IM10601-L

You Can Do It! — How to Write Programs for Inventor

Brian Ekins
Autodesk

Learning Objectives

- Learn how to create a simple Visual Basic .NET program.
- Discover the basic concepts of Inventor's programming interface
- Learn how to create a program that works with Inventor's parameters
- Learn how to create a program that works with Inventor's iProperties

Description

This lab is an introduction to customizing Inventor using Inventor's programming interface. Inventor is a general-purpose design tool, but as an Inventor user you have unique needs. Using Inventor's Application Programming Interface (API) you can customize Inventor to be a tool specifically for you. Writing programs can often be intimidating, but this lab will cover the very basics of using Microsoft Visual Basic .NET and Inventor's programming interface to create custom applications. You'll have hands-on experience creating your own program that works with Inventor to drive parameters and work with iProperties.

Your AU Expert

Brian Ekins began working in the CAD industry in the 1980s and has worked in various positions, including CAD administrator, applications engineer, instructor, CAD API designer, and consultant. He has worked for Autodesk, Inc., since 1998. He is the designer of the programming interfaces for Inventor and Fusion 360, and he also supports developers and customers in their use of these technologies. Brian also enjoys designing real-world things and writing programs for fun at home.

Introduction to Programming Inventor

Congratulations on wanting to do more with Inventor than what the basic product can do. By writing programs you can automate difficult and repetitive operations and add completely new functionality, which makes using the software easier and reduces errors. If you're new to programming there will be a lot of new concepts but my intent is to start slowly and demonstrate how you can improve your knowledge and skills little by little.

Don't expect to become an expert at programming Inventor in the next 1 ½ hours. This is meant to be a quick introduction to give you a small test of what it's like to program Inventor.

This paper describes concepts and provides step-by-step instructions for two simple programs. The step-by-step instructions are highlighted in yellow, so if you just want to go through the examples you can focus on those and read the conceptual information later.

What Do I Need to Program Inventor?

To write a program for Inventor there are many choices about which language to use. A common way to get started writing programs for Inventor is to use the Visual Basic for Applications (VBA) that comes with Inventor. Here are some important things to understand about VBA.

- Free and comes with Inventor
- The best for Inventor RAD (Rapid Application Development)
- The best for debugging Inventor code
- Old technology
- Limited to macros that run in Inventor (can't create an add-in)
- Difficult to share programs
- Limited system functionality
- Custom dialogs are very limited

As you can see there are some good and bad points about VBA. I use it frequently for quickly testing a small bit of code and to make sure a certain workflow works the way I think it should. This is taking advantage of it as a RAD tool and its debugging capabilities. VBA was specifically designed to work with API's like Inventors and does it very well. The problem is that the language itself is old and somewhat limiting. I've chosen Visual Basic .NET for this lab because of its overall better capabilities.

Visual Basic .NET is an updated version of the Visual Basic used in VBA. It's very similar, but there are also some significant differences.

- New language that is similar to VBA
- Better language features
- Very powerful library (.NET)
- Creates exe's and dll's (dll's are needed to create add-ins)
- Easy to share programs
- Useful for programs besides customizing Inventor
- Excellent custom dialogs

You can also use other languages to program Inventor. C# is a sibling to Visual Basic .NET and provides essentially the same capabilities with a language that more closely resembles C++. Whether to use Visual Basic .NET or C# is more of a personal preference rather than a technical decision. My personal opinion is that Visual Basic is an easier language if you're not already programming in C++. If you are comfortable with C++, C# might be easier.

You can also program in C++, but it's much more difficult than Visual Basic or C#. There are also many other languages that can be used but they aren't recommended because of limitations and you'll have difficulty finding any samples in those languages or getting help if you do run into any problems.

To use Visual Basic .NET you'll need to install Microsoft's Visual Studio. Visual Studio is an Integrated Development Environment (IDE), which means it's an application that provides a user interface for writing programs. Visual Studio supports many languages including Visual Basic .NET, C# and C++. There are several versions of Visual Studio and all of them can be used when writing programs for Inventor.

Visual Studio Professional – This is the full, paid for, version of Visual Studio. There are several flavors of this too, where the more expensive versions have more capabilities for working with larger teams of programmers and include additional support from Microsoft. The base version of Visual Studio Professional is all you need to write any programs for Inventor. It retails for $499.

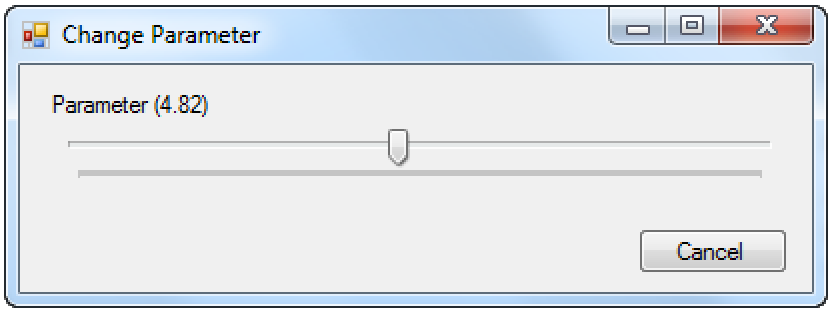
Visual Studio Community – This is also a full version of Visual Studio but is free. Along with the price, it comes some with some licensing restrictions in who can use it. Its use is limited to individual developers, open source projects, academic research, education, and small non-enterprise professional teams (those with less than 250 PCs or less than $1 Million US Dollars in annual revenue).

Visual Studio Express – This is a limited version of Visual Studio that is free. It has some limited capabilities but doesn't have the same license restrictions as the Community version and can be used to create commercial software. For most programs, the Express edition is fully capable. The primary limitation is that it's not as easy to write Inventor add-ins, although it is still possible.

Installing Visual Studio
For this lab, the screen shots are using Visual Studio Community, because that's what is installed on the Autodesk University lab computers. However, any of the versions of Visual studio will work and you shouldn't notice any differences in what we'll cover. You can find the installers for Visual Studio Express and Community on the Microsoft website. <u>If you're installing Visual Studio Express, you want to pick the "Express for Desktop" edition.</u> For community, there is only one edition and it contains everything.

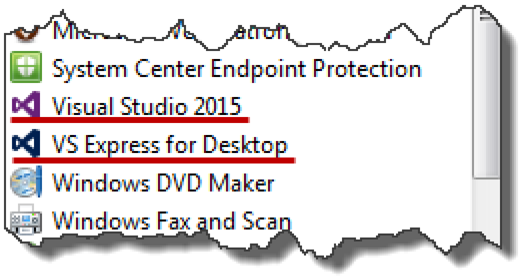
Program One

The first project is to create a dialog, as shown below, which changes a parameter within the active Inventor document as you move the slider.



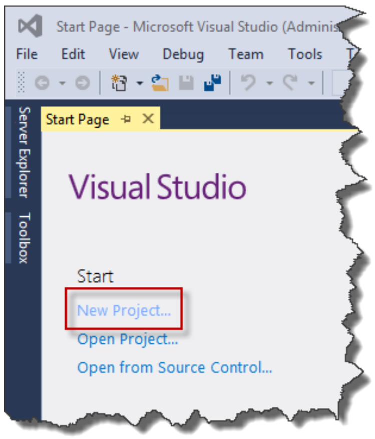
Creating a Visual Basic Project

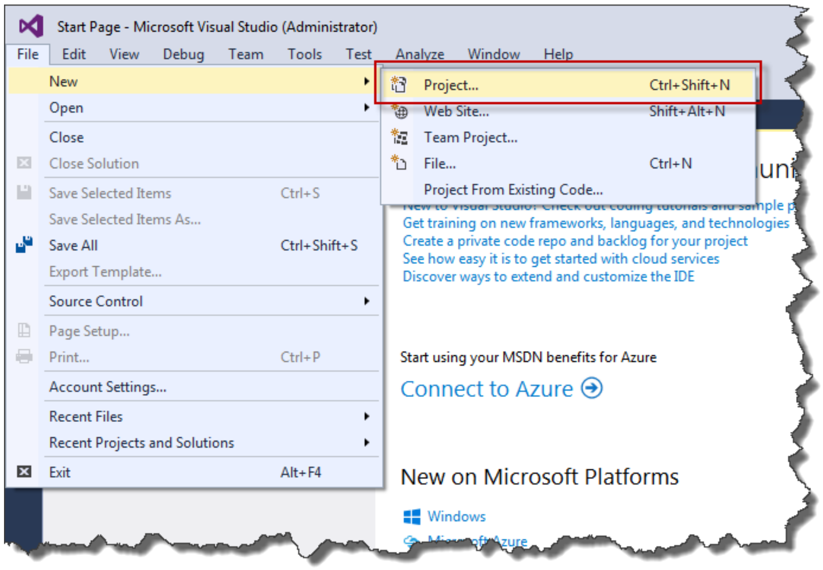
Once Visual Studio is installed you can run it from the Start menu as shown below.



When you first run it, you'll see a dialog that will let you choose the development settings and color theme. Because I use multiple languages I use the "General" development settings but if you're only using Visual Basic, "Visual Basic" is probably more appropriate. In actual use, there really isn't much difference. For the color theme, choose whatever you like.

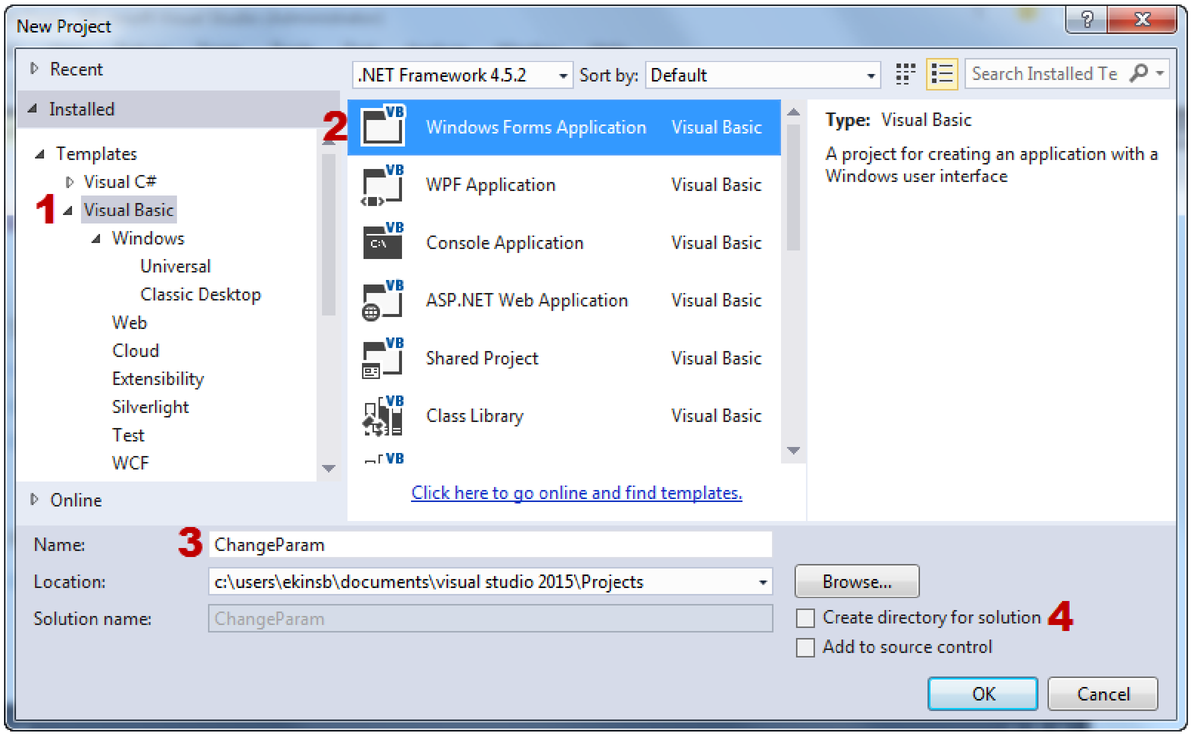
The first step in creating your program is to create a new project. A project is a file that contains all of the project settings and a list of the files associated with the project. To create a new project use the "New Project…" link on the start page or you can use the "Project…" command in the File->New menu, both of which are shown below.


Start Page - Microsoft Visual Studio (Adminis
File Edit View Debug Team Tools
Start Page
Server Explorer Toolbox
Visual Studio
Start
New Project...
Open Project...
Open from Source Control...


Start Page - Microsoft Visual Studio (Administrator)
File Edit View Debug Team Tools Test Analyze Window Help
New
Open
Close
Close Solution
Save Selected Items Ctrl+S
Save Selected Items As...
Save All Ctrl+Shift+S
Export Template...
Source Control
Page Setup...
Print... Ctrl+P
Account Settings...
Recent Files
Recent Projects and Solutions
Exit Alt+F4
Project... Ctrl+Shift+N
Web Site... Shift+Alt+N
Team Project...
File... Ctrl+N
Project From Existing Code...
Get training on new frameworks, languages, and technologies
Create a private code repo and backlog for your project
See how easy it is to get started with cloud services
Discover ways to extend and customize the IDE
Start using your MSDN benefits for Azure
Connect to Azure
New on Microsoft Platforms
Windows

This will display the “New Project” dialog as shown below, where you should follow these steps:

1. Choose “Visual Basic” from the list of templates on the left hand side of the dialog.
2. Choose the “Windows Forms Application” template from the center list.
3. Enter a name for the project. I’m using “ChangeParam”.
4. Uncheck the “Create directory for solution” box.


New Project

Recent

Installed

Templates
Visual C#
1 Visual Basic
Windows
Universal
Classic Desktop
Web
Cloud
Extensibility
Silverlight
Test
WCF

Online

.NET Framework 4.5.2 | Sort by: Default | Search Installed Te

2 Windows Forms Application	Visual Basic
WPF Application	Visual Basic
Console Application	Visual Basic
ASP.NET Web Application	Visual Basic
Shared Project	Visual Basic
Class Library	Visual Basic

Click here to go online and find templates.

Type: Visual Basic

A project for creating an application with a Windows user interface

Name: 3 ChangeParam

Location: c:\users\ekinsb\documents\visual studio 2015\Projects

Solution name: ChangeParam

Browse...

☐ Create directory for solution 4

☐ Add to source control

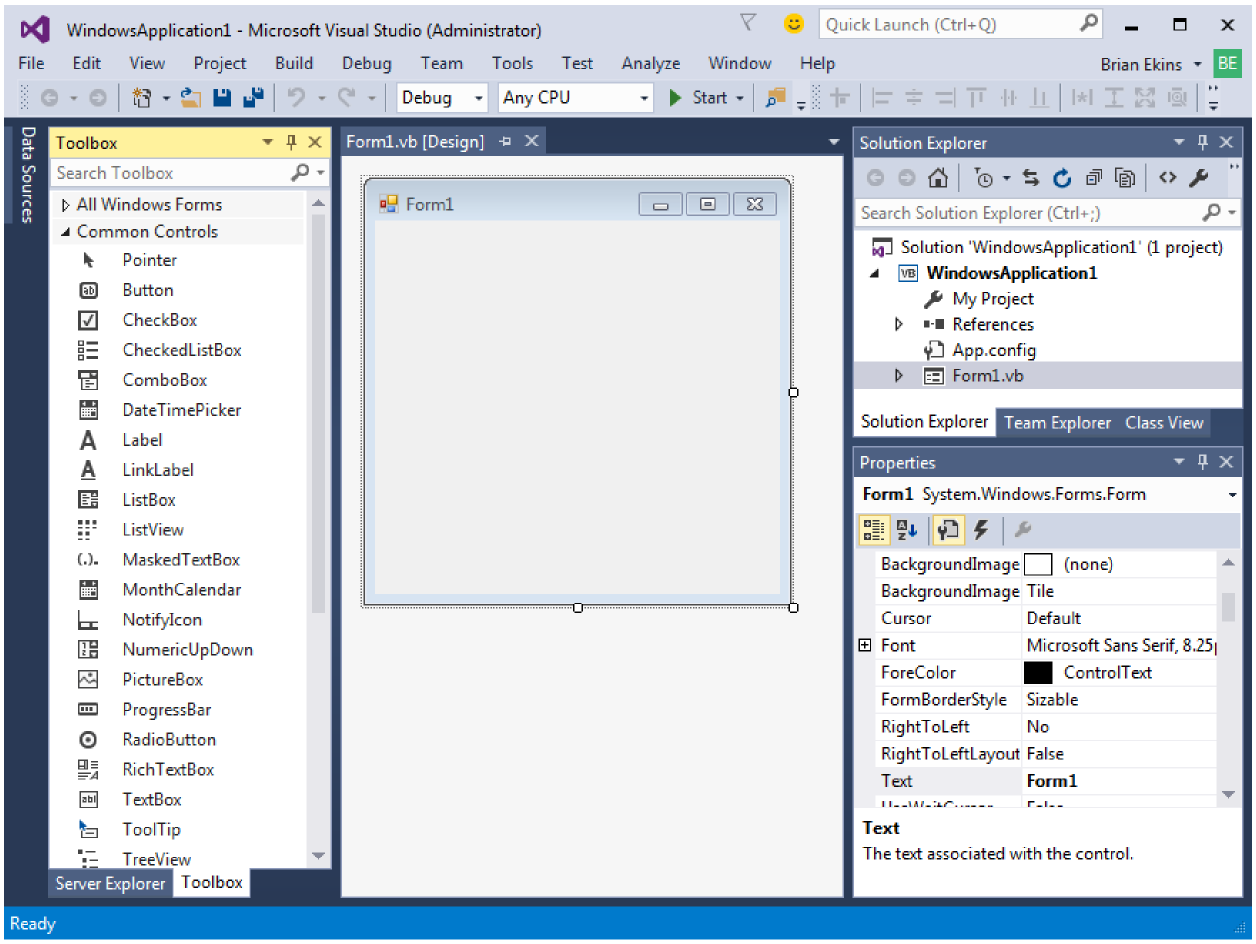
OK | Cancel

If the “Create directory for solution” checkbox is checked, then in addition to your project a *solution* is also created. A solution can own multiple projects and is generally used for larger, more complicated programs that are divided up into multiple projects. It’s likely that you’ll never need to use solutions and unchecking the “Create directory for solution” results in a simpler directory structure. If it’s checked then a directory is created for the solution and a subdirectory is created for the project. By unchecking this option a solution is still created but it is created in the project directory.

A Quick Overview of Visual Studio

Visual Studio is a very powerful IDE (Integrated Development Environment) that provides a suite of tools for creating software. It's likely that you'll never need most of the functionality that Visual Studio provides, however there are some basic functions that you'll use frequently

The main area of Visual Studio contains the form that you'll be defining. This area will also contain the code window once we get to the coding portion of the project. The **Solution Explorer** is a tree shown in the upper-right. This lists the files and external references for this project. The **Properties** window is in the lower-right and lists the properties of whatever is currently selected. In the picture below, the form is selected so the Properties window is displaying the properties of the Form object. The **Toolbox,** shown on the left, can be optionally shown and is where you access the various controls that can be added to a form. There are also toolbars and menus to access additional functionality.



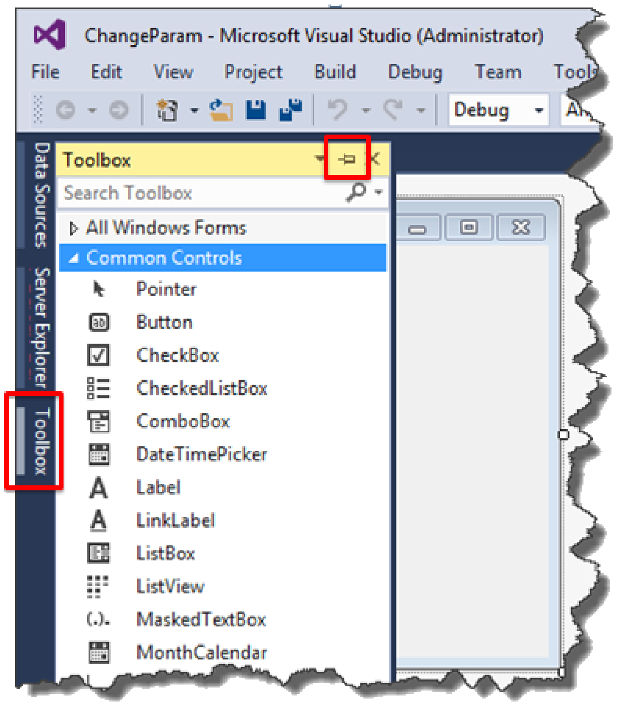
BackgroundImage	(none)
BackgroundImage	Tile
Cursor	Default
Font	Microsoft Sans Serif, 8.25
ForeColor	ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	**Form1**

Text
The text associated with the control.

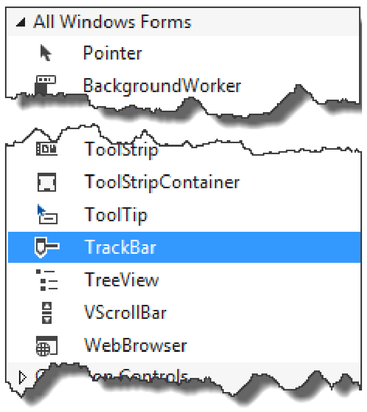
The "Visual" Part of Visual Basic

For the next step in creating the program we'll build the dialog. Using Visual Basic or C#, you can design a dialog visually without writing any code.

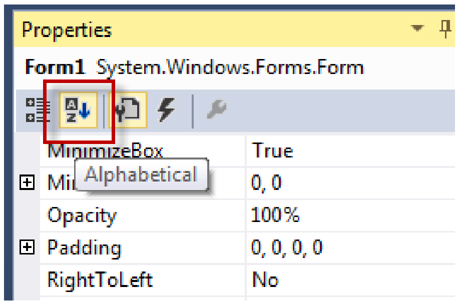
When you create a new project using the "Windows Forms Application" template, the project already has a single form associated with it. It's a blank form that you can add controls to provide whatever user interaction your program needs. The controls that you can place on the form are found in the Toolbox panel which is available on the left hand side of the Visual Studio window, as shown below. Clicking on "Toolbox" on the left-hand side of the window, as shown below, will display the "Toolbox" panel. I find it convenient when I'm designing a form to click the push-pin at the upper right of the "Toolbox" panel so that it remains visible and is easier to access.



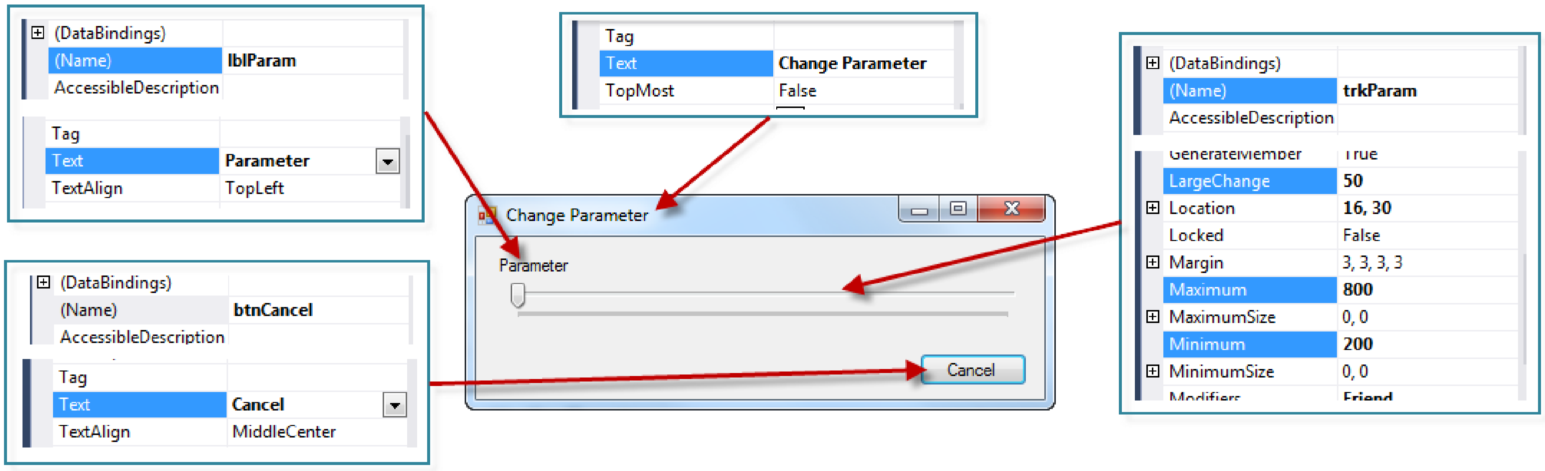
The form we're going to design for this first project is fairly simple and contains five controls; one button, two trackbars (or sliders) and two labels. The contents of the Toolbox are categorized to help organize the controls and make them easier to find. Usually you can use the "Common Controls" portion because it does contain the most commonly used controls. The Button and Label controls are in the "Common Controls" list as shown above. The TrackBar control is not in this list and you'll need to expand the "All Windows Forms" and look for "TrackBar", as shown to the right.



Below is a picture of the form and the properties of each control that you'll be changing. To place a control onto your form you can drag it from the Toolbox to the Form. Once it's on the form you can move and resize it and then use the Properties window to change its properties. Don't worry if it doesn't look exactly like the form shown below, just get it close. The Maximum and Minimum values of the track bar are being set with the idea that the parameter will vary from 2.0 to 8.0, which we'll see later. It can be easier to find the properties if you sort them alphabetically using the option shown to the right.



MinimizeBox	True
Mi...	0, 0
Opacity	100%
Padding	0, 0, 0, 0
RightToLeft	No



(DataBindings)	
(Name)	lblParam
AccessibleDescription	
Tag	
Text	Parameter
TextAlign	TopLeft

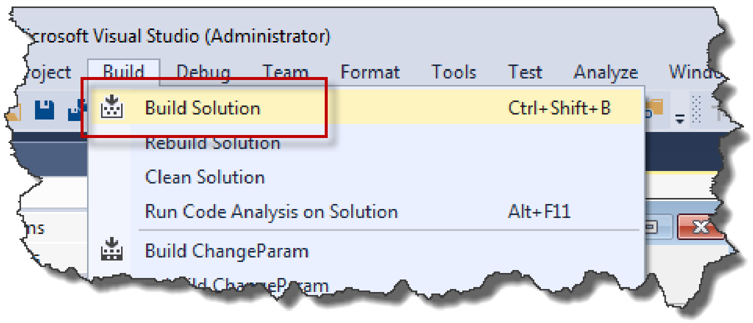
Tag	
Text	Change Parameter
TopMost	False

(DataBindings)	
(Name)	trkParam
AccessibleDescription	
GenerateMember	True
LargeChange	50
Location	16, 30
Locked	False
Margin	3, 3, 3, 3
Maximum	800
MaximumSize	0, 0
Minimum	200
MinimumSize	0, 0
Modifiers	Friend

(DataBindings)	
(Name)	btnCancel
AccessibleDescription	
Tag	
Text	Cancel
TextAlign	MiddleCenter

You'll notice above that the name of each control has been changed. The name is the unique identifier for each control and doesn't affect how the control looks. You should name the control something that makes sense to you. In the example above we have a label and a trackbar that will be used to change the value of a parameter. For the first set, I named them lblParam1 and trkParam. It's common practice to add a prefix to the name of control names that indicates the type of control. "lbl" indicates that the control is a label and "trk" is used for a trackbar. "btn" is used for a button. This isn't required but is a standard practice and does end up being useful when you do the programming part of the project.

Even at this point you can compile and run the program. To do that, click the **Start** command in the main toolbar, as shown below. Your form will appear and you can move the trackbar and click the Cancel button, but nothing will happen because you haven't defined the instructions for what's supposed to happen. Those instructions are the code, which we'll look at now.


crosoft Visual Studio (Administrator)
oject
Build
Debug
Team
Format
Tools
Test
Analyze
Windo
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Build ChangeParam

The Coding Part of Visual Basic

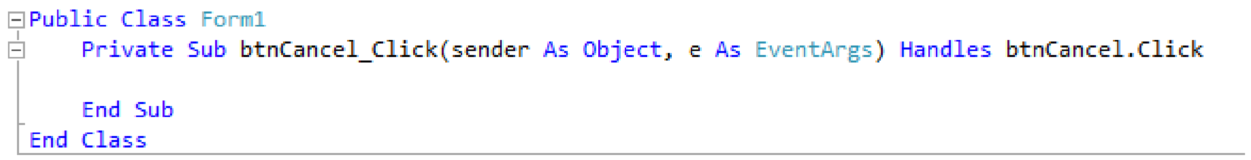
Once you've built your form, the next step is to add the code to tell Visual Basic what to do when the user interacts with your form. Each of the controls you placed on the form is available within your code as an *object*. You might also hear the term "class". You can think of a "class" as a blueprint that describes an object and the "object" is the physical thing has been created using the blueprint. Each object you create is an *instance* of the class. For example, I can have a blueprint that describes a chair but there can be things that can be unique for each chair built like the color or the height. These are the properties of the object.

Objects also support *methods*. These are actions that a particular object supports. For example, maybe I can tell the chair to move to a particular location. Objects can also support *events*. These are actions that happen to the object that it notifies you about. A chair would likely support a "sit" event to tell you when someone has started sitting in it.

You call methods or events by referencing an object use a "." and then the name of the property or method. For example, to set the value of the trackbar to 500 you would use the code below which sets the Value property to 500.

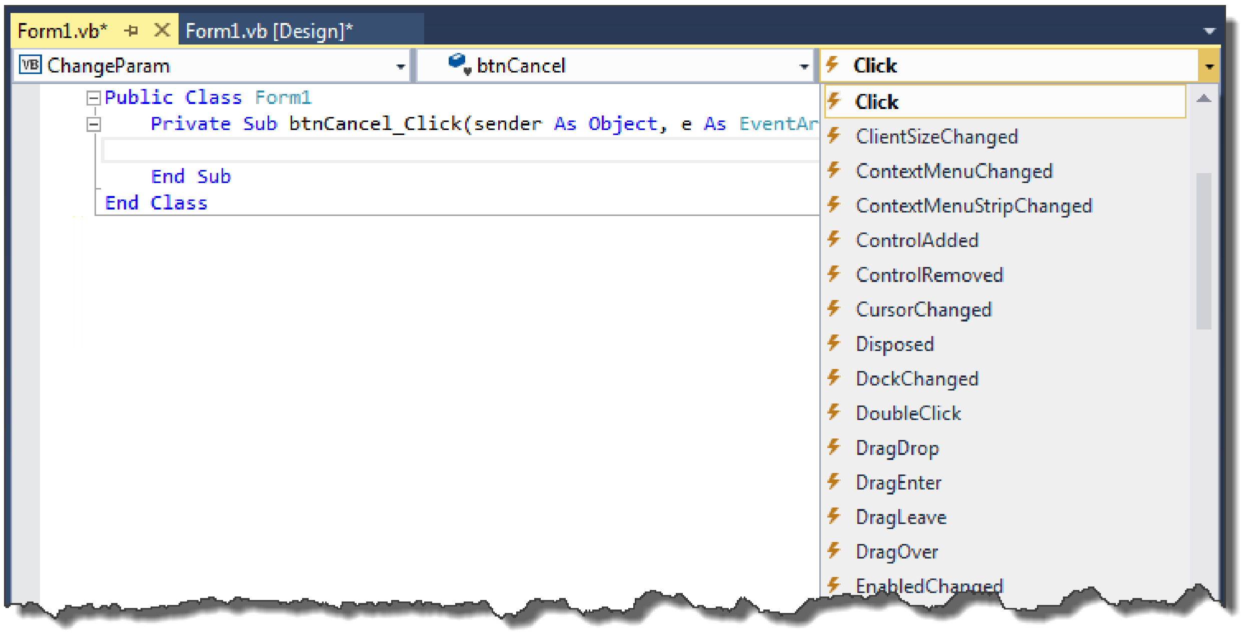
```
trkParameter.Value = 500
```

When writing code for a form you are typically responding to events. For example, you want to close the form when the user clicks the "Cancel" button so you need to add code to respond to the button's click event. To be able to respond to the button's click event you need add the handler code. Visual Basic makes this easy. If you double-click on the Cancel button, Visual Basic will open the Code window and will have created the event handler for the Click event, as shown below. The bntCancel_Click Sub that's been created will be called by Visual Basic whenever the "Cancel" button is clicked.



```
Public Class Form1
    Private Sub btnCancel_Click(sender As Object, e As EventArgs) Handles btnCancel.Click

    End Sub
End Class
```

Double-clicking on a control creates the event handler for the most common event associated with that control. Most controls support many events and you can access the other events from the Code window. There are three drop-downs at the top of the Code window. The first drop-down, which is “ChangeParam” in the example below, is the project. The next drop-down, which is “btnCancel” in this example, is the name of the control. If you click this dropdown you’ll see a list of all of the controls you’ve placed on the form. The next drop-down is a list of the events that are supported by the currently selected control. If the event is shown in bold, then there is already code in your program for that event and selecting it will take you that code. Otherwise it will create a new event handler for that event.



```
Public Class Form1
    Private Sub btnCancel_Click(sender As Object, e As EventAr

    End Sub
End Class
```

For the Cancel button click event add the `Me.Close()` code as shown below. Notice in the code above that the Click event handler is defined within a Class, which is the form. Using “Me” is referring to the owning class which in this case is the form so the code below is calling the Close method of the form object. The form is telling itself to close.

```
Private Sub btnCancel_Click(sender As Object, e As EventArgs) Handles btnCancel.Click
    Me.Close()
End Sub
```

To call an object’s methods and properties, you refer to the object by name, use a period, and then the name of the method or property. This is where spending time to rename your controls with a logical name pays off. The Close method is a simple method that doesn’t require any arguments but many methods need additional information that you pass through arguments. Below is an example of calling the Scale method on a button control and passing the scale factor as an argument.

```
btnCancel.Scale(1.5)
```

Properties are accessed in a similar way by using the object name, a period, and the property name but instead of just calling them like a method you can get and set their values. The code below is setting the Text property of the button, which changes the visible text on the button.

```
btnCancel.Text = "Abort"
```

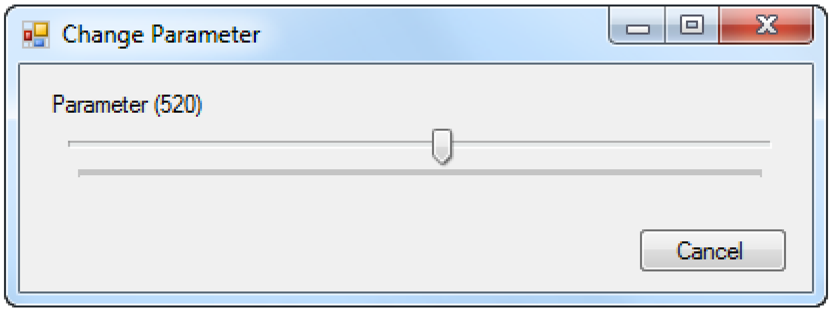
The code below sets the value of the label control to be whatever the current value of the trackbar is.

```
lblParam.Text = trkParam.Value
```

So now let's add some code to the trackbar to have it do something more interesting when you slide it. Double-clicking on a trackbar will create a Scroll event handler which is fired whenever the slider is moved. Add the code shown below for trkParam.

```
Private Sub trkParam_Scroll(sender As Object, e As EventArgs) Handles trkParam.Scroll
    lblParam.Text = "Parameter (" & trkParam.Value & ")"
End Sub
```

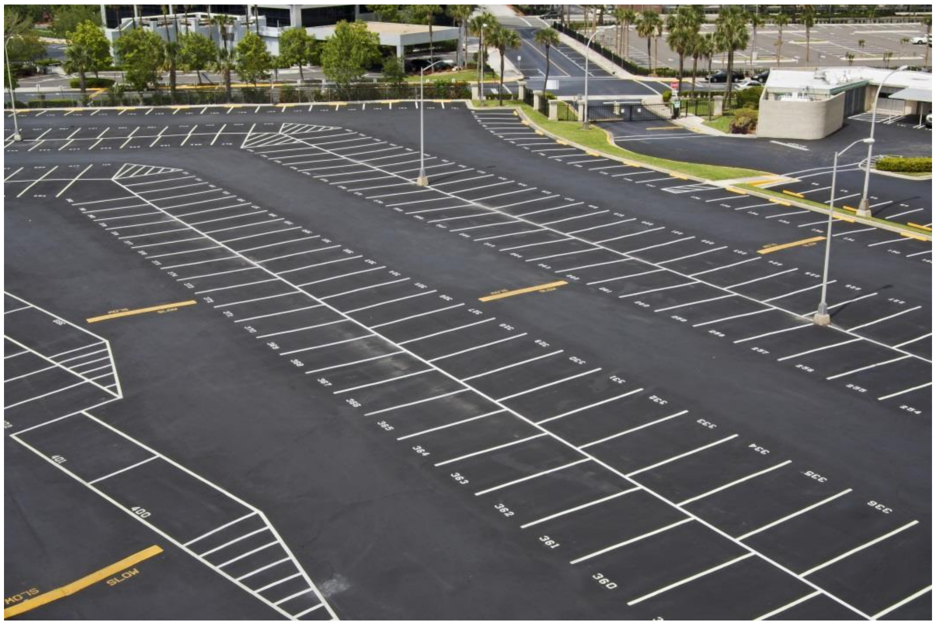
Now re-build your project using the "Build Solution" command in the Build menu and run your program using the "Start" command in the main toolbar. Try sliding the trackbar. You'll see that the current value of the trackbar is now being displayed in the label. It can vary between 200 and 800 because of the trackbar properties you set earlier. Clicking the Cancel button will dismiss the form.



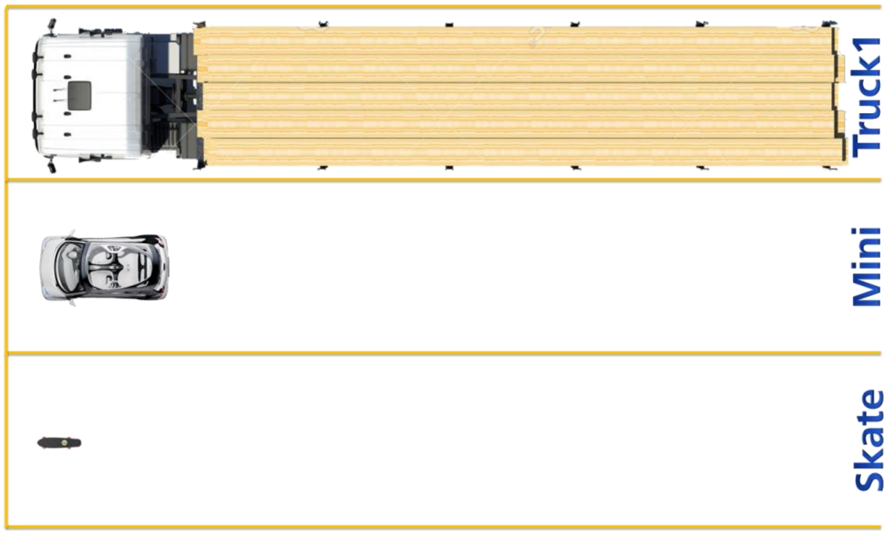
General Visual Basic Programming Concepts

Here are some general concepts that you need to understand when writing programs using Visual Basic, or any programming language. This is a very quick overview of a few concepts that we'll use in the sample program. Of course Visual Basic has a lot more capabilities that aren't discussed here.

Variables
Variables are something that are used in almost every program but they are also a concept that I've seen new programmers struggle with. Hopefully this explanation helps. Variables are a way for you to temporarily store numbers, strings, objects, and other types that Visual Basic supports. One way to visualize a variable is as a parking space in a parking lot like the one shown below. Notice how each parking space is uniquely identified by a number.



In your program you "declare" a variable, which using the analogy above is the equivalent of creating a parking space for the value. When you declare a variable you give it a name. This is the unique identifier for the variable and can be anything that makes sense to you but also has to follow some basic rules, (no spaces, can't start with a number, no special characters, and must be different than a few reserved keywords). In the picture below, three parking spaces have been created called Truck1, Mini, and Skate, where a truck is parked in the first space, a car in the second, and a skateboard in the third.



Creating a large parking space when you're going to use it for something small seems a bit wasteful. Even though the space is large and it might be storing something small, it can still only store one thing at a time. If you assign a new value to a variable it takes the place of the previous value.

Visual Basic is more efficient with variables and requires you to specify the type of value that will be stored in that variable. It's possible to create a variable as type "Object", like shown below, that can store anything but it's not recommended.

```
Dim Mini As Object
```

A better practice is to declare the variable as the specific type you want it to hold. In the example below, there are two variables declared as the Car type. Notice how the declared parking space is now just large enough to hold a car. In fact if you try to assign a truck or even a skateboard to that parking space Visual Basic will complain that the types don't match.

```
Dim Mini1 As Car
Dim Mini2 As Car
```

Mini1
Mini2

Some variable types that are commonly used in Visual Basic are:

- **Double** – This is used for floating point numbers. There is also another type called Single that can be used for floating point numbers but it's not as accurate. Inventor uses the Double type for all floating point numbers.

  ```
  Dim angle As Double = 30.5
  ```

- **Long, Integer** – These are used for whole numbers. T Long is used by Inventor and can store much larger numbers.
  - Long – Numbers up to 9,223,372,036,854,775,807
  - Integer – Numbers up to 2,147,483,647

  ```
  Dim numHoles As Long = 20
  ```

- **String** – This is used to store any textual information.

  ```
  Dim message As String = "This is the message"
  ```

A common misunderstanding is the difference between a number and a String when the string contains what looks like a number. For example, look at two variables below.

```
Dim longNum As Long = 136
Dim stringNum As String = "136"
```

The first one is a true number and is stored internally as 136. The second isn't a number but is the three characters "1", "3", and "6". This is a special case where all of the characters happen to represent numbers. Internally this is stored as (49 51 54), which are the look-up codes into a table that correspond with "1", "3", and "6".

When writing programs for Inventor it will be common to read numbers from a file or get a value typed by the user which will be the String (series of characters) representation of the number. There are utilities to convert Strings that can be a number into a true number. You can also convert a number into its String equivalent. In some cases, Visual Basic will do this conversion automatically so it might not seem like there is a difference even though there is.

This is demonstrated in the code you typed in earlier to react to the change in the track bar value, as shown below. The trkParam.Value property returns an Integer, which is a true number, but it's being combined with Strings and assigned to a label, which expects a String. This works because of the "&" operator which is used to combine strings. When it encounters a number it automatically attempts to convert it to the equivalent String.

```
lblParam.Text = "Parameter 1 (" & trkParam.Value & ")"
```

- **Inventor Types** – You'll see shortly that there are also types specific to Inventor.
- **Other Types** – There are other less commonly used types like Date, Byte, and Currency.

Subs and Functions

All of your code will exist within a Sub or Function. I'm not going to go into much detail here, but it is something you'll want to read about and get comfortable with. A Sub or Function is a grouping of code statements that you can execute by calling the sub or function. When you call a function or sub, you can pass it additional information it will need to perform whatever task it needs to do. The difference between a Sub and a Function is that a Sub doesn't return a value and a Function does. For this program we're going to rely heavily on some Subs that Visual Basic will create automatically as event handlers. An event handler is a sub that Visual Basic calls when a certain action occurs. An event handler sub is where you added the code to close the form earlier. In fact, all of the code you've written so far has been in event handlers.

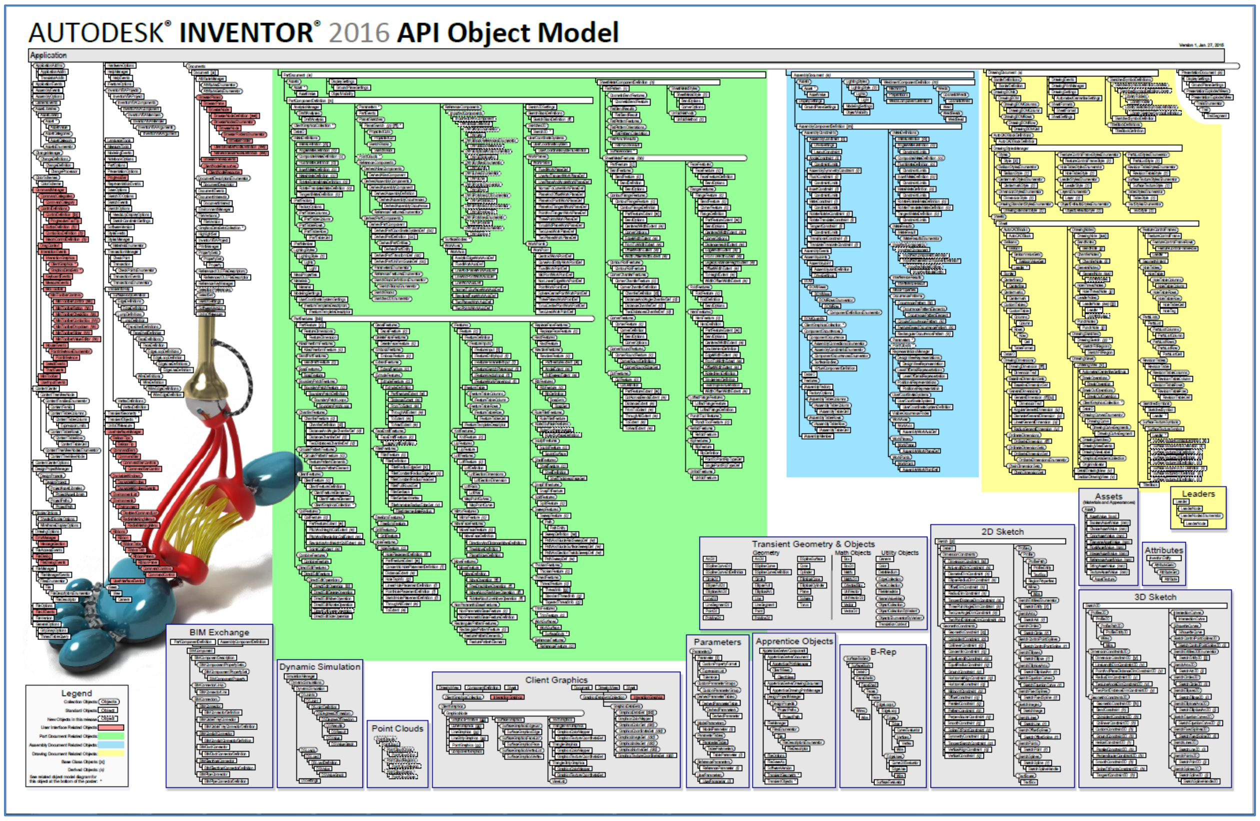
Working with Inventor

So far all of the code you've written has nothing to do with Inventor. Visual Basic is a generic programming tool that you can use to create programs that are completely independent of Inventor. However, in this case we want our program to interact with Inventor. The controls you added to the form are available as objects in your program where you can call their methods and properties and respond to their events. The same concepts apply when accessing Inventor from your program except now you'll be accessing Inventor specific objects and using methods, properties, and events supported by those objects.

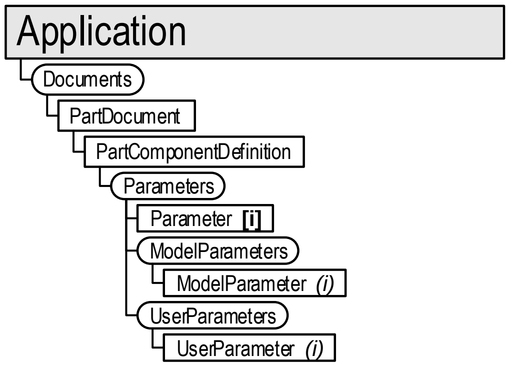
Inventor's Object Model

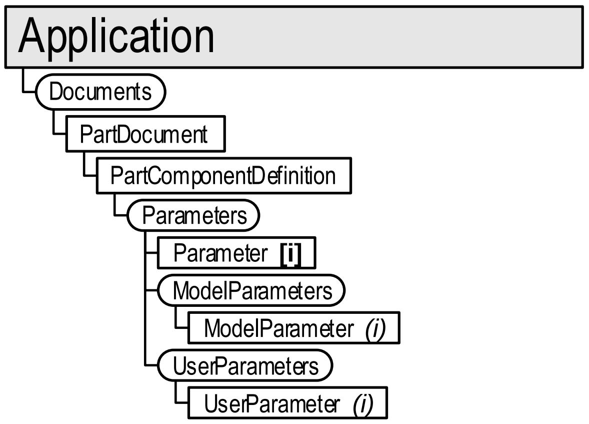
Inventor's programming interface is exposed as a set of objects. In fact, in Inventor 2016 there are 1444 objects which have 4661 methods, 16763 properties, and 172 events. That's a lot of functionality and can be very intimidating. However, as you'll see, you don't need to understand all 1444 objects to be able to program Inventor. Instead you just need to understand the few objects that pertain to the area of Inventor you want to work with. We'll be working with parameters and iProperties.

Inventor's objects are organized in what's referred to as the "Object Model" or "Object Hierarchy". 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 entire model that contains the objects we'll need for the parameter program and we can ignore the rest of the 1433 objects. The top object is the "Application" object and represents Inventor as a whole. This 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 is represented in the object model shown to the right.





If you're familiar with object orient programming and class diagrams, the object model is NOT a class diagram. Instead is shows ownership between objects. You can think of the Application object as owning everything under it, the Documents object owning every under it, and so on. You use properties on each object to access its children.

All of the boxes shown in the object model represent objects but some of them are a special type of object and are represented by boxes with rounded corners such as Documents, Parameters, ModelParameters, and UserParameters. These objects are known as *collection* objects. They're special for a couple of reasons. First, they are API utility objects and don't correlate to anything in the user interface. Second, collections provide access to a set of existing objects and in many cases they also support the creation of new objects. All collection objects support the *Count* property and the *Item* method. The Count property tells you how many objects are in the collection and the Item method returns a specific item from the collection. For example, the UserParameters object provides access to the user created parameters in a specific document. Its Count property tells you how many user parameters there are and the Item method lets you access a specific user parameter, either by an index number or by name. In addition, the UserParameters object supports the Add method that creates a new user parameter.

Let's look at the code below to see how this all works. The important thing to notice in this code is how it is traversing the object model shown above to get down to the Parameters object. The Documents object is skipped by using the ActiveDocument property to get the active document.

Line 2 is declaring a variable as "Inventor.Application" type. The "Inventor." prefix is used in the declaration to specify which library the object is defined within. Line 3 is calling the Visual Basic GetObject function and passing it the ID of Inventor, "Inventor.Application". If Inventor is running, this will get the Application object and return it. In this example it is assigning the result to the variable named "invApp".

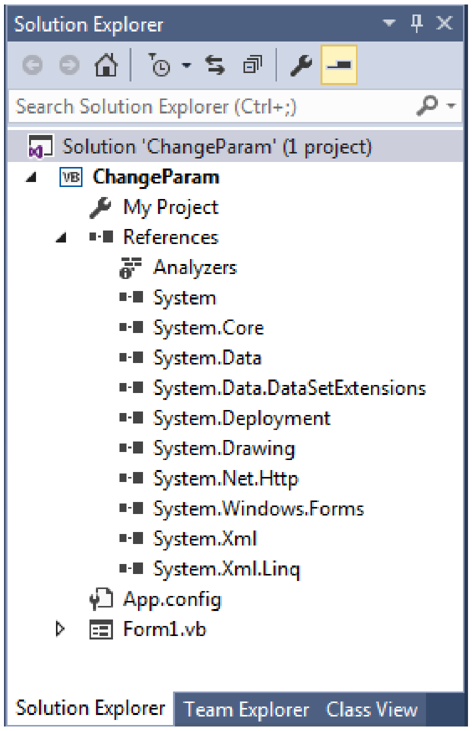
Line 5 is declaring a new variable as PartDocument type and line 6 calls the ActiveDocument property of the Application object, which returns whatever document is currently active and assigns it to the variable named "partDoc". The code assumes that a part document is active and will fail otherwise.

Line 8 calls the ComponentDefinition property of the PartDocument object. This returns a PartComponentDefinition object and then the Parameters property is called on it which returns a Parameters object and assigns this to the variable named "params".

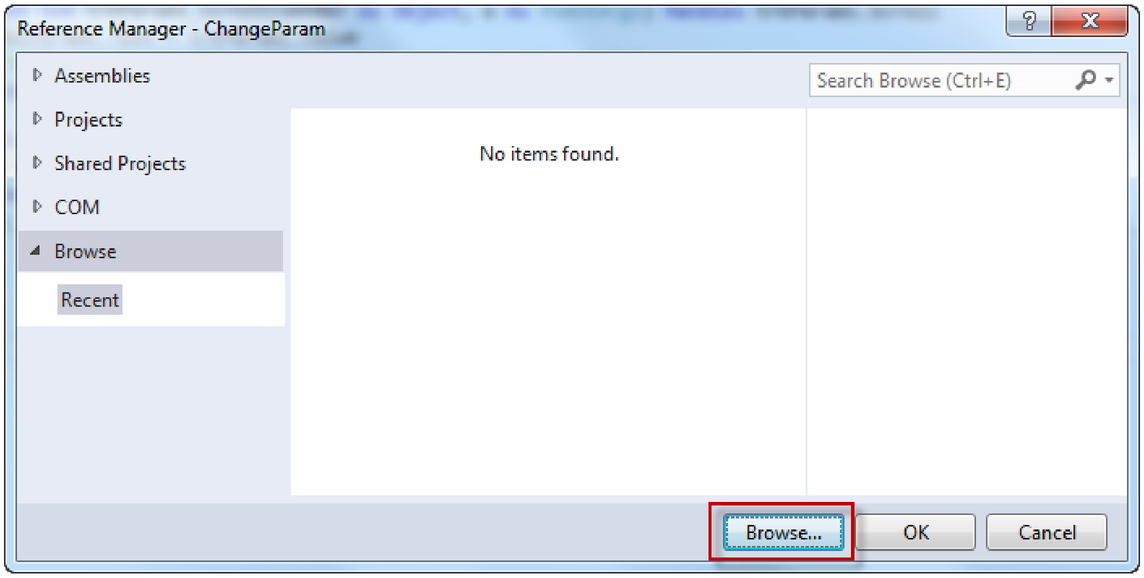
```
1)  ' Get the Inventor Application object.
2)  Dim invApp As Inventor.Application
3)  invApp = GetObject(, "Inventor.Application")

4)  ' Get the active document.  This assumes it's a part document.
5)  Dim partDoc As Inventor.PartDocument
6)  partDoc = invApp.ActiveDocument

7)  ' Get the Parameters collection.
8)  params = partDoc.ComponentDefinition.Parameters
```

Adding a Reference to Inventor

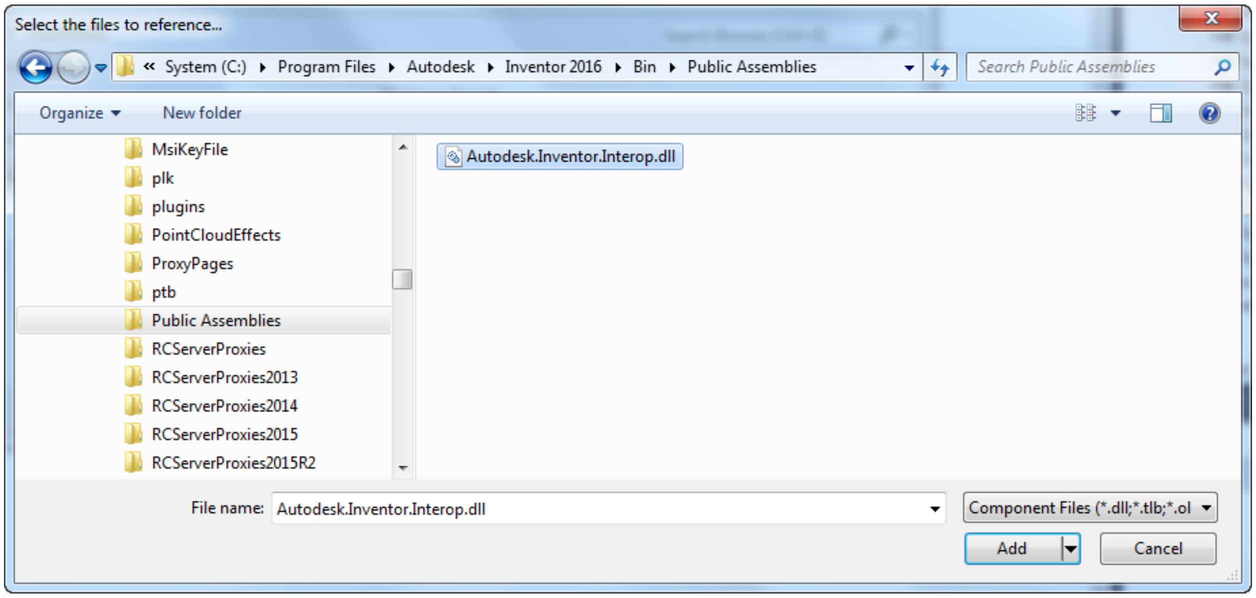
The code above refers to the “Inventor” library when it’s declaring the Inventor related variables. To be able to do this you need to make your Visual Basic project aware of Inventor. Your program is already referencing several .NET libraries to be able to accomplish what it’s doing. You can see those references by expanding the “References” node in the Solution Explorer, as shown below.



To be able to work with Inventor, you need to add a reference to the library that defines Inventor’s programming interface. You do this by using the “Add Reference…” command from the Project menu. In the **Reference Manager** dialog and clicking the “Browse…” button, as shown below.

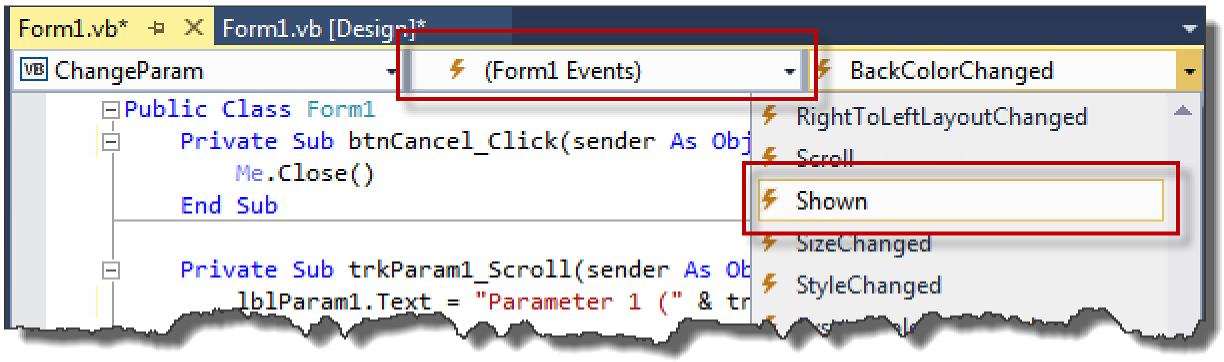


Use the dialog to browse to “C:\Program Files\Autodesk\Inventor 2016\Bin\Public Assemblies” and choose the file “Autodesk.Inventor.Interop.dll”, as shown below. Click “Add” and then “OK” in the **Reference Manager** dialog.



File name: Autodesk.Inventor.Interop.dll

Let’s add some more code to your program to be able to access Inventor. The workflow the program is designed for is that Inventor is running and the part that contains the parameters you want to change is open. You run your program and slide the track bars in the form to modify the model. With this workflow it will be best if your program connects to Inventor when the program is first executed. To do this you can react to the form shown event which Visual Basic calls when the form is first displayed. Create the Sub to handle the form Shown event. Do this by using the drop-downs at the top of the code window. Choose “Form1 Events” in the second drop-down and “Shown” in the third drop-down. This will create a new sub for handling the event when the form is shown.



```
Public Class Form1
    Private Sub btnCancel_Click(sender As Obj
        Me.Close()
    End Sub

    Private Sub trkParam1_Scroll(sender As Ob
        lblParam1.Text = "Parameter 1 (" & tr
```

Enter the code shown below into the event handler.

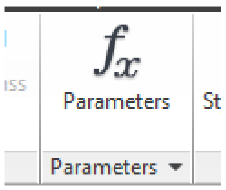
```
Public Class Form1
    Dim partDoc As Inventor.PartDocument
    Dim param As Inventor.Parameter

    Private Sub Form1_Shown(sender As Object, e As EventArgs) Handles Me.Shown
        ' Get the Inventor Application object.
        Dim invApp As Inventor.Application
        invApp = GetObject(, "Inventor.Application")

        ' Get the active document.  This assumes it's a part document.
        partDoc = invApp.ActiveDocument

        ' Get the Parameters collection.
        Dim params As Inventor.Parameters
        params = partDoc.ComponentDefinition.Parameters

        ' Get the Parameter using its name.
        param = params.Item("Length")
    End Sub

    Private Sub trkParam_Scroll(sender As Object, e As EventArgs) Handles trkParam.Scroll
        lblParam.Text = "Parameter (" & trkParam.Value & ")"
    End Sub

    Private Sub btnCancel_Click(sender As Object, e As EventArgs) Handles btnCancel.Click
        Me.Close()
    End Sub
End Class
```

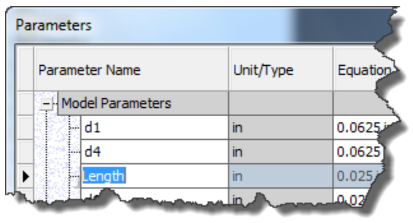
Now when you run your program it will connect to Inventor, get the Parameters collection from the active document, and get the parameter named “Length”.

Notice how the partDoc and param variables are declared outside of the event handler. That’s because of something known as *scope*. “Scope” is the idea of variables only being available to certain portions of your program. Because invApp and params are all declared inside the Form1_Shown sub, their scope is only within that sub. This means that other subs don’t have access to those variables. Delcaring partDoc and param outside of the sub but within the form class allows everything within the form class to use those variables.

The code above may seem like a lot of work just to get a parameter. However, as a user of Inventor you do essentially the same thing when you use it.

1. Getting the Inventor Application object is the same as finding a running Inventor in the task bar.
2. Getting the active document is a free operation in the UI because the active document is always available.
3. Getting the Parameters object is the same as running the Parameters command to open the Parameters dialog.
4. And getting the parameter named “Length” is the same as finding that parameter in the dialog.

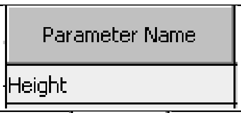


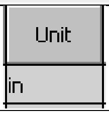

Parameters

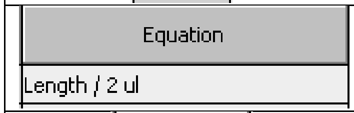
Parameter Name	Unit/Type	Equation
Model Parameters		
d1	in	0.0625
d4	in	0.0625
Length	in	0.025
	in	0.02

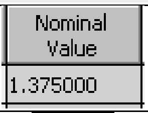
Like all collections, the Parameters collection supports the Item method that allows you to get an item within the collection. The Item method has a single argument to specify which item in the collection you want. All collections support using a number as the argument where the first item in the collection is 1. Some collections, like the Parameters collection, also support a String as input where the String is the name of the parameter. The code above gets the parameter named "Length". If there isn't a parameter named "Length" then the call will fail. It is also case sensitive so you need to be careful to match the case of the actual parameter.

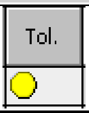
The Parameter object supports several properties which are shown in the chart below the portion of the user interface that is its equivalent.

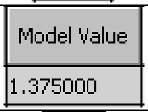
 Parameter Name
Height
	`Parameter.Name`
	
	`Ex: Parameter.Name = "Length"`

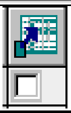
 Unit
in
	`Parameter.Units`
	
	`Ex: Parameter.Units = "mm"`

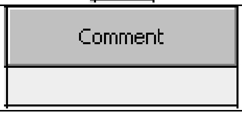
 Equation
Length / 2 ul
	`Parameter.Expression`
	
	`Ex: Parameter.Expression = "Height/2 + 5 in"`

 Nominal Value
1.375000
	`Parameter.Value`
	
	`Ex: MsgBox("Nominal Value: " & Parameter.Value & " cm")`

 Tol.
●
	`Parameter.ModelValueType`
	
	`Ex: Parameter.ModelValueType = kUpperValue`

 Model Value
1.375000
	`Parameter.ModelValue`
	
	`Ex: MsgBox("Model Value: " & Parameter.ModelValue & " cm")`

 ☐	`Parameter.ExposedAsProperty`
	
	`Ex: Parameter.ExposedAsProperty = True`

 Comment	`Parameter.Comment`
	
	`Ex: Parameter.Comment = "This is the comment."`

The final task is to change the value of the parameter as the track bar is scrolled. When you created the track bar you changed its properties to set the min value to 200 and the max to 800 with the idea that the parameter will actually vary between 2.0 and 8.0. All we need to do is divide the trackbar value by 100 to get the usable value. Change the existing Scroll event to the following:

```
    Private Sub trkParam_Scroll(sender As Object, e As EventArgs) Handles trkParam.Scroll
        Dim newValue As Double = trkParam.Value / 100

        param.Value = newValue
        partDoc.Update()

        lblParam.Text = "Parameter (" & newValue & ")"
    End Sub
```

The code above sets the value of the parameter and then calls the Update method of the PartDocument object. Calling the Update method is the same as clicking the 🗲 button in the QAT.

Parameter Expression vs. Value

The code above changes the value of the parameter by using the Expression property of the Parameter object. The Parameter object also has a property called Value. It's important to understand the difference between these two when working with parameters. They're both useful but usually one will be more applicable than the other depending on what you're doing.

The Expression property is exactly the same as editing the value of a parameter using the Parameters dialog. Let's say you're working in a document that has inches set as the length unit and there is a parameter that controls the length of something. If you type "6" for the value, it is interpreted as six inches. But you can also type "6 mm" to override the default unit type. In addition you can get much more complex by using equations, calling functions, and referencing other parameters. For example you could enter something like "sqrt(3 in * 2 cm) + Length". If you get the value of the Expression property it will return the full equation. The type of the Expression property is a string and it's evaluated by Inventor to compute the actual numerical value of the parameter.

The Value property is an API specific capability. It provides direct access to the real value of the parameter. The type of the Value property is a Double and it's always in the internal units of Inventor which means lengths are ALWAYS centimeters and angles are ALWAYS radians. If you set the Value property, Inventor will create an appropriate expression and replace the current expression.

In general, if you've gotten input from the user then you should use the Expression property because you'll get Inventor's advanced expression evaluation behavior. If you're computing the value, (for example, you have a function that computes the length of a part), you should typically use the Value property. This eliminates the need for you to convert the computed value into a string. However, remember that the Value property is always in internal units. In this example program I chose to use the Expression property because I want the value to be in whatever units the user has chosen as the current document units. The program is changing the "Length" parameter from 2 to 8. If the document is an inch document it will change from 2 inches to 8 inches. If it's a millimeter document it will change from 2 millimeters to 8 millimeters.

Running Your Program

Create a simple part that contains a parameter named "Length". Run your program and move the trackbar. You should see your part dynamically change.

Making it Better

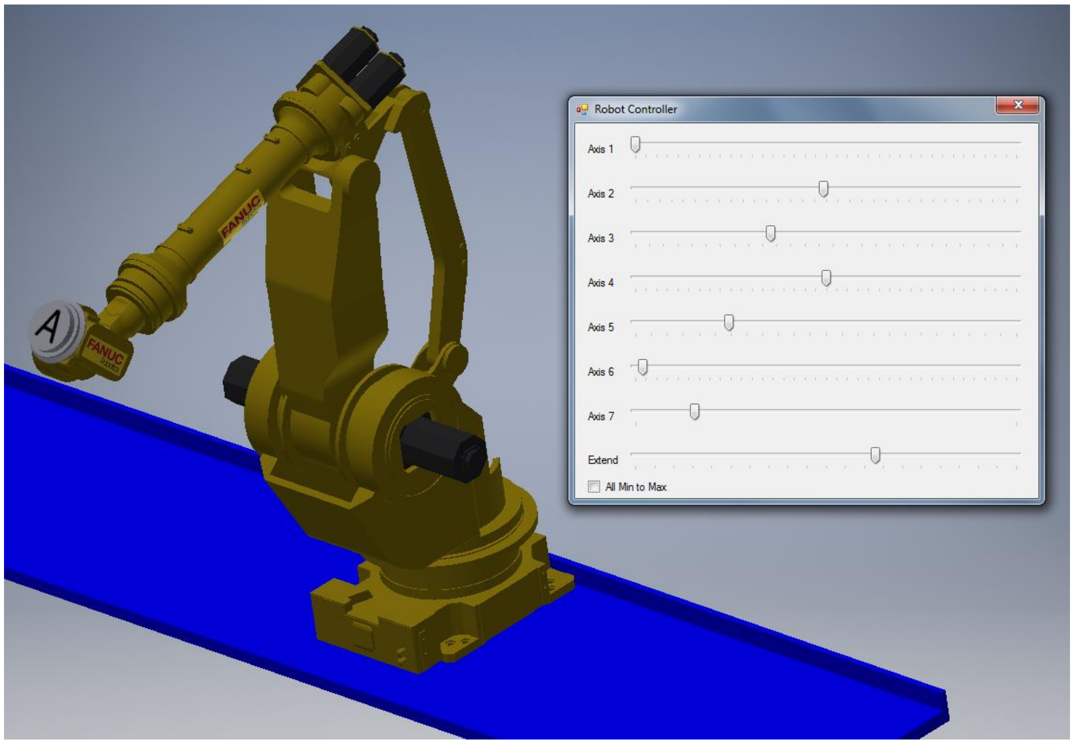
In the short time we have for this lab there are several things we've had to skip over but they are important for a more robust and full featured program. Here's a quick list of some things to consider.

1. What if you run the program and Inventor isn't running?
2. What if you run the program and a part isn't open.
3. What if you want the program to work in a part or an assembly?
4. When you start the program the sliders don't show the current value of the parameters.
5. What if the parameters being edited don't exist in the open document?
6. What if I don't want to hard-code the variable names and their min/max values?
7. How do I run my program from Inventor?

There's a larger code sample that handles all but the last of these issues that you can look at and see my approach to providing the additional functionality. I discuss possible solutions to the last question at the end of this paper.

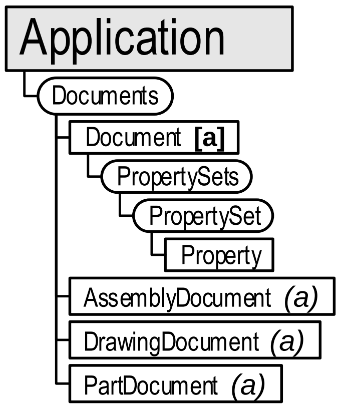
So Many Possibilities

What we've done here is very simple but hopefully gives you an idea of how the API for parameters works. Below is an example of a robot where the all of robot's axes are controlled using track bars in a dialog. This is a practical example of how something simple like you've done can be expanded and applied to a real-world problem.



Working with iProperties

Another area of the API that is commonly used is iProperties. The picture to the right is the portion of the object model that applies to iProperties. The objects that provide access to iProperties are the PropertySets, PropertySet and Property objects, which we'll look at below. They are accessed through the Document object.



Derivation

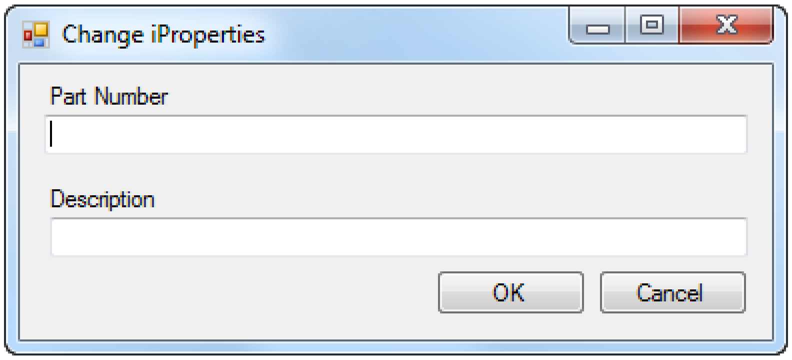
A useful topic to understand when working with the API is something called *derivation*. When working with Inventor, you never use a generic document but instead it's always a specific type of document, i.e. Part, Assembly, Drawing, etc. The same is true with parameters. In the previous program we were using a generic Parameter object, but parameters are always a specific type, i.e. Model, User, Reference, etc. The programming terminology is that the Parameter object is the *base* class and the specific parameter types (ModelParameter, UserParameter, etc.) are *derived* from it. The same is true with documents; Document is the base class and the other types are derived from it. As I said earlier, the object model chart shows ownership, but it also tries to show these derived relationships. In this case the bold "a" **[a]** in square brackets indicates that the Document object is a base class and the italic "a" in parentheses *(a)* indicates that the AssemblyDocument, DrawingDocument, and PartDocumet objects that derive from Document.

One way to think of this is like taxonomy in biology. Document can be considered a mammal object and the objects that derive from it can be considered more specific types like cow, dog, or dolphin. The mammal object defines all of the methods and properties that apply to all mammals. The derived objects automatically inherit everything from the base class and then add additional functionality that's unique to that object.

All documents share a lot of similar functionality, but obviously there is functionality that's specific to each type of document. The important thing in this case is that iProperties is functionality that all documents share so we can write the program below to work with the Document object and it will automatically work for parts, assemblies, drawings, etc.

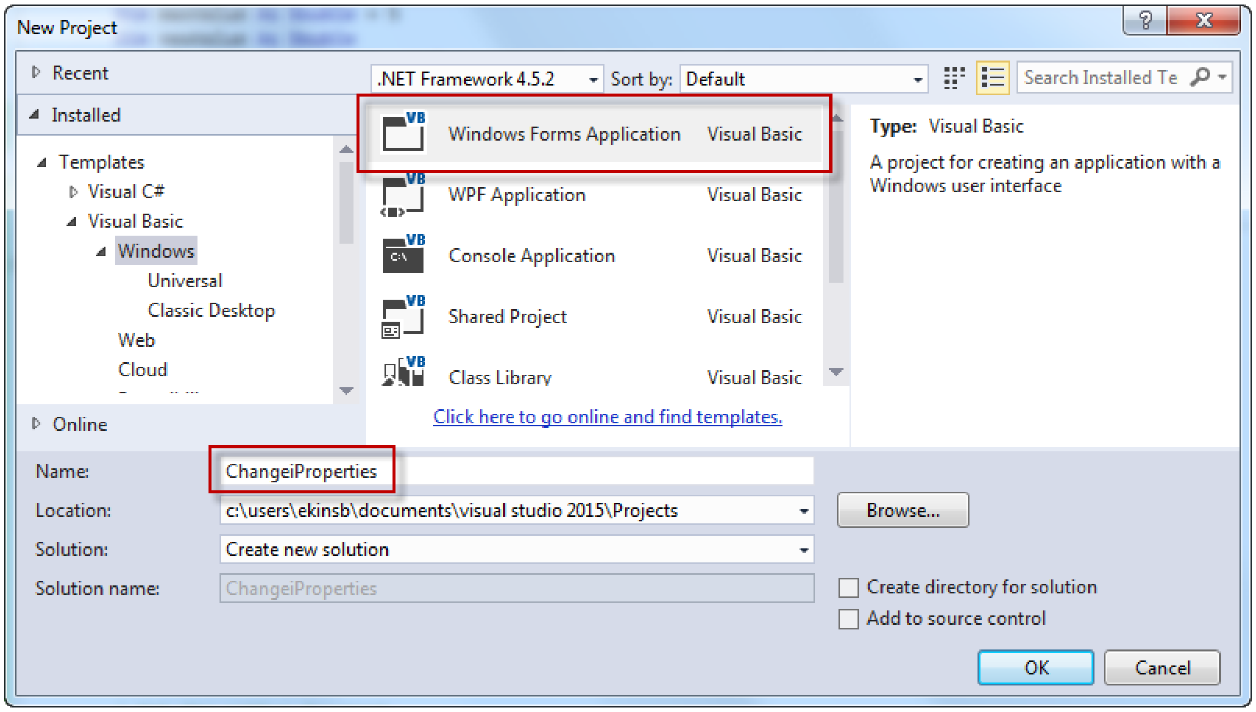
Program Two

The second project is to create a dialog, as shown below, which changes iProperty values in the active document.



Build your Form

Create a new Windows Form project. I named mine “ChangeiProperties”.


New Project
Recent
Installed
Templates
Visual C#
Visual Basic
Windows
Universal
Classic Desktop
Web
Cloud
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Te
Windows Forms Application Visual Basic
WPF Application Visual Basic
Console Application Visual Basic
Shared Project Visual Basic
Class Library Visual Basic
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual Basic

A project for creating an application with a Windows user interface

Name: ChangeiProperties

Location: c:\users\ekinsb\documents\visual studio 2015\Projects

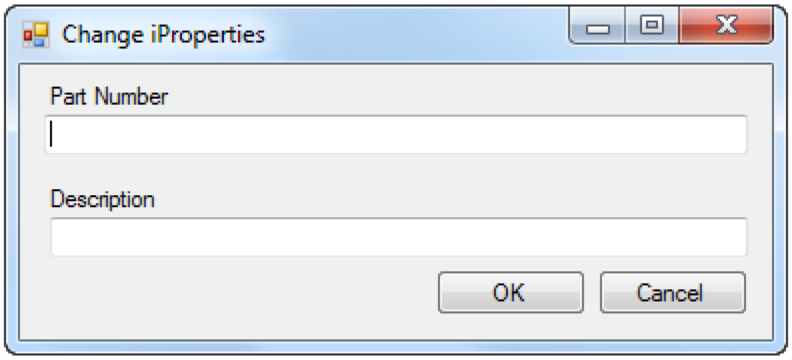
Solution: Create new solution

Solution name: ChangeiProperties

☐ Create directory for solution

☐ Add to source control

Add controls to the form so it looks similar to what’s shown below. In this example, there are two labels, two text boxes, and two buttons.



For this example, we won’t do anything as the user edits the text box values. It’s possible to listen to events and react as the text is changed, but there’s no reason to do that In this case because all of the work will happen when either the OK or Cancel buttons are clicked.

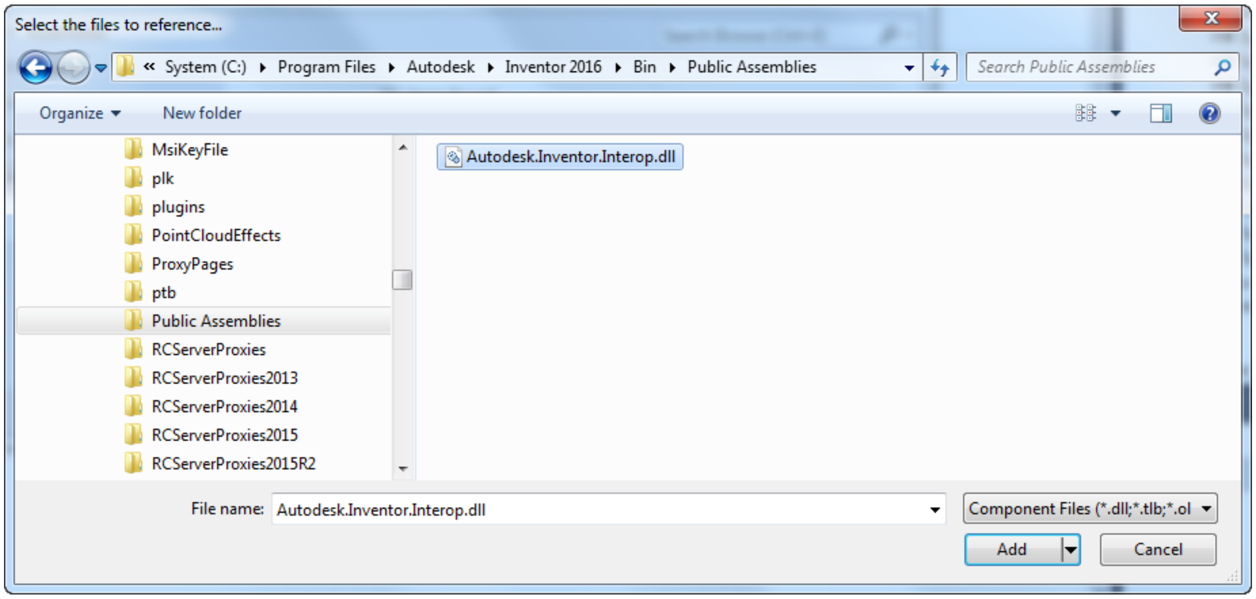
When the Cancel button is clicked, nothing should happen except that the form will be dismissed. This is done exactly the same as in the parameter sample. Create the event handler for the Click event and add the code shown below.

```
Private Sub btnCancel_Click(sender As Object, e As EventArgs) Handles btnCancel.Click
    Me.Close()
End Sub
```

When the OK button is clicked you need to read the values from the text boxes and assign those to the correct properties in the active document. Let's break this down into the steps needed and then work through each one.

1. Connect to Inventor.
2. Get the active document.
3. Get the iProperty objects that represents the part number and description.
4. Assign the text from the part number and description text boxes to the associated iProperty object.
5. Dismiss the form.

Add a reference to the Inventor library, just the same as you did before using "Add Reference…" command from the Project menu.



File name: Autodesk.Inventor.Interop.dll

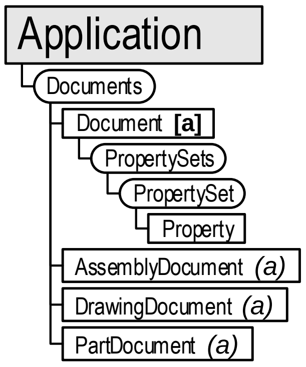
1. **Connect to Inventor**
   Create the event handler for the Click event of the OK button and add the code shown below. This is the same as in the parameter example except we're doing it in reaction to a different event.

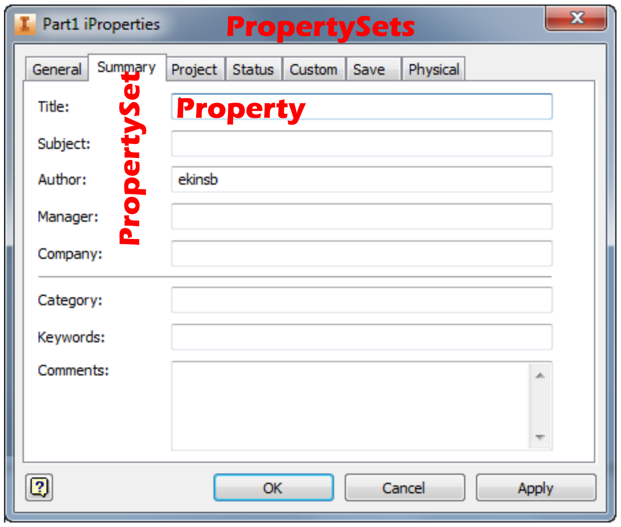
```
Private Sub btnOK_Click(sender As Object, e As EventArgs) Handles btnOK.Click
    ' Get the Inventor Application object.
    Dim invApp As Inventor.Application
    invApp = GetObject(, "Inventor.Application")
End Sub
```

2. **Get the active document**
   Add the code below to get the active document.

```
Private Sub btnOK_Click(sender As Object, e As EventArgs) Handles btnOK.Click
    ' Get the Inventor Application object.
    Dim invApp As Inventor.Application
    invApp = GetObject(, "Inventor.Application")

    ' Get the active document.
    Dim doc As Inventor.Document
    doc = invApp.ActiveDocument
End Sub
```

3. **Get the iProperty that represents the part number.**
   To get to a specific iProperty you need to understand how iProperties are structured in the Inventor Object Model. You can see from the diagram that you get a PropertySets object from the Document object. This is a collection of PropertySet objects. A PropertySet is a collection of Property objects. The best way to visualize these is to think of the ProperySets object as being equivalent to the iProperties dialog. It is the high-level object that provides overall access. A PropertySet object provides the equivalent of each tab in the dialog and the Property object is the equivalent of each value.





Title:	
Subject:	
Author:	ekinsb
Manager:	
Company:	
Category:	
Keywords:	
Comments:	

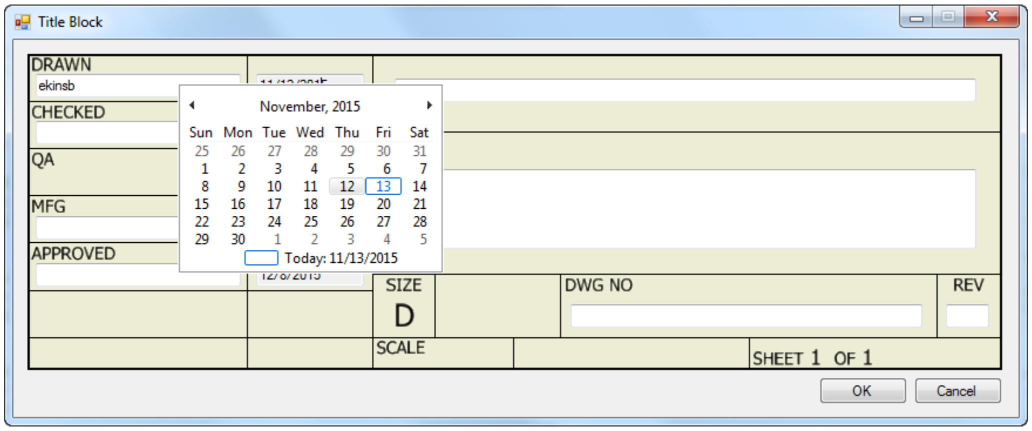
What this means is that to get to a particular property you first need the document that contains the property, then you need to know which property set the property is in and finally the name of the specific property. Below is a chart that contains a list of the standard property sets and their properties. The bold names are the names of the PropertySet objects. Under each property set is a list of the names of the Property objects and what type of value they are. You can see that most of them are in the "Design Tracking Properties" property set. An interesting one is the first property set listed; "Inventor User Defined Properties". This property set contains the properties you create when using the "Custom" tab of the iProperties dialog. The name of these properties is whatever you named them. The two properties we care about here are "Description" and "Part Number" which are both part of the "Design Tracking Properties" property set.

Inventor User Defined Properties	
Inventor Summary Information	
Author	String
Comments	String
Keywords	String
Last Saved By	String
Revision Number	String
Subject	String
Thumbnail	Picture
Title	String
Inventor Document Summary Information	
Category	String
Company	String
Manager	String
Design Tracking Properties	
Appearance	String
Authority	String
Catalog Web Link	String
Categories	String
Checked By	String
Cost	Currency
Cost Center	String
Creation Time	Date
Date Checked	Date
Defer Updates	Boolean
Density	Double
Description	String
Design Status	Long
Designer	String
Document SubType	String
Document SubType Name	String
Engineer	String
Engr Approved By	String
Engr Date Approved	Date
External Property Revision Id	String
Flat Pattern Area	Double
Flat Pattern Length	Double
Flat Pattern Width	Double
Language	String
Last Updated With	String
Manufacturer	String
Mass	Double
Material	String
Material Identifier	String
Mfg Approved By	String
Mfg Date Approved	Date
Parameterized Template	Boolean
Part Icon	Picture
Part Number	String
Part Property Revision Id	String
Project	String
Proxy Refresh Date	Date
Sheet Metal Area	String
Sheet Metal Length	String
Sheet Metal Rule	String
Sheet Metal Width	String
Size Designation	String
Standard	String
Standard Revision	String
Standards Organization	String
Stock Number	String
SurfaceArea	Double
Template Row	String
User Status	String
Valid MassProps	Long
Vendor	String
Volume	Double
Weld Material	String

Add the highlighted code below to use the names listed in the previous table to get the "Design Tracking Properties" property set and the "Description" and "Part Number" properties from it. The code then assigns the current value of the text boxes in the dialog to the iProperties.

```
Private Sub btnOK_Click(sender As Object, e As EventArgs) Handles btnOK.Click
    ' Get the Inventor Application object.
    Dim invApp As Inventor.Application
    invApp = GetObject(, "Inventor.Application")

    ' Get the active document.
    Dim doc As Inventor.Document
    doc = invApp.ActiveDocument

    ' Get the "Design Tracking Properties" property set.
    Dim designTrackPropSet As Inventor.PropertySet
    designTrackPropSet = doc.PropertySets.Item("Design Tracking Properties")

    ' Get the "Description" property from the property set.
    Dim descProp as Inventor.Property
    descProp = designTrackPropSet.Item("Description")

    ' Set the value of the property using the current value of the text box.
    descProp.Value = txtDescription.Text

    ' Get the "Part Number" property from the property set.
    Dim partNumProp as Inventor.Property
    partNumProp = designTrackPropSet.Item("Part Number")

    ' Set the value of the property using the current value of the text box.
    partNumProp.Value = txtPartNumber.Text
End Sub
```

The program you just created is very simple and doesn't really provide that much of a benefit over using the iProperties dialog itself. However, here's a more real-world example that's just more of what you've already done where you can easily see how a task specific program can simplify something like filling out a title block. Not only is the interface more intuitive but you can also provide lists of valid choices and validate the data that's entered.



DRAWN ekinsb

12/8/2015

November, 2015

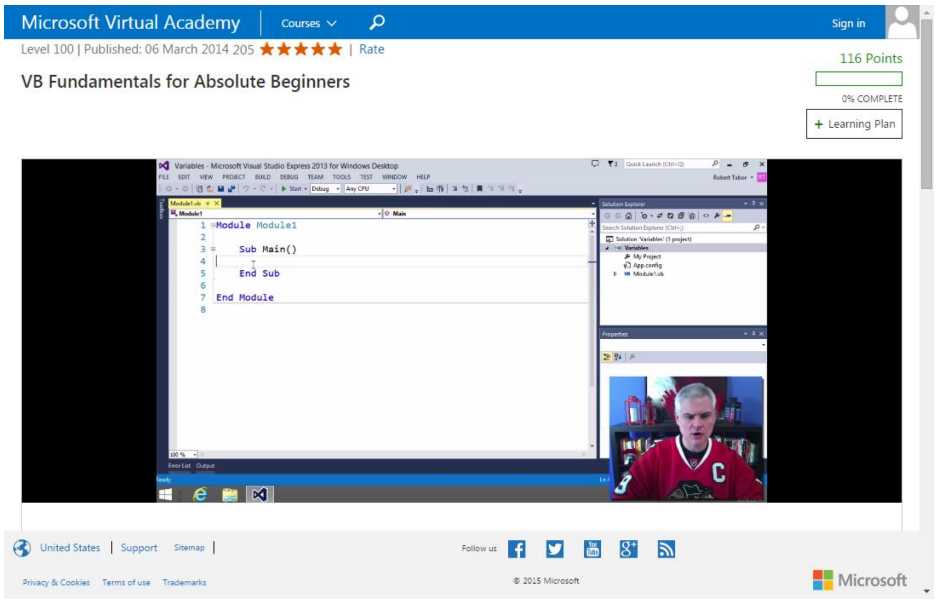
Sun	Mon	Tue	Wed	Thu	Fri	Sat
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

Today: 11/13/2015

What's Next?

This lab was intended to give you a small taste of what it's like to use Inventor's programming interface. Now that you've gotten started, where do you go from here? There are two things you need to learn more about to be an effective Inventor programmer; your programming language and Inventor's API.

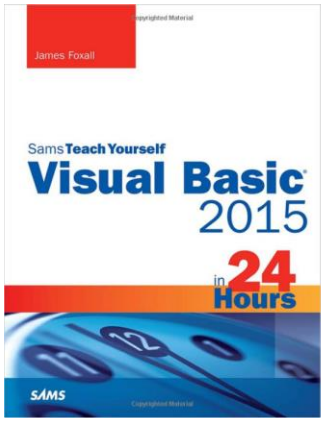
To learn more about Visual Basic there are some useful books and there are also some very good online resources. I would recommend starting with "Microsoft Virtual Academy" (MVA). This contains many videos on many programming topics. There is an excellent course called "VB Fundamentals for Absolute Beginners" by Bob Tabor that is a series of 26 video lessons that fills in some gaps we didn't have time to cover here and goes over a lot more. This is enjoyable to use and is free.

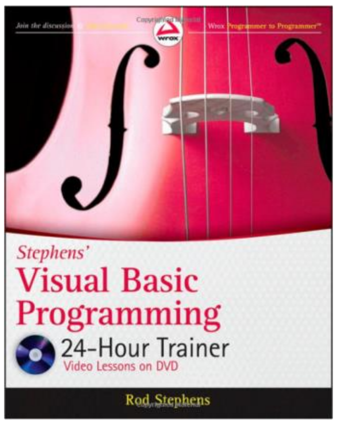

Microsoft Virtual Academy
Courses
Sign in
Level 100 | Published: 06 March 2014 205 ★★★★★ | Rate
VB Fundamentals for Absolute Beginners
116 Points
0% COMPLETE
+ Learning Plan
United States | Support Sitemap |
Follow us
Privacy & Cookies Terms of use Trademarks
© 2015 Microsoft
Microsoft

```
Module Module1

    Sub Main()

    End Sub

End Module
```

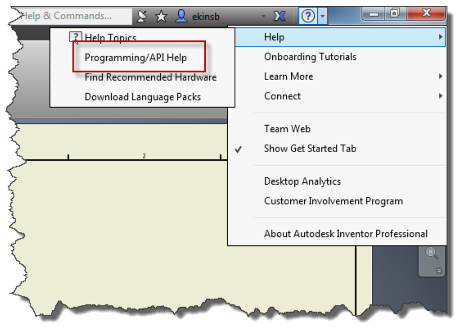
Microsoft also has some other Visual Basic training material as part of their MSDN material. Their "Introduction to the Visual Basic Programming Language" material is very good and is also free.

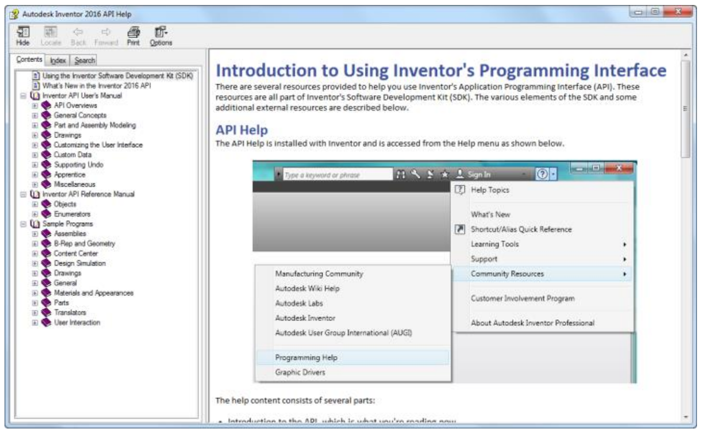
For books, there are a few books I've used. Some have become dated and were targeted towards people moving from the older version of Visual Basic to Visual Basic .NET. For new programmers, I've found the "Visual Basic 2015 in 24 Hours" to be good and also recently discovered "Stephens' Visual Basic Programming" which includes video lessons on DVD.





For learning more about Inventor's programming interface, there are several resources available. There is programming help that you can access from within Inventor, as shown below.





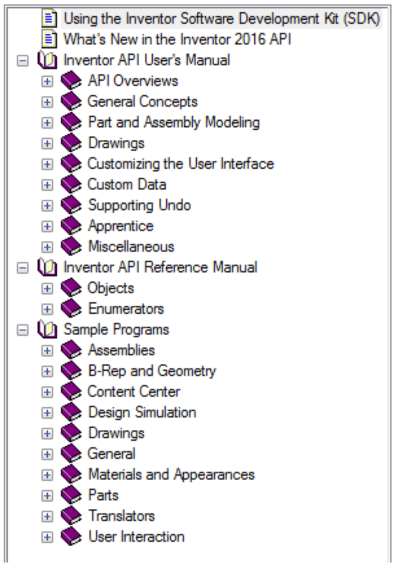
Introduction to Using Inventor's Programming Interface

There are several resources provided to help you use Inventor's Application Programming Interface (API). These resources are all part of Inventor's Software Development Kit (SDK). The various elements of the SDK and some additional external resources are described below.

API Help

The API Help is installed with Inventor and is accessed from the Help menu as shown below.

The help content consists of several parts:



The programming help consists of several sections. The "Using the Inventor Software Development Kit (SDK)" topic discusses additional resources including some additional samples and tools that are delivered with Inventor.

The "What's New in Inventor" section lists the enhancements made in that release.

The "Inventor API User's Manual" session is a series of papers that provide a discussion of various API topics.

The "Inventor API Reference Manual" section is the heart of the documentation and includes a description of every object and their methods, properties, and events. These also link to sample programs that demonstrate those particular objects or functions.

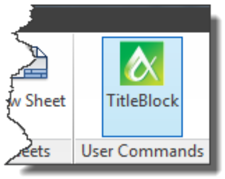
The "Sample Programs" section is a categorized list of small samples that are also linked to from the object and function topics in the reference manual.

Sharing and Running Your Program

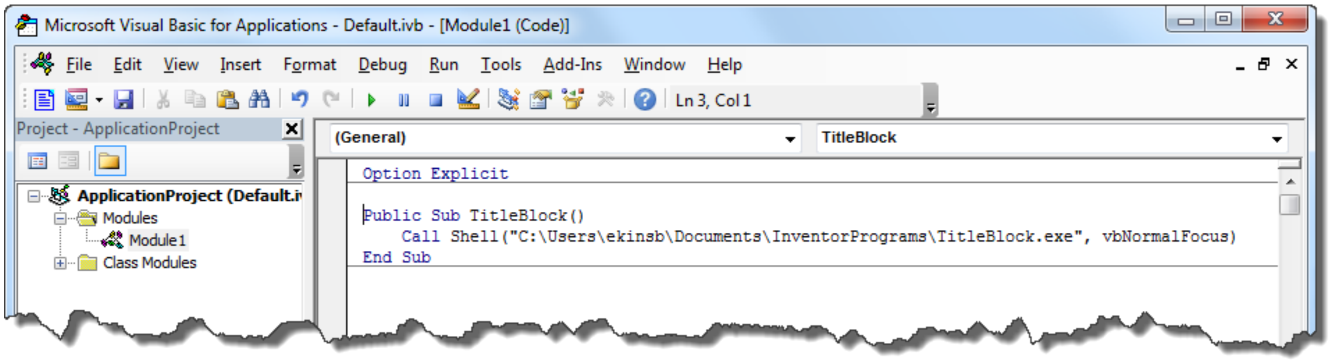
One more thing that I mentioned I earlier I would talk about is how to share and run your program. What you've created for both of the programs is a standard Windows executable. If you browse in your project's folder you'll see a "bin" directory. Inside of that is a "Debug" directory and inside of that will be your exe file, along with several other files. For these simple programs, the other files are just used for debugging purposes and can be deleted.

To run your program outside of Visual Studio you just need to run the executable file, typically by double-clicking the file from Windows Explorer. The program can be located anywhere. To share your program with someone else you just need to give them the executable and then they can run it on their system.

Running your programs from outside Inventor is not ideal but is the easiest way to get started writing programs with Visual Basic that work with Inventor. There are a couple of possibilities to provide easier access to your program. The first is to create a small VBA macro that executes it which you can add a button in the ribbon to run the VBA macro and if you want you can even create a custom icon for it. Below is an example of what my button to run the macro that will execute the title block program.

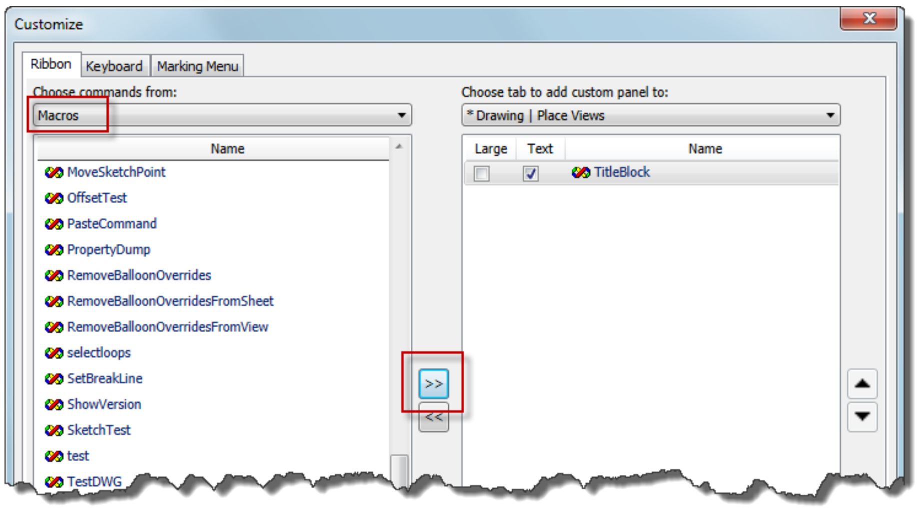


To do this, create a VBA macro that will execute your program. Open the VBA editor in Inventor and in the Module1 module of the ApplicationProject add the code shown below. If the code window isn't shown, double-click on "Module1". The path to the exe will be to wherever it is on your system.


Microsoft Visual Basic for Applications - Default.ivb - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Ln 3, Col 1
Project - ApplicationProject
ApplicationProject (Default.i
Modules
Module1
Class Modules
(General)
TitleBlock

```
Option Explicit

Public Sub TitleBlock()
    Call Shell("C:\Users\ekinsb\Documents\InventorPrograms\TitleBlock.exe", vbNormalFocus)
End Sub
```

In Inventor, have the ribbon where you want your button currently displayed, right-click on the ribbon, and choose the "Customize User Commands…". This will display the Customize command shown below. Choose "Macros" from the drop-down at the top of the command list. This will display all of your macros. Pick your macro and then click the ">>" button in the middle of the dialog. This will move that macro over to the right-hand side of the dialog. You can then choose if text should be displayed and if a large or small icon should be used. A default icon will be used. Now you have one-click easy access to your program.

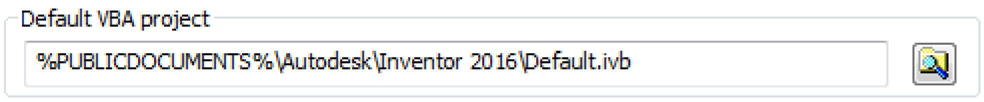


If you want a custom icon you need to create a 16x16 image for a small icon or a 32x32 image for a large icon. This currently must be a bmp type of image. The name of the image is important. The images for the TitleBlock macro are:

Module1.TitleBlock.Small.bmp

Module1.TitleBlock.Large.bmp

Where the name is "*ModuleName*.*MacroName*.*IconSize*.bmp" where IconSize can be "Small" or "Large". The files need to be in the same directory as the VBA project file. You can find out that location on your system by looking at the "File" tab of the "Application Options" dialog, as shown below. On my computer this is "C:\Users\Public\Documents\Autodesk\Inventor 2016".



%PUBLICDOCUMENTS%\Autodesk\Inventor 2016\Default.ivb

The other option to making your program easily accessible is to convert it to an add-in. This makes it very easy to share your program. You can deliver a set of files, or even an installer, and the receiver either copies the files to a specified location or just runs the installer. The next time they start Inventor they'll see the new commands. If you get serious about writing programs you'll want to start writing add-ins. They use the same API but just support this additional way of being loaded by Inventor.