Eezeescript Host API Reference

Using Eezeescript

This section provides a quick introduction for integrating Eezeescript in a .NET solution and using the API.

Integrating Eezeescript in a .NET solution

The Eezeescript host API can be integrated into an existing .NET solution by adding a reference to the `Eezeescript.dll` assembly. All API classes are defined within an "Eezeescript" namespace that must be either specified with the `using` clause or prefixed to the API classes.

Preparing the scripting environment

The scripting system is initialised by creating one or more instances of the **ScriptManager** class. Each instance represents a scripting environment where scripts can be loaded and executed and also provides a "global" variable scope that the scripts can use to share data.

```
ScriptManager scriptManager = new ScriptManager();
```

Loading Scripts

Once a script manager is available, scripts can be loaded by creating instances of the **Script** class. The script object's constructor requires a reference to the script manager and a name to identify the script. By default, this name corresponds to a disk filename.

```
Script script = new Script(scriptManager, "NPC_Wizard.ezs");
```

Preparing scripts for execution

A script object represents only the programming instructions contained within and not it's execution state. To execute the script, an instance of the **ScriptContext** class must be created. The class's constructor requires a reference to the script to be executed or to one of the script's named blocks if any other defined. A script reference implies that the main code block will be executed. If a named block is specified, only the code within the block and other blocks called within it will be executed. The script context provides execution control and

access to the execution state in terms of the variables defined during execution, the next statement to be executed, and so on. The ScriptContext class represents a running instance of a script and hence, multiple instances of the same script object can be executed within the same script manager by creating multiple script contexts referencing the same script.

```
// create a context for the script's main code block
ScriptContext scriptContext = new ScriptContext(script);

// also creates a context for the script's main block
ScriptContext scriptContext = new ScriptContext(script.MainBlock);

// create a context for one of the script's named blocks
ScriptBlock scriptBlock = script.Blocks["WanderAround"];
ScriptContext scriptContext = new ScriptContext(scriptBlock);
```

Executing Scripts

The script context object allows execution of the referenced script via the three variants of its `Execute` method, allowing execution of scripts for an indefinite amount of time, for a given time interval or up to a maximum number of executed statements.

The first method variant allows the referenced script block to execute indefinitely or until the end of the block is reached. If the script contains an infinite loop, this method will block indefinitely, unless an interrupt is generated. The `Execute` method returns the total number of statements executed since its invocation.

```
// execute indefinitely, or until termination, or until
// a script interrupt is generated
scriptContext.Execute();
```

The second variant of the `Execute` method allows the script context to execute up to a given maximum number of statements. The script context may break out of execution before the maximum is reached if there are no more statements to process or an interrupt is generated.

```
// execute up to a maximumum of 10 statements
scriptContext.Execute(10);
```

The third variant of the `Execute` method accepts a **TimeSpan** defining the maximum time interval allowed for script execution. The method may break out of execution earlier than the given interval if there are no more statements to process or an interrupt is generated. Given a script with a good balance of different statements, a possible use of this method is to determine the speed of the scripting system on the target environment in terms of statements executed per second.

```
// execute for up to 10 milliseconds
TimeSpan tsInterval = new TimeSpan(0, 0, 0, 0, 10);
scriptContext.Execute(tsInterval);
```

The second and third variants of Execute may be used to implement a virtual multi-threaded scripting environment. Global variables may be used as semaphores to synchronise concurrently running scripts.

The RPG cut-scene demo included with the EezeeScript library illustrates this concept. Each character in the demo is controlled by a separate script. Global boolean variables are used to allow one character to signal another character to perform an action. It should be noted that for this particular example, a cut-scene might be easier to implement using a single unified script with access to all relevant characters. However, the point of the demo is to demonstrate as many of the features as possible, including virtual multi-threading, inter-script communication and integration with the host application.

Figure 1 RPG Cut-Scene Demo

Interrupting and resetting scripts

A script context will normally execute its referenced script block indefinitely, for a given time interval, until a given maximum number of statements are executed or until there are no more statements to process.

In some cases it is desirable to break execution prematurely, such as to return control to the host when specific statements are executed, or because a script is too computationally intensive to execute in one go.

Eezeescript provides two ways for generating script interrupts:

- using the YIELD script instruction to explicitly break execution at a specific point in the script, or
- enabling the script context object's **InterruptOnCustomCommand** property to generate an interrupt automatically whenever a custom command is executed

The RPG cut-scene demo uses both approaches to allow each character script to be written as if it is meant to run in isolation from other scripts. Control is returned to the associated character object whenever a custom command is processed. This in turn allows the character object to queue and process the corresponding actions. While the actions are in progress, the script context is kept in a suspended state. Once the actions are complete, execution of the script is resumed. The scripts also implement "busy-waiting" synchronisation by setting global boolean variables and using `WHILE – ENDWHILE` looping statements. A YIELD instruction in the loop prevents indefinite script lockup.

NPC 1 Script

```
// initialise signal
SET g_OnTheWay TO TRUE

// walk to meeting point
NPC_MOVE 400 0

// signal arrival to NPC 2
SET g_OnTheWay TO FALSE
```

NPC 2 Script

```
// wait for arrival of NPC1
WHILE g_OnTheWay
    YIELD
ENDWHILE



// talk to NPC 1
NPC_SAY "Here you are!" 100
```

An executing script may be reset via the script context's `Reset` method. The net effect of invoking this method is that the local variables defined in the context are lost, the execution frame stack is cleared and the statement pointer is reset to point to the first statement in the script block referenced by the script context. Global variables are not affected and persist after a script context is reset.

Custom Script Loaders

To allow for loading of scripts from other sources, such as an archive file, network or database, a custom script loader class can be developed and bound to the script manager to be used for loading the script. The script loader class may be any class that implements the **ScriptLoader** interface. The loader is used to retrieve the script specified in the Script class constructor and also any additional include scripts defined within the original script.

```
// custom script loader class
public class MyScriptLoader
    : ScriptLoader
{
    public List<String> LoadScript(String strResourceName)
    {
        // loader implementation here...
    }
}
:
:
// in initialisation code...
ScriptLoader scriptLoader = new MyScriptLoader();
scriptManager.Loader = scriptLoader;
```

Accessing the local and global variable scope

One approach for allowing a script to communicate with or control the host application entails the latter polling the local variables of the associated script context and global variables of the associated script manager by respectively querying the **LocalVariables** and **GlobalVariables** properties of script context object.

```
// get value of local variable
int iScore = (int) scriptContext.LocalVariables["PlayerScore"];
```

```
// get value of global variable
bool bNewQuest = (bool)
    scriptContext.GlobalVariables["g_WizardQuestAvailable"];
```

Custom command extensions

A more powerful alternative to allow a script to interface with the host application is to register custom commands with the script manager and assign a script handler to a script context. The script handler in turn provides an implementation for the custom commands.

Custom commands are first defined by creating an instance of the **CommandPrototype** class to define the command's name and parameters.

```
// define command to move player
CommandPrototype commandPrototype
    = new CommandPrototype("Player_Move");
// add int parameter for x offset
commandPrototype.AddParameterType(typeof(int));
// add int parameter for y offset
commandPrototype.AddParameterType(typeof(int));
```

Once the command prototype is defined, it can be registered with the script manager. The command prototype ensures that a corresponding custom command is recognised during

runtime and that the parameters passed alongside correspond in number, type and order of those defined in the prototype.

```
// register new custom command
scriptManager.RegisterCommand(commandPrototype);
```

Implementing custom commands

Unless a script handler is bound to a script context, custom commands are only validated against their corresponding prototype when processed. Any class that implements the **ScriptHandler** interface may be used to handle custom script commands. The owner of the script context is a suitable choice for the handler. The handler interface requires the implementer class to provide methods to be executed whenever a custom command is processed, a local or global variable is changed, or an interrupt is generated.

```
public class Player
    // player class is its own script handler
    : ScriptHandler
{
    // called by script context when custom command is processed
    public void OnScriptCommand(ScriptContext scriptContext,
        String strCommand, List<object> listParameters)
    {
        // identify command by name
        if (strCommand == "Player_Move")
        {
            // read X and Y movement offsets
            m_iMoveX = (int)listParameters[0];
            m_iMoveY = (int)listParameters[1];
        }

        // other command implementations go here...
    }

    // called by script context when local/global var set or changed
    public void OnScriptVariableUpdate(ScriptContext scriptContext,
        String strIdentifier, object objectValue)
    {
        // do something, e.g. display for debug purposes
    }

    // called by script context when on interrupt
    public void OnScriptInterrupt(ScriptContext scriptContext)
    {
        // do something, e.g. update host state
    }

}
```

The script handler object is bound to the respective context via the **Handler** property.

```
// script passed via constructor
public Player(..., Script script, ...)
{
    :
    // prepare context
    m_scriptContext = new ScriptContext(script);

    // bind self as a script handler
    m_scriptContext.Handler = this;
    :
}
```

The ability to define a different handler for every script context allows a custom command to have different implementations and / or contexts depending on the script contexts to which respective script handlers are bound. For example, the implementation for a “Move x y” command within a game might affect the movement of a player character, a non-player character or a projectile.

Error Handling

API usage errors, compilation errors and script runtime errors are handled by throwing a **ScriptException**. Each exception may contain inner exceptions at an arbitrary number of levels. For compiler and runtime errors, the exception also provides a **Statement** property identifying the statement where the compilation or runtime error occurred.