AUTODESK UNIVERSITY

SK17917

iLogic and the Inventor API

Brian Ekins
Autodesk, Inc.

Description

This course is in response to a lot of the questions that I see on the Inventor Customization forum. I believe many of these questions are the result of a limited understanding of what iLogic really is. iLogic by itself has very limited functionality but it does provide a way of accessing and using the entire Inventor API. Having an understanding of Inventor's API and other development tools that are available can help you to be both a better iLogic developer and also to recognize when iLogic isn't the best tool for a job and when it will be better to use something else.

Learning Objectives

- Gain a better understanding of what iLogic really is.
- Learn what Inventor's API is and its basic concepts.
- Learn about the available tools for working with the API and writing programs.
- Learn how to use existing VB and VBA samples in iLogic rules.

Your AU Expert

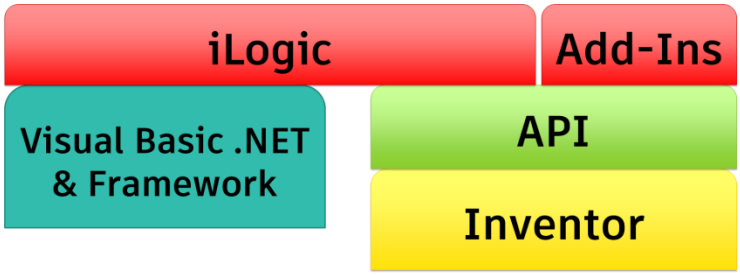
Brian began working in the CAD industry in the early 80's, and he has worked in various positions, including CAD administrator, applications engineer, instructor, CAD API (application programming interface) designer, and consultant. As an applications engineer, he specialized in modeling complex forms (boat hulls, jewelry, shoes, etc.) and customizing the software to better fit customer requirements. He moved into API design, where he designed the programming interface for Solid Edge software, and he has worked for Autodesk, Inc., since 1998, where he is the designer of the programming interfaces for Inventor and is now focusing on Fusion 360 software. Brian also uses the software himself to design and build furniture at home.

This course is in response to a lot of the questions that I see on the Inventor Customization forum. I'm hopeful that having a technical understanding of what iLogic actually is and what other tools are available will help you to better use iLogic or to know when to choose other tools instead when they're more appropriate for your task.

What is iLogic?

Technically, iLogic is an Inventor add-in that provides the ability to write rules to perform configurations of Inventor parts, assemblies, and drawings. A rule is a VB.NET function that typically includes the use of iLogic specific objects and functions. iLogic also provides a development environment where you write and edit your rules.

The diagram below illustrates the dependencies of iLogic. iLogic is an add-in, like any of the other add-ins in Inventor so it is dependent on the Inventor API and the API is dependent on Inventor. iLogic is also dependent on Visual Basic .NET and the .NET Framework.



Does this talk of add-ins, objects, functions, and development environments sound a lot like programming? I have some news for you; if you're writing rules in iLogic you are programming. iLogic attempts to hide that fact from you by providing a simplified interface that wraps a small portion of Inventor's API and it also tries to hide some other typical programming constructs. Even the terminology that iLogic uses tries to hide the fact that you're programming, for example it uses the term "rule" instead of "program" and "snippets" instead of "objects", "methods", "properties" and "functions". Within this simplified iLogic world, things work ok but that world is very small and limited. A majority of tasks that people use iLogic for cannot be done while staying within the small world of iLogic but because an iLogic rule is actually a VB.NET program you also have access to the full Inventor API. The world of the Inventor API is much bigger but is also more complex and requires an understanding of more basic programming techniques.

An analogy of iLogic vs. the Inventor API is a very basic point-and-shoot camera versus a full featured SLR. In many cases the point-and-shoot camera is good enough and is certainly much easier to get started using and may be all you ever need if you just take the occasional picture. But if you end up taking a lot of pictures you soon begin to see its deficiencies and would like to be able to take other types of pictures that you currently can't.

Here's a simple rule that stays within the limited world of iLogic that checks the value of a custom iProperty, changes the value of a parameter, and suppresses and unsuppresses a couple of features.

```
If iProperties.Value("Custom", "PartType") = "One" Then
    Parameter("Length") = 15
    Feature.IsActive("Fillet1") = False
    Feature.IsActive("Hole1") = True
Else If iProperties.Value("Custom", "PartType") = "Two" Then
    Parameter("Length") = 9
    Feature.IsActive("Fillet1") = True
    Feature.IsActive("Hole1") = False
End If

Parameter.UpdateAfterChange = True
```

The rule stays within the iLogic world and uses iLogic specific objects to do all of the work. I've highlighted the iLogic specific statements in the code below.

```
If iProperties.Value("Custom", "PartType") = "One" Then
    Parameter("Length") = 15
    Feature.IsActive("Fillet1") = False
    Feature.IsActive("Hole1") = True
Else If iProperties.Value("Custom", "PartType") = "Two" Then
    Parameter("Length") = 9
    Feature.IsActive("Fillet1") = True
    Feature.IsActive("Hole1") = False
End If

Parameter.UpdateAfterChange = True
```

In the code above "iProperties", "Parameter", and "Feature" are objects defined in the iLogic library and "Value", "IsActive", and "UpdateAfterChange" are properties supported by those objects. The code that isn't highlighted is standard Visual Basic .NET code to add some logic to how the code is executed. The iLogic objects act as simplified wrappers over part of the Inventor API, the Excel API, and a few other things. It's possible to do the same actions that iLogic does by using those technologies directly rather than go through the iLogic.

iLogic has several advantages over some other methods of programming Inventor:

- Simpler to use.
- Is part of the Inventor installation.
- Is simple to write a program that reacts to changes in an Inventor file.
- Very good for part and assembly configuration.
- Simplifies the use of Excel.
- Can access the full Inventor API but then it becomes more complex.

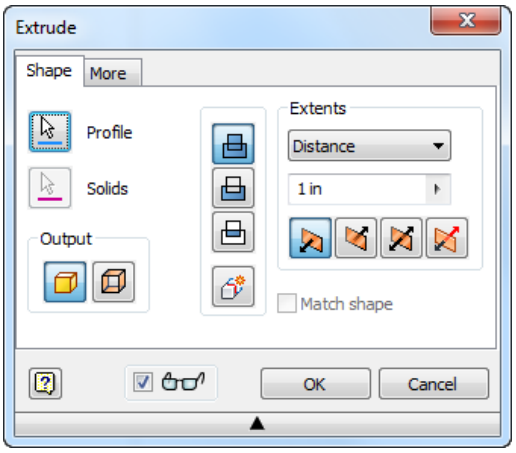
However, there are also disadvantages to using iLogic:

- You are very limited with what can be done with only iLogic.
- The iLogic editor is not very good.
- It's difficult to debug your programs.
- Cannot write add-ins or create custom commands.
- Can't use Apprentice.
- Can't do external batch processing type of programs.

My opinion is that there isn't a perfect solution that solves everything and iLogic is a great solution for certain problems but terrible for others. It's best to have an understanding of the options available and the pros and cons of each. In the rest of this paper I want to briefly discuss some of the other options and spend more time on how some of those options can actually benefit your use of iLogic. I also want to introduce the basic concepts of Inventor's API so hopefully it won't be so intimidating to use when iLogic doesn't have the capabilities you need.

Inventor's User and Programming Interface

As a typical Inventor user, when you use Inventor you tell Inventor what to do by using its "User Interface" (UI) and running commands. The commands guide you through collecting whatever input the command requires and then it performs some type of operation on the model. The command collects the required information and then internally it calls a "request" and passes it the collected information and the request does the actual work to modify the model. For example, the extrude command as shown to the right, guides the user through picking a profile, defining which type of operation to perform, and specifying the extent of the extrusion. Nothing happens to the model until you click "OK" at which point the Extrude commands passes the information its collected to the internal Extrude request and the request does the work to create the extrude feature.



What's described above is the "User" interface to Inventor, but there is also a "Programming Interface". This is referred to as an "Application Programming Interface" or "API". Inventor's programming interface is similar to the user interface in that it provides a way to pass the required information to the same internal requests that the commands use to create the final result. But instead of using a visual user interface you're writing a program to collect and provide the input. It's certainly not as easy as using the user interface but it's important to see and understand that there are parallels between using the User Interface and Programming Interface and looking for these will help you to better understand and use the API.

Programming Tools

No matter if you're using iLogic or writing advanced add-ins you need to write the program code. The user interface to write code is referred to as an "Integrated Development Environment" because it integrates all of the tools needed to write a program into a single application. In some cases, like with iLogic, this interface is delivered with and integrated into the application, which is Inventor in this case. Visual Basic for Applications" (VBA) is another IDE that is provided with Inventor.

The iLogic development environment is one of the weak points of iLogic. Besides providing easy access to snippets, it provides very little functionality to help you write or debug your programs. The other IDE's are far more advanced and can significantly increase your productivity as you write programs. Unfortunately, there's not a single obvious choice of an IDE when writing Inventor programs but instead there are several options and each has its own pros and cons. Here's a brief discussion of the most common development environments.

VBA

VBA or "Visual Basic for Applications" is delivered as part of Inventor so you already have it. VBA is especially good for debugging Inventor's API. VBA was designed specifically for the kind of API that Inventor has. The problem with VBA is that it uses an older version of the Visual Basic language that is different from the version of Visual Basic that iLogic uses which is Visual Basic .NET. The languages are mostly the same but the small difference makes it so you can't always just copy and paste a VBA program into your iLogic rule and have it work. In a section below I'll go into the differences between VBA and iLogic and show how to convert VBA programs to .NET so they can be used in iLogic.

Even though programs written in VBA aren't 100% compatible with iLogic I believe it's still useful to have a basic understanding of VBA for a few reasons. First, the majority of existing sample programs for the Inventor API is written in VBA so it's important to be able to take advantage of those samples. Second, VBA is probably the best environment for quickly prototyping functionality to try and figure out how to use the API to accomplish the task you're trying to do.

Visual Studio

Visual Studio is a full professional development environment from Microsoft. It supports several languages including Visual Basic .NET. Visual Studio is not installed as part of Inventor but is something you'll have to get separately. Because Visual Studio uses Visual Basic .NET, any code you write in it will be compatible with iLogic. The Visual Studio code editor is very powerful and is better than the VBA editor. Debugging in Visual Studio is also very good but not as good as VBA, at least not when you're talking about debugging Inventor API code.

The Visual Basic .NET language also provides some very nice functionality that VBA does not have. And with Visual Basic .NET you'll have access to the .NET Framework library which is very powerful and makes a lot of system level functionality like working with files and accessing the registry available and easy to use.

There are several versions of Visual Studio; the full professional version that you'll have to pay for and two free versions. A license of **Visual Studio Professional** is $499. This comes with several languages, including Visual Basic .NET and has all the capabilities you would ever need for programming Inventor. With it you can create standalone executables that connect to Inventor, utilities that use Apprentice, and you can create sophisticated add-ins.

There is also **Visual Studio Community**. This is free and has all of the capabilities that Visual Studio Professional has. Of course since it's free there is a catch and the catch with Visual Studio Community is how it's licensed. It's free for individual developers, classrooms, academic research, open source projects, and organizations with up to 5 users that make less than $1 million dollars in annual revenue. This version of Visual Studio is relatively new.

Finally, there is **Visual Studio Express**. Express is also free but instead of license restrictions it's missing some functionality and doesn't have all of the features that Visual Studio Professional or Community has. These missing features aren't important when programming Inventor except that it makes it more difficult to debug add-ins. The open question with Visual Studio Express is if it will continue to be made available by Microsoft now that Visual Studio Community is available. So far, Microsoft is providing both but the Express version not as easy to find on the Microsoft Website and my guess is that it will be retired at some point in the future so it would be a good idea to download a copy while it is still available.

Writing Code

As I said before, one of the worst things about iLogic is its code editor. It does very little to help you write and validate your code. As discussed above, there are other IDE's but unfortunately

none of the them are the best in all cases. The first problem with using an IDE other than iLogic is that they don't understand the iLogic specific code like that highlighted in the example below. As discussed earlier, the iProperties, Feature, and Parameter objects used here are iLogic specific and aren't understood by VBA or Visual Studio. However, if the rule you're writing is primarily using the Inventor API, VBA and Visual Studio can be a great option to use to write your code. Visual Studio is probably the better choice because it uses Visual Basic .NET which is the same as iLogic. But Visual Studio also has a slightly bigger learning curve to get started than VBA.

```
If iProperties.Value("Custom", "PartType") = "One" Then
    Parameter("Length") = 15
    Feature.IsActive("Fillet1") = False
    Feature.IsActive("Hole1") = True
End If

Parameter.UpdateAfterChange = True
```

iLogic hides some of the basic programming constructs so you need to understand what's really happening to be able to use Visual Studio. When you run your rule, a VB.NET function ends up being executed. iLogic automatically creates and wraps your code in a "Main" function. The program above actually ends up getting run internally as:

```
Public Sub Main()
    If iProperties.Value("Custom", "PartType") = "One" Then
        Parameter("Length") = 15
        Feature.IsActive("Fillet1") = False
        Feature.IsActive("Hole1") = True
    End If

    Parameter.UpdateAfterChange = True
End Sub
```

If you try running this program in Visual Studio or VBA it will fail because it doesn't know about the iLogic iProperties, Parameter, or Feature objects. So you can't run pure iLogic programs in Visual Studio, however, you can run Visual Studio programs in iLogic because your iLogic programs are really VB.NET programs. And you can convert any VBA program to be VB.NET compatible so they can also be used.

The Basics of Inventor's API

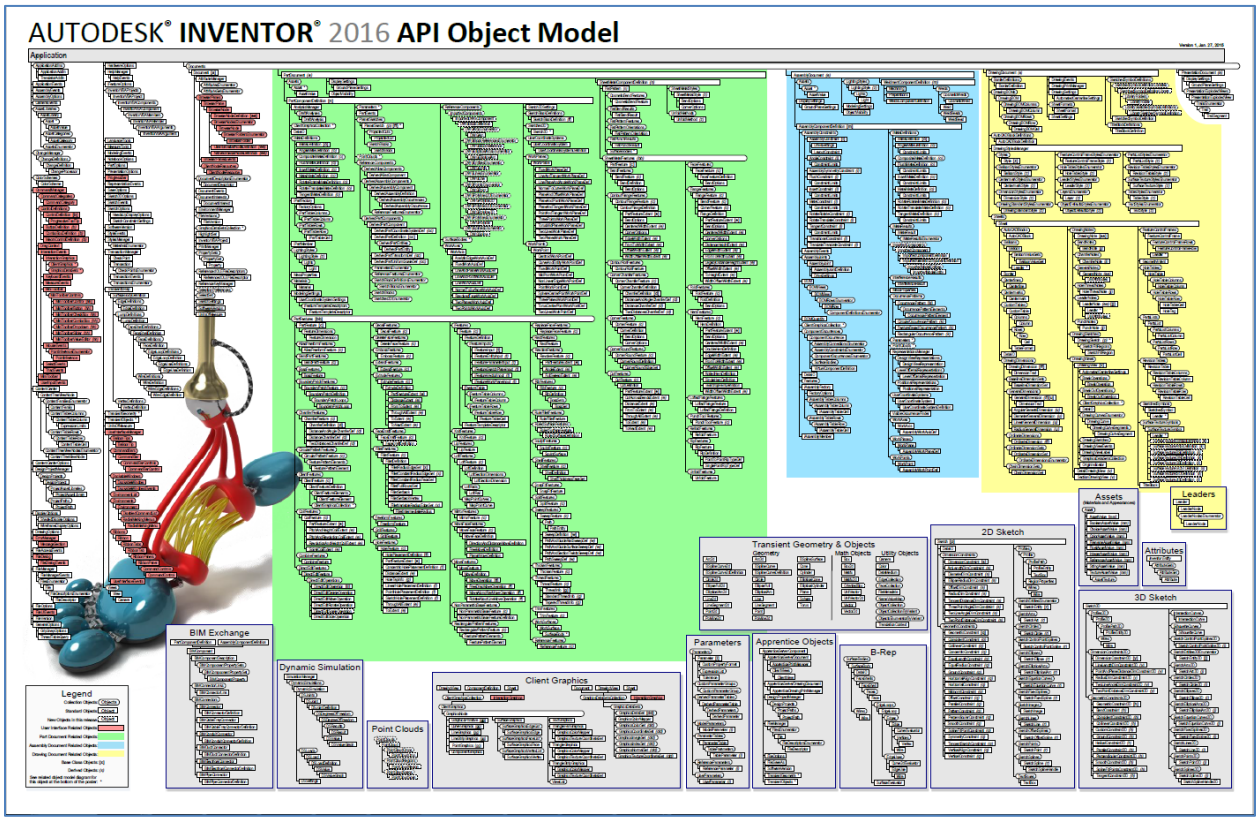
Inventor's API is very powerful and provides the ability to do most of the things you can do using the User interface. Many of the applications delivered with Inventor are using the API to do their work in Inventor; Tube & Pipe, Cable & Harness, Frame Generator, Inventor Studio and others. And in fact, iLogic is also an Inventor add-in and is using the API to do everything it does in Inventor. Teaching you to use the API it out of the scope of this class but I can introduce some basic concepts to help you get started and to help you better make sense of any existing sample programs you might see.

Inventor Object Model

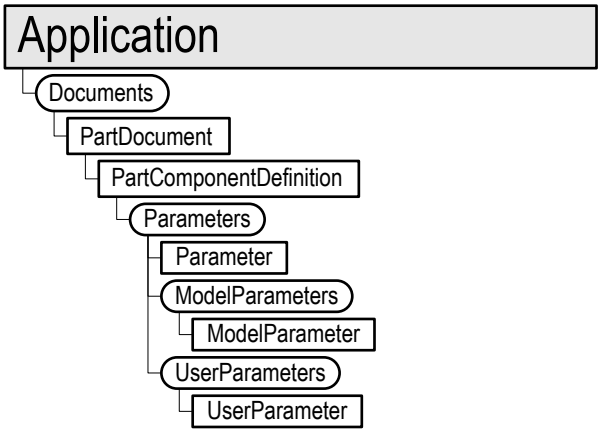
Inventor's programming interface is exposed as a set of objects. In fact, in Inventor 2016 there were 1444 objects which have 4661 methods, 16763 properties, and 172 events. That's a lot of functionality and can be very intimidating. However, as you'll see, you don't need to understand

all of it to be able to use the API. Instead you just need to understand the few objects that pertain to the area of Inventor you want to work with, which we'll see in a minute.

Inventor's objects are organized in what's referred to as the "Object Model". This is represented as the structure shown below where most of the 1444 objects are displayed.



To the right is a small section of the object model that contains most of the objects related to parameters. The top object is the "Application" object and represents Inventor as a whole. If you have the Application object you have access to the entire API. The Application object supports various properties and methods that let you interact with the application. Many of these properties return other objects. For example, the Application object has a property called "Documents" that returns a Documents object. The Documents object provides access to the PartDocument object which provides access to the PartComponentDefinition object, and so on. This structure is represented in the object model shown to the right.



There are two different types of objects represented in the object model. Standard objects, where most of them have a one-to-one correspondence to things you're familiar with in Inventor, and Collection objects. Collection objects are the objects shown in the boxes with rounded corners. They're API specific and are used to provide access to a group of similar objects and it's through collections that you create new objects. In the chart above the Documents collection object provides access to all of the open documents and has methods to let you open existing

documents and create new documents. The Parameters collection provides access to all existing parameters, regardless of their type, and provides access to other collection objects that provide access to parameters of a specific type. For example, the UserParameters collection provides access to the parameters the user created and lets you create new user parameters.

The object model might look complex but as a user of Inventor most of it should be somewhat familiar. The organization of the objects in the object model is based on ownership. What would you say is the owner of a parameter? Wouldn't it be the part or assembly document that the parameter is in? That's the structure the object model represents. Understanding this structure is key because you have to traverse the object model to get to the specific object you want. To get a parameter, you first get the document that contains the parameter and then you can get the parameter inside it. Without going into the details of why, there is an intermediate object in the chart above called the PartComponentDefinition. Just know that it's this object that actually owns all of the part specific information and that each part document has a single PartComponentDefinition associated with it.

When you use the iLogic Parameter object to change the value of two parameters, like that shown below, iLogic is really making Inventor API calls to do the actual work.

```
Parameter("Length") = 15
Parameter("Width") = 10
```

Below is the equivalent code using the Inventor API.

```
Public Sub Main()
    ' Connect to the Application object and get the active document.
    Dim invApp As Inventor.Application
    invApp = GetObject(,"Inventor.Application")
    Dim partDoc As PartDocument = invApp.ActiveDocument

    ' Get the Parameters collection.
    Dim params as Parameters = partDoc.ComponentDefinition.Parameters

    ' Change the values of the parameters "Length" and "Width".
    Dim param = params.Item("Length")
    param.Expression = "15"

    param = params.Item("Width")
    param.Expression = "10"

    ' Update the document.
    partDoc.Update()
End Sub
```

Units in iLogic and Inventor's API

There are some very basic differences between writing iLogic and API code. One of these is how they each handle units. When using iLogic and getting and setting values it is always in the current document units. When using the API it is the internal units, which are always centimeters for length and radians for angles.

The API behavior may seem strange at first but it actually makes writing applications easier because the units are always consistent. For example, if I was to have the following iLogic code and different users with different document units using that code they would get different results.

For example, if one user has their document units set to inch this will set the length to 15 inches but another user that has their document units set to millimeters will have a length of 15 millimeters.

```
Parameter("Length") = 15
```

When setting parameters using the API there are actually two options for setting the value. You can use the “Value” property which for lengths is always centimeters or you can use the “Expression” property which can be any valid expression, just like what the user would type in and then it’s evaluated the same as in the UI. For example the code below gets the parameter named length and sets the Expression to be “15” which will be 15 of whatever the default units are. I could also set it to “15 in” so then it will always be inches or I could even do something like “width / 2” to create an equation. When I query the Expression property I’ll get back the same string.

```
Dim param = params.Item("Length")
param.Expression = "15"
```

I can also use the Value property. Notice that the value I’m assigning is not a string but a real value. With the Value property, lengths are always in centimeters and angles are always in radians.

```
param.Value = 15
```

Because with use of units is consistent with the API you can write code that always assumes lengths are centimeters and angles are radians and it will work the same in all cases. But you’re probably saying that you or your users don’t want to work in centimeters or radians and you would be right. But you can handle that too. Any time you need to get values from the user or display results you can convert them from the current document units. But for larger applications the interaction with the user is a small portion of the code and the rest of the code never needs to worry about units.

To work with units you can use the UnitsOfMeasure object. For example, if I use the API to get the length of a line and want to display that to the user I can use the following. Remember that the API will always return the length in centimeters and the user could have set their document units to anything. The example below allows the user to key in a value which is then evaluated as a length using the GetValueFromExpression method which will return the value in centimeters. The value entered by the user could be a simple value like “12” where the units would be assumed to be the current document units. Or it could be a value that specifies the units like “14 mm”. Or it can be a complex expression containing equations and references to other parameters like “(Width + Height) / 2”. The important thing is that as the programmer you don’t care what the user enters because you just pass it into the GetValueFromExpression method and get back centimeters so the rest of your program that does the work can always assume everything is centimeters.

```
Dim partDoc As PartDocument = ThisApplication.ActiveDocument
Dim uom As UnitsOfMeasure = partDoc.UnitsOfMeasure

Dim userValue As String = InputBox("Enter value")
Dim realValue As Double = uom.GetValueFromExpression(userValue, uom.LengthUnits)
MsgBox("Value: " & realValue)
```

When you need to display something back to the user you can also use the UnitsOfMeasure object but in this case use the GetStringFromValue method which takes a value and unit type, just the same as the GetValueFromExpression, but in this case returns a string that is in the units specified and formatted according to the current document settings for units. If the user's current units are inch the code below will display "17.984 in". If the user's current units are meters it will display "0.457 m". In both cases, three decimals are shown because that's the precision defined in the document settings. But the important thing to see here is that as the programmer you don't need to care about any of that but rely on Inventor to do any needed conversions and formatting.

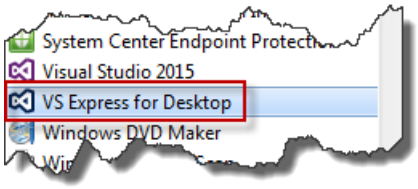
```
Dim length As Double = 45.67874687
MsgBox(uom.GetStringFromValue(length, uom.LengthUnits))
```

Writing Code in Visual Studio

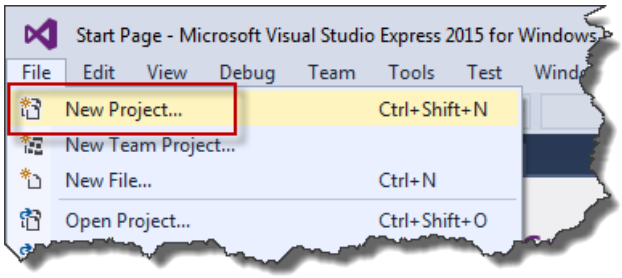
I've already said that iLogic code is actually Visual Basic .NET code but it can also useiLogic specific objects. Most programs you'll find online aren't using iLogic so they won't use any of the iLogic specific objects and many of the programs I see being written in iLogic don't use these objects either. For these programs, it's much more efficient to write the bulk of the program in Visual Studio and then copy it into a rule at the end. Visual Studio helps you as you write the code with code hints and as long as you don't use any of the iLogic specific objects you can even run and debug your program using Visual Studio. Below are the steps to run some code in Visual Studio.

For this example, I'll be using Visual Studio Express, but the same workflow applies to any of the different versions.

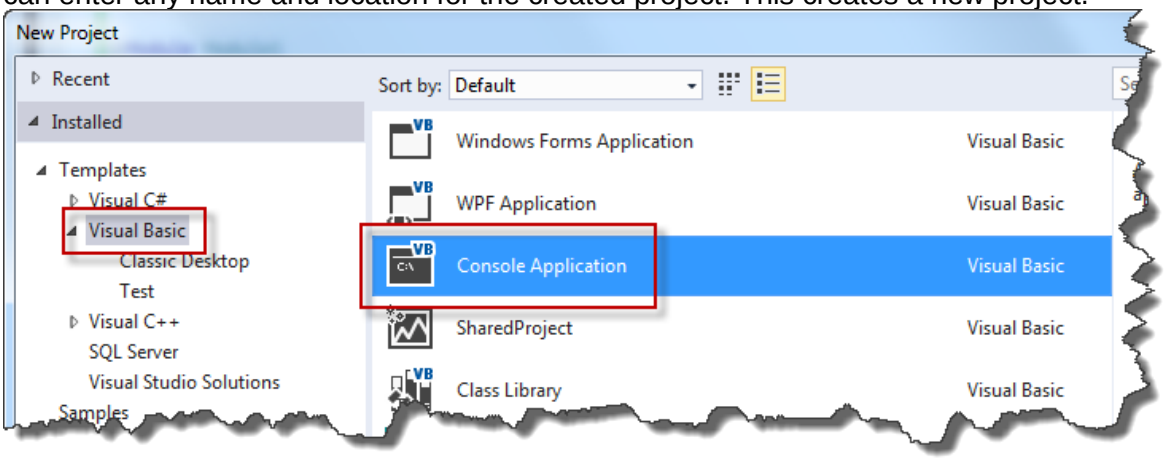
1. Start Visual Studio from the Windows **Start** menu or from the desktop.



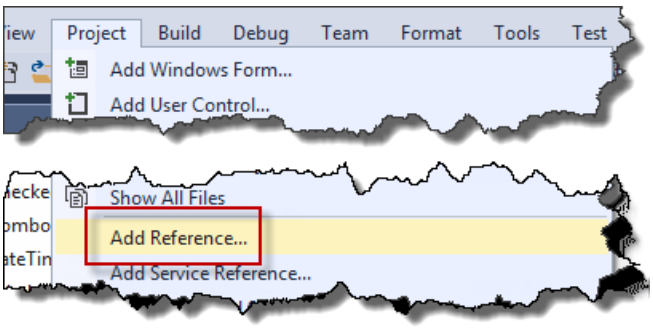
2. From the **File** menu or the start page, choose **New Project…**



3. Choose “Visual Basic” in the Templates list and the “Console Application” template. You can enter any name and location for the created project. This creates a new project.

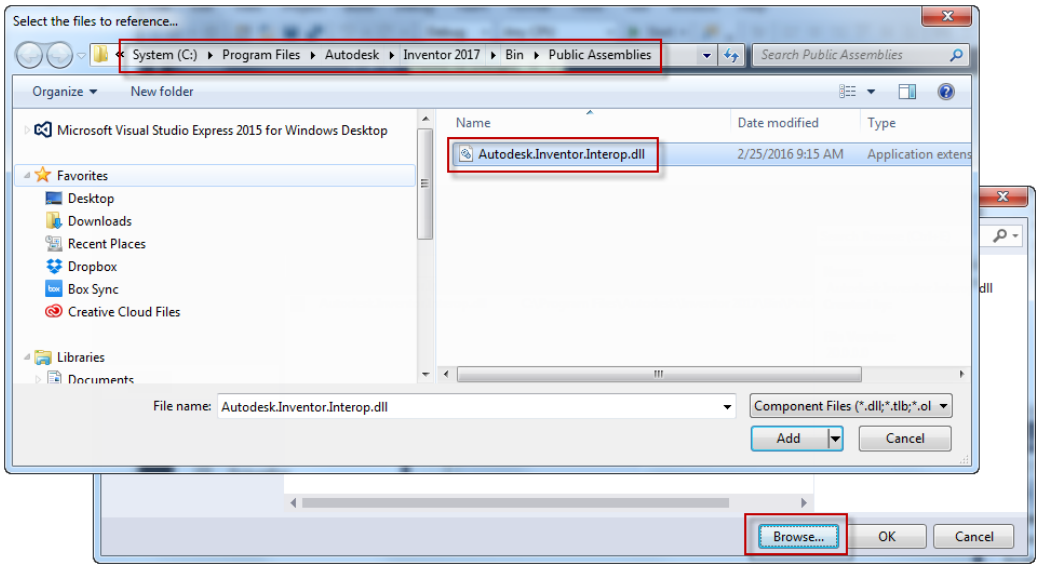


4. Visual Studio doesn’t know about Inventor so you need to add a reference to the Inventor library. Use the “Add Reference…” command in the “Project” menu to add a reference to the Inventor library to your project.



In the “Reference Manager” dialog, choose the “Browse…” button at the bottom and browse to the “Autodesk.Inventor.Interop.dll” file for the version of Inventor you’re using. In the example below, the path is:

```
C:\Program Files\Autodesk\Inventor 2017\Bin\Public Assemblies
```



5. Enter the code shown below into the code window. The “Imports Inventor” line at the top allows you to use the Inventor objects without requiring you to use the library name. For example you can use

   ```
   Dim extrude As ExtrudeFeature
   ```

   instead of

   ```
   Dim extrude As Inventor.ExtrudeFeature
   ```

   The Dim line is declaring a variable named “ThisApplication” as an Inventor Application object and then calling the `GetObject` function to get the Application object from the running instance of Inventor. Notice in the `GetObject` call that there is a comma and then the “Inventor.Application”. The comma is required.

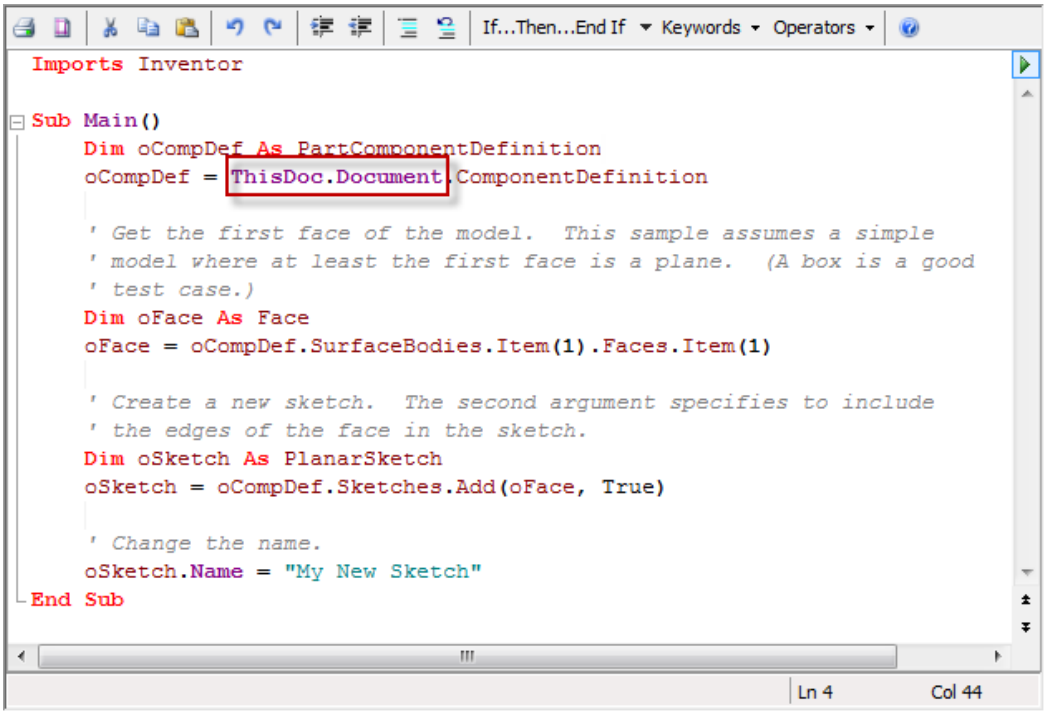
```
Imports Inventor

Module Module1
    Sub Main()
        Dim ThisApplication As Inventor.Application = GetObject(, "Inventor.Application")
    End Sub
End Module
```

6. You can now write, execute, and debug your code in Visual Studio. The only limitation is that you can’t use any iLogic specific objects because they’re not supported in Visual Studio. However, all of the Inventor API is supported.

   One of the sample programs from the Inventor API help has been copied into the program above. The highlighted portions are reported as errors by Visual Studio saying that Set statements are no longer supported. After deleting the Set statements the program can be run from Visual Studio.

```
Imports Inventor

Module Module1
    Sub Main()
        Dim ThisApplication As Inventor.Application = GetObject(, "Inventor.Application")

        ' Set a reference to the part component definition.
        ' This assumes that a part document is active.
        Dim oCompDef As PartComponentDefinition
        Set oCompDef = ThisApplication.ActiveDocument.ComponentDefinition

        ' Get the first face of the model.  This sample assumes a simple
        ' model where at least the first face is a plane.  (A box is a good
        ' test case.)
        Dim oFace As Face
        Set oFace = oCompDef.SurfaceBodies.Item(1).Faces.Item(1)

        ' Create a new sketch.  The second argument specifies to include
        ' the edges of the face in the sketch.
        Dim oSketch As PlanarSketch
        Set oSketch = oCompDef.Sketches.Add(oFace, True)

        ' Change the name.
        oSketch.Name = "My New Sketch"
    End Sub
End Module
```

Here's the same program copied into an iLogic rule. Notice that creating and setting the ThisApplication has been removed and using the `ActiveDocument` property has been replaced with the iLogic specific `ThisDoc.Document`.



The iLogic specific API has some functions to provide shortcuts to accessing the Inventor API. You can directly access the Application object by using the `ThisApplication` function. This is the most important because you now have access to the entire Inventor API. You can also access the document that the rule is running within using `ThisDoc.Document`

Examining the API using VBA

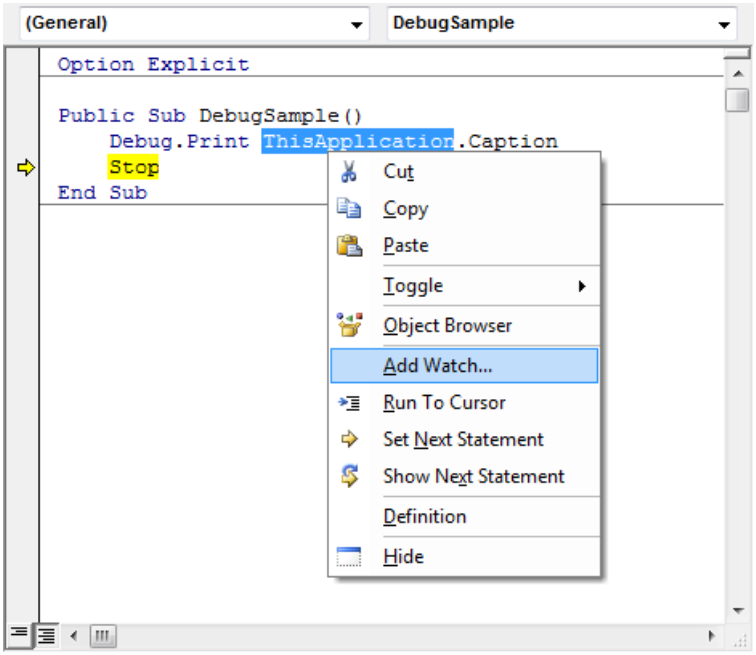
I made the statement above that VBA is the best for debugging and is the best tool for prototyping to determine if something is possible or not. VBA is typically the first thing I open when researching an Inventor API question. Below are some examples of how I use it.

VBA Debugger – Using the Watch Window

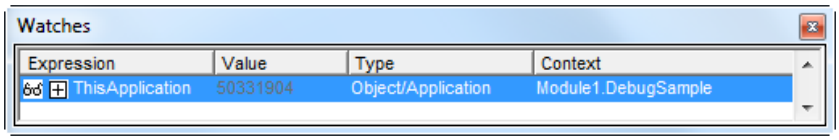
The VBA debugger is particularly useful when looking at Inventor's API. Many of the other languages will also support debugging the Inventor objects but they don't do it as cleanly as VBA does. With just the simple VBA macro below you can get a lot of information about Inventor's API.

```
Public Sub DebugSample()
    Debug.Print ThisApplication.Caption
    Stop
End Sub
```

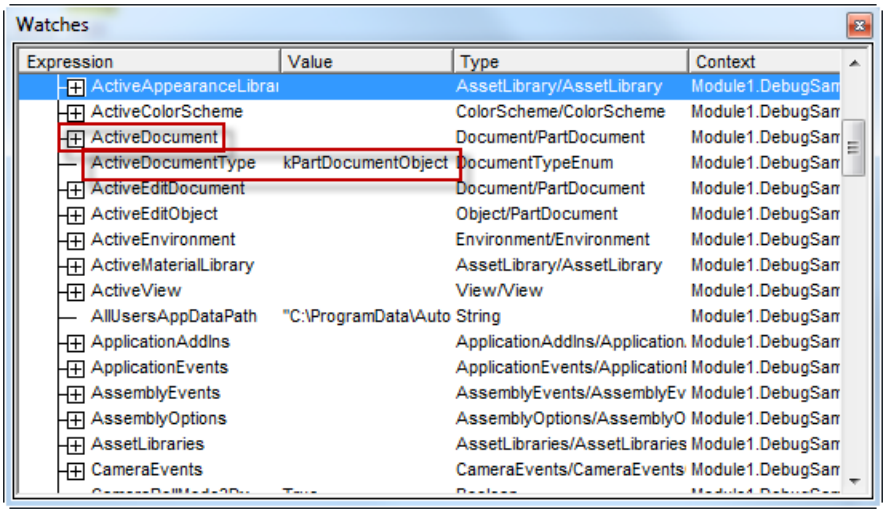
If I run the macro above it will break at the Stop statement so I can debug. If I right-click on "ThisApplication" and pick the "Add Watch…" option, as shown below, VBA will display ThisApplication in the "Watches" window (which it will open if it's not already open).



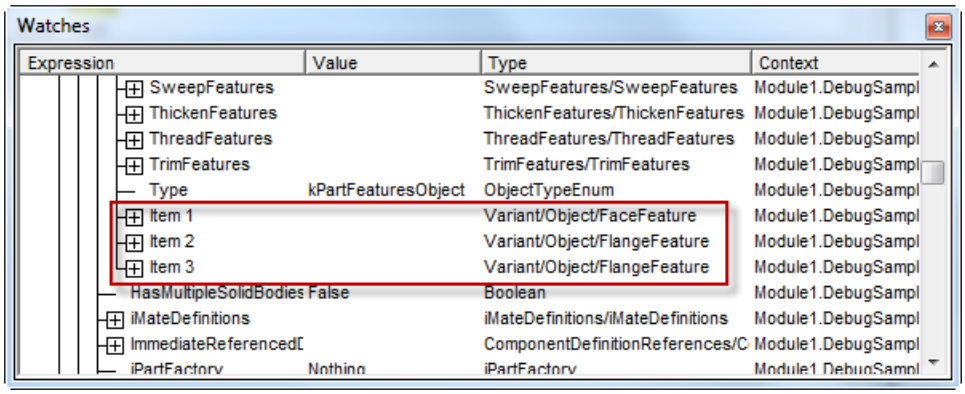
Below is what you'll initially see in the Watches window. It shows you that you're looking at the expression "ThisApplication" which is of type "Application".



The Watches window becomes especially useful when you click the “+” next to “ThisApplication”. This expands the tree and shows all of the properties supported by that object and their current values. In the picture below, ThisApplication has been expanded and the list has been scrolled down to see some of the more interesting properties. For properties that return a simple value you can see that value in the window. For example, the ActiveDocumentType property returns a value indicating what type of document is currently active in Inventor. In this example, it returns kPartDocumentObject, indicating a part is active. The properties that return other objects have a “+” next to them and can be expanded to see the properties of that object. For example, clicking the “+” next to the ActiveDocument property will show me all of the values of the properties on the PartDocument object returned by the ActiveDocument property.



In the example below, “ActiveDocument”, “ComponentDefinition”, and finally “Features” were clicked where you can see the specific feature collection objects and the high level collection of all features where you can see there are three features in this part; one face and two flange features.



What the Watches window does is provide a “live” view of Inventor’s Object Model. You can click through the objects using the same structure you see in the large object model poster but this lets you view it in the context of an open document and see all of the properties and their values, not just the objects.

Here's a modified version of the previous VBA program that makes it a bit easier to examine a specific object. You need to first select the object you want to look at then run the macro and add the variable "obj" to the Watches window.

```
Public Sub DebugSample()
    Dim obj As Object
    Set obj = ThisApplication.ActiveDocument.SelectSet.Item(1)
    Stop
End Sub
```

It's important to understand that using this tool isn't going to highlight the answer to your question but it will allow you to do some investigation work without writing any code and many times you'll be able to find the answer you're looking for.

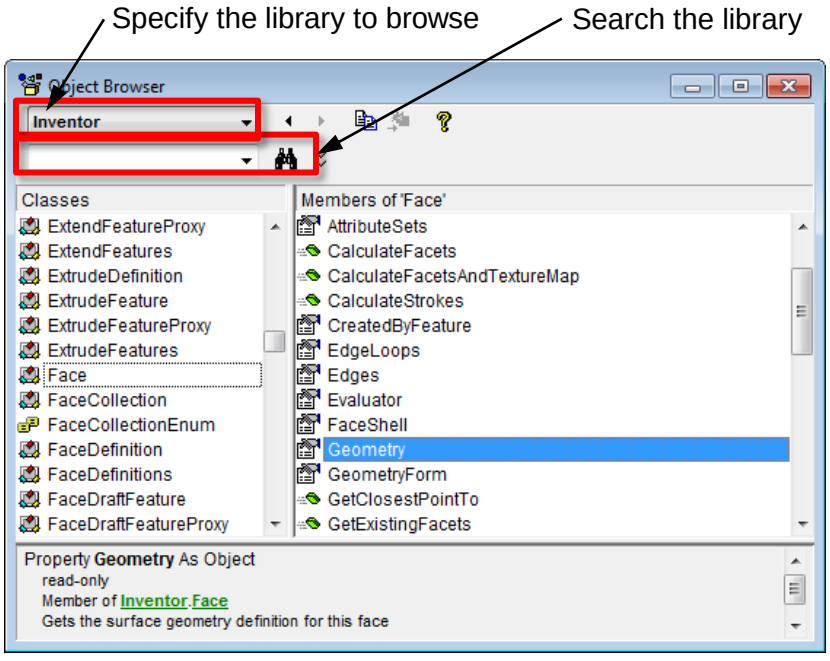
For some more examples of how useful this can be here are some questions I found in the Inventor Customization discussion group that the Watches window makes it relatively easy to find the answer.

VBA Debugger – Stepping Through Code

When first learning the Inventor API, the VBA debugger can be very useful to get an understanding of how to use the API by looking at sample code and programs others have made public. You can step line-by-line through the code and look at values as the program runs to figure out what the program is doing and how it works. Finding an existing sample that is close to what you want and then stepping through it to get a full understanding of how the program works will allow you to more easily make changes to the existing program to add the functionality you need. This deeper understanding also makes it easier for you to pick pieces out of existing programs and put them together into your program to accomplish what you need.

VBA Object Browser

The Object Browser is a tool that lets you look at the objects supported by a particular COM library. Visual Studio also supports an Object Browser but I've always found the VBA Object Browser to be easier to use. Using the Object Browser you can view the methods, properties, and events that each object supports. It has a short description of the object or function and is context sensitive so that if you press F1 it will take you to Inventor's API help for the current topic. You can also use the Object Browser to search the library.



Converting VBA Code

If you have Visual Studio, converting VBA code to VB.NET is significantly simplified because it will point out most of the issues as errors. There are a few other differences that are not so easily found and show up either as run-time errors or incorrect results. The issues I think you're most likely to encounter are listed below.

1. **The Set keyword is no longer used.** In VBA code you'll see a lot of statements that include the "Set" keyword at the beginning of the line, like the line shown below. This is a simple way to identify code as coming from VBA.

```
Set oDoc = ThisApplication.ActiveDocument
```

   These are very easy to catch and fix because Visual Studio will point them out as an error and you just need to delete "Set" to get the statement shown below.

```
oDoc = ThisApplication.ActiveDocument
```

2. **Parentheses are required around function arguments.** VBA has strange rules regarding the use of parentheses. If you call a function and don't need the return value, the parentheses are optional and if you do use parentheses you have to use the Call statement. The two statements below are valid VBA code.

```
MsgBox "This is a test."
Call MsgBox("This is a test.")
```

   In VB.Net you must always use parentheses around arguments and the Call statement is optional so the code below is what you'll typically see in VB.NET

```
MsgBox("This is a test.")
```

3. **VB.NET requires fully qualified enumeration constants.** In VBA when you use an enum value you only have to use the name of the value but VB.NET requires the name of the enumeration and the value. The statement below is valid in VBA, does not work in VB.Net.

```
oExtrude.Operation = kJoinOperation
```

   In VB.Net you must fully qualify the use of kJoinOperation by specifying the name of the enumeration as shown below.

```
oExtrude.Operation = PartFeatureOperationEnum.kJoinOperation
```

   These are easy to catch and fix since VB.Net identifies them as errors and IntelliSense does most of the work for you to create the fully qualified name as you start typing to enter the correct name.

4. **Arrays have a lower bound of 0 (zero).** This is a difference that you have to be careful when converting VBA code to VB.NET. In VBA the default lower bound of an array is 0 but it's common to use the Option Base statement to change this to 1. It's also common in VBA to specify the lower and upper bounds in the array declaration as shown below.

```
Dim adCoords(1 To 9) As Double
```

In VB.Net the above statement is not valid. The lower bound of an array is always 0. The equivalent statement in VB.Net is:

```
Dim adCoords(8) As Double
```

This creates an array that has an upper bound of 8. Since the lower bound is zero the array can contain 9 values. This can be confusing for anyone familiar with other languages where the declaration is the size of the array rather than the upper bound. If your VBA program was written assuming a lower bound of 1, adjusting the lower bound to 0 shifts all of the values in the array down by one index. You'll need to change the index values everywhere the array is used.

5. **Variable scope.** The scope of variables within functions is different with VB.Net. Variable scope is now limited to be within code blocks where-as VBA was only limited to within a function. If you copy the VBA function below, (which works fine in VBA), into a VB.Net program it will fail to compile. The last two uses, (underlined in the sample below), of the variable strSuppressed report that the variable is not declared. In this example strSuppressed is declared within the If Else block and is only available within that block.

```
' VBA
Public Sub ShowState(ByVal Feature As PartFeature)
    If Feature.Suppressed Then
       Dim strSuppressed As String
       strSuppressed = "Suppressed"
    Else
       strSuppressed = "Not Suppressed"
    End If

    MsgBox "The feature is " & strSuppressed
End Sub
```

Here's a version of the same function modified to work correctly in VB.Net. The declaration of the strSuppressed variable has been moved outside the If Else block, and now has scope within the entire sub.

```
' VB.Net
Public Sub ShowState(ByVal Feature As PartFeature)
    Dim strSuppressed As String
    If Feature.Suppressed Then
       strSuppressed = "Suppressed"
    Else
       strSuppressed = "Not Suppressed"
    End If

    MsgBox "The feature is " & strSuppressed
End Sub
```

There are some other differences too but I think it's less likely that you'll run into these.

6. **Function arguments default to ByVal.** In VBA, arguments default to ByRef which means the function can change the values and the modified values are passed back to the caller. With ByVal, the value of the variable is passed in but is local so even if the value changes it's not passed back to the caller. Here's an example to illustrate what this means. The VBA Sub below takes a feature as input and returns suppression and dimension information about the feature.

```
Sub GetFeatureInfo( Feature As PartFeature, Suppressed As Boolean,
                    DimensionCount As Long)
```

   In VBA this code works fine since it's optional to specify whether an argument is ByRef or ByVal and if you don't specify one it defaults to ByRef. In this example the Suppressed and DimensionCount arguments need to be ByRef since they're used to return information to the caller. The Feature argument can be declared as ByVal since it's not expected to change. VB.Net requires you to declare each variable as ByRef or ByVal. If it isn't specified for an argument, VB.Net automatically sets it to ByVal when you paste in your VBA code. Because of that, this example won't run correctly because the Suppressed and DimensionCount arguments won't return the correct values. They need to be changed to ByRef arguments for the sub to function as expected.

7. **Some data types are different.** There are two changes here that can cause problems. First, the VBA type **Long** is equivalent to the VB.Net **Integer** type. If you're calling a method that was expecting a Long or an array of Longs in VBA, that same code will give you a type mismatch error in VB.Net. Changing the declaration from Long to Integer will fix it.

   Second, the *Variant* data type isn't supported in VB.Net. If you have programs that use the Variant type, just change those declarations to the new *Object* type instead.

8. **Arrays of changing size are handled differently in VB.Net.** This and the next item are a couple of array related issues that you might run into. You can't specify the type when you use the ReDim statement re-dimension an array. Specifying a type will result in an error in VB.Net

```
' VBA
ReDim adCoords(18) As Double
```

```
' VB.Net
Redim adCoords(18)
```

9. **Declaring an array in VB.Net does not initialize it.** This will likely be another common problem encountered that's not obvious what the problem is from the error. The VBA code below will fail in .Net resulting in a type mismatch error. This is easily fixed by initializing the value to an empty array, as shown (open and closed braces).

```
' VBA
Dim adStartPoint() As Double
Dim adEndPoint() As Double
Call oEdge.Evaluator.GetEndPoints(adStartPoint, adEndPoint)
```

```
' VB.Net
Dim adStartPoint() As Double = {}
Dim adEndPoint() As Double = {}
oEdge.Evaluator.GetEndPoints(adStartPoint, adEndPoint)
```