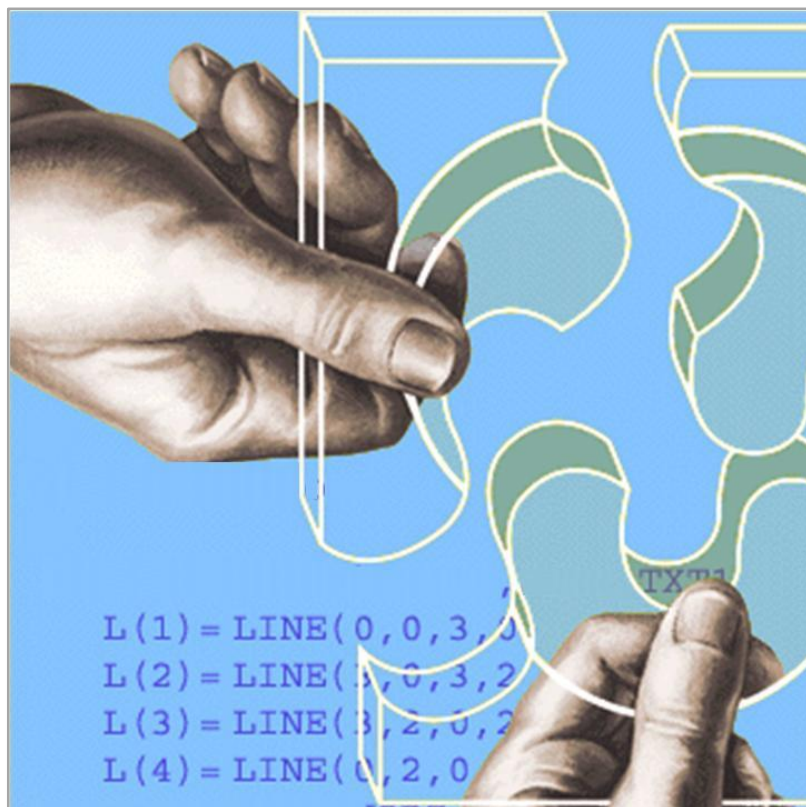


# Getting Started with SNAP

Revision 1.0. August 2011



# SIEMENS

© 2011 Siemens Product Lifecycle Management Software Inc.

# Table of Contents

<b>Chapter 1: Introduction .....</b>	<b>1</b>
What Is SNAP ?.....	1
Purpose of this Guide .....	1
Where To Go From Here .....	1
Other Documentation.....	2
Example Code.....	3
<b>Chapter 2: Using the NX Journal Editor.....</b>	<b>4</b>
System Requirement – The .NET Framework.....	4
Typographic Conventions .....	4
Example 1: Hello World.....	4
Example 2: Creating Simple Geometry.....	6
Example 3: Some More Interesting Geometry.....	7
Example 4: Getting Input from the User.....	8
Example 5: Using Vectors.....	9
Example 6: Using .NET Tools.....	9
Example 7: WinForms (The Hard Way).....	11
What Next ? .....	12
<b>Chapter 3: Using Visual Studio Express .....</b>	<b>13</b>
Getting Started.....	13
Installing SNAP Templates.....	14
Example 1: Hello World Again.....	14
Example 2: Declaring Variables .....	16
Example 3: WinForms Again.....	18
Example 4: Hello World Yet Again (the Hard Way) .....	21
Example 5: Toolpath Simulation.....	23
<b>Chapter 4: The Visual Basic Language .....</b>	<b>25</b>
The Development Process.....	25
Structure of a Visual Basic Program.....	25
An Example Program.....	26
Lines of Code.....	27
Built-In Data Types .....	28
Declaring and Initializing Variables .....	28
Data Type Conversions.....	29
Arithmetic and Math.....	29
Logical Values & Operators.....	30
Arrays .....	30
Strings .....	31
Enumerations.....	32
Other Types of Collections .....	32
Nothing .....	32
Decision Statements .....	33
Looping.....	33
Functions and Subroutines .....	34
Optional Arguments for Functions .....	35
Arrays as Function Arguments.....	35
Classes.....	36
Shared Functions .....	37
Object Properties .....	38
Hierarchy & Inheritance.....	38
<b>Chapter 5: SNAP Concepts &amp; Architecture .....</b>	<b>39</b>
Relationship of SNAP to NX Open .....	39
SNAP Files.....	39
The SNAP Architecture.....	40
SNAP Design Principles.....	41
<b>Chapter 6: Positions, Vectors, and Points.....</b>	<b>43</b>
Positions.....	43
Vectors .....	44
Points.....	45
<b>Chapter 7: Curves.....</b>	<b>47</b>
Lines .....	47
Arcs and Circles.....	47
Splines.....	48
Bezier Curves.....	49
<b>Chapter 8: Feature Concepts .....</b>	<b>51</b>
What is a Feature ?.....	51
Features Versus Bodies.....	51
Feature Display Properties.....	52
More Feature/Body Confusion .....	53
Feature Parameters – the Number Class.....	53
Creating Features .....	54
Other Feature Functions.....	54
History-Free Mode.....	55
<b>Chapter 9: Simple Solids and Sheets.....</b>	<b>56</b>
Creating Primitive Solids.....	56
Extruded Bodies.....	56
Revolved Bodies.....	57
B-surfaces.....	58
<b>Chapter 10: Object Properties &amp; Methods .....</b>	<b>60</b>
NXObject Properties.....	60
Curve and Edge Properties .....	62
Face Properties .....	64
<b>Chapter 11: NX Block-Based Dialogs.....</b>	<b>65</b>
When to Use Block-Based Dialogs .....	65
The Overall Process .....	65
Using Block Styler .....	66
Template Code.....	67
Callback Details.....	69
Precedence of Values .....	69
Getting More Information .....	69
<b>Chapter 12: Simple Input and Output .....</b>	<b>70</b>
Entering Numbers and Strings.....	70
Choosing from Menus.....	71
Specifying Positions, Vectors, and Planes.....	71
The Role of DLX Files .....	72
Writing Output .....	72
Windows Output.....	73
<b>Chapter 13: Selecting NX Objects .....</b>	<b>74</b>
The Basic Idea.....	74
A Bit More Detail.....	74
Types, Subtypes, and TypeCombos .....	76
Selecting Curves, Edges, and Faces.....	77
Using the Cursor Ray.....	78
Multiple Selection.....	78
Selection by Database Cycling.....	79
<b>Chapter 14: The Jump to NX Open.....</b>	<b>80</b>
The NX Open Inheritance Hierarchy .....	80
Sessions and Parts.....	81
Object Collections.....	81
Features and Builders.....	82
Exploring NX Open By Journaling.....	82
The “FindObject” Problem .....	82
<b>Chapter 15: Troubleshooting.....</b>	<b>84</b>
Using the NX Log File .....	84
If You Don’t Have .NET Framework V4.....	84
If You Don’t Have a Snap Authoring License .....	85
If You Try to Create Features in History-Free Mode.....	85
Can’t Find Visual Studio Templates.....	86
Can’t Find dlx Files.....	86

# 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 API's, 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 average 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're old enough to 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.

Like any .NET library, **SNAP** 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.

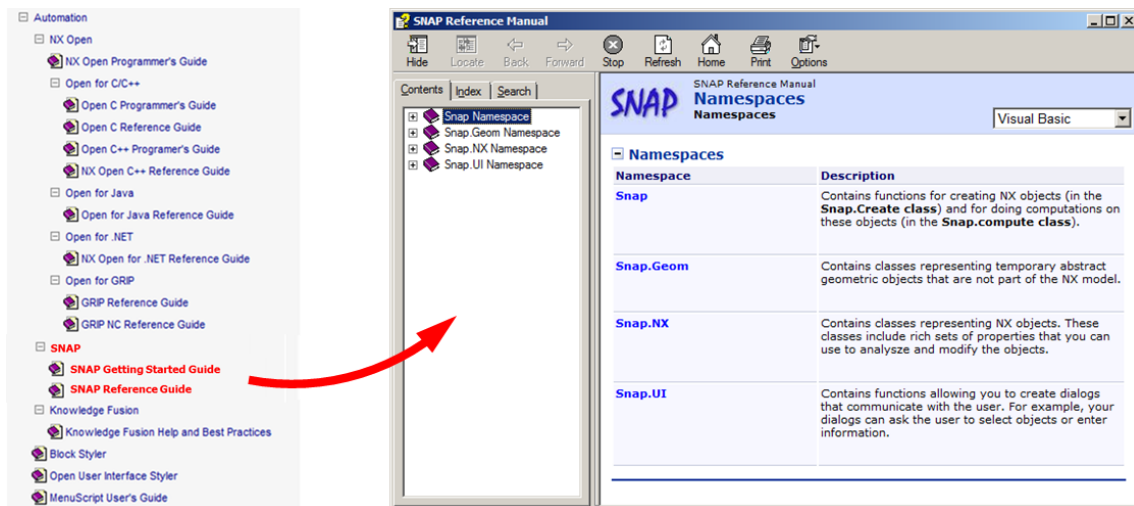
In Chapter 5, we provide a brief overview of **SNAP** concepts and architecture. It's not really necessary for you to know all of this, but understanding the underlying principles might help you to learn things more quickly.

Chapters 6 through 13 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 Manual or by using the Object Browser in Visual Studio.

In Chapter 14, 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 understand. If you find that [SNAP](#) alone does not provide everything you need, you can use NX Open to plug the gaps.

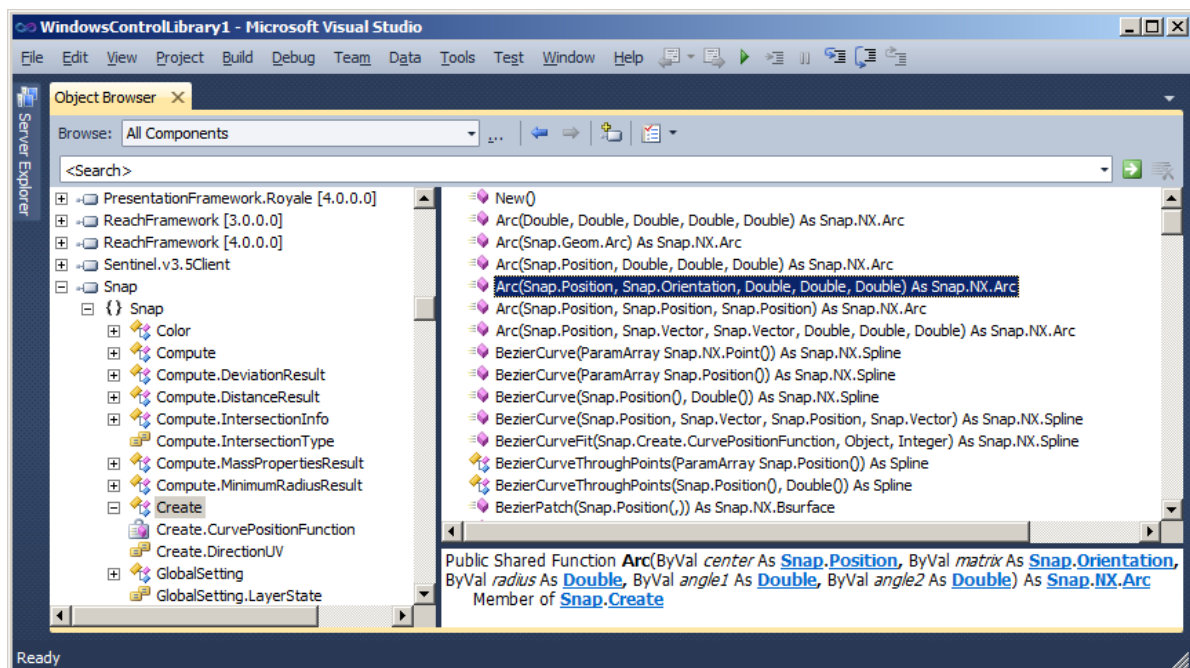
## Other Documentation

The definitive source of information about the capabilities of [SNAP](#) is the [SNAP](#) Reference Manual, 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.

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



This won't give you access to the example programs and explanatory text that are in the Reference Manual, 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 Manual 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.

## 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 Manual, there are several hundred small 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 useful, but will probably find them helpful in understanding function usage.
- There are some examples in C:\Program Files\Siemens\NX8.0\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 useful things.

If you’ve read everything, and you’re still stuck, you can contact Siemens GTAC support, or you can ask questions in the NX Languages Forum [here](#).

## 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 good way 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. This requires a small setup effort, but provides a much nicer development environment.

### System Requirement – The .NET Framework

To use **SNAP**, you need a fairly recent version of the .NET Framework – version 4.0 or newer. To check which version you have, look in your **Windows\Microsoft.NET\Framework** folder, and see if it contains a folder named v4.0.xxxx. Alternatively, you can use the “Programs and Features” Control Panel to check. If you don't have version 4 or later, 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”.

### 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.

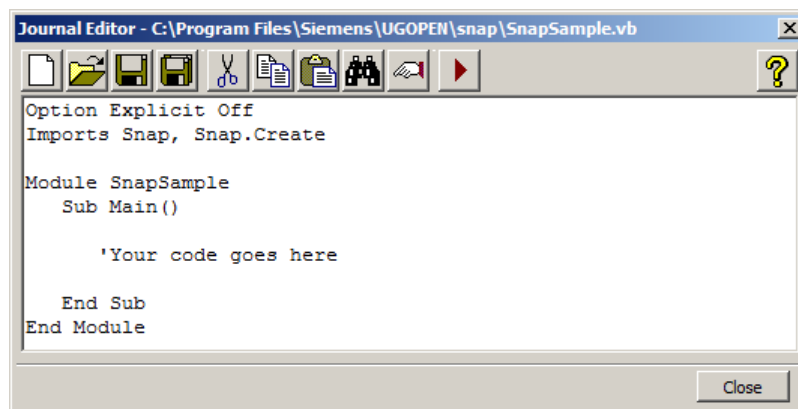
### Example 1: Hello World

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.

We will write the code in Visual Basic, using the NX Journal Editor. The steps are as follows:

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 the location where NX is installed, such as **C:\Program Files\Siemens\NX 8.0\UGOPEN\SNAP**. You should see this text:



The text 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

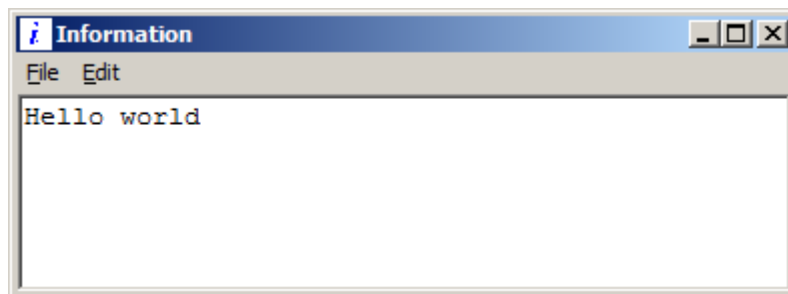
    End Sub
End Module
```

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 messed up, or the line endings get lost. 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 find the example code in [C:\Program Files\Siemens\NX8.0\UGOPEN\SNAP\Examples](#) instead.

Once you have the right text in the Journal Editor, delete the line that says “Your code goes here” 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 far right). This will send your code to the Visual Basic compiler, producing some object code (which you won't see), and then it will run that object code. The NX Information window should appear, like this:



If you receive some sort of error, rather than the output shown above, there are a few things that could conceivably have gone wrong:

- Maybe you typed something incorrectly, in which case the 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
- Maybe your system doesn't have any nx\_snap\_author licenses available (either because you didn't purchase any, or they are all in use by other people)

There is a troubleshooting guide in Chapter 15 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 prepared to just accept it as magic, you can skip directly to example 2.

Lines of code	Explanation
<code>Option Explicit Off</code> <code>Imports Snap, Snap.Create</code>	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.
<code>Module SnapSample</code>	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".
<code>Sub Main</code>	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.
<code>InfoWindow.WriteLine("Hello world")</code>	We call a SNAP function to write a line of text to the NX Info Window
<code>End Sub</code>	The end of our subroutine, Main
<code>End Module</code>	The end of our module

## Example 2: Creating Simple Geometry

In this next example, we create some simple geometry. Again, run NX, create a new part file, and then choose Tools→Journal→Edit (or press Alt+F11). In the Journal Editor window, open the file [SnapSample.vb](#).

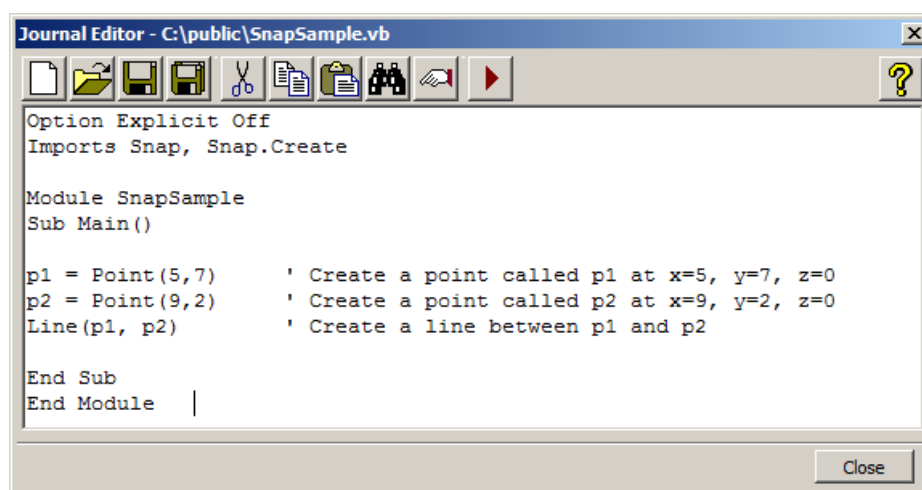
As before, delete the line that says "Your code goes here", and type in the following code. If you don't like typing, you can copy the code from here and paste it into the Journal Editor window. The paste operation in the Journal Editor is available on the right-mouse-button menu, or you can use the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



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

You may be wondering how you're supposed to know that the **SNAP** function for creating a line is just "Line", and not "CreateLine", or "BuildLine", or "Line\_Between\_Two\_Points". The answer is that all the **SNAP** functions are documented in detail, along with example code, in the [SNAP Reference Manual](#). 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.

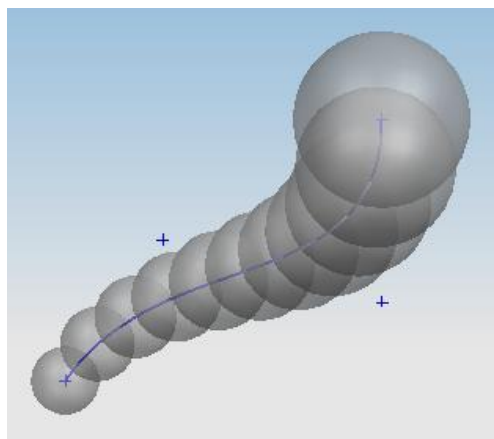
```
p1 = Point(0, 0) : p2 = Point(6, 0) : p3 = Point(6, 6) : p4 = Point(6, 6, 6)

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                         ' Calculate point on spine curve
    p = spine.Position(t)                 ' Calculate radius
    r = factor^count                      ' Create sphere
    Sphere(p, 2*r)
Next
```

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).



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

Lines of code	Explanation
<code>spine = BezierCurve(p1, p2, p3, p4)</code>	Creates a Bezier curve from the four points p1, p2, p3, p4. A Bezier curve is just a simple kind of spline curve.
<code>For count = 0 to 10</code> <code>&lt;the body of our loop&gt;</code> <code>Next</code>	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.
<code>t = count*0.1</code>	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
<code>p = spine.Position(t)</code>	Calculates a position on the curve “spine” at the parameter value t.
<code>r = factor^count</code>	Calculates a radius value by raising “factor” to the power “count”
<code>Sphere(p, 2*r)</code>	Creates a sphere with center location p and diameter 2*r

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 doing 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
```

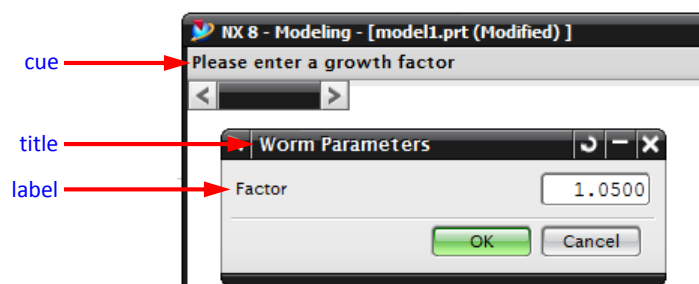
The statement `Circle(p, v, r)` creates a circle with center p, normal vector v, and radius r. You can try increasing the value of n to something very large so see how fast **SNAP** code can create geometry. If it takes longer than 10 seconds to create 5000 circles, maybe it's time for you to buy a new 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. You can build far more sophisticated dialogs using either the NX Block Styler 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 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. 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.

## 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. At the time of writing (mid-2010), there are about 10,000 “classes” in the framework, organized into several hundred categories called “namespaces”. Some of the more interesting ones are listed below

Namespace	Description
System	Base types like <a href="#">String</a> , <a href="#">DateTime</a> , <a href="#">Boolean</a> , plus <a href="#">arrays</a> , math functions, etc.
System.Collections	Provides <a href="#">collections</a> used in programming, such as <a href="#">lists</a> , <a href="#">queues</a> , <a href="#">stacks</a> , etc.
System.Data	Functions to access data and data services.
System.Diagnostics	Debugging 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 <a href="#">streams</a> , such as files
System.Management	Provides system information, such free disk space, CPU utilization, etc.
System.Media	Provides you the ability to play system sounds and .wav files.
System.Messaging	Networking and <a href="#">.NET Remoting</a>
System.Text	Supports various encodings, <a href="#">regular expressions</a> , and string manipulation tools
System.Threading	Helps facilitate multithreaded programming.
System.Timers	Allows you to raise an event on a specified interval.
System.Windows.Forms	Tools for building graphical user interfaces (menus, toolbars and dialogs)
System.Xml	Reading, writing and processing <a href="#">XML</a> data

For more details, see [http://en.wikipedia.org/wiki/Framework\\_Class\\_Library](http://en.wikipedia.org/wiki/Framework_Class_Library).

In this example, we will use a .NET function for reading data from a bitmap file. 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.

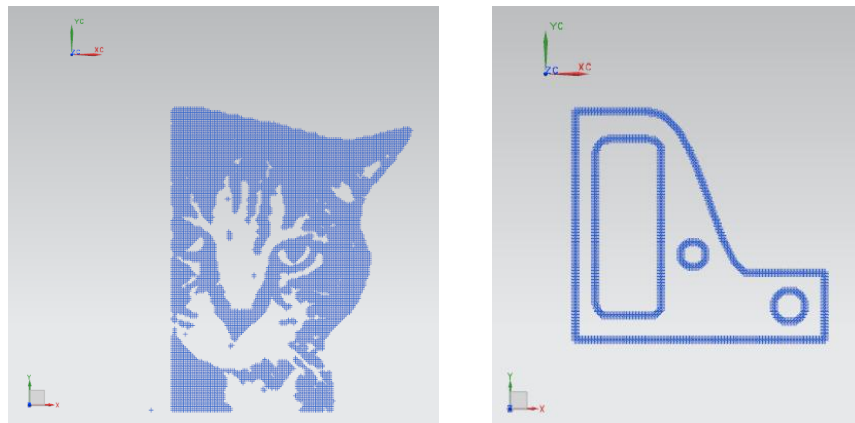
```
Sub Main
  pixels = New System.Drawing.Bitmap("C:\sammie.jpg")      ' Reads the file. Change this !!

  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 be valuable – 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, you can modify the code as follows:

```
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.

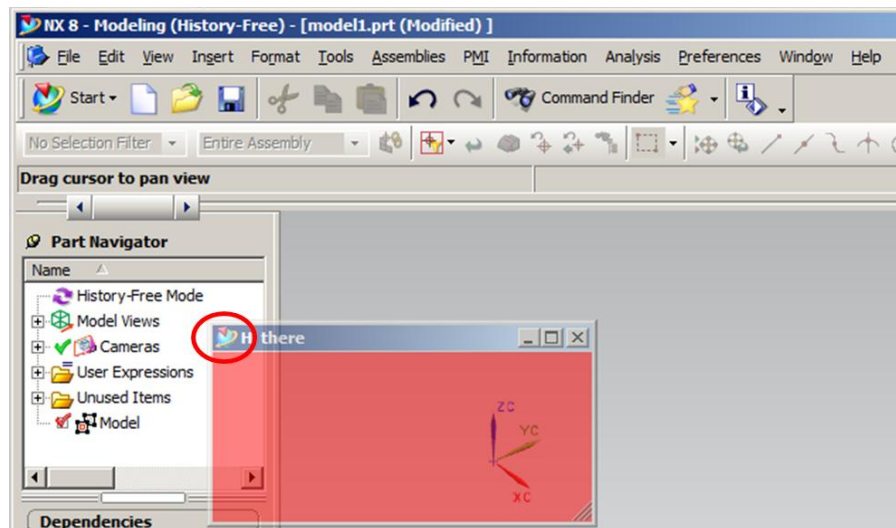
## Example 7: WinForms (The Hard Way)

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 Styler has similar tools, and produces dialogs that are more consistent with the rest of NX, as explained in chapter 11. 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.WinForms()
    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.WinForms](#), 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.

Currently, our WinForm doesn't do anything, so let's change that. Modify the code in [SnapSample.vb](#) as follows:

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

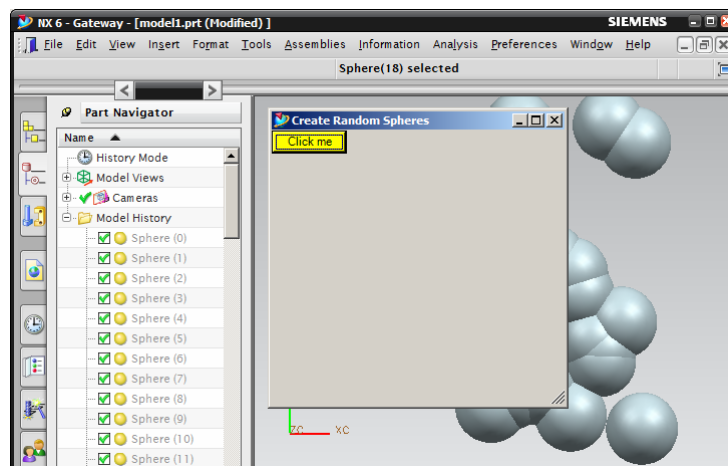
Module SnapSample
    Dim WithEvents myButton As Button           'A button for our form
    Dim rand As Random                          'A .NET random number generator

    Sub Main()
        rand = New Random()
        myForm = New Snap.UI.WinForms()
        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

    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
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 have added a few lines of code that create a “button” and place it on our form. Also, we added a new subroutine called [Handler](#). This type of subroutine is called an “event handler” – specifically, it handles 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 you 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.



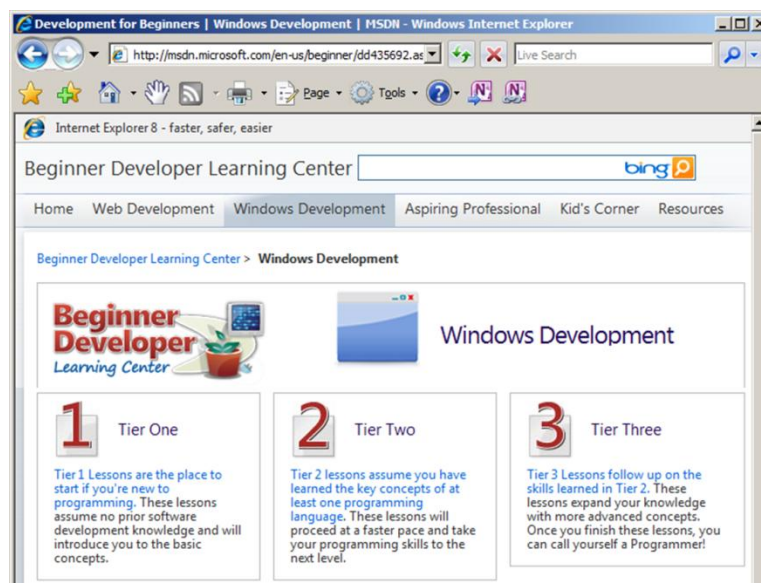
# Chapter 3: Using Visual Studio Express

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 2010 Express package.

## Getting Started

If you already have some version of Visual Studio installed on your computer, and you are familiar with it, you can skip this section and proceed directly to the first example.

Microsoft and others provide a large amount of tutorial material for Visual Studio. If you have no programming experience at all, perhaps the best place to start is at [Microsoft’s “Beginner Developer” web page](http://msdn.microsoft.com/en-us/beginner/dd435692.aspx) for Windows development:



This page has links to many tutorial documents and videos, and some of the links are reproduced in this document, for easy access. Similar Microsoft pages are available in other languages, and you may prefer one of these if your native language is not English.

The “Tier One” category includes several introductions to Visual Studio programming and the Visual Studio Express IDE

### 1 Tier One: Windows Development

In the first tier, learn about the foundations of Windows development. You'll also discover many features of Visual C#, Visual Basic, and Visual C++, and the corresponding Visual Studio 2008 Express Editions. Then, write your first Visual Basic program. Finally, depending on your knowledge level, you may want to follow the Bits & Bytes series to learn computer basics.

Introduction to Windows Development

- > [Introduction to Windows as a Platform](#)

Visual Basic: Windows Development

- > [Introduction to the Visual Basic Programming Language](#)
- > [Introduction to Visual Basic 2008 Express Edition](#)
- > [Visual Basic 2008 Express Feature Tour](#)
- > [Creating Your First Visual Basic Program](#)

[This particular video](#), begins by telling you how to download and set up the Visual Basic Express Edition (the 2008 version). Downloading and installing the software is fairly trivial, anyway, so if you want to do it without any help,

you can go directly to [the download page](#), and skip the first 4 minutes of the video. The download is not huge (about 70 MB) so it shouldn't take long.

After the installation is complete, you should see Microsoft Visual Basic 2010 Express Edition on your Programs menu, and you should see a folder called [Visual Studio 2010](#) in your [My Documents](#) folder.

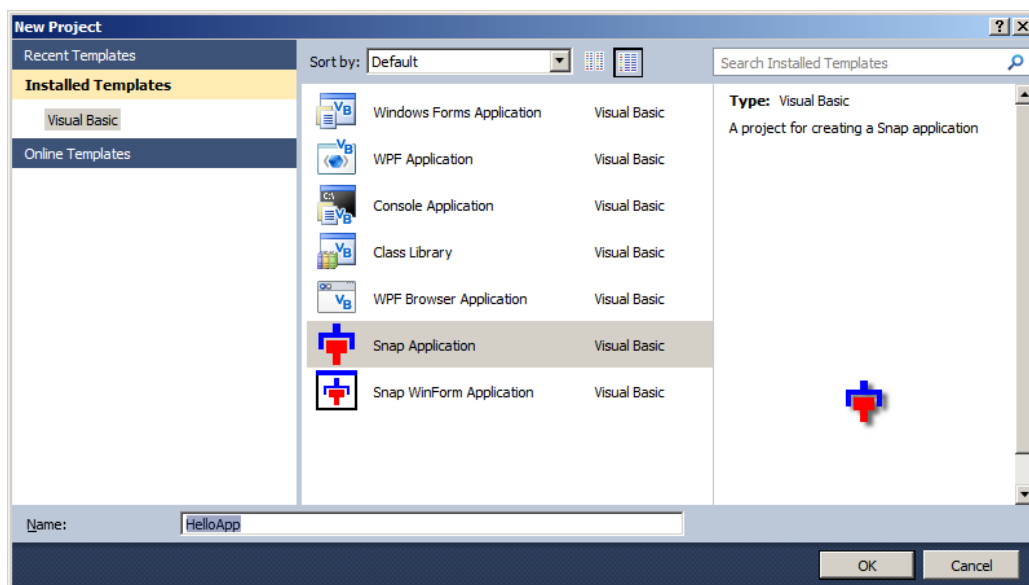
## Installing [SNAP](#) Templates

After installing Visual Studio, you should install a couple of custom templates that can be used as convenient starting points when developing [SNAP](#) programs. You will find two zip files called [SnapTemplateVB.zip](#) and [SnapWinFormTemplateVB.zip](#) in a location like `C:\Program Files\Siemens\NX 8.0\UGOPEN\snap`. Copy these two zip files into `<My Documents>\Visual Studio 2010\Templates\Project Templates\Visual Basic`.

**Note:** you should **not** extract the contents from the zip files; just **copy the zip files themselves**.

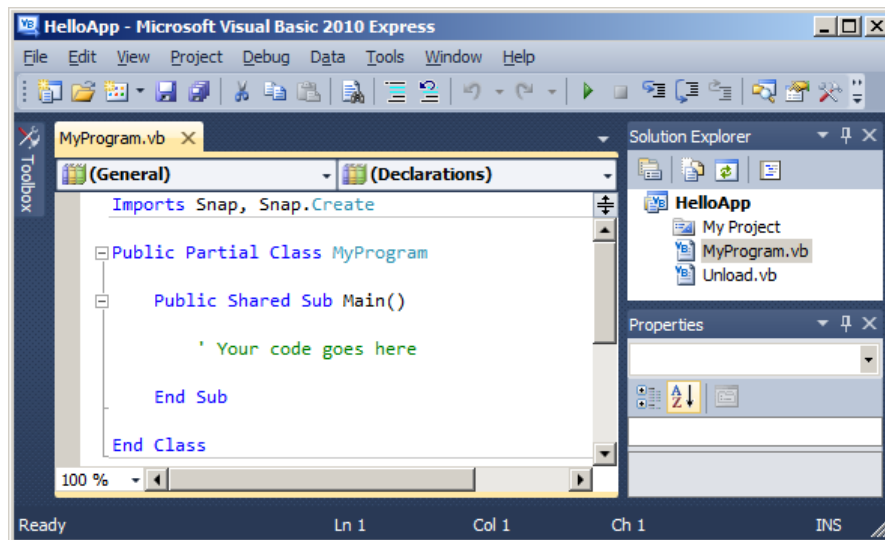
## Example 1: Hello World Again

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

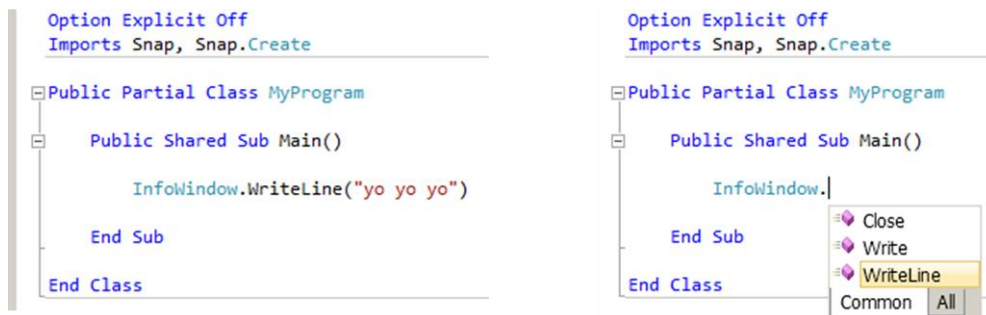


Choose the “Snap Application” template. This is a special custom template designed to serve as a convenient starting point for certain kinds of [SNAP](#) applications. Also, give your project a suitable name – something like “HelloApp” would be good.

The Snap Application template gives you a framework for a simple [SNAP](#) application, as shown here:

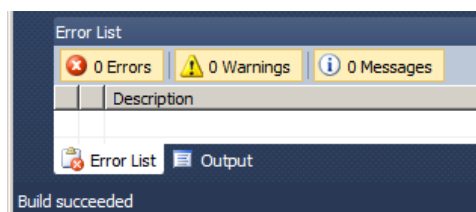


In the left-hand pane, you can see some familiar VB code which the template has placed in a file called [MyProgram.vb](#) for you. We need to make a couple of changes to this code: add `Option Explicit Off` at the top, and add a line that outputs our message to the listing window, as shown here:

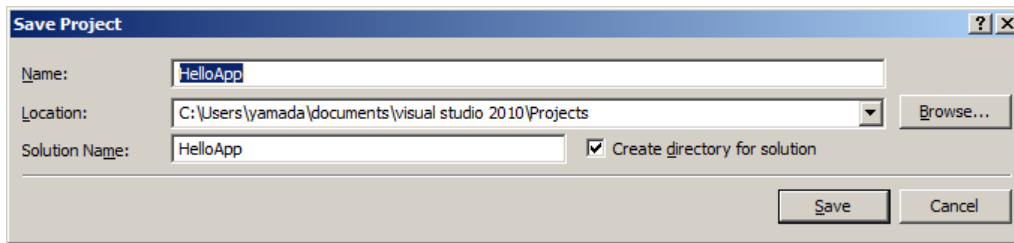


You should type the new code, rather than just copying and pasting it, because some interesting things happen as you type (as you saw in the tutorial videos, if you watched them). In fact, it's interesting to type the entire 7 lines of code. You will find that you actually only have to type 5 lines – 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. All of this is called “Intellisense” by Microsoft’s marketeers. Despite its silly name, you’ll find it very helpful as your programming activities progress. Also, notice that Visual Studio has automatically made comments green, literal text red, and language keywords blue, to help you distinguish them.

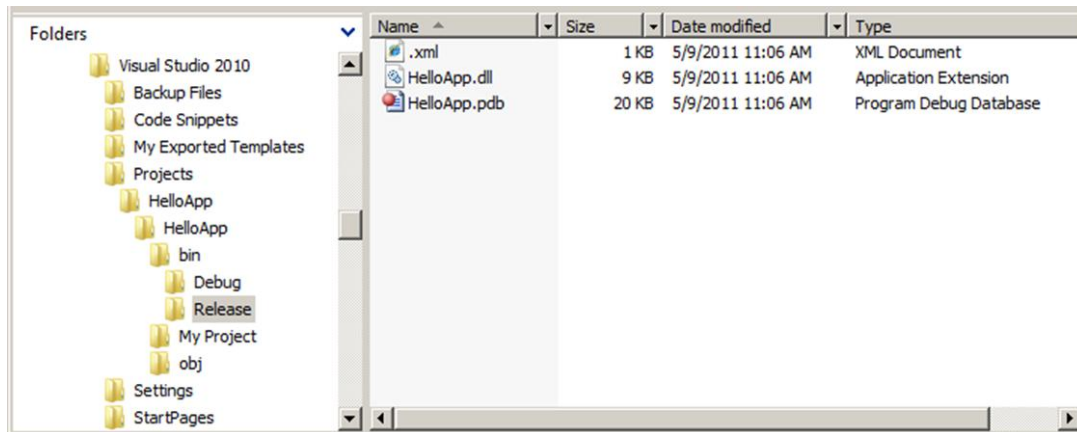
Next, you are ready to compile (or “build”) your code into an executable application. To do this, go to the Debug menu and choose Build HelloApp. After a few seconds, you should get some good news about the build succeeding down in the bottom left-hand corner of the window:



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

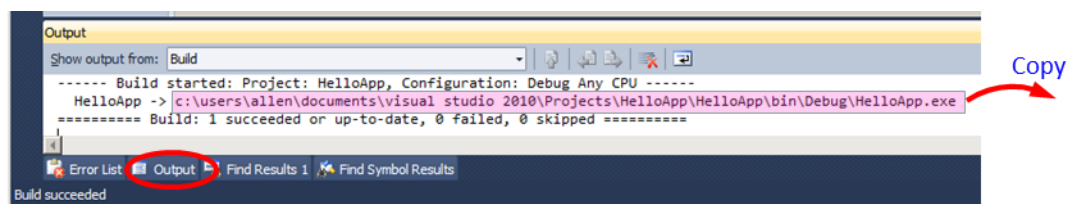


Finally, we are ready to run our new application. From within NX, choose File → Execute → NX Open (or press Ctrl+U). Your version of the NX user interface might not have the Execute option installed in the File menu, but the Ctrl+U shortcut will work anyway. A dialog will appear that allows you to find your executable. The executable will be called **HelloApp.dll**, and it will be located in the folder **<My Documents> Visual Studio 2010\Projects\HelloApp\bin\Release** 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\Debug** folder, rather than the **bin\Release** folder. If you still can't find it, it's probably because you forgot to save your project or you forgot to set the file type filter correctly.

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



You can just copy the pathname of the newly-created application (highlighted in pink above) and paste it into the “Execute” dialog within NX. You only have to do this once per NX session, because NX will remember the location for you.

## 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.

So, start up Visual Studio, and 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
```

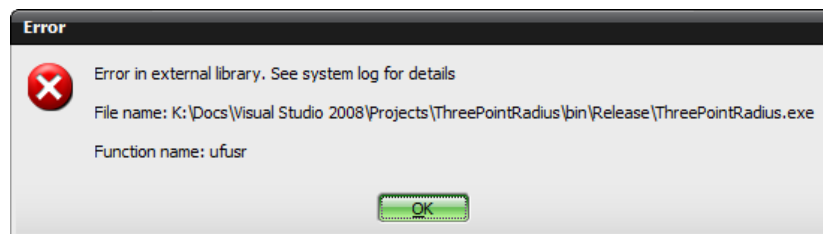
Again, you can gain some experience with Intellisense if you type this code, rather than copying and pasting it. The only thing that's new here is the function `GetPosition`, which allows you to get a point location from the user by means of the usual NX Point Subfunction.

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”. The project will still build successfully, but when you run it from within NX, you'll get an error message like this:



If you hunt around the NX System Log, you will find some more error messages, most notably these ones

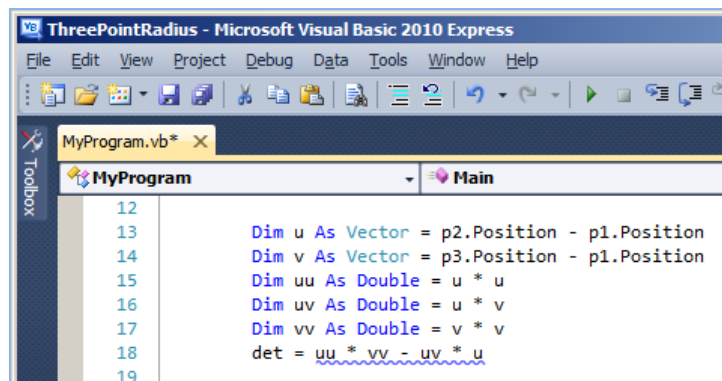
```
+++ Overload resolution failed because no Public '-' can be called with these arguments:
Argument matching parameter 'u' cannot convert from 'Double' to 'Vector'.
```

Obviously it would be much better to discover errors like this earlier, 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, when we type the next line, 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 (though sometimes in rather unhelpful terms)

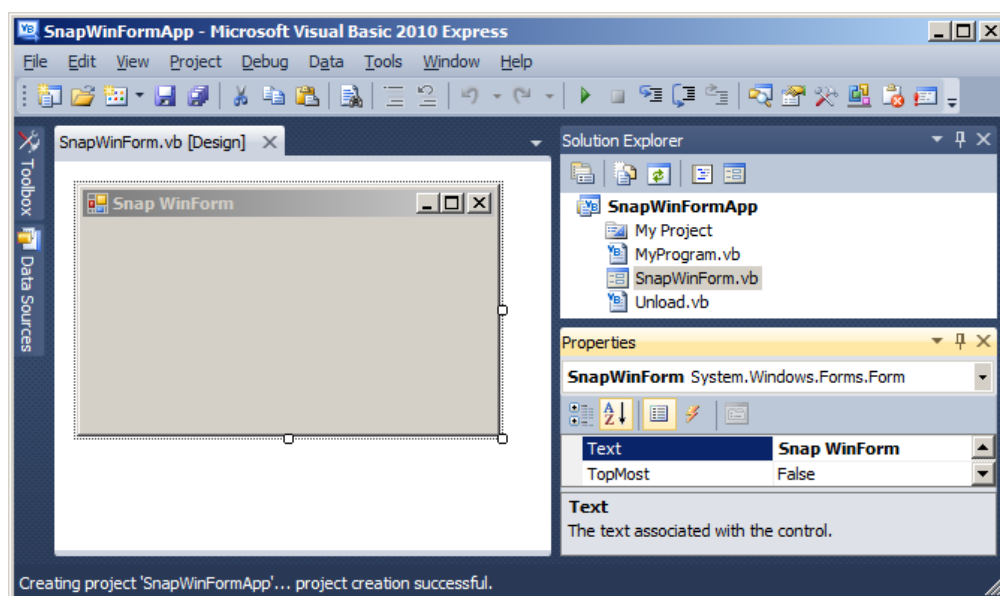


Up until now, our applications have been very simple, so there was not much justification for the extra effort of declaring variables. But, as you start to write more complex applications, you will definitely want the compiler to help you find your mistakes. And it can do this very effectively if you 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.

### Example 3: WinForms Again

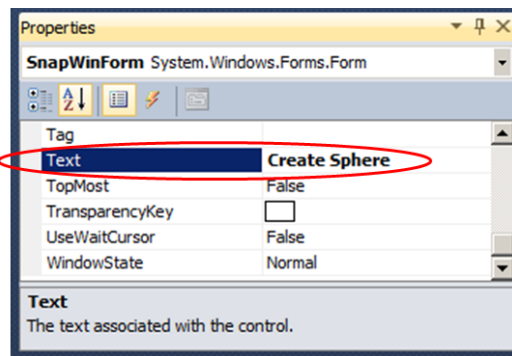
One of the nice things about Visual Studio is the set of tools it provides for designing user interface dialogs using Windows Forms (WinForms, for short). We’re going to recreate the “Create Random Spheres” dialog from the previous chapter, but it will be much easier this time, using Visual Studio, and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

Run Visual Studio Express, choose New Project from the File menu. Instead of choosing the Snap Application template, as we did before, chose the Snap WinForm Application template. Call your new project SnapWinFormApp. Your new project will look something like this:

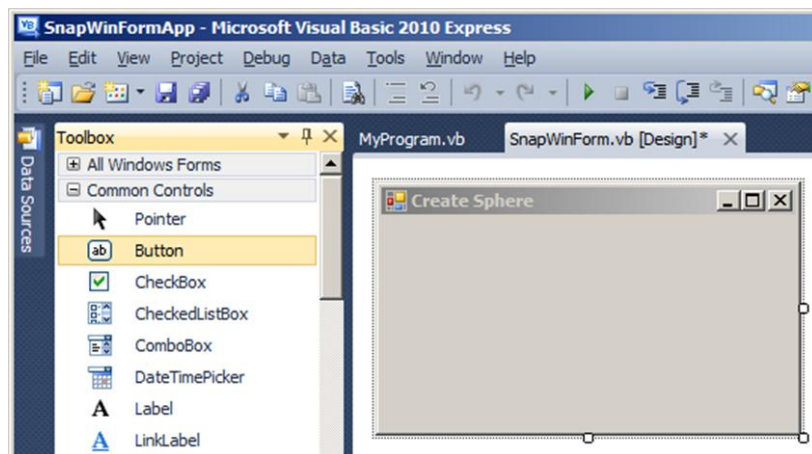


You may need to double-click on SnapWinForm.vb to see the new Windows form in the left-hand pane. In the lower right-hand pane, all the “properties” of the new WinForm are listed, along with their values. As you can see, the form has a property called “Text”, and this property currently has the value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

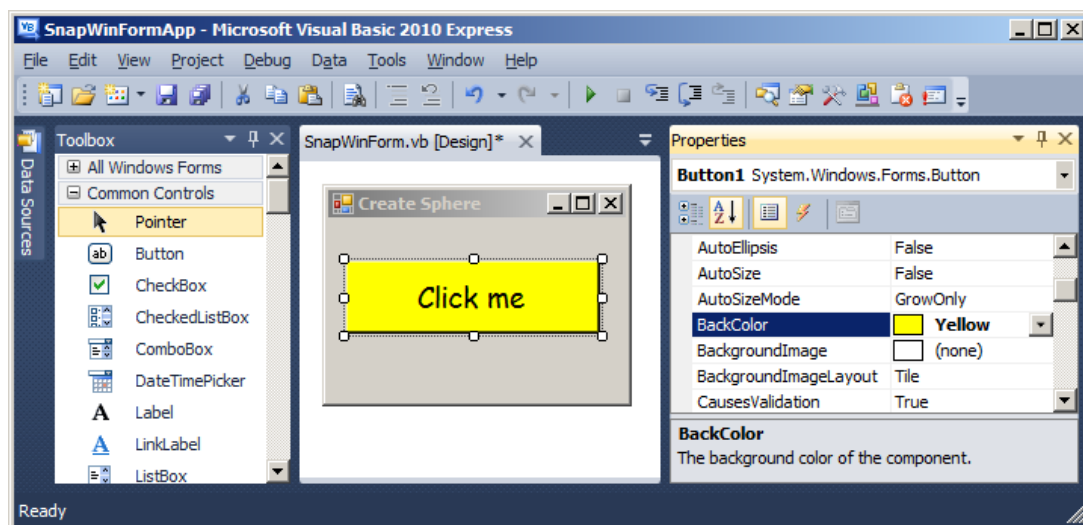




Next, as before, we need to add a button to our form. On the left-hand side of the Visual Studio window, you should see a Toolbox containing various types of user interface objects. If you don't see the Toolbox, choose it from the View menu, or press Ctrl+Alt+X.



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 perform some useful function. 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
```

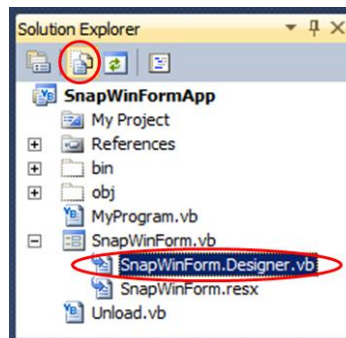
The function you see is an event handler for the button's "click" event. Currently, it doesn't do anything, but you can edit it as shown to make the click event create a sphere, or whatever else you want it to do.

```
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. If you want to see this code, click on the Show All Files button at the top of the Solution Explorer window, and double-click on the file named [SnapWinForm.Designer.vb](#).



As the comments in the file say, you are not supposed to edit this code directly – that's why it's hidden from you. To display our dialog, we need a couple of lines of code in Sub Main:

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

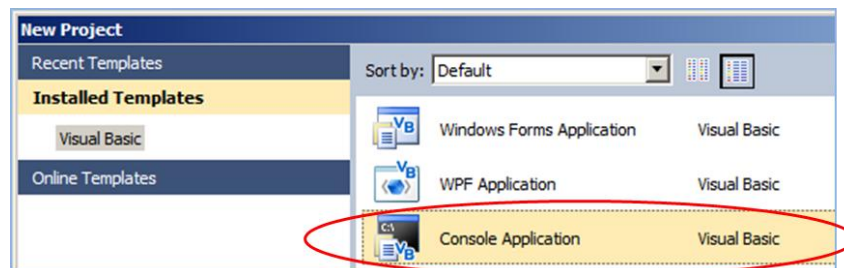
As before, we're using `form.ShowDialog` to display the dialog, so it will be "modal", which means that we can't do anything else until we close the form. There is also `myForm.Show`, which creates a non-modal form, but to use this, you have to change the `GetUnloadOption` function in the file [Unload.vb](#). 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.

This concludes our brief introduction to WinForms. If you want to learn more, there are many books and on-line tutorials that show you how to create WinForm-based user interfaces. You may have already watched [this video](#), which we recommended earlier, and there are thousands of others.

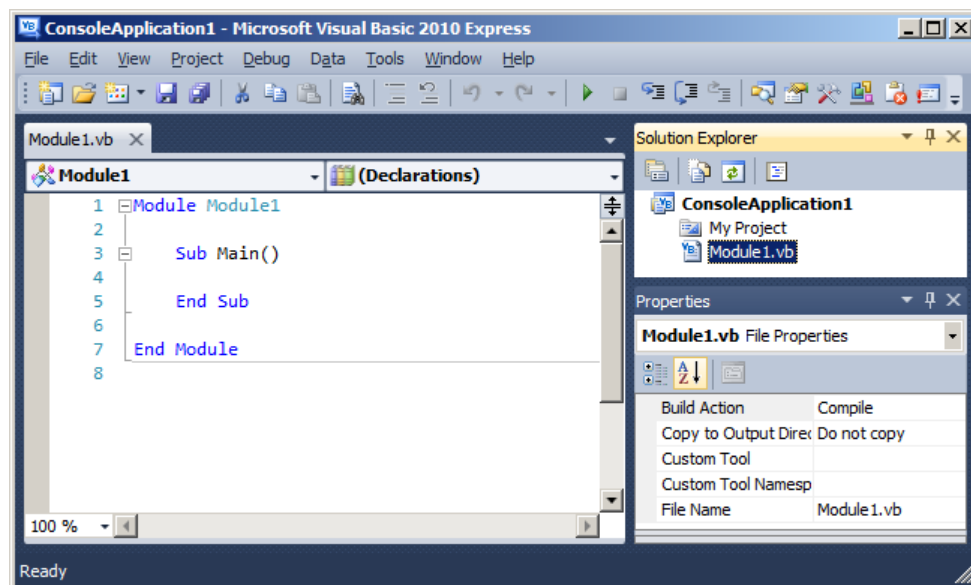
## Example 4: Hello World Yet Again (the Hard Way)

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 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.

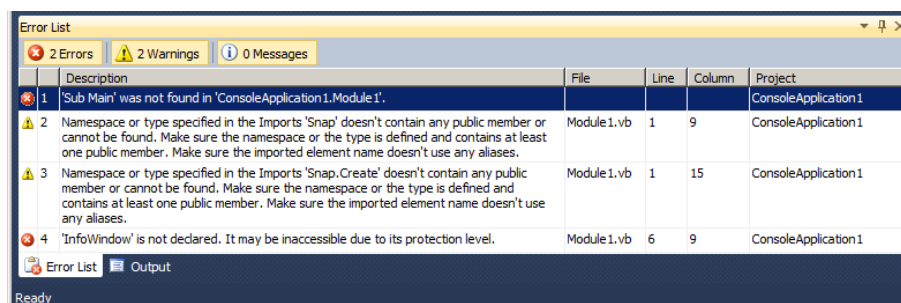
Run Visual Studio Express, and choose New Project from the File menu. You will see the available set of project templates. But, this time, instead of choosing the Snap Application template, choose the Console Application one:



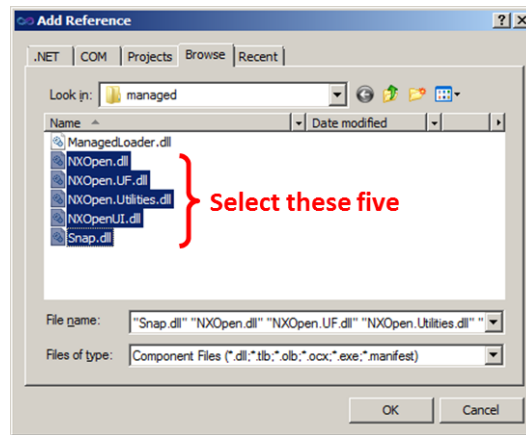
The Console Application template gives you a framework for a simple application, as shown here:



In the left-hand pane, you can see the VB code in the file `Module1.vb`. Delete this code and paste (or type) the contents of `SnapSample.vb` in its place. You will see that this causes some error and warning messages to appear in the list at the bottom of the window

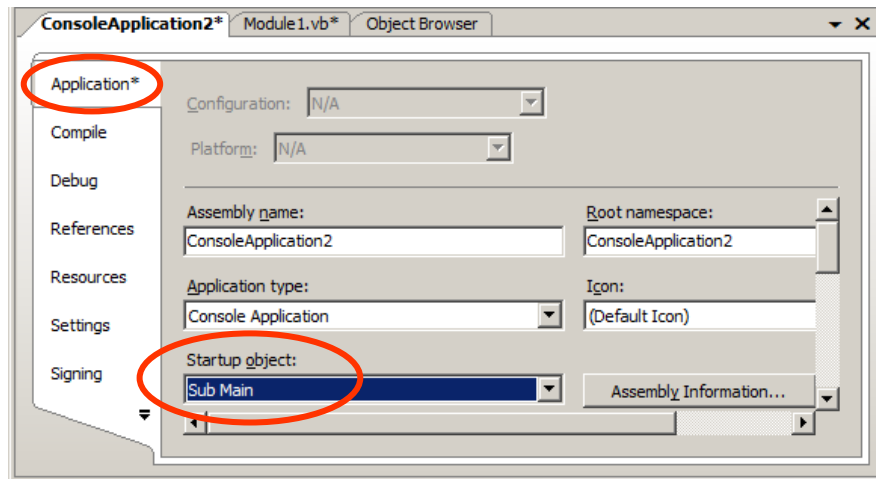


Most of the problems arise because our code is using the `SNAP` library, and this is not connected in any way to our current project. So the compiler doesn't know anything about `Snap`, `Snap.Create`, or the `InfoWindow` function. To fix this, we need to add a "reference" to the `SNAP` library. From the Project menu, choose Add Reference. In the dialog that appears, click on the Browse tab, and navigate to a folder called "managed" inside your UGII folder. The pathname will typically be something like `C:\Program Files\Siemens\NX 8.0\UGII\managed`.



You will see five DLLs – the Snap DLL and 4 NX Open 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 to have references that you don't actually need, so select all 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". The Snap Application template that we used last time already includes these references, 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 earlier.

The remaining error message says that "Sub Main was not found". As we have mentioned before, the "Main" subroutine is where your code starts executing when you run it. So, in effect, the project is complaining that it doesn't know where to start. This is easy to fix. From the Project Menu, choose Properties. In the dialog that appears, choose the Application tab, if it's not already selected. On the Startup object pull-down menu, choose Sub Main. This should cause the last of the error messages to disappear.



Now you can build and run the application, as before.

## Example 5: Toolpath Simulation

Our last example shows some calculations on surfaces, and simulates the positioning of a tool. 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

    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

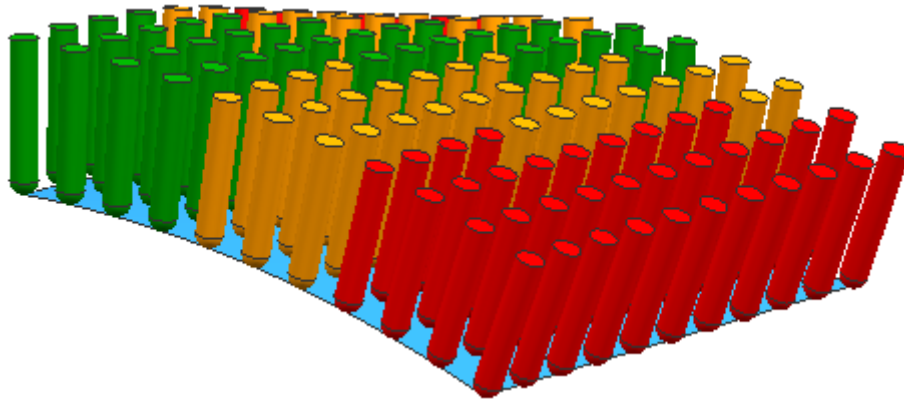
End Class
```

Note that we are using functions from the [System.Drawing](#) class, and 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. This information is then used to create and display a model of the cutting 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:





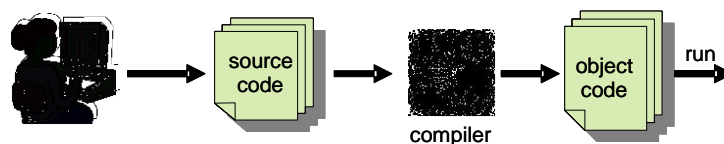
# Chapter 4: The Visual Basic Language

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, so our description 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](#).

## 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 involves lots 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 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.

Instead of placing option statements within the source code itself, you can specify compilation options in the properties of a Visual Studio project, which is often more convenient. Option statements must be placed at the beginning of a source file, and they apply only to the source file in which they appear.

### 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 System.Console

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: "
        WriteLine(message & area)          ' Write out the area value
        ReadLine()
    End Sub

    ' Function to calculate the area of a circle
    Function CircleArea(r As Double) As Double
        Dim pi As Double = 3.14
        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 table below gives more details:

Lines of code	Explanation
<code>Option Explicit On</code>	Tells the compiler that it should give you an error message if you fail to declare any variables
<code>Imports System.Console</code>	Allows you to refer to functions in the System.Console class using short names
<code>Dim radius As Double = 3.75</code>	Declares a variable of type Double, gives it the name radius, and stores the value 3.75 in it.
<code>Dim area As Double</code>	Declares another variable of type Double, and names it area
<code>area = CircleArea(radius)</code>	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.
<code>Dim message As String = "Area is: "</code>	Declares and initializes a variable of type String
<code>WriteLine(message &amp; area)</code>	Calls a function named WriteLine to write text on the system console (a DOS command prompt window). This function lives in the System.Console class, so its full name is System.Console.WriteLine. We can use the shortened name here because we wrote <code>"Imports System.Console"</code> up above.
<code>ReadLine()</code>	Calls a function named ReadLine, which, in this context, essentially just waits for the user to press a key
<code>' Function to calculate circle area</code>	This is a "comment". Comments are descriptive text to help you and other readers understand the code. They are ignored by the compiler.
<code>Function CircleArea(r As Double) As Double</code>	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.
<code>Dim pi As Double = 3.14</code>	Defines a variable called pi and gives it the value 3.14. Of course the value of $\pi$ is not really 3.14. It would be much better to use the value from the built-in constant Math.PI
<code>Dim area As Double = pi * r * r</code>	Calculates the area, and stores it in a newly declared variable called area
<code>Return area</code>	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 a bit 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

Visual Basic, like most programming languages, uses variables for storing values. Every variable has a name, and a data type (which determines the kind of data that the variable can store).

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

Data 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	4.9E-324 to 1.8 E+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 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 **For** and **Imports**).

## 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. Widening conversions never fail. A *narrowing conversion* may involve conversion from one type to another type that is not guaranteed to be able to contain it, so it 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 = 500.037
Dim roughWeight As Integer
roughWeight = CInt(weight)              ' Cast weight to an integer
roughWeight = CType(weight, Integer)    ' Different technique
```

Casts can be performed with the general `CType` function, or with more specific functions like `CInt`. The result is exactly the same.

The set of implicit conversions depends on the compilation environment and the `Option Strict` statement. If you have `Option Strict On` at the start of your file, only widening conversions may occur implicitly. With `Option Strict Off`, all 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’s slightly unusual 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 an answer of 1.0, you'll get 0.9999999999999999. Tiny errors like this usually don't matter in engineering applications. But, in cases where they do, you should 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](#) if and only if both of the operands are [True](#)
- **Or**: the result is [True](#) if either or both of the operands is [True](#)
- **Xor**: the result is [True](#) if one and only 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 value 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

## Strings

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

```
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, to avoid this deletion/recreation.

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`.

## 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 will need to 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 = Console.ReadLine()             ' Read a name from the console
    nameList.Add(name)                    ' Add it to our list
Loop Until name = ""                     ' Keep going until a blank line is entered
```

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.

## 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 `Console.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"
Console.WriteLine(greeting)
Console.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, `Console.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:

```
Dim p1 As Snap.NX.Point = Snap.Create.Point(2, 3,|
Point(x As Double, y As Double, z As Double) As Snap.NX.Point
Create a point from x, y, z coordinates
z: z-coordinate. Optional. Default = 0
```

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:

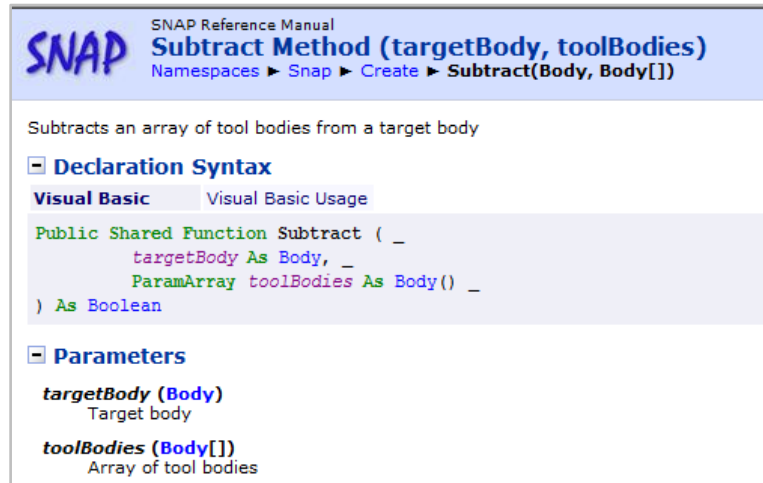
```
Dim values As Double() = { 3, 5, 7 }
mean = Average(values)
```

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 Manual](#) shows us that the specifications for the Subtract function are:



The screenshot shows the SNAP Reference Manual page for the **Subtract Method (targetBody, toolBodies)**. The breadcrumb trail is **Namespaces > Snap > Create > Subtract(Body, Body[])**. The description states: "Subtracts an array of tool bodies from a target body".

**Declaration Syntax**

**Visual Basic** Visual Basic Usage

```
Public Shared Function Subtract ( _  
    targetBody As Body, _  
    ParamArray toolBodies As Body() _  
) As Boolean
```

**Parameters**

- targetBody (Body)**  
Target body
- toolBodies (Body[])**  
Array of tool bodies

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 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 a 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) * 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.

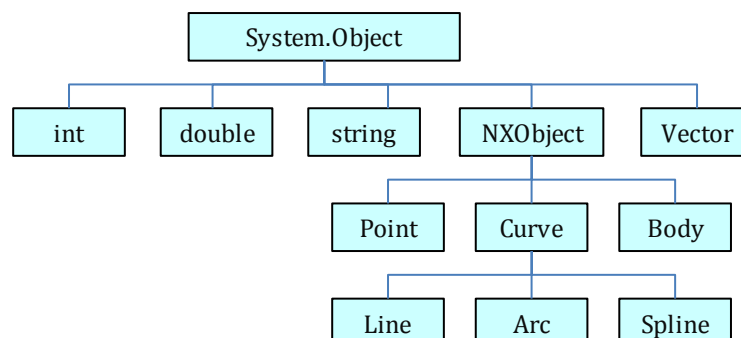
Each property has a name. To get or set the property, you use a “dot” followed by the name of the property. So, if `myCircle` is an arc, then you refer to its center as `myCircle.Center`, and its radius as `myCircle.Radius`.

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 point 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:



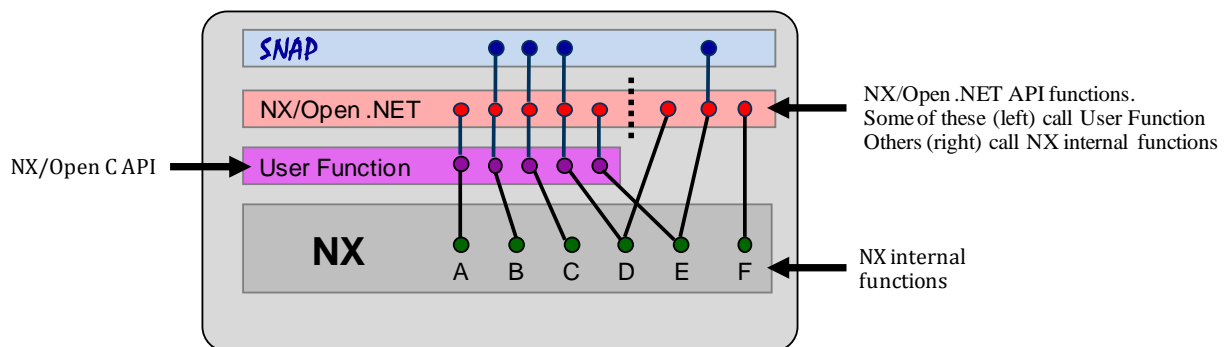
The tables in the following chapters indicate the types of objects we will be using, and their properties. You might think you will need to keep these tables handy as you are writing code, so that you know what properties are available. But, this is not the case assuming you are using a modern IDE (Integrated 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 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, but they might be interesting, and they might help 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

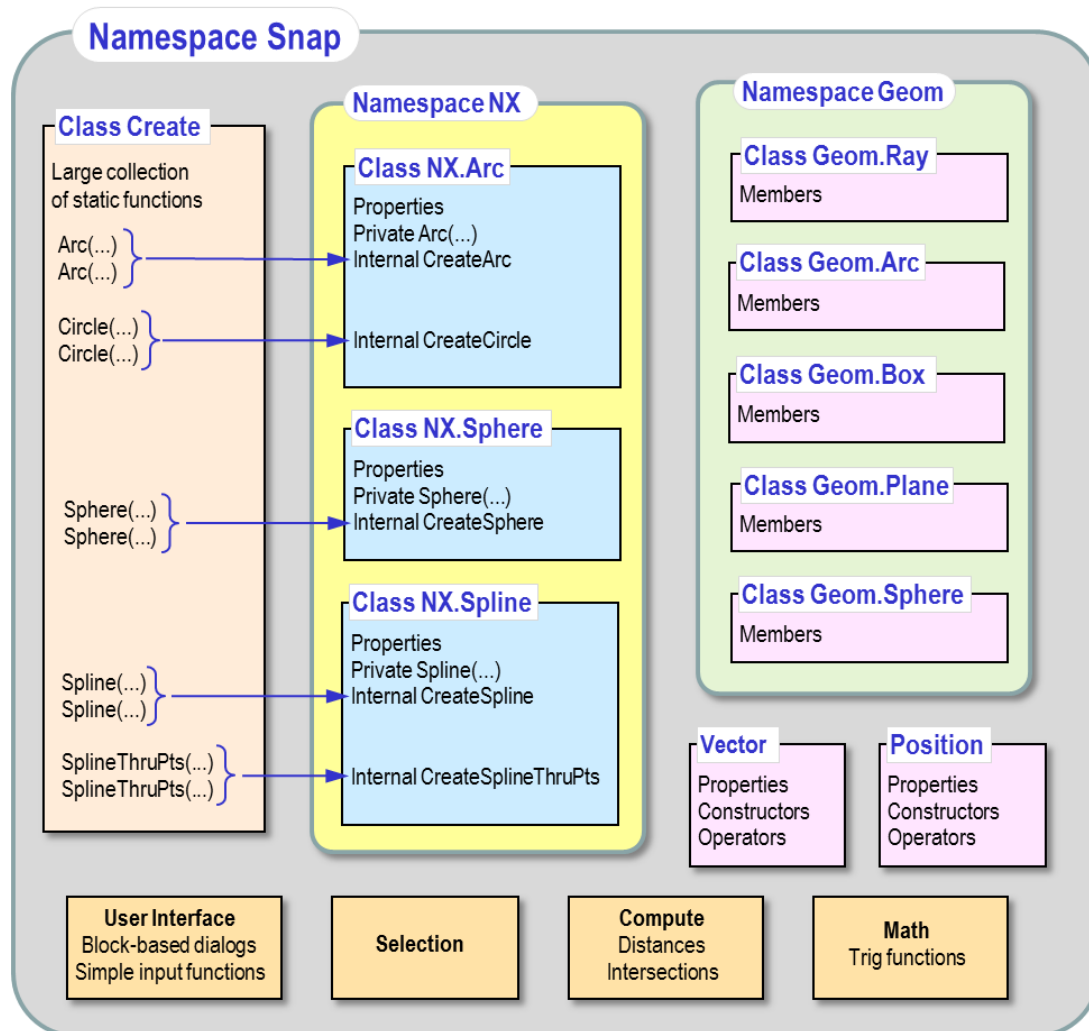
The *SNAP* functions reside in an assembly called *Snap.dll*, which you can find in the "managed" folder in your NX installation; the path name will be `C:\Program Files\Siemens\NX 8.0\UGII\managed`, or something like that. As you know from earlier examples, your code must have a "reference" to this 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 Manual is contained in a file called *Snap.chm*, which is 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 NXOpen 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.

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](#))

The [Geom](#) namespace is the home of various abstract geometric objects, which are often used as function arguments or to hold other temporary results. These objects are different from NX objects because they are transient, rather than permanent or persistent. [Geom](#) objects are never stored in NX part files (or anywhere else, typically) so they disappear as soon as your [SNAP](#) program finishes executing. 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.

## [SNAP](#) Design Principles

This section outlines some design principles that we have generally followed in the development of [SNAP](#).

### 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.

## 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 Manual, 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.

## 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

This chapter and 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 Manual](#) 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 add a few declarations. Alternatively, you can put the [Option Explicit Off](#) directive at the top of your code, so that the compiler won't complain about undeclared variables. Generally, many people consider this to be risky and counterproductive, but it's OK for these little snippets of example code.

### 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
<a href="#">Position(x As Double, y As Double, z As Double)</a>	From three rectangular coordinates.
<a href="#">Position(x As Double, y As Double)</a>	From xy-coordinates (assumes z=0).
<a href="#">Position(coords As Double[])</a>	From an array of 3 coordinates.
<a href="#">Position(p As NXOpen.Point3d)</a>	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
<a href="#">Vector(x As Double, y As Double, z As Double)</a>	From three rectangular components.
<a href="#">Vector(x As Double, y As Double)</a>	From xy- components (assumes z=0).
<a href="#">Vector(coords As Double[])</a>	From an array of three coordinates.
<a href="#">Vector(v As NXOpen.Vector3d)</a>	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
<code>Cross(u As Vector, v As Vector)</code>	Vector	Cross product (vector product) of two vectors.
<code>UnitCross(Vector u, Vector v)</code>	Vector	Unitized cross product of two vectors.
<code>Unit(u As Vector)</code>	Vector	Unitizes a given vector.
<code>Norm(u As Vector)</code>	Double	Norm (length) of a vector.
<code>Norm2(u As Vector)</code>	Double	Norm squared of a vector.
<code>Angle(u As Vector, v As Vector)</code>	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:

<code>w = u + v</code>	' Vector w is the sum of vectors u and v
<code>v = -v</code>	' Reverses the direction of the vector v
<code>w = 3.5*u - r*v/2</code>	' Multiplying and dividing by scalars
<code>u = p - q</code>	' Subtracting two Positions gives a Vector
<code>r = u*v</code>	' Dot product of vectors u and v
<code>w = Vector.Cross(u,v)</code>	' Cross product of vectors u and v
<code>w = (w*u)*u + (w*v)*v</code>	' Various products
<code>w = Vector.Cross(u, v)/2</code>	' A random pointless calculation
<code>r = Vector.Norm(u)</code>	' Calculates the length (norm) of u
<code>p = p + 3*Vector.AxisX</code>	' Moves p along the x-axis by three units
<code>p.X = p.X - 3</code>	' 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
<code>Point(x As Double, y As Double, z As Double)</code>	From x, y, z coordinates
<code>Point(x As Double, y As Double)</code>	From xy-coordinates (assumes z=0)
<code>Point(p As Position)</code>	From a position
<code>Point(coords As Double[])</code>	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 the 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
```

# Chapter 7: Curves

This chapter briefly outlines the SNAP functions for creating and editing curves (lines, arcs, and splines)

## Lines

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

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

## Arcs and Circles

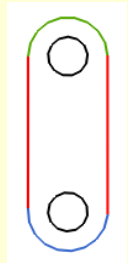
Functions for creating circular arcs are:

Function	Inputs and Creation Method
<code>Arc(center As Position, axisX As Vector, axisY As Vector, radius As Double, angle1 As Double, angle2 As Double)</code>	From center, axes, radius, angles.
<code>Arc(center As Position, matrix As Orientation, radius As Double, angle1 As Double, angle2 As Double)</code>	From center, orientation matrix, radius, angles.
<code>Arc(center As Position, radius As Double, angle1 As Double, angle2 As Double)</code>	From center point, radius, angles, parallel to the XY-plane.
<code>Arc(cx As Double, cy As Double, radius As Double, angle1 As Double, angle2 As Double)</code>	From center coordinates, radius, angles, lying in the XY-plane.
<code>Fillet(p0 As Position, pa As Position, p1 As Position, double radius)</code>	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	Returns	Inputs and Creation Method
<code>Circle(center As Position, axisX As Vector, axisY As Vector, radius As Double)</code>	<code>Arc</code>	Circle from center, axes, radius.
<code>Circle(center As Position, radius As Double)</code>	<code>Arc</code>	From center point, radius, parallel to the XY-plane.
<code>Circle(cx As Double, cy As Double, radius As Double)</code>	<code>Arc</code>	From center coordinates, radius, lying in the XY-plane.
<code>Circle(center As Position, axisZ As Vector, radius As Double)</code>	<code>Arc</code>	From center, normal, radius.
<code>Circle(p1 As Position, p2 As Position, p3 As Position)</code>	<code>Arc</code>	Through three points.

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

<pre> 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 </pre>	
--	---

The properties of arc objects are as follows:

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

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

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

## 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:  $t[0], \dots, t[n + k - 1]$

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 function. The basic functions for creating splines are:

Function	Inputs and Creation Method
<code>Spline(knots As Double[], poles As Position[], weights As Double[])</code>	Rational spline from knots, poles, and weights.
<code>Spline(knots As Double[], poles As Position[])</code>	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
<code>SplineThroughPoints(intPoints As Position[], nodes As Double[], knots As Double[])</code>	Spline passing through given points.
<code>SplineThroughPoints(intPoints As Position[], knots As Double[])</code>	Spline passing through given points.
<code>SplineThroughPoints(intPoints As Position[])</code>	Cubic spline passing through given points.

In the third 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  $S(\text{nodes}[i]) = \text{intPoints}[i]$  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 some example pseudo-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) ' Create a 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
<code>BezierCurve(ParamArray poles As Position[])</code>	Polynomial Bezier curve
<code>BezierCurve(poles As Position[], weights As Double[])</code>	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
<code>BezierCurveThroughPoints( ParamArray intPoints As Position[])</code>	Bezier curve passing through given points
<code>BezierCurveThroughPoints( intPoints As Position[], nodes As Double[])</code>	Bezier curve passing through given points

The second function allows you to specify nodes (the parameter values at which interpolation will occur). Choosing different parameter nodes will make a big difference to the shape of the curve, as the following example illustrates:

```

Dim q0 As New Position (0,0)
Dim q1 As New Position (4,1)
Dim q2 As New Position (5,0)
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
<code>Position()</code>	Poles	<code>get, set</code>	Array of 3D points representing poles (control points).
<code>Double()</code>	Weights	<code>get, set</code>	Array of weight values (these must be >0)
<code>Double()</code>	Knots	<code>get</code>	Array of knot values.
<code>Integer</code>	Degree	<code>get</code>	The degree of the spline, $m$ (equal to order - 1).
<code>Integer</code>	Order	<code>get</code>	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)
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

```

## Chapter 8: 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 sphere; 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 Manual](#).

### 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, and which 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 some 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

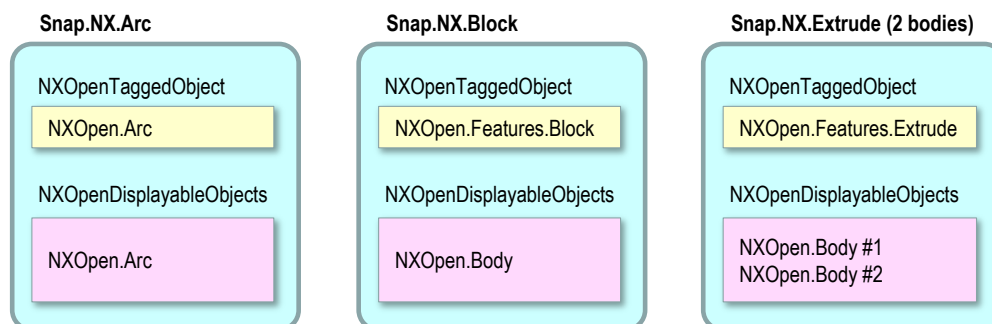
Like most other objects in [SNAP](#), the [NX.Feature](#) class is derived from [NX.NXObject](#), so it inherits all the [NX.NXObject](#) properties described in chapter 10. For our discussion here, the most important of these properties are the [NXOpenTaggedObject](#) property and the [NXOpenDisplayableObject](#) 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 [NXOpenDisplayableObject](#) 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 [NXOpenDisplayableObject](#) array has a single entry, which is an [NXOpen.Body](#).

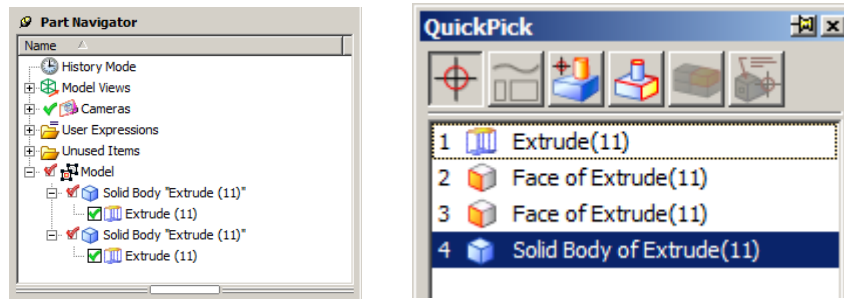
The most complex case is a feature that produces several bodies. To illustrate how this works, let’s consider a specific example. Suppose we create two circles, and use them to make a single Extrude feature. Obviously two bodies will be created. In the [NX.Extrude](#) feature, the [NXOpenTaggedObject](#) is an [NXOpen.Features.Extrude](#), and the [NXOpenDisplayableObject](#) 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:

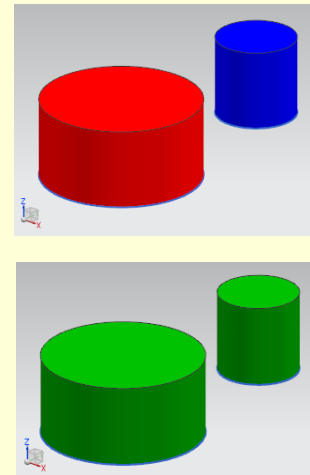
```
'Create two circles, and extrude them
Dim disk0 As NX.Arc = Circle(0, 0, 2)    ' Center at (0,0), radius = 2
Dim disk1 As NX.Arc = Circle(0, 5, 1)    ' Center at (0,5), radius = 1
Dim pegs As NX.Extrude = Extrude( {disk0, disk1}, Vector.AxisZ, 2)

'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 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 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:

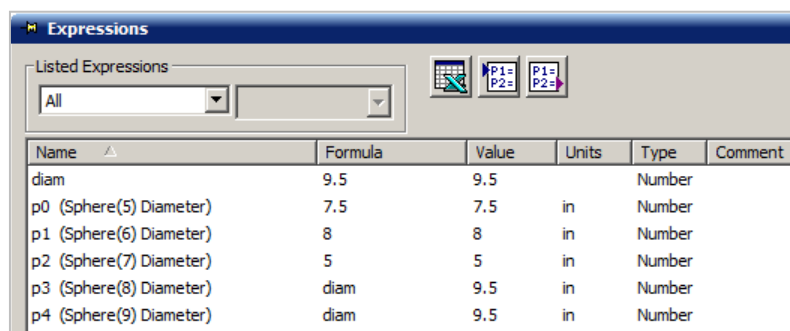
```
s1 = Sphere(center, 7.5)           ' Double 7.5 converted to Number
s2 = Sphere(center, 8)             ' Integer 8 converted to Number
s3 = Sphere(center, "5")           ' String "5" converted to Number (pointless)

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 real value 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 Integers to an array of Numbers. You have to use `numberDistances`, instead.

## Creating Features

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

Some of these will be described briefly in chapter 9; please refer to the [SNAP Reference Manual](#) for further details.

## Other Feature Functions

The `Snap.NX.Feature` class has 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 Manual](#).

## History-Free Mode

If you try to execute [SNAP](#) functions that create features while in History-Free mode, you will receive an error message. 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
```

# Chapter 9: 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 Manual; the following is just a sample of what is available:

Function	Inputs and Creation Method
<code>Block(origin As Position, xLength As Number, yLength As Number, zLength As Number)</code>	Create a block feature aligned with the absolute coordinate system axes
<code>Cylinder(basePoint As Position, direction As Vector, length As Number, diameter As Number)</code>	Create a cylinder feature
<code>Cylinder(basePoint As Position, endPoint As Position, diameter As Number)</code>	Create a cylinder feature
<code>Sphere(x As Double, y As Double, z As Double, d As Number)</code>	Create a sphere feature
<code>Sphere(center As Position, d As Number)</code>	Create a sphere feature
<code>Torus(center As Vector, axis As Vector, a As Double, b As Double)</code>	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 Manual; the following are just two typical functions:

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

The inputs to the Extrude functions have the following meanings:

<b>curves</b>	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.
<b>axis</b>	Extrusion direction. The magnitude of this vector is not significant
<b>distances/length</b>	Extents or length of the extrusion (measured from the input curves)
<b>draftAngle</b>	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:

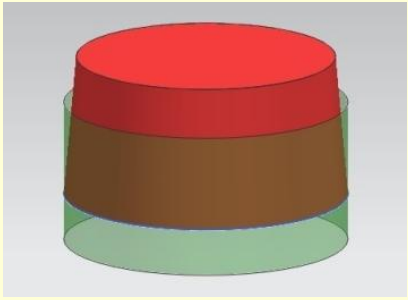
```

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 Manual; the following are just two typical functions:

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

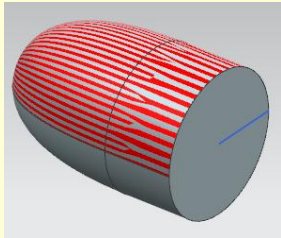
The meanings of the inputs are as follows

<b>curves</b>	An array of curves or edges to be revolved (the “section”)
<b>axisPoint</b>	Point on the axis of revolution
<b>axisVector</b>	Vector along the axis of revolution (magnitude doesn't matter)
<b>angles</b>	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, c1, c2, c3)      ' Solid
RevolveSheet( {c1, c2}, axisPoint axisVector, angles)  ' Sheet
```



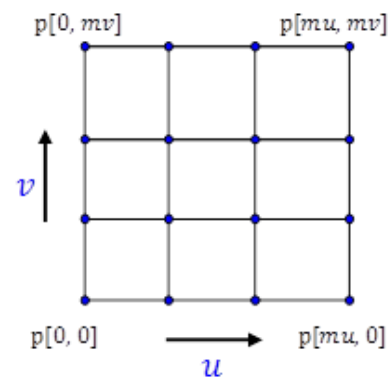
B-surfaces

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:  $u[0], \dots, u[nu + ku - 1]$
- KnotsV -- An array of  $nv + kv$  knot values:  $v[0], \dots, v[nv + kv - 1]$

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  $p[0, 0], p[0, 1], \dots, p[0, mv]$  lie along edge  $u = 0$  ( $0 < v < 1$ )
- Poles  $p[0, 0], p[1, 0], \dots, p[mu, 0]$  lie along edge  $v = 0$  ( $0 < u < 1$ )



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

- $p[0, 0] = S(0, 0)$
- $p[0, mv] = S(0, 1)$
- $p[mu, 0] = S(1, 0)$
- $p[mu, mv] = S(1, 1)$

If weights are not specified, or they are all equal, the result is polynomial surface; otherwise it is 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:

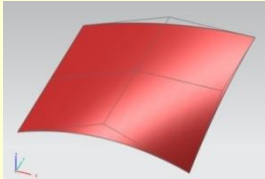
Function	Result
BezierPatch(poles As Position[,])	Polynomial Bezier patch
BezierPatch(poles As Position[,], weights As Double[,])	Rational Bezier patch



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
<a href="#">BsurfaceThroughPoints(intPoints As Position[,], nodesU As Double[], nodesV As Double[], knotsU As Double[], knotsV As Double[])</a>	Bsurface through points
<a href="#">BsurfaceThroughPoints(intPoints As Position[,], degreeU As Integer, degreeV As Integer)</a>	Bsurface through points
<a href="#">BezierPatchThroughPoints (intPoints As Position[,], nodesU As Double[], nodesV As Double[])</a>	Bezier patch through points
<a href="#">BezierPatchThroughPoints(intPoints As Position[,])</a>	Bezier patch through points

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

<pre> 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) </pre>	
---	---

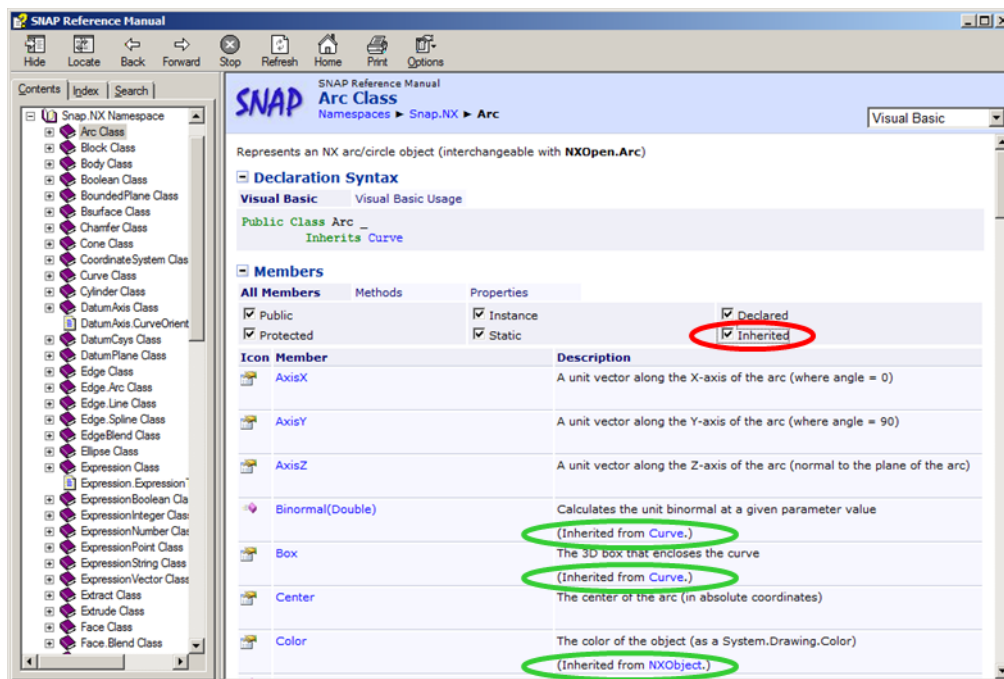
In addition to the simple B-surface creation functions listed above, there are several other **SNAP** functions that create b-surface geometry; the [ThroughCurves](#) and [ThroughCurveMesh](#) features are the most important examples, and, of course, you can find these described in the [SNAP Reference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

# Chapter 10: 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 Manual](#), 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 Manual](#), you can control whether or not inherited members are displayed by clicking in the check-box circled in red below:



The green highlighting shows two members that [NX.Arc](#) inherits from [NX.Curve](#), and one that it inherits from [NX.NXObject](#). Both 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
<a href="#">Snap.NX.ObjectTypes.Type</a>	<a href="#">ObjectType</a>	<a href="#">get</a>	The object's type
<a href="#">Snap.NX.ObjectTypes.SubType</a>	<a href="#">ObjectSubType</a>	<a href="#">get</a>	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 <code>System.Drawing.Color</code> )
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 (thin, medium, or thick)
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.ObjectFont.Dashed
disk.LineWidth = Globals.ObjectWidth.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, as we explained in chapter 8.

## 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 Manual](#):

Function	Description
<a href="#">DeleteAttributes(AttributeType)</a>	Deletes all attributes of a given type
<a href="#">GetAttributeInfo()</a>	Gets an array of <a href="#">AttributeInformation</a> structures
<a href="#">GetAttributeStrings()</a>	Get the object’s attributes as strings
<a href="#">GetIntegerAttribute(String)</a>	Returns the value of an attribute of type “Integer”
<a href="#">GetStringAttribute(String)</a>	Returns the value of an attribute of type “String”
<a href="#">SetBooleanAttribute(String, Boolean)</a>	Creates and/or sets the value of an attribute of type “Boolean”
<a href="#">SetDateTimeAttribute(String, DateTime)</a>	Creates and/or sets the value of an attribute of type “Time”
<a href="#">SetRealAttribute(String, Double)</a>	Creates and/or sets the value of an attribute of type “Real”
<a href="#">Name</a>	The name of the object (aka “custom name”, sometimes)
<a href="#">NameLocation</a>	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 [NXOpen](#) objects, as follows:

Data Type	Property	Access	Description
<a href="#">NXOpen.TaggedObject</a>	<a href="#">NXOpenTaggedObject</a>	<a href="#">get</a>	The enclosed <a href="#">NXOpen.TaggedObject</a>
<a href="#">NXOpen.Tag</a>	<a href="#">NXOpenTag</a>	<a href="#">get</a>	The tag of the <a href="#">NXOpen.TaggedObject</a>
<a href="#">NXOpen.DisplayableObject[]</a>	<a href="#">NXOpenDisplayableObjects</a>	<a href="#">get</a>	Array of associated displayable objects
<a href="#">NXOpen.DisplayableObject</a>	<a href="#">NXOpenDisplayableObject</a>	<a href="#">get</a>	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 almost always contains several points.

The following code illustrates this:

```
'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 pegBody0 As NX.Body = CType(pegs.NXOpenDisplayableObjects(0), NX.Body)
Dim pegBody1 As NX.Body = CType(pegs.NXOpenDisplayableObjects(1), NX.Body)

'Change the colors of these two bodies
pegBody0.Color = System.Drawing.Color.Red
pegBody1.Color = System.Drawing.Color.Blue
```

## 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. Given a point on a curve (defined by a parameter value  $u$ ), we can ask for a variety of different values at this point, 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)

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  $0 < u < 1$ . 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  $u = 0.5 * (\text{MinU} + \text{MaxU})$ . Here are some examples to illustrate the usage:

<code>myCircle = Circle(3, 5, 2)</code>	' Circle of radius 2 with center at (3,5,0)
<code>point90 = myCircle.Position(90)</code>	' Point at 90 degrees - will be (3,7,0)
<code>tang90 = myCircle.Tangent(90)</code>	' Tangent at 90 degrees - will be (-1,0,0)
<code>deriv0 = myCircle.Derivative(0)</code>	' First derivative at start - will be (0, 0.0174533, 0)
<code>curv = myCircle.Curvature(0.83)</code>	' 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  $\pi/180$ ). 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 – you can ask what body the edge lies on, and what faces lie on either side of it.

## Face Properties

Like edges, faces have both evaluator functions and topological 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( <i>u</i> As Double, <i>v</i> As Double)	Point on surface
Vector	Normal( <i>u</i> As Double, <i>v</i> As Double)	Unit surface normal
Vector	DerivDu( <i>u</i> As Double, <i>v</i> As Double)	Partial derivative wrt <i>u</i>
Vector	DerivDv( <i>u</i> As Double, <i>v</i> As Double)	Partial derivative wrt <i>v</i>
Double[]	Parameters( <i>p</i> As Position)	Parameters at position <i>p</i> (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 edges, faces have “topological” properties that describe their relationship to other objects in their body. If *myFace* is a *Snap.NX.Face* object, then *myFace.Body* gives the body that the face lies on, and *myFace.Edges* gives its array of edges.

### 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)

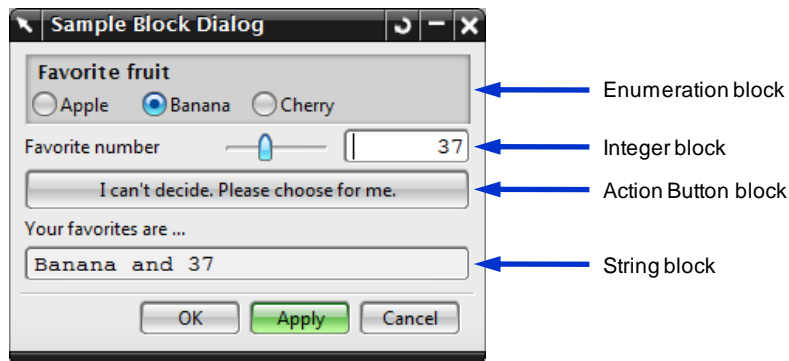
' Get the Geom (a Geom.Cylinder object)
Dim cylinderSurface As Geom.Face.Cylinder = cylinderFace.Geometry

' 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 11: NX Block-Based Dialogs

Since NX5, new dialogs in NX have been built from a common collection of “blocks”. So, for example, this dialog consists of four blocks, whose types are indicated by the labels to the right



Each type of block has a specific 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
- 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 NX Open and [SNAP](#), so your add-on applications can look and behave like the rest of NX. This chapter tells you how to do this.

## 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 for creating WinForm-based dialogs. 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 Winforms 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 much easier. Finally, Block-Based dialogs are portable across computer platforms, and support Journaling. 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.

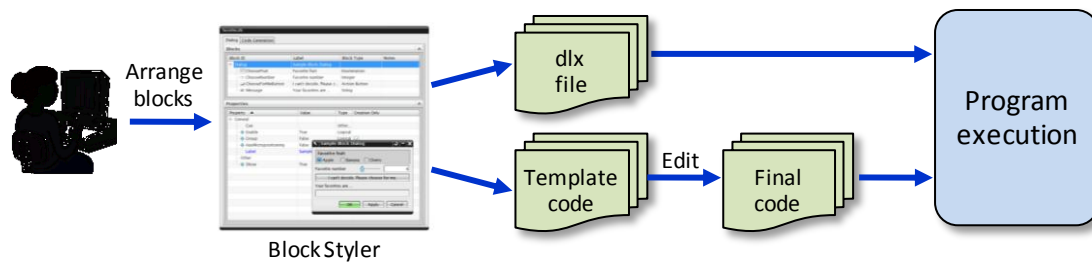
## The Overall Process

The overall process of developing a Block-Based dialog is as follows:

- You use Block Styler to choose the blocks you want, and arrange them on your dialog
- Block 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.

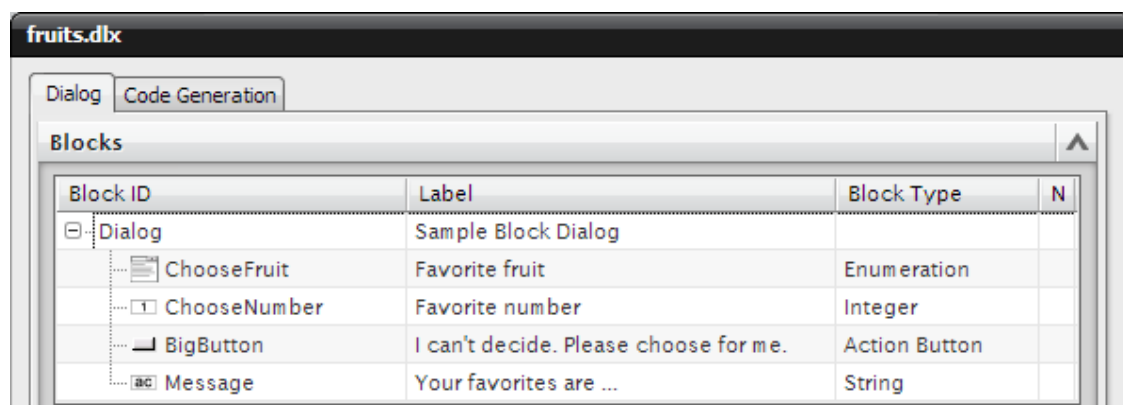




## Using Block Styler

Instructions for using Block 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.

Access Block Styler via Start → All Applications → Block Styler, and use it make the dialog shown earlier. It has four blocks, as shown here:



You can leave all the properties set to their original default values except for the changes shown below:

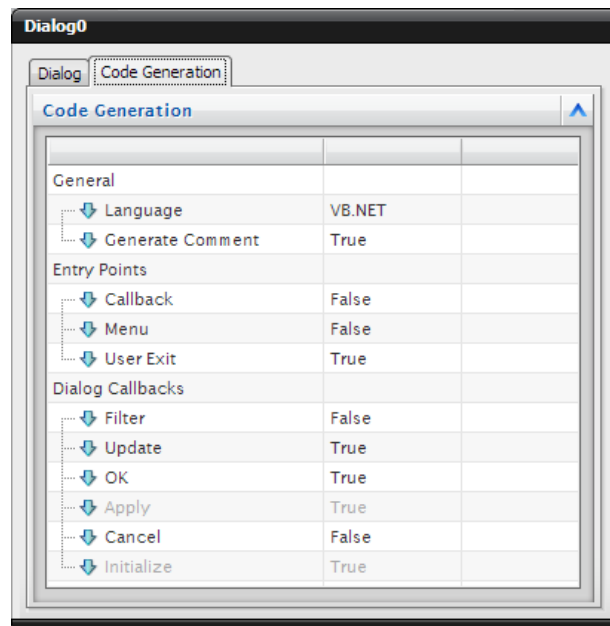
Block	Property	Value
ChooseFruit	Block ID	ChooseFruit
	Label	Favorite fruit
	PresentationStyle	Radio Box
	Layout	Horizontal
	Value	Apple Banana Cherry

Block	Property	Value
ChooseNumber	Block ID	ChooseNumber
	Label	Favorite number
	PresentationStyle	ScaleKeyin
	ScaleLimits	False
	ShowScaleValue	False
	MinimumValue	1
	MaximumValue	99

Block	Property	Value
BigButton	Block ID	BigButton
	Label	I can't decide. Please choose for me

Block	Property	Value
Message	Block ID	Message
	Label	Your favorites are ...
	PresentationStyle	Wide
	ReadOnlyString	True

Once you have arranged the blocks the way you want on your dialog, choose the Code Generation tab, and set the options as you see here



Finally, choose File → Save, which will generate two files called [FavoritePicker.vb](#) and [FavoritePicker.dlx](#).

## Template Code

When you save a dialog in Block Styler, a Visual Basic file is 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.

The contents of the VB file will depend on the options you chose in BlockStyler. 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 remove the error checking, of course.

At the top of the VB file, you will see code like this...

```
Public Class FavoritePicker

    Inherits Snap.UI.BlockDialog

    Public Shared theFavoritePicker As FavoritePicker
    Private ChooseFruit           As Snap.Block.UI.Enumeration ' Block type: Enumeration
    Private ChooseNumber          As Snap.Block.UI.Integer      ' Block type: Integer
    Private BigButton             As Snap.Block.UI.Button       ' Block type: Button
    Private Message               As Snap.Block.UI.String        ' Block type: String

    and so on ...
```

As you can see, we are defining a new class called “FavoritePicker” to represent instances of our dialog. The last four lines declare variables to hold the four blocks that make up a “FavoritePicker” dialog.

Then, further down, you will see a constructor:

```
Public Sub New(dialogPath As String)
    MyBase.New(dialogPath)
    theDialog.AddApplyHandler(AddressOf ApplyCallback)
    theDialog.AddUpdateHandler(AddressOf UpdateCallback)
    theDialog.AddCancelHandler(AddressOf CancelCallback)
    theDialog.AddInitializeHandler(AddressOf InitializeCallback)
End Sub
```

Most of this code is adding “handlers” to our dialog, as we requested when we saved the dialog from Block Styler. These handlers will allow us to write code that responds to “events” in the dialog. For example, when the user clicks the “Apply” button in the dialog, a function called “ApplyCallback” 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 we examine the “Main” routine:

```
Public Shared Sub Main()
    randomFruit = new System.Random()
    randomNumber = new System.Random()
    theFavoritePicker = New FavoritePicker(@"C:\dialogs\FavoritePicker.dlx")
    theFavoritePicker.Show()
    theFavoritePicker.Dispose()
End Sub
```

Here we define two random number generators that we will use to implement the behaviour of the dialog, as you will see later. Then we create a new “FavoritePicker” dialog, and display it using the “Show” function.

Also, note how we have placed a “@” before the pathname of the dlx file. This causes the compiler to interpret the string completely literally, and prevents any possible confusion that might be caused by colons and \ characters.

The most interesting part is the code you write yourself, in the “callback” functions. Usually, the “update” callback has most of the important code. In our case:

```
Public Function UpdateCallback(ByVal block As NXOpen.BlockStyler.UIBlock) As Integer
    If block Is BigButton.NXOpenBlock Then
        ChooseFruit.SelectedIndex = randomFruit.Next(3)      ' random index in range 0 to 2
        ChooseNumber.Value = randomNumber.Next(99) + 1      ' random value in range 1 to 99
        ApplyCallback()
    End If
    Return 0
End Function
```

The update callback is called whenever the user changes anything in some block in the dialog. It receives a UIBlock value identifying the block that was used. So, then, the code typically has several “if” clauses that do different things depending on which block the user changed. Our case is very simple -- we only have one “if” clause because the **BigButton** button is the only block that we want to react to immediately. As you can see, if the user tells us to choose a fruit and a number for him (by clicking on the **BigButton** button), we just choose them randomly.

The other interesting callback is the “Apply” one:

```
Public Function ApplyCallback() As Integer
    Dim fruit As String = ChooseFruit.SelectedItem
    Dim number As String = ChooseNumber.Value.ToString()
    Message.Value = fruit & " and " & number
    Return 0
End Function
```

It retrieves the currently-chosen values from the ChooseFruit and ChooseNumber blocks, combines them into a text string, and outputs them via the Message block.

## Callback Details

The use of the Update callback and the Apply callback were illustrated above. There are actually several more callbacks that you might find useful. 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 functions:

Callback function name	When NX calls this function
<a href="#">UpdateCallback</a>	When the user changes something in the dialog
<a href="#">OkCallback</a>	When the user clicks the OK button
<a href="#">ApplyCallback</a>	When the user clicks the Apply button
<a href="#">CancelCallback</a>	When the user clicks the Cancel button
<a href="#">InitializeCallback</a>	Just before values are loaded from “dialog memory” (see below)
<a href="#">DialogShownCallback</a>	Just before the dialog is displayed
<a href="#">FocusNotifyCallback</a>	When focus is shifted to a block that cannot receive keyboard entry
<a href="#">KeyboardFocusNotifyCallback</a>	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

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. 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 corresponding dlx file are used, then
- (2) Values and options specified in the Initialize callback are applied, then
- (3) Values from dialog memory are applied, and finally
- (4) Values and options specified in the DialogShown callback are applied

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 can not. 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 Styler. Also, the NXOpen samples folder contains eight examples of Block Styler dialogs. Its location is typically [C:\Program Files\Siemens\NX 8.0\UGOPEN\SampleNXOpenApplications\.NET\BlockStyler](#)

## Chapter 12: Simple Input and Output

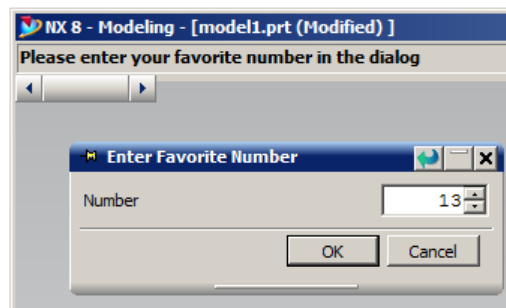
To communicate with the user of your programs, you can either use WinForms, as we saw in Chapter 3, or NX Block-Based dialogs, which were described in Chapter 11. Both techniques allow you to build very rich user interfaces, but sometimes this is a lot more than you need. Sometimes, you just want the 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 sort of 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)
```

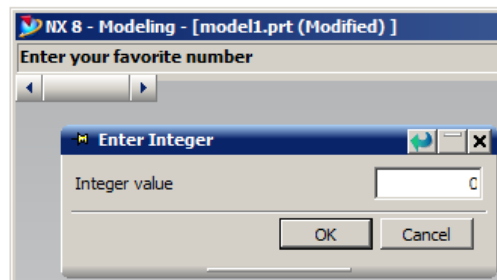
This code produces the following display in NX: (### fix picture)



As you can see from the documentation, most 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:

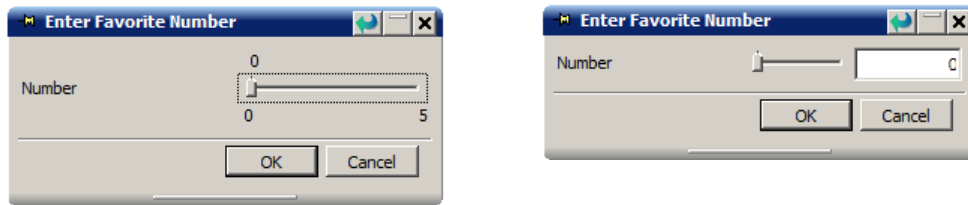
```
Dim favorite As Integer = GetInteger("Enter your favorite number")
```

This produces the display



Which uses the Keyin presentation style.

The Scale and ScaleKeyIn options allow the use of sliders to enter numbers:



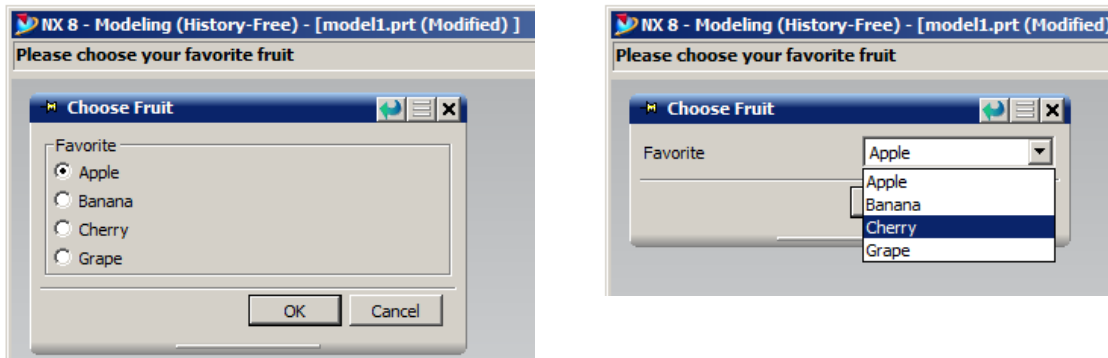
The [GetDouble](#) function provides very similar capabilities, but for double (floating point) numbers. The [GetString](#) function has a Wide option that allows a longer line of text to be displayed. All of the dialogs described above are examples of the usual kind of NX “block-based” dialog. They 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 As = Snap.UI.Block.EnumPresentationStyle.RadioButton
Dim choice = GetChoice(fruits, cue, title, label)
```

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:

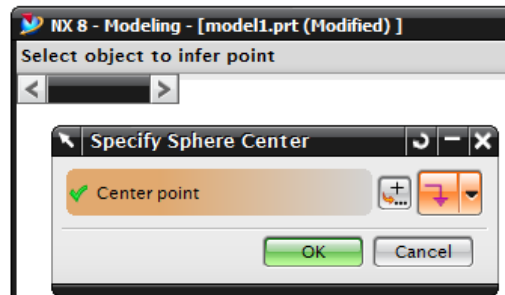
```
Dim choice As Integer = GetChoice( { "Apple", "Banana", "Cherry" } )
```

## 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)
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.

## The Role of DLX Files

As mentioned above, the simple dialogs created by the [Snap.UI.Input](#) functions are the usual NX block-based dialogs, albeit very simple ones. You may recall from Chapter 11 that block-based dialogs are constructed using “DLX” files, but we have not mentioned those here because you typically do not need to worry about them. In fact, the DLX files used by our simple dialogs are accessible, and you can modify them, if you want to. The typical location is something like `C:\Program Files\Siemens\NX 8.0\UGOPEN\snap\dialog`. The file `double.dlx` is used by the [GetDouble](#) function, and so on. So, if you modify the file `double.dlx` (using NX Block Styler), you can influence the design of the dialog produced by the [GetDouble](#) function.

## 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
<a href="#">Write(output As Integer)</a>	Write an integer to the Information window.
<a href="#">Write(output As Double)</a>	Write a double to the Information window.
<a href="#">Write(output As String)</a>	Write a text string to the Information window.
<a href="#">Write(output As Position)</a>	Write a Position to the Information window.
<a href="#">Write(output As Vector)</a>	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

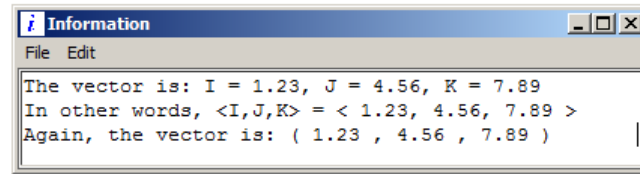
s1 = String.Format("The vector is: I = {0}, J = {1}, K = {2}", v.X, v.Y, v.Z)
s2 = String.Format("In other words, <I,J,K> = < {0}, {1}, {2} >", v.X, v.Y, v.Z)
s3 = "Again, the vector is: " & v.ToString()

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 string manipulation techniques in Visual Basic in many books and on-line tutorials.

## Windows Output

If you need more than just monospaced black English text, you may want an output medium that's richer than the NX Info window. The standard Windows utilities give you some other options. The simplest approach is to use the [System.Windows.Forms.MessageBox](#) class. 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



More general Windows forms give you a vast range of output possibilities, of course.

# Chapter 13: 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. This chapter describes the available tools and how to use them.

## The Basic Idea

In [SNAP](#), selection of NX objects is supported by the [Snap.UI.Selection](#) class. The general process of selection is as follows:

- 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, NX.ObjectTypes.Type.Line)

dialog.Title = "Selection Demo"
dialog.Scope = NXOpen.Selection.SelectionScope.AnyInAssembly
dialog.IncludeFeatures = False

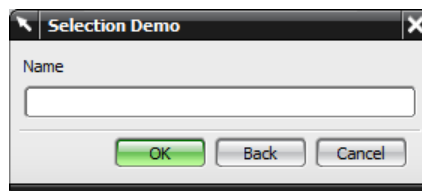
Dim result As Selection.Result = dialog.Show()

If result.Response <> NXOpen.Selection.Response.Cancel Then
    result.Object.IsHidden = true
End If
```

As you can see, we created a [Selection.Dialog](#) object, modified some of its properties, and then displayed it to allow the user to select objects and provide a response. As a result of this, some useful information (like the list of selected objects, for example) gets returned in a [Selection.Result](#) object. The [Snap.Selection](#) class provides the [Selection.Dialog](#) object, plus some static functions for constructing various kinds of selection dialogs.

## A Bit More Detail

We saw the basic selection process illustrated in the code above. When that 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).

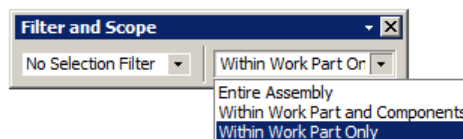
Following are the details of the variables that affect the behavior of the dialog:

Argument	Type	Meaning
<code>cue</code>	<code>String</code>	The message displayed in the Cue line
<code>type</code>	<code>NX.ObjectTypes.Type.Line</code>	The type of object we want to select
<code>dialog.Title</code>	<code>String</code>	The title displayed at the top of the dialog
<code>dialog.Scope</code>	<code>NXOpen.Selection.SelectionScope</code>	The scope of the selection, explained below
<code>dialog.includeFeatures</code>	<code>Boolean</code>	If true, selecting features is allowed
<code>dialog.Result</code>	<code>Selection.Result</code>	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 `type` argument indicates 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 response 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 the following:

```
Dim cue = "Please select a line to be hidden"
Dim dialog = Selection.SelectObject(cue, 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 static function called `SelectObject` to create a `Selection.Dialog`. A couple of controlling variables are input to this function, since they have no reasonable default values. Other variables are available as properties of the dialog that can be modified 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.

## 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 the selection function, as we did in the example 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", Type.DatumPlane)
result = dialog.Show()

' Select a body (solid or sheet)
cue = "Please select a body (solid or sheet)"
dialog = Selection.SelectObject(cue, 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 `SelectObject` function, as follows:

```
'To select a point or a line
cue = "Please select a point or a line"
Dim types As NX.ObjectTypes.Type() = { Type.Point, Type.Line }
dialog = Selection.SelectObject(cue, types)
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 most of the `SelectObject` functions.

Here are two examples: first, selecting an ellipse:

```
' To select an ellipse
Dim type = NX.ObjectTypes.Type.Conic
Dim subType = NX.ObjectTypes.SubType.ConicEllipse
dialog = Selection.SelectObject(cue, type, subtype)
result = dialog.Show()
```

and then selecting a note:

```
' To select a note
Dim noteType = Type.DraftingEntity
Dim noteSubtype = SubType.DraftingEntityNote
dialog = Selection.SelectObject(cue, 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
Dim type = NX.ObjectTypes.Type.Face
Dim subtype = NX.ObjectTypes.SubType.FacePlane
dialog = Selection.SelectObject("Select a face", 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](#) selection 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 Select 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 Select 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
Dim combos As TypeCombo() = { circularEdgeCombo, cylinderFaceCombo }
dialog = Selection.SelectObject("Select hole", combos)
```

## Selecting Curves, Edges, and Faces

Quite often, we will want to allow the user to select a curve of any type (line, arc, conic, or spline). Of course, you could do this using the techniques described above. We would just write:

```
Dim types As NX.ObjectType() = { Type.Line, Type.Arc, Type.Conic, Type.Spline }
dialog = Selection.SelectObject("Select curve", types)
```

We have reduced the typing by using [Imports Snap.NX.ObjectTypes](#), but it’s still quite a lot of work. This is such a common situation that [SNAP](#) provides special SelectCurve and SelectCurves functions that make things easier.

There is a similar situation when selecting faces. Suppose you wanted the user to be able to select a cylindrical or conical face. You could certainly define a TypeCombo array with two elements to do this. But, to make things easier, there are special functions called SelectFace and SelectFaces that allow you to pass an array of face subtypes directly, like this:

```
' To select a cylindrical or conical face
Dim faceSubTypes As NX.ObjectSubType() = { Type.FaceCylinder, Type.FaceCone }
dialog = Selection.SelectFace(cue, 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 SelectICurve (an “ICurve” is either a curve or an edge). For example, the code

```
dialog = Selection.SelectICurve(cue, Type.Circle)
```

will allow the user to select either a circle or a circular edge. The type specified in the last argument controls both curve types and edge types, all in one.

## 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 selectedLine As NX.Line = CType(selectionResult.Object, NX.Line)
            Dim result As Snap.Compute.DistanceResult
            result = Compute.ClosestPoints(selectedLine, selectionResult.CursorRay)
            Point(result.Point1)
        End If

    End Sub

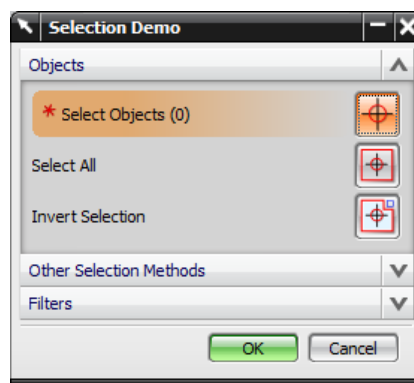
End Class
```

Since `result.Object` is of type `NXObject`, we have to use the `CType` operator to convert it to an `NX.Line` object. Then, once we have an `NX.Line` object, we can use the `Compute.ClosestPoints` function to find the point on this line that's closest to the cursor ray. This 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 result of our calculations.

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

So far, all our examples have involved selection of a single object. Selecting multiple objects uses very similar techniques, but the functions you use have names ending in "s". So, to create a dialog that allows selection of several objects, you would call `Selection.SelectObjects` (plural), rather than your old friend `Selection.SelectObject` (singular). This will cause the standard NX multi-selection dialog to appear



which 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 array functions to operate on the list of selected objects. For example, **Objects.Length** gives you the number of objects selected, and **Objects.Take(3)** gives you the first 3 of them.

## 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.Geom is Geom.Plane
            face.Color = Green
        End If
    Next face
Next body
```



# Chapter 14: 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 6, 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.



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. Also used 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, faceted bodies, datum objects, CAE objects. Displayable objects have colors, fonts, and other appearance attributes.

### 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 [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. 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 notes above cover 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. The code generated by journaling is sometimes verbose and difficult to read. But it's worth persevering, because hidden within that code is an example call showing you exactly the function you need.

## 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(2) 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(2) 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 10, 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)    ### wrong. Covariance ??
End If
```

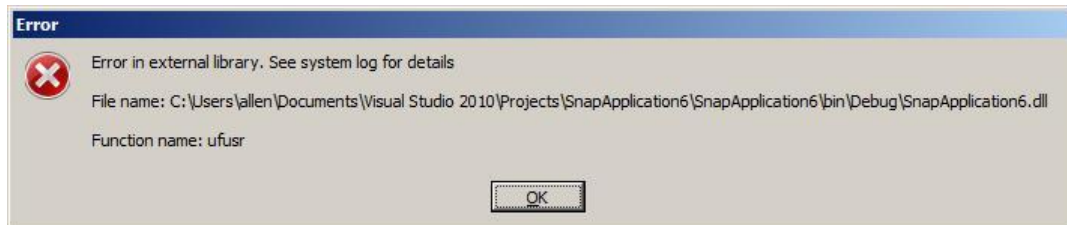
For the third case (blanking all the objects with certain characteristics), we will need to cycle through all the objects in our model, finding the ones that meet our criteria, and then pass these to the BlankObjects function. See the last section in chapter 13 for information about cycling through the objects in a part file.

# Chapter 15: 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. The syslog text is quite helpful here, as is often the case.

## If You Don’t Have .NET Framework V4

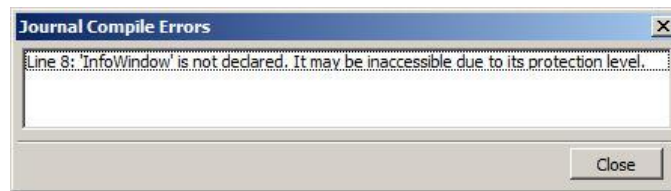
To use [SNAP](#), you need a fairly recent version of the .NET Framework (version 4.0 or newer) installed on your computer. If you don’t have this, then, the first time you try to run any code in the Journal Editor, you will receive this mysterious error message:



You will find similar text in the NX syslog, too. To check which version you have, look in your [Windows\Microsoft.NET\Framework](#) folder, and see if it contains a folder named v4.0.xxxx. Alternatively, you can use the “Programs and Features” Control Panel to check. If you don’t have version 4 or later, 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”.

## If You Don't Have a Snap Authoring License

From time to time, you may receive error messages like this:



In this particular example, the compiler is complaining that it can't find something called InfoWindow. If your code used the SNAP Line or Arc functions, the compiler would complain about those being "inaccessible", too. Now we know that InfoWindow and Line and Arc are all parts of **SNAP**, so this means that the compiler isn't able to access the **SNAP** assembly (Snap.dll). This may be because an environment variable is set incorrectly, so that the compiler is looking for Snap.dll in the wrong place. But, more likely, it means that no nx\_snap\_author license could be obtained. If you check down near the bottom of the syslog, it will tell you 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 8.0\ugii\managed\NXOpen.Utilities.dll as a reference item

Adding C:\Program Files\Siemens\NX 8.0\ugii\managed\NXOpen.dll as a reference item

Adding C:\Program Files\Siemens\NX 8.0\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

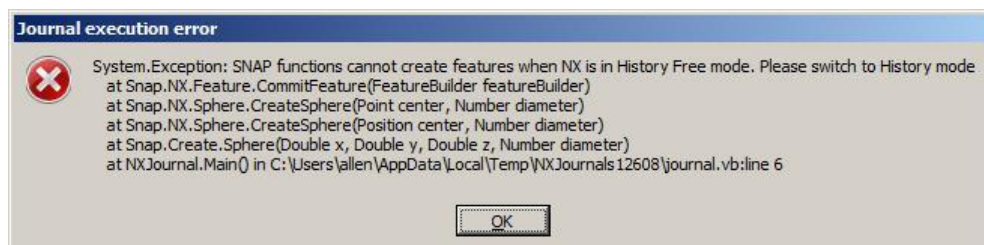
\*\*\*\*\*

Cannot obtain authoring license

So, as you can see, the system was able to add references to the various NXOpen dlls, but was not able to add Snap.dll because of a licensing failure. The failure could be due to a disconnect with your licensing server, but more likely it's because no nx\_snap\_author license is available for you.

## If You Try to Create Features in History-Free Mode

As explained at the end of chapter 8, 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.

## Can't Find Visual Studio Templates

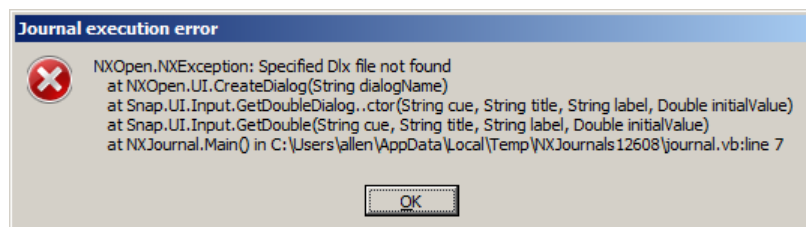
When you start working through the examples in chapter 3, you may find that the [SNAP](#) project templates ([SnapTemplateVB](#) and [SnapWinFormTemplateVB](#)) are not listed in the “New Project” dialog in Visual Studio. There are a few possible causes for this problem. First, maybe you forgot to copy the template ZIP files, as instructed near the beginning of chapter 3. You can find two zip files called [SnapTemplateVB.zip](#) and [SnapWinFormTemplateVB.zip](#) in a location like `C:\Program Files\Siemens\NX 8.0\UGOPEN\snap`. You should copy these two zip files into `<My Documents>\Visual Studio 2010\Templates\Project Templates\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 similarity of names.

Finally despite the warning in big red letters in chapter 3, maybe you unzipped the two zip files. You should not do this – Visual Studio cannot use them if they are unzipped.

## Can't Find dlx Files

The simple input dialogs described in chapter 12 depend on some dlx files that are typically found in a location like `C:\Program Files\Siemens\NX 8.0\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).