Bethesda Tutorial Papyrus Introduction to Properties and Functions

creationkit.com/index.php

Contents

Overview

This tutorial assumes you've already completed the Hello World Tutorial and the Variables and Conditionals Tutorial. This will be a longer tutorial than the first two, and introduce you to some more advanced topics. Hang in there!

You will learn:

- About things glossed over in previous tutorials:
  - The first line of a script, and what it means to "extend" a script
  - How to add tool tips to your script using {}
- How to use properties and hook them up in the editor
- How to create and use a function

First Line

Before going further, lets take a look at some things we've glossed over in the previous tutorials.

If you've been following along, looking at the top of your script, you'll see this first line:

```
Scriptname HelloWorldScript extends ObjectReference
```

- ***Scriptname*** *HelloWorldScript extends ObjectReference*: the first line of any script starts with this.
- *Scriptname* ***HelloWorldScript*** *extends ObjectReference*: you are naming your script here. In this case "HelloWorldScript" any time any other script needs to refer to this one, it will use the name you give it here.
- *Scriptname HelloWorldScript* ***extends*** *ObjectReference*: this is a special word that means essentially, this script is based on another script that already exists.
- *Scriptname HelloWorldScript extends* ***ObjectReference***: this script is extending (based on) this other script, in this case the "ObjectReference" script.

Extending a script

Virtually all of the scripts you write will need to "extend" another script. When you extend a script, you are saying, "My script is the same thing as this other script, plus whatever extra stuff I've added to it."

Which script you are extending is also a way of saying to the game, "My script is this type of object."

For example, if you are adding a script to an object that is going to be a reference in the world (like our pillar for example) your script will need to "extend" the "ObjectReference" script. If you want to add a script to a Quest you would "extend" the "Quest" script. And so on.

As you become more familiar with scripting, you will become more familiar with the contents of the scripts you are extending. For now, just know that your script will need to extend another script that is based on the type of object you are attaching your script to.

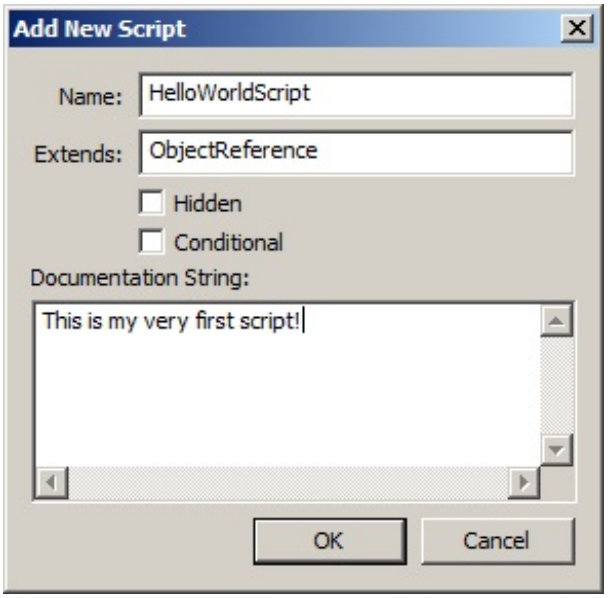
Adding Tooltips

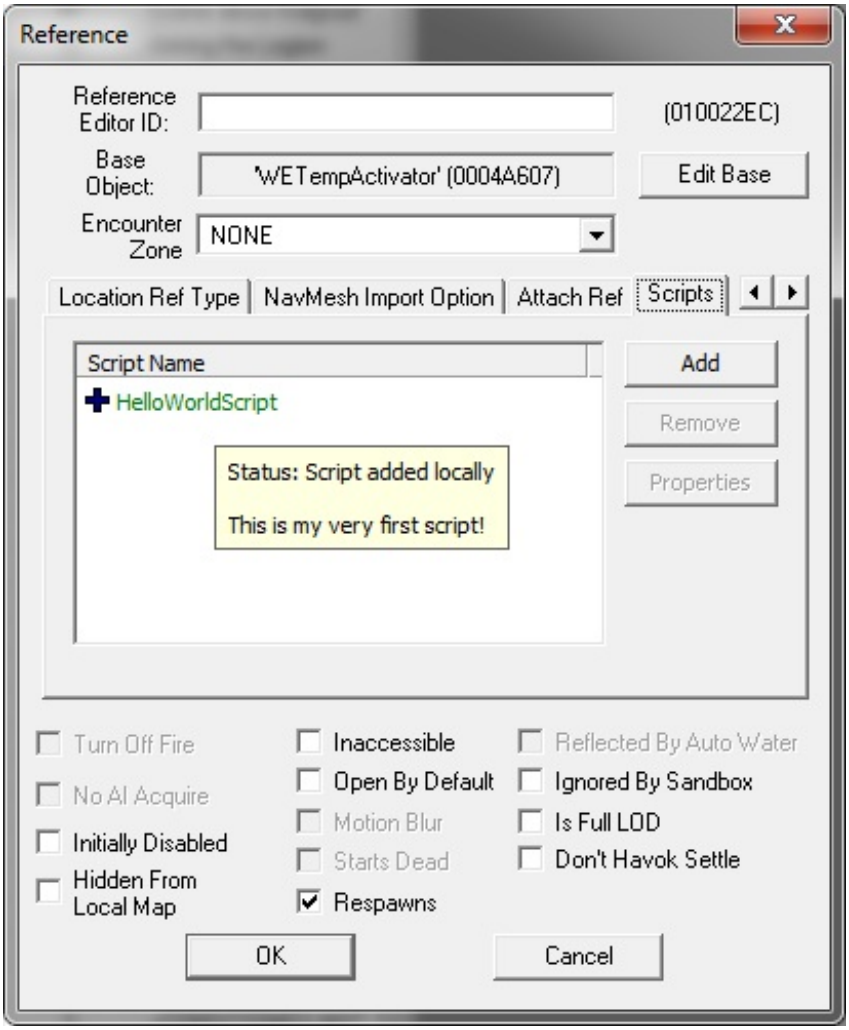
Wondering where the second line in your script came from? The one that looks like this:

```
{This is my very first script!}
```

When you created your new script and entered the documentation string, this line was created for you:

This now shows up as a tool tip when you hover over your script in the scripts tab:





If you want to change the tooltip, just change the text between the curly-braces {}

You can also add tooltips to help you manage your properties. Which brings us to:

Properties

The following definition of a Property comes from cipscis.com

---

> *Because scripts are not internal parts of data files, and the compiler is not an integral part of the Creation Kit, you cannot directly refer to information from a data file in a script. If you want to refer to something in your data file, such as a particular actor, then you need to use something else as an intermediate. This "something else" is called a property.*
>
> *A property allows your script to use information in a data file by providing a special interface that they can both access. From your script's point of view, a property is an arbitrary piece of information of a certain specified type, like "Actor" or "Quest". From the Creation Kit's point of view, it is a point at which information of that type can be inserted into the instance of your script attached to the object you're editing.*
>
> *Declaring properties is very similar to declaring variables, except there are a couple of extra things you need to do, and properties cannot be declared inside functions.*

Lets go thru an example of a typical property declaration.

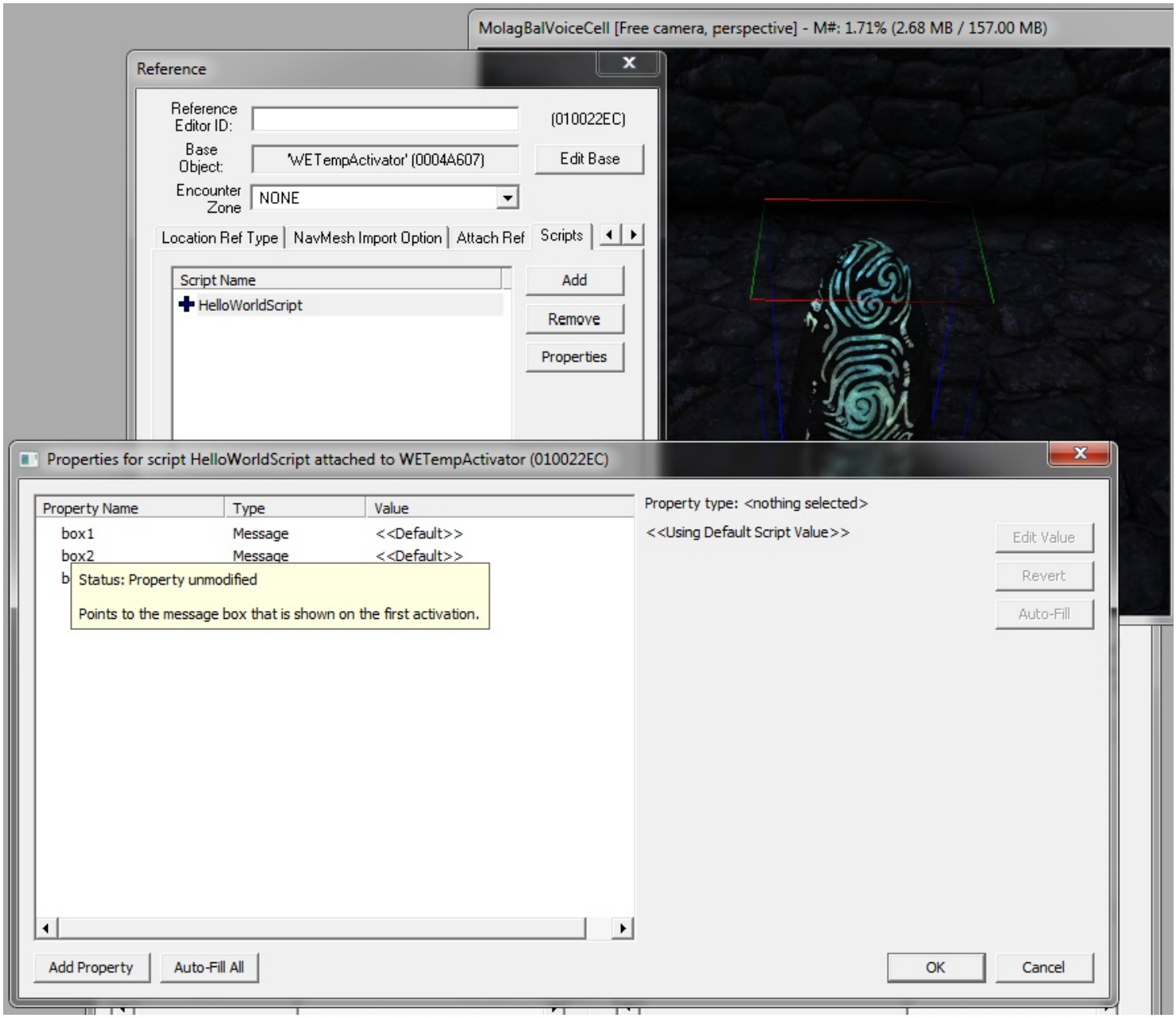
Add this above the "int count" line in your script:

```
Message property box1 auto
{Points to the message box that is shown on the first activation.}

Message property box2 auto
{Points to the message box that is shown on the second activation.}

Message property box3 auto
{Points to the message box that is shown on the third activation.}
```

- ***Message*** *property box1 auto*: The type of the property. In this case "Message." (This is similar to how we defined the "count" variable in our script to be of the type "int.")
- *Message* ***property*** *box1 auto*: We are saying this message is NOT a variable. It IS a property.
- *Message property* ***box1*** *auto*: We are saying the name of this property is "box1". (This is similar to how we defined the name of our "count" variable)
- *Message property box1* ***auto***: This is a special keyword that you just have to remember to use when you are declaring properties. (Almost all properties will use the "auto" keyword. Don't worry about why this is for now. Just remember to add "auto" when you are declaring properties.).

You'll notice the curly-braces here {}. That will provide the tool tips. Let's take a look:

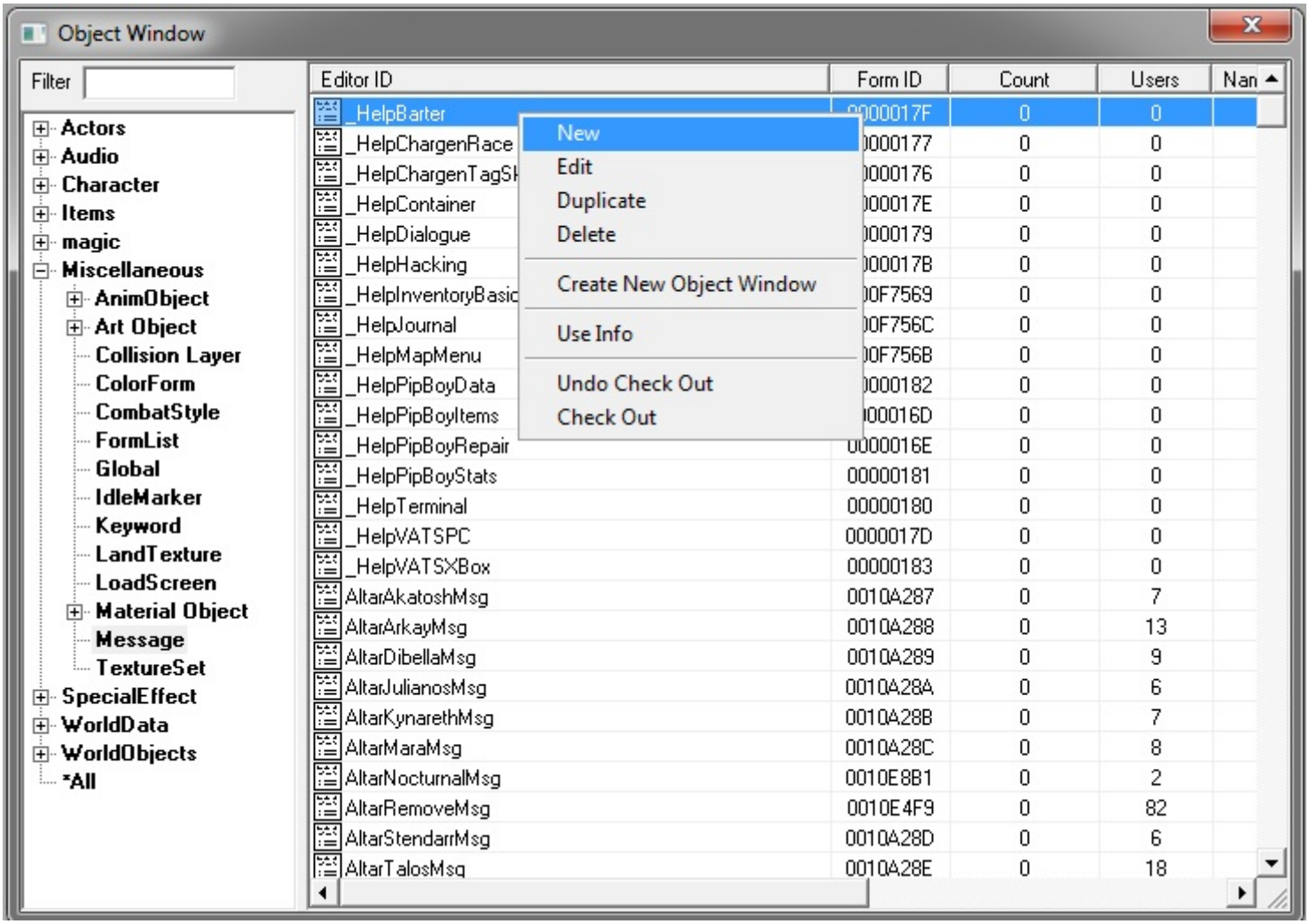
Hit file->save. Close the script window. And hit the properties button on the pillar reference form. You should see a list of properties. Hover over their names.



Creating Messageboxes

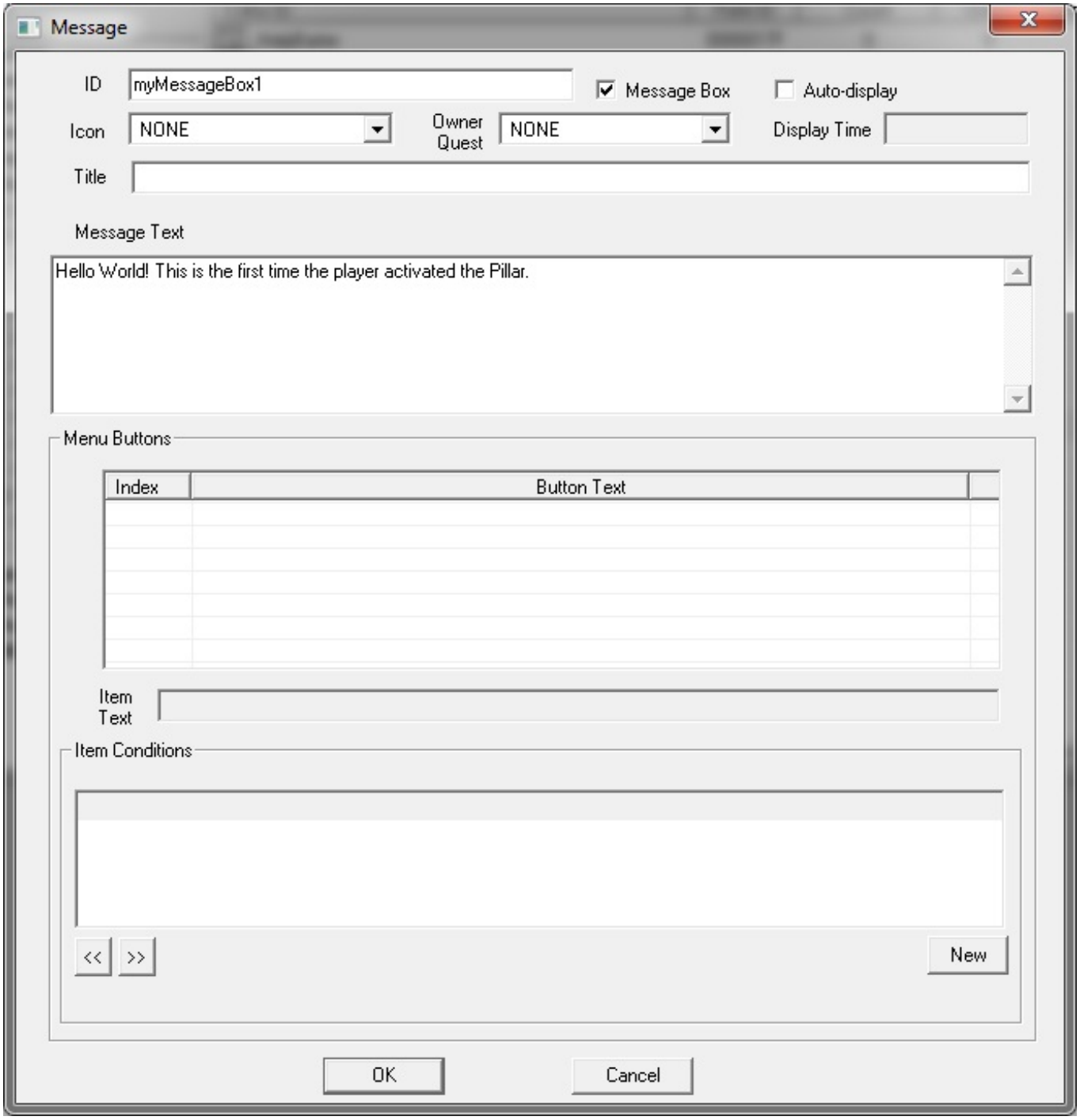
Now we will create the Message objects that we will hook up to our properties.

In the Object Window expand the "Miscellaneous" category, click on the "Messages" category. In the list of messages RIGHT CLICK, and select "New" from the context menu.



Enter the following:

- ID: myMessageBox1
- Message Text: Hello World! This is the first time the player activated the Pillar.



Create two more message boxes with the following:

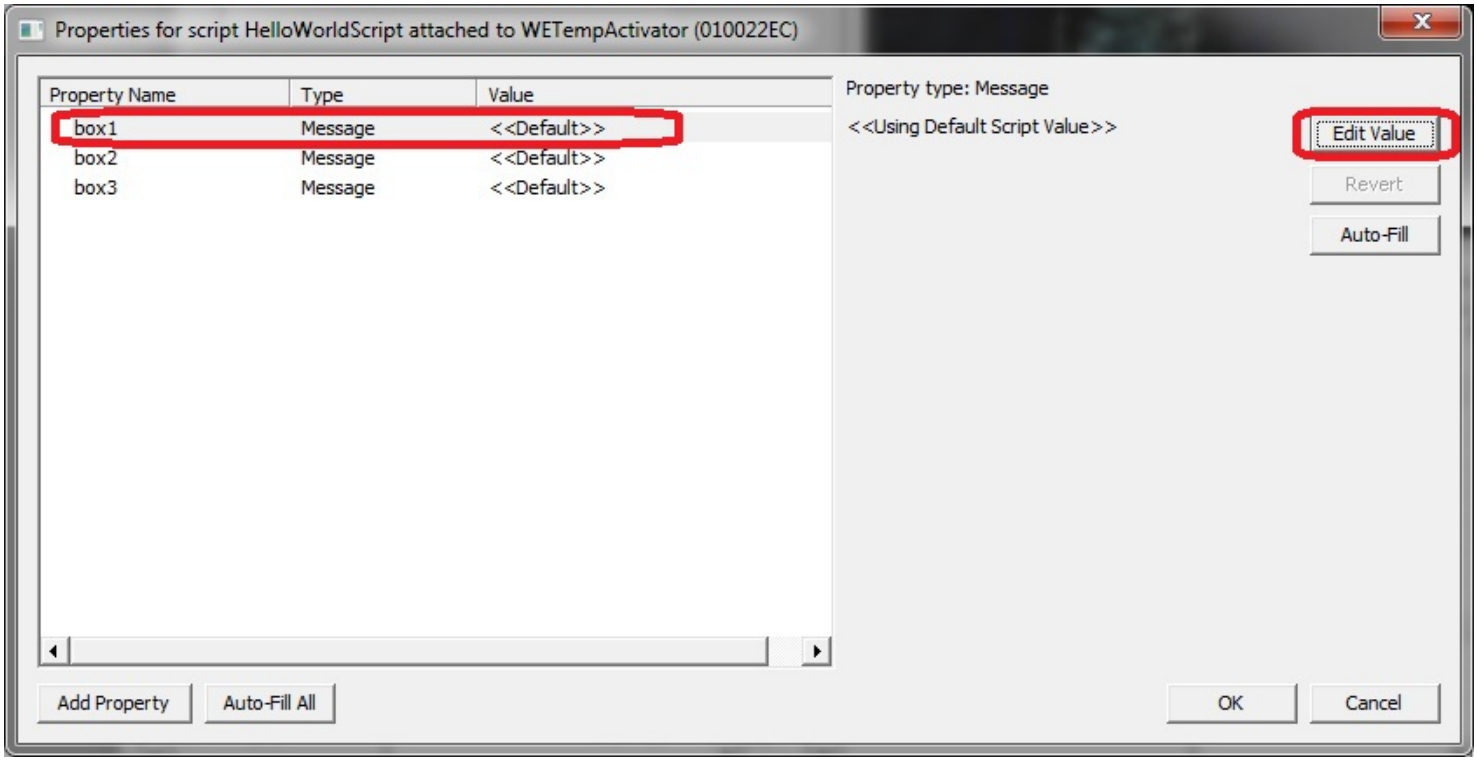
- ID: myMessageBox2
- Message Text: This is the second time the player activated the Pillar.
- ID: myMessageBox3
- Message Text: It's been three or more times activating the Pillar.

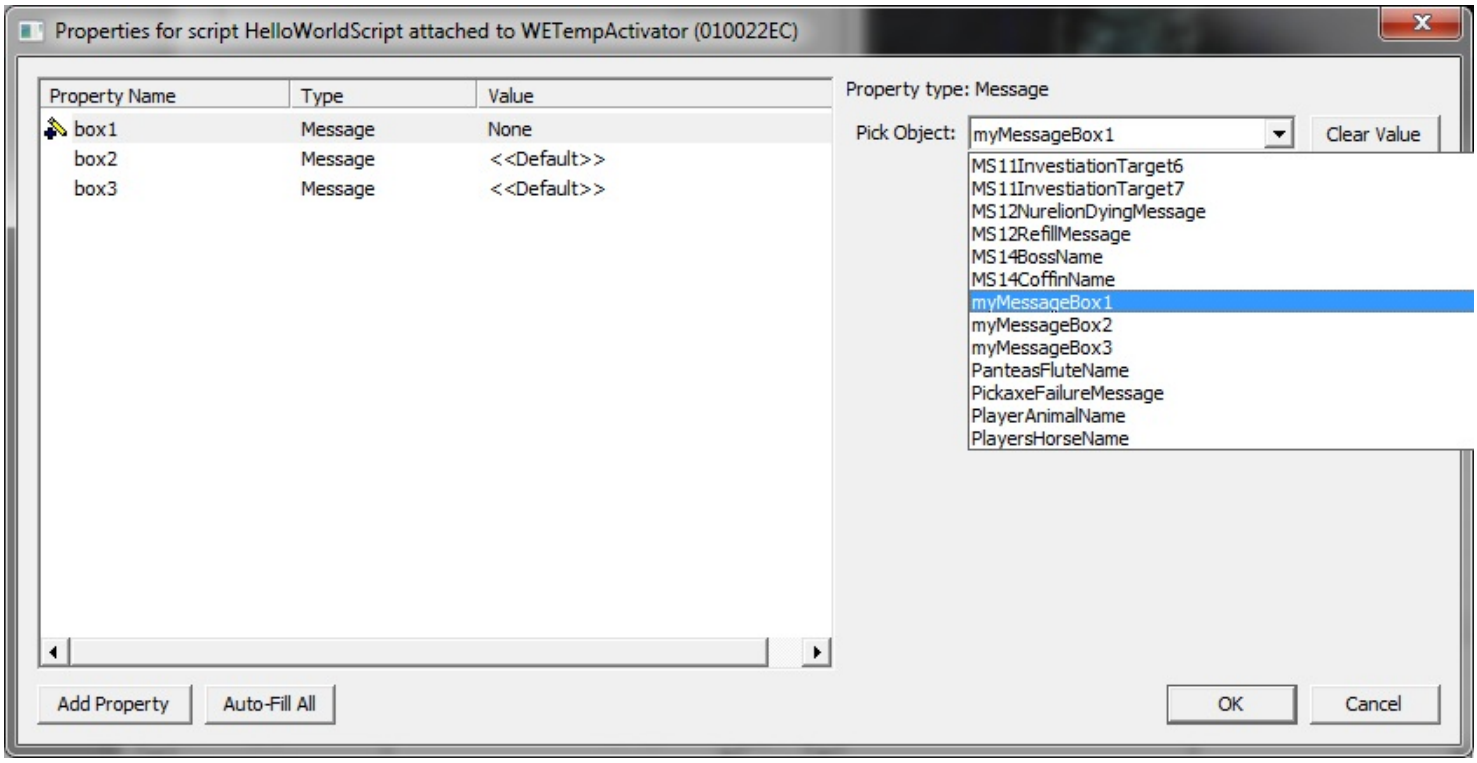
Hooking up the message boxes to the properties in the script

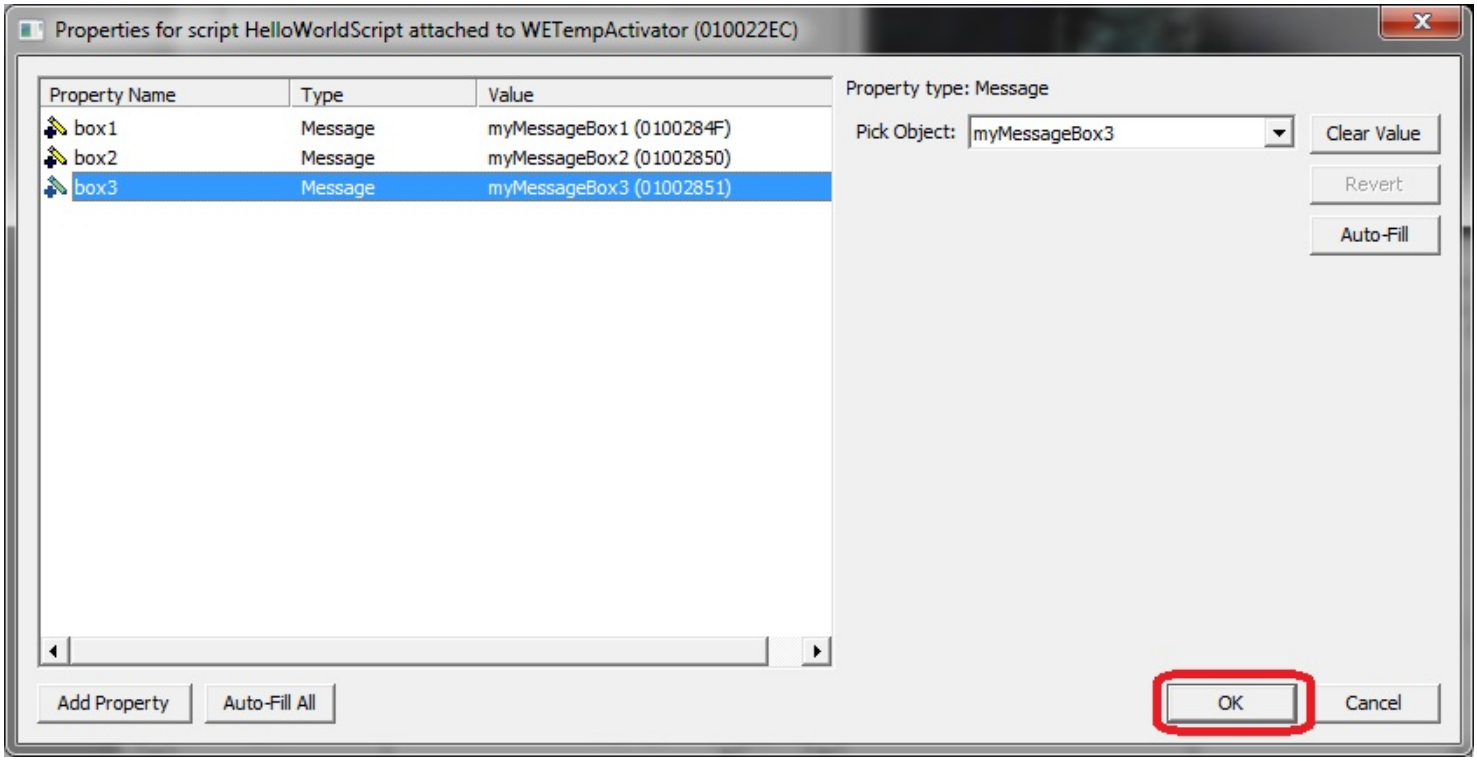
Go back and open the Properties window of your script attached to the Pillar.

Click on the line with "box1" and then hit the "edit value" button. This will create a "Pick Object" drop down from

which you should select the myMessageBox1 item. Similarly select myMessageBox2, and myMessageBox3 for box2, and box3 properties respectively.







Make sure you hit "OK" to close the reference window (if you don't you will lose the changes you made), then save your plugin.

So what we have essentially done here is make a connection between our properties (box1, box2, and box3) to three message objects in the editor (myMessageBox1, myMessageBox2, and myMessageBox3).

After confirming you have each of the three message boxes hooked up to the proper properties, we are ready to continue editing our script.

Calling functions on Properties

Now let's get these messageboxes to show up in game.

Open up your script to edit, and REPLACE these lines:

```
if count == 1
   Debug.MessageBox("Hello, World!")
elseif count == 2
   Debug.MessageBox("Hello, World. Again!")
else
   Debug.MessageBox("Hello, World. Enough already!")
endif
```

WITH THESE lines:

```
if count == 1
   box1.Show()
elseif count == 2
   box2.Show()
else
   box3.Show()
```

```
endif
```

What we are doing here is calling the **Function** "Show()" on each of the message objects in the box1, box2, and box3 properties. The period (.) means the function to right ("Show()") should be called on the object to the left (the message inside the "box1" property).

Save the script, **hit OK** to close the reference edit form. Then **save your plugin**. And run the game! (COC MolagBalVoiceCell)

You should see the text of each of the messages come up each time you click the pillar.

(If you get a warning about "cannot call show() on a none object" go back and make sure you have your properties hooked up properly to the message objects, and that you hit okay on all the open forms and saved your plugin. Then try again.)

Okay, now let's create a function!

Creating Function

Just like we called "MessageBox()" function on Debug, and "Show()" on the messages inside our properties. We can call a function we create in our own script. But before we can call a new function, we must create it.

Add this to the bottom of your script.

```
Message function GetMessage(int currentCount)
  Message chosenMessage

  if currentCount == 1
      chosenMessage = box1
  elseif currentCount == 2
      chosenMessage = box2
  else
      chosenMessage = box3
  endif

  Return chosenMessage

endFunction
```

Let's break down the first line:

- ***Message*** *function GetMessage(int currentCount)*: Here we are saying this function is going to return a Message object. Note that this is optional - functions are not required to return anything.
- *Message* ***function*** *GetMessage(int currentCount)*: Here we are saying "this is a function" similar to how we used "property" when declaring a property.
- *Message function* ***GetMessage****(int currentCount)*: This is the name of the function. It is what we will use to "call" it from our script.
- *Message function GetMessage****(int currentCount)***: Everything in parentheses () are the "arguments" we will need to give to our function when we "call" it. In this case we will need to supply an INTeger. We are naming our argument "currentCount." We will use this name in the function to refer to the value that we pass to the function when we call it.

The next line:

- ***Message*** *chosenMessage*: we are declaring a variable, whose type is Message. (Just like we declared our "count" variable as a type "int" we can make a variable of any kind of object. In this case it's a "Message" object.
- *Message* ***chosenMessage***: we are naming the variable "chosenMessage"

In the next section we are making chosenMessage be one of the Message object that are in box1, box2, or box3 properties. (Again, note the differences between "==" and "=")

Finally we:

```
 Return chosenMessage
```

This means that when we call the GetMessage() function, it will give to us the Message object that is currently in the chosenMessage property.

And now, we will change the script to make use of our new function.

REPLACE these lines:

```
if count == 1
   box1.Show()
elseif count == 2
   box2.Show()
else
   box3.Show()
endif
```

WITH THIS line:

```
 GetMessage(count).Show()
```

What we are doing here is calling GetMessage() and passing in count as the parameter. Because GetMessage(count) will return a Message, we can call Show() on that, and it will cause the Message to show in game.

So your script should now look like this:

```
 Scriptname HelloWorldScript extends ObjectReference
 {This is my very first script!}

 Message property box1 auto
 {Points to the message box that is shown on the first activation.}

 Message property box2 auto
 {Points to the message box that is shown on the second activation.}

 Message property box3 auto
 {Points to the message box that is shown on the third activation.}

 int count  ;stores the number of times this object has been activated
```

```
Event OnActivate(ObjectReference akActionRef)
   count = count + 1

   GetMessage(count).Show()
endEvent

Message function GetMessage(int currentCount)
  Message chosenMessage

  if currentCount == 1
      chosenMessage = box1
  elseif currentCount == 2
      chosenMessage = box2
  else
      chosenMessage = box3
  endif

  Return chosenMessage

endFunction
```

When we run the game, we'll see the message boxes pop up correctly. (COC MolagBalVoiceCell)

What's next

That's it for the basic tutorials. The next one demonstrates how you might make a special boss who reanimates guys.