Mojo#

Contenido

Introducción

Sintaxis

- **Literales:**
  - **Numéricos**. Enteros y coma flotante. Se utiliza el caracter '.' para indicar la coma flotante.
  - **Booleanos**. true o false.
  - **Texto**. Cualquier texto entre comillas. ("")
- **Operadores** (Siempre que se use una expresión, debe estar entre paréntesis)
  - **Unarios**
    - **Formato:** operador [literal / (expresión)]
    - **not** Niega la expresión que le sigue.
  - **Binarios**
    - **Formato:** [literal / (expresión)] operador [literal / (expresión)]
    - **Comparadores:**
      - **==** (Igualdad)
      - **>=** (Mayor o igual)
      - **<=** (Menor o igual)
      - **>** (Mayor)
      - **<** (Menor)
    - **Matemáticos**
      - **+**
      - **-**
      - *****
      - **/**
    - **Lógicos**
      - **and** (and lógico)
      - **or** (or lógico)
- **Variables**
  - **Definición**
    - **Formato:** var [nombredevariable]. (¡¡Ojo con el punto!!)
  - **Acceso**
    - **Formato:** [nombredevariable].get
  - **Asignación**
    - **Formato:** [nombredevariable] = [literal / (expresión)]
- **Control de Flujo**
  - **If:** Salto condicional
    - **Formato:**
    - **if (**literal / expresión**)** Cualquier número de expresiones **endif**
- **Métodos Definidos**
  - Permite ejecutar métodos definidos en unas clases especiales.
  - **Formato:** NombredeClase**.**NombreDeMetodo**(**parametros**)**

Definir Métodos

El sistema permite extender el lenguaje mediante la creación de nuevos métodos. Estos métodos serán los que doten al lenguaje de mayor expresividad y potencia.

Por ejemplo: Queremos añadir un sistema de log, que además de imprimir en consola, escriba en un fichero que podamos utilizar después para depurar. Sin embargo, en nuestro lenguaje, que queremos mantener simple, queremos que el usuario lo utilice mediante la llamada **Logger.Log("Mensaje")**.

Para definir este tipo de métodos, debemos crear una nueva clase, que herede de la clase **TM.Machine.Class**. En esta clase, podremos escribir todo el código que queramos con total normalidad, accediendo a todas las caracterísitcas de C# y Unity. Al heredar de la máquina virtual, tendremos a nuestra disposición acceso a la propia máquina, así como a la pila.

Si queremos que alguno de nuestros métodos pueda ser utilizado en el editor de Mojo#, deberemos marcarlo con el atributo **[ClassMethod]**. Este atributo permite indicar el **valor devuelto** por la función y un texto de ayuda. La función, en **código de C# no puede recibir ni devolver ningún valor.** El intercambio de parámetros debe hacerse a través de la pila. Para ello, se utilizarán los métodos **Pop()** para obtener los parámetros, y el método **Push()** para devolver un valor.

```
namespace TM.Machine
{
    public class Logger : Class
    {
        public Logger(VirtualMachine vm, Stack stack) : base(vm, stack){}

        [ClassMethod("void", "Log String to Console")]
        public void Log()
        {
            Debug.Log(Stack.PopString());
        }
    }
}
```

En el ejemplo de la imagen, se definen la clase Logger y el método Log. Este método toma el último valor de la pila, transformado a string, y lo imprime a través de la consola de Unity. Este método será accesible en Mojo# como **Logger.Log(**[Expresión a imprimir]**).**

SmartObjects

Los SmartObjects son objetos capaces de almacenar y ejecutar código Mojo#. Por ejemplo, si queremos que un Trigger ejecute código ante algún evento, necesitaremos crear una nueva clase, SmartTrigger, que herede de **SmartObject**. Dentro de esta clase, añadiremos los métodos que queramos que sean "programables". Para hacerlos accesibles al sistema, hay que marcarlos con el atributo **[SmartMethod].** En el constructor se puede añadir una ayuda para facilitar su uso. Finalmente, en su interior debemos hacer una llamada a **Execute(**"NombreDelMetodo**")**.

```
[SmartMethod("Method Executed On Trigger Enter")]
public void OnHit()
{
    Execute("OnHit");
}
```

Finalmente, debemos escribir un disparador. El disparador será el encargado de lanzar el método que hemos definido. Por ejemplo, el evento **OnHit()** queremos que se dispare cuando se produzca el **OnTriggerEnter(Collider other)** de Unity. Simplemente haremos la llamada.

```
public class SmartTrigger : SmartObject
{
    [SmartMethod("Method Executed On Trigger Enter")]
    public void OnHit()
    {
        Execute("OnHit");
    }

    private void OnTriggerEnter(Collider other)
    {
        OnHit();
    }
```

Anexo I: Class Methods

• **Asset**

- void **Asset.Enable**(string assetName, bool enable) : *Set Asset to enable Status*

• **Environment**

- void **Environment.ChangeWeather**(string selectedWeather) : *Set the selected Weather. Values: SUNNY - CLOUDY - RAINY - ICY - HAIL – SNOW*
- void **Environment.ChangeDayTime**(string selectedDayTime) : *Set the selected Weather. Values: DAY - DUSK - NIGHT - DAWN*
- void **Environment.SetIntensity**(string selectedIntensity) : *Set the selected Intensity. Values: NONE - LOW - MODERATE – HIGH*
- void **Environment.ModifyAsphalt**(string selectedAsphalt) : *Set the selected Asphalt Modifier. Values: AQUAPLANING - OIL - DEGRADEDASPHALT*

• **Locution**

- void **Locution.Play**(string locutionToPlay, bool wait) : *Play the selected locution. If there's any locution playing, it will stop.*
- void **Locution.PlayOneShot**(string locutionToPlay, bool wait) : *Play the selected locution just once. To play it again you have to do a **Locution.Restart**(). If there's any locution playing, it will stop.*
- void **Locution.Restart**() : *Allow to play again any locution*
- void **Locution.Stop**() : *Stop the selected locution*

• **Logger**

- void **Logger.Log**(string message) : *Log String to Console*

• **Objective**

- void **Objective.Increase**(string objectiveId) : *Increase the Objective punctuation by one*
- void **Objective.Decrease**(string objectiveId) : *Decrease the Objective punctuation by one*
- void **Objective.Declare**(string objectiveId, number goalValue) : *Declare new Objective with goalValue repetitions*

• **Timer**

- void **Timer.WaitSeconds**(float seconds) : *Wait until continue execution*

• **Traffic**

- void **Traffic.ChangeIntensity**(string selectedIntensity) : *Set the Traffic Intensity. Values: NONE - LOW - MODERATE – HIGH*
- void **Traffic.Cross**(string selectedEntity, string selectedDifficulty) : *Makes an entity cross in front of User Vehicle. Values for Entity: PEDESTRIAN - ANIMAL - BICYCLE - TORNADO - DOG - HORSE - COW - PIGValues for Difficulty: EASY - MEDIUM - HARD*

• **Trigger**

- void **Trigger.Enable**(string triggerName, bool enable) : *Set Trigger to enable Status*

• **UserVehicle**

- number UserVehicle.Speed() : Returns the User Vehicle Speed
- number UserVehicle.Gear() : Returns the User Vehicle Gear
- void UserVehicle.Respawn(string spawnPointName) : Respawns the vehicle on the selected SpawnPoint
- void UserVehicle.Enable(bool isEnabled) : Enable or disable vehicle controls

Anexo 2: SmartObjects

- **SmartAsset**
  - **SmartAsset.OnHit :** Method Executed On Asset Hit
- **SmartExercise**
  - **SmartExercise.OnStart :** Method Executed On Exercise Start
  - **SmartExercise.OnFinish :** Method Executed On Exercise Ended
- **SmartSpawnPoint**
  - **SmartSpawnPoint.OnEnter :** Method Executed On SpawnPoint Enter
  - **SmartSpawnPoint.OnStay :** Method Executed On SpawnPoint Stay
  - **SmartSpawnPoint.OnExit :** Method Executed On SpawnPoint Exit
- **SmartTrigger**
  - **SmartTrigger.OnHit :** Method Executed On Trigger Enter
  - **SmartTrigger.OnStay :** Method Executed On Trigger Stay
  - **SmartTrigger.OnExit :** Method Executed On Trigger Exit
- **SmartUserVehicle**
  - **SmartUserVehicle.OnStall :** Method Executed On User Vehicle Stalled
  - **SmartUserVehicle.OnHit :** Method Executed On User Vehicle Hitted