action that you can apply).

If you *left* click on the status fields of a file you can toggle display of extended information relating to that field, e.g. CSharpatron conversion errors, compile errors (if available) or external references. With either error list shown, you can click on a line number to jump to that error in your Unity associated text editor. If the X-Refs list is visible, you can click the text at the top left of this panel to toggle between X-Ref *types* and X-Ref *files*. The 'files' view will show direct, inherited (italic) and circular (yellow) dependencies.

By *right* clicking a file you can bring up a context menu showing file specific actions.

Context Menu

- **View .js.** Jump to the given file's .js version. This isn't available once a file is 'live' (deactivate it first).
- **View .csTest.** Jump to the C# converted form of this file. This isn't available once a file is 'live' (the 'View .cs' command takes its place).
- **View .cs.** Jump to the C# version of this file. This is only available one a file is 'live'.
- **Force build state refresh.** (For a 'live' file.) This forces a re-test of the compilation success/fail state for a file. It can be useful in rare cases where successful compilation fails to update a previous error state.
- **Re-Convert file from .JS.** Re-convert the file from the .js original. This will overwrite the current .csTest version of the file with a new one (be careful if you have made edits!). This option isn't available if a C# version of the file is active (deactivate it first).
- **Remove from Workspace.** This will remove the given file from the workspace, also deleting the .csTest version of the file. Only available if a file isn't 'live'. NOTE: this action cannot be undone (at least not without re-converting a file).

Workspace Commands

There are a number of commands generally available in the lower part of the Workspace View.

- **Remove[X].** Remove selected files from the workspace - synonymous with using 'Remove from Workspace' on each file.
- **Re-Convert[X].** Re-convert selected files to build new .csTest representations and capture new conversion log output for them. This is synonymous with selecting 'Re-Convert' on each file. NOTE: it pays to convert multiple files at a time since initialization work is only performed once.
- **Toggle 'live' [X].** This is synonymous with clicking the 'live' icons of each selected file. If a file is currently not live it will go live and vice versa. It often pays to group multiple files for activation toggling since build time is significantly reduced.
- **View Documentation.** This will open up this .pdf file in your system's associated .pdf viewer.
- **Destroy Workspace.** This will de-activate any active files and remove *every* file from your workspace. Be careful: any .csTest/.cs edits will be lost!
- **Rebuild Error Log.** Every time the converter runs, the log file is replaced with output reflecting solely the last conversion file-set. However, conversion log entries for *every* file are retained in Workspace.txt; this command will rebuild an error log with contributions from all files in your workspace.
  NOTE: when you rebuild the error log the order will reflect the current sort order of the Workspace View file-list.
- **View Error Log.** Open the current error log in your Unity associated browser.
- **Rebuild Stubs File.** A 'stubs' file is critical for usage of CSharpatron's file shadowing feature (see 'How File Shadowing Works'). This option will (re) build a 'stubs' file containing entries from every file in your workspace that isn't currently shadowed. You can run this command at any time to bring the stubs file up to date.
- **DeShadow Files.** This will look at the current dependency states of every shadowed file. Any files whose X-Refs are now fully fulfilled (i.e. which are now compiled to C# in either full or shadowed form) will be de-shadowed to become fully active C# files. You can see for yourself if a file is ready to be de-shadowed by using the X-Ref *file-view* mode. You will likely find that most files have a huge number of X-Refs so de-shadowing them won't be possible until very late in the conversion process.
- **Finalize.** This command appears once all workspace files are 'live' (i.e. compiled in C#) and you have no compile errors. The command effectively completes the conversion process by moving .cs files back from the 'Plugins' folder to your original source file locations. The stubs file will be deleted. NOTE: 'Finalize' will *not* delete either your workspace file or the original Unityscript backup files stored in 'CSharpatronWork/Backup'. It will be up to you to remove those files manually when you are ready to do so.

Under the Hood

Working Folders and Special Files

During CSharpatron usage, the converter will create several sub-folders within your project's Asset folder.

- **CSharpatronWork**. Two important files are written to this location:
  - **Log.txt**. This is the log output for the *last* conversion operation. NOTE: you can rebuild a full version of this file at any time using the **Rebuild Log File** command.
  - **Workspace.txt**. This file records all files that you have converted (and effectively added to your workspace). For each file it remembers conversion errors, compile errors and external references. *This file is critical to the ongoing conversion process so be sure not to delete it any time you're still working with a workspace!*
- **CSharpatronWork/Backup**. This is where your original **.js** files are backed-up any time that you 'go live' with a converted file. The files are renamed with the extension **.jsOrig** so that they are no longer 'seen' by the Unityscript compiler. *It is strongly recommended that you don't edit or delete these files during CSharpatron usage!*
- **Plugins/CSharpatronConverted.** This is where 'live' **.cs** files are written by CSharpatron. The location of this folder is critical: since it is within 'Plugins' this means that Unity will compile these files *before* attempting compilation of the .js files that still reside within your project's source folder (and so a new .cs file can go live and its types are still visible to your .js files during their own compilation, whereas if the .cs files were written to the same place as the .js originals, compilation order wouldn't permit the UnityScript compile to see those types and you'd get lots of 'unknown type' errors). For more information about script compilation order, see here.

Another important file is written to the top level of Plugins/CSharpatronConverted:

- **ConversionTemp.cs**. This is the file typically referred to as the 'stubs' file. It can be generated after file conversion by the **ReBuild Stubs File** command, and will be incrementally updated when the **Build stubs during conversion** option is ticked.
  The file contains stub versions of every class in your project that can be used to support 'file shadowing'. For more information on the role this file plays see here.
  NOTE: if you see compile errors in ConversionTemp.cs then this suggests that (probably) you have built a stubs file that doesn't know about all the types in your project (most likely there are files you haven't converted?). ...Or there's some bug in the generation of the stubs file (hopefully not!). In either case, provided you have no currently shadowed files, you can simply delete ConversionTemp.cs without consequence - remember it can be rebuilt at any time.

How Does CSharpatron Understand Variable Types?

CSharpatron utilizes a language feature called 'Reflection' that is built into .Net/Mono. This makes it possible to look up precise type information for any *loaded* (i.e. successfully compiled) assembly. It is for this reason that you need to always be working with valid, compiled script and why you should only attempt to convert scripts for platforms that match your active Unity project platform.

As an example: if you are working on a iPhone project, you may tick UNITY_EDITOR and UNITY_IPHONE. If you have any other options ticked and your script contains script within corresponding #if blocks then you will most likely see many errors.

Conversion Output

As a file is converted there are several types of information that may be logged.

- **Warnings**. These are typically informing you of changes that CSharpatron has made in the output file. You won't be specifically 'warned' for *every* little change that is made, but there are a number of specific types where you may want to be aware of and/or review changes.

- **Errors.** These are problems encountered by CSharpatron which mean that a portion of a file couldn't be converted (perhaps just a single statement, perhaps more if that first error had knock on effects). Regular errors don't mean that a converted file wasn't written out, but do tell you that the written file is unlikely to compile cleanly. There are two important sub-types of error that relate to manual fixes you will need to make in a file:
  - **Fix .JS.** These are items that will certainly block a file from compiling in C#. They are things that you should fix in the **.js** version of the file before then *re-converting* it. Upon the next conversion attempt you should see no recurrence of the original error and most likely other related errors will have gone away too.
    In practice there are only a few 'Fix .JS' error types that you are likely to see. You can read about them in Appendix B.
  - **Fix .CS**. These are things that need to be fixed in the output **.csTest/.cs** file. Because Unity will automatically start compiling these files if activated, you will certainly see C# compiler errors related to the same things; making fixes in the .csTest version of a file will often make most sense.
    In practice all errors in this category actually relate to use of Unityscript's 'Function' type and the need to explicitly configure how you wish to replace those types in C# (i.e. which kind of *delegate* and what its parameter and return types may be).
    You may in fact choose to fix delegate related errors by placing **Type Hints** in your .js files. You can see examples in Appendix A11.

- **Fatal errors**. These are issues that caused CSharpatron to completely abort conversion of a file (i.e. no output could be written). They can essentially occur for two reasons:
  a. You have some kind of formatting or compile error in your source file.

b. There is a bug or omission in CSharpatron. *<ahem>*

If you look in the log file you'll see extended error information and perhaps a stack-trace relating to a Fatal Error. Below this the log should tell you the last line successfully converted and so enable you to determine whether you're dealing with type a) or b).

In general CSharpatron is designed to be very robust in its handling of errors - even fatal errors will only disrupt a single file conversion if you are converting a larger group of files.

How a File Goes Live

When you click the lightning -bolt icon to make a file 'live' (or if is part of a multi-file **Toggle 'Live'** command), CSharpatron does the following:

- (If the file is not being 'shadowed') The .js version of your file is moved to a location that mirrors the original source file location but as a sub-folder of **'CSharpatronWork/Backup'**.
  - The file is renamed to become **<yourfile>.origJS**. With this new extension, Unity will no longer try to compile it.
- The .csTest version of the file is moved to a location that mirrors the original source file location but as a sub-folder of **'Plugins/CSharpatronConverted'**.
  - If the file is being shadowed, CSharpatron will enclose the entire file content within a custom namespace.
  - The file is renamed to become **<yourFile>.cs.** With this new extension, Unity will initiate compile of the file.
- (If the file is not being 'shadowed'.) The file **<yourFile>.js.meta** is copied to the folder **'Plugins/CSharpatronConverted'** and renamed **<yourFile>.cs.meta**. This is a critical step since it is how we're able to avoid any mapped object losing any script connections.

If you deactivate a file then these steps are reversed and you're left with the .js file active and the .csTest version sitting alongside it.

If you activate multiple files at once you'll find that the order in which the compiler builds them appears to be completely arbitrary, and while two or three may compile in parallel, all other files will be blocked until any compile issues in those chosen files are resolved. In practice I find that taking just a few files live at a time can avoid quite a bit of confusion (albeit this is a tradeoff against build-time).

Remember, if you do see relatively cryptic sounding C# errors it may be worth checking back over the CSharpatron log for the file: perhaps you'll see something you forgot to fix. ...Usually CSharpatron's errors will be more straightforward to understand, especially if C# is new to you.

How File Shadowing Works

To help you to work around dependency issues that can make activating your C# files a truly unpleasant ordeal, CSharpatron implements 'file shadowing'.

The principle is this: when you try to activate a C# file that has one or more externally referenced files *that isn't already compiled in C#*, CSharpatron will activate the file in 'shadowed' mode.

Behind the scenes what happens is that CSharpatron creates and manages a new C# namespace that contains 'shadowed' versions of your project classes. Initially this namespace will contain 'stub' versions of *every single type* declared in your (workspace) conversion files.

Before 'shadowing' can proceed, you first need CSharpatron to generate the .cs file that contains all of the necessary type information. This is the file **'Plugins/CSharpatronConverted/ConversionTemp.cs'**, also referred to as the 'stubs' file. To build this file you use the command **Build Stubs File**. Once you have a stubs file it will need to be kept up to date should you re-convert any source files. This will be handled automatically if you enable the Conversion Option **Build 'stubs' file during conversion**.
NOTE: a few liberties are taken when building the 'stubs' file, e.g. all fields are declared public, 'Function' and 'Array' type params (not valid in C#) become 'object'. ...We just need to make a file that as simply as possible captures the types and interfaces for your project so that other files can compile against it.

> NOTE: You can use the command **Build Stubs File** to rebuild the stubs file *at any time*, so this should be your first action should you encounter any build issue related to ConversionTemp.cs.

As files are activated in 'shadow' form, their .js version *remains* active (so your project remains compilable and runnable), and custom versions of their .cs counterparts go live too, where each such file has been automatically enclosed within the shadow namespace. For each 'shadowed' file, CSharpatron will remove the stub file entries for that file since the .cs file now provides 'real' definitions for all elements.

In essence the shadowing mode allows you to test the compilation state for all converted files and to locate and fix errors long before dependency issues would have otherwise allowed for useful compilation.

In testing, I attempted to convert my project using a version of CSharpatron where I hadn't yet implemented 'shadowing' mode. **Ugh!** What a world of hurt those file dependencies are... 'Shadow' mode FTW!

Conversions

Although there are many shared elements between Unityscript and C# - they both sit on top of the same .Net framework, both access most of the same libraries, share similar syntax - there are a many small syntactic differences and a few major conceptual differences that make manual file conversion a tedious and error prone chore.

CSharpatron attempts to automate as many aspects of file conversion as possible whilst remaining 100% consistent with original functionality.

I will outline the various conversions and fix-ups that take place and a few things that CSharpatron doesn't handle.

File structure adjustments

There are a few approaches to structuring classes within Unityscript files.

a) Don't declare any class, i.e. a file implicitly represents a MonoBehaviour derived class of the same name as the file. The file may contain other explicitly declared classes. ...All functions and vars defined in the file become part of the 'File Class'.
b) Explicitly declare a class within the file that has the same name as the file and may derive from MonoBehaviour, a different class or be a base class with no inheritance. The file may contain other explicitly declared classes.
c) Declare one or more classes where none of those classes share the same name as the file.

In C# there is no inferred automatic class generation (type A) and so CSharpatron must manually create an equivalent class. Obviously it needs to consider type B and type C files too - in those cases it just converts your own class declaration into C# syntax.

A challenge when working with type A files comes when your file includes other explicitly declared classes. Those classes may occur anywhere in the file, and under Unityscript compilation rules they are *not* subclasses of your main file-class as you might expect, but in fact possess the same (global) scope as the file class. CSharpatron must gather together everything that relates to the file class into a single contiguous block that can be framed within a suitable class declaration. Any other classes will be relocated - in front of the file-class as it turns out - in order to keep their global scope in C# form.

'Using' statements and type-names

CSharpatron will automatically convert any Unityscript 'import' directive to C#'s 'using' command. In addition the converter will always add:

```
using UnityEngine
using System
```

Other namespaces may also be added according to the types that your script makes use of, e.g. if you use generic containers such as List<> or Dictionary<>, you will see 'using System.Collections.Generic' added.

Whenever a namespace is included, converted types will use correspondingly shortened names. Where shortened names present ambiguity, e.g. the type 'Random' could mean 'System.Random' or 'UnityEngine.Random', then the fully qualified name is used.

Pragmas

Pragmas 'strict', 'downcast' and 'explicit' are all removed by the converter since these aren't valid in C# .

Attributes

All attributes are converted from Unityscript to C# syntax, e.g.

```
@ExecuteInEditMode                      [ExecuteInEditMode]
@System.NonSerialized                   [System.NonSerialized]
@script RequireComponent (Camera)       [RequireComponent (typeof(Camera))]
```

As a special case, RequireComponent is also relocated to just prior to the main (Mono derived) class definition.

Simple syntactic changes

There are a few areas in which syntax differs between Unityscript and C#. The following are all automatically converted by CSharpatron:

- Variable declarations uses a different syntax, require inversion of type/name, e.g.

```
var myVar: int;                         int myVar
```

- Function declaration uses a different syntax, requires inversion of type/name for parameters, e.g.

```
function myFunc(myVar: int): int        int myFunc(int myVar)
{}                                      {}
```

- Generic types have a slightly different syntax:

```
var myList: List.<int>;          List<int> myList;
```

- Unityscript's 'boolean' type becomes 'bool' for C#.
- Unityscript's 'String' type becomes 'string' for C#.
- Unityscript's 'Number' type because a 'double' for C#.
- Unityscript's 'base' keyword (to represent a parent class) is replaced by 'super' in C#. CSharpatron also has to relocate where callbacks to base class constructors occur - they must come between a constructor's prototype and its method body. See Appendix A15.

Serializable attribute

In Unityscript, any class that doesn't derive from MonoBehaviour is internally tagged with the 'Serializable' attribute. CSharpatron will explicitly add this attribute for all types - unless they possess a 'NonSerialized' attribute - since failing to to so would cause all mapped objects to lose any data that used these types.

Field and type visibility

In Unityscript variables and methods are public by default and may be made private with the 'private' keyword. In C# the default becomes private and the 'public' keyword must be used. The same holds true for classes and enums.
CSharpatron will automatically add or remove the appropriate keyword to achieve equivalent visibility in all cases.

Default assignment to variables

When declaring variables in Unityscript, it is common practice to rely on them having been assigned a default value by the compiler. In C# there is no default assignment (see here). Consequently *CSharpatron will automatically always assign a default, type-appropriate value to each non-assigned variable declaration*.

Reserved variable names

Any language prevents you from giving names to your variables that conflict with language keywords (or at least try to do so you'll likely see cryptic errors!). While C# shares many keywords with Unityscript, there are some differences and *many* additional keywords in C#. You may well find that you have variables with names such as 'string', 'object' or 'params' that won't be permitted in C#. You can see a list of C# keywords here.

CSharpatron will automatically fix any *local* variable names that would be illegal in C#. For *class* vars you will be prompted to make manual fixes for any transgressions. This is necessary because the knock-on effects of class scoped changes can easily spread to multiple files, plus you probably don't want a class var to receive some weird, automatically adjusted name!

Variable types

In Unityscript a variable can be declared without type and will be implicitly typed upon first assignment (and unless a file contains '#pragma strict', variable types can be reassigned at will). It is also common practice to omit specific declaration of a type and let the compiler determine the type according to what is assigned.

In C#, variables are always strictly typed, usually via explicit type declaration, or, for local (function) variables declared using the 'var' keyword, their type will be determined from their assignment (assignment must occur in the same line as declaration).

In order to perform the most thorough conversion, CSharpatron has to determine types in a very strict manner. By default it will automatically convert your untyped/assignment-typed Unityscript vars into explicitly typed C# variable declarations. (If you prefer to have locals use the 'var' keyword then you can change this behavior via the 'Conversion/Local Var Conversion' configurable option).

You can see examples of CSharpatron's type inference in Appendix A1- A7.

Please note that CSharpatron does not support type re-assignment; if #pragma strict isn't present in a file they you will always see a warning...

Implicit 'new' fix-ups

Unityscript doesn't care if you use 'new' when allocating a new type: if you use it, fine, if you don't then it'll be implicitly added whenever a type constructor is 'called'. In C# use of 'new' is mandatory and CSharpatron will add the keyword in all contexts that require it. See examples in Appendix A2, A3, A5.

Array types

Unityscript and C# use different syntax for assignment to array types. In C# this syntax includes the requirement to 'new' an assigned array of constants.

- There are two optional approaches for representing multi-dimensional arrays, e.g. int [,] ('multi-dimensional') and int[][] ('jagged').
- Of these, CSharpatron will always convert a multi-dimensional Unityscript array into a 'jagged' array type (effectively an array of arrays, e.g. int[][]).
- CSharpatron will fully convert initialized Unityscript arrays into the equivalent C# syntax (using the 'jagged' array form if dimensions are greater than one).
- Array's declared in C# assembles may also be a mix of multi-dimensional and jagged form. From v1.1 CSharpatron should handle working with such arrays.

Array length is only accessible via array.**Length** in C# rather than Unityscript's optional array.length form. See array examples in Appendix A5.

Type casts

Unityscript seldom requires you to use type-casts since type handling is so geared around type *inference*; type promotions are handled silently. Where casts are used, the only form available is the postfix style 'as' operator.

The exact opposite is true in C# - with expressions involving multiple types, you must be *totally* explicit about how type promotions occur, very little is ever just inferred. C# supports prefixed bracket-style casts and postfixed 'as' casts. A primary difference between the casts is that 'as' casts may fail (yielding a null result that you can test for) and may consequently be used only on what C# calls 'nullable' types.

As CSharpatron converts your script it is fully aware of all element types, and where required it will automatically insert type-casts. In some cases this may seem like a *lot,* but only in very rare circumstances will you find them to be unnecessary! NOTE: CSharpatron does respect implicit cast operators that may be declared for specific assignment of a type and shouldn't apply casts where these are valid.

When you're doing math involving multiple vector types things can look especially ugly, not least since you're used to those automatic type promotions. However, without suitable casts the C# compiler *will* give you errors complaining about operator ambiguity.

Where an element to be cast is a literal, CSharpatron will, in preference to inserting a cast, reformat the literal to achieve the same result.

See type-cast examples in Appendix A12 - A14.

Numeric literals

There are differences in how decimal literals are formatted in Unityscript and C#

```
var myFloat = 1.0; // or...      float myFloat = 1.0f;
var myFloat = 1.0f;              float myFloat = 1.0f;
var myDouble = 1.0d;             double myDouble = 1.0;
```

CSharpatron will automatically reformat literals as appropriate, taking into account the preference you can set using the Conversion Option **Infer unqualified decimal literals as float**. *Please note that this option defaults to 'on'.*

If you use scientific notation then similar suffix conversions will occur. CSharpatron will also consider the range of a scientific notation value and if too large for a float will promote to a double. See more examples in Appendix A2.

Character literals

Unityscript doesn't support character literals. As a strange workaround when calling functions requiring character parameters you can use form "X"[0], effectively accessing the first/only element of a string to obtain the required 'char'.

- When CSharpatron encounters these element 0 string accesses it will convert them to an actual C# literal.
- In the case of calling certain string functions (e.g. 'Split'), you may see the docs list only prototypes that expect a 'char[]' and yet your function calls passes only a single 'char'. In these cases you're actually utilizing a .Net feature where the final array typed parameter of a function is declared a 'params' field and may receive one or more comma de-limited values of the given parameter type.

Access to static class members

Unityscript allows you to access static variables and methods by prefixing them with the class name, or by accessing them via a type instance as though they were non-static. In C# only the former approach is permitted. C# will automatically made adjustments as required. See example in Appendix A16.

Implicit boolean comparisons

In Unityscript you can simply place a variable name into any 'test' context (e.g. an 'if' statement or ternary operator) and a binary test of the variable is inferred. If the variable is an integer, the test is implicitly 'if intVar != 0', or for a complex type it is 'if typeVar != null'. Prefixing the variable name with a '!' will negate the inferred test, e.g. making the 'int' case 'if intVar == 0'.
In C# only boolean types can be used in this way. For all other types, CSharpatron will infer the correct implied test and adjust source output accordingly. See examples in Appendix A8.

Type comparison

In Unityscript a type-name can be used in a comparison operation against a Type. In C# the typeof() operator must be used.

```
if (v.GetType == MyClass)        if (v.GetType() == typeof(MyClass)
```

Another way to compare type is using Unityscript's 'instanceof' operator. This is replaced by the 'is' operator in C#:

```
if (v instanceof MyClass)        if (v is MyClass)
```

In Unityscript typeof() can be used to test the type of a variable instance. In C# this isn't possible. From v1.1 CSharpatron makes the following fix-up.

```
if (typeof(myVar)==String)       if(myVar.GetType()==typeof(string))
```

Function declaration

In addition to inversion of var-name and type, there are some other language differences when declaring a function:

- C# requires that functions returning void explicitly declare this.
- C# requires that if a function is to act as virtual base then it must be declared 'virtual'.
- C# requires that if a function is an override of a virtual base, then it has to be declared 'override'.

CSharpatron makes these adjustments as required. You can see examples of these conversions in Appendix A15.

Calling functions with 'ref' and 'out' parameters

Unityscript has no support for explicitly declared reference parameters but it *will* happily call library methods whose fields may be declared with *reference* or *output* qualifiers. In C# it is required that when making such function calls, the qualifiers 'ref' or 'out' are prefixed to the relevant parameters of the client's function call. CSharpatron will automatically detect when these qualifiers are necessary and add them accordingly. See examples in Appendix A26.

'Function' type and C# delegates

Unityscript offers the type 'Function' that makes it possible to assign a function to a variable in order implement object callbacks. You can literally pass any function to a 'Function' type without there being any need to declare input or output parameter types. You can even assign functions of varying type to the same 'Function' var, even with '#pragma strict' defined (i.e. Function vars are dynamically typed).

C# offers a number of ways to achieve function indirection, with 'delegates' being the most obvious replacement for your 'Function' instances. ...Of course you won't be surprised to hear that a delegate has to be declared in a *totally* explicit manner - you tell it the exact types of inputs and - if you want to have one - the output type too. There are a few forms of delegate class with the simplest being the 'Action<X>' and 'Func<X, Y>' generic types where you would use 'Action' in cases with no return type and 'Func' where there is (the final generic field is the return type).

A limitation of Mono 2.0 (as currently used in Unity) is that it implements an earlier version of the CLR which only supports up to five generic type parameters for Action and Func. Where you need more than five types you can use a different class to create a custom 'Delegate<>' type that has the exact prototype that you need.

Conversion of Function vars to delegates is frankly the weakest aspect of script conversion using CSharpatron. Available reflection data only reports the underlying type of a Function (this being 'Boo.Lang.ICallable' which cannot be used in C#), and without trawling through *all* files trying to determine the function types you are assigning to your Function vars, there's simply no way to

automatically infer the type fields required in your substitute delegate. (There is one exception: if you assign a function to a local Function var CSharpatron can infer the exact prototype in this case).

Consequently in most cases where Functions var are declared or referenced you will see a conversion error and a prompt to manually fix the issue. You can do that either in the output C# file, or - *since making fixes at the .JS stage and reconverting is neater* - CSharpatron allows you to supply 'type-hints' that enable it to fix at least all var *declarations* at conversion time.

See Appendix A11.

Anonymous methods

Unityscript allows you to declare anonymous methods that are assigned to a variable, e.g.

```
var testFunc = function() { Debug.Log( "Hello" ); };
var testFunc2 = function( a: String, b: String ) { Debug.Log( a + b ); };
var testFuncRet = function( a: String, b: String ): String { return a + b; };
```

From CSharpatron v1.1, these would be converted to:

```
public Action testFunc = delegate() { Debug.Log( "Hello" ); };
public Action<string, string> testFunc2 = delegate( string a, string b )
{ Debug.Log( a + b ); };
public Func<string, string, string> testFuncRet = delegate( string a, string b )
{ return a + b; };
```

Please note that although Unityscript allows you to declare and assign to an anonymous method var as separate operations, C# requires that declaration and assignment occur within the *same* line; you may see errors relating to this.

Extension methods

Although these cannot be declared in Unityscript they can be utilized in .js files. From v1.1 CSharpatron should parse them correctly.

Unityscript doesn't seem to mind if you utilize an extension method without using parentheses (i.e. as if it were a property). This isn't valid for C# and an attempt to do so with yield a 'Fix .JS' conversion error (JS18).

Properties

From v1.2 CSharpatron will correctly convert Unityscript properties (class methods using the 'get'/'set' syntax) to C# properties with corresponding 'get'/'set' methods.

'Finalize' functions

Any such functions are automatically converted into C# class destructors to avoid C# compile error: 'CS0249: Do not override `object.Finalize()`. Use destructor syntax instead'. See discussion here.

Multi variable assignment

Unityscript and C# both support assignment chains where multiple assignees received a single right-hand-side value. C#'s stricter typing means that assignments that were valid in Unityscript may not be so in C#. If you consider a statement such as 'vec3Array = transformArray = null' you can see why: assigning a var of type 'Transform[]' to an array of type 'Vector3[]' clearly makes no sense! CSharpatron will split apart such an assignment ensuring that each type receives a valid assignment, casting if necessary. See examples in Appendix A14.

Modification of 'foreach' loop iteration vars

The Unityscript loop format 'for X in Y' is converted into a C# 'foreach' loop. There are some differences however:

- A 'foreach' loop requires that the type of the iteration var be explicitly declared. This is handled by CSharpatron.
- You may not write to the iteration var in a 'foreach' loop. CSharpatron will detect any attempt to do so and log a 'Fix .JS' style error (JS05).

Assignment to value type return values

In C# it isn't legal to write to components of a 'value type' if that type is itself a return from another value type. For example if you write a line such as 'myTransform.position.x = 0.0' then this will yield a compile error in C#. (The logic here being that reading from a value type (e.g. the Transform class) will return a *copy* of the accessed field. Writing to a copy would obviously be pointless!)
Fixing these errors is slightly involved. Two approaches are:

```
Vector3 temp = myTransform.position;
temp.x = 0.0f;
myTransform.position = temp;
// or
myTransform.position = new Vector3( 0.0f,
                                    myTransform.position.y,
                                    myTransform.position.z );
```

CSharpatron will automatically fix up the assignment using the *first* approach (at least this is true from from v1.1 - it actually used the second approach in the previous release). The approach should work in all cases barring multi-var assignment. Where consecutive lines access fields of the same host, CSharpatron

should combine these smartly to minimize use of temp vars. If a fix-up cannot be made then an error is logged. You can see more examples in Appendix A24.

Local variable scoping differences

Unityscript's local variable scoping rules *appear* to be (since I know of no official documentation) very simple, and decidedly 'loose'. ...You can declare a variable anywhere within a function and it remains accessible at any later point within the function, irrespective of the relative nesting levels of declaration and access. Because the var remains in scope for the remainder of the function, it is common-place (in my script at least!) to find yourself reusing certain common variables - e.g. loop iteration vars - multiple times. In these cases the first 'for-loop' declares and initializes 'i', each subsequent loop just reassigns and iterates using the same 'i'.

C# employs far stricter scoping rules: a variable is accessible only within the scope it is declared (although its name is *reserved* for the remainder of the function).

Because of these differences, naïve conversion of function script would yield many out-of-scope access issues for C#. To address this, CSharpatron will move *where* variables are declared (as required) to place them in a scope appropriate to declaration and *all* accesses.

Obviously the converter will only make these changes very carefully and in rare cases may end up logging an error if a safe redeclaration point can't be determined.

You can see examples of scope fix-ups in Appendix A21 - A23.

Type inference when not using #pragma strict

Where Unityscript files do not specify #pragma strict, new local variables may be introduced to your scripts just by introducing a new name on the left-hand side of an assignment, either within general function scope, or in the initial field of a 'for' loop. C# still requires a concrete type, so CSharpatron will infer type from the assignment and insert explicit declaration for each such variable.
*NOTE: CSharpatron does **not** support re-typing of vars -the type is set once, upon first assignment. This is mainly because C# doesn't support this, so deducing the right fix-up in all cases would be no small feat!*

Coroutines & Coroutine usage

Unityscript hides virtually all complexity when handling coroutines. For C# more specific formatting is required.

- C# requires all coroutines to return the type 'IEnumerator'.
- C# requires the yield command to return an object that implements the 'IEnumerator' interface
  - ...Or null.
  - The standard classes: WaitForSeconds, WWW, WaitForEndOfFrame all implement 'IEnumerator'.

Unityscript 'in' operator

Although I've yet to find any documentation on this, Unityscript *does* support Javascript's 'in' operator, used to determine if an array or container holds a given value, e.g. 'if value in container'. Since C# offers no equivalent operator CSharpatron will just log an error should an instance of this be identified.

Switch statements

There are a number of restrictions upon switch statement formatting in C# compared with Unityscript.

- In C# you cannot have a (non empty) 'case' block that just drops through into the 'case' below: if this is desired behavior you must specifically engineer it. This can be done using C#'s 'goto' keyword.
- In C# you *must* have a 'break' in the final 'case'/'default' clause of the switch.
- In C# it's mandatory that the switch test var and all case conditions are of the same type.
- In C# it's illegal to have a   Type as a 'case' condition: the 'switch' statement must be reworked. CSharpatron will log error JS14. A simple fix is to change the switch test to reference a type *name*, e.g. 'myVar.GetType().**Name**'. You can then just put quotes around the types in your 'case' conditions.
- In C# all case conditions must be *constants*.

CSharpatron can fix the first three of these but not the last two. You can see examples in Appendix A18 - A19.

'while (1)' becomes 'while (true)'

CSharpatron will convert this common 'loop always' style to avoid type error/casting in C#.

parseInt, parseFloat

These methods aren't available (or necessary) in C#. CSharpatron will automatically replace them suitable type-casts.

Add component / component look-up methods

When adding a component or looking up component using any of the various methods to do so, there are three different approaches that you may use:

```
var comp: MyComp = go.GetComponent( "MyComp" );    // By name
var comp: MyComp = go.GetComponent( MyComp );      // By type
var comp: MyComp = go.GetComponent.<MyComp>();     // Generic method, preferred
```

CSharpatron will automatically convert any usage of any of these these functions to use the preferred, better performing, generic form. See examples in Appendix A10.

UnityEngine.Object.Instantiate

Seemingly against all language principles, use of this function in Unityscript doesn't just return a UnityEngine.Object and demand a cast to the actually instanced type, it can actually return an instance whose type matches the 'clone object' type of the first parameter. For example (in a non #pragma strict file at least), the following **is** legal Unityscript:

```
var obj = Instantiate(objPrefab, pos, rotation); // Returns Object type surely?
obj.transform.parent = otherObj.transform;       // Nope, and don't call me Shirley.
```

A literal C# conversion expects an Object type return and errors accordingly on the following line. To work around this apparent Unityscript magic, CSharpatron has special treatment for Instantiate() inferring a return type to match that of the clone object.

@CustomEditor

An implication of this attribute is another piece of Unityscript voodoo: it makes it so that the 'target' field of the Editor class can be accessed as if it were of the class type declared in the attribute's parameter!

CSharpatron wasn't really written with conversion of editor scripts in mind and v1.0 didn't support this attribute. However from CSharpatron v1.1 '@CustomEditor' **is** properly supported in a manner that makes type informed access to the 'target' field possible . To achieve this, whenever an editor function references 'target' the access is changed to be 'target_cs'. This new var is automatically inserted at the top of the function and assigned a suitably type-cast version of 'target'. Conversion of editor scripts now becomes significantly more likely to succeed!

Limitations

Reflection Limits

As previously stated, CSharpatron makes extensive use of Mono's type reflection capabilities in order to understand underlying types as it parses your source. This gives it capabilities that go well beyond any other source-level converters (that I'm aware of), but the approach does have a few inherent limitations.

- Only script visible to the compiler can be parsed. This means that any code within comments or non-active #if conditions *cannot* be converted.
- All source must be in a valid, fully compiling state; syntax errors in your script may trip up CSharpatron.
- As you make fixes to source files you need to wait for the compiler to build those files before they can be successfully re-converted (in fact the Workspace window blocks on this). Effectively you are waiting for the modified assembly to be re-loaded after compilation completes.

NOTE: please see documentation on switching platform/build-defines under Usage.

Armed with the ability to look up information on any already compiled class, CSharpatron can tackle most things you can throw at it. However where it's necessary to convert from a Unityscript only feature to a different approach in C# (e.g. going from Unityscript's 'Function' type to C# delegates), reflection doesn't help us. The converter does as much as possible to help, but in some cases is limited by the fact that it only knows about one file at a time so, e.g. if you use type-hinting to help convert 'Function' fields declared in one file, a different file that might include calls to those Function types still can't understand their type. (At least not until the declaring files become 'live' in C# form and reflection of the converted types becomes possible).

Auto Fix-up Limits

The converter goes a long way in terms of trying to fix up problems it encounters, but there are situations where where the 'right' fix is entirely subjective, and so you'll just see an error logged. Some examples:

- Illegally named function vars are automatically fixed because it is practical for CSharpatron to safely fix up all references. For illegally named *class* vars there is no auto-fix since access to those fields may extend beyond the declaring file.
- Where a function declaration declares a return type but none is supplied (something Unityscript *doesn't* warn about), what *should* be returned is something that demands careful programmer consideration!
- No initialized hash-table support. This syntax doesn't exist in C#.
- Non constant case statement fields: you'll probably want to implement an if/else ladder, but maybe not?
- In Unityscript you can assign a public class field to a private class field. In C# this is illegal (unless the private field is static). I don't understand the underlying cause of this limitation (?) but clearly it wouldn't be reasonable for CSharpatron to 'blindly' make a variable static in order to fix the assignment; this is a fix-up that demands programmer judgment!

#pragma strict

CSharpatron's parsing was primarily written to assume that your input .js files contain #pragma strict (since such script is already closer in nature to C# than script without the #pragma). That said, there's nothing stopping you from running the convert on non #pragma strict files, and in many cases it can still generate clean, fully typed C# script.

The big limitation is that if your non #pragma strict file re-assigns variables with types that differ from the first assignment, CSharpatron won't know to adjust it's internal representation of the var's type (in fact it will most likely try to cast the assignment to match the original type - this may well make good sense in all but the most perverse cases!).

When using CSharpatron on a non #pragma strict file, you will always see a warning about this type inference limitation and you should probably always check through all warnings to ensure that conversions *do* make sense.

UnityScript 'Array' Class

Unityscript's untyped Array class has no direct equivalent in C#, although there are several ways to perform the same task. In truth, use of this class has long been discouraged due to its inferior performance and lack of type safety. Use of the generic list (List.<T>) is often recommended in its place.

Any usage of the Array type will be caught as errors in CSharpatron's log file. The best approach for conversion is probably to switch your scripts to use a generic list *before* attempting conversion to C#. Discussion of this topic can be found here.

Script File Location

Unity compiles scripts in a multi-pass manner with a strict order (see docs here). Files may only reference types declared in their own pass or an *earlier* one. For this reason, in order for C# files to be made 'live' one at a time (when there are likely still Unityscript files dependent upon them) CSharpatron has to move them to a compilation pass earlier than the original Unityscript version. At present the converter is written to only support conversion of files in the general project space that are moved into a Plugins subfolder when made active. *If you need to convert editor scripts within the Editor folder or Standard Assets then you would currently need to work around this limitation.*

You will find that only .js scripts not sitting below the Editor, Plugins or Standard Assets folders will be considered for conversion.

Trouble-shooting & Tips

If you are encountering conversion errors that don't make sense, be sure to consider the following:

- **It is critical that you keep your project in a cleanly compiling state throughout the conversion process!** Basically whenever there are build errors this means that CSharpatron is unable to access the corresponding assembly and all of the Reflected type information it would contain. This will lead to conversion errors (unknown types) and may cause UI errors/confusion (e.g. seemingly incorrect reporting of XRef statuses).
  - **Definitely don't attempt an initial conversion if you have errors in your .js files.**
  - If you see errors in C# files as they are made active you can deactivate files to 'clean' the error state as quickly as possible, and then focus attention on fixing issues in those files.
  - Especially if you have a complex project, it really pays to take it slow with C# file activation, perhaps 1-4 at a time. (If you make too many active at once and an error is encountered in the first file all other files will remain in a confusing 'pending' state because Unity won't even attempt to compile them until the initial file's errors are resolved).

- Don't forget that the build defines you set for CSharpatron must match those that you have selected for your current editor build: the converter relies on having built assemblies in which to look up all class type information. You will need to match both the platform define and any custom defines you may have configured in your project's build options.

- If you're finding that CSharpatron is unable to recognize certain types that reside with APIs beyond your own game-code, then this may be because those APIs are explicitly excluded by the converter. At run-time CSharpatron builds a database of type information from the vast pool of types accessible using reflection. To reduce this initialization time we specifically exclude a range of Mono assemblies that (very subjectively!) don't seem relevant to most Unity game developers. You can use the option Conversion Option **Filter libs list** to stop this from happening.

- Occasionally you may find that it makes more sense to explicitly declare a variable type in your .js file and re-convert rather than letting CSharpatron infer type. An example would be when you need a var to be of base type so that various derived type objects can be assigned to it. CSharpatron's approach of looking at the first assignment can't help in this case!

- You are blocked from converting files in **Editor**, **Plugins** or **Standard Assets** folders because these special folders have fixed positions in Unity's compilation order making it impossible for CSharpatron to relocate files as it needs to for successful compilation (see Script File Location). A workaround for this is to create a new, temporary project containing the reserved folder(s) you wish to reconvert. Rename the folders (e.g. with some temporary suffix). You can now convert the files and move the resultant C# versions to their original locations within your original project (and delete the .js originals).

- You will most likely find that both CSharpatron and the C# compiler will reveal at least a few errors in your underlying .js files. Especially if a C# error doesn't appear to make sense, be sure to check back to the original .js source - perhaps a long-standing bug will suddenly be explained!

- If you start to see compile errors popping up in your .js files when .cs are made live then this is most likely due to 'import' directives in those now removed .js files no longer being present during Unityscript compilation. ...Really, every script file that uses classes from a system library (e.g. a container class from System.Collections.Generic) should include the necessary namespace. However since files are compiled in a block that encompasses all inter-dependent files, you can often get away without needing the namespace import in every single file. To fix your new .js errors you just need to import the necessary namespace into the affected file or update code to use 'full' names for all access to relevant types.

- If you're getting compile errors in ConversionTemp.cs then most likely this is because your workspace doesn't include the original scripts for all types that are being referenced. In these cases you should ensure that you don't have **Build stubs file during conversion** ticked and you can safely delete ConversionTemp.cs. ...When needed you can just use the **Rebuild Stubs File** button to re-generate the file.
  If you still see ConversionTemp.cs errors this suggest a support issue, however be aware that this file can be deleted at will in order to return to a cleanly compiling state (at the expense of file-shadowing not being possible).

Array Conversions
When converting from an Unityscript Array to a C# compatible type there are a few obvious obvious choices you might want to replace it with:

- Where array elements are all of the same type and the array is populated once, either at point of initialization or by incrementally adding members up to a known maximum, a **built-in array** is your best choice.
- Where array elements are all of the same type and the array is used dynamically, either when an unknown number of elements are to be added or content will be changing dynamically, a **generic list (List<T>)** is your best option. See documentation here.
- Where array elements may each be of differing types you can use an **ArrayList**. See documentation here. Note that there are benefits to using a list of fixed type (better error checking by compiler, no need for type-casts upon access) so only use this type when you really need type variance and a generic list isn't an option.

On occasion you may be better served with a more specific container, e.g. a Hashtable, Dictionary, Stack, or Queue.

Since all of these optional types are usable in Unityscript, the best approach is to make array conversions in your .js files *prior* to conversion (or else use the converter to locate usage, make your changes and then *re-convert* those files).

If you are using any container types other than a built-in array, then you will need to add an import directive at the top of your file:

```
import System.Collections.Generic;    // For List<T>, Dictionary<T>, Stack<T>, etc...
import System.Collections;            // For ArrayList...
```

Some other useful syntax to know:

```
// Built in arrays...
var myArray: int[];                   // Declare an uninitialized array variable
var myArray = new int[ 10 ];          // Declare an empty array with room for 10 elements
var myArray = [ 1, 2, 3, 4 ];         // Create an array filled with the given value
var arraySize = myArray.length;       // Get the length of an array

// Generic lists...
var myList = new List.<int>();        // Create a new, empty list ready to hold 'ints'
myList.Add( 1 );                      // Add elements...
myList.Add( 2 );                      // NOTE: List<T> can't be initialized in .js :(
myList.RemoveAt( 0 );                 // Remove the first element in the list
myList.RemoveAt( myList.count - 1 );  // Remove the last element in the list

myList.AddRange( myArray );           // Add contents of built-in array to a list
var myArray = myList.ToArray();       // Initialize a built-in array with list elements
```

There is an excellent post on the various list and container types available to you here.

Function to Delegate conversions
Conversion of Unityscript Function variables to C# delegates can be fiddly because CSharpatron has no way to infer how you use a given Function var and specifically what the prototypes of functions you assign to that var will be.

Your best tool to convert over a Function var are type-hints. These are comments that you add to a .js file prior to conversion. You need to add a type-hint just prior to each Function reference, e.g. on the line before declaration of a Function var, or preceding a function that accepts Function type parameters.

Here are some examples:

```
// !CSTYPE! onEnterCallback: Action<TriggerObject,Collider>; onExitCallback:
Action<TriggerObject,Collider>;
function AcquireProximityTrigger( layerMask: LayerMask,
                                  triggerRadius: float,
                                  onEnterCallback: Function,
                                  onExitCallback: Function,
                                  maintainInTriggerLists: boolean,
                                  tagList: String[] ) { … }


// !CSTYPE! onBlocked: Action; onBreak: Action;
function WaitForNavigation( targetInfo: AITaskNavTo.TargetInfo, onBlocked: Function,
onBreak: Function ) { … }


// !CSTYPE! onPlacementCallback:Func<GrabbedObject,bool>;
var onPlacementCallback: Function;
```

And here's the converted output:

```
public void AcquireProximityTrigger( LayerMask layerMask,
                                     float triggerRadius,
                                     Action<TriggerObject, Collider> onEnterCallback,
                                     Action<TriggerObject, Collider> onExitCallback,
                                     bool maintainInTriggerLists,
                                     string[] tagList ) { … }

public IEnumerator WaitForNavigation( AITaskNavTo.TargetInfo targetInfo,
                                      Func<bool> onBlocked,
                                      Func<bool> onBreak ) { … }

public Func<GrabbedObject, bool> onPlacementCallback;
```

Even with type hints, you may still encounter conversion errors at the point that Function vars are *invoked* if this occurs in a separate file to the var's declaration (a symptom of CSharpatron converting a single file at a time). In such cases you can usually ignore these errors and proceed with activation of the converted file, however you will still need the declaring file to be already compiling in C# for your file to build without error.

A more complicated case is where you are assigning a function to a Function var that receives more than 4 parameters (see why here). The best approach here is to use type hints as usual, e.g.

```
// !CSTYPE! captureFunc:
Action<Camera,Plane[],PropLocator[],PropMeshInfo,PropLocator.Flags>;

var captureFunc: Function;
```

Which should get you through conversion... However the C# compiler will complain about the delegate having too many generic fields. To fix this, you could edit your .cs/.csTest file to use a custom delegate type, e.g.

```
public delegate void PropCaptureDelegate( Camera camera, Plane[] frustumPlanes,
PropLocator[] candidates, PropMeshInfo pmi, PropLocator.Flags captureFlag );
public PropCaptureDelegate captureFunc;
```

Limits of File-shadowing

File-shadowing (see here) is a powerful tool to circumvent dependency issues and allow to to get C# files into a compilable state but it does have some limitations.

At any moment during conversion, the 'stubs' file (CSharpatronTemp.cs) must supply type information for every type *not* currently compilable in C# form. When building this file, as usual, Reflection is used to look up the necessary information. However, some types have no known (or valid) representation in C#, e.g. our usual friends: Unityscript Arrays and Function vars and also Boo functions. Any such fields are just assigned the type 'object' within the stubs file which makes the file perfectly compilable, but doesn't help at all when a shadowed file tries to reference the type (leading to unknown type errors).

The solution in all such cases are Dependency Hacks. ...So the principle here is simple: just comment out any lines that are throwing unknown type errors, being sure to comment each case with the dependency hack 'tag' (by default, this would be // !CSDEP!). This should permit the file to now compile. Using the tag has two purposes: firstly CSharpatron will clearly show any file that is 'live' but contains a dependency hack by coloring its icon *cyan*. Secondly, once you have all files compiling, have de-shadowed them (so dependencies are no longer an issue), you can remove the dependency hacks by sorting your workspace by the 'Live' field to see the cyan icons all grouped, and then using your text editor to search for and uncomment each hack.

I used dependency hacks in a few places for my own project conversion and if you're working with a large or complicated project its pretty likely you'll have places that may need to use this approach too.

Manually de-activating a C# file

Hopefully you'll never need this information, however if Unity is shut-down or crashes at a time when you have one or more active C# files containing errors then you'll may need to follow this process to get back to an error-free state *(and so gain access to a window layout that includes CSharpatron!)*.

NOTE: Close Unity before doing any of this and use Finder/Windows Explorer to perform these file operations. Please follow the steps very carefully to avoid potentially breaking script links within your mapped GameObjects!

- Move the broken C# file *and its .meta file* from **Plugins/CSharpatronConverted/<yourPath>** back to the original location (**<yourPath>**).
    - Rename the C# file as **<yourFile>.csTest**.
    - Rename the meta file from **<yourFile>.cs.meta** to **<yourFile>.js.meta**.
- Move the original .js file from **CSharpatronWork/Backup/<yourPath>** to the original location (**<yourPath>**).
    - Rename the file from **<yourFile>.jsOrig** to **<yourFile>.js**.

If you are using Source Control it may just be easier to revert the 'removal' of the .js/.js.meta/.csTest files and the 'addition' of the .cs file.

You'll also need to make repairs in the workspace file in order for CSharpatron to be kept in sync:

- Load the project file **'CSharpatronWork/Workspace.txt'**. You'll need to find the block relating to your file, it'll start with **'$<yourFile>'**.
- Delete the line containing **'@isLive'**. If present, also delete **'@isShadowed'**.
    - *Be very careful not to accidentally delete or damage Workspace.txt - there's lots of information in here that would be difficult to retrieve.*

NOTE: You can find some more 'best practices' mentioned in the Case Study appendix.

Support

If you encounter problems using CSharpatron, especially fatal errors, recurring conversion issues or errors in the building of the 'stubs' file, please let me know and I'll do my best to issue a fix.

Thanks for buying (or considering) CSharpatron. Best of luck with your conversions!

Please email: **csharpatron@spoonsized.com**.

Appendix A - Example Conversions

The following are all examples of script conversions performed by CSharpatron.

Original Unityscript	CSharpatron converted C#

A1

```
// Typed var declarations in Unityscript...
var intVar1: int;
var floatVar1: float;
var boolVar: boolean;
var stringVar1: String;
```

```
// Equivalent declarations in C#...
int intVar1;
float floatVar1;
bool boolVar;                        // Type name adjusted for C#
string stringVar1;                   // Type name adjusted to C# 'built-in type' form
```

A2

```
// Many (most?) declarations in Unityscript infer var type..
var intVar1 = 1;
intVar2 = 1;              // when not #pragma strict
var longVar1 = 1L;
var hexVar1 = 0x12345;
var hexVar2 = 0x12345L;
var floatVar1 = 1.0;
var floatVar2 = 1.0;
var floatVar3 = .1d;
var floatVar4 = 3.142F;
var sciVar1 = 5e+2;
var sciVar2 = 3e-200;
var charVar1 = "a"[0];
var charVar2: char = "a"[0];
var charVar3 = "\n"[0];
var str1 = 'Noodle';
var str2 = 1.0 + 'apple';
var str3 = "a" + "b";
var str4 = "a" + "b"[0];
var str5 = 'Rocko\'s Modern Life';
var intArr = [1, 2];
var s1 = ['"hello"', "world"];
var f1 = [1 + (3 * 4), 2, 3.5];
```

```
// CSharpatron will infer and declare an exact type in 99% of cases...
int intVar1 = 1;
int intVar2 = 1;                     // Var decl added for first assign to new var name.
long longVar1 = 1L;
int hexVar1 = 0x12345;               // Hex format not a problem
long hexVar2 = 0x12345L;             // Hex longs too
float floatVar1 = 1.0f;              // Added suffix (assuming 'decimals as floats' = true) OR...
double floatVar2 = 1.0;              // (Different output if 'decimals as floats' = false)
double floatVar3 = .1;               // Use of double specifier, short-hand supported too
float floatVar4 = 3.142F;            // Fully qualified constant always converted as float
float sciVar1 = 5e+2f;               // Scientific notation receives suffix too
double sciVar2 = 3e-200;             // Always double because single precision range is exceeded
char charVar1 = 'a';                 // Char type inferred, conversion to real char literal
char charVar2 = 'a';                 // Same output for explicit form
char charVar3 = '\n';                // Not confused by escaped chars!
string str1 = "Noodle";              // Double quotes mandatory for C# strings
string str2 = 1.0f + "apple";                // Type inferred from expression, quotes converted
string str3 = "a" + "b";
string str4 = "a" + 'b';                     // Conversion to 'real' char literal
string str5 = "Rocko's Modern Life";         // Quotes converted, fix-up for \' within string
int intArr = new int[] { 1, 2 };             // Reformatted array, implied 'new'
string[] s1 = new string[] {"\"hello\"", "world"}; // ...Array fixes plus quote fix-ups
float[] f1 = new float[] {1 + (3 * 4), 2, 3.5f};   // Type inference from varied field types
```

Original Unityscript	CSharpatron converted C#

```
// Untyped var declarations, continued...
var vecVar1 = Vector3(1.0, 2.0, 3.0);
var listVar = new List.<float>();
var dictVar = Dictionary.<String, int>();
var elemFromList = listVar[0];
var elemFromDict = dictVar["thing"];

// Complex assignments can make var type less obvious...
var complex1 = 5 | 4;
var complex2 = 5 || 4;
var complex3 = (5 + 3) && 2;
var complex4 = (5 * (2 + 3)) ^ 1;
var complex5 = 3 | (5 + 3);
var complex6 = 3.0 / (5 + 3);
var complex7 = ( (3 / (5.0 + 3) * 24) % 3) + 9;
var complex8 = 3 / (5 + 3.0);
var complex10 = complex1 && complex2;

// Multi-dimensional arrays...
var array2d =
[
    [ 1, 2, 3 ],
    [ 4, 5, 6 ],
    [ 7, 8, 9 ]
];
var anotherArray = array2d;
var arrayElem = array2d[1];

// Types derived from lib properties...
var testRnd1 = Random.value;
var testRnd2 = Random.Range( 1.0, 5.0 );
var normVec = vecVar1.normalized;
var dotProd = Vector3.Dot(vecVar1, vecVar2);
```

A3
```
// Untyped var declarations, continued...
Vector3 vecVar1 = new Vector3( 1.0f, 2.0f, 3.0f );
List<float> listVar = new List<float>();                // Fix-up for different generic syntax
Dictionary<string, int> = new Dictionary<string, int>();
float elemFromList = listVar[0];                        // CSharpatron fully understands generics
int elemFromDict = dictVar["thing"];
```

A4
```
// CSharpatron parses types in each expression, promoting according to operator precedence
int complex1 = 5 | 4;
bool complex2 = (5 != 0) || (4 != 0);                   // Comparison fix-ups necessary for boolean op.
bool complex3 = ((5 + 3) != 0) && (2 != 0);
int complex4 = (5 * (2 + 3)) ^ 1;
int complex5 = 3 | (5 + 3);
float complex6 = 3.0f / (5 + 3);
float complex7 = ((3 / (5.0f + 3) * 24) % 3) + 9;
float complex8 = 3 / (5 + 3.0f);                        // Float element means type promoted to float
bool complex10 = (complex1 != 0) && complex2;     // Infer type from exp. with inferred types!
```

A5
```
// Multi-dimensional arrays...
int[][] array2d = new int[][]
{
    new int[] { 1, 2, 3 },                  // Each sub array requires its own 'new'
    new int[] { 4, 5, 6 },
    new int[] { 7, 8, 9 }
};
int[][] anotherArray = array2d;        // Inferred type info is remembered throughout conversion
int[] arrayElem = array2d[1];          // Array accesses are fully understood
```

A6
```
// Types derived from lib properties...
float testRnd1 = UnityEngine.Random.value;  // Full type name to avoid ambiguity with System.Random
float testRnd2 = UnityEngine.Random.Range( 1.0f, 5.0f );
Vector3 normVec = vecVar1.normalized;
float dotProd = Vector3.Dot(vecVar1, vecVar2);
```

Original Unityscript

```
// Operators have return types just like regular functions...
var matrixVar = Matrix4x4.identity;
var quatVar = Quaternion.identity;
var vecResult1 = quatVat * vecVar;
var vecResult2 = matrixVar.GetColumn( 1 ) - vecResult2;
var vecResult3 = vecResult1 / floatVar;
var matResult = matrixVar * matrixVar;


// Conversion of type to bool for compare is automatic in US:
if ( boolVar ) Debug.Log( "Boolean is true" );
if ( intVar ) Debug.Log( "Int is non zero" );
if ( floatVar ) Debug.Log( "Float non zero" );

if ( !intVar ) Debug.Log( "Int is zero" );
if ( !!intVar ) Debug.Log( "Int is non-zero" );
if ( classVar )
{
    Debug.Log( "Class var is not null" );
}


// Ternary operators...
var t1 = classVar.myBool ? Screen.width: 640;
var t2 = intVar > 5 ? 1.0 * Screen.width: 640.0;
var t3 = intVar ? true: false;


// There are three ways to look up Unity components...
var go = GameObject();
var mf1: MeshFilter = go.GetComponent( MeshFilter );
var mf2: MeshFilter = go.GetComponent( "MeshFilter" );
// Preferred approach:
var mf3: MeshFilter = go.GetComponent.<MeshFilter>();


var subObj = go.Find( "Nemo" );
```

Converted C#

A7
```
// CSharpatron infers var type from expressions involving custom operators...
Matrix4x4 matrixVar = Matrix4x4.identity;
Quaternion quatVar = Quaternion.identity;
Vector3 vecResult1 = quatVar * vecVar;              // Type inferred from operator*
Vector3 vecResult2 = matrixVar.GetColumn( 1 ) - vecResult2;
Vector3 vecResult3 = vecResult1 / floatVar;
Matrix4x4 matResult = matrixVar * matrixVar;
```

A8
```
// In C# non bool types must be explicitly compared with 0/null...
if ( boolVar ) Debug.Log( "Boolean is true" );        // No change for actual boolean compare
if ( intVar != 0 ) Debug.Log( "Int is non zero" );    // Made explicit comparison to avoid C# err.
if ( floatVar != 0.0f ) Debug.Log( "Float non zero" );

if ( intVar == 0 ) Debug.Log( "Int is zero" );        // Not confused by negation!
if ( intVar != 0 ) Debug.Log( "Int is non-zero" );    // Not confused by confusing negation!
if ( classVar != null )                               // Comparison value chosen according to type
{
    Debug.Log( "Class var is not null" );
}
```

A9
```
// Ternary operators require optional values of equivalent type in C#...
int t1 = classVar.myBool ? Screen.width: 640;
float t2 = intVar > 5 ? 1.0f * Screen.width: 640.0f;  // 640 converted to float for type consistency
bool t3 = intVar != 0 ? true: false;
```

A10
```
// CSharpatron always converts to preferred look-up style using generic function GetComponent<>...
GameObject go = new GameObject();
MeshFilter mf1 = go.GetComponent<MeshFilter>();
MeshFilter mf2 = go.GetComponent<MeshFilter>();


MeshFilter mf3 = go.GetComponent<MeshFilter>();


GameObject subObj = go.Find( "Nemo" );
```

Original Unityscript

```
// Indirectly calling a function is trivial in Unityscript...
function TestFunc( intParam: int ): boolean
{
    return true;
}
var funcCallback = TestFunc;  // Var has type 'Function'

// A Unityscipt 'Function' var can be assigned absolutely any
// parameter types with no pre-declaration.
// It can return any type with no pre-declaration.
// CSharpatron can infer nothing useful from this type :(
var callbackVar1: Function;




var callbackVar2: Function;



// Function receiving Function type params...
function UpdateStuff( Function testCallback,
                      Function updateCallback )
{
    ...
    if ( testCallback( intVar ) )
    {
        updateCallback( stringVar, intVar );
    }
    ...
}
```

CSharpatron converted C#

```
// For C# calling a function indirectly is just as easy (using a Delegate) but is strictly typed...
bool TestFunc( int intParam )
{
    return true;
}
Func<int,bool> funcCallback = TestFunc;// Direct assignment to local var *can* infer delegate type



// Type inference for class scoped delegates & unassigned local vars is NOT possible.
// In these cases CSharpatron will log an error and write dummy values. MANUAL FIX-UP IS REQUIRED.
Func<PARAM_FIX_ME, RETURN_FIX_ME> callbackVar1 = null;

// However, type hints can be used. These are comment blocks that prefix unknown types.
// Their format is: '!CSTYPE! varName:<DelegateType>[,[repeat]]'

// !CSTYPE! callbackVar1:Action<string>;
Action<string> callbackVar2 = null;
// You can include as many (upcoming) named vars as you want in a single block.
// !CSTYPE! testCallback:Func<int,bool>; updateCallback:Action<string,int>;
bool UpdateStuff( Func<int,bool> testCallback,
                  Action<string,int> updateCallback )
{
    ...
    if ( testCallback( intVar ) )
    {
        updateCallback( stringVar, intVar );
    }
    ...
}
```

A11

```
// Unityscript almost never needs type-casts...
intVar = MyEnum.Value;
enumVar = 0;
prefab = prefabArray[PrefabType.MoonWizard];
intResult = sizeArray[enumType];
intArray[0] = floatVar;
intArray[0] = -floatVar;
avgTime /= frameCount;
var curve: Curve = t==0 ? Bezier(pts): CatmullRom(pts, true);
var go: GameObject = UnityEngine.Object.Instantiate("boo",
                        Vector3.zero, Quaternion.identity);



floatVar = 0;
doubleVar = 1;
unsignedVar = 5;
var str = "Num:" + (array != null ? array.length: "n/a");


// Multiple assignments with varying types is fine in US...
floatVar1 = intVar1 = floatVar2 = 12345.0;



// Unityscript doesn't have a problem with this:
transformArray = vecArray = null;
```

A12
A13
A14

```
// C# requires casts whenever types are converted. CSharpatron automatically adds them, as needed...
intVar = (int)MyEnum.Value;
enumVar = (MyEnumType)0;
prefab = prefabArray[(int)PrefabType.MoonWizard];
intResult = sizeArray[(int)enumType];
intArray[0] = (float)floatVar;
intArray[0] = (float)(-floatVar);          // Bracketing required in this case
avgTime /= (float)frameCount;
Curve curve = t==0 ? (Curve)new Bezier(pts): (Curve)new CatmullRom(pts, true);
GameObject go = (GameObject)UnityEngine.Object.Instantiate("boo", Vector3.zero, Quaternion.identity);

// CSharpatron works hard to avoid making unnecessary casts. In some cases, e.g. when a statement
// includes literals the literal may be reformatted to achieve the same effect...
floatVar = 0.0f;
doubleVar = 1.0;
unsignedVar = 5;       // Integer constants are implicitly 'signed' but CSharpatron won't cast them
string str = "Num:" + (array != null ? "" + array.Length: "n/a"); // Used '"" +' to promote to string

// C# doesn't let you assign one type to a different type without a cast. Assignment must be split...
floatVar1 = floatVar2 = 12345.0f;
intVar1 = (int)floatVar1;          // Int var receives suitably cast value

// C# won't let you assign a Vector3[] to a Transform[]! CSharpatron separates out elements...
transformArray = null;
vecArray = null;
```

Original Unityscript	CSharpatron converted C#

```
// A simple Unityscript class...
// NOTE: any non MonoBehaviour derived class is serializable.    A15
class TestBase
{
    var publicVar: int;
    private var privateVar: int;
    static var staticVar: boolean;
    function TestBase( intVar: int )
    {}
    function Foo() { Debug.Log( "Base" ); }
    private function Bar() {}
    static function Bar() {}
}


class TestDerived extends TestBase
{
    function TestDerived( intVar: int )
    {
        super( intVar );
    }
    function Foo() { Debug.Log( "Derived" ); }
}




// Calling a static function in Unityscript...          A16
TestBase.Bar();
// ...or...
myBaseTypeVar.Bar();


// In Unityscript the 'instanceof' operator compares type...
if ( myVar instanceof UnityEngine.GameObject ) { ... }    A17
```

```
// CSharpatron's conversion of the same class to C#...
[System.Serializable]      // Added to remain consistent with US - must be explicitly added in C#
public class TestBase                                           // 'public' keyword added
{
    public int publicVar;                                       // 'public' keyword added
    int privateVar;                                             // 'private' keyword removed
    public static bool staticVar;                               // 'public' keyword added
    public TestBase( int intVar )
    {}                                                          // 'public' keyword added
    public virtual void Foo() { Debug.Log( "Base" ); }   // 'public', 'virtual' keywords added
    void Bar() {}                                               // 'private' keyword removed
    public static void Bar() {}                                 // 'public' keyword added
}


[System.Serializable]
public class TestDerived: TestBase                              // 'public' keyword added
{
    public TestDerived( int intVar )
        : base( intVar )                // 'super' becomes 'base'. Repositioned as C# requires
    {
    }
    public override void Foo() { Debug.Log( "Derived" ); } // 'public', 'override' keywords added
}


// Calling a static function in C# must always be done using the class name...
TestBase.Bar();
// So this version is automatically fixed:
TestBase.Bar();


// C# uses the 'is' operator instead...
if ( myVar is GameObject ) { ... }     // (Type also simplified by CSharpatron)
```

Original Unityscript	CSharpatron converted C#

```
// Unityscript switch statements are very open to abuse...
    switch( state )
    {
        case 1: break;
        case 2:
        case 3:
            Debug.Log( "Shared state" );
            break;
        case 4:
            Debug.Log( "4" );
        case 5:
            Debug.Log( "5" );
        default:
    }
```

A18

```
// In C# 'drop through' must be specifically engineered...
    switch( state )
    {
        case 1: break;
        case 2:                          // Drop-through ok here since case 2 is empty
        case 3:
            Debug.Log( "Shared state" );
            break;
        case 4:
            Debug.Log( "4" );       // Drop-through here is illegal. CSharpatron will *warn*
            goto case 5;            // ...but use 'goto' to achieve controlled drop-through
        case 5:
            Debug.Log( "5" );
            goto default;           // Controlled drop-through using 'goto'
        default:
            break;                  // Closing break is mandatory. CSharpatron added one here
    }
```

```
// Mixing types in switch statements not a problem in US...
enum TestEnum { Val1, Val2 };

    switch( intVar )
    {
        case TestEnum.Val1: break;
        case TestEnum.Val2: break;
    }
```

A19

```
// C# requires consistent type for the case comparisons...
public enum TestEnum { Val1, Val2 };

    switch( intVar )
    {
        case (int)TestEnum.Val1: break;
        case (int)TestEnum.Val2: break;
    }
```

```
// In Unityscript type names can be used in comparisons...
    if (myVar.GetType() == MyClass)
    {
        ...
    }
```

A20

```
// In C# you must use the typeof() operator when comparing a Type...
    if (myType.GetType() == typeof(MyClass))
    {
        ...
    }
```

Original Unityscript

```
// US local variable scoping is extremely loose and can very
// easily leave you open to errors, e.g. this is legal script:
{
    if (testVar > 0)
    {
        var intVar = 1;
    }
    else
    {
    }
    DoSomething(intVar);
    // How is intVar still in a usable scope?
    // What does it contain? (0 prob. but who knows for sure?)
}
// Unityscript forbids re-use of a var name within a function,
// even when you might expect scope to be different...
{
    for(var i = 0; i < 16; i++) { DoSomething(i); }
    // Resetting loop vars like this is very common...
    for(i = 0; i < 5; i++) { DoSomethingElse(i); }
}
// Of course you might occasionally exploit how scoping works:
{
    for(var i = 0; i < 16; i++) { DoSomething(i); }
    for( ; i > 0; i--) { DoSomethingElse(i); }
}
// Another case, this one non #pragma strict...
{
    for(i = 0; i < 16; i++) { DoSomething(i); }
    i = 8;
    DoSomethingElse(i);
}
```

CSharpatron converted C#

A21

```
// C# has clearly defined scoping rules. CSharpatron will *warn* but because scoping problems are
// extremely prevalent (especially with C#'s stricter rules) it will also fix up bad scoping...
{
    int intVar = 0;          // Var declaration moved to a scope compatible with all usage
    if (testVar > 0)
    {
        intVar = 1;          // Now just an assignment
    }
    else
    {
    }
    DoSomething(intVar);    // Now we can be sure what's getting passed...
}
```

A22

```
// C# scoping rules won't permit 'reuse' of a loop var as in the Unityscript original.
// CSharpatron will automatically declare a new var...
{
    for(int i = 0; i < 16; i++) { DoSomething(i); }
    for(int i = 0; i < 5; i++) { DoSomethingElse(i); }   // New var declaration
}
// ...CSharpatron is smart enough to account for such a case...
{
    int i = 0;                                          // Declaration moved outside loop
    for(i = 0; i < 16; i++) { DoSomething(i); }
    for( ; i > 0; i--) { DoSomethingElse(i); }
}
```

A23

```
// This one is handled in a similar way, NOTE: need for var decl automatically inferred.
{
    int i = 0;                                          // Declaration moved outside loop
    for(i = 0; i < 16; i++) { DoSomething(i); }
    i = 8;
    DoSomethingElse(i);
}
```

Original Unityscript

```
// Script like the following is often used in Unityscript:
  var t = obj.transform;
  ...
  t.position.x = 0.0;
  t.eulerAngles.y += step;
  t.eulerAngles.z = 180.0;




// 'for x in y' style loops require attention in C#...
  for ( obj in objArray )
  {
      obj.DoSomething();
  }


// In some cases these loops will require manual fixes, e.g.
  for ( col in colorArray )
  {
      col = Color.green;
  }


// In Unityscript, passing params 'by reference' isn't
// something that you have to think about...
  Vector3.OrthoNormalize(myRight, myUp, myFwd);
  ang = Mathf.SmoothDampAngle(ang, targ, vel, 0.5, 20.0);
  hit = Physics.Raycast(pos, fwd, out hitResult, 5.0f, mask);
```

CSharpatron converted C#

A24
```
// In C# it is illegal to *write* to a 'value type return value'. CSharpatron makes these fixes:
  Transform t = obj.transform;
  ...
  var tmp_cs1 = t.position;          // Assignment to new temp var
  tmp_cs1.x = 0.0f;                  // Modify temp var
  t.position = tmp_cs1;              // Re-assign from temp
  var tmp_cs2 = t.eulerAngles;
  tmp_cs2.y += step;                 // Access to shared fields on consecutive lines...
  tmp.cs2.z = 180.0f;                // ...is optimized to utilize the same temp var
  t.eulerAngles = tmp_cs2;
```

A25
```
// The loop becomes a 'foreach' loop. The element type must be explicitly declared...
  foreach ( GameObject obj in objArray )
  {
      obj.DoSomething();
  }


// ...In C# the loop iteration var is READ ONLY. CSharpatron will flag the following as an error...
  foreach ( Color col in colorArray )
  {
      col = Color.green;          // ERROR! Manual fix required since resolution is case dependent!
  }
```

A26
```
// In C# these function calls all have parameters that must receive additional qualifiers...
// CSharpatron automatically adds them, as required.
  Vector3.OrthoNormalize(ref myRight, ref myUp, ref myFwd);
  ang = Mathf.SmoothDampAngle(ang, targ, ref vel, 0.5f, 20.0f);
  hit = Physics.Raycast(pos, fwd, out hitResult, 5.0f, mask);
```

Original Unityscript:

```
// Coroutines usage requires very little syntax in US…
  StartCoroutine(MyCoroutine1(10));
  // Alternatively...
  StartCoroutine("MyCoroutine2");


void MyCoroutine1(someVar: int)
{
    ...
    // Just containing a 'yield' makes this a coroutine
    yield;
}


void MyCoroutine2()
{
    while(1)
    {
        ...
        yield WaitForSeconds(0.5);
    }
}


// You might write a function to kick off a coroutine...
void StartMyCoroutine()
{
    return StartCoroutine( MyCoroutine(0) );
}


// Unityscript properties are converted (v1.2)...
var health: float;
public function get Health(): int {return health;}
public function set Health(value: int) {health = value;}
```

A27

CSharpatron converted C#:

```
// In C# coroutines require more careful treatment...
  StartCoroutine(MyCoroutine1(10));
  // Alternatively...
  StartCoroutine("MyCoroutine2");      // This one can be stopped with StopCoroutine("MyCoroutine2")


public IEnumerator MyCoroutine1(int someVar)
{
    ...
    // C# always requires that you return an IEnumerator object or 'null' so CSharpatron fixes it:
    yield return null;
}


public IEnumerator MyCoroutine2()
{
    while(true)
    {
        ...
        yield return new WaitForSeconds(0.5f);
    }
}



// This will actually cause a CSharpatron ERROR since the declared return type of the function
// would need to change (to become IEnumerator) which would suggest the function to be a coroutine
// when it isn't! ...Just remove the 'return'.
```

A28

```
// Converted properties...
float health;
public int Health { get {return health;} set {health = value;} }
```

Appendix B - Common Conversion Errors

Whenever CSharpatron encounters unexpected formatting, script that cannot be parsed, or it makes a significant fix-up, it may write out an error or warning to the log file. There are many internal error checks, but in general there are two classes of message:

a) Warnings or errors that are 'expected' - i.e. things that are fairly normal to see when processing normal, correctly formatted files. These may well be issues that require your attention; they will be things that you might expect to see during the conversion process.

b) Warnings or errors that should *not* normally be seen. These should only occur if a user attempts to convert badly formatted source *or* if the input is highlighting a bug or omission in CSharpatron.

This appendix will discuss output message of the *first* type to help you understand the possible action you should take.

'Fix .JS' Error Types

These are errors highlighting problems encountered by CSharpatron that you *will* need to manually resolve (within the original .js source) before a file can be successfully re-converted into compilable C#.

JS00: Non void function failed to return a value. This will cause a C# error.
You have a .js function that declares a return type but the function fails to actually return anything.
Fix: add missing return statement.

JS01: Hashtable initialization not supported in C#. Type not converted.
CSharpatron encountered in-line initialization of a hash-table. C# doesn't possess equivalent syntax (?) and so CSharpatron is unable to offer any automatic fix-up.
Fix: you will need to initialize your hash table in a different way - perhaps runtime assignment of values. Perhaps the hashtable could be replaced with generic Dictionary usage?

JS03: Cannot automatically fix-up use of .js (only) 'Array' type: <varname>. Suggest replacement with a generic List.
A usage of a Unityscript array type has been encountered. This container type is not available in C# (in fact its use tends to be discouraged in Unityscript too). CSharpatron has no automatic conversion for this since there are a number of possible replacements.
Fix: you should replace usage of the Array type with another container class. Typically a generic List (List<T>) makes a great replacement candidate. There's plenty of discussion to be found on this subject, e.g. here.

JS05: Assignment to var <varname> is illegal in C# (read-only 'foreach' iteration var?)
You have script that is writing to the iteration var of a Unityscript 'for X in Y' style loop. This loop type is converted to a 'foreach' loop in C#. It isn't legal to write to the iteration var of a 'foreach' loop.
Fix: Update the loop to become an indexed type and re-convert.

JS06: Multi-assignment includes an assignee that is a value type return (not assignable in C#). This can't be auto-fixed. Please separate out any <var>.<field> assignee.
You have a multi-assignment statement that includes a field that is a value type return, e.g. something like

```
myVec = transform.position.x = 0.0;
```

This kind of write isn't legal in C# and although CSharpatron has automatic fix ups for value type return writes, it currently cannot handle this fix in the context of a multi-var assignment. See here.
Fix: separate out the write to the value type return field and re-convert.

JS07: Use of class var/property <identifier> whose name is illegal in C#.
There are many reserved keywords in C#. CSharpatron has encountered a class scoped symbol that would be illegal. CSharpatron can automatically fix-up names for locals, but doesn't attempt to do so for class vars since the type may be accessed by other files plus any new name is something you'd most likely prefer to be in control of!
Fix: rename the var (being careful to avoid C# keywords - see here), and re-convert.

JS08: A function can't 'return' a co-routine in C#; auto addition of StartCoroutine() not possible here.
CSharpatron would usually automatically add StartCoroutine for you, but cannot do so if the called coroutine is part of a return statement. (This is because in C# a coroutine returns IEnumerator and so making such a fix would need to alter the prototype of the function which would have a ripple effect across multiple files and also denote the function as a coroutine when it isn't).
Fix: remove the 'return' statement and re-convert.

JS09: Identifier name <name> is discouraged since double-underscore is reserved for C# compiler usage. (Auto rename failed)
The converter encountered a symbol whose name would be illegal in C#. In normal (local var) cases auto-renaming would occur but isn't currently supported in this double underscore case.
Fix: rename the variable and re-convert.

JS10: Identifier name <name> is not permitted in C# (most likely a reserved keyword). (Auto rename failed)
The converter encountered a symbol whose name would be illegal in C#. In normal (local var) cases auto-renaming would occur but wasn't possible in this case.
Fix: rename the variable and re-convert.

JS11: Var declaration <name> would redefine a var in the same scope. Automatic fix-up is not possible.
The converter encountered a symbol whose name would be illegal in C#. In normal (local var) cases auto-renaming would occur but wasn't possible in this case.
Fix: rename the variable and re-convert.

JS12: 'in' keyword has no equivalent in C#.
Despite not being documented anywhere I've been able to find, I've witnessed an apparently functioning Unityscript file that employs a Javascript style 'in' operator (which tests for the presence of a given property/key in a supplied array/container). On the basis of lack of documentation and C# not possessing an equivalent keyword, CSharpatron offers no automatic fix-up of this!
Fix: replace the 'x in container' format with a function query appropriate to the container type, e.g. myList.Contains(), myDict.ContainsKey() etc.

JS14: A switch expression cannot be of 'System.Type' in C#. (Convert the type to a string?)
The converter encountered a switch statement with a case condition of type 'System.Type' (i.e. a class-name). This isn't permitted in C#.
*Fix: change your switch statement to test the type's name, e.g. 'var.GetType().**Name**' and update your case statements to string format accordingly.*

JS15: Type for parameter <name> is missing. It can only be inferred as 'System.Object' which is probably wrong.
While parsing a class based field, CSharpatron couldn't locate any type information for the field. This may be because the field is part of an assembly that hasn't finished compilation.
Fix: check that the assembly containing the type is fully compiled (most likely Assembly-UnityScript or a counter-part).

JS16: Assignment to a value type return value, this isn't legal in C#. Auto fix-up is not possible for type field <type-field>.
A write to a value type return field was encountered (e.g. attempting to write to a vector field of a position belonging to a transform: accessing transform.position returns the position by value rather than by reference and so while a component read is valid, a write would be storing data to a temporary rather the the transform's actual position instance).   In most case auto-fix up is possible, but in this case CSharpatron's current fix-up method is unable to achieve this. See 'here'.
Fix: you will need to manually update the field write, e.g. from 'myVar.field = value' to something like 'VarType temp = myVar; temp.field = value; myVar = temp'.

JS17: Assignment to a value type return value, this isn't legal in C#. Auto fix-up is not possible for type <type>
See JS16.

JS18: Class var type not explicitly defined for <varName> (will default to 'object'.) This is probably unintentional and will lead to unexpected casts in C#.
CSharpatron found a class variable declaration with no defined type. In Unityscript this would silently receive 'Object' type and could easily be being used in odd ways without you having any idea (e.g. Unityscript will let you assign an 'int' value to an 'Object' and later compare it with another 'int' without any warning).
Fix: You would do best to give the variable a deliberate type (even if that is still 'Object') and then re-convert.

'Fix .CS' Error Types

These are errors highlighting problems encountered by CSharpatron that you will need to fix before a converted file can possibly become compilable. Fixes should be made in the .csTest or .cs version of a file.

CS00: Delegate type <type> must be fixed by user.
CSharpatron located an instance of a Unityscript **Function** type var. The output source will have a placeholder delegate type written in place of the 'Function' type-name, e.g. 'Func<PARAM_FIXME, RETURN_FIXME>'. This error is telling you that you will need to manually fix up those delegate type-names to something that suits your use of the variable.
Fix: fix the type-names in your .csTest/.cs file as suggested, or optionally supply a type-hint in your .js file and re-convert. See more discussion on fixing up delegates in Appendix A11 or here.

General Error Types

These are errors highlighting some kind of internal problem during the conversion process. In some cases they are 'harmless' by-products of **Fix .JS** or **Fix .CS** errors. In other cases they may indicate a problem with your source file or even a bug in CSharpatron.

G00. Source file is too large to process (<size>). Limit = 1024k
CSharpatron has a file size limit of 1024k for source .js files. Anything close to this size will also likely take a *long* time to convert! ...Sorry.

G51: 'in' keyword has no C# equivalent in this context.
Typically a by-product of JS12.

G56: Delegate call where type is still a 'Function' or types are unknown. Try to make the declaring class a live C# file and then Re-Convert.
When working to replace use of Unityscript Function vars with C# delegates you will face errors in the files that declare those vars and errors in files that use them (or which perhaps pass in function references to those types either by writing to them directly or calling registration functions). This error will be seen in cases of the latter type where script is attempting to make use of a delegate type.

Fix: there's probably not much that can be done here other than the error's suggestion: don't try to activate this file until the file declaring the delegate has been activated first. This was the approach I took with my own project conversion and these errors just 'fell away' as expected with suitable file activation ordering.

G60: System.Array type deprecated.
Most likely a by-product of JS03.

G115: Scope ended without var assignment supporting type inference for local <name>
CSharpatron finished parsing a variable scope without being able to determine the type of a variable (i.e. it wasn't assigned). This the error form of W29 which you will only see if you choose 'Only Inferred Type' for the 'Local var conversion' option.
Fix: Use 'Prefer Inferred Type' or ensure that the variable in question has either a declared type of an assignment to infer from.

Warnings

These are typically just information telling you of a fix-up that the converter performed or a (probably) harmless issue encountered during conversion.

W00: Class <name> will be relocated outside of Mono class scope to maintain global name scope in C#.
This is informing you that a class will be relocated with the .cs file in order to allow the main file class to be contiguous within its own (now made explicit) class declaration. See here.

W01: Failed to find '#pragma strict'. Please note that CSharpatron only supports typing by *first* assignment.
CSharpatron was primarily written to convert source files with #pragma strict declared, i.e. which introduce new vars using the 'var' keyword, and which assign type to a var just once - either at point of declaration or upon first assignment. Non #pragma strict files can certainly be converted but if your script includes reassignments that change var-type, CSharpatron will likely try to type-cast them and *won't* update its internal understanding of your var's type. ...*This is just a warning to* **beware**.

W03: Auto fix-up added a 'break' before switch end as required by C#.
This is letting you know that a switch statement was found with a trailing 'case' or 'default' that lacked a 'break' keyword in Unityscript. This would be illegal in C# so CSharpatron added one.

W06: Function <name> declares no return type but returns non-void. Reflection derived type <type> will be used.
You have a function that declares no return type but which is returning a non-void type nevertheless! CSharpatron can lookup the active return type using Reflection and fix up the function prototype as required by C#.

W07: Function declared 'virtual' for C#: <func-name>
A function prototype is being converted where the function has been determined to be a virtual base. Unityscript doesn't need this to be explicitly stated but C# does.

W08: Function declared as 'override' for C#: <func-name>
A function prototype is being converted where the function has been determined to be an override of a virtual base function. Unityscript doesn't need this to be explicitly stated but C# does.

W09: C# doesn't permit drop-through from non-empty case. Auto fix-up inserted <goto case/default>.
CSharpatron found a non-empty switch case that was dropping through to the case below it in Unityscript. This may be deliberate and so CSharpatron fixes this up for C# by adding an explicit 'goto' the case in question (otherwise you'd see an error). ...Then again, this might not be deliberate, hence this warning!

W11: Reused 'for' loop var <varname> is not in scope for C#. A new var of the same name and type has been automatically declared.
Unityscript's weird scoping rules mean that unless you like thinking up a custom loop var name for every single loop, then chances are you have plenty of functions that declare one var in their first loop instance, and then re-use that same var in subsequent loops within the function. C#'s scoping rules don't permit this, and so CSharpatron adds explicit declarations for any 'reused' loop vars. This warning is just letting you know that technically at least, a new var was introduced into your script.

W12: Can't infer type from 'null' assignment: <varname>. [Type will be determined by first assignment.]
Type information for class vars can (usually) be looked up via Reflection. This isn't the case for local vars, and so in cases where a type isn't declared and no assignment occurs, CSharpatron will leave the type 'open' until the first assignment. This warning is just to let you know - sometimes that first assignment might not yield the type you expect (e.g. if you're declaring a base type var that is to be assigned various derived class types).

W13: Class var type not explicitly defined, will default to 'object'. This may be unintentional and may lead to unexpected casts in C#.
This warning denotes that a *class var* has no declared type in Unityscript. This may be unintentional and will most likely lead to ugly casts upon usage.

W14: Found class var before 'using <libs>' insertion point - output will be ugly! Ensure .js 'imports' or '#pragmas' (e.g. 'strict') come first in your file.
In order to determine where it can neatly add new 'using' commands along with file class's class declaration CSharpatron considers the placement of any 'import' directives or #pragmas - things typically top-most in a file. This warning is letting you know that we found a class var *earlier* in the file than either of these things: output may well be ugly!

W15: Auto fix-up for assignment to a value type return value (not legal in C#): <var-field>.
CSharpatron identified and automatically fixed up a write to a value type return (e.g. a component of a vector belonging to a transform). See JS06 or JS16 or here for more information about this.

W19: Multi variable assignment split apart due to C# type incompatibilities.
The converter found an instance where multiple left-hand assignees are receiving a single right-hand-side value. In this case one or more the assignees isn't compatible with the type of its neighbor in the assignment chain. To fix this CSharpatron has split apart the assignment to span multiple lines with RHS values type-cast as required.

W20: Cannot determine return type of delegate ('Function') call.
An instance of a delegate call where the return type is unknown. See also CS00.

W22: Access to static field must be updated for C#. Auto fix-up replaced <original-access> with class-name <fixup-access>.
CSharpatron identified usage of a static field or function that is accessed via a class instance. In C# this isn't legal syntax - the class name is the only valid prefix. CSharpatron fixed up the access accordingly.

W23: Constructor's parent call-back relocated outside of function body as required by C#.
C# requires that the callback to a parent class's constructor is positioned between a function prototype and the function body. CSharpatron found and fixed such an instance.

W24: Local identifier name <var-name> is illegal in C#. Fixing up as <new-var-name>. All references will be updated to reflect name change.
CSharpatron found a local variable with a name illegal in C#. The name was automatically adjusted and any references within the function updated accordingly.

W25: Var declaration <var-name> would redefine a var in the same scope. Fixing up as <new-var-name>. All references will be updated to reflect name change.
Certain variable scoping fix-ups can encounter the need to rename a variable declaration to avoid collision with a var of the same name that is still in scope. This is such an instance; references to the var have been updated too.

W26: Converted 'char' field to 'char[]' for C#. Perhaps this is a call to a string function expecting 'char[]' that was being given a "X"[0] element in Unityscript?
This is warning that a function call which in Unityscript was passing a 'char', has been updated to pass a 'char[]' array in C#. String.Split is an example of a function that requires this fix-up.

W27: Converted Finalize() call into destructor for C#.
A call to Finalize() was converted into a C# destructor. This is to avoid C# compile error: 'CS0249: Do not override `object.Finalize()'. Use destructor syntax instead'. See discussion here.

W28: Relocated var decl for <var-name> to fix out-of-scope C# access.
CSharpatron located a reference to a variable that appears to be 'in-scope' for Unityscript but would not be so for C#. To address this, the declaration for the variable has been moved to a position whose scope is mutually compatible with the original declaration as well as this attempted access. You may see more than one of these warnings for a given var. This is a consequence of CSharpatron moving the var to a broader and broader scope in order to encompass all accesses. See Appendix A21.

W29: Scope ended without var assignment supporting type inference for local <var>. Falling back to 'object'.
The converter has reached the end of a scope that contains a variable declaration lacking enough information for a type to be inferred. The variable will be assumed to be of 'object' type. This may not actually be correct if in Unityscript there was an 'out of scope' assignment to the type, but typically seems a better option then allowing the variable to be left untyped. See also: **W12**.

W30: Void return in function with non-void return declared. Fix-up will return a suitable 'zero' value.
CSharpatron just reached a 'void' return statement within a function for which the declared return type is non-void. A zero value of suitable type will be automatically returned instead.

W31: Class var(s) for Mono class <class-name> will be relocated to reside within C# class definition.
In a file containing classes other than just the main Mono derived file-class, CSharpatron just found a block of vars relating to the file class which will need to be relocated in order to sit within the contiguous class declaration that our output file needs to contain. See here.

W32: Class method <func-name> for Mono class <class-name> will be relocated to reside within C# class definition.
Similar to W31 but in this case it is a file-class *function* that will need to be relocated.

W33: No assignment of 'for' loop var. Zero value will be assigned to match type.
A 'for' loop failed to make an assignment to a loop var. CSharpatron will initialize the var with zero.

W34: Added declaration of 'for' loop var. Type inferred as: <typeName>
A 'for' loop failed to declare a variable and no previous variable with the same name was found. A new var declaration will be automatically added.

To help give you an idea of how to (best?) use CSharpatron to convert many files - in fact an entire project - I'll offer a run through of how I went about converting my own project using the tool.

First let me just say that my codebase is *not* straightforward and perhaps not especially typical since I have a great deal of complex, custom code and quite a bit of statically initialized data. Before writing CSharpatron I actually considered converting the project by hand (using other available tools to help), but after working my way through a few files, I came to the realization that I would certainly be looking at many *weeks* of work!

In total the code-base comprises **267 .js files totaling 107,075 lines**. I have files ranging from a handful of lines all the way up to a few over-grown monsters of 2000+ lines. I believe I make use of pretty much every aspect of UnityScript to some degree, plus I have a few files with code taken from the Unity wiki and the like, so I figure (hope!) that this is a pretty good test environment for CSharpatron.

So, on a fine Summer's morning, I began the conversion...

- First I brought up the CSharpatron Control Center (Windows/CSharpatron) and docked this in a clearly visible spot.
- I configured *Conversion Options* as I needed: one *Custom define*, a tick for UNITY_EDITOR and UNITY_IPHONE in the *Platform defines*, and also a tick for *Convert all caps class var decl to 'const'* since that suits my code style. All other *Conversion Options* and *Layout Options* I left at their defaults since those made sense for me.
- I decided to convert all my source files in one go. The easy way to do this was to select my top-level 'Source' folder since this contains all of my .js source files (CSharpatron doesn't care what *else* is in there, it picks up just the .js files). After a moment the convert button appeared: *Convert (267)*.
- I clicked *Convert* ...and went off to play Battleheart Legacy for a while!
- When I checked back later, I found all the files converted and now listed - along with all conversion log information - in CSharpatron's Workspace View. My overall error status was:

Status	Time	Total time
Initial conversion of all 267 scipt files. **0 Fatal errors. 405 Errors. 2269 Warnings.**	23m	23m

Of the errors, there were **275 'Fix .JS'** errors and **74 'Fix .CS'** (these all being related to my fairly extensive use of the Function type).

Of the remaining 56 errors, 42 were knock-on effects from Function vars not being convertible, 12 were knock -on effects of Unityscript Array vars not being convertible and the last 2 were bad assignments that I have no idea why Unityscript wasn't catching!

I decided to first fix up the the the 'Fix .JS' errors, focusing on one type of error at a time.

First up, I went through all errors relating to illegal property naming. These were all class variables named things like 'object', 'string', or 'params'. Fixing these was a little fiddly since use of each class instance typically spread into several other files. Along the way I also ran into the super helpful error message *'Quack is not a generic definition'* which it turned out was a consequence of some non pragma strict Unity Editor files (which I wasn't converting) referencing types that I had now renamed in my project source! ...Anyhow, with my fixes complete, I reconverted the modified file and saw a nice payoff:

Status	Time	Total time
Fix up illegal class variable names **0 Fatal errors. 209 Errors. 2322 Warnings.**	45m	1h 08m

Next I had three files still making use of the Unityscript Array type. Bah - how had I not fixed this before! Still, not difficult to resolve - in each case it was simple to switch the Array type for a generic List. Files reconverted. More good progress…

Status	Time	Total time
Removing use of Unityscript Array type **0 Fatal errors. 168 Errors. 2383 Warnings.**	30m	1h 38m

Now looking at miscellaneous problems with just a few instances of each type:

- 4 functions where I was failing to return a value in functions declaring a return type. Thanks Unityscript! Easily fixed.
- 15 cases where I was writing to value type return values in ways that CSharpatron couldn't fix using its current fix-up implementation (a small number compared to the many it had fixed, but a ToDo note added for possible future improvement (NOTE: addressed in v1.1)).
- 7 places where non void functions were failing to actually return a value.
- 2 places where I was writing to a foreach iteration var.
- 2 switch statements whose case statements were relying on Type fields. (Fixed up by changing the switch value to 'var.GetType().**Name**' and substituting string based cases.)

Status	Time	Total time
Fixing various limited instance issues **0 Fatal errors. 126 Errors. 2402 Warnings.**	25m	2h 03m

At this point I had fixed all of the **'Fix .JS'** issues. I decided that rather than fix up Function type issues 'post' conversion, I preferred to fix them as far as possible with more edits to my .js files. I went through all

cases where I was declaring Function type vars adding suitable 'Type Hints' before each declaration (see Appendix A11).

...This was easily the most taxing part of my conversion work since figuring out the exact prototype in each case was sometimes awkward. Worst of all: in several cases I had exploited the fact that one Function var can be assigned totally different functions, both with and without return types!

Anyhow, with type-hints added, reconverting those files gave the results I was looking for:

Status	Time	Total time
Adding type-hints for all 'Function' vars **0 Fatal errors. 30 Errors. 2524 Warnings.**	50m	2h 53m

Now I had eliminated all **'Fix .JS'** and '**Fix .CS'** errors, and was left with just 30 unqualified errors, all relating to Function conversion issues in files not responsible for that actual Function var declaration. A quick scan through warnings suggested nothing alarming - lots of fixes for bad local var naming, out of scope variables, static field fix-ups, etc. ...Just CSharpatron doing its job.

You may have noticed that as I was fixing errors, my warning count actually increased. This is mainly because errors can block full parsing, so only once they're removed do we see all the automatic fixes that CSharpatron wants to make along with their associated warnings.

Generally warning count shouldn't be cause for concern - warnings are emitted mainly to offer you a record of more significant fixes just in case you encounter problems when finally running your project.

I decided it was time to finally start activating C# files.

• Easy files first: I had 37 files with 0 X-Refs or just 1 already fulfilled X-Ref (a utility class already written in C#). Not anticipating any compile errors in these mostly simple files, I took them all live in one go using multi-select, toggle status. Wait for the build. No errors. ...Next!
• I double clicked the X-Ref sort button to help highlight more zero-dependency files ready to go live.
• 6 more files now had zero dependencies. I activated them all using group select again. Compile. Sort.
• Now 3 more with no dependencies. Group-select. Activate.

Status	Time	Total time
Make zero dependency files live **0 Fatal errors. 30 Errors. 2524 Warnings.**	10m	3h 03m

Now I had 46 live C# files representing 7% of my project. All clean compiles. Nice. But… No more zero dependency activations.

TIP: at this point in the conversion process everything that has been done to the project is 100% testable. I ran the game to confirm that yes, with 7% C# now active it still functioned the same! Reassuring...

Ok, time to rely on 'file shadowing' to make it possible to activate more C# files...

• First step of working with shadowed files: I used the **ReBuild Stubs File** command to generate the file CSharpatronTemp.cs. I also ticked the Conversion Option **Build stubs file during conversion** in case I needed to reconvert anything.

I'm not sure there's a best ordering strategy for making files live via shadowing… For me, what seemed logical was to try and get simpler files activated first (i.e. ones with lower X-Ref counts, lower line counts). . After a while working through files like this, I switched to a different strategy: switch sort-mode to 'File-path' and focus on groups of related files.

My process for essentially the remainder of my project conversion was:

• Activate a group of 3 or 4 files. Wait for compile.
• Roughly 1 in 10 files would throw up C# errors. These would either be:
   a) Problems relating to attempted use of delegate types for which declaring files hadn't yet been made live. (As I mentioned before: the stubs file 'cheats' when it comes to delegates and just pretends that they are of 'object' type. That obviously doesn't fly once you have a file trying to make concrete use of a delegate var or function with delegate parameters)
   b) Some bizarre, eye-opening error related to Unityscript's loose (virtually non existent?!) type checking. See Problems Revealed.
   c) Something that CSharpatron hadn't been able to fully fix.
• When encountering type a) issues, I would make a judgment call: if there were a handful or less references to fix in a file then I would comment out the offending lines, *making absolutely certain to add a Dependency Hack tag*. Where more fixes would have been necessary, I cancelled activation of the file with the intent of making it live later, once I had activated the class responsible for declaring the delegate in question.
• Type b) errors actually proved to be very simple to fix in every case. I fixed each of these as they came up and moved on.
• Type c) issues actually proved to be extremely rare. See Manual Fixes Required.

I had a lot of files to work through, and in most cases their external file dependencies were huge (like 190+ files out of 267!), so for me there was little chance of actually running the game with any of the shadowed

files made fully active until the last file was activated. (Although just to be clear, the game was still runnable at all times just still operating with the .js versions of those shadowed .cs files.)

With the last file finally activated I had three last things to do:

- I used the **DeShadow** command to make all my 'shadowed' files truly active. There was a lot of stuff happening at this point: 221 files having their .js versions made inactive (and backed up), having their .cs versions edited to remove them from the 'shadow' namespace. Then one final *long* build as the compiler churned through all the newly edited files.
- I removed Dependency Hacks - basically just re-instating commented out lines. Of course I used the Dependency Hack tag as a search string to find all occurrences. (With related types now compiled to C# all of these edited files now compiled fine too).
- And finally: I used the **Finalize** command to copy C# files back from their temporary location within 'Plugins' and back to my source folder. This also cleaned up work files (including the 'stubs' file), but didn't remove the actual CSharpatron project or backed up Unityscript files.

The final work summary:

Status	Time	Total time
Make high dependency files active, fix C# errors **0 Fatal errors. 30 Errors. 2524 Warnings.**	1h 25	4h 28m

As I suspected none of those 30 errors had actually impeded my progress, all just fell away.

Throughout the whole process of getting the C# files compiling I encountered **30** files that had C# compiler errors. **8** of these files had Function dependency issues that I used Dependency Hacks to circumvent. The other **22** exhibited errors you can see documented below. Mostly they were ugly Unityscript source elements - *or actual hidden bugs* - revealing themselves (see Problems Revealed). Just a handful of issues were things that CSharpatron hadn't been able to resolve (see Manual Fixes Required).

The Moment of Truth

With conversion complete, no dependency hacks and no C# compile errors, my project was finally ready to be run in its new, C# form. With great anticipation I hit the *Play* button and…

EXCEPTION!

Two in fact.

I'm guessing that switching files to C# must change startup order somehow? ...For whatever reason I had two system classes that were falling over in initial update passes due to 'null' fields. I added '!= null' tests in each case and ran again. This time...

SUCCESS!

...And then one more run-time issue: an exception within my save-game class. It turns out that if you write an 'int' to a hash-table then accidentally read it back as a 'float' Unityscript is perfectly happy. C# not so much. Anyhow...

My project is now running in C# with no apparent side effects, in fact - less bugs than before! *Woohoo.*

UPDATE for v1.1:
After optimization work, CSharpatron's conversion times have dropped significantly.
For my full project, the initial conversion now takes just under 12 minutes rather than 23.

Build Time Improvement

The conversion process made it fairly clear that most files in my project have *huge* dependency lists. This isn't really a shock since unlike languages like C or C++ where header files must be included in order to utilize external types (and that makes you *think* about dependency overhead), the .Net languages really encourage you to go buck-wild and use whatever types you need. The upshot of this is that making a simple change in a typical file actually causes the majority of the project to be rebuilt.

Having done some timing tests based one such file I can report:

Typical build time for Unityscript:	**56s**
Typical build time for C#:	**18s**

A nice visible reward for the conversion effort!

Problems Revealed

I don't believe that I am generally a 'sloppy' programmer, but these were all problems unearthed by my project conversion, things that Unityscript had let me 'get away with' where a stricter language would have caught the error on first compile:

- An instance where the Unityscript compiler had let me type 'if (keys[i] === key)'. *(CSharpatron error)*
- An instance where a function had a for loop: for (i = 0; i < xpt.values.length; i++) where the loop var 'i' hadn't been declared at any prior point in the function (and no class var 'i' either). ...And this *was* a #pragma strict file! *(CSharpatron error)*
- Seven instances where non void functions were failing to return a value. *(CSharpatron errors)*
- Many instances of functions where a return type had inadvertently been omitted but which were returning non void types. *(CSharpatron auto-fix warning)*
- One switch statement with a case that was dropping through unintentionally. *(CSharpatron auto-fix warning)*
- An instance where Unityscript didn't see any problem with 'System.Int32.Convert(stringVar * float)' that should have been 'System.Int32.Convert(stringVar) * float'. *(C# compile error)*
- Two cases where I had local vars with the same name as class vars where both symbols were being referenced on the same line, e.g. 'color = Mathf.Lerp(color, pulse, frac)'. *(C# compile error)*
- An instance where Unityscript was happily letting me compare a float and a Vector3. *(C# compile error)*
- A switch statement where a case value was accidentally duplicated. *(C# compile error)*
- Four errors due to the fact that Unityscript doesn't care if a virtual overload whose base returns non-void has no return statement itself. *(C# compile error)*
  - Interestingly, if a base method has no return but a derived version attempts to return something, *that* gives an error!
- An instance where a function returning a bool had a return statement within an if statement but none outside of it. *(C# compile error)*
- Several instances where I was accidentally calling parent constructors passing them uninitialized class vars instead of the passed-on constructor parameters I intended. *(C# compile error)*
- An instance where I was using 'as TypeX' to cast a function's return type but had inadvertently called the wrong method and Unityscript was in fact assigning a completely unrelated type - that C# would have considered non-convertible - without any warning! *(C# compile error)*
- An instance where I had somehow managed to add a 'new' operator in front of a static 'typeVar.func()' call that Unityscript had no problem with. *(C# compile error)*
- Even with #pragma strict present, Unityscript doesn't care if you don't declare the type of a function parameter (I had about 20 accidental cases of this). *(C# compile error)*
  - Such parameters are just left as type 'object' so whether you ever see the problem is completely down to what the function might do with those inputs.
- An instance where I accidentally wrote 'nextUpdateState = X' when I should have put 'nextUpdate**Stage** = X'. (Again, this was a #pragma strict file - shouldn't that catch this sort of thing?). *(C# compile error)*
- In Unityscript accessing a hashtable element that isn't of the expected type 'just works', in C# you get a type-cast exception. *(C# run-time error)*

During development of CSharpatron I also came across:

- A place where an out of scope variable was *always* being used uninitialized. *(CSharpatron error)*
- Six places where out of scope vars could potentially be used with uninitialized values. Three of these directly explained erratic hard to repro bugs that existed in my game.
- One place where a copy/paste had led to an out of scope var from earlier in a function being used instead of a intended, new loop var.

NOTE: These kind of problem are now fixed automatically since CSharpatron will 're-scope' and initialize variables such that all accesses are within the same scope.

Manual Fixes Required

These were the few fixes I had to make which weren't in direct response to CSharpatron errors but which instead arose from C# compilation:

- I had a foreach loop iterating a Hashtable. CSharpatron had inferred 'object' as an appropriate type for the iteration var but that didn't work with how the iteration var was being used: I needed to update the type manually.
- I had to remove a try/catch block from a coroutine function (this was apparently causing an internal compiler error in C# - weird).
- C# doesn't allow non constants as switch case values. I had a couple of switches that I needed to turn into if/else ladders.
- I had a class that employed a function dictionary of .js type 'Dictionary.<string, Function>. I had to rework that a little since C# delegates are strictly typed and this dictionary was being assigned functions with two different prototypes.
- Beyond the general need to fix-up (or pre-hint) all Function to delegate conversions, I had one case in particular that was a little more complicated due to the number of parameters… Although it wouldn't have been a limitation in a more modern version of .Net, Mono 2.0's older .Net implementation has a limit of five types for Func and four for Action types. A solution for this is to define a custom delegate type, e.g.

```
// Would like to have hinted like this:
// !CSTYPE! captureFunc: Action<Camera,Plane[],PropLocator[],PropMeshInfo,PropLocator.Flags>;
// But five params is too many. So instead we declare a new delegate type...
public delegate void PropCaptureDelegate(Camera camera, Plane[] frustumPlanes, PropLocator[] candidates, PropMeshInfo pmi, PropLocator.Flags captureFlag );

    public PropCaptureDelegate captureFunc;
```

Conclusions

Conversion of my project proved to be straightforward, and was (I think) very fast for such a complicated code-base.

Out of my **267** files, after fixing 'Fix .JS' issues and adding Type Hints to address Function conversion, **237** of those files (89%) compiled perfectly *without a single manual edit to the CSharpatron's .cs output*. None of the remaining files required more than a few quick fix-ups.

I do wish I could have done more to automate conversion of Functions-to-delegates - these were the cause of the vast majority of manual fix-up work that I had to do. Unfortunately I just can't see any viable path that's any better than 'type-hinting'… The main problem is simply that the Function type is so flexible and that some use-cases simply demand a re-think in order to transition into an environment supporting only statically-defined types.

I was especially relieved to see that not a single mapped object lost data during the conversion process, and that barring those two (not-obviously-conversion-related fixes), the game still ran at the end of it!

LIke I said earlier, I believe my project definitely sits at the 'complicated' end of the conversion spectrum. *For comparison I also did a test conversion of Unity's 'Penelope' example project and that converted with zero errors and was running in C# in less than 5 minutes!* I suspect many people would find their experience closer to that one.

Anyhow, thanks for reading - I hope CSharpatron is able to save you as much time as it would have saved me (had I not had to write it first!).

☺

A couple of take away thoughts:

- This conversion process revealed quite a high number of latent bugs in my game. I really have more confidence in the quality of my C# codebase than I had in the Unityscript I started with - not something I would have expected.
- Writing complex scripts in a loosely typed language is a very bad idea. I won't be sorry to leave Unityscript behind me!

Appendix D - Extras

Reflection Tools

To help with development of CSharpatron I implemented some simple tools that can dump out Reflection derived type information to the Unity log. I found them useful so figured others else might find them interesting or useful too…

To use these tools, open up the control center's 'Extras' callout. You will see:

- **Show reflected types.** Use this tool to display the names of types present within available assemblies. If you click 'Show' leaving the tex-box *empty* then you'll get a list of *all* types, grouped by assembly. …It's interesting to see the insane number of classes we have access to!
  Anything you type in the tex-box is treated as a filter tag, so if you type 'Random' then click 'Show' you'll see all types that include the text 'Random' within their name.

- **Show members of type.** If you know the name of a type that's of interest, you can look up all members of that type using this option. What you type is treated as a filter tag so if you lookup 'String' you'll see the members of all types containing 'String' in their name (48 classes for me). A more specific tag like 'System.String' shows fewer types (4). *If you want only an exact type the supply a full type name and hold down Alt as you click the 'Show' button.*
  NOTE: output for a given class includes all inherited fields. (It would be nice to actually list the inherited types and interfaces but that's not quite as simple as it sounds and I haven't had a need that justifies this so far!)

- **Show members with name.** If you remember the name of a method or field but can't remember which type contains it, you can use this one. The text you provide is a filter tag unless you hold down *Alt* as you click the 'Show' button, for example if I search for 'Substring' I see about 25 results spread across 14 types. Holding down *Alt,* this becomes 9 results across 8 types.
  NOTE: this tool takes a while to return a result; it has to churn through a *lot* of type information!

- **Extension methods.** Click the 'Show' button to see log output of every public extension method, sorted by 'extended class' (new in v1.1).

Each of these tools just writes results as a single Log string so copy/pasting into a file is very straightforward.

Change Log

v1.2

- Added support for conversion of Unityscript 'Properties'.
- Added retry mechanism to all file operations to avoid the potential for 'sharing violation' exceptions e.g. if VS Intellisense parsing is occurring during converter operation.
- Added support for 'same line' style properties. This is valid for certain types only: HideInInspector, System.NonSerializedAttribute, System.NonSerialized and SerializeField.
- Added a warning message any time that the user's project isn't in a compiling state.

v1.12

- Fix for converter freeze that could occur with a certain source formatting style.

v1.1

- Fix for error case when reading workspace file when running under Windows (causing 'Int32.Parse' crash cycle).
- Improved support for multi-var arrays declared in Unityscript (e.g. int[,])
- Support for mixed format multi-dimensional arrays declared in a C# assembly (e.g. int[,][])
- Fix for a freeze bug that could occur if source contained an abbreviated float constant of the form '1f'.
- Reworked handling for auto fix-up of assignment to value type return. New method works in all cases barring multi var assignment.
- Prevented an unhelpful type-cast to 'IEnumerable' that could be added in some circumstances.
- Fixed a case where a var declared as a Unityscript Array could fail to yield an error.
- Fixed possible confusing error message pending assignment of a Unityscript Array to an otherwise untyped var.
- Fixed potential parsing bug when processing initialized arrays (could have caused bad type-cast bracketing).
- Added a debug 'line-step' feature to help one user to track down a 'lock-up' bug.
- Added support for Javascript 'Number' type (converts to C# 'double')
- UnityEngine.Object.Instantiate return type now implicitly set to match first param (to match the similar magical upcast that Unityscript can apparently perform without the need to cast!)
- @CustomEditor is now respected and used to infer the type of the editor class var 'target'. This works wonders for conversion error counts in editor related files.
- Extension methods (from C# assemblies) are now parsed correctly when used in Unityscript.
- Added a 'DumpExtensionMethods' debug tool (under Extras).
- Fixed bogus 'read-only' error when script was writing to 'this.member'.
- Fixed possibility for negative numeric constants to be misinterpreted.
- Fixed bug that could cause broken stubs file generation.
- Improved detection of implicitly convertible types in binary ops to fix some spurious casts.
- 'Rebuild stubs file' is now blocked if there are outstanding fatal errors (to avoid generation of an incomplete file).
- Reworked intParse and floatParse to just insert type-casts as appropriate.
- Improved wording for a couple of common warnings.
- Improved handling for typeof() operator. If supplied a non-class name, CSharpatron now auto-converts to var.GetType()
- Added new 'About' callout in Control Center panel to make version and contact details a little more accessible.
- Added support for anonymous methods.
- Major optimization to conversion times, now approximately twice as fast as v1.03.
- Fixed parsing problem in 'for(var x:type in y)' style loops where 'type' included template brackets.
- Fixed fatal error parsing nested 'if' statements of the form 'if (x) if (y) {}'.
- Added ability to 'DestroyWorkspace' after 'Finalize'.

v1.03

- Fixed a problem where initializing database could encounter file exceptions when attempting to load all assemblies for parsing.

v1.02

- First published version