**Editing ScriptEase Libraries: Making and Editing Story Components**

This tutorial will introduce you to the Neverwinter Nights Library Editor in ScriptEase and help you understand the mechanisms for creating your own causes, effects, descriptions, and activities. Ultimately, the default Neverwinter Nights library should cover most of your scripting needs, but this tutorial will introduce you to ScriptEase’s unique development features if you ever need to make your own. This tutorial assumes you have completed the basic ScriptEase tutorials and have experience with NWScript.

Overview: we’ll be making (1) a simple cause about picking up a unique item, (2) a complex cause dealing with the internal hours of the game, (3) a swarm effect to generate objects, (4) a simple description to determine if a creature is dead and (5) a simple activity about an annoying NPC that follows the Play around.

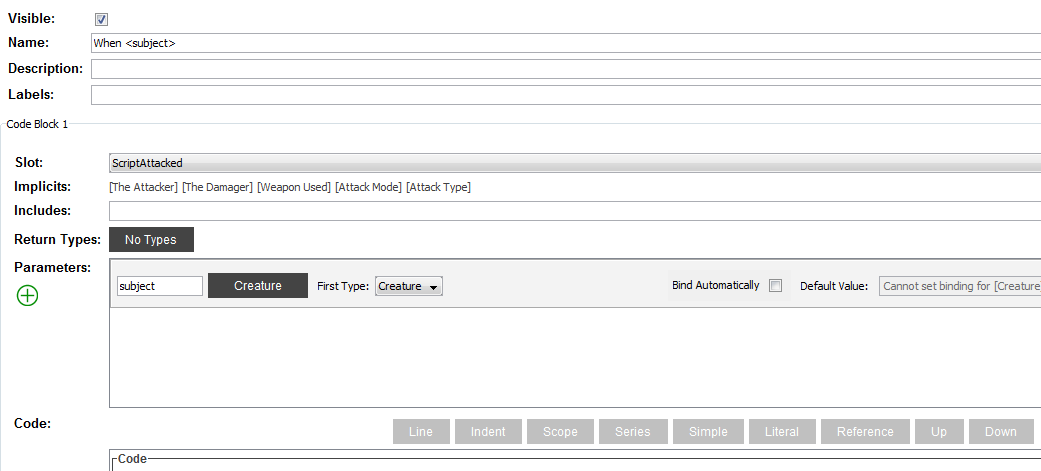
Creating a Library

The libraries in ScriptEase hold the story components we need to generate our game scripts. We’ll begin by making a new Library.

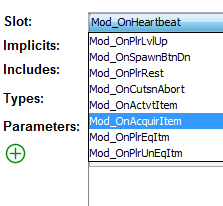
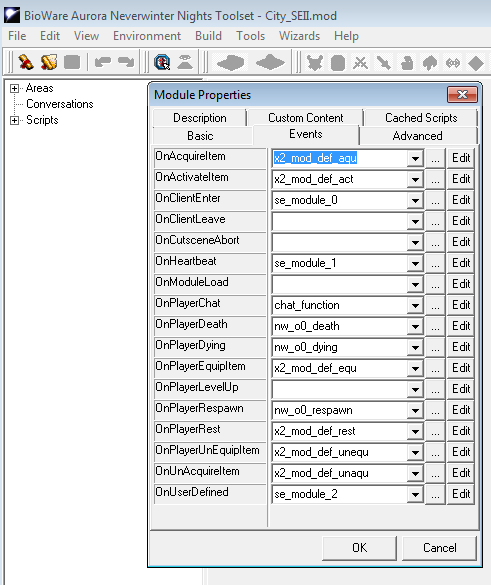
1. First, open ScriptEase. Under the “Library” drop down menu, highlight the “Neverwinter Nights” slot and pick “New Library”.
   1. The default library is read only, but you can still look at it.
2. You must give the library a name, such as “My Library”. You should also provide an author name and description.
3. Click Finish. The library will load up. It will be empty, but the default library’s components will be shown in grey in the library pane. These are there in case you want to duplicate any of them or use any of the effects for descriptions.
4. Excellent, now for the fun part.

Making a Cause

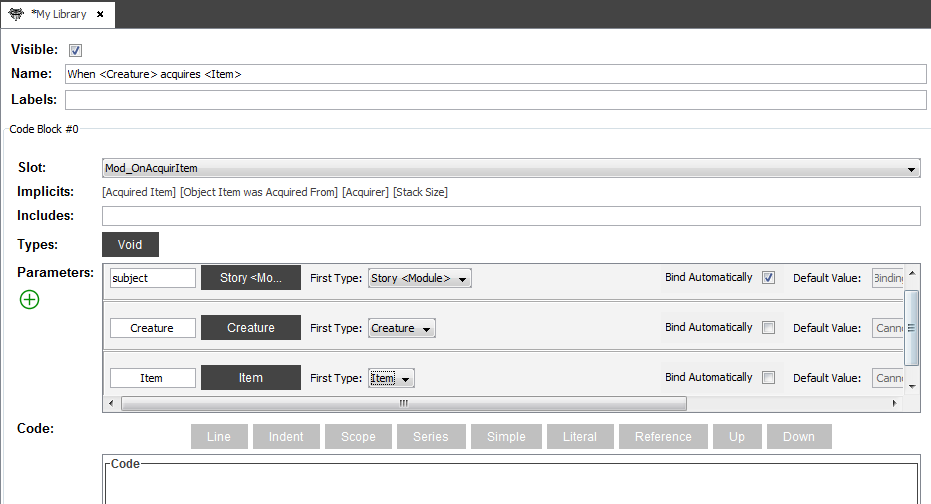
1. In your new library click “File->New->Cause” which will bring you to this screen.



1. First, in the “Name” box type “When <Creature> acquires <Item>”. The angle brackets denote the parameters of the cause.
2. There is already a cause about acquiring items called Notice that each of the parameters is a green implicit parameter. An implicit parameter is prebound by ScriptEase to the appropriate object when the cause fires. As such we can’t change them.   
     
   Usually, users would use the Acquirer and Acquired Item with two pairs of descriptions and questions to find out who used what. However, we want to be able to specify a particular creature and item.
3. You can leave the “Labels” box blank. We usually use these for effects when we want them to mark them as effects that occur immediately (“NOW”) or are placed in an objects queue (“TODO”). Note that they don’t have any functionality; they’re just labels.
4. Every parameter has one or more types. One of the parameters is special in that it is the object on which the event occurs. It is listed first in the “Parameters:” field and its name is “subject.” It cannot be deleted. By default, its type is “Creature.” In our case, we want this first parameter to be the story module since in Neverwinter Nights the event for acquiring an item happens on the module object.   
     
   It is unusual that this is not an event on the creature that acquired the item. In NWN several events occur on the “Story <Module>” rather than on the individual objects. Even though the “Story <Module>” does not appear explicitly in the “Name:” field, it must be used in creating this Cause. Therefore, go to the “Parameters:” field and click on the small black box next to [subject] labeled “Creature”. This will bring up a pop-up screen with a list of types. Deselect “Store” and select “Story <Module>” under Game Objects. Doing this lets us place scripts in the right slot in the Module’s Events panel instead of using the slots related to the Creature.



1. Now we need to pick an appropriate event slot for the first parameter (subject) in the “Parameter:” field. This is done by going to the “Slot:” field and picking from the drop down list. For this cause we need the “Mod\_OnAcquirItem” event.
   1. When you pick the “Mod\_OnAcquireItem” even, this will update the “Implicits:” field with [Acquired Item], [Acquirer], [Stack Size] and [Object Item was Acquired by] since these implicits are part of the event. Ignore them. They will not appear as parameters in our new Cause, since they are not referred to in the “Name:” field using angled brackets as delimiters.
2. Tick the box next to “Bind Automatically” since the first parameter (subject) in the “Parameter:” field has type “Story <Module>”. If another type was used for the first parameter, this box should not be ticked. ScriptEase will bind the “Story <Module>” automatically once this box is ticked. It can do this since there is only one module for each ScriptEase story.
3. Some ScriptEase components (Causes, Effects, Descriptions) can return an object so we must specify a return type. In NWN the return type of all Causes must be set to Void. Use the pop-up dialog in the “Types” field to change the type to “Void.”
4. Now back to “Parameters”. The reason we added “<>” angle brackets to “Creature” and “Item” in our name is because we want to define them as parameters. To do this, click on the green (+) symbol under “Parameters”. This will add an empty parameter.
   1. Name this parameter “Creature” and change its type to [Creature].
   2. Add a second parameter called “Item” and change it’s type to [Item].
   3. Your screen should look like this.



Adding Code

Now that that’s done let’s add in what we actually want the Cause to do. If you check out the default “When <Acquirer> acquires…” cause you’ll notice that in the “Code” section is almost completely empty. The only thing in the “Code” box is a purple “Format Reference” box with the text field “Children”. This is because most causes are, essentially, containers for effects that do the legwork. If you head over to the Effects tab and click on one, you’ll see that the “Code” section is full of code. Why? Think of a cause as an “If” statement.

if(a creature acquires an item){

Do something (the something is an effect)

}

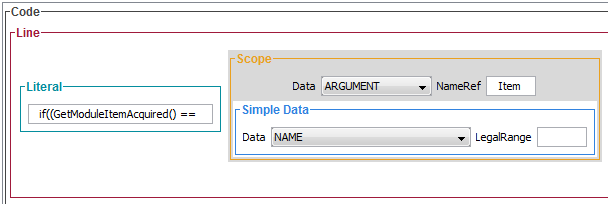
This works perfectly well for most things, but there is room for specificity. But first let me describe some of the things we’ll be using. Next to the “Code” field there are a 7 coloured buttons that will be our tools. If you have not selected the “Code” box below the buttons, they will be greyed out. These buttons add various code fragments that will tell ScriptEase II how to generate the code.

* Line: This adds a Line box to the Code. This means that everything you add to this box will be on one line in the generated scripts.
* Indent: adds an indent to the generated code. Use it for readability. If you’re using this with lines, the lines must be inside of the indent block.
* Scope: Scope tells fragments where to get data from. We’ll be using it to scope into the parameters we created for this component.
* Series: We won’t be using Series here, but they act like loops.
* Simple: Simples are used to access data. We usually use them inside of Scopes and Series fragments.
* Literal: Literals will output exactly what you type in as code.
* Reference: References an existing format, which is a group of code. We have one called “children” that writes the code for anything inside of a cause.

Try looking at other effects to see how these are used together to generate code. Compare them to the outputted code in NWScript.

Now, let’s get started.

1. Select the Code box and click on the Line button. This will add a Line box. Select the new Line box and click on Literal to add a Literal box inside of it.
   1. In the Literal Box text field, type if((GetModuleItemAcquired == .Remember the implicits from earlier? We need to make sure that the item we want to specify is the same as the implicit [Acquired Item]. We do this so that we can control when a Cause will act. If we did not add in these conditions, then the Cause may fire it’s Effects when any item is picked up, or if a specific item is picked up by any creature.
2. Next reselect the Line Box and add a Scope box by pressing the Scope button. This will add the Scope box next to the Literal we added in step 1. If you find that the Scope box is added underneath the Line box, that is because you had the Code box selected. Delete it and select the Line box. Next select the Scope box and add a Simple box inside of it.
   1. In the NameRef field type “Item.” This will let us use the Item Parameter that we created earlier as an argument in our code.
   2. So far it should look like this:

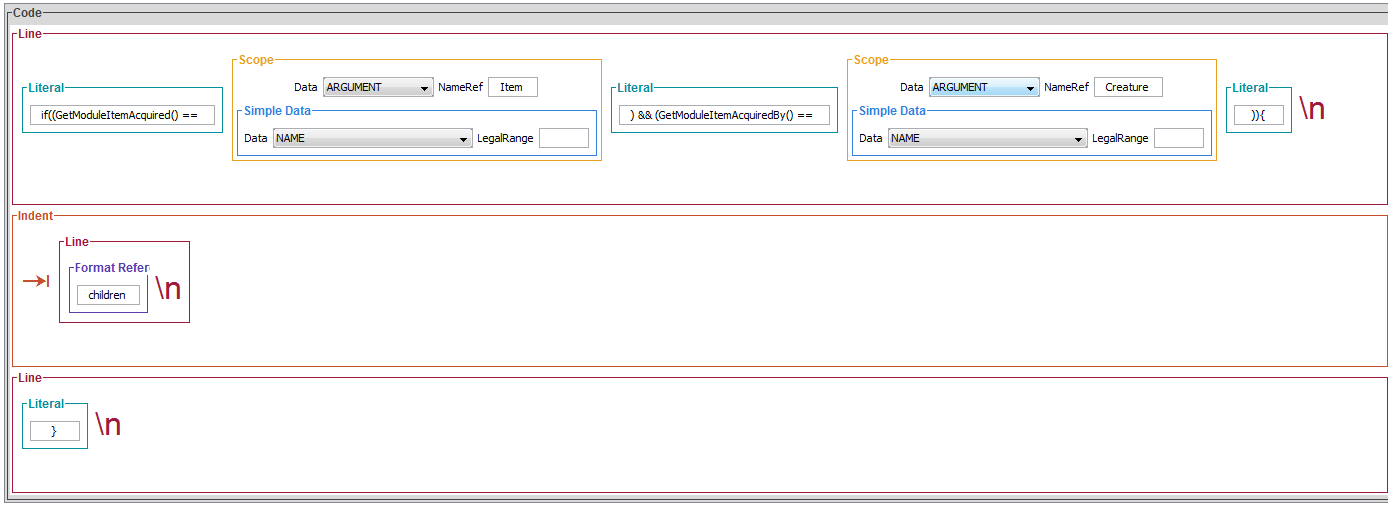


1. Reselect the Line Box and add another Literal. This time in the type field add ) && (GetModuleItemAcquiredBy() == . Same as in step 2, we want to make sure that we are adding conditions to our Cause so that the right creature acquiring the right item will set off any Effect we have planned. This piece of code will be used to compare our Creature to the Implicit [Acquirer].
2. Repeat step 2 by adding a Scope and Simple combined box, this time using “Creature” as the NameRef text.
3. Reselect the Line Box and add another Literal with the text )){

Fantastic! We’ve made our first line. Now let’s add the boxes we need so that the Cause will fire the Effect.

1. Select the Code Box that contains the first Line.
   * 1. Add an Indent
     2. Inside the Indent add a Line
     3. Indents can hold many Lines, but Lines can’t hold many Indents without looking disorganized.
     4. Inside the Line Box add a Reference.
     5. Type “children” inside the text box of the Reference. Remember, this is the box necessary for executing effect.
2. Lastly, reselect the Code box and add another Line, inside the Line add another Literal and type }

Your screen should look like this:



If this were in the toolset script editor it would look like this.

if((GetModuleItemAcquired() == Item) && (GetModuleItemAcquiredBy == Creature)){

Format reference : DO THE EFFECT

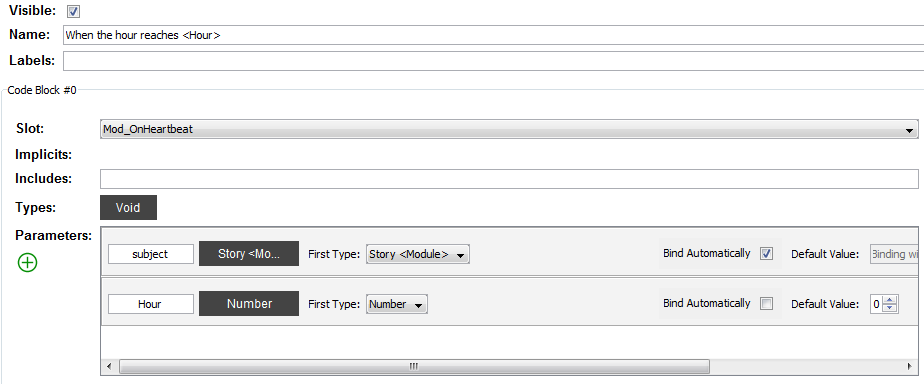
}

Congrats! You’ve made a new Cause!

A Little More Challenging

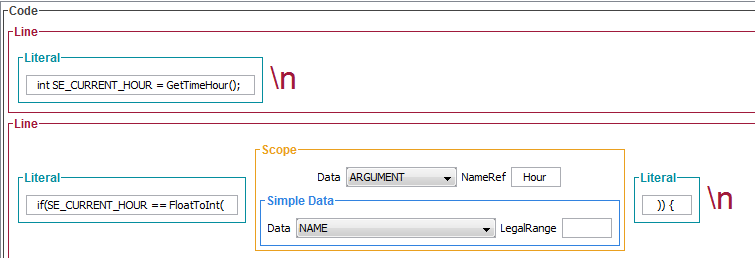
This cause will be used to determine the exact hour inside of the Aurora engine. This can be used to make time specific behaviours or create schedules for your characters to make them more dynamic. This will be more precise than just relying on the “OnHeartbeat” Module Property.

1. Start by making a new cause (“File->New->Cause”). Name it “When the hour reaches <hour>” and leave the Label Blank.
2. In the “Parameters” box, change the [subject]’s type to “Story <Module>” and bind it automatically. Change the “Types:” field above to “Void”
3. In the “Slot:” field select “Mod\_OnHeartBeat” from the drop down menu.
4. Add one parameter by clicking on the green (+) sign. Name it “Hour” and change its type to Number. Check Bind Automatically.



Onto the Code.

1. Add a Line box and then a Literal box inside of it. We’re going to create a local int variable called SE\_CURRENT\_HOUR. Use the GetTimeHour() method and assign SU\_CURRENT\_HOUR its value.
   1. IMPORTANT! A good practice in making your own causes and effects is naming variables with the SE\_CAPITAL\_LETTERS naming convention. This will help prevent errors if there is another variable with the same name and will make it more readable if you were to look at the scripts in the toolset.
2. Reselect the Code box and add another Line. Inside the Line add a Literal and type in if(SE\_CURRENT\_HOUR == FloatToInt(
   1. This will be our way of comparing the actual hour of the game to the hour we have selected.
3. Next add a Scope/Simple Box, and in the NameRef field type “Hour”
4. Close the if statement with another Literal with brackets: )){

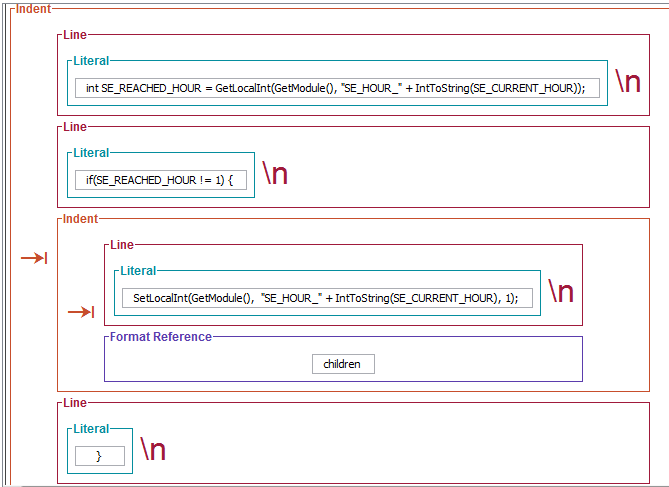
So far it should look like this:

1. Next add an Indent to the Code block. Inside, add another Line+Literal Box. We’re going to be making another variable as a check to make sure there aren’t any errors in our scripting.
   1. Name the integer SE\_REACHED\_HOUR and assign it the value of the GetLocalInt(GetModule(), “SE\_HOUR\_” + IntToString(SE\_CURRENT\_HOUR));.
      1. The reason why we use “SE\_HOUR\_” + InToString instead of just InToString is for the same reason as above. It is safer to use the “SE\_” naming convention to prevent conflicts with other variable names.
2. Select the Indent box, and add another Line+Literal. We will be making an if statement that will be checking if SE\_REACHED\_HOUR does not equal 1.
   1. if(SE\_REACHED\_HOUR != 1){
   2. Next add another Indent with a Line+Literal box inside of it. We will be setting our “SE\_HOUR\_” variable we created above to 1 with

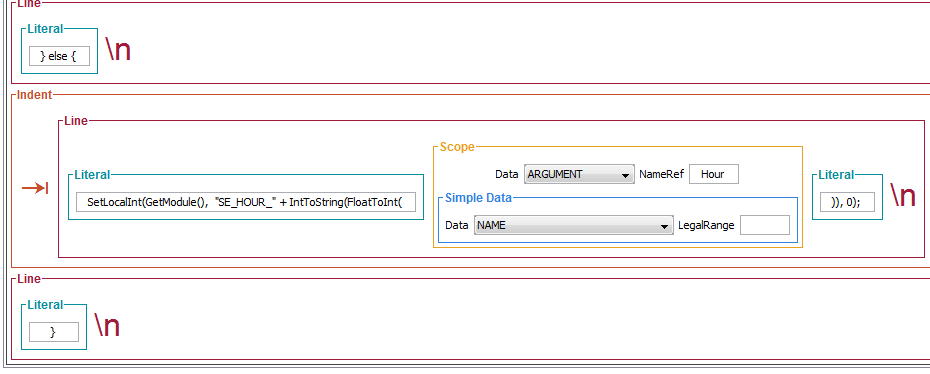
SetLocalInt(GetModule(), “SE\_HOUR\_” + IntToString(SE\_CURRENT\_HOUR), 1);

* 1. Back to the inner Indent, right below our SetLocalInt() box, add a Reference, and make sure that its text box reads “children”
  2. Back to the outer Indent box, add another Line+Literal box and close the loop.

1. Your new block of code should look like this:



1. The reason we’re using the inner loop is to make sure that we aren’t firing the Effect in the Format Reference block every six seconds, which is what the OnHeartBeat module script does. In plain terms, this Cause checks to see if the hour you want is the current hour of the module. If it is and it hasn’t been “seen” before, it will fire its Effects and then exit the loop until the next hour. Which leads us into the final step of the Cause.
2. We’ll be making an else statement to update our “SE\_HOUR\_” + IntToString variable from above to make sure that our Cause will actually fire when it reaches the right hour.
3. Start by selecting the original Code box and adding a Line+Literal box. In the Literal text field type } else {
4. Add an Indent below the previous step, and add a Line+Literal within it. The text of the Literal should read SetLocalInt(GetModule(), “SE\_HOUR\_” + IntToString(FloatToInt(
5. Reselect the Line box from step 14 and add a Scope+Simple box. In the NameRef field type “Hour”. In the Simple box, NAME should already be selected from the drop down menu.
6. Close the method by reselecting the Line box and adding a Literal that reads )), 0);
7. Finally select the original Code box, and add in a Line+Literal closing the else statement with a }

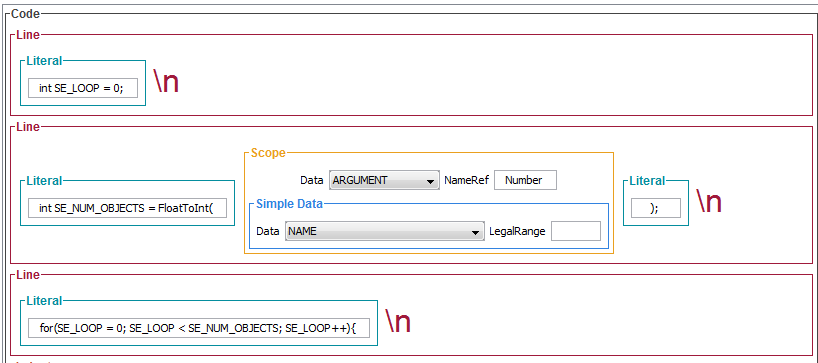


Second Cause done!

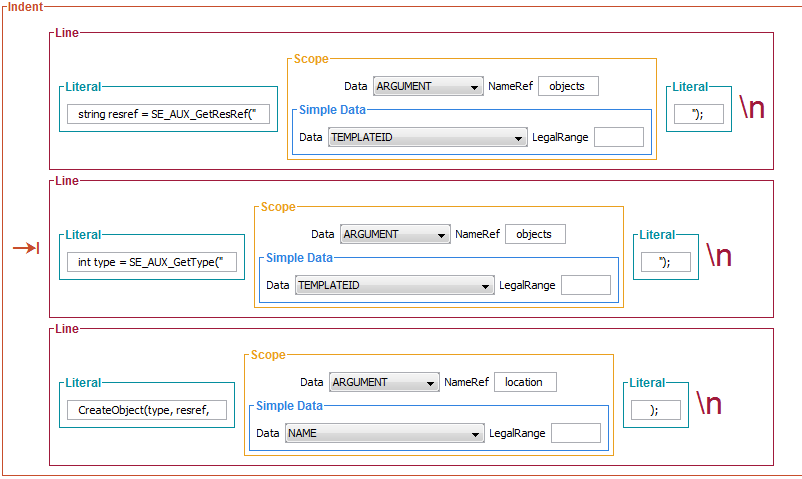
Making an Effect

Now that we have a couple causes under our belts, let’s make an effect.

1. First start off by selecting “File->New->Effect.”
2. In the “Name” box, type: “Create a <number> of <objects> at <location>” These will be our parameters for the effect. In the “Labels” box, type: “NOW” since this will be an instant effect. (Reminder: the other Effect label is “TODO” which fires in sequence)
3. Next we will set the “Type” of the effect. Click on the “No Types” button next to the “Types:” text.
   1. Click on “Void” under “Game Constants” and hit okay.
4. Now we need to declare our variables. Add three parameters to the Parameters field using the green (+) button. Name them “Objects” “Location” and “Number”.
   1. For the “Objects” parameter select “Item, Waypoint, Creature, Placeable and Store” from the Types pop-up. Change the “First Type” drop down to “Item.” We are selecting multiple types in order to use multiple kinds of game objects.
   2. For “Location” change its type and First Type to “Location”
   3. For “Number” change its type and First Type to “Number”
   4. They should not (and cannot) be bound automatically.
5. Next add a Line+Literal and in the Literal box create an integer named “SE\_LOOP” and initialize it to 0;
6. Select the Code box, and add another Line+Literal. Inside the Literal type
   1. int SE\_NUM\_OBJECTS = FloatToInt(
7. Reselect the Line box we just added and add a Scope+Simple box next to the Literal and in the NameRef field type Number. In the Simple Data field the NAME drop down should already be selected.
8. Add another Literal within the Line and Close the FloatToInt method with the text );
9. Finally we’re going to make a for loop using the variables we initialized above. Select the original Code box and add a Line+Literal. In the text field type for(SE\_LOOP = 0; SE\_LOOP < SE\_NUM\_OBJECTS; SE\_LOOP++){



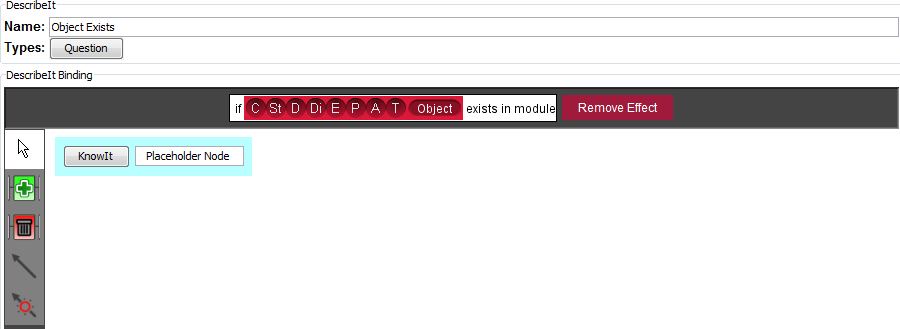
1. Now to add the content of our For loop. Selecting the Code Box, add an Indent with a Line+Literal inside. Inside the Literal type string resref = SE\_AUX\_GetResRef(“.
   1. The SE\_AUX\_ prefix is from a pre-generated ScriptEase script containing methods that assist in ScriptEase’s function. Check out the i\_se\_aux.nss file in the translator’s resources/includes directory, or in a module edited with SE2, to see other methods we have.
   2. We use this to retrieve the Blueprint ResRef of a unique object so that we can later use it with the CreateObject() Neverwinter Nights function. CreateObject requires the ResRef of an object and can be difficult to use with “Edit Copy” items as it can show you a copy of the blueprint resref but not the object created from a blueprint resref itself. More steps are required to use a ‘unique’ resref, so to make things easier we’ll use the pre-created ScriptEase methods.
   3. Now we’ll add the Scope+Simple box. Change the NameRef to Objects in the Scope box. In the Simple box (not the Scope box) change the choice in the Data drop down menu to “TEMPLATEID”. The TemplateID is a stronger version of a tag.
   4. Next add another Literal box and close the brackets of the method “ ); ”
2. Re-select the Indent box and add another Line+Literal box. In this Literal type in the following: int type = SE\_AUX\_GetType(“
   1. The SE\_AUX\_GetType method returns the type of the object so that later methods know what to create. For example if the object is a creature or an item or a store. That’s why, when we were changing the Type of the Objects parameter, we selected 5 possible types that Objects could be.
   2. Add another Scope+Simple box and complete the step from 10.c. Once again the Nameref will be “Objects” and the choice in the Data drop down menu will be “TEMPLATEID”
   3. Complete by adding another Literal and closing the brackets.
3. Re-select the Indent and add the third and final Line+Literal box. This final Literal box we’ll be adding in the method that will create our objects using the two previous variable we created in 10 and 11. In the Literal field type CreateObject(type, resref, to start the method.
   1. Next, select the Line box we just added and add in another Scope+Simple box. Change the Scope box NameRef text field to “Location”. In the Scope box make sure that the Data drop down menu has “NAME” chosen (not TEMPLATEID). We only need a simple tag from our “Location” so we don’t need the more specialized TEMPLATEID option to get the resref.
   2. Close the method with another Literal that has the end );
4. Almost done! Finally reselect the original Code box and add in the final Line+Literal box. In this Literal we’ll be closing the “For” loop with } in the Literal text field.
5. Your screen should look like this:



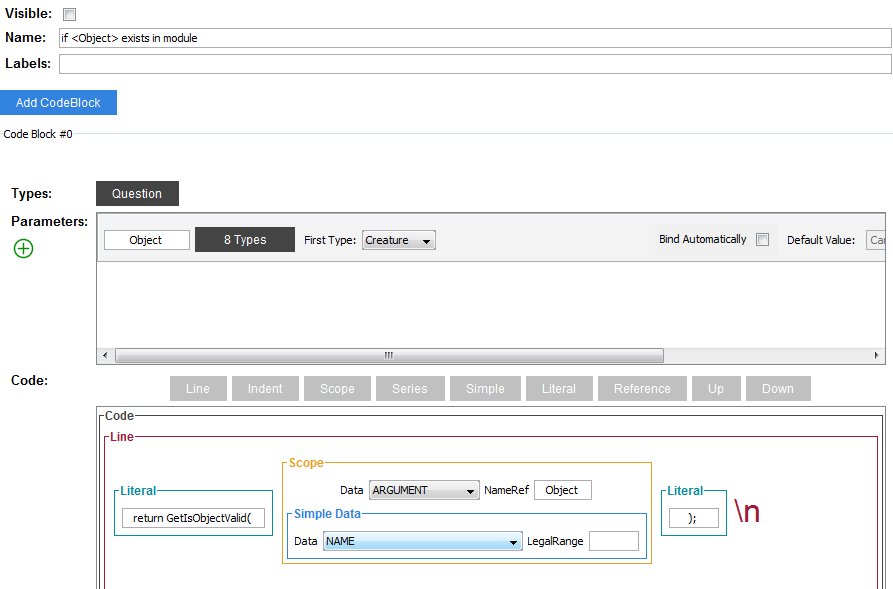
Congratulations! You’ve created an effect.

Making a Description

Before we can start making a description we need to make another effect. The reason being is the effect houses the actual code used by ScriptEase. For example the is a description that describes if, well, an object exists in the module. However when you click on the Description in the Library editor you’ll find this screen:

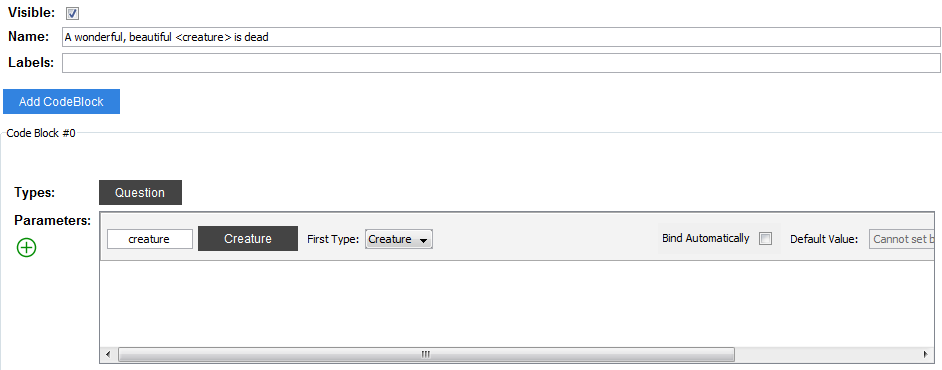


As you can see, there is no code that will give us the answers to if the object exists. So how do we evaluate this question? That’s where our Effects come in. For the most part Effects have a visible reaction in ScriptEase; Effects “Do” something whether it is creating monsters or adding in a magical animation to characters. However, in the Library editor there are a number of special effects that don’t have a visible effect, but are necessary for Descriptions to functions. These Effects (located in the Effects tab; they are greyed out and not available when creating an actual Story) are “return methods.” For example, the Effect that relates to “Object Exists describes….” description is the “If Object Exists” Effect shown below.

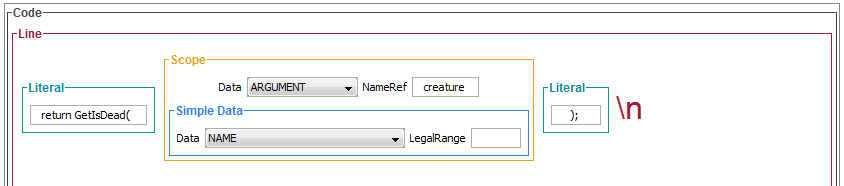


This Effect returns TRUE if the object we are specifying exists. Therefore, in order to create our own Descriptions we must first create another Effect.

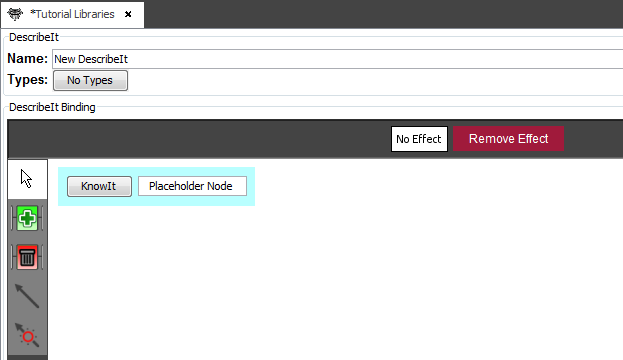
1. First create a new effect by clicking “File->New->Effect”
2. Name it “A wonderful, beautiful <creature> is dead” (Side note: there is already an “If creature is dead” Effect, but for the most part almost all the Effects needed for Descriptions have already been created)
3. In previous Effects we’ve changed the “Types:” field to “Void” since we weren’t returning a value. Instead we were performing an independent action. In this case, we’ll be changing the “Types:” field to “Question.” For this effect we will be returning a True or False value to describe if a creature is alive (False) or dead (True).
4. Next add a Parameter. Name it “creature” and change its type to Creature.



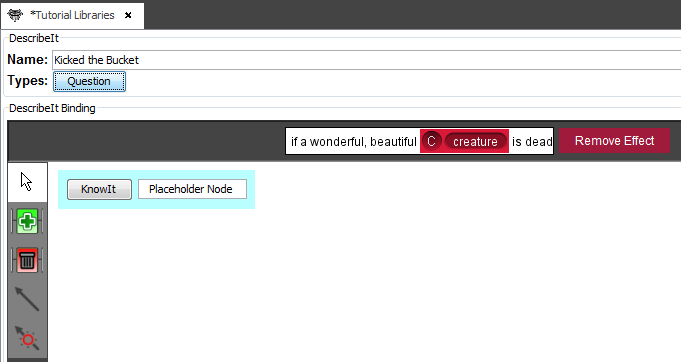
1. Selecting the Code Box, add a Line+Literal box. In the Literal text field type in the following return GetIsDead(
2. Next, selecting the Line box, add in a Scope+Simple. Change the NameRef to “creature” and keep the Simple Data drop down menu as “NAME”.
3. Close the method by adding in another Literal with );



1. Pretty simple. Now to make our description. Click “File->New->Description” which will create a new blank Description populated by a single empty “New DescribeIt” slot in the left Descriptions panel. Your new Description will start out looking like this.



1. First, change the name to “Kicked the Bucket.” We don’t use angle brackets in this name because aren’t using any of the name words as parameters.
2. Change the “Types:” field to “Question.” Whenever you make a new Description, make sure that it is the same type as the Effect you want to use with it. It will not work otherwise, the same way a method will throw an error if you try to return a string when it is expecting an integer. Since our “if a wonderful, beautiful <creature> is dead” Effect has a “Question” type, we want to make the Description match.
3. Next click on the Effects tab. This will bring up the list of Effects you might want to use, but won’t change the right Description panel. This is so you can drag the Effect you want into the Description.
   1. Click and drag the “if a wonderful, beautiful <creature> is dead” Effect from the Effects panel into the white box that says “No Effect”



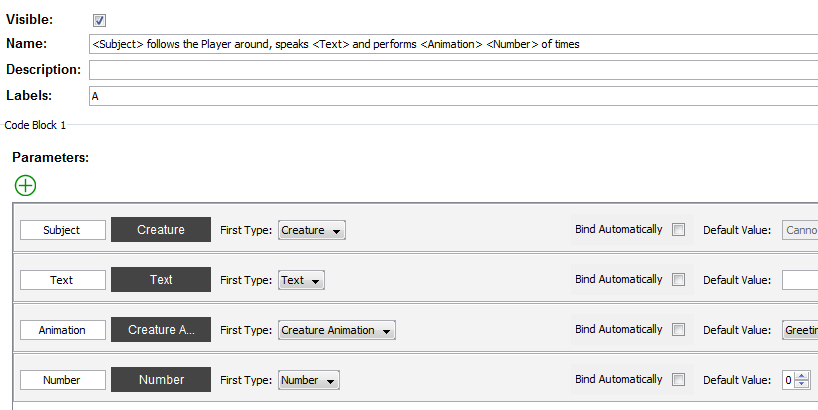
1. And viola! Your new Description is complete!

Making an Activity!

Now that you’ve made Causes, Effects and a Description, we’ll move on to making an Activity. Activities are similar to Descriptions in that they use Effects instead of hard code to perform an action. We use Activities to save time creating repetitive effects and actions in our story. For example, say we have an NPC standing guard at the gate. Everytime the PC walks past we want the NPC to perform a different animation and say a new line. Each time we wanted to change the NPC’s actions we would have to drag in a new animation effect, a new ‘say text’ effect, and potentially a new ‘distance between the pc and npc’ description. To save time dragging them into the story and populating their slots, we create Activities to reuse.

For our Activity, we want to create an Activity where an NPC will follow a Player around repLet’s get started.

1. First create a new Activity by clicking “File -> New -> Activity”.
2. Head the Activities tab and select your new Activity. In the “Name:” field type: “<Subject> follows the Player around, speaks <Text> and performs <Animation> <Number> of times”
3. Next add three parameters: “Subject”, “Text”, “Animation” and “Number”. Change their types to “Creature”, “Text”, “Creature Animation” and “Number” respectively. Your screen should look like this.



Our new Activity provides us with a new implicit statement. These implicits will be used to populate the slots of the Effects and Descriptions we will eventually be using in our Activity. 

1. Next we’ll be adding in the actions we want this Activity to perform. First we’ll start by defining the Player and the Player’s Location.
2. Drag in two Descriptions:
   1. “<The Player Character> describes The Player Character”
   2. “<Location>” describes the location of “<Object>”
3. Drag “The Player Character” description into the slot of the Location Description.
4. Now that we have our Descriptions, we’ll drag in the Effects we want. Start by finding the “<Mover><move>s to <Location>” and drag that beneath our Descriptions. Populate the Mover slot with the <Subject> implicit above. and the Location slot with our new Location implicit.
5. Next, drag the “<Speaker> speaks <Text> at <Volume> volume” and populate the Speaker slot with the Subject implicit, the Text slot with the Text implicit