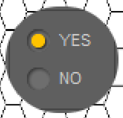
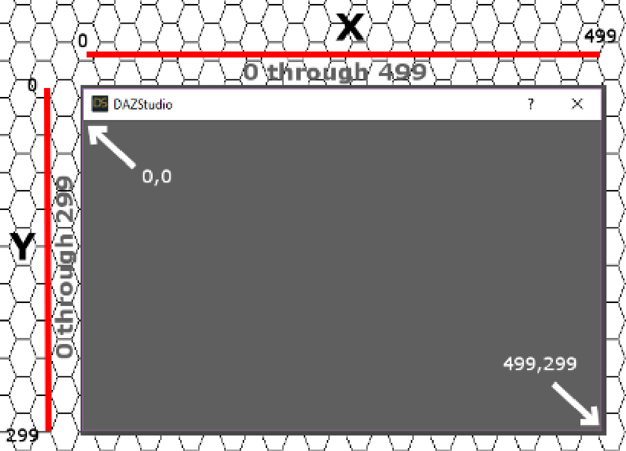
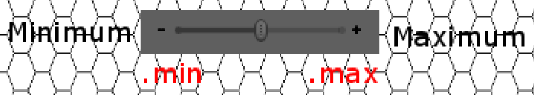
SCRIPTING

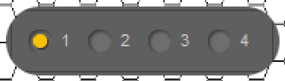
Made Simple

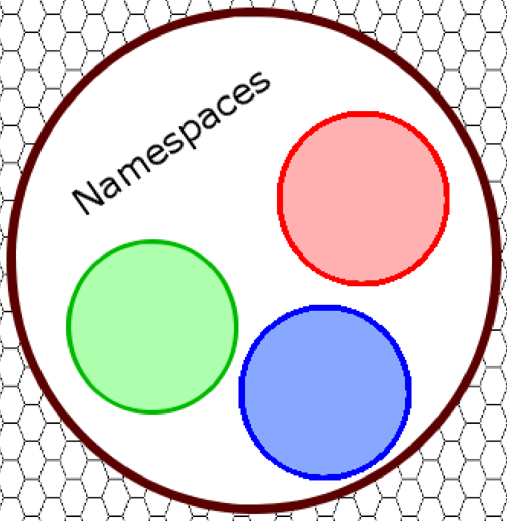
Breaking News
Current EVENTS











Program Daz Studio with Your Own Code

Input and Output Controls

SCRIPTING MADE SIMPLE
Volume 5: Input and Output Controls

TABLE OF CONTENTS

Required Software

To use DAZ Scripting, you must first install the DAZ Studio application. For the purposes of this tutorial, we used version 4.10. The latest version is available free from DAZ 3D using the link below.

Get DAZ Studio
{https://www.daz3d.com/get_studio}

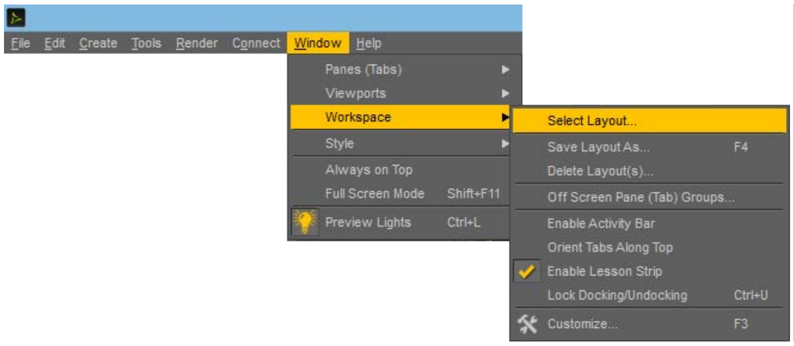
NOTE:
The following **Preparation** and **Review** sections are abbreviated overviews of material covered in: SMS Volumes 1, 2, 3, and 4.

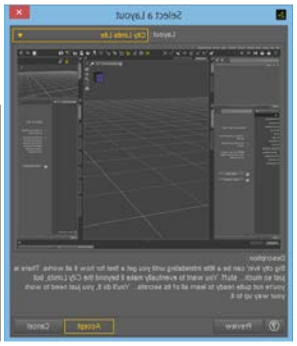
Preparation

Layout and Style

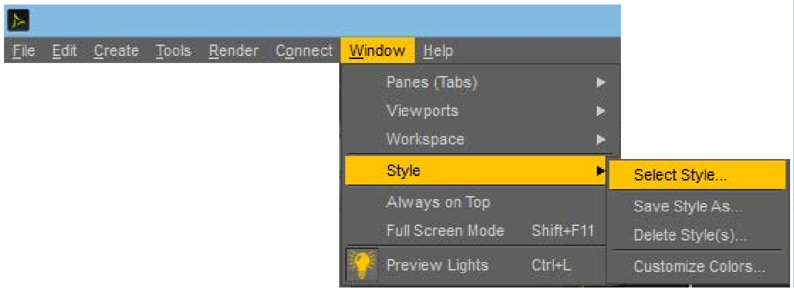
For this tutorial, we will use Layout "**City Limits Lite**" and Style "**Darkside**".

To change your layout, on menu select **Window / Workspace / Select Layout**. In popup window, select **City Limits Lite** in Layout field and click Accept button.





To change your style, on menu select **Window / Style / Select Style**. In popup window, select **Darkside** in Style field and click the Accept button.



Script IDE Pane(tab)

You will be using the built-in Script IDE pane in DAZ Studio for the scripts we will be writing. If you do not see the Script IDE pane or tab anywhere on your workspace, you can open it up from the main menu using **Window / Panes (Tabs) / Script IDE**.

Review

Topics covered in previous volumes of the Scripting Made Simple series for DS:

Volume 1: Intro to Daz Script

Case Sensitivity, End Of Line, Scripts Folder
Commenting Your Code, Naming Conventions, Common Prefixes

Volume 2: Mathematics and Looping

Mathematical Symbols, Assignments, Strings, Operations
Fractions, Combining Strings, Incrementing and Decrementing
Combining Operation with Assignment, Precedence of Operators
Grouping with Parenthesis, Comparing Values, Blocking Code
Conditional Statements, If - Else, Switch - Case,
Looping Statements, For, While, Do - While, Parsing Errors

Volume 3: Strings, Dates and More

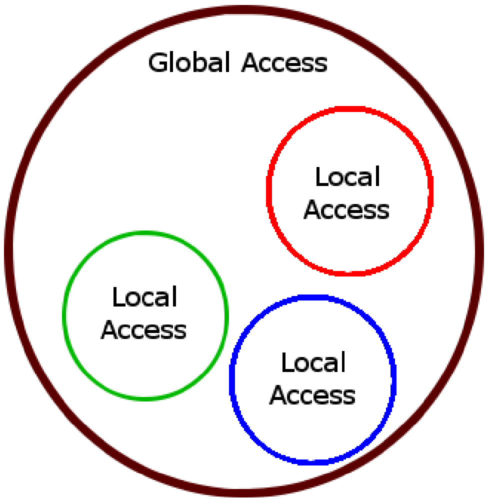
Whole Numbers, Positive and Negative, Integers, Real Numbers
Math, Expressions, Absolute Value, Ceiling of, Floor it,
Power of, Square Root, Round off, Random Number
Minimum of, Maximum of, “E”, Exponent, Logarithm
Strings, Character at, Trim off, Lower Case, Upper Case
Left-most/Right-most Characters, Middle Characters
Empty String, ASCII, Char to Code, Code to Char
Dates, Current Date, Universal Time, New Objects, Date to String
Year, Month, Day of Month, Day of Week, Hour, Minute

Volume 4: Arrays and Creating Functions

Customize IDE, Preferences, Current Line, Indents and Tabs
Paragraph Markers, Minimizing Actions, Debug/Output Panel, Log File
Arrays, Declaring Array, Element Naming, Determine Array
Counting Elements, Length vs. Indexing, Undefined Array, Defining Elements
Advanced Concepts, Nesting, Labels, User-Defined Functions, Basic Types
Parts of Function, Designing Functions, Function Type 1, Function Type 2
Function Type 3, Method toString, Function Type 4, Method forEach

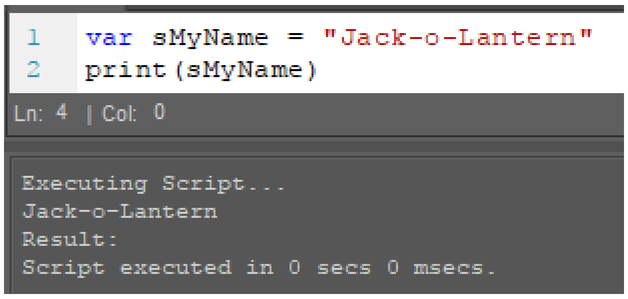
Namespace (Scope)

Now is a good time to introduce the concept of namespaces (or scope of use) for items in your script. The scope of an item is basically where it can be called or referred to within your scripting code, and is usually determined by "where" it is declared in your code. The two scopes that can be applied to variables and objects are the global namespace and any local namespaces created in your code.



Global Namespace

Items in the global namespace can be used anywhere in your script. These items are globally accessible meaning that they can be accessed and/or changed from anywhere within your script. There is only one global namespace and that is your whole application. Items declared at the beginning of your script can be used anywhere, so they are considered to be in the global namespace.



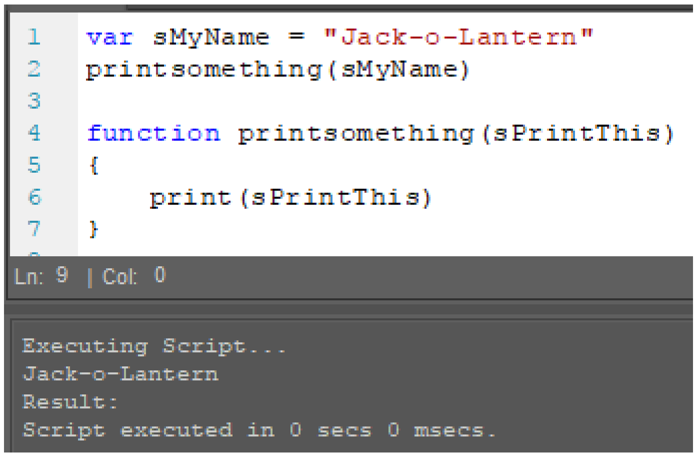
```
var sMyName = "Jack-o-Lantern"
print(sMyName)
```

Ln: 4 | Col: 0

```
Executing Script...
Jack-o-Lantern
Result:
Script executed in 0 secs 0 msecs.
```

Local Namespace

Items declared in a local namespace such as a function can only be called or referred to within that functions code block. The following example looks simple enough. Even though it is not obvious, the variable ***sPrintThis*** is a local variable for the function ***printsomething*** and cannot be used outside of that block.



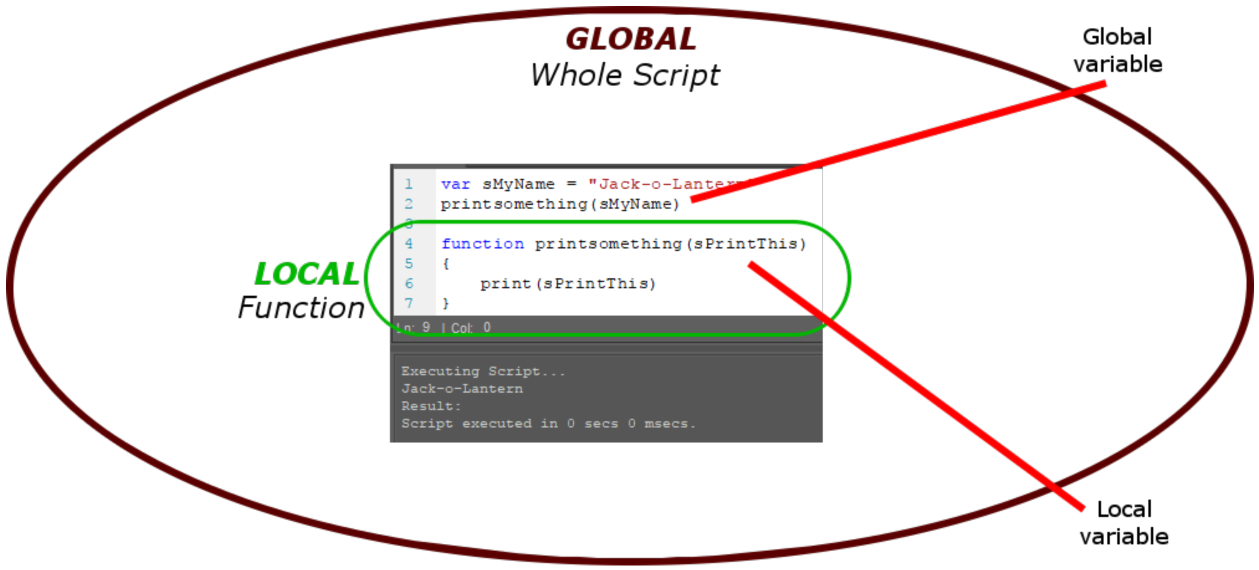
```
var sMyName = "Jack-o-Lantern"
printsomething(sMyName)

function printsomething(sPrintThis)
{
    print(sPrintThis)
}
```

```
Ln: 9  | Col: 0
```

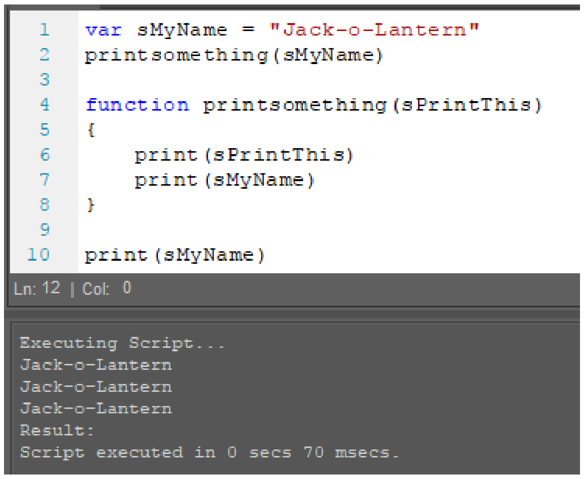
```
Executing Script...
Jack-o-Lantern
Result:
Script executed in 0 secs 0 msecs.
```

Global vs. Local

Sure that all sounds good, but perhaps it still doesn't make sense. Let's discuss the issue by demonstrating with variables. Remember, ***global variables*** are declared at the beginning of your script, or outside any blocks of code, while ***local variables*** are declared within a block of code like in the function as shown below.



The concept sounds easy enough, but how can you prove that these ***namespace*** boundaries really exist? By coding, testing, and debugging of course! Let's look at that code again and test to see if the variable **sMyName** is truly in the global namespace by printing it inside and outside of the function's code block.

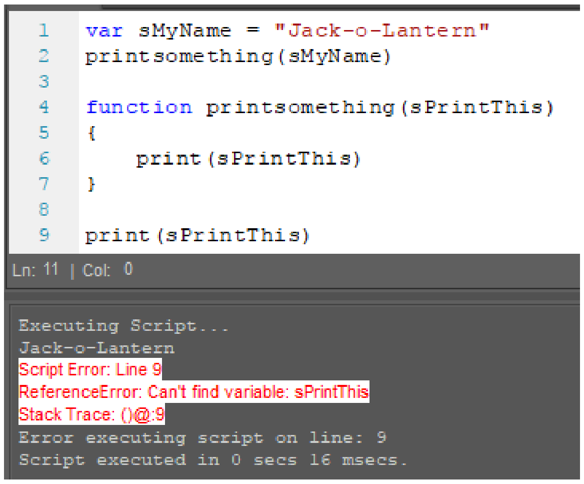


```
var sMyName = "Jack-o-Lantern"
printsomething(sMyName)

function printsomething(sPrintThis)
{
    print(sPrintThis)
    print(sMyName)
}

print(sMyName)
```

Ln: 12 | Col: 0

```
Executing Script...
Jack-o-Lantern
Jack-o-Lantern
Jack-o-Lantern
Result:
Script executed in 0 secs 70 msecs.
```

SUCCESS! The data contained in the variable **sMyName** was able to be passed to the function for printing, and it was also directly printed both inside and outside the function's code block meaning that it is in the global namespace. That may have been quite obvious, so let's try to test the variable **sPrintThis** and see if it accessible outside of the function's code block.



```
1   var sMyName = "Jack-o-Lantern"
2   printsomething(sMyName)
3
4   function printsomething(sPrintThis)
5   {
6       print(sPrintThis)
7   }
8
9   print(sPrintThis)
```

Ln: 11 | Col: 0

```
Executing Script...
Jack-o-Lantern
Script Error: Line 9
ReferenceError: Can't find variable: sPrintThis
Stack Trace: ()@:9
Error executing script on line: 9
Script executed in 0 secs 16 msecs.
```

As you can see, we received an error for line 9 where we tried to access the variable **sPrintThis** outside of the block of code for the function where it was declared. That was because it was declared by and is local to that function.

Behind the Scenes

Practically everyone today uses a graphics based operating system (OS), whether it be Microsoft's Windows, the Apple Mac's OSX, or a Linux GUI like Gnome or KDE. What most computer users do not realize is that the OS is constantly monitoring the computer system itself along with all of the attached peripherals like keyboard, mouse, network, and touch screen monitors (just to mention a few).

Events

So exactly how does an app know when you have clicked somewhere on the screen with your mouse, or pressed some key on the keyboard? Well, it's that constant monitoring we mentioned above. When the OS detects that something has occurred on one of the many devices that it is watching, it generates what is called an "event". Applications developed with most languages, including Daz Script, can be coded to respond to certain events when they occur.

As you further your level of experience and develop your skill set, you need to have a basic understanding of how the OS and Applications interact. When the OS or an application is waiting for an event to occur, this is often called "*listening*" for an event. When an application is executing some task because an event has occurred, this is called "*handling*" the event, and the function or method that is called is referred to as an event handler.

Input / Output

Many of the Daz Scripts that you will write can perform as designed without any interaction from the user. However, there are times when a script either needs to tell the user about something which is called "***output***", or get more information from the user which is called "***input***". You will be able to code your input and output needs using a variety of objects and methods available in Daz Script.

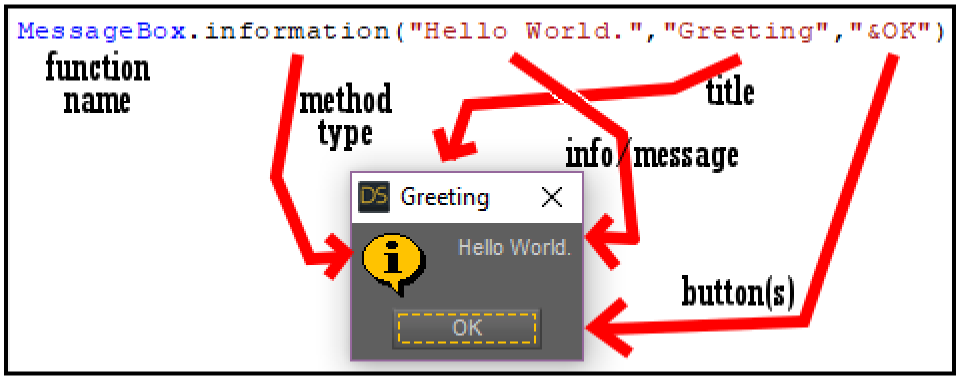
We will cover how to get input and display output in a variety of ways in the upcoming sections on the MessageBox and Widgets. We will demonstrate how to write your scripting code with and without the use of system events.

MessageBox

The MessageBox object can be used by your scripts to interact with the user by providing information that is important for the user to know, or to get simple information from the user for your script to use. Because of how this object works, it can be considered both an input and output control for your scripts. Here is the basic formatting of the MessageBox object.

MessageBox.**type**(info,title,buttons)

The main components are the object name "MessageBox" followed by the method (which in this case is the *type* of message). The parameters you provide are the information to display (*info*), a name or title for the message window (*title*), and one or more button names (*buttons*) as required by your scripting code. Below is the classic "Hello World" window in the MessageBox format with all of the components labeled.



```
MessageBox.information("Hello World.","Greeting","&OK")
```

There are four types of message boxes as shown below available for your scripting needs which will automatically display an icon within the popup window. The type you choose to use is completely up to your requirement.

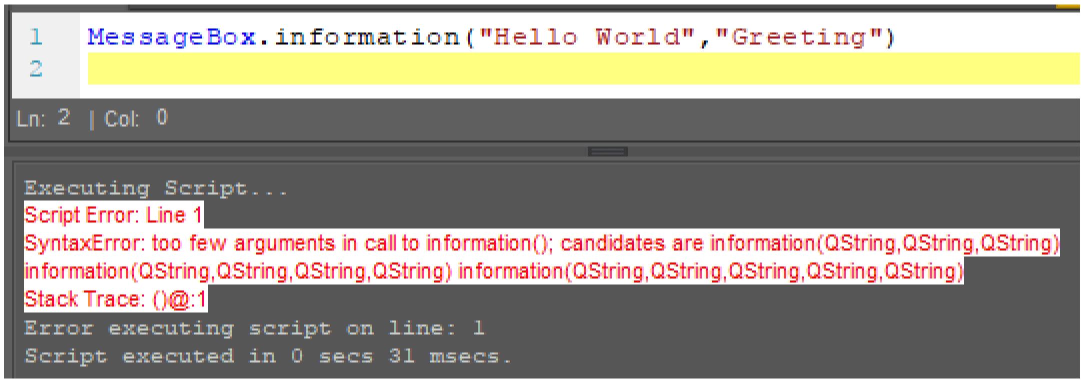
ⓘ	information	⊗	critical
ⓘ	question	⚠	warning

Buttons

While each of the four types of MessageBox can have up to three (3) buttons, by nature of their designated type each method has a required minimum number of buttons as shown below.

information	= **1**	critical	= **1**
question	= **2**	warning	= **2**

Why do you think that you are not allowed to use the MessageBox object without any buttons? Because Daz Studio would have no way of knowing the user is finished with the popup and should return to running the script. In this example, we demonstrate what will happen without any buttons.



```
MessageBox.information("Hello World","Greeting")

Ln: 2 | Col: 0

Executing Script...
Script Error: Line 1
SyntaxError: too few arguments in call to information(); candidates are information(QString,QString,QString)
information(QString,QString,QString,QString) information(QString,QString,QString,QString,QString)
Stack Trace: ()@:1
Error executing script on line: 1
Script executed in 0 secs 31 msecs.
```

It just makes sense that *information* and *critical* would only need one button to acknowledge the message, whereas *question* and *warning* would require at least two buttons for responses like “Yes” and “No”, or perhaps “Continue” and “Exit”.

When buttons are used for any method, Daz Script uses zero-based indexing. It is common practice to name any variables to capture button responses accordingly.

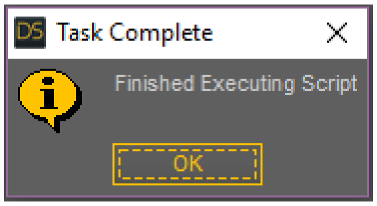
button0	**button1**	**button2**
0	*1*	*2*
	index	

Note that even though the button text is within quotes it often begins with the ampersand (&) sign before the character you want to be the keyboard equivalent of clicking the button. For example, you would use “&OK” if you want the button to display OK and use Alt-O as the keyboard shortcut.

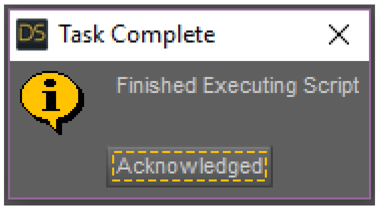
Information method

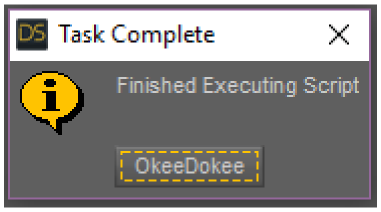
The simplest form of the MessageBox object is using the ***information*** method to give the user an update and wait until they click an OK button to continue. This method does not have a need to capture any of the button data because it is usually used just for an acknowledgement of the message by the user. For instance, let's say that you want to notify the user that your script has finished executing.

MessageBox.**information**("*Finished Executing Script*","*Task Complete*","&OK")



You can use any text for MessageBox buttons that suits your liking or needs.

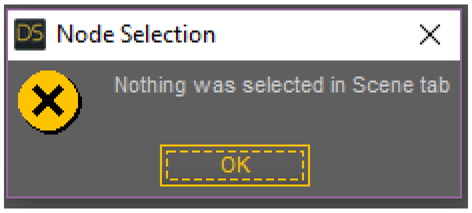




Critical method

The ***critical*** method also does not need to capture any data from the user, but is used when the message is very important to what is happening in Daz Studio. For instance, let's say that your script is expecting the user to select something in the Scene tab before executing the script. If your script determines that nothing is selected, it can notify the user before exiting.

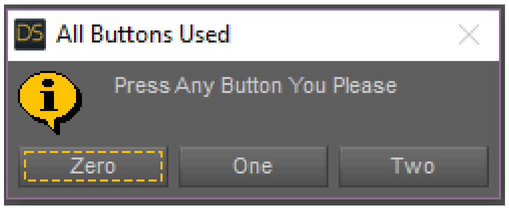
MessageBox.**critical**("*Nothing was selected in Scene tab*","*Node Selection*","&OK")



Button Pressed

Up to now, we have not captured any return values from the MessageBox methods. If we had, the value would have always been zero (0) for the examples that we used. This is because the buttons have assigned values based upon the zero-indexing principle, and we only defined one button for the ***information*** and ***critical*** methods. Please note that we could have used up to three buttons if we really needed to. If you use more than one button, you must separate each by the comma (,) and encapsulate the text for each in quotes.

MessageBox.**information**("*Press Any Button You Please*","*All Buttons Used*","**&Zero**","**&One**","**&Two**")



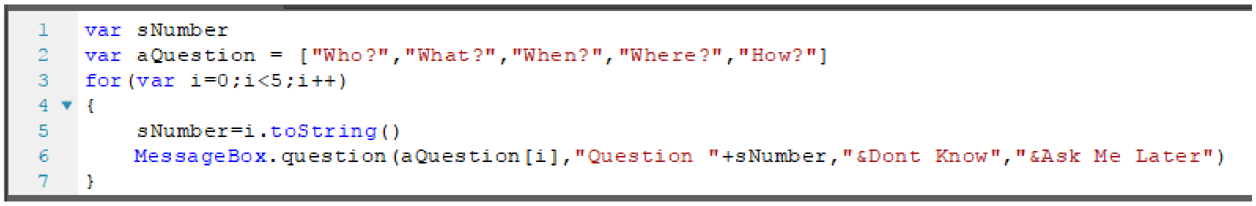
All Buttons Used

Press Any Button You Please

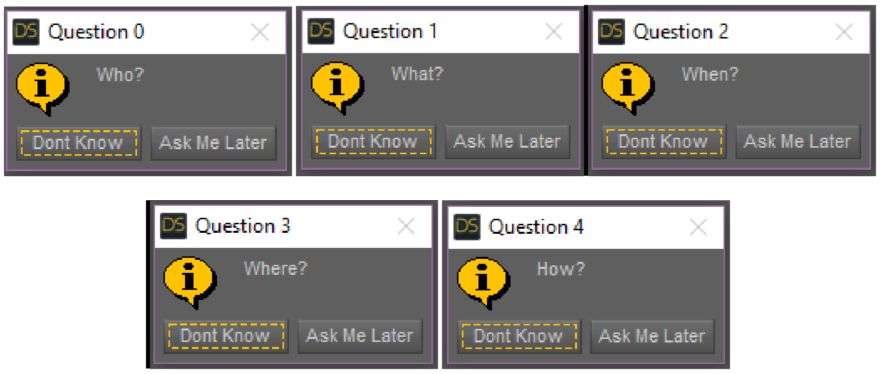
Passing Strings

You should note that when using the MessageBox object, you can pass string variables containing text to the methods for various parts of the popup. Here is the formatting along with a working example.

MessageBox.**method**(*String1*,*String2*,*String3*,*String4*,*String5*)



```
var sNumber
var aQuestion = ["Who?","What?","When?","Where?","How?"]
for(var i=0;i<5;i++)
{
    sNumber=i.toString()
    MessageBox.question(aQuestion[i],"Question "+sNumber,"&Dont Know","&Ask Me Later")
}
```



Question 0 — Who?

Question 1 — What?

Question 2 — When?

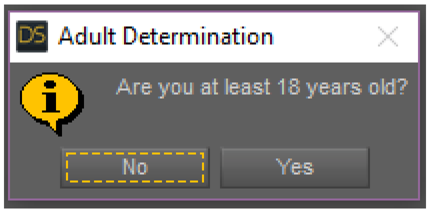
Question 3 — Where?

Question 4 — How?

Question method

The ***question*** method is nice for prompting the user to answer questions with two parts like Yes/No or True/False. Because it displays the question mark icon, the user already knows it is asking a question. For this method, you must use at least two buttons.

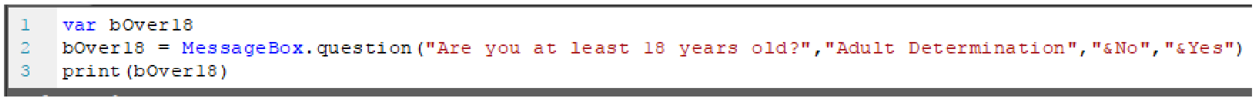
MessageBox.**question**("*Are you at least 18 years old?*","*Adult Determination*","&No","&Yes")



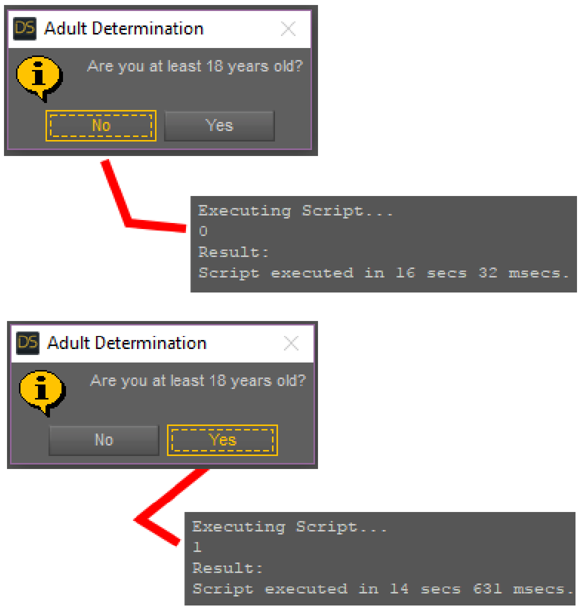
Adult Determination

Are you at least 18 years old?

But wait! We haven't captured any return values. And exactly how do you think we will do that? Easy! Just assign the MessageBox object to a variable to capture the index number of the button pressed. Of course you will need to write your script to match the text you used for each button name.



```
var bOver18
bOver18 = MessageBox.question("Are you at least 18 years old?","Adult Determination","&No","&Yes")
print(bOver18)
```



Adult Determination

Are you at least 18 years old?

```
Executing Script...
0
Result:
Script executed in 16 secs 32 msecs.
```

Adult Determination

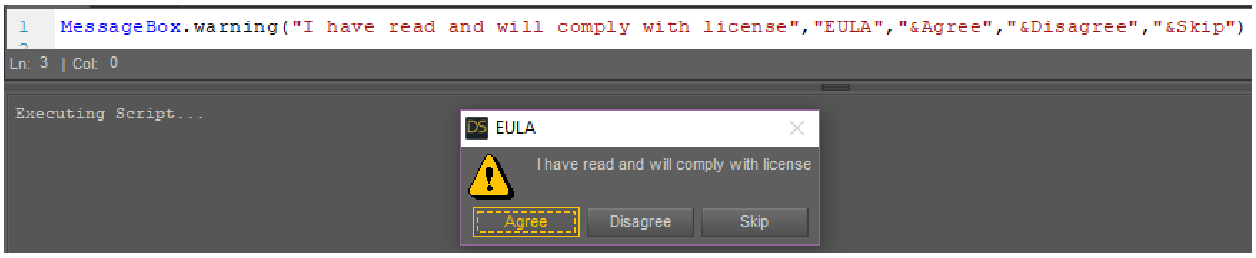
Are you at least 18 years old?

```
Executing Script...
1
Result:
Script executed in 14 secs 631 msecs.
```

As you can see, we received a "0" and a "1" for the two different button clicks.

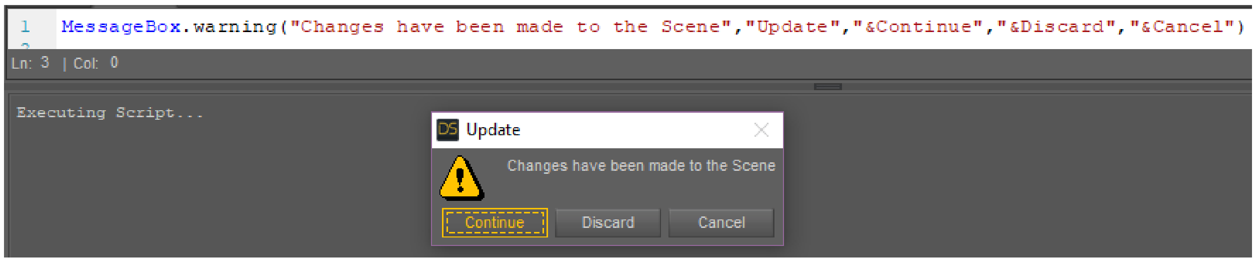
Warning method

Even though it is possible to use each of the methods in any way desired, the ***warning*** method is used to get the users attention for very important issues. Two examples are shown below; 1) you have incorporated a license agreement of some sort into your script, an 2) your script has made changes to the scene that can be undone.



```
MessageBox.warning("I have read and will comply with license","EULA","&Agree","&Disagree","&Skip")
```

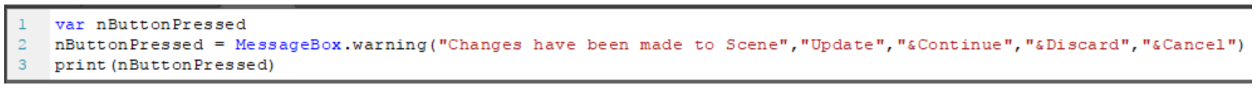
```
Ln: 3 | Col: 0
Executing Script...
```



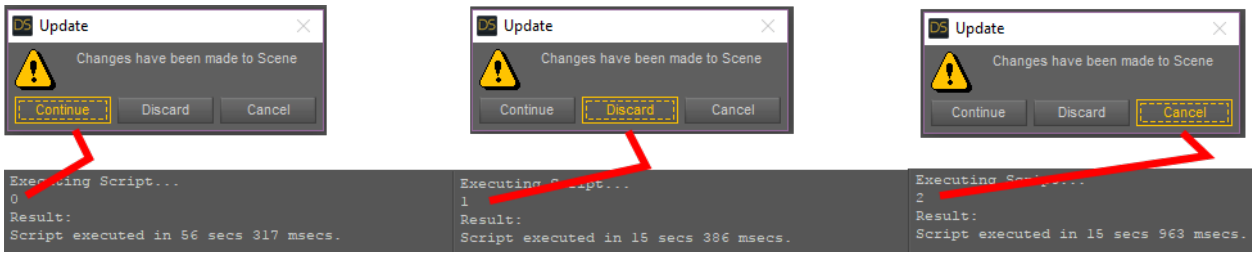
```
MessageBox.warning("Changes have been made to the Scene","Update","&Continue","&Discard","&Cancel")
```

```
Ln: 3 | Col: 0
Executing Script...
```

Again, to use the button responses from the user, simply assign the MessageBox object to a variable and use a conditional or case selection to determine what to do.



```
var nButtonPressed
nButtonPressed = MessageBox.warning("Changes have been made to Scene","Update","&Continue","&Discard","&Cancel")
print(nButtonPressed)
```



```
Executing Script...
0
Result:
Script executed in 56 secs 317 msecs.
```

```
Executing Script...
1
Result:
Script executed in 15 secs 386 msecs.
```

```
Executing Script...
2
Result:
Script executed in 15 secs 963 msecs.
```

Widgets

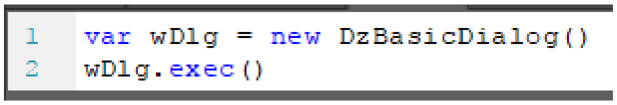
Introduction

In Daz Script, widgets are usually graphical components used to interact with the user. These include such items as the popup window, text labels, clickable buttons, moveable sliders, and more. The basic naming convention for widget items is to prefix the given name with the "w" character. For instance, if we create a widget for a cancel button, we would declare it as something like "**w**CancelButton".

Dialogs

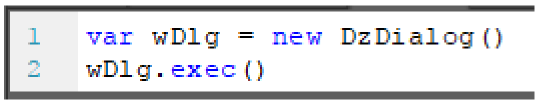
You can think of a "**dialog**" as a popup window to interact with the user to both give information and receive inputs. There are two kinds of dialogs that you have to choose from for use in your scripts; ***DzBasicDialog*** and ***DzDialog***. The one you choose will depend on what your use for the dialog will be. Both dialogs are objects, so when assigned to a variable using "*new*" that variable will inherit methods and properties from the object. Once declared, the dialog (popup) remains dormant until it is executed to interact with the user using the "*exec*" method. In almost every case, you will be adding more widgets to the dialog before it is executed (more about this later).

var *xxxxx* = new **DzBasicDialog**()



```
var wDlg = new DzBasicDialog()
wDlg.exec()
```

var *xxxxx* = new **DzDialog**()

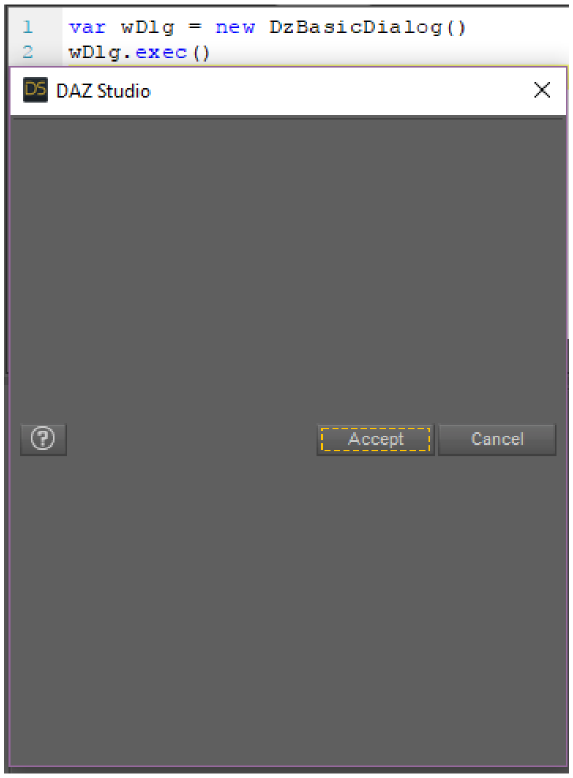


```
var wDlg = new DzDialog()
wDlg.exec()
```

You can use whatever naming convention suits your project workflow; however, it is common practice to use "**wDlg**" as the variable name for dialog widgets.

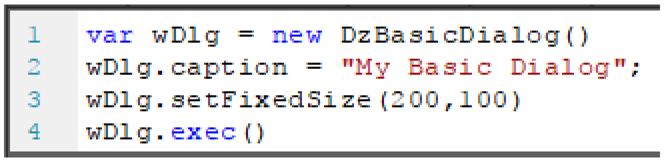
DzBasicDialog

You will use this dialog when you need some simple interaction with the user of your script. It comes with two default buttons and here is how it looks by default.

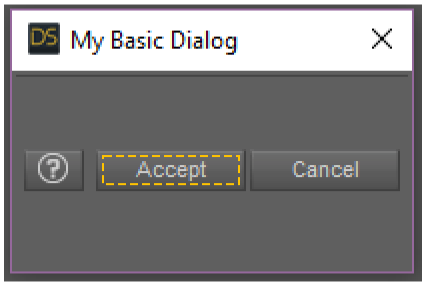


```
var wDlg = new DzBasicDialog()
wDlg.exec()
```

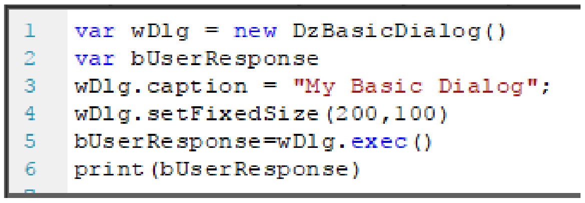
It looks a bit awkward, so let's make some quick changes. You can add a title with the "**caption**" method, and you can size it with the "**setFixedSize**" method.



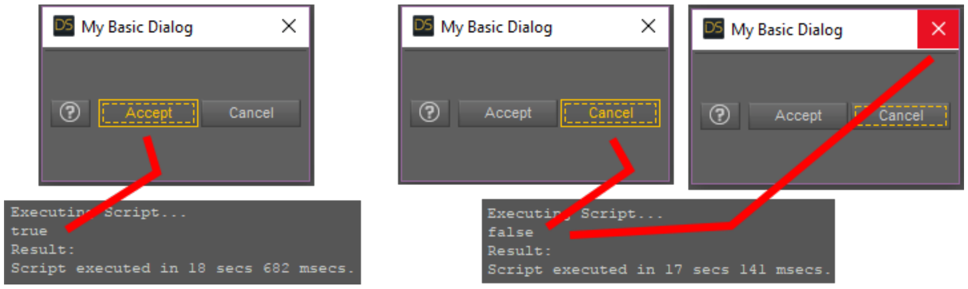
```
var wDlg = new DzBasicDialog()
wDlg.caption = "My Basic Dialog";
wDlg.setFixedSize(200,100)
wDlg.exec()
```



It's getting better, but there still isn't any message on it, and how do I know which button was clicked? Let's start with the buttons; you can capture the user's response by assigning the dialog object itself to a variable. The ***Accept*** button will return "*true*" while the ***Cancel*** and ***Close*** buttons will return "*false*".



```
var wDlg = new DzBasicDialog()
var bUserResponse
wDlg.caption = "My Basic Dialog";
wDlg.setFixedSize(200,100)
bUserResponse=wDlg.exec()
print(bUserResponse)
```



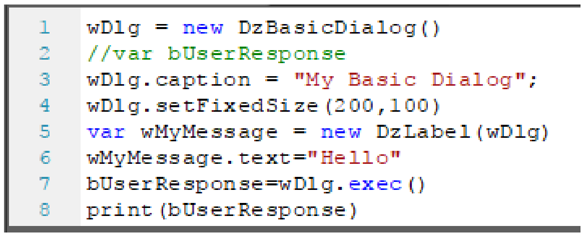
```
Executing Script...
true
Result:
Script executed in 18 secs 682 msecs.
```

```
Executing Script...
false
Result:
Script executed in 17 secs 141 msecs.
```

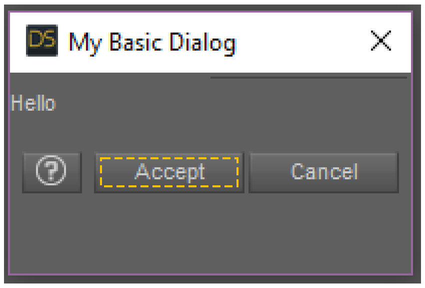
DzLabel

As for the message to our user, we will need to add a label onto our dialog using the **DzLabel** widget. When declaring input/output widgets, you must include the name of the dialog upon which they will appear.

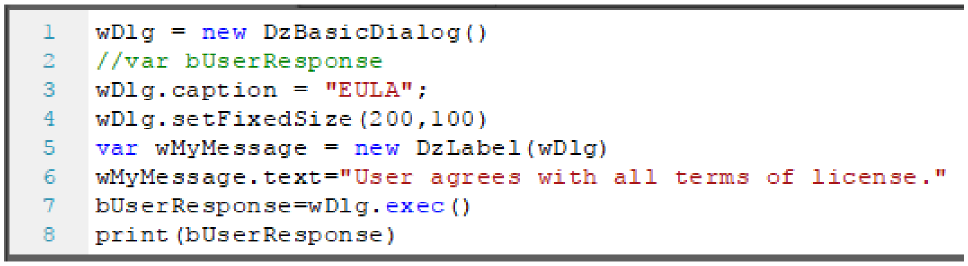
var *xxxxx* = new **DzLabel**(*dialogname*)



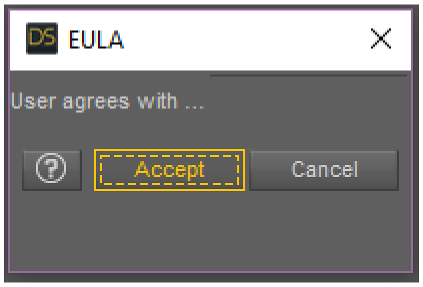
```
wDlg = new DzBasicDialog()
//var bUserResponse
wDlg.caption = "My Basic Dialog";
wDlg.setFixedSize(200,100)
var wMyMessage = new DzLabel(wDlg)
wMyMessage.text="Hello"
bUserResponse=wDlg.exec()
print(bUserResponse)
```



Ok, now let's try to make our dialog something practical and needed. How about agreeing with all the particulars of the end user license agreement? That should work as a good real-world example.



```
wDlg = new DzBasicDialog()
//var bUserResponse
wDlg.caption = "EULA";
wDlg.setFixedSize(200,100)
var wMyMessage = new DzLabel(wDlg)
wMyMessage.text="User agrees with all terms of license."
bUserResponse=wDlg.exec()
print(bUserResponse)
```

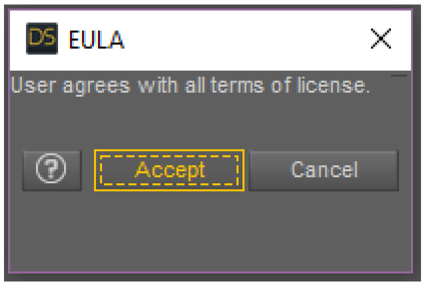


EULA

User agrees with ...

Wow, that did not work so well; we cannot read the complete message line. It appears that even the label has a default size. Let's try that again, but this time before defining it, we will resize the label just like we did the dialog.

wMyMessage.**setFixedSize**(190,15)



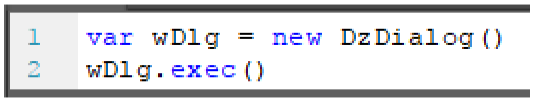
EULA

User agrees with all terms of license.

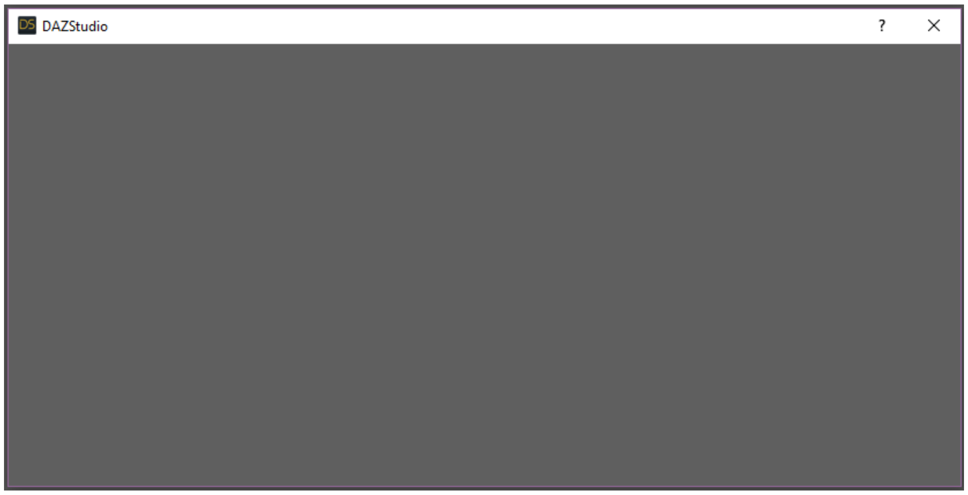
SUCCESS! That looks much better.

DzDialog

The DzBasicDialog looks good, but what if I want to create my own look and feel? The ***DzDialog*** object is like a blank slate giving you all of the room you need for your artistic freedom when it comes to designing dialogs.



```
var wDlg = new DzDialog()
wDlg.exec()
```



Event Handling

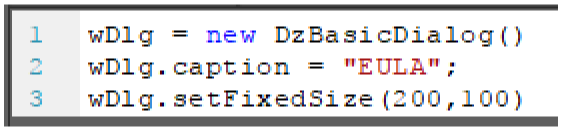
Connect is the function by which you define a widget as an event handler. In actuality, when the widget is first created it will have inherited events applicable to the type of widget it is. For instance, buttons on your dialog will be clicked on or pressed and this is an event to the OS. Your script must connect event(s) that you are watching for to the user-defined functions that you create to handle the event.

connect(*widget*,*event*,*function*)

This will all become clearer in the next section where we create our first widget that is an event handler (a button of course, as you may have already guessed).

Property vs. Method

Before we continue on to our next widget, let's quickly review how to recognize the difference between properties and methods for objects. Those are the words that you find following the period "." after the object's name. A property will have an assignment with the equals "=" sign, while a method will have a parameters section with the parenthesis "()" symbols. Note there may or may not be data in the parameters section. In the example below, ***caption*** is a property whilst ***setFixedSize*** is a method.



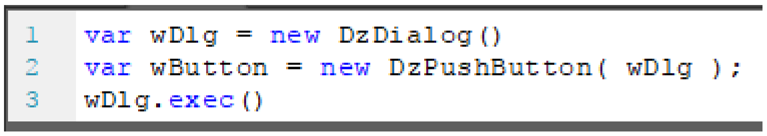
```
wDlg = new DzBasicDialog()
wDlg.caption = "EULA";
wDlg.setFixedSize(200,100)
```

DzPushButton

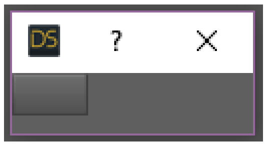
The ***DzPushButton*** widget will be one of the most used widgets in your arsenal as almost every popup has some sort of buttons on it. Here is the overall structure of the command line used to declare and define your new button.

var ***wButton*** = new **DzPushButton**(*dialog*)

As always, you must first have a dialog to place the new widget on. Then you will create the button with your desired name.

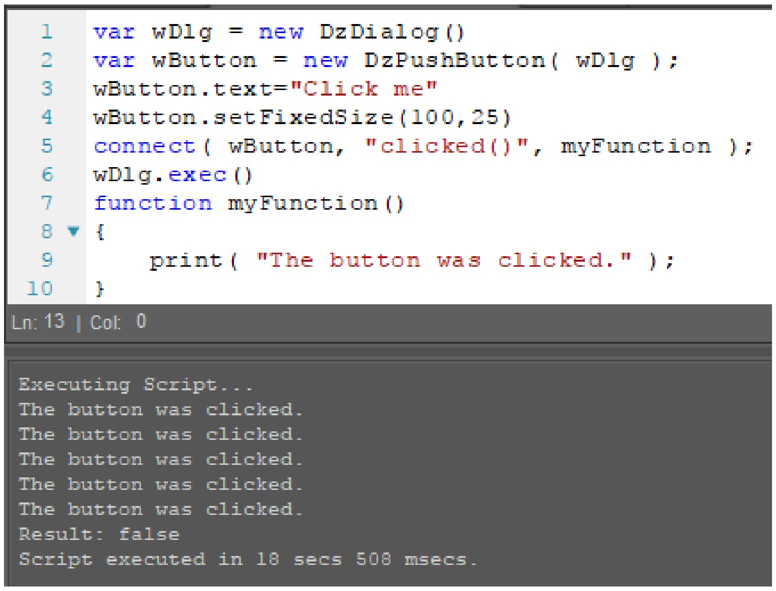


```
var wDlg = new DzDialog()
var wButton = new DzPushButton( wDlg );
wDlg.exec()
```



This doesn't look very exciting, huh? But did you notice how the dialog box resized itself to fit the widgets placed upon it? Let's continue on to make it more practical.

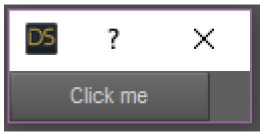
Now we need to place some text on the button using the ***text*** property and size it to fit our desires with the ***setFixedSize*** method. Lastly, the event that we will waiting for will be the clicked() event.



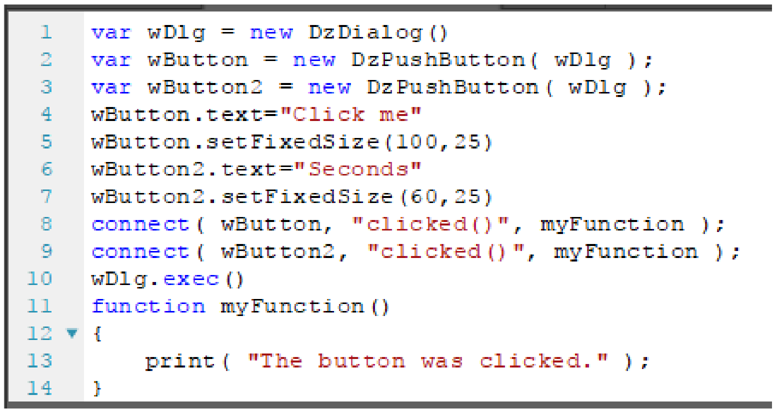
```
var wDlg = new DzDialog()
var wButton = new DzPushButton( wDlg );
wButton.text="Click me"
wButton.setFixedSize(100,25)
connect( wButton, "clicked()", myFunction );
wDlg.exec()
function myFunction()
{
    print( "The button was clicked." );
}
```

```
Ln: 13 | Col: 0
```

```
Executing Script...
The button was clicked.
The button was clicked.
The button was clicked.
The button was clicked.
The button was clicked.
Result: false
Script executed in 18 secs 508 msecs.
```

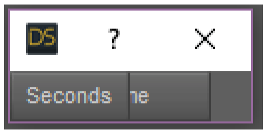


SUCCESS! We clicked the push button five times and everything worked great. Let's add a second "wButton2" to make our dialog more interesting, but use the same *myFunction* call for the second button. Do you think this is going to work?

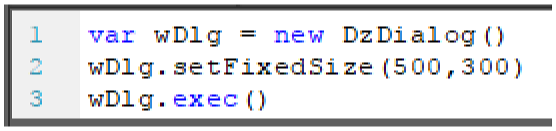


```
var wDlg = new DzDialog()
var wButton = new DzPushButton( wDlg );
var wButton2 = new DzPushButton( wDlg );
wButton.text="Click me"
wButton.setFixedSize(100,25)
wButton2.text="Seconds"
wButton2.setFixedSize(60,25)
connect( wButton, "clicked()", myFunction );
connect( wButton2, "clicked()", myFunction );
wDlg.exec()
function myFunction()
{
    print( "The button was clicked." );
}
```

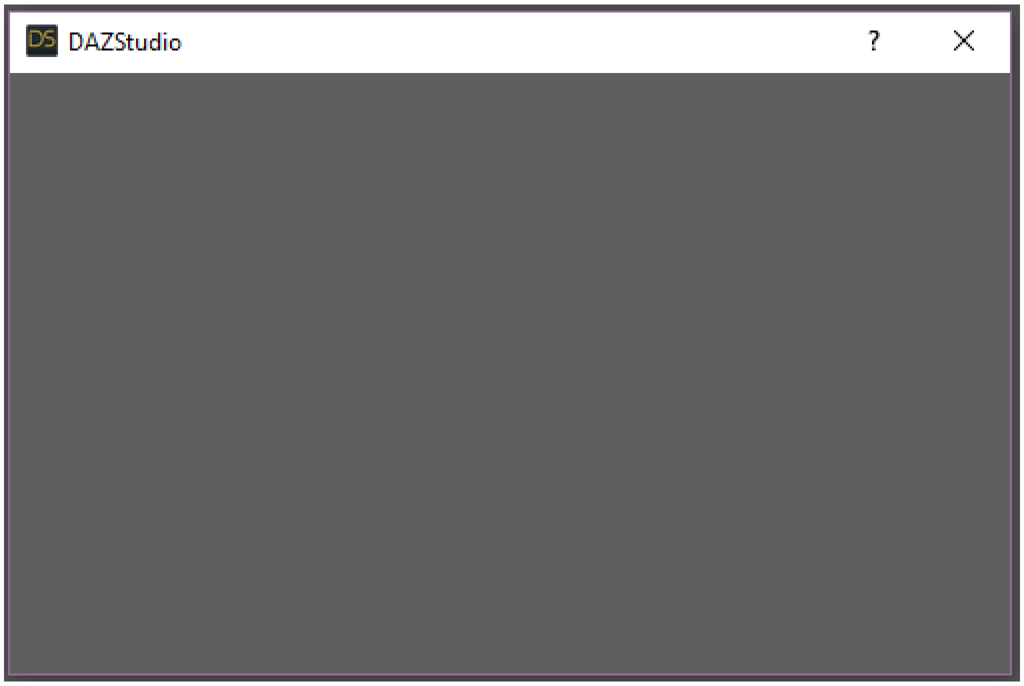
Ouch! This is probably not what you were expecting. Even though we sized the second button to be smaller, it is still covering our first button. Sure, you can click on *wButton2* and if needed click on the right edge of the *wButton*, but that is just not practical. Not to mention that this isn't a very impressive looking popup.



Before we correct this problem, we will need to cover the basic concepts of pixel sizing and the 2D coordinate system. As you may recall, earlier on the dialog box automatically resized itself for our first button. However, that did not work for our two button demo. The good news is that you can force the dialog box to be whatever size fits your needs. When you create a dialog box, it has a width and height that are measured in dots that are called pixels. The pixels that comprise the width of the dialog box are counted from left to right starting with zero. The pixels that comprise the height are counted from top to bottom stating with zero. You can use the ***setFixedSize*** method to change the size of your dialog box.

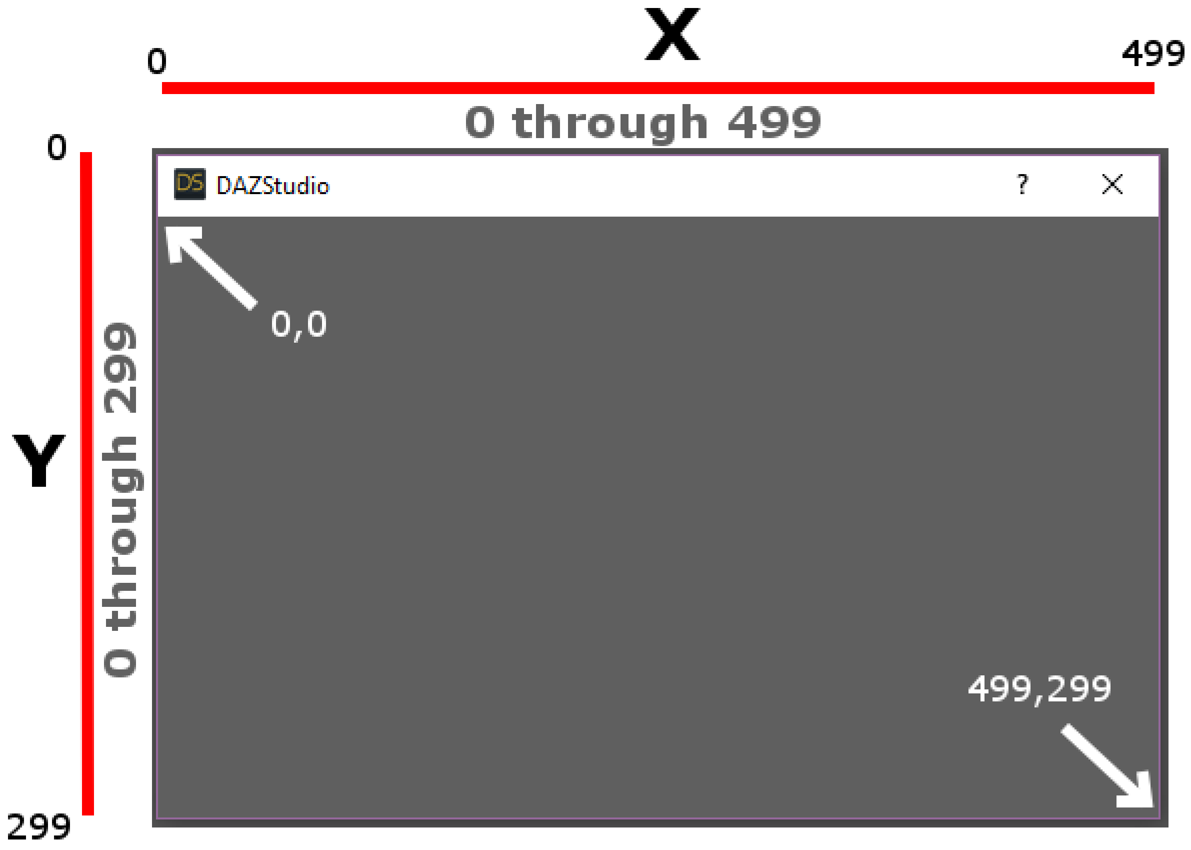


```
var wDlg = new DzDialog()
wDlg.setFixedSize(500,300)
wDlg.exec()
```



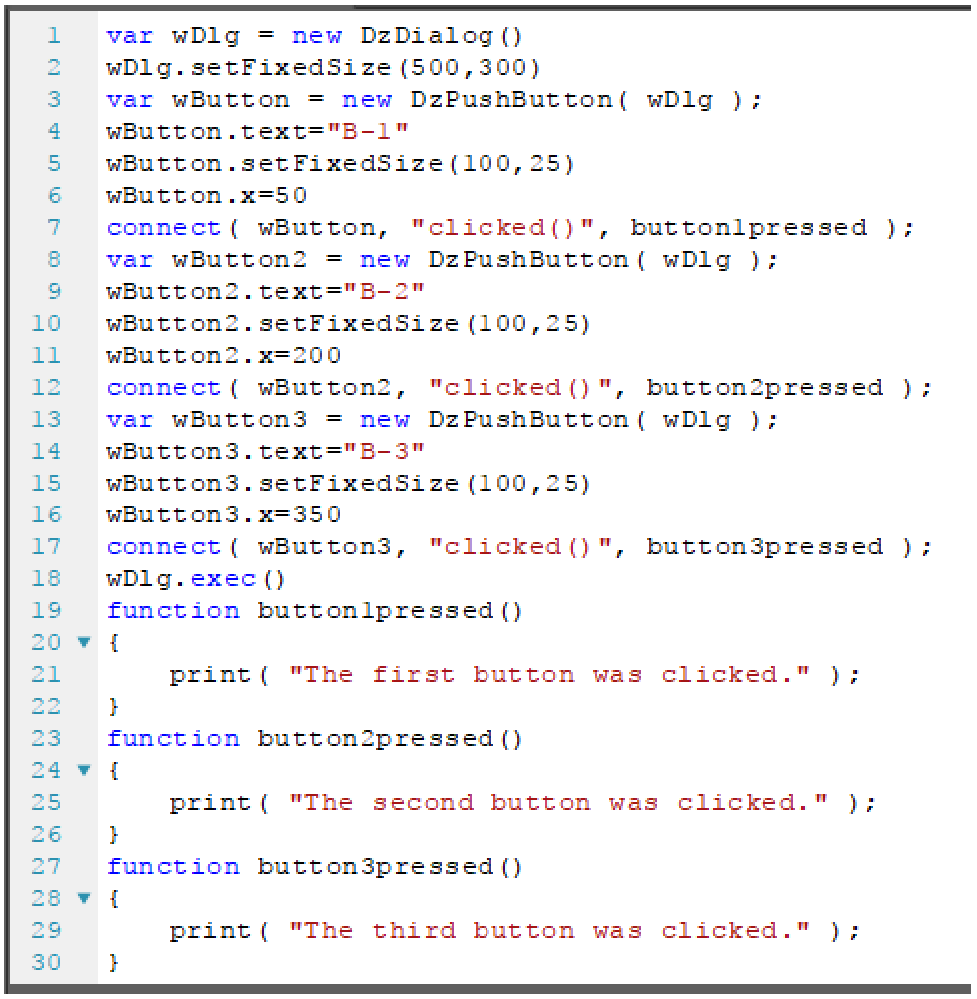
2D Coordinates

Pixel locations for the width of the dialog box are called the "**X**" direction. Pixel locations for the height of the dialog box are called the "**Y**" direction. The X and Y directions are referred to as the **axis** for that direction. Pixel locations start with zero (remember that zero-based indexing) and go either to the right or down depending on which axis they are being counted. Together, these two pixel positions comprise a specific location because you can pinpoint an exact spot on the dialog box using the pixel locations in both directions. Pixel locations are written using the X and Y coordinates separated by a comma like **x,y** but you will often see them written using a set of parenthesis like **(x,y)**. In the figure below, you can see the coordinates for the upper-left and lower-right pixels (locations) on the dialog box. This universal design means that there are 150,000 unique pixel locations (500 x 300) for the dialog box popup. Here is the dialog box we resized showing how the 500 physical pixels in width are counted from 0 through 499, and the 300 physical pixels in height are counted from 0 through 299.



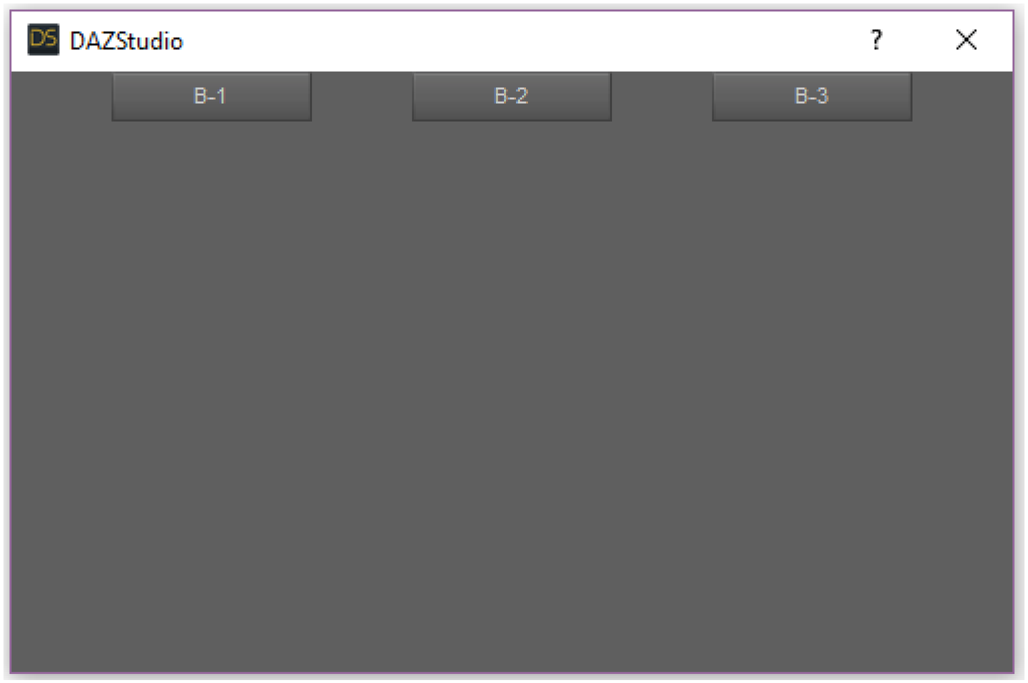
Positioning Widgets

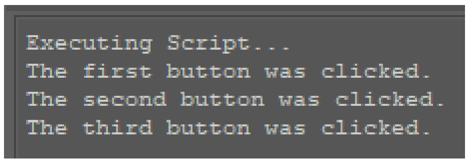
You can control the placement of your widgets on the dialog box using the 2D coordinate system. Most widgets have both "***x***" and "***y***" properties that you can set. The default value for both of these properties is zero (0). These two properties together represent the **x,y** pixel location for the upper-left corner of the widget. Now it makes sense why one button was covering the other button. You can change (set) one or both properties to move the widget to any location you desire. With that in mind, let's try to see if we can move our second button away from the first and add a third button for even more effect. We will make all three buttons the same size and leave the **y** position at the default zero so they align at the top. We will also have separate functions for each button click.



```
var wDlg = new DzDialog()
wDlg.setFixedSize(500,300)
var wButton = new DzPushButton( wDlg );
wButton.text="B-1"
wButton.setFixedSize(100,25)
wButton.x=50
connect( wButton, "clicked()", button1pressed );
var wButton2 = new DzPushButton( wDlg );
wButton2.text="B-2"
wButton2.setFixedSize(100,25)
wButton2.x=200
connect( wButton2, "clicked()", button2pressed );
var wButton3 = new DzPushButton( wDlg );
wButton3.text="B-3"
wButton3.setFixedSize(100,25)
wButton3.x=350
connect( wButton3, "clicked()", button3pressed );
wDlg.exec()
function button1pressed()
{
    print( "The first button was clicked." );
}
function button2pressed()
{
    print( "The second button was clicked." );
}
function button3pressed()
{
    print( "The third button was clicked." );
}
```

SUCCESS! We now have a fully functional multi-button script with input from the user in the form of button clicks and output to the user in the form of printing which button was clicked.

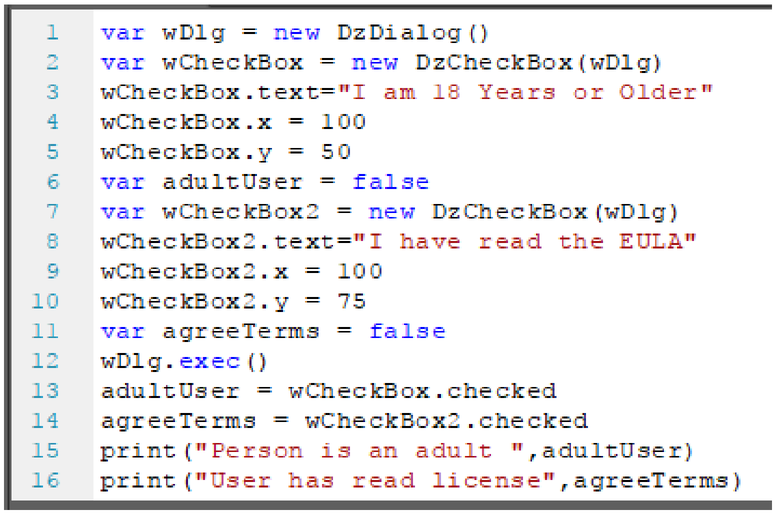




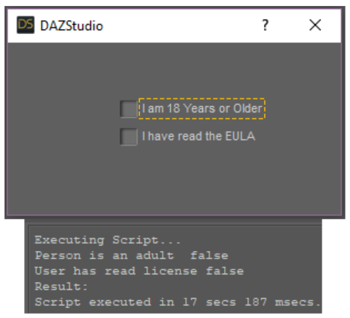
```
Executing Script...
The first button was clicked.
The second button was clicked.
The third button was clicked.
```

DzCheckBox

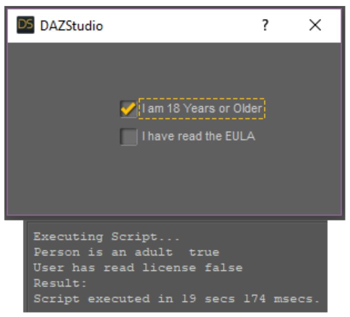
You will use checkboxes when you want the user to check or uncheck their response to an inquiry you place on the widget. When using more than one checkbox, the user can select or unselect given choices independently of each other. The properties for the **DzCheckBox** include ***text*** and ***x,y*** positioning so you can customize it to your liking. You can capture the *return* value, a Boolean ***true*** or ***false*** for the state (checked or unchecked), by assigning it to a variable.



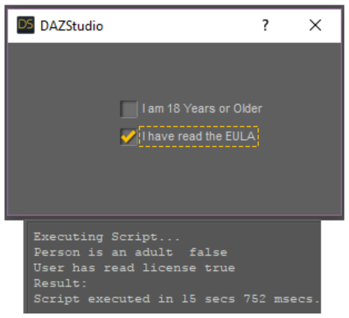
```
var wDlg = new DzDialog()
var wCheckBox = new DzCheckBox(wDlg)
wCheckBox.text="I am 18 Years or Older"
wCheckBox.x = 100
wCheckBox.y = 50
var adultUser = false
var wCheckBox2 = new DzCheckBox(wDlg)
wCheckBox2.text="I have read the EULA"
wCheckBox2.x = 100
wCheckBox2.y = 75
var agreeTerms = false
wDlg.exec()
adultUser = wCheckBox.checked
agreeTerms = wCheckBox2.checked
print("Person is an adult ",adultUser)
print("User has read license",agreeTerms)
```



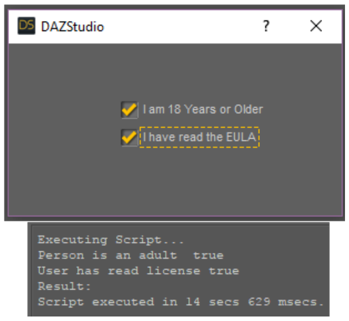
```
Executing Script...
Person is an adult  false
User has read license false
Result:
Script executed in 17 secs 187 msecs.
```



```
Executing Script...
Person is an adult  true
User has read license false
Result:
Script executed in 19 secs 174 msecs.
```



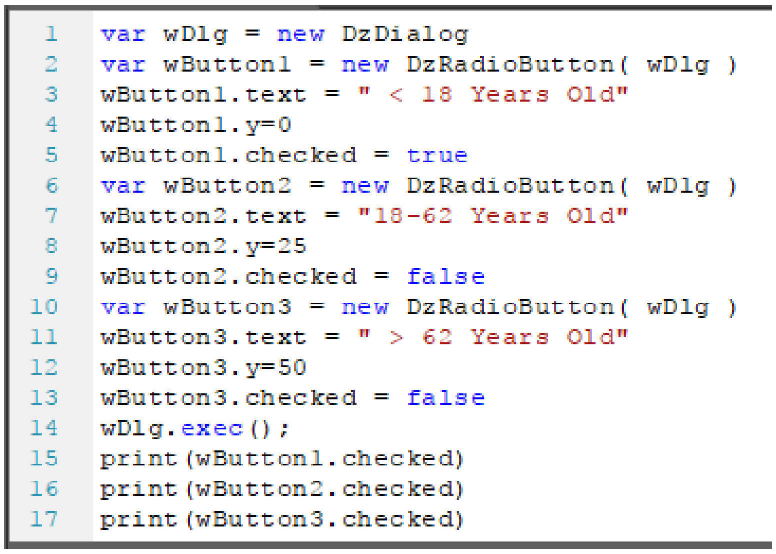
```
Executing Script...
Person is an adult  false
User has read license true
Result:
Script executed in 15 secs 752 msecs.
```



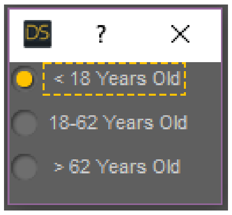
```
Executing Script...
Person is an adult  true
User has read license true
Result:
Script executed in 14 secs 629 msecs.
```

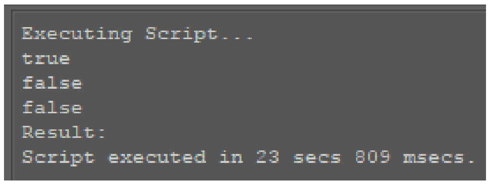
DzRadioButton

Unlike the checkbox, when using ***DzRadioButton***(s) the user can only select one of the available options. All radio buttons defaults to false, so you should set one to true when they are declared. If you accidentally set more than one to true, the last one whose value is changed will be the selected one. The properties for the **DzRadioButton** also include ***text*** and ***x,y*** positioning. Normally you would see radio buttons stacked vertically, however you can position them as you prefer.

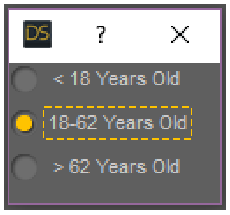


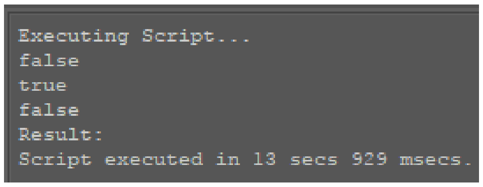
```
var wDlg = new DzDialog
var wButton1 = new DzRadioButton( wDlg )
wButton1.text = " < 18 Years Old"
wButton1.y=0
wButton1.checked = true
var wButton2 = new DzRadioButton( wDlg )
wButton2.text = "18-62 Years Old"
wButton2.y=25
wButton2.checked = false
var wButton3 = new DzRadioButton( wDlg )
wButton3.text = " > 62 Years Old"
wButton3.y=50
wButton3.checked = false
wDlg.exec();
print(wButton1.checked)
print(wButton2.checked)
print(wButton3.checked)
```



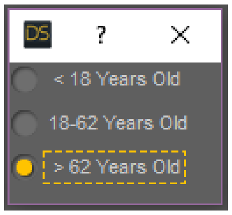


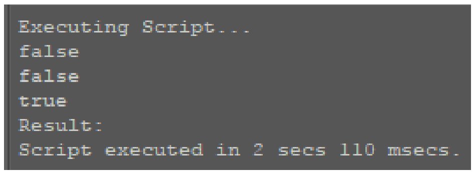
```
Executing Script...
true
false
false
Result:
Script executed in 23 secs 809 msecs.
```





```
Executing Script...
false
true
false
Result:
Script executed in 13 secs 929 msecs.
```



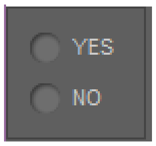


```
Executing Script...
false
false
true
Result:
Script executed in 2 secs 110 msecs.
```

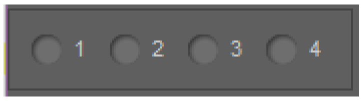
Grouping Widgets

There will be times when you need to group together several widgets. This can be done for the buttons we previously scripted, however it is more often used for check boxes and radio buttons. For example, if you needed two different sets of radio buttons, they will have to work independently of each other in order for your script to function properly. There are various methods to accomplish this; however in this section we will discuss **DzVButtonGroup** and **DzHButtonGroup** widgets.

First, you create the main dialog widget and any grouping types that you desire. The ***V*** is for vertically arranged buttons, and the ***H*** for horizontally arranged buttons. Create as many groups as you need for your script.

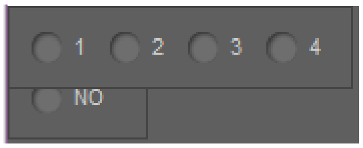


Vertical



Horizontal

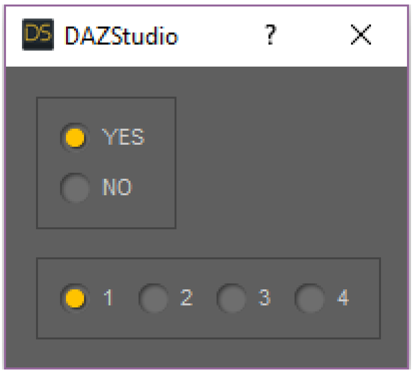
You will then have to position the groups that you create on the main dialog so that they do not overlap each other (unless that is the desired effect). You can do that using the ***X*** and ***Y*** properties for each group.



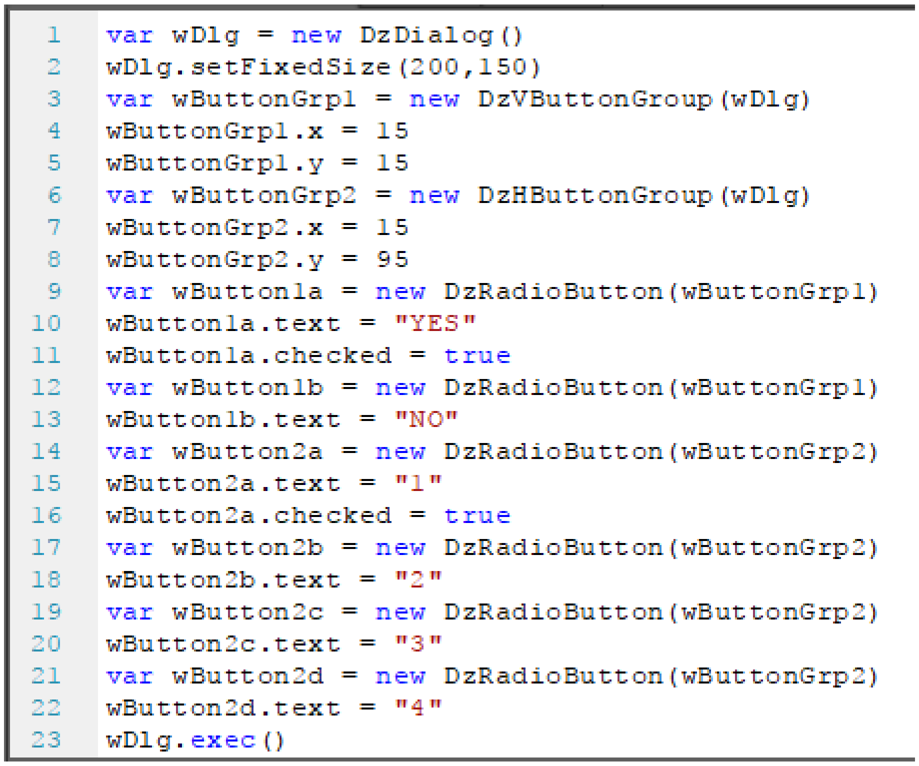
Overlapping Groups

Please note that the images shown above include buttons that were created in the next paragraph in order to demonstrate the described features or effects.

Lastly, you create any buttons you need designating the group widget for which they are intended. Be sure to set at one of your buttons for each group to default as ***true*** so it will appear enabled when your dialog appears.

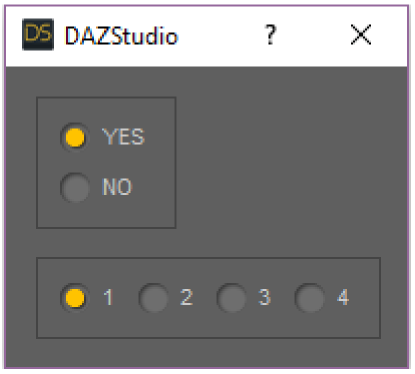


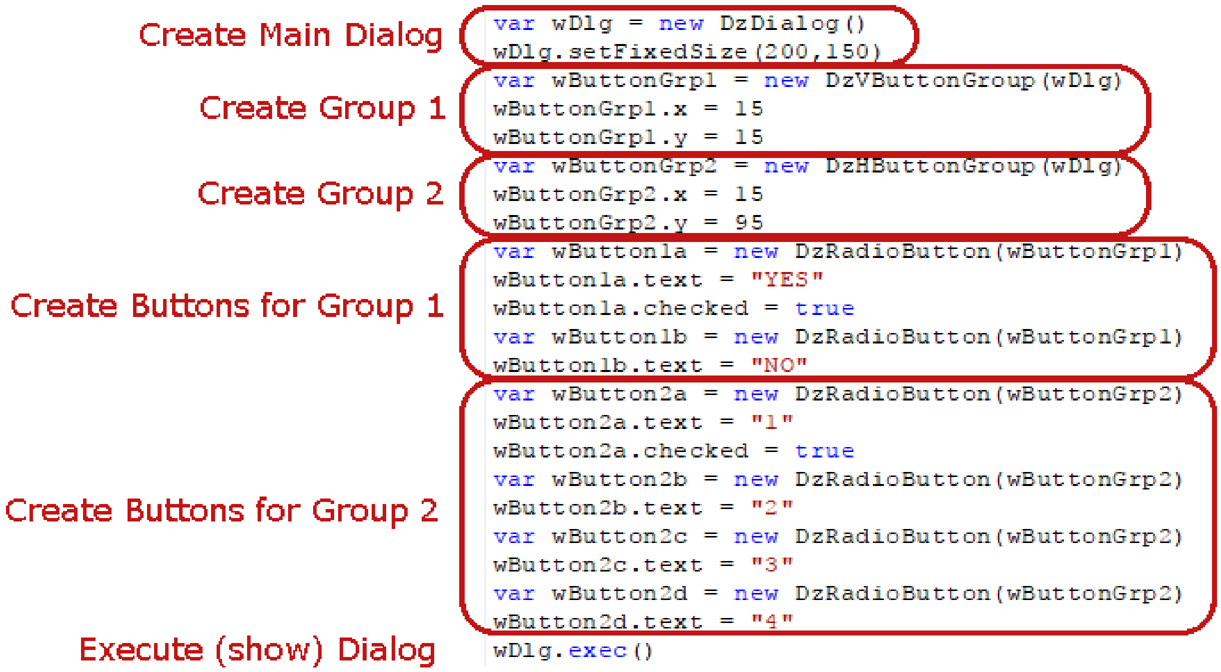
Sound complicated? Don't worry because it is simpler that it sounds. Here is how it was all done. You can find the annotated script on the next page.



```
var wDlg = new DzDialog()
wDlg.setFixedSize(200,150)
var wButtonGrp1 = new DzVButtonGroup(wDlg)
wButtonGrp1.x = 15
wButtonGrp1.y = 15
var wButtonGrp2 = new DzHButtonGroup(wDlg)
wButtonGrp2.x = 15
wButtonGrp2.y = 95
var wButton1a = new DzRadioButton(wButtonGrp1)
wButton1a.text = "YES"
wButton1a.checked = true
var wButton1b = new DzRadioButton(wButtonGrp1)
wButton1b.text = "NO"
var wButton2a = new DzRadioButton(wButtonGrp2)
wButton2a.text = "1"
wButton2a.checked = true
var wButton2b = new DzRadioButton(wButtonGrp2)
wButton2b.text = "2"
var wButton2c = new DzRadioButton(wButtonGrp2)
wButton2c.text = "3"
var wButton2d = new DzRadioButton(wButtonGrp2)
wButton2d.text = "4"
wDlg.exec()
```

Here is the dialog again along with details of how it was created.





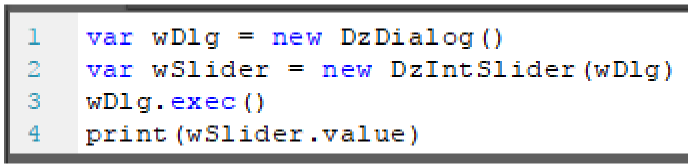
```
var wDlg = new DzDialog()
wDlg.setFixedSize(200,150)
var wButtonGrp1 = new DzVButtonGroup(wDlg)
wButtonGrp1.x = 15
wButtonGrp1.y = 15
var wButtonGrp2 = new DzHButtonGroup(wDlg)
wButtonGrp2.x = 15
wButtonGrp2.y = 95
var wButton1a = new DzRadioButton(wButtonGrp1)
wButton1a.text = "YES"
wButton1a.checked = true
var wButton1b = new DzRadioButton(wButtonGrp1)
wButton1b.text = "NO"
var wButton2a = new DzRadioButton(wButtonGrp2)
wButton2a.text = "1"
wButton2a.checked = true
var wButton2b = new DzRadioButton(wButtonGrp2)
wButton2b.text = "2"
var wButton2c = new DzRadioButton(wButtonGrp2)
wButton2c.text = "3"
var wButton2d = new DzRadioButton(wButtonGrp2)
wButton2d.text = "4"
wDlg.exec()
```

The Perfect Dialog

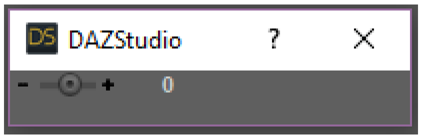
When creating custom dialogs, you will have to experiment with the ***size*** and ***position*** of each widget to get the look and feel you desire.

Sliders

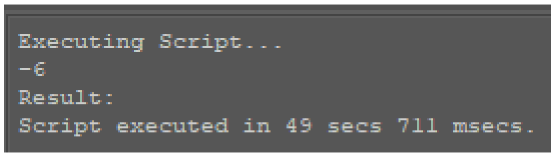
Oh my gosh! If you think you like buttons, you are going to love sliders! Here is the script for a simple slider control using **DzIntSlider** widget and how it looks.



```
var wDlg = new DzDialog()
var wSlider = new DzIntSlider(wDlg)
wDlg.exec()
print(wSlider.value)
```

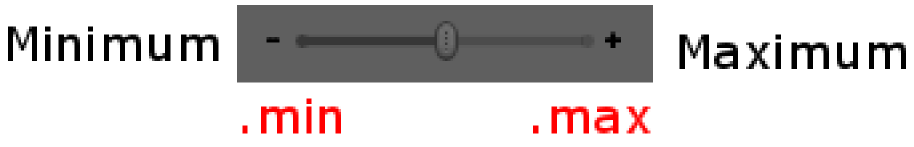


Not very impressive, but it does work. The default starting value for your slider will be zero (0), but you can change this using the ***value*** property. You will also use this property to retrieve any value selected by the user. You can see here where we ran the script and ended the dialog with a value of “-6” on the slider.

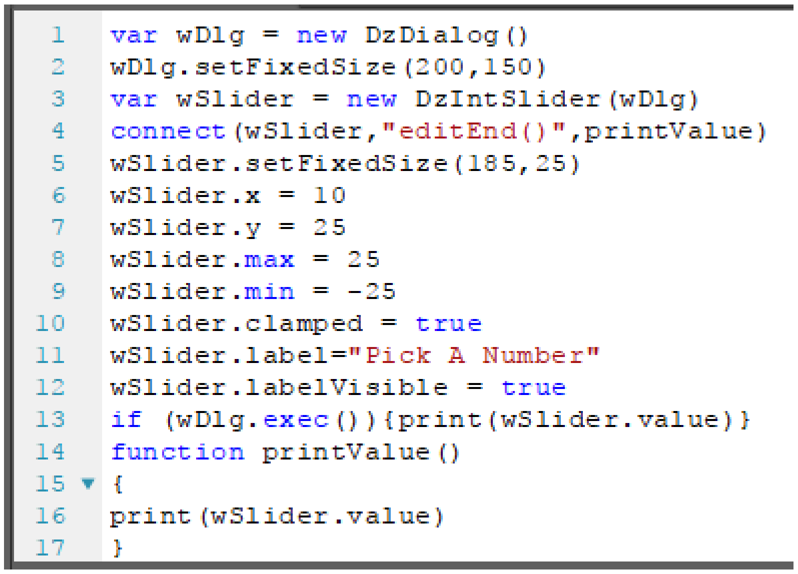


```
Executing Script...
-6
Result:
Script executed in 49 secs 711 msecs.
```

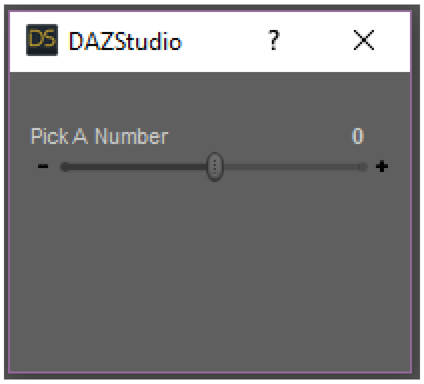
Ok, now let’s make it look much more appealing and usable. Just like the many other widgets, you can set the size/position of sliders using the ***setFixedSize*** method and ***x, y*** properties. To further define your slider, you can use the ***min*** and ***max*** properties to set the minimum and maximum values that you want to retrieve.



Lastly, what good is a slider widget if the user does not know what to use it for? We can solve this with the ***label*** and ***labelVisible*** properties. Some widgets are a little more advanced and therefore take a bit more coding to properly develop. Here is our finished script for a fully functional slider widget.

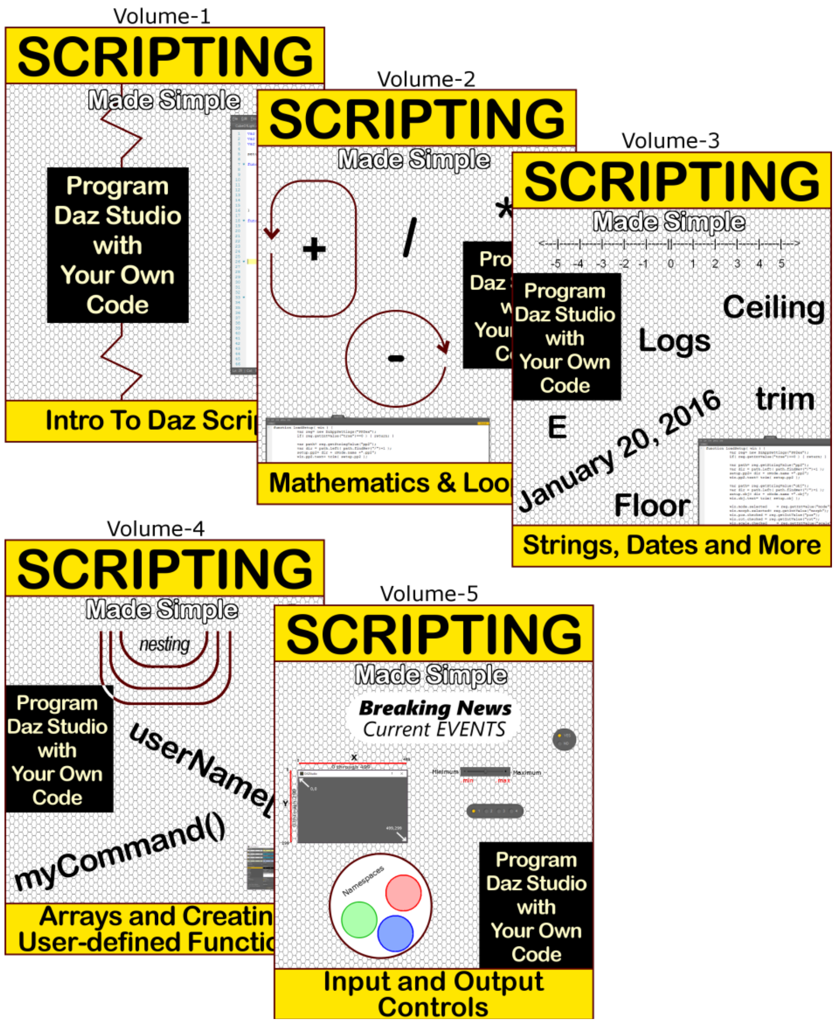


```
var wDlg = new DzDialog()
wDlg.setFixedSize(200,150)
var wSlider = new DzIntSlider(wDlg)
connect(wSlider,"editEnd()",printValue)
wSlider.setFixedSize(185,25)
wSlider.x = 10
wSlider.y = 25
wSlider.max = 25
wSlider.min = -25
wSlider.clamped = true
wSlider.label="Pick A Number"
wSlider.labelVisible = true
if (wDlg.exec()){print(wSlider.value)}
function printValue()
{
print(wSlider.value)
}
```



Conclusion

We hope that you have enjoyed this tutorial and have found some inspiration to further develop your scripting experience by using the available input and output controls in the Daz Studio environment. Please watch our website for other volumes in the Scripting Made Simple series. Future tutorial volumes will cover such topics as the 3D space, controlling objects in your scene, advanced mathematics, and more...



BE YOURSELF AND SUCCEED
Winterbrose
Arts and Graphics