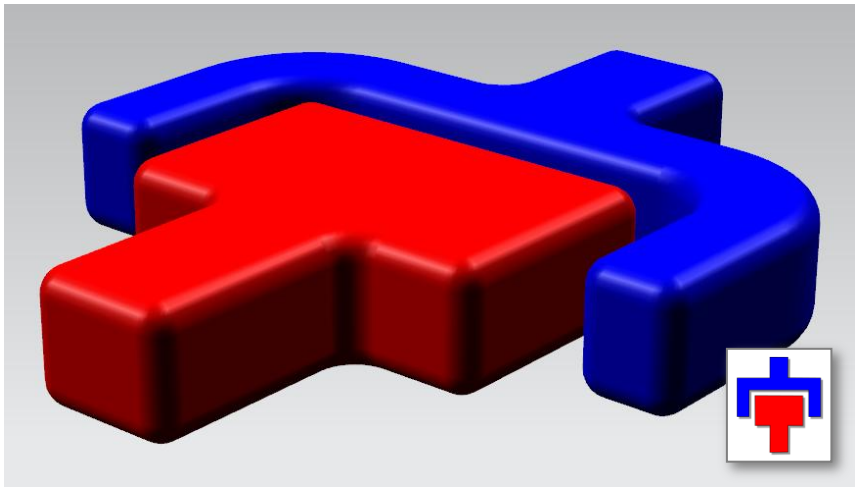
Getting Started with SNAP

Revision 10.0. October 2014



SIEMENS

Table of Contents

Chapter 1: Introduction

What Is SNAP ?

S.N.A.P. stands for Simple NX Application Programming. It's an Application Programming Interface (API) that lets you write programs to customize or extend NX. The benefit is that small applications created this way can often speed up repetitive tasks, and capture important design process knowledge.

NX already has other APIs, of course, including GRIP, NX Open, and Knowledge Fusion (KF), so you may be wondering why yet another one is needed. The GRIP language has not been enhanced for many years, so it's very much behind the times. NX Open and KF are enormously broad and powerful, but the power comes with a lot of complexity, and many people find it difficult to even get started. So, the main point of SNAP is that it's designed to be learned quickly by average NX users — people who have little or no previous programming experience. The focus is on simplicity and ease of learning, so that typical users can write small programs to improve their productivity without a lot of study and preparation. Since SNAP is based on NX Open, you can smoothly graduate to NX Open programming later, if you want.

You may have noticed that SNAP sounds a little like GRIP. This is not an accident. Although it's based on completely new development and entirely different technology, SNAP is very similar to GRIP in spirit and purpose. So, if you remember GRIP, and you liked it, we hope you'll like SNAP, too.

If you'd like a little background information, please keep reading here. If you can't wait, and you just want to start writing code immediately, please skip to chapter 2, where we show you how to proceed.

Purpose of this Guide

This guide is a beginner's introduction to programming using SNAP. It will get you started in writing your first few applications, and give you a sample of some of the things that are possible with SNAP.

You don't need to have any programming experience to read this document, but we assume you have some basic knowledge of NX and Windows. If you are an experienced programmer, the only benefits of this document will be the descriptions of programming techniques specific to NX.

SNAP is just a .NET library, so it can be used with any .NET-compliant language. In this document, we focus on the Visual Basic (VB) language, but in most cases it will be obvious how to apply the same techniques in other .NET languages, such as C#, C++, IronPython, F#, etc.

Where To Go From Here

The next two chapters show you how to write programs in two different environments. If you have no programming experience, you won't understand much of the code you see. That's OK — the purpose of these two chapters is to teach you about the programming environments and capabilities, not about the code.

Chapter 2 discusses programming using the NX Journal Editor. The only real advantage of this environment is that it requires no setup whatsoever — you just access the Journal Editor from within NX, and you can start writing code immediately. But, by the time you reach the end of the examples in chapter 2, you will probably be growing dissatisfied with the Journal Editor, and you will want to switch to a true "Integrated Development Environment" (IDE) like Microsoft Visual Studio.

Chapter 3 discusses Microsoft Visual Studio. We explain how to download and install a free version, and how to use it to develop SNAP programs. If you have some programming experience, and you already have Visual Studio installed on your computer, you might want to skip through chapter 2 very quickly, and jump to chapter 3.

Chapter 4 provides a very quick and abbreviated introduction to the Visual Basic (VB) programming language. A huge amount of material is omitted, but you will learn enough to start writing SNAP programs in VB. If you already know Visual Basic, or you have a good book on the subject, you can skip this chapter entirely.

In Chapter 5, we provide a brief overview of SNAP concepts and architecture. It's not really necessary for you to know all of this, but understanding the underlying principles might help you to learn things more quickly.

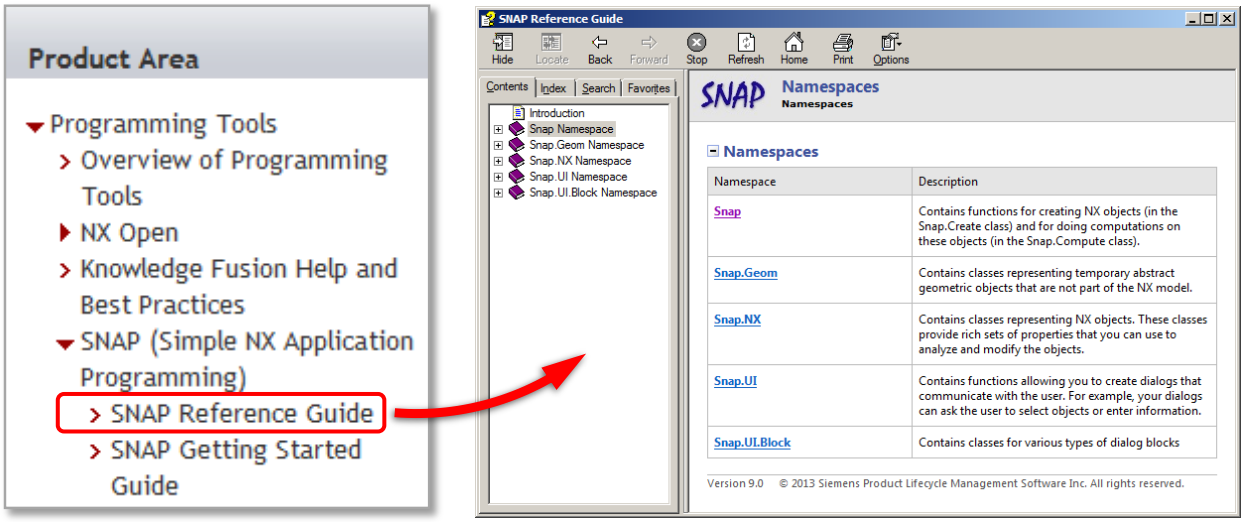
Chapters 6 through 15 provide brief descriptions of some SNAP functions, and examples of their uses. We focus on basic techniques and concepts, so we only describe a small subset of the available functions. You can get more complete information either from the SNAP Reference Guide or by using the Object Browser in Visual Studio.

In chapter 16, we explain how NX Open works. After you have been writing SNAP programs for a while, you will understand some basic principles, and NX Open should be easier to approach. If you find that SNAP alone does not provide everything you need, you can use NX Open to plug the gaps.

Finally, in chapter 17, we tell you how to deal with some common problems, if they should arise.

Other Documentation

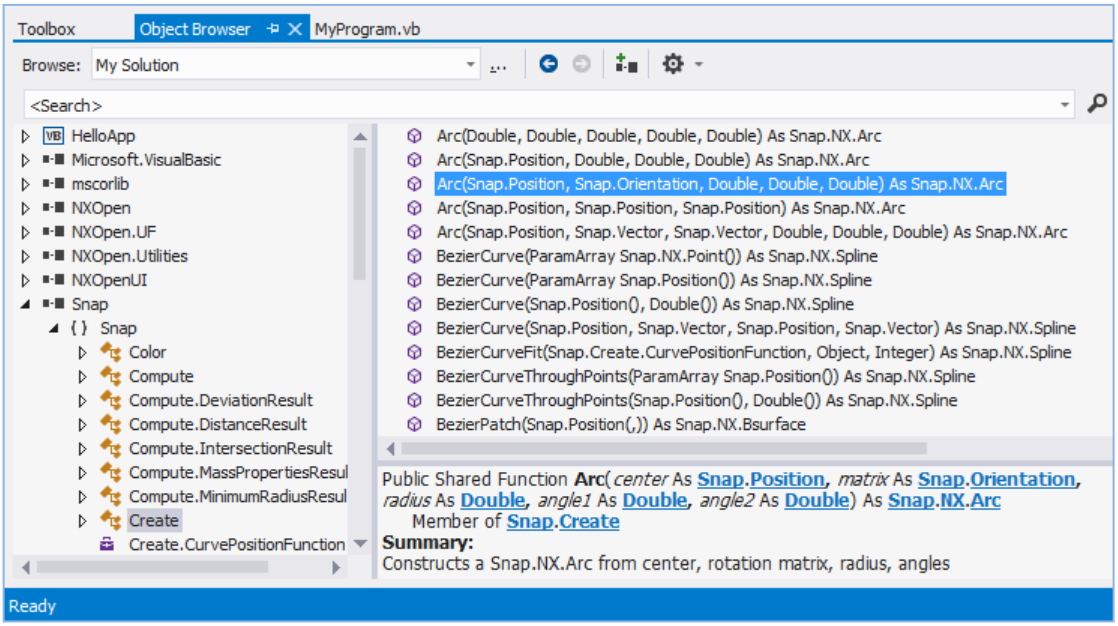
The definitive source of information about the capabilities of SNAP is the SNAP Reference Guide, which you can find in the NX documentation set in the location shown below:



The document is fully indexed and searchable, so we hope you'll be able to find the information you need. It describes all SNAP functions in detail, and includes several hundred sample programs.

If you get tired of clicking through all the security warnings that appear when you access the NX documentation, you can fix this. In Internet Explorer, choose Tools→Internet Options→Advanced. Scroll down to the Security set of options near the bottom of the list, and check "Allow active content to run in files on My Computer".

In Visual Studio, another option is to use the Object Browser, which you can access from the View menu:



The Object Browser won't let you see the example programs and explanatory remarks that are in the Reference Guide, but it might be easier to access while you're in the middle of writing some code.

Actually, you may find that you don't need either the SNAP Reference Guide or the Visual Studio Object Browser, because all the information you need about calling a function is given by Visual Studio "intellisense" as you type.

If you have some experience with the GRIP language, then there's a document called "SNAP and NX Open for GRIP Enthusiasts" that might be helpful to you. It explains SNAP and NX Open programming in terms that are likely to be familiar to people who have used GRIP, and shows you how to map GRIP functions to SNAP and NX Open ones. You can find that document in the standard NX documentation set, in roughly the same place that you found this one.

Example Code

Once you understand the basic ideas of SNAP, you may find that code examples are the best source of help. You can find example programs in several places:

- In this guide. There are about a dozen example programs in chapters 2 and 3, along with quite detailed descriptions. Also, the later chapters contain many "snippets" of code illustrating various programming techniques.
- In the SNAP Reference Guide, there are several hundred example programs that show you how to use the functions described there. These are all very small programs, and very few of them do anything that is truly valuable, but you will probably find them helpful in understanding function usage.
- There are some examples in [...NX]\UGOPEN\SNAP\Examples. There are two folders: the one called "Getting Started Examples" contains the examples from this guide, and the "More Examples" folder contains some larger examples that try to do more useful things. Here (and in the remainder of this document), [...NX] denotes the folder where the latest release of NX is installed, which is typically C:\Program Files\Siemens\NX 10, or something similar.

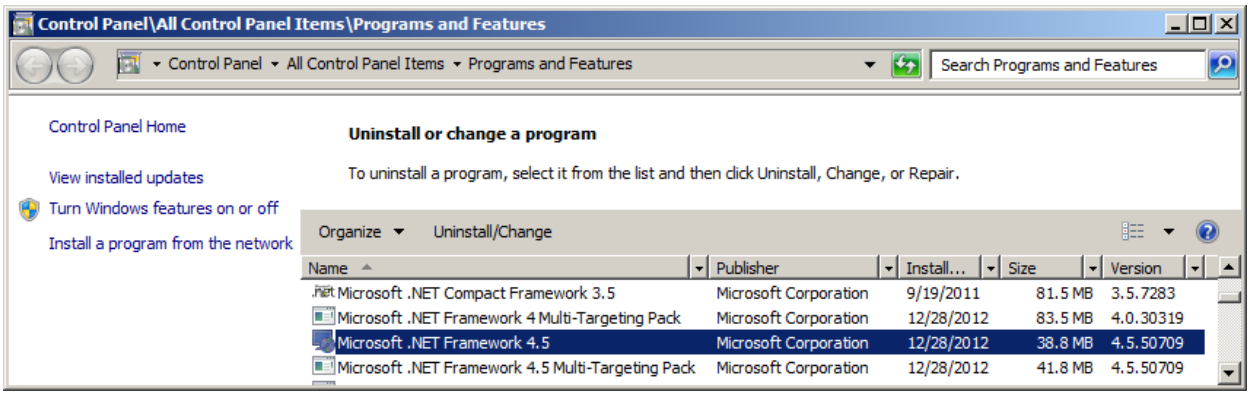
If you've read everything, and you're still stuck, you can contact Siemens GTAC support, or you can ask questions in the NX Customization and Programming Forum at the Siemens PLM Community site.

Chapter 2: Using the NX Journal Editor

In this chapter, we will discuss creation of simple programs using the NX Journal Editor. This is not a very supportive environment in which to write code, but it's OK for very simple programs, and it requires no setup. In the next chapter, we will discuss the use of Microsoft Visual Studio, instead. This requires a small preparation effort, but provides a much nicer development environment.

System Requirement — The .NET Framework

To use SNAP with NX 10, you need version 4.5 of the .NET Framework, or newer. It's possible that you have several versions installed (which is quite OK) — you can use the "Programs and Features" Control Panel to check:



If you don't have version 4.5 or later, please download and install it from this Microsoft site.

Typographic Conventions

In any document about programming, it's important to distinguish text that you're supposed to read from code that you're supposed to type (which the compiler will read). In this guide, program text is either enclosed in yellowish boxes, as you see at the top of the next page, or it's shown in `this blue font`. References to filenames, pathnames, functions, classes, namespaces, and other computerish things are written in this dark blue color. The dark blue is fairly close to black, so as not to cause too much clutter and distraction.

Licensing — SNAP and MiniSNAP

SNAP is fairly inexpensive, but it's not free — you need to purchase a "SNAP Author" license (nx_snap_author) in order to use the full package. This license allows you to:

- Run code that calls SNAP functions in the NX Journal Editor
- "Sign" the compiled programs you write (so that other people can run them more easily)
- Run compiled programs that call SNAP functions, even if they have not been signed

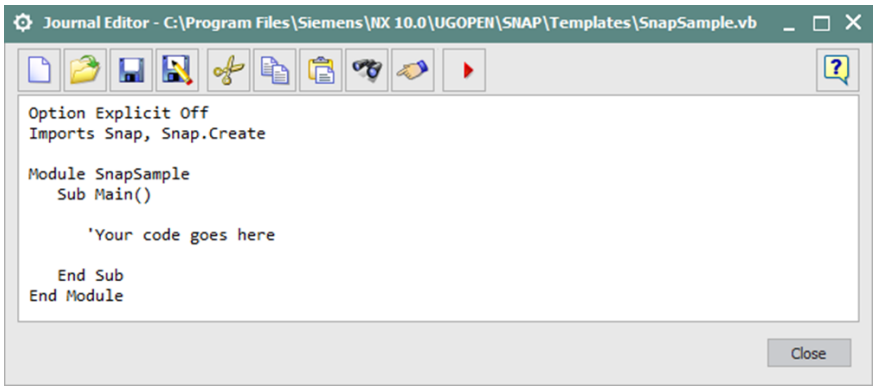
But we want you to be able to experiment with SNAP, so we provide a free scaled-down version called MiniSNAP. The capabilities of MiniSNAP are quite limited, but it does have enough functions to let you work through the examples in the next two chapters. All you have to do is replace the word "Snap" with "MiniSnap" in your code. Experience has shown that you'll probably forget to do this, so we'll remind you from time to time.

Example 1: Hello World

When learning a new programming language or environment, it's traditional that the first program you write should be one that simply outputs the text "Hello World". We will follow that tradition here, too.

Run NX, create a new part file, and then choose Tools→Journal→Edit (or press Alt+F11). The Journal Editor window will appear. You may see some text in this window, but you can ignore it.

Click on the second icon in the Journal Editor toolbar and open the file SnapSample.vb, which you can find in [...NX]\UGOPEN\SNAP\Templates. Remember that [...NX] is just shorthand for the location where NX is installed, which is typically somewhere like C:\Program Files\Siemens\NX 10. You should see some text like this:



What you see here is the framework for a simple Visual Basic program. The framework itself won't do anything interesting until we fill in some real content, which we will do shortly.

If you can't find the file SnapSample.vb, for some reason, it's no great loss — you can just type the text shown above into the Journal Editor, or copy it from here:

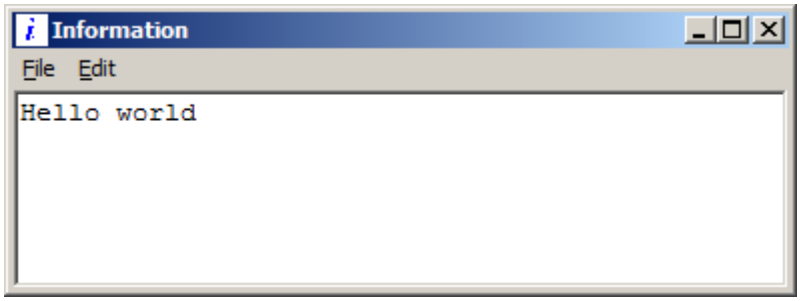
```
Option Explicit Off
Imports Snap, Snap.Create

Module SnapSample
   Sub Main()

      'Your code goes here

   End Sub
End Module
```

Within the Journal Editor, you can find the Cut/Copy/Paste functions on the right-mouse-button menu, or you can use the standard keyboard shortcuts (Ctrl+X, Ctrl+C, Ctrl+V). Copying text from a PDF file doesn't always work very well, and you may find that the pretty indentation gets ruined, or the line endings get lost. Generally, the formatting of the text doesn't matter, except for readability, but you will have to fix the line endings, if they get messed up. If you find that copying/pasting from this document is troublesome, you can get the example code from [...NX]\UGOPEN\SNAP\Examples\GS Guide instead.

Once you have the right text in the Journal Editor, delete the entire line that says "`Your code goes here`" (including the initial quotation mark) and insert the following line in its place:

```
InfoWindow.WriteLine("Hello world")
```

Our code is now complete, so we're ready to run it. Click on the "Play" icon in the toolbar (the red triangle arrow at the upper right). This will send your code to the Visual Basic compiler, which will translate it into "object code" that your computer can execute. You won't see this executable code, but it will immediately be run, and this should cause the NX Information window to appear, like this:



If you receive some sort of error, rather than the output shown above, here are some possible causes:

- Maybe you typed something incorrectly, in which case the compiler will probably complain that it can't understand what you wrote. An error message will tell you in which line of code the problem occurred. The description of the error might not be all that helpful, but the line number should be.
- Maybe you don't have an up-to-date version of the .NET framework installed, as mentioned above. This may cause a mysterious error that reports an "Invalid attempt to load library".
- Maybe your system doesn't have any nx_snap_author licenses available (perhaps you didn't purchase any, or they are all in use by other people). In this case, you can use MiniSNAP instead of SNAP: just change the second line of code to read "`Imports MiniSnap, MiniSnap.Create`".
- Maybe you neglected to delete the quotation mark at the beginning of the line "`Your code goes here`", in which case your code will run without any errors, but the NX Information window will not appear

There is a troubleshooting guide in chapter 17 that will help you figure out what went wrong, and get it fixed. Fortunately, you will only have to go through the troubleshooting exercise once. If you can get this simple "hello world" program to work, then all the later examples should work smoothly, too.

Once you have successfully run the program, you might want to save your work. If so, use the Save As icon on the Journal Editor toolbar, and save your file as HelloWorld.vb, or something like that.

Next we're going to analyze this code briefly, to understand what it did. If you're not interested in this, and you're willing to just accept it as magic, you can skip directly to example 2.

Lines of code	Explanation
`Option Explicit Off` `Imports Snap, Snap.Create`	Don't worry about this stuff, for now. It's a standard framework that will appear in all the programs you write, for a while.
`Module SnapSample`	All code has to belong to either a "class" or a "module". This line says that our code is going to belong to a module called "SnapSample".
`Sub Main`	All code has to belong to some subroutine or function. This says that our code will belong to a subroutine called Main. The name "Main" is special — this is the place where your code typically starts executing.
`InfoWindow.WriteLine("Hello world")`	We call a SNAP function to write a line of text to the NX Info Window
`End Sub`	The end of our subroutine, Main
`End Module`	The end of our module

Example 2: Creating Simple Geometry

In this next example, we create some simple geometry. With NX running, and a part file open, choose Tools→Journal→Edit (or press Alt+F11). In the Journal Editor window, open the file SnapSample.vb.

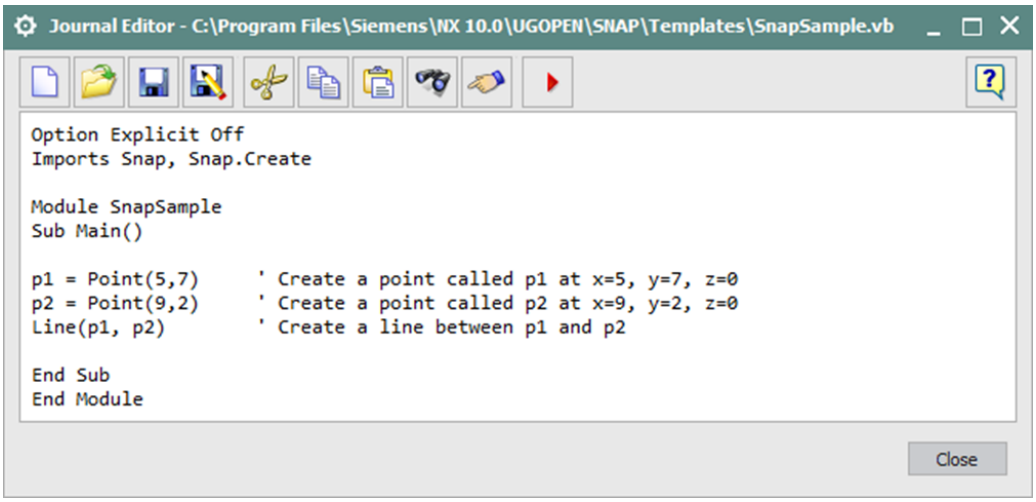
As before, delete the line that says "Your code goes here", and type in the following code. If you don't like typing, you can copy the code from here and paste it into the Journal Editor window. The paste operation in the Journal Editor is available on the right-mouse-button menu, or you can use the standard Ctrl+V shortcut.

```
p1 = Point(5,7)      ' Create a point called p1 at x=5, y=7, z=0
p2 = Point(9,2)      ' Create a point called p2 at x=9, y=2, z=0
Line(p1, p2)         ' Create a line between p1 and p2
```

Obviously this code just creates two points and a line. Note that we didn't have to provide z-coordinates for the two points; we omitted them and SNAP just assumed them to be zero.

This code also introduces the concept of "comments" (shown in green above). Any text between a single quote mark (') and the end of the line is considered to be a comment. These comments are ignored by the compiler — they are just a way of documenting the code and making it easier for people to understand.

The Journal Editor window doesn't support color, so your code will look like this



Change "Snap" to "MiniSnap" (twice) in the second line if you don't have a SNAP Author license.

Click on the "Play" icon (the red triangle on the upper right), and your code will be executed, producing two points and a line in the NX window. It's a fairly small line, so you may have to Zoom in (or Ctrl+F) to see it.

For a little more information about creating points and lines, please refer to chapter 6 and chapter 7. More detailed discussion is provided in the SNAP Reference Guide, along with example code,. More on this later.

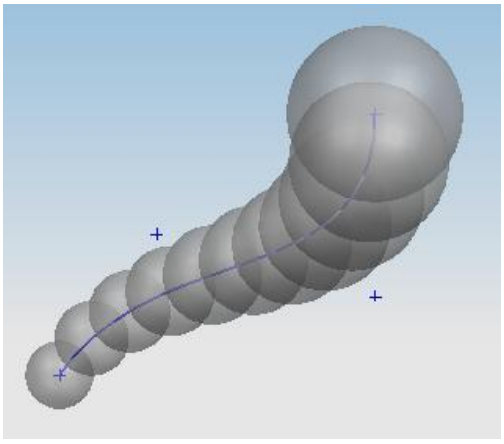
Example 3: Some More Interesting Geometry

In this example, we will create some slightly more interesting geometry, and will also introduce a technique for repetitive actions (looping). So, as usual, start up the Journal Editor window, and open the file SnapSample.vb.

As before, delete the line that says "`Your code goes here`", and copy/paste the following code in its place.

```
p1 = Point(0, 0)  :  p2 = Point(6, 0)  :  p3 = Point(6, 6)  :  p4 = Point(6, 6, 6)

spine = BezierCurve(p1, p2, p3, p4)     ' Create centerline of worm shape

factor = 1.1                            ' The "growth factor" for the worm shape

For count = 0 to 10                     ' Step along spine curve
   t = count*0.1
   p = spine.Position(t)                ' Calculate point on spine curve
   r = factor^count                     ' Calculate radius
   Sphere(p, 2*r)                       ' Create sphere
Next
```

Replace "Snap" with "MiniSnap" if you need to, and then click on the "Play" icon, and something like this should appear in the NX window (I made the spheres transparent to show the spline curve inside).



Note that this code won't work if NX is in History-Free mode, as explained in chapter 10 and chapter 17.

The meanings of the more interesting lines of code are as follows:

Lines of code	Explanation
`spine = BezierCurve(p1, p2, p3, p4)`	Creates a Bezier curve called "spine" from the four points p1, p2, p3, p4. A Bezier curve is just a simple kind of spline curve. See chapter 7 for more information about splines.

`For count = 0 to 10`
`  <the body of our loop>`
`Next`
	This is a repetitive "loop" process. The statements between the "For" statement and the "Next" statement are executed 11 times, with the variable called "count" set equal to 0, 1, 2, …, 10 successively.

`t = count*0.1`	Calculates a parameter value, t, based on the count. So, as the loop repeats, t gets values 0.0, 0.1, 0.2, …, 1.0
`p = spine.Position(t)`	Calculates a position on the curve "spine" at the parameter value t.
`r = factor^count`	Calculates a radius value by raising "factor" to the power "count"
`Sphere(p, 2*r)`	Creates a sphere with center location p and diameter 2*r. For more information about creating simple, solids, please see chapter 8.

There are many different ways to write this same code. For example, you can get rid of the variable t, and just write

```
p = spine.Position(count*0.1)
```

In fact, you can squeeze the entire body of the loop into just one statement, if you really want to:

```
Sphere( spine.Position(count*0.1), 2*factor^count )
```

but this just makes the code harder to read. You can create circles instead of spheres by using the following loop instead of the original one. Place this code after the line "`factor = 1.1`":

```
n = 50

For index = 0 to n
  t = index*(1.0/n)
  spinePoint   = spine.Position(t)
  spineTangent = spine.Tangent(t)
  power = 10*t
  radius = factor^power
  Circle(spinePoint, spineTangent, radius)
Next
```

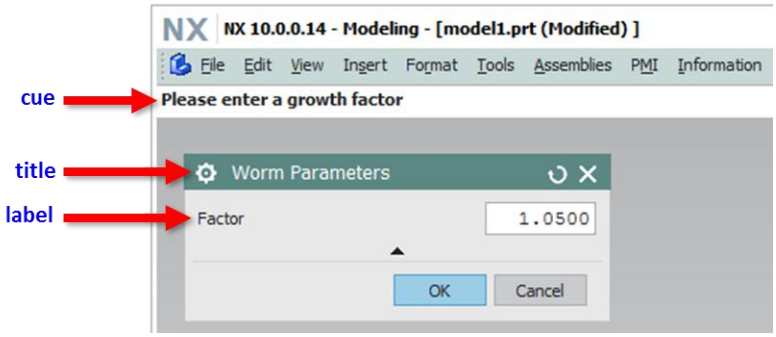
The statement `Circle(p, v, r)` creates a circle with center p, normal vector v, and radius r. You can try increasing the value of n to something larger than 50, to so see how fast SNAP code can create geometry. If it takes longer than about 10 seconds to create 5000 circles, maybe it's time to go shopping for a new computer. ☺

Example 4: Getting Input from the User

In the previous example, the spheres gradually grew in size as we moved along the "spine" of the worm. The growth factor was set to a constant, 1.1. Next we are going to allow the user to choose this growth factor. Take the previous example, and put the following code in place of the line "`factor = 1.1`":

```
cue = "Please enter a growth factor"
title  = "Worm Parameters"
label = "Factor"
factor = Snap.UI.Input.GetDouble(cue, title, label, 1.05)
```

When you run this code, the cue text will appear in the NX cue line, and a dialog will appear, asking you to enter a value for the growth factor:



This dialog might be hidden behind the main NX window, in which case you will have to hunt for it. Enter a value, and then click on OK, and the familiar worm-shaped geometry will appear.

The GetDouble function is a very simple way to get input from the user. Other simple input functions are explained in chapter 12. You can build far more sophisticated dialogs using either NX block-based dialogs or the .NET tools for constructing Windows Forms. We have written the code in four lines, for clarity, but in practice you would probably put `Imports Snap.UI.Input` at the top of the file, and then write just one line, like this:

```
factor = GetDouble("Please enter a growth factor", "Worm Parameters", "Factor", 1.05)
```

If you enter a growth factor greater than around 1.2, you will get a fairly strange result, because the spheres used for the body of the "worm" will get very large. You might want to add some code to enforce "design standards" for these worm objects. For example, immediately after the GetDouble call, you could write:

```
If factor > 1.2 Then factor = 1.2
```

Example 5: Using Vectors

In this example, we construct a circular arc through three points to estimate the radius of curvature of a spline curve at its mid-point. Change "Snap" to "MiniSnap" if you need to, and then paste the following code into the file SnapSample.vb in the usual place inside the "Main" subroutine:

```
myCurve = BezierCurve(Point(0,0), Point(1,0), Point(1,1))
p1 = myCurve.Position(0.5 - 0.0001)        ' A tiny bit before the mid-point
p2 = myCurve.Position(0.5)                 ' At the mid-point
p3 = myCurve.Position(0.5 + 0.0001)        ' A tiny bit after the mid-point

u = p2 - p1         ' Vector from p1 to p2
v = p3 - p1         ' Vector from p1 to p3
uu = u*u            ' Dot product of u with itself
uv = u*v
vv = v*v
det   =  uu*vv - uv*uv                  ' Determinant for solving linear equations
alpha = (uu*vv - uv*vv) / (2 * det)     ' Bad -- should check first that det is not zero !
beta  = (uu*vv - uu*uv) / (2 * det)
rvec = alpha * u + beta * v             ' Radius vector
radius = Vector.Norm(rvec)              ' Radius is length (norm) of this vector

InfoWindow.WriteLine(radius)            ' Output the radius to the Info window
```

When you run this code, a small spline curve will be created, and the value 0.707106784745667 should be output to the NX Information window. This is (roughly) the radius of curvature of the spline at its mid-point.

As this code shows, SNAP has built-in support for 3D vectors. You can add them, subtract them, form dot and cross products, measure lengths and angles, and so on. You perform these operations using the natural arithmetic operators: if u and v are vectors, then u+v is their sum, and u*v is their dot product, and so on. Please see chapter 6 for more information about working with vectors and positions.

Example 6: Using .NET Tools

The .NET framework provided by Microsoft has a huge number of useful functions that we can easily call from our code. You can read and write files, access data stored in databases, create and manipulate various types of images, work with text, interact with the Windows OS and applications like MS Word and Excel, and many other things. There are several thousand "classes" in the framework, organized into several hundred categories called "namespaces". Some of the more interesting ones are listed on this Wikipedia page. They include:

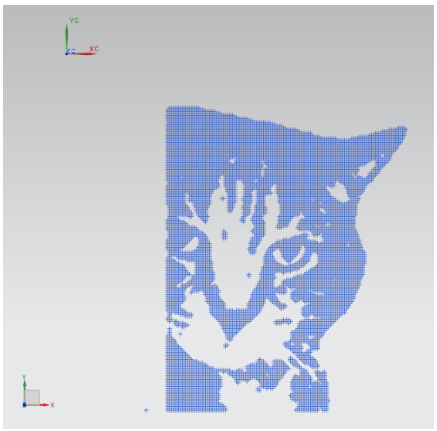
Namespace	Description
`System`	Base types like String, DateTime, Boolean, plus arrays, math functions, etc.
`System.Collections`	Provides collections used in programming, such as lists, queues, stacks, etc.
`System.Data`	Functions to access data and data services.
`System.Diagnostics`	Diagnostic tools such as event logging, performance counters, debugging, etc.
`System.Drawing`	Bitmap and vector graphics, imaging, printing, and text services.
`System.IO`	Allows you to read from and write to different streams, such as files
`System.Management`	Provides system information, such as free disk space, CPU utilization, etc.
`System.Media`	Provides you the ability to play system sounds and .wav files.
`System.Messaging`	Networking and .NET Remoting
`System.Text`	Supports various encodings, regular expressions, and string manipulation tools
`System.Threading`	Helps facilitate multithreaded programming.
`System.Timers`	Timers and stop watches for measuring system performance
`System.Windows.Forms`	Tools for building graphical user interfaces (menus, toolbars and dialogs)
`System.Xml`	Reading, writing and processing XML data

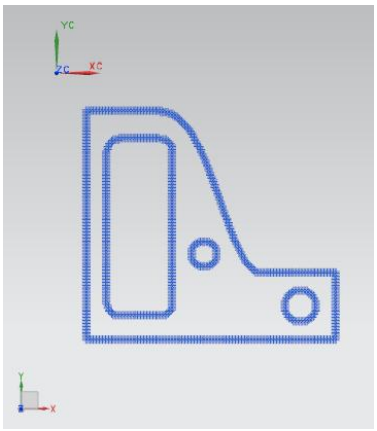
In this example, we will use a .NET function for reading data from an image. To run this example, you need a small bitmap file. If it's too large, the code will only read the top left 200x200 area of pixels. The file should be in either BMP or JPG format. A small image you scribbled in Microsoft Paint will work, or a small photograph. You have to modify the second line of code below to indicate where your bitmap file is located. The code loops through the pixels in the image, and creates an NX point wherever it finds a dark pixel.

```
Sub Main
  pixels = New System.Drawing.Bitmap("C:\sammie.jpg")     ' Reads the file. Change this !!

  width  = System.Math.Min(pixels.Width,  200)            ' Limit the width to 200
  height = System.Math.Min(pixels.Height, 200)            ' Limit the height to 200

  For x = 1 To width-1                                ' Loop over the pixels in the image
    For y = 1 To height-1
      pixelColor = pixels.GetPixel(x, y)              ' Read the pixel color at location (x,y)
      brightness = pixelColor.GetBrightness()         ' Measure the brightness of the color
      If brightness < 0.4 Then Point(x, -y)           ' If dark, create a point (but flip y)
    Next y
  Next x
End Sub
```

The interesting part of this code is the second line. Here we are calling a function from the .NET System.Drawing namespace. This function reads the contents of the bitmap file, and stores the data in a two-dimensional array called "pixels". This function has a lot of built-in intelligence — it knows how data is organized in BMP files and JPG files (and other bitmap files, too, actually), so you don't have to understand any of this. As is often the case, the .NET framework does all the hard work for you. The function Math.Min is another .NET framework function. The .NET System.Math namespace includes all the basic math functions you would expect, such as Sin, Cos, Tan, Sqrt, etc. Also, note that we had to write `Point(x, -y)` instead of just `Point(x, y)`, because bitmap coordinate systems generally have y increasing downwards.

Here are the results I got from two simple images:





The one on the left is just for fun, but the one on the right might actually have some practical value — you could use the points to fit NX curves to the image data, for example.

If you want your points to mimic the colors of the pixels in your image, remove the line that says "`If brightness < 0.4 Then Point(x, -y)`", and put the following in its place:

```
If brightness < 0.4 Then
     pt = Point(x, -y)            ' If dark, create a point (but flip y)
     pt.Color = pixelColor        ' Give the point the correct color
End If
```

Depending on the brightness and contrast of your image, you may have to adjust the "0.4" value to get good results.

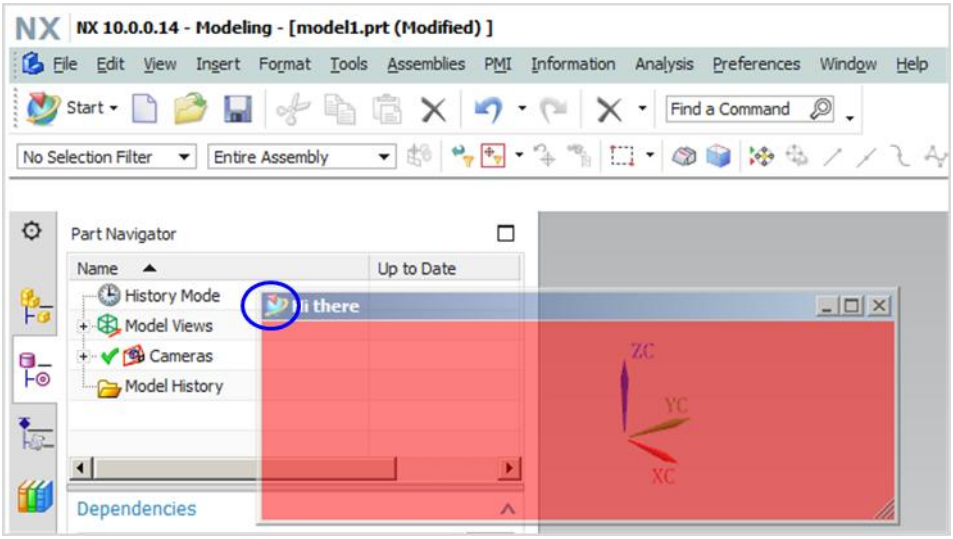
Example 7: WinForms (The Hard Way)

The .NET framework provides a wide variety of tools for designing user interface dialogs. These dialogs are called Windows Forms (WinForms, for short). The NX Block UI Styler has similar tools, and produces dialogs that are more consistent with the rest of NX, as explained in chapter 13 and chapter 14. But WinForms are more flexible, and you may find them useful in some cases. Designing WinForm-based user interfaces is actually much easier if you use an IDE like Visual Studio, and we will see how to do this in the next chapter. For now, we will create a very simple WinForm, to show the basic concepts.

Copy and Paste the following code into the file SnapSample.vb:

```
Sub Main
  myForm = New Snap.UI.WinForm()
  myForm.BackColor = System.Drawing.Color.Red
  myForm.Opacity = 0.5
  myForm.Text = "Hi there"
  myForm.ShowDialog()
End Sub
```

When you run this application, you should see a WinForm appear, like this:



The WinForm is pretty boring, but it does have all the standard Windows functionality — you can move it around, resize it, minimize it, and so on, in the usual way. Since we called Snap.UI.WinForm, we got a special NX-style WinForm, not a generic one. It has the NX icon in its top left corner, which will help the user understand that it's associated with NX. Also, the main NX Window is the "parent" of our new form, which means that our form will be minimized and restored along with the NX window, and will never get hidden underneath it. Actually, in the current scenario, our form is "modal", which means that you have to close it before you do anything with the NX window, so the parenting arrangement doesn't have much value. We got this modal behavior because we called myForm.ShowDialog to display our form. There is also myForm.Show, which creates a non-modal form, but this doesn't work in the Journal Editor.

The next few lines of code adjust various properties of the form — we give it a red color, make it 50% transparent, and put the words "Hi there" in its title bar. There are dozens of properties that influence the appearance and behavior of a WinForm, but it's best to wait until the next chapter to explore these, because it's very easy using Visual Studio.

To stop your code running and get back to the Journal Editor, you need to close the WinForm. You do this in the usual way — click on the "X" in the top right corner.

Next, let's add a button to our WinForm. Modify the code in SnapSample.vb as follows:

```
Option Explicit Off
Imports Snap, Snap.Create
Imports System, System.Windows.Forms, System.Drawing.Color

Module SnapSample
  Dim WithEvents myButton As Button       'A variable to hold a button
  Dim rand As Random                      'A variable to hold a random number generator

  Sub Main()
    rand = New Random()                         'Create a random number generator
    myForm = New Snap.UI.WinForm()              'Create a Windows form
    myForm.Text = "Create Random Spheres"
    myButton = New Button()                     'Create a button
    myButton.BackColor = Yellow                 'Color it yellow
    myButton.Text = "Click me"                  'Put some text on it
    myForm.Controls.Add(myButton)               'Add it to our form
    myForm.ShowDialog()                         'Display our form
  End Sub

End Module
```

First, note that we have added another line of "Imports" statements at the top of the file. These allow us to abbreviate the names in our code. So, for example, we can refer to Yellow instead of the full name System.Drawing.Color.Yellow, and we can refer to Random instead of System.Random.

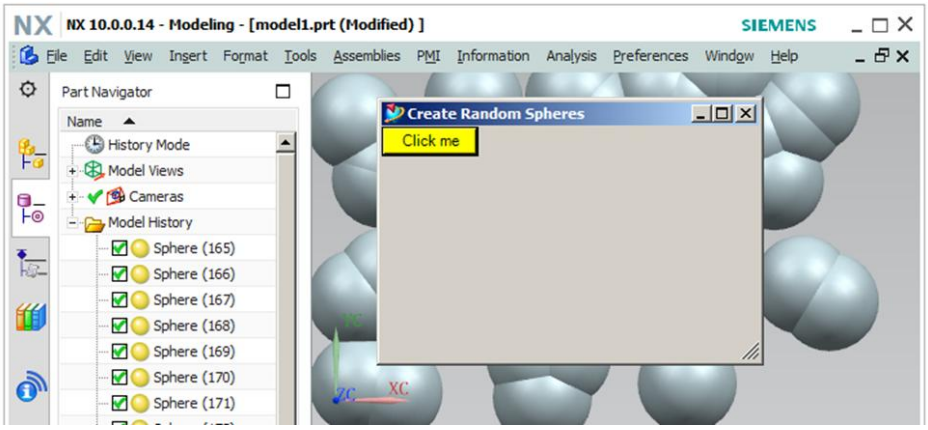
As you can see, we used the "New" keyword when creating the random number generator, the form, and the button. We have never used "New" when creating NX objects like points and splines, and you may be wondering why these two types of objects get treated differently. The answer is given in chapter 5, in the section entitled "Constructors vs. Static Functions". Don't worry about it for now — just accept that the "New" keyword isn't needed when you're creating NX objects. Or, if the curiosity is overwhelming, you can read about this topic in chapter 5.

Try running this code. You will see that the form is displayed, but nothing happens if you click on the yellow button. To change this, place the following code down near the bottom, just before the line that says "End Module".

```
Sub Handler(ByVal sender As Object, ByVal e As EventArgs) Handles myButton.Click
  x = rand.NextDouble()         'Get a random x-coordinate
  y = rand.NextDouble()         'Get a random y-coordinate
  Sphere(x, y, 0, 0.2)          'Create a sphere at (x,y,0) with diameter 0.2
End Sub
```

This is a new "subroutine" (denoted by the keyword "Sub"). So, now we have two subroutines — one called "Main" and one called "Handler". This is a new situation, for us, but it's a fairly typical one — as your code gets longer, it's easier to understand if you break it up into several subroutines.

The new function is an event handler for the "click" event of the yellow button. In other words, this code gets executed whenever you click on the yellow button in the form. As you can see, every time you click the button, the code will create a randomly-located sphere.



Designing buttons and writing event handlers is much easier in Visual Studio, as we will see in the next chapter.

What Next ?

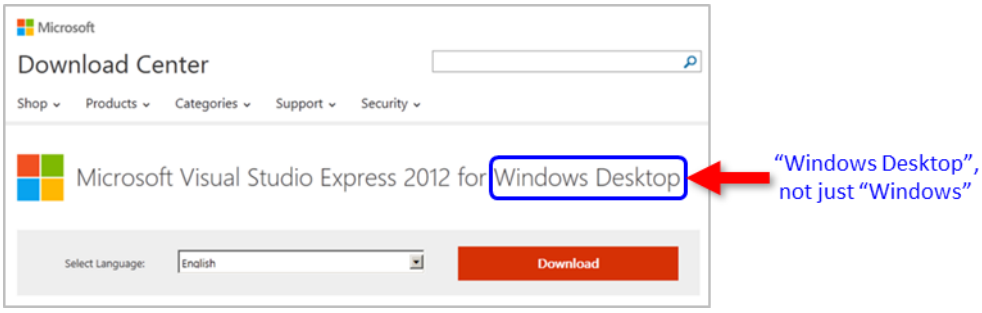
The examples in this chapter have given you a brief glimpse at some of the things you can do with SNAP. Using the Journal Editor, we were able to start programming immediately, and we saw that SNAP allows us to build simple user interfaces, do calculations, and create NX geometry. If you liked what you saw in this chapter, you'll probably like the next one, too. It shows you some further examples of SNAP capabilities, and also some much easier and more pleasant ways to write code.

Chapter 3: Using Visual Studio Express

In the previous chapter, we developed code using the NX Journal Editor. This is a convenient starting point, since it requires no setup, but it is really a fairly primitive environment. Except for very short programs, it is far better to use a more powerful "integrated development environment" (IDE). The Microsoft Visual Studio "Express" editions are free-ware single-language lightweight versions of the Microsoft Visual Studio IDE used by many professional programmers. The idea, according to Microsoft, is to provide streamlined, easy-to-use IDEs for less serious users, such as hobbyists, students, and people like you. Express Editions are available for the Visual Basic, C#, and C++ programming languages. In this chapter, we will be focusing on the Visual Basic 2012 Express package.

Installing Visual Studio

If you already have some version of Visual Studio 2012 installed on your computer, and you are familiar with it, you can skip this section and proceed directly to the first example. If not, then the first step is to install Visual Studio 2012 Express for Windows Desktop, which you can download from here, or several other places.



If you can't find the web page (because the Microsoft folks have moved it again), just search the internet for "Visual Studio 2012 Express". Make sure you get the "for Windows Desktop" version. A common mistake is to download the "for Windows" version, instead, but this is for building Windows store apps, so it's not what we want. Follow the instructions to download the package and complete the installation. After you're done, you should see Microsoft Visual Studio 2012 Express on your Programs menu, and you should see a folder called Visual Studio 2012 in your My Documents folder. If you run into trouble, it might help to watch this video.

Visual Studio 2013 Express should work, too, though it is not officially supported with NX 10. Older versions of Visual Studio will not work because they don't allow you to use version 4.5 of the .NET Framework.

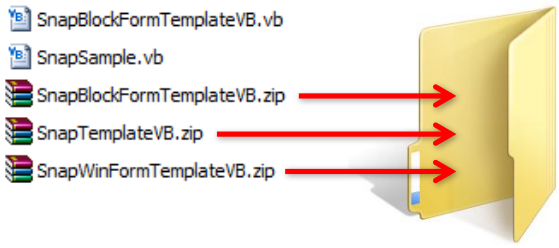
Unfortunately, the Visual Studio Express download is much larger than it was when SNAP was first conceived — it has grown from around 80 MB to over 600MB. If you don't have the patience or disk space to handle a package this large, you can try the SharpDevelop IDE, instead. It's only around 15MB, and provides everything you need. The instructions you read in this document won't match SharpDevelop exactly, but it should be fairly easy to adapt.

In the examples in this chapter, we'll provide step-by-step instructions for writing the code, just as we did in chapter 2, so it should be easy to follow. But if you'd like to get some additional information about the Visual Basic language or Visual Studio, then one good place to start is this series of videos. There is a huge amount of other tutorial material available on the internet, and you might find other sources preferable, especially if your native language is not English.

Installing SNAP Templates

After installing Visual Studio, you should install three custom templates that we will be using as convenient starting points when developing SNAP programs. You will find three zip files in [...NX]\UGOPEN\SNAP\Templates. Again, remember that [...NX] is just shorthand for the location where NX is installed, which is typically somewhere like C:\Program Files\Siemens\NX 10. The names of the files are SnapTemplateVB.zip, SnapBlockFormTemplateVB.zip, and SnapWinFormTemplateVB.zip. Copy these three zip files into the folder [My Documents]\Visual Studio 2012\Templates\ProjectTemplates\Visual Basic.

For added clarity, here are the same instructions in pictorial form:




```
[My Documents]
  Visual Studio 2012
    Templates
      ProjectTemplates
        Visual Basic
```

Unfortunately, experience has shown that people often do this step wrong, so we're going to yell at you ...

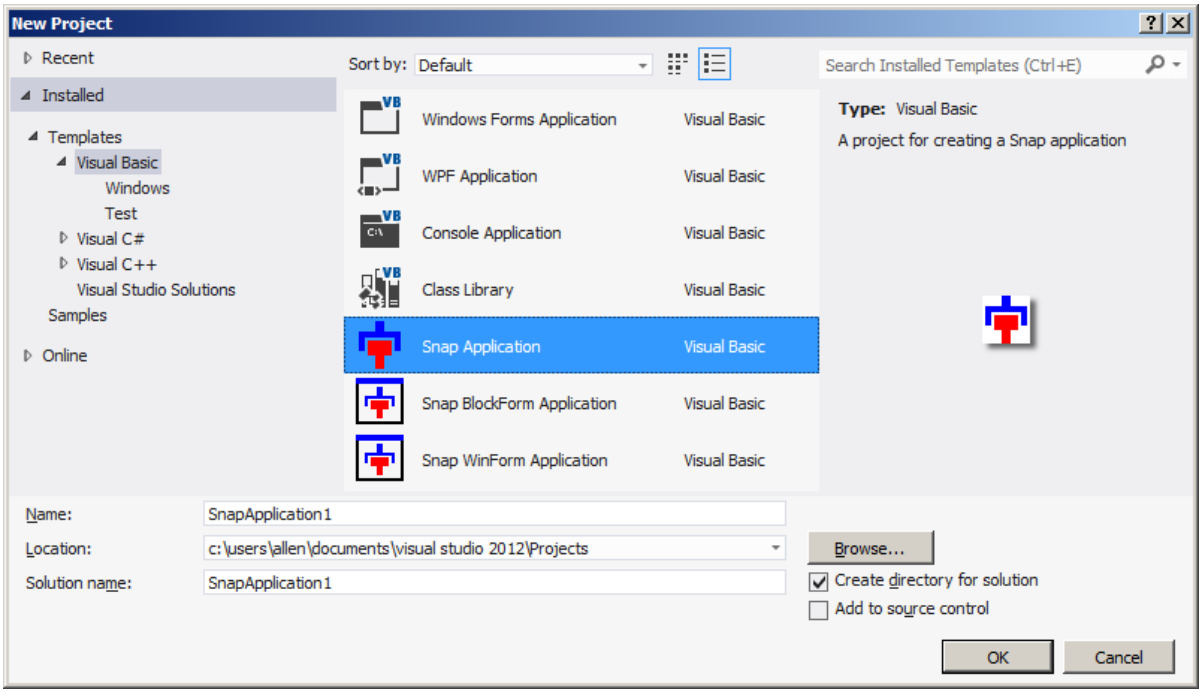
NOTE: please **do not** extract the contents from the zip files; just **copy the zip files themselves**.

Licensing Issues Again

As mentioned in the previous chapter, we provide a free scaled-down version of SNAP called MiniSNAP. The capabilities of MiniSNAP are quite limited, but it does have enough functions to let you work through the examples in this chapter and the previous one. As you saw in the previous chapter, you can run code that calls MiniSNAP functions in the NX Journal Editor, even if you don't have a SNAP authoring license. In this chapter, we will be compiling our code within Visual Studio to produce DLLs. If this code calls MiniSNAP functions, then, again, it will still work even if you don't have a SNAP authoring license However, as in the previous chapter, you will need to change the code to import "MiniSnap", rather than "Snap".

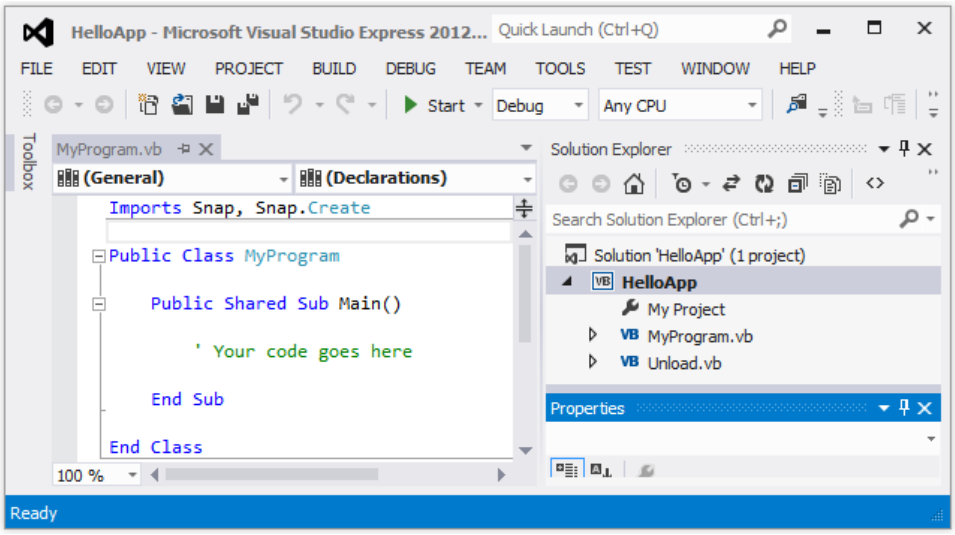
Example 1: Hello World Again

Our first exercise is to create a "Hello World" application again. Sorry, we know it's boring, but it's a tradition. After you get Visual Studio Express installed and running, choose New Project from the File menu. A "project" is the name Visual Studio uses for a collection of related files. You will see a list of available project templates

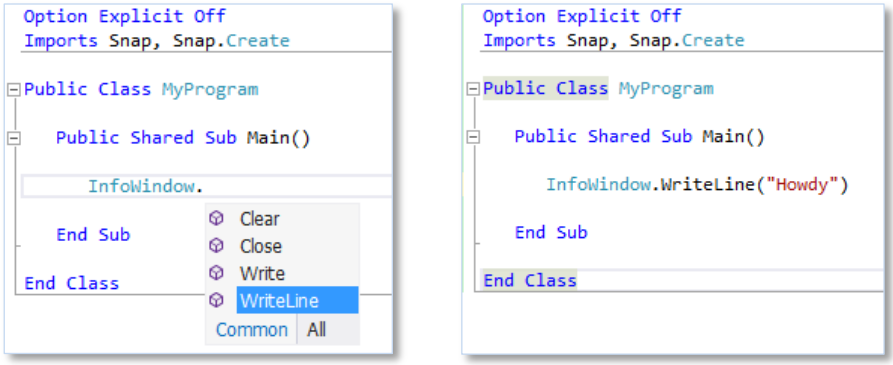


Choose the "Snap Application" template. This is a special custom template designed to serve as a convenient starting point for certain kinds of SNAP applications. Also, give your project a suitable name — something like "HelloApp" would be good.

The Snap Application template gives you a framework for a simple SNAP application, as shown here:

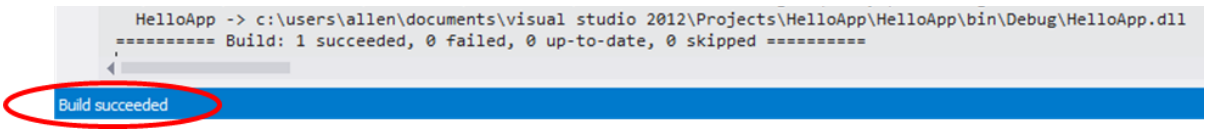


In the left-hand pane, you can see some familiar VB code, which the template has placed in a file called MyProgram.vb for you. We need to make a couple of changes to this code: add `Option Explicit Off` at the top, and add a line that outputs some message to the listing window, as shown here:

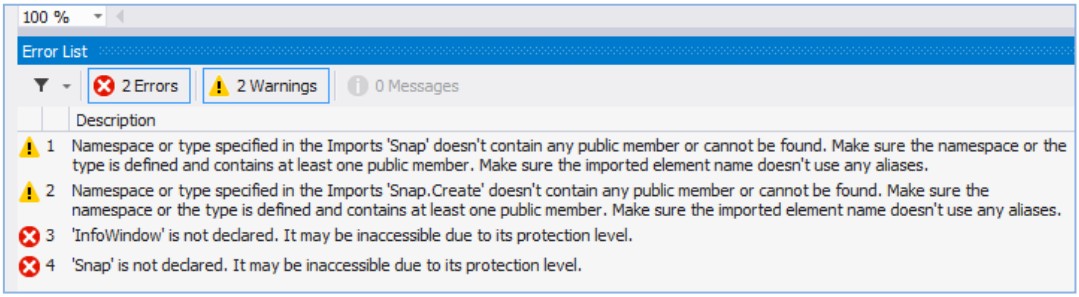


You should type the new code, rather than just copying and pasting it, because some interesting things happen as you type (as you saw in the tutorial videos, if you watched them). In fact, it's interesting to type the entire 7 lines of code. You will find that you actually only have to type 5 lines — Visual Studio will type the other two for you. Generally, Visual Studio helps you by suggesting alternatives, completing words, correcting mistakes, showing you documentation, and so on. To accept the highlighted alternative, you can either press Tab, or type another character, like a period or a parenthesis. All of this is called "Intellisense" by Microsoft's marketeers. Despite its dubious name, you'll find it very helpful as your programming activities progress. Also, notice that Visual Studio automatically makes comments green, literal text red, and language keywords blue, to help you distinguish them.

Next, you are ready to compile (or "build") your code into an executable application. To do this, go to the Debug menu and choose Build HelloApp, or press Ctrl+Shift+B, which will send your code to the VB compiler. The compiler will translate your code into an executable form that your computer can run, and will store this in a file called HelloApp.dll. The extension "dll" stands for "Dynamic Link Library", which is a type of file that holds executable code. You should get the good news about the build succeeding down at the bottom left:



On the other hand, if you're unlucky, you might get some error messages like these:

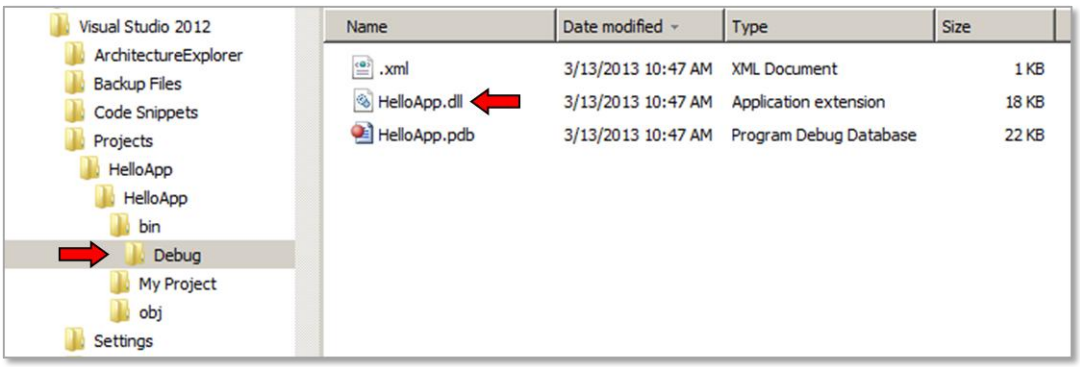


It's not very likely that this problem will occur, so we don't want to interrupt the flow by discussing all the details here. The possible causes and corrective actions are described in chapter 16.

At some point, you should save your project by choosing Save All from the File menu. Visual Studio will offer to save in your Projects folder, whose path is typically something like [My Documents]\Visual Studio 2010\Projects.

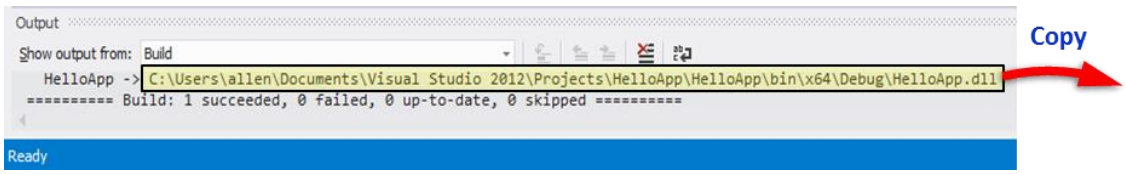
Now, we are ready to run our new application. From within NX, choose File→Execute→NX Open (or press Ctrl+U). Your version of the NX user interface might not have the Execute option installed in the File menu, but the Ctrl+U shortcut will work anyway.

A dialog will appear that allows you to find your executable. As mentioned earlier, it will be called HelloApp.dll, and it will be located in [My Documents]\Visual Studio 2012\Projects\HelloApp\HelloApp\bin\Debug along with two other files that you don't need to worry about.



To see HelloApp.dll, make sure you set the "Files of type" filter in the NX dialog to "Dynamic Loadable Libraries (*.dll)". Double-click on HelloApp.dll, and a friendly greeting should appear in your NX Listing window. If you can't find your application, try looking in the bin\Release folder, rather than the bin\Debug folder. If you still can't find it, it's probably because you forgot to save it, or you didn't set the file type filter correctly.

There's a useful trick that allows you to locate your executable quickly. When you build the application, some text like this will appear in the "Output" pane at the bottom of your Visual Studio window:



If the output pane is not visible, press Ctrl+Alt+O to display it (that's the letter O, not the number zero). You can then just copy the pathname of the newly-created application (highlighted in yellow above) and paste it into the "Execute" dialog within NX. This technique is highly recommended — it avoids all the hunting around folders that we described above, and it ensures that you are running the code that you just built. You only have to do this once per NX session, because NX will remember the location for you.

If you don't have a SNAP Author license, you will need to change "Snap" to "MiniSnap" in the first line of code, as you did in chapter 2.

Example 2: Declaring Variables

This example is a variation on Example 4 from the previous chapter — we will do some vector calculations to compute the radius of a circle through three points. But this time we will declare the variables we use, to see how this affects things.

If your previous project is still open in Visual Studio, close it by choosing File→Close Project. Then choose New Project from the File menu, use the Snap Application template to create a project, and give it the name ThreePointRadius, or something like that.

As before, add the line `Option Explicit Off` at the top of the file. For reasons explained below, this is the last time we're going to do this in our examples.

Then, replace the line "`Your code goes here`" with the following code

```
p1 = Snap.UI.Input.GetPosition("Specify first point")    ' Get first point from user
p2 = Snap.UI.Input.GetPosition("Specify second point")   ' Get second point
p3 = Snap.UI.Input.GetPosition("Specify third point")    ' Get third point

u = p2.Position - p1.Position         ' Vector from p1 to p2
v = p3.Position - p1.Position         ' Vector from p1 to p3
uu = u * u                            ' Dot product of vectors
uv = u * v
vv = v * v
det = uu * vv - uv * uv                    ' Determinant for solving linear equations
alpha = (uu * vv - uv * vv) / (2 * det)    ' Bad code !! Should check that det is not zero
beta  = (uu * vv - uu * uv) / (2 * det)
rvec = alpha * u + beta * v                ' Radius vector
radius = Vector.Norm(rvec)                 ' Radius is length (norm) of this vector

InfoWindow.WriteLine(radius)               ' Output to listing window
```

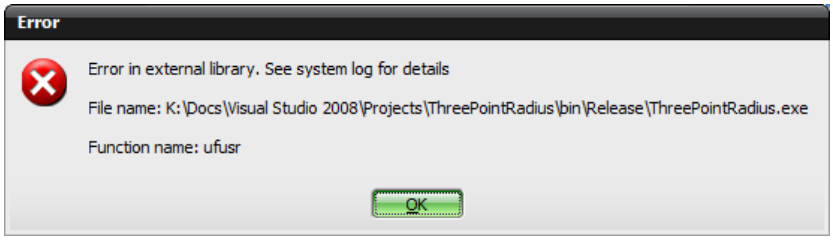
Again, you can gain some experience with Intellisense if you type this code, rather than copying and pasting it. The only thing that's new here is the function GetPosition, which allows you to get a point location from the user by means of the usual NX Point Subfunction.

As before, you can save this project, build it, and run it from within NX using File→Execute→NX Open (or Ctrl+U).

Now let's see what happens if you make a typing error. Change the line that calculates "det" to read

```
det = uu * vv - uv * u
```

In other words, change the last term from "uv" to "u". Then build the project and try running it again. It will still build successfully, but when you run it from within NX, you'll get an error message like this:



If you choose Help→Log File from within NX, and hunt around the NX System Log, you will find some more error messages about 50 lines from the bottom, most notably these ones

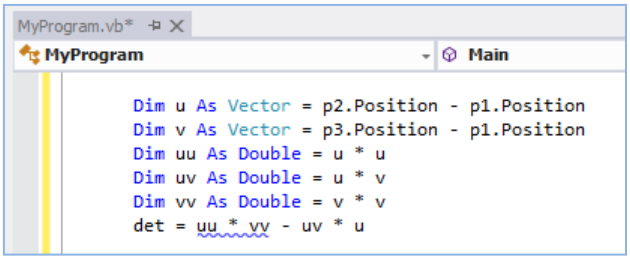
```
+++ Overload resolution failed because no Public '-' can be called with these arguments:
Argument matching parameter 'u' cannot convert from 'Double' to 'Vector'.
```

Obviously it would be much better to discover errors like this earlier, as you're writing the code, rather than when you run the application. And, in fact, you can, if you change the way you write the code, and give the compiler a little more information. The key is a process called "declaring" variables, which lets us tell the compiler about their types.

To see how this works, change your code to read:

```
Dim u As Vector = p2.Position - p1.Position
Dim v As Vector = p3.Position - p1.Position
Dim uu As Double = u * u
Dim uv As Double = u * v
Dim vv As Double = v * v
```

The phrase "`Dim u As Vector`" tells the compiler that the variable `u` is supposed to hold a Vector, and so on. So, the compiler now knows that `u` and `v` are vectors, and `uu`, `uv`, and `vv` are numbers (doubles). So `uv*u` is a vector, and the expression `uu*vv – uv*u` is trying to subtract a vector from a number, which obviously doesn't make sense. So we get a "squiggly underline" error indicator, and we know immediately that we have made a mistake. And, if you hover your mouse over the mistake, a message will appear telling you what you did wrong:



Up until now, our applications have been very simple, so there was not much justification for the extra effort of declaring variables. But, as you start to write more complex applications, you will definitely want the compiler to help you find your mistakes. And it can do this very effectively if you declare your variables. Actually, many programming languages require you to declare all variables. Visual Basic is an exception — if you use the "`Option Explicit Off`" directive at the start of your code, as we have been doing, then you don't have to. But declaring variables is a good thing, so we're going to do it from now on. For further discussion of declaring variables (and avoiding or shortening declarations), please see chapter 4.

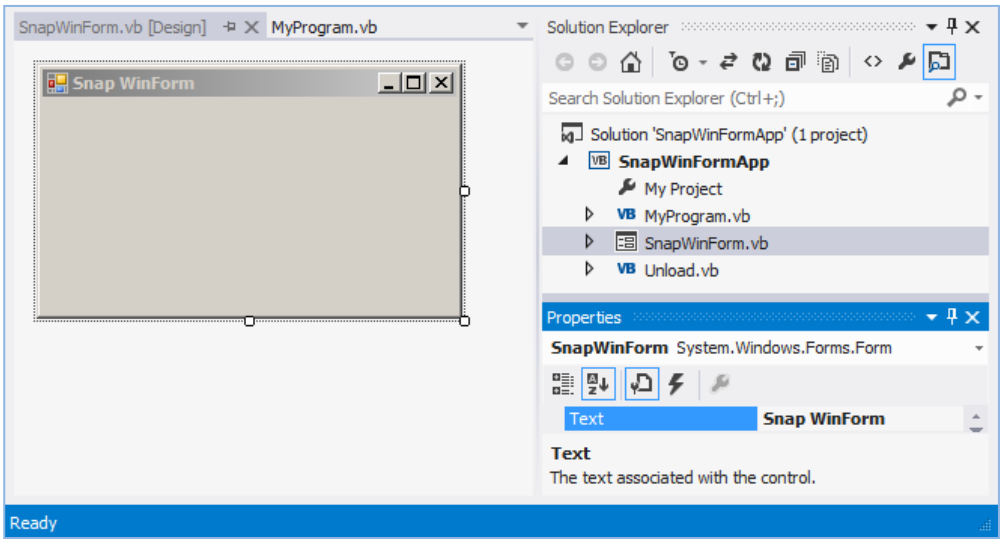
Again, remember that if you don't have a SNAP Author license, you will need to change "Snap" to "MiniSnap" in the first line of code, as you did in chapter 2.

Example 3: WinForms Again

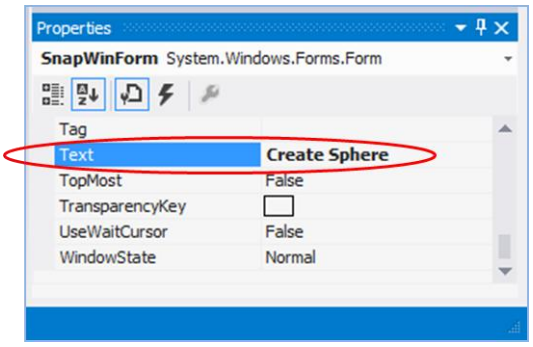
One of the nice things about Visual Studio is the set of tools it provides for designing user interface dialogs using Windows Forms (WinForms, for short). We're going to recreate the "Create Random Spheres" dialog from the previous chapter, but it will be much easier this time, using Visual Studio, and the dialog will look nicer.

Run Visual Studio Express, and choose New Project from the File menu. Instead of choosing the Snap Application template, chose the Snap WinForm Application template this time. Call your new project SnapWinFormApp.

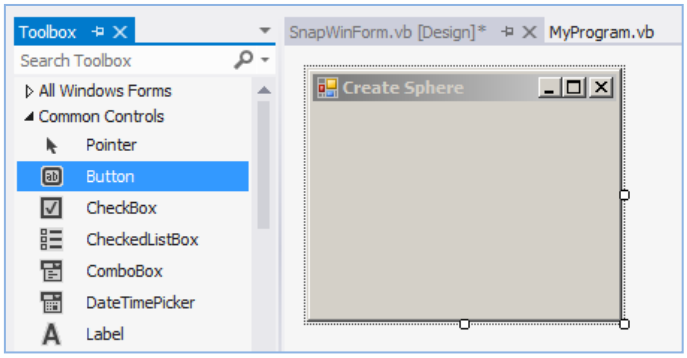
Your new project will look something like this:



You may need to double-click on SnapWinForm.vb to see the new Windows form in the left-hand pane. In the lower right-hand pane, all the "properties" of the new WinForm are listed, along with their values. As you can see, the form has a property called "Text", and this property currently has the value "Snap WinForm". This property actually represents the text in the title bar of the dialog. Edit this text to read "Create Sphere". When you do this, you will see that the dialog title bar changes, too.

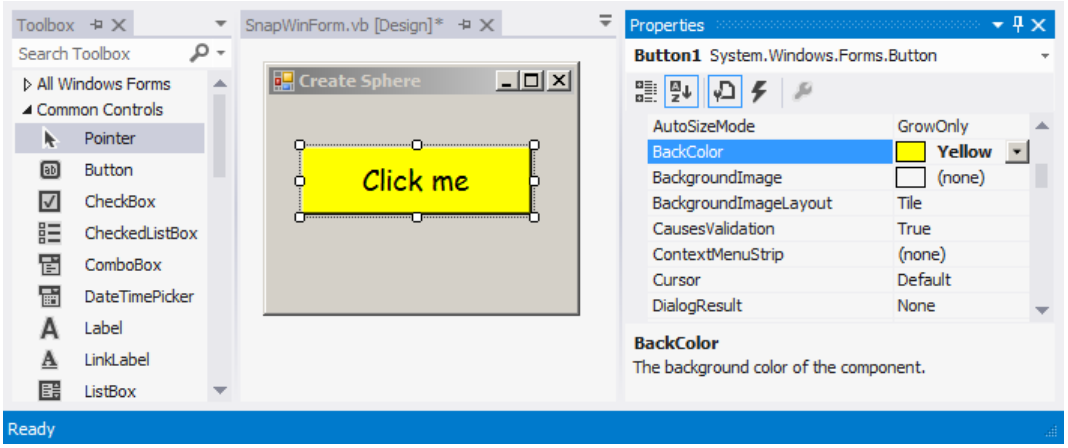


Next, as before, we're going to add a button to our form. On the left-hand side of the Visual Studio window, you should see a Toolbox containing various types of user interface objects. If you don't see the Toolbox, choose it from the View menu, or press Ctrl+Alt+X.



Click on the "Button" object. The cursor will change to a small "+" sign, and you can then use it to graphically draw a button on the form. Initially, the button will be labeled with the text "Button1", but you can change this to "Click me" or whatever you want by editing the text property of the button, just as we edited the text property of the form.

You can edit other properties of the button, too, like the font used and the background color. Your result might be something like this:



Also, you can adjust the sizes of the button and the form by dragging on their handles:

Next, let's make the button do something useful. Double-click the button, and a code window will appear, like this:

```
Imports Snap, Snap.Create

Public Class SnapWinForm

  Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click

  End Sub
End Class
```

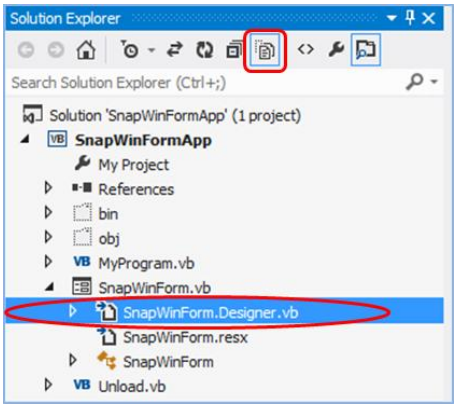
The function you see is an event handler for the button's "click" event. Currently, it doesn't do anything, but you can edit it as shown below to make the click event create a sphere, or whatever else you want it to do.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
  InfoWindow.WriteLine("Creating a sphere")
  Sphere(0, 0, 0, 10)
End Sub
```

When we created this dialog manually, in the previous chapter, you may recall that we wrote code like this:

```
myForm.Text = "Create Random Spheres"
myButton = New Button()                    'Create a button
myButton.BackColor = Color.Yellow          'Color it yellow
myButton.Text = "Click me"                 'Put some text on it
myForm.Controls.Add(myButton)              'Add it to our form
```

This same sort of code exists in our current project, too, but it was written for us by Visual Studio, and it's somewhat hidden, because you're not supposed to edit it. To see this code, click on the Show All Files button at the top of the Solution Explorer window, and then double-click on the file named SnapWinForm.Designer.vb.



To display our dialog, we have a couple of lines of code in Sub Main in the file MyProgram.vb:

```
Public Shared Sub Main()
  Dim form As New SnapWinForm()
  form.ShowDialog()
End Sub
```

As before, we're using form.ShowDialog to display the dialog, so it will be "modal", which means that we can't do anything else until we close the form. There is also myForm.Show, which creates a non-modal form, but to use this, you have to change the GetUnloadOption function in the file Unload.vb. Specifically, you have to modify this function to return Snap.UnloadOption.AtTermination instead of Snap.UnloadOption.Immediately. If you fail to do this, your dialog will disappear a second or two after it's displayed, so you'll probably never see it.

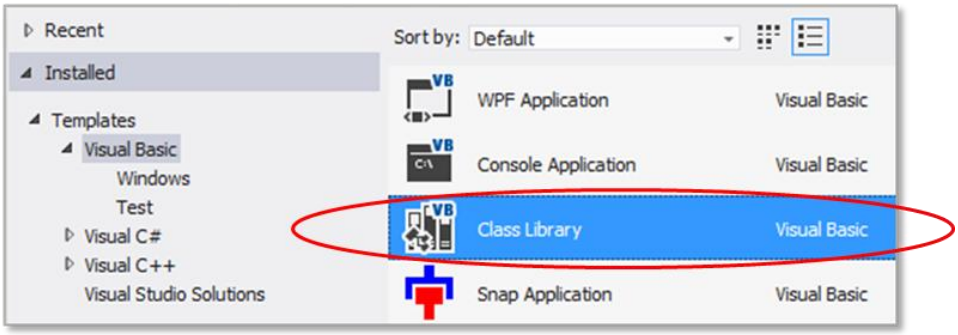
Build the project, and run it from within NX, as usual. When your dialog appears, you can click on your button to create spheres. When you get bored with this, click the "X" to close your dialog.

If you want to learn more about creation of WinForm-based user interfaces, there are many books and on-line tutorials available on the subject, including this series of videos.

Example 4: Hello World Yet Again (the Hard Way)

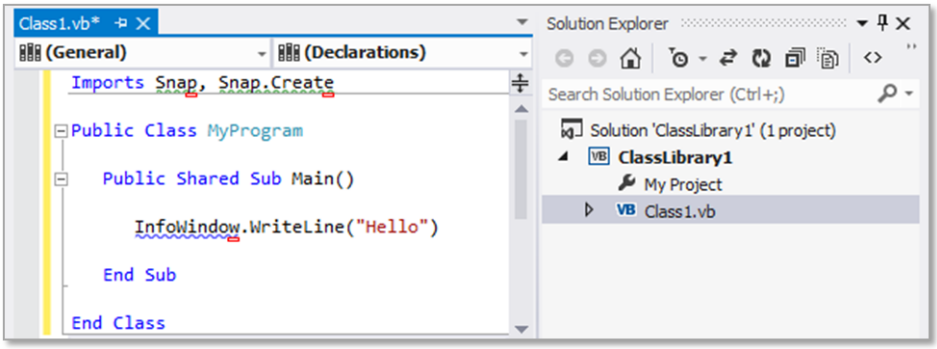
Sorry, but we're going to create a "Hello World" application yet again. This time, we're going to do it without getting any assistance from the SNAP template we used last time. This will help you understand what is happening "behind the scenes" so that you will know what to do if you run into problems later. If you're not interested in this, you can skip to the next example.

Run Visual Studio Express, and choose New Project from the File menu. You will see the available set of project templates. But, this time, instead of choosing the Snap Application template, choose the Class Library one:

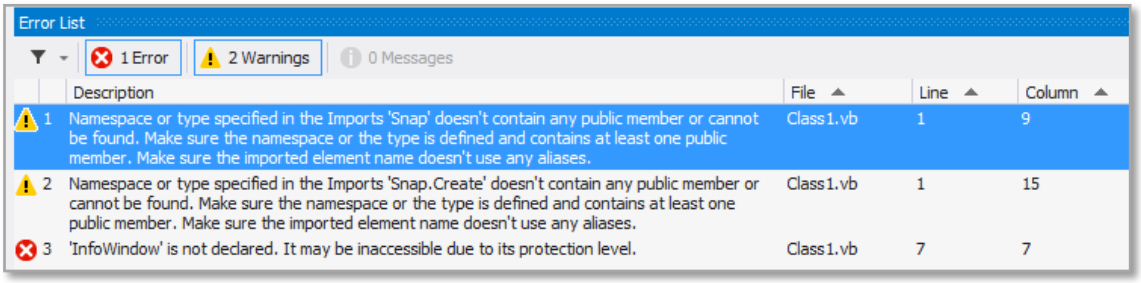


You might be thinking that you could use the "Console Application" template, instead. Unfortunately, there are some technical reasons why this will not work — on some systems, it will lead to a mysterious "failed to load image" error when you try to run your application from within NX. Please see chapter 17 for more details.

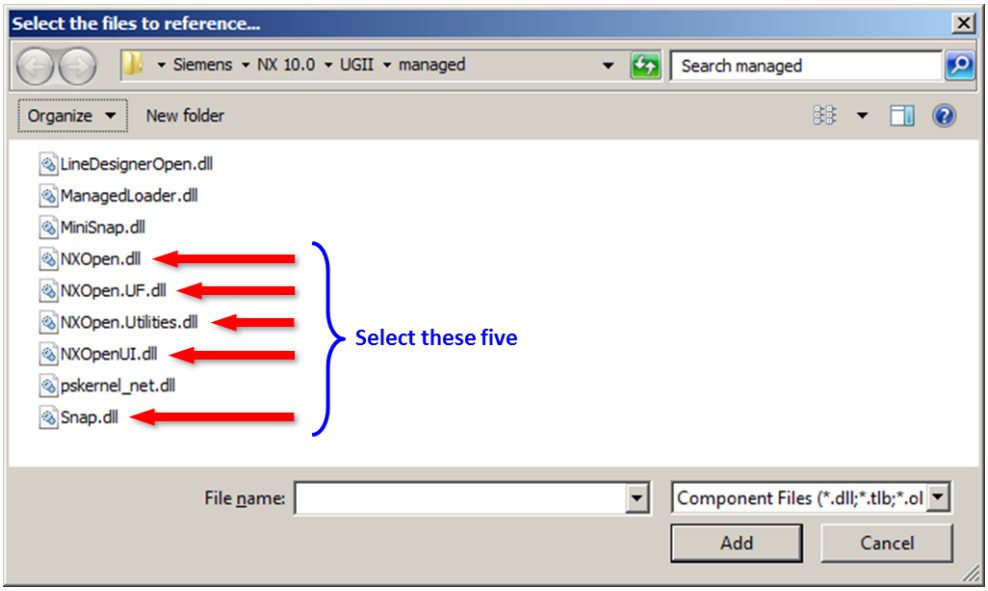
This Class Library template gives you a framework for a Visual Basic class definition. You will see a file called Class1.vb that contains a couple of lines of code. Delete this code and paste (or type) the contents of SnapSample.vb in its place. Delete the first line (the one that says `Option Explicit Off`). Also, delete the line that says `Your code goes here`, and replace it by `InfoWindow.WriteLine("Hello world")`, as we have done several times before. You should end up with something that looks like this:



As you're typing, you might notice that the usual "intellisense" doesn't work. This is the first indication that something is wrong. Also, you will see several squiggly underlines, and some error and warning messages in the list at the bottom of the window:



Most of the problems arise because our code is using the SNAP library, and this is not connected in any way to our current project. So the compiler doesn't know anything about Snap, Snap.Create, or the InfoWindow function. To fix this, we need to add a "reference" to the SNAP library. From the Project menu, choose Add Reference. In the dialog that appears, click on the Browse tab, and navigate to the folder [...NX]\UGII\managed:



You will see nine DLLs. We only need the SNAP DLL in this example, but quite often you'll need the NXOpen DLLs, too. It doesn't really hurt too much to include references that you don't actually need, so select five DLLs, as shown above, and click OK. Your project now has references to the SNAP and NX Open libraries, and this should remove the complaints about them "containing no public members". Now you can build and run the application, as usual.

The Snap Application template that we used previously already includes the references to the SNAP and NX Open libraries, so you didn't have to add them manually. But, it's useful to know how to do this when you need to. For example, to use some of the .NET Framework functions listed in Example 6 in the previous chapter, you may have to add references to the assemblies where they reside. If you forget to do this, you will get "type not defined" errors, like the ones we saw above. Please see chapter 17 for more information about problems with references.

A project based on the Class Library template has another deficiency — it doesn't include a GetUnloadOption function. This means that NX won't know how to "unload" your code after it has finished executing — in some sense, NX "holds onto" your code, and won't let it go. So, if you try to change your code and rebuild the project, you'll get an error message telling you that you "can't access the file because it is being used by another process". The other process is NX, and you'll have to terminate NX to get it to release its hold on your DLL so that you can rebuild it. The Snap Application Template provides a GetUnloadOption function for you, so you won't have these sorts of problems. Writing your own GetUnloadOption function is fairly simple. The code is as follows:

```
Public Function GetUnloadOption(ByVal dummy As String) As Integer
    Return Snap.UnloadOption.Immediately
End Function
```

You have to place this code in the same class or module as your "Main" function — in our case, this means inside the SnapSample module. So, you just need to paste this code immediately before the line that says "`End Module`". Please look up GetUnloadOption in the SNAP Reference Guide for more information about unloading code.

Example 5: Toolpath Simulation

Our next example simulates the positioning of a tool on a surface. As before, run Visual Studio Express, and create a new project using the Snap Application template. Edit the code to read as follows:

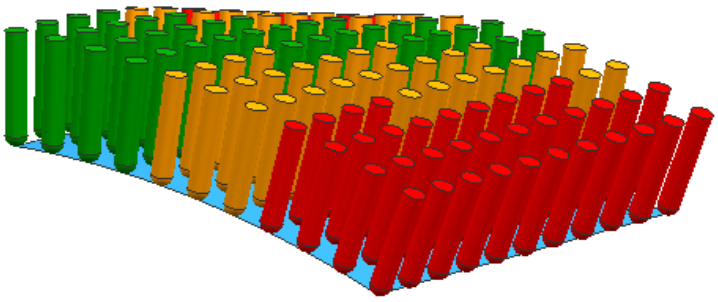
```
Imports Snap, Snap.Create
Imports System.Drawing.Color

Public Partial Class MyProgram

    Public Shared Sub Main()

      Dim p As Position(,) = New Position(2,2) {}
      Dim h As double = 0.4
      p(0,0) = {0,0,0}  :  p(0,1) = {0,1,0}  :  p(0,2) = {0,2,0}
      p(1,0) = {1,0,h}  :  p(1,1) = {1,1,h}  :  p(1,2) = {1,2,h}
      p(2,0) = {2,0,0}  :  p(2,1) = {2,1,h}  :  p(2,2) = {2,2,h}
      Dim patchBody As NX.Body = BezierPatch(p)
      Dim face As NX.Face = patchBody.Faces(0)

      Dim nu As Integer = 10  :     Dim uStep As Double = 1.0/nu
      Dim nv As Integer = 10  :     Dim vStep As Double = 1.0/nv

      Dim diameter = 0.1    ' Tool diameter
      Dim length   = 0.5    ' Tool length

      Dim u, v As Double
      Dim point As Position
      Dim axis As Vector

      For i As Integer = 0 To nu
        For j As Integer = 0 To nv
          u = i*uStep
          v = j*vStep
          point = face.Position(u,v)                ' Point on surface
          axis  = face.Normal(u,v)                  ' Surface normal = tool axis
          ShowTool(diameter, length, point, axis)    ' Display the tool at this position
        Next j
      Next i

End Sub

    Shared Sub ShowTool(diameter As Double, length As Double, point As Position, axis As Vector)
      Dim toolCenter As Position = point + 0.5* diameter*axis
      Dim toolSphere As NX.Body = Sphere(toolCenter, diameter)
      Dim toolShaft As NX.Body = Cylinder(toolCenter, axis, length, diameter)
      Dim tool As NX.Body = Unite(toolSphere, toolShaft)
      tool.Color = Green
      Dim angle As Double = Vector.Angle(axis, Vector.AxisZ)
      If angle >  8 Then tool.Color = Orange
      If angle > 15 Then tool.Color = Red
    End Sub

End Class
```

The key calculations here were performed by the `face.Position` and `face.Normal` functions, which are explained further in chapter 9.

Note that we are using functions from the System.Drawing class, so we will need to add a reference to its assembly in order for this to work. To do this, the process is similar to the one we used in Example 4: from the Project menu, choose Add Reference, click on the .NET tab, and find System.Drawing in the long list of assemblies that appears. Once you have done this, you should be able to build your project successfully.

The first part of the code is just defining a surface. In a real application, you would probably ask the user to select the surface, instead of creating it within your code. Then there are two For loops that step across the surface, calculating position and surface normal at each point, and creating/displaying the tool by calling ShowTool.

The ShowTool function constructs the tool by uniting a sphere and a cylinder. It also checks the inclination of the tool from vertical. If the inclination is somewhat large (greater than 8 degrees) the tool is colored orange, and if it's very large (greater than 15 degrees), the tool is colored red. Typical results are shown here:

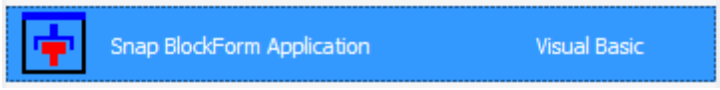


Example 6: A BlockForm User Interface

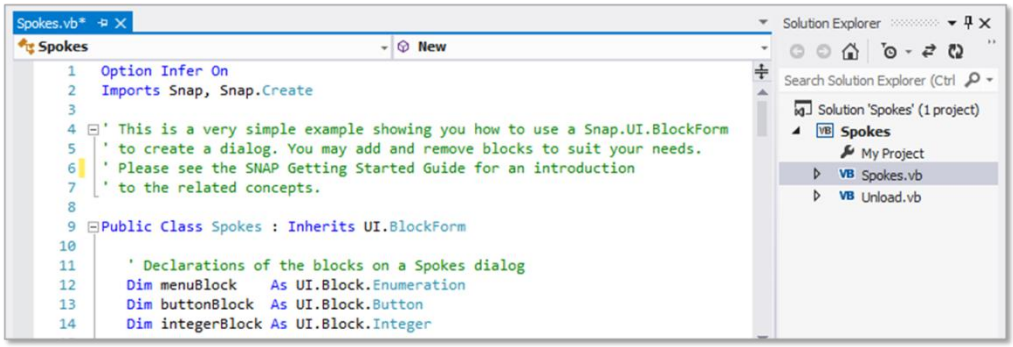
As we saw earlier, you can use WinForms to construct rich graphical user interfaces for your applications. But sometimes WinForm-based user interfaces don't look and behave like the rest of NX, so they can seem out of place. If we want more NX-like appearance and behavior, we should use an NX "block-based" user interface, instead. Block-based user interfaces are discussed in detail in chapter 13 and chapter 14, so this example will just provide a quick introduction.

This example is different from all the previous ones because it requires a SNAP authoring license. Using MiniSNAP won't work because it doesn't support block-based dialogs.

In Visual Studio, choose "New Project". Choose the Snap BlockForm Application template



and type in a name down at the bottom of the dialog. Please use the name "Spokes", because this will make it easier to follow the descriptions below. After you click OK, Visual Studio will create a new project containing two files. One is the usual "Unload" file, which contains nothing new or interesting. The other is a file called Spokes.vb, which contains the skeleton of a BlockForm-based application, as shown here:



Note that we have turned on line numbers using Tools → Options → Text Editor → All Languages → General, to make it easier to find the things referred to in the instructions below.

The first task is an easy one – you just have to delete a few things:

- Near the end of the file, you will see five functions called `OnShow`, `OnOK`, `OnApply`, `OnCancel`, and `OnUpdate`. Delete all of these except `OnApply`.
- Delete the lines that mention `buttonBlock` (lines 30, 31, and 13, in that order).
- Delete the "instruction" comments. This is optional, but doing it will make the code tidier.

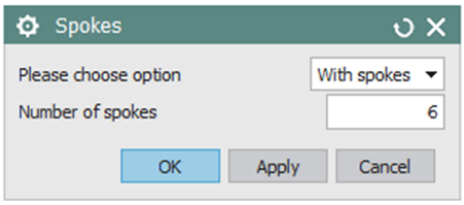
Change the code within the constructor (the `Sub New` function) to read as follows. Or, instead of editing, you can just copy/paste the code below, of course.

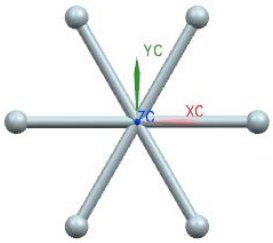
```
' Constructor for a Spokes dialog object
Public Sub New()

    Me.Title = "Spokes"
    Me.Cue = "Please enter information"

    ' Create an option menu block
    menuBlock = New UI.Block.Enumeration()
    menuBlock.Label = "Please choose option"
    menuBlock.Items = {"Balls only", "With spokes"}

    ' Create an Integer block
    integerBlock = New UI.Block.Integer("Number of spokes", 6)

    ' Add all the blocks to the BlockForm
    Me.AddBlocks(menuBlock, integerBlock)

End Sub
```

The `OnApply` function will get called when the user clicks the Apply button on our dialog, so let's put some code in there, so that the Apply button does something interesting. Change the `OnApply` function to read as follows:

```
Public Overrides Sub OnApply()
   Dim n = integerBlock.Value
   Dim delta = 360.0/n
   For i = 0 To n-1
      Dim x = Snap.Math.CosD(i*delta)
      Dim y = Snap.Math.SinD(i*delta)
      Sphere( {x,y,0}, 0.05 )
      If menuBlock.SelectedItem = "With spokes" Then
         Cylinder( {0,0,0}, {x,y,0}, 0.02 )
      End If
   Next
End Sub
```

Build the project, and run it from within NX. The following dialog should appear, and clicking the Apply button or the OK button should produce a design like the one shown on the right:





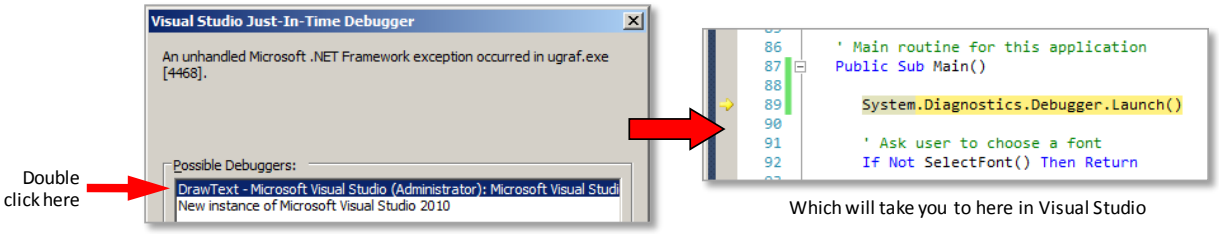
If you're interested in knowing more about Block-Based dialogs, please read chapter 13 and chapter 14.

Debugging in Visual Studio

The full version of Visual Studio (but not the Express edition) provides an excellent debugger that lets you step through your code one line at a time, watching what's happening as it executes. In particular, you can set "breakpoints" that pause the execution of your code, allowing you to examine variable values. This is a very good way to find problems, obviously. The techniques used with SNAP and NX Open programs are a little unusual because you are debugging code called by a "Main" function that you don't have access to (because it's inside NX). This means that using the normal "Start Debugging" command within Visual Studio is not appropriate. There are two alternative approaches, as outlined below, but neither of these is available in Visual Studio Express editions.

Using Debugger.Launch

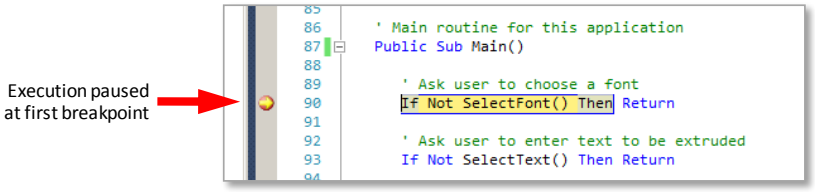
First, you write `System.Diagnostics.Debugger.Launch()` somewhere near the beginning of your code, and then you run your application in the normal way using File→Execute→NX Open. When execution reaches the `Debugger.Launch` call, the Just-In-Time Debugger dialog will appear, asking you which debugger you want to use:



Double-click on the debugger for your current project, as shown in the picture above, and you will be taken back to Visual Studio with your code "paused" at the `Debugger.Launch()` line, ready to begin stepping through it.

Using Attach To Process

Within Visual Studio, choose Tools→Attach to Process (or press Ctrl+Alt+P), and double-click on the NX process (ugraf.exe) in the list of available processes. Again, run your application using File→Execute→NX Open, and you will arrive back in Visual Studio with your code "paused" at the first breakpoint.



Regardless of which of the two approaches you used, you are now ready to step through your code. The available options are shown in the Debug menu or on the Debug Toolbar within Visual Studio. For information on how to use the debugger facilities, please consult one of the many tutorials available on the internet.

Chapter 4: The Visual Basic Language

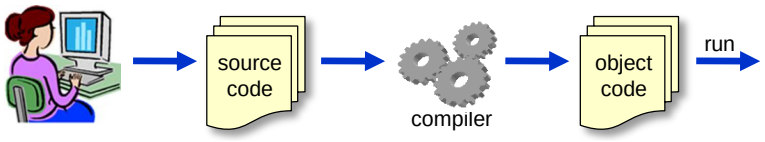
One of the strengths of NX Open and SNAP is that they are based on standard mainstream programming languages. This means there are many excellent tools you can use (like Visual Studio), and there's lots of tutorial and help material available. This chapter provides an introduction to the Visual Basic language (which we have been using for all of our examples). There are many places where you can learn more about Visual Basic (like this series of videos, for example), so our description here will be very brief.

When looking for books and on-line tutorials, you should be aware that the Visual Basic language has evolved significantly over the years. What we are using here is Visual Basic for .NET. Older versions (like Visual Basic 6, for example), are quite different. So, when you start reading, make sure you are using fairly modern materials. If you really want the complete story, you can read the Microsoft documentation on this web page.

If you prefer to use the C# language, instead of Visual Basic, then these videos should be helpful.

The Development Process

The basic process of creating a program in Visual Basic (or any other language) is shown below



The process is quite simple, but unfortunately it typically involves quite a lot of programmer jargon. The Visual Basic statements you write are known as "source code". This code is typically contained in one or more text files with the extension ".VB". Your source code is then sent to a compiler, which converts it into "object code" that your computer can actually understand and run. The object code is sometimes referred to as an "executable" or a "library", or an "assembly", and is held in a file with the extension ".EXE" or ".DLL".

Structure of a Visual Basic Program

A Visual Basic program has standard building blocks, typically present in the following sequence:

- **Option** statements
- **Imports** statements
- The **Main** procedure
- **Class** and **Module** elements

Option Statements

Option statements establish ground rules for subsequent code. `Option Explicit On` ensures that all variables are declared, which may make debugging easier. `Option Strict On` applies more strict rules to variable type conversions, which helps prevent problems that can occur when you transfer information between variables of different types. `Option Infer On` asks the compiler to try to guess the types of your variables, which reduces the need for declarations, as explained a little later, on page 29.

If you place Option statements in your source code, they must be placed at the beginning of a source file, and they apply only to the source file in which they appear. Another approach is to specify compilation options in the properties of a Visual Studio project, in which case they apply to the entire project. This is often more convenient.

Imports Statements and Namespaces

Placing an **Imports** statement at the beginning of a source file allows you to use abbreviated names within that file (rather than longer "fully qualified" ones), which reduces your typing effort. For example, suppose you will frequently be using the System.Console.WriteLine function to output text. If you write `Imports System.Console` at the beginning of your source file, then you can refer to this function as simply `WriteLine` whenever you need it.

In Visual Basic, the thing that appears in an Imports statement can be either a class or a namespace. Classes are explained later in this chapter. Namespaces help you to organize large quantities of code into related subgroups and to distinguish different uses of the same name. Suppose you had a large application that performed operations on both fish and musical instruments. This probably isn't very likely, but it provides a convenient illustration. You might invent two namespaces called Instruments and Fish to hold your code. You could use the name Bass within both of these namespaces, because Instruments.Bass and Fish.Bass would be two different names. If you wrote Imports Instruments at the top of a code file, you could use the name Bass instead of Instruments.Bass. If you wrote both Imports Instruments and Imports Fish, then you would create a problem, of course, because then the name Bass would be ambiguous.

The Main Procedure

The **Main** procedure is the "starting point" for your application — the first procedure that is accessed when you run your code. **Main** is where you would put the code that needs to be accessed first.

Classes, Modules, and Files

Each line of executable code must belong to some class or module. Classes are explained near the end of this chapter. For now, you can consider a class to be a related collection of code and data fields, often representing some generic type of object. A module is really a special simplified type of class. Modules are not as flexible as classes, and they are not used as much in real-world applications, but we use them in this document because they provide a convenient way to temporarily manage smallish snippets of code. As you may recall, the NX Journaling function always produces code that is packaged into a Module. Many people advocate placing each class in its own source file, and giving this source file the same name as the class, but, you can place several classes in a single file, if you want to. Conversely, you do not have to put an entire class within a single file — by using the "partial class" capability, you can split a class definition into several files, which is often useful.

An Example Program

The listing below shows a simple program containing most of the elements mentioned above. Don't try to compile and run this program right now; let's just read it and understand it, for the time being.

```
Option Explicit On
Imports Snap

Module MyProgram

   Sub Main()
      Dim radius As Double = 3.75
      Dim area As Double
      area = CircleArea(radius)                  ' Call function to calculate area
      Dim message As String = "Area is: "
      InfoWindow.WriteLine(message & area)       ' Write out the area value
   End Sub

   ' Function to calculate the area of a circle
   Function CircleArea(r As Double) As Double
      Dim pi As Double = System.Math.PI
      Dim area As Double = pi * r * r
      Return area
   End Function

End Module
```

The program starts with an Option statement and an Imports statement. Then there is a single module called "MyProgram" that holds all the executable code. Inside this module there is a "Main" procedure, as always, and then another function called CircleArea.

The following table gives more details:

Lines of code	Explanation
`Option Explicit On`	Tells the compiler that it should give you an error message if you fail to declare any variables
`Imports Snap`	Allows you to refer to functions in the Snap namespace using short names
`Dim radius As Double = 3.75`	Declares a variable of type Double, gives it the name radius, and stores the value 3.75 in it.
`Dim area As Double`	Declares another variable of type Double, and names it area
`area = CircleArea(radius)`	Calls a function named CircleArea, which is defined below. The variable radius is used as the input to this function, and the output returned from the function is written into the variable named area.
`Dim message As String = "Area is: "`	Declares and initializes a variable of type String
`InfoWindow.WriteLine(message & area)`	Calls a function named InfoWindow.WriteLine to write text on the NX Info window. This function lives in the Snap namespace, so its full name is Snap.InfoWindow.WriteLine. We can use the shortened name here because we wrote "`Imports Snap`" above.
`' Function to calculate circle area`	This is a "comment". Comments are descriptive text to help you and other readers understand the code. They are ignored by the compiler.
`Function CircleArea(r As Double) As Double`	This is the heading for the definition of a function named CircleArea. The text in parentheses says that, when this function is called, it should receive as input a variable of type Double, which will be referred to as "r". As output, the function will return an item of type Double.
`Dim pi As Double = System.Math.PI`	Defines a variable called pi and gives it the value π (accurate to around 15 decimal places).
`Dim area As Double = pi * r * r`	Calculates the area, and stores it in a newly declared variable called area
`Return area`	Returns the value area as the output of the function

Lines of Code

Generally, you place one statement on each line of your source file. But you can put several statements on a single line if you separate them by the colon (:) character. So, for example, you might write

```
x1 = 3    :    y1 = 5    :    z1 = 7
x2 = 1    :    y2 = 2    :    z2 = 9
```

A statement usually fits on one line, but when it is too long, you can continue it onto the next line by placing a space followed by an underscore character (_) at the end of the first line. For example:

```
Dim identityMatrix As Double(,)  =  {  {1, 0, 0}, _
                                       {0, 1, 0}, _
                                       {0, 0, 1}  }
```

Actually, in modern versions of Visual Basic, the underscores are often unnecessary, since the compiler can figure out by itself when a line of code is supposed to be a continuation of the one before it.

Note that "white space" (space and tab characters) don't make any difference, except in readability. The following three lines of code do exactly the same thing, but the first is much easier to read, in my opinion:

```
y = 3.5 * ( x + b*(z - 1) )
y=3.5*(x+b*(z-1))
y     =3.5 * (  x+b * (z - 1)  )
```

Built-In Data Types

In Visual Basic, as in most programming languages, we use variables for storing values. Every variable has a name, by which we can refer to it, and a data type, which determines the kind of data that the variable can hold.

Some of the more common built-in data types are shown in the following table:

Type	Description	Examples	Approximate Range of Values
Integer	A whole number	1, 2, 999,–2, 0	–2,147,483,648 through 2,147,483,647
Double	Floating-point number	1.5, –3.27, 3.56E+2	4.9×10^{-324} to 1.8×10^{308}, positive or negative
Char	Character	"x"c, "H"c, "山"c	Any Unicode character
String	String of characters	"Hello", "中山"	Zero up to about 2 billion characters
Boolean	Logical value	True, False	True or False
Object	Holds any type of data		Anything

Note that variables of type Double can use scientific notation: the "E" refers to a power of 10, so 3.56E+2 means 3.56×10^2, which is 356, and 3.56E-2 means 0.0356. There are many other built-in data types, including byte, decimal, date, and so on, but the ones shown above are the most useful for our purposes.

Declaring and Initializing Variables

To use a variable, you first have to declare it (or, this is a good idea, at least). It's also a good idea to give the variable some initial value at the time you declare it. Generally, a declaration/initialization takes the following form:

```
Dim <variable name> As <data type> = <initial value>
```

So, some examples are:

```
Dim n As Integer = -45
Dim triple As Integer = 3*n
Dim biggestNumberExpected As Integer = 999
Dim diameter As Double = 3.875
Dim companyName As String = "Acme Incorporated"
```

For more complex data types, you use the "new" keyword and call a "constructor" to declare and initialize a new variable, like this:

```
Dim <variable name> As New <data type>(constructor inputs)
```

```
Dim v As New Vector(1, 0, 0)
Dim generator As New System.Random()
Dim myButton As New System.Windows.Forms.Button()
```

A variable name may contain only letters, numbers, and underscores, and it must begin with either a letter or an underscore (not a number). Variable names are **NOT** case sensitive, so `companyName` and `CompanyName` are the same thing. Also, variable names must not be the same as Visual Basic keywords (like `Dim` or `Integer`).

There are some ways to omit or shorten variable declarations, as explained in the next section.

Omitting Variable Declarations

When you're just experimenting with small programs, declaring variables is sometimes not very helpful, and the extra typing and text just interfere with your thought process. If you put `Option Explicit Off` at the beginning of your program, then this will prevent the compiler from complaining about missing declarations, and this might make your life easier (for a while, anyway). On the other hand, as we saw in chapter 3, declaring variables helps the compiler find mistakes for you, so it's valuable.

When you write `Option Explicit Off`, the compiler doesn't know the types of undeclared variables, so it assumes that they are all of type System.Object. As we will see later, all objects in Visual Basic are derived (either directly or indirectly) from System.Object, so a variable of this type can hold any value whatsoever, and any assignment statement will work, no matter how peculiar:

```
circ = 3.75               ' circ is of type System.Object
circ = "hello"            ' so this strange assignment works
circ = Circle(2, 3, 1)    ' and so does this
r = circ.Radius           ' works, but we get no help from Intellisense
```

When a variable is of type System.Object, you don't get much help from Visual Studio Intellisense. When you type the dot in the fourth line of code above, you might be hoping to see a helpful list of the properties of an NX.Arc object, but you won't, because the compiler thinks that `circ` is a System.Object, not an NX.Arc.

If you get tired of declaring variables, but you still want the compiler to find your mistakes, and give you helpful Intellisense hints, then a good compromise is `Option Infer On`. With this option, the compiler tries to guess the type of a variable, based on its initialization or first usage. The code looks like this:

```
Dim x = 3.75                  ' Compiler guesses that x is of type Double
Dim y = Math.SinD(x)          ' Compiler guesses that y is of type Double
Dim greeting = "hello"        ' Compiler guesses that greeting is of type String
Dim circ = Circle(2, 3, 1)    ' Compiler guesses that circ is of type NX.Arc
Dim r = circ.Radius           ' Intellisense helps us, now
```

The word `Dim` before a variable is what prompts the compiler to start guessing. You are still declaring the variables `x`, `y`, `greeting`, and `circ`, but you don't have to tell the compiler their types, because it can guess from the context. This can cut down on a lot of repetition, and make your code much easier to read. In the following, the second three lines of code are much clearer than the first three, and just as safe:

```
Dim p1 As NX.Point = Point(3,4)
Dim q1 As NX.Point = Point(7,9)
Dim a1 As NX.Line = Line(p1, q1)

Dim p2 = Point(2,5)
Dim q2 = Point(6,8)
Dim a2 = Line(p2, q2)
```

In the examples later in this document, and in the SNAP Reference Guide, we will sometimes use `Option Infer On` to make the code shorter and easier to read.

You have to be a little careful, sometimes, because the guessing isn't foolproof. Consider the following code:

```
Dim r = 3                       ' Compiler assumes that r is an Integer
Dim circ = Circle(0, 0, 5.75)   ' Create circle with radius 5.75
r = circ.Radius                 ' Error or unwanted rounding
```

The compiler will infer that r is an Integer. So, in the third line of code, we're trying to assign a Double value to an Integer variable, and we'll either get an error message, or the value of `circ.Radius` will be rounded to 6 (instead of 5.75) when it's stored in the variable `r`. To avoid this sort of problem, you can write `Dim r = 3.0` in the first line, which will tell the compiler that r is supposed to be a Double.

Data Type Conversions

Conversion is the process of changing a variable from one type to another. Conversions may either be *widening* or *narrowing*. A widening conversion is a conversion from one type to another type that is guaranteed to be able to contain it (from Integer to Double, for example), so it will never fail. In a narrowing conversion, the destination variable may not be able to hold the value (an Integer variable can't hold the value 3.5), so the conversion may fail.

Conversions can be either *implicit* or *explicit*. Implicit conversions occur without any special syntax, like this:

```
Dim weightLimit As Integer = 500
Dim weight As Double = weightLimit      ' Implicit conversion from Integer to Double
```

Explicit conversions, on the other hand, require so-called "cast" operators, as in the following examples.

```
Dim weight As Double = 500.637
Dim roughWeight As Integer
roughWeight = CInt(weight)              ' Cast weight to an integer (rounding occurs)
roughWeight = CType(weight, Integer)    ' Different technique, but same result
```

Casts can be performed with the general CType function, or with more specific functions like CInt. The result is exactly the same — the weight value is rounded and we get roughWeight = 501.

The set of allowable implicit conversions depends on the `Option Strict` setting. If you use `Option Strict On`, only widening conversions may occur implicitly. With `Option Strict Off`, both widening and narrowing conversions may occur implicitly.

It is possible to define your own conversion operators, and, in fact, we do this quite often in SNAP to make things more convenient for you. This topic is discussed further in chapter 5 and chapter 6.

Arithmetic and Math

Arithmetic operators are used to perform the familiar numerical calculations on variables of type Integer and Double. The only operator that might be slightly unexpected is "^", which performs exponentiation (raises a number to a power). Here are some examples:

```
Dim m As Integer = 3
Dim n As Integer = 4
Dim p1, p2, p3, p4, p5 As Integer
p1 = m + n              ' p1 now has the value 7
p2 = 2*m + n - 1        ' p2 now has the value 9
p3 = 2*(m + n) - 1      ' p3 now has the value 13
p4 = m / n              ' p4 now has the value 1. Beware !!
p5 = m ^ n              ' p5 now has the value 81
```

Even though m and n are both integers, performing a division produces a Double (0.75) as its result. But then when you assign this value to the Integer variable p4, it gets rounded to 1. With either Integer or Double data types, dividing by zero will cause trouble, of course.

The System.Math namespace contains all the usual mathematical functions, so you can write things like:

```
Dim rightAngle As Double = System.Math.PI / 2
Dim cosine As Double = System.Math.Cos(rightAngle)
Dim x, y, r, theta As Double
theta = System.Math.Atan2(3, 4)        ' theta is about 0.6345 (radians)
x = System.Math.Cos(theta)             ' x gets the value 0.8
y = System.Math.Sin(theta)             ' y gets the value 0.6
r = System.Math.Sqrt(x*x + y*y)
```

Note that the trigonometric functions expect angles to be measured in radians, not in degrees. In SNAP, angles are always expressed in degrees, not in radians, since this is more natural for most people. So, SNAP has its own set of trigonometric functions (SinD, CosD, TanD, AsinD, AcosD, AtanD, Atan2D) that use degrees, instead.

If you have `Imports Snap.Math` at the top of your file, then the code from above can be written more clearly as:

```
Dim rightAngle As Double = 90
Dim cosine As Double = CosD(rightAngle)
Dim x, y, r, theta As Double
theta = Atan2D(3, 4)                ' theta is about 36.87 (degrees)
x = CosD(theta)                     ' x gets the value 0.8
y = SinD(theta)                     ' y gets the value 0.6
r = System.Math.Sqrt(x*x + y*y)
```

Other useful tools include hyperbolic functions (Sinh, Cosh, Tanh), logarithms (Log and Log10), and absolute value (Abs). Visual Studio Intellisense will show you a complete list as you type.

In floating point arithmetic (with Doubles), small errors often occur because of round-off. For example, calculating 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 (10 times) won't give you 1.0, you'll get 0.99999999999999989, instead. Tiny errors like this usually don't matter in engineering applications. But, in cases where they do, you can use the Decimal data type, instead of Double. Arithmetic is much slower with Decimal variables, but more precise.

Logical Values & Operators

Visual Basic provides a set of relational operators that perform some comparison between two operands and return a Boolean (true or false) result. Briefly, these operators are, =, <, >, <=, >=, <>. Their meanings are fairly obvious, except perhaps for the last one, which means "is not equal to".

Also, there are some logical operators that act on Boolean operands. They are:

- `And`: the result is True when both of the operands are True
- `Or`: the result is True when at least one of the operands is True
- `Xor`: the result is True when exactly one of the operands is True
- `Not`: this is a unary operator. The result is True if the operand is False

Using these operators, we can construct complex conditions for use in If statements and elsewhere:

```
Dim four As Integer = 4
Dim five As Integer = 5
Dim six  As Integer = 6
Dim m, n As Integer
Dim b1, b2, b3, b4, b5, b6 As Boolean

b1 = (four = five)                      ' Result is False
b2 = (six < five)                       ' Result is False
b3 = (four <> five)                     ' Result is True
b4 = ("four" < "five")                  ' Result is False. String comparison is alphabetical !
b5 = (four < five) And (five < six)     ' Result is True
b6 = (m < n) Or (m >= n)                ' Result is True (regardless of values of m and n)
```

Arrays

An array is a collection of values that are related to each other in some way, and have the same data type. Within an array, you can refer to an individual element by using the name of the array plus a number. This number has various names: index, offset, position, or subscript are some common ones. The term "offset" is perhaps the best, since it highlights the fact that the numbering starts at zero — the first element of the array has an offset of zero.

In the following code, the first line declares and initializes an array variable that holds the number of people who work on each floor of an office building. It says that 5 people work on the ground floor, 27 on the first floor, and so on. Then the second and third lines read values from the people array.

```
Dim people As Integer() = {5, 27, 22, 31}
Dim groundFloorPeople As Integer = people(0)     '  5 people work on the ground floor
Dim  firstFloorPeople As Integer = people(1)     ' 27 people work on the first floor
```

Note that the style of array declaration shown here is perfectly legal, but it is not the usual one. Most VB programmers would write `Dim people() As Integer`, but I think the style shown above makes more sense — it says that people is an Integer() (i.e. it is an Integer array). If you want to declare and initialize the array separately, then you write something like:

```
Dim people As Integer()        ' Declares people as an array of integers
people = New Integer(3) {}     ' Initialises the "people" array variable
people(0) = 5                  ' Initialise the elements of the array, one by one
people(1) = 27
people(2) = 22
people(3) = 31
```

In this case, you need to place an integer between the parentheses in the declaration. Note that the number you use is the upper bound of the array (the highest index), which is one less than the number of elements in the array. So, in the example above, the "`New Integer(3)`" gave us an array of **four** integers with indices 0, 1, 2, 3. If you have experience with C-style programming languages, this can be very confusing, so please beware.

You can also create two-dimensional (and higher dimension) arrays using declarations like

```
Dim identityMatrix As Double(,) = { {1,0,0}, {0,1,0}, {0,0,1} }
```

The .NET framework provides many useful functions for working with arrays. For example:

- The Length property returns the total number of elements in the array
- The GetUpperBound method returns the highest index value for the specified dimension
- The Sort method sorts the elements of a one-dimensional array
- The Find and FindIndex methods allow you to search for specific items

Other Types of Collections

The .NET Framework includes the System.Collections namespace, which provides many useful "collections" that are more general than the arrays described above. For example, there are Lists, Dictionaries (Hash Tables), Queues, Stacks, and so on. You should use a List (rather than an array) when you don't know in advance how many items you will need to store. Here is a simple example:

```
Dim nameList As New List(Of String)     ' Create a list of strings
Dim name As String
Do                                  ' Loop to collect names
  name = GetName()                  ' Get the next name, somehow
  nameList.Add(name)                ' Add it to our list
Loop Until name = ""                ' Keep going until a blank name is encountered
```

There is also a general collection called an ArrayList, which can hold elements of different types. So, you can write:

```
Dim myList As New ArrayList()
myList.Add("apple pie")
myList.Add(System.Math.PI)
Dim x as Double = myList(1)        ' Gives x the value 3.14159625 etc.
```

Like a List, an ArrayList expands dynamically as you add elements. Though the ArrayList type is more general, you should use the List type, where possible, since it is faster and less error-prone. Most of the "collection" types support the same capabilities as arrays, such as indexing, counting, sorting, searching, and so on.

Strings

A String is essentially an array of characters. You can declare and initialize a string with one statement like:

```
Dim myString As String = "Hello, World!"
```

You can extract characters from a String just as if it were an array of characters:

```
Dim alphabet As String = "ABC"
Dim c0 As Char = alphabet(0)    ' Sets c0 equal to "A"
Dim c1 As Char = alphabet(1)    ' Sets c1 equal to "B"
Dim c2 As Char = alphabet(2)    ' Sets c2 equal to "C"
```

You can "concatenate" two strings (join them together into one) using either the "+" or "&" operators. Also, there are many useful functions available for working with strings; some of them are: Trim, ToUpper, ToLower, SubString, StartsWith, Compare, Copy, Split, Remove and Length. For example:

```
Dim firstName As String = "Jonathon"
Dim  lastName As String = "Smith"
Dim  nickName As String = firstName.Substring(0, 3)        ' Sets nickName = "Jon"
Dim  fullName As String = firstName & " " & lastName       ' Sets fullName = "Jonathon Smith"
Dim  greeting As String = "Hi, " & nickName                ' Sets greeting = "Hi, Jon"
```

Strings are immutable, which means that once you assign a value to one, it cannot be changed. Whenever you assign another value to a string, or edit it in some way, you are actually creating a new copy of the string variable and deleting the old one. If you are doing a lot of modifications to a string variable, use the StringBuilder type, instead, because it avoids this deletion/recreation and gives much better performance.

Any .NET object can be converted to String form using the ToString method. So, for example, this code

```
Dim pi As Double = System.Math.PI
Dim piString As String = pi.ToString()
```

will place the string "3.14159265358979" in the variable `piString`.

Enumerations

Enumerations provide a convenient way to work with sets of related constants. You can give names to the constants, which makes your code easier to read and modify. For example, in SNAP, there is an enumeration that represents the various types of line font that can be assigned to an object. In shortened form, its definition might look something like this:

```
Enum LineFont
   Solid  = 0
   Dashed = 1
   Dotted = 2
End Enum
```

Having made this definition, the symbol LineFont.Dotted now permanently represents the number 2. The benefit is that a statement like `myFont = LineFont.Dotted` is much easier to understand than `myFont = 2`.

Nothing

Some of the data types we have discussed above can have a special value called Nothing (or "null" in some other programming languages). For example, strings, arrays, and objects can all have the value Nothing. Visual Basic provides a special function called IsNothing to make it easy to test for this value. Note that Nothing does not indicate a string with no characters, or an array with zero length, as the following code illustrates:

```
Dim nullString As String = Nothing       ' A String variable with value = Nothing
Dim zeroLengthString As String = ""      ' A String with zero length (no characters)

Dim b1 As Boolean = IsNothing(nullString)           ' True
Dim b2 As Boolean = IsNothing(zeroLengthString)     ' False
```

Simple data types like Integers, Doubles, Vectors and Positions cannot have the value Nothing, ordinarily — there is no such thing as a null integer or a null Position. This is actually quite inconvenient, at times. For example, in a function that computes the point of intersection of two curves, it would be natural to return Nothing if the curves don't actually intersect. Fortunately, recent versions of Visual Basic provide a solution via a technology called "nullable value types": by placing a question mark (?) after a variable type, you can indicate that it should be allowed to hold the value Nothing, in addition to its "regular" values. Then you can use the `HasValue` function to find out whether or not the variable holds a "real" value, rather than Nothing, as the following code shows:

```
Dim s1 As NX.Spline = BezierCurve( {0,0,0}, {1,0,0}, {2,1,0} )
Dim s2 As NX.Spline = BezierCurve( {0,1,0}, {2,0,0}, {4,0,0} )

Dim nearPoint as new Position(1.5, 0.5, 0)

' Try to compute an intersection point
Dim intPoint As Position? = Compute.Intersect(s1, s2, nearPoint)

' If the returned value is not Nothing, write it out
If intPoint.HasValue Then InfoWindow.Write(intPoint.Value)
```

Actually, `Position?` is an abbreviation for `Nullable(Of Position)`, and you may see the longer form in documentation, sometimes.

Decision Statements

Simple decisions can be implemented using the `If Then Else` construct, as shown in the following tax computation. It assumes that we have already defined two variables called `income` and `tax`

```
If income < 27000 Then
  tax = income * 0.15                       ' 15% tax bracket
ElseIf income < 65000 Then
  tax = 4000 + (income - 27000) * 0.25      ' 25% tax bracket
Else
  tax = 4000 + (income - 65000) * 0.35      ' 35% tax bracket
End If
```

If there were only two tax brackets, we wouldn't need the `ElseIf` clause, so our code could be simpler:

```
If income < 27000 Then
  tax = income * 0.15                       ' 15% tax bracket
Else
  tax = 4000 + (income - 65000) * 0.35      ' 35% tax bracket
End If
```

This could be simplified even further:

```
tax = income * 0.15                         ' 15% tax bracket
If income > 27000 Then
  tax = 4000 + (income - 65000) * 0.35      ' 35% tax bracket
End If
```

Finally, we can compress the If statement into a single line, if we want to:

```
If income > 27000 Then tax = 4000 + (income - 65000) * 0.35          ' 35% tax bracket
```

Looping

It is often useful to repeat a set of statements a specific number of times, or until some condition is met, or to cycle through some set of objects. These processes are all called "looping". The most basic loop structure is the `For ... Next` loop, which takes the following form

```
For i = 0 To n
  a(i) = 0.5 * b(i)
  c(i) = a(i) + b(i)
Next
```

The variable `i` is called the loop counter. The statements between the `For` line and the `Next` line are called the body of the loop. These statements are executed n+1 times, with the counter `i` set successively to 0, 1, 2, ..., n. It is often convenient to declare the counter variable within the `For` statement. Also, you can append the name of the counter variable to the Next statement, which sometimes improves clarity, especially in "nested" loops like this:

```
For i As Integer = 0 To m
  For j As Integer = 0 To n
    c(i, j) = a(i) + b(j)
  Next j
Next i
```

Several other looping constructs are available, including:

- The `For Each...Next` construction runs a set of statements once for each element in a collection. You specify the loop control variable, but you do not have to determine starting or ending values for it.
- The `Do...Loop` construction allows you to test a condition at either the beginning or the end of a loop structure. You can also specify whether to repeat the loop while the condition remains True or until it becomes True.

Functions and Subroutines

In many cases, you will call a "function" to perform some task. For example, you call the Math.Sqrt function to calculate the square root of a number, or you call the Snap.InfoWindow.WriteLine function to write out text. Sometimes the function is one that you wrote yourself, but, more often, it's part of some library of functions written by someone else (like SNAP). You pass inputs to a function when you call it, the code inside the function is executed, and then (sometimes) it returns some value to you as output. The function provides a convenient place to put a block of code, so that it's easy to re-use. Here are some examples of function calls:

```
' Some calls to the Math.Sqrt function
Dim x, y, z As Double
x = 3
y = Math.Sqrt(x)
z = Math.Sqrt(5)
Dim root2 As Double = Math.Sqrt(2)

' Some calls to the WriteLine function
Dim greeting As String = "Hello"
Snap.InfoWindow.WriteLine(greeting)
Snap.InfoWindow.WriteLine("Goodbye")

' Some calls to Snap functions
Dim p1, p2 As Snap.NX.Point
p1 = Snap.Create.Point(3, 5, 7)
p2 = Snap.Create.Point(2, 4, 6)
Snap.Create.Line(p1, p2)
```

In Visual Basic, a function that does not return a value is called a "Subroutine" or just a "Sub". In the code above, Snap.InfoWindow.WriteLine is a subroutine, but Math.Sqrt, Snap.Create.Point, and Snap.Create.Line are not. Even if a function does return a value, you are not obligated to use this value. For example, in the code above, we didn't use the value returned from the Snap.Create.Line function. A function can have any number of inputs (or "arguments") including zero.

Near the start of this chapter, we saw an example of a function (CircleArea) that you might have written yourself:

```
Function CircleArea(r As Double) As Double
   Dim pi As Double = 3.14
   Dim area As Double = pi * r * r
   Return area
End Function
```

Since you have the source code of this function, you could just use this code directly, instead of calling the function, but we would not recommend this approach; calling functions makes your code less repetitive, easier to read, and easier to change. The general pattern for a function definition is:

```
Function <FunctionName>(arguments) As <ReturnType>
   <body of the function>
End Function
```

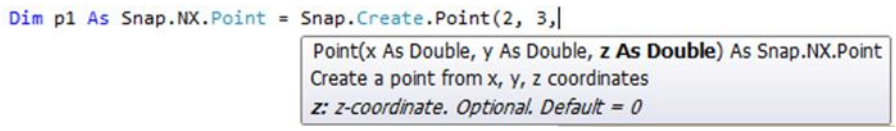
Some further examples are:

```
Function RectangleArea(width As Double, height As Double) As Double     ' Area of a rectangle
Function Average(m As Double, n As Double) As Double                    ' Average of two numbers
Function Average(values As Double()) As Double                          ' Avg of list of numbers
Function Cube(center As Position, size As Double) As Snap.NX.Body       ' Create a cube
```

Note that it's perfectly legal to have several functions with the same name, provided they have different types of inputs. This technique is called "overloading", and the function name is said to be "overloaded". For example, the function name "Average" is overloaded in the list of function definitions above. When you call the function, the compiler will decide which overload to call by looking at the types of inputs you provide.

Optional Arguments for Functions

In some cases, there is a reasonable default value for a function argument, so supplying that argument as an input can be optional when you call the function. The SNAP Point function is a good example. Creating points on the XY-plane is very common, so it is reasonable to use z = 0 as the default value for the third input to the Point function. When you write a call to this function in Visual Studio, Intellisense will tell you that the third input is optional, and what default value will be used if you do not supply one, like this:



An extreme example is the SNAP Print function — it has 11 arguments, but they are all optional.

Arrays as Function Arguments

It's quite common (especially in SNAP) to have functions that receive arrays of objects as input. We saw an example above — we had a function called Average whose input was an array of Double values. When you have a small number of items, packing them into an array just so that you can call some function is an inconvenience and a distraction. Instead of writing:

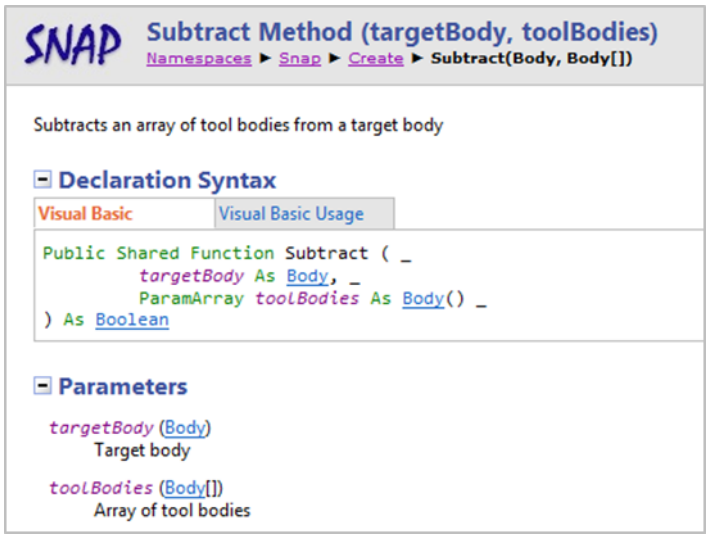
```
Dim values As Double() = { 3, 5 , 7 }
mean = Average(values)
```

it would be nice if we could dispense with the array and just write:

```
mean = Average(3, 5, 7)
```

It's especially annoying in functions like Snap.Create.Subtract. This function receives an array of tool bodies that are to be subtracted from a target body. But subtracting a single tool body is a very common case, and building an array with one element is pretty silly. Fortunately, Visual Basic provides us with a way to avoid this inconvenience.

The SNAP Reference Guide shows us that the specifications for the Subtract function are:



As you can see, the array of tool bodies is marked with the word "ParamArray". What this means is that you can input a list of individual bodies, rather than an array. The following code shows the available alternatives:

```
Dim target As NX.Sphere = Sphere( {0,0,0}, 6 )
Dim toolA  As NX.Sphere = Sphere( {0,3,0}, 1 )
Dim toolB  As NX.Sphere = Sphere( {0,0,3}, 1 )

Dim toolBodies As NX.Body() = { toolA.Body, toolB.Body }

Subtract( target, toolA )              ' Subtract one tool body
Subtract( target, toolA, toolB )       ' Subtract two tool bodies
Subtract( target, {toolA, toolB} )     ' Array constructed on the fly
Subtract( target, toolBodies )         ' Using explicit array of tool bodies
```

The last four lines of code are just to illustrate the available alternatives; you can't run this code exactly as shown, because the Subtract function "consumes" its tool bodies. The last three calls to the Subtract function all produce exactly the same result.

Classes

In addition to the built-in types described earlier, Visual Basic allows you to define new data types of your own. The definition of a new user-defined data type is held in a block of code called a class. The class represents a generic object, and a specific concrete object of this type is called an "instance" of the class. So, for example, we might have a "Sphere" class that represents spheres in general, and the specific sphere object with center at (0,0,0) and radius = 3 would be an instance of this Sphere class.

New objects defined by classes have **fields**, **properties** and **methods**. Fields and properties can be considered as items of data (like the radius of a sphere), and a method is a function that does something useful with an object of the given class (like calculating the volume of a sphere). Properties are described in the next section, but, for now, you can think of a property as just a field with a smarter and safer implementation — it provides controlled read/write access to a hidden field.

A class typically includes one or more functions called “constructors” that are used to create new objects. So, a typical class definition might look like this:

```
Public Class Ball

   Public Center As Position     ' Field to hold center point (should be a property, really)
   Public Radius As Double       ' Field to hold radius value (should be a property, really)

   ' Constructor, given a point and a radius
   Sub New(center As Position, r As Double)
      Me.Center = center
      Me.Radius = r
   End Sub

   ' Constructor, given center coordinates and radius
   Sub New(x As Double, y As Double, z As Double, r As Double)
      MyClass.New(New Position(x, y, z), r)
   End Sub

   ' Function (method) to calculate volume
   Public Function Volume() As Double
      Return (4 / 3) * System.Math.PI * Me.Radius ^ 3
   End Function

   ' Function (method) to draw a ball
   Public Sub Draw()
      ' Code omitted
   End Sub

End Class
```

Note that the constructors are “overloaded” — there are two of them, with different inputs. To create a ball object, you call a constructor using the New keyword. Properties and methods are both accessed using a “dot” notation. As soon as you type a period in Visual Studio, Intellisense will show you all the available fields, properties and methods.

In this class, Center and Radius are both public fields, so you can access them directly. It would be safer to make them private fields and provide properties to access them. By doing this, we could prevent the calling code from making balls with negative radius, for example. Code to use the Ball class looks like this:

```
Dim myBall As New Ball(x, y, z, r)                ' Create a ball named "myBall"
myBall.Radius = 10                                ' Change its Radius property (or field)
Dim mass As Double = density * myBall.Volume()    ' Use the Volume method
myBall.Draw()                                     ' Display the ball
```

Note that the first line of code uses a convenient shorthand notation. The full form would have been

```
Dim myBall As Ball = New Ball(x, y, z, r)
```

Shared Functions

In the example above, we had a class called “Ball”, and this class contained functions (methods) like Volume and Draw that operated on balls. This is the “object-oriented programming” view of life — the world is composed of objects that have methods operating on them. This is all very nice, but some software doesn't fit naturally into this model. Suppose for example that we had a collection of functions for doing financial calculations — for calculating things like interest, loan payments, and so on. The functions might have names like SimpleInterest, and LoanPayment, etc. It would be natural to gather these functions together in a class named FinanceCalculator. But the situation here would be fundamentally different from the “ball” class. The SimpleInterest function lives in the FinanceCalculator class, but it doesn’t operate on FinanceCalculator objects. Saying it another way, the SimpleInterest function is associated with the FinanceCalculator class itself, not with instances of the FinanceCalculator class.

Functions like this are called “Shared” functions in Visual Basic (or “static” functions in many other languages). You have already seen this word many times before because the “Main” function is always Shared. By contrast, the functions Volume and Draw in the Ball class are called Member functions or Instance functions. So, in short, the FinanceCalculator class is simply a collection of Shared functions. This is a common situation, so Visual Basic has a special construct to support it — a class that consists entirely of Shared functions is called a Module.

Calls to Member functions and Shared functions look the same in our code, but they are conceptually different. For example, look at:

```
Dim myBall As New Ball(x, y, z, r)
Dim v As Double = myBall.Volume()
Dim payment As Double = FinanceCalculator.LoanPayment(20000, 4.5)
```

Both the second and third lines use the “dot” notation to refer to a function. But, in these two cases, the thing that comes before the “dot” is different. In myBall.Volume on the second line, myBall is an object (of type Ball), but in FinanceCalculator.LoanPayment on the third line, FinanceCalculator is a class.

Object Properties

Each type of object we create in a VB program typically has a set of “properties” that we can access. For example, a point has a position in space, an arc has a center and a radius, a curve has a length, and a solid body has a density. In all cases, you can read (or “get”) the value of the property, and in many cases, you can also write (or “set”) the value. Setting a property value is often a convenient way to modify an object.

If you are familiar with the GRIP language, these properties are exactly analogous to GRIP EDA (entity data access) symbols.

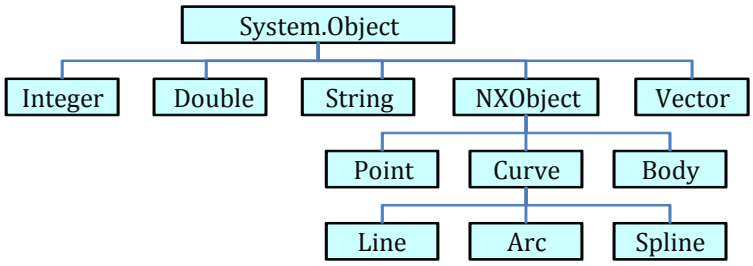
Each property has a name. To get or set the property, you use a “dot” followed by the name of the property. So, if myCircle is an arc, then you refer to its center as myCircle.Center, and its radius as myCircle.Radius.

If p1, p2, p3 are three given positions, then we can write code like this:

```
c1 = Circle(p1, p2, p3)    ' Creates a circle through three positions p1, p2, p3
r = c1.Radius              ' Gets the radius of the circle
c1.Center = p2             ' Moves the circle, placing its center at position p2
```

Hierarchy & Inheritance

Object methods and properties are hierarchical. In addition to its own particular properties, a given object also has all the properties of object types higher up in the object hierarchy. So, for example, since a Line is a kind of Curve, it has all the properties and methods of the Curve type, in addition to the particular ones of lines. We say that the Line type “inherits” properties and methods from the Curve type. A portion of the hierarchy is shown below:



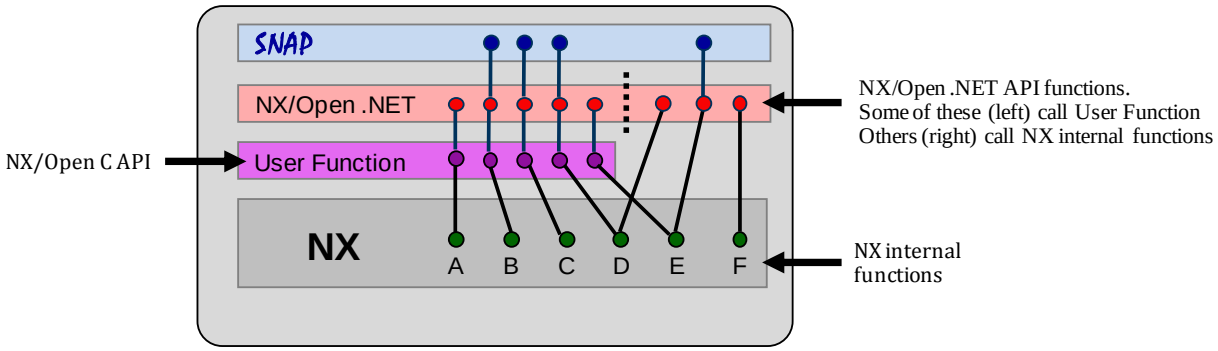
As you can see, every object is derived from System.Object, and therefore inherits certain mysterious properties from it (like the Finalize, GetHashCode, and MemberwiseClone functions). The tables in the following chapters indicate the types of objects we will be using, and their properties. You might think you will need to keep these tables handy as you are writing code, so that you know what properties are available. But, this is not the case assuming you are using a modern IDE (Integrated Development Environment) to write your code. In a good IDE (like Visual Studio), as soon as you type a dot, a list of available properties and methods will appear, and all you have to do is choose the one you want. Some enthusiasts like to say that “the code writes itself” ☺

Chapter 5: SNAP Concepts & Architecture

Now that you understand a little about Visual Basic objects and classes, we can explain how SNAP works. The details are somewhat technical, and it's not really necessary that you understand them all, but they might be interesting, and they might help to clarify some things if you get confused occasionally.

Relationship of SNAP to NX Open

The programming interfaces for NX have evolved over many years. Earlier generations are still supported and still work, even though they have been superseded by newer APIs and are no longer being enhanced. These older tools included an API called "User Function" or "UFUNC" that was designed to support applications written in the Fortran or C languages. The name of the User Function C API was subsequently changed, and it is now known as the NX Open C API, or sometimes just the Open C API. This API is old-fashioned, by today's standards, but it is extremely rich, fairly well documented, and still widely used. A large part of the NX Open .NET API (a portion called the NXOPEN.UF namespace) was actually created by building "wrappers" around NX Open C functions. Newer NX Open .NET functions are built directly on top of internal NX functions, so they by-pass the NX Open C layer. The SNAP layer is built on top of NX Open .NET. In fact, you can think of it as a "sugar coating" that makes NX Open.NET easier to digest. The layering is shown in the diagram below:



SNAP Files

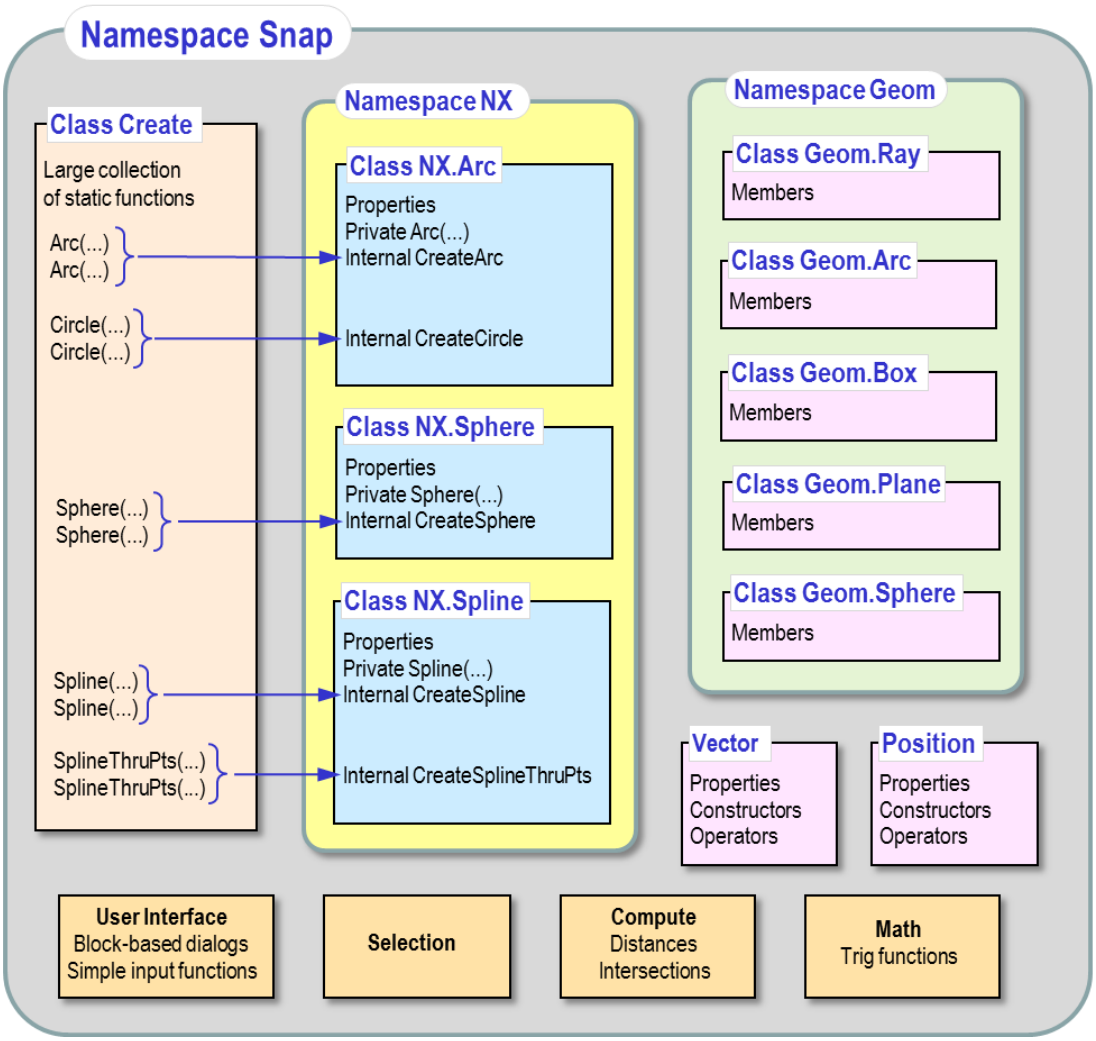
The SNAP functions reside in an assembly called Snap.dll, which you can find in the folder [...NX]\UGII\managed, where, as usual, [...NX] is just shorthand for the location where NX is installed, which is typically somewhere like C:\Program Files\Siemens\NX 10. As you know from earlier examples, your code must have a "reference" to this dll in order to use SNAP functions.

In this same location, you will find another file called Snap.xml. This file contains detailed reference information about SNAP functions, which gets displayed by the Object Browser and the Intellisense facility in Visual Studio.

Finally, the SNAP Reference Guide is contained in a file called SNAP_Reference.chm, located in your UGDOC folder.

The SNAP Architecture

A subset of the basic SNAP architecture is shown below. Lots of items are omitted for clarity, but the diagram shows a representative sample of some of the most important elements and how they are inter-related.



At the top level, there is an overall namespace called Snap. According to Microsoft rules, this should actually be called SiemensPLM.Snap, but that's too long, so we broke the rules and just called it Snap.

Within this namespace, there is another namespace called NX that contains classes with names like Snap.NX.Point, Snap.NX.Arc, Snap.NX.Sphere, and so on, which correspond to the various types of objects found in an NX part file. These classes are primarily important because of the properties they provide. So, as we saw earlier, if c1 is an object of type NX.Arc, then c1.Radius is its radius, c1.Center is the location of its center, and c1.Color is its color. The NX classes are (roughly) in one-to-one correspondence with NX Open objects — for example, NX.Spline is just a simple wrapper around NXOpen.Spline, and NX.Sphere is a wrapper around an NXOpen.Features.Sphere object, and so on. There are implicit conversions that make it easy to use an NX.Spline and an NXOpen.Spline interchangeably.

Creating NX Objects

The constructors in the NX classes are private, and are not exposed in the SNAP API, so you can't use them to create new objects. Instead, your programs create new NX objects by calling Shared (static) functions in the Create class. The confusing thing (possibly) is that the functions in the Create class also have names like Point, Line, Arc, Sphere, Extrude, and so on. We have typically written `Imports Snap.Create` and `Option Explicit Off` at the top of each of our code files, and this allows us to call these functions very conveniently, like this:

```
p1 = Point(1,3)
p2 = Point(5,6)
Line(p1, p2)
```

But, without the convenience tricks, if we wrote out this code in full, it would be:

```
Dim p1 As Snap.NX.Point         ' Declare p1 to be an object of type Snap.NX.Point
p1 = Snap.Create.Point(1,3)     ' Give p1 a value by calling the Snap.Create.Point function
Dim p2 As Snap.NX.Point         ' Declare p2
p2 = Snap.Create.Point(5,6)     ' Give p2 a value
Snap.Create.Line(p1, p2)        ' Create a line by calling Snap.Create.Line
```

So, as you can see there are really two things called "Point" — a class (whose full name is Snap.NX.Point) and a function (whose full name is Snap.Create.Point). In most situations, Visual Basic can keep all these concepts straight, but it might get confused occasionally, and you'll have to help it out.

If you're thoroughly confused by this seemingly pointless duplication, just remember two things:

- To **declare** a Point object, use Snap.NX.Point, as in Dim myPoint As Snap.NX.Point
- To **create** a Point object, call the function Snap.Create.Point (which you can often abbreviate to just plain Point)

Geom Objects

The Geom namespace is the home of various abstract geometric objects. Logically, the Position and Vector objects belong in the Geom namespace, too, but they are raised up a level because they are used so often and we want to make them easy to access. Geom objects, Positions, and Vectors are different from NX objects because they are transient, rather than permanent or persistent. They are never stored in NX part files, (or anywhere else, typically), so they disappear as soon as your SNAP program finishes executing. Also, unlike NX objects, Geom objects are often unbounded (infinite in extent). Within your programs, Geom objects, Positions, and Vectors can be very useful in a number of ways. For example:

- When you calculate a location on a curve, the result is a Position, not an NX.Point object.
- To perform mirroring, you typically use a Geom.Surface.Plane, not an NX.DatumPlane
- Information about face geometry is returned to you in the form of a Geom.Surface object.

SNAP Design Principles

This section outlines some design principles that we have generally followed in the development of SNAP. If you understand these principles, then SNAP functions should be predictable — you will often be able to guess how they work without reading the detailed documentation.

The Work Part

SNAP functions always deal with the Work Part. When you create objects, they are stored in the Work Part. When you make enquiries about objects in a part file, or part specific settings, the part file used is always the Work Part. So, if your SNAP program needs to do operations in several part files, you must change the Work Part within your code.

Coordinate Systems

All coordinates in SNAP are expressed relative to the Absolute Coordinate System. This is slightly inconvenient sometimes, but using a single fixed global coordinate system seems less confusing, ultimately, and it also allows SNAP to interoperate better with NX Open. Sometimes you may want to map coordinates between the Absolute Coordinate System (ACS) and the Work Coordinate System (WCS); the Snap.NX.CoordinateSystem class contains functions called MapAcsToWcs and MapWcsToAcs to do this for you.

Angles

All angles in SNAP are measured in degrees. The standard math functions in the .NET System.Math class expect their arguments to be in radians, so SNAP provides some alternative functions in the Snap.Math class that work in degrees. Following an old Fortran convention, the SNAP functions have names that end with "D". So, for example, `Snap.Math.SinD(45)` is the same as `System.Math.Sin(Math.PI/4)`.

Function Returns

The object returned by a SNAP function is the primary thing the function calculates or creates, not an error indicator or other flag. Or, looking it another way, the primary result(s) are returned as the value of the function, not via the function arguments. When reading about Visual Basic, you will probably see examples where functions return information via their arguments (using a concept called "pass by reference", or "ByRef"). This is a common source of confusion, so we avoid it in SNAP. In some cases, this means we have to invent little structures to serve as function results. For example, selection of an object returns the object, the cursor ray, and the user's response. All of these things are packaged into a SelectionResult object that is returned by the selection function.

Constructors vs Static Functions

The use of the `New` keyword is a common source of confusion. To create a point, we write `p1 = Point(x,y,z)`, but to create a Position, we write `p2 = New Position(x,y,z)`. Why does one of these use the `New` keyword while the other one doesn't? The bewilderment is quite understandable. When we use the `New` keyword, we are calling a constructor function to create an object. This is the pattern typically used when creating .NET framework objects like arrays, lists, and strings. Non-NX objects like Positions, Vectors and Geom objects follow this same constructor pattern, too. But, in SNAP, we always use a static function, rather than a constructor, to create an NX object (i.e. an object that resides in an NX part file). That's why creating a Point doesn't require the `New` keyword. Using constructors in SNAP would not be practical because it severely restricts the range of function names that can be used. For example, function names like Circle, BezierCurve, and LineTangent would not be usable.

Error Handling

SNAP functions indicate failure by throwing exceptions, not by returning "error flags". You may then "catch" these exceptions in your code, if you choose to. If you don't catch an exception, it is passed upwards to successive calling functions until it is either handled or it causes your program to terminate.

In most of the examples in this document and in the SNAP Reference Guide, the error handling is omitted because it would somewhat obscure the main points that we are trying to explain. But in real code, error handling is important, of course, and should not be omitted. To find out more about exceptions and how to handle them in your code, please refer to this video or other Visual Basic programming tutorials.

Abbreviations

The names of functions and properties in SNAP are generally not abbreviated. Sometimes this means that there are a lot of characters to type (like Snap.Compute.MassPropertiesResult.RadiusOfGyration), but, fortunately, Visual Studio Intellisense does most of this typing for you. If we used abbreviations, you would always have to guess how we abbreviated — the radius of gyration property could be Snap.Comp.MassPropsResult.RadGyr or Snap.Comp.MassResult.RoG, or any of dozens of other possibilities. Unabbreviated names are also much easier to understand for people whose native language is not English.

Properties

To get information about an object in SNAP, you almost always use properties, rather than "Get" or "Ask" functions. For example, if circ is an NX.Arc object, its radius is circ.Radius, not circ.GetRadius() or circ.AskRadius. In many cases, properties are also writable, so you can use them to modify an object. Using properties rather than Get/Set functions cuts the number of functions in half, and makes your code more readable. The concept will be familiar to you if you've ever used EDA (Entity Data Access) symbols in GRIP.

Chapter 6: Positions, Vectors, and Points

The next few chapters briefly outline the SNAP functions available for performing simple tasks. The function descriptions are fairly brief, since we are just trying to show you the range of functions available. The SNAP Reference Guide has much more detailed information, and this detailed information will also be presented to you as you are writing your code, if you use a good development environment like Visual Studio. Specifically, as soon as you type an opening parenthesis following a function name, a list of function inputs will appear, together with descriptions. You can also get complete information about any function or object by using the Object Browser in Visual Studio.

Following the descriptions of functions, we often give small fragments of example code, showing how the functions can be used. The examples are very simple, but they should still be helpful. To keep things brief, the example code is often not complete. For example, declarations are often left out, and a complete Main function is only included very rarely. If you actually want to compile the example code, you will typically need to make some additions.

Positions

A Position object represents a location in 3D space. After basic numbers, positions and vectors are the most fundamental objects in geometry applications, so we will describe them first. Note that a Position is not a real NX object. Positions only exist in your SNAP program — they are not stored permanently in your NX model (or anywhere else). So, as soon as your program has finished running, all your Position objects are gone. In this sense, they are just like the numerical variables that you use in your programs. If you want to create a permanent NX object to record a location, you should use a Snap.NX.Point, not a Position. You can use the following functions to create a Position:

Function	Inputs and Creation Method
`Position(x As Double, y As Double, z As Double)`	From three rectangular coordinates.
`Position(x As Double, y As Double)`	From xy-coordinates (assumes z=0).
`Position(coords As Double[])`	From an array of 3 coordinates.
`Position(p As NXOpen.Point3d)`	From an NXOpen.Point3d object.

Since this is the first of many similar tables, we will describe this one in some detail. In the first column, you see a formal description of the types of inputs you should provide when calling the function. So, for the first form, you have to provide three variables of type "double". In the third row, you see the notation "`coords As Double[]`", which indicates that coords is a variable of type Double[] — in other words, it is an array of doubles. The second column has a brief description of what the function does.

These functions are all constructors, so, when calling them, we have to use the "New" keyword in our code. Here are some examples:

```
Dim p As New Position(3,5,8)        ' Creates a position "p" with coordinates (3,5,8)
Dim q As New Position(1.7, 2.9)     ' Creates a position "q" with coordinates (1.7, 2.9, 0)
Dim x As Double() = { 3, 5, 8 }     ' Creates an array of three numbers
Dim w As New Position(x)            ' Creates a position from the array
```

Within SNAP, we have implemented implicit conversion functions that convert an array of three doubles or an NXOpen.Point3d object into a Position. This means that you do not have to perform a "cast" when you write assignment statements like this:

```
Dim p, q As Position
Dim point As New NXOpen.Point3d(3, 4, 5)
Dim coords As Double() = {6, 7, 8}
p = point        ' Implicit conversion -- no cast required
q = coords       ' Implicit conversion -- no cast required
```

This conversion facility provides a very succinct and natural way of defining Positions; you can write things like:

```
Dim p1, p2 As Position
p1 = { 1, 2, 3 }
p2 = { 4, 6, 9.75 }
```

Position object properties are as follows:

Data Type	Property	Access	Description
Double	X	get, set	The x- coordinate of the position
Double	Y	get, set	The y-coordinate of the position
Double	Z	get, set	The z-coordinate of the position
Double	PolarTheta	get	Angle of rotation in the XY-plane, in degrees
Double	PolarPhi	get	Angle between the vector and the XY-plane, in degrees

Note that the PolarTheta and PolarPhi angles are returned in degrees, not radians, as is standard in SNAP.

Positions are very important objects in CAD/CAM/CAE, so they receive special treatment in SNAP. To make our code shorter and easier to understand, many Position functions have been implemented as operators, which means we can use normal arithmetic operations (like +, –, *) instead of calling functions to operate on them. So, if u, v, w are Positions, then we can write code like this:

```
Dim centroid As Position = (u + v + w)/3    ' Centroid of a triangle
w = w + 3*Vector.AxisX                      ' Moves w along the x-axis by three units
w.X = w.X - 3                               ' Moves it back again
```

As you can see from the first line of code above, addition and scalar multiplication of Positions is considered to be legal. In fact, only certain types of expressions like this make sense, but we have no good way to distinguish between the proper ones and the improper ones, so we allow all of them.

Vectors

A vector object represents a direction or a displacement in 3D space. Like Positions, Vectors only exist in your SNAP program — they are not stored permanently in your NX model (or anywhere else). You can use the following constructor functions to create Vector objects:

Function	Inputs and Creation Method
Vector(x As Double, y As Double, z As Double)	From three rectangular components.
Vector(x As Double, y As Double)	From xy- components (assumes z=0).
Vector(coords As Double[])	From an array of three coordinates.
Vector(v As NXOpen.Vector3d)	From an NXOpen.Vector3d object.

SNAP has implicit conversion functions that convert an array of three doubles or an NXOpen.Vector3d object into a Vector, so again we do not have to perform casts, and we can define vectors conveniently using triples of numbers:

```
Dim u, v, w As Vector
w = New Vector(3,5,8)              'Creates a vector with components (3, 5, 8)

Dim vec3d As New NXOpen.Vector3d(3, 4, 5)
Dim coords As Double() = {6, 7, 8}

u = vec3d                         ' Implicit conversion -- no cast required
v = coords                        ' Implicit conversion -- no cast required

u = { 3.0, 0.1, 0.1 }             ' Nice simple definitions of vectors
v = { 0.1, 3.0, 0.1 }
w = { 0.1, 0.1, 3.0 }
```

Some functions for manipulating vectors are shown in the following table:

Function	Returns	Result
`Cross(u As Vector, v As Vector)`	`Vector`	Cross product (vector product) of two vectors.
`UnitCross(Vector u, Vector v)`	`Vector`	Unitized cross product of two vectors.
`Unit(u As Vector)`	`Vector`	Unitizes a given vector.
`Norm(u As Vector)`	`Double`	Norm (length) of a vector.
`Norm2(u As Vector)`	`Double`	Norm squared of a vector.
`Angle(u As Vector, v As Vector)`	`Double`	Angle between two vectors, in degrees

SNAP also provides three built-in unit vectors called AxisX, AxisY, AxisZ corresponding to the coordinate axes.

Vector object properties are as follows:

Data Type	Property	Access	Description
`Double`	`X`	`get, set`	The x-component of the vector
`Double`	`Y`	`get, set`	The y-component of the vector
`Double`	`Z`	`get, set`	The z-component of the vector
`Double`	`PolarTheta`	`get`	Angle of rotation in the XY-plane, in degrees
`Double`	`PolarPhi`	`get`	Angle between the vector and the XY-plane, in degrees

Vectors are very important objects in CAD/CAM/CAE, so they receive special treatment in SNAP. To make our code shorter and easier to understand, many Vector functions have been implemented as operators, which means we can use normal arithmetic operations (like +, –, *) instead of calling functions to operate on them. So, if p and q are Positions, u, v, w are Vectors, and r is a "scalar" (an Integer or a Double), then we can write code like this:

```
w = u + v                   ' Vector w is the sum of vectors u and v
v = -v                      ' Reverses the direction of the vector v
w = 3.5*u - r*v/2           ' Multiplying and dividing by scalars
u = p - q                   ' Subtracting two Positions gives a Vector
r = u*v                     ' Dot product of vectors u and v
w = Vector.Cross(u,v)       ' Cross product of vectors u and v
w = (w*u)*u + (w*v)*v       ' Various products
w = Vector.Cross(u, v)/2    ' A random pointless calculation
r = Vector.Norm(u)          ' Calculates the length (norm) of u
p = p + 3*Vector.AxisX      ' Moves p along the x-axis by three units
p.X = p.X - 3               ' Moves it back again
```

Points

Points might seem a lot like Positions, but they are quite different. A Point is an NX object, which is permanently stored in an NX part file; Positions and Vectors are temporary objects that exist only while your SNAP program is running. Despite the large conceptual difference, we will sometimes use the word "point" when we really mean "position", just because it sounds better in some contexts.

When you call any of the following functions, a point is created in your Work Part:

Function	Inputs and Creation Method
`Point(x As Double, y As Double, z As Double)`	From x, y, z coordinates
`Point(x As Double, y As Double)`	From xy-coordinates (assumes z=0)
`Point(p As Position)`	From a position
`Point(coords As Double[])`	From an array of 3 coordinates

The properties of Point objects are as follows:

Data Type	Property	Access	Description
Double	X	get, set	The x-coordinate of the point.
Double	Y	get, set	The y-coordinate of the point.
Double	Z	get, set	The z-coordinate of the point.
Position	Position	get, set	The position vector of the point.

There are many functions that require Positions as inputs. If we have a Point, instead of a Position, we can always get a Position by using the Position property of the point. So, if myPoint is a Point, and we want to create a sphere (which requires a Position for the center) we can write:

```
Sphere(myPoint.Position, radius)
```

Since their X, Y, Z properties can be set (written), it's easy to move points around, as follows:

```
p1 = Point(1, 2, 5)
p2 = Point(6, 8, 0)
p1.Z = 0                    ' Projects p1 to the xy plane
p1.Y = p2.Y                 ' Aligns p1 and p2 -- gives them the same y-coordinate
```

Note that the functions for creating Point objects are shared (static) functions in the Snap.Create class, not constructors, so we don't need the New keyword.

Chapter 7: Curves

This chapter briefly outlines the SNAP functions for creating and editing curves (lines, arcs, and splines). For further details, please look at the Snap.Create class in the SNAP Reference Guide.

Lines

The Snap.Create class contains several functions for creating lines, as follows:

Function	Inputs and Creation Method
`Line(x0 As Double, y0 As Double, z0 As Double, x1 As Double, y1 As Double, z1 As Double)`	Given x, y, z coordinates of end-points
`Line(x0 As Double, y0 As Double, x1 As Double, y1 As Double)`	Given x, y coordinates of end-points (z assumed zero)
`Line(p0 As Position, p1 As Position)`	Between two positions
`Line(p0 As Point, p1 As Point)`	Between two points (NX.Point objects)

The following fragment of code creates two points and two lines in your Work Part:

```
p1 = Point(3,5)                    ' Creates a point at (3,5,0) in your Work Part
q = New Position(2,4,6)            ' Creates a position
p2 = Point(q)                      ' Creates a point from the position q
Dim c As NX.Line = Line(p1, p2)    ' Creates a line between points p1 and p2
Line(1,3, 6,8)                     ' Creates a line from (1,3,0) to (6,8,0)
```

Notice how z-coordinates can be omitted, in some cases. Since it's quite common to create curves in the xy plane, we provide special shortcut functions for doing this, so that you don't have to keep typing zeros for z-coordinates.

The properties of lines are:

Data Type	Property	Access	Description
`Position`	`StartPoint`	`get, set`	Start point (point where t = 0).
`Position`	`EndPoint`	`get, set`	End point (point where t = 1).
`Vector`	`Direction`	`get`	A unit vector in the direction of the line.

The StartPoint and EndPoint properties can be set, so you can use them to edit a line, like this:

```
Dim myLine As NX.Line = Line(2,3,7,8)     ' Creates a line between (2,3,,0) and (7,8,0)
myLine.EndPoint = {7,8,5}                 ' Moves the end-point to (7,8,5)
```

The NX.Line class also inherits some useful properties from NX.Curve, such as Arclength:

```
Dim myLine As NX.Line = Line(0,0,3,4)
Dim length As Double = myLine.Arclength
```

The arclength of a line is just the distance between its end-points, of course, but the Arclength property makes the calculation a little more convenient and easier to read.

Arcs and Circles

Functions for creating circular arcs are:

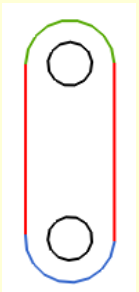
Function	Inputs and Creation Method
`Arc(center As Position, axisX As Vector, axisY As Vector, radius As Double, angle1 As Double, angle2 As Double)`	From center, axes, radius, angles.
`Arc(center As Position, matrix As Orientation, radius As Double, angle1 As Double, angle2 As Double)`	From center, orientation matrix, radius, angles.
`Arc(center As Position, radius As Double, angle1 As Double, angle2 As Double)`	From center point, radius, angles, parallel to the XY-plane.
`Arc(cx As Double, cy As Double, radius As Double, angle1 As Double, angle2 As Double)`	From center coordinates, radius, angles, lying in the XY-plane.
`Fillet(p0 As Position, pa As Position, p1 As Position, double radius)`	Fillet arc from three points.

There is no specific object of type "circle". A circle is just the name we give to an arc whose start and end angles differ by 360 degrees. So, the functions listed below simply produce arc objects as their output. But, creating circles is often simpler than creating more general circular arcs, so we provide these special functions for doing this.

Function	Inputs and Creation Method
`Circle(center As Position, axisX As Vector, axisY As Vector, radius As Double)`	Circle from center, axes, radius.
`Circle(center As Position, radius As Double)`	From center point, radius, parallel to the XY-plane.
`Circle(cx As Double, cy As Double, radius As Double)`	From center coordinates, radius, lying in the XY-plane.
`Circle(center As Position, axisZ As Vector, radius As Double)`	From center, normal, radius.
`Circle(p1 As Position, p2 As Position, p3 As Position)`	Through three points.

Here is a simple program that creates this linkage bar using lines and arcs:

```
Dim length As Double = 8
Dim width As Double = 4
Dim half As Double = width/2
Dim holeDiameter As Double = half
Line(-half, 0, -half, length)        ' Left side
Line( half, 0,  half, length)        ' Right side
Arc(0, length, half, 0, 180)         ' Top semi-circle (green)
Arc(0, 0, half, 180, 360)            ' Bottom semi-circle (blue)
Circle(0, length, holeDiameter/2)    ' Top hole
Circle(0, 0, holeDiameter/2)         ' Bottom hole
```



The properties of arc objects are as follows:

Data Type	Property	Access	Description
Double	Radius	get, set	Radius of arc.
Position	Center	get, set	Center of arc (in absolute coordinates).
Vector	AxisX	get	A unit vector along the X-axis of the arc (where angle = 0).
Vector	AxisY	get	A unit vector along the Y-axis of the arc (where angle = 90).
Vector	AxisZ	get	A unit vector along the Z-axis of the arc (normal to its plane).
double	StartAngle	get	Start angle (in degrees).
double	EndAngle	get	End angle (in degrees).
Position	StartPoint	get	Start-point of the arc (where angle = StartAngle).
Position	EndPoint	get	End-point of the arc (where angle = EndAngle).

Here is some example code showing the use of point, line, and arc properties:

```
c1 = Circle(0, 0, 4)          ' Creates a circle with radius 4 at the origin
p1 = New Position(1, 1)       ' Creates a position
p2 = New Position(4, 7)       ' Creates another position
myLine = Line(p1, p2)         ' Constructs a line between the two positions
v = myLine.Direction          ' Gets the direction vector of the line - (0.6, 0.8, 0)
c1.Center = p1 + 10*v         ' Moves the circle to the position (7, 9, 0)
c1.Radius = 5                 ' Change the radius to 5
Point(p1.X, p2.Y)             ' Creates a point at (1,7)
```

Splines

The SNAP functions for handling splines use a fairly conventional NURBS representation that consists of:

- Poles — An array of n 3D vectors representing poles (control vertices)
- Weights — An array of n weight values (which must be strictly positive)
- Knots — An array of $n + k$ knot values: $t[0], \ldots, t[n + k - 1]$

The order and degree of the spline can be calculated from the sizes of these arrays, as follows:

- Let n = number of poles = Poles.Length
- Let $npk = n + k$ = number of knots = Knots.Length

Then the order, k , is given by $k = npk - n$. Finally, as usual, the degree, m , is given by $m = k - 1$.

You may not be familiar with the "weight" values associated with the poles, since these are not very visible within interactive NX — you can see them in the Info function, but you can't modify them. So, in this case, the SNAP API actually gives you more power than interactive NX. Generally, the equation of a spline curve is given by a rational function (the quotient of two polynomials). This is why spline curves are sometimes known as NURBS (Non-Uniform Rational B-Spline) curves. If the weights are all equal (and specifically if they are all equal to 1), then some cancellation occurs, and the equation becomes a polynomial. The basic functions for creating splines are:

Function	Inputs and Creation Method
Spline(knots As Double[], poles As Position[], weights As Double[])	Rational spline from knots, poles, and weights.
Spline(knots As Double[], poles As Position[])	Polynomial spline from knots and poles.

We are using the same array notation as before, so Position[] means an array of positions, and Double[] means an array of double values.

There are also functions that allow you to create spline curves that interpolate (pass through) given points.

Function	Inputs and Creation Method
`SplineThroughPoints(points As Position[], nodes As Double[], knots As Double[])`	Spline with given nodes and knots passing through given points.
`SplineThroughPoints(points As Position[], degree As Integer)`	Spline with given degree passing through given points.
`SplineThroughPoints(points As Position[], startTangent As Vector, endTangent As Vector)`	Cubic spline interpolating given points and start and end tangents.

In the first form, note that you can specify the parameter values (the node values) at which interpolation will occur. So, specifically, the resulting spline S will satisfy S(nodes[i]) = points [i] for $0 < i \leq n$. Choosing different node values will make a big difference to the shape of the curve. In interactive NX, you have no control over these node values — the system just chooses some reasonable values for you.

Here is an example snippet of code:

```
Dim p0 As New Position(0,0)                                  ' Define some points (positions)
Dim p1 As New Position(1,2)
Dim p2 As New Position(2,5)
Dim p3 As New Position(3,7)
Dim myPoints As Position() = { p0, p1, p2, p3 }               ' Put the points into an array
Dim curve As NX.Spline = SplineThroughPoints(myPoints, 3)     ' Cubic spline through the points
```

Bezier Curves

A Bezier curve is just a spline that consists of only one segment. But, creating Bezier curves is often simpler than creating more general splines, so we provide these special functions for doing this. There is no specific object of type "Bezier curve", so the functions listed below simply produce NX.Spline objects as their output. The basic functions are:

Function	Result
`BezierCurve(ParamArray poles As Position[])`	Polynomial Bezier curve
`BezierCurve(poles As Position[], weights As Double[])`	Rational Bezier curve.

In the first function, the array of poles is marked with the word "ParamArray", which means that you can input a list of individual positions, rather than an array. The following code shows the two possible techniques:

```
Dim p1, p2, p3 As Position
p1 = {1, 0, 0}  :  p2 = {2, 1, 0}  :  p3 = {4, 2, 0}
Dim poleArray As Position() = { p1, p2, p3 }

BezierCurve(p1, p2, p3)    ' Using individual positions
BezierCurve(poleArray)     ' Using an array of positions (same result)

BezierCurve( {1, 0, 0}, {2, 1, 0}, {4, 2, 0} )    ' Yet another approach
```

There are also functions to create Bezier curves that interpolate (pass through) given points:

Function	Inputs and Creation Method
`BezierCurveThroughPoints( ParamArray intPoints As Position[])`	Bezier curve passing through given points
`BezierCurveThroughPoints( intPoints As Position[], nodes As Double[])`	Bezier curve passing through given points

The second function allows you to specify nodes (the parameter values at which interpolation will occur).

Using different nodes will make a big difference to the shape of the curve, as the following example illustrates:

```
Dim q0 As New Position (0,0)   :   Point(q0)
Dim q1 As New Position (4,1)   :   Point(q1)
Dim q2 As New Position (5,0)   :   Point(q2)
Dim intPoints As Position() = { q0, q1, q2 }     ' Three points we want our curve to pass through
Dim nodes1 As Double() = { 0, 0.4, 1 }           ' Parameter values to assign to the 3 points
Dim nodes2 As Double() = { 0, 0.8, 1 }           ' A different set of parameter values
BezierCurveThroughPoints(intPoints, nodes1)      ' Curve through the points q0, q1, q2
BezierCurveThroughPoints(intPoints, nodes2)      ' Another curve through q0, q1, q2
```

The results look like this:

You don't have this sort of control in interactive NX — the system just chooses the node values for you.

The properties of spline objects are as follows:

Data Type	Property	Access	Description
Position()	Poles	get, set	Array of 3D points representing poles (control points).
Double()	Weights	get, set	Array of weight values (these must be >0)
Double()	Knots	get	Array of knot values.
Integer	Degree	get	The degree of the spline, *m* (equal to order – 1).
Integer	Order	get	The order of the spline, *k* (equal to degree + 1).

Here is some example code:

```
Dim q0 As New Position(0,0)         ' Points we want our spline to pass through
Dim q1 As New Position(1,2)
Dim q2 As New Position(2,5)
Dim q3 As New Position(3,7)
Dim qpts As Position() = { q0, q1, q2, q3 }

Dim mySpline As NX.Spline
mySpline = SplineThroughPoints(qpts, 3)

Dim m As Integer = mySpline.Degree             ' Get the degree of the spline (3)
Dim pole0 As Position = mySpline.Poles(0)      ' Get first pole – will be (0,0,0)
Dim pole1 As Position = mySpline.Poles(1)      ' Get the second pole
```

It would be reasonable to assume that you can edit a spline by writing code like `mySpline.Poles(0) = {3,0,0}`. However, due to an unfortunate quirk in the .NET support for array properties, this does not work. You have to do this sort of thing, instead:

```
Dim myPoles As Position() = mySpline.Poles      ' Get the poles array
myPoles(0) = {3,0,0}                            ' Modify a pole (or several)
mySpline.Poles = myPoles                        ' Set the Poles property
```

So, as you can see, you can set the array of poles, but you can't set individual ones (not directly, anyway).

Chapter 8: Simple Solids and Sheets

This chapter briefly outlines the SNAP functions that are available for creating solid and sheet bodies.

Creating Primitive Solids

The Snap.Create class provides a variety of functions for creating simple solid primitives (blocks, cylinders, cones, spheres, and tori). These functions actually create features. For example, the Snap.Create.Block function creates an NX.Block object, which is interchangeable with NXOpen.Features.Block. The complete suite of functions is documented in the SNAP Reference Guide; the following is just a very small sample of what is available:

Function	Inputs and Creation Method
`Block(origin As Position, xLength As Number, yLength As Number, zLength As Number)`	Create a block feature aligned with the absolute coordinate system axes
`Cylinder(basePoint As Position, direction As Vector, length As Number, diameter As Number)`	Create a cylinder feature
`Cylinder(basePoint As Position, endPoint As Position, diameter As Number)`	Create a cylinder feature
`Sphere(x As Double, y As Double, z As Double, d As Number)`	Create a sphere feature
`Sphere(center As Position, d As Number)`	Create a sphere feature
`Torus(center As Vector, axis As Vector, a As Double, b As Double)`	Create a solid torus.

Most of the numerical parameters (xLength, diameter, etc.) are Number objects, so you can supply either Double or String values as inputs.

There is no Torus feature in NX, so the Snap.Create.Torus function actually creates a Revolve feature in the Work Part. Here's a program that produces a toy four-spoke steering wheel design using simple primitive solids:

```
Dim diameter, rimDiameter, a, b As Double
diameter = 300
rimDiameter = 40
a = diameter/2
b = rimDiameter/2
Dim origin As Position = Position.Origin
Dim rim, hub As NX.Feature
rim = Torus(origin, Vector.AxisZ, a, b)
hub = Sphere(origin, 2*b)
Dim spokes As NX.Cylinder() = new NX.Cylinder(3) {}
Spokes(0) = Cylinder(origin,  Vector.AxisX, a, b)
Spokes(1) = Cylinder(origin, -Vector.AxisX, a, b)
Spokes(2) = Cylinder(origin,  Vector.AxisY, a, b)
Spokes(3) = Cylinder(origin, -Vector.AxisY, a, b)
```

Extruded Bodies

The Snap.Create class provides several functions for creating extruded shapes. Each of these functions returns an NX.Extrude object, which is interchangeable with NXOpen.Features.Extrude. The complete suite of functions is documented in the SNAP Reference Guide; the following are just two typical examples:

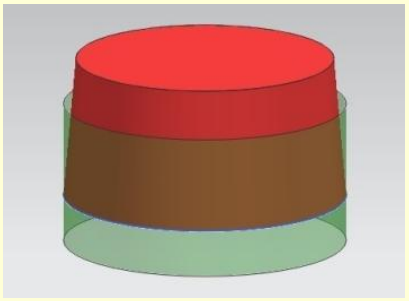
Function	Result
`Extrude(curves As ICurve[], axis As Vector, distances As Number[], draftAngle = null)`	Create an extruded feature (which might be a solid or a sheet)
`ExtrudeSheet(curves As ICurve[], axis As Vector, length As Number, draftAngle As Number)`	Create an extruded sheet feature

The inputs to the Extrude functions have the following meanings:

`curves`	An array of curves to be extruded (the "section"). These are actually "icurve" objects, which means they can either be wire-frame curves or edges of bodies.
`axis`	Extrusion direction. The magnitude of this vector is not significant
`distances/length`	Extents or length of the extrusion (measured from the input curves)
`draftAngle`	The draft angle, in degrees. This is an optional input, and if you don't supply it the draft angle will be zero. If the draft angle is positive, the cross-sectional shape will grow smaller as you travel in the direction of the axis vector.

Again, the numerical parameters are Number objects, so you can supply either Double or String inputs. Here are two simple examples showing the use of the Extrude functions:

```
Dim section As NX.Arc = Circle(0, 0, 3)
Dim axis As Vector = Vector.AxisZ
Dim length As Double = 3
Dim extents As Number() = { -1, 2 }
Dim draft As Double = 5

Dim e1, e2 As NX.Extrude
e1 = Extrude( {section}, axis, length, draft)      ' Solid
e2 = ExtrudeSheet( {section}, axis, extents)       ' Sheet

e1.Color = System.Drawing.Color.Red
e2.Color = System.Drawing.Color.Green
e2.Translucency = 80
```



Revolved Bodies

The Snap.Create class provides several functions for creating revolved shapes. Each of these functions returns an NX.Revolve object, which is interchangeable with NXOpen.Features.Revolve. The complete suite of functions is documented in the SNAP Reference Guide; the following are just two typical examples:

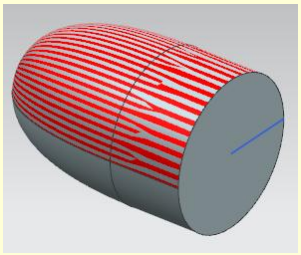
Function	Result
`Revolve(curves As ICurve[], axisPoint As Position, axisVector As Vector)`	Create a complete 360 degree revolved feature
`RevolveSheet(curves As ICurve[], axisPoint As Position, axisVector As Vector, angles As Number[])`	Create a revolved sheet feature

The meanings of the inputs are as follows

`curves`	An array of curves or edges to be revolved (the "section")
`axisPoint`	Point on the axis of revolution
`axisVector`	Vector along the axis of revolution (magnitude doesn't matter)
`angles`	Angular extents of the revolved shape, in degrees, measured from the input curves

Here are two simple examples showing the use of the Revolve functions:

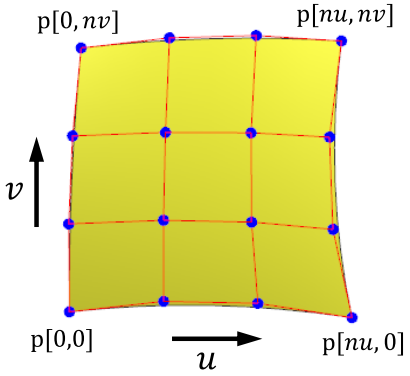
```
Dim c1, c2, c3 As NX.Curve
c1 = BezierCurve({0, 0, 0}, {0, 1, 0}, {2, 1, 0})
c2 = Line(2, 1, 0, 3, 1, 0)
c3 = Line(3, 1, 0, 3, 0, 0)
Dim section As NX.Curve = { c1, c2, c3 }
Dim axisPoint As Position = Position.Origin
Dim axisVector As Vector = Vector.AxisX
Dim angles As Number() = { 0, 180 }

Revolve(section, axisPoint, axisVector)                 ' Solid
RevolveSheet( {c1, c2}, axisPoint, axisVector, angles)  ' Sheet
```



B-surfaces

The b-surface representation we use in SNAP is very similar to the spline representation shown earlier. We have:

- Poles — A 2D array of $nu \times nv$ 3D positions representing poles
- Weights — A 2D array of $nu \times nv$ weight values
- KnotsU — An array of $nu + ku$ knot values: $u[0], \ldots, u[nu + ku - 1]$
- KnotsV — An array of $nv + kv$ knot values: $v[0], \ldots, v[nv + kv - 1]$



Again, as with splines, the orders and degrees of the surface can be inferred from the sizes of the pole and knot arrays. Also, note how the poles are arranged with respect to the surface parameterization:

- Poles p[0, 0], p[0, 1], ... ,p[0, nv] lie along edge $u = 0$ ($0 < v < 1$)
- Poles p[0, 0], p[1, 0], ... ,p[nu, 0] lie along edge $v = 0$ ($0 < u < 1$)

So, in particular, the poles at the corners of the surface are:

- p[0, 0] = S(0, 0)
- p[0, nv] = S(0, 1)
- p[nu, 0] = S(1, 0)
- p[nu, nv] = S(1, 1)

If weights are not specified, or they are all equal, the result is polynomial surface; otherwise it is a rational surface. The basic functions for creating b-surfaces are:

Function	Result
`Bsurface(poles As Position[,], knotsU As Double[], knotsV As Double[])`	Polynomial b-surface
`Bsurface(poles As Position[,], weights As Double[,]` `knotsU As Double[], knotsV As Double[])`	Rational b-surface

A Bezier patch is just a b-surface that consists of only one patch. But, creating Bezier patches is often simpler than creating more general b-surfaces, so we provide special functions for doing this. There is no specific object of type "Bezier patch", so the functions listed below simply produce NX.Bsurface objects as their output.

The basic functions are:

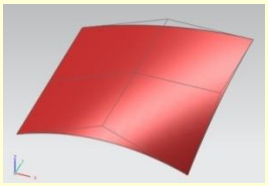
Function	Result
`BezierPatch(poles As Position[,])`	Polynomial Bezier patch
`BezierPatch(poles As Position[,], weights As Double[,])`	Rational Bezier patch

Instead of creating a b-surface from poles, you may wish to specify an array of points through which the surface should pass. SNAP provides functions called BsurfaceThroughPoints and BezierPatchThroughPoints to help you do this.

Function	Result
`BsurfaceThroughPoints(intPoints As Position[,], nodesU As Double[], nodesV As Double[], knotsU As Double[], knotsV As Double[])`	Bsurface through points
`BsurfaceThroughPoints(intPoints As Position[,], degreeU As Integer, degreeV As Integer`	Bsurface through points
`BezierPatchThroughPoints (intPoints As Position[,], nodesU As Double[], nodesV As Double[])`	Bezier patch through points
`BezierPatchThroughPoints(intPoints As Position[,])`	Bezier patch through points

Here is some example code that creates a simple B-surface (a Bezier patch, actually):

```
Dim p As Position(,) = New Position(2,2) {}
Dim h As double = 0.4
p(0,0) = {0,0,0}  :  p(0,1) = {0,1,0}  :  p(0,2) = {0,2,0}
p(1,0) = {1,0,h}  :  p(1,1) = {1,1,h}  :  p(1,2) = {1,2,h}
p(2,0) = {2,0,0}  :  p(2,1) = {2,1,h}  :  p(2,2) = {2,2,h}

BezierPatch(p)
```



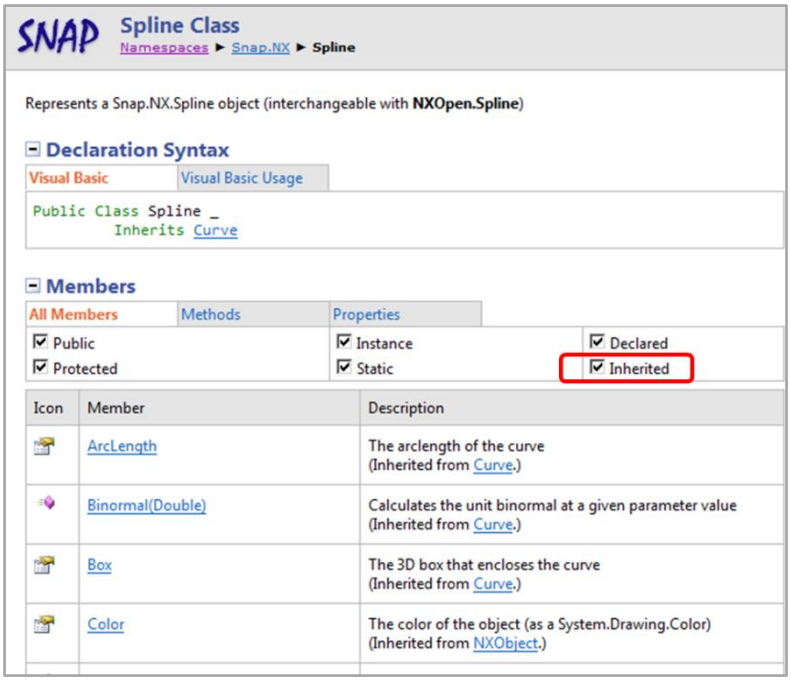
In addition to the simple B-surface creation functions listed above, there are several other SNAP functions that create b-surface geometry; the ThroughCurves and ThroughCurveMesh features are the most important examples, and, of course, you can find these described in the SNAP Reference Guide.

Chapter 9: Object Properties & Methods

The objects in the Snap.NX namespace have a rich set of properties that let us get information about the objects and (in some cases) modify them. The complete properties of each object are documented in the SNAP Reference Guide, so the overview provided here is just to help you understand the basic concepts.

As we mentioned in chapter 4, objects inherit properties from the parent classes from which they are derived, in addition to having properties of their own. So, for example, because NX.Arc inherits from NX.Curve, which in turn inherits from NX.NXObject, an NX.Arc object has all the properties of an NX.Curve and all the properties of an NX.NXObject, in addition to specific properties of its own.

In the SNAP Reference Guide, you can control whether or not inherited members are displayed by clicking in the check-box circled in red below:



As you can see, there are three members that NX.Spline inherits from NX.Curve, and one that it inherits from NX.NXObject. All three of these will be hidden if you uncheck the "inherited" box.

NXObject Properties

The NX.NXObject class is the highest level in the SNAP object hierarchy, so its properties are very important because they trickle down to all the lower-level objects. The properties can be divided into several categories, as outlined below:

Type and SubType Properties

Each SNAP object has an ObjectType property and an ObjectSubType property, which you will often use to make decisions about how to process the object. These properties are read-only, of course — you can not change the type of an object.

Data Type	Property	Access	Description
`Snap.NX.ObjectTypes.Type`	`ObjectType`	`get`	The object's type
`Snap.NX.ObjectTypes.SubType`	`ObjectSubType`	`get`	The object's subtype

Suppose the user has selected an object, for example. You might want to test whether this object is an ellipse before processing it.

The code to do this would be as follows:

```
Dim thing As NX.NXObject = ...
Dim myType    As Snap.NX.ObjectTypes.Type =  thing.ObjectType
Dim mySubType As Snap.NX.ObjectTypes.SubType =  thing.ObjectSubType
If myType = NX.ObjectTypes.Type.Conic And mySubType = NX.ObjectTypes.SubType.ConicEllipse Then
    'Do something
End If
```

You can reduce the typing by putting `Imports NX.ObjectTypes` at the top of your file. In some cases, it might be more convenient to test the type of an object using the standard Visual Basic `TypeOf` function. For example, the code above could be written as:

```
Dim thing As NX.NXObject = ...
If TypeOf thing Is Snap.NX.Ellipse
    'Do something
End If
```

Display Properties

The display-related properties of an NX.NXObject are as follows:

Data Type	Property	Access	Description
Integer	Layer	get, set	The layer on which the object resides
Boolean	IsHidden	get, set	If true, indicates that the object is hidden (blanked)
Color	Color	get, set	The color of the object (as a System.Drawing.Color)
Integer	LineFont	get, set	The line font used to draw the object (solid, dashed, etc.)
Integer	LineWidth	get, set	The line width used to draw the object
Integer	Translucency	get, set	The translucency of the object (from 0 to 100)

The following code illustrates the use of these properties:

```
'Create a circle
Dim axis As New Vector(1,2,5)
Dim disk As NX.Arc = Circle(Position.Origin, axis, 100)

' Change its color, linefont and linewidth
disk.Color = System.Drawing.Color.Blue
disk.LineFont = Globals.Font.Dashed
disk.LineWidth = Globals.Width.Thin

' Create a translucent enclosing box
Dim box As NX.Block = Block(Orientation.Identity, disk.Box.MinXYZ, disk.Box.MaxXYZ)
box.Color = System.Drawing.Color.Yellow
box.Translucency = 50

' Hide (blank) the circle
disk.IsHidden = True

' Move the box (block) to layer 200
box.Layer = 200
```

Note that the Color attribute is a standard .NET System.Drawing.Color. The Snap.Color class has some functions for correlating traditional NX colors with System.Drawing colors. Also, note that we can change the color of an NX.Block object, even though it is a feature, which is not a displayable object. This topic is explained further in chapter 10.

Attribute Properties

For technical reasons, attributes cannot be implemented as "real" properties, so they are accessed via old-fashioned "Get" and "Set" functions. A few of the available functions are listed below, and the complete set is documented in the SNAP Reference Guide:

Function	Description
`DeleteAttributes(AttributeType)`	Deletes all attributes of a given type
`GetAttributeInfo()`	Gets an array of AttributeInformation structures
`GetAttributeStrings()`	Get the object's attributes as strings
`GetIntegerAttribute(String)`	Returns the value of an attribute of type "Integer"
`GetStringAttribute(String)`	Returns the value of an attribute of type "String"
`SetBooleanAttribute(String, Boolean)`	Creates and/or sets the value of an attribute of type "Boolean"
`SetDateTimeAttribute(String, DateTime)`	Creates and/or sets the value of an attribute of type "Time"
`SetRealAttribute(String, Double)`	Creates and/or sets the value of an attribute of type "Real"
`Name`	The name of the object (aka "custom name", sometimes)
`NameLocation`	The position at which the name of the object is displayed

NXOpen Connection Properties

Every SNAP object wraps or "encloses" a corresponding NXOpen.TaggedObject object and is associated with one or more NXOpen.DisplayableObject objects. The SNAP object has properties that let you access the corresponding NX Open objects, as follows:

Data Type	Property	Access	Description
`NXOpen.TaggedObject`	`NXOpenTaggedObject`	`get`	The enclosed NXOpen.TaggedObject
`NXOpenTag`	`NXOpenTag`	`get`	The tag of the NXOpen.TaggedObject
`NXOpen.DisplayableObject[]`	`NXOpenDisplayableObjects`	`get`	Array of associated displayable objects
`NXOpen.DisplayableObject`	`NXOpenDisplayableObject`	`get`	The first element of the array

In simple cases, the NXOpenDisplayableObjects array has only a single element, which is the same as the NXOpenTaggedObject. You can refer to this single object either as NXOpenDisplayableObjects[0] or as NXOpenDisplayableObject. Things are more complex when dealing with features, since a single feature may correspond to several displayable objects. For example, an Extrude feature sometimes contains several bodies, a Split feature always contains several bodies, and a PointSet feature would typically contain several points. This is explained further in chapter 10.

In addition, each SNAP object has a property that returns its enclosed NX Open object in a variable of the right type (rather than returning a generic NXOpen.TaggedObject). This is convenient because it removes the need for casting the NXOpen.TaggedObject. So, for example, you can write:

```
Dim snapLine As NX.Line = Line(0, 2, 3, 5, 7, 1)        ' A Snap.NX.Line object
Dim nxopenLine As NXOpen.Line = snapLine.NXOpenLine     ' Its enclosed NXOpen.Line object
```

In most situations, you won't need to access the enclosed NX Open objects, because (for the most part) the Snap.NX object and the NX Open object can be used interchangeably. But, from time to time, you may find this technique to be useful. The last section in chapter 16 has more information on this subject.

Curve and Edge Properties

In SNAP, wire-frame curves and edges are both represented by an object called an ICurve. Technically, an ICurve is an "interface" that is implemented by both the Snap.NX.Curve class and the Snap.NX.Edge class, but you don't need to understand the details of this — just remember that an ICurve can represent either a curve or an edge. You will find that most SNAP functions use ICurves as input, rather than curves or edges, so, as far as possible, curves and edges can be used interchangeably. Referring to "curves or edges" all the time gets tiresome, and "ICurve" is an unfamiliar term to many people, so often we just say "curve" when we really mean "ICurve" or "curve or edge".

Evaluators

Some of the most useful methods when working with ICurves (either curves or edges) are the so-called "evaluator" functions. At a given location on a curve (defined by a parameter value u), we can ask for a variety of different values, such as the position of the point, or the tangent or curvature of the curve. The evaluators available in SNAP are as follows:

Returns	Function	Value calculated
`Position`	`Position(u As Double)`	Point on the curve or edge
`Vector`	`Derivative(u As Double)`	First derivative vector
`Vector[]`	`Derivatives(u As Double, n As Integer)`	Curve derivatives up to order n (plus position)
`Vector`	`Tangent(u As Double)`	Unit tangent vector
`Vector`	`Normal(u As Double)`	Unit normal
`Vector`	`Binormal(u As Double)`	Unit binormal
`Double`	`Curvature(u As Double)`	Curvature
`Double`	`Parameter(p As Position)`	Parameter at position p (on or near the curve)

In other software systems, a common approach is to "normalize" the parameter value (u) that is passed to these sorts of evaluator functions, so that it lies in the range $0 \le u \le 1$. But the constant normalizing and denormalizing of parameter values can be tedious and confusing, so we never do this in SNAP. In the SNAP approach, each curve has a minimum parameter value, MinU, and a maximum parameter value, MaxU, and you should not assume that MinU = 0 or MaxU = 1. Actually, for lines and splines it **is** always true that MinU = 0 and MaxU = 1, but this is not the case for circles, ellipses, and a few other edge/curve types. To avoid confusion, if you want information about the point half-way along a curve, you should always use u = 0.5*(MinU + MaxU). Here are some examples to illustrate the usage:

```
myCircle = Circle(3, 5, 2)              ' Circle of radius 2 with center at (3,5,0)
point90 = myCircle.Position(90)         ' Point at 90 degrees – will be (3,7,0)
tang90  = myCircle.Tangent(90)          ' Tangent at 90 degrees – will be (-1,0,0)
deriv0  = myCircle.Derivative(0)        ' First derivative at start – will be (0, 0.0349066, 0)
curv    = myCircle.Curvature(0.83)      ' Curvature at any point will be 0.5
```

You may have to think a little, or dig out your old calculus books, to understand the result we got for the derivative vector at the start point (hint: the y-component is $2\pi/180$). Don't worry about this. The main point here is that the derivative vector does ***not*** necessarily have unit length.

Edge Topology Properties

The main difference between an edge and a curve, of course, is that an edge is part of a body, whereas a curve is not. Because of this, an edge has "topological" properties that a curve does not have, which describe how the edge is connected to other items (faces, edges, vertices) within the body. You can do basic topology enquiries using the Snap.NX.Face and Snap.NX.Edge objects, but, for more advanced applications, please read about the Snap.Topology namespace in the SNAP Reference Guide. It provides Loop, Fin, and Vertex objects that allow you to get much more detailed information about object topology.

Edge Geometry Properties

To obtain geometric information about an edge, you first get its Geom object, and then use the properties of the Geom object. For example, the following code gets the radius of a circular edge:

```
Dim edge1 As NX.Edge = myBody.Edges(0)                      ' Get an edge
Dim arc1 As NX.Edge.Arc = CType(edge1, Snap.NX.Edge.Arc)    ' Cast to Edge.Arc
Dim arcGeom As Snap.Geom.Curve.Arc = arc1.Geometry          ' Get its geometry
Dim r = arcGeom.Radius                                      ' Get radius
```

Face Properties

Like edges, faces have both evaluator functions, topological properties, and geometric properties.

Evaluators

As with curves, we can call "evaluator" functions to calculate certain values at a given point on a surface (or a face). So, as you might expect, we can call functions to obtain the location of the point, the surface normal at the point, and so on. To indicate which point we're interested in, we have to give two parameter values, traditionally denoted by u and v. Here are the values we can calculate at a point on a face:

Returns	Function	Value calculated
`Position`	`Position(u As Double, v As Double)`	Point on surface
`Vector`	`Normal(u As Double, v As Double)`	Unit surface normal
`Vector`	`DerivDu(u As Double, v As Double)`	Partial derivative wrt u
`Vector`	`DerivDv(u As Double, v As Double)`	Partial derivative wrt v
`Double[]`	`Parameters(p As Position)`	Parameters at position p (on or near surface)

To demonstrate the use of these functions, assume we have already defined an array of poles, which we will use to build a Bezier patch:

```
Dim patch As NX.Body = BezierPatch(poles)
Dim myFace As NX.Face = patch.Faces(0)      ' Get the face of the patch body
Dim p1, p2 As Position
Dim n1, n2, derU, derV As Vector
p1 = myFace.Position(0,0)                   ' Point at one corner of face
n1 = myFace.Normal(1,1)                     ' Surface normal at opposite corner
p2 = myFace.Position(0.4, 0.5)              ' Point roughly in middle of face
derU = myFace.DerivDu(1,1)                  ' Partial derivative wrt u at corner
derV = myFace.DerivDv(1,1)                  ' Partial derivative wrt v at corner
n2 = Vector.Cross(derU, derV)               ' Cross product – will be parallel to n1 above
```

Face Topology Properties

Like an edge, a face has "topological" properties that describe its relationship to other objects in its body. If myFace is a Snap.NX.Face object, then myFace.Body is the body that the face lies on, and myFace.Edges is its array of edges. More detailed information can be obtained by using the functions in the Snap.Topology namespace.

Face Geometry Properties

To get information about the geometry of a face, you first get its Geom object, and then use the properties of the Geom object. For example, the following code gets the radius of a cylindrical face:

```
' Create an extrusion
Dim profile As NX.Arc = Arc( {0,0,0}, {2,1,0}, {4,3,0} )
Dim sheet As NX.Extrude = Extrude( {profile}, Vector.AxisZ, 5)

' Get the face of the extrusion, and cast to NX.Face.Cylinder
Dim cylinderFace As NX.Face.Cylinder = CType(sheet.Faces(0), NX.Face.Cylinder)

' Get the Geom (a Geom.Cylinder object)
Dim cylinderSurface As Geom.Surface.Cylinder = cylinderFace.Geometry

' Get the diameter and radius
Dim r As Double = cylinderSurface.Diameter/2

' Get the axis direction of the cylinder
Dim axis As Vector = cylinderSurface.AxisVector
```

Chapter 10: Feature Concepts

The Snap.Create class contains a wide variety of functions for creating “features”. At one extreme, features can be very simple objects like blocks or spheres; at the other extreme, features like ThroughCurveMesh can be quite complex. In this chapter, we explain what a feature is, and give some samples of the SNAP functions that create them. As usual, the full details can be found in the SNAP Reference Guide.

What is a Feature ?

Though you have probably created hundreds of features while running NX interactively, perhaps you never stopped to think what a “feature” really is. So, here is the definition ...

A feature is a collection of objects created by a modeling operation that remembers the inputs and the procedure used to create it.

The inputs used to create the feature are called its “parents”, and the new feature is said to be the “child” of these parents. This human family analogy can be extended in a natural way to provide a wealth of useful terminology. We can speak of the grandchildren or the ancestors or the descendants of an object, for example, with the obvious meanings. An object that has no parents (or has been disconnected from them) is said to be an “orphan”, or sometimes a “dumb” object, or an “unparameterized” one.

The inputs and the procedure are also known as the “history” of the object, or the “recipe”, or the “parameters”. There is no shortage of terminology in this area.

Features Versus Bodies

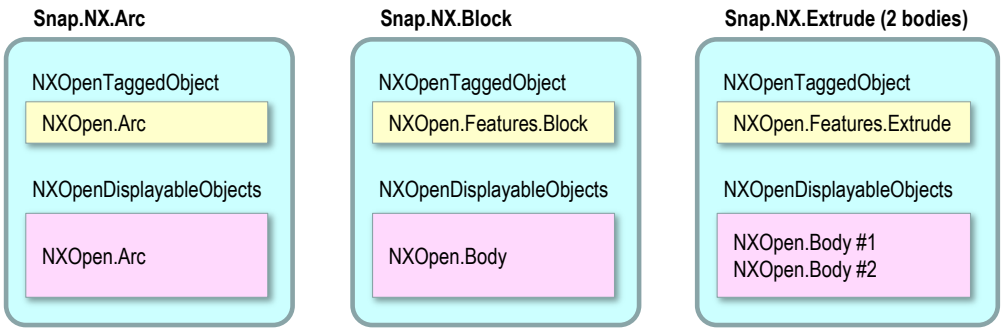
Like most other objects in SNAP, the NX.Feature class is derived from NX.NXObject, so it inherits all the NX.NXObject properties described in chapter 9. For our discussion here, the most important of these properties are the NXOpenTaggedObject property and the NXOpenDisplayableObjects array.

A simple object like an arc or a spline is itself a “displayable” object in NX, which means it has color, a hidden/shown property, and other display attributes. So, in a Snap.NX.Arc object, the NXOpenDisplayableObjects array has a single entry, which is the same as the NXOpenTaggedObject (which is an NXOpen.Arc).

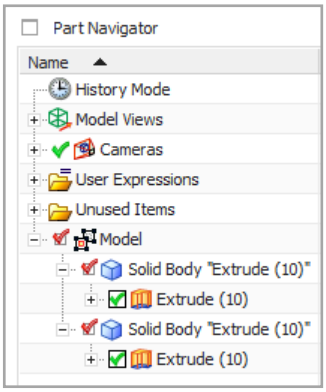
In the case of a simple feature like a Snap.NX.Block, the NXOpenTaggedObject is an NXOpen.Features.Block, and the NXOpenDisplayableObjects array has a single entry, which is an NXOpen.Body.

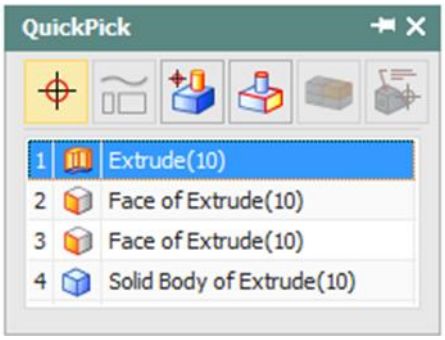
The most complex case is a feature that produces several bodies. To illustrate how this works, let’s consider a specific example. Suppose we create two circles, and use them to make a single Extrude feature. Obviously two bodies will be created. In the NX.Extrude feature, the NXOpenTaggedObject is an NXOpen.Features.Extrude, and the NXOpenDisplayableObjects array will have two entries, each of which is an NXOpen.Body.

The following diagram illustrates the three cases



In interactive NX, you can see the distinction between a feature and its bodies in the Part Navigator (if you turn off TimeStamp Order) and in the QuickPick window. Here is what you will see for our two-body Extrude example:





Feature Display Properties

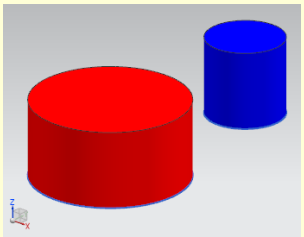
In the NX.Arc example described above, it's fairly obvious how display properties like color should be handled — they are simply attached to the enclosed NXOpen.Arc object. The NX.Block example is also fairly straightforward. There is a minor issue because a feature is not a displayable object, but the Block feature consists of a single body, which **is** a displayable object, so we can use this to hold the display properties of the feature. The third example (an NX.Extrude feature with two bodies) is the only one that's slightly complicated. Here, we have two bodies, and each of them has its own display properties. So, what happens when I manipulate the color of this Extrude feature? The answers are:

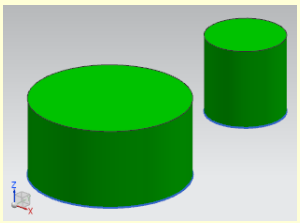
- If you set the color of the feature, this will set the color of each body in the feature
- If you get the color of the feature, you'll get the color of the first body in the feature (which is unpredictable)

Other display properties, like line-width, hidden/shown status and layer are handled the same way.

So, in SNAP code, although it appears that you can manipulate the display properties of features, what's really happening is that you are indirectly manipulating the properties of the underlying displayable objects. When the feature consists of a single body, the SNAP functions do exactly what you would expect, and the scheme is very convenient. When a feature has several bodies, you may have to think a little, sometimes. The following code illustrates the situation:

```
'Create two circles, and extrude them
Dim disk0 As NX.Arc = Circle(0, 0, 2)     ' Center at (0,0), radius = 2
Dim disk1 As NX.Arc = Circle(0, 5, 1)     ' Center at (0,5), radius = 1
Dim pegs As NX.Extrude = Extrude( {disk0, disk1}, Vector.AxisZ, 2)

'Get the two displayable objects of the Extrude feature (two bodies)
Dim b0 As NX.Body = CType(pegs.NXOpenDisplayableObjects(0), NX.Body)
Dim b1 As NX.Body = CType(pegs.NXOpenDisplayableObjects(1), NX.Body)

'Change the colors of these two bodies
b0.Color = System.Drawing.Color.Red
b1.Color = System.Drawing.Color.Blue

' Get the color of the feature (color of first body)
Dim pegColor As System.Drawing.Color = pegs.Color

' Make the feature green (makes both bodies green)
pegs.Color = System.Drawing.Color.Green
```





The code above was deliberately written in a rather roundabout way to illustrate the role of the NXOpenDisplayableObjects array. In practice, it would be much simpler to write b0 = pegs.Bodies(0), and b1 = pegs.Bodies(1) rather than doing the convoluted conversion shown above.

More Feature/Body Confusion

As we saw above, when working with display properties, it is useful to blur the distinction between a feature and its constituent bodies, especially in the common case where the feature has only one body. We will see here that this blurring is also convenient in modeling.

There are many modeling and computation functions that expect to receive bodies as input. Examples are Boolean operation functions, trimming, splitting, computing mass properties, and so on. Since most of the basic creation functions produce features, the output of these functions will not be immediately usable, unless we make some accommodation. For example, consider the following code:

```
Dim s1 As NX.Sphere = Sphere(0,0,0, 2)
Dim s2 As NX.Sphere = Sphere(1,0,0, 1)
Dim cut As NX.Boolean = Subtract(s1, s2)
Dim volume As Double = Compute.Volume(cut)
```

The Subtract function expects two bodies as input, but s1 and s2 are features, so we would not expect the Subtract to work. Similarly, the Volume function expects to receive a body, so we would not expect this to work, either. We could fix the code, of course, by getting bodies from the features before calling Subtract and Volume, but this would make our code harder to write and harder to understand. It would be much better if the code shown above just worked, without any further fuss. To make this possible, we again confuse features and their bodies. Inside SNAP, there is an implicit conversion that silently converts a feature to a body whenever necessary. For the sake of safety, this is only done if the feature consists of a single body; an exception will be raised if you try to use a multi-body feature someplace where a body is expected.

Feature Parameters — the Number Class

When creating features, it is sometimes desirable to use Strings as function arguments to represent numerical quantities like lengths and diameters. This allows expressions to be used to define feature parameters, which means we can establish numerical relationships between features, and make them easier to edit interactively.

On the other hand, in typical programs, we would expect numerical quantities to be represented by Double variables, rather than strings. So, at first, it appears that two creation functions might be needed for each feature — one receiving Strings as input, and one receiving Double values. So, to define a Sphere feature, for example, we would have these two functions

- `Snap.Create.Sphere(center As Position, diameter as String)` — diameter given as a String, and
- `Snap.Create.Sphere(center As Position, diameter as Double)` — diameter given as a Double

This would cause a huge amount of duplication, so we need some way around the problem. In SNAP, the solution is to use a "Number" object that can represent either a Double or a String. So, in place of the two functions listed above, we have just one function for creating a sphere

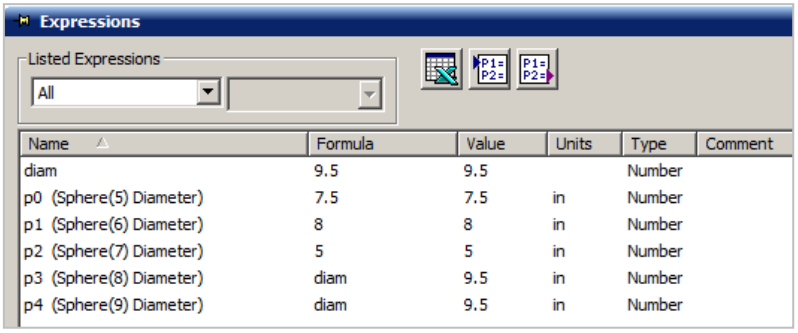
- `Snap.Create.Sphere(center As Position, diameter as Number)` — diameter given as a Number object

It would be very inconvenient if we had to explicitly convert Doubles and Strings into Number objects to create features, so this conversion is done silently and implicitly, behind the scenes. Typical code looks like this:

```
s1 = Sphere(center, 7.5)            ' Double 7.5 converted to Number
s2 = Sphere(center, 8)              ' Integer 8 converted to Number
s3 = Sphere(center, "5")            ' String "5" converted to Number (pointless)

Dim diamString As String = "diam"
Expression(diamString, 9.5)

s4 = Sphere({0,0,20}, diamString)   ' diamString converted to Number
s5 = Sphere({0,0,50}, diamString)   ' diamString converted to Number
```

In the creation of s1 and s2, it looks as if we are just passing a Double and an Integer to the Sphere function — we don't have to think about Number objects at all. The same is true in the creation of s3, where a string gets silently converted to a Number. The major benefit is shown in the creation of spheres s4 and s5.

Here we have used the same string variable, diamString as the diameter of both spheres, which means they are both now controlled by a common expression, as shown below:



Name	Formula	Value	Units	Type	Comment
diam	9.5	9.5		Number	
p0 (Sphere(5) Diameter)	7.5	7.5	in	Number	
p1 (Sphere(6) Diameter)	8	8	in	Number	
p2 (Sphere(7) Diameter)	5	5	in	Number	
p3 (Sphere(8) Diameter)	diam	9.5	in	Number	
p4 (Sphere(9) Diameter)	diam	9.5	in	Number	

This makes it easy to edit these two spheres in a coordinated way, either programmatically or interactively.

There are a few functions that receive an array of Number objects as input. Unfortunately, we cannot use the implicit behind-the-scenes conversion trick to convert an array of Doubles into an array of Number objects, so, in this case, you have to actually declare variables as Number objects. Consider the following code:

```
Dim shape As NX.Line() = Rectangle( {0,0,0}, {4,6,0})
Dim axis As Vector = Vector.AxisZ

Dim doubleDistances As Double() = { -1, 5 }
Extrude(shape, axis, doubleDistances)             ' Doesn't even compile

Dim numberDistances As Number() = { -1, 5 }
Extrude(shape, axis, numberDistances)             ' Works fine
```

Trying to create an Extrude using the array `doubleDistances` won't work — the compiler will complain that it can't convert an array of Doubles to an array of Numbers. You have to use `numberDistances`, instead.

More Feature Functions

The Snap.Create class has several dozen functions for creating features. These include:

- Simple primitive solids (block, cylinder, cone, sphere, torus)
- Extrude and revolve features
- Free-form features like ThroughCurves and ThroughCurveMesh
- Edge and face blends
- Boolean, sew, trim, and split
- Offsetting and Thickening
- Datum axes, datum planes and datum coordinate systems

In addition, there are several other functions for working with features. For example, you can:

- Find all the bodies, faces, or edges in a feature
- Control whether or not a feature is suppressed
- Find the parents of a feature

As usual, the complete details are given in the SNAP Reference Guide.

History-Free Mode

If you try to execute a SNAP function that creates a feature while NX is in History-Free mode, you will receive an error message. The error details are described further in chapter 17. To avoid this problem, your code should set Snap.Globals.HistoryMode to True before creating any features, like this:

```
Globals.HistoryMode = True
Dim cubeFeature As NX.Block = Block(Position.Origin, 10, 10, 10)
Dim cubeBody As NX.Body = cubeFeature.Body
Globals.HistoryMode = False
Dim faces As NX.Face() = cubeBody.Faces
cubeBody.Color = System.Drawing.Color.Red
```

The code above switches HistoryMode back to False after creating the Block feature. This will destroy the Block feature, and so the cubeFeature variable will no longer be usable. On the other hand, the cubeBody variable was assigned a value before setting HistoryMode = False, and bodies are not affected by this switch, so this variable's value is still usable in subsequent code. So, for example, we can get the faces of the cubeBody, or change its color, as shown.

Chapter 11: Assemblies

Introduction

Unless you're in the brick business, most of your products will probably be assemblies — combinations of simpler lower-level items, rather than just homogeneous hunks of material. This chapter outlines how NX represents assemblies, and describes the SNAP functions that you can use to work with them. The SNAP functions that are available at present are mostly focused on reading information about assemblies, rather than creating them, so their main use is for extracting information and writing reports of one sort or another. Typically, your code will traverse though the items in an assembly, gathering information (from attributes, usually), and writing this into a report document of some kind.

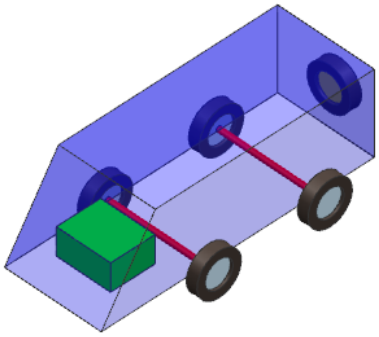
Many of the code examples given below are just fragments, as usual. Complete working code and the part files for a simple car assembly are provided in the folder [...NX]\UGOPEN\SNAP\Examples\More Examples\CarAssembly.

Note that some of the code in this chapter will work properly only if the car assembly is fully loaded.

The Obligatory Car Example

Following the time-honored traditions of assembly modeling, we will use a simple car as an example throughout this chapter (though this version looks more like a van, actually).

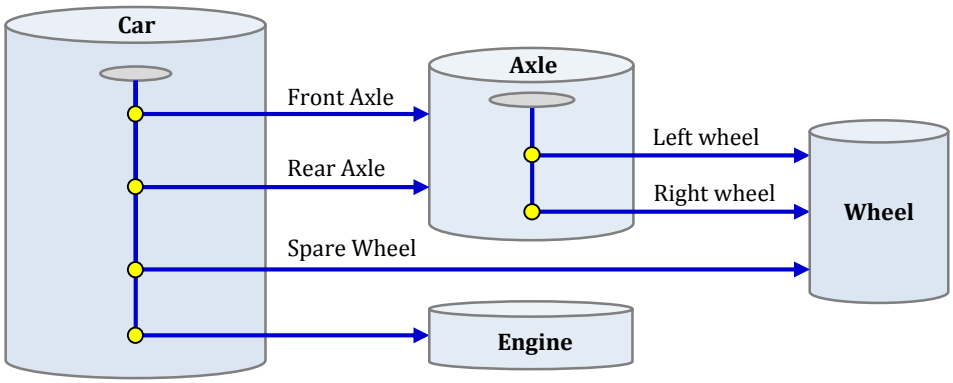
Descriptive Part Name	Component Name	Part Name
Sections		
Car_Assembly		Car_Assembly
Wheel_Part	SPARE_WHEEL	Wheel_Part
Engine_Part	ENGINE	Engine_Part
Axle_Assembly	REAR_AXLE	Axle_Assembly
Wheel_Part	REAR_RIGHT_WHEEL	Wheel_Part
Wheel_Part	REAR_LEFT_WHEEL	Wheel_Part
Axle_Assembly	FRONT_AXLE	Axle_Assembly
Wheel_Part	FRONT_RIGHT_WHEEL	Wheel_Part
Wheel_Part	FRONT_LEFT_WHEEL	Wheel_Part



As you can see, the car consists of an engine (the green block), an exterior shape (the blue thing), two axles, and a spare wheel. Each axle consists of a shaft and two wheels. The exterior shape is a sheet body in Car_Assembly.prt, so you don't see it in the Assembly Navigator. Similarly, the red shaft is a solid body in Axle_Assembly.prt, so you don't see this, either.

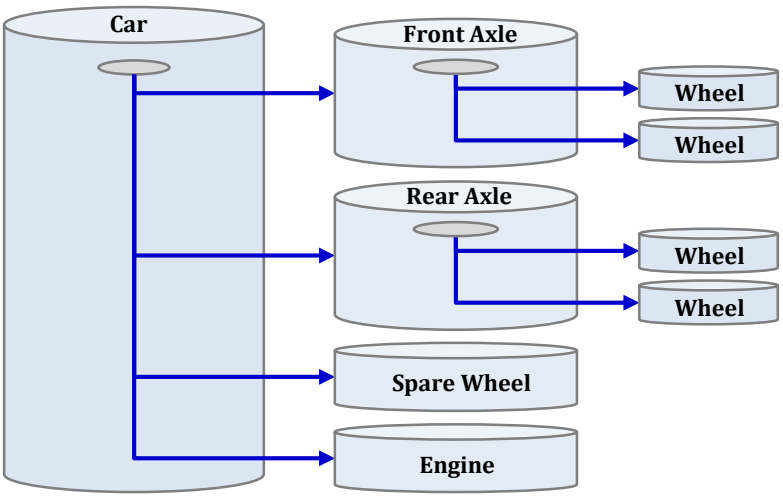
Trees, Roots, and Leaves

Let's use our car model to explain some terminology. Graphically, its structure looks like this:



This diagram accurately reflects the structure of the data stored in NX. Notice that the wheel part is stored only once, even though the car has five wheels (the four main ones and a spare).

However, diagrams like this are difficult to draw, in more complex situations, so we will usually draw them as shown below, instead, with items repeated:



The top-level car assembly has four subassemblies: two axles, a spare wheel, and an engine. The axle assembly, in turn, has two subassemblies, namely its left and right wheels. In this situation, the axles, spare wheel and engine are said to be **children** of the car assembly. Or looking at it the other way around, the car assembly is said to be the **parent** of each of these four. This human-family terminology can be extended further: we might say that each of the four main wheels is a grandchild of the car assembly, and all the parts shown are **descendants**, and so on.

Note that this is the reverse of feature terminology. In the feature modeling world, if object-A and object-B are constituents of object-C (in the sense that they are used to create object-C), then they are called the parents of object-C, not its children. This inconsistency is unfortunate, but it's very well established, and is not likely to change, so we have to live with it.

In addition to the parent-child terminology, there are some useful terms that we can borrow from computer science. A computer scientist would regard the assembly structure as a **tree**, and the various parts and assemblies would be called the **nodes** in the tree. The node at the top of a sub-tree (denoted by the  symbol in the diagram) is called the **root** node of that sub-tree. Nodes at the bottom (like the wheels and engine) are called **leaf** nodes; these are easy to identify because they have no children. Trees in computer science are strange — their roots are always at the top, and their leaves are at the bottom ☺.

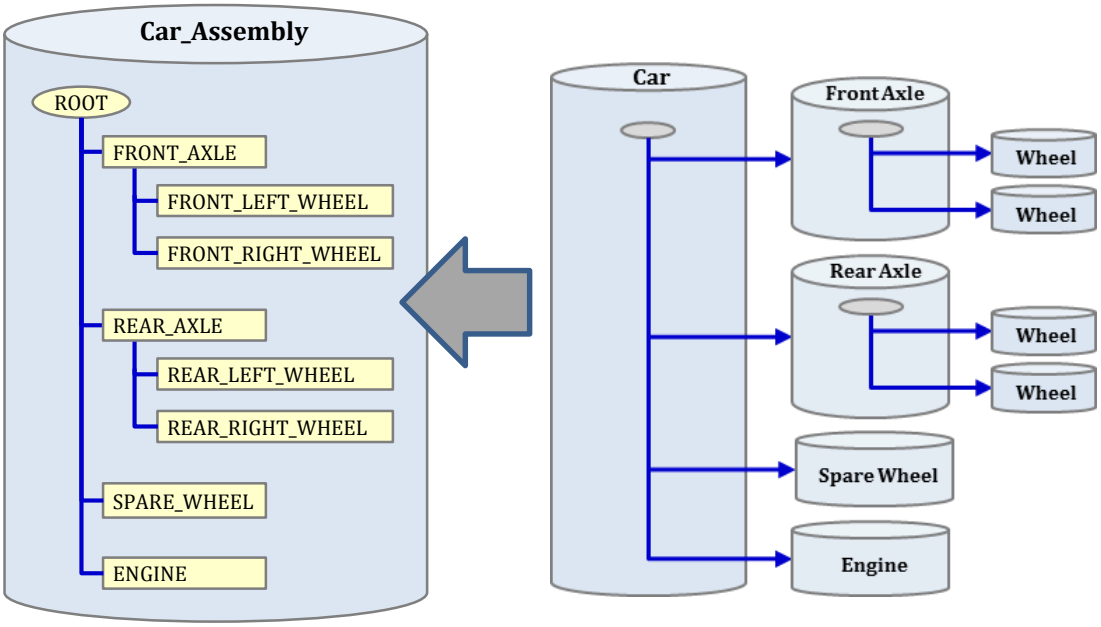
In engineering, a leaf node in an assembly tree is sometimes referred to as a piece part. This is a somewhat misleading term because it suggests that the part consists of a single solid body, which is not always true. To avoid any possible misunderstandings, we will use the term "leaf" in this document.

We can measure the **depth** of a node in a tree by counting its ancestors, including parents, grandparents, and so on, up to the root node of the tree. So, in our car example, the car itself is at depth zero, the axles and engine are at depth $= 1$, and the four main wheels are at depth $= 2$. In NX documentation, nodes with depth $= 1$ (i.e. immediately below the root node) are sometimes known as "top level" nodes.

Components and Prototypes

Suppose we have an assembly, and we want to write out a report describing its structure. Each part knows about its child subassemblies, so we could do this by writing code that "walks" from part to part, recording the parent-child relationships. We would start at the top of the tree with the car assembly file. Using the information in this file, we would find out that there are four children, and we could "walk" to each of these four children to get information about grandchildren, and so on. This process would certainly work, but it has a problem — we have to open each part file so that we can look inside to get information about its children. Opening hundreds of part files might be very slow (depending on their locations), and we may not even have permission to open some of them, so we need a better way.

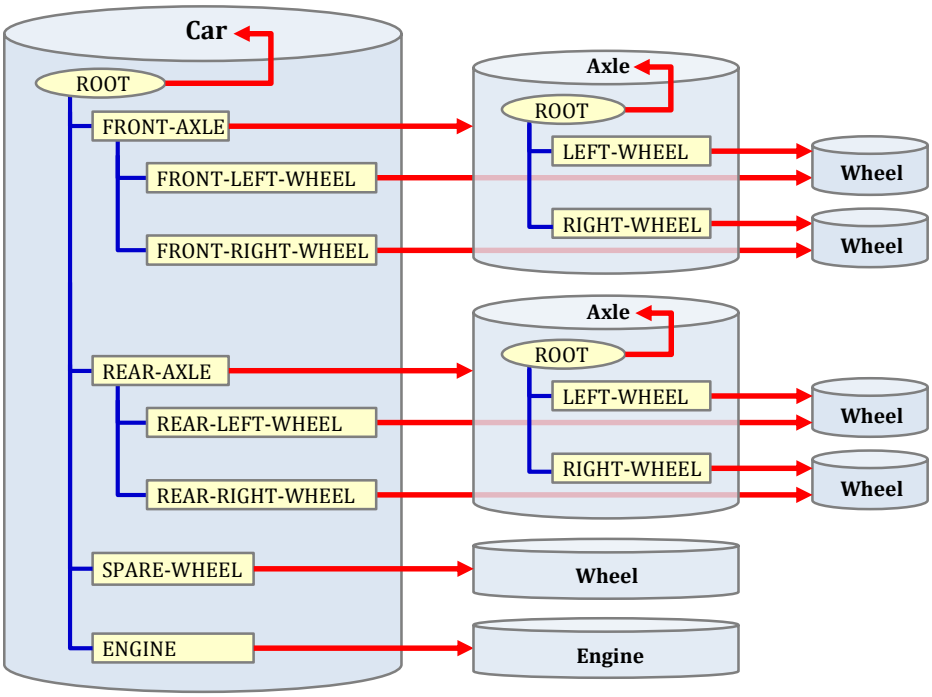
The NX solution is to store a replica of the assembly tree within each part file, as shown here:



The yellow items are called "**Components**" or sometimes "**Part Occurrences**". The tree of components inside a part file replicates the tree structure of the subassemblies themselves. So, if we want to know about this structure, we can simply traverse through the tree of component objects in the file, without opening any other part files.

A part file that represents an assembly has a special **RootComponent** object, which serves as the root node of its tree of components. You can get to all the other components in the part file by traversing downwards from the RootComponent. The RootComponent will be `Nothing` if the part file is not an assembly.

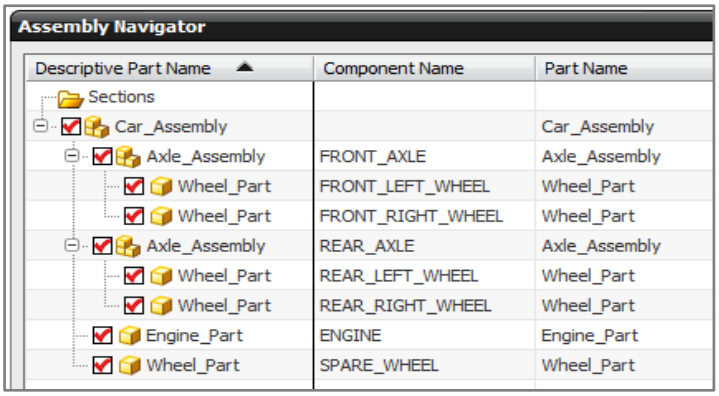
Each component contains a list of links to its children, a link to its parent, and a link to the corresponding part file, which is called the **Prototype** of the component. In the diagram below, the parent-child relationships are shown by the blue lines, and the component-to-prototype links are shown as red arrows:



So, for example, as you can see, the axle part is the prototype corresponding to each of the components FRONT_AXLE and REAR_AXLE. Or, looking at it the other way around, FRONT_AXLE and REAR_AXLE are occurrences of the axle part.

As mentioned before, a root component is not a "real" component, so its prototype link has a special meaning — it "loops back" and refers the part in which the root component resides.

This correspondence between components and their associated prototype parts is also displayed in the Assembly Navigator, as shown here:



In SNAP, components are represented by Snap.NX.Component objects, whose properties and methods are summarized in the table below:

Property or Method	Description
`Parent`	The parent component of this component
`Children`	An array containing the child components of this component
`Descendants`	A collection containing children, grandchildren, and all other descendants
`Leaves`	A collection containing all leaves descended from this component
`Prototype`	The prototype part of this component
`Depth`	The depth of this component in the tree hierarchy
`IsLeaf`	If True, indicates that this component is a leaf node of the component tree
`Position`	The position of this component in the parent part (discussed later)
`Orientation`	The orientation of this component in the parent part (discussed later)

Many additional properties and methods are inherited from NX.NXObject. For example, you can change the color of a component, hide it, move it between layers, assign attributes to it, and so on.

If you look at the documentation for Component.Descendants and Component.Leaves, you will see that they are both IEnumerable<Component> collections, which may be a new concept for you. You can look up the technical meaning of IEnumerable, but, in practical terms, the main point is that you can use these collections in `For Each` loops, as we will show below.

Cycling Through Descendants

There are many situations where it is useful to cycle through all the subassemblies of a given assembly, doing some operation on each of them. As mentioned above, we do this by traversing the nodes of the component tree. The simplest operation is to just get all the children of a given part (say the work part). The code to do this is as follows:

```
' Get the root component of the work part
Dim workPart As NX.Part = Snap.Globals.WorkPart
Dim root As NX.Component = workPart.RootComponent

' Get the array of child components of the root component
Dim children As NX.Component() = root.Children

' Cycle through the child components, writing out names of prototypes
For Each child As NX.Component In children
   Dim proto As NX.Part = child.Prototype       ' Get the prototype part
   Dim protoName As String = proto.Name         ' Get the part name
   InfoWindow.WriteLine(protoName)              ' Write it out
Next
```

If you run this code with Car_Assembly.prt as your work part, it will produce the following listing:

```
Engine_Part.prt
Wheel_Part.prt
Axle_Assembly.prt
Axle_Assembly.prt
```

Actually, if you are running NX in "managed" mode (with Teamcenter), rather than native mode, you will probably see names like Engine_Part/A, rather than Engine_Part.prt. Similar comments apply to the other examples below that use part names and pathnames. We'll mention this from time to time, to remind you.

Cycling through leaf nodes is similar. Suppose that root is the node whose leaf nodes we are interested in. We can write out the name of each leaf component plus the file name of the corresponding prototype part like this:

```
For Each leaf As NX.Component In root.Leaves
   Dim compName As String = leaf.Name                    ' Name of component
   Dim proto As NX.Part = leaf.Prototype                 ' Get the prototype part
   Dim protoName As String = proto.Name                  ' Get the part name
   InfoWindow.WriteLine(compName & " ; " & protoName)    ' Write it out
Next
```

If you run this code with Car_Assembly.prt as your work part, the result in native mode will be the following listing. The listing has been reformatted a little, to make it easier to read:

```
FRONT_LEFT_WHEEL ;  Wheel_Part.prt
FRONT_RIGHT_WHEEL ; Wheel_Part.prt
REAR_LEFT_WHEEL ;   Wheel_Part.prt
REAR_RIGHT_WHEEL ;  Wheel_Part.prt
SPARE_WHEEL ;       Wheel_Part.prt
ENGINE ;            Engine_Part.prt
```

As our last example, let's write out all the descendants of some root, along with each one's depth and parent:

```
For Each comp As NX.Component In root.Descendants
   Dim compName As String , parentName As String
   If comp.IsRoot Then
      compName = "[ROOT]"
      parentName = "[NO PARENT]"
   Else
      compName = comp.Name
      parentName = comp.Parent.Name
   End If

   If comp.Depth = 1 Then parentName = "[ROOT]"

   InfoWindow.Write("Depth: " & comp.Depth.ToString & " ;  ")
   InfoWindow.Write("Component: " & compName & " ;  ")
   InfoWindow.WriteLine("Parent: " & parentName)
Next
```

Note that we wrote special code to handle two cases:

- if the component is a root node, then it has no name and no parent
- if the component is at depth = 1, then its parent is a root node, which has no name

If you run this code with the car assembly as your work part, the results (with a little reformatting) will be:

```
Depth: 0 ;  Component: [ROOT] ;             Parent: [NO PARENT]
Depth: 1 ;  Component: FRONT_AXLE ;         Parent: [ROOT]
Depth: 2 ;  Component: FRONT_LEFT_WHEEL ;   Parent: FRONT_AXLE
Depth: 2 ;  Component: FRONT_RIGHT_WHEEL ;  Parent: FRONT_AXLE
Depth: 1 ;  Component: REAR_AXLE ;          Parent: [ROOT]
Depth: 2 ;  Component: REAR_LEFT_WHEEL ;    Parent: REAR_AXLE
Depth: 2 ;  Component: REAR_RIGHT_WHEEL ;   Parent: REAR_AXLE
Depth: 1 ;  Component: SPARE_WHEEL ;        Parent: [ROOT]
Depth: 1 ;  Component: ENGINE ;             Parent: [ROOT]
```

Indented Listings

Listings of parts in an assembly are easier to understand if they are indented, since the indentation makes the hierarchical structure more visible. First, a simple function that creates a string of spaces for use in indenting:

```
Public Shared Function Indent(level As Integer) As String
   Dim space As Char = " "c
   return new String(space, 3*level)     ' Indent 3 spaces for each level
End Function
```

Once we have this function, creating indented listings is straightforward. The Depth property is the key to getting the indenting correct, as the following code shows:

```
For Each comp As NX.Component In workPart.RootComponent.Descendants
   Dim indentString As String = Indent(comp.Depth)
   Dim compName As String = comp.Name
   If comp.IsRoot Then compName = "[ROOT]"
   InfoWindow.WriteLine(indentString & compName)
Next
```

This produces the following nicely indented listing:

```
[ROOT]
   FRONT_AXLE
      FRONT_LEFT_WHEEL
      FRONT_RIGHT_WHEEL
   REAR_AXLE
      REAR_LEFT_WHEEL
      REAR_RIGHT_WHEEL
   SPARE_WHEEL
   ENGINE
```

Recursive Traversals

As we saw above, it's very easy to traverse through the components of an assembly by using the Descendants and Leaves collections. From time to time, though, you may need to do this sort of traversal one level at a time, which you can do by using a technique called recursion. The basic idea is to write a recursive function, which is one that calls itself. This might sound like a strange idea, but it provides a very convenient way of traversing a tree, as in the following code:

```
Public Shared Sub Main()
   Dim workPart As NX.Part = Snap.Globals.WorkPart
   Dim root As NX.Component = workPart.RootComponent
   DoSomething(root)
End Sub

Public Shared Sub DoSomething(comp As NX.Component)
   InfoWindow.WriteLine(comp.Name)
   For Each child As NX.Component In comp.Children
      DoSomething(child)
   Next
End Sub
```

As you can see, the `DoSomething` function is recursive — it calls itself. So, what happens when the system executes the line of code that says `DoSomething(root)` in the Main function? Well, first of all, the name of the root component will be written out. Then, `DoSomething` is applied to each of the children of root, causing their component names to be written out. But, then, through the magic of recursion, applying `DoSomething` to a child causes `DoSomething` to be applied to its children, in turn, and so on. In the end, the result is that `DoSomething` gets applied to all the descendants of root, so all of their names are written to the Info window. Of course, in practice, you would probably replace the `InfoWindow.WriteLine` call with some more interesting code, but the principle would be exactly the same.

Tricks with LINQ

LINQ is a Microsoft technology that was introduced in 2007 to provide data query functions in .NET languages. Fortunately for us, the System.Linq namespace in the .NET framework contains a large collection of functions that are useful for processing lists of NX objects (like components).

To illustrate the techniques, suppose we have a large assembly in which each part has a "Vendor" attribute. Our car example has this attribute, but a more complex assembly would be better. Anyway, we can write a `GetVendor` utility function for use in our queries, as follows:

```
Public Shared Function GetVendor(part As NX.Part) As String
   Return part.GetStringAttribute("Vendor")
End Function
```

Then, if we wanted to get all the parts supplied by the vendor Acme, we might write:

```
Dim AcmeList As New List(Of NX.Part)

For Each comp As NX.Component In root.Descendants
   Dim proto As NX.Part = comp.Prototype
   If GetVendor(proto) = "Acme"
      AcmeList.Add(proto)
   End If
Next
```

This works fine, but LINQ gives us a new way to do things. We can write code like this, instead:

```
Dim AcmeList = From comp In root.Descendants
               Let proto = comp.Prototype          ' Get the prototype of each component
               Where GetVendor(proto) = "Acme"     ' Get those where vendor = Acme
               Select name = proto.Name            ' Get the prototype filenames
               Distinct                            ' Remove duplicates
               Order By name                       ' Sort aphabetically
```

This code again gets a list of parts supplied by Acme, but, in addition, it removes the duplicates from the list, and sorts it. As you can see, the general approach is to "chain" the operations together — the output from one operation becomes the input to the next. All the LINQ operators are designed to use IEnumerable collections as their inputs and outputs, so the chaining works smoothly.

As a last example, here's some magic that creates a list of unique parts in the car assembly, along with their counts, and then outputs this list:

```
Dim partList = From comp In root.Leaves
               Group By partTag = comp.Prototype.NXOpenTag
               Into partGroups = Group, Count()

For Each item In partList
   InfoWindow.Write("Count = " & item.Count & " ;  ")
   Dim part As Snap.NX.Part = Snap.NX.Part.Wrap(item.partTag)
   InfoWindow.WriteLine("Part = " & part.Name)
Next
```

The Linq approach often produces code that is shorter and easier to understand, so it's worth knowing about it, even though it is a somewhat advanced topic. But, code like the last example can be pretty puzzling, so it's probably best to avoid this kind of trickery until you become an expert. Making your code clear is much more important than making it concise and slick.

Component Positions & Orientations

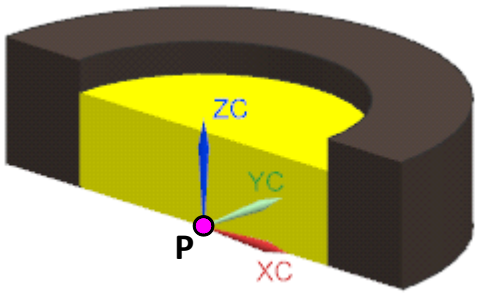
When you insert a part into an assembly, it is typically re-positioned and re-oriented somehow. The position and orientation information are held within an NX component object, and you can retrieve it as follows:

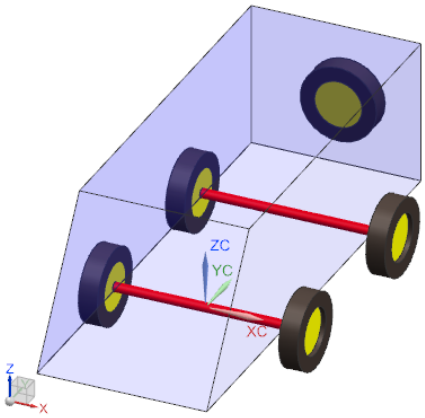
```
For Each comp As NX.Component In workPart.RootComponent.Descendants
   Dim location As Snap.Position = comp.Position
   Dim orientation As Snap.Orientation = comp.Orientation
   Dim axisZ As Snap.Vector = orientation.AxisZ
   InfoWindow.WriteLine("comp = " & comp.Name)
   InfoWindow.Write("   comp.Position = " & location.ToString("F0") & " ;  ")
   InfoWindow.Write("   comp.Orientation.AxisZ = " & axisZ.ToString("F0") & vbCrLf)
Next
```

If you run this code with the car assembly as your work part, the resulting listing will include the following:

```
FRONT_LEFT_WHEEL  ; comp.Position = (  950, 0, 0 )   ;  comp.Orientation.AxisZ = (  1, 0, 0 )
FRONT_RIGHT_WHEEL ; comp.Position = ( -950, 0, 0 )   ;  comp.Orientation.AxisZ = ( -1, 0, 0 )
REAR_LEFT_WHEEL   ; comp.Position = ( 950, 2000, 0 ) ;  comp.Orientation.AxisZ = (  1, 0, 0 )
SPARE_WHEEL       ; comp.Position = ( 0, 3050, 650 ) ;  comp.Orientation.AxisZ = (  0, 1, 0 )
```

To understand what this means, let's first look at how the wheel part itself was designed. The left-hand picture below shows a section view in the wheel part. As you can see, the inside center of the rim (the purple point labeled "P") is at the origin, and the rotational axis of the wheel is along the z-axis.





When the front left wheel gets inserted into the car assembly, this point P gets placed at (950, 0, 0). So, if `comp` is the FRONT_LEFT_WHEEL component, then `comp.Position` is (950, 0, 0). Similarly, the REAR_LEFT_WHEEL component has `Position` = (950, 2000, 0).

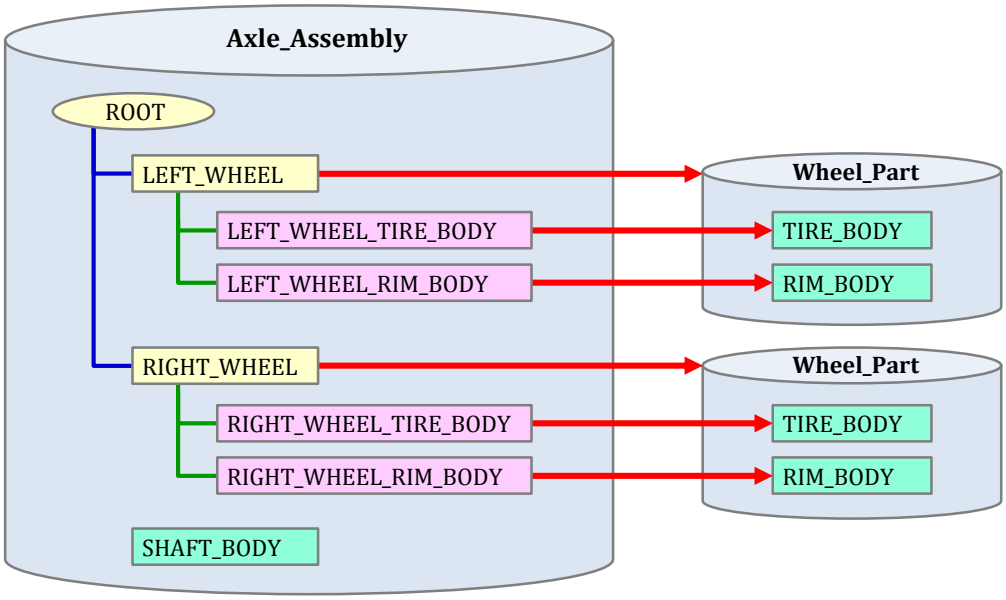
Orientations are a bit more interesting: when the front left wheel gets inserted into the car assembly, its z-axis gets aligned with the x-axis of the car. So, if `comp` is the FRONT_LEFT_WHEEL component, then `comp.Orientation.AxisZ` is (1, 0, 0). On the right-hand side of the car, the wheel is flipped, of course, so, the FRONT_RIGHT_WHEEL has its `AxisZ` in the opposite direction, equal to (–1, 0, 0). Similarly, the SPARE_WHEEL component has `Orientation.AxisZ` = (0, 1, 0).

We could also study the Orientation.AxisX and Orientation.AxisY properties of various components, of course. But, in the case of an axi-symmetric object like a wheel, these are not important.

These sorts of transformations are often explained using matrices. But this usually causes confusion, so it's better to think in terms of repositioning of vectors, as we have done above.

Object Occurrences

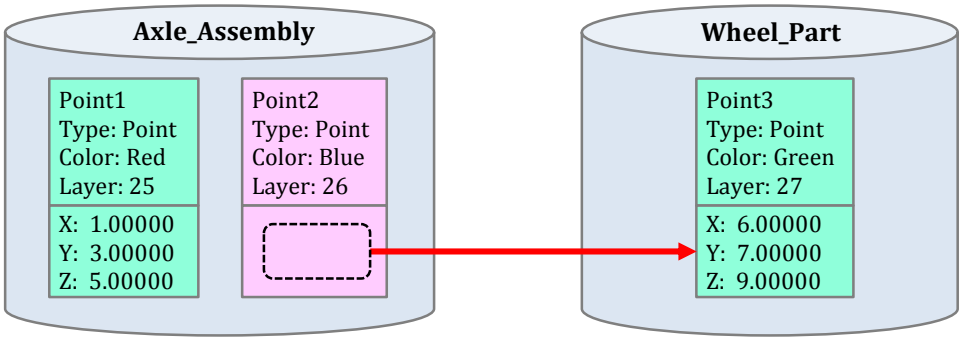
When a part is inserted into an assembly, we know that an occurrence of this part (i.e. a component object) gets created in the parent assembly. But, the story doesn't end there. In additional to the occurrence of the inserted part itself, the system also creates occurrences of all the objects inside it. To understand what happens, let's look at the structure of the Axle part in our car example. As we know, this part contains a solid body representing a shaft, plus two components (LEFT_WHEEL and RIGHT_WHEEL) which are occurrences of Wheel_Part. The wheel part contains two solid bodies called TIRE_BODY and RIM_BODY. The structure is shown in the diagram below:



Looking at the top half of the diagram, we see that the wheel part has been inserted into the axle assembly. As a result of this, a part occurrence called LEFT_WHEEL has been created in the Axle_Assembly part. But, in addition to this, we see the pink boxes, LEFT_WHEEL_TIRE_BODY and LEFT_WHEEL_RIM_BODY. These are **object occurrences**; LEFT_WHEEL_TIRE_BODY is an occurrence of TIRE_BODY, and LEFT_WHEEL_RIM_BODY is an occurrence of RIM_BODY. We say that these object occurrences are **members** of the LEFT_WHEEL component, as indicated by the green lines. The red arrows show how part and object occurrences both refer back to the original objects, which are called their **prototypes**. Only solid bodies are shown in the diagram, but, in fact, the LEFT_WHEEL component will have members that are occurrences of ***all*** the objects in the wheel part.

In many ways, the LEFT_WHEEL_TIRE_BODY occurrence looks and behaves just like a normal solid body in the axle part. You can blank it, move it to another layer, assign attributes to it, or even calculate its weight and center of gravity. But, on the other hand it is fundamentally different from SHAFT_BODY, which is a "real" solid body. The difference is that SHAFT_BODY includes its own geometric data, whereas LEFT_WHEEL_TIRE_BODY merely has links to geometric data that actually reside in the wheel part. So, in some sense, an occurrence is a "phantom" or "proxy" object, rather than a "real" one. Or, borrowing some terminology from Microsoft Office products, we might say that an occurrence is a "linked" object, whereas a "real" object like SHAFT_BODY is an "embedded" one. The technology used in NX is completely different, but the basic concept is similar.

The diagram below shows the difference between the data structures of occurrence and "real" objects, using a simple example of three point objects in the axle and wheel parts:



Point1 is embedded in the axle part, and Point2 is an occurrence whose prototype (Point3) resides in the wheel part. As usual, green boxes denote "real" embedded objects and pink ones denote occurrences. As you can see,

Point2 has a color and a layer, but it has no coordinate data of its own. Whenever we ask for the coordinates of Point2, they will be derived by suitably transforming the coordinates of Point3.

The diagram above illustrates another important fact: even though Point2 is an occurrence, its object type is still "Point". There is no special "occurrence" type in NX; any NX object can either be an occurrence (a linked object), or a "real" local embedded one. An NX.NXObject has a property IsOccurrence, which allows you to find out whether or not it's an occurrence. Then, if IsOccurrence is True, there are ProtoType and OwningComponent properties with the obvious meanings. The following code shows how these properties can be used:

```
For Each obj As NX.NXObject In workPart.Objects
   If obj.IsOccurrence And obj.ObjectType = NX.ObjectTypes.Type.Body
      Dim occName As String = obj.Name
      Dim protoName As String = obj.Prototype.Name
      InfoWindow.Write("Occurrence: " & occName & "  ; ")
      InfoWindow.Write("Owning component: " & obj.OwningComponent.Name & "  ; ")
      InfoWindow.WriteLine("Prototype: " & protoName)
   End If
Next
```

Note that you have to cycle through workPart.Objects in order to find object occurrences; if you cycle through workPart.Bodies (for example), you won't find bodies that are occurrences, you will only find the ones that are embedded in the work part.

If you run this code with Axle_Assembly.prt as your work part, the output will be as follows:

```
Occurrence: RIGHT_TIRE_BODY  ; Owning component: RIGHT-WHEEL  ; Prototype: TIRE_BODY
Occurrence: RIGHT_RIM_BODY   ; Owning component: RIGHT-WHEEL  ; Prototype: RIM_BODY
Occurrence: LEFT_TIRE_BODY   ; Owning component: LEFT-WHEEL   ; Prototype: TIRE_BODY
Occurrence: LEFT_RIM_BODY    ; Owning component: LEFT-WHEEL   ; Prototype: RIM_BODY
```

Other Topics

NX/Open has a very rich and complex collection of functions for working with assemblies. After reading the material in this chapter, you should be ready to start using the NXOpen functionality, too. There are older functions in the NXOpen.UF.UFAssem class, along with several example programs, and some useful explanatory notes. Then, in addition, there are some newer functions in the NXOpen.Assemblies namespace. In NXOpen, note that you have to get the root component via a ComponentAssembly object, rather than directly from a part object.

Chapter 12: Simple Input and Output

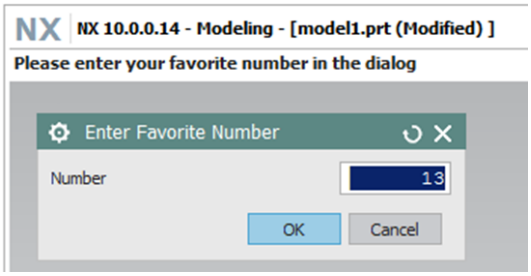
To communicate with the user of your programs, you can either use WinForms, as we saw in chapter 3, or NX block-based dialogs, which we'll describe in chapters 14 and 15. Both techniques allow you to build very rich user interfaces, but sometimes this is a lot more than you need. Sometimes, you just want the user to enter a number, or you just want to write out a bit of text. This chapter describes some easy ways to do this simple input and output.

Entering Numbers and Strings

The Snap.UI.Input class has several functions that allow the user to enter information. The simplest ones display a dialog that prompts the user to enter an integer, a floating point number (a double), or a text string. The following code provides an example:

```
Dim cue, title, label As String
cue   = "Please enter your favorite number in the dialog"
title = "Enter Favorite Number"
label = "Number"
Dim initialValue As Integer = 13
Dim favorite As Integer = Snap.UI.Input.GetInteger(cue, title, label, initialValue)
```

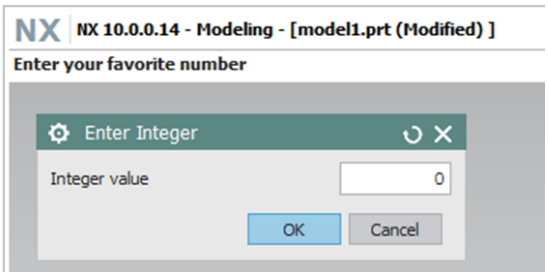
This code produces the following display in NX:



As you can see from the documentation, almost all of the inputs to the Snap.UI.Input.GetInteger function are optional. If you omit most of the optional inputs, and put `Imports Snap.UI.Input` at the top of your source file, the code above can be abbreviated to just:

```
Dim favorite As Integer = GetInteger("Enter your favorite number")
```

This produces the following display:



The GetDouble function provides very similar capabilities, for entering double (floating point) numbers, and the GetString function allows entry of a string of text. Also, there are similar functions called GetIntegers, GetDoubles, and GetStrings, which let you enter several items of data, rather than just one.

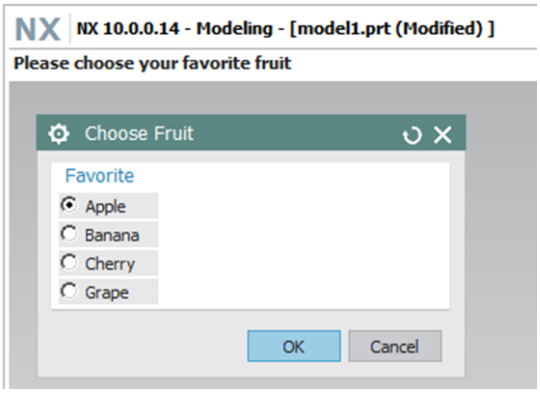
All of the dialogs described above are examples of the usual kind of NX "block-based" dialog. They are very simple examples, because each dialog has only one block, but they still have all the expected properties and behavior — you can collapse them, reset them to their default values, and so on.

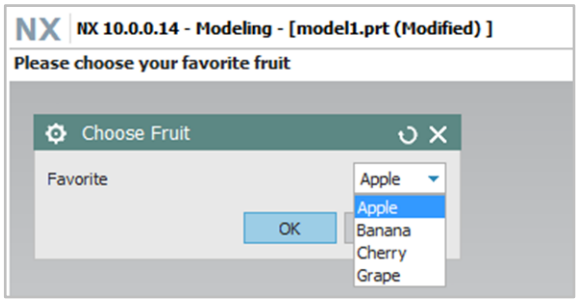
Choosing from Menus

Another simple input function allows the user to choose from a list of alternatives. The function is called GetChoice, and the following code shows how to use it (with `Option Infer = On`, for brevity):

```
Dim cue = "Please choose your favorite fruit"
Dim title = "Choose Fruit"
Dim label = "Favorite"
Dim fruits As String() = {"Apple", "Banana", "Cherry", "Grape"}
Dim style = Snap.UI.Block.EnumPresentationStyle.RadioBox
Dim choice = GetChoice(fruits, cue, title, label, style)
```

The "style" argument allows you to specify whether you want the choices displayed as a list of radio buttons or as a pull-down menu, as shown below:





Except for the list of items, all the other inputs are optional, and have somewhat reasonable default values, so you can display a simple menu with just one line of code, like this:

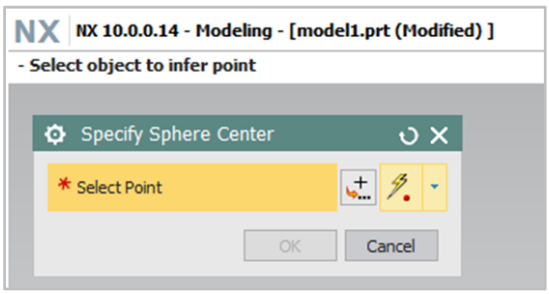
```
Dim choice As Integer = GetChoice( { "Apple", "Banana", "Cherry", "Grape" } )
```

Specifying Positions, Vectors, and Planes

The function GetPosition displays a dialog that allows the user to specify a position using the standard NX "Point Subfunction". For example, the following code:

```
Dim title = "Specify Sphere Center"
Dim label = "Center point"
Dim center As Position = GetPosition(title, label).Position
Sphere(center, 10)
```

displays this dialog



The dialog contains a single "SpecifyPoint" block, which is exactly the same as the ones that appear within NX or in block-based dialogs that you construct yourself. Similarly, the GetVector function displays a dialog containing a SpecifyVector block, GetPlane lets the user specify a plane, and so on.

Writing Output

We will often want to output text from our programs, to record results or provide other information. The easiest way to do this is to write the text to the NX Information window (also known as the "listing" window in the past). The Snap.InfoWindow class provides many functions for doing this. The important functions are Write and WriteLine. The design is modeled after the System.Console class, so the WriteLine functions append a "return" to their output, while the Write ones do not. Some simple examples of the Write functions are:

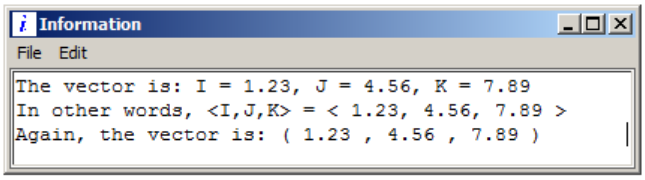
Function	Inputs and Creation Method
`Write(output As Integer)`	Write an integer to the Information window.
`Write(output As Double)`	Write a double to the Information window.
`Write(output As String)`	Write a text string to the Information window.
`Write(output As Position)`	Write a Position to the Information window.
`Write(output As Vector)`	Write a Vector to the Information window.

The function for writing strings is very flexible because there are a great many .NET functions to help you compose your string. Also the SNAP Position and Vector objects have ToString functions that assist you further.

So, for example, you can write things like :

```
Dim v As Vector = {1.23, 4.56, 7.89}
Dim s1, s2, s3 As String

s1 = String.Format("The vector is: I = {0}, J = {1}, K = {2}", v.X, v.Y, v.Z)
s2 = String.Format("In other words, <I,J,K> = < {0}, {1}, {2} >", v.X, v.Y, v.Z)
s3 = "Again, the vector is: " & v.ToString()

InfoWindow.WriteLine(s1)
InfoWindow.WriteLine(s2)
InfoWindow.WriteLine(s3)
```

That code produces the following output in the Info window:



You can find out more about the Visual Basic string manipulation techniques in many books and on-line tutorials.

Windows Output

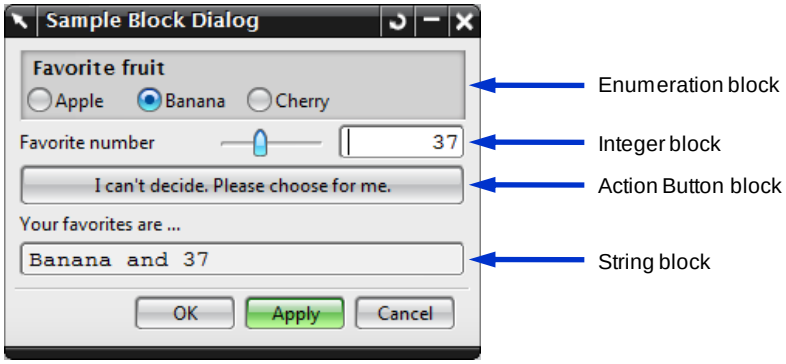
Windows forms give you a vast range of output possibilities, of course, far beyond what you can do with the NX Info window. The simplest approach is to use the System.Windows.Forms.MessageBox class. For example, this code:

```
Dim title As String = "Thanks Very Much"
Dim text As String = "ありがとう !!"
MessageBox.Show(text, title, MessageBoxButtons.OK, MessageBoxIcon.Information)
```

produces this display

Chapter 13: Block-Based Dialogs

Since around 2007, the NX user interface has been based on “block-based” dialogs, so-called because they are built from a common collection of user interface “blocks”. So, for example, this dialog consists of four blocks, whose types are indicated by the labels to the right



Each block has a specific type and purpose. So, looking at the four examples from the dialog above:

- An Enumeration block presents a set of options to the user, and asks him to choose one of them
- An Integer block allows the user to enter an integer (by typing, or by using a slider, for example)
- An Action Button block performs some action when the user clicks on it
- A String block displays text that the user can (sometimes) edit

Blocks of any given type are used in many different dialogs throughout NX. Application developers build dialogs from blocks, rather than from lower-level items. This reduces programming effort for NX developers, and guarantees consistency. Constructing a new Enumeration block (for example) requires very little code, and this new Enumeration block is guaranteed to look and behave in exactly the same way as all other Enumeration blocks within NX.

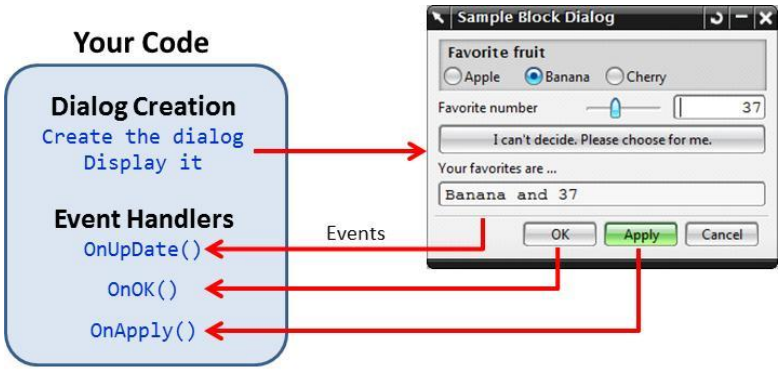
You can construct these same “block-based” dialogs in SNAP, so your add-on applications can look and behave like the rest of NX. This chapter and the next one tell you how to do this. In fact, there are two ways to create block-based dialogs in SNAP. One approach (using BlockForm objects) is described in this chapter, and the other one (using BlockDialog objects and Block UI Styler) is described in chapter 14.

When to Use Block-Based Dialogs

As we saw earlier, you can use Windows Forms (WinForms) to create dialogs for your SNAP applications, and Visual Studio has some very nice tools to help you do this. So, you may be wondering why you should use block-based dialogs instead. WinForm dialogs are very rich and flexible, so there may be times when they are appropriate. On the other hand, block-based dialogs are rigid and highly structured, because they enforce NX user interface standards. Unless the added flexibility of a WinForm brings some significant benefit, it’s better to have a block-based dialog whose appearance and behavior are consistent with the rest of NX. Also, achieving NX-like behavior in a WinForm-based dialog sometimes requires a great deal of work. This is especially true of dialogs that have accompanying graphical feedback (like Selection and the Point, Vector and Plane Subfunctions). For these kinds of situations, implementation using blocks is usually much easier. So, in short, we recommend using block-based dialogs unless the added flexibility of WinForms provides some large benefit that outweighs the drawbacks of inconsistency and increased development cost.

How Block-Based Dialogs Work

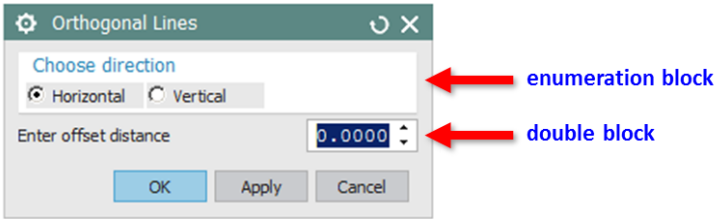
The diagram below shows how your code interacts with a block-based dialog



First, your code creates and displays the dialog. Then, when the user starts to interact with the dialog, NX sends messages back to your code, telling you what "events" occurred in the dialog. For example, NX might tell you that the user entered some number, or clicked on the Apply button. Your code should have functions called "event handlers" or "callbacks" that determine what should happen (if anything) in response to each event. The event handlers must have special names, so that NX knows how to call them; for example, the event handler called when the user clicks the Apply button must be named either "OnApply" or "ApplyCallback", depending on the circumstances. If you want to create some geometry when the user clicks the Apply button, you would put the code to create this geometry in your OnApply function.

Our Example — OrthoLines

In this chapter, you'll learn how to create dialogs using Snap.UI.BlockForm objects. To illustrate the process, we'll use this simple dialog that lets the user create "infinite" lines in the horizontal or vertical directions in the XY-plane.



It only has two blocks — an "Enumeration" block to let the user choose either horizontal or vertical, and a "Double" block in which the user enters the offset distance (the distance from the line to the origin). So, if the user chooses "horizontal" and enters an offset distance of 300, for example, he'll get the line Y = 300.

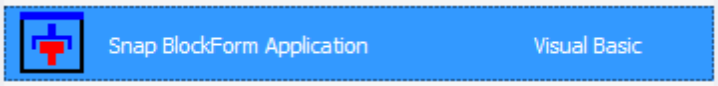
Our lines won't be truly infinite, they'll just be very long. The idea is that some tasks are best accomplished by making oversized lines and then trimming them later.

The process described below is quite lengthy. If you get lost (or bored) at some point, you can find the completed example in [...NX]\UGOPEN\SNAP\Examples\GS Guide\OrthoLines1.

Using the Snap BlockForm Template

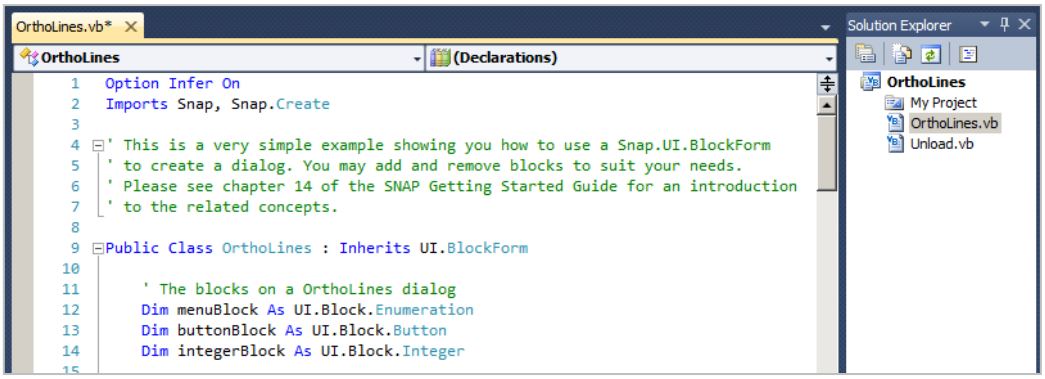
In this example, we're going to assume that you're using Visual Studio. If you've forgotten how to build and run projects using Visual Studio, please take a quick look at example 1 in chapter 3 to remind yourself.

In Visual Studio, choose "New Project". Choose the Snap BlockForm Application project template

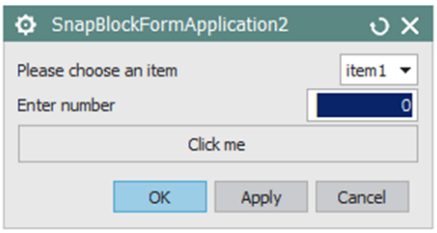


and type in a name down at the bottom of the dialog. As always, it's wise to give your project a meaningful name. We recommend the name "OrthoLines", because this will make it easier to follow the descriptions below. After you click OK, Visual Studio will create a new project containing two files.

One is the usual "Unload" file, which contains nothing new or interesting. The other is a file called OrthoLines.vb, which contains the skeleton of a BlockForm-based application, as shown here:

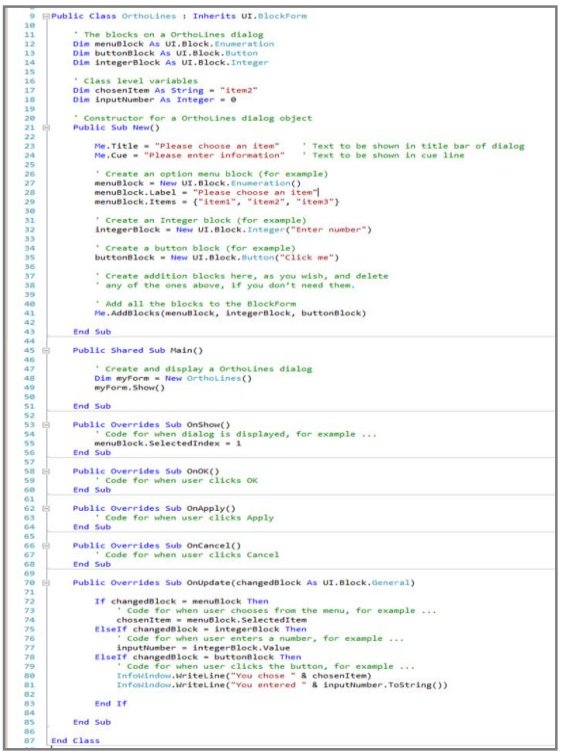


You can immediately build this project (press Ctrl+Shift+B in Visual Studio), and run it from within NX (using Ctrl+U). Please refer to example 1 in chapter 3 if you've forgotten how to do this. This dialog will be displayed:



This isn't the dialog we want, of course, but at least it's nice to see something displayed successfully. If you enter some information, and click the button, you'll see that the dialog responds as you would expect. Click the Cancel button on the dialog when you're finished playing with it.

If you browse through the code in OrthoLines.vb, you'll see that it has a few major sections, as outlined below:



Declarations of blocks that will belong to the dialog

The **Constructor** (the **Sub New** function, in which you create new blocks and add them to the dialog

The **Main** function, which creates and displays a dialog

Event handlers (OnApply, OnUpdate, etc.), which respond when the user interacts with the dialog

You don't need to understand the details, for now, and you certainly don't have to try to read the tiny text in the picture above. Just scan through the code and familiarize yourself with the basic high-level structure.

We don't care about the Main function, but make sure you know how to find the **declarations**, **constructor**, and **event handlers**, because we'll be referring to these sections later on.

Next we'll see how to change this code to get the dialog we want.

The Dialog Title and Cue

In the file OrthoLines.vb, find the constructor (the `Sub New` subroutine). It's about 20 lines from the top of the file. Near the start of the constructor, you will see two lines of code that define `Me.Title` and `Me.Cue`. Change them to read as follows:

```
Me.Title = "Orthogonal Lines"     ' Text to be shown in title bar of dialog
Me.Cue = "Whatever"               ' Text to be shown in the cue line
```

As the comments say, these two lines define the text strings that are shown in the title bar of the dialog and in the cue line. It doesn't matter what cue text you use, since no-one ever reads the cue line, anyway ☺.

Declaring and Creating Blocks

Currently, the dialog has three blocks. We're going to delete these three, and create two new ones of our own. We have to make changes in two places — in the declarations of the blocks, and in the block creation code.

First, let's change the declarations. About 10 lines from the top of the file, you will see three lines of code that declare blocks called `menuBlock`, `buttonBlock`, and `integerBlock`. Delete these three lines and replace them with:

```
' Declarations of the blocks on a OrthoLines dialog
   Dim directionBlock As UI.Block.Enumeration
   Dim offsetBlock As UI.Block.Double
```

So, now we have declared two blocks called `directionBlock` and `offsetBlock`, which we'll be using on our dialog.

Next, we need to change the code that creates blocks. In the middle of the constructor function you will find the following code, which creates the three original blocks:

```
' Create an Enumeration block (for example)
menuBlock = New UI.Block.Enumeration()
menuBlock.Label = "Please choose an item"
menuBlock.Items = {"item1", "item2", "item3"}

' Create an Integer block (for example)
integerBlock = New UI.Block.Integer("Enter number")

' Create a Button block (for example)
buttonBlock = New UI.Block.Button("Click me")
```

Since we're going to replace these three blocks with two of our own, you can delete this code. Replace it with the following code, instead:

```
' Create the direction choice block
directionBlock = New UI.Block.Enumeration()
directionBlock.Label = "Choose direction"
directionBlock.Items = {"Horizontal", "Vertical"}

' Create the offset distance block
offsetBlock = New UI.Block.Double("Offset distance")
```

Now that we've created `directionBlock` and `offsetBlock`, we can add them to our dialog. To do this, we just modify the last line of the constructor function to read as follows:

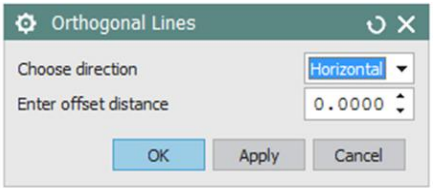
```
' Add the blocks to the BlockForm
Me.AddBlocks(directionBlock, offsetBlock)
```

After all the changes outlined above, the code in your constructor should now look like this:

```
' Constructor for a OrthoLines dialog object
Public Sub New()

    Me.Title = "Orthogonal Lines"    ' Text to be shown in title bar of dialog
    Me.Cue   = "Whatever"            ' Text to be shown in cue line

    ' Create the direction choice block
    directionBlock = New UI.Block.Enumeration()
    directionBlock.Label = "Choose direction"
    directionBlock.Items = {"Horizontal", "Vertical"}

    ' Create the offset distance block
    offsetBlock = New UI.Block.Double("Offset distance")

    ' Add all the blocks to the BlockForm
    Me.AddBlocks(directionBlock, offsetBlock)

End Sub
```

At this point, it would be nice to build and run the project, to see what we have accomplished. But, you can't do this because the code still has several references to the `menuBlock` object, which no longer exists (because we deleted it). You can see that the word `menuBlock` has squiggly underlining everywhere, indicating that there's something amiss. The easiest way to fix this is to just delete all the code that mentions `menuBlock`. You can delete the `OnShow`, `OnOK`, and `OnCancel` functions entirely, because we're not going to use them. Also, you can delete the code inside the `OnUpdate` function. We are going to use the `OnUpdate` function, later, so don't delete the whole thing – just delete the lines inside the function, but leave the first and last lines intact.

Once you've done all this, you can build and run your project, and it should produce a dialog that looks like this:



This isn't quite what we want, but it's getting close. The first problem is that we have no up-down "spin" arrows on the field where we enter the offset distance. This is easy to fix — just change the definition of the `offsetBlock` a little. Specifically, you should add a line of code that sets the `PresentationStyle` property, as follows:

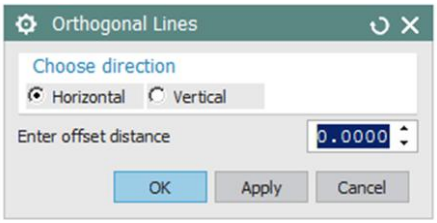
```
' Create the offset distance block
offsetBlock = New UI.Block.Double("Enter offset distance")
offsetBlock.PresentationStyle = UI.Block.NumberPresentationStyle.Spin
```

The next problem is that the horizontal/vertical choice is displayed as an option menu, not as a pair of radio buttons. Again, we can fix this by adjusting the properties of our `directionBlock` object. After the lines that set the `directionBlock.Label` and `directionBlock.Items` properties, put two more lines:

```
directionBlock.PresentationStyle = UI.Block.EnumPresentationStyle.RadioBox
directionBlock.Layout = UI.Block.Layout.Horizontal
```

It looks like a lot to type, but Intellisense will do most of the work for you. The first line says we want radio buttons, and the second line says we want them arranged horizontally.

Once this is done, you can build and run, and you should get the desired dialog:



The dialog should behave correctly — you can click on the radio buttons, type numbers, and use the OK, Apply, and Cancel buttons. Go ahead and try it. At this point, though, nothing very useful will happen as a result of your typing and clicking. We'll fix that in our next step.

The OnApply Event Handler

When the user interacts with our dialog, NX will take note of what he does, and send messages back to our code. Specifically, every time the user performs some action in the dialog, NX will call the associated "event handler" function within our code. For example, if the user clicks the "Apply" button, NX will call our `OnApply` function (since this is the event handler for an "Apply" event). Whatever code we put inside our OnApply function will then get executed, so we can respond to the "Apply" event in a useful way.

So, let's begin by making the Apply button do something interesting. Change the `OnApply` function to read:

```
Public Overrides Sub OnApply()
   InfoWindow.WriteLine("You clicked the Apply button")
End Sub
```

Build and run the project. When the dialog appears, click on the Apply button, and this should cause a message to be displayed in the NX Info window. This is not terribly exciting, admittedly, but it shows that the basic mechanism is working — when the user clicks the Apply button, the code in our OnApply event handler is getting executed.

You should try clicking the OK button, too. You will see that this also causes the same message to appear in the Info window. This is because the default implementation of the `OnOK` event handler just calls the `OnApply` function and then closes the dialog. So, our `OnApply` code is getting executed when the user clicks OK, also.

Of course, what we'd really like to do is create a line when the user clicks the Apply button. Here's a new version of the OnApply function that will do exactly that. Type it in, or copy/paste it, as usual.

```
Public Overrides Sub OnApply()
    Dim infinity As Double = 5000
    Dim d As Double = offsetBlock.Value
    If directionBlock.SelectedItem = "Horizontal" Then
        Line(-infinity, d, infinity, d)     ' Create a horizontal line
    Else
        Line(d, -infinity, d, infinity)     ' Create a vertical line
    End If
End Sub
```

This code shows the typical pattern of an event handler — you retrieve information from the dialog blocks, and then use this information to do what the user requested. As you can see, we use the `SelectedItem` property of `directionBlock` to decide whether to create a horizontal or vertical line, and we read the offset distance from the `offsetBlock.Value` property. We're assuming that the user has set these values appropriately before clicking the Apply button. The value we're using for `infinity` is arbitrary, of course, and you will probably want to change it to something larger if you design aircraft or ships.

If you build and run this code, you should find that it works nicely. Entering some information and clicking Apply will create a line, as we expect. Clicking OK will also create a line, for the reasons outlined above. Happily, this is exactly what we want.

To make our code a bit cleaner, and to prepare for the steps ahead, let's re-organize a little. For reasons that will become clear later, we're going to package the code that creates an infinite line into a nice tidy function. Copy the following code, and place it somewhere inside the OrthoLines class. Right at the bottom, just before the `End Class` line is a good place for it.

```
Private Function CreateLine() As NX.Line
   Dim infinity As Double = 5000
   Dim d As Double = offsetBlock.Value
   If directionBlock.SelectedItem = "Horizontal" Then
      Return Line(-infinity, d, infinity, d)      ' Horizontal line
   Else
      Return Line(d, -infinity, d, infinity)      ' Vertical line
   End If
End Function
```

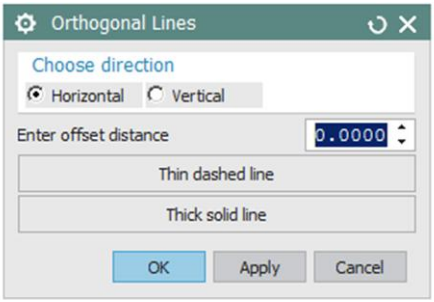
Note that we have made the function Private, since it wouldn't make sense to use it outside the OrthoLines class. Now that we have this CreateLine function, we can make a much simpler version of our OnApply function, like this:

```
Public Overrides Sub OnApply()
    CreateLine()
End Sub
```

The basic version of your OrthoLines function is now complete. Congratulations. In the next section we'll add a little more functionality to it, and learn how to use the OnUpdate function.

The OnUpdate Event Handler

Suppose we want to create two different kinds of infinite lines – thin dashed ones and thick solid ones. A convenient way to do this would be to place two new buttons on our dialog, like this:



Let's suppose that we're going to call these new buttons `thinDashedButton` and `thickSolidButton`. As before, the first thing we'll do is declare these two buttons. Add two more lines to the declaration section near the top of the file, which will give you a total of four declarations, like this:

```
' Declarations of the blocks on a OrthoLines dialog
Dim directionBlock As UI.Block.Enumeration
Dim offsetBlock As UI.Block.Double
Dim thinDashedButton As UI.Block.Button
Dim thickSolidButton As UI.Block.Button
```

Next, we need to add code to create the two new buttons. Near the end of the constructor, just before the `Me.AddBlocks` line, insert the following two lines:

```
' Create the two buttons
thinDashedButton = New UI.Block.Button("Thin dashed line")
thickSolidButton = New UI.Block.Button("Thick solid line")
```

And, finally, we add the two buttons to the dialog, along with the other two blocks:

```
' Add all the blocks to the BlockForm
Me.AddBlocks(directionBlock, offsetBlock, thinDashedButton, thickSolidButton)
```

You can build the project and run this code, and it should produce the dialog shown above. But, of course, the new buttons won't do anything until we write some event handler code for them.

The event handler code for the two new buttons should go in the `OnUpdate` function, like this:

```
Public Overrides Sub OnUpdate(changedBlock As UI.Block.General)
   Dim myLine As NX.Line

   If changedBlock = thinDashedButton
      myLine = CreateLine()
      myLine.LineWidth = Globals.Width.Thin
      myLine.LineFont = Globals.Font.Dashed
   End If

   If changedBlock = thickSolidButton
      myLine = CreateLine()
      myLine.LineWidth = Globals.Width.Thick
      myLine.LineFont = Globals.Font.Solid
   End If

End Sub
```

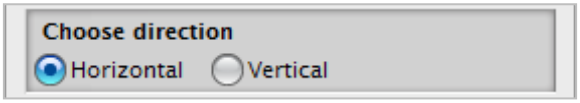
You can see now why we wrote the `CreateLine` function — because we need to call it in two places in this code.

NX calls our `OnUpdate` function whenever the user does anything with ***any*** block on the dialog. As you can see, the `OnUpdate` function receives a UI block called `changeBlock` as input, which tells us which block the user "touched". We write a series of "`If`" clauses that test the value of `changedBlock`, and do different things in different cases. If we find that `changedButton = thinDashedButton`, for example, then we know that the user clicked the `thinDashedButton` button, so we create a line that's thin and dashed.

Of course, it's possible that the user changed the line direction or the offset distance (rather than clicking one of our two buttons). We could put some more code in the `OnUpdate` function to handle these events, too, if we wanted. But let's quit here. Build and run the project, and have some fun making infinite lines.

Making Custom Re-Usable UI Blocks

You may find that you repeatedly use the same blocks, with the same properties, on many different BlockForm dialogs. For example, in the previous example, we created an Enumeration block that had radio buttons for choosing between horizontal and vertical.



If you use this sort of block over and over again, in different places, then it makes sense to capture its definition in a re-usable way. To do this, we might define a new class called HorizVertChoice as follows:

```
Public Class HorizVertChoice   :   Inherits Snap.UI.Block.Enumeration
   Public Sub New()
      Me.Label = "Choose direction"
      Me.Items = {"Horizontal", "Vertical"}
      Me.PresentationStyle = UI.Block.EnumPresentationStyle.RadioBox
      Me.Layout = UI.Block.Layout.Horizontal
   End Sub
End Class
```

Using this class, the definition of `directionBlock` in the OrthoLines constructor becomes very simple; it's just:

```
directionBlock = New HorizVertChoice()
```

Another example is a re-usable "UnitSlider" block that lets the user enter some value between 0 and 1:

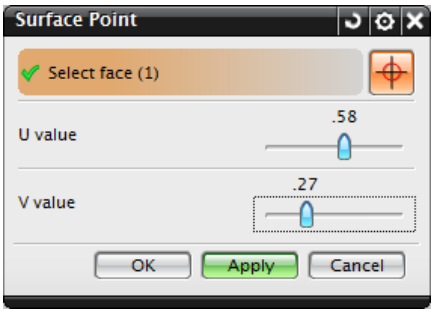
You can define this new class of block with the following code

```
Public Class UnitSlider   :   Inherits Snap.UI.Block.Double
    Public Sub New(label As String)
        Me.Label = label
        Me.PresentationStyle = UI.Block.NumberPresentationStyle.Scale
        Me.MinimumValue = 0
        Me.MaximumValue = 1
        Me.ScaleLimits = False
    End Sub
End Class
```

Then, using two UnitSlider blocks, you could easily create a function for defining a point at (u, v) parameter values on a surface, for example. The code (in part) would be

```
selectSurf = New Snap.UI.Block.SelectObject()
selectSurf.SetFilter(Snap.NX.ObjectTypes.Type.Face)
selectSurf.LabelString = "Select face"

separatorLine = New Snap.UI.Block.Separator()

uSlider = New UnitSlider("U value")
vSlider = New UnitSlider("V value")

Me.AddBlocks(selectSurf, separatorLine, uSlider, separatorLine, vSlider)
```

and this would produce a dialog looking like this:



The Apply event handler would just use the slider values to create a point, like this:

```
Public Overrides Sub OnApply()
   Point(selectedFace.Position(uSlider.Value, vSlider.Value))
End Sub
```

Precedence of Values

In many situations, the values the user enters into a dialog are stored internally, so that they can be reloaded and used as default values the next time the dialog is displayed. You may have noticed this happening in the example above. This facility is called “dialog memory”. If your code is trying to control the contents of a dialog, it is important to understand how this reloading from dialog memory fits into the overall process. The chain of events is as follows:

(1) Values and options from the block creation code are used, then …

(2) Values from dialog memory are applied, and then …

(3) Values and options specified in the OnShow event handler are applied, and finally …

(4) The dialog is displayed

So, you can see that values and options you define in your block creation code might get overwritten by values from dialog memory. Since the OnShow event handler is executed just before the dialog is displayed, it gives you one last chance to set properties the way you want. On the other hand, your initial block creation code can set values that the OnShow event handler cannot. So, in short, setting properties in your block creation code gives you broader powers, but the OnShow event handler gives you stronger ones.

More Information

There are several examples of dialogs created with BlockForms in the folder
[…NX]\UGOPEN\SNAP\Examples\More Examples

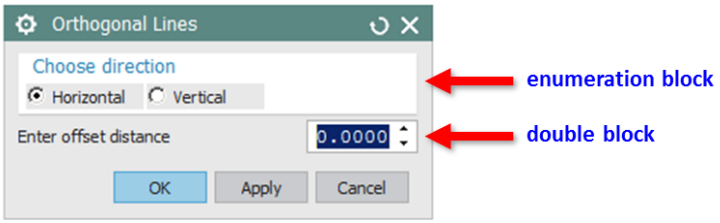
Chapter 14: Using Block UI Styler

In the previous chapter, we saw how we can create simple block-based dialogs using Snap.UI.BlockForm objects. In this chapter, we'll discuss an alternative approach that uses Snap.UI.BlockDialog objects, instead.

These two types of dialogs contain exactly the same kinds of blocks, and they behave in exactly the same way from the user's point of view. The main difference between the two is how you create them:

- To create a BlockForm, you write fragments of code to define blocks and add them to your dialog;
- To create a BlockDialog, you use the NX Block UI Styler to define blocks and arrange them on your dialog

We'll use the same "OrthoLines" example that we used in the previous chapter. As you may recall, this provides a simple dialog that lets the user create "infinite" lines in the horizontal or vertical directions in the XY-plane.



It only has two blocks – an "Enumeration" block to let the user choose either horizontal or vertical, and a "Double" block in which the user enters the offset distance (the distance from the line to the origin).

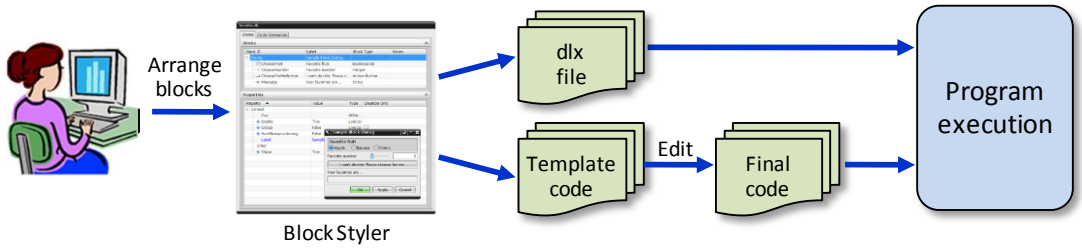
If you don't want to create this dialog yourself, using the instructions in this chapter, then you can find a completed version in [...NX]\UGOPEN\SNAP\Examples\More Examples\OrthoLines2.

The Overall Process

The overall process of developing a BlockDialog is as follows:

- You use Block UI Styler to choose the blocks you want, and arrange them on your dialog
- Block UI Styler creates a "dlx" file, and also some template code
- You edit the template code to define the behaviour you want
- At run-time, NX uses the dlx file plus your code to control the appearance and operation of the dialog

The process is illustrated in the following figure, and further details are provided below.

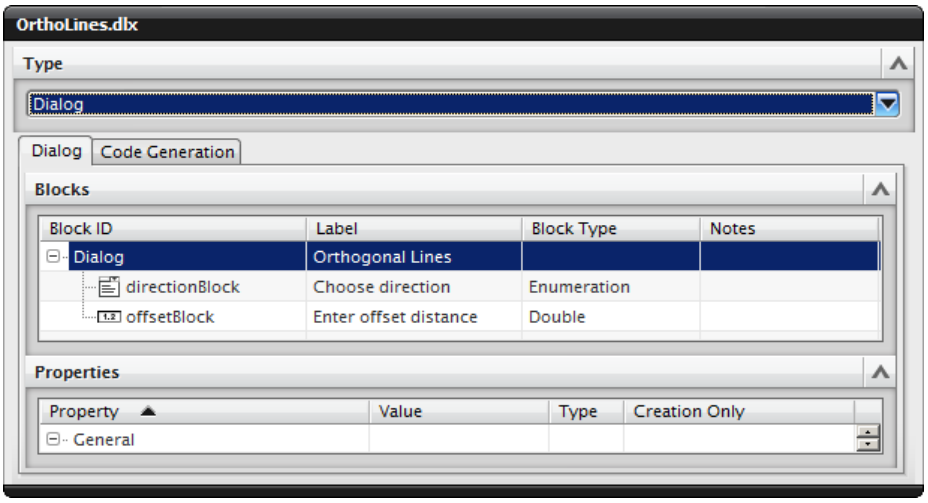


Using Block UI Styler

Instructions for using Block UI Styler are provided in the NX User Manual, but it is largely self-explanatory. Choosing a block type from the Block Catalog adds a new block to your dialog. You can then adjust its properties as desired. The process is similar to the one for designing WinForms that we saw in chapter 3.

In NX, access Block UI Styler via Start→All Applications→Block UI Styler. We could use Block UI Styler to create the dialog from scratch, but let's just open the file OrthoLines.dlx in Block UI Styler, instead — it has the dialog definition already created for you. You can find it in [...NX]\UGOPEN\SNAP\Examples\OrthoLines2.

The dialog has two blocks (`directionBlock` and `offsetBlock`), which you will see listed in Block UI Styler:



If you click on one of the blocks shown above, its properties will be shown in the lower half of the Block UI Styler window, and you can edit them as you wish. Some of the more important properties are shown below:

Block	Property	Value
directionBlock	Block ID	directionBlock
	Label	Choose direction
	PresentationStyle	Radio Box
	Layout	Horizontal
	Value	Horizontal Vertical

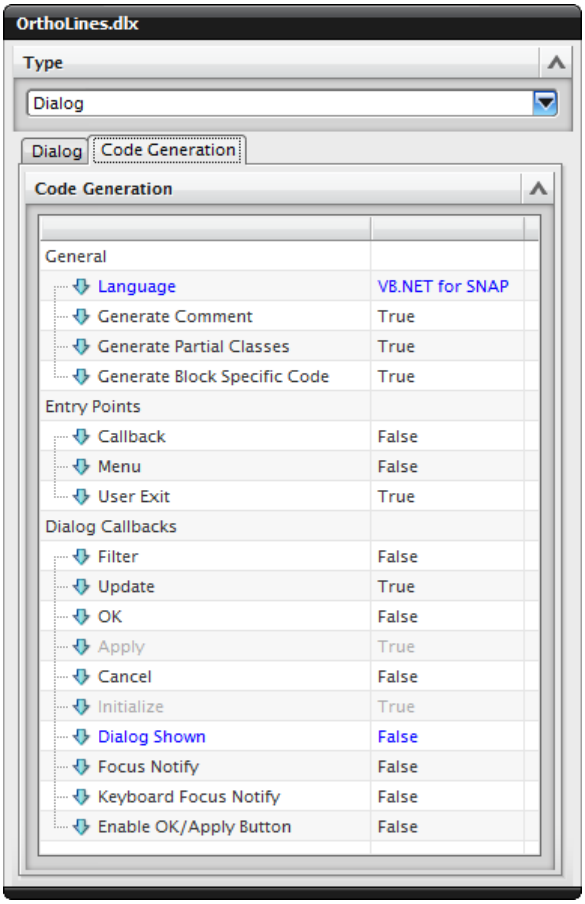
Block	Property	Value
offsetBlock	Block ID	offsetBlock
	Label	Enter offset distance
	PresentationStyle	Spin

If you're observant, you might see the correlation between the properties listed in Block UI Styler and the code we wrote when defining blocks to place on a BlockForm in chapter 12. In chapter 12, our code was:

```
directionBlock.Label = "Choose direction"
directionBlock.PresentationStyle = UI.Block.EnumPresentationStyle.RadioBox
directionBlock.Layout = UI.Block.Layout.Horizontal
directionBlock.Items = {"Horizontal", "Vertical"}
```

So you can see that Block UI Styler is really just providing us with a tabular way to edit the properties of blocks, instead of writing code.

When you have established all the blocks and properties you want, switch to the Code Generation tab in Block UI Styler, and define the settings as shown below:



Finally, choose File→Save, which will generate two VB files, called OrthoLines.vb and OrthoLines.Private.vb, and another file called OrthoLines.dlx.

Template Code

When you save a dialog in Block UI Styler, two Visual Basic files are created containing template code. The idea is that you "fill in the blanks" in this template code to define the way you want your dialog to behave. Actually, using our example from above, you should only modify the code in the OrthoLines.vb file. As its name implies, the other generated file (OrthoLines.Private.vb) is "private", and you are not supposed to edit it by hand. In fact, if you change the design of the dialog using Block UI Styler, saving the changes will over-write the "private" file, so your hand-crafted edits would be lost, anyway.

The contents of the VB files will depend on the options you chose in Block UI Styler. If you want to run your code from the Journal Editor, you should set "Generate Partial Classes" to False. This will force all the generated code to be placed in one file, which you can just open and execute from within the Journal Editor. If you generate two files (as described above), neither of them can be executed from within the Journal Editor.

The code shown below is a bare minimum. We have removed all the error-checking and most of the comments, in order to focus clearly on the essential concepts. In real working code, you should not do this, of course.

You're not really supposed to be looking at the "private" file, but if you choose to snoop, you will see code like this:

```
Partial Public Class OrthoLines
    Inherits Snap.UI.BlockDialog
    Public Shared theOrthoLines As OrthoLines
    Private directionBlock As Snap.UI.Block.Enumeration     ' Block type: Enumeration
    Private offsetBlock As Snap.UI.Block.Double             ' Block type: Double

and so on ...
```

As you can see, we are defining a new class called "OrthoLines" to represent instances of our dialog. Then there are two lines that declare variables to hold the two blocks that make up a "OrthoLines" dialog.

Then, further down, you will see a constructor:

```
Public Sub New(theDlxFileName As String)
   Me.NXOpenBlockDialog = New Snap.UI.BlockDialog(theDlxFileName).NXOpenBlockDialog
   Me.NXOpenBlockDialog.AddApplyHandler(AddressOf ApplyCallback)
   Me.NXOpenBlockDialog.AddInitializeHandler(AddressOf InitializeCallback)
End Sub
```

Most of this code is adding "event handler" callbacks to our dialog, as we requested when we saved the dialog from Block UI Styler. Inside these handler functions, we can write code that responds to "events" in the dialog. For example, when the user clicks the "Apply" button in the dialog, the "ApplyCallback" function will be called, so any code we place in that function (see below) will be executed. In this way, we can make the Apply button do something useful when the user clicks it.

Next, let's look at the contents of the OrthoLines.vb file (which we are intended to edit). Again, we have removed some error checking code to make the concepts clearer. First, there is the "Main" routine:

```
Public Shared Sub Main()
   Dim theDlxFileName As String = "OrthoLines.dlx"
   theOrthoLines = New OrthoLines(theDlxFileName)
   theOrthoLines.Show()
   theOrthoLines.Dispose()
End Sub
```

The first line defines the pathname of your dlx file, which you will have to change, depending on where you chose to place this file. The next two lines are automatically generated code that create a new "OrthoLines" dialog, and display it using the "Show" function.

As we saw in the previous chapter, the most interesting part of a dialog implementation is the code you put in the event handler functions, since this code determines how the dialog will react. When working with BlockDialog objects, we normally use the term "callback" rather than "event handler", but the meaning is the same. In fact, the event handler functions used with BlockDialog objects all have the word "callback" in their names.

In the "ApplyCallback" function, we'll use the same code we used in the OnApply function in the previous chapter:

```
Public Function ApplyCallback() As Integer
    Dim infinity As Double = 5000
    Dim d As Double = offsetBlock.Value
    If directionBlock.SelectedItem = "Horizontal" Then
        Line(-infinity, d, infinity, d)     ' Create a horizontal line
    Else
        Line(d, -infinity, d, infinity)     ' Create a vertical line
    End If
End Sub
```

As we have seen before, we use the `SelectedItem` property of `directionBlock` to decide whether to create a horizontal or vertical line, we read the offset distance from the `offsetBlock.Value` property, and then we create a line.

If you build and run this code, you should find that it works as expected. Entering some information and clicking Apply will create a line.

The behavior of the dialog will be exactly the same as the one we created in chapter 12. This is because, essentially, the two dialogs **are** the same. One is a BlockForm, and the other is a BlockDialog, and we constructed them using different techniques. But these differences are superficial — the fundamental point is that we used the same blocks, so we'll get the same appearance and behavior.

Callback Details

We've discussed the Update event handler and the Apply event handler quite a bit in the last two chapters. But some additional event handlers (callbacks) are available, especially with BlockDialog objects. The complete list of available callbacks is shown in the Code Generation tab of the Block UI Styler, and there you can choose the ones for which you want "stub" code generated. The table below indicates when NX calls each of these:

Callback function name	When NX calls this function
`UpdateCallback`	When the user changes something in the dialog
`OkCallback`	When the user clicks the OK button
`ApplyCallback`	When the user clicks the Apply button
`CancelCallback`	When the user clicks the Cancel button
`InitializeCallback`	Just before values are loaded from "dialog memory" (see below)
`DialogShownCallback`	Just before the dialog is displayed (see below)
`FocusNotifyCallback`	When focus is shifted to a block that cannot receive keyboard entry
`KeyboardFocusNotifyCallback`	When focus is shifted to a block that can receive keyboard entry

The OK, Apply and Cancel callbacks should each return an integer value. In the Cancel callback, this returned value is ignored, so its value doesn't matter. In the OK and Apply callbacks, returning zero will cause the dialog to be closed, and a positive value will cause it to remain open.

Precedence of Values, Again

As with BlockForm objects, it's sometimes important to understand what happens before a BlockDialog gets displayed, so that you can control its contents. The situation is a little more complex with BlockDialog objects, because the dlx file is involved, and there are more callbacks. Here is what happens:

(1) Values and options from the corresponding dlx file are used, then …

(2) Values and options specified in the InitializeCallback function are applied, and then …

(3) Values from dialog memory are applied, and then next…

(4) Values and options specified in the DialogShownCallback function are applied, and then finally …

(5) The dialog is displayed

So, you can see that values and options you set in the Initialize callback might get overwritten by values from dialog memory. Since the DialogShown callback is executed later, it does not suffer from this drawback. On the other hand, the Initialize callback can set values that the DialogShown callback cannot. So, in short, the Initialize callback gives you broader powers, but the DialogShown callback gives you stronger ones.

Getting More Information

This is a very simple example, of course. In more realistic cases, there will likely be much more code, but the basic structure will remain the same. The standard NX documentation set includes a manual describing the details of Block UI Styler. Also, the NXOpen samples folder contains eight examples of Block UI Styler dialogs. Its location is typically [...NX]\UGOPEN\SampleNXOpenApplications\.NET\BlockStyler

Chapter 15: Selecting NX Objects

In order to perform some operation on an NX object, the user will often have to select it, first. So, we need some way to support selection in our SNAP programs. You can use either a free-standing Selection.Dialog object, or a SelectObject block on a block-based dialog. The two approaches have much in common, and this chapter describes both of them.

Selection Dialogs

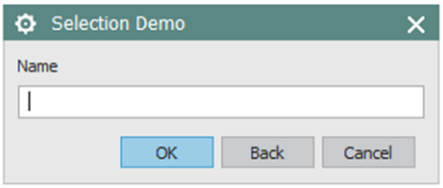
One way to support selection in SNAP is to use the tools in the Snap.UI.Selection class. The general process is:

- You construct a Selection.Dialog object
- You adjust its characteristics and behavior, if necessary
- You display it, so that it can gather information from the user
- A Selection.Result is returned to you, containing useful information that you can use in your program

Here is a short snippet of code illustrating this process:

```
Dim cue = "Please select a line to be hidden"
Dim dialog As Selection.Dialog = Selection.SelectObject(cue)

dialog.SetFilter(NX.ObjectTypes.Type.Line)
dialog.Title = "Selection Demo"
dialog.Scope = Selection.Dialog.SelectionScope.AnyInAssembly
dialog.IncludeFeatures = False

Dim result As Selection.Result = dialog.Show()

If result.Response <> NXOpen.Selection.Response.Cancel Then
    result.Object.IsHidden = true
End If
```

When the code shown above is executed, a small dialog appears giving the user the opportunity to select a line.



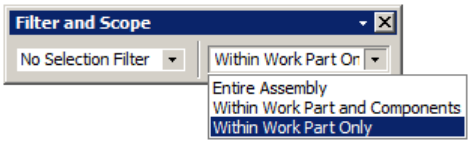
If the user selects a line and clicks OK, the selected line will be available to your code in the Selection.Result object, so you can do whatever you want with it. In the example above, we chose to make the line hidden (blanked).

Following are some details of the variables that affect the behavior of the dialog:

Argument	Type	Meaning
cue	String	The message displayed in the Cue line
dialog.Title	String	The title displayed at the top of the dialog
dialog.Scope	NXOpen.Selection.SelectionScope	The scope of the selection, explained below
dialog.IncludeFeatures	Boolean	If true, selecting features is allowed
dialog.Result	Selection.Result	Results returned from the selection process

The cue and dialog.Title variables are self-explanatory, so we won't discuss them further.

The scope argument indicates the domain from which the user will be allowed to select objects. In this case, we have specified that the selection scope should be the work part. The scope options correspond exactly to the choices shown by the Scope menu on the Selection toolbar in interactive NX.



The includeFeatures argument does exactly what it says — it determines whether or not the dialog will allow the user to select features.

The SetFilter function determines what type of object the dialog will allow the user to select. The NX Selection Filter will be pre-set according to the value of the type argument, and this restricts the user to choosing only certain types of objects. There are several other ways of specifying the types of entities that will be eligible for selection. Details are given below.

The Selection.Result object returned by the function has several fields. The most important ones are Result.Object, which indicates which object was selected, and Result.Response, which indicates how the user interacted with and closed the dialog (whether he clicked OK or Cancel, for example). The example code shows the typical process — you normally check the value of the response and then do something to the selected object based on this value.

In many cases, the default values of a Selection.Dialog object will be just what you need, so you won't need to adjust them before showing the dialog. Also, we can reduce our typing by putting `Imports NX.ObjectTypes` at the top of our file and by taking advantage of the Visual Basic "infer" option. By using all these tricks, the code shown above can typically be shortened to something like the following:

```
Dim dialog = Selection.SelectObject("Please select a line to be hidden")
dialog.SetFilter(NX.ObjectTypes.Type.Line)
Dim result = dialog.Show()
If result.Response <> NXOpen.Selection.Response.Cancel Then
    result.Object.IsHidden = true
End If
```

As you can see, we use a shared (static) function called SelectObject to create a Selection.Dialog. A cue string is input to this function, since this has no reasonable default value. Other variables are available as properties of the dialog that you can modify after it has been created. But these properties have plausible default values that you often will not need to modify, which saves you from writing a few lines of code.

SelectObject Blocks

Sometimes, you will want to support selection inside a larger block-based dialog, rather than using a standalone selection dialog. To do this, you place a SelectObject block on your dialog. As we know from earlier chapters, SNAP provides two types of block-based dialogs: BlockDialog objects, and BlockForm objects. We'll be using a BlockForm in the example below. The basic steps are as follows:

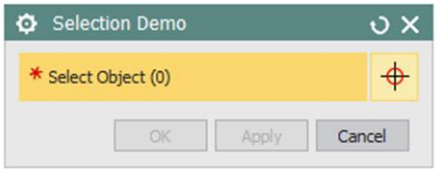
- You create a BlockForm object
- You create a SelectObject block
- You adjust the block's characteristics and behavior, if necessary
- You add the SelectObject block to the BlockForm
- You display the BlockForm, so that it can gather information from the user
- You retrieve useful information from the BlockForm, so that you can use in your program

Although they are created by different processes, and in different contexts, SelectObject blocks are quite similar to standalone SelectObject dialogs. In particular, they both use the same kind of SetFilter functions, so, you only have to learn the SetFilter techniques once.

Here is a snippet of code illustrating the use of SelectObject block on BlockForm:

```
Dim dialog As New BlockForm()
dialog.Title = "Selection Demo"

Dim selectionBlock As New Block.SelectObject()

selectionBlock.Cue = "Please select a line to be hidden"
selectionBlock.SetFilter(NX.ObjectTypes.Type.Line)
selectionBlock.MaximumScope = Block.SelectionScope.AnyInAssembly

dialog.AddBlocks(selectionBlock)

Dim response = dialog.Show()

If response <> UI.Response.Cancel
    selectionBlock.SelectedObjects(0).IsHidden = true
End If
```

When this code is executed, a small dialog appears, giving the user the opportunity to select a line:



If the user selects a line and clicks OK, the line will be hidden (blanked).

Just as we saw with the Selection.Dialog earlier, there is a SetFilter function that determines what type of object the block will allow the user to select.

After the user has selected some objects, these are available in the SelectedObjects property of the block, so you can retrieve them and process them however you wish.

Types, Subtypes, and TypeCombos

There are several different ways to specify the types of objects that are to be eligible for selection. In simple cases, you can just pass a single type parameter to a SetFilter function, as we did in the examples above. Two further examples of this simple approach are:

```
Dim dialog As Selection.Dialog
Dim result As Selection.Result
Dim cue As String

' Select a datum plane
dialog = Selection.SelectObject("Select a datum plane")
dialog.SetFilter(Type.DatumPlane)
result = dialog.Show()

' Select a body (solid or sheet)
cue = "Please select a body (solid or sheet)"
dialog = Selection.SelectObject(cue)
dialog.SetFilter(Type.Body)
result = dialog.Show()
```

In this example, and in most of the following ones, we'll assume that we have Imports Snap.NX.ObjectTypes at the top of our file. This allows us to write Type.Body instead of NX.ObjectTypes.Type.Body, and so on.

Though you don't really need to know this, the values in the enumeration NX.ObjectType correspond very closely with the values in NXOpen.UF.Constants. So, for example, NX.ObjectTypes.Type.Line actually has the same value as UFConstants.UF_line_type. But the values in NX.ObjectTypes are much easier to find and use, since there are far fewer of them.

In more complex cases, you might want to specify that two or more different types of objects should be selectable. You do this just by passing an array of types to a SetFilter function, as follows:

```
'To select a point or a line
cue = "Please select a point or a line"
dialog = Selection.SelectObject(cue)
dialog.SetFilter(Type.Point, Type.Line)
result = dialog.Show()
```

As you may know, certain types of NX objects have "subtypes". For example "ellipse" is a subtype of the type "conic curve", and "note" is a subtype of "drafting entity". You can see this type-subtype structure when you browse the NX type hierarchy, and you also see a somewhat modified version when you use Detailed Filtering in interactive NX. To obtain finer control of the selection process you can pass both types and subtypes to SetFilter functions.

Here are two examples: first, selecting an ellipse:

```
' To select an ellipse
dialog = Selection.SelectObject(cue)
dialog.SetFilter(Type.Conic, SubType.ConicEllipse)
result = dialog.Show()
```

and then selecting a note:

```
' To select a note
dialog = Selection.SelectObject(cue)
Dim type = NX.ObjectTypes.Type.DraftingEntity
Dim subtype = NX.ObjectTypes.SubType.DraftingEntityNote
dialog.SetFilter(type, subtype)
result = dialog.Show()
```

A somewhat more interesting example is selection of a planar face. The code is as follows:

```
'To select a planar face
dialog = Selection.SelectObject("Select a face")
Dim type = NX.ObjectTypes.Type.Face
Dim subtype = NX.ObjectTypes.SubType.FacePlane
dialog.SetFilter(type, subtype)
result = dialog.Show()
```

In this example, note that "Face" is a type, and "FacePlane" is a subtype. If you are familiar with NX Open selection functions, you will notice that the approach used here is quite different from the "MaskTriple" technique they use. There are SNAP SetFilter functions that allow you to continue using the MaskTriple approach, if you want to, but it is not likely that you will ever need them. We recommend that you use the new type-subtype approach shown here, since it is usually much simpler.

A combination of a type and a subtype is known as a TypeCombo. You can bundle a type and a subtype together into a TypeCombo, and pass this to a SetFilter function, instead of passing the type and subtype separately. Of course, if you only have one type-subtype combination, it's easier to pass these directly to the SetFilter function — there's no point in using a TypeCombo. The TypeCombo technique only becomes useful when you want to specify several type-subtype combinations, which you can do by using a TypeCombo array.

This is illustrated in the following example, where we want to allow the user to select either a circular edge or a cylindrical face (because either of these could represent a hole in a part, perhaps):

```
'TypeCombo for circular edges
Dim type1 = Type.Edge
Dim subtype1 = SubType.EdgeCircle
Dim circularEdgeCombo = New TypeCombo(type1, subtype1)

'TypeCombo for cylindrical faces
Dim type2 = Type.Face
Dim subtype2 = SubType.FaceCylinder
Dim cylinderFaceCombo = New TypeCombo(type2, subtype2)

'To select either circular edge or a cylindrical face
dialog = Selection.SelectObject("Select hole")
Dim combos As TypeCombo() = { circularEdgeCombo, cylinderFaceCombo }
dialog.SetFilter(combos)
```

Selecting Faces, Curves and Edges

Quite often, we will want to allow the user to select a face of any type (planar, cylindrical, conical, etc.). In other situations, you might want the user to be able to select only cylindrical or conical faces. Of course, you could do this using the TypeCombo techniques described above, but it would be quite a lot of work. This is such a common situation that SNAP provides special SetFaceFilter functions that make things easier. You can just pass an array of face subtypes directly to a SetFaceFilter function, like this:

```
' To select a cylindrical or conical face
dialog = Selection.SelectObject(cue)
Dim faceSubTypes As SubType() = { SubType.FaceCylinder, SubType.FaceCone }
dialog.SetFaceFilter(faceSubTypes)
```

Another common situation is selection of both curves and edges. Again, you can do this by using arrays of TypeCombos, but SNAP provides an easier approach using a function called SetCurveFilter.

For example, the following code defines a SelectObject block that will allow the user to select either a circle or a circular edge. The argument of the SetCurveFilter function controls both curve types and edge types, all in one.

```
Dim selectionBlock As New Block.SelectObject()
selectionBlock.Cue = "Please select a circle"
selectionBlock.SetCurveFilter(Type.Circle)
```

Using the Cursor Ray

You can think of selection as a process of shooting an infinite line (the cursor ray) at your model. The object that gets selected is one that this ray hits, or the one that's closest to the ray. Sometimes, rather than just knowing which object was selected, you want to know where on the object the user clicked. You can figure this out by using the cursor ray. For example, you can find out where the ray intersects the model, or you can find out which end of a curve is closest to the ray. The following example shows a typical application — we use the cursor ray to create a point at the location on a line where the user clicked to select it:

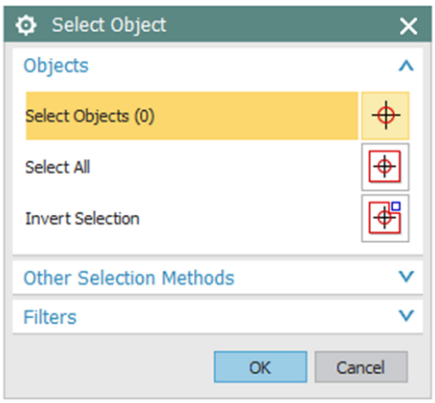
```
Imports Snap, Snap.Create, Snap.UI, Snap.NX.ObjectTypes, Snap.Compute

Public Class SelectionTest

   Public Shared Sub Main()

      Dim cue As String = "Please select a line"
      Dim dialog As Selection.Dialog = Selection.SelectObject(cue, Type.Line)
      Dim selectionResult As Selection.Result = dialog.Show()
      If selectionResult.Response <> NXOpen.Selection.Response.Cancel Then
         Dim result As Snap.Compute.DistanceResult
         result = Compute.ClosestPoints(selectionResult.Object, selectionResult.CursorRay)
         Point(result.Point1)
      End If

   End Sub

End Class
```

The selectionResult.Object is the line the user selected. We use the Compute.ClosestPoints function to find the point on this line that's closest to the cursor ray. The result of this calculation contains two points; the first one, Point1, is on the line, and the second one is on the ray. We create a point at Point1 to show the location.

In a parallel view, the cursor ray is parallel to the z-axis vector of the view. In perspective views, things are somewhat more complicated, but the code shown above still works.

Multiple Selection

Selecting multiple objects uses very similar techniques. Selection.Dialog objects and SelectObject blocks both have an AllowMultiple property, and you just have to set this to True. On a Selection.Dialog, this will cause the standard NX multi-selection dialog to appear



This dialog allows the user to select objects in all the usual ways. As with single selection, the available options in the selection filter will be pre-set to restrict the range of different object types that are selectable.

The selection result contains an array called Objects that holds all the selected objects. Typically, your code will cycle through this array, doing something to each object in turn. For example:

```
dialog = Selection.SelectObjects(cue, type)
Dim result As Selection.Result = dialog.Show()
For Each obj In result.Objects
   obj.Color = System.Drawing.Color.Red
Next
```

You can use standard .NET functions to operate on the array of selected objects. For example, Objects.Length gives you the number of objects selected, and Objects.ConvertAll lets you convert it to some other type.

Selection by Database Cycling

Another way to "select" objects is to gather them while cycling through an NX part file. In this case, the selection is done by your code, rather than by the user, but some of the ideas are somewhat similar, so the topic is included in this chapter.

To get all the objects of a certain type in a given part file, you use various "collection" properties of the Snap.NX.Part class. For example, the Curves array gives you all the curves in a part file, and the Bodies collection gives you all the bodies. You can then cycle through one of these collections using the usual For Each construction, doing whatever you want to each object in turn. Often, you will be dealing with the work part, which you can obtain as Snap.Globals.WorkPart. This first example hides all the wire-frame curves in the work part:

```
Dim workPart As Snap.NX.Part = Snap.Globals.WorkPart

For Each curve In workPart.Curves
   curve.IsHidden = True
Next
```

This example moves all the sheet bodies in the work part to layer 200:

```
For Each body In workPart.Bodies
   If body.ObjectSubType = Snap.NX.ObjectTypes.SubType.BodySheet
      body.Layer = 200
   End If
Next
```

Finally, this last example makes all the planar faces green:

```
For Each body In workPart.Bodies
   For Each face In body.Faces
      If TypeOf face is NX.Face.Plane
         face.Color = System.Drawing.Color.Green
      End If
   Next face
Next body
```

A New Way

Using `For Each` to cycle through collections of objects is the traditional approach, but there's a new way that offers some improvements in readability. Suppose you have a plate with a large number of holes, and you want to gather together the larger ones, and sort them by their diameter. The old way is to do something like this:

```
' Get all the arcs (holes) in the work part
Dim holes = Snap.Globals.WorkPart.Arcs

' Cycle through them, gathering up the big ones
Dim bigHoles = New List(Of NX.Arc)()
For Each h In holes
   If h.Radius > 5 Then bigHoles.Add(h)
Next

' Then do the sorting (somehow)
```

But there's a tidier way: if you put `Imports System Linq` at the top of your file, then you can write:

```
' Filter to include only the big holes
Dim bigHoles = holes.Where(Function(hole) hole.Radius > 5)

' Sort the holes according to radius
Dim sortedHoles = holes.OrderBy(Function(hole) hole.Radius)

'Chain together the filtering and sorting
Dim sortedBigHoles = holes.Where((Function(h) h.Radius > 5)).OrderBy(Function(h) h.Radius)
```

This is certainly much shorter, and it's easier to understand, in my opinion. It uses LINQ queries and some new things called "lambda expressions", which are special types of "anonymous functions". Look up these terms on the internet, if this approach looks useful to you. If nothing else, you'll be able to amaze your impressionable colleagues by using terms like "lambda expression", and "anonymous function".

Chapter 16: The Jump to NX Open

As we have mentioned before, the design of SNAP is focused on simplicity, rather than completeness. So, at some point, you will find that SNAP doesn't do what you need, and you'll want to use NX Open to supplement it.

The standard NX documentation tells you how to call any of the thousands of functions available in NX Open, but many people find it hard to see the "big picture", so they don't know where to start. This chapter explains the conceptual model behind NX Open programming, to make it easier to find the functions you need.

As a sample problem, let's suppose we want to create some simple object like a sphere or a circular arc or a point in an NX Open program. This is easy using SNAP, of course, as we saw in chapter 7 and chapter 8, but we're going to do it using NX Open, instead, to demonstrate the principles involved.

The NX Open Inheritance Hierarchy

The first thing to consider is the hierarchical structure of NX object types. There are hundreds of different object types, so the complete picture is difficult to understand (or even to draw). The simplified diagram below shows us the path from the top of the hierarchy down to the simple objects we are interested in.

- **NXRemotableObject**
  - **TaggedObject**
    - **NXObject**
      - **DisplayableObject**
        - Body
        - DatumAxis
        - DatumPlane
        - Edge
        - Face
        - FacetedBody
        - Sketch
        - SmartObject
          - Point
          - Curve
            - Line
            - Conic
              - Arc
              - Ellipse
            - Spline
          - CoordinateSystem
          - Axis
          - Direction
          - Plane
          - Scalar
          - Xform
      - **Feature**
        - BodyFeature
          - Sphere
      - **NXMatrix**
      - **Expression**
      - **BasePart**
        - Part
    - **Builder**
      - **FourPointSurfaceBuilder**
      - **FeatureBuilder**
        - BooleanBuilder
        - SphereBuilder
    - **BaseSession**
      - **Session**
  - **TaggedObjectCollection**
    - **BaseFeatureCollection**
    - **PartCollection**
    - **FeatureCollection**
  - **MathUtils**

So, we see that a Point is a kind of "SmartObject", which is a kind of "DisplayableObject", and so on. The details are given later, but briefly, here are the roles of the more important object types:

RemotableObject
Used for collections of preferences and also as the basis of the "UF" classes

TaggedObject
Used for lists of objects, for selections, and for "builders" (to be described later)

NXObject
Used for Part objects, and for objects that live inside NX part files, but are not displayed — views, layouts, expressions, lights, and so on. NXObjects have names and other non-graphical attributes.

DisplayableObject
Includes most of the object types familiar to users. Things like annotations, bodies, facetted bodies, datum objects, CAE objects. Displayable objects have colors, fonts, and other appearance attributes. Note that NX features are not displayable objects, as we explained earlier, in chapter 10

SmartObject
Includes points, curves, and some object types used as components of other objects when implementing associativity.

Sessions and Parts

Typical NX objects (the ones we're discussing, here, anyway) reside in part files. So, the first thing we must do is identify a part file in which our new objects will be created. The relevant code is:

```
Dim mySession As NXOpen.Session = NXOpen.Session.GetSession()  ' Get the current NX session
Dim parts As NXOpen.PartCollection = mySession.Parts           ' Get the session's PartCollection
Dim myWorkPart As NXOpen.Part = parts.Work                     ' Get the Work Part
```

As you can see, we first get the current NX session object by calling GetSession. Every session object has a PartCollection object called "Parts" which we obtained in the second line of code. Then we get the Work Part from this PartCollection. Of course, as always, we could have reduced our typing by putting `Imports NXOpen` at the top of our code file.

In addition to the Work Part, there are other useful objects that you will probably want to initialize at the beginning of your program. Examples are the Display Part, the "UI" object, the "Display" object, the UFSession object, and so on. You will see code like this near the top of almost every NX Open program:

```
Dim theSession As Session = Session.GetSession()     ' Assumes "Imports NXOpen" above
Dim parts As PartCollection = theSession.Parts
Dim theWorkPart As Part = parts.Work
Dim theDisplayPart As Part = parts.Display
Dim theUfSession As UF.UFSession = UF.UFSession.GetUFSession()
Dim theDisplay As DisplayManager  = theSession.DisplayManager
Dim theUI As UI = UI.GetUI()
```

Object Collections

We saw above that there are specific NX object types corresponding to the Point and Arc objects we are planning to create. It might seem natural that the Point class will contain a function for creating points, and the Arc class will contain a function for creating arcs. But, it doesn't work this way. The NX Open view is that a part file contains "collections" of different object types. So, for example, given a Part object named myPart, there is a collection called myPart.Points that contains all the Point objects in the part. Similarly, myPart.Arcs is a collection that contains all the arcs in this part, and myPart.Curves includes all the curves.

These collections are the key to creating new objects in a part file. In the NX Open view of the world, when you create a new point, you are adding a new point object to some PointCollection object. Specifically, if you create a new point in myPart, you are adding to the PointCollection called myPart.Points. So, the CreatePoint function can actually be found in the PointCollection class, and you use it as follows to create a point:

```
Dim coords As new Point3d(3, 5, 0)                    ' Define coordinates of point
Dim workPart As Part = parts.Work                     ' Get the Work Part
Dim points As PointCollection = workPart.Points       ' Get the PointCollection of the Work Part
Dim p1 As Point = points.CreatePoint(coords)          ' Create the point (add to collection)
p1.SetVisibility(SmartObject.VisibilityOption.Visible)
```

The last line of code is necessary because an NXOpen.Point is a "SmartObject", which is invisible by default. If you try hard, you can create a point (an invisible one, again) with a single line of code. Here it is:

```
Dim p1 As Point = NXOpen.Session.GetSession().Parts.Work.Points.CreatePoint( New Point3d(3,5,0) )
```

Following this scheme, you might expect that a LineCollection object would have functions for creating lines, an ArcCollection object would have functions for creating arcs, and so on. This is partly correct, but the truth is a little more complex:

- A LineCollection object has one function (CreateFaceAxis) for creating lines
- An ArcCollection object has no functions for creating arcs
- A SplineCollection object has no functions for creating splines
- A CurveCollection object has about a dozen functions for creating lines, arcs, and conics

So, to create a line in the work part, the code is:

```
Dim curves As CurveCollection = theWorkPart.Curves      ' Get the CurveCollection of the Work Part
Dim p1 As New Point3d(1, 2, 3)                          ' Define start point of line
Dim p2 As New Point3d(5, 4, 7)                          ' Define   end point of line
curves.CreateLine(p1, p2)                               ' Create the line (add to collection)
```

These collections are very useful if you want to cycle through all the objects of a certain type within a given part file. For example, to perform some operation on all the points in a given part file (myPart) you can write:

```
For Each pt As Point In theWorkPart.Points
  ' Do something with pt
Next
```

A Part object has many collection objects that can be used in this way, including collections of bodies, faceted bodies, annotations, dimensions, drawing sheets, and so on. See the documentation for the NXOpen.Part class for further details.

Note that, even though the Line class (for example) does not help us create lines, it does help us edit them and get information from them. Once we have created a Line object, we can use functions and properties in the Line class to modify it. For example, there are functions like NXOpen.Line.SetStartPoint and NXOpen.Line.SetEndPoints.

Features and Builders

The discussion above covers the case of simple geometry like points, lines, and arcs. Next, let's look at a more complex object like our Sphere feature. The code to build a sphere feature is as follows:

```
[1]  Dim nullSphere As NXOpen.Features.Sphere  = Nothing
[2]  Dim mySphereBuilder As NXOpen.Features.SphereBuilder
[3]  mySphereBuilder = theWorkPart.Features.CreateSphereBuilder(nullSphere)
[4]  mySphereBuilder.Property1 = <whatever you want>
[5]  mySphereBuilder.Property2 = < whatever you want >
[6]  Dim myObject As NXOpen.NXObject = mySphereBuilder.Commit()
[7]  mySphereBuilder.Destroy()
```

So, the general approach is to

- create a "builder" object (line [3])
- modify its properties as desired (lines [4] and [5])
- "commit" the builder to create a new object (line [6])

As you can see in line [3] above, the functions to create various types of "builder" objects are methods of a FeatureCollection object, and we can get one of these from workPart.Features.

A SphereBuilder object is fairly simple, but other feature builders are very complex, with many properties that you can set.

Exploring NX Open By Journaling

The NX Open API is very rich and deep — it has thousands of available functions. This richness sometimes makes it difficult to find the functions you need. Fortunately, if you know how to use the corresponding interactive function in NX, the journaling facility will tell you which NX Open functions to use, and will even write some sample code for you. You choose Tools→Journal→Record to start recording, run through the desired series of steps, and then choose Tools→Journal→Stop Recording. The code generated by journaling is verbose and is often difficult to read. But it's worth persevering, because hidden within that code is an example call showing you exactly the function you need. You can indicate which language should be used in the recorded code by choosing Preferences→User Interface→Tools→Journal. The available choices are C#, C++, Java, Python, and Visual Basic.

The “FindObject” Problem

When you use a journal as the starting-point for an application program, one of the things you need to do is remove the “FindObject” calls that journaling produces. This section tells you how to do this.

A journal records the exact events that you performed during the recording process. If you select an object during the recording process, and do some operations on it, the journal actually records the name of that object. So, when you replay the journal, the operations will again be applied to this same named object. This is almost certainly not what you want — you probably want to operate on some newly-selected object, not on the one you selected during journal recording. Very often, objects with the original recorded names don’t even exist when you are replaying the journal, so you’ll get error messages.

To clarify further, let’s take a specific example. Suppose your model has two objects in it — two spheres named SPHERE(23) and SPHERE(24). If you record a journal in which you select all objects in your model, and then blank them, then what gets recorded in the journal will be something like this:

```
Dim objects1(1) As DisplayableObject

Dim body1 As Body = CType(workPart.Bodies.FindObject("SPHERE(23)"), Body)
objects1(0) = body1

Dim body2 As Body = CType(workPart.Bodies.FindObject("SPHERE(24)"), Body)
objects1(1) = body2

theSession.DisplayManager.BlankObjects(objects1)
```

If you replay this code, it’s just going to try to blank SPHERE(23) and SPHERE(24) again, which is probably useless. There’s a good chance that SPHERE (23) and SPHERE (24) won’t exist at the time when you’re replaying the journal, and, even if they do, it’s not likely that these are the objects you want to blank. Clearly we need to get rid of the “FindObject” calls, and add some logic that better defines the set of objects we want to blank. There are a few likely scenarios:

- Maybe we want to blank some objects that were created by code earlier in our application
- Maybe we want to blank some objects selected by the user when our application runs
- Maybe we want to blank all objects in the model, or all the objects that have certain characteristics

The first of these is easy: if we created the objects in our own code, then presumably we assigned them to program variables, and they are easy to identify:

```
Dim myBall0 As NX.Body = Sphere(1,2,1, 5).Body
Dim myBall1 As NX.Body = Sphere(1,4,3, 7).Body

Dim objects1(1) As DisplayableObject

objects1(0) = myBall0
objects1(1) = myBall1

theSession.DisplayManager.BlankObjects(objects1)
```

For the second case, we need to add a selection step to our code as outlined in chapter 14, and then blank the objects the user selects when the journal is replayed. Something like this:

```
Dim cue = "Please select the objects to be blanked"
Dim dialog As Selection.Dialog = Selection.SelectObjects(cue)

Dim result As Selection.Result = dialog.Show()

If result.Response <> NXOpen.Selection.Response.Cancel Then
    theSession.DisplayManager.BlankObjects(result.Objects)
End If
```

For the third case (blanking all the objects with certain characteristics), we will need to cycle through all the objects in our model, finding the ones that meet our criteria, and then pass these to the BlankObjects function. See the last two sections in chapter 15 for information about cycling through the objects in a part file.

Mixing SNAP and NX Open

As we have seen, NX Open functions provide enormous power and flexibility, but SNAP functions are usually much easier to find and understand. So, there may well be situations where you will want to use SNAP and NX Open functions together. To do this, you will need to convert SNAP objects into NX Open objects, and vice-versa. We have tried to make these conversions as convenient as possible, so that SNAP and NX Open code can live together in peace and harmony.

A SNAP object is just a simple wrapper around a corresponding NX Open object — for example, a Snap.NX.Spline object is just a wrapper that encloses an NXOpen.Spline, and a Snap.NX.Sphere is a wrapper around an NXOpen.Features.Sphere object, and so on. So, if you have an NXOpen object, you can "wrap" it to create a Snap.NX object. In the other direction, if you have a Snap.NX object, you can "unwrap" it to get the NXOpen object that it encloses. There are hidden "implicit" conversions that do this wrapping and unwrapping for you, so often things just work without any extra effort. For example:

```
Dim snapPoint As Snap.NX.Point = Point(3,5,9)
Dim session As NXOpen.Session = NXOpen.Session.GetSession()
session.Information.DisplayPointDetails(snapPoint)
Dim pt As NXOpen.Point3d = snapPoint.Position
```

In the third line, we are passing a Snap.NX.Point object to a function that expects to receive an NXOpen.Point. But the implicit conversion is invoked behind the scenes, and the function call just works as expected. Similarly, in the fourth line, we are assigning a Snap.Position object to an NXOpen.Point3d object, and this works, too.

However, there are times when the implicit conversions don't work, and you need to do something more explicit. For example, if you want to use NXOpen member functions or properties, then you have to get an NXOpen object from your SNAP object first. So suppose, for example, that we have a Snap.NX.Sphere object called `snapSphere`, and we write the following code:

```
snapSphere.HideParents()                          ' Fails
Dim version = snapSphere.TimeStamp                ' Fails
```

Both lines of code will fail, because a Snap.NX.Sphere object does not have a HideParents method or a TimeStamp property. So, to proceed, you have to "unwrap" to get the enclosed NXOpen.Features.Sphere object. You can do this in a couple of different ways, as shown below:

```
CType(snapSphere, NXOpen.Features.Sphere).HideParents()     ' Works, but a bit clumsy
snapSphere.NXOpenSphere.HideParents()                       ' Nicer: use NXOpenSphere property
```

The first line just uses the standard VB CType function to do the conversion, and the second line uses the `NXOpenSphere` property. The second approach, using properties, is the most convenient, so there are several properties that let you get NXOpen objects from SNAP objects in this same way. For example, if `snapSphere` is a Snap.NX.Sphere object, again, then

- `snapSphere.NXOpenSphere` is the enclosed NXOpen.Features.Sphere object
- `snapSphere.NXOpenTag` is the NXOpen tag of this NXOpen.Features.Sphere object
- `snapSphere.SphereBuilder` is the "builder" object for the NXOpen.Features.Sphere

Going in the other direction (from NXOpen to SNAP) is not quite so streamlined. The approach using properties is not available, so you have to call the `Wrap` function to create a new SNAP object from the NXOpen one, like this:

```
Dim coords = New NXOpen.Point3d(3, 6, 8)
Dim workPart As NXOpen.Part = Snap.Globals.WorkPart.NXOpenPart
Dim nxopenPoint As NXOpen.Point = workPart.Points.CreatePoint(coords)
Dim snapPoint As NX.Point = NX.Point.Wrap(nxopenPoint.Tag)        ' Create a Snap.NX.Point
Dim location As Position = snapPoint.Position                     ' Use its Position property
```

In the fourth line of code, we first get the tag of the NXOpen.Point object. Then we call the Wrap function, which gives us a new Snap.NX.Point object that "wraps" it. Then, in the last line, we can use the Position property of this new Snap.NX.Point object.

As we saw above, the `Wrap` function receives an NXOpen.Tag as input. So, if you are working with older NXOpen functions that use tags, interoperability with SNAP is even easier. For example:

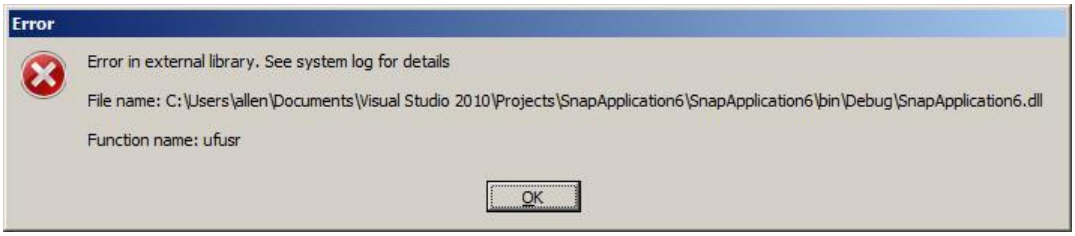
```
Dim ufSession = NXOpen.UF.UFSession.GetUFSession()
Dim pointTag As NXOpen.Tag
Dim coords As Double() = {2, 6, 9}
ufSession.Curve.CreatePoint(coords, ByRef pointTag)
Dim snapPoint As NX.Point = NX.Point.Wrap(pointTag)
```

Chapter 17: Troubleshooting

This chapter describes a few things that might go wrong as you are working through the examples in this guide, and how you can go about fixing them. If they occur at all, you will probably encounter these problems fairly early in your learning process. But then, once you solve them, they will probably not re-appear, and you should be able to continue your education without any further disruptions.

Using the NX Log File

If things go wrong in a SNAP program, you might receive a message like this:



The "external library" is your code, and the message is telling you there's something wrong with it. The "system log" that the message mentions is the NX Log File (traditionally known as the NX "syslog"), which you can access via the Help→Log File command from within NX. This log file typically contains a large amount of text, some of which can be very useful in diagnosing problems. After an error, the useful information is usually at the bottom of the syslog, so you should start at the end and work backwards in your search for information. The typical text, about 50 lines from the end of the syslog, will look something like this:

```
NXOpen.NXException: Attempt to use an object that is not alive
   at NXOpen.TaggedObject.get_Tag()
   at NXOpen.DisplayableObject.Blank()
   at Snap.NX.NXObject.set_IsHidden(Boolean value)
   at SnapApplication6.MyProgram.Main() in   SnapApplication6\MyProgram.vb:line 9
*** EXCEPTION: Error code  3600041 in line 1987 of <blah, blah, blah>
+++ Attempt to use an object that is not alive
```

I deliberately caused this error by deleting an object and then trying to "Blank" it (make it hidden). As you can see, NX is quite rightly complaining that I am attempting to use an object that is no longer alive, and this caused the get_Tag function to fail . The syslog text is quite helpful here, as is often the case.

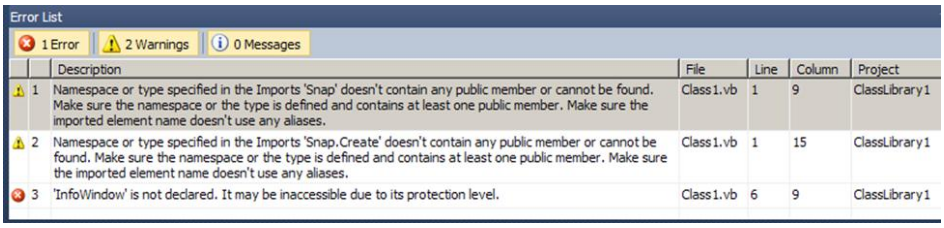
Invalid Attempt to Load Library

To use SNAP, you need to have a fairly recent version of the .NET Framework installed on your computer. For any version of NX, the Release Notes document lists the required version; NX 10 requires .NET Version 4.5. If you don't have the correct .NET version installed, you will receive this mysterious error message

the first time you try to run any code in the Journal Editor. You will find similar text in the NX syslog, too. To check which version(s) of the .NET Framweork you have installed, look in your Windows\Microsoft.NET\Framework folder, or use the "Programs and Features" Control Panel. If you don't have the correct version, please download and install it from this Microsoft site. If you find that the link to the Microsoft site is broken, you can easily find the download by searching the internet for ".NET Framework".

No Public Members; Inaccessible Due to Protection Level

At some point in your work, you may encounter a mysterious error message saying "Namespace or type specified in the Imports "Snap" doesn't contain any public member or cannot be found". And there may be further complaints saying that some function may be "inaccessible due to its protection level".



If you run into this problem at all, it will probably be the first time you try to build a SNAP application in Visual Studio. It arises because your code is using the SNAP library, and this is not connected in any way to your current project. So the compiler doesn't know anything about Snap, Snap.Create, or the InfoWindow function.

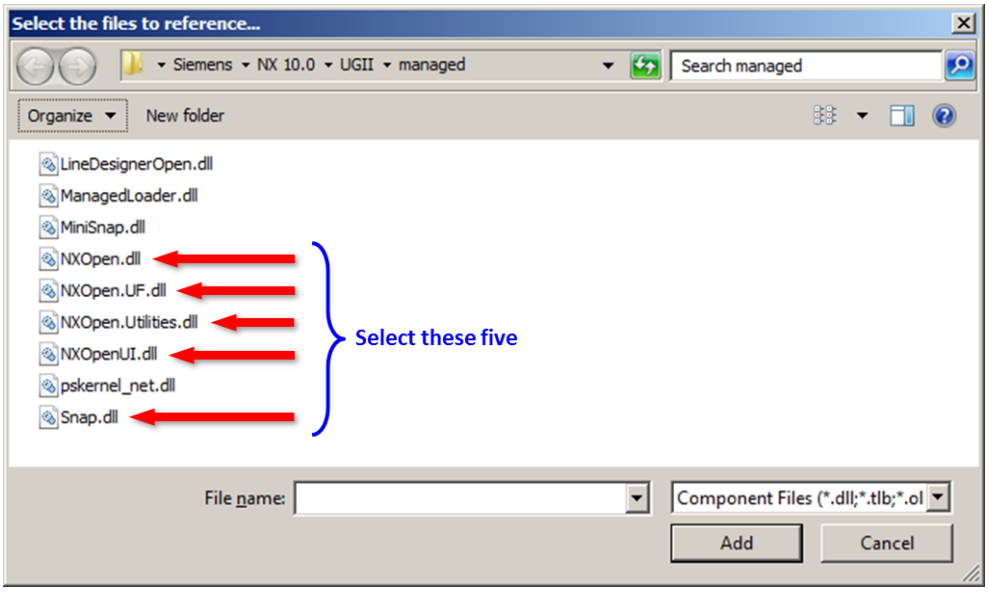
There are several possible causes of this disconnection

- Your project does not have a reference to the SNAP library
- You have a reference, but it's "broken" (pointing to the wrong place)
- The reference is correct, but a licensing problem is preventing the SNAP library from being loaded

The first two problems (missing or broken references) will happen only when using Visual Studio. When you run code in the Journal Editor, referencing of SNAP and NX/Open libraries is all handled inside NX, so it's not likely to go wrong. The third problem (licensing) can happen in both environments. Let's look at the three issues in turn:

Missing References

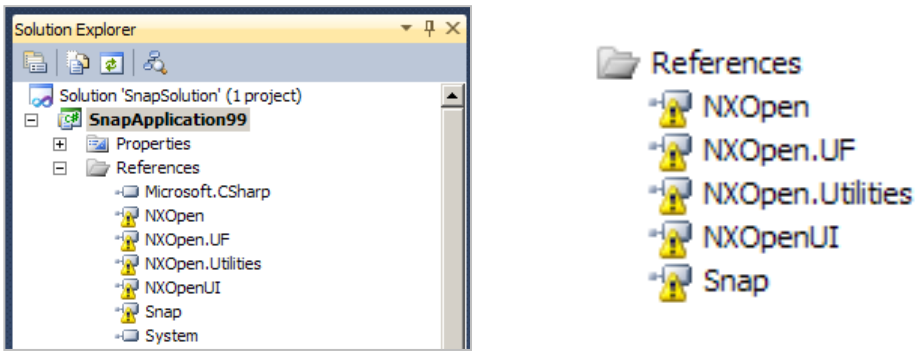
To see if this is the cause of the problems, look in the References folder in the Solution Explorer pane (usually in the upper right of the Visual Studio window). If you don't see SNAP listed there, then things aren't going to work. This problem could arise because you used some generic template (rather than a SNAP template) to create your project, as we described in example 4 in chapter 3. Fortunately, this problem is easy to fix. From the Project menu, choose Add Reference. In the dialog that appears, click on the Browse tab, and navigate to [...NX]\UGII\managed.



Select the five needed DLLs, as shown above, and click OK. Your project now has references to the SNAP and NX Open libraries, and this should stop the complaints.

Broken References

Maybe your project includes references to the SNAP and NX Open libraries, but these references are "broken" (pointing to the wrong locations). You can find out if this is the case by looking in the References folder in Solution Explorer, again. The little yellow triangular "caution" signs indicate broken references:



The SNAP application templates use the UGII_ROOT_DIR environment variable to establish the references, so, if this environment variable is set incorrectly, the references won't work. To fix the problem, you have to delete the broken references and create new ones. Right-click on each reference in Solution Exporer, and choose "Remove". Then create new references as described in the previous section. If your UGII_ROOT_DIR environment variable is set incorrectly, you should fix it, or else you'll get annoying broken references in every project you create.

Licensing Problems

Unlike the first two problems, this one can happen when you are working either in the Journal Editor or in Visual Studio. In the Journal Editor, the error message might look like this:

Again, this means that the compiler isn't able to access the SNAP assembly (Snap.dll). If you are using Visual Studio, the first thing you should do is make sure your references are correct, as outlined in the previous two sections. If you have done that, and the problem persists, take a look in the NX Log File (the "syslog"). Working backwards through the syslog, you'll first find something you already know:

```
Journal execution results...
Syntax errors:
Line 8: 'InfoWindow' is not declared. It may be inaccessible due to its protection level.
```

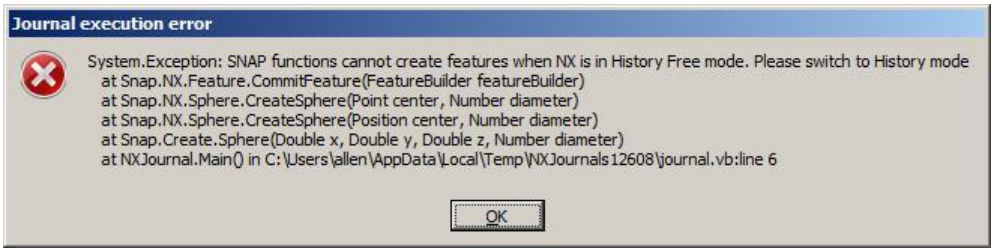
But, then, a few lines before this, you may see something like the following:

```
Adding C:\Program Files\Siemens\NX 10\ugii\managed\\NXOpen.Utilities.dll as a reference item
Adding C:\Program Files\Siemens\NX 10\ugii\managed\\NXOpen.dll as a reference item
Adding C:\Program Files\Siemens\NX 10\ugii\managed\\NXOpen.UF.dll as a reference item
Evaluating whether to add Snap library:
************** Licensing Information **************
Server ID : For Internal Siemens PLM Use Only
 Webkey Access Code : server module
 License File Issuer : Siemens PLM, Inc,
 Flex Daemon Version : 0.0
 Vendor Daemon Version String : No Version
**************************************************
Cannot obtain authoring license
```

As you can see, the system was able to add references to the various NXOpen dlls, which means that your license server must be working OK. But trying to add Snap.dll failed because of a licensing failure. The failure probably means that no nx_snap_author license is available for you. If you're just experimenting, you may be able to work around this problem by using MiniSNAP instead of SNAP. But, if you're doing serious work with SNAP, maybe it's time to buy a few more authoring licenses.

Cannot Create Features in History-Free Mode

As explained at the end of chapter 10, a SNAP program cannot create features if NX is in "history free" mode. When running from the Journal Editor, you will receive an error message something like this:



Similar text will be written to the System Log:

```
Journal execution results...
Runtime error:
System.Exception: SNAP functions cannot create features when NX is in History Free mode. Please switch to
History mode
   at Snap.NX.Feature.CommitFeature(FeatureBuilder featureBuilder)
   at Snap.NX.Sphere.CreateSphere(Point center, Number diameter)
   at Snap.NX.Sphere.CreateSphere(Position center, Number diameter)
   at Snap.Create.Sphere(Double x, Double y, Double z, Number diameter)
   at NXJournal.Main() in C:\Users\allen\AppData\Local\Temp\NXJournals12608\journal.vb:line 6
```

The particular error messages shown above result from trying to create a Sphere feature. Trying to create any other type of feature would cause a similar error, of course. As the message suggests, you should set Snap.Globals.HistoryMode = True before creating any features.

Visual Studio Templates Missing

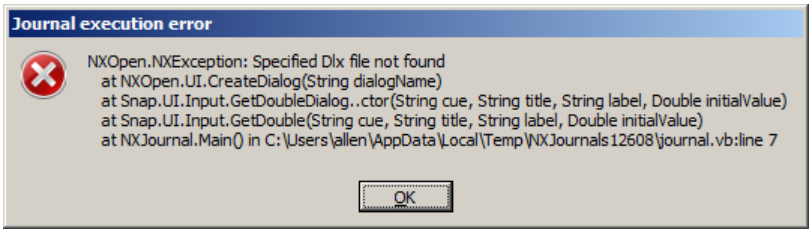
When you start working through the examples in chapter 3, you may find that the SNAP project templates (SnapTemplateVB, SnapWinFormTemplateVB and SnapBlockFormTemplateVB) are not listed in the "New Project" dialog in Visual Studio. There are a few possible causes for this problem. First, maybe you forgot to copy the template zip files, as instructed near the beginning of chapter 3. You can find the three necessary zip files in the folder [...NX]\UGOPEN\SNAP\Templates. You need to copy these three files into the folder [My Documents]\Visual Studio 2012\Templates\ProjectTemplates\Visual Basic.

You may find other folders with names like C:\Program Files\Microsoft Visual Studio\Common\IDE\Templates if you hunt around your disk. None of these are the correct destination for the SNAP templates, despite the unfortunate similarity of names.

Finally, despite the warning in big red letters in chapter 3, maybe you unzipped the three zip files. You should not do this — Visual Studio cannot use them if they are unzipped.

Dlx File Not Found

The simple input dialogs described in chapter 12 depend on some dlx files that are located in the folder [...NX]\UGOPEN\SNAP\dialog. The file double.dlx is used by the GetDouble function, and so on. If the system cannot find these dlx files, for some reason, you will receive error messages like this:

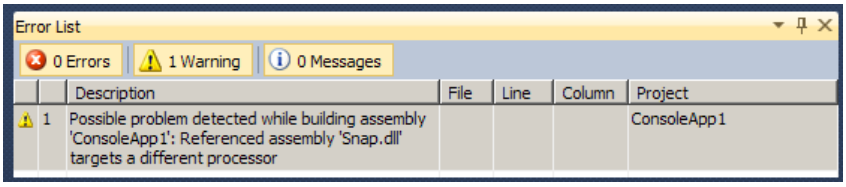


and similar messages will be written to the System Log. NX locates the needed dlx files by using the UGII_BASE_DIR environment variable. If this is not set correctly, then simple input dialogs will not work (and many other things will go wrong, too, actually).

Failed to Load Image

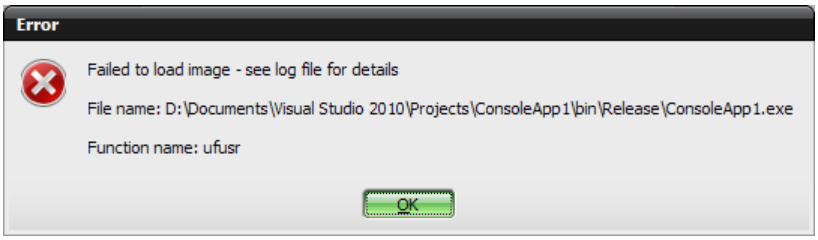
The "Failed to Load Image" error occurs when there is a mismatch between the type of your NX installation and the type of SNAP application you created. Specifically, you will get this error if you have a 64-bit version of NX but you try to run a 32-bit SNAP application.

If you're watching carefully, you will see an indication of this problem when you build your SNAP application. You will receive a warning like the one shown below:



The message indicates that the SNAP dll targets a 64-bit processor (since it is part of 64-bit NX), but your application (ConsoleApp1) targets a 32-bit processor. But this is only a warning, so the build will succeed.

However, when you try to run your SNAP application, you will get this error:



If you look in the NX syslog, will find something like this:

```
The reason ...\ConsoleApp1.exe failed to load was:
Cannot classify image ...\ConsoleApp1.exe
```

Again, this indicates that 64-bit NX was unable to load and run your application because it was built for a 32-bit architecture. With the full version of Visual Studio, you can avoid this problem by specifying what type of application you want to build. But in Visual Studio Express, there is less flexibility in this area, so you have to be careful to base your projects on the right type of template. With the Visual Studio "Console Application" template, the default target is a 32-bit architecture, so you may run into problems if you are using a 64-bit version of NX. If you always use the SNAP project templates we provide, then things should go smoothly.

That's All Folks

This seems like a strange way to end our tour of SNAP, but having a separate "wrap up" chapter would be even more ridiculous, so we'll just stop here. We hope this introduction has been useful to you, and that you will want to explore SNAP further. As we have told you many times before, you can find out (much) more about the details of the available functions by consulting the SNAP Reference Manual. Bon voyage!