Autodesk University 2008

# Inventor® API:
# Exploring iProperties and Parameters

Brian Ekins – Autodesk

**DE101-1** This session provides an overview of how the Inventor Application Programming Interface (API) works. It begins with a brief introduction of some basic API concepts and then puts them into practice using iProperties and Parameters. iProperties and Parameters are two of the most used areas of Inventor and are also commonly accessed through the programming interface. The use of the API will be demonstrated and example programs that use these concepts will be shown.

**About the Speaker:**
Brian is a designer for the Autodesk Inventor® programming interface. He began working in the CAD industry over 25 years ago in various positions, including CAD administrator, applications engineer, CAD API designer, and consultant. Brian was the original designer of the Inventor® API and has presented at conferences and taught classes throughout the world to thousands of users and programmers.

brian.ekins@autodesk.com

# API Terminology and Basic Concepts

Before looking at iProperties and Parameters here's a brief review of the various terms and concepts used by Inventor's programming interface.

### API

API stands for *Application Programming Interface*. Any application that supports driving it with a program has some type of API. Inventor supports a programming interface that uses Microsoft technology called *COM Automation*. This is the same technology that Word and Excel use for their API's. If you've ever programmed them then many of the concepts you learned will transfer over when programming Inventor. If you haven't programmed them, the things you learn when programming Inventor will give you a head-start if you do need to program them in the future.

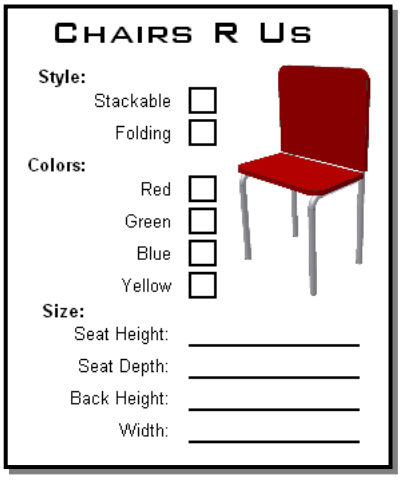
### SDK

SDK stands for *Software Development Kit*. Many applications provide a software development kit with the application to help support developers that want to write programs for the application. A software development kit typically consists of documentation, tools, and samples that can help you understand and use the application's API. The documentation for Inventor's API is installed when you install Inventor and is accessible through the **Help►Additional Resources►Programming Help** command. Inventor also provides an SDK that is copied onto your disk when Inventor is installed. You'll need to take the extra step and install the SDK to have access to the additional tools and samples. You do this by running Inventor 2009\SDK\DeveloperTools.msi

### Object Oriented Programming

Object Oriented Programming is a programming methodology used by Inventor to provide its programming interface. It provides a logical way of organizing an API and makes it easier to use. Here are the object oriented terms you'll need to be familiar when using Inventor's API

**Object** – A programming object usually represents a logical entity but can also serve to group related functionality. Some examples of objects in Inventor's API are the Parameter, Property, ExtrudeFeature, and MateConstraint objects. Some Inventor API objects also represent more abstract concepts like Vector, SketchOptions, and UnitsOfMeasure. The functionality that a particular object supports is exposed through its *Methods*, *Properties*, and *Events.* Here's a non-programming example to illustrate these concepts.




**Properties** - A company that sells chairs might allow a customer to design a chair by filling out an order form like the one shown to the right. The options on the order form define the various properties of the desired chair. By setting the properties to the desired values the customer is able to describe the specific chair they want. An API object has properties that let you get and set information associated with it. For example, an ExtrudeFeature object supports the Name and Suppression properties.

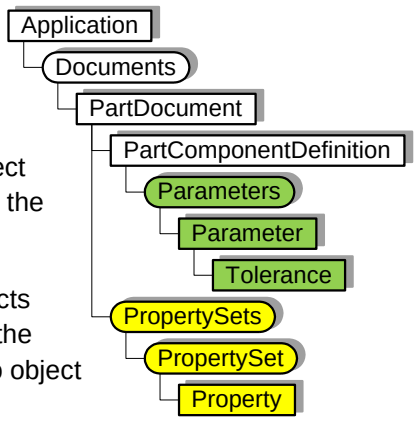
**Methods** - Methods are basically instructions that the object understands. In the real world, these are actions you would perform with the chair. For example, you could move the chair, cause it to fold up, or throw it in the trash. In the programming world the objects are smart and instead of you performing the action you tell the object to perform the action itself; move, fold, and delete.

**Events** – For the chair example, events would be like adding sensors to the chair so you would be notified anytime anyone sat on the chair, or if it was moved. In Inventor events notify you when certain things happen in Inventor.

All objects have a defined set of methods, properties, and events. These represent the settings, actions, and notifications that are associated with that object.

### Object Model

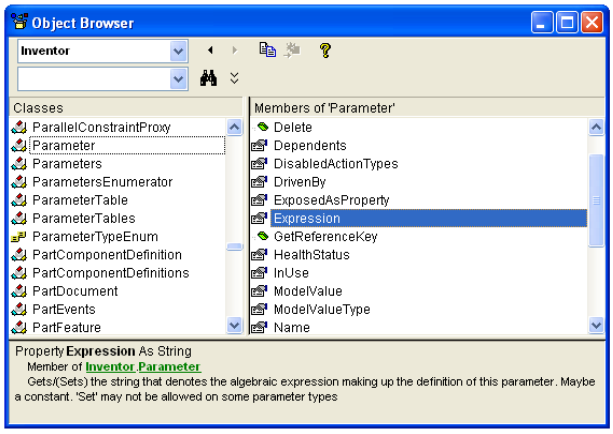
With the type of API that Inventor uses the objects are exposed through something known as the *Object Model* or *Object Hierarchy*. The object model provides Inventor objects in a structured, organized way. The object model typically defines the parent-child relationships between objects. To access a specific object you need to first access some object within the object model. From that object you can them move through the hierarchy to get to the specific object you want.




The picture to the right shows the object model for the commonly used objects associated with parameters and iProperties. You'll typically gain access to the object model through the Application object and then traverse from object to object to get to the specific object you want. You'll see examples of this later.

### Object Browser

The Object Browser is a tool within VBA that allows you to see all of the objects, methods, properties, and events for Inventor's API. For the methods, properties, and events, it also shows you the arguments and for properties indicates if the property is read-write or read-only. This is a very useful tool when using Inventor's API. You can access the Object Browser from Inventor's VBA environment by pressing the F2 key. You open the VBA environment by pressing Alt+F11.




The Object Browser is also a convenient entry to Inventor's programming help since it is context sensitive. You can pick any object, method, property, or event in the browser and press F1 and it will open the help window for the selected item.

### Collection Objects

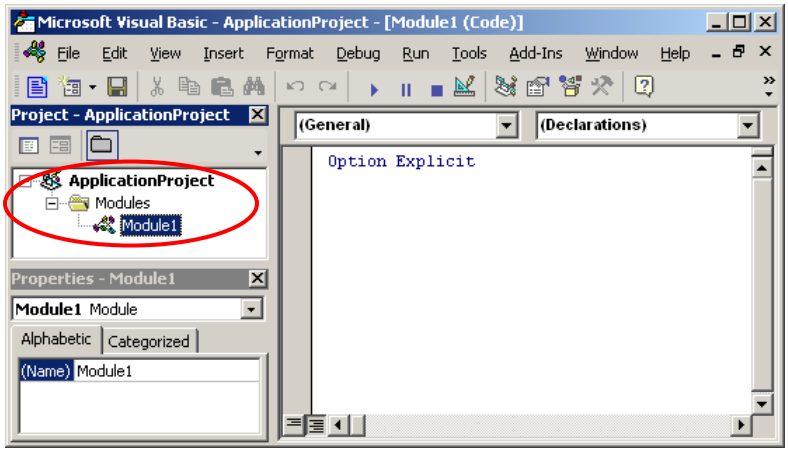
A Collection is a special type of object that provides access to a group of other objects. To support this capability all collections have the following two properties; Count and Item. The Count property tells you how many objects are in the collection. The Item property returns an item from the collection. You can always specify which item to return by using its index within the collection and some collections also let you specify the item by name. An example of a collection object is the Parameters object. Through this object you have access to all of the parameters within a document. You'll see examples of the use of collection objects later in this paper.

## What Development Environment to Use?

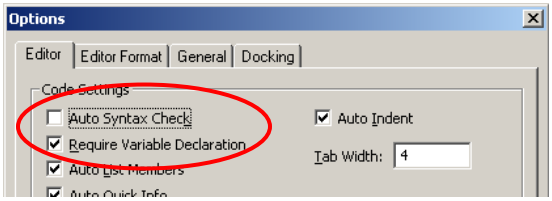
To write a program you need to use some type of programming language. There are many choices out there and they all have various advantages and disadvantages. If you're new to programming or

Inventor's API, I would recommend using Inventor's VBA (Visual Basic for Applications). VBA comes free with Inventor and is easier to use than any of the other languages for programming Inventor. Here's just enough of an introduction to VBA that you can try out the programs shown later.

To open the VBA interface you must first have Inventor running and then you can either press Alt+F11 or use the **Tools►Macro►Visual Basic Editor** command. There's a default VBA project created that you will typically write your programs in that's called "ApplicationProject". The project window shows the available projects and their contents. Expand the tree so you can see Module1 and double-click on it, as shown below. That will open the code window so you can begin programming.

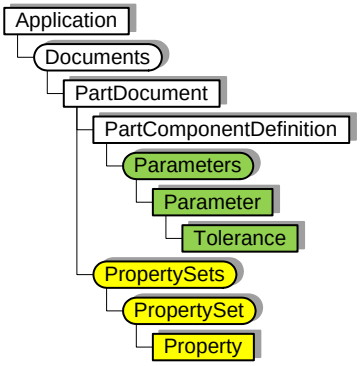



To make things a little smoother I would also recommend changing a couple of the VBA settings. In VBA run the **Tools►Options** command and change the **Auto Syntax Check** and **Require Variable Declaration** settings as shown below.




## Accessing Documents in the API

To do anything with Inventor's API you need to gain access to the object that supports the functionality you want. If you look at the object model diagram to the right you'll notice that properties and parameters are both under the PartDocument object. To access parameters or properties you need to first gain access to the document that contains them.




There are actually several different types of document objects with PartDocument, AssemblyDocument, and DrawingDocument being the most common types. There is another type called the *Document* object that is a special kind of object that can represent any of the other document types. Using the Document object is very convenient in many programs because it allows a single program to handle any type of document without requiring you to special case for each specific type of document.

Before looking specifically at properties and parameters, let's look at some of the most common ways of accessing documents using Inventor's programming interface.

### Get the Document the end-user is editing

The most common technique for getting a document is to get the document the end-user is currently editing. This is also the easiest since the Application object provides direct access to this document through its ActiveDocument property.

```
Public Sub GetActiveDocumentSample()
    ' Get the active document.
    Dim oDoc As Document
    Set oDoc = ThisApplication.ActiveDocument

    MsgBox "Got " & oDoc.FullFileName
End Sub
```

### Opening an existing Document

You can use the API to open an existing document. The Document object that represents the document just opened is returned.

```
Public Sub OpenDocumentSample()
    ' Open an existing document from disk.
    Dim oDoc As Document
    Set oDoc = ThisApplication.Documents.Open("C:\Temp\Part.ipt", True)

    ' Call a property of the Document to show its name.
    MsgBox "Opened " & oDoc.FullFilename
End Sub
```

### Create a new document

New documents are created using the Add method of the Documents collection. You specify the type of document and the template to use. You can use the GetTemplateFile method to get the default template filename. This is demonstrated below.

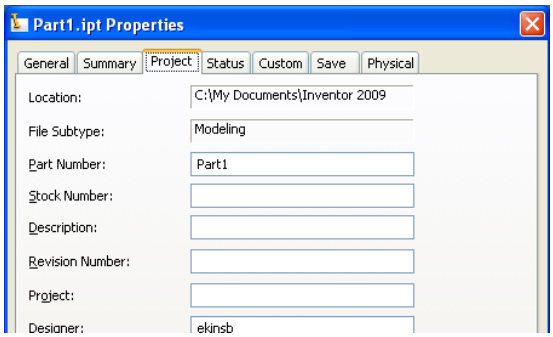
```
Public Sub CreateDocumentSample()
    ' Create a new part document using the default template.
    Dim oDoc As Document
    Set oDoc = ThisApplication.Documents.Add(kPartDocumentObject, _
            ThisApplication.FileManager.GetTemplateFile(kPartDocumentObject))

    MsgBox "Created " & oDoc.DisplayName
End Sub
```

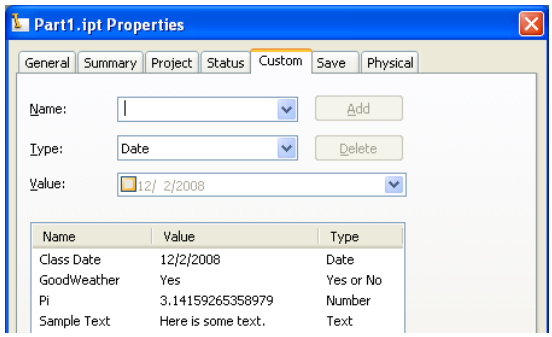
## iProperties Through the API

### Through the User-Interface

Before looking at the programming interface for iProperties let's do a quick review of iProperties from an end-user perspective. iProperties are accessed through the iProperties dialog. There are several tabs at the top that organize the available properties. There are also some tabs on this dialog that are not actually iProperties. For example, the General tab provides access to information about the document, the Save tab provides options that control how the thumbnail image is captured, and the Physical tab provides the various physical properties of the part or assembly document. If you need access to this information it is available through the API but not through the iProperties portion of the API that we're discussing here.

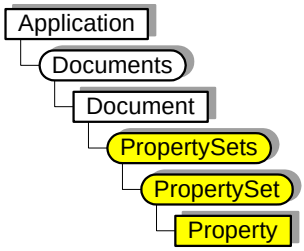
iProperties are accessed through the Properties dialog as shown below. The various properties are organized by tabs. The picture below on the left illustrates a typical tab of the Properties dialog where you have access to a specific set of properties. The set of iProperties shown on each tab are predefined and cannot be changed. However, using the Custom tab (shown in the picture on the right) you can add additional properties to the document. This allows you to associate any information you want with the document.




## Through the API

The object model for iProperties is shown to the right. As discussed earlier, properties are owned by documents and to get access to properties you go through the document that owns them.




### Property Naming

Even though the programming interface for iProperties is simple, people still tend to struggle with it. I believe this is primarily because of not knowing how to access a specific property. Before discussing iProperties in detail, a brief discussion of naming is appropriate to help describe the concepts Inventor uses. The picture to the right shows a person and three different ways to identify this person. His full legal name is a good way to identify him, although a bit formal. His social security number is good, but not very user friendly. His nickname, although commonly used, can have problems since it's not as likely to be unique and Bill could change it. The point is that there are three ways to identify this person, each one with its own pros and cons.

Legal Name:
William Harry Potter

Nickname: Bill

SSN: 365-58-9401

Like Bill, iProperties also have several names. Understanding this single concept should help you overcome most of the problems other people have had when working with iProperties. The various iProperty objects, their names and best use suggestions are described below.

### PropertySets Objects

The PropertySets object serves as the access point to iProperties. The PropertySets object itself doesn't have a name but is simply a collection object that provides access to the various PropertySet objects. Using the Item method of the PropertySets object you specify which PropertySet object you want. The Item method accepts an Integer value indicating the index of the PropertySet object you want, but more important, it also accepts the name of the PropertySet object you want. The next section discusses PropertySet objects and their various names.

**PropertySet Objects**

The PropertySet object is a collection object and provides access to a set of iProperties. The PropertySet object is roughly equivalent to a tab on the Properties dialog. The Summary, Project, Status, and Custom tabs of the dialog contain the iProperties that are exposed through the programming interface. There are four PropertySet objects in an Inventor document; Summary Information, Document Summary Information, Design Tracking Properties, and User Defined Properties.

PropertySet objects are named as a way to identify a particular PropertySet object. A PropertySet object has three names; Name, Internal Name, and Display Name. Using the previous analogy of Bill, the Name is equivalent to his legal name, the Internal Name is equivalent to his social security number, and the Display Name is equivalent to his nickname. Let's look at a specific example to illustrate this. There is a PropertySet object that has the following names:

Name: Inventor Summary Information
Internal Name: {F29F85E0-4FF9-1068-AB91-08002B27B3D9}
DisplayName: Summary Information

Any one of these can be used as input to the Item method in order to get this particular PropertySet object. I would suggest always using the Name for the following reasons. The Name cannot be changed, is guaranteed to be consistent over time, and is an English human readable string. The Internal Name cannot be changed and will remain consistent but it is not very user-friendly and makes your source code more difficult to read. The Display Name is not guaranteed to remain constant. The Display Name is the localized version of the name and will change for each language. A chart showing the names of the four standard PropertySet objects is on page 13 of this paper.

Below is an example of obtaining one of the PropertySet objects. In this case the summary information set of iProperties.

```
Public Sub GetPropertySetSample()
    ' Get the active document.
    Dim invDoc As Document
    Set invDoc = ThisApplication.ActiveDocument

    ' Get the summary information property set.
    Dim invSummaryInfo As PropertySet
    Set invSummaryInfo = invDoc.PropertySets.Item("Inventor Summary Information")
End Sub
```

**Property Objects**

A Property object represents an individual property. Each Property object also has three names; Name, Display Name, and ID. Many of the same principles discussed for PropertySet objects applies here. The Name is an English string that is guaranteed to remain constant. The Display Name is the localized version of the Name and can change, so it's not a reliable method of accessing a particular property. The ID is a number and is similar to the Internal Name of the PropertySet object, but is a simple Integer number instead of a GUID. I would recommend using the name of the property to access a specific one. Below is an example of the three identifiers for a particular property.

Name: Part Number
DisplayName: Part Number
ID: 5 or kPartNumberDesignTrackingProperties

The following code gets the iProperty that represents the part number.

```
Public Sub GetPropertySample()
    ' Get the active document.
    Dim invDoc As Document
    Set invDoc = ThisApplication.ActiveDocument

    ' Get the design tracking property set.
    Dim invDesignInfo As PropertySet
    Set invDesignInfo = invDoc.PropertySets.Item("Design Tracking Properties")

    ' Get the part number property.
    Dim invPartNumberProperty As Property
    Set invPartNumberProperty = invDesignInfo.Item("Part Number")
End Sub
```

You may see program samples that use identifiers like kPartNumberDesignTrackingProperties to specify a specific property. These identifiers are defined in the Inventor type library and provide a convenient way of specifying the ID of a property. For the part number, instead of specifying the ID as 5 you can use kPartNumberDesignTrackingProperties. This makes your code more readable. If you want to use the ID instead of the Name you need to use the ItemByPropId property instead of the standard Item property of the PropertySets object. For consistency I would recommend using the Name in both cases.

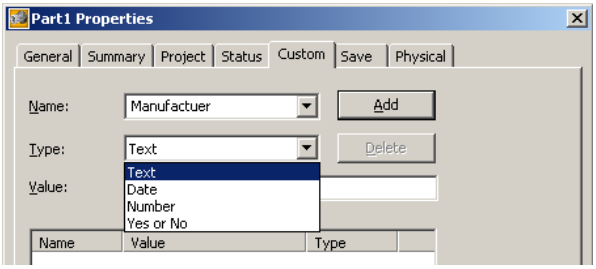
Now that we have a reference to a specific property we can use its programming properties to get and set the value. The Property object supports a property called Value that provides this capability. For example, the line below can be added to the previous sample to display the current part number.

```
MsgBox "The part number is: " & invPartNumberProperty.Value
```

This next line sets the value of the part number iProperty.

```
invPartNumberProperty.Value = "Part-001"
```

The Value property uses an interesting programming feature that is useful to understand when working with iProperties. The Value property is of type Variant. A Variant is a special type that can represent almost any other type. For example, a Variant can hold a String, Double, Date, Object, or most any other type. iProperties take advantage of this since they allow the value to be one of many different types. You see this when working with custom (user defined) properties, as shown in the picture below. When creating or modifying a custom iProperty you define the type; Text, Date, Number, or Yes or No. This results in an iProperty being created where the value contains a String, Date, Double, or Boolean value.




When setting the value of any of Inventor's predefined iProperties, Inventor forces them to be the correct type and will convert them automatically, when possible. If you supply a value that can't be converted to the expected type, it will fail. In the table on page 13 of this paper, the type of each property is listed. Most of the standard iProperties are Strings, with a few Date, Currency, Boolean, and Long values.

There is also one other type called IPictureDisp. This type is used for the thumbnail picture associated with a document. Using this you can extract and set the thumbnail picture for a document.

**Creating Properties**

When you create custom iProperties using the Properties dialog, as shown in the previous picture, you specify the type of property you're going to create; Text, Date, Number, or Yes or No. When you create them using Inventor's programming interface you don't explicitly specify the type but it is implicitly determined based on the type of variable you input.

New iPropertiies can only be created within the Custom (user defined) set of properties. New iProperties are created using the Add method of the PropertySet object. The sample below illustrates creating four new iProperties, one of each type.

```
Public Sub CreateCustomProperties()
    ' Get the active document.
    Dim invDoc As Document
    Set invDoc = ThisApplication.ActiveDocument

    ' Get the user defined (custom) property set.
    Dim invCustomPropertySet As PropertySet
    Set invCustomPropertySet = invDoc.PropertySets.Item( _
                           "Inventor User Defined Properties")

    ' Declare some variables that will contain the various values.
    Dim strText As String
    Dim dblValue As Double
    Dim dtDate As Date
    Dim blYesOrNo As Boolean

    ' Set values for the variables.
    strText = "Some sample text."
    dblValue = 3.14159
    dtDate = Now
    blYesOrNo = True

    ' Create the properties.
    Dim invProperty As Property
    Set invProperty = invCustomPropertySet.Add(strText, "Test Test")
    Set invProperty = invCustomPropertySet.Add(dblValue, "Test Value")
    Set invProperty = invCustomPropertySet.Add(dtDate, "Test Date")
    Set invProperty = invCustomPropertySet.Add(blYesOrNo, "Test Yes or No")
End Sub
```

A common task is when you have a value you want to save as a custom property within a document. If the property already exists you just want to update the value. If the property doesn't exist you want to create it with the correct value. The code below demonstrates getting the volume of a part and writing it to a custom property named "Volume". With the volume as an iProperty it can be used as input for text on a drawing. The portion of this macro that gets the volume and formats the result is outside the scope of this paper but helps to demonstrate a practical use of creating and setting the value of an iProperty.

```
Public Sub UpdateVolume()
    ' Get the active part document.
    Dim invPartDoc As PartDocument
    Set invPartDoc = ThisApplication.ActiveDocument

    ' Get the volume of the part.  This will be returned in
    ' cubic centimeters.
    Dim dVolume As Double
    dVolume = invPartDoc.ComponentDefinition.MassProperties.Volume

    ' Get the UnitsOfMeasure object which is used to do unit conversions.
    Dim oUOM As UnitsOfMeasure
    Set oUOM = invPartDoc.UnitsOfMeasure

    ' Convert the volume to the current document units.
    Dim strVolume As String
    strVolume = oUOM.GetStringFromValue(dVolume, _
                    oUOM.GetStringFromType(oUOM.LengthUnits) & "^3")

    ' Get the custom property set.
    Dim invCustomPropertySet As PropertySet
    Set invCustomPropertySet = _
              invPartDoc.PropertySets.Item("Inventor User Defined Properties")

    ' Attempt to get an existing custom property named "Volume".
    On Error Resume Next
    Dim invVolumeProperty As Property
    Set invVolumeProperty = invCustomPropertySet.Item("Volume")
    If Err.Number <> 0 Then
        ' Failed to get the property, which means it doesn't exist
        ' so we'll create it.
        Call invCustomPropertySet.Add(strVolume, "Volume")
    Else
        ' We got the property so update the value.
        invVolumeProperty.Value = strVolume
    End If
End Sub
```

**Apprentice and Properties**

Apprentice is a programming component separate from Inventor. It is delivered with Inventor and also as part of Inventor View (which is freely available at www.autodesk.com). Apprentice provides a small subset of Inventor's capabilities as a component that runs within your application. Apprentice doesn't have a user-interface but only a programming interface. For the information it provides access to it can be significantly faster than the equivalent program in Inventor. Apprentice provides full access to properties. The program below illustrates the use of Apprentice. Apprentice cannot be used from within Inventor's VBA or an Add-In but must be used from an exe program or from within another application's VBA environment (i.e. Excel).

One interesting difference between using Inventor and Apprentice is saving any changes you make. To save any changes you make in Inventor you save the document. The Save method of the Document object does this. However, when working with Apprentice you don't have to save the entire document since it's possible to only save the iProperty changes to the document, which is much faster than saving the entire document. The FlushToFile method of the PropertySets object saves any iProperty changes. The sample below demonstrates this by opening a document using Apprentice, changing a property value, saving the change, and closing everything.

```
Public Sub ApprenticeUpdate()
    ' Declare a variable for Apprentice.
    Dim invApprentice As New ApprenticeServerComponent

    ' Open a document using Apprentice.
    Dim invDoc As ApprenticeServerDocument
    Set invDoc = invApprentice.Open("C:\Temp\Part1.ipt")

    ' Get the design tracking property set.
    Dim invDTProperties As PropertySet
    Set invDTProperties = invDoc.PropertySets.Item("Design Tracking Properties")

    ' Edit the values of a couple of properties.
    invDTProperties.Item("Checked By").Value = "Bob"
    invDTProperties.Item("Date Checked").Value = Now

    ' Save the changes.
    invDoc.PropertySets.FlushToFile

    ' Close the document and release all references.
    Set invDoc = Nothing
    invApprentice.Close
    Set invApprentice = Nothing
End Sub
```

## Example Property Programs

There are some associated sample programs that provide more complete examples of the various topics covered in this paper and provide practical examples in various programming languages of how to use Inventor's programming interface.

PropertySamples.ivb

CopyProperties – Copies a set of properties from a selected document into the active document.

DumpPropertyInfo – Displays information about all of the property sets and their properties.

Properties.xls

An Excel file where each sheet contains a list of properties and values. There are two Excel macros that are executed using the buttons on the first sheet.

Set Properties of Active Document – Copies the properties of the selected Excel sheet into the active document.

Set Properties of Documents – Copies the properties of the selected Excel sheet into all of the Inventor documents in the selected directory. This sample uses Apprentice.

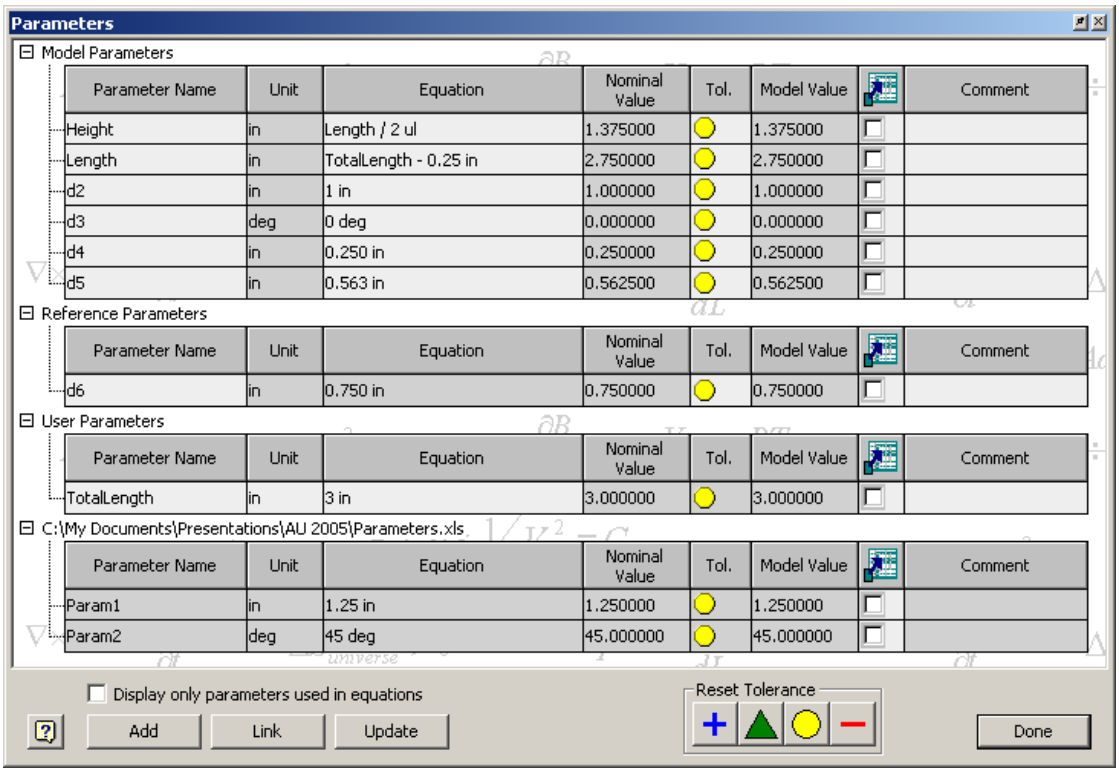
| **Inventor User Defined Properties,** {D5CDD505-2E9C-101B-9397-08002B2CF9AE} | | | |
|---|---|---|---|
| | | | |
| **Inventor Summary Information,** {F29F85E0-4FF9-1068-AB91-08002B27B3D9} | | | |
| **Property Name** | **Id** | **Id Enum** | **Type** |
| Title | 2 | kTitleSummaryInformation | String |
| Subject | 3 | kSubjectSummaryInformation | String |
| Author | 4 | kAuthorSummaryInformation | String |
| Keywords | 5 | kKeywordsSummaryInformation | String |
| Comments | 6 | kCommentsSummaryInformation | String |
| Last Saved By | 8 | kLastSavedBySummaryInformation | String |
| Revision Number | 9 | kRevisionSummaryInformation | String |
| Thumbnail | 17 | kThumbnailSummaryInformation | IPictureDisp |
| | | | |
| **Inventor Document Summary Information,** {D5CDD502-2E9C-101B-9397-08002B2CF9AE} | | | |
| **Property Name** | **Id** | **Id Enum** | **Type** |
| Category | 2 | kCategoryDocSummaryInformation | String |
| Manager | 14 | kManagerDocSummaryInformation | String |
| Company | 15 | kCompanyDocSummaryInformation | String |
| | | | |
| **Design Tracking Properties,** {32853F0F-3444-11D1-9E93-0060B03C1CA6} | | | |
| **Property Name** | **Id** | **Id Enum** | **Type** |
| Creation Time | 4 | kCreationDateDesignTrackingProperties | Date |
| Part Number | 5 | kPartNumberDesignTrackingProperties | String |
| Project | 7 | kProjectDesignTrackingProperties | String |
| Cost Center | 9 | kCostCenterDesignTrackingProperties | String |
| Checked By | 10 | kCheckedByDesignTrackingProperties | String |
| Date Checked | 11 | kDateCheckedDesignTrackingProperties | Date |
| Engr Approved By | 12 | kEngrApprovedByDesignTrackingProperties | String |
| Engr Date Approved | 13 | kDateEngrApprovedDesignTrackingProperties | Date |
| User Status | 17 | kUserStatusDesignTrackingProperties | String |
| Material | 20 | kMaterialDesignTrackingProperties | String |
| Part Property Revision Id | 21 | kPartPropRevIdDesignTrackingProperties | String |
| Catalog Web Link | 23 | kCatalogWebLinkDesignTrackingProperties | String |
| Part Icon | 28 | kPartIconDesignTrackingProperties | IPictureDisp |
| Description | 29 | kDescriptionDesignTrackingProperties | String |
| Vendor | 30 | kVendorDesignTrackingProperties | String |
| Document SubType | 31 | kDocSubTypeDesignTrackingProperties | String |
| Document SubType Name | 32 | kDocSubTypeNameDesignTrackingProperties | String |
| Proxy Refresh Date | 33 | kProxyRefreshDateDesignTrackingProperties | Date |
| Mfg Approved By | 34 | kMfgApprovedByDesignTrackingProperties | String |
| Mfg Date Approved | 35 | kDateMfgApprovedDesignTrackingProperties | Date |
| Cost | 36 | kCostDesignTrackingProperties | Currency |
| Standard | 37 | kStandardDesignTrackingProperties | String |
| Design Status | 40 | kDesignStatusDesignTrackingProperties | Long |
| Designer | 41 | kDesignerDesignTrackingProperties | String |
| Engineer | 42 | kEngineerDesignTrackingProperties | String |
| Authority | 43 | kAuthorityDesignTrackingProperties | String |
| Parameterized Template | 44 | kParameterizedTemplateDesignTrackingProperties | Boolean |
| Template Row | 45 | kTemplateRowDesignTrackingProperties | String |
| External Property Revision Id | 46 | kExternalPropRevIdDesignTrackingProperties | String |
| Standard Revision | 47 | kStandardRevisionDesignTrackingProperties | String |
| Manufacturer | 48 | kManufacturerDesignTrackingProperties | String |
| Standards Organization | 49 | kStandardsOrganizationDesignTrackingProperties | String |
| Language | 50 | kLanguageDesignTrackingProperties | String |
| Defer Updates | 51 | kDrawingDeferUpdateDesignTrackingProperties | Boolean |
| Size Designation | 52 | | String |
| Categories | 56 | kCategoriesDesignTrackingProperties | String |
| Stock Number | 55 | kStockNumberDesignTrackingProperties | String |
| Weld Material | 57 | kWeldMaterialDesignTrackingProperties | String |

## Parameters through the API

Before looking at the programming interface that provides access to the parameters lets first go through a brief overview of how parameters work from the perspective of an end-user, and then look at them form the perspective of an API user.

### Through the User-Interface

Each part or assembly document has its own set of parameters. You access these parameters using the **Parameters** command  , which displays the Parameters dialog as shown below. Let's look at the general functionality supported by all parameters.




**Parameter Name** – All parameters have a unique name. These names are editable by the end-user (except for table parameters where the name is controlled by the spreadsheet).

**Unit** – All parameters have a specified unit type. The unit type is editable for user parameters. For table parameters it is controlled by the spreadsheet. For all other types, the unit type is based on the current default units defined for the document.

**Equation** – The equation defines the value of the parameter. The equation is editable by the end-user for all parameter types except for reference and table parameters. The equation specified for a parameter must result in units that match the unit specified for that parameter. For example, if the parameters a, b, and c exist and are defined to be inch units, the equation "a * b" is invalid for c because the resulting units of multiplying two length units together (inch) will be an area (square inches). The equation "a + b" is valid because the resulting unit is still a length (inch).

Equations can also call functions. For example “d0 * sin(d5)” is a valid equation, assuming the parameters d0 an d5 exist and are the correct unit type. To find a description of all of the supported functions search for “functions in edit boxes” in Inventor’s online help. All of the functions listed can be used within parameter equations.

**Nominal Value** – The nominal value displays the resulting value of the equation.

**Tolerance** – The tolerance setting in the Parameters dialog specifies which value to use as the model value within the part or assembly. You can choose to use the nominal (which is the default), the upper, lower, or median value.

**Model Value** – The model value shows the resulting value after the tolerance setting has been applied to the nominal value. This value will be used when recomputing the part or assembly.

**Export Parameter** – The export parameter check box defines whether this parameter is exported as an iProperty or not. If checked, this parameter is exported as a custom iProperty and is also selectable when deriving a part.

**Comment** – The comment field allows the end-user to add a description to a parameter. The comment is editable for all parameter types except for table parameters which are controlled by the spreadsheet.

## Parameter Types

The parameters are grouped by type within the dialog. Let’s look at each of these parameter types and their functionality.

**Model Parameters** – Model parameters are created automatically whenever you create something within Inventor that is dependent on a parameter, i.e. sketch dimensions, features, assembly constraints, iMates, etc. You cannot edit the unit type or delete model parameters.

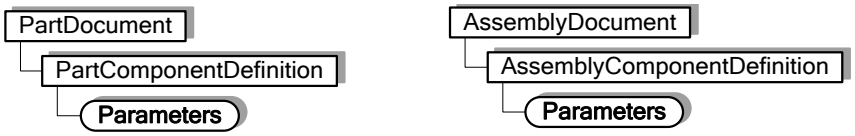
**Reference Parameters** – Reference Parameters are created automatically whenever a sketch dimension is created and is designated as a driven dimension. By default, dimension constraints are driving dimensions. That is, they drive or control the geometry. By toggling a dimension constraint to be driven, the geometry controls it. A reference dimension constraint can be created by placing a dimension constraint that will over-constrain the sketch. A warning dialog pops up, notifying you of the problem but allows you to proceed with the placement of the dimension as a driven dimension. You can also toggle any existing dimension constraint from driving to driven using the Driven Dimension command,  . You cannot change the unit type or the equation of a reference parameter and you cannot delete a reference parameter.

**User P**arameters - User parameters are created by the end-user using the Add button on the Parameters dialog. You have complete control over user parameters and can edit the unit and equation and can also delete them.

**Table Parameters** – Table parameters are created automatically whenever an Excel spreadsheet is linked to the parameters using the Link button on the Parameters dialog. Each linked spreadsheet is listed in the parameters dialog with the parameters defined in that spreadsheet nested within that sheet in the Parameters dialog. You can’t edit any of the information associated with a table parameter except to specify that you want to export it. Everything else is defined the by the spreadsheet.

## Parameters Programming Interface

Now let's look at the programming interface for parameters. The parameter functionality is exposed through the Parameters object. This object is obtained using the Parameters property of either the PartComponentDefinition or AssemblyComponentDefinition object as shown below.




The VBA code shown below obtains the Parameters object. It will work for either a part or assembly document.

```
Dim oParameters As Parameters
Set oParameters = ThisApplication.ActiveDocument.ComponentDefinition.Parameters
```

Below is an example that displays the number of parameters in the active document. It also incorporates error handling when attempting to get the Parameters object. The other samples have omitted error handling to make them easier to read.

```
Public Sub GetParameters()
    ' Get the Parameters object.  This uses error handling and will only be successful
    ' if a part or assembly document is active.
    Dim oParameters As Parameters
    On Error Resume Next
    Set oParameters = ThisApplication.ActiveDocument.ComponentDefinition.Parameters
    If Err Then
        ' Unable to get the Parameters object, so exit.
        MsgBox "Unable to access the parameters.  A part or assembly must be active."
        Exit Sub
    End If
    On Error Goto 0

    ' Do something with the Parameters object.
    MsgBox "The document has " & oParameters.Count & " parameters in it."
End Sub
```

Let's look at a simple program that uses the Parameter object to change the value of a parameter named "Length". Although it is simple, this program demonstrates the parameter functionality that is most commonly used and is frequently the only parameter functionality needed for many programs.

```
Public Sub SetParameter()
    ' Get the Parameters object.  Assumes a part or assembly document is active.
    Dim oParameters As Parameters
    Set oParameters = ThisApplication.ActiveDocument.ComponentDefinition.Parameters

    ' Get the parameter named "Length".
    Dim oLengthParam As Parameter
    Set oLengthParam = oParameters.Item("Length")

    ' Change the equation of the parameter.
    oLengthParam.Expression = "3.5 in"

    ' Update the document.
    ThisApplication.ActiveDocument.Update
End Sub
```

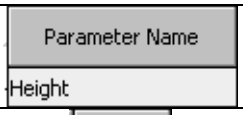
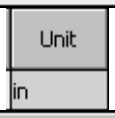
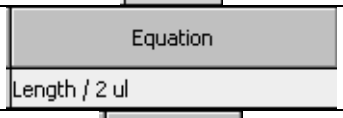
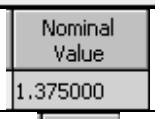
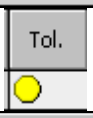
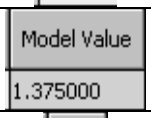
This program starts by getting the Parameters object, just as the previous sample demonstrated. The Parameters object is a typical collection object. It supports the Count property which returns the number

of items in the collection and it supports the Item property which lets you access the items within the collection. The Item property will accept a Long value indicating the index of the Parameter within the collection you want and it will also accept a String, which specifies the name of the Parameter you want. When specifying the name it is case sensitive. If the name you specify does not exist the call will fail. In this example, as long as there is a parameter named "Length", the call of the Item property will return the parameter and assign it to the variable oLengthParam.

In the next line the Expression property of the Parameter object is used to set the expression. The Expression property provides read and write access to the equation of the parameter. The terms "Expression" and "Equation" are synonymous when working with parameters. The expression is set using a String that defines a valid equation. The same rules apply here as when setting a parameter interactively in the Parameters dialog.

Just as when you edit parameters interactively, you need to tell the model to update to see the results. Interactively, you run the Update command  . In the API, the equivalent is the Update method of the Document object.

Below is a chart that shows the parameter functionality and the equivalent API function.

| | |
|---|---|
|  | `Parameter.Name`<br><br>`Ex: Parameter.Name = "Length"` |
|  | `Parameter.Units`<br><br>`Ex: Parameter.Units = "mm"` |
|  | `Parameter.Expression`<br><br>`Ex: Parameter.Expression = "Height/2 + 5 in"` |
|  | `Parameter.Value`<br><br>`Ex: MsgBox "Nominal Value: " & Parameter.Value & " cm"` |
|  | `Parameter.ModelValueType`<br><br>`Ex: Parameter.ModelValueType = kUpperValue` |
|  | `Parameter.ModelValue`<br><br>`Ex: MsgBox "Model Value: " & Parameter.ModelValue & " cm"` |
|  | `Parameter.ExposedAsProperty`<br><br>`Ex: Parameter.ExposedAsProperty = True` |
|  | `Parameter.Comment`<br><br>`Ex: Parameter.Comment = "This is the comment."` |

There are a few new concepts to understand when working with parameters through the API versus using them in the user-interface. Some of these are also general concepts that apply to other areas of the API as well. Below is a discussion of some of the properties above and these new concepts.

**Expression**

We looked at the Expression property in an earlier example. Remember that this is exactly the equivalent of the Equation field in the parameters dialog. Setting the Expression property has the same rules as entering it manually in the parameters dialog.

### Value

As shown in the chart above, the Value property is the equivalent of the Nominal Value field in the parameters dialog. However, there are some differences between the Nominal Value field in the dialog and the Value property or the API. First, whenever values are passed in the API, either in or out, they use Inventor's internal units; **distance values are always in centimeters** and **angle values are always in radians**. This isn't true of the Expression property since it is just a String that's describing a value. The Value property returns a Double value and is the real internal value of that parameter after its expression has been evaluated. This concept also applies when reading the ModelValue property.

The idea of a consistent internal unit system applies throughout the entire API and is not limited to parameters. For example if you get the length of a line through the API it will always be in centimeters regardless of the unit type the end-user has specified in the Document Options dialog.

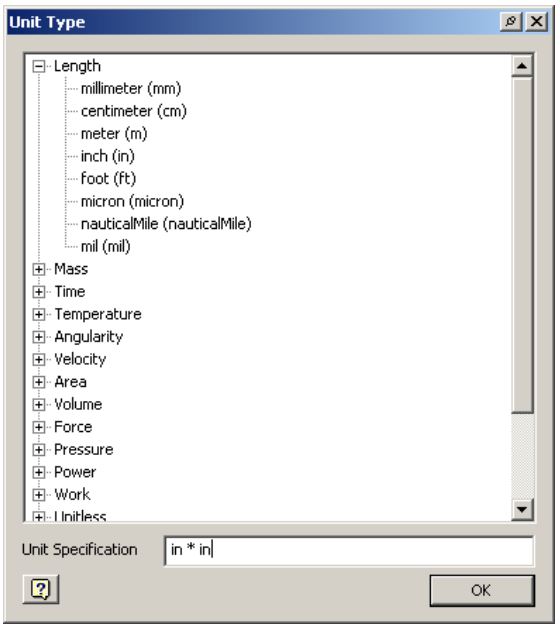
Another difference between the Nominal Value field in the dialog and the Value property of the API is that the Value property is editable. That is you can set the parameter using a Double value instead of having to create a string that represents the value. The same concept of internal units applies when setting the Value property; it is always in internal database units.

### Unit

The Unit property returns a String identifying the unit type associated with the parameter. The Unit property can be set using a String that defines a valid unit or you can use one of the predefined unit types in UnitsTypeEnum. The two statements below are both valid and have the same result.

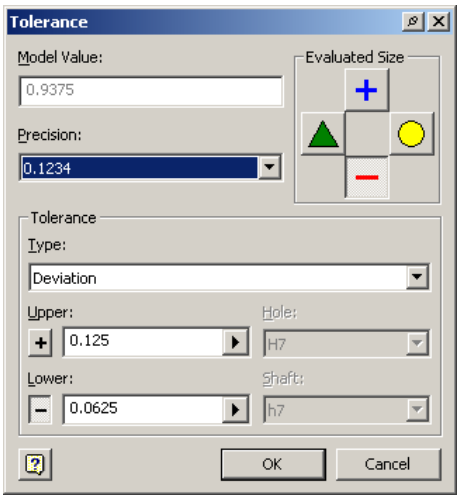
```
Parameter.Unit = "in"
Parameter.Unit = kInchLengthUnits
```

The UnitsTypeEnum contains a list of the commonly used units. It's the same list of units that are available if you click on the Units field in the parameters dialog and the "Unit Type" dialog is displayed, as shown to the right. The primary benefit of using a String is that you can define unit types that are not available in the enum list. For example, square inches are not a predefined unit type but by specifying the string "in * in" or "in ^ 2" you can define square inch units. A unit for acceleration could be defined as "(m / s) / s" or "m / (s^2)". Almost any engineering unit can be defined by combining the predefined base units.




### Tolerance

The Tolerance property of the Parameter object returns a Tolerance object. Using this object you can get and set the various tolerance options for the parameter. The Tolerance dialog shown below illustrates setting a tolerance for a parameter interactively. (You access this dialog interactively by selecting the Equation field of a parameter in the Parameters dialog and clicking the arrow at the right of the field to select the "Tolerance…" option.) This example sets a deviation type of tolerance with a +0.125/-0.0625 tolerance value. It also sets the evaluated size to be the lower value and a precision of 4 decimal places. The code below uses the Tolerance object to define the same tolerance.



The line below sets the tolerance using Strings to define the tolerance values. When using a String the unit can be defined as part of the String or it will default to the document units.

```
Call oParam.Tolerance.SetToDeviation("0.125 in", "0.0625 in")
```

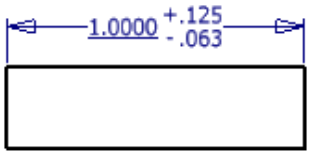
The line below accomplishes exactly the same result but specifies the tolerances using Double values. When providing a Double value, instead of a String, it is always in internal units. In this case the dimension is a length so it is in centimeters.

```
Call oParam.Tolerance.SetToDeviation(2.54 / 8, -2.54 / 16)
```

These last two lines define the number of decimal places to display for the model value and specifies that the lower value is to be used as the evaluated sized.

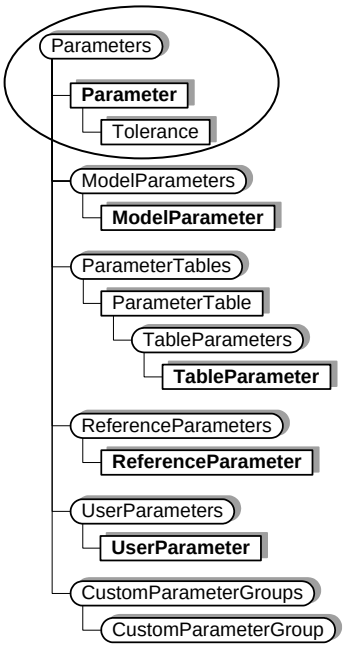
```
oParam.Precision = 4
oParam.ModelValueType = kLowerValue
```

Here's the result in the part.




| Parameter Name | Unit | Equation | Nominal Value | Tol. | Model Value | | Comment |
|---|---|---|---|---|---|---|---|
| d0 | in | 1 in | 1.000000 | | 0.937500 | | |

## Creating Parameters




You may have noticed that so far in the discussion of the API we have not discussed the different types of parameters (model, user, reference, or table). The code we've looked at deals with all parameter types as a single generic "Parameter" type. The line of code shown below will work regardless of what type of parameter "Length" is.

```
Set oLengthParam = oParameters.Item("Length")
```

Shown to the right is the complete object model for parameters. The previous samples have just used the circled portion. The Item property of the Parameters collection object provides access to all of the parameters in the document, regardless of their type. Also from the Parameters collection object you can access other collections that contain the parameters of a specific type. The Item property of those collections will return only the parameters of that specific type. It's also through these type specific collections that you create new parameters. The sample below illustrates creating a user parameter. Notice that it uses the UserParameters collection object.

```
Dim oUserParams As UserParameters
Set oUserParams = oCompDef.Parameters.UserParameters

Dim oParam As Parameter
Set oParam = oUserParams.AddByExpression("NewParam1", "3", _
                     kInchLengthUnits)
```

The example above uses the AddByExpression method of the UserParameters collection to create a new parameter called “NewParam1”. The value of the parameter is “3” and the units are inches. Because the units are specified to be inches the value of “3” is interpreted to be 3 inches. Below is a similar example.

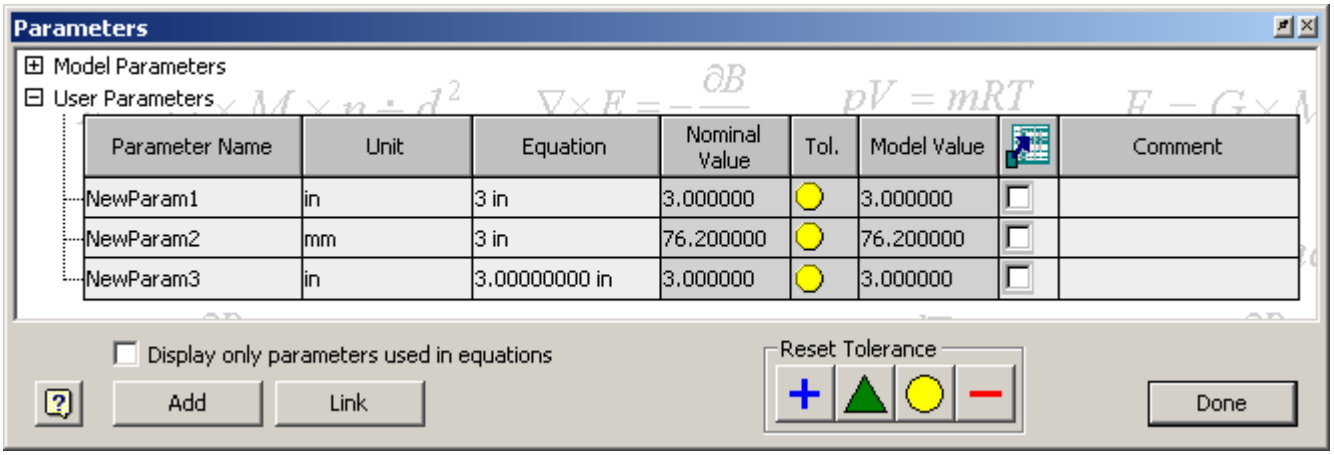
```
Set oParam = oUserParams.AddByExpression("NewParam2", "3 in", _
                     kMillimeterLengthUnits)
```

The example above uses the same method to create the parameter named “NewParam2”. In this case the units are specified to be millimeters but because the expression also defines the unit, the resulting parameter value will still be 3 inches, but the parameter unit will be millimeters. The code below uses the AddByValue method to create a new parameter.

```
Set oParam = oUserParams.AddByValue("NewParam3", 3 * 2.54, kDefaultDisplayLengthUnits)
```

The AddByValue method is similar to the AddByExpression method except instead of the second argument being a String that defines the expression of the parameter it is a Double that defines the parameter value using internal units. This means that for lengths the value specified is **always in centimeters**. The units for AddByValue are defined just the same as they were for AddByExpression. However, this sample demonstrates the use of a special enum value within the UnitsTypeEnum. The unit type kDefaultDisplayLengthUnits specifies that the API is to use whatever the default length unit is for the document. This is the length that the user specifies in the Document Options dialog. When you use this enum, or the equivalent kDefaultDisplayAngleUnits for angles, you’re not specifying a particular unit but only the type of unit; typically length or angle. The specific unit type is picked up from the document. Since the value you specify is always in internal units, (centimeters or radians), it doesn’t matter to you what specific units the end-user has chosen to work in.

The end result of the previous lines of code creates the parameters shown below. The default units for the document are inches. That’s why NewParam3 has the unit type “in”.




Through the user-interface you can only create user and table parameters. (Table parameters are created by linking an Excel spreadsheet.) Model and reference parameters are created automatically as you add dimension constraints to the model. Through the API it is possible to create model and reference parameters. They’re created the same way as user parameters except you use the Add methods of the ModelParameters and ReferenceParameters objects. For most programs you will want to create user parameters. Model and reference parameters are typically created by applications that want to expose values to the end-user with the behavior of model or reference parameters.

### Units of Measure

There's another area of the API that can be useful when working with parameters. The UnitsOfMeasure object is obtained using the UnitsOfMeasure property of the Document object. Each document has its own UnitsOfMeasure object since each document can have different default units defined. The following illustrates getting the UnitsOfMeasure object when a form is initialized and assigning it to a global variable within the form.

```
Private m_oUOM As UnitsOfMeasure

Private Sub UserForm_Initialize()
    Set m_oUOM = ThisApplication.ActiveDocument.UnitsOfMeasure
End Sub
```

Typically the UnitsOfMeasure object is used whenever you need to interact with the end-user and have them input values or you need to display a value back to the end-user. For example, perhaps your program displays a dialog to allow the end-user to specify a length in a part. If you want your program to have the same behavior as standard Inventor commands you should support units and equations. For example, if the end-user types "4 in", how would you handle it? Or if they type "4 cm + d0" what would you do? It would be a lot of work to correctly handle these inputs if you had to do it on your own. Instead you can take advantage of the UnitsOfMeasure object and let it do the work for you. The examples below describe and illustrate the most common uses of this object.

The code below illustrates using the CompatibleUnits method to check if an expression defines a valid length. It's implemented in the Change event of a text box. If the expression isn't valid it changes the text color of the text box to red to indicate to the end-user the expression is invalid.

```
Private Sub TextBox1_Change()
    On Error Resume Next
    ' Check that the expression is a valid length.
    Dim bCompatible As Boolean
    bCompatible = oUOM.CompatibleUnits(TextBox1.Text, kDefaultDisplayLengthUnits, _
                  "1", kDefaultDisplayLengthUnits)

    ' Check if it was successful and change the text to the appropriate color.
    If Err or Not bCompatible Then
        TextBox1.ForeColor = vbRed
    Else
        TextBox1.ForeColor = vbWindowText
    End If
End Sub
```

The code below illustrates using the GetValueFromExpression method to evaluate an equation and get the resulting value. The first argument is the String that represents the equation. The second argument specifies the unit type that the expression should represent. In the example below it's specifying that the expression should be whatever the current default length units for the document are. You would use kDefaultDisplayAngleUnits if the expression represents an angle. The value passed back is in database units.

```
' Get the real value of the input string.
Dim dValue As Double
dValue = oUOM.GetValueFromExpression(txtBox1.Text,  kDefaultDisplayLengthUnits)
```

Let's look at an example of input and the corresponding output from the previous example. If the user enters "5" in your dialog's text box and the current length units for the document are inches it is interpreted by the GetValueFromExpression method to be 5 inches. The value returned is 12.7 which is the equivalent of 5 inches in centimeters. The great thing about this method is that you don't need to do anything special to handle the various units and equations. You just pass any equation into the function and as long as Inventor can evaluate it, it will pass back the result.

Another common use of the UnitsOfMeasure object is to display results back to the end-user. If you perform a calculation in your program that is in a known unit you can use the UnitsOfMeasure object to format a String to display to the end-user. The example below illustrates this.

```
MsgBox "Length: " & oUOM.GetStringFromValue(dValue, kDefaultDisplayLengthUnits)
```

This example uses the GetStringFromValue method to get a correctly formatted String to display to the end-user. For example, if you've calculated the length of an object and have the value, represented by the variable dValue in the sample, this method will return a String that represents that value in the current default length units of the document. Remember that the units for a length input value will always be centimeters and for an angle will always be radians. In this case if you passed in the value 12.7 and the current length unit of the document is inches, Inventor will return the string "5.00 in".

This also helps to illustrate the power of the UnitsOfMeasure object and how easy it makes interacting with the end-user. In this case you don't need to worry about any of the document units. The end-user can specify any unit type for the document and it doesn't matter to you. You just do your calculations in internal units and anytime you need to display a result to the end-user you pass in the value and Inventor provides the correctly formatted String to display to the end-user.

### Putting it all Together

The parameters portion of the API is fairly simple but it can become powerful when you show your creativity in applying this functionality towards solving your specific problems. The power in using the API with parameters is that it allows you to quickly change one or more parameters based on any input you choose. Here are some examples of potential uses of the parameter portion of the API.

Delivered with Inventor is a programming sample whose primary function is to change parameter values. It also takes advantage of other API functionality to create a complete set of functionality. The sample program is delivered in the SDK\Samples\VB\AddIns\Analyze directory where Inventor was installed. The readme.txt file describes how to run the sample. In this sample you choose from the parameters in the active document and for each parameter choose a starting and ending value and the number of steps. The program then computes intermediate values for the parameters between the starting and ending values. It assigns the values to the parameters for each step and updates the assembly. The result is that you see the assembly animating as it is driven by the parameters.

Another common use of the parameter portion of the API is to drive the sizes of parts and assemblies based on external input. Most commonly this is done for some type of configuration application. For example, an order for a particular product can be made and communicated to you. Based on this order you are able to determine which parts are needed to create the specified product. This logic is up to you but could be based on information in a database or just logic within your program. You can then edit the parameters of parts and assemblies to get the desired sizes and create the desired finished product.

## Example Parameter Programs

There are some associated sample programs that provide more complete examples of working with parameters. Each sample is described in more detail in a readme located with the sample.

**Configurator** - This sample is in the Configurator directory and consists of several files including a Visual Basic program. This example copies a part and drawing to a defined new name and edits the parameters to specified sizes.

**Animate –** This sample is in the Animate directory and consists of several files including a VBA project. This example lets you specify starting and ending values for a set of parameters and animates the changes.

**ParameterEdit** - This sample uses Excel to define new values for a set of parameters and then changes the values in the active part or assembly.