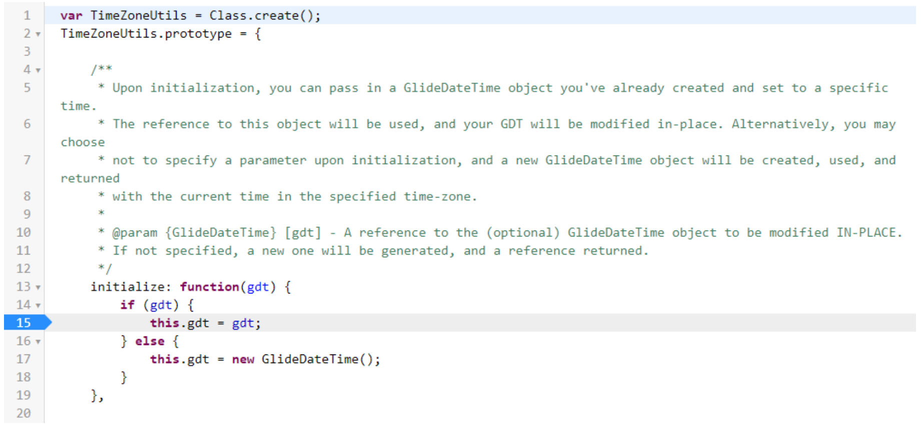
SERVER-SIDE DEBUGGING

When it comes to debugging server-side behavior, you can add breakpoints to your code using the script editor, by clicking on the line number. For example, in the below screenshot, this Script Include has a breakpoint on *line 15*:



```
1   var TimeZoneUtils = Class.create();
2   TimeZoneUtils.prototype = {
3
4       /**
5        * Upon initialization, you can pass in a GlideDateTime object you've already created and set to a specific
    time.
6        * The reference to this object will be used, and your GDT will be modified in-place. Alternatively, you may
    choose
7        * not to specify a parameter upon initialization, and a new GlideDateTime object will be created, used, and
    returned
8        * with the current time in the specified time-zone.
9        *
10       * @param {GlideDateTime} [gdt] - A reference to the (optional) GlideDateTime object to be modified IN-PLACE.
11       * If not specified, a new one will be generated, and a reference returned.
12       */
13      initialize: function(gdt) {
14          if (gdt) {
15              this.gdt = gdt;
16          } else {
17              this.gdt = new GlideDateTime();
18          }
19      },
20
```

If you open the Script Debugger (**System Diagnostics > Script Debugger**), then all scripts with breakpoints you've created will show there. While debugging, any time those scripts are executed, the transaction will pause[1] and allow you to inspect variables in the local scope and see what might be causing any aberrant behavior.

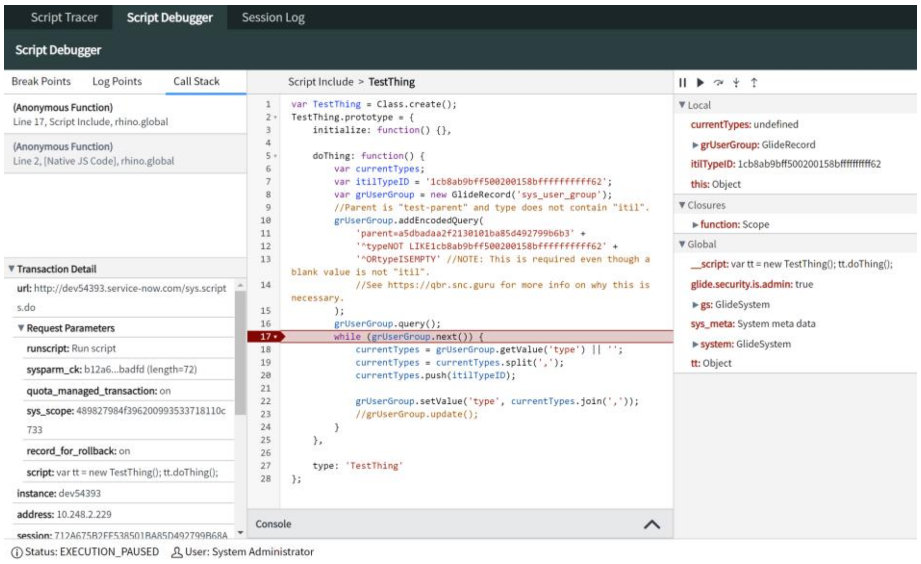
Once you have the server-side script debugger window open and at least one breakpoint added into your script, trigger any synchronous server-side script you want to debug (such as by calling a Script Include method in **Scripts – Background** or by doing some action which triggers a Business Rule you're trying to debug), and it should pause at your breakpoint. At that point, switch over to the debugger window and you should see something like this:

[1] *If you notice that the debugger isn't pausing on your breakpoint and you've already tried stopping/starting the debugger, consider that certain lines of code may not actually be "executed" in the typical way that we think of code executing. That is, the JavaScript runner is made up of a bunch of different parts, each responsible for different jobs. The lexer and scoping engine for example, don't "execute code" in the way you might expect the actual JS "Engine" to. Only code executed by the Engine can be paused on. This means that if you do something like "hoist" a variable (which is a good idea much of the time!), and* ***declare*** *it without* ***initializing*** *it to a value, the JS Engine will not pause on the line in which you've declared the variable, even if you add a breakpoint there! I know this sounds a little arcane, but consider the following code:*

```
var myName;
myName = 'Tim';
var yourName = 'Bob';
```

Fig. 2.01

A breakpoint on ***lines 2 or 3*** *would pause execution just as you'd expect, but a breakpoint on* ***line 1*** *would not.*



Script Tracer | Script Debugger | Session Log

Script Debugger

Break Points | Log Points | Call Stack

(Anonymous Function)
Line 17, Script Include, rhino.global

(Anonymous Function)
Line 2, [Native JS Code], rhino.global

▼ Transaction Detail
url: http://dev54393.service-now.com/sys.scripts.do
▼ Request Parameters
runscript: Run script
sysparm_ck: b12a6...badfd (length=72)
quota_managed_transaction: on
sys_scope: 489827984f396200993533718110c733
record_for_rollback: on
script: var tt = new TestThing(); tt.doThing();
instance: dev54393
address: 10.248.2.229
session: 712A675B2FF538501BA85D492799B68A

Script Include > TestThing

```
var TestThing = Class.create();
TestThing.prototype = {
    initialize: function() {},

    doThing: function() {
        var currentTypes;
        var itilTypeID = '1cb8ab9bff500200158bffffffffff62';
        var grUserGroup = new GlideRecord('sys_user_group');
        //Parent is "test-parent" and type does not contain "itil".
        grUserGroup.addEncodedQuery(
            'parent=a5dbadaa2f2130101ba85d492799b6b3' +
            '^typeNOT LIKE1cb8ab9bff500200158bffffffffff62' +
            '^ORtypeISEMPTY' //NOTE: This is required even though a blank value is not "itil".
            //See https://qbr.snc.guru for more info on why this is necessary.
        );
        grUserGroup.query();
        while (grUserGroup.next()) {
            currentTypes = grUserGroup.getValue('type') || '';
            currentTypes = currentTypes.split(',');
            currentTypes.push(itilTypeID);

            grUserGroup.setValue('type', currentTypes.join(','));
            //grUserGroup.update();
        }
    },

    type: 'TestThing'
};
```

Console

▼ Local
currentTypes: undefined
▶ **grUserGroup:** GlideRecord
itilTypeID: 1cb8ab9bff500200158bffffffffff62
this: Object

▼ Closures
▶ **function:** Scope

▼ Global
__script: var tt = new TestThing(); tt.doThing();
glide.security.is.admin: true
▶ **gs:** GlideSystem
sys_meta: System meta data
▶ **system:** GlideSystem
tt: Object

Status: EXECUTION_PAUSED   User: System Administrator

The line highlighted in red is the line we're currently paused on. The highlighted line has not yet executed; it's the line which *will* execute on the next "**step**" we tell the debugger to take.

If you look at the top-right of the above image, you'll see a section called **Local**, in which all of our locally defined variables are visible. This allows us to inspect current variable values as our code executes. Unfortunately, sometimes this doesn't work. Certain arrays and objects are shown as "Native JavaScript Array" or some arbitrary-sounding memory location address. As of the Paris release of ServiceNow though, we can get around this using the **Console** at the bottom of the debugger, which we'll discuss in more detail below.

Just above the **Local** section, you'll see five buttons that allow you to control the debugger, step through your code, and step into or out of function calls.

The function of these buttons may not be obvious when first glancing at their icons, so let's talk about what each of them does. Going in order from left to right, these are their functions:

1. **Stop Debugging**: This button, which looks like a pause icon, will stop the debugger entirely. Any code which is currently paused, will **resume**, and no further debugging will happen (even if your code hits another breakpoint) until you click the button again, to re-start the debugger.
2. **Resume Execution**: This button, which looks like a 'play' icon, will not halt debugging entirely, but will resume unmanaged execution of your code. Unlike the **Stop Debugging** button though, this button **will** pause execution at the *next* breakpoint it hits. For example, if you have a breakpoint *inside* a `while` loop, execution will pause the first time you loop through that code. You can "step" through the code from there and follow its execution, or you can just press the **Resume Execution** button, and your code will resume. However, the *next* time the loop runs, your code will pause again, and control will be given back to you via the debugger, just like the first time through.
3. **Step Over Next Function Call**: This is the button you're likely to click the most. It basically says, "execute the current highlighted line (highlighted in red) and pause execution on the next line". This is very similar to the **Step Into Next Function Call** button, except that it will stay at the current "layer" of the call-stack. This means that if the line to be executed is a function-call, unlike the "Step Into" button, this "Step Over" button will not step *into* the called function, and pause execution *inside* the called function, but will instead execute the called function in its entirety (without debugging *that* function line-by-line) and pause on the next line of the *current* function. Hence the name: Step ***Over*** rather than Step ***Into***.

4. **Step Into Next Function Call**: As alluded to above, rather than staying at the current "layer" of the call stack, if possible, this button will step ***into*** the function being called on the highlighted line of code being debugged. If the highlighted line of code executes an outside Script Include, another method within an existing Script Include, or any other type of script that you have access to, the debugger will switch to that script, adding it to the **Call Stack** section in the panel on the left of the debugger window, and will allow you to debug execution of *that* script line-by-line, even if you didn't have a breakpoint in that called function. Note that I said, "any type of script *that you have access to*". There are some scripts you won't be able to step into; black-box code, such as any call to the ServiceNow GlideSystem API (for example, `gs.nowDateTime()`). In cases like this, pressing the **Step Into** button won't do any harm, and will simply behave as though you clicked the **Step Over** button.
5. **Step Out of Current Function**: This button is the exact opposite of the **Step Into** button. If you look at the **Call Stack** section in the panel on the left side of the debugger, you can see what scope is just "above" your current "layer" of the call stack. This makes it easy to **Step Into** a function call, have a look at how the first few lines are executing, then easily return to the function you were debugging previously (the one which called this other function), by clicking the **Step Out** button.

The Script Debugger is an exceptionally useful, and largely under-utilized tool for exploring the behavior of your server-side code. I ***strongly*** recommend becoming extremely familiar with this tool, its capabilities and limitations, and how to use it to effectively debug your own code during a rubber duck debugging[2] exercise.

Pro-tip: *Only* ***synchronous*** *operations can be debugged since the debugger has to pause the thread as it executes in order to debug it. This means that certain things like inbound API calls cannot be debugged.*

*That said, for REST APIs in particular, you can trigger an API call using the Rest API Explorer (**System Web Services > REST > REST API Explorer**) to run it synchronously and debug your Scripted REST APIs (SRAPIs).*

DEBUGGER CONSOLE

In addition to letting you see the exact values of all of your local variables, monitor execution line-by-line, step into and out of other functions, and generally become a wizard of debugging, the debugger has one more trick up its sleeve. As of the **Paris** release of ServiceNow, the debugger now has a **Console**.

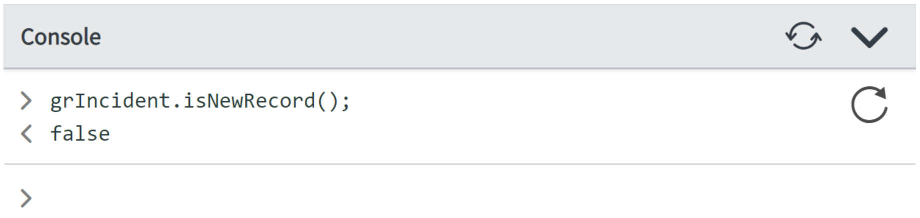
You may have seen the Console section at the bottom of some of the screenshots in the previous section. Collapsed by default, it gives you just what you'd expect: a console into which you can write code. This can be monumentally useful for things like logging *actual* values, including (a) variables which don't show up meaningfully in the **Local** panel at the top right of the debugger, and (b) values not stored in discrete variables, such as the value returned by `grIncident.getRowCount()`.

To get a value to print out in the console, you don't use `console.log()` like in a browser console, and you don't use `gs.info()` or `gs.log()` like in a background script. Instead, you simply type in some code, and whatever the last line of code[3] evaluates to, will be printed in the console. For example, if I'm debugging a Business Rule, and I simply type in the console `current.isNewRecord();`, either `true` or `false` will be printed to the console (depending on whether the Business Rule is executing on a new or existing record, obviously).

[2] *"**Rubber Ducking**", also known as "**Rubber Duck Debugging**" is the practice of explaining, out loud, exactly what you expect your code to be doing, literally line-by-line, to an inanimate object, such as a rubber duck.*

See rubberduckdebugging.com for more info.

[3] *If you write a line of code and press Enter in the debugger console, that line will execute. You can write multiple lines of code – even an entire self-executing function – by instead using SHIFT+Enter to create new lines, then pressing Enter when you're ready to execute the whole thing. Just remember that whatever the **last line of code evaluates to**, is what will be printed in the console.*

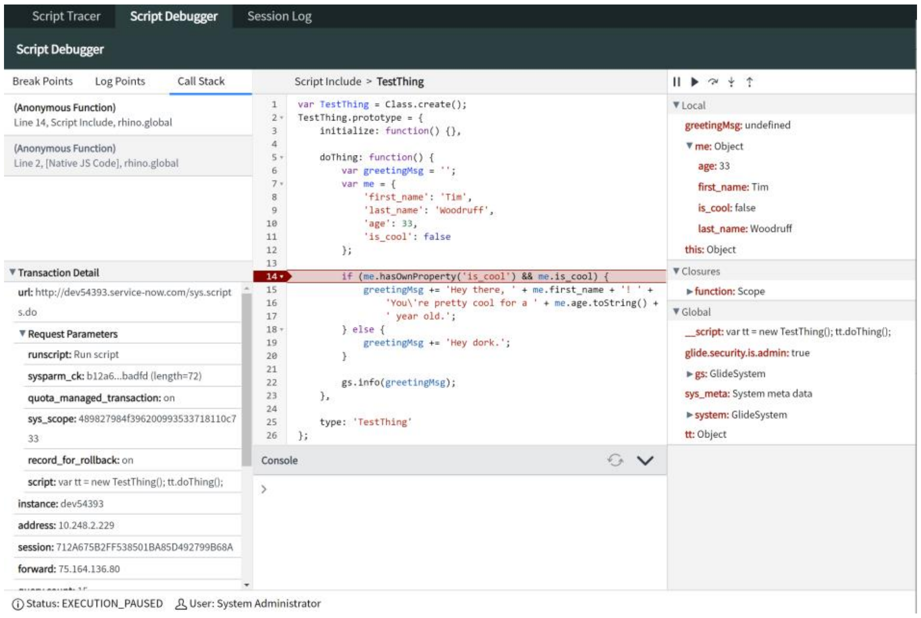


```
> grIncident.isNewRecord();
< false

>
```

Example of running a line of code in the console, and the result being evaluated and logged below.

That's pretty rad, but the console can do much more than just log stuff for us. Consider the following screenshot:



(Anonymous Function)
Line 14, Script Include, rhino.global

(Anonymous Function)
Line 2, [Native JS Code], rhino.global

▼ Transaction Detail
url: http://dev54393.service-now.com/sys.scripts.do
▼ Request Parameters
runscript: Run script
sysparm_ck: b12a6...badfd (length=72)
quota_managed_transaction: on
sys_scope: 489827984f396200993533718110c733
record_for_rollback: on
script: var tt = new TestThing(); tt.doThing();
instance: dev54393
address: 10.248.2.229
session: 712A675B2FF538501BA85D492799B68A
forward: 75.164.136.80

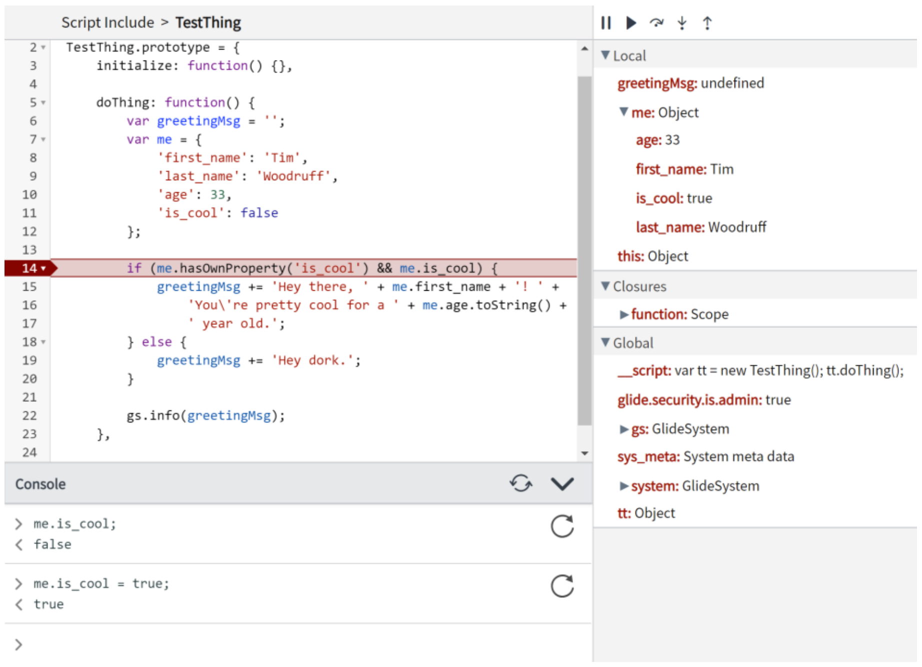
```
1   var TestThing = Class.create();
2   TestThing.prototype = {
3       initialize: function() {},
4
5       doThing: function() {
6           var greetingMsg = '';
7           var me = {
8               'first_name': 'Tim',
9               'last_name': 'Woodruff',
10              'age': 33,
11              'is_cool': false
12          };
13
14          if (me.hasOwnProperty('is_cool') && me.is_cool) {
15              greetingMsg += 'Hey there, ' + me.first_name + '! ' +
16                  'You\'re pretty cool for a ' + me.age.toString() +
17                  ' year old.';
18          } else {
19              greetingMsg += 'Hey dork.';
20          }
21
22          gs.info(greetingMsg);
23      },
24
25      type: 'TestThing'
26  };
```

▼ Local
greetingMsg: undefined
▼ me: Object
age: 33
first_name: Tim
is_cool: false
last_name: Woodruff
this: Object
▼ Closures
▶ function: Scope
▼ Global
__script: var tt = new TestThing(); tt.doThing();
glide.security.is.admin: true
▶ gs: GlideSystem
sys_meta: System meta data
▶ system: GlideSystem
tt: Object

Status: EXECUTION_PAUSED   User: System Administrator

This script is paused on line 14, and you can see that the variable `me` has already been declared and initialized, with the property `is_cool` set to `false`. Therefore, if we were to continue execution, we would see that the `else` block fires, as the condition in the `if` block is not met.

That is obviously unjust, untrue, and cruel beyond measure. Tim Woodruff is *way cool*! Therefore, as the debugger is paused on the line on which the condition in the if block is to be evaluated, we'll use the debugger to modify the value of `me.is_cool` so that it properly reflects Tim's totally tubular coolness.



```
2   TestThing.prototype = {
3       initialize: function() {},
4
5       doThing: function() {
6           var greetingMsg = '';
7           var me = {
8               'first_name': 'Tim',
9               'last_name': 'Woodruff',
10              'age': 33,
11              'is_cool': false
12          };
13
14          if (me.hasOwnProperty('is_cool') && me.is_cool) {
15              greetingMsg += 'Hey there, ' + me.first_name + '! ' +
16                  'You\'re pretty cool for a ' + me.age.toString() +
17                  ' year old.';
18          } else {
19              greetingMsg += 'Hey dork.';
20          }
21
22          gs.info(greetingMsg);
23      },
24
```

```
> me.is_cool;
< false

> me.is_cool = true;
< true

>
```

```
▼ Local
  greetingMsg: undefined
  ▼ me: Object
      age: 33
      first_name: Tim
      is_cool: true
      last_name: Woodruff
  this: Object
▼ Closures
  ▶ function: Scope
▼ Global
  __script: var tt = new TestThing(); tt.doThing();
  glide.security.is.admin: true
  ▶ gs: GlideSystem
  sys_meta: System meta data
  ▶ system: GlideSystem
  tt: Object
```

As you can see from the screenshot above, in the **Local** panel at the top-right of the debugger, the `is_cool` property of the `me` object is now – justly[4] – set to `true`[5], and all is right with the world again.

But wait! Our code is still paused! The condition on line 14 has not yet been evaluated! Is it possible that the fiddling we did in the console will actually change how our code runs? **Indeed, it is!**

Of course, any changes you make through the console **only apply for this specific debugging instance**, and won't change how your code will execute in the future. However, this functionality does give you the ability to test out various changes **mid-execution**, *while* your code is running, and see how it behaves! That makes this one of the **most powerful tools in your entire debugging arsenal**!

If you've never taken the time to get familiar with it before, let today be the day! Go, now! Put this book down and go experiment with the Script Debugger!

[4] If you doubt the veracity of this statement, then allow me to present as evidence, the fact that Tim once wore mirrored sunglasses to class every day for like 3 months in middle school.

Checkmate, chum.

*[5] If a variable's value is modified through the console, it won't actually immediately show the new value of the variable in the **Local** panel of the debugger until the panel is updated by, for example, stepping into the next line of code. However, you can also simply collapse and re-expand any object definition in the **Local** panel or the **Function** section in the **Closures** panel in order to see the updated values without having to step into the next line of code.*