User Guide

Dynamic C#

Runtime C# scripting

Trivial Interactive

Version 1.0.5

Dynamic C# allows runtime loading of assemblies and C# scripts at runtime making it trivial to add modding support to you game by allowing your users to write C# scripts. In addition, Dynamic C# also allows strict security restrictions to be enforced as specified by the developer meaning that external code can run safely.

Limitations

- Cannot run under .net subset (Requires .Net 2.0).
- AOT platforms such as IOS are not supported.
- Scripts must be compiled before they can be executed.

Features

- Compile and run C# scripts at runtime.
- Fast Execution – once compiled, external scripts run as fast as game scripts.
- Allows for modding support to be added easily.
- Pre-load security checks mean that unsafe code can be identified and discarded.
- Support for loading assemblies and C# scripts.
- All scripts use custom namespaces to prevent clashing type names.
- Support for non-concrete interface using script proxies.
- Simple and clean API for accessing types and proxies.
- Cached member tables for quick reflection.
- Automatic construction of types using the appropriate method (AddComponent, CreateInstance, new)
- Comprehensive .chm documentation of the API for quick and easy reference.
- Fully commended C# source code included.

Support for PC, Mac, and Linux platforms. Dynamic C# may work without issue on other platforms however we will only offer support for the listed platforms.

Installation

In order to use the compiler functionality of Dynamic C# you will first need to install an additional compiler package which contains all the extra scripts and assemblies that are required. By default, these files are not included directly in the project because they depend upon the API compatibility level being set at '.Net 2.0' as opposed to '.Net 2.0 Subset'. This would result in compile time errors on import which would not make for a good first impression.

In order to keep the process as simple as easy as possible, we have included a convenient installer window which will take care of importing the required files. Alternatively if you prefer to understand what changes are need to be made to your project then you can install the compiler package manually.

1. **Automatic Install**

   First you will need to open the installer window by selecting the following menu item 'Tools -> Dynamic C# -> Installer'. Once you have done that you should see the following window:

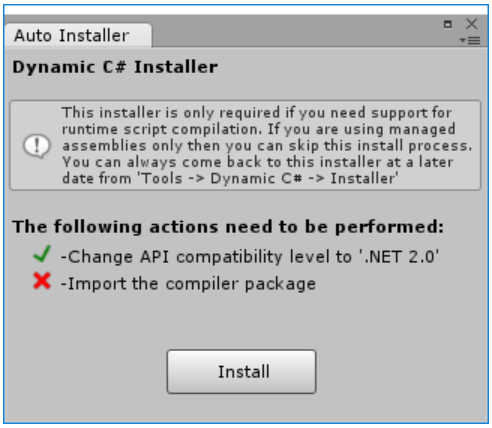


Figure 1

   Depending upon your project settings you may have to perform one or more actions in order to install the compiler module. Items that have not been completed are marked with a red 'X' and items that do not need to be installed are marked with a green 'tick'.

   The install process is as simple as clicking the 'Install' button and waiting for the install to complete which should only take a matter of seconds. After a small delay, you should see that all required actions have been performed and you now have all green ticks.

2. **Manual Install**

   A manual install is only recommended for more experienced Unity users as it involves changing project settings.

   The first thing you will need to do is open the player settings for your build platform which can be done by selecting the following menu item 'Edit -> Project Settings -> Player'. The inspector window should now display the current project settings. Scroll down until you find the 'Api Compatibility Level' option under the 'Optimization' header as shown in figure 2.

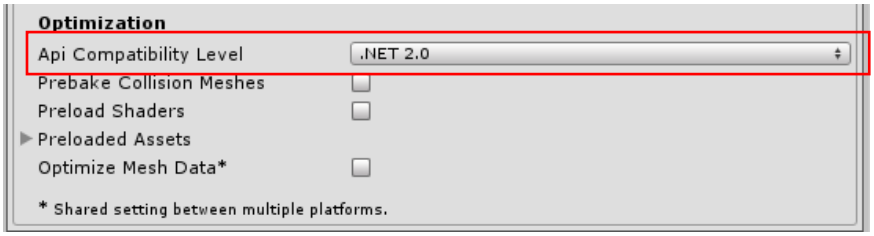


Figure 2

If the option is set as '.Net 2.0' then you do not need to change anything and can skip this step. If the value is set to '.Net 2.0 Subset' then you will need to change the value to '.Net 2.0'.

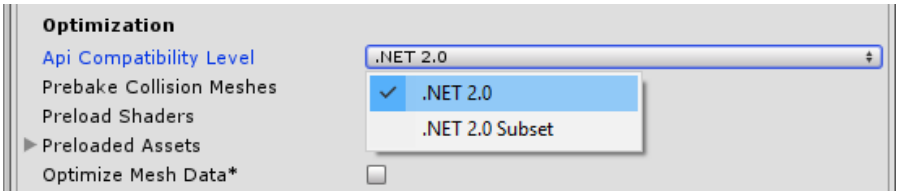


Figure 3

Once you have change the Api level you are ready to install the compiler module. The compiler module is distributed as a '.unitypackage' and can simply be imported into the project. Locate the package at 'DynamicCSharp/Resources/Editor/CompilerPackage.unitypackage' and double click the file to import into the current project.

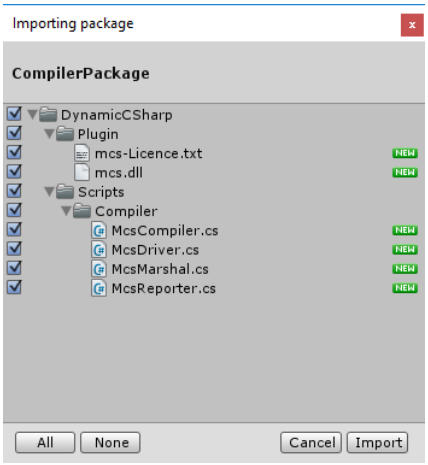


Figure 4

Now you can simply select 'import' and Unity will take care of the rest. The package will be extracted into the correct locations under the 'DynamicCSharp' root folder.

Quick Start

This section will cover the steps required to get up and running as quickly as possible. More detailed information is provided later in the document.

1. **(Optional) Install Compiler**

   If you want to be able to compile C# scripts at runtime then you will need to install the compiler module. The compiler module requires that the API compatibility level is set to .Net 2.0 instead of the default 'Subset'. For this reason the compiler is included separately and must be installed before use.

   To install the compiler, see the previous 'Installation' section.

   > **Note:** *Projects that cannot change from '.Net 2.0 Subset' can still use Dynamic C# however you will only be able to load pre-compiled managed assemblies as opposed to un-compiled C# scripts.*

2. **Create a Domain**

   The next thing you will need to do is create a Script Domain where all your external code will be loaded into. The following C# code shows how a domain can be created:

C# Code

```
1  using UnityEngine;
2  using DynamicCSharp;
3
4  public class Example : MonoBehaviour
5  {
6      // Our script domain reference
7      private ScriptDomain domain = null;
8
9      // Called by Unity
10     void Start()
11     {
12         // Should we initialize the compiler?
13         bool initCompiler = true;
14
15         // Create the domain
16         domain = ScriptDomain.CreateDomain("MyDomain",
initCompiler);
17     }
```

   The domain is created using the static 'CreateDomain' method. The second argument is particularly important here and will depend upon the last step. If you have installed the compiler then you can pass 'true' to the method otherwise you will need to pass 'false'. If the compiler has not been installed and you pass 'true' to the method then you will receive and exception about the missing compiler module.

3. Load External Code

Once you have a domain, you are now ready to load any external C# code or assemblies. There are a number of methods that you can use for loading assemblies or C# code. For a basic example, we have defined the C# code we want to load as a string named 'source'.

C# Code

```
using UnityEngine;
using DynamicCSharp;

public class Example : MonoBehaviour
{
    private ScriptDomain domain = null;

    // The C# source code we will load
    private string source =
        "using UnityEngine;" +
        "class Test : MonoBehaviour" +
        "{" +
        "    void SayHello()" +
        "    {" +
        "        Debug.Log(\"Hello World\");"    +
        "    }"                                  +
        "}";

    void Start()
    {
        // Create the domain - We are using C# code so we need the
compiler
        domain = ScriptDomain.CreateDomain("MyTestDomain", true);

        // Compile and load the source code
        ScriptType type =
domain.CompileAndLoadScriptSource(source);
    }
```

The main load method here is the call to 'CompileAndLoadScriptSource'. This method will invoke the compiler which generates a managed assembly. The main type for that assembly (in this case 'Test') is then automatically selected and returned as a Script Type.

For more information on loading C# code and assemblies take a look at 'LoadingAssemblies' and 'LoadingScripts' section.

4. Create an instance of the type

Once you have a Script Type loaded, the next thing you will want to do is create an instance of that type. There are a number of methods that allow you to do this but for this example we will use the basic 'CreateInstance' method of Script Type.

Previous code has been omitted to keep the examples short

C# Code

```
1  using UnityEngine;
2  using DynamicCSharp;
3
4  public class Example : MonoBehaviour
5  {
6      void Start()
7      {
8          // Create the domain - We are using C# code so we need the
9  compiler
10         domain = ScriptDomain.CreateDomain("MyTestDomain", true);
11
12         // Compile and load the source code
13         ScriptType type =
14 domain.CompileAndLoadScriptSource(source);
15
16         // We need to pass a game object because 'Test' inherits
17 from MonoBehaviour
18         ScriptProxy proxy = type.CreateInstance(gameObject);
19     }
}
```

At this point you now have an external C# script attached to a game object as a component. All of the expected mono behaviour events will be called such as 'Start' and 'Update' and you can interact with the Unity API from the external script.

5. Call a Method

This next step is a simple example of how you are able to call custom methods to provide a similar event system as a mono behaviour. For example, let's say we want to call the 'SayHello' method when a script is constructed.

C# Code

```
using UnityEngine;
using DynamicCSharp;

public class Example : MonoBehaviour
{
    void Start()
    {
        // Create the domain - We are using C# code so we need the
compiler
        domain = ScriptDomain.CreateDomain("MyTestDomain", true);

        // Compile and load the source code
        ScriptType type =
domain.CompileAndLoadScriptSource(source);

        // We need to pass a game object because 'Test' inherits
from MonoBehaviour
        ScriptProxy proxy = type.CreateInstance(gameObject);

        // Call the 'SayHello' method
        proxy.Call("SayHello");
    }
}
```

We use the method 'Call' in this example but there is another method called ' SafeCall' which might be more appropriate in this scenario. The 'SafeCall' method will catch any exceptions throw by the target method which could prove useful when dealing with external code.

For more information about cross communication between external code and the application, take a look at the 'Interface Approaches' section.

6. **Congratulations, you have just compiled and loaded a basic C# script at runtime. Now you can go on to do awesome things!**

Concepts

Script Domain

Dynamic C# uses the concept of Script Domains which you can think of as a container for any externally loaded code. You must create a Script Domain before you are able to load any external code and all subsequent loading will be handled by that domain.

If you are particularly adept in C# then you may be familiar with 'AppDomains'. It is worth noting that Dynamic C# does not create a separate AppDomain for external scripts but instead loads all code into the current domain. A Script Domain simply acts as a filter allowing only external code to be visible to the user.

As well as acting as a container, a Script Domain is also responsible for the loading of C# code or assemblies, as well as security validation to ensure that any loaded code does not make use of potentially dangerous assemblies or namespaces. For example, by default access to 'System.IO' is disallowed.

Script Assembly

A Script Assembly is a wrapper class for a managed assembly and includes many useful methods for finding Script types that meet a certain criteria. For example, finding types that inherit from UnityEngine.Object.

In order to obtain a reference to a Script Assembly you will need to use one of the 'LoadAssembly' methods of the Script Domain class. Depending upon settings, the Script Domain may also validate the code before loading to ensure that there are no illegal referenced assemblies or namespaces.

Script Assemblies also expose a property called 'MainType' which is particularly useful for external code that defines only a single class. For assemblies that contain more than one types, the MainType will be the first defined type in that assembly.

If you need more control of the assembly then you can use the 'RawAssembly' property to access the 'System.Reflection.Assembly' that the Script Assembly is managing.

> **Note:** *Any assemblies or scripts that are loaded into a Script Domain at runtime will remain until the application ends. Due to the limitations of managed runtime, any loaded assemblies cannot be unloaded.*

Script Type

A Script Type acts as a wrapper class for 'System.Type' and has a number of unity specific properties and methods that make it easier to manage external code. For example, you can use the property called 'IsMonoBehaviour' to determine whether a type inherits from MonoBehaviour.

The main advantage of the Script Type class is that it provides a number of methods for type specific construction meaning that the type will always be created using the correct method.

- For types inheriting from MonoBehaviour, the Script Type will require a GameObject to be passed and will use the 'AddComponent' method to create an instance of the type.

- For types inheriting from ScriptableObject, the Script Type will use the ‘CreateInstance’ method to create an instance of the type.
- For normal C# types, the Script Type will make use of the appropriate construction based upon the arguments supplied (if any).

This abstraction makes it far simpler to create a generic loading system for external code.

Script Proxy

A Script Proxy is used to represent an instance of a Script Type that has been created using one of the ‘CreateInstance’ methods. Script Proxies are a generic wrapper and can wrap Unity instances such as MonoBehaviours components as well as normal C# instances.

A Script Proxy implements the IDisposable interface which handles the destruction of the instance automatically based upon its type.

- For instances inheriting from MonoBehaviour, the Script Proxy will call ‘Destroy’ on the instance to remove the component
- For instance inheriting from ScriptableObject, the Script Proxy will call ‘Destroy’ on the instance to destroy the data.
- For normal C# instances, the script proxy will release all references to the wrapped object allowing the garbage collector to reclaim the memory.

Note: Y*ou are not required to call ‘Dispose’ on the Script Proxy. It is simply included to provide a generic, type independent destruction method.*

Global Settings

Dynamic C# uses a number of global settings that can be modified within the Unity editor by opening the following menu item 'Tools -> Dynamic C# -> Settings'. You should see the following window:

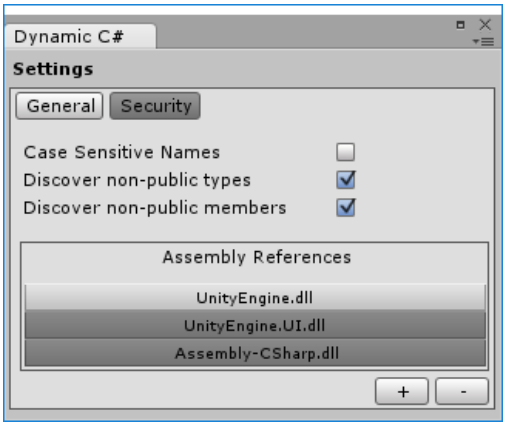


Figure 5

There are two main tabs in the settings window to categorize the settings. This following section will cover all the options in each tab and what they do.

General

The general settings tab contains a number of generic settings that modify the loading and runtime behaviour of Dynamic C#.

- **Case Sensitive Names:** When enabled, all method that use type names or member names will search using case sensitive string matching as opposed to non-case sensitive matching when the option is disabled.
- **Discover Non-Public Types:** When enabled, Dynamic C# will include private, protected and internal types in its searches as opposed to just public types. It is recommended that you keep this options enabled because you will typically want to access all types within an external assembly regardless of their exposure.
- **Discover Non-Public Members:** Just like with types, members can also be non-public and this option will determine whether private, protected and internal members are accessible as opposed to just public when the option is disabled. It is recommended that this options also remains enabled.
- **Assembly References:** The assembly references collection allows you to specify what assemblies are referenced by runtime compiled code. Any code that is run through the runtime compiler will reference all of these assemblies.

> **Note:** *It is important that any assemblies specified in this collection are guaranteed to be loaded at runtime otherwise all compile requests will fail. One way to do this is to include the assembly in your Unity project as Unity will then ensure that it is loaded at runtime. The other alternative is to load the assembly manually at runtime by calling Assembly.Load. This will need to be done before compiling any code with Dynamic C#.*

Add Reference: In order to add an assembly reference to the collection you will need to click the '+' icon at the bottom of the list which will open an input dialog as shown below:

Figure 6

Enter the name of the referenced assembly into the field making sure to include the '.dll' file extension and then select 'OK'. You should now see your reference name appear in the collection.

Remove Reference: In order to remove a reference from the collection you can simply select the reference name by clicking on it, and then hit the '-' icon at the bottom of the list which will cause the selected item to be removed.

Security

The security tab contains all settings related to code security and validation and is where you can setup restrictions to ensure that external code does not access undesirable API's such as System.IO.

- **Security Check Code:** When enabled, Dynamic C# will attempt to validate all code before it is loaded into the script domain. If the security checks fail then the code will not be loaded. It is highly recommended that this option remains enabled as you may not have any control over the external code being loaded.
- **Assembly Reference Restrictions:** The assembly reference restrictions list allows you to add any number of reference restrictions which will be checked when external code is loaded. A reference restriction is simply the name of an assembly that must not be referenced by loaded code. A good candidate for a restriction would be 'UnityEditor.dll' as it is not available at runtime.

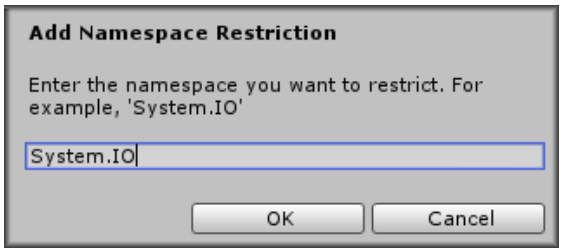
Add Reference Restriction: In order to add an assembly reference restriction to the collection you will need to click the '+' icon at the bottom of the list which will open an input dialog as shown below:

Figure 7

Enter the name of the restricted assembly into the field making sure to include the '.dll' file extension and then select 'OK'. You should now see your reference name appear in the collection and it will be included in the runtime security checks.
Remove Reference Restriction: In order to remove a reference from the collection you can simply select the reference name by clicking on it, and then hit the '-' icon at the bottom of the list which will cause the selected item to be removed.

- **Assembly Namespace Restrictions:** The assembly namespace restrictions list allows you to specify individual namespaces that should not be accessed by external code. Simply specify the full namespace name and any loaded code that references that namespace will fail verification causing the code to be discarded. A good candidate for namespace restriction would be 'System.IO' to prevent access to the file system. It is also highly recommended that 'System.Reflection' is added to the restrictions because it could potentially be used as a workaround for already restricted namespaces.

  Add Namespace Restriction: In order to add a namespace restriction to the collection you will need to click the '+' icon at the bottom of the list which will open an input dialog as shown below:



- *Figure 8*

  Enter the name of the restricted namespace into the field making sure to spell it correctly and then select 'OK'. You should now see your namespace appear in the collection and it will be included in the runtime security checks.
  Remove Namespace Restriction: In order to remove a namespace from the collection you can simply select the reference name by clicking on it, and then hit the '-' icon at the bottom of the list which will cause the selected item to be removed.

Loading Assemblies

It is possible to load external code as compiled assemblies using Dynamic C#. There are a number of load methods that are provided that allow you to achieve this, all of which will perform additional security verification checks if enabled. All of these methods are called directly on an instance of a Script Domain which must be created in order to use Dynamic C#.

The assembly loading methods are as follows:

- **LoadAssembly(string):** Attempts to load a managed assembly from the specified file path.
- **LoadAssembly(AssemblyName):** Attempts to load a managed assembly with the specified assembly name. Note that due to limitations, this method cannot security check code so it is recommended to use another 'Load' method were possible.
- **LoadAssembly(byte[]):** Attempts to load a managed assembly from its raw byte data. This is useful if you already have the assembly in memory or are downloading it from a remote source or similar.
- **LoadAssemblyFromResources(string):** This is a Unity specific load method any will attempt to load an assembly from the specified TextAsset in the resources folder.

All of these load methods return a Script Assembly which can be used to access Script Types using a number of 'Find' methods.

For more information on loading methods, take a look at the separate API documentation included with the package.

Loading Scripts

One of the main features of Dynamic C# is that it allows C# source code to be compiled at runtime meaning that you can easily add features such as modding support without relying on slower interpreted languages such as Lua. Since the code is compiled before use, you will have the same performance as if the script was included in the game in the first place.

There are a number of useful methods that you can use. The following methods can compile and load C# source code at runtime:

- **CompileAndLoadScriptFile(string):** Attempts to compile the C# source code in the specified file and load the resulting Script Type into the Script Domain.
- **CompileAndLoadScriptFiles(params string[]):** Attempts to compile all of the specified C# source files as a batch and loads the resulting assembly into the Script Domain.
- **CompileAndLoadScriptSource(string):** Attempts to compile the C# source code specified in the string argument and load the resulting Script Type into the Script Domain.
- **CompileAndLoadScriptSources(params string[]):** Attempts to compile all of the specified C# source code and loads the resulting Script Assembly into the Script Domain.

For more information on the compile methods, take a look at the separate API documentation included with the package.

Interface Approaches

Once you have loaded an assembly or script into a Script Domain and created a Script Proxy instance, the next step you will likely want to take is to communicate with the types in some way

There are 2 main types of communication that you can use to interact with external scripts and assemblies:

Generic Communication

Generic communication is considered as a non-concrete type of communication meaning that the type you want to communicate with is not known at compile time. This poses a few issues because you are unable to simply call a method on an unknown type. Fortunately Dynamic C# includes a basic generic communication system that works using reflection and allows any class member to be accessed without knowing the runtime type.

A Script Proxy is used to communicate with external code using string identifiers to access members. The following example shows how to create your own magic method type events:

C# Code

```
1  using UnityEngine;
2  using DynamicCSharp;
3
4  public class Example : MonoBehaviour
5  {
6      // This example assumes that 'proxy' is created before hand
7      ScriptProxy proxy;
8
9      void Start()
10     {
11         // Call the method 'OnScriptStart'
12         proxy.SafeCall("OnScriptStart");
13     }
14
15     void Update()
16     {
17         // Call the method 'OnScriptUpdate'
18         proxy.SafeCall("OnScriptUpdate");
19     }
}
```

The above code shows how a method with the specified name can be called at runtime using the 'SafeCall' method. The following methods can be used to call methods on external scripts:

- Call: The call method will attempt to call a method with the specified name and upon error will throw an exception. Any exceptions thrown as a result of invoking the target method will also go unhandled and passed up the call stack so it is recommended that you use 'SafeCall' unless you want to implement your own error handling.
- SafeCall: The SafeCall method is essentially a wrapper for the 'Call' method and handles any exceptions that it throws. If the target method is not found then this method will fail silently.

When calling a method is it also very useful to be able to pass arguments to that method. Dynamic C# allows any number of arguments to be passed provided that the passed types are known to both

the game and the external script beforehand. A good candidate for this Unity types such as Vector3 as compiled scripts are set to reference the UnityEngine.dll by default. The target method must also accept the same argument list otherwise calling the method will fail.

Dynamic C# also includes a way of accessing fields and properties of external scripts provided that their name is known beforehand. Again communication is achieved via the proxy but instead of calling a method you use either the 'Fields' or 'Properties' property of the proxy.

1. **Fields**

   Fields can have their values read from or written to so long as the assigned type matches the field type. If the types do not match then a type mismatch exception may be thrown.

   Unlike methods, there is no safe alternative for accessing fields using this method. If you want to be safe when accessing fields then you should catch any exceptions thrown.

   The following code shows how a field called 'testField' can be modified:

C# Code

```
1  using UnityEngine;
2  using DynamicCSharp;
3
4  public class Example : MonoBehaviour
5  {
6      // This example assumes that 'proxy' is created before hand
7      ScriptProxy proxy;
8
9      void Start()
10     {
11         // This example assumes that 'testField' is an int
12
13         // Read the value of the field
14         int old = proxy.Fields["testField"];
15
16         // Print to console
17         Debug.Log(old);
18
19         // Set the value of the field
20         proxy.Fields["testField"] = 123;
21     }
}
```

2. **Properties**

   Properties are a little different to fields because they are not required to implement a get and a set method. This means that certain properties cannot be written to or read from which means you have to be all the more careful.

   If you attempt to read from or writ to a property that does not support it, then a target exception may be thrown.

   As with fields, there is no safe alternative for accessing properties. If you want to be safe when accessing fields then you should catch any exceptions thrown.

The following code shows how a property called ‘testProperty’ can be accessed. The method is very similar with fields and properties.

C# Code

```
using UnityEngine;
using DynamicCSharp;

public class Example : MonoBehaviour
{
    // This example assumes that 'proxy' is created before hand
    ScriptProxy proxy;

    void Start()
    {
        // This example assumes that 'testProperty' is an int

        // Read the value of the property
        int old = proxy.Properties["testProperty"];

        // Print to console
        Debug.Log(old);

        // Set the value of the property
        proxy.Properties["testProperty"] = 456;
    }
}
```

Interface Communication

The second communication method is fairly more advanced than the previous method however it will allow for concrete type communication as opposed to loose string based communication. It should also offer improved runtime performance since it does not rely on reflection to access type members.

The implementation involves creating a shared interface containing any number of base classes or C# interfaces that all external code must inherit from. The best way to do this would be to create a separate managed assembly containing these shared base types and make it available to the external code, however it is possible (although not advised) to create these base classes directly within your Unity project and allow the external code to reference the 'Assembly-CSharp' assembly containing all of your game code. The reason this method is not recommended is that it might allow external code to use your game scripts as well as the base classes which may be undesirable.

Providing a guide for creating a separate interface assembly is beyond the scope of this documentation however there are a number of very useful Unity specific tutorial online to cover this.

Once you have defined your interface then you are able to load and call the external code as if it were part of your game.

As an example, we will use the following C# interface to show how the process would work:

C# Code

```
1  using UnityEngine;
2
3  public interface IExampleBase
4  {
5      void SayHello();
6
7      void SayGoodbye();
8  }
```

As you can see the interface contains 2 methods which must be implemented and for now we will assume that this interface is defined in an assembly called 'ExampleInterface.dll'.

In order for runtime compiled code to access this assembly we will need to add it to the assembly references in the global settings. See the 'Add Assembly Reference' section in the 'Global Settings' section for information on how to do this.

Once the assembly has been added to the references then we are ready to compile and load our external code. We will now require our external code to implement this interface in order for us to load it into the game. If it does not then we will treat it as if there are no valid defined types. Our external code is simply as follows:

C# Code

```
using UnityEngine;

public class Test: IExampleBase
{
    void SayHello()
    {
        Debug.Log("Hello");
    }

    void SayGoodbye()
    {
        Debug.Log("Goodbye");
    }
}
```

As you can see, the example code is very basic and will simply print to the Unity console when one of its two methods are called.

The next step it to compile and load the code into a Script Assembly using one of the many load method of our Script Domain. As in the previous section, we will store the C# code from the above example directly in a string in order to keep the example simple but you can load the source code from any location you need. The following code will compile and load the source into a Script Domain:

C# Code

```
using UnityEngine;

class Example : MonoBehaviour
{
    private string source =
        "using UnityEngine;"                +
        "public class Test : IExampleBase"  +
        "{"                                 +
        "    void SayHello()"               +
        "    {"                             +
        "        Debug.Log("Hello");"       +
        "    }"                             +
        "    void SayGoodbye()"             +
        "    {"                             +
        "        Debug.Log("Goodbye");"     +
        "    }"                             +
        "}";

    void Start()
    {
        // Create the domain
        ScriptDomain domain =
ScriptDomain.CreateDomain("TestDomain", true);

        // Compile the source code
        ScriptAssembly assembly =
domain.CompileAndLoadScriptSources(source);
    }
}
```

At this point, we now have our source code compiled and loaded into our Script Doman and the next step is to make use of our interface and find all types that inherit from it. The Script Assembly class contains a number of useful methods for accessing Script Types that inherit from other types. The following code shows how we can access all types that inherit from our 'IExampleBase' interface that we created earlier.

Previous code has been omitted to keep the examples short

C# Code

```
using UnityEngine;

class Example : MonoBehaviour
{
    void Start()
    {
        // Compile the source code
        ScriptAssembly assembly =
domain.CompileAndLoadScriptSources(source);

        // Find all types that implement the 'IExampleBase'
interface
        ScriptType[] types =
assembly.FindAllSubtypesOf<IExampleBase>();
    }
}
```

As you can see, we now have an array of Script Types, all of which implement our 'IExampleBase' interface. That means we can be sure that all of these types implement both methods defined in the interface so the next thig we can do is call those methods on each type.

Previous code has been omitted to keep the examples short

C# Code

```
using UnityEngine;

class Example : MonoBehaviour
{
    void Start()
    {
        // Find all types that implement the 'IExampleBase'
interface
        ScriptType[] types =
assembly.FindAllSubtypesOf<IExampleBase>();

        foreach(ScriptType type in types)
        {
            // Create a raw instance of our type
            IExampleBase instance =
type.CreateRawInstance<IExampleBase>();

            // Call its methods as you would expect
            instance.SayHello();
            instance.SayGoodbye();
        }
    }
}
```

As you would expect, before we can you the type we need to create an instance of it which is done using the ‘CreateRawInstance’ of the Script Type. The main different this method has when compared with the ‘CreateInstance’ methods is that the concrete type is returned as opposed to a managing Script Proxy. This means that we can access the result directly as our ‘IExampleBase’ interface and the conversion will work fine. After that we now have an instance of the ‘Test’ class defined earlier stored as the ‘IExampleBase’ interface meaning that we can now call the methods directly.

Although the interface approach requires more setup to get working, it is worth the effort as you gain type safety as well as extra performance when compared with the proxy communication method. This is due to the fact that proxies rely on reflection under the hood in order to call methods and access members which will always be slower than simply calling a method.

Report a Bug

At Trivial Interactive we test our assets thoroughly to ensure that they are fit for purpose and ready for use in games but it is often inevitable that a bug may sneak into a release version and only expose its self under a strict set of conditions.

If you feel you have exposed a bug within the asset and want to get it fixed then please let us know and we will do our best to resolve it. We would ask that you describe the scenario in which the bug occurs along with instructions on how to reproduce the bug so that we have the best possible chance of resolving the issue and releasing a patch update.

http://trivialinteractive.co.uk/bug-report/

Request a feature

Dynamic C# was designed as a complete runtime coding solution, however if you feel that it should contain a feature that is not currently incorporated then you can request to have it added into the next release. If there is enough demand for a specific feature then we will do our best to add it into a future version.

http://trivialinteractive.co.uk/feature-request/

Contact Us

Feel free to contact us if you are having trouble with the asset and need assistance. Contact can either be made by the contact options on the asset store or buy the link below.

Please attempt to describe the problem as best you can so we can fully understand the issue you are facing and help you come to a resolution. Help us to help you :-)

http://trivialinteractive.co.uk/contact-us/