AUTODESK UNIVERSITY

SD20619

Introduction to Inventor API Automation: Where Should You Start?

Nathan Bowser
MESA, Inc.

Learning Objectives

- Recognize the importance of learning VB.NET, and know where to go to gain that knowledge
- Learn how to capitalize on the Inventor API Help to transfer VB.NET skills to the Inventor API
- Learn how to identify and utilize resources available for advancing your programming knowledge
- Learn how to debug programs directly within the iLogic interface using a debug viewer

Description

If you are an Inventor software user interested in learning how to automate your design process using the Inventor software API (application programming interface), this class is designed specifically for you. The class will act as a guide, giving those with no programming or automation experience a road map to success for working with the Inventor software API. For simplicity, this class will focus on programming with the Inventor software API directly in the iLogic interface, and we will identify the resources available for new users to develop their programming skills. The course will provide students with the tools to learn, including how to program in VB.NET, how to use the API and API Help, where to find sample programs, how to debug API programs within the iLogic interface, and where to seek individual help. The course will give students the information and confidence they need to develop the skills necessary to quickly begin programming their own Inventor software customizations. This session features Inventor and Inventor Professional.

Your AU Expert

Nathan Bowser is a Mechanical Designer for MESA, Inc. (an Autodesk Silver Partner and an Autodesk Authorized Training Center). He is an Inventor Certified Professional with several years of experience training students in Inventor Sheet Metal and iLogic. Nathan's industry experience comes from seven years of service in the Manufacturing sector. Nathan has been using Autodesk Inventor since 2011 and, as a Mechanical Designer, works hands-on with the software every day designing and drafting. Nathan has been using iLogic and the Inventor API since Inventor 2011, and has been consulting with customers in their use of these automation techniques for over three years.

nathan.bowser@mesa-cad.com

MESA, Inc.

Introduction

As a front-line Inventor user and consultant, at an Autodesk Partner and Authorized Training Center, for over four years, I have had the opportunity to see Inventor used in many industries and applications. From architectural and civil to manufacturing and product design, I have had an opportunity to work with companies that span a wide range of Autodesk products and applications.

Despite the large differences in the work these companies do, a few things remain common. Today, companies are working hard to do more with less. Many of them have identified their shortcomings and inefficiencies and are looking to advance their knowledge and capabilities to continue to improve their processes.

One area, ripe for success, is through the automation of tasks which waste engineering time and resources, or introduce avoidable errors. More and more, companies are turning to automation to reduce the time it takes to complete a task, or to remove opportunities which could foster mistakes. Many companies, however, lack the resources or expertise in this area and are seeking to develop individuals with the knowledge and capabilities to take on such endeavors.

My goal in this course is to provide you with a guide, which you can use over the following weeks and months to develop the skills necessary to program your own inventor customizations. Through determination you can accomplish this goal, and through both professional and personal development, outfit yourself with the tools required to complete any automation tasks you might encounter.

Speaking the Language:

The first, and likely the largest, hurdle in becoming an Inventor API user is the programming language. A programming language is a formal computer language, or constructed language, designed to communicate instructions to a machine, particularly a computer. Just like learning to speak a new language takes time and dedication, so too does learning a new programming language. The Inventor API can be programmed with a number of languages (vb.net, c#, etc), however, for the beginner, I recommend Visual Basic .NET, or VB.NET.

VB.what?

VB.NET isn't the best programming language for every application, but it is the best place to start for a new programmer interested in learning to program using the Inventor API. I advise those with no other programming experience to begin their journey into the Inventor API by first learning the fundamentals of VB.NET. I recommend VB.NET for two main reasons.

Simplicity

First, VB.NET is relatively easy to learn (as far as languages go), and has a simpler, English-based syntax. Much of the code in VB.NET reads more like a conversation rather than an obscure compilation of characters. Compare the following two samples.

VB.NET Sample

```
Dim i As Integer
For i = 0 To 100
     If i = 7 Then
          MessageBox.Show(i)
     End If
Next
```

C# Sample

```
For (Int i=0; i<=100; i++)
{
    If (i == 7) {
      MessageBox.Show(i);
    }
}
```

Note the additional characters (parentheses, brackets, semi-colons, etc.) as well as the non-literal syntax required in the C# sample. While the end result is the same, it is easier to understand the VB.NET version without any other experience. C# is often regarded as a more powerful programming language, but it can be intimidating and less forgiving for those new to programming.

Flexibility

Second, and possibly more importantly, iLogic, the built-in automation tool within Inventor, is based on the VB.NET language. So learning to program in VB.NET will allow you to program fluently both when using iLogic and when building stand-alone applications and Add-ins using the API.

Starting with VB.NET allows a new programmer to begin learning the basics and applying them right away using iLogic. Later, after becoming more proficient and confident, they can begin expanding their abilities to include stand-alone applications and custom Inventor Add-ins.

I'm Sold! Now what?

So you're ready to learn VB.NET? You're eager to start your journey in automation and excited to take that first step! One of the best resources available to new programmers interested in getting the basics of VB.NET under their belt is a free programming course available directly from Microsoft through the Microsoft Virtual Academy. Here is how Microsoft describes the virtual academy:

> *"MICROSOFT VIRTUAL ACADEMY PROVIDES FREE ONLINE TRAINING BY WORLD-CLASS EXPERTS TO HELP YOU BUILD YOUR TECHNICAL SKILLS AND ADVANCE YOUR CAREER. MAKE IT YOUR DESTINATION OF CHOICE TO GET STARTED ON THE LATEST MICROSOFT TECHNOLOGIES AND JOIN THIS VIBRANT COMMUNITY."*

With the help of the Virtual Academy, you'll be learning something new in no time. You can find the Microsoft Virtual Academy at https://mva.microsoft.com/. Once you have created a free account you will find a wealth of programming knowledge covering many platforms, programs, and languages.

The primary resource for you to begin with will be "Visual Basic Fundamentals for Absolute Beginners", taught by Bob Tabor. This course covers everything you will need to know and more about VB.NET, and will help you to get comfortable in the Visual Studio programming IDE (Integrated Development Environment). I highly recommend while you are learning VB.NET that you download a free version of the Visual Studio application and program alongside Bob as he shows you the ins-and-outs of Visual Basic.

In addition to the training available through the Microsoft Virtual Academy, I've included, in **Appendix A** of this document, a quick reference guide for new programmers containing some VB.NET functions and samples that are used frequently when building programs for Inventor.

iLogic vs. The API

Once you've learned the basic skills necessary to work with VB.NET you'll find yourself ready to apply your knowledge for Inventor Automation. A common question I see beginners asking:

> "What is the difference between programming in iLogic and programming in the API?"

To answer that question, we need to first understand what iLogic is.

What is iLogic?

iLogic is an Autodesk Inventor Add-in included with all Inventor Packages. It is based on the VB.NET language, so it gives the user access to the same level of programming within Inventor that they have using the API. In addition to the base functionality of Visual Basic, iLogic adds an additional layer of functionality which is specifically designed for controlling common Inventor tasks. These are exposed to the iLogic user via iLogic's snippets.

Essentially, iLogic is similar to the Visual Studio IDE, allowing you to build and run your program with access to some advanced integrated functionality but built right into Inventor. This allows for the elimination of some of the tasks required when programming with Inventor such as creating or connecting to an Inventor session, accessing and opening Inventor documents, accessing parameters and iProperties, and connecting to Excel documents. These functions, and more, are already incorporated into iLogic and allow a simpler way for new users to connect with Inventor. In this way, there isn't really a difference from iLogic and the API. The API is at the core of iLogic, and iLogic resides on top of it and bolsters its functionality.

Why do I care about the API then?

While much, if not all, of the functionality of the API is available through the iLogic interface, it still has its drawbacks. For all of iLogic's benefits (built-in class functions, simple Inventor integration, and shortcuts and snippets) there are a number of areas where it falls short.

First and foremost, the iLogic interface lacks the powerful IntelliSense capabilities of Visual Studio. IntelliSense is a group of features which help you track the parameters you have created, offers the ability to autocomplete code as you are typing, and helps you learn more about the code you are using. Essentially, IntelliSense is designed to make programming faster and easier. You can find out more about these features and more at msdn.microsoft.com.

Secondly, iLogic lacks direct debugging capabilities; the ability to step through the code line-by-line as it executes. Debugging is a common task used while testing a program that is nearly invaluable, allowing the programmer to identify and correct errors faster and easier. Being able to examine exactly what a program is doing and what values are currently contained in a variable significantly reduces the potential aggravation involved in identifying and eliminating bugs.

Which one is best?

Given the benefits and drawbacks of iLogic, it might be difficult to decide where to begin. The lack of IntelliSense can make writing code a bit of a chore and without debugging a new user can easily be overwhelmed by a simple problem.

Despite those issues, I feel that, for the uninitiated, iLogic is still a great place to get your feet wet. Building a stand-alone program or add-in using the API requires understanding of Visual Studio projects, resources, references, as well as how to utilize application interfaces.

While those tasks are not terribly difficult, the ability to jump right into the code without that overhead makes iLogic more approachable for someone for which programming is not their only job responsibility.

Using iLogic allows you to program just what you need, right when you need it, and with a few tricks we can help overcome a few of the drawbacks of iLogic to help close the gap between the two.

Transferring VB.NET skills to Inventor

As we have discussed, learning the Visual Basic programming language is essential to programming in Inventor. However, the skills and knowledge to do so must be applied to Inventor through the understanding and mastery of the Inventor API.

What is this API we've heard so much about?

Part of understanding what an API actually is, is incorporated right into its name.

AN APPLICATION PROGRAMMING INTERFACE IS A SET OF ROUTINES, PROTOCOLS, AND TOOLS FOR BUILDING SOFTWARE APPLICATIONS. AN API SPECIFIES HOW SOFTWARE COMPONENTS SHOULD INTERACT AND APIS ARE USED WHEN PROGRAMMING GRAPHICAL USER INTERFACE (GUI) COMPONENTS. A GOOD API MAKES IT EASIER TO DEVELOP A PROGRAM BY PROVIDING ALL THE BUILDING BLOCKS. A PROGRAMMER THEN PUTS THE BLOCKS TOGETHER.

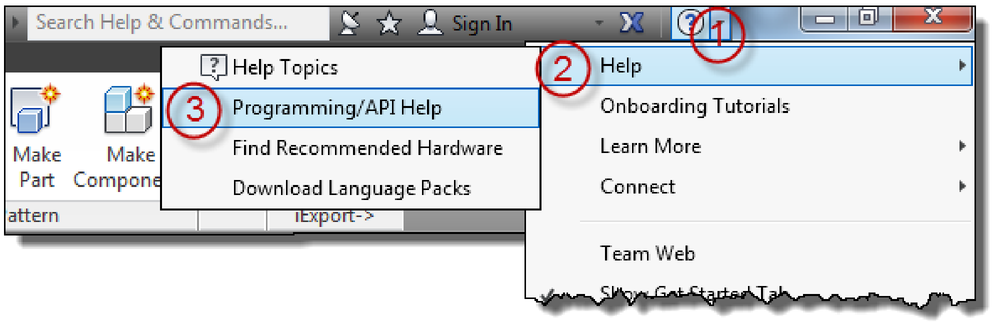
The Inventor API exposes to the user a library of object classes, variables, and functions which they can use to interface with, communicate to, and take control of the Inventor Application.

How do I use the API?

The trick to using the Inventor API successfully is in understanding what objects, variables, and functions exist within the API and how to interface with those components to achieve your goal. Luckily, that information is readily available; you just have to know where to look.

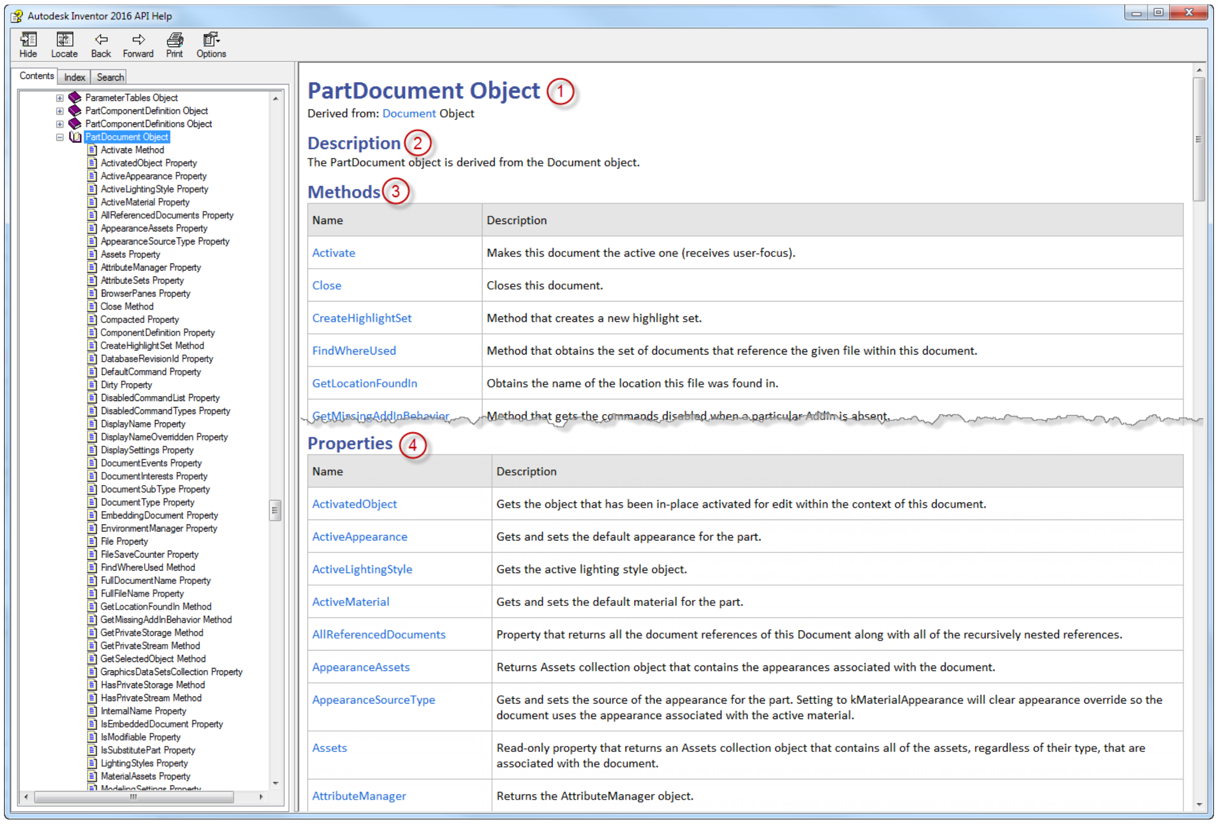
Inventor Programming/API Help Documentation

Autodesk has included a thorough, unifying documentation guide available right within Inventor under the Information drop-down → *Help* pop-out → *Programming/API Help*



The API Help Documentation allows you to explore the API and discover all the object classes, methods and properties available for use in Inventor Automation Programs. It includes detailed descriptions of the methods and properties for each object class and often contains examples and sample programs to demonstrate the different functionalities of those objects.

Let's examine the ***PartDocument Object*** help page in Depth.


Autodesk Inventor 2016 API Help

Hide Locate Back Forward Print Options

Contents | Index | Search

- ParameterTables Object
- PartComponentDefinition Object
- PartComponentDefinitions Object
- PartDocument Object
  - Activate Method
  - ActivatedObject Property
  - ActiveAppearance Property
  - ActiveLightingStyle Property
  - ActiveMaterial Property
  - AllReferencedDocuments Property
  - AppearanceAssets Property
  - AppearanceSourceType Property
  - Assets Property
  - AttributeManager Property
  - AttributeSets Property
  - BrowserPanes Property
  - Close Method
  - Compacted Property
  - ComponentDefinition Property
  - CreateHighlightSet Method
  - DatabaseRevisionId Property
  - DefaultCommand Property
  - Dirty Property
  - DisabledCommandList Property
  - DisabledCommandTypes Property
  - DisplayName Property
  - DisplayNameOverridden Property
  - DisplaySettings Property
  - DocumentEvents Property
  - DocumentInterests Property
  - DocumentSubType Property
  - DocumentType Property
  - EmbeddingDocument Property
  - EnvironmentManager Property
  - File Property
  - FileSaveCounter Property
  - FindWhereUsed Method
  - FullDocumentName Property
  - FullFileName Property
  - GetLocationFoundIn Method
  - GetMissingAddInBehavior Method
  - GetPrivateStorage Method
  - GetPrivateStream Method
  - GetSelectedObject Method
  - GraphicsDataSetsCollection Property
  - HasPrivateStorage Method
  - HasPrivateStream Method
  - InternalName Property
  - IsEmbeddedDocument Property
  - IsModifiable Property
  - IsSubstitutePart Property
  - LightingStyles Property
  - MaterialAssets Property
  - ModelingSettings Property

PartDocument Object ①

Derived from: Document Object

Description ②

The PartDocument object is derived from the Document object.

Methods ③

Name	Description
Activate	Makes this document the active one (receives user-focus).
Close	Closes this document.
CreateHighlightSet	Method that creates a new highlight set.
FindWhereUsed	Method that obtains the set of documents that reference the given file within this document.
GetLocationFoundIn	Obtains the name of the location this file was found in.
GetMissingAddInBehavior	Method that gets the commands disabled when a particular AddIn is absent.

Properties ④

Name	Description
ActivatedObject	Gets the object that has been in-place activated for edit within the context of this document.
ActiveAppearance	Gets and sets the default appearance for the part.
ActiveLightingStyle	Gets the active lighting style object.
ActiveMaterial	Gets and sets the default material for the part.
AllReferencedDocuments	Property that returns all the document references of this Document along with all of the recursively nested references.
AppearanceAssets	Returns Assets collection object that contains the appearances associated with the document.
AppearanceSourceType	Gets and sets the source of the appearance for the part. Setting to kMaterialAppearance will clear appearance override so the document uses the appearance associated with the active material.
Assets	Read-only property that returns an Assets collection object that contains all of the assets, regardless of their type, that are associated with the document.
AttributeManager	Returns the AttributeManager object.

The PartDocument Object represents an Inventor Part (.ipt). This is not an instance of the part within the assembly, known as an occurrence. It is the actual document of the part that resides on your computer.

The Help file consistently provides information for any object in the same manner. Let's break down the information provided in the Help File for the PartDocument Object.

1. Object Name - Indicates the syntax used in the API to define the object class.
2. Description – Provides a brief description of the object. It also indicates if the object inherits from another Object. If so, you might also want to review that objects help page.
3. Methods - Also referred to as *behaviors*, methods indicate actions that the object class can perform, such as ***Activate***, ***Close***, ***Update***, etc.
4. Properties – Also referred to as *data*, properties provide information about the object class, such as ***DisplayName***, ***FullFileName***, ***DocumentType***, etc.

Let's look at the following sample of code involving a PartDocument Object

```
1. Dim oDoc As Document
2. oDoc = ThisDoc.Document
3. Dim oPartDoc As PartDocument
4. If oDoc.DocumentType = DocumentTypeEnum.kPartDocumentObject
5.      oPartDoc = oDoc
6.      MessageBox.Show("Display Name: " & oPartDoc.DisplayName)
7. Else
8.      MessageBox.Show("Current Document is not of Type: PartDocument")
9. End If
```

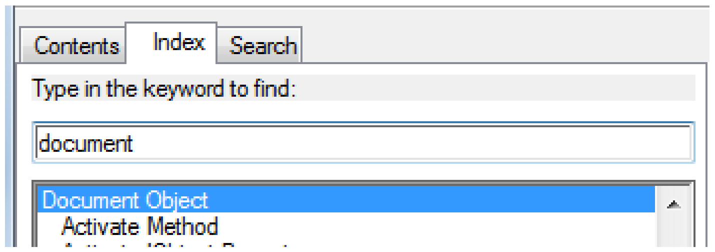
Let's analyze that code, line-by-line, and explain what the instructions are telling the computer.

Line 1. Create (`Dim`) a new variable of type `Document` named `oDoc`
Line 2. Pass the current document (`ThisDoc.Document`) into the newly created `oDoc` Variable
Line 3. Create (`Dim`) a new variable of type `PartDocument` named `oPartDoc`
Line 4. Test if the `DocumentType` of `oDoc` is equal to `DocumentTypeEnum.kPartDocumentObject`
If the test passes (condition results in *True*) lines 5 and 6 will be executed
Line 5. Assign the current document stored in `oDoc` into the recently created `oPartDoc` variable
Line 6. Print a message box to the user containing the Part Document's `DisplayName` string
Line 7. If the test failed (condition resulted in *False)* line 8 will be executed, instead of 5 and 6
Line 8. Print out a message box to the user indicating the Document was not a PartDocument
Line 9. The *If* statement requires an *End if* to indicate the end of the code block

Notice that during a single run of the code only one path is possible. Either the Document is of type PartDocument or not, therefore, lines 5 and 6 will only execute if the test is True. Likewise, line 8 will only execute if the test if False.

Most of the information about the above code, and the methods and properties used within it, are available within the Inventor Programming/API Help Documentation.

The documentation can be searched in multiple ways. You can view the *Contents* in alphabetical order, you can peruse the *Index* by Keyword, or you can *Search* the documentation objects. All three are useful tools in locating the content you need quickly.



Type in the keyword to find:

document

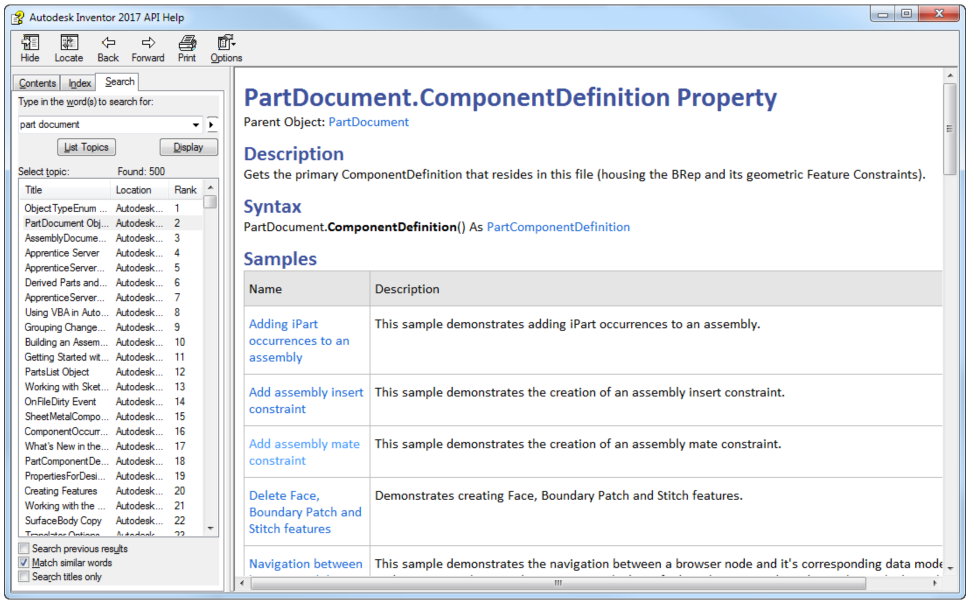
Document Object
Activate Method

Programming Samples

Also included in the Programming Help Documentation are numerous samples. These often provide more perspective on how the methods and properties are applied in practice.

Samples for a given item, if available, are listed at the very bottom of the objects page.

Below is the list of samples for the ***PartDocument.ComponentDefinition Property***.



Type in the word(s) to search for:

part document

Select topic: Found: 500

Title	Location	Rank
ObjectTypeEnum ...	Autodesk...	1
PartDocument Obj...	Autodesk...	2
AssemblyDocume...	Autodesk...	3
Apprentice Server	Autodesk...	4
ApprenticeServer...	Autodesk...	5
Derived Parts and...	Autodesk...	6
ApprenticeServer...	Autodesk...	7
Using VBA in Auto...	Autodesk...	8
Grouping Change...	Autodesk...	9
Building an Assem...	Autodesk...	10
Getting Started wit...	Autodesk...	11
PartsList Object	Autodesk...	12
Working with Sket...	Autodesk...	13
OnFileDirty Event	Autodesk...	14
SheetMetalCompo...	Autodesk...	15
ComponentOccurr...	Autodesk...	16
What's New in the...	Autodesk...	17
PartComponentDe...	Autodesk...	18
PropertiesForDesi...	Autodesk...	19
Creating Features	Autodesk...	20
Working with the ...	Autodesk...	21
SurfaceBody Copy	Autodesk...	22

☐ Search previous results
☑ Match similar words
☐ Search titles only

PartDocument.ComponentDefinition Property

Parent Object: PartDocument

Description

Gets the primary ComponentDefinition that resides in this file (housing the BRep and its geometric Feature Constraints).

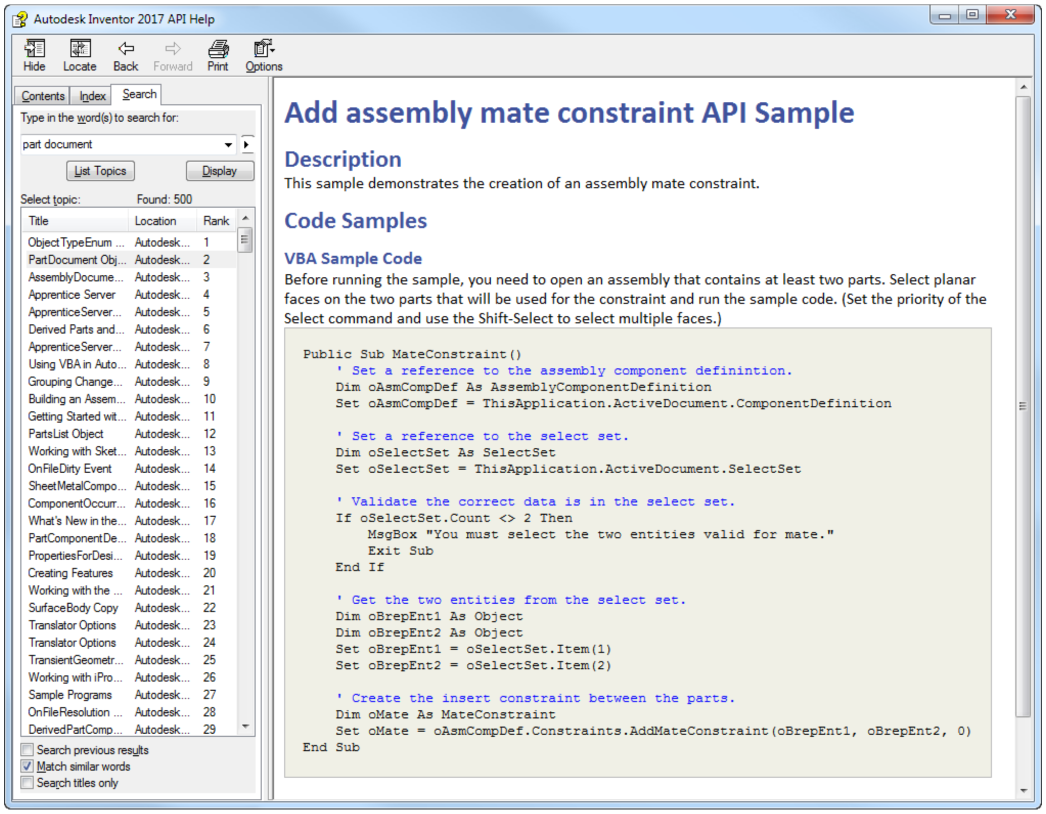
Syntax

PartDocument.**ComponentDefinition**() As PartComponentDefinition

Samples

Name	Description
Adding iPart occurrences to an assembly	This sample demonstrates adding iPart occurrences to an assembly.
Add assembly insert constraint	This sample demonstrates the creation of an assembly insert constraint.
Add assembly mate constraint	This sample demonstrates the creation of an assembly mate constraint.
Delete Face, Boundary Patch and Stitch features	Demonstrates creating Face, Boundary Patch and Stitch features.
Navigation between	This sample demonstrates the navigation between a browser node and it's corresponding data mode

Below is the sample for ***Add Assembly Mate Constraint*** as seen above



Type in the word(s) to search for:

part document

Select topic: Found: 500

Title	Location	Rank
ObjectTypeEnum ...	Autodesk...	1
PartDocument Obj...	Autodesk...	2
AssemblyDocume...	Autodesk...	3
Apprentice Server	Autodesk...	4
ApprenticeServer...	Autodesk...	5
Derived Parts and...	Autodesk...	6
ApprenticeServer...	Autodesk...	7
Using VBA in Auto...	Autodesk...	8
Grouping Change...	Autodesk...	9
Building an Assem...	Autodesk...	10
Getting Started wit...	Autodesk...	11
PartsList Object	Autodesk...	12
Working with Sket...	Autodesk...	13
OnFileDirty Event	Autodesk...	14
SheetMetalCompo...	Autodesk...	15
ComponentOccurr...	Autodesk...	16
What's New in the...	Autodesk...	17
PartComponentDe...	Autodesk...	18
PropertiesForDesi...	Autodesk...	19
Creating Features	Autodesk...	20
Working with the ...	Autodesk...	21
SurfaceBody Copy	Autodesk...	22
Translator Options	Autodesk...	23
Translator Options	Autodesk...	24
TransientGeometr...	Autodesk...	25
Working with iPro...	Autodesk...	26
Sample Programs	Autodesk...	27
OnFileResolution ...	Autodesk...	28
DerivedPartComp...	Autodesk...	29

☐ Search previous results
☑ Match similar words
☐ Search titles only

Add assembly mate constraint API Sample

Description

This sample demonstrates the creation of an assembly mate constraint.

Code Samples

VBA Sample Code

Before running the sample, you need to open an assembly that contains at least two parts. Select planar faces on the two parts that will be used for the constraint and run the sample code. (Set the priority of the Select command and use the Shift-Select to select multiple faces.)

```
Public Sub MateConstraint()
    ' Set a reference to the assembly component definintion.
    Dim oAsmCompDef As AssemblyComponentDefinition
    Set oAsmCompDef = ThisApplication.ActiveDocument.ComponentDefinition

    ' Set a reference to the select set.
    Dim oSelectSet As SelectSet
    Set oSelectSet = ThisApplication.ActiveDocument.SelectSet

    ' Validate the correct data is in the select set.
    If oSelectSet.Count <> 2 Then
        MsgBox "You must select the two entities valid for mate."
        Exit Sub
    End If

    ' Get the two entities from the select set.
    Dim oBrepEnt1 As Object
    Dim oBrepEnt2 As Object
    Set oBrepEnt1 = oSelectSet.Item(1)
    Set oBrepEnt2 = oSelectSet.Item(2)

    ' Create the insert constraint between the parts.
    Dim oMate As MateConstraint
    Set oMate = oAsmCompDef.Constraints.AddMateConstraint(oBrepEnt1, oBrepEnt2, 0)
End Sub
```

Let’s pull the text out of that sample to get a better look

```
Public Sub MateConstraint()
    ' Set a reference to the assembly component definintion.
    Dim oAsmCompDef As AssemblyComponentDefinition
    Set oAsmCompDef = ThisApplication.ActiveDocument.ComponentDefinition

    ' Set a reference to the select set.
    Dim oSelectSet As SelectSet
    Set oSelectSet = ThisApplication.ActiveDocument.SelectSet

    ' Validate the correct data is in the select set.
    If oSelectSet.Count <> 2 Then
        MsgBox "You must select the two entities valid for mate."
        Exit Sub
    End If

    ' Get the two entities from the select set.
    Dim oBrepEnt1 As Object
    Dim oBrepEnt2 As Object
    Set oBrepEnt1 = oSelectSet.Item(1)
    Set oBrepEnt2 = oSelectSet.Item(2)

    ' Create the insert constraint between the parts.
    Dim oMate As MateConstraint
    Set oMate = oAsmCompDef.Constraints.AddMateConstraint(oBrepEnt1,
oBrepEnt2, 0)
End Sub
```

This sample should get you well on your way to creating your own Mate Constraints in your rule. Unfortunately, it isn’t as simple as cut-and-paste to use the sample in an iLogic Rule.

The samples in the help are written for VBA programming. While the VB.NET programming language is very similar to VBA, there are a few differences that you will have to address.

The primary issue I experience when using the samples in my iLogic or VB.NET code is the `Set` keyword, which is required in VBA when assigning one object to another. Since the set keyword is obsolete in VB.NET, we can simply remove it. So the beginning of the sample would become:

```
Dim oAsmCompDef As AssemblyComponentDefinition
oAsmCompDef = ThisApplication.ActiveDocument.ComponentDefinition
```

There are many more differences between VBA and VB.NET but this is the primary issue regarding the programming language you will experience in attempting to use the samples.

The second issue I experience when using the samples is that enumerators are not implicitly known in iLogic. Here is a snippet from one of the samples which creates a new part document

```
' Create a new part document, using the default part template.
Dim oPartDoc As PartDocument
Set oPartDoc = ThisApplication.Documents.Add(kPartDocumentObject, _
      ThisApplication.FileManager.GetTemplateFile(kPartDocumentObject))
```

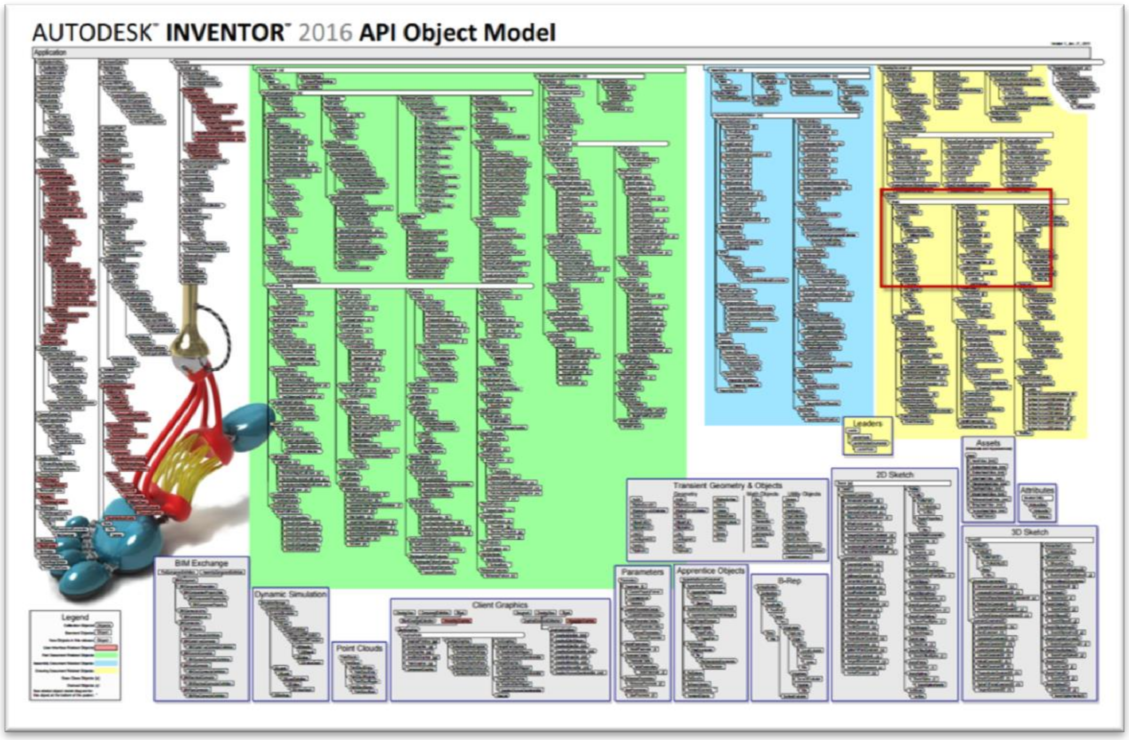
To use the sample we would have to use `DocumentTypeEnum.kPartDocumentObject` instead

**Note that this issue has been resolved in Inventor 2017*

Inventor API Object Model

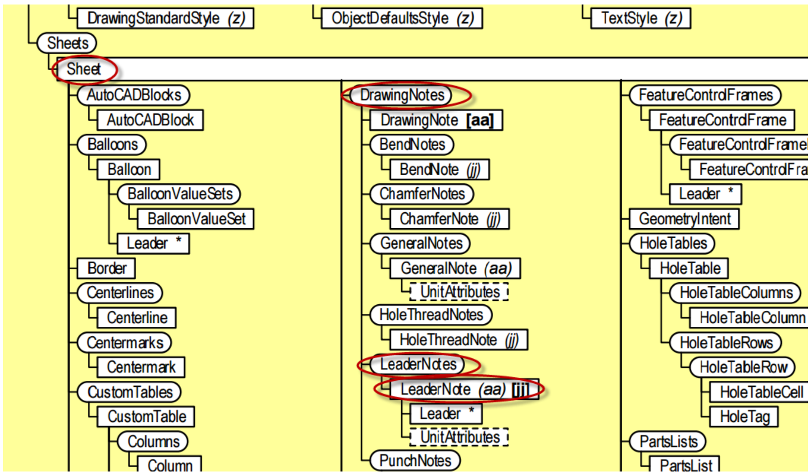
In addition to the API help documentation, Autodesk provides a detailed map of the connections within the API, known as an object model. The *Inventor 2016 Object Model* can be found Here.

The object Model specifies how all the different elements of the API fit together and how each object relates to one another. Using the Object model is analogous to reading a map. If you know your starting point and your destination, it is simply a matter of discerning the path which will get you where you need to go.



Take the following clipping from the Drawing section above. Assume I have a drawing sheet object variable called `oSheet` and want to access the first leader note on the sheet. I can reach that leader note as follows:

```
oSheet.DrawingNotes.LeaderNotes.Item(1)
```



Additional Resources

In addition to the help documentation and object model provided by Autodesk, there are a number of other resources available for you to continue to grow your automation abilities.

When I have a question or problem, I use a search engine, like Google, to search for my issue and often find a result which matches or is similar to what I am looking for. Those searches often lead me to one or more the following resources. They each hold a wealth of knowledge and will be valuable resources to help you to expand your capabilities.

Manufacturing DevBlog http://adndevblog.typepad.com/manufacturing/inventor/

This semi-official Autodesk blog is maintained by the Developer Technical Services team at Autodesk and while the blog covers more programs than Inventor, the Inventor portion contains a bevy of useful content. Information available here often comes from inside knowledge of the inner workings of the software and API, enabling you to get answers to questions that are not addressed in the API Documentation.

Mod the Machine http://modthemachine.typepad.com/

This private blog, authored by Autodesk Programming Experts Brian Ekins and Adam Nagy, is a great place to find code samples and snippets which you can use right away in your program to save time or help you over a particularly difficult hurdle.

From the Trenches with Autodesk Inventor http://inventortrenches.blogspot.com/

Another private blog, this one produced by the ever-helpful Curtis Waguespack, contains information and samples that often aligns closely with the needs of the end user. Curtis is the author of the ‘Mastering Autodesk Inventor’ series of books and brings with him the experience of a daily user of the software; hence the blog’s name ‘From the Trenches’

Autodesk Knowledge Network https://knowledge.autodesk.com/

The Autodesk Knowledge Network, while not exhaustive, contains a great deal of articles that help to address more non-specific topics. It’s definitely not a one stop shop for developers, but it does contain something for everyone.

Autodesk Inventor Customization Forum http://forums.autodesk.com/...

The Inventor Forums is a community of thousands of Inventor users all coming together to share their unique knowledge and skills. As you begin programming with the Inventor API, you start a journey that hundreds of people before you have made. Without a doubt you will have questions and the Inventor Customization forum is likely to contain the answer you seek, just like many others before you. If you have exhausted all of the options above and still find you can’t resolve your problem, feel free to post your question to the customization forum, and more than likely, you’ll get the answer you need and help pave the road for the next Inventor user to come after you.

Debugging iLogic

One of the drawbacks of using iLogic, as we have covered, is the lack of IDE support for real time debugging. Although there is no way to implement line-by-line debugging in iLogic, we do have an option to use VB.NET's diagnostic tools to debug key parts of our code in real time. Doing so will give us much of the benefits of a traditional debugger right in iLogic.

Trace

Just like the military uses tracer rounds to observe the path of a projectile as it is fired, we can employ traces on our code to follow the path of the program as it operates. To do this, we will employ the VB.Net Diagnostics class, which provides a set of methods that help you trace the execution of your code

To allow access to the trace abilities, we must import the diagnostics class into our program, and then we can employ the Trace classes' write line method. Review the code snippet below:

```
Imports System.Diagnostics

Sub Main
    Trace.WriteLine("iLogic: 'Status' Process Initiated")
    For i = 0 To 10
        Trace.WriteLine("iLogic: 'Info' i = " & i)
    Next
    Trace.WriteLine("iLogic: 'Status' Process Complete")
End Sub
```

The first line of code *Imports* the *System.Diagnostics* class which contains the classes used to generate debug information. Note that *Imports* appear before your *Sub Main* and other routines.

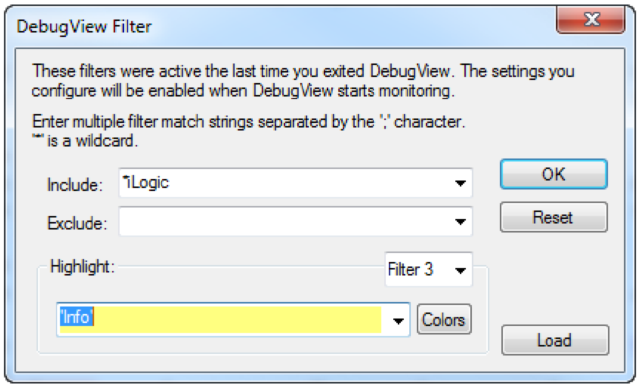
Once we have imported the *Diagnostics* class, we have access to *Diagnostics.Trace.WriteLine.* Since *Trace* only exists within the *Diagnostics* class, it is non-ambiguous. Therefore we do not need to specify *Diagnostics.Trace* on each call of the *Trace* class, simplifying our code.

The *Trace.WriteLine("string")* will write the string passed into it to the systems diagnostics functions. In order to see the debug information, however, we will need to use a Debug Viewer.

I use the free Microsoft application, **DebugView**, available at technet.microsoft.com. DebugView captures all debug calls for your entire system, so you will need to pare down the traces to only those which are related to iLogic. In order to do that you will need to configure the DebugView filter.

I've chosen to preface all my traces with iLogic. In doing so, I can Include only those traces which contain the text **iLogic*. I also like to color code my traces by type, so I have included a type designation after the iLogic string like 'Status', 'Info' , and 'Error'.

Using these options allow you to tailor the debugger to your particular needs and preferences.



These filters were active the last time you exited DebugView. The settings you configure will be enabled when DebugView starts monitoring.

Enter multiple filter match strings separated by the ';' character. '*' is a wildcard.

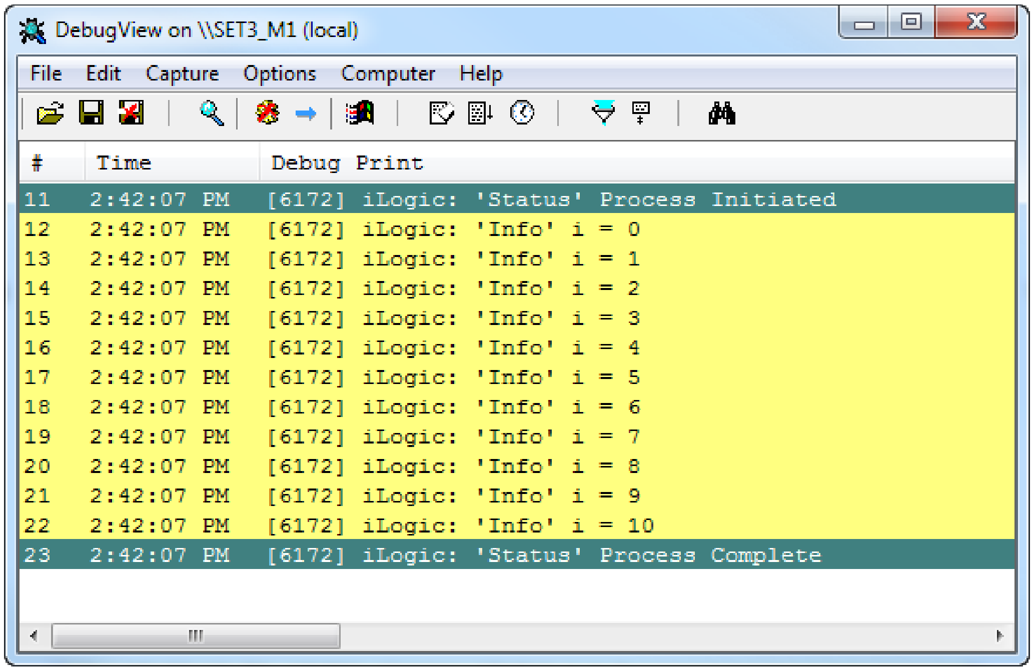
Include: *iLogic

Exclude:

Highlight: Filter 3

Info

Here is what the trace of the sample code would look like with the filters and formatting applied.


DebugView on \\SET3_M1 (local)
File Edit Capture Options Computer Help

```
#   Time         Debug Print
11  2:42:07 PM   [6172] iLogic: 'Status' Process Initiated
12  2:42:07 PM   [6172] iLogic: 'Info' i = 0
13  2:42:07 PM   [6172] iLogic: 'Info' i = 1
14  2:42:07 PM   [6172] iLogic: 'Info' i = 2
15  2:42:07 PM   [6172] iLogic: 'Info' i = 3
16  2:42:07 PM   [6172] iLogic: 'Info' i = 4
17  2:42:07 PM   [6172] iLogic: 'Info' i = 5
18  2:42:07 PM   [6172] iLogic: 'Info' i = 6
19  2:42:07 PM   [6172] iLogic: 'Info' i = 7
20  2:42:07 PM   [6172] iLogic: 'Info' i = 8
21  2:42:07 PM   [6172] iLogic: 'Info' i = 9
22  2:42:07 PM   [6172] iLogic: 'Info' i = 10
23  2:42:07 PM   [6172] iLogic: 'Status' Process Complete
```

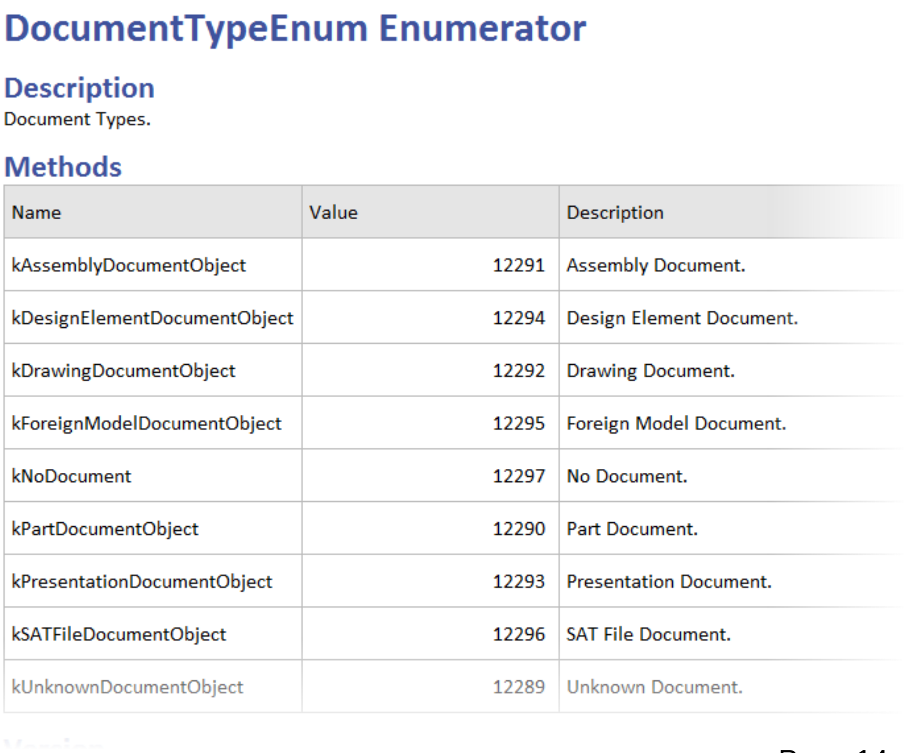
Using these tools, you can add as many traces to your code as required. I typically add traces before and after potentially error-prone calls, as well as including traces to output critical data values after they have been assigned to ensure the data matches my expectations.

Often, the data type I'm working with is an *object* but the debugger can only accept text strings. In those cases, I'll output the *object*.Name or the *object*.Value to the debugger, which will provide me with enough information to know which object is stored in my variable.

I also find it useful at times to output the *Type* of the current object, like *oDoc.DocumentType*.

Keep in mind, however, that Inventor will return the integer value, which you will need to compare to the enumerator list which you can locate in the API Help file.

So for our *PartDocument* object from eariler, we would see *12290* in the debug output, which we can identify as doc type *kPartDocumentObject*



DocumentTypeEnum Enumerator

Description

Document Types.

Methods

Name	Value	Description
kAssemblyDocumentObject	12291	Assembly Document.
kDesignElementDocumentObject	12294	Design Element Document.
kDrawingDocumentObject	12292	Drawing Document.
kForeignModelDocumentObject	12295	Foreign Model Document.
kNoDocument	12297	No Document.
kPartDocumentObject	12290	Part Document.
kPresentationDocumentObject	12293	Presentation Document.
kSATFileDocumentObject	12296	SAT File Document.
kUnknownDocumentObject	12289	Unknown Document.

Summary

Learning to build custom Inventor Automations is a worthwhile endeavor which rewards success with valuable lifelong skills. It does, however, take time and dedication to accomplish. There are many snares that can befall a new programmer and diminish their confidence and ambition.

In my personal attempts to develop my own skills in this area, I have been slowed and stopped many times. I have discovered difficulties for which I assumed I lacked the knowledge or abilities to overcome. I became discouraged, and resigned to failure. I did over time, albeit slowly, discover the tools available to move forward. Each time a new tool presented itself, it assisted me over a hurdle and allowed me to proceed.

It is my hope that by understanding programming languages better, knowing the tools available to you and how to use them, and possessing a path to follow and a foundation to build upon, you can experience success in an expedited fashion.

By fully understanding the task ahead, and best utilizing the resources available to assist you, I'm certain you will find success in your undertaking.

Appendix A

VB.Net Quick Reference Guide for iLogic

Variable Data Types: a few common types only, many more types exist

`Boolean`	-	Can contain **True** of **False**
`Char`	-	Can contain a single Unicode Character (a-Z, 0-9, etc)
`Date`	-	Can contain a Date and Time
`Double`	-	Can hold a numerical value which contains a decimal component (1.333$\bar{3}$)
`Integer`	-	Can hold whole numbers, positive and negative ($-9.2x10^{18}$ to $9.2x10^{18}$)
`String`	-	Can hold text values (capacity of approximately 2 Million Unicode characters)

Arithmetic Operators:

^	-	Raises one operand to the power of another
+	-	Adds two operands
-	-	Subtracts second operand from the first
*	-	Multiplies both operands
/	-	Divides one operand by another and returns a decimal result
\	-	Divides one operand by another and returns an integer result
`MOD`	-	Modulus Operator and remainder of after an integer division

Comparison Operators:

=	-	Checks if the values of two operands are equal
<>	-	Checks if the values of two operands are not equal
>	-	Checks if the value on the left if greater than the value on the right
<	-	Checks if the value on the left if less than the value on the right
>= / <=	-	Checks if the value on the left is greater than / less than the value on the right

String Manipulation functions: a few common functions only, many more functions exist

```
Dim myText As String = "String"
myText.Chars(3) 'Returns the character in the 3rd position (i)
Len(myText) 'Returns the number of characters in the string (6)
Left(myText, 3) 'Returns the Left 3 characters from the string (Str)
Right(myText,2) 'Returns the Right 2 characters from the string (ng)
Mid(Mytext,3,2) 'Returns 2 characters starting from the third (ri)
Mytext.Remove(3,2) 'Removes 2 characters starting from the third (Stng)
myText.ToUpper 'Converts all the characters in the string to Upper Case
myText.ToLower 'Converts all the characters in the string to Lower Case
myText.Trim 'Removes any leading or trailing white-space characters
"Text: " & myText 'Use "&" to concatenate two or more strings
```

`=      - The assignment operator; assigns the value on right to the variable on left`

```
myText = "This is some Text"
myNumber = 387
```

Appendix A – Continued

`Dim` – Short for Dimension, dim is used for variable declaration and storage allocation for variables

```
Dim myText As String     'Creates a new String variable called myText
Dim myNumber As Integer 'Creates a new Integer variable called myNumber
```

`Main()` – The entry point of the program; required for creating additional functions or sub-routines
Functions return a value, while Subs perform an operation but do not return a value

```
Sub Main()  'Main Code goes here
    'might contain calls to functions and sub-routines
    Dim myText As String
    myText = GetText()    'Calls the GetText Function per below
    Routine(myText)       'Calls the Routine Sub per below, w/ argument
End Sub

Function GetText() As String  'Does not accept any arguments but could
    Return "This is some text" 'Function returns a value
End Function

Sub Routine(text As String)  'Accepts a single string argument
    MessageBox.Show(text)
End Sub                       'Sub does not return a value
```

`If, Then` – allows the user to check if a condition exists and executes a statement based on the results

```
If myNumber < 100 Then        ' Is myNumber less than 100
    myText = "Small"
Else If myNumber < 500 Then   ' Else If  statement is Optional
    myText = "Medium"
Else                          ' Else statement is also optional
    myText = "Large"
End If
MessageBox.Show(myText)
```

`Select Case` – allows the user to check a single expression for multiple different possible values

```
Dim myVariable As Integer
Select Case myText
    Case "Small"
        myVariable = 1
    Case "Medium"
        myVariable = 2
    Case "Large"
        myVariable = 3
    Case Else                 'Case Else handles all non-specified cases
        myVariable = -1
End Select
```

`Do While Loop` – allows the user to execute a block of code repeatedly until a condition is met

```
Dim i As Integer = 0
Do While i < 10 'while condition can be here or after Loop at end
    MessageBox.Show(i)
    i += 1        'this is the same as i = i + 1
Loop  'while can also be placed here, insuring at least one execution
```

Appendix A – Continued

`For Each` – allows the user to loop through a group of items in a collection

```
Dim myArray() As Integer = {1, 3, 5, 7, 9}
For Each arrayItem As Integer In myArray
    MessageBox.Show(arrayItem)
Next
```

Sample 1: This sample demonstrates testing which document type is currently open in Inventor

```
Dim oDoc As Document
oDoc = ThisDoc.Document

Dim oDocType As String = ""

Select Case oDoc.DocumentType
    Case 12290 'DocumentTypeEnum.kPartDocumentObject
        oDocType = "Part"
    Case 12291 'DocumentTypeEnum.kAssemblyDocumentObject
        oDocType = "Assembly"
    Case 12292 'DocumentTypeEnum.kDrawingDocumentObject
        oDocType = "Drawing"
    Case Else
        oDocType = "Other"
End Select
MessageBox.Show("Document is of type: " & oDocType)
```

Sample 2: This sample demonstrates Looping through Occurrences and Documents in an Assembly in Inventor

```
Dim oDoc As Document
oDoc = ThisDoc.Document

If oDoc.DocumentType <> 12291 'DocumentTypeEnum.kAssemblyDocumentObject
    MessageBox.Show("An Assembly must be open to run this rule")
    Exit Sub
End If

'loop through assembly occurrences
Dim oOccs As ComponentOccurrences
oOccs = oDoc.ComponentDefinition.Occurrences

Dim occString As String = ""

For Each oOcc As ComponentOccurrence In oOccs
    occString += oOcc.Name & vbCr
Next
MessageBox.Show (occString, "Occurrrences")

'loop through all documents referenced by the assembly
Dim docString As String = ""

For Each oRefDoc As Document In oDoc.AllReferencedDocuments
    docString += IO.Path.GetFileName(oRefDoc.FullFileName) & vbCr
Next
MessageBox.Show(docString, "Documents")
```

Appendix A – Continued

Sample 3: This sample demonstrates accessing Standard and Custom iProperty Values in Inventor

```
Dim oDoc As Document
oDoc = ThisDoc.Document

Dim oPropSets As PropertySets
oPropSets = oDoc.PropertySets

Dim oPropSet As PropertySet
oPropSet = oPropSets.Item("Design Tracking Properties")

Dim oPartNumProp As Inventor.Property
oPartNumProp = oPropSet.Item("Part Number")

Dim oCustPropSet As PropertySet
oCustPropSet = oPropSets.Item("Inventor User Defined Properties")

Dim oCustomProp As Inventor.Property
Dim oCustPropName As String = "MyProperty"
Dim oCustPropValue As Integer = 6

Try
    oCustomProp = oCustPropSet.Item(oCustPropName)
Catch
    oCustomProp = oCustPropSet.Add(oCustPropValue, oCustPropName)
End Try

MessageBox.Show("Part Number: " & oPartNumProp.Value & vbCr & _
                "My Property: " & oCustomProp.Value)
```

Sample 4: This sample demonstrates using the translator add-in to generate a PDF with custom settings in Inventor

```
Dim oDoc As Document = ThisDoc.Document
If oDoc.DocumentType <> 12292 Then Exit Sub

Dim oPDFAddIn As TranslatorAddIn
oPDFAddIn = ThisApplication.ApplicationAddIns.ItemById _
                ("{0AC6FD96-2F4D-42CE-8BE0-8AEA580399E4}")
oContext = ThisApplication.TransientObjects.CreateTranslationContext
oContext.Type = IOMechanismEnum.kFileBrowseIOMechanism
oOptions = ThisApplication.TransientObjects.CreateNameValueMap
oDataMedium = ThisApplication.TransientObjects.CreateDataMedium

If oPDFAddIn.HasSaveCopyAsOptions(oDataMedium, oContext, oOptions) Then
    oOptions.Value("All_Color_AS_Black") = 0
    oOptions.Value("Remove_Line_Weights") = 1
    oOptions.Value("Vector_Resolution") = 2400
    oOptions.Value("Sheet_Range") = Inventor.PrintRangeEnum.kPrintAllSheet
End If

oDataMedium.FileName =
IO.Path.GetFileNameWithoutExtension(oDoc.FullFileName) & ".pdf"
oPDFAddIn.SaveCopyAs(ThisDoc.Document, oContext, oOptions, oDataMedium)
```