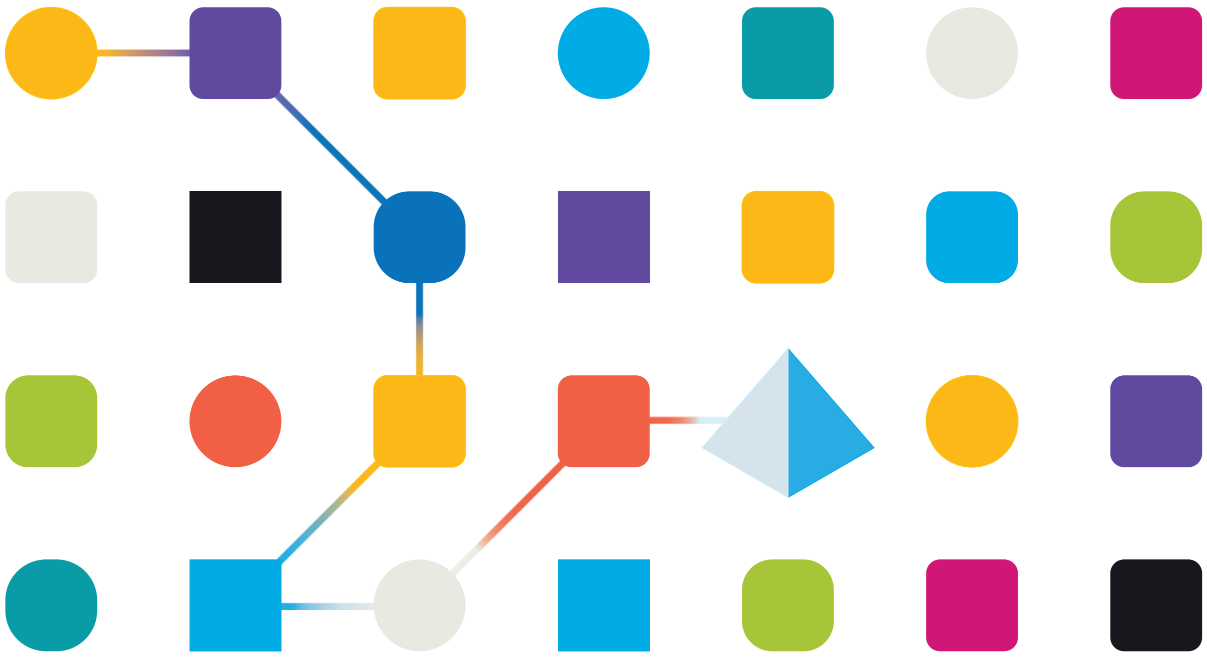
blueprism®

Code Stages

Developer Guide

Document Revision 1.0



Trademarks and copyright

Contents

Figures

Introduction

Blue Prism has many tools to create automations between systems. It is sometimes necessary to leap beyond the provided tools and perform an action that is totally customised. Perhaps a customer has a legacy application that is too costly to replace or perhaps the knowledge that created it has now left the company. There are many reasons why a code stage could be useful. The main one being, to do something that you just cannot accomplish using the standard Blue Prism tools.

Knowledge

There is always a cost to using a flexible tool such as a code stage. This being knowledge of .NET development. Code stages allow code to be written in Visual Basic.NET, C# and Visual J#. We will focus on C# in this document.

Now, if you are looking to do something more complex than perhaps a simple calculation, you may need to learn more about VB.NET or C#. That is beyond the scope of this paper but there are plenty of resources around to assist with this.

On the other hand, if you are a .Net coding veteran, this should be quite enjoyable for you. There are a few things to be aware of, particularly in the area of how the code elements are arranged. Blue Prism also provides a development paradigm called single-responsibility out of the box. Uncle Bob would be so pleased.

Solution Files

This file can be found on our Blue Prism Github repository at https://github.com/blue-prism/Code-Stage-Training.

To use this asset, import the bprelease file named *Blue Prism DX – Code Stage* The repository contains all the files we refer to in this paper and provide a working example of code stages and how to get the most out of them.

The Project

We are going to build an object that will perform several tasks:

1. Add together 2 numbers. This will initially be pre-set values in the .net code but we will expand this out to include parameters.
2. Call a separate routine to undertake a single task of returning us a set of Fibonacci numbers. We will take those numbers and produce a Blue Prism Collection.
3. Finally, we will look at calling both Python and PowerShell scripts from a code stage.

Create the Business Object

We are going to create a business object, that will sit under the default node of the objects tree. To do this, right-click on the objects node and choose Create Object.

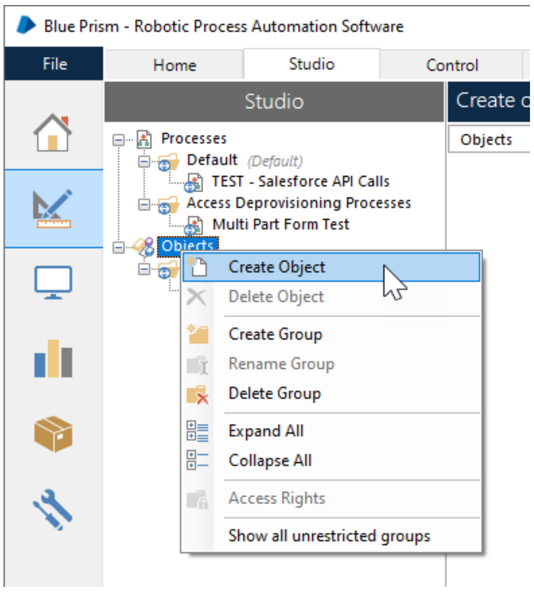
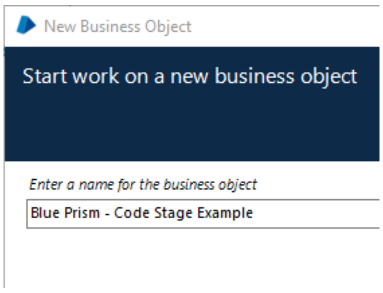


Figure 1 Create Object

When the dialog appears, enter a name for your new business object as we have done below.


New Business Object

Start work on a new business object

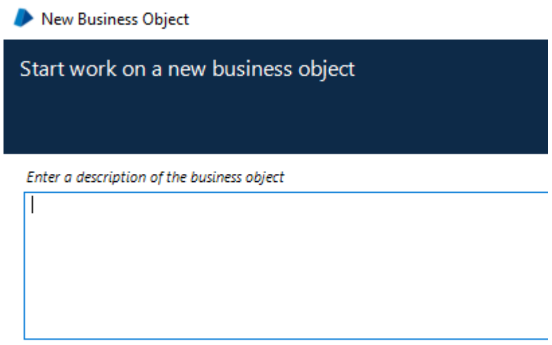
Enter a name for the business object

Blue Prism - Code Stage Example

Figure 2 Name the Business Object

The next dialog will present you with the option of a more detailed description of your business object.

Its up to you, but right now, we will leave this empty.


New Business Object

Start work on a new business object

Enter a description of the business object

Figure 3 Business Object Description

The next click of the mouse and we have our business object created and it will show up in the studio list.

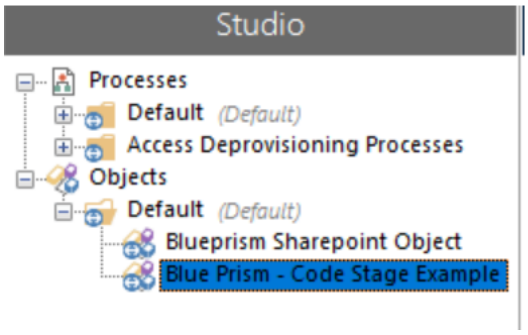
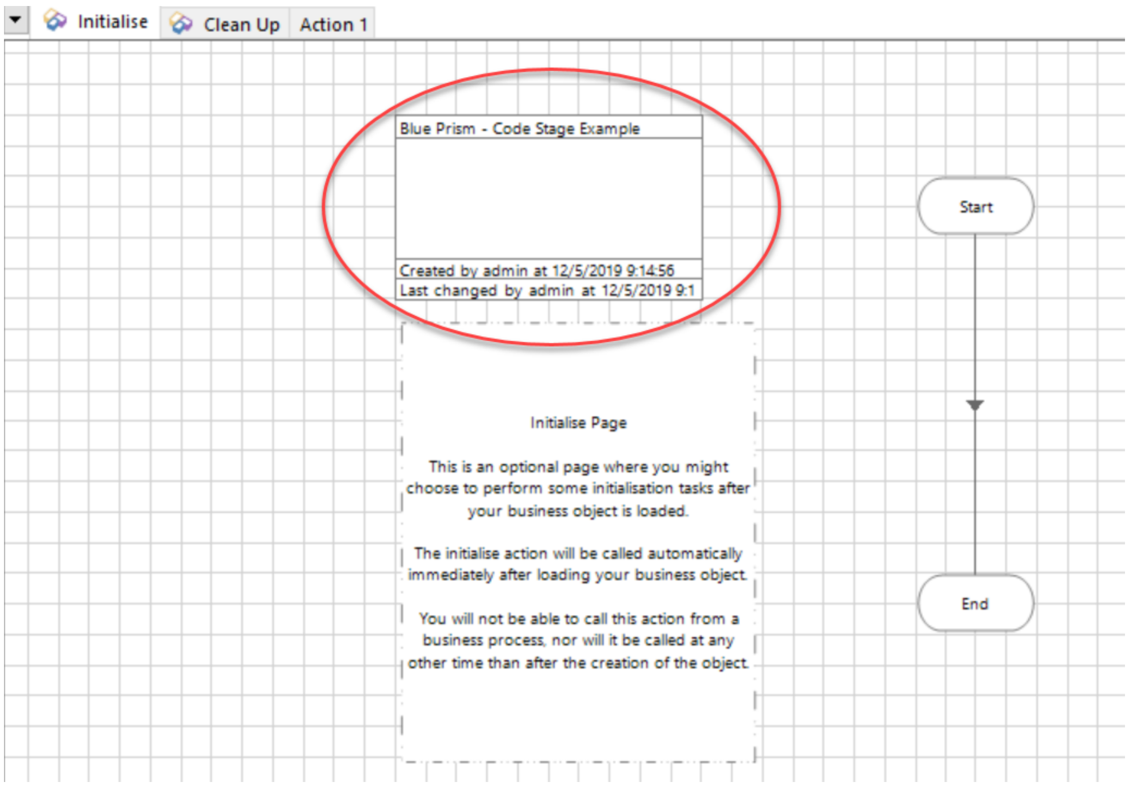

Studio
Processes
Default (Default)
Access Deprovisioning Processes
Objects
Default (Default)
Blueprism Sharepoint Object
Blue Prism - Code Stage Example

Figure 4 Business Object Created

Now the creation part is done, lets get on with making our object do something for us. Double click it and maximize the window.


Initialise
Clean Up
Action 1
Blue Prism - Code Stage Example
Created by admin at 12/5/2019 9:14:56
Last changed by admin at 12/5/2019 9:1
Start
End

Initialise Page

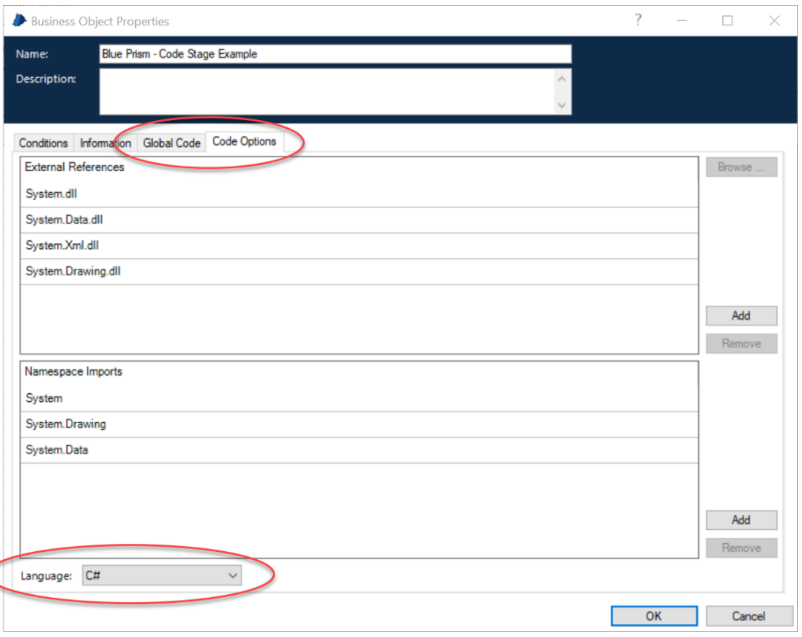
This is an optional page where you might choose to perform some initialisation tasks after your business object is loaded.

The initialise action will be called automatically immediately after loading your business object.

You will not be able to call this action from a business process, nor will it be called at any other time than after the creation of the object.

Figure 5 Business Object Initialize Page

So, this is the initialize page, the first thing we need to do when working with code stages is to decide our coding language. So, the area in ellipse, double click it and we will see how we select that.



Name: Blue Prism - Code Stage Example
Description:

External References
System.dll
System.Data.dll
System.Xml.dll
System.Drawing.dll

Namespace Imports
System
System.Drawing
System.Data

Language: C#

Figure 6 Business Object Properties

So, here we have 2 tabs that are of immediate interest to us. The one shown, *Code Options*, allows us to select our choice of .net language. These are Visual Basic, C# and Visual J#. For our purposes we will focus on Visual Basic and C#.

The *External References* and *Namespace Imports* are used to include libraries of .NET functions that exist in other files. Examples of these could be an SQL Server or MySQL database library or perhaps a library for machine learning. This is where code stages really come into their own. While not every business object needs one, they are there when you do.

The Global Code tab is exactly that, it's for code that can called throughout our business object. The benefit to this is only having a single area to update when things need to change. Rather than writing the same routine in each code stage, we can take advantage of this feature, reduce the amount of programming code we write and benefit from easier maintenance.

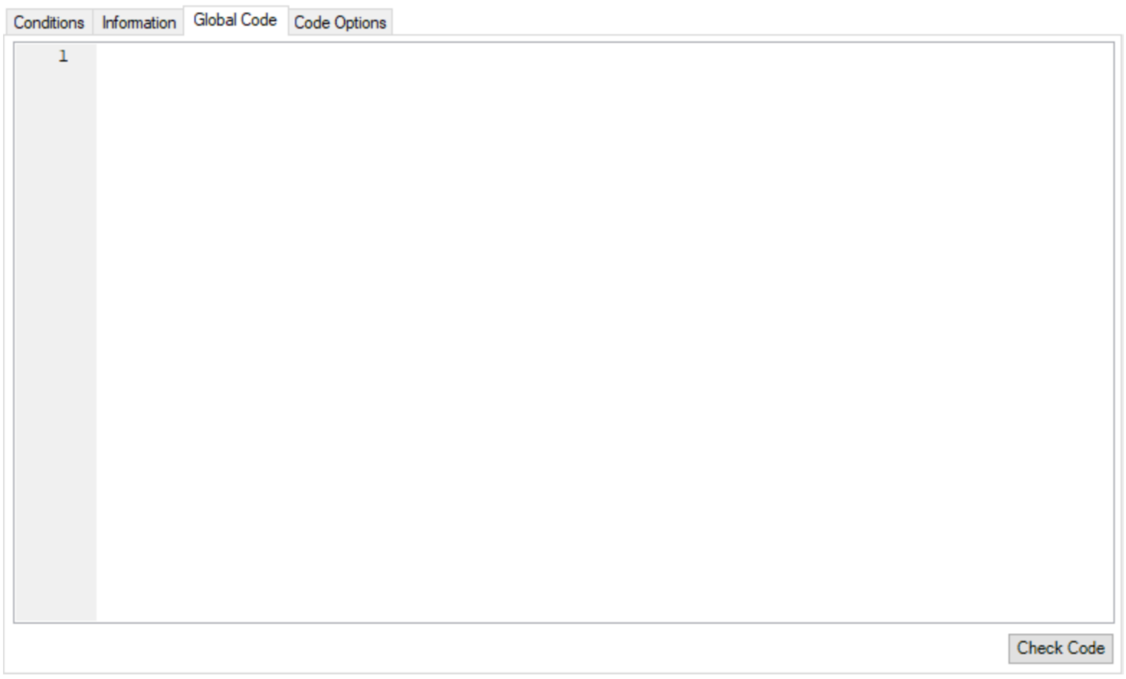


Figure 7 Global Code Tab

As you can see, there is no global code right now. In a little while we will revisit this window and populate it with some code.

Adding the Code Stage

So, we have 3 tabs on view in our object. These are the Initialise, Clean Up and for now, Action 1. Now any work that goes on in the Initialize tab, executes when the object is first created, so any set up should be done here. Likewise, in the interest of being tidy scouts and guides, we should leave the campsite better than when we arrived, so in the clean up we put any necessary tear down code. This will allow resources to be released and reduce the chance of memory leaks and other hard to find problems. Seasoned programmers might liken these stages to constructors and destructors.

To make our page more professional, we are going to rename the page and call it Add Numbers. This helps when the object is published and is available to be used by our (or others) processes. To do this we right-click the mouse on the Action 1 tab and choose Rename from the drop down.

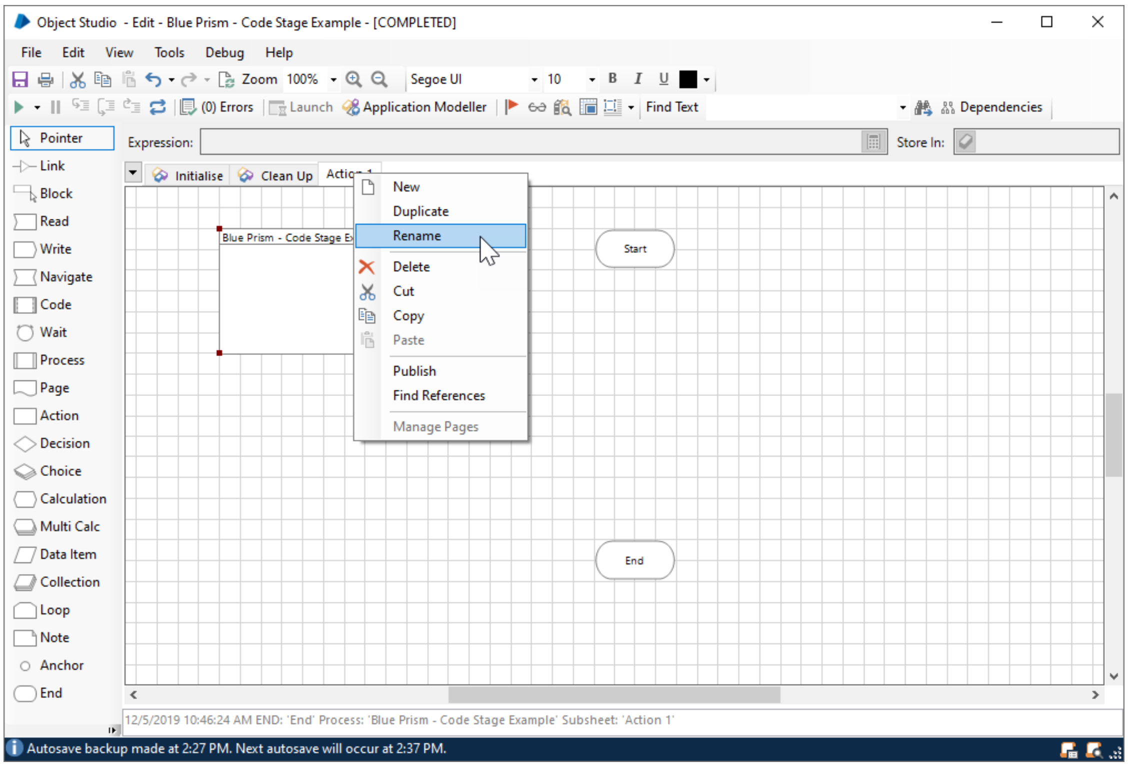
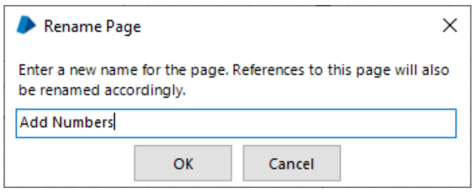

Object Studio - Edit - Blue Prism - Code Stage Example - [COMPLETED]
File Edit View Tools Debug Help
Zoom 100%
Segoe UI 10
(0) Errors Launch Application Modeller Find Text Dependencies
Pointer Link Block Read Write Navigate Code Wait Process Page Action Decision Choice Calculation Multi Calc Data Item Collection Loop Note Anchor End
Expression: Store In:
Initialise Clean Up Action 1
New Duplicate Rename Delete Cut Copy Paste Publish Find References Manage Pages
Blue Prism - Code Stage Ex
Start
End
12/5/2019 10:46:24 AM END: 'End' Process: 'Blue Prism - Code Stage Example' Subsheet: 'Action 1'
Autosave backup made at 2:27 PM. Next autosave will occur at 2:37 PM.

Figure 8 Renaming the Action

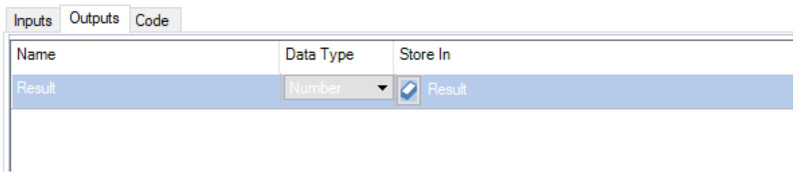

Rename Page

Enter a new name for the page. References to this page will also be renamed accordingly.

Add Numbers

Figure 9 Renaming the Action

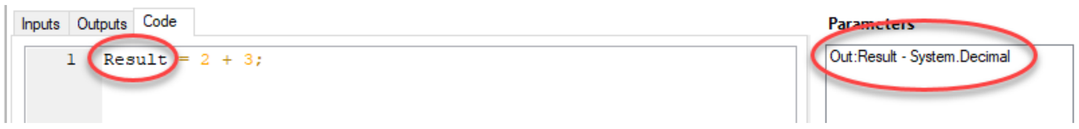
Our first task is to simply add together 2 numbers, using a code stage. Now, this is *not* something to use a code stage for. The reason being that this can be accomplished far easier using a Blue Prism calculation stage. In order to get a value from our code, we are going to create an output parameter. This will create a variable inside the code stage for us to put the result of our calculation into. Note the name *Result* as it will be seen how this is used in the code pane.



Name	Data Type	Store In
Result	Number	Result

Figure 10 Code Stage Output Parameter

We have our code stage and we now have something for it to do.



```
Result = 2 + 3;
```

Parameters
Out:Result - System.Decimal

Figure 11 Code Pane

So, the output parameter we created earlier is visible on the parameters list on the right of the image.

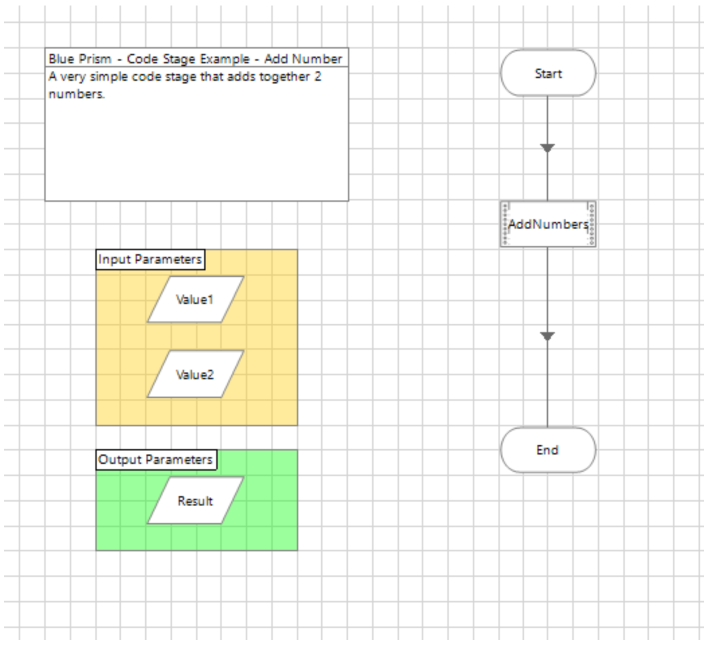
> Note, that we created this parameter as Blue Prism type number. This is translated into .net as a decimal.

Now when this code stage executes, the result of 5 will be put into our Result data item.

As you can see from the above, the values 2+3 are what we call *hard-coded*. This is great if we always want the result of 5. If we need another value, then we need to alter our code. This resultantly then means our code is *high maintenance*. It also has *zero reusability* as it would likely need to be changed every single time. So, lets solve this. Let's use some parameters.

We are going to add input parameters to our code stage and then take that a little further with parameters for our whole action which will then reduce the maintenance needed on our code and increase its reusability.

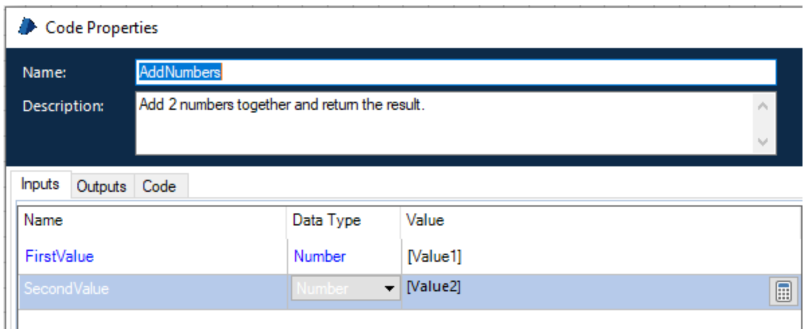
So, for our parameters, we need to create 2 data items of type number. We will call them Value1 and Value2, so not much imagination needed there.



Blue Prism - Code Stage Example - Add Number
A very simple code stage that adds together 2 numbers.

Figure 12 Data Items to be used for Parameter Values

So, now we have our data items, we can use them in our code stage.



Code Properties

Name: AddNumbers

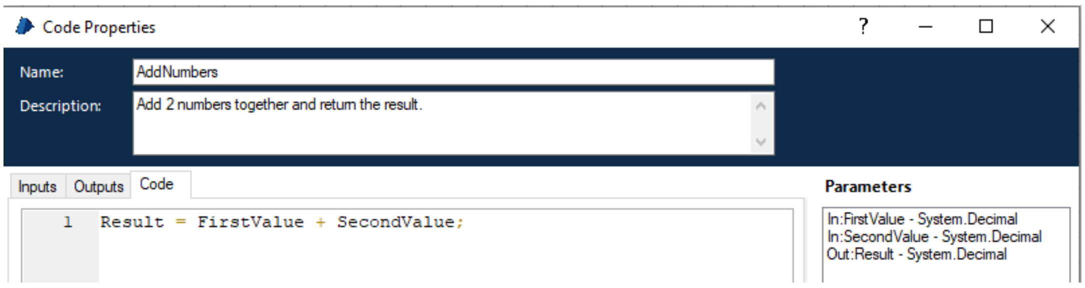
Description: Add 2 numbers together and return the result.

Inputs | Outputs | Code

Name	Data Type	Value
FirstValue	Number	[Value1]
SecondValue	Number	[Value2]

Figure 13 Code Stage Input Parameters

Now we can review our original code in our code stage and replace the number values with our newly created parameters.



Code Properties

Name: AddNumbers

Description: Add 2 numbers together and return the result.

Inputs | Outputs | Code

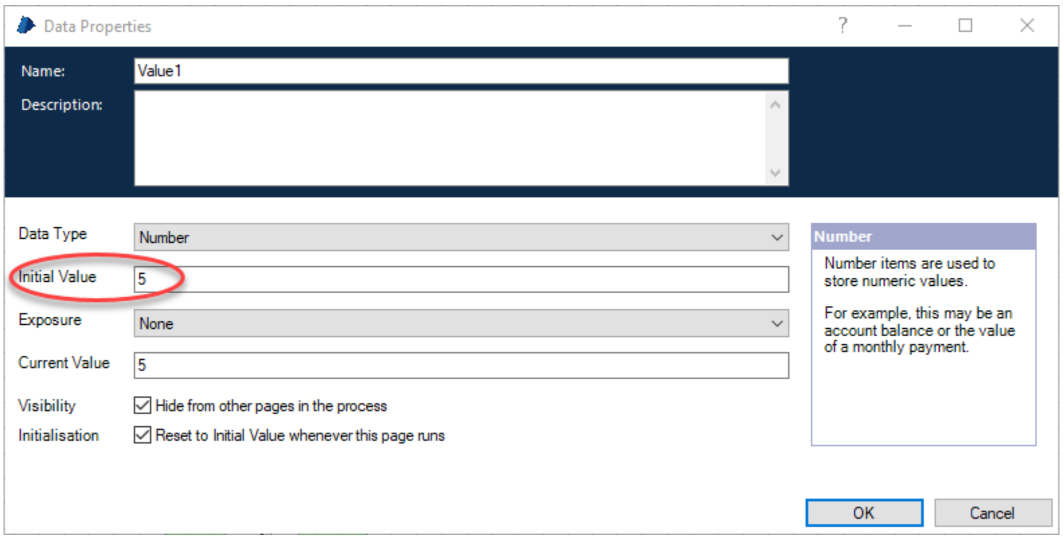
```
Result = FirstValue + SecondValue;
```

Parameters

In:FirstValue - System.Decimal
In:SecondValue - System.Decimal
Out:Result - System.Decimal

Figure 14 Code Stage Parameters in use

We can now, modify the values of our data items and those values will then be passed into our code stage. The benefit here is that it is easier to change the values of the data items than it is to change the actual code in the code stage. So, if we set the initial value, by double-clicking the data item and place a numeric value into the Initial Value field of our data items as shown below, we can predict what the result will be, but at the same time, it's easier to maintain.



Data Properties

Name: Value1

Description:

Data Type: Number

Initial Value: 5

Exposure: None

Current Value: 5

Visibility: ☑ Hide from other pages in the process

Initialisation: ☑ Reset to Initial Value whenever this page runs

Number

Number items are used to store numeric values.

For example, this may be an account balance or the value of a monthly payment.

Figure 15 Parameters with values

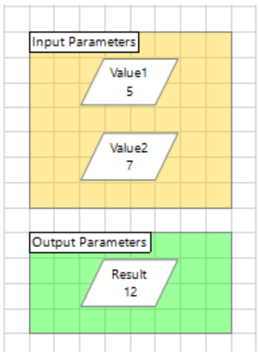
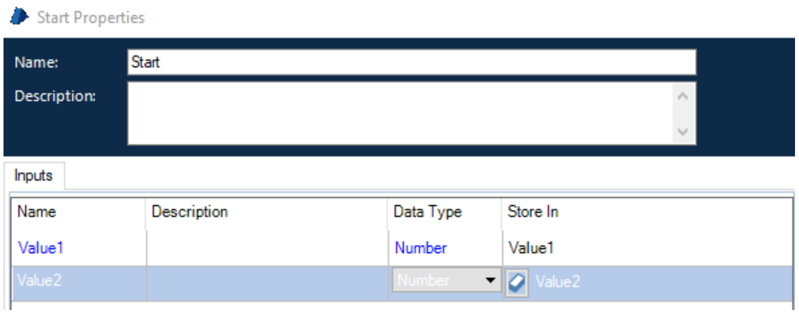


Figure 16 Parameters with values

We have made this action almost maintenance free and at the same time increased its usability. We can now take this a stage further by creating input parameters for the whole action. This is done by double-clicking on the Start stage.



Name: Start
Description:

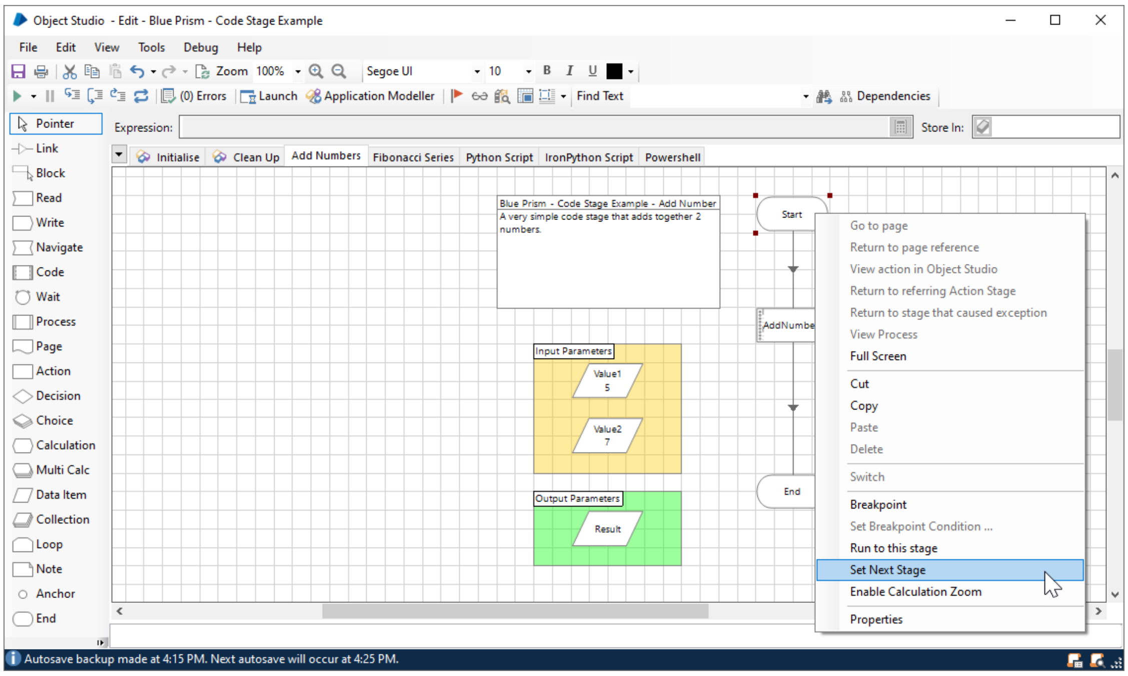
Name	Description	Data Type	Store In
Value1		Number	Value1
Value2		Number	Value2

Figure 17 Stage Parameters

As you can see, the stage when it is called now from a process, will have 2 input values. These values will be stored in the data items we created. At this point, we have reached maximum reusability for our stage and minimum maintenance. This has been accomplished through the use of parameters, which allow different values to be passed in without having to change anything.

Executing Our Code Stage

We need to have a starting position so we might test our business object. So, right-click the mouse on the start stage of the object on our Add Numbers page. Select *Set Next Stage*.



Blue Prism - Code Stage Example - Add Number
A very simple code stage that adds together 2 numbers.

Autosave backup made at 4:15 PM. Next autosave will occur at 4:25 PM.

Figure 18 Set Next Stage

This will set the colour of the Start stage and we can now use our controls to execute our code stage.

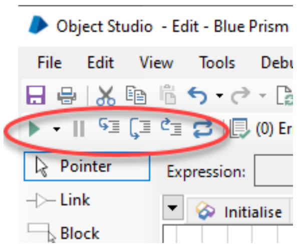


Figure 19 Debugging Controls

The controls shown above, from left to right are:

- Play (Run)
- Pause
- Step
- Step Over
- Step Out
- Reset

Debug: Play

There is some additional functionality to the Play (Run) button. You may see that is has a small downward pointing triangle on it and if this is clicked you will see the following:

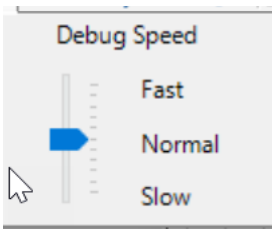


Figure 20 Debug Speed

It is possible to debug at another speed, whether that be fast or slow, it aids debugging as it allows you to speed up the running of your objects actions, over areas you are confident are working correctly. Now, something to be aware of, there is no debugging or stepping through the actual code you have written at this time. If you need to debug values of items then it makes sense to use data items and pass them in as parameters, they can be removed later when everything is working how you expect. We will look at an example of this later.

Debug: Pause

Exactly as you would expect, this allows a running process to be halted. Clicking on the play button again resumes execution from the current selected location.

Debug: Step

In order to work through your process, it helps to be able to step through and monitor each stage one at a time. This button requires a manual click for each stage in the process. Note that any processes that involve other pages, or object will jump to those stages and you will be able to work your way through them individually.

Debug: Step Over

Once you have satisfactorily debugged your process, you may not want to keep stepping through it. The step over button allows a whole stage to be run, without interaction from the user.

Debug: Step Out

If you are in a stage and want to move back to the calling routing, you can click on step out and the remaining steps will be executed, and you will be returned to the step following the one you just entered.

Debug: Reset

This resets the state of any business object or stage. This can also be used to refresh changes in other parts of Blue Prism. However, changes made to web API objects will require the object to be saved and reloaded. This was the case to version 6.5.

Code Stage 2 – Something a little more useful

Our previous code stage was not really very useful for anything beyond demonstrating the environment and how the various tools are used.

So, lets do something a little more interesting, lets calculate the Fibonacci series between 0 and 50 and let's put them into a Blue Prism Collection. So, we have 2 things to do here, build the series of numbers and then format their output. We can use the same object as before, but we will add a new page this will appear as a new function of our business object when we get to using it in a process.

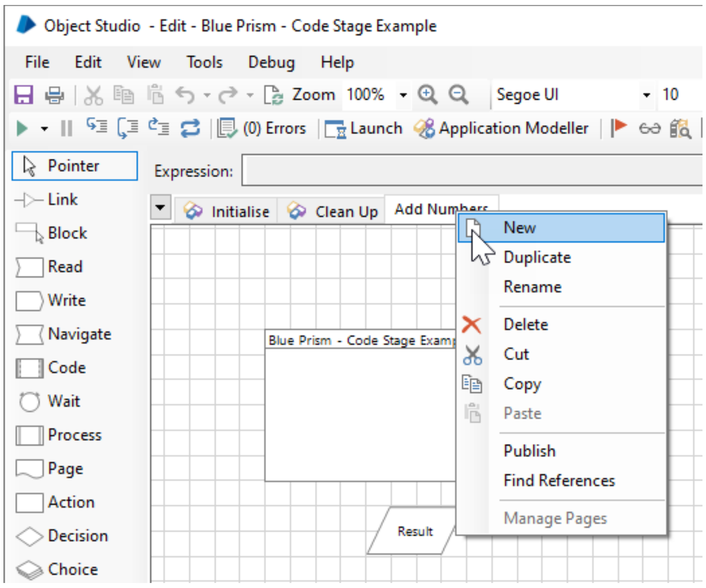
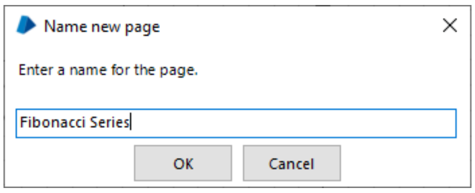


Figure 21 Create New Page

As this is a slightly more complex page, its worth discussing how is it best to write code stages. If your code stage is just a few lines and its code you are familiar with, then go ahead just write it into the Blue Prism code stage. If, however, the code is long, refers to other methods, it just might be worth making use of VS Code or even Visual Studio itself. The reason for this is that, Visual Studio is an excellent development environment and has all the tools you could possibly ever need for software development. The main benefit to using it here though lies in the fact that you can test the logic of your code before you move it to Blue Prism. This way you know your code works and all you then must do is make it work inside of Blue Prism.

So, lets create our new page



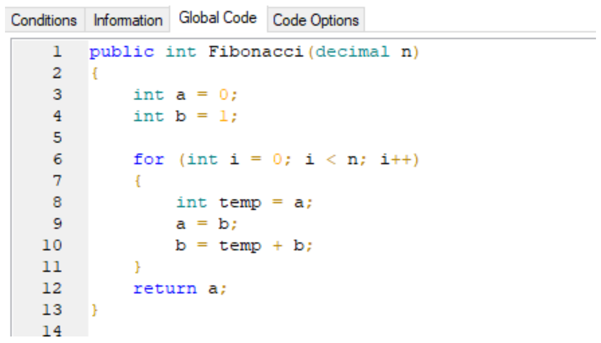
Enter a name for the page.

Fibonacci Series

Figure 22 Name our new page

Now, for this code stage, we are going to use a global code item, or method. Our method is going to do the Fibonacci heavy lifting for us, but because this function is iterative, it needs to be called repeatedly. Iterative coding can be complex, but it can also be very efficient.

So, we are going to take a quick jump back to our Initialize page and look at the properties of our object, we will write the code in the global code window.


Conditions | Information | Global Code | Code Options

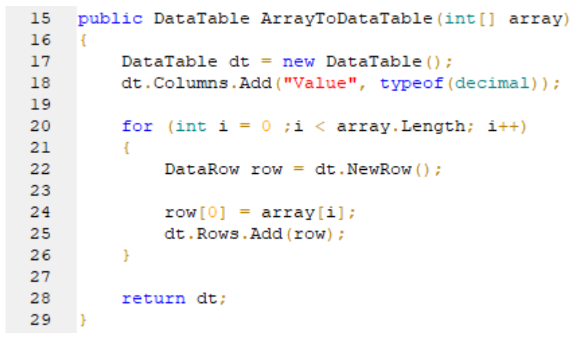
```
public int Fibonacci(decimal n)
{
    int a = 0;
    int b = 1;

    for (int i = 0; i < n; i++)
    {
        int temp = a;
        a = b;
        b = temp + b;
    }
    return a;
}

```

Figure 23 The Fibonacci Code

Now this code is fine for generating the Fibonacci sequence and it also demonstrates another nice concept. That of *single responsibility*.

> NOTE: Single responsibility is the 'S' in SOLID. For more information search for SOLID and the name Robert Martin (Uncle Bob). The benefit this provides is that this code does one thing and one thing only.

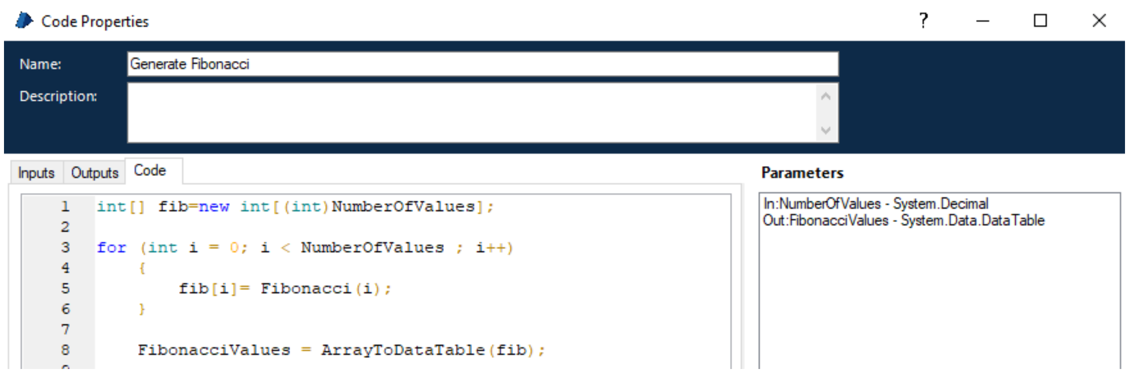
It would have been easy to add in the code to convert the array but that is better if it's in its own method too. So, here is our code to convert the array into a *datatable*. The *datatable* being the complimentary data type to a Blue Prism collection. By having the 2 methods separate, it is easier to maintain, after all, you might not want to always place your Fibonacci values into a collection. You might have other plans for that data.



```
public DataTable ArrayToDataTable(int[] array)
{
    DataTable dt = new DataTable();
    dt.Columns.Add("Value", typeof(decimal));

    for (int i = 0 ;i < array.Length; i++)
    {
        DataRow row = dt.NewRow();

        row[0] = array[i];
        dt.Rows.Add(row);
    }

    return dt;
}
```

Figure 24 ArrayToDataTable Conversion Method

So, now we have our two global methods, we now need to return to our coding stage and make use of them.


Code Properties

Name: Generate Fibonacci
Description:

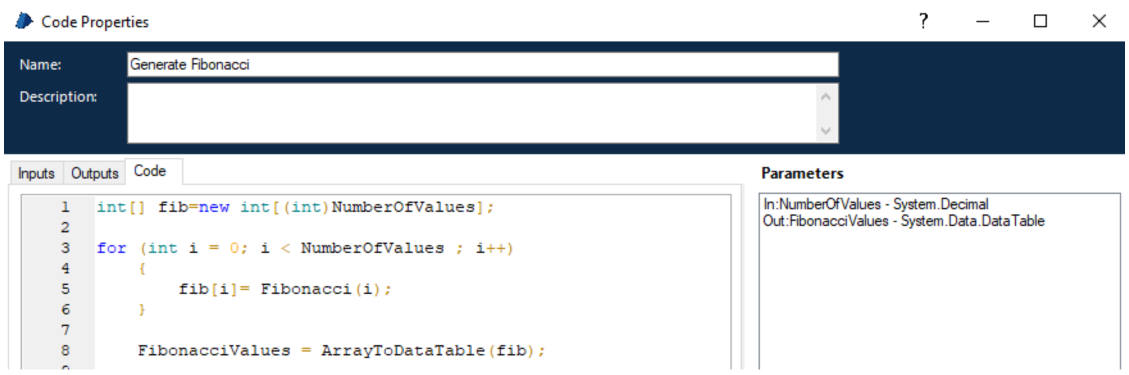
Inputs | Outputs | Code

```
int[] fib=new int[(int)NumberOfValues];

for (int i = 0; i < NumberOfValues ; i++)
    {
        fib[i]= Fibonacci(i);
    }

    FibonacciValues = ArrayToDataTable(fib);
```

Parameters
In:NumberOfValues - System.Decimal
Out:FibonacciValues - System.Data.DataTable

Figure 25 The Code Stage

So, as you can see, we are preparing an integer array, and building a Fibonacci series using the value held in the *NumberOfValues* data item. This parameter is created in the same way we created it in Add Numbers. You can also see that we have an output parameter declared, called *FibonnaciValues*. As you can see in the parameters list on the right of the image above, it is assuming the type of DataTable.



Code Properties

Name: Generate Fibonacci

Description:

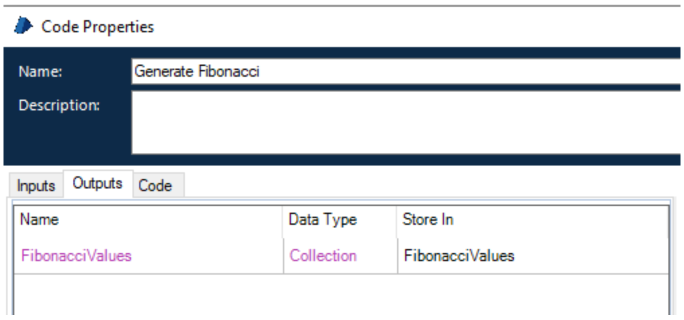
Inputs | Outputs | Code

```
int[] fib=new int[(int)NumberOfValues];

for (int i = 0; i < NumberOfValues ; i++)
    {
        fib[i]= Fibonacci(i);
    }

    FibonacciValues = ArrayToDataTable(fib);
```

Parameters

In:NumberOfValues - System.Decimal
Out:FibonacciValues - System.Data.DataTable

Figure 26 Code Stage Parameters



Code Properties

Name: Generate Fibonacci

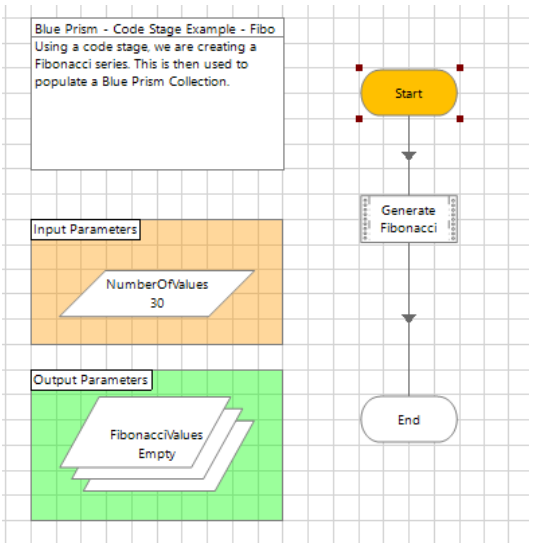
Description:

Inputs | Outputs | Code

Name	Data Type	Store In
FibonacciValues	Collection	FibonacciValues

Figure 27 Code Stage Parameters

It is worth being aware that Number values, are translated to System.Decimal in C#. So, where we have used *NumberOfValues* above to define the size of the array, it is necessary to cast this value to an integer. Our input parameter is defined as a number and our output parameter is defined as a Collection. This output maps nicely to the datatable that we used in our code. All there is to do now, is run our code and look at the outputs.



Blue Prism - Code Stage Example - Fibo
Using a code stage, we are creating a Fibonacci series. This is then used to populate a Blue Prism Collection.

Figure 28 Start

By right-clicking and choosing Set Next Stage, we can position our code execution point at the start stage. Note that currently our collection *FibonacciValues* is still empty. Our NumberOfValues has its initial value set to 30. You can set this to your own value, bearing in mind there will be numerical limits to the results.

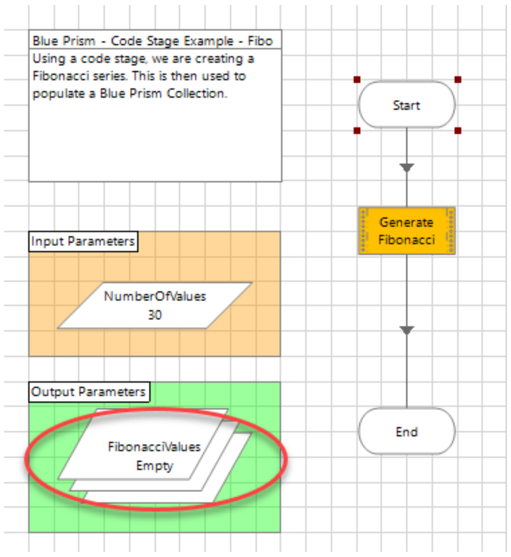

Blue Prism - Code Stage Example - Fibo
Using a code stage, we are creating a Fibonacci series. This is then used to populate a Blue Prism Collection.
Input Parameters
NumberOfValues
30
Output Parameters
FibonacciValues
Empty
Start
Generate Fibonacci
End

Figure 29 Execute

By clicking on the step button we can advance our execution point to our code stage. At the next press of the step button our code will be executed, and the results will be seen in the FibonacciValues collection data item.

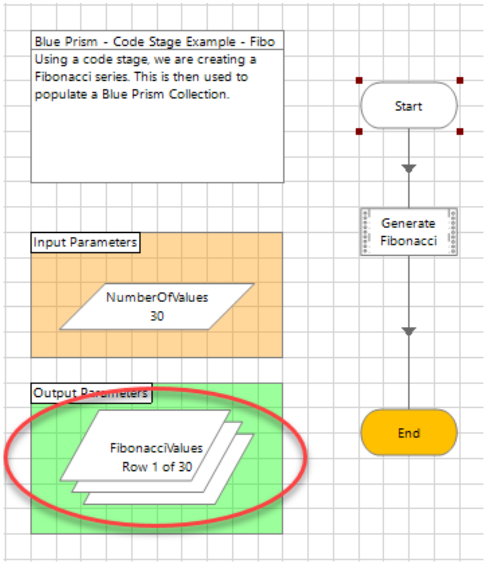
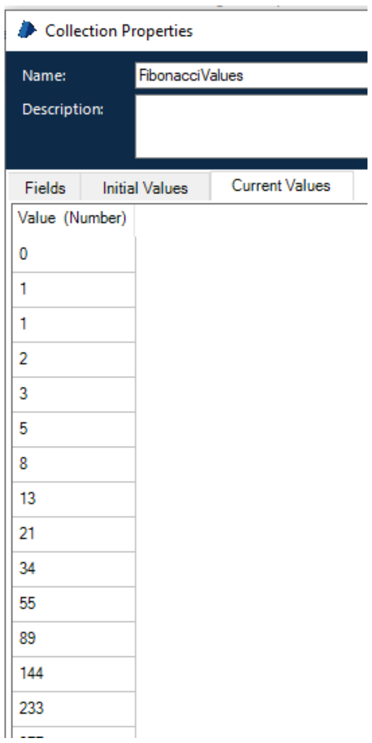

Blue Prism - Code Stage Example - Fibo
Using a code stage, we are creating a Fibonacci series. This is then used to populate a Blue Prism Collection.
Input Parameters
NumberOfValues
30
Output Parameters
FibonacciValues
Row 1 of 30
Start
Generate Fibonacci
End

Figure 30 Results

Our code has successfully run and returned us the first 30 Fibonacci series. This is verified by examining the Collection data item FibonacciValues.


Collection Properties

Name: FibonacciValues

Description:

Value (Number)
0
1
1
2
3
5
8
13
21
34
55
89
144
233

Figure 31 Fibonacci Series Collection Data Item

Going Further

So, now we know how to do this with a code stage, what else might we do? How about call a Python or a Powershell script? What about dynamic languages, can we use Blue Prism there? Let's look.

The interesting thing here is that to execute either of these script types, we don't have to use a code stage. We can make use of the Environment Utility VBO that is supplied with Blue Prism, however if you really want to use a code stage go ahead and knock yourself out.

Python Scripts

Blue Prism comes complete with a set of VBO's. While these are installed on your PC when you install Blue Prism, they won't be installed in your environment until you specifically do so. One of those VBO's is called Utility – Environment. This asset has a heap of functionality in relation to the windows environment on the machine which the process is running. A whole host of functionality for managing processes, of which Start Process is the one we are interested in.

In order to use this VBO asset, its necessary to download the VBO, if you have never done this before then here we go. From the main control page of Blue Prism, click the file menu and choose import.

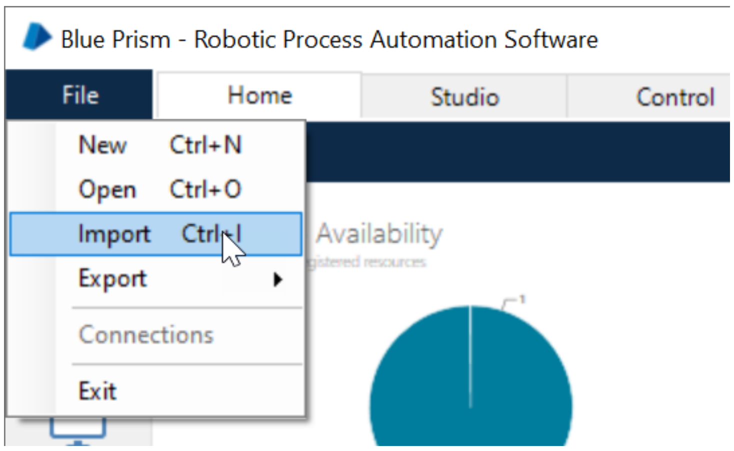


Figure 32 Blue Prism File Import Menu

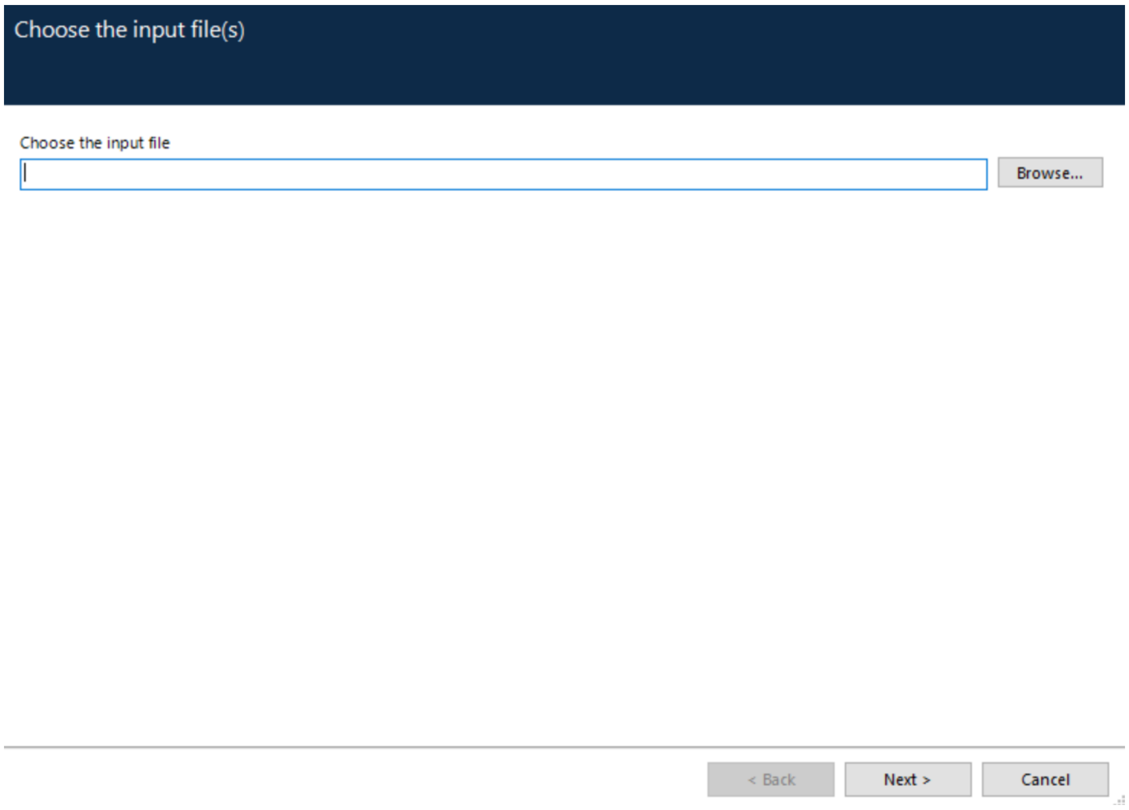
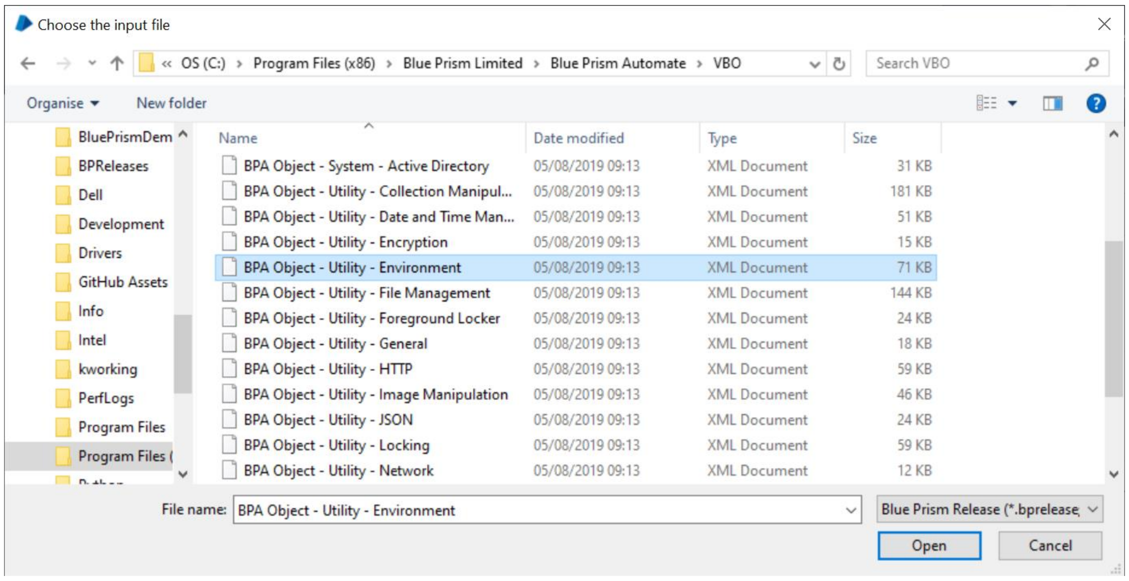


Figure 33 Blue Prism Import File Dialog

Click on Browse… and the following file dialog will appear. You will need to navigate to the following location, if Blue Prism is installed on your C: drive.

C:\Program Files (x86)\Blue Prism Limited\Blue Prism Automate\VBO



Name	Date modified	Type	Size
BPA Object - System - Active Directory	05/08/2019 09:13	XML Document	31 KB
BPA Object - Utility - Collection Manipul...	05/08/2019 09:13	XML Document	181 KB
BPA Object - Utility - Date and Time Man...	05/08/2019 09:13	XML Document	51 KB
BPA Object - Utility - Encryption	05/08/2019 09:13	XML Document	15 KB
BPA Object - Utility - Environment	05/08/2019 09:13	XML Document	71 KB
BPA Object - Utility - File Management	05/08/2019 09:13	XML Document	144 KB
BPA Object - Utility - Foreground Locker	05/08/2019 09:13	XML Document	24 KB
BPA Object - Utility - General	05/08/2019 09:13	XML Document	18 KB
BPA Object - Utility - HTTP	05/08/2019 09:13	XML Document	59 KB
BPA Object - Utility - Image Manipulation	05/08/2019 09:13	XML Document	46 KB
BPA Object - Utility - JSON	05/08/2019 09:13	XML Document	24 KB
BPA Object - Utility - Locking	05/08/2019 09:13	XML Document	59 KB
BPA Object - Utility - Network	05/08/2019 09:13	XML Document	12 KB

File name: BPA Object - Utility - Environment

Figure 34 Locate the Utility Environment VBO Asset

If you select the Environment VBO, as shown, and click open, the dialog will disappear, and the previous file import dialog will be exposed. Click next here. You should see a progress bar move across the dialog and it should display the message that the release has been imported into the database. You may now click on finish. Repeat this procedure for the Utility – File Management VBO asset. We will use this to read in our text file.

So, now we return to our business object and create the action that will execute our python script. Now, as we have been working on the Fibonacci series, we will continue with this except this time our series will be calculated by a Python script and written to a local text file on the hard drive. We will then read in that text file and populate a Blue Prism collection with the values in that text file. As we are going to call a python script it will be necessary to have Python installed. This was tested with Python 3.8 which is available as a Windows installer from https://www.python.org our installation location is non-standard so you may need to alter pathnames to meet your requirements.

So, as we have done before, create a new page, and ours is called Python Script. Feel free to name yours however you wish. Add an action stage to our newly created page.

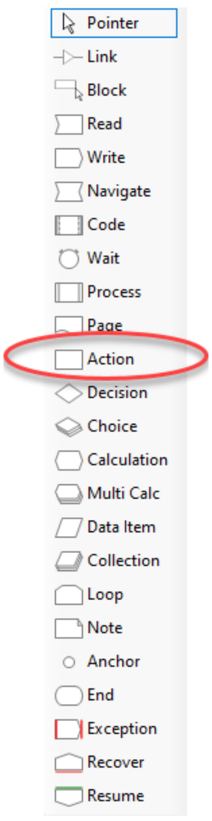
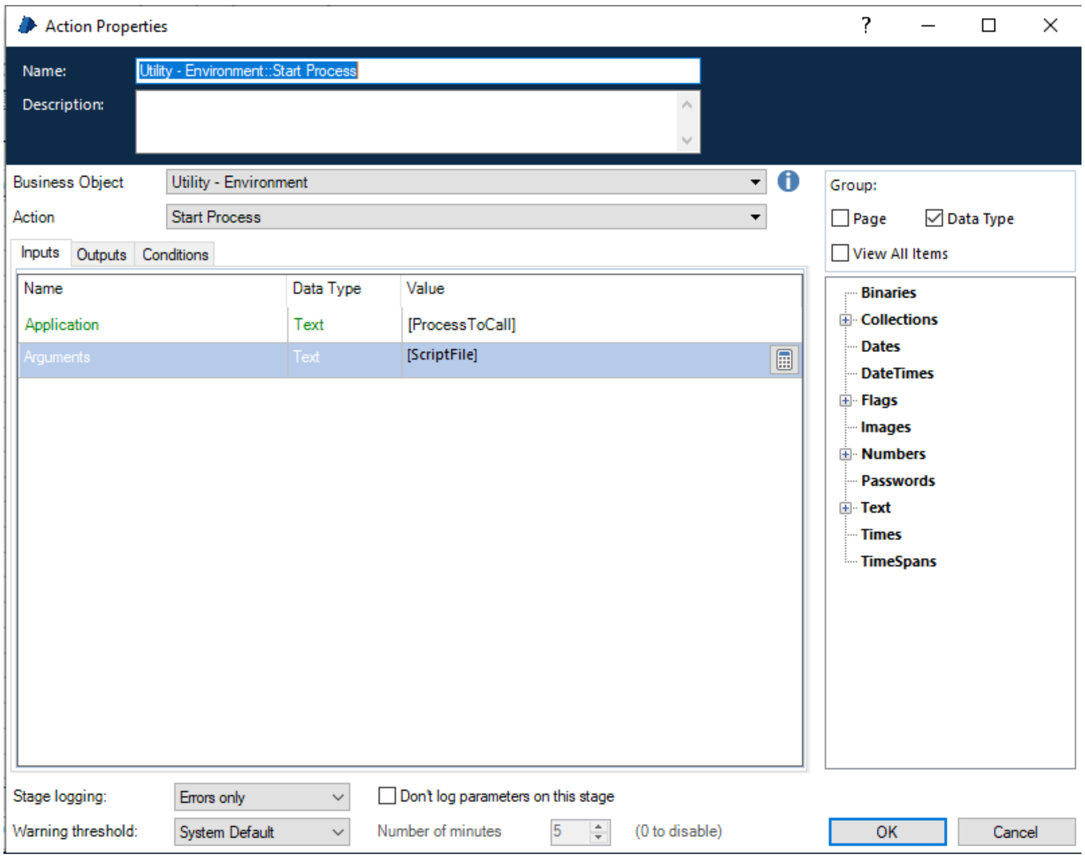


Figure 35 Blue Prism Toolbox

Set the business object drop down to Utility – Environment and the action to Start Process. Set the input values appropriate for your system. There are no output parameters for this stage. As you can see we are using Data Items to carry our parameter values.



Action Properties

Name: Utility - Environment::Start Process

Description:

Business Object: Utility - Environment

Action: Start Process

Inputs | Outputs | Conditions

Name	Data Type	Value
Application	Text	[ProcessToCall]
Arguments	Text	[ScriptFile]

Stage logging: Errors only

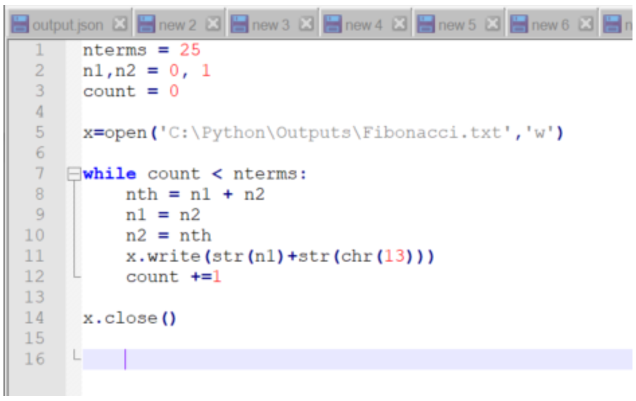
Don't log parameters on this stage

Warning threshold: System Default

Number of minutes 5 (0 to disable)

Figure 36 Action Stage Input Parameters

Our python script is as follows:



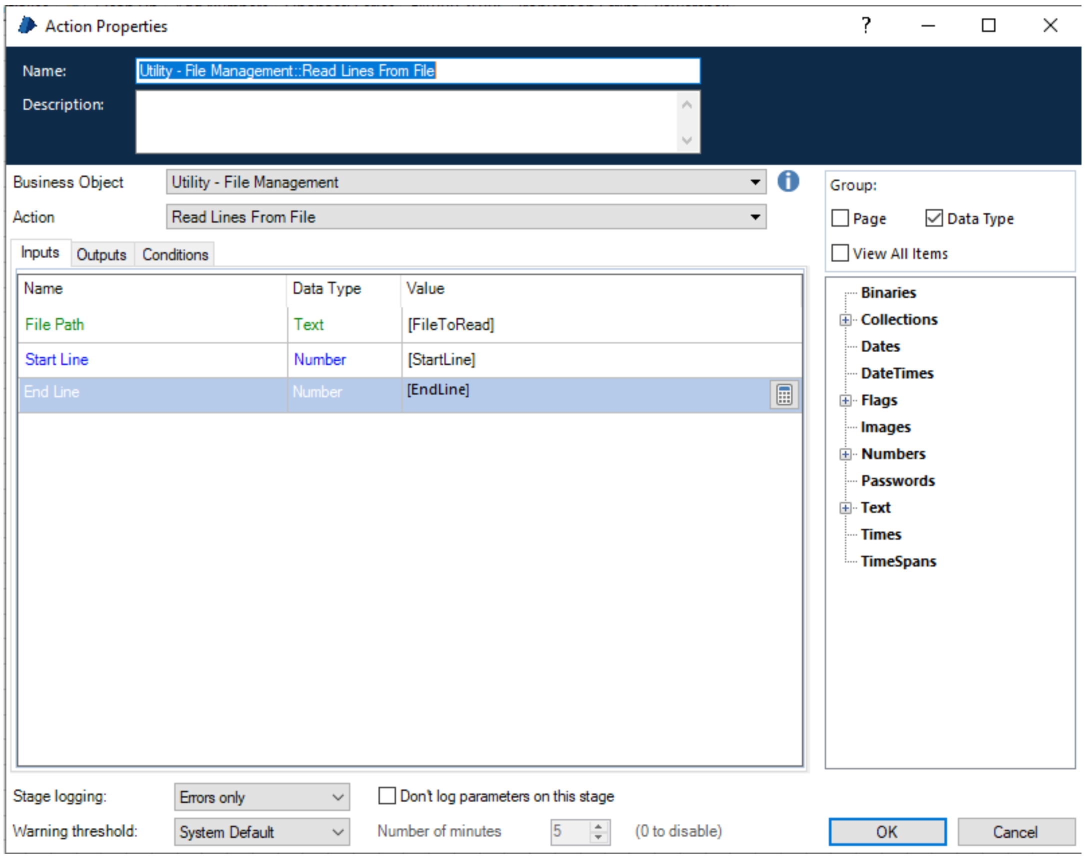
```python
nterms = 25
n1,n2 = 0, 1
count = 0

x=open('C:\Python\Outputs\Fibonacci.txt','w')

while count < nterms:
    nth = n1 + n2
    n1 = n2
    n2 = nth
    x.write(str(n1)+str(chr(13)))
    count +=1

x.close()

```

Figure 37 Fibonacci Series Python Script

This file, Fibonacci.py, can be found at the GitHub site for the accompanying assets for this exercise. The python script has a path where a file is written to, this may need to be altered for your environment.

The next step is to read in the text file and populate a Blue Prism Collection. We can do this with another installed VBO. The Utility – File Management will enable us to read in the file contents and process it however we wish.

So, let's look at our object now. Here are the settings for the File Management Utility VBO, to read the file into a Blue Prism Collection.



Action Properties

Name: Utility - File Management::Read Lines From File

Description:

Business Object: Utility - File Management

Action: Read Lines From File

Inputs | Outputs | Conditions

Name	Data Type	Value
File Path	Text	[FileToRead]
Start Line	Number	[StartLine]
End Line	Number	[EndLine]

Group: Page, ☑ Data Type, View All Items

Binaries, Collections, Dates, DateTimes, Flags, Images, Numbers, Passwords, Text, Times, TimeSpans

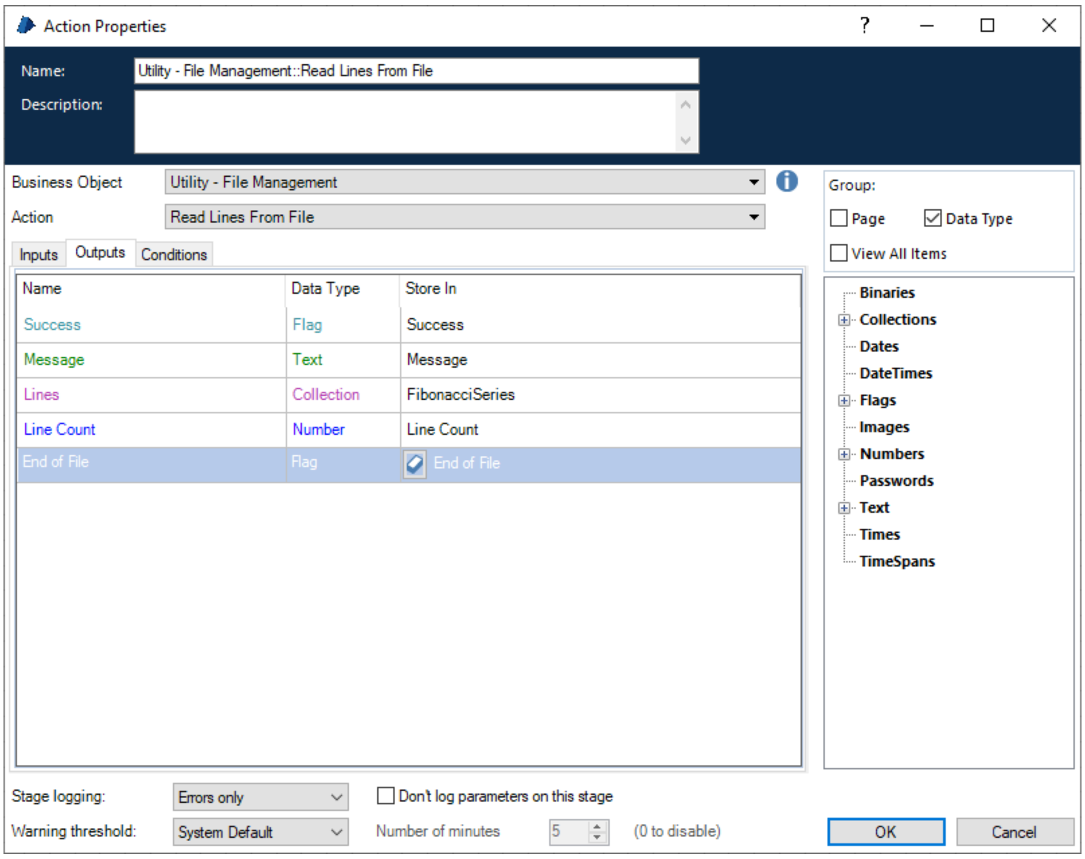
Stage logging: Errors only

Don't log parameters on this stage

Warning threshold: System Default — Number of minutes 5 (0 to disable)

OK | Cancel

Figure 38 File Management Utility Read Lines from File – Input and Output Parameters



Action Properties

Name: Utility - File Management::Read Lines From File

Description:

Business Object: Utility - File Management

Action: Read Lines From File

Inputs | Outputs | Conditions

Name	Data Type	Store In
Success	Flag	Success
Message	Text	Message
Lines	Collection	FibonacciSeries
Line Count	Number	Line Count
End of File	Flag	End of File

Group: Page, ☑ Data Type, View All Items

Binaries, Collections, Dates, DateTimes, Flags, Images, Numbers, Passwords, Text, Times, TimeSpans

Stage logging: Errors only

Don't log parameters on this stage

Warning threshold: System Default — Number of minutes 5 (0 to disable)

OK | Cancel

Figure 39 File Management Utility Read Lines from File – Input and Output Parameters

Our process is now a little more complicated than before we have stages for executing the Python script and producing the output file. The next stage reads the file into Blue Prism and populates a collection data item with the values found.

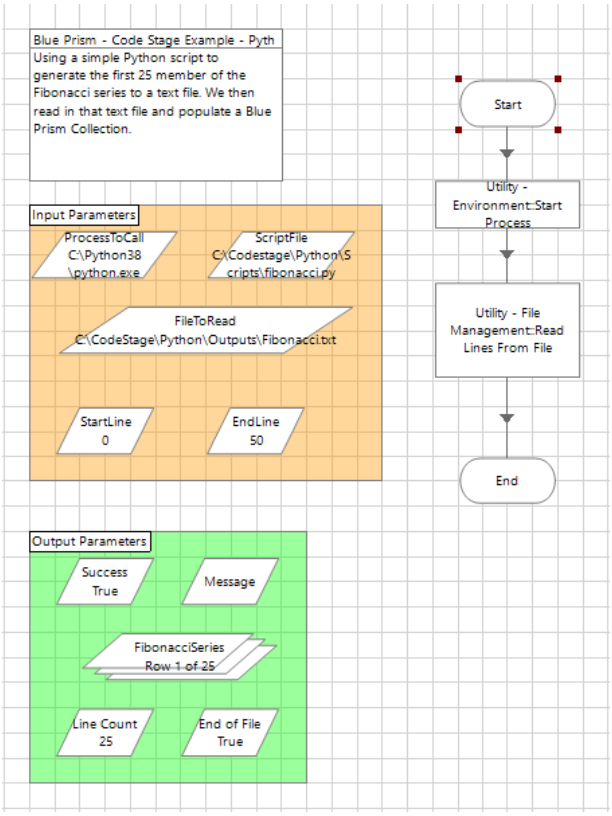
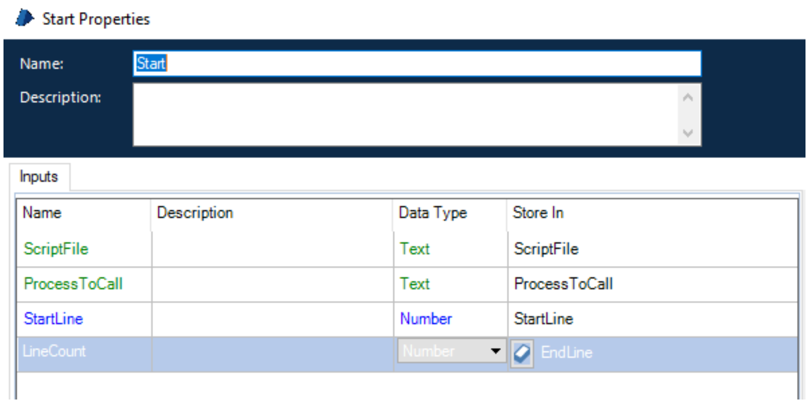

Blue Prism - Code Stage Example - Pyth
Using a simple Python script to generate the first 25 member of the Fibonacci series to a text file. We then read in that text file and populate a Blue Prism Collection.
Input Parameters
ProcessToCall C:\Python38\python.exe
ScriptFile C:\Codestage\Python\Scripts\fibonacci.py
FileToRead C:\CodeStage\Python\Outputs\Fibonacci.txt
StartLine 0
EndLine 50
Output Parameters
Success True
Message
FibonacciSeries Row 1 of 25
Line Count 25
End of File True
Start
Utility - Environment:Start Process
Utility - File Management:Read Lines From File
End

Figure 40 Our Python Script Action

From the above image you can see the action has executed successfully, the success variable is set to true, message is empty which confirms there were no errors, FibonacciSeries is populated with 25 items and the end of file wasn't reached but we know there were only 25 numbers to read in and finally the line count is set on 25.

This is just a method by which we can execute a Python script from inside of Blue Prism, technically it isn't being executed inside of Blue Prism, but you might have the circumstances where this is exactly what you need.

Now to make maximum use of the action, we would add parameters to the start stage that cover all the input parameters that we have created. This way, when it is called from a Blue Prism process, we gain maximum benefit from its functionality.



Start Properties

Name: Start

Description:

Inputs

Name	Description	Data Type	Store In
ScriptFile		Text	ScriptFile
ProcessToCall		Text	ProcessToCall
StartLine		Number	StartLine
LineCount		Number	EndLine

Figure 41 Start Parameters for the Action

Using IronPython

Let’s look now at executing a Python script from inside of a code stage in Blue Prism. For this we will use the IronPython extensions. It is recommended for this that you have access to at least Visual Studio Code or Visual Studio itself.

IronPython is an open-source implementation of the Python programming language that is integrated with the .NET framework. More details of IronPython can be found at http://ironpython.net.

IronPython can be installed into Visual Studio through nuget.

We are going to use a variant to our original file, IPFibonacci.py and execute this inside of a Blue Prism Code stage. We need to use a different file as IronPython is based on Python 2.7, whereas our previous Python work was based on the current 3.8 release. The only difference between the 2 files is the way the pathname is constructed.

Now, in order to make this work from Blue Prism we need to add the IronPython.dll assembly to the Global Assembly Cache (GAC). This is done with a utility called *gacutil*. This utility is best called from the Visual Studio Command prompt window that is available under your Visual Studio application group on the windows menu. In order to modify the global assembly cache, it is necessary to execute this command prompt as an administrator. This is done by right clicking the application and choosing the ‘More’ option and then ‘Run as Administrator’.

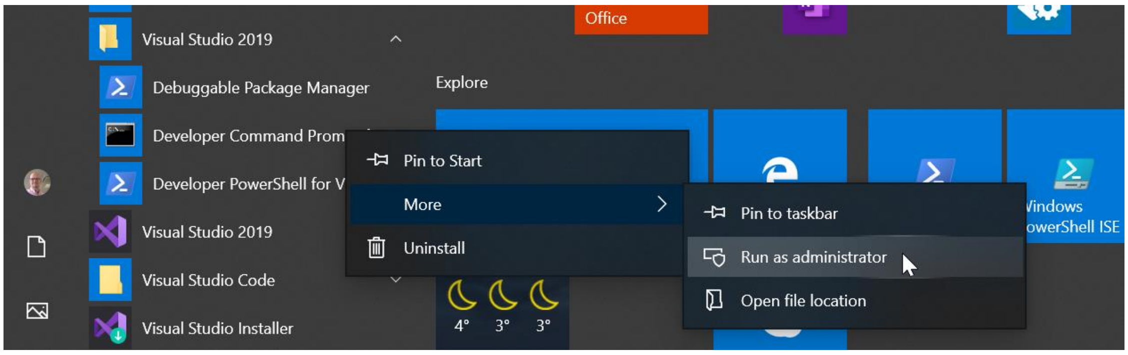


Figure 42 Launching the Developer Command Prompt as Administrator

Once you have the command prompt open you will need to locate the file IronPython.dll. Now the recommended way to develop code stages like this is to code them initially in Visual Studio or Visual Studio Code, VS Code is free and there is the community version of Visual Studio that can be used for non-commercial projects. This allows you to get the code right along with any needed references. Given that some references may need to be downloaded from Nuget, it will make it easier to install the necessary file into the GAC.

So, when you have your prototype application in Visual Studio, get the path for the file IronPython.dll, this will be visible in the references node and then look at the path property. In the developer command window change the directory to this path and then you are ready to install the assembly into the GAC.

This is done with the following command:

Gacutil /i ironpython.dll

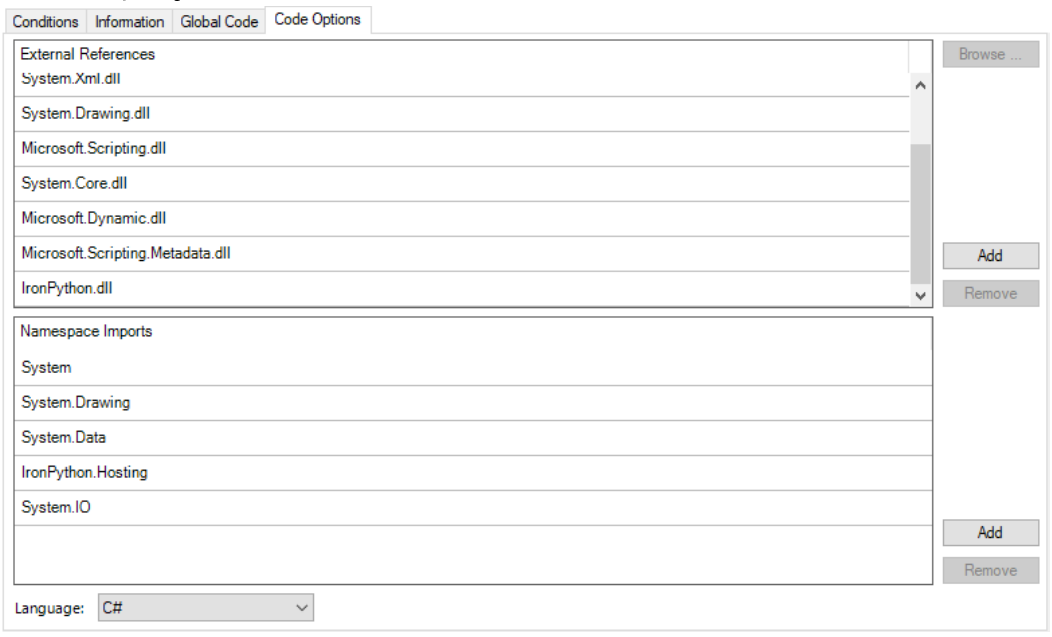
If this is successful, an appropriate message will be displayed, and you can then close the command window.

Iron Python Business Action

Once we are happy that our code is working in Visual Studio, we can take the code across to our Blue Prism business object and place it into the code window. In order to make this code work, we need to tell Blue Prism about the additional assembly references that we require.

Those additional references are as follows:

- Microsoft.Scripting.dll
- System.Core.dll
- Microsoft.Dynamic.dll
- Microsoft.Scripting.Metadata



External References
System.Xml.dll
System.Drawing.dll
Microsoft.Scripting.dll
System.Core.dll
Microsoft.Dynamic.dll
Microsoft.Scripting.Metadata.dll
IronPython.dll

Namespace Imports
System
System.Drawing
System.Data
IronPython.Hosting
System.IO

Language: C#

Figure 43 Code Options for the Iron Python References

With the references in place, we should now be able to run our IronPython action. The complete action is shown in fig.40

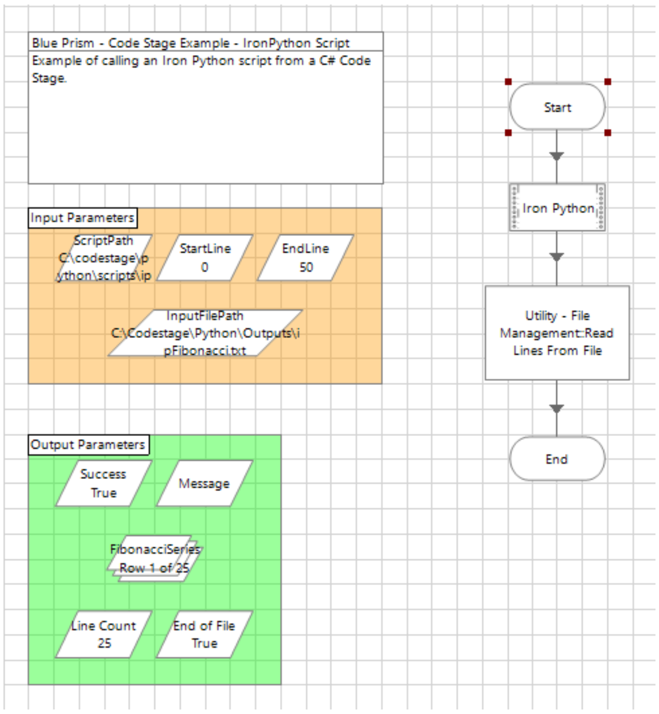
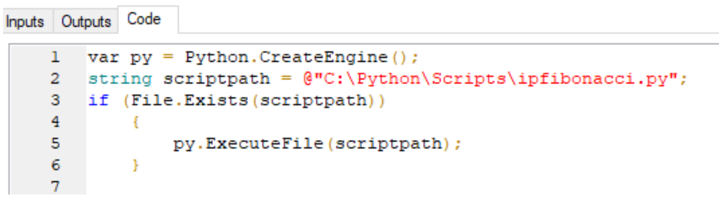

Blue Prism - Code Stage Example - IronPython Script
Example of calling an Iron Python script from a C# Code Stage.
Input Parameters
ScriptPath C:\codestage\python\scripts\ip
StartLine 0
EndLine 50
InputFilePath C:\Codestage\Python\Outputs\ipFibonacci.txt
Output Parameters
Success True
Message
FibonacciSeries Row 1 of 25
Line Count 25
End of File True
Start
Iron Python
Utility - File Management:Read Lines From File
End

Figure 44 The IronPython Action

As you can see it resembles the previous Python script action, indeed there is very little difference in other than how the python code is being executed. The file is created in IronPython and then read into Blue Prism using the same File Management Utility. The FibonacciSeries collection data item can be seen populated with the content of the file as it was previously.

Here is the code that is in use in the IronPython code stage.



```
var py = Python.CreateEngine();
string scriptpath = @"C:\Python\Scripts\ipfibonacci.py";
if (File.Exists(scriptpath))
    {
        py.ExecuteFile(scriptpath);
    }

```

Figure 45 The IronPython Code Stage

We have covered a lot of ground over the last 2 actions. We will now round off with a look into executing a PowerShell script.

Powershell Script

We can call a Powershell script in a similar manner to how we called a Python script. We use the Environment Utility and call the Start Process Action.

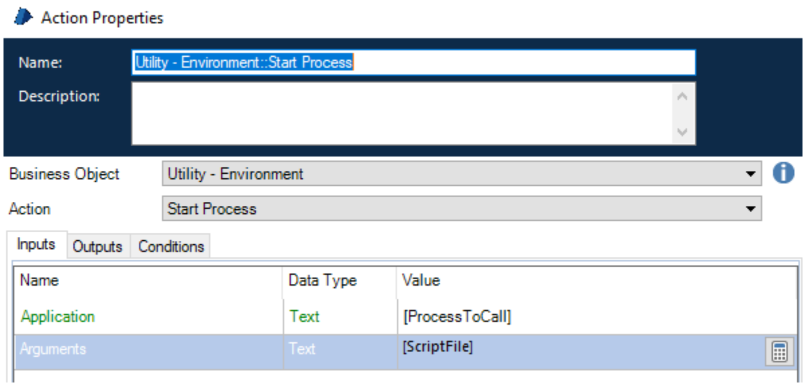
In the example files there is a PowerShell script named directorylisting.ps1. It will perform a directory listing

$a = dir "C:\\program files (x86)\\Blue Prism Limited\\Blue Prism Automate" -Recurse

$a | out-file -encoding ascii c:\\codeexamples\\powershell\\outputs\\directorylisting.txt

This script produces a recursive directory listing of the Blue Prism Automate directory and stores it in a variable. That variable value is then piped into another PowerShell command to output it into a text file.

Here's our Action to execute the Powershell script.



Action Properties

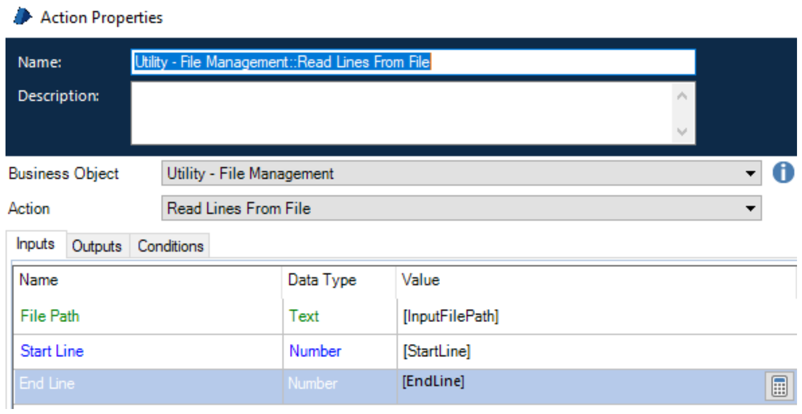
Name:	Utility - Environment::Start Process
Description:	
Business Object	Utility - Environment
Action	Start Process

Inputs | Outputs | Conditions

Name	Data Type	Value
Application	Text	[ProcessToCall]
Arguments	Text	[ScriptFile]

Figure 46 Start Process – Executing a Powershell Script

When our script executes, it will write the output file into the c:\codeexamples\powershell\outputs folder. The next stage of our action will read in the listing file and place it into a Blue Prism collection.



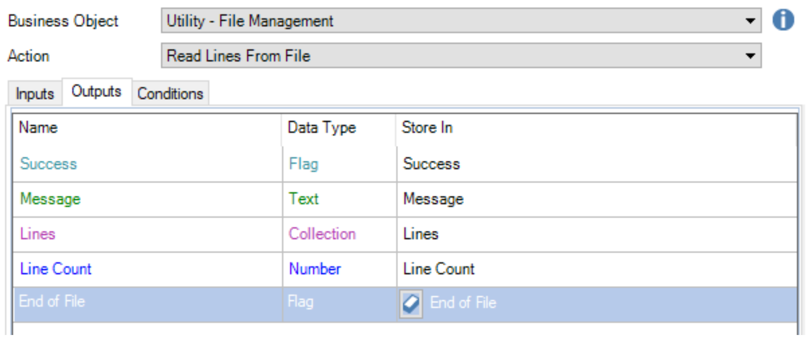
Action Properties

Name:	Utility - File Management::Read Lines From File
Description:	
Business Object	Utility - File Management
Action	Read Lines From File

Inputs | Outputs | Conditions

Name	Data Type	Value
File Path	Text	[InputFilePath]
Start Line	Number	[StartLine]
End Line	Number	[EndLine]

Figure 47 Read File – Input Parameters



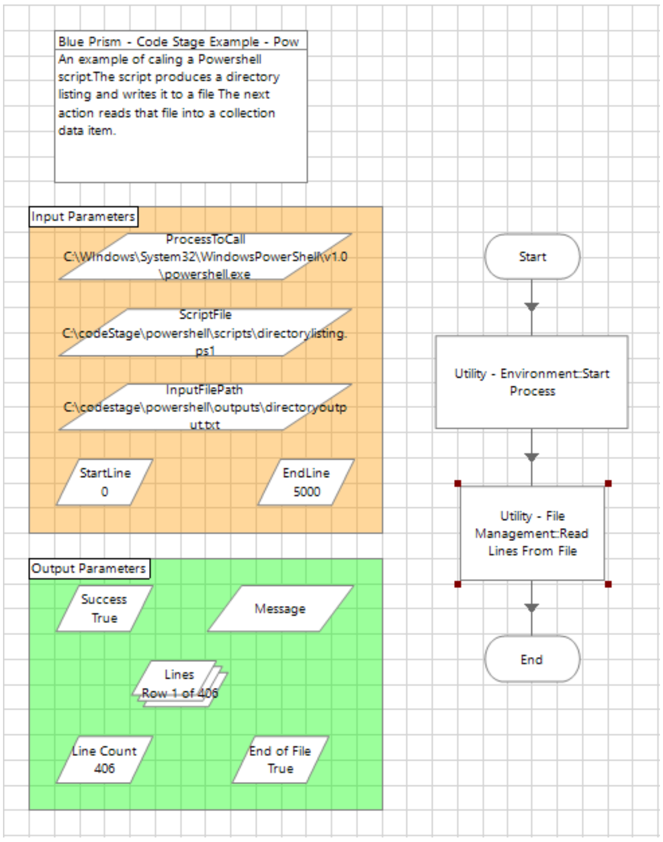
Business Object	Utility - File Management
Action	Read Lines From File

Inputs | Outputs | Conditions

Name	Data Type	Store In
Success	Flag	Success
Message	Text	Message
Lines	Collection	Lines
Line Count	Number	Line Count
End of File	Flag	End of File

Figure 48 Read File – Output Parameters

Finally, the complete Powershell action.


Blue Prism - Code Stage Example - Pow
An example of caling a Powershell script The script produces a directory listing and writes it to a file The next action reads that file into a collection data item.

Input Parameters
ProcessToCall
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ScriptFile
C:\codeStage\powershell\scripts\directorylisting.ps1
InputFilePath
C:\codestage\powershell\outputs\directoryoutput.txt
StartLine
0
EndLine
5000
Output Parameters
Success
True
Message
Lines
Row 1 of 406
Line Count
406
End of File
True
Start
Utility - Environment::Start Process
Utility - File Management::Read Lines From File
End

Figure 49 The Powershell Action

Conclusion

Code stages provide that level of functionality that allows you to go that extra mile. For those things that you just can't make work using the normal Blue Prism tools. At the same time, some things that you may think that could only be accomplished with a code stage can be done using existing Blue Prism tools.

You should also now have a good understanding on the use of parameters and storing their values in data items. This makes your objects more reusable and should help to reduce maintenance.