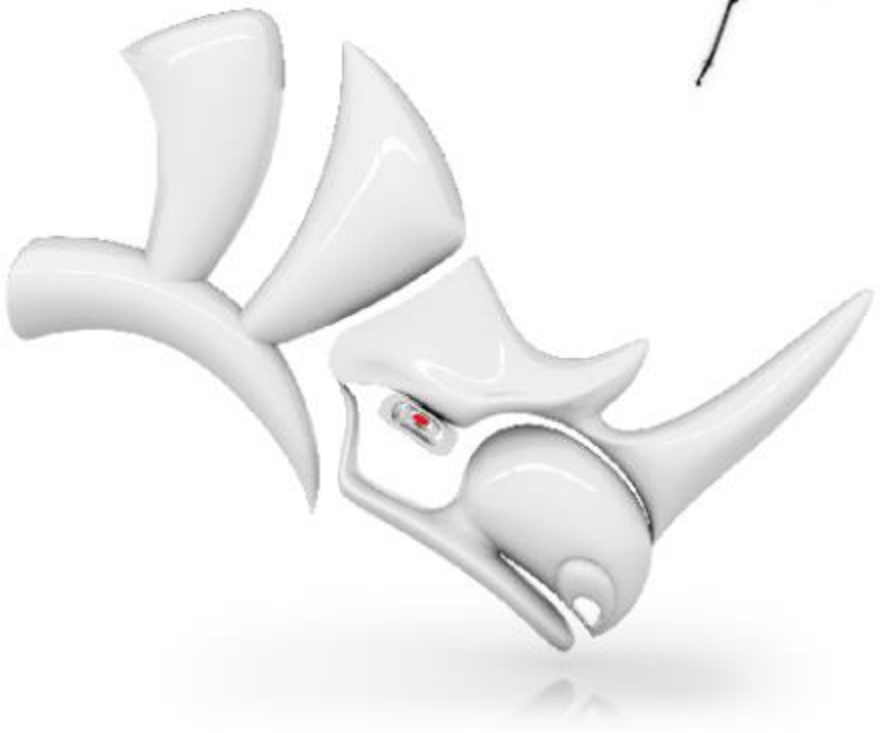
Essential Guide to C# Scripting for Grasshopper



Rajaa Issa
Robert McNeel & Associates

Table of Contents

Preface

This manual is intended for designers who are experienced in Grasshopper visual scripting, and would like to take their skills to the next level to create their own custom scripts using C# programming language. This guide does not assume nor require any background in programming. The document is divided into three parts. The first explains the general interface and parts of the C# scripting component in Grasshopper. The second reviews the basics of the C# DotNet programming language. The third part covers the main geometry types and functions in the RhinoCommon SDK. The guide includes many examples that are also available as a Grasshopper file available with the download of this guide. Note that all examples are written in version 6 of Rhino and Grasshopper.

I would like to acknowledge the excellent technical review by Mr. Steve Baer of Robert McNeel and Associates. I would also like to acknowledge Ms. Sandy McNeel for reviewing the writing and formatting of this document.

Rajaa Issa

Robert McNeel & Associates

Technical Support

Scripting, SDK and other developer help
There is a lot of information and samples in the McNeel Developer site
http://developer.rhino3d.com/

Forums and discussion groups
The Rhino community is very active and supportive and is a great place to post questions and look for answers. There are discussion sections reserved for scripting.
http://discourse.mcneel.com/

Chapter One: Grasshopper C# Component

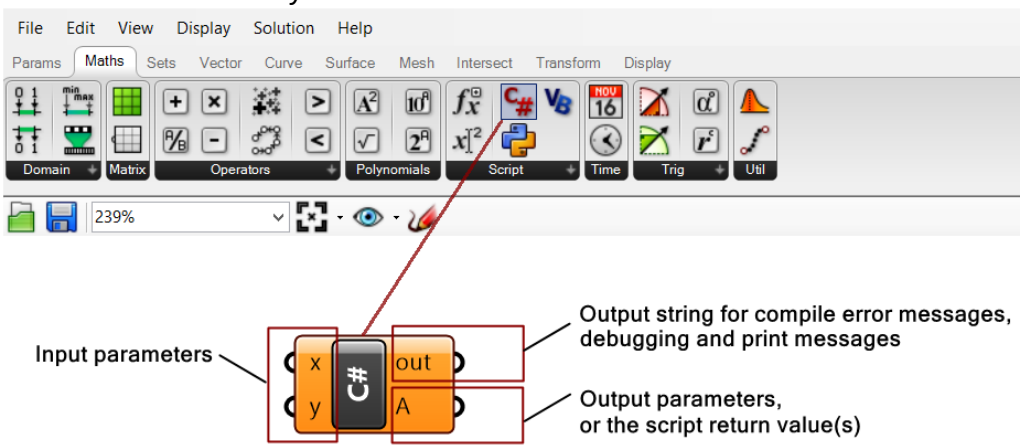
1_1: Introduction

Grasshopper supports multiple scripting languages such as ***VB.NET***, ***C#*** and ***Python*** to help develop custom components using the Rhino and Grasshopper SDKs (software development kit). Rhino publishes a cross-platform ***SDK*** for ***.NET*** languages called ***RhinoCommon***. The documentation of the SDK and other developer resources are available at http://developer.rhino3d.com/

1_2: C# component interface

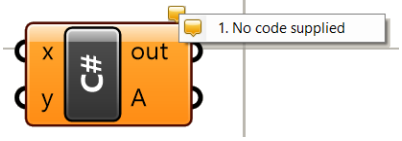
The scripting components are integrated within Grasshopper and have a similar interface to that of other typical components. They can read input and produce output, and have an editor to write custom code with access to ***RhinoCommon***. They are used to create specialized code and workflows not supported by other Grasshopper components. You can also use them to simplify, optimize and streamline your definitions by combining multiple functions.

To add an instance of the C# script component to the canvas, drag and drop the component from the ***Script*** panel under the ***Maths*** tab. The default script component has two inputs and two outputs. The user can change the names of input and output, set the input data type and data structure and also add more inputs and outputs parameters or remove them. We will explain how to do all that shortly.



Figure(1): The C# component and its location in the toolbar. **x**: first input parameter. **y**: second input parameter. **Out**: output string with compiling messages. **A**: Returned output of type object.

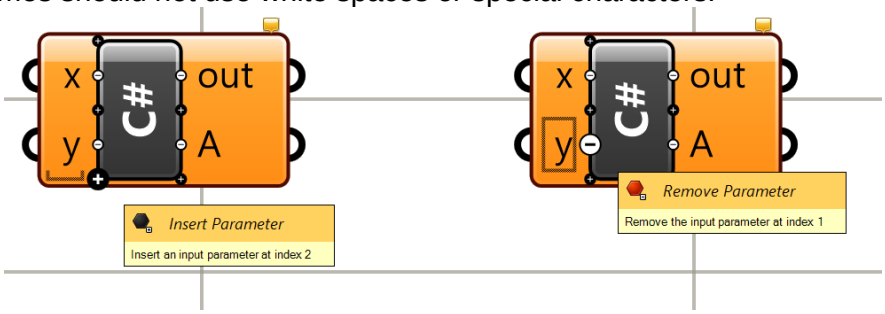
When you first drag to canvas, the component shows with the color orange indicating a warning. This is because there is no code typed inside it to start with. If you click on the bubble at the top-right corner, you can see what the warning is as shown in figure 2.



Figure(2): The default C# component in Grasshopper

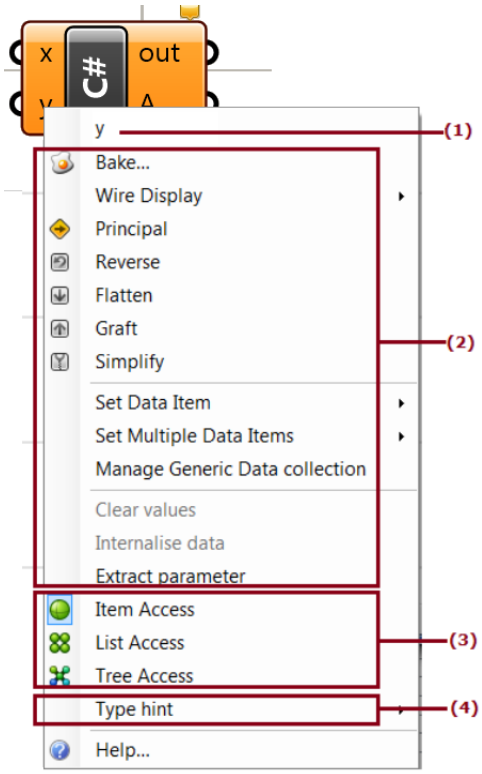
1_3: The input parameters

By default, there are two input parameters named ***x*** and ***y***. It is possible to edit the parameters' names, delete them or add new ones. If you zoom in, you will notice a few "+" and "-" signs appearing. You can click on those to add or remove parameters. You can also right-mouse click on a parameter to change its name. Note that the names of the parameters and their types are passed to the main function inside the script component, which is called ***RunScript***. It is a good practice to set the input and output parameter names to reflect what each parameter does. Parameter names should not use white spaces or special characters.



Figure(3): Zoom in to add a remove input parameters by pressing the + and - signs

If you right-mouse click on an input parameter, you will see a menu that has four parts as detailed in Figure 4 below.



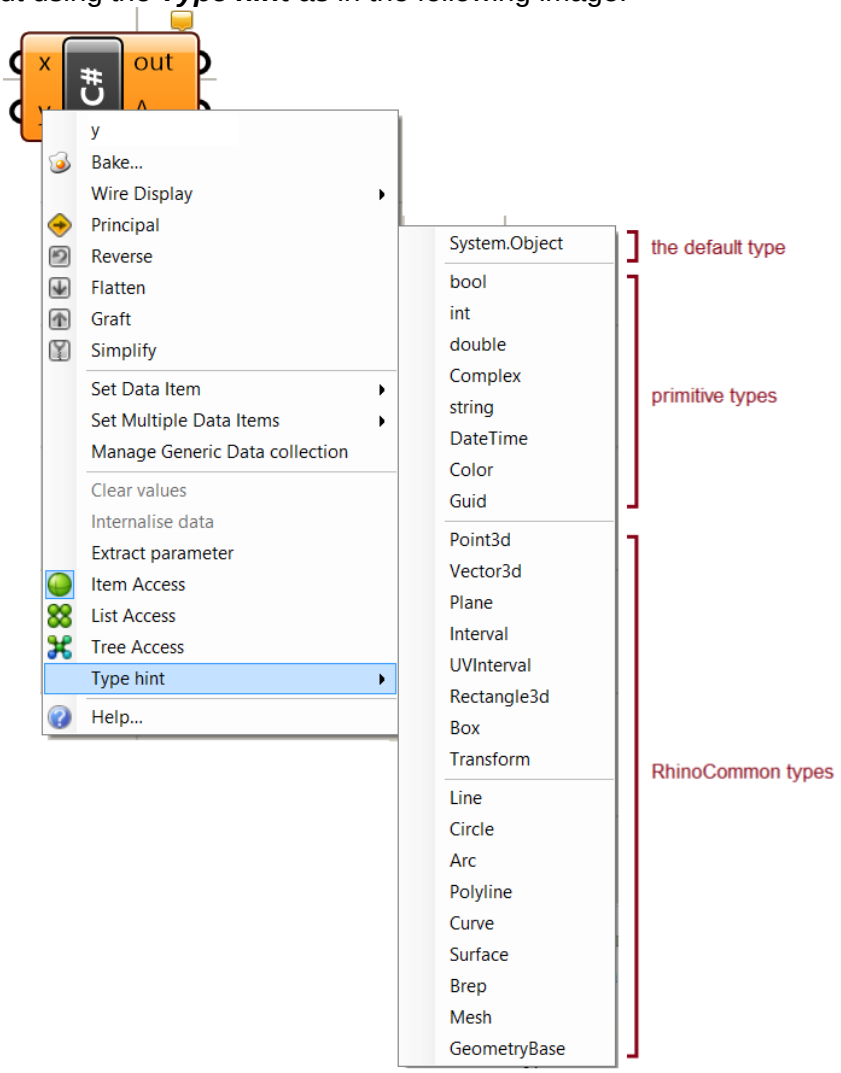
Figure(4): Expand the input parameter menu (access by right-mouse click on the parameter).

The input parts are:

1. **Name**: you can click on it and type a new parameter name.

2. **General attributes**: common to most other GH components.
3. **Data access**: to indicate whether the input should be accessed as a single item, a list of items or as a tree[1].
4. **Type**: Input parameters are set by default to be of an ***object*** type. It is best to specify a type to make the code more readable and efficient. Types can be primitive, such as ***int*** or ***double***, or ***RhinoCommon*** types that are used only in Rhino such as ***Point3d*** or ***Curve***. You need to set the type for each input.

Just like all other GH components, you can hook data to the input parameters of the script components. Your script component can process most data types. You can specify the data type of your input using the ***Type hint*** as in the following image.



Figure(5): The type hint in the input allows setting input parameters to specific type

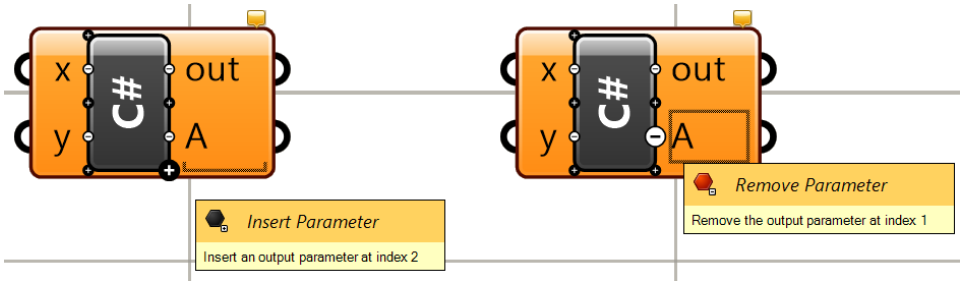
The ***Type hint***, gives access to many types and can be divided into three groups:

[1] We will explain how to traverse and navigate data access options in some detail later in this chapter.

1- ***System.Object***. if you do not specify the input type, GH assigns the base type ***System.Object***. The ***System.Object*** is the root type that all other types are built on. Therefore it is safe to assign *System.Object* type to any data type.
2- ***Primitive*** types. Those are made available by the ***.NET*** framework.
3- ***RhinoCommon*** types. Those are defined in the ***RhinoCommon SDK***.

1_4: The output parameters

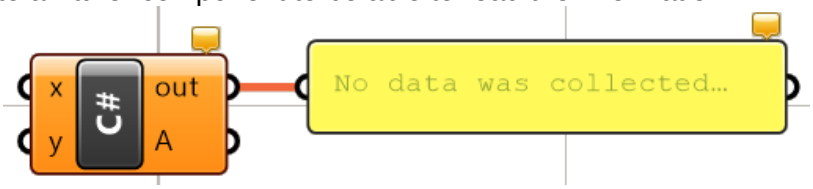
Just like with input parameters, you can add or remove output parameters by zooming in, then use the "+" or "-" signs. You can also change the name of the output. However, there is no data access or data types that you can set. They are always defined as ***System.Object***, and hence you can assign any type, and as defined by your script and GH, take care of parsing it to use downstream.



Figure(6): Zoom in to add or remove an output parameter by pressing the + and - gadgets

1_5: The out string

The ***out*** string is used to list errors and other useful information about your code. You can connect the ***out*** to a ***Panel*** component to be able to read the information.



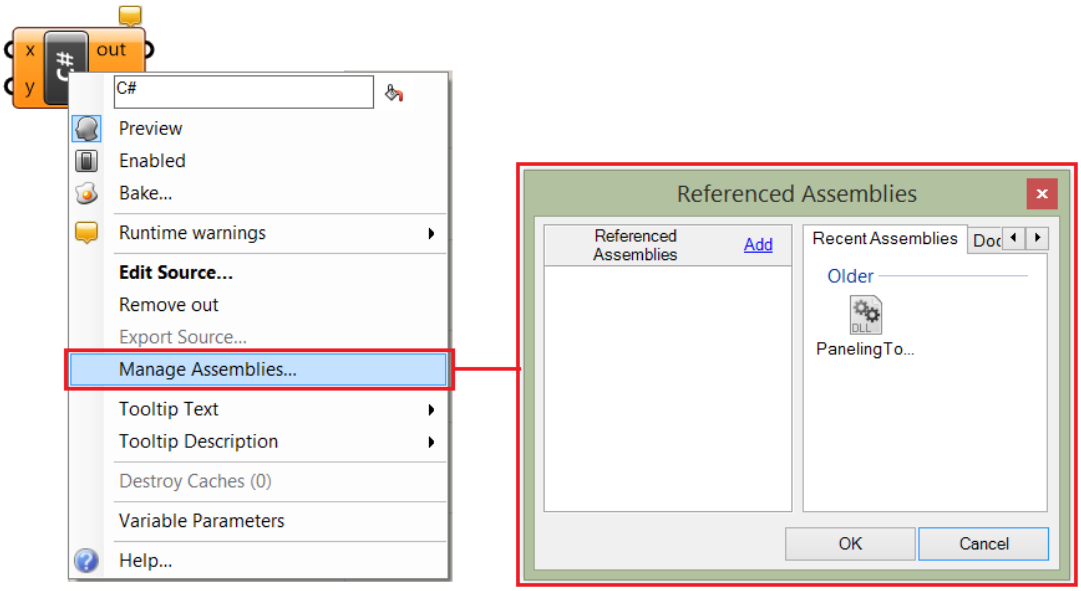
Figure(7): The out parameter includes compile and runtime messages

There are two types of messages that print to the out string.

1- ***Compile-time*** messages. These include compiler errors and warnings about your code. This is very helpful information to point you to the lines in code that the compiler is complaining about, so that you can correct the errors.
2- ***Runtime*** messages. You can print any text to the ***out*** string to track information generated inside your code during execution.

1_6: Main menu

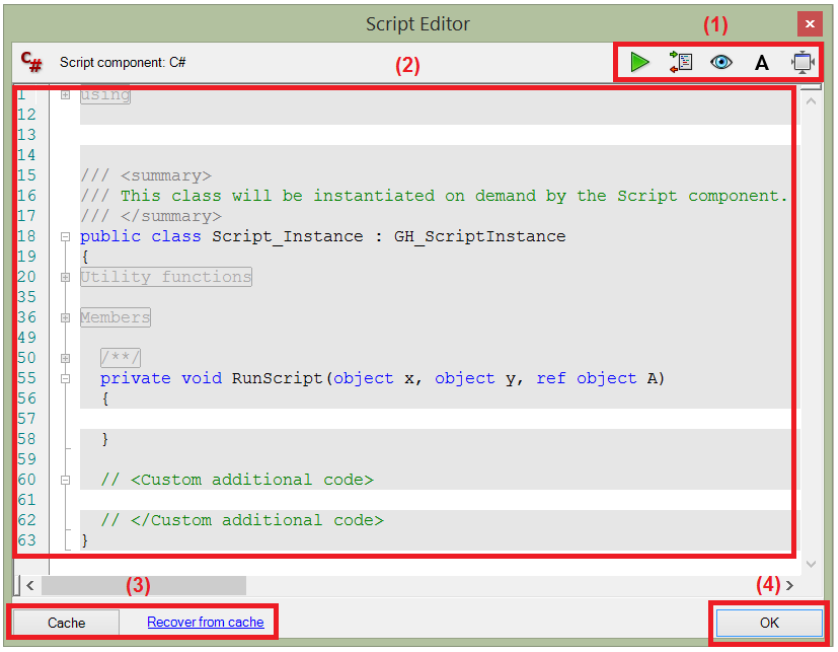
You can access the main component menu with a mouse-right-click while hovering over the middle of the scripting component. Most of the functions are similar to other GH components, but there are a couple specialized functions such as ***Edit Source...*** to open the code editor and ***Manage Assemblies*** to help add external libraries to access in your script.



Figure(8): Manage assemblies allows adding external libraries to access inside the scripting component

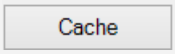
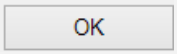
1_7: Code editor

To show the code editor window for the C# script component, you need to double click in the middle of the component (or right-mouse-click the middle, and select ***Edit Source…***). The code editor consists of 4 parts; the toolbar, the code, the caching tools and the ***OK*** button as in the following figure 9.

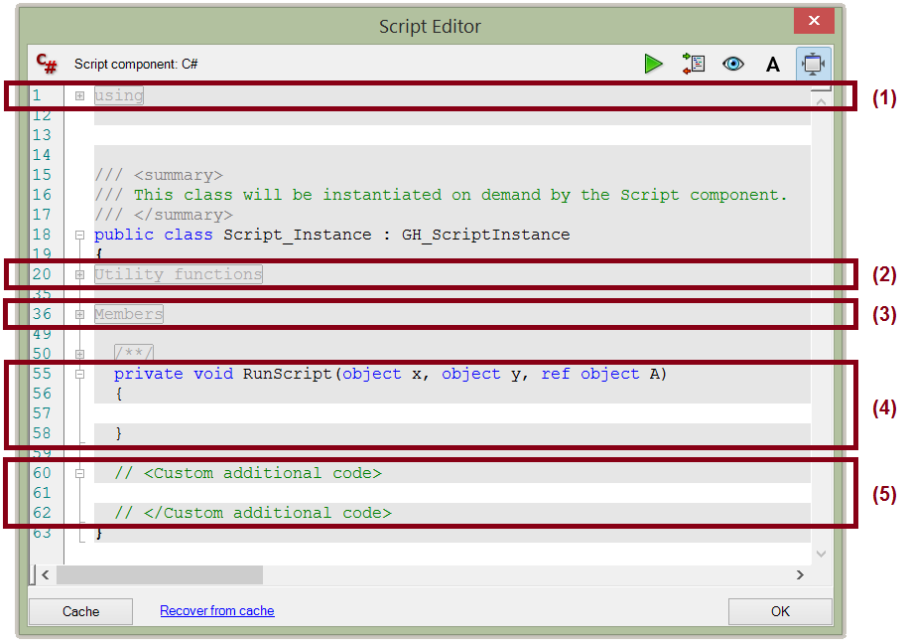


Figure(9): The code editor has 4 parts. (1) toolbar, (2) code, (3) cache, (4) OK

The top and bottom toolbars have multiple buttons. The following explains what each one means.

	Runs the code without closing the editor
	If you need to run some code once before calling your ***RunScript*** or afterwards, then there are functions that you can place in your additional code section. This may be used to initialize or reset some variables or settings before and after running your script.
	Insert additional overrides that are related to preview. This helps you create a custom preview.
	Set the editor font.
	If selected, the editor will shrink when it goes out of focus (you click outside the editor).
	When clicked, the current content of your script is saved. Even if you make modifications, you can revert to a previous state by clicking "Recover from cache". There are time stamps of the saved states.
	Click to close the editor and run the script.

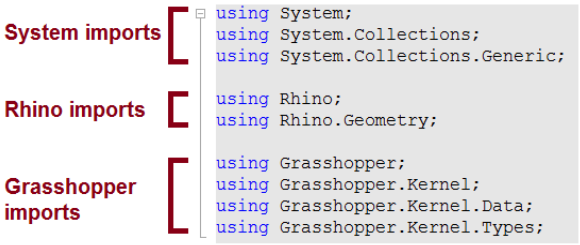
The code part of the scripting component has five sections as in figure 10. Next, we will expand and explain each of the parts.



Figure(10): The 5 parts of the code section of the C# scripting component: (1) imports, (2) utility functions, (3) Members, (4) the RunScript function, and (5) custom additional code.

1_7_1: Imports

There are assemblies that you can access and use in your code. Most of them are the .Net system ones, but there are also the Rhino and Grasshopper assemblies that give access to the Rhino geometry classes and the Grasshopper types and functions.



Figure(11): Default imports in the C# scripting component

You can also add your custom libraries (using the ***Manage Assemblies***). Externally referenced libraries appear at the bottom of the import list.

1_7_2: Utility functions

This section defines a few useful utilities that you can use inside your code mostly to debug or communicate useful information. You can use the ***Print()*** functions to send strings to the ***out*** output parameter. You can use the ***Reflect()*** functions to gain information regarding classes and their methods. For example, if you wish to get more information about the data that is available through the input parameter ***x***, you can add ***Reflect(x)*** to your code. As a result, a string with type method information will be written to the ***out*** parameter.

```
#region Utility functions
  /// <summary>Print a String to the [Out] Parameter of the Script component.</summary>
  /// <param name="text">String to print.</param>
  private void Print(string text)...
  /// <summary>Print a formatted String to the [Out] Parameter of the Script component.</summary>
  /// <param name="format">String format.</param>
  /// <param name="args">Formatting parameters.</param>
  private void Print(string format, params object[] args)...
  /// <summary>Print useful information about an object instance to the [Out]
  ///          Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj)...
  /// <summary>Print the signatures of all the overloads of a specific method to the [Out]
  ///          Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj, string method_name)...
#endregion
```

Figure(12): Utility functions the come by default with the C# scripting component

1_7_3: Members

Members are useful variables that can be utilized in your code. All are read only access, which means that you cannot change in your code. Members include a one and only instance to the active Rhino document, an instance to the Grasshopper document, the current GH scripting component and finally the iteration variable, which references the number of times the script component is called (usually based on the input and data access). For example, when your input parameters are single items, then the script component runs only once, but if you input a list of values, say 10 of them, then the component is executed 10 times (that is assuming that the ***data access*** is set to a ***single item***). You can use the ***Iteration*** variable if you would like your code to do different things at different iterations, or to simply analyze how many times the component is called.

```
#region Members
  /// <summary>Gets the current Rhino document.</summary>
  private readonly RhinoDoc RhinoDocument;
  /// <summary>Gets the Grasshopper document that owns this script.</summary>
  private readonly GH_Document GrasshopperDocument;
  /// <summary>Gets the Grasshopper script component that owns this script.</summary>
  private readonly IGH_Component Component;
  /// <summary>
  /// Gets the current iteration count. The first call to RunScript() is associated with Iteration==0.
  /// Any subsequent call within the same solution will increment the Iteration count.
  /// </summary>
  private readonly int Iteration;
#endregion
```

Figure(13): Member region includes read-only variables that point to the document and the component. Also includes an integer variable for the iteration count.

1_7_4: The RunScript

This is the main function where you write your code. The signature of the ***RunScript*** includes the input and output parameters along with their data types and names. There is white space between the open and closed parentheses to indicate the region where you can type your code.

```
  /// <summary>
  /// This procedure contains the user code. Input parameters are provided as regular arguments,
  /// Output parameters as ref arguments. You don't have to assign output parameters,
  /// they will have a default value.
  /// </summary>
  private void RunScript(object x, object y, ref object A)
  {

  }
```

Figure(14): The RunScript is the main function within which the user can put their code.

1_7_5: Custom additional code

You can place all additional functions and variables in the ***<Custom additional code>*** area just below the ***RunScript***. Your main code inside the ***RunScript*** can reference the additional members and functions placed in this portion of the code.

```
  // <Custom additional code>

  // </Custom additional code>
```

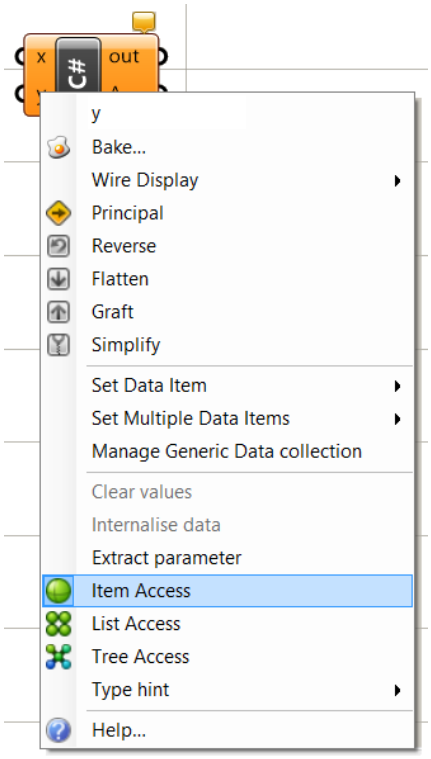
Figure(15): There is a designated region to put the additional custom code outside the RunScript function.

1_8: Data access

This topic requires knowledge in C# programming. If you need to review or refresh you knowledge in C# programming, please review chapter two before reading this section. Grasshopper scripting components, just like all other GH components, can process three types of data access; ***item access***, ***list access*** and ***tree access***.

Icon in GH	Data Access
	**Item Access**
	**List Access**
	**Tree Access**

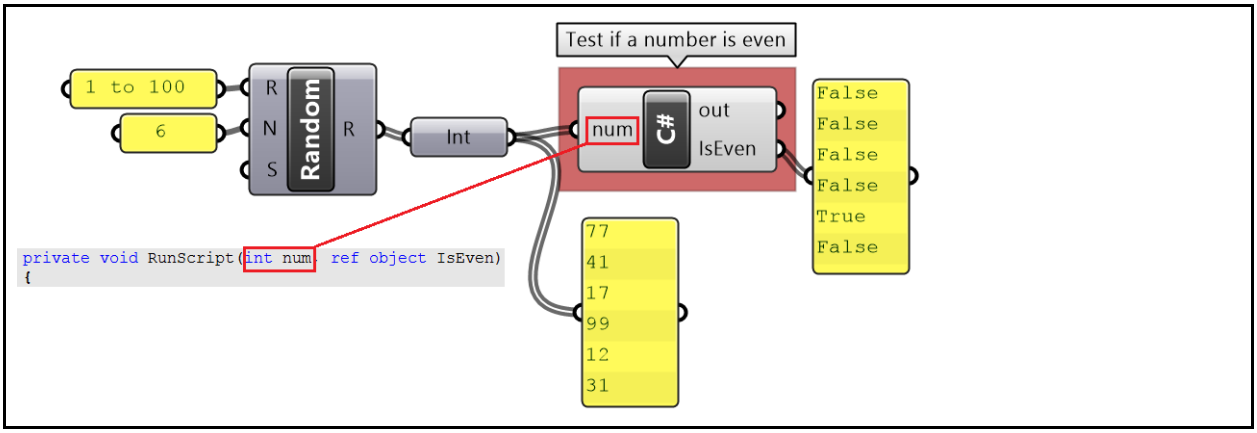
You need to right-mouse-click on the input parameter to set its data access, otherwise it is set to ***item access*** by default. We will explain what each access means, and how data is processed inside the component in each case.



Figure(22): Set the input parameter to be an ***item access*** to indicate that input in processed one element at a time

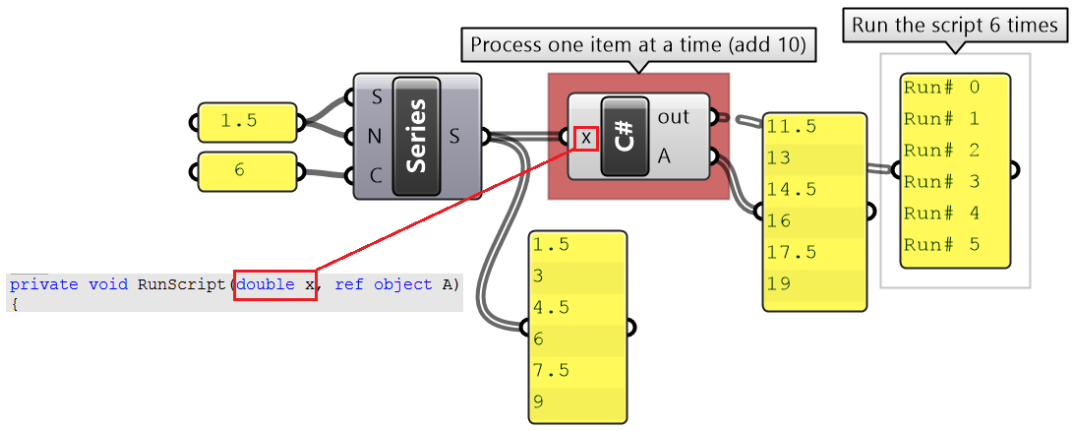
1_8_1: Item access

Item access means that the input is processed one item at a time. Suppose you have a list of numbers that you need to test if they are odd or even. To simplify your code, you can choose to only process one number at a time. In this case, ***item access*** is appropriate to use. So even if the input to ***num*** is a list of integers, the scripting component will process one integer at a time and run the script a number of times equal to the length of the list.



```
private void RunScript(int num, ref object IsEven)
{
    int mod = num % 2;
    IsEven = (mod == 0 ? true : false);
}
```

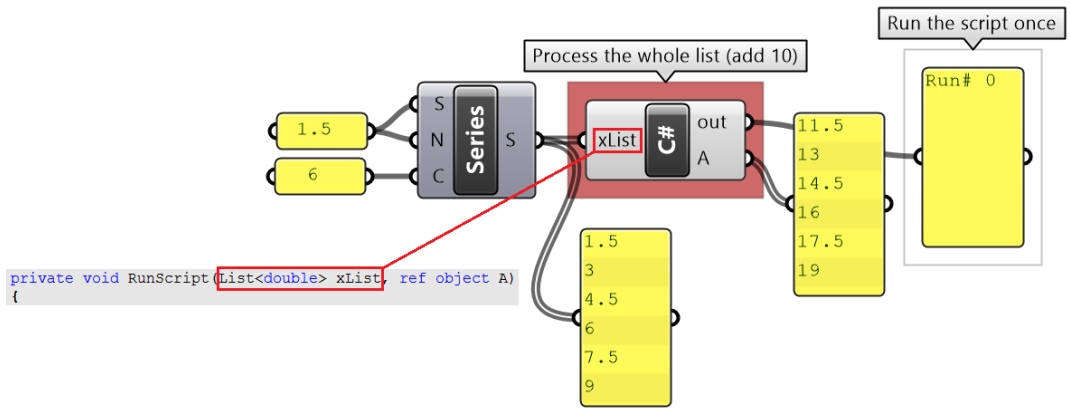
With ***item access***, each element in a list is processed independently from the rest of the elements. For example if you have a list of 6 numbers {1, 2, 3, 4, 5, 6} that you like to increment each one of them by some number, let's say 10, to get the list {11, 12, 13, 14, 15, 16}, then you can set the data access to ***item access***. The script will run 6 times, once for each element in the list and the output will be a list of 6 numbers. Notice in the following implementation in GH, ***x*** is input as a single item of type ***double***.




```
private void RunScript(double x, ref object A)
{
    A = x + 10;
    Print("Run# " + Iteration);
}
```

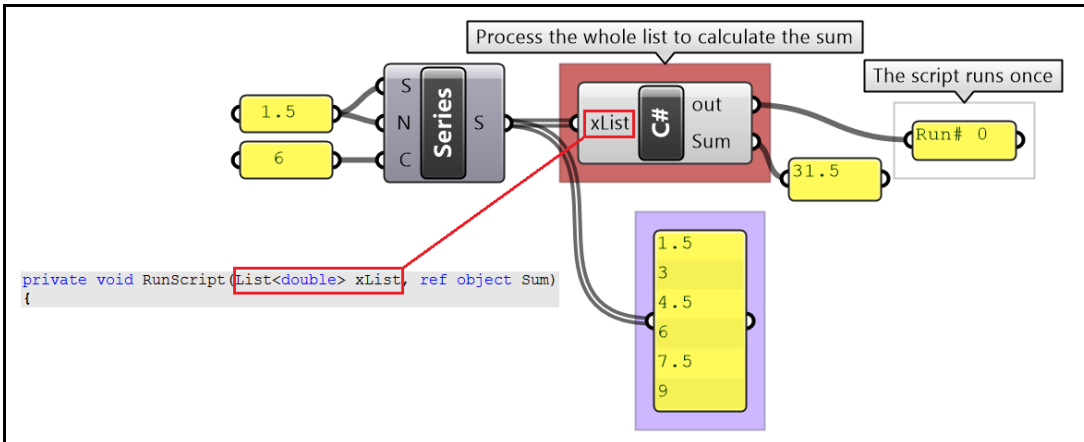
1_8_2: Lists access

List access means that the whole list is processed all in one call to the scripting component. In the previous example, we can change the data access from ***item access*** to ***list access*** and change the script to add 10 to each one of the items in the list as in the following. Note that the script runs only once.




```
private void RunScript( List<double> xList, ref object A)
{
    List<double> newList = new List<double>();
    foreach( var x in xList)
      newList.Add(x + 10);
    Print("Run# " + Iteration);
    A = newList;
}
```

The ***list access*** is most useful when you need to process the whole list in order to find the result. For example, to calculate the sum of a list of input numbers, you need to access the whole list. Remember to set the data access to ***list access***.

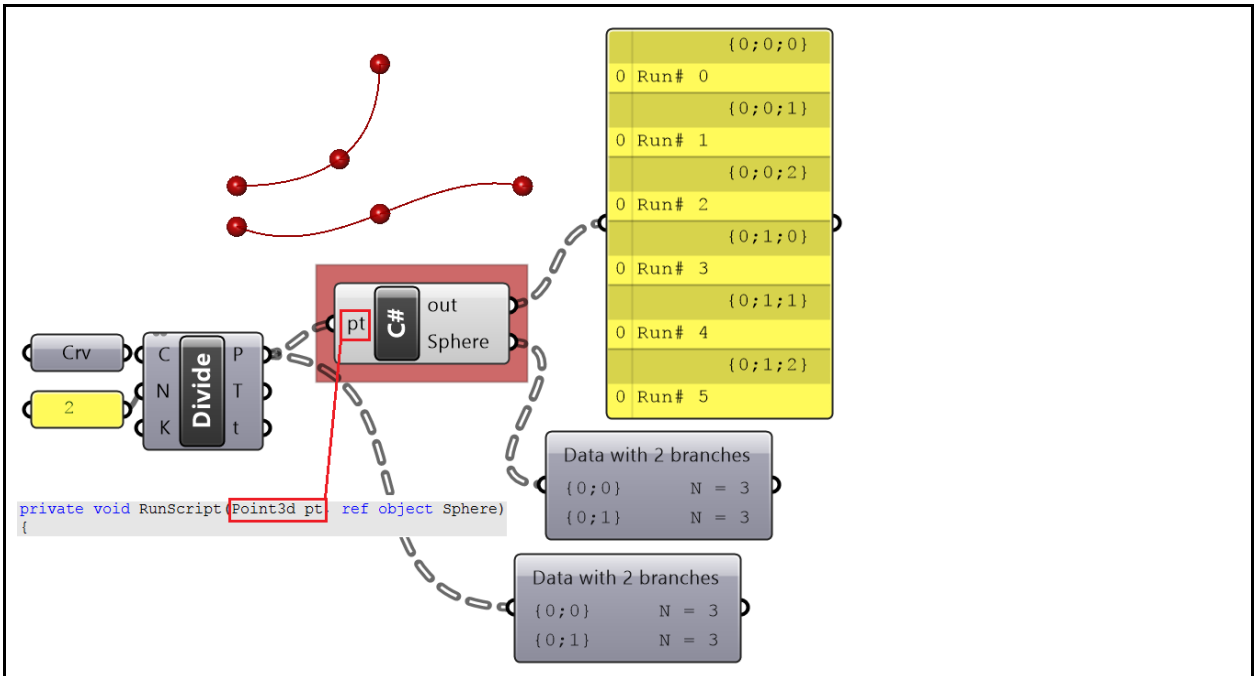



```
private void RunScript( List<double> xList, ref object Sum)
{
    double xSum = 0;
    foreach( var x in xList)
       xSum = xSum + x;
    Print("Run# " + Iteration);
    Sum = xSum;
}
```

1_8_3: Tree access

Tree data access in Grasshopper is an advanced topic. You might not need to deal with trees, but if you encounter them in your code, then this section will help you understand how they work and how to do some basic operations with them.

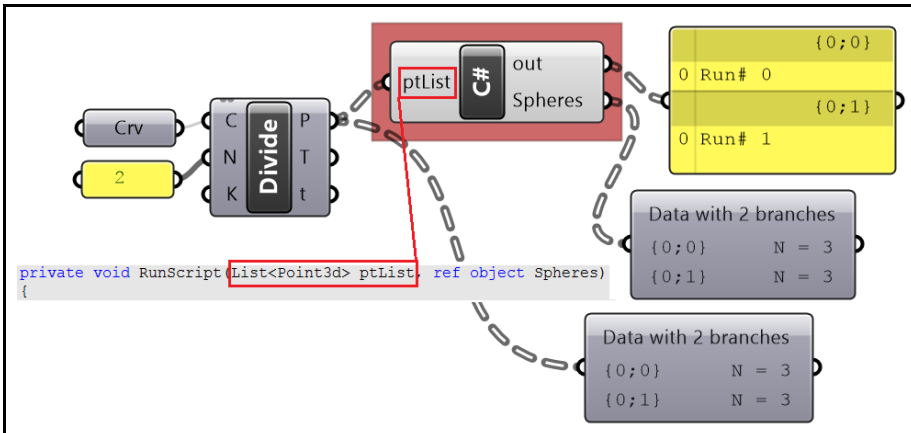
Trees (or multi-dimensional data) can be processed one element at a time or one branch (path) at a time or all branches at the same time. That will depend on how you set the data access (item, list or tree). For example, if you divide two curves into two segments each, then we get a tree structure that has two branches (or paths), each with three points. Now suppose you want to write a script to place a sphere around each point. The code will be different based on how you set the ***data access***. If you make it ***item access***, then the script will run 6 times (once for each point), but the tree data structure will be maintained in the output with two branches and three spheres in each branch. Your code will be simple because it deals with one point at a time.




```
private void RunScript( Point3d pt, ref object Sphere)
{
    Sphere ptSphere = new Sphere(x, 0.5);
    Print("Run# " + Iteration);
    Sphere = ptSphere;
}
```

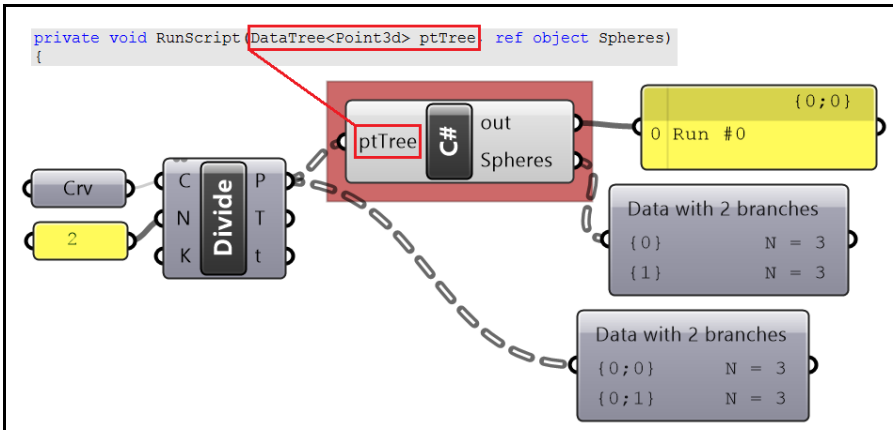
It should be ""pt" instead of "x"

The same result can be achieved with the ***list access***, but this time your code will have to process each branch as a list. The script runs 2 times, once for each branch.




```
private void RunScript( List<Point3d> ptList, ref object Spheres)
{
  List<Sphere> sphereList = new List<Sphere>();
  foreach (Point3d pt in ptList) {
    Sphere sphere = new Sphere(pt, 0.5);
    sphereList.Add(sphere);
  }
  Print("Run# " + Iteration);
  Spheres = sphereList;
}
```

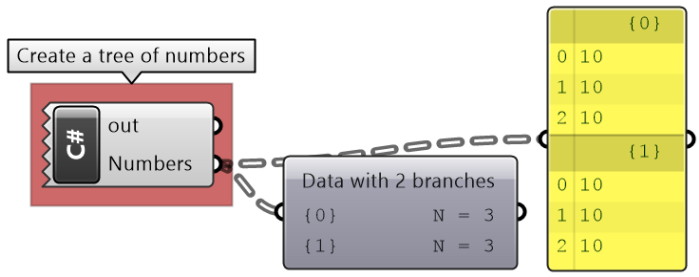
The ***tree access*** allows you to process the whole tree and your code will run only once. While this might be necessary in some cases, it also complicates your code.




```
private void RunScript( DataTree<Point3d> ptTree, ref object Spheres)
{
  //Declare a tree of spheres
  DataTree<Sphere> sphereTree = new DataTree<Sphere>();
  int pathIndex = 0;
  foreach (List<Point3d> branch in ptTree.Branches) {
    List<Sphere> sphereList = new List<Sphere>();
    GH_Path path = new GH_Path(pathIndex);
    pathIndex = pathIndex + 1;
    foreach (Point3d pt in branch)
```

```
    {
      Sphere sphere = new Sphere(pt, 0.5);
      sphereList.Add(sphere);
    }
    sphereTree.AddRange(sphereList, path);
  }
  Print("Run# " + Iteration);
  Spheres = sphereTree;
}
```

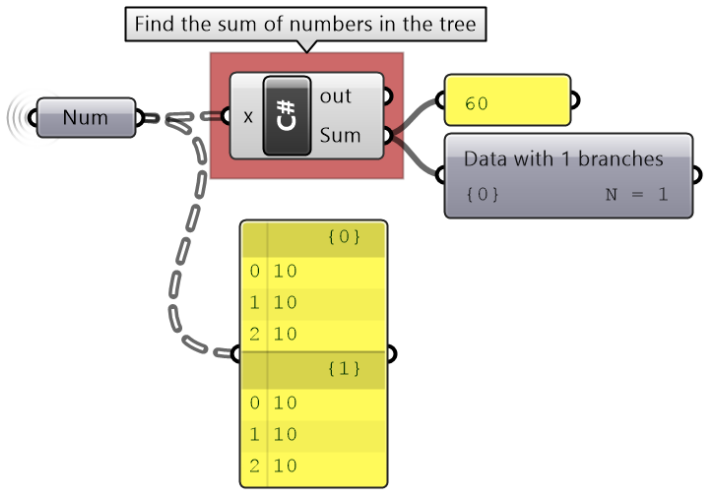
The tree data structure is a special data type that is unique to Grasshopper[2]. If you need to generate a tree inside your script, then the following shows how to do just that.




```
private void RunScript( ref object Numbers)
{
  DataTree<double> numTree = new DataTree<double>();
  int pathIndex = 0;
  for (int b = 0; b <= 1; b++) {
    List<double> numList = new List<double>();
    GH_Path path = new GH_Path(pathIndex);
    pathIndex = pathIndex + 1;
    for (int i = 0; i <= 2; i++) {
      numList.Add(10);
    }
    numTree.AddRange(numList, path);
  }
  Numbers = numTree;
}
```

The following example shows how to step through a given data tree of numbers to calculate the overall sum.

[2] For more details about Grasshopper data structures, please refer to the "*Essential Algorithms and Data structures for Grasshopper*". Free download is available here: https://www.rhino3d.com/download/rhino/6.0/essential-algorithms




```
private void RunScript( DataTree<double> x, ref object Sum)
{
    double xSum = 0;
    foreach (List<double> branch in x.Branches){
      foreach (double num in branch) {
        xSum = xSum + num;
      }
    }
    Sum = xSum;
}
```

Chapter Two: C# Programming Basics

2_1: Introduction

This chapter covers basic C# programming concepts. It serves as an introduction and quick reference to the language syntax. It is not meant to be complete by any measure, so please refer to the C# resources available online and in print[3]. All examples in this chapter are implemented using the Grasshopper C# component.

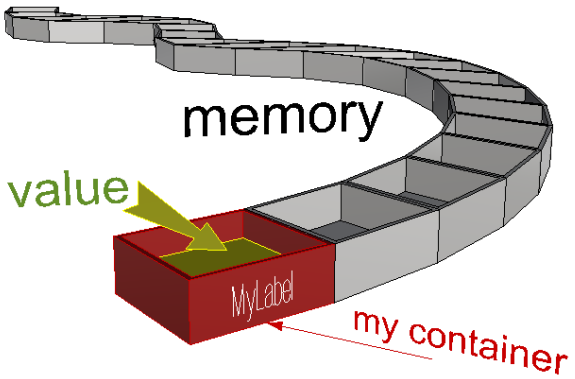
2_2: Comments

Comments are very useful to describe your code in plain language. They are useful as a reminder for you, and also for others to help them understand your code. To add a comment, you can use two forward slashes // to signal that the rest of the line is a comment and the compiler should ignore it. In the Grasshopper C# code editor, comments are displayed in green. You should enclose a multi-line comment between /* and */ as in the following example.

```
// The compiler ignores this line
/*
  The compiler ignores all lines enclosed
  within this area
*/
```

2_3: Variables

You can think of variables as labeled containers in your computer's memory where you can store and retrieve data. Your script can define any number of variables and label them with the names of your choice, as long as you do not use spaces, special characters, or reserved words by the programming language. Try to always variable names that are descriptive of the data you intend to store. This will make it easier for you to remember what kind of information is stored in each variable.



Figure(16): How variables are stored in the computer memory

Your script uses variable names to access the value stored in them. In general, you can assign new values to any variable at any point in your program, but each new value wipes out the old

[3] The Microsoft documentation is a good resource: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/

one. When you come to retrieve the value of your variable, you will only get the last one stored. For example, let's define a variable and name it ***x***. Suppose we want ***x*** to be of type integer. Also,suppose we would like to assign it an initial value of 10. This is how you write a statement in C# to declare and assign an integer:

```
int x = 10;
```

Let us dissect all the different parts of the above statement:

int	The type of your data. ***int*** is a special keyword in ***C#*** that means the type of the variable is a signed integer (can be a positive or negative whole number)
x	The name of the variable.
=	Used for assignment, and it means that the value that follows will be stored in the variable x.
10	The initial value stored in the ***x*** variable.
;	The Semicolon is used to end a single statement[4].

In general, when you declare a variable, you need to explicitly specify the ***data type***.

2_4: Operators

Operators are used to perform arithmetic, logical and other operations. Operators make the code more readable because you can write expressions in a format similar to that in mathematics. So instead of having to use functions and write ***C=Add(A, B)***, we can write ***C=A+B***, which is easier to read. The following is a table of the common operators provided by the C# programming language for quick reference:

Type	Operator	Description
Arithmetic Operators	^	Raises a number to the power of another number.
	*	Multiplies two numbers.
	/	Divides two numbers and returns a floating-point result.
	\	Divides two numbers and returns an integer result.
	%	Remainder: divides two numbers and returns only the remainder.
	+	Adds two numbers or returns the positive value of a numeric expression.
	-	Returns the difference between two numeric expressions or the negative value of a numeric expression

[4] For more details about C# statements refer to the Microsoft documentation: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements

Assignment Operators	=	Assigns a value to a variable
	*=	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.
	+=	Adds the value of a numeric expression to the value of a numeric variable and assigns the result to the variable. Can also be used to concatenate a String expression to a String variable and assign the result to the variable.
	-=	Subtracts the value of an expression from the value of a variable and assigns the result to the variable.
Comparison Operators	<	Less than
	<=	Less or equal
	>	Greater than
	>=	Greater or equal
	==	Equal
	!=	Not equal
Concatenation Operators	&	Generates a string concatenation of two expressions.
	+	Concatenate two string expressions.
Logical Operators	&&	Performs a logical conjunction on two Boolean expressions
	!	Performs logical negation on a Boolean expression
	||	Performs a logical disjunction on two Boolean expressions

2_5: Namesapces

Namespaces are very useful to group and organize classes especially for large projects. The ***.NET*** uses namespaces to organize its classes. For example ***System*** namespace has many classes under it including the ***Math*** class, so if you like to calculate the square root of a number, your code will look like the following:

```
double num = 16;
double sqrNum = System.Math.Sqrt(num);
```

Notice how you use the namespace followed by "." to access the classes within that namespace. If you do not want to type the namespace each time you need to access one of the classes within that namespace, then you can use the ***using*** keyword as in the following.

```
using System;
```

```
double num = 16;
double sqrNum = Math.Sqrt(num);
```

2_6: Data

Data types refer to the kind of data stored in the variable. There are two main data types. The first is supplied by the programming language and involves things like numbers, logical true or false, and characters. Those are generally referred to as primitive or built-in types. Variables of any data type can be composed into groups or collections.

2_6_1: Primitive data types

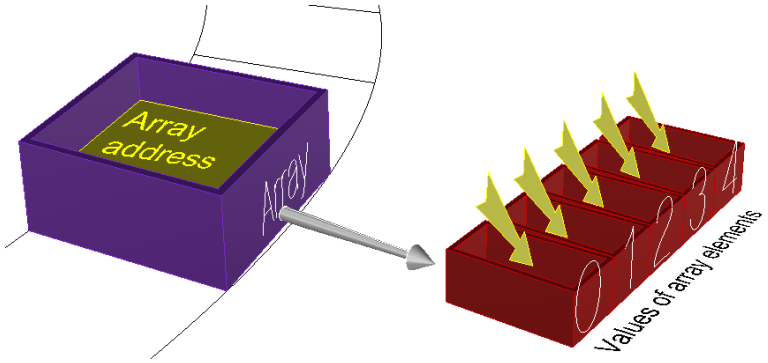
Primitive types refer to the basic and built-in types provided by the programming language. Following examples declare variables of primitive data types:

Declare primitive data types	Notes
double pi = 3.1415;	***double***: big number with decimal point.
bool pass = true;	***bool***: set to either true or false. Used mainly to represent the truth value of a variable or a logical statement.
char initial = 'R';	***char***: stores exactly one character.
string myName = "Mary";	***string***: a sequence of characters.
object someData = "Mary";	***object*** type can be used to store data of any type. The use of ***object*** type is inefficient and should be avoided if at all possible.

2_6_2: Collections

In many cases, you will need to create and manage a group of objects. There are generally two ways to group objects: either by organizing them in arrays or using a collection. Collections are more versatile and flexible, and they allow you to expand or shrink the group dynamically. There are many ways to create collections, but we will focus on ordered ones that contain elements of the same data type. For more information, you can reference the literature.

Arrays
Arrays are a common way to assemble an ordered group of data. Arrays are best suited if you already know the values you are storing, and do not need to expand or shrink the group dynamically.



Figure(17): How arrays are stored in the computer memory

For example, you can organize the days of the week using an array. The following declares and initializes the weekdays array in one step.

```
//Declare and initialize the days of the week array
string[ ] weekdays = {"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};
```

You can also declare the array first with a specific dimension representing the number of elements to allocate in memory for the array, then assign the values of the elements in separate statements. In C# language, each statement ends with a semicolon.

```
//Declare the days of the week in one statement, then assign the values later
string[ ] weekdays = new string[7];
weekdays[0] = "Sunday";
weekdays[1] = "Monday";
weekdays[2] = "Tuesday";
weekdays[3] = "Wednesday";
weekdays[4] = "Thursday";
weekdays[5] = "Friday";
weekdays[6] = "Saturday";
```

Let's take the statement that sets the first value in the weekdays array in the above.

```
weekdays[0] = "Sunday";
```

Sunday is called an ***array element***. The number between brackets is called the ***index*** of the element in the array. Notice that in ***C# language*** the array always starts with index=0 that points to the first element. The last index of an array of seven elements therefore equals 6. If you try to retrieve data from an invalid index, for example 7 in the weekday example above, then you will get what is called an ***out of bound error*** because you are basically trying to access a part of the memory that you did not allocate for your array and it can lead to a crash.

Lists

If you need to create an ordered collection of data dynamically, then use a ***List***. You will need to use the ***new*** keyword and specify the data type of the list. The following example declares a new list of integers and appends elements incrementally in subsequent statements. Note that the indices of a list are also zero based just like arrays.

```
//Declare a dynamic ordered collection using “List”
List<string> myWorkDays =  new List<string>();

//Add any number of new elements to the list
myWorkDays.Add("Tuesday");
myWorkDays.Add("Wednesday");
myWorkDays.Add("Thursday");
myWorkDays.Add("Friday");
```

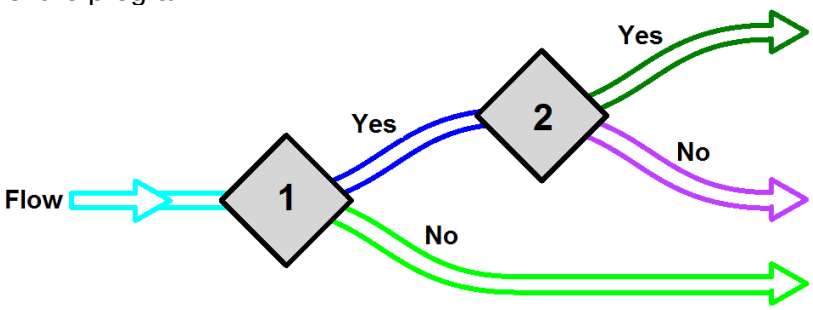
You need to use the keyword ***new*** to declare an instance of a ***List***.. We will explain what it means to be a reference type with some more details later.

2_7: Flow control

The flow of your scripts indicates the order of code execution. Sometimes you might need to branch or loop through specific statements several times. For example, you might want to branch your script to implement a different sequence based on some condition. Other times, you might need to repeat a certain sequence multiple times. We will discuss three important mechanisms to control the flow of a program; conditional statements, loops and methods.

2_7_1: Conditional statements

Conditional statements allow you to decide on which part of your code to use based on some condition that can change during run time. Unlike regular programming instructions, conditional statements can be described as decision-making logic. Conditional statements help control and regulate the flow of the program.



Figure(17): Conditional statements 1 and 2, and how they affect the flow of the program.

The following script examines a number variable and prints “zero” if it equals zero:

```
if (myNumber == 0)
{
    Print("myNumber is zero");
}
```

Description	
if	Keyword indicating the start of the if statement
(myNumber == 0)	The condition of the if statement enclosed by parentheses

{
  Print("myNumber is zero");
}
	The if statement block enclosed between curly brackets. The block is executed only if the condition evaluates to true, otherwise it is skipped.

The following code checks if a number is between 0 and 100:

```
if (myNumber >= 0 && myNumber <= 100)
{
   Print("myNumber is between 0 and 100");
}
```

Sometimes the script needs to execute one block when some condition is satisfied, and another if not. Here is an example that prints the word ***positive*** when the given number is greater than zero, and prints ***less than 1*** if not. It uses the ***if… else*** statement.

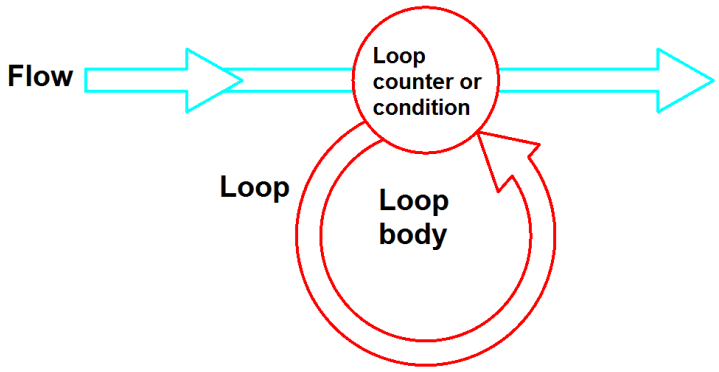
```
if (myNumber > 0)
{
   Print("positive");
}
else
{
   Print( "less than 1" );
}
```

You can use as many conditions as you need as shown in the following example.

```
if (myNumber > 0)
{
   Print("myNumber is positive");
}
else if (myNumber < 0)
{
   Print( "myNumber is negative" );
}
else
{
   Print( "myNumber is zero" );
}
```

2_7_2: Loops

Loops allows you to run the body of your loop a fixed number of times, or until the loop condition is no longer true.



Figure(18): Loops in the context of the overall flow of the program

There are two kinds of loops. The first is iterative where you can repeat code a defined number of times, and the second is conditional where you repeat until some condition is no longer satisfied.

Iterative loops: for loop

This is a common way of looping when you need to run some lock of code a specific number of times. Here is a simple example that prints numbers from 1 to 10. You first declare a counter and then increment in the loop to run the code specific number of times. Notice that the code statements that you would like to repeat are bound by the block after the for statement.

```
for (int i = 1; i <= 10; i++)
{
  //Convert a number to string
  Print( i.ToString() );
}
```

Description	
`for (int i = 1; i <= 10; i++)`	The for statement. It has 4 parts: **for(... ; … ; ...)**: the for keyword and loop condition and counter **i=1**: the counter initial value **i<=10**: the condition: if evaluates to true, then execute the body of the for loop **i++**: increment the counter (by 1 in this case)

`{`
`  //Convert a number to string`
`  Print( i.ToString() );`
`}`
	The body of the for loop. This is the code that is executed as long as the condition is met.
	Note that if the condition of the for loop is always true, then the program will enter what is called an “infinite loop” that leads to a crash. For example if the condition was “i>0” instead of “i<=10” then it will cause an infinite loop.

The loop counter does not have to be increasing, or change by 1. The following example prints the even numbers between 10 and -10. You start with a counter value equal to 10, and then change by -2 thereafter until the counter becomes smaller than -10.

```
for (int i = 10; i >= -10; i = i-2)
{
   Print( i.ToString() );
}
```

If you happen to have an array that you need to iterate through, then you can set your counter to loop through the indices of your array. Just remember that the indices of arrays are zero based and therefore you need to remember to loop from index=0 to the length of the array minus 1, or else you will get an out-of-bound error.

```
string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char[ ] letters = alphabet.ToCharArray();
for (int i = 0; i < letters.Length; i++)
{
  Print( i.ToString() + " = " + letters[i] );
}
```

Here is another example that iterates through a list of places.

```
//List of places
List< string > placesList = new List< string >();
placesList.Add( "Paris" );
placesList.Add( "NY" );
placesList.Add( "Beijing" );

int count = placesList.Count();
//Loop starting from 0 to count -1 (count = 3, but last index of the placesList is 2)
for (int i = 0; i < count; i++)
{
    Print( "I have been to " + placesList[i] );
}
```

Iterative Loops: foreach loop

You can use the ***foreach*** loop to iterate through the elements of an array or a list without using a counter or index. This is a less error prone way to avoid out of bound error. The above example can be rewritten as follows to use ***foreach*** instead of the ***for*** loop:

```
//List of places
List< string > placesList = new List< string >();
placesList.Add( "Paris" );
placesList.Add( "NY" );
placesList.Add( "Beijing" );

//Loop
foreach (string place in placesList)
{
    Print(place);
}
```

Conditional Loops: while loop

Conditional loops are ones that keep repeating until certain conditions are no longer true. You have to be very careful when you use conditional loops because you can be trapped looping infinitely until you crash when your condition continues to be true.

Some problems can be solved iteratively but others cannot. For example if you need to find the sum of 10 consecutive positive even integers starting with 2, then this can be solved iteratively with a ***for*** loop. Why? This is because you have a specified number of times to loop (10 in this case). This is how you might write your loop to solve your problem.

```
int sum = 0;
for (int i = 1; i <= 10; i++)
{
    sum = sum + ( i*2 );
}
Print( sum.ToString() );
```

On the other hand, if you need to add consecutive positive even integers starting with 2 until the sum exceeds 1000, then you cannot simply use an iterative ***for*** loop because you do not know how many times you will have to loop. You only have a condition to stop looping. Here is how you might solve this problem using the conditional ***while*** loop:

```
int sum = 0;
int counter = 0;
int number = 2;

//Loop
while (sum < 1000)
{
   sum = sum + number;
   //Make sure you increment the counter and the number
   counter = counter + 1;
   number = number + 2;
}

//Remove last number
sum -= number;
counter --;

Print("Count = " + counter);
Print("Sum = " + sum);
```

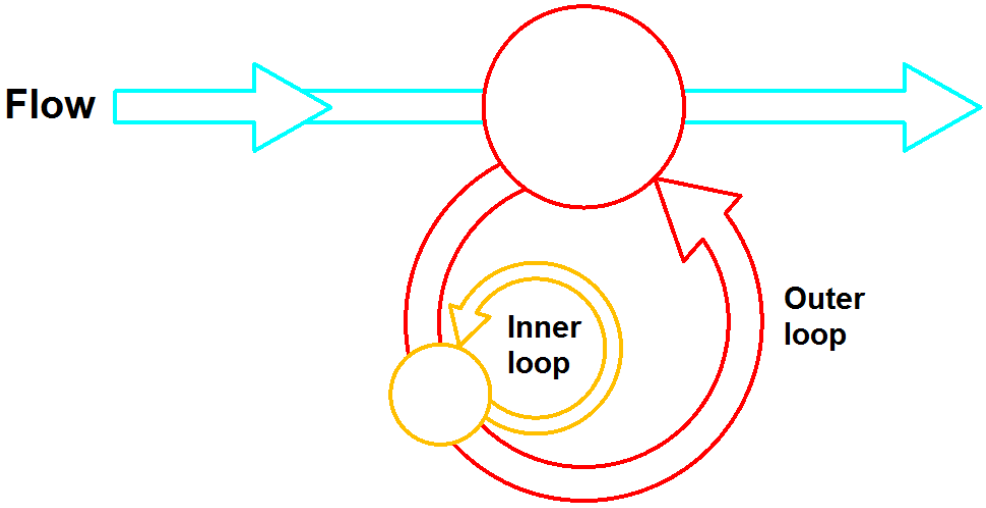
Here is another example from mathematics where you can solve the ***Collatz conjecture***. It basically says that you can start with any natural positive integer, and if even then divide by 2, but if odd, then multiply by 3 and add 1. If you repeat the process long enough, then you always converge to 1[5]. The following example prints all the numbers in a ***Collatz conjecture*** for a given number:

```
int x = 46;
while(x > 1)
{
  Print( x.ToString() );
  if( x % 2 == 0 )
    x = x / 2;
  else
    x = x * 3 + 1;
}
Print( x.ToString() );
```

[5] For details about the Collatz conjecture, check http://en.wikipedia.org/wiki/Collatz_conjecture

Nested loops

A nested loop is a loop that contains another loop inside it. You can nest as many loops as you need, but keep in mind that this can make the logic harder to follow.



Figure(19): Nested loops

For example, suppose you have the following list of words as input: {apple, orange, banana, strawberry} and you need to calculate the number of letters ***a***. You will have to do the following:

1- Loop through each one of the fruits. This is the outer loop.
2- For each fruit, loop through all the letters. This is the inner, or the nested, loop.
3- Whenever you find the letter "a" increment your counter by 1.
4- Print the counter.

```
//Declare and initialize the fruits
string[ ] fruits =  {"apple", "orange", "banana", "strawberry"};
int count = 0;

//Loop through the fruits
foreach (string fruit in fruits)
{
  //loop through the letters in each fruit
  foreach (char letter in fruit)
  {
    //Check if the letter is 'a"
    If (letter == 'a')     {
      count = count + 1;
    }
  }
}
Print( "Letter a is repeated " + count + " times" );
```

Using break and continue inside loops

The ***break*** statement is used to terminate or exit the loop while the ***continue*** statement helps skip one loop iteration. Here is an example to show the use of both. It checks if there is an orange in a list of fruits, and counts how many fruits have at least one letter r in their name.

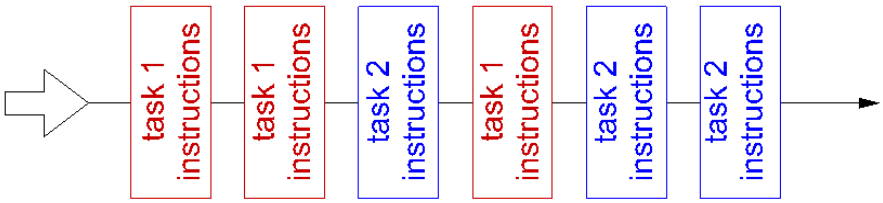
```
//Declare and initialize the fruits
string[ ] fruits =  {“apple”, “orange”, “banana”, “strawberry”};

//Check if there is at least one orange
foreach (string fruit in fruits)
{
  //Check if it is an orange
  if (fruit == "orange")
  {
    //Check if the letter is ‘a"
    Print( "Found an orange" );

    //No need to check the rest of the list, exit the loop
    break;
  }
}

//Check how many fruits has at least one letter ‘r’ in the name
int count = 0;
//Loop through the fruits
foreach (string fruit in fruits)
{
  //loop through the letters in each fruit
  foreach (char letter in fruit)
  {
    //Check if the letter is ‘r"
    if (letter == ‘r’)     {
      count = count + 1;

      //No need to check the rest of the letters, and skip to the next fruit
      continue;
    }
  }
}
Print( "Number of fruits that has at least one letter r is: " + count );
```

2_8: Methods

2_8_1: Overview

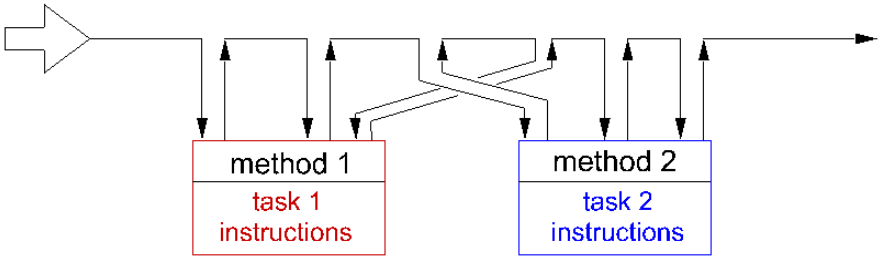
A method[6] is a self-contained entity that performs a specific task. The program can call it any number of times. Functions are basically used to organize your program into sub-tasks. The following is an example of the flow of a program that uses two distinct tasks repeatedly, but does not implement as a separate function, and hence has to retype at each encounter.

[6] The term “method” refers to functions associated with classes, but this guide may use them interchangeably.



Figure(20): Sequential flow of the script where tasks are repeated

Most programming languages allow you to organize your tasks into separate modules or what is called functions or methods. You can think of Grasshopper components each as a function. This allows you to write the instructions of each task once and give it an identifying name. Once you do that, anytime you need to perform the task in your program, you simply call the name of that task. Your program flow will look like the following:



Figure(21): Methods can be called and executed as many times as the script requires

Whenever you need to perform the same task multiple times in different places in your code, you should consider isolating it in a separate function. Here are some of the benefits you will get from using functions:

1- Break your program into manageable pieces. It is much easier to understand, develop, test and maintain smaller chunks of code, than one big chunk.
2- Write and test the code once, and reuse instead of writing it again. Also update in one place if a modification is required at any time.

Writing functions might not always be easy and it needs some planning. There are four general steps that you need to satisfy when writing a function:

1- Define the purpose of your function and give it a descriptive name.
2- Identify the data that comes into the function when it is called.
3- Identify the data that the function needs to hand back or return.
4- Consider what you need to include inside the function to perform your task.

For example, suppose you would like to develop a function that checks whether a given natural number is a prime number or not. Here are the four things you need to do:

1- **Purpose & name:** Check if a given number is prime. Find a descriptive name for your function, for example "***IsPrimeNumber***".
2- **Input parameters:** Pass your number as type integer.
3- **Return:** True if the number is prime, otherwise return false.
4- **Implementation**: Use a loop of integers between 2 and half the number. If the number is not divisible by any of those integers, then it must be a prime number. Note that there might be other more efficient ways to test for a prime number that can expedite the execution.

```
bool IsPrimeNumber( int x)
{
   bool primeFlag = true;

   for( int i = 2; i < (x / 2); i++)
   {
     if( x % i == 0)
     {
       primeFlag = false;
       break; //exit the loop
     }
   }
   return primeFlag;
}
```

Let us dissect the different parts of the ***IsPrimeNumber*** function to understand the different parts.

The function or method part	Description

bool IsPrimeNumber(…)
{
…
}
	The name of the function proceeded by the return type. The function body is enclosed inside the curly parenthesis.

(int x)	Function parameters and their types.

bool primeFlag = true;
 	
for(int i = 2; i < (x / 2); i++)
{
if(x % i == 0)
{
primeFlag = false;
break; //exit the loop
}
}
	This is called the *body* of the function. It includes a list of instructions to examine whether the given number is prime or not, and append all divisible numbers.

return primeFlag;	Return value. Use return keyword to indicate the value the function hands back.

2_8_2: Method parameters

Input parameters are enclosed within parentheses after the method name. The parameters are a list of variables and their types. They represent the data the function receives from the code that calls it. Each parameter is passed either ***by value*** or ***by reference***. Keep in mind that variables themselves can be a value-type such as primitive data (int, double, etc.), or reference-types such as lists and user-defined classes. The following table explains what it means to pass value or reference types by value or by reference.

Parameters	Description
Pass by value a **value-type** data such as int, double, bool	The caller passes a copy of the actual value of the parameter. Changes to the value of a variable inside the function will not affect the original variable passed from the calling part of the code.

Pass by reference a **value-type** data	Indicates that the address of the original variable is passed and any changes made to the value of that variable inside the function will be seen by the caller.
Pass by value a **reference-type** data such as lists, arrays, object, and all classes	Reference Type data holds the address of the data in the memory. Passing a Reference Type data by value simply copy the address, so changes inside the function will change original data. If the caller cares that ita data is not affected by the function, it should duplicate the data before passing it to any function.
Pass by reference a **reference-type** data	Indicates that the original reference of the variable is passed. Again, any changes made to the variable inside the function are also seen by the caller.

As you can see from the table, whether the parameter is passed as a copy (by value) or as a reference (by reference), is mostly relevant to ***value-type*** variables such as ***int***, ***double***, ***bool***, etc. If you need to pass a **reference-type** such as objects and lists, and you do not wish the function to change your original data (act as though the variable is passed by value), then you need to duplicate your data before passing it. All input values to the GH ***RunScript*** function (the main function in the scripting component) are duplicated before being used inside the component so that input data is not affected.

Let's expand the ***IsPrimeNumber*** function to return all the numbers that the input is divisible by. We can pass a list of numbers to be set inside the function (***List*** is a reference type). Notice it will not matter if you pass the list with the ***ref*** keyword or not. In both cases, the caller changes the original data inside the function.

```
private void RunScript(int num, ref object IsPrime, ref object Factors)
{
    //Assign variables to output
    List<int> factors = new List<int>();
    IsPrime= isPrimeNumber(num, factors);
    Factors = factors;
}
//Note: "List" is a reference type and hence you can pass it by value (without "ref" keyword)
//         and the caller still get any changes the function makes to the list
bool IsPrimeNumber( int num, ref List<int> factors )
{
    //all numbers are divisible by 1
    factors.Add(1);
    bool primeFlag = true;
    for( int i = 2; i < (num / 2); i++)
    {
      if( num % i == 0)
      {
        primeFlag = false;
        factors.Add( i); //append number
      }
    }
    //all numbers are divisible by the number itself
    factors.Add(num);

    return primeFlag;
}
```

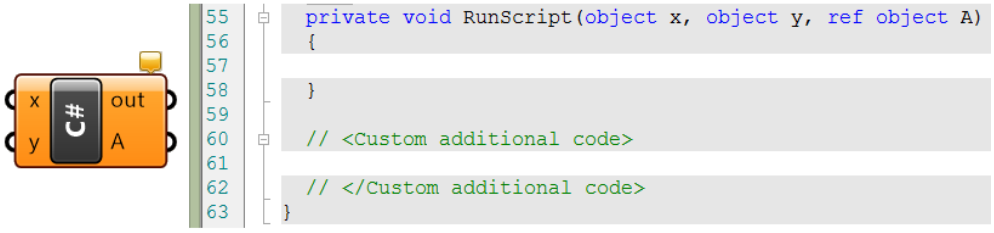
IsPrimeNumber/ isPrimeNumber there is no constant variable

Passing parameters ***by reference*** using keyword ***ref*** is very useful when you need to get more than one ***value-type*** back from your function. Since functions are allowed to return only one

value, programmers typically pass more parameters by reference as a way to return more values. Here is an example of a division function that returns success if the calculation is successful, but also returns the result of the division using the ***rc*** parameter that is passed ***by-reference***.

```
public bool Divide(double x, double y, ref double rc)
{
   if (Math.Abs(y) < 1e-100)
      return false;
   else
      rc = x / y;
   return true;
}
```

As we explained before, the ***RunScript*** is the main function that is available is the ***C#*** and other script components. This is where the user main code lives. Notice that the scripting component output is passed by reference. This is what you see when you open a default ***C#*** script component in Grasshopper.



Code part	Description

{
 	
}
	Curly parentheses enclose the function body of code.

RunScript	This is the name of the main function.
RunScript(…)	Parentheses after the function name enclose the input parameters. This is the list of variables passed to the function.
object x	*x* is the first input parameter. It is passed *by value* and its type is *object*. That means changes to "x" inside the ***RunScript*** does not change the original value.
object y	*y* is the second input parameter. It is passed *ByVal* and its type is *Object*.
ref object A	*A* is the third input parameter. It is passed *by reference* and its type is *object*. It is also an output because you can assign it a value inside your script and the change will be carried outside the ***RunScript*** function.
//Custom additional code	*Comment indicating where you should place your additional functions, if needed.*

2_9: User-defined data types

We mentioned above that there are built-in data types and come with and are supported by the programming language such as int, double, string and object. However, users can create their own custom types with custom functionality that suits the application. There are a few ways to create custom data types. We will explain the most common ones: enumerations, structures and classes.

2_9_1: Enumerations

Enumerations help make the code more readable. An enumeration "provides an efficient way to define a set of named integral constants that may be assigned to a variable"[7]. You can use enumerations to group a family of options under one category and use descriptive names. For example, there are only three values in a traffic light signal.

```
enum Traffic
{
  Red = 1,
  Yellow = 2,
  Green = 3
}
Traffic signal = Traffic.Yellow;
if( signal == Traffic.Red)
    Print("STOP");
else if(signal == Traffic.Yellow)
    Print("YIELD");
else if(signal == Traffic.Green)
    Print("GO");
else
    Print("This is not a traffic signal color!");
```

2_9_2: Structures

A structure is used to define a new ***value-type***. In C# programming , we use the keyword ***struct*** to define new structure. The following is an example of a simple structure that defines a number of variables (fields) to create a custom type of a colored 3D point. We use ***private*** access to the fields, and use ***properties*** to ***get*** and ***set*** the fields.

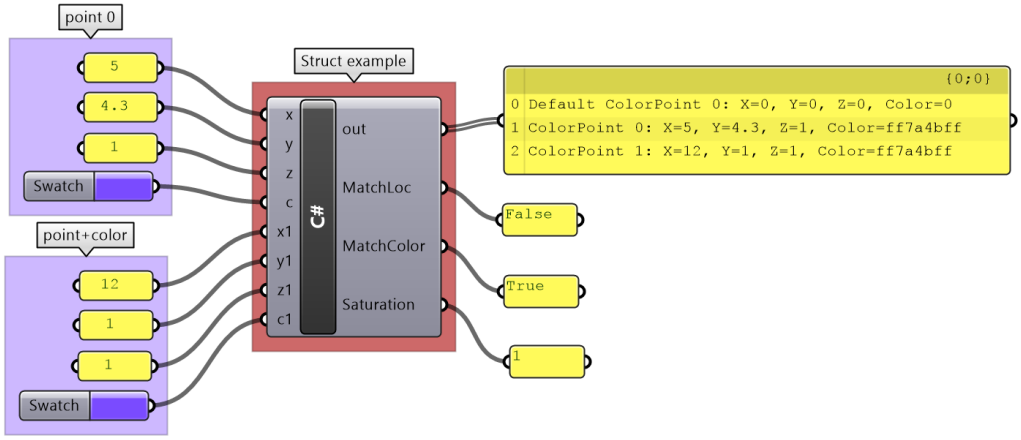
```
struct ColorPoint{
   //fields for the point XYZ location and color
   private double _x;
   private double _y;
   private double _z;
   private System.Drawing.Color _c;
   //properties to get and set the location and color
   public double X { get { return _x; } set { _x = value; } }
   public double Y { get { return _y; } set { _y = value; } }
   public double Z { get { return _z; } set { _z = value; } }
   public System.Drawing.Color C { get { return _c; } set { _c = value; } }
}
```

[7] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/enumeration-types

As an example, you might have two instances of the ***ColorPoint*** type, and you need to compare their location and color. Notice that when you instantiate a new instance of ***ColorPoint*** object, the object uses a default constructor that sets all fields to “0”:

```
ColorPoint cp0 = new ColorPoint();
//Using default constructor sets the fields to zero
Print("Default ColorPoint 0: X=" + cp0.X + ", Y=" + cp0.Y + ", Z=" + cp0.Z + ", Color=" + cp0.C.Name);
//set fields
cp0.X = x;
cp0.Y = y;
cp0.Z = z;
cp0.C = c;
Print("ColorPoint 0: X=" + cp0.X + ", Y=" + cp0.Y + ", Z=" + cp0.Z + ", Color=" + cp0.C.Name);

ColorPoint cp1 = new ColorPoint();
//set fields
cp1.X = x1;
cp1.Y = y1;
cp1.Z = z1;
cp1.C = c1;
Print("ColorPoint 1: X=" + cp1.X + ", Y=" + cp1.Y + ", Z=" + cp1.Z + ", Color=" + cp1.C.Name);

//compare location
MatchLoc = false;
if(cp0.X == cp1.X && cp0.Y == cp1.Y && cp0.Z == cp1.Z)
  MatchLoc = true;

//compare color
MatchColor = cp0.C.Equals(cp1.C);
```

This is the output you get when implementing the above struct and function in a C# component:



Structs typically define one or more constructors to allow setting the fields. Here is how we expand the example above to include a constructor.

```
struct ColorPoint{
  //fields
  private double _x;
  private double _y;
  private double _z;
  private System.Drawing.Color _c;
```

```
    //constructor
    public ColorPoint(double x, double y, double z, System.Drawing.Color c)
    {
      _x = x;
      _y = y;
      _z = z;
      _c = c;
    }
    //properties
    public double X { get { return _x; } set { _x = value; } }
    public double Y { get { return _y; } set { _y = value; } }
    public double Z { get { return _z; } set { _z = value; } }
    public System.Drawing.Color C { get { return _c; } set { _c = value; } }
}
```

You can use properties to only set or get data. For example you can write a property that get the color saturation:

```
    //property
    public double Saturation { get { return _c.GetSaturation(); } }
```

We can rewrite the ***Saturation*** property as a method as in the following:

```
    //method
    public double Saturation() { return _c.GetSaturation(); }
```

However, methods typically include more complex functionality, multiple input or possible exceptions. There are no fixed rules about when to use either, but it is generally acceptable that properties involve data, while methods involve actions. We can write a method to calculate the average location and color of two input ColorPoints.

```
    //method
    public static Average (ColorPoint a, ColorPoint b)
    {
      ColorPoint avPt = new ColorPoint();
      avPt.X = (a.X + b.X) / 2;
      avPt.Y = (a.Y + b.Y) / 2;
      avPt.Z = (a.Z + b.Z) / 2;
      avPt.C = Color.FromArgb((a.C.A + b.C.A) / 2, (a.C.R + b.C.R) / 2, (a.C.G + b.C.G) / 2, (a.C.B + b.C.B) / 2);
      return avPt;
    }
```

Structures can include members and methods that can be called even if there is no instance created of that type. Those use the keyword ***static***. For example, we can add a member called OriginBlack that creates a black point located at the origin. We can also include a ***static*** method to compare if two existing points have the same color, as in the following.

```
    //static methods
    public static bool IsEqualLocation(ColorPoint p1, ColorPoint p2)
    {
      return (p1.X == p2.X && p1.Y == p2.Y && p1.Z == p2.Z);
    }
```

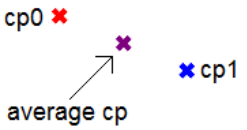
```
public static bool IsEqualColor(ColorPoint p1, ColorPoint p2)
{
  return p1.C.Equals(p2.C);
}
```

Because structs are value types, if you pass your colored points to a function, and change the location (x,y,z), it will only be changed inside or within the scope of that function, but will not affect the original data, unless explicitly passed by reference as in the following example:

```
void ChangeLocationByValue(ColorPoint cp)
{
  cp.X += 10;
  cp.Y += 10;
  cp.Z += 10;
  Print(" Inside change location= (" + cp.X + "," + cp.Y + "," + cp.Z + ")");
}
void ChangeLocationByReference(ref ColorPoint cp)
{
  cp.X += 10;
  cp.Y += 10;
  cp.Z += 10;
  Print(" Inside change location= (" + cp.X + "," + cp.Y + "," + cp.Z + ")");
}
```

Using our ***ColorPoint*** struct, the following is a program that generates 2 colored points, compares their location and color, then finds their average:

```
//create 2 instances of ColorPoints type
ColorPoint cp0 = new ColorPoint(1, 1, 1, System.Drawing.Color.Red);
ColorPoint cp1 = new ColorPoint(1, 1, 1, System.Drawing.Color.Blue);
//compare location
bool matchLoc = ColorPoint.isEqualLocation(cp0, cp1);
//compare color
Bool matchColor = ColorPoint.isEqualColor(cp0, cp1);
//Output average point and color
ColorPoint avPt = ColorPoint.Average(cp0, cp1);
```



This should be lowercase

Structures have limitations, most notable is the inability to build a hierarchy of objects. Also, they are not preferred for objects that are large in their size in bytes.

2_9_3: Classes

Classes help create new data types that are ***reference-type***. They also have added functionality compared to structures. The main one is that they support creating a hierarchy of types where each new level inherits the members and methods of the level above it. Many of the geometry types in ***RhinoCommon*** use classes to define them. Let us redefine the ***ColorPoint*** above as a class that inherits from a generic ***Point*** class.

```
class CPoint{
  //fields
  private double _x;
```

```
    private double _y;
    private double _z;
    //constructors
    public CPoint() : this(0,0,0) //default calls another constructor
    {
    }
    public CPoint(double x, double y, double z)
    {
      _x = x;
      _y = y;
      _z = z;
    }
    //properties
    public double X { get { return _x; } set { _x = value; } }
    public double Y { get { return _y; } set { _y = value; } }
    public double Z { get { return _z; } set { _z = value; } }
    //static methods
    public static bool IsEqualLocation(CColorPoint p1, CColorPoint p2)
    {
      return (p1.X == p2.X && p1.Y == p2.Y && p1.Z == p2.Z);
    }
}

class CColorPoint: CPoint{
    //fields
    private System.Drawing.Color _c;

    //constructors
    public CColorPoint() : base(),default
    {
      _c = System.Drawing.Color.White;
    }
    public CColorPoint(double x, double y, double z, System.Drawing.Color c) : base(x, y, z)
    {
      _c = c;
    }
    //properties
    public System.Drawing.Color C { get { return _c; } set { _c = value; } }
    //static methods
    public static bool IsEqualColor(CColorPoint p1, CColorPoint p2)
    {
      return p1.C.Equals(p2.C);
    }
    //method
    public static CColorPoint Average (CColorPoint a, CColorPoint b)
    {
      CColorPoint avPt = new CColorPoint();
      avPt.X = (a.X + b.X) / 2;
      avPt.Y = (a.Y + b.Y) / 2;
      avPt.Z = (a.Z + b.Z) / 2;
      avPt.C = Color.FromArgb((a.C.A + b.C.A) / 2, (a.C.R + b.C.R) / 2, (a.C.G + b.C.G) / 2, (a.C.B + b.C.B) / 2);
      return avPt;
    }
}
```

The comma should be removed

System.Drawing before Color

You can use the same program written above for ***ColorPoint struct*** to create an average point between 2 color points. The only difference is that if you now pass an instance of the class ***CColorPoint*** to a function, it will be passed by reference, even without using the ***ref*** keyword. Here are the key differences between structures and classes:

Struct	Class	What does it mean?
Value type	Reference type	Value types (struct) are cheaper to allocate and deallocate in memory, however the assignment of a large value type can be more costly than a reference type (class).
Passed to functions by value	Passed to functions by reference	When you pass a parameter to a function as an instance of a reference type, changes inside the function affect all references pointing to that same instance. Value types are copied when passed by value.
Cannot use inheritance	Can use inheritance	Use classes when building a hierarchy of objects or when the data type is large, and use structs when creating types that are generally small, short lived and embedded inside other objects.

2_9_4: Value vs reference types

It is worth stressing the two data classifications: ***value-types*** and ***reference-types***. We touched on that topic when introducing methods and how parameters are passed by value or by reference. There are also differences in how the two are stored and managed in memory. Here is a summary comparison between the two classifications:

	Value data types	Reference data types
Examples	All built-in numeric data types (such as Integer, double, bool, and char), Arrays, Structures.	Lists Classes
Memory storage	Stored inline in the program stack.	Stored in the heap.
Memory management	Cheaper to create and clear from memory especially for small data.	Costly to clear (use garbage collectors), but more efficient for big data.
When passed as parameters in methods	Passes a copy of the data to methods, which means that the original data is not changed even when altered inside the method.	Passes the address of the original data, and hence any changes to the data inside the method changes the original data as well.

2_9_5: Interface

You may need your structures and classes to implement some common functionality. For example, you may need to check if two instances of the same type are equal. In this case, you would like to make sure that you use the same method name and signature. To ensure that, you can define that functionality inside a separate entity called ***interface***, and have your structure and classes implement that same ***interface***.

The .NET Framework provides interfaces, but you can also define your own. For example, we can implement an ***IEquatable*** interface in the ***ColorPoint struct*** as in the following.

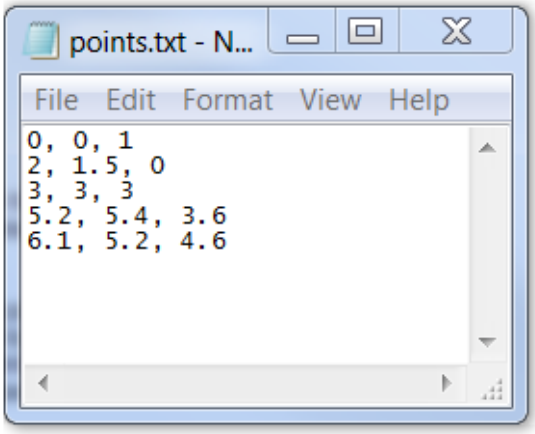
```
struct ColorPoint: IEquatable<ColorPoint>
{
   //fields
   private double _x;
   private double _y;
   private double _z;
   private System.Drawing.Color _c;
   //properties
   public double X { get { return _x; } set { _x = value; } }
   public double Y { get { return _y; } set { _y = value; } }
   public double Z { get { return _z; } set { _z = value; } }
   public System.Drawing.Color C { get { return _c; } set { _c = value; } }
   //implement Equals method inside IEquatable interface - if omitted, you'll get error
   public bool Equals( ColorPoint pt )
   {
      return (this._x == pt._x && this._y == pt._y && this._z == pt._z && this._c == pt._c);
   }
}
```

2_10: Read and write text files

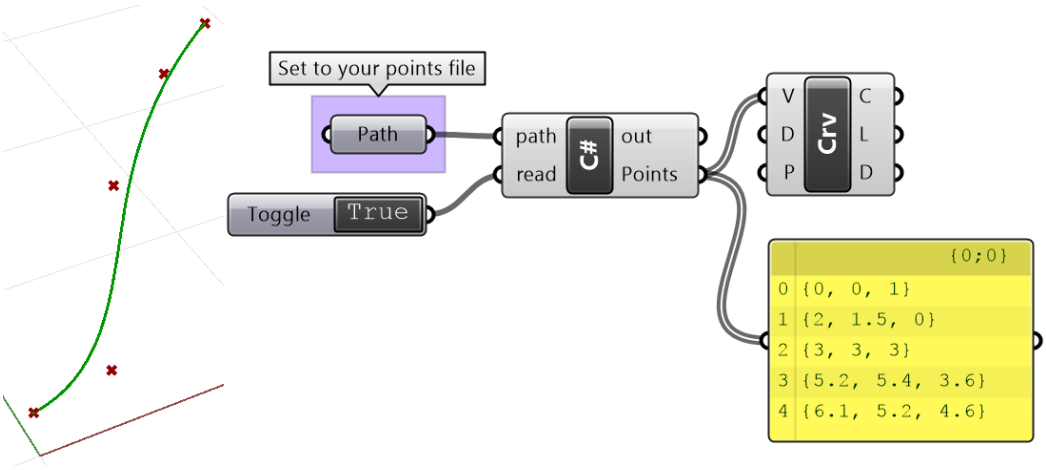
There are many ways to read from and write to files in **C#** and many tutorials and documents are available online. In general, reading a file involves the following:

- Open the file. Generally you need a path to point to.
- Read a string (whole text or line by line).
- Tokenize the string using some delimiter characters (such as a comma, space, etc.).
- Cast each token to the appropriate data type.
- Store the result in your data structure.

We will discuss a simple example that parses points from a text file. Using the following text file format, we will read each line as a point and use the first value as x coordinate, the second as y and the third as z of the point. We will then use these points to create a NURBS curve.



The input to the scripting component is the path to the text file, and the output is an array of points. You can then use these points as input to create a curve in Grasshopper.



Here is the code inside the script component. It reads the file line by line, then parses each into a point assuming that each point is stored in one line and the coordinates are comma separated.

References: using System.IO

```
//Read the file
string[] lines = File.ReadAllLines(path);

//Declare list of points (Point3d is a data type in RhinoCommon SDK)
List<Point3d> pts = new List<Point3d>();

//Loop through lines
foreach (string line in lines)
{
    //Tokenize line into array of strings separated by ","
    string[] parts = line.Split(",".ToCharArray());

    //Convert each coordinate from string to double
    double x = Convert.ToDouble(parts[0]);
    double y = Convert.ToDouble(parts[1]);
    double z = Convert.ToDouble(parts[2]);

    pts.Add(new Point3d(x, y, z));
}
Points = pts;
```

The above code does not do any validation of the data and will only work if the text file has no errors such as empty lines, invalid numbers, or simply does not follow the expected format. Validating the data before using it is a good practice that helps make your program robust and avoid crashes and errors.The following is a rewrite of the same code, but with validation (highlighted).

The code works better without this part

```
if ((!File.Exists(path)) || !read) {
    //Show message box
    System.Windows.Forms.MessageBox.Show("File does not exist or could not be read");
    //Give feedback in the component out parameter
    Print("Exit without reading");
```

```
    return;
}

//Read the file
string[] lines = File.ReadAllLines(path);

//Check that file is not empty
if ((lines == null))
{
    Print("File has no content. Exit without reading");
    return;
}

//Declare list of points (Point3d is a data type in RhinoCommon SDK)
List<Point3d> pts = new List<Point3d>();

//Characters to remove
var charsToRemove = new string[] { " ", ")", "(", "[", "]", "{", "}" };

//Loop through lines
foreach (string line in lines)
{
    //Trim invalid char
    var tLine = line;
    foreach (var c in charsToRemove)
    {
        tLine = tLine.Replace(c, string.Empty);
    }

    if (String.IsNullOrEmpty(line)) continue;
    //Tokenize line into array of strings separated by ","
    string[] parts = tLine.Split(",".ToCharArray());

    //Make sure that each line has exactly 3 values
    if (parts.Length != 3)
        continue;

    //Convert each coordinate from string to double
    double x = Convert.ToDouble(parts[0]);
    double y = Convert.ToDouble(parts[1]);
    double z = Convert.ToDouble(parts[2]);

    pts.Add(new Point3d(x, y, z));
}
A = pts;
```

2_11: Recursive functions

Recursive functions are functions that call themselves until a stopping condition is met. Recursion is not very common to use, but is useful in data search, subdividing and generative systems.

Just like conditional loops, recursive functions need to be designed and tested carefully for all possible input; otherwise you run the risk of crashing your code.

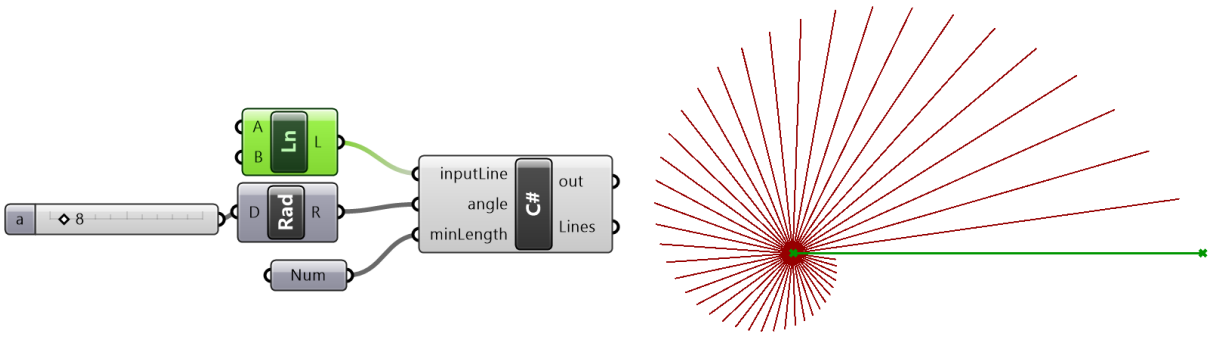
The following example takes an input line then makes a copy that is shorter and rotated. It keeps doing that for each newly generated line until the line length becomes less than some minimum length. The following table shows the input and output parameters.

Function parameters		
Input parameters	**Line** inputLine	Starting line
	double angle	Angle in radians
	double minLength	The stopping condition
Output parameters	**Line** [] A	The result as an array of lines

To compare the two approaches, wWe will solve the problem recursively and iteratively ~~to be able to compare the two approaches~~. Note that not all recursive cases can be solved iteratively. Also some cases are easier to solve recursively that iteratively or vice versa. If you do not design your recursive function carefully, you can end up in an infinite loop.

In the following recursive solution, note that inside the ***DivideAndRotate*** function includes:

- A stopping condition to exit the function.
- A call to the same function from within the function (recursive function call themselves).
- The result (array of lines) is passed by reference to keep adding new lines to the list.

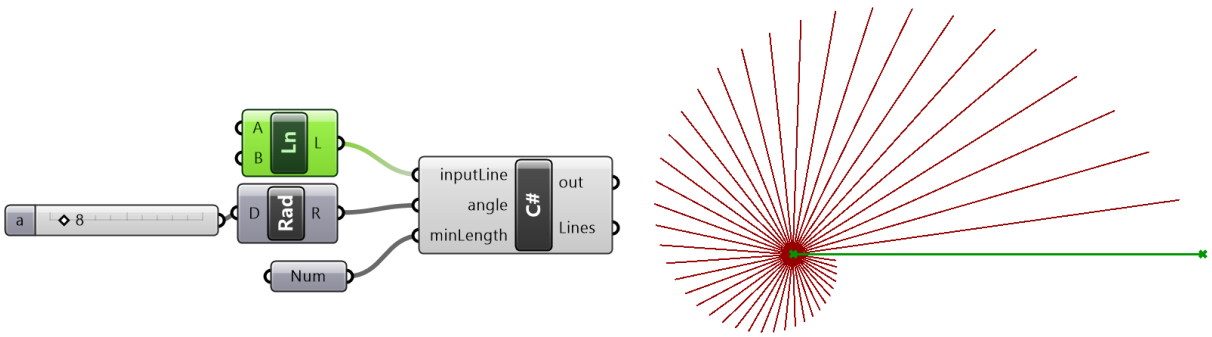



```
private void RunScript(Line inputLine, double angle, double minLength, ref object Lines)
{
    //Declare all lines
    List<Line> allLines = new List<Line>();

    //Append input line as a first line in the list of lines
    allLines.Add(inputLine);

    //Call recursive function
    DivideAndRotate(inputLine, ref allLines, angle, minLength);
```

```
  //Assign return value
  Lines = allLines;
}
public void DivideAndRotate(Line currLine, ref List<Line> allLines, double angle, double minLength)
{
  //Check the stopping condition
  if (currLine.Length < minLength)
    return;

  //Take a portion of the line
  Line newLine = new Line();
  newLine = currLine;

  Point3d endPt = new Point3d();
  endPt = newLine.PointAt(0.95);

  newLine.To = endPt;

  //Rotate
  Transform xform = default(Transform);
  xform = Transform.Rotation(angle, Vector3d.ZAxis, newLine.From);
  newLine.Transform(xform);

  allLines.Add(newLine);

  //Call self
  DivideAndRotate(newLine, re  ines, angle, minLength);
}
```

We can solve the same problem using an iterative solution through a conditional loop. Note that:

- The stopping condition appears in the ***while*** loop condition.
- The ***DivideAndRotate*** function parameters are passed by value (not changed inside the function). The function simply calculates and returns the new line.




```
private void RunScript(Line inputLine, double angle, double minLength, ref object Lines)
{
  //Declare all lines
  var allLines = new List<Line>();
  //Find current length
  double length = inputLine.Length;
  Line newLine = default(Line);
  newLine = inputLine;
  //Loop until length is less than min length
```

```
    while (length > minLength) {
      //Generate the new line
      newLine = DivideAndRotate(newLine, angle);
      //Add to list
      allLines.Add(newLine);
      //Stopping condition
      length = newLine.Length;
    }
    //Assign return value
    Lines = allLines;
```

```
}
public Line DivideAndRotate(Line currLine, double angle)
{
    //Take a portion of the line
    Line newLine = new Line();
    newLine = currLine;
    Point3d endPt = new Point3d();
    endPt = newLine.PointAt(0.95);
    newLine.To = endPt;
    //Rotate
    newLine.Transform(Transform.Rotation(angle, Vector3d.ZAxis, newLine.From));
    //Function return
    return newLine;
}
```

Chapter Three: *RhinoCommon* Geometry

3_1: Overview

RhinoCommon is the ***.NET SDK*** for Rhino. It is used by Rhino plug-in developers to write ***.NET*** plug-ins for Rhino and Grasshopper. All Grasshopper scripting components can access ***RhinoCommon*** including all geometry objects and functions. The full ***RhinoCommon*** documentation is available here:
https://developer.rhino3d.com/api/RhinoCommon

In this chapter, we will focus on the part of the ***SDK*** dealing with Rhino geometry. We will show examples of how to create and manipulate geometry using the Grasshopper C# component.

The use of the geometry classes in the ***SDK*** requires basic knowledge in vector mathematics, transformations and NURBS geometry. If you need to review or refresh your knowledge in these topics, then refer to the "*Essential Mathematics for Computational Design*", a free download is available from here:
https://www.rhino3d.com/download/rhino/6/essentialmathematics

If you recall from Chapter 2, we worked with value types such as ***int*** and ***double***. Those are system built-in types provided by the programming language, ***C#*** in this case. We also learned that you can pass the value types to a function without changing the original variables (unless passed by reference using the ***ref*** keyword). We also learned that some types such as ***objects***, are always passed by reference. That means changes inside the function also changes the original value.

The system built-in types whether they are value or reference types, are often very limiting in specialized programming applications. For example, in computer graphics, we commonly deal with points, lines, curves or matrices. These types need to be defined by the ***SDK*** to ease the creation, storage and manipulation of geometry data. Programming languages offer the ability to define new types using ***structures*** (value types) and ***classes*** (reference types). The ***RhinoCommon SDK*** defines many new types as we will see in this chapter.

3_2: Geometry structures

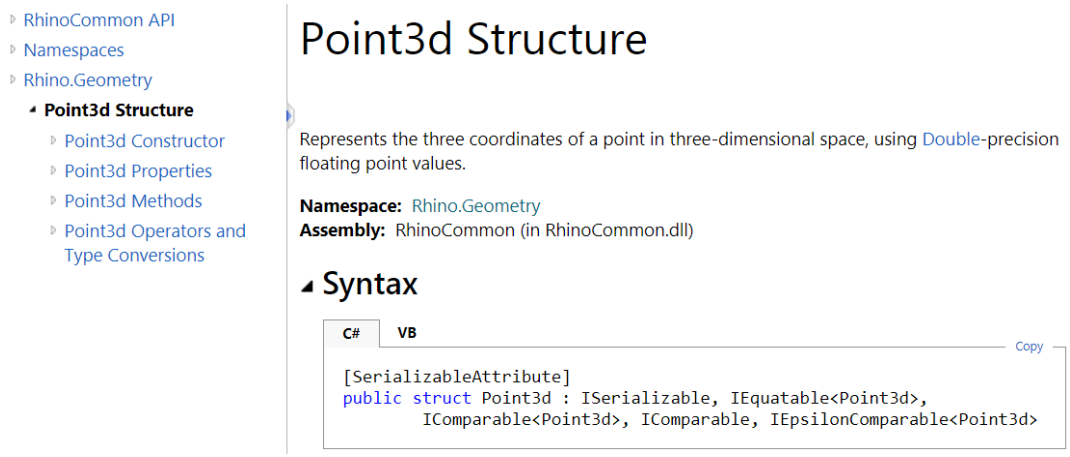
RhinoCommon defines basic geometry types using structures. We will dissect the ***Point3d*** structure and show how to read in the documentation and use it in a script. This should help you navigate and use other structures. Below is a list of the geometry structures.

Structures	Summary description
Point3d	Location in 3D space. There are other points that have different dimensions such as: **Point2d** (parameter space point) and **Point4d** to represent control points.
Vector3d	Vector in 3D space. There is also Vector2d for vectors in parameter space
Interval	Domain. Has min and max numbers
Line	A line defined by two points (from-to)

Plane	A plane defined by a plane origin, X-Axis, Y-Axis and Z-Axis
Arc	Represents the value of a plane, two angles and a radius in a subcurve of a circle
Circle	Defined by a plane and radius
Ellipse	Defined by a plane and 2 radii
Rectangle3d	Represents the values of a plane and two intervals that form an oriented rectangle
Cone	Represents the center plane, radius and height values in a right circular cone.
Cylinder	Represents the values of a plane, a radius and two heights -on top and beneath- that define a right circular cylinder.
BoundingBox	Represents the value of two points in a bounding box defined by the two extreme corner points.This box is therefore aligned to the world X, Y and Z axes.
Box	Represents the value of a plane and three intervals in an orthogonal, oriented box that is not necessarily parallel to the world Y, X, Z axes.
Sphere	Represents the plane and radius values of a sphere.
Torus	Represents the value of a plane and two radii in a torus that is oriented in 3D space.
Transform	4x4 matrix of numbers to represent geometric transformation

3_2_1 The Point3d structure

The ***Point3d***[8] type includes three ***fields*** (X, Y and Z). It defines a number of ***properties*** and also has ***constructors*** and ***methods***. We will walk through all the different parts of ***Point3d*** and how it is listed in the ***RhinoCommon*** documentation. First, you can navigate to ***Point3d*** from the left menu under the ***Rhino.Geometry*** namespace. When you click on it, the full documentation appears on the right. At the very top, you will see the following:



Here is a break down of what each part in the above ***Point3d*** documentation means:

[8] Point3d documentation:
https://developer.rhino3d.com/api/RhinoCommon/html/T_Rhino_Geometry_Point3d.htm

Part	Description
Point3D Structure Represents the ...	Title and description
Namespace: Rhino.Geometry	The namespace that contains ***Point3d***[9]
Assembly: RhinoCommon (in RhinoCommon.dll)	The assembly that includes that type. All geometry types are part of the RhinoCommon.dll
[SerializableAttribute]	Allow serializing the object[10]
public struct Point3d	**public**: public access: your program can instantiate an object of that type **struct**: structure value type. **Point3d**: name of your structure
: ISerializable, IEquatable<Point3d>, IComparable<Point3d>, IComparable, IEpsilonComparable<Point3d>	The **":"** is used after the struct name to indicate what the struct implements. Structures can implement any number of ***interfaces***. An ***interface*** contains a common functionality that a structure or a class can implement. It helps with using consistent names to perform similar functionality across different types. For example **IEquatable** interface has a method called "**Equal**". If a structure implements **IEquatable**, it must define what it does (in ***Point3d***, it compares all X, Y and Z values and returns true or false).[11]

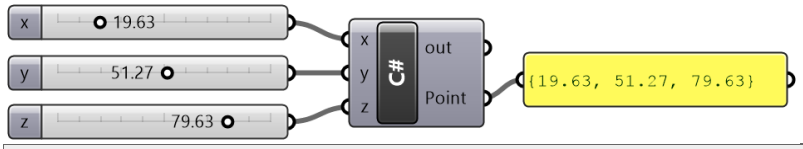
Point3d Constructors:
Structures define constructors to instantiate the data. One of the ***Point3d*** constructors takes three numbers to initialize the values of X, Y and Z. Here are all the constructors of the ***Point3d*** structure .

Constructors

	Name	Description
	Point3d(Point3d)	Initializes a new point by copying coordinates from another point.
	Point3d(Point3f)	Initializes a new point by copying coordinates from a single-precision point.
	Point3d(Point4d)	Initializes a new point by copying coordinates from a four-dimensional point. The first three coordinates are divided by the last one. If the W (fourth) dimension of the input point is zero, then it will be just discarded.
	Point3d(Vector3d)	Initializes a new point by copying coordinates from the components of a vector.
	Point3d(Double, Double, Double)	Initializes a new point by defining the X, Y and Z coordinates.

The following example shows how to define a variable of type ***Point3d*** using a GH C# component.

[9] Namespace in .NET: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/namespaces/
[10] For details, refer to Microsoft documentation: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/
[11] Interfaces: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/index



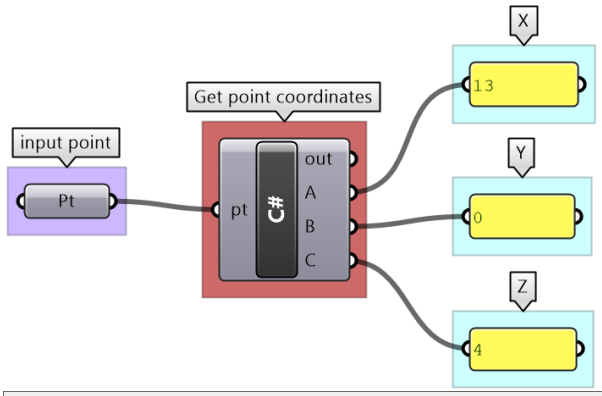

```
private void RunScript(double x, double y, double z, ref object Point)
{
  //Create an instance of a point and initialize to x, y and z
  Point3d pt = new Point3d(x, y, z);

  //Assign the point "pt" to output
  Point = pt;
}
```

Point3d Properties:
Properties are mostly used to “get” and/or “set” the fields of the structure. For example, there are the “X”, “Y” and “Z” properties to get and set the coordinates of an instance of ***Point3d***. Properties can be ***static*** to get specific points, such as the origin of the coordinate system (0,0,0). The following are ***Point3d*** properties as they appear in the documentation:

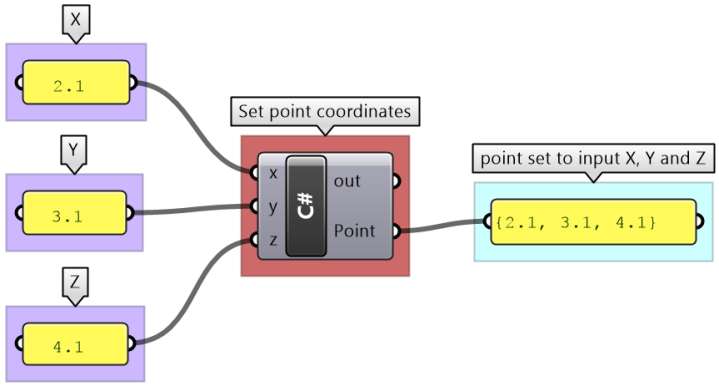
Properties

	Name	Description
	IsValid	Each coordinate of the point must pass the IsValidDouble(Double) test.
	Item	Gets or sets an indexed coordinate of this point.
	MaximumCoordinate	Gets the largest (both positive and negative) valid coordinate in this point, or RhinoMath.UnsetValue if no coordinate is valid.
	MinimumCoordinate	Gets the smallest (both positive and negative) coordinate value in this point.
S	Origin	Gets the value of a point at location 0,0,0.
S	Unset	Gets the value of a point at location RhinoMath.UnsetValue,RhinoMath.UnsetValue,RhinoMath.UnsetValue.
	X	Gets or sets the X (first) coordinate of this point.
	Y	Gets or sets the Y (second) coordinate of this point.
	Z	Gets or sets the Z (third) coordinate of this point.

Here are two GH examples to show how to get and set the coordinates of a ***Point3d***.




```
private void RunScript(Point3d pt, ref object A, ref object B, ref object C)
{
    //Assign the point coordinates to output
    A = pt.X;
    B = pt.Y;
    C = pt.Z;
}
```

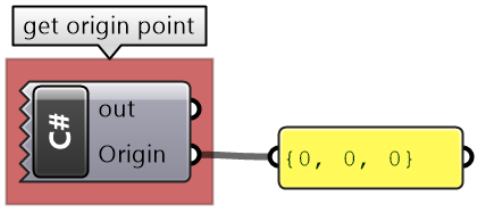



```
private void RunScript(double x, double y, double z, ref object Point)
{
    //Declare a new point
    Point3d newPoint = new Point3d(Point3d.Unset);

    //Set "new_pt" coordinates
    newPoint.X = x;
    newPoint.Y = y;
    newPoint.Z = z;

    //Assign the point coordinates to output
    Point = newPoint;
}
```

newPoint

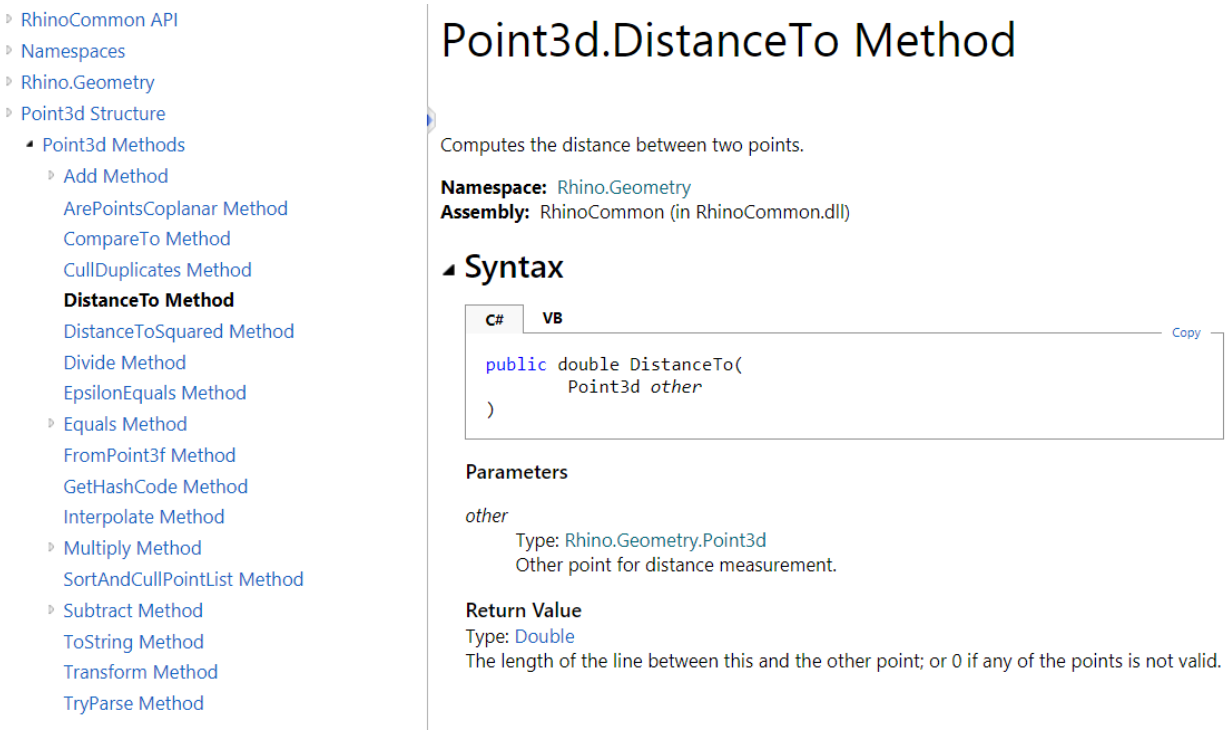
Static properties get or set a generic data of that type. The above example uses the static ***Point3d*** method ***Unset*** to unset the new point. Another example that is commonly used is the

origin property in ***Point3d*** to get the origin of the coordinate system, which is (0,0,0) as in the following example.

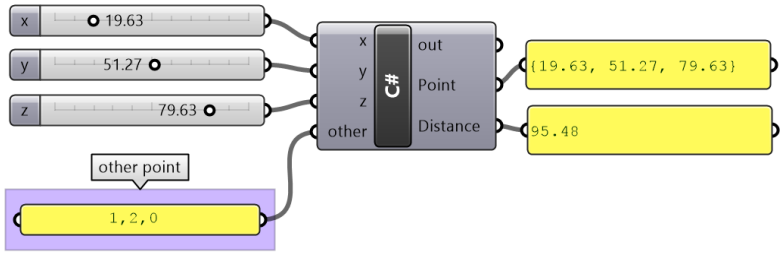



```
private void RunScript( ref object Origin)
{
  Origin = Point3d.Origin;
}
```

Point3d Methods:
The methods are used to help inquire about the data or perform specific calculations, comparisons, etc. For example, ***Point3d*** has a method to measure the distance to another point. All methods are listed in the documentation with full details about the parameters and the return value, and sometimes an example to show how they are used. In the documentation, ***Parameters*** refer to the input passed to the method and the ***Return Value*** describes the data returned to the caller. Notice that all methods can also be navigated through the left menu.



Here is how the ***DistanceTo*** method is used in an example.




```
private void RunScript( double x, double y, double z, Point3d other, ref object Point, ref object Distance)
{
    //Create an instance of a point and initialize to x, y and z
    Point3d pt = new Point3d(x, y, z);
    //Calculate the distance from "pt" to the "other" input point
    double dis = pt.DistanceTo(other);
    //Assign the point "point" to the A output
    Point = pt;
    //Assign the distance to the B output
    Distance = dis;
}
```

The ***DistanceTo*** method does not change the fields (the X, Y and Z). However, other methods such as ***Transform*** change the fields. The ***Transform*** method resets the X, Y and Z values to reflect the new location after applying the transform.

- RhinoCommon API
- Namespaces
- Rhino.Geometry
- Point3d Structure
  - Point3d Methods
    - Add Method
    - ArePointsCoplanar Method
    - CompareTo Method
    - CullDuplicates Method
    - DistanceTo Method
    - DistanceToSquared Method
    - Divide Method
    - EpsilonEquals Method
    - Equals Method
    - FromPoint3f Method
    - GetHashCode Method
    - Interpolate Method
    - Multiply Method
    - SortAndCullPointList Method
    - Subtract Method
    - ToString Method
    - **Transform Method**
    - TryParse Method

Point3d.Transform Method

Transforms the present point in place. The transformation matrix acts on the left of the point. i.e., result = transformation*point

Namespace: Rhino.Geometry
Assembly: RhinoCommon (in RhinoCommon.dll)

Syntax

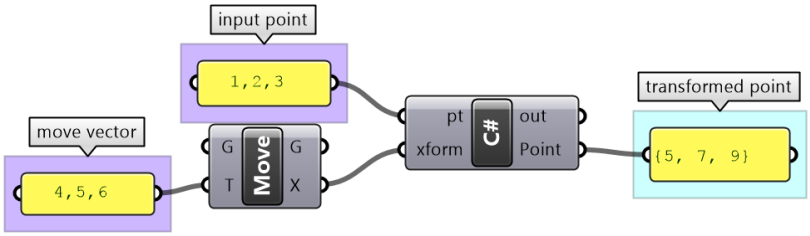
C# VB Copy

```
public void Transform(
        Transform xform
)
```

Parameters

xform
Type: Rhino.Geometry.Transform
Transformation to apply.

Here is an example that uses ***Point3d.Transform***.

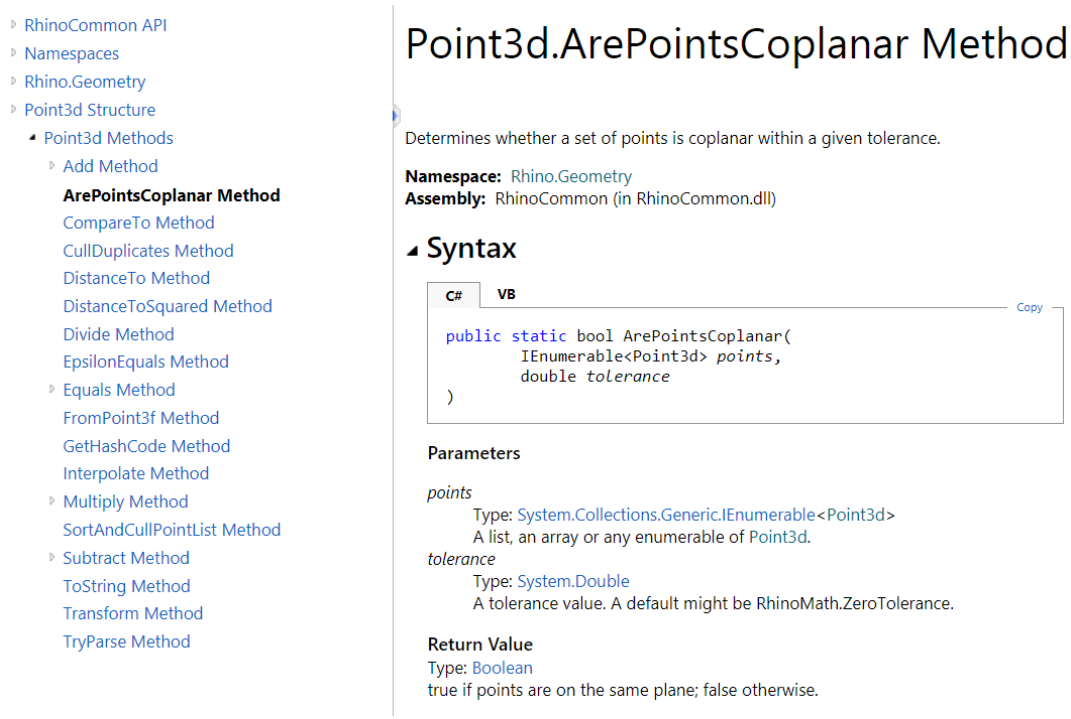



```
private void RunScript(Point3d pt, Transform xform, ref object Point)
{
    //Create an instance of a point and initialize to input point
    Point3d newPoint = new Point3d(pt);
    //Transform the point
    newPoint.Transform(xform);
    //Assign the point "new_pt" to the A output
    Point = newPoint;
}
```

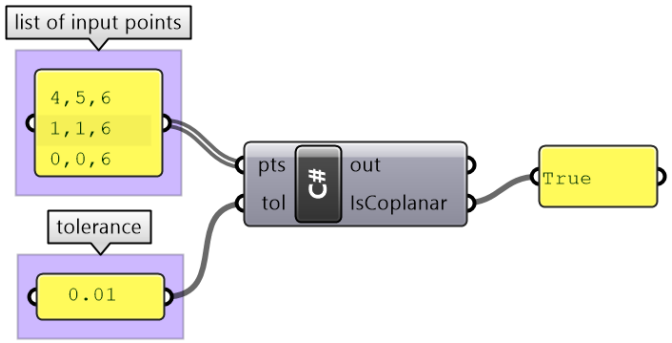
Point3d static methods:
Point3d has ***static*** methods that are accessible without instantiating an instance of ***Point3d***. For example, if you would like to check if a list of given instances of points are all coplanar, you can call the static method ***Point3d.ArePointsCoplanar*** without creating an instance of a point. Static methods have the little red “s” symbol in front of them in the documentation.

S	ArePointsCoplanar	Determines whether a set of points is coplanar within a given tolerance.

The ***ArePointsCoplanar*** has the following syntax, parameters and the return value.



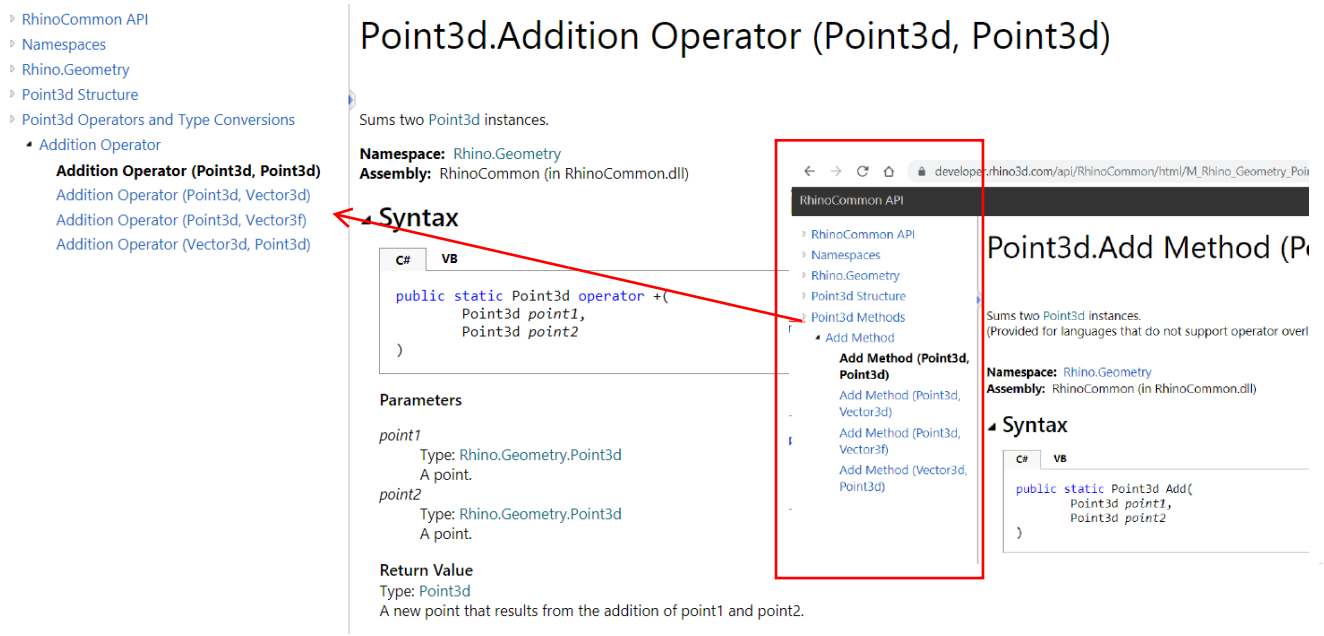
Here is an example that uses the static method ***Point3d.ArePointsCoplanar***.



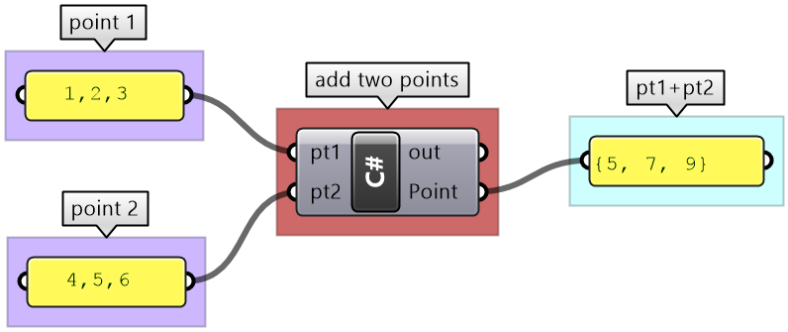

```
private void RunScript(List<Point3d> pts, double tol, ref object IsCoplanar)
{
    //Test if the list of input points are coplanar
    bool coplanar = Point3d.ArePointsCoplanar(pts, tol);

    //Assign the co-planar test to output
    IsCoplanar = coplanar;
}
```

Point3d Operators:
Many Structures and Classes in RhinoCommon implement operators whenever relevant. Operators enable you to use the “+” to add two points or use the “=” to assign the coordinates of one point to the other. ***Point3d*** structure implements many operators and this simplifies the coding and its readability. Although it is fairly intuitive in most cases, you should check the documentation to verify which operators are implemented and what they return. For example,

the documentation of adding 2 ***Point3d*** indicates the result is a new instance of ***Point3d*** where X, Y and Z are calculated by adding corresponding fields of 2 input ***Point3d***.



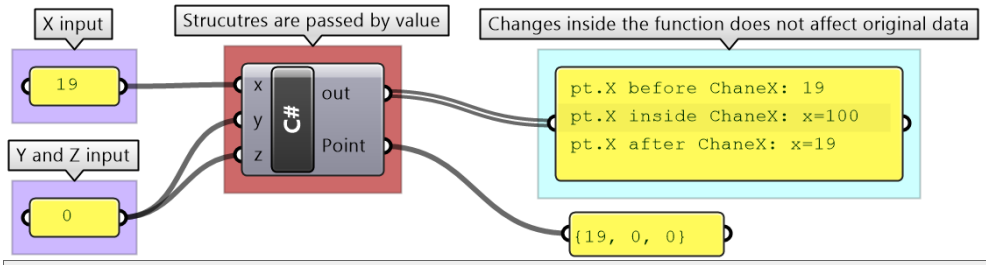
Note that all operators are declared ***public*** and ***static***. Here is an example that shows how the "+" operator in ***Point3d*** is used. Note that the "+" returns a new instance of a ***Point3d***.




```
private void RunScript(Point3d pt1, Point3d pt2, ref object Point)
{
    //Add 2 points and assign result to output
    Point = pt1 + pt2;
}
```

Point3d as a function parameter:

Point3d is a value type because it is a structure. That means if you pass an instance of ***Point3d*** to a function and change its value inside that function, the original value outside the function will not be changed, as in the following example:




```
private void RunScript(double x, double y, double z, Point3d pt2, ref object Point)
{
  Point3d pt = new Point3d(x, y, z);
  Print("pt.X before ChangeX: " + pt.X);

  //Call a function to change the value of X in the point
  ChangeX(pt);

  Print("pt.X after ChangeX: x=" + pt.X);
  Point = pt;
}

public void ChangeX(Point3d pt)
{
  pt.X = 100;
  Print("pt.X inside ChangeX: x=" + pt.X);
}
```

3_2_2: Points and vectors

RhinoCommon has a few structures to store and manipulate points and vectors[12]. Take for example the double precision points. There are three types of points that are commonly used listed in the table below.

Z as Double is missing

Class name	Member variables	Notes
Point2d	**X** as Double **Y** as Double	Used for parameter space points.
Point3d	**X** as Double **Y** as Double	Most commonly used to represent points in three dimensional coordinate space
Point4d	**X** as Double **Y** as Double **Z** as Double **W** as Double	Used for grips or control points. Grips have weight information in addition to the 3D location.

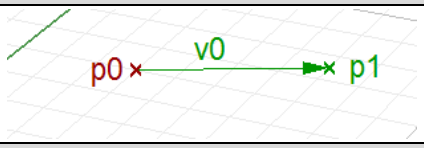
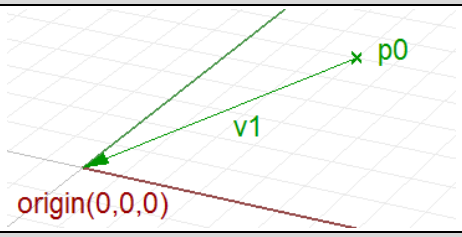
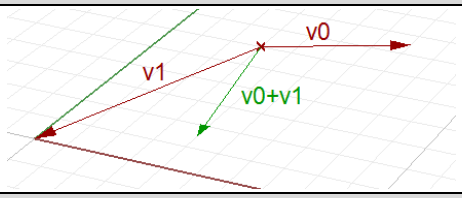
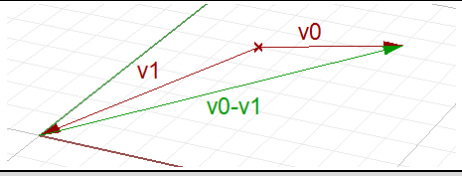
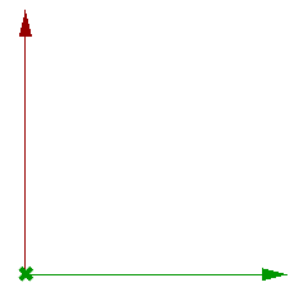
As for vectors, there are two main types.

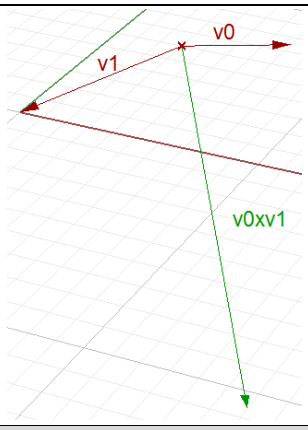
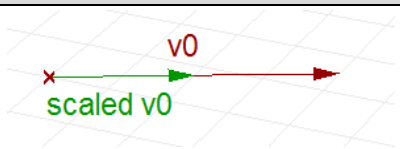
Class name	Member variables	Notes
Vector2d	**X** as Double **Y** as Double	Used in two dimensional space
Vector3d	**X** as Double	Used in three dimensional space

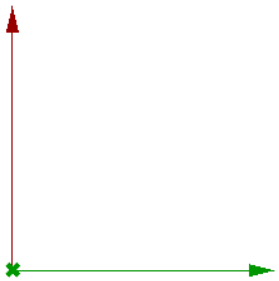
[12] For more detailed explanation of vectors and points, please refer to the "Essential Mathematics for Computational Design", a publication by McNeel.
https://www.rhino3d.com/download/rhino/6/essentialmathematics

	Y as Double **Z** as Double	

The following are a few point and vector operations with examples of output. The script starts with declaring and initializing a few values, then applying some operations.

Create a new instance of a point and vector	
Point3d p0 = new Point3d(3, 6, 0); **Vector3d** v0 = new Vector3d(4, 1.5, 0); double[illegible]tor = 0.5;	
Move a point by a vector	
Point3d p1 = new Point3d(**Point3d**.Unset); p1 = p0 + v0;	
Distance between 2 points	
double distance = p0.DistanceTo(p1);	
Point subtraction (create vector between two points)	
Vector3d v1 = new Vector3d(Vector3d.Unset); v1 = **Point3d**.Origin - p0;	
Vector addition (create average vector)	
Vector3d addVectors = new Vector3d(**Vector3d**.Unset); addVectors = v0 + v1;	
Vector subtraction	
Vector3d subtractVectors = v0 - v1;	
Vector dot product (if result Is positive number then vectors are In the same direction)	
double dot = v0 * v1; /*--- For example if ***v0*** = <10, 0, 0> ***v1*** = <0, 10, 0> then ***dot*** = 0 */	
Vector cross product (result is a vector normal to the 2 input vectors)	

Vector3d cross = new Vector3d(**Vector3d**.Unset); cross = **Vector3d**.CrossProduct(v0, v1);	
Scale a vector	
Vector3d scaleV0 = new Vector3d(Vector3d.Unset); scaleV0 = factor * v0;	
Get vector length	
double v0Length = v0.Length;	
Use vector operations to get the angle between 2 vectors	

// Unitize the input vectors
v0.Unitize();
v1.Unitize();
double dot = v0 * v1;
 	
// Force the dot product of the two input vectors to
// fall within the domain for inverse cosine, which
// is -1 <= x <= 1. This will prevent runtime
// "domain error" math exceptions.
if ((dot < -1.0))
  dot = -1.0;
if ((dot > 1.0))
  dot = 1.0;
double angle = System.Math.**Acos**(dot);
	
	v0 = <10, 0, 0>
	v1 = <0, 10, 0>
	dot = 0
	angle = acos(0) = 90 degrees

3_2_3: Lightweight curves

RhinoCommon defines basic types of curves such as lines and circles as structures and hence most of them are value types. The mathematical representation is easier to understand and is typically more light-weight. If needed, it is relatively easy to get the Nurbs approximation of these curves using the method ***ToNurbsCurve***. The following is a list of the lightweight curves.

Lightweight Curves Types	
Line	Line between two points
Polyline	Polyline connecting a list of points (not value type)
Arc	Arc on a plane from center, radius, start and end angles
Circle	Circle on a plane from center point and radius
Ellipse	Defined by a plane and 2 radiuses

The following shows how to create instances of different lightweight curve objects:

Declare and initialize 3 new points

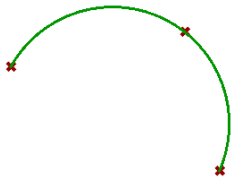
```
Point3d p0 = new Point3d(0, 0, 0);
Point3d p1 = new Point3d(5, 1, 0);
Point3d p2 = new Point3d(6, -3, 0);
```

Create an instance of a ***Line***

```
//Create an instance of a lightweight Line
Line line = new Line(p0, p1);
```

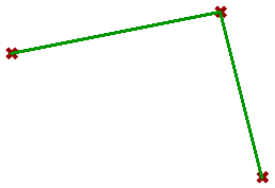
Create an instance of a ***Arc***

```
//Create an instance of a lightweight Arc
Arc arc = new Arc(p0, p1, p2);
```



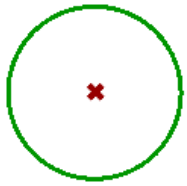
Create an instance of a ***Polyline***

```
//Put the 3 points in a list
Point3d[ ] pointList = {p0, p1, p2};

//Create an instance of a lightweight Polyline
Polyline polyline = new Polyline(pointList);
```



Create an instance of a ***Circle***

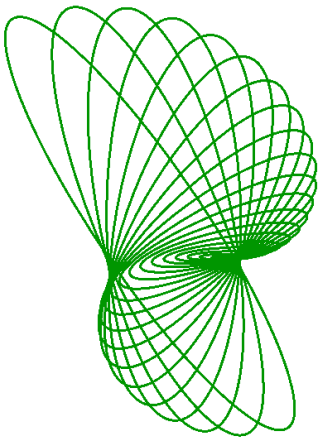
```
double radius = 3.5;

//Create an instance of a lightweight Circle
Circle circle = new Circle(p0, radius);
```



Create an list of instances of an ***Ellipse***

```
double angle = 0.01; //angle in radian
int ellipseCount = 20; //number of ellipses
double x = 1.5; // shift along X axis

//Declare a new list of ellipse curve type
List<Ellipse> ellipseListist = new List<Ellipse>();

//Use a loop to create a number of ellipse curves
Plane plane = Plane.WorldXY;

for (Int i = 1; i <= ellipseCount; i++) {
    Point3d pt = new Point3d(x, 0, 0);
    Vector3d y = new Vector3d(0,1,0);
    plane.Rotate(angle * i, y , pt);

    //Declare and instantiate a new ellipse
    Ellipse ellipse = new Ellipse(plane, (double) i / 2, i);

    //Add the ellipse to the list
    ellipseList.Add(ellipse);
  }
```

ellipseList



3_2_4: Lightweight surfaces

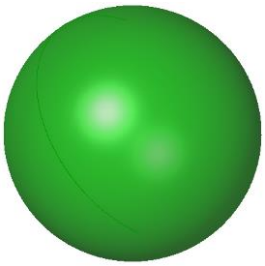
Just like with curves, ***RhinoCommon*** defines a number of lightweight surfaces that are defined as structures. They can be converted to Nurbs surfaces using the ***ToNurbsSurface()*** method. They include common surfaces such as cones and spheres. Here is a list of them:

Lightweight Surface Types	Description
Sphere	Defined by a center (or a plane) and a radius.
Cylinder	Defined by a circle and height
Cone	Defined by the center plane, radius and height
Torus	Created from a base plane and two radii

The following shows how to create instances of different lightweight surface objects:

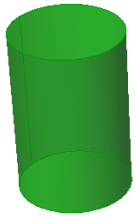
Create an instance of a ***Sphere***

```
Point3d center = Point3d.Origin;
double radius = 7.5;

Sphere sphere = new Sphere(center, radius);
```



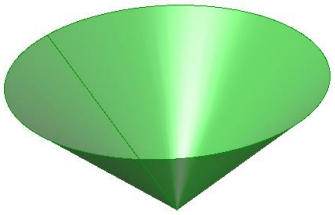
Create an instance of a ***Cylinder***

```
double height = 7.5;
double radius = 2.5;
Point3d center = Point3d.Origin;
Circle baseCircle = new Circle(center, radius);

Cylinder cy = new Cylinder(baseCircle, height);
```



Create an instance of a ***Cone***

```
double height = 7.5;
double radius = 7.5;

Cone cone = new Cone(Plane.WorldXY, height, radius);
```

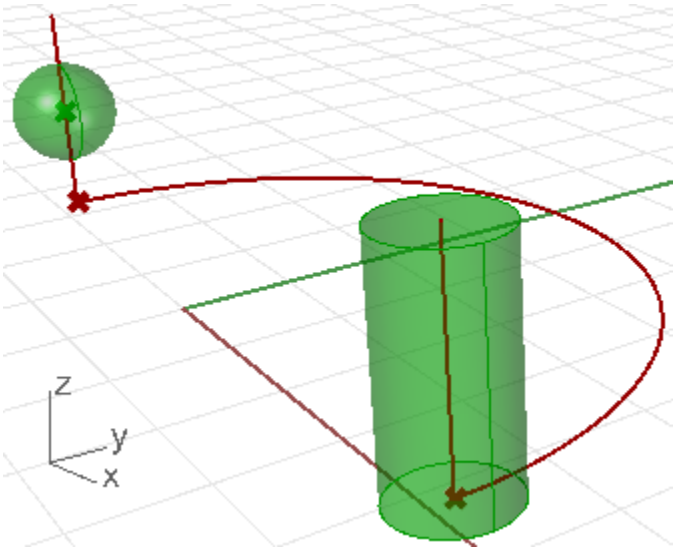


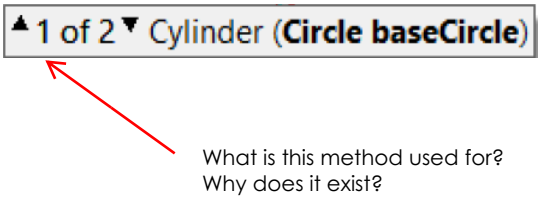
Create an instance of a ***Torus***

```
double minorRadius = 2.5;
double majorRadius = 7.5;

Torus torus = new Torus(Plane.WorldXY, majorRadius,
minorRadius);
```

3_2_5: Other geometry structures

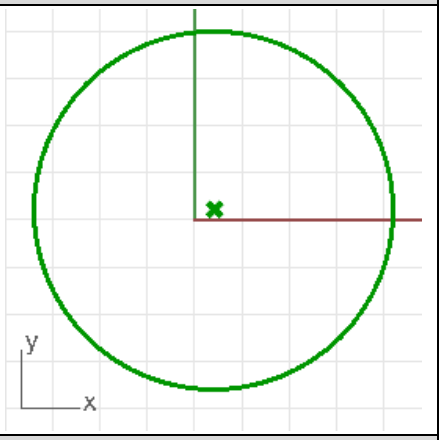
Now that we have explained the ***Point3d*** structure in some depth, and some of the lightweight geometry structures you should be able to review and use the rest using the ***RhinoCommon*** documentation. As a wrap up, the following example uses eight different structures defined in the ***Rhino.Geometry*** namespace. Those are ***Plane***, ***Point3d, Interval, Arc***, ***Vector3d***, ***Line***, ***Sphere***, and ***Cylinder***. The goal is to create the following composition.





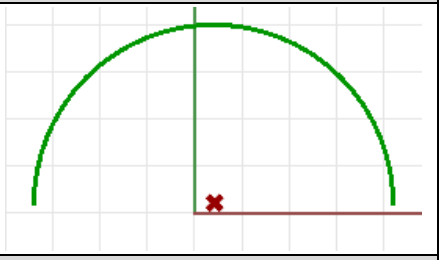
1- Create an instance of a circle on the xy-plane, center (2,1,0) and a random radius between 10 and 20

```
//Generate a random radius of a circle
Random rand = new Random();
double radius = rand.Next(10, 20);

//Create xy_plane using the Plane static method WorldXY
Plane plane = Plane.WorldXY;

//Set plane origin to (2,1,0)
Point3d center = new Point3d(2, 1, 0);
plane.Origin = center;

//Create a circle from plane and radius
Circle circle = new Circle(plane, radius);
```



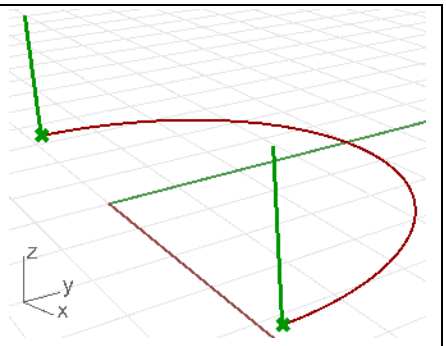
2- Create an instance of an arc from the circle and angle interval between 0 and Pi

```
//Create an arc from an input circle and interval
Interval angleInterval = new Interval(0, Math.PI);

Arc arc = new Arc(circle, angleInterval);
```



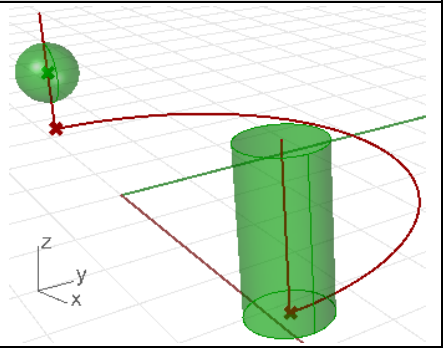
3- Extract the end points of the arc and create a vertical lines with length = 10 units

```
//Extract end points
Point3d startPoint = arc.StartPoint;
Point3d endPoint = arc.EndPoint;

//Create a vertical vector
Vector3d vec = Vector3d.ZAxis;
//Use the multiplication operation to scale the vector by 10
vec = vec * 10;

//Create start and end lines
Line line1 = new Line(startPoint, vec);
Line line2 = new Line(endPoint, vec);
```



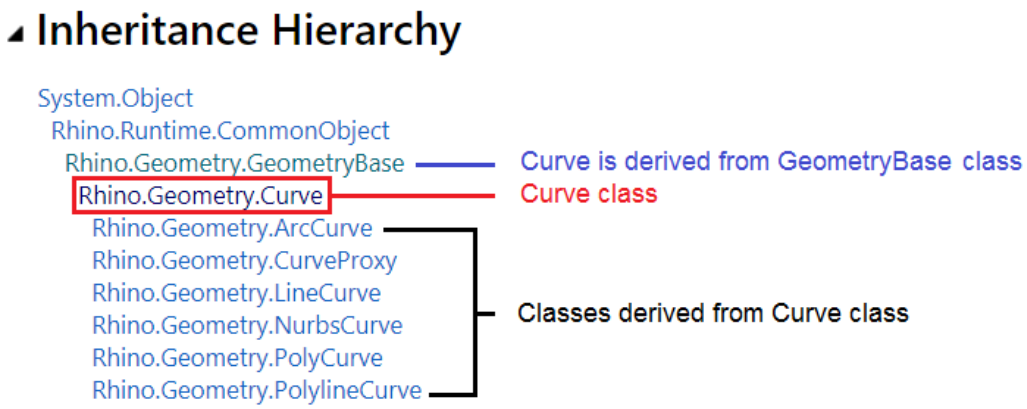
4- Create a cylinder around the start line with radius = line height/4, and a sphere around the second line centered in the middle and with radius = line height/4

```
//Create a cylinder at line1 with radius = 1/4 the length
double height = line1.Length;
double radius = height / 4;
Circle circle = new Circle(line1.From, radius);
Cylinder cylinder = new Cylinder(c_circle, height);

//Create a sphere at the center of line2 with radius = 1/4 the length
Point3d sphereCenter = line2.PointAt(0.5);
Sphere sphere = new Sphere(sphereCenter, radius);
```



3_3: Geometry classes

Just like structures, classes enable defining custom types by grouping other types together along with some custom methods and events. A class is like a blueprint that encapsulates the data and the behavior of the user-defined type. But, unlike structures, classes allow ***inheritance*** which enables defining a hierarchy of types that starts with a generic type and branches into more specific types. For example, the ***Curve*** class in ***RhinoCommon*** branches into specialized curve types such as ***ArcCurve*** and ***NurbsCurve***. The following diagram shows the hierarchy of the ***Curve*** class:



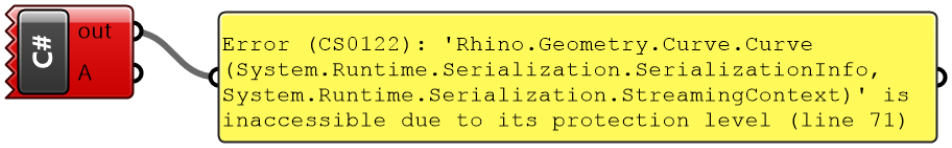
Most geometry classes are derived from the ***GeometryBase*** class. The following diagram shows the hierarchy of these classes.

- **GeometryBase** — Abstract class (cannot create an instance of an abstract class)
- **AnnotationBase**
  - **Dimension** — Abstract class derived from AnnotationBase
    - AngularDimension
    - Centermark
    - LinearDimension — Classes derived from **Dimension** class
    - OrdinateDimension
    - RadialDimension
  - Leader — Classes derived from **AnnotationBase** class
  - TextEntity
- **Brep**
- **BrepLoop**
- **Curve** — Abstract Curve class
  - ArcCurve
  - CurveProxy
    - BrepEdge
    - BrepTrim — Classes derived from **Curve** base class
  - LineCurve
  - NurbsCurve
  - PolyCurve
  - PolylineCurve
- **DetailView**
- **Hatch**
- **InstanceDefinitionGeometry**
- **InstanceReferenceGeometry**
- **Light**
- **Mesh**
- **MorphControl**
- **Point**
  - BrepVertex — Class derived from **Point** base class
- **Point3dGrid**
- **PointCloud**
- **Surface** — Abstract Surface class
  - Extrusion
    NurbsSurface
    PlaneSurface
    - ClippingPlaneSurface — Classes derived from **Surface** base class
  - RevSurface
    SumSurface
    SurfaceProxy
    - BrepFace
- **TextDot**

You can instantiate an instance of most of the classes above. However, there are some classes that you cannot instantiate. Those are usually up in the hierarchy such as the ***GeometryBase***, ***Curve*** and ***Surface***. Those are called ***abstract*** classes.

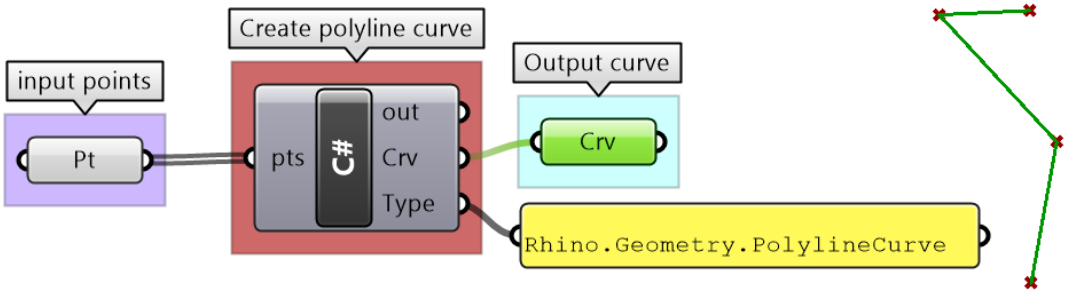
- **Abstract Classes**: The "***GeometryBase***" in ***RhinoCommon*** is one example of an abstract class. You cannot create an object or instantiate an instance of an abstract class. The purpose is to define common data and functionality that all derived classes can share.

- **Base Classes**: refer to parent classes that define common functionality for the classes that are derived from them. The ***Curve*** class is an example of a base class that also happens to be an abstract (cannot instantiate an object from it). Base classes do not have to be abstract though.



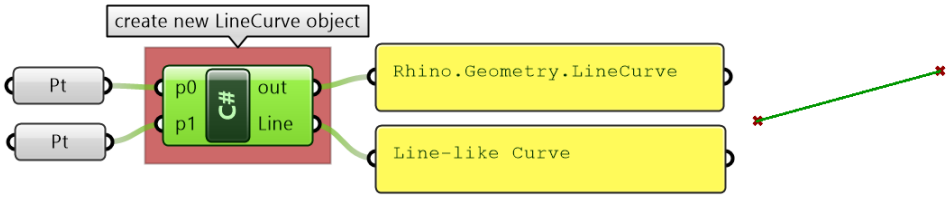

```
private void RunScript(ref object A)
{
  //ERROR: attempt to create an instance of the abstract "Curve" class
  Rhino.Geometry.Curve crv = new Rhino.Geometry.Curve();
}
```

- **Derived Classes**: inherit the members of a class they are derived from and add their own specific functionality and implementation. The ***NurbsCurve*** is an example of a derived class from the ***Curve*** class. The ***NurbsCurve*** can use all members in ***Curve*** class methods. The same is true for all other classes derived from ***Curve***, such as ***ArcCurve***, ***PolyCurve***. The following example shows how to create a new instance of the ***PolylineCurve*** class.




```
private void RunScript(List<Point3d> pts, int degree, ref object Crv, ref object Type)
{
  //Declare and create a new instance of a polyline curve from points
  var crv = new Rhino.Geometry.PolylineCurve(pts);
  //Assign curve to A output
  Crv = crv;
  //Assign curve type to B output
  Type = crv.GetType();
}
```

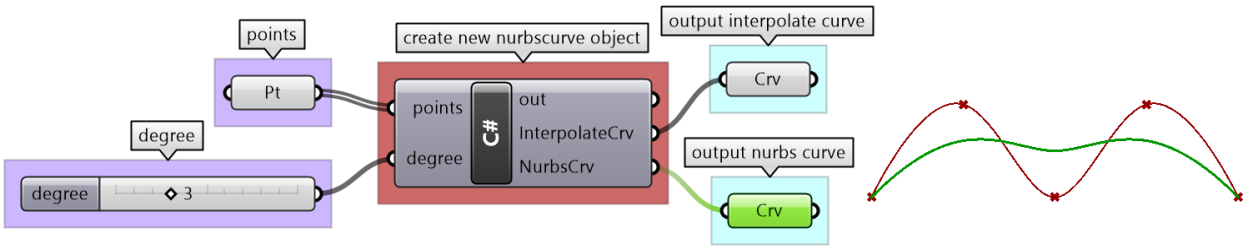
The most common way to create an instance of a class is to use the ***new*** keyword when declaring the object.




```
private void RunScript( Point3d p0, Point3d p1, ref object Line )
{
    //Create an instance of a point object and unset
    var lineCurve = new Rhino.Geometry.LineCurve(p0, p1);
    Print(lineCurve.ToString());
    Line = lineCurve
}
```

I tested that I could use: var, Cure, or LineCurve. Which would you recommend

There is another common way to create instances of classes. That is to use special methods in some classes to help create and initialize a new instance of an object. For example, the ***Curve*** class has ***static*** methods to create a variety of curve types as in the following example. Notice that you do not need to use ***new*** in this case.




```
private void RunScript(List<Point3d> points, int degree, ref object A, ref object B)
{
    //Declare a curve variable
    Rhino.Geometry.Curve inter_crv = default(Rhino.Geometry.Curve);
    //Create new instance of a curve from interpolate points
    inter_crv = Rhino.Geometry.Curve.CreateInterpolatedCurve(points, degree);

    //Declare a curve variable
    Rhino.Geometry.Curve nurbs_crv = default(Rhino.Geometry.Curve);
    //Create new instance of a curve from control points
    nurbs_crv = Rhino.Geometry.Curve.CreateControlPointCurve(points, degree);

    //Assign output
    A = inter_crv;
    B = nurbs_crv;
}
```

The following table summarizes the different ways to create new instances of objects, which applied to both class and structure types.

DIfferent way to create a new instance of an object

1- Use the class constructor

Need to use the ***new*** keyword. For example, the following creates a line from two points.

```
Rhino.Geometry.LineCurve lc = new Rhino.Geometry.LineCurve(p0, p1);
```

Note that each class may have a number of constructors that include different sets of parameters. For example the ***LineCurve*** class has the following constructors:

LineCurve()	Initializes a new instance of the LineCurve class.
LineCurve(Line)	Initializes a new instance of the LineCurve class, by retrieving its value from a line.
LineCurve(LineCurve)	Initializes a new instance of the LineCurve class, by copying values from another linear curve.
LineCurve(SerializationInfo, StreamingContext)	Protected constructor used in serialization. ← **Protected Constructor**
LineCurve(Point2d, Point2d)	Initializes a new instance of the LineCurve class, by setting start and end point from two 2D points.
LineCurve(Point3d, Point3d)	Initializes a new instance of the LineCurve class, by setting start and end point from two 3D points.
LineCurve(Line, Double, Double)	Initializes a new instance of the LineCurve class, by retrieving its value from a line and setting the domain.

Many times, there are “protected” constructors. Those are used internally by the class and you cannot use them to create a new instance of the object with them. They are basically locked. The ***LineCurve*** class has one marked in the image above.

2- Use the class static *Create* methods

Some classes include a ***Create*** method to generate a new instance of the class. Here is an example:

```
Rhino.Geometry.NurbsCurve nc = NurbsCurve.Create(isPeriodic, degree, controlPoints);
```

You can find these methods in the ***RhinoCommon*** help when you navigate the class “members”. Here are different ways to create a ***NurbsCurve*** for example and how they appear in the help.

Create	Constructs a 3D NURBS curve from a list of control points.
CreateFromArc(Arc)	Gets a rational degree 2 NURBS curve representation of the arc. Note that the parameteriza does not match arc's transcendental paramaterization.
CreateFromArc(Arc, Int32, Int32)	Create a uniform non-ratonal cubic NURBS approximation of an arc.
CreateFromCircle(Circle)	Gets a rational degree 2 NURBS curve representation of the circle. Note that the parameter does not match circle's transcendental paramaterization. Use GetRadianFromNurbFormPara GetParameterFromRadian() to convert between the NURBS curve parameter and the transc
CreateFromCircle(Circle, Int32, Int32)	Create a uniform non-ratonal cubic NURBS approximation of a circle.
CreateFromEllipse	Gets a rational degree 2 NURBS curve representation of the ellipse. Note that the parameterization of the NURBS curve does not match with the transcendenta the ellipsis.
CreateFromLine	Gets a non-rational, degree 1 Nurbs curve representation of the line.
CreateParabolaFromFocus	Creates a parabola from focus and end points.
CreateParabolaFromVertex	Createsa a parabola from vertex and end points.
CreateSpiral(Point3d, Vector3d, Point3d, Double, Double, Double, Double)	Creates a C1 cubic NURBS approximation of a helix or spiral. For a helix, you may have radi spiral radius0 == radius0 produces a circle. Zero and negative radii are permissible.
CreateSpiral(Curve, Double, Double, Point3d, Double, Double, Double, Double, Int32)	Create a C2 non-rational uniform cubic NURBS approximation of a swept helix or spiral.

3- Use the static *Create* methods of the parent class

There are times when the parent class has “Create” methods that can be used to instantiate an instance of the derived class. For example, the Curve class has few static methods that a derived class like NurbsCurve can use as in the example:

```
Rhino.Geometry.Curve crv= Curve.CreateControlPointCurve(controlPoints, degree);
Rhino.Geometry.NurbsCurve nc = crv as Rhino.Geometry.NurbsCurve;
```

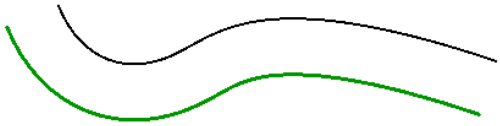
For the full set of the ***Curve Create*** methods, check the ***RhinoCommon*** documentation. Here is an example of a few of them.

S	CreateControlPointCurve(IEnumerable<Point3d>)	Constructs a control-point of degree=3 (or less).
S	CreateControlPointCurve(IEnumerable<Point3d>, Int32)	Constructs a curve from a set of control-point locations.
S	CreateCurve2View	Creates a third curve from two curves that are planar in different construction p same as each of the original curves when viewed in each plane.
S	CreateInterpolatedCurve(IEnumerable<Point3d>, Int32)	Interpolates a sequence of points. Used by InterpCurve Command This routine
S	CreateInterpolatedCurve(IEnumerable<Point3d>, Int32, CurveKnotStyle)	Interpolates a sequence of points. Used by InterpCurve Command This routine
S	CreateInterpolatedCurve(IEnumerable<Point3d>, Int32, CurveKnotStyle, Vector3d, Vector3d)	Interpolates a sequence of points. Used by InterpCurve Command This routine
S	CreateMeanCurve(Curve, Curve)	Constructs a mean, or average, curve from two curves.
S	CreateMeanCurve(Curve, Curve, Double)	Constructs a mean, or average, curve from two curves.
S	CreatePeriodicCurve(Curve)	Removes kinks from a curve. Periodic curves deform smoothly without kinks.
S	CreatePeriodicCurve(Curve, Boolean)	Removes kinks from a curve. Periodic curves deform smoothly without kinks.

4- Use the return value of a function

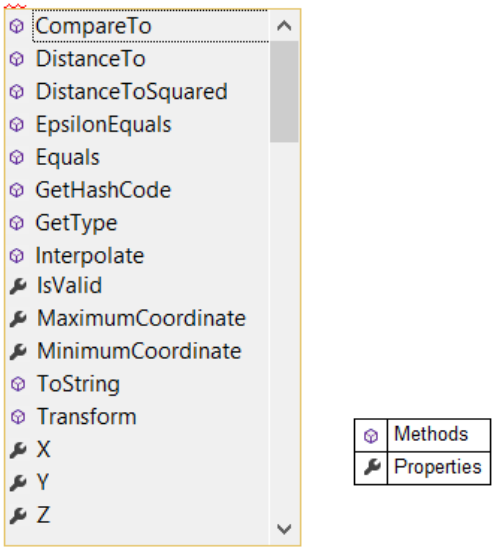
Class methods return values and sometimes those are new instances of objects. For example the ***Offset*** method in the ***Curve*** class returns a new array of curves that is the result of the offset.

```
Curve[ ] offsetCurves = x.Offset( Plane.WorldXY, 1.4, 0.01, CurveOffsetCornerStyle.None );
```



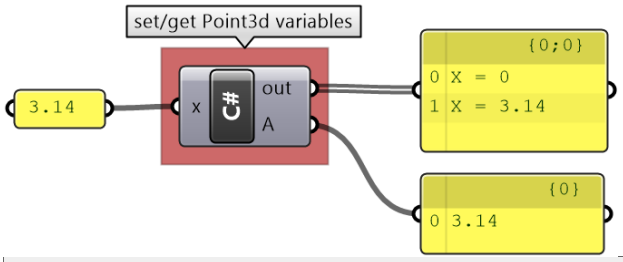
Once you create an instance of a class or a structure, you will be able to see all class methods and properties through the auto-complete feature. When you start filling the method parameters, the auto-complete will show you which parameter you are at and its type. This is a great way to navigate all available methods for each class and be reminded of what parameters are needed. Here is an example from a ***Point3d*** structure. Note that you don’t always get access to all the methods via the auto-complete. For the complete list of properties, operations and methods of each class, you should use the ***RhinoCommon*** help file.

```
Rhino.Geometry.Point3d pt = new Rhino.Geometry.Point3d(0, 0, 0);
pt.
```



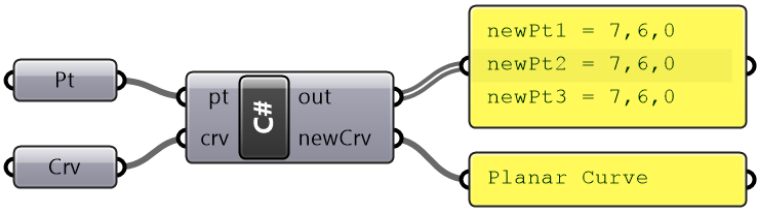
Figure(23) Auto-complete help navigate the type methods

Classes and structures define properties to set and retrieve data. Each property has either a ***Get*** or a ***Set*** method, or both. The following example shows how to get and set the coordinates of a ***Point3d*** object.




```
private void RunScript(double degree, ref object A)
{
    //Create an instance of a point object and initialize to origin (0,0,0)
    Point3d pt = new Point3d(0, 0, 0);
    //Get coordinates and print to "output"
    Print("X = " + pt.X);
    //Set the x coordinate to a new value from input
    pt.X = x;
    //Print the new x value
    Print("X = " + pt.X);
    //Get the pt "x" value and assign to output
    A = pt.X;
}
```

Copying data from an existing class to a new one can be done a few different ways depending on what the class supports. The following example shows how to copy data between two points using three different ways. It also shows how to use the ***DuplicateCurve*** method of the ***Curve*** class to create a new identical instance of a ***Curve***.




```
private void RunScript(Point3d pt, Curve crv, ref object newCrv)
{
    //Different ways to copy data between objects
    //Use the constructor when you instantiate an instance Of the Point3d class
    Point3d newPt1 = new Point3d(pt);
    Print("new pt1 = " + newPt1.ToString());

    //Use the "= Operator" If the Class provides one
    Point3d newPt2 = new Point3d(Point3d.Unset);
    newPt2 = pt;
    Print("new pt2 = " + newPt2.ToString());

    //Copy the properties one by one
    Point3d newPt3 = new Point3d(Point3d.Unset);
    newPt3.X = pt.X;
    newPt3.Y = pt.Y;
    newPt3.Z = pt.Z;
    Print("new pt3 = " + newPt3.ToString());

    //Some geometry  classes provide "Duplicate" method that Is very efficient to use
    newCrv = crv.DuplicateCurve();
}
```

3_3_1: Curves

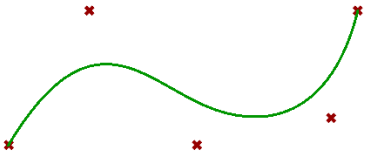
The ***RhinoCommon SDK*** has the ***abstract Rhino.Geometry.Curve*** class that provides a rich set of functionality across all curves. There are many classes derived from the parent ***Curve*** class and we will learn about how to create and manipulate them. The following is a list of the classes derived from the ***Curve*** class.

Curve Types	Notes
ArcCurve	Used to create arcs and circles
LineCurve	Used to create lines
NurbsCurve	Used to create free form curves
PolyCurves	A curve that has multiple segments joined together
PolylineCurve	A curve that has multiple lines joined together
CurveProxy	Cannot instantiate an instance of it. Both BrepEdge and BrepTrim types are derived from the CurveProxy class.

Create curve objects:
One way to create a curve object is to use the create methods available as ***static*** methods in the parent ***Rhino.Geometry.Curve*** class. Here is an example.

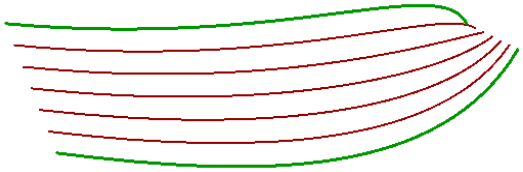
Create an instance of a ***NurbsCurve*** from control points and degree

```
Curve nc = Rhino.Geometry.Curve.CreateControlPointCurve(points, degree);
```



Create an array of tween curves using two input curves and a the number of tween curves

```
//Declare a variable of type Curve
Curve[ ] tweenCurves = null;

//Create an array of tween curves
tweenCurves = Rhino.Geometry.Curve.CreateTweenCurves(curve0, curve1, count, 0.01);
```



Another way to create new curves is to use the constructor of the curve with the ***new*** keyword. The following are examples to create different types of curves using the constructor or the ***Create*** method in the class. You can reference the ***RhinoCommon*** help for more details about the constructors of each one of the derived curve classes.

Declare and initialize 3 new points

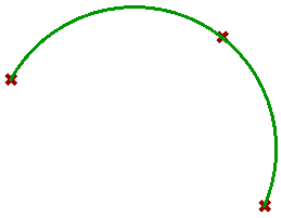
```
Point3d p0 = new Point3d(0, 0, 0);
Point3d p1 = new Point3d(5, 1, 0);
Point3d p2 = new Point3d(6, -3, 0);
```

Create an instance of a ***LineCurve*** using the class constructor and ***new*** keyword

```
//Create an instance of an LineCurve
LineCurve line = new LineCurve(p0, p1);
```

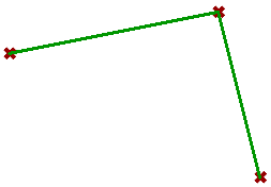
Create an instance of a ***ArcCurve*** using the class constructor and ***new*** keyword

```
//Create an instance of a lightweight Arc to pass to the
//  constructor of the ArchCurve class
Arc arc = new Arc(p0, p1, p2);

//Create a new instance of ArcCurve
ArcCurve arcCurve = new ArcCurve(arc);
```



Create an instance of a ***PolylineCurve*** using the class constructor and ***new*** keyword

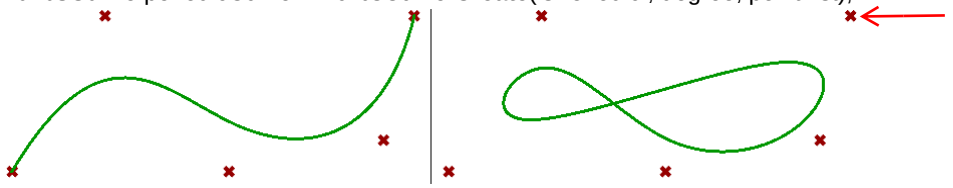
```
//Put the 3 points in a list
Point3d[ ] pointList = {p0, p1, p2};

//Create an instance of an PolylineCurve
PolylineCurve polyline = new PolylineCurve(pointList);
```



Create one open and one closed (periodic) curves using the **Create** function of the **NurbsCurve** class

```
bool isPeriodic = false;
int degree = 3;
NurbsCurve openCurve = NurbsCurve.Create(isPeriodic , degree, pointList);
```

```
isPeriodic = true;
NurbsCurve periodicCurve = NurbsCurve.Create(isPeriodic , degree, pointList);
```

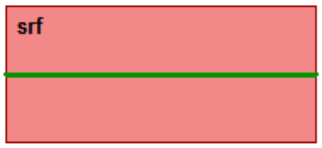


This is using undefined points in the code.

Curves can also be the return value of a method. For example offsetting a given curve creates one or more new curves. Also the surface ***IsoCurve*** method returns an instance of a curve.

Extract iso-curve from a surface. A new instance of a curve is the return value of a method

```
//srf = input surface, p = input parameter
var isoCurve = srf.IsoCurve(0, p);
```



When we wrote the code, it was divided by the surface distance not the percentage (UV).

Multiple offset of curve

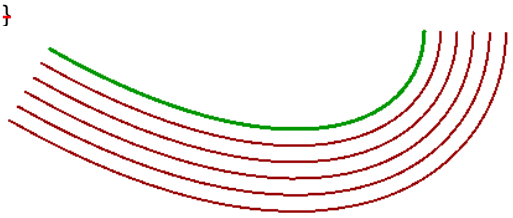
using System.Linq;

```
private void RunScript(Curve crv, int num, double dis, double tol, Plane plane)
{
  //Declare the list of curve
  List<Curve> crvs = new List<Curve>();
  Curve lastCurve = crv;
  for (int i = 1; i <= num; i++)
  {
    Curve[ ] curveArray = last_crv.Offset(plane, dis, tol, CurveOffsetCornerStyle.None);

    //Ignore if output is multiple offset curves
    if (crv.IsValid && curveArray.Count() == 1) {
      //append offset curve to array
      crvs.Add(curveArray[0]);

      //update the next curve to offset
      lastCUrve = curveArray[0];
    }
    else
      break;
  }
}
```

lastCurve

lastCurve

*Length (if not using Linq)



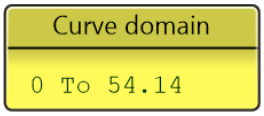
Curve methods:

Each class can define methods that help navigate the data of the class and extract some relevant information. For example, you might want to find the endpoints of a curve, find the tangent at some point, get a list of control points or divide the curve. The ***AutoComplete*** helps to quickly navigate and access these methods, but you can also find the full description in the ***RhinoCommon*** help.

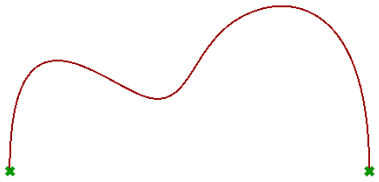
Keep in mind that a derived class such as NurbsCurve can access not only its own methods, but also the methods of the classes it was derived from. Therefore an instance of a ***NurbsCurve***, can access the ***NurbsCurve*** methods and the ***Curve*** methods as well. The methods that are defined under the ***Curve*** are available to all of the classes derived from the **Curve** class such as ***LineCurve***, ***ArcCurve***, ***NurbsCurve***, etc. Here are a few examples of curve methods.

<table>
<tr><th colspan="2">Some of the Curve and NurbsCurve methods</th></tr>
<tr><td>

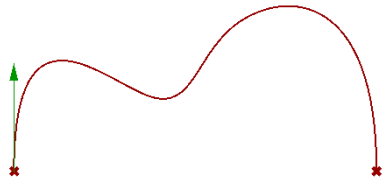
```
//Get the domain or interval of the curve
Interval domain = crv.Domain;
```

</td><td> 
</td></tr>
<tr><td>

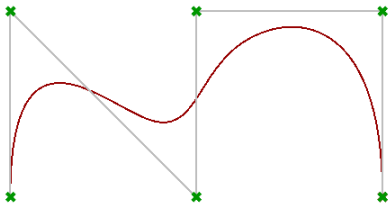
```
//Get the start and end points of a curve
Point3d startPoint = crv.PointAtStart;
Point3d endPoint = crv.PointAtEnd;
```

</td><td>  </td></tr>
<tr><td>

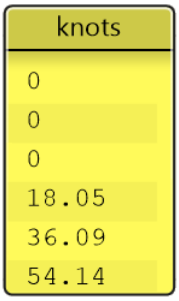
```
//Get the tangent at start of a curve
Vector3d startTangent = crv.TangentAtStart;
```

</td><td>  </td></tr>
<tr><td>

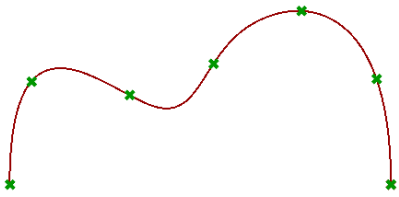
```
//Get the control points of a NurbsCurve (nc)
List<Point3d> cpList = new List<Point3d>();
int count = nc.Points.Count;

                    Nurbs Curve = crv.ToNurbsCurve ();
//Loop to get all cv points
for (int i = 0; i <= count - 1; i++) {
    ControlPoint cp = nc.Points[i];
    cpList.Add(cp.Location);
}
```

</td><td>  </td></tr>
<tr><td>

```
//Get the knot list of a NurbsCurve (nc)
List<double> knotList = new List<double>();
int count = nc.Points.Count;

//Loop to get all knots values
for (int i = 0; i <= count - 1; i++) {
    double knot = nc.Knots[i];
    knotList.Add(knot);
}
```

</td><td> 
</td></tr>
<tr><td>

```
//Divide curve (crv) by number (num)
//Declare an array of points
Point3d[ ] points = { };

//Divide the curve by number
crv.DivideByCount(num, true, out points);
```

</td><td>  </td></tr>
</table>

3_3_2: Surfaces

There are many surface classes derived from the abstract ***Rhino.Geometry.Surface*** class. The ***Surface*** class provides common functionality among all of the derived types. The following is a list of the surface classes and a summary description.

Surface derived Types	Notes
Extrusion	Represents surfaces from extrusion. It is much lighter than a NurbsSurface
NurbsSurface	Used to create free form surfaces
PlaneSurface	Used to create planar surfaces
RevSurface	Represents a surface of revolution
SumSurface	Represents a sum surface, or an extrusion of a curve along a curved path.
SurfaceProxy	Cannot instantiate an instance of it. Provides a base class to brep faces and other surface proxies.

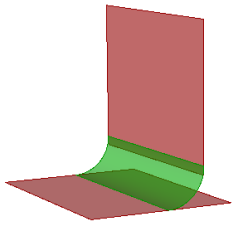
Create surface objects:
One way to create surfaces is by using the static methods in the ***Rhino.Geometry.Surface*** class that start with the keyword ***Create***. Here are some of these create methods..

The ***Create*** methods in ***Rhino.Geometry.Surface*** class	
CreateExtrusion	Constructs a surface by extruding a curve along a vector.
CreateExtrusionToPoint	Constructs a surface by extruding a curve to a point.
CreatePeriodicSurface	Constructs a periodic surface from a base surface and a direction.
CreateRollingBallFillet	Constructs a rolling ball fillet between two surfaces.
CreateSoftEditSurface	Creates a soft edited surface from an existing surface using a smooth field of influence.

The following is an example to create a fillet surface between 2 input surfaces given some radius and tolerance.

```
private void RunScript(Surface srfA, Surface srfB, double radius, double tol, ref object A)
{
   //Declare an array of surfaces
   Surface[ ] surfaces = {};

   //Check for a valid input
   if (srfA != null && srfB != null) {
      //Create fillet surfaces
      surfaces = Surface.CreateRollingBallFillet(srfA, srfB, radius, tol);
    }
 }
```

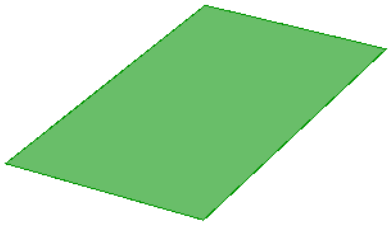
If the surface is made out of 3 points it doesn't work... Do you know the reason why?



However, the most common way to create a new instance of a derived surface type is to either use the constructor (with ***new*** keyword), or the ***Create*** method of the derived surface class. Here are a couple examples that show how to create instances from different surface types.

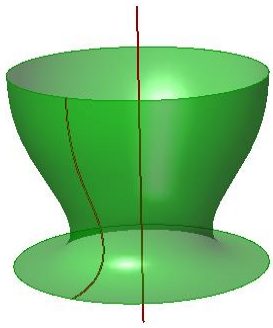
Create an instance of a ***PlaneSurface*** using the constructor and ***new*** keyword

```
var plane = Plane.WorldXY;
var x_interval = new Interval(1.0, 3.5);
var y_interval = new Interval(2.0, 6.0);

//Create planar surface
var planeSrf = new PlaneSurface(plane, x_interval, y_interval);
```



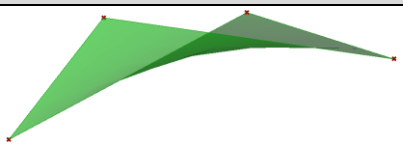
Create an instance of a ***RevSurface*** from a line and a profile curve

```
RevCurve revCrv = … //from input
Line revAxis = … //from input

//Create surface of revolution
var revSrf = RevSurface.Create(revCrv, revAxis);
```



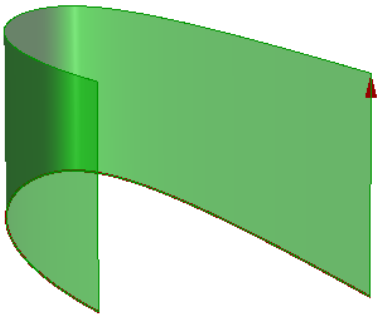
Create an instance of a ***NurbsSurface*** from a list of control points

```
List<Point3d> points = … //from input

//Create nurbs surface from control points
NurbsSurface ns = null;
ns = NurbsSurface.CreateThroughPoints(points, 2, 2, 1, 1, false, false);
```



Create an instance of a ***NurbsSurface*** from extruding a curve in certain direction

```
Curve cv = … //from input
Vector3d dir = … //from input

//Create a nurbs surface from extruding a curve in some dir
var nSrf = NurbsSurface.CreateExtrusion(crv, dir);
//Create an extrusion from extruding a curve in some dir
var ex = Extrusion.Create(crv, 10, false);

//Note that in Grasshopper 1.0 there is no support to extrusion objects
and hence the output within GH is converted to a nurbs surface. The
only way to bake an Extrusion instance is to add the object to the active
document from within the scripting component as in the following
Rhino.RhinoDoc.ActiveDoc.Objects.AddExtrusion(ex);
```



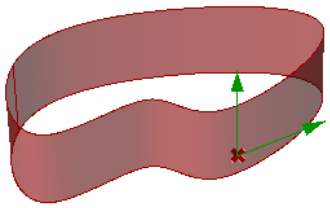
Surface methods:

Surface methods help edit and extract information about the surface object. For example, you might want to learn if the surface is closed or if it is planar. You might need to evaluate the surface at some parameter to calculate points on the surface, or get its bounding box. There are also methods to extract a lightweight geometry out of the surface. They start with “Try” keyword. For example, you might have a RevSurface that is actually a portion of a torus. In that case, you can call TryGetTorus. All these methods and many more can be accessed through the surface methods. A full list of these methods is documented in the ***RhinoCommon SDK*** documentation.

Examples of the ***Surface*** and ***NurbsSurface*** methods

```
Surface srf = … //from input
//Check if the input surface is closed in the u or v direction
bool isClosedU = srf.IsClosed(0);
bool isClosedV = srf.IsClosed(1);

//Check if the surface is planar at zero tolerance
bool isPlanar = srf.IsPlanar();
```

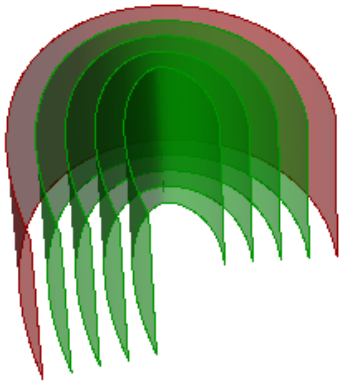
```
Surface srf = … //from input
double u = 0.5;
double v = 0.5;
//Declare and instantiate the evaluation point and derivative vectors
Point3d evalPt = new Point3d(Point3d.Unset);
Vector3d[ ] derivatives = {};

//Evaluate the surface and extract the first derivative to get tangents
srf.Evaluate(u, v, 1, out evalPt, out derivatives);

Vector3d tanU = derivatives[0];
Vector3d tanV = derivatives[1];
```



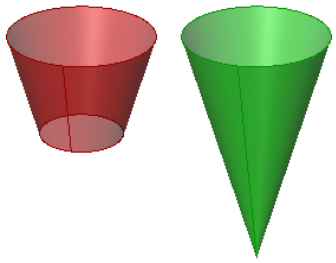
```
//Declare the list of surfaces
List<Surface> srfs = new List<Surface>();

Surface lastSrf = srf;
for (int i = 1; i <= num; i++) {
    Surface offset_srf = last_srf.Offset(dis, tol);
    if (srf.IsValid) {
        //append offset surface to array
        srfs.Add(offset_srf);
        //update the next curve to offset
        lastSrf = offset_srf;
    }
    else
        break;
}
```

srf = ... (from intput)
num = ... (from input)
dis = ... (from input)
tol = ... (from input)

lastSrf



```
//Declare and instantiate an axis and a curve
Line axis = new Line (Point3d.Origin, new Point3d(0, 0, 10));
LineCurve crv = new LineCurve(new Point3d(2, 0, 0), new
Point3d(3.5, 0, 5));

//Create surface of revolution
RevSurface revSrf = RevSurface.Create(crv, axis);

//Try to get a Cone
Cone cone;
if( revSrf.TryGetCone(out cone))
    Print("Cone was successfully create from surface");
```



3_3_3: Meshes

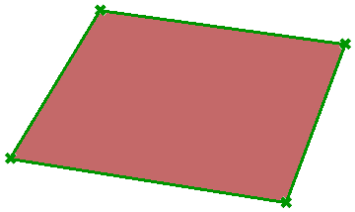
Meshes represent a geometry class that is defined by faces and vertices. The mesh data structure basically includes a list of vertex locations, faces that describe vertices connections and normal of vertices and faces. More specifically, the geometry lists of a mesh class include the following.

Mesh geometry lists	Description
Vertices	Of type MeshVertexList - includes a list of vertex locations type Point3f.
Normals	Of type MeshVertexNormalList - includes a list of normals of type Vector3f.
Faces	Of type MeshFaceList - includes a list of normals of type MeshFace.
FaceNormals	Of type "MeshFaceNormalList" - includes a list of normals of type Vector3f.

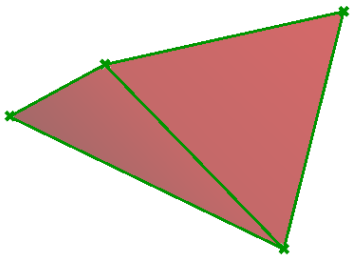
Create mesh objects:

You can create a mesh from scratch by specifying vertex locations, faces and compute the normal as in the following examples.

```
//Create a simple rectangular mesh that has 1 face
//Create a new instance of a mesh
Rhino.Geometry.Mesh mesh = new Rhino.Geometry.Mesh();

//Add mesh vertices
mesh.Vertices.Add(0.0, 0.0, 0.0); //0
mesh.Vertices.Add(5.0, 0.0, 0.0); //1
mesh.Vertices.Add(5.0, 5.0, 0.0); //2
mesh.Vertices.Add(0.0, 5.0, 0.0); //3

//Add mesh faces
mesh.Faces.AddFace(0, 1, 2, 3);

//Compute mesh normals
mesh.Normals.ComputeNormals();

//Generate any additional mesh data
mesh.Compact();
```



```
//Create simple triangular mesh that has 2 faces
//Create a new instance of a mesh
Rhino.Geometry.Mesh mesh = new Rhino.Geometry.Mesh();

//Add mesh vertices
mesh.Vertices.Add(0.0, 0.0, 0.0); //0
mesh.Vertices.Add(5.0, 0.0, 2.0); //1
mesh.Vertices.Add(5.0, 5.0, 0.0); //2
mesh.Vertices.Add(0.0, 5.0, 2.0); //3

//Add mesh faces
mesh.Faces.AddFace(0, 1, 2);
mesh.Faces.AddFace(0, 2, 3);

//Compute mesh normals
mesh.Normals.ComputeNormals();

//Generate any additional mesh data
mesh.Compact();
```

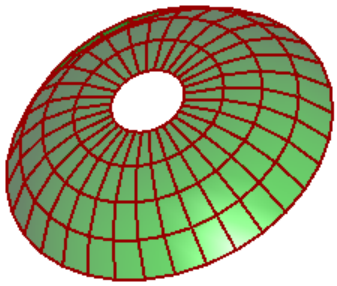


The Mesh class includes many ***CreateFrom*** static methods to create a new mesh from various other geometry. Here is the list with description as it appears in the ***RhinoCommon*** help.

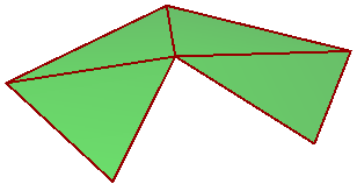
S	CreateFromBox(BoundingBox, Int32, Int32, Int32)	Constructs new mesh that matches a bounding box.
S	CreateFromBox(Box, Int32, Int32, Int32)	Constructs new mesh that matches an aligned box.
S	CreateFromBox(IEnumerable<Point3d>, Int32, Int32, Int32)	Constructs new mesh from 8 corner points.
S	CreateFromBrep(Brep)	Constructs a mesh from a brep.
S	CreateFromBrep(Brep, MeshingParameters)	Constructs a mesh from a brep.
S	CreateFromClosedPolyline	Attempts to create a Mesh that is a triangulation of a closed polyline.
S	CreateFromCone(Cone, Int32, Int32)	Constructs a solid mesh cone.
S	CreateFromCone(Cone, Int32, Int32, Boolean)	Constructs a mesh cone.
S	CreateFromCurvePipe	Constructs a new mesh pipe from a curve.
S	CreateFromCylinder	Constructs a mesh cylinder
S	CreateFromLines	Creates a mesh by analizing the edge structure. Input lines could be from the extraction of edges from an original mesh.
S	CreateFromPlanarBoundary(Curve, MeshingParameters)	Do not use this overload. Use version that takes a tolerance parameter instead.
S	CreateFromPlanarBoundary(Curve, MeshingParameters, Double)	Attempts to construct a mesh from a closed planar curve.RhinoMakePlanarMeshes
S	CreateFromPlane	Constructs a planar mesh grid.
S	CreateFromSphere	Constructs a mesh sphere.
S	CreateFromSurface(Surface)	Constructs a mesh from a surface
S	CreateFromSurface(Surface, MeshingParameters)	Constructs a mesh from a surface
S	CreateFromTessellation	Attempts to create a mesh that is a triangulation of a list of points, projected on a plane, including its holes and fixed edges.

Here are a couple examples to show how to create a mesh from a brep and a closed polyline.

```
//Set mesh parameters to default
var mp = MeshingParameters.Default;

//Create meshes from brep
var newMesh = Mesh.CreateFromBrep(brep, mp);
```



```
//Create a new mesh from polyline
var newMesh = Mesh.CreateFromClosedPolyline(pline);
```



Navigate mesh geometry and topology:
You can navigate mesh data using ***Vertices*** and ***Faces*** properties. The list of vertices and faces are stored in a collection class or type with added functionality to manage the list efficiently. ***Vertices*** list for example, is of type ***MeshVertexList***. So if you need to change the locations of

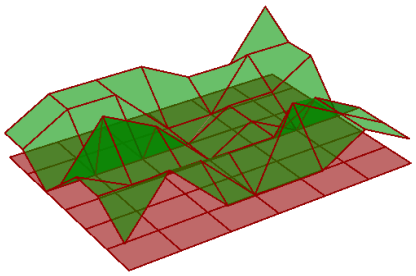
mesh vertices, then you need to get a copy of each vertex, change the coordinates, then reassign in the vertices list. You can also use the ***Set*** methods inside the ***MeshVertexList*** class to change vertex locations.

The following example shows how to randomly change the ***Z*** coordinate of a mesh using two different approaches. Notice that mesh vertices use a ***Point3f*** and not ***Point3d*** type because mesh vertex locations are stored as a single precision floating point.

Change the location of mesh vertices by assigning the new value to the ***Vertices'*** list

```
//Change mesh vertex location
int index = 0;
Random rand = new Random();

foreach (Point3f loc in mesh.Vertices) {
    Point3f newLoc = loc;
    newLoc.Z = rand.Next(10) / 3;
    mesh.Vertices[index] = newLoc;
    index = index + 1;
}
```

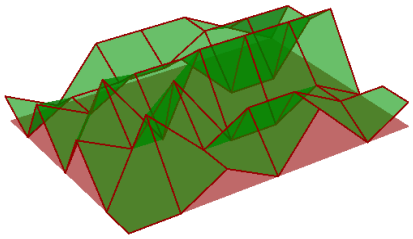
Mesh mesh = …// from input



Change the location of mesh vertices using the ***SetVertix*** method

```
Mesh mesh = … // from input

//Create a new instance of random generator
Random rand = new Random();
for (int i = 0; i <= mesh.Vertices.Count - 1; i++){
    //Get vertex
    Point3f loc = mesh.Vertices[i];
    loc.Z = rand.Next(10) / 3;
    //Assign new location
    mesh.Vertices.SetVertex(i, loc);
}
```

SetVertex

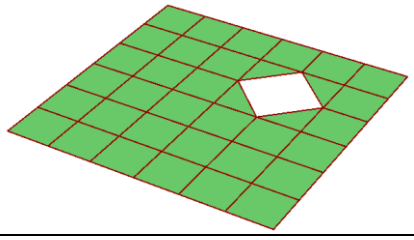


Here is an example that deletes a mesh vertex and all surrounding faces

Delete mesh faces randomly

```
Mesh mesh = … // from input
int index = 32;
//Remove a mesh vertex (make sure index falls within range)
if (index >= 0 & i < mesh.Vertices.Count)
{
    mesh.Vertices.Remove(index, true);
}
```

index

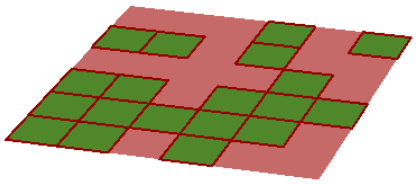


You can also manage the ***Faces*** list of a mesh. Here is an example that deletes about half the faces randomly from a given mesh.

Delete mesh faces randomly

```
List<int> faceIndices = new List<int>();
int count = mesh.Faces.Count;

//Create a new instance of random generator
Random rand = new Random();

for (int i = 0; i <= count - 1; i += 2){
```

using System.Linq;
Mesh mesh = … // from input



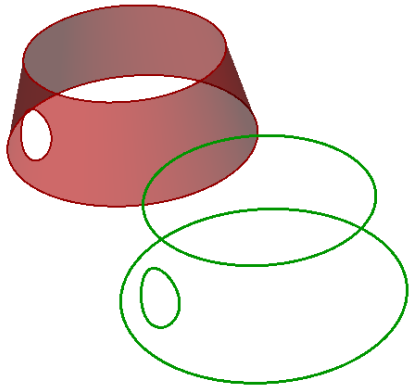
```
    int index = rand.Next(count);
    faceIndices.Add(index);
}

//delete duplicate indexes
List<int> distinctIndices = faceIndices.Distinct().ToList();

//delete faces
mesh.Faces.DeleteFaces(distinctIndices);
```

Meshes keep track of the connectivity of the different parts of the mesh. If you need to navigate related faces, edges or vertices, then this is done using the mesh topology. The following example shows how to extract the outline of a mesh using the mesh topology.

Mesh topology example: extract the outline of a mesh

```
// Get the mesh's topology
Rhino.Geometry.Collections.MeshTopologyEdgeList meshEdges =
mesh.TopologyEdges;

List<Line> lines = new List<Line>();

// Find all of the mesh edges that have only a single mesh face
for (int i = 0; i <= meshEdges.Count - 1; i++) {
    int numOfFaces = meshEdges.GetConnectedFaces(i).Length;

    if ((numOfFaces  == 1)) {
        Line line = meshEdges.EdgeLine(i);
        lines.Add(line);
    }
}
```

Mesh mesh = … // from input

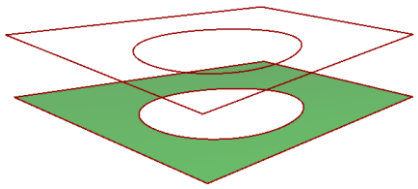


Mesh methods:

Once a new mesh object is created, you can edit and extract data out of that mesh object. The following example extracts naked edges out of some input mesh.

Extract the outline of a mesh

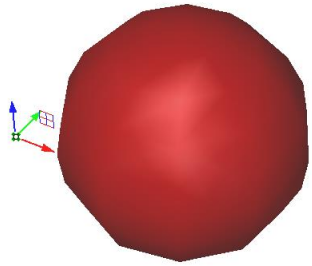
```
Mesh mesh = … // from input

//Declare an array of polylines
Polyline[ ] naked_edges = {};

//Create a new mesh from polyline
nakedEdges = mesh.GetNakedEdges();
```



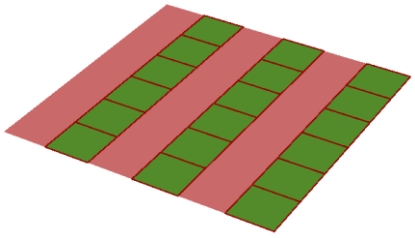
Test if a given point is inside a closed mesh

```
Mesh mesh = … // from input
Point3d pt = … // from input

//Test if the point is inside the mesh
mesh.IsPointInside(pt, tolerance, true);
```

double tolerance = … // from input



Delete every other mesh face, then split disjoint meshes	

Mesh mesh = … // from input
List<int> faceIndeces = new List<int>();
 	
//Loop through faces and delete every other face
for (int i = 0; i <= mesh.Faces.Count - 1; i += 2) {
faceIndeces.Add(i);
}
//delete faces
mesh.Faces.DeleteFaces(faceIndeces);
//Split disjoint meshes
Mesh[ ] meshArray = mesh.SplitDisjointPieces();
	

3_3_4: Boundary representation (Brep)

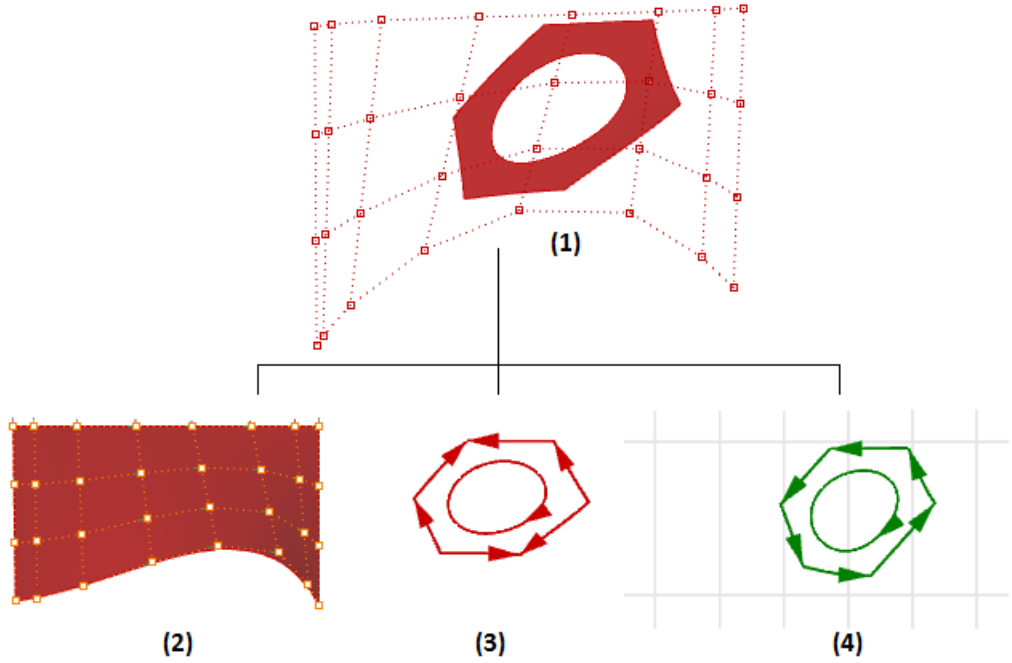
The boundary representation is used to unambiguously represent trimmed nurbs surfaces and polysurfaces. There are two sets of data that are needed to fully describe the 3D objects using the boundary representation. Those are geometry and topology.

Brep geometry:
Three geometry elements are used to create any Breps:

1- The 3D untrimmed nurbs surfaces in the modeling space.

2- The 3D curves, which are the geometry of edges in modeling space.

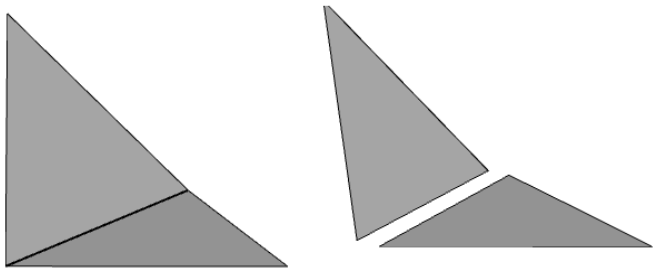
3- The 2D curves, which are the geometry of trims in the parameter space.



Figure(24): Geometry elements of a typical trimmed nurbs surface.
(1) Trimmed surface with control points turned on. (2) Underlying 3-D untrimmed surface. (3) 3-D curves (for the edges) in modeling space. (4) 2-D curves (for the trims) in parameter space

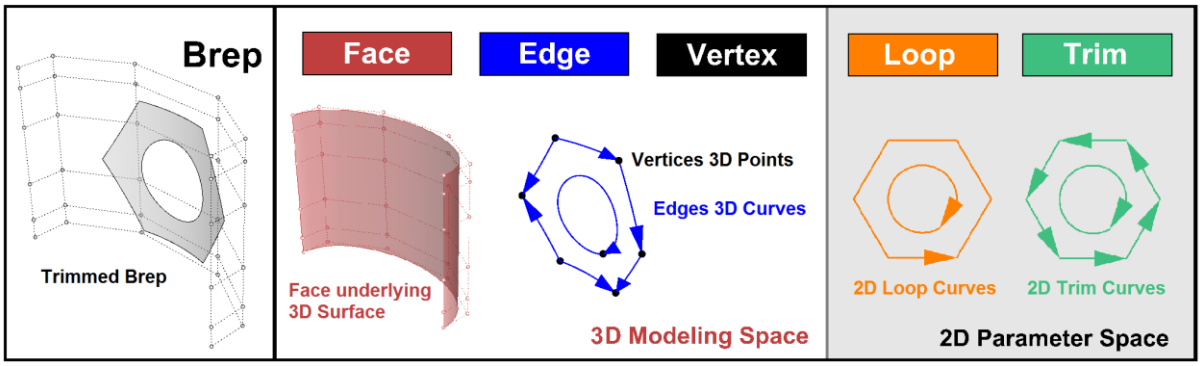
Brep topology:
A topology refers to how different parts of the 3-D object are connected. For example we might have two adjacent faces with one of their edges aligned. There are two possibilities to describe

the relationship or connectivity between these two faces. They could either be two separate entities that they can be pulled apart, or they are joined in one object sharing that edge. The topology is the part that describes such relationships.



Figure(25): Topology describes connectivity between geometry elements. Two adjacent faces can either be joined together in one polysurface or are two separate faces that can be pulled apart.

Brep topology includes faces, edges, vertices, trims and loops. The following diagram lists the Brep topology elements and their relation to geometry in the context of ***RhinoCommon***.



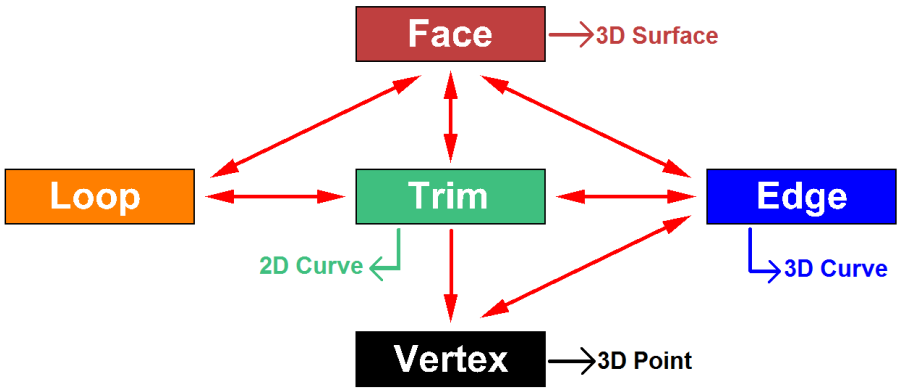
Figure(26): The Brep topology elements and their relation to geometry

The topology elements of the Brep can be defined as follows:

Topology	Referenced Geometry	Description
Vertices	3D points (location in 3D space)	They describe the corners of the brep. Each vertex has a 3D location. They are located at the ends of edges and are shared between neighboring edges.
Edges	3D Nurbs curves (in 3D modeling space)	Edges describe the bounds of the brep. Each references the 3D curve, two end vertices and the list of trims. If an edge has more than one trim that means the edge is shared among more than one face. Multiple edges can reference the same 3D curve (typically different portion of it).
Faces	3D underlying Nurbs surfaces (in 3D modeling space)	Faces reference the 3D surface and at least one outer loop. A face normal direction might not be the same as that of the underlying surface normal. Multiple faces can reference the same 3D surface (typically different portion of it).

Loops	2D closed curves of connected trims (in 2D parameter space)	Each face has exactly one outer loop defining the outer boundary of the face. A face can also have inner loops (holes). Each loop contains a list of trims.
Trims	2D Curves (in 2D parameter space)	Each trim references one 2D curve and exactly one edge, except in singular trims, there is no edge. The 2D curves of the trims run in a consistent direction. Trims of outer or boundary of the face run anti-clockwise regardless of the 3D curve direction of its edge. The 2D curves of the trims of the inner loops run clockwise.

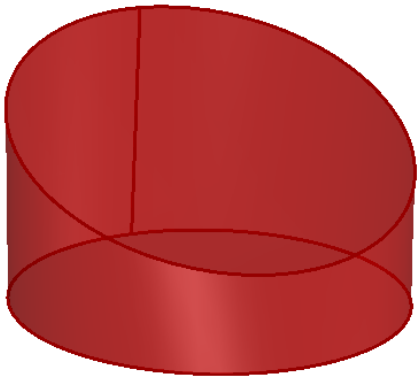
Each brep includes lists of geometry and topology elements. Topology elements point to each other and the geometry they reference which makes it easy to navigate through the brep data structure. The following diagram shows navigation paths of the brep topology and geometry elements.



Figure(27): Navigation diagram of the ***Brep*** data structure in ***RhinoCommon***.

The topology of the ***Brep*** and navigating different parts

```
Brep brep = … //from input
//BrepFace topology (3D modeling space)
Rhino.Geometry.Collections.BrepFaceList faces = brep.Faces;
for(int fi = 0; fi < faces.Count; fi++)
   {
     BrepFace face = faces[fi];
     //Get Adjacent faces
     var aFaces = face.AdjacentFaces();
     //Get Adjacent edges
     var aEdges = face.AdjacentEdges();
     //Get face loops
     var faceLoops = face.Loops;
     //Get the 3D untrimmed surface
     var face3dSurface = face.UnderlyingSurface();
}

//BrepLoop topology (2D parameter space)
Rhino.Geometry.Collections.BrepLoopList loops = brep.Loops;
for(int li = 0; li < loops.Count; li++)
{
     BrepLoop loop = loops[li];
     //Get loop face
     var loopFace = loop.Face;
     //Get loop trims
```



```
    var loopTrims = loop.Trims;
    //Get loop 2D and 3D curves
    var loop2dCurve = loop.To2dCurve();
    var loop3dCurve = loop.To3dCurve();
}

//BrepEdge topology (3D modeling space)
Rhino.Geometry.Collections.BrepEdgeList edges = brep.Edges;
for(int ei = 0; ei < edges.Count; ei++)
{
    BrepEdge edge = edges[ei];
    //Get edge faces
    var eFaces_i = edge.AdjacentFaces();
    //Get edge start and end vertices
    var eStartVertex = edge.StartVertex;
    var eEndVertex = edge.EndVertex;
    //Get edge trim indices
    var eTrimIndeces = edge.TrimIndices();
    //Get edge 3D curve
    var e3dCurve = edge.EdgeCurve;
}

//BrepTrim topology (2D parameter space)
Rhino.Geometry.Collections.BrepTrimList trims = brep.Trims;
for(int ti = 0; ti < trims.Count; ti++)
{
    BrepTrim trim = trims[ti];
    //Get the edge
    var trimEdge = trim.Edge;
    //Get trim start and end vertices
    var trimStartVertex = trim.StartVertex;
    var trimEndVertex = trim.EndVertex;
    //Get trim loop
    var trimLoop = trim.Loop;
    //Get trim face
    var trimFace = trim.Face;
    //Get trim 2D curve
    var trim2dCurve = trim.TrimCurve;
}

//BrepVertex topology (3D modeling space)
Rhino.Geometry.Collections.BrepVertexList vertices = brep.Vertices;
for(int vi = 0; vi < vertices.Count; vi++)
{
    BrepVertex vertex = vertices[vi];
    //Get vertex edges
    var vEdges = vertex.EdgeIndices();
    //Get vertex location
    var vPoint = vertex.Location;
}
```

In polysurfaces (which are ***Breps*** with multiple faces), some geometry and topology elements are shared along the connecting curves and vertices where polysurface faces join together. In the following example, the two faces F0 and F1 share one edge E0 and two vertices V0 and V2.

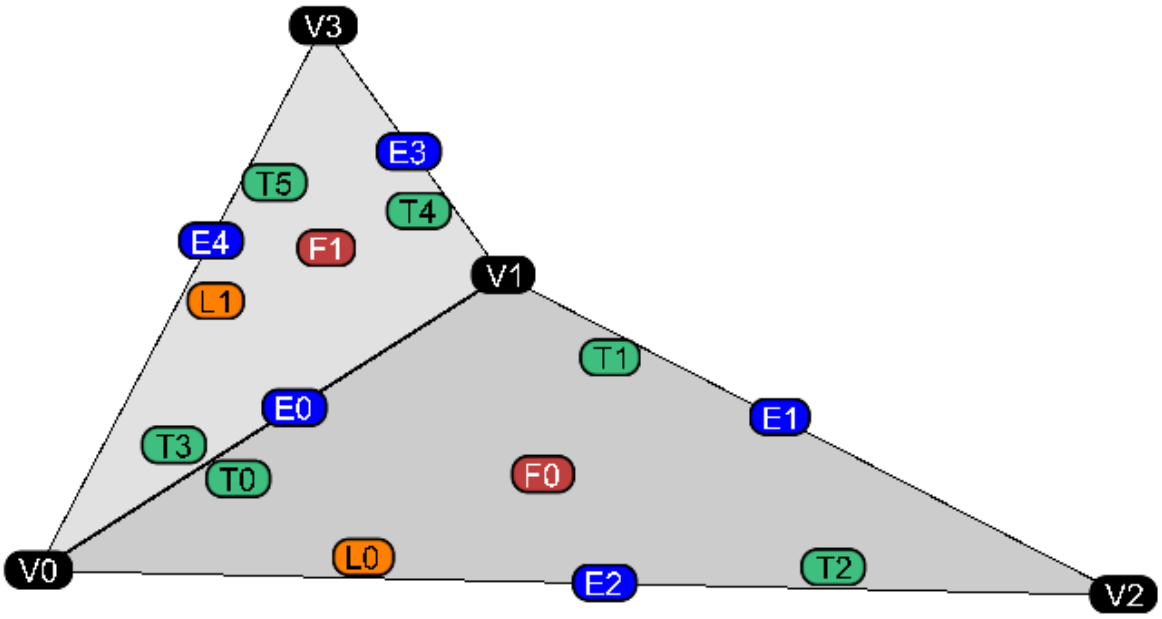
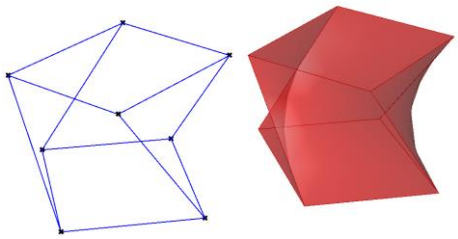


Figure (28): Vertices, edges and 3-D curves are shared between neighboring faces. Edge (E0) and Vertices (V0 and V2) are shared between the two faces (F0 & F1)..

As illustrated before, **Breps** has a rather complex data structure and it is useful to learn how to navigate this data. The following examples show how to get some of the geometry parts.

Example to count geometry and topology elements of a ***Brep*** and then extract the 3D geometry

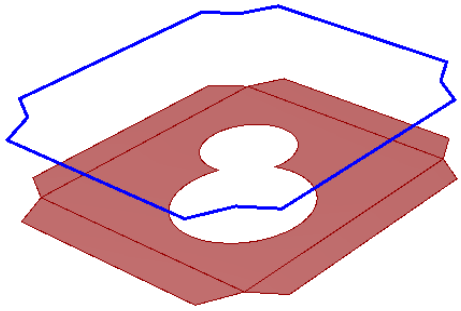
```
Brep brep = … //from input
//Print the number of geometry elements
Print("brep.Vertices.Count = {0}", brep.Vertices.Count);
Print("brep.Curves3D.Count = {0}", brep.Curves3D.Count);
Print("brep.Curves2D.Count = {0}", brep.Curves2D.Count);
Print("brep.Surfaces.Count = {0}", brep.Surfaces.Count);

//Print the number of topology elements
Print("brep.Trims.Count = {0}", brep.Trims.Count);
Print("brep.Loops.Count = {0}", brep.Loops.Count);
Print("brep.Faces.Count = {0}", brep.Faces.Count);
Print("brep.Edges.Count = {0}", brep.Edges.Count);

//Extract 3d geometry elements
var V = brep.Vertices;
var C = brep.Curves3D;
var S = brep.Surfaces;
```



```
brep.Vertices.Count = 8
brep.Curves3D.Count = 12
brep.Curves2D.Count = 48
brep.Surfaces.Count = 6
brep.Trims.Count = 24
brep.Loops.Count = 6
brep.Faces.Count = 6
brep.Edges.Count = 12
```

Example to extract the outline of a ***Brep*** ignoring all holes

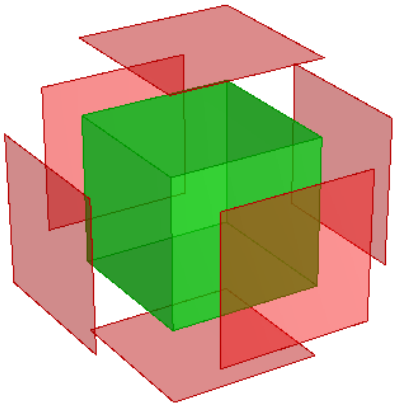
```
Brep brep = … //from input
//Declare outline list of curves
List<Curve> outline = new List<Curve>();
//Check all the loops and extract naked that are not inner
foreach (BrepLoop eLoop in brep.Loops) {
   //Make sure the loop type is outer
   if (eLoop.LoopType == BrepLoopType.Outer) {
      //Navigate through the trims of the loop
      foreach (BrepTrim trim in eLoop.Trims) {
         //Get the edge of each trim
         BrepEdge edge = trim.Edge;
```



```
            //Check if the edge has only one trim
            if (edge.TrimCount == 1) {
                //Add a copy of the edge curve to the list
                outline.Add(edge.DuplicateCurve());
            }
        }
    }
}
```

Example to extract all the faces of a ***Brep*** box and move them away from the center

```
Brep brep = … //from input
//Declare a new list of faces
List<Brep> faces = new List<Brep>();

//Find the center
Point3d center = brep.GetBoundingBox(true).Center;
for (int i = 0; i <= brep.Faces.Count - 1; i++) {
    //Extract the faces
    int[ ] iList = { i };
    Brep face = brep.DuplicateSubBrep(iList);
    //Find face center
    Point3d faceCenter = face.GetBoundingBox(true).Center;
    //Find moving direction
    Vector3d dir = new Vector3d();
    dir = faceCenter - center;
    //Move the face and add to the list
    face.Translate(dir);
    faces.Add(face);
}
```

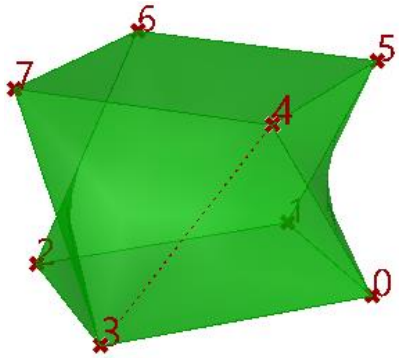


Create Brep objects:

The Brep class has many ***Create*** methods. For example, if you need to create a twisted box, then you can use the ***CreateFromBox*** method in the ***Brep*** class as in the following. You can also create a surface out of boundary curves or create a solid out of bounding surfaces.

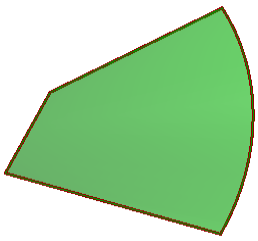
Create a ***Brep*** from corners

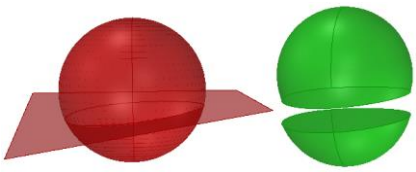
```
List<Point3d> corners = … // from input

//Create the brep from corners
Brep twistedBox = Brep.CreateFromBox(corners);
```



Create a ***Brep*** from bounding edge curves

```
List<Curve> crvs = … // from input

//Build the brep from edges
Brep edgeBrep = Brep.CreateEdgeSurface(crvs);
```



Create ***Breps*** from bounding surfaces	
List<**Brep**> breps = … // from input double tol = … // from input //Build the brep from corners **Brep**[ ] solids = Brep.CreateSolid(breps, tol);	

Brep methods:

Here is a list of some of the commonly used create methods found under the ***Brep*** class. For the full list and details about each method, please refer to the ***RhinoCommon SDK*** help.

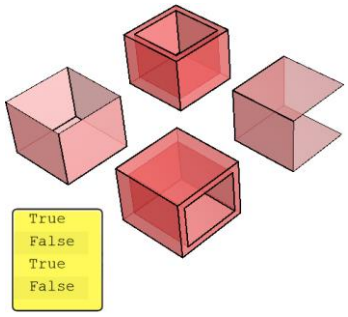
S	CreateBaseballSphere	Creates a brep representation of the sphere with two similar trimmed NURBS surfaces, and no singularities.
S	CreateBooleanDifference(Brep, Brep, Double)	Compute the Solid Difference of two Breps.
S	CreateBooleanIntersection(Brep, Brep, Double)	Compute the Solid Intersection of two Breps.
S	CreateBooleanUnion(IEnumerable<Brep>, Double)	Compute the Boolean Union of a set of Breps.
S	CreateEdgeSurface	Constructs a coons patch from 2, 3, or 4 curves.
S	CreateFromBox(IEnumerable<Point3d>)	Constructs new brep from 8 corner points.
S	CreateFromCone	Constructs a Brep representation of the cone with a single face for the cone, an edge along the cone seam, and vertices at the base and apex ends of this seam edge. The optional cap is a single face with one circular edge starting and ending at the base vertex.
S	CreateFromCornerPoints(Point3d, Point3d, Point3d, Double)	Makes a brep with one face.
S	CreateFromCornerPoints(Point3d, Point3d, Point3d, Point3d, Double)	make a Brep with one face.
S	CreateFromCylinder	Constructs a Brep definition of a cylinder.
S	CreateFromLoft	Constructs one or more Breps by lofting through a set of curves.
S	CreateFromMesh	Create a brep representation of a mesh
S	CreateFromRevSurface	Constructs a brep form of a surface of revolution.
S	CreateFromSweep(Curve, Curve, Boolean, Double)	Sweep1 function that fits a surface through a profile curve that define the surface cross-sections and one curve that defines a surface edge.
S	CreateFromSweep(Curve, Curve, Curve, Boolean, Double)	General 2 rail sweep. If you are not producing the sweep results that you are after, then use the SweepTwoRail class with options to generate the swept geometry.
S	CreateFromTaperedExtrude(Curve, Double, Vector3d, Point3d, Double, ExtrudeCornerType)	Extrude a curve to a taper making a brep (potentially more than 1)
S	CreateOffsetBrep	Offsets a Brep.
S	CreatePatch(IEnumerable<GeometryBase>, Surface, Double)	Constructs a brep patch. This is the simple version of fit that uses a specified starting surface.
S	CreatePipe(Curve, Double, Boolean, PipeCapMode, Boolean, Double, Double)	Creates a single walled pipe
S	CreateShell	Creates a hollowed out shell from a solid Brep. Function only operates on simple, solid, manifold Breps.
S	CreateSolid	Constructs closed polysurfaces from surfaces and polysurfaces that bound a region in space.
S	CreateTrimmedPlane(Plane, Curve)	Create a Brep trimmed plane.

The ***Brep*** methods serve multiple functions. Some are to extract information such as finding out if the brep is solid (closed polysurface), others perform calculations in relation to the brep such as area, volume or find a point inside a solid. Some methods change the brep such as cap

holes or merge coplanar faces. Methods that operate on multiple instances of breps include joining or performing some boolean operations. Following are examples to show the range of ***Brep*** methods.

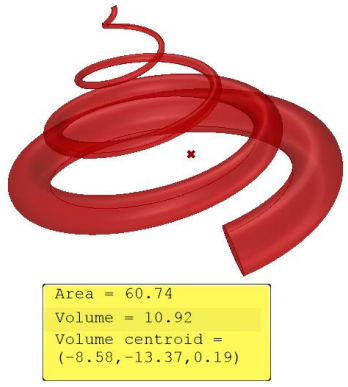
Example methods to extract ***Brep*** information: Find if a brep is valid and is a solid.

```
Brep[ ] breps = … // from input

List<bool> isSolid = new List<bool>();
foreach( Brep brep in breps)
    if( brep.IsValid )
        isSolid.Add(brep.IsSolid);
```



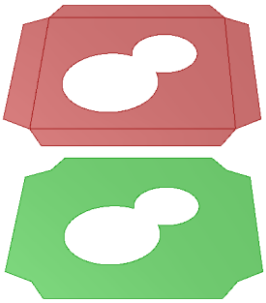
Example to calculate a brep area, volume and centroid

```
Brep brep = … // from input
//Declare variable
double area = 0;
double volume = 0;
Point3d vc…efault(Point3d);

//Create a new instance of the mass properties classes
VolumeMassProperties volumeMp = VolumeMassProperties.Compute(brep);
AreaMassProperties areaMp = AreaMassProperties.Compute(brep);

//Calculate area, volume and centroid
area = brep.GetArea();          //or use: area = areaMp.Area
volume = brep.GetVolume();  //or use: volume = volumeMp.Volume
centroid = volumeMp.Centroid;
```



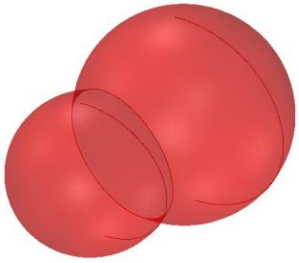
~~Example methods that change a brep topology: merge coplanar faces in a brep~~

```
Brep brep = … // from input
double tol = 0.01;

bool merged = brep.MergeCoplanarFaces(tol);
```



Example methods that operate on multiple breps: Boolean union 2 breps

```
List<Brep> breps = … // from input

//Boolean union the input breps
Brep[ ] unionBrep = Brep.CreateBooleanUnion(breps, tol);
```

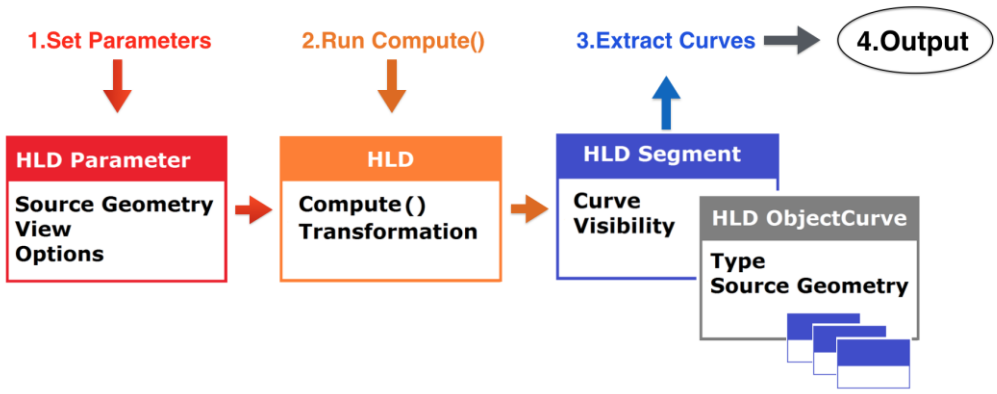


3_3_5: Other geometry classes

There are many important classes under the ***Rhino.Geometry*** namespace that are not derived from ***GeometryBase***. Those are commonly used when writing scripts to create and manipulate geometry. We used a couple of them in the examples. You can find the full list of ***RhinoCommon*** classes in the documentation. Most of these classes are derived directly from the ***C#*** abstract class ***System.Object***. This is the inheritance hierarchy for these classes.

System.Object	Abstract class in DotNet
AreaMassProperties	To calculate area, volume and centroid
Arrowhead	Create arrowhead
BezierCurve	Create a Bezier curve
BrepRegion	
BrepRegionFaceSide	
DevelopableSrf	Create a surface that can be unrolled flat
HiddenLineDrawing	All HiddenLineDrawing classes are related to ***Make2D*** functionality
HiddenLineDrawingObject	
HiddenLineDrawingObjectCurve	
HiddenLineDrawingParameter	
HiddenLineDrawingPoint	
HiddenLineDrawingSegment	
Matrix	A data structure that holds transformation information
MeshDisplacementInfo	
MeshExtruder	
MeshParameters	
MeshNgon	
MeshPart	
MeshPoint	

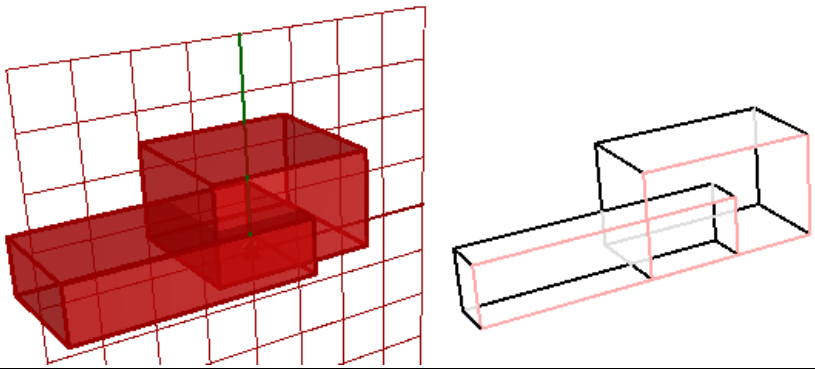
One important cluster of classes in this list has to do with extracting 2D outlines of the geometry, (***Make2D)***. These classes start with ***HiddenLineDrawing*** keyword. The following diagram shows the sequence of creating the drawing and which classes are used in each step.



Figure(29): Workflow and classes used to extract 2D drawing of the geometry

The first step involves setting the parameters such as the source geometry, view and all other options. Once the parameters are set, the HLD core class can calculate all the lines and hand back to the HLD Segment class with all visibility information and associated source. The user can then use this information to create the final drawing.

Create 2D drawing from 3D geometry (Make2D)



```
List<GeometryBase> geomList = … //from input
List<Plane> cplaneList = … //from input

//Use the active view in the Rhino document
var _view = doc.Views.ActiveView;
//Set Make2D Parameters
var _hldParams = new HiddenLineDrawingParameters
{
       AbsoluteTolerance = doc.ModelAbsoluteTolerance,
       IncludeTangentEdges = false,
       IncludeHiddenCurves = true
       };
   _hldParams.SetViewport(_view.ActiveViewport);

   //add objects to hld_param
   foreach (var geom in geomList){
     _hldParams.AddGeometry(geom, Transform.Identity, null);
}

//Add clipping planes
foreach (var cplane in cplaneList ){
     _hldParams.AddClippingPlane(cplane);
}

//perform HLD calculation
var  visibleList = new List<Curve>();
var  hiddenList = new List<Curve>();
var  secVisibleList = new List<Curve>();
var  secHiddenList = new List<Curve>();
var _hld = HiddenLineDrawing.Compute(_hldParams, true);
if (_hld != null)
{
   //transform
   var flatten = Transform.PlanarProjection(Plane.WorldXY);
   BoundingBox pageBox = _hld.BoundingBox(true);
   var delta2D = new Vector2d(0, 0);
   delta2D = delta2D - new Vector2d(pageBox.Min.X, pageBox.Min.Y);
   var delta3D = Transform.Translation(new Vector3d(delta2D.X, delta2D.Y, 0.0));
```

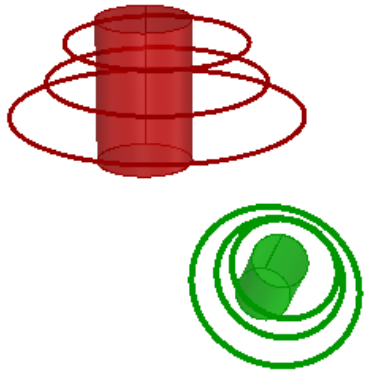
```
    flatten = delta3D * flatten;
    //add curves to lists
    foreach (HiddenLineDrawingSegment hldCurve in .Segments)
    {
      if (hldCurve == null ||
          hldCurve.ParentCurve == null ||
          hldCurve.ParentCurve.SilhouetteType == SilhouetteType.None)
        continue;
      var crv = hldCurve.CurveGeometry.DuplicateCurve();
      if (crv != null)
      {
        crv.Transform(flatten);
        if (hldCurve.SegmentVisibility == HiddenLineDrawingSegment.Visibility.Visible)
        {
          if (hldCurve.ParentCurve.SilhouetteType == SilhouetteType.SectionCut)
            secVisibleList.Add(crv);
          else
            visibleList.Add(crv);
        }
        else if (hldCurve.SegmentVisibility == HiddenLineDrawingSegment.Visibility.Hidden)
        {
          if (hldCurve.ParentCurve.SilhouetteType == SilhouetteType.SectionCut)
            secHiddenList.Add(crv);
          else
            hiddenList.Add(crv);
        }
      }
    }
}
```

3_4: Geometry transformations

All classes derived from ***GeometryBase*** inherit four transformation methods. The first three are probably the most commonly used which are ***Rotate***, ***Scale*** and ***Translate***. But there is also a generic ***Transform*** method that takes a ***Transform*** structure and can be set to any transformation matrix. The following example shows how to use the scale, rotate and translate methods on a list of geometry objects.

Use different transformation methods (scale, Rotate and Translate)

```
List<GeometryBase> objs = … //from input

//Create a new list of geometry objects
List<GeometryBase> newObjs = new List<GeometryBase>();

foreach (GeometryBase obj in objs) {
    //scale, rotate And move
    obj.Scale(factor);
    obj.Rotate(angle, Vector3d.YAxis, Point3d.Origin);
    obj.Translate(dir);

    //add to list
    newObjs.Add(obj);
}
```

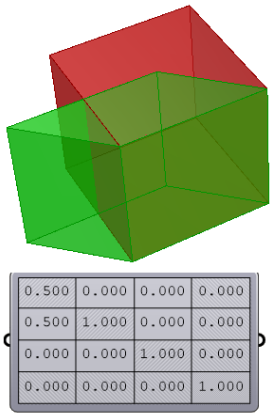


The ***Transform*** structure is a 4x4 matrix with several methods to help create geometry transformations. This includes defining a shear, projection, rotation, mirror and others. The structure also has functions that support operations such as multiplications and transpose. For

more information about the mathematics of transformations, please refer to “The Essential Mathematics for Computational Design”. The following examples show how to create a shear transform and planar projection.

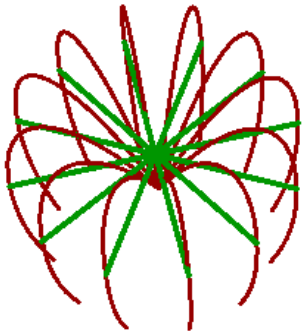
Create shear transformation and output the matrix

```
Brep brep = … //from input

//Create a shear Transform
Plane p = Plane.WorldXY;
var v = new Vector3d(0.5, 0.5, 0);
var y = Vector3d.YAxis;
var z = Vector3d.ZAxis;

var xform = Transform.Shear(p, v, y, z);

//shear the brep
brep.Transform(xform);
```



Create planar projection transform to project curves

```
List<GeometryBase> objs = … //from input

//Create a new list of geometry objects
List<GeometryBase> newObjs = new List<GeometryBase>();

foreach (GeometryBase obj in objs) {
    //Create a project transform
    obj.Transform(Transform.PlanarProjection(Plane.WorldXY));

    //add to list
    newObjs.Add(obj);
}
```



Chapter Four: Design Algorithms

4_1: Introduction

In this chapter we will implement a few examples to create mathematical curves and surfaces and solve a few generative algorithms using C# in Grasshopper.

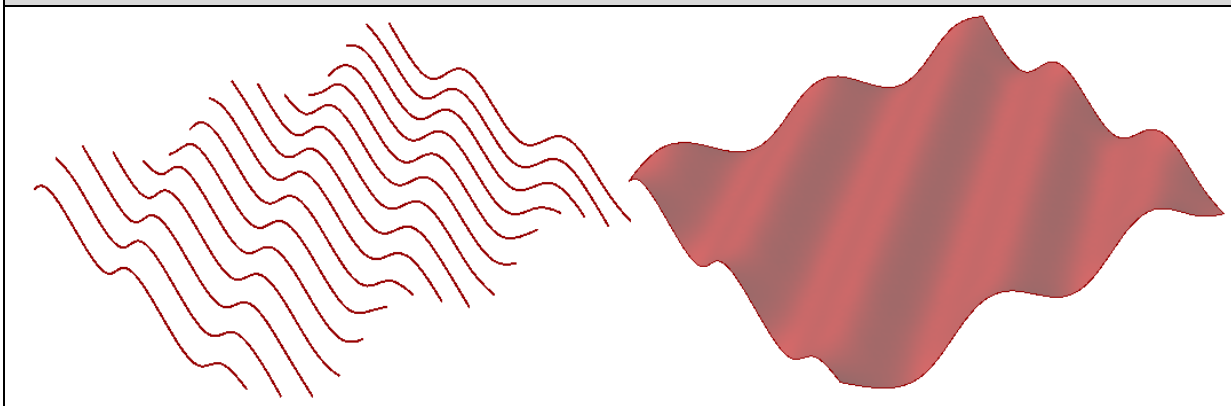
4_2: Geometry algorithms

Using scripting, it is relatively easy to create curves and surfaces that follow certain mathematical equations. You can generate control points to create smooth Nurbs, or interpolate points to connect and create the geometry.

4_2_1: Sine curves and surface

The following example shows how to create NurbsCurves and NurbsSurfaces and a lofted Brep using the sine of an angle.

Create curves and surface using the sine equation



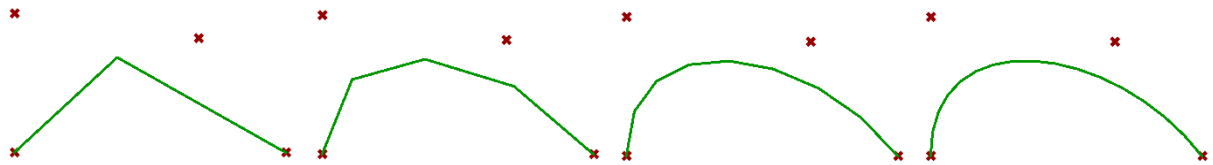
```
private void RunScript(int num, ref object OutCurves, ref object OutSurface, ref object OutLoft)
{
  //list of all points
  List<Point3d> allPoints = new List<Point3d>();
  //list of curves
  List<Curve> curves = new List<Curve>();

  for(int y = 0; y < num; y++)
  {
     //curve points
     List<Point3d> crvPoints= new List<Point3d>();
     for(int x = 0; x < num; x++)
     {
       double z = Math.Sin(Math.PI / 180 + (x + y));
       Point3d pt = new Point3d(x, y, z);
       crvPoints.Add(pt);
       allPoints.Add(pt);
     }
     //create a degree 3 nurbs curve from control points
     NurbsCurve crv = Curve.CreateControlPointCurve(crvPoints, 3);
     curves.Add(crv);
  }
```

```
  //create a nurbs surface from control points
  NurbsSurface srf = NurbsSurface.CreateFromPoints(allPoints, num, num, 3, 3);

  //create a loft brep from curves
  Brep[ ] breps = Brep.CreateFromLoft(curves, Point3d.Unset, Point3d.Unset, LoftType.Tight, false);

  //Assign output
  OutCurves = curves;
  OutSurface = srf;
  OutLoft = breps;
}
```

4_2_2: De Casteljau algorithm to interpolate a Bezier curve

You can create a cubic Bezier curve from four input points. De Casteljau algorithm is used in computer graphics to evaluate the Bezier curve at any parameter. If evaluated at multiple parameters, then the points can be connected to draw the curve any resolution. The following example shows a recursive function implementation to interpolate through a Bezier curve.

De Casteljau algorithm to draw a Bezier curve with 2, 4, 8 and 16 segments



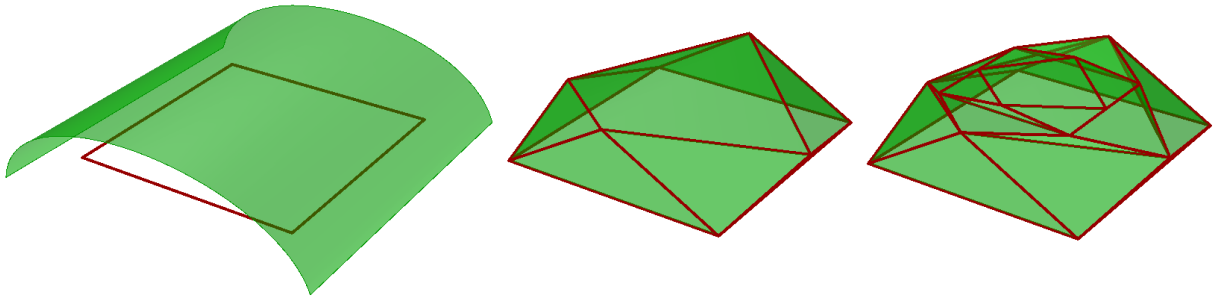
```
private void RunScript(Point3d pt0, Point3d pt1, Point3d pt2, Point3d pt3, int segments, ref object BezierCrv)
{
   if(segments < 2)
     segments = 2;

   List<Point3d> bezierPts = new List<Point3d>();
   bezierPts.Add(pt0);
   bezierPts.Add(pt1);
   bezierPts.Add(pt2);
   bezierPts.Add(pt3);

   List<Point3d> evalPts = new List<Point3d>();
   double step = 1 / (double) segments;
   for(int i = 0; i <= segments; i++)
   {
     double t = i * step;
     Point3d pt = Point3d.Unset;
     EvalPoint(bezierPts, t, ref pt);
     if(pt.IsValid)
       evalPts.Add(pt);
   }
   Polyline pline = new Polyline(evalPts);
   BezierCrv = pline;
}
```

```
void EvalPoint(List<Point3d> points, double t, ref Point3d evalPt)
{
   //stopping condition - point at parameter t is found
   if(points.Count < 2)
     return;
   List<Point3d> tPoints = new List<Point3d>();
   for(int i = 1; i < points.Count; i++)
   {
```

```
      Line line = new Line(points[i - 1], points[i]);
      Point3d pt = line.PointAt(t);
      tPoints.Add(pt);
    }
    if(tPoints.Count == 1)
      evalPt = tPoints[0];
    EvalPoint(tPoints, t, ref evalPt);
}
```

4_2_3: Simple subdivision mesh

The following example takes a surface and closed polyline, then creates a subdivision mesh. It pulls the mid points of the polyline edges to the surface to then subdivide and pull again.



```
private void RunScript(Surface srf, List<Polyline> inPolylines, int degree, ref object OutPolylines, ref object
OutMesh)
{
    //instantiate the collection of all panels
    List<Polyline> outPanels = new List<Polyline>();
    //limit to 6 subdivisions
    if( degree > 6)
      degree = 6;
    for(int i = 0; i < degree; i++)
    {
      //outer polylines
      List<Polyline> plines = new List<Polyline>();
      //mid polylines
      List<Polyline> midPlines = new List<Polyline>();
      //generate subdivided panels
      bool result = SubPanelOnSurface(srf, inPolylines.ToArray(), ref plines, ref midPlines);
      if( result == false)
        break;
      //add outer panels
      outPanels.AddRange(plines);
      //add mid panels only in the last iteration
      if(i == degree - 1)
        outPanels.AddRange(midPlines);
      else //subdivide mid panels only
        inPolylines = midPlines;
    }
    //Create a mesh from all polylines
    Mesh joinedMesh = new Mesh();
    for(int i = 0; i < outPanels.Count; i++)
    {
      Mesh mesh = Mesh.CreateFromClosedPolyline(outPanels[i]);
      joinedMesh.Append(mesh);
    }
    //make sure all mesh faces normals are in the same general direction
```

```
    joinedMesh.UnifyNormals();

    //Assign output
    OutPolylines = outPanels;
    OutMesh = joinedMesh;
}
```

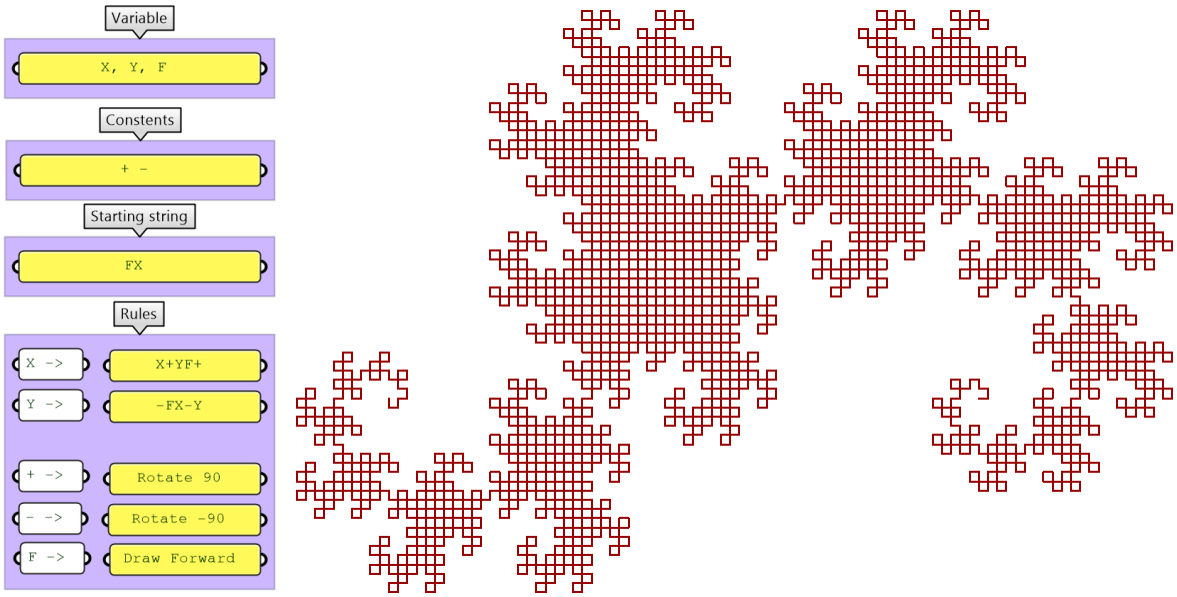
```
bool SubPanelOnSurface( Surface srf, Polyline[] inputPanels, ref List<Polyline> outPanels, ref List<Polyline>
midPanels)
{
    //check for a valid input
    if (inputPanels.Length == 0 || null == srf)
      return false;
    for (int i = 0; i < inputPanels.Length; i++)
    {
      Polyline ipline = inputPanels[i];
      if (!ipline.IsValid || !ipline.IsClosed)
        continue;
      //stack of points
      List<Point3d> stack = new List<Point3d>();
      Polyline newPline = new Polyline();
      for (int j = 1; j < ipline.Count; j++)
      {
        Line line = new Line(ipline[j - 1], ipline[j]);
        if (line.IsValid)
        {
          Point3d mid = line.PointAt(0.5);
          double s, t;
          srf.ClosestPoint(mid, out s, out t);
          mid = srf.PointAt(s, t);
          newPline.Add(mid);
          stack.Add(ipline[j - 1]);
          stack.Add(mid);
        }
      }
      //add the first 2 point to close last triangle
      stack.Add(stack[0]);
      stack.Add(stack[1]);
      //close
      newPline.Add(newPline[0]);
      midPanels.Add(newPline);

      for (int j = 2; j < stack.Count; j = j + 1)
      {
        Polyline pl = new Polyline { stack[j - 2], stack[j - 1], stack[j], stack[j - 2] };
        outPanels.Add(pl);
      }
    }
    return true;
}
```

4_3: Generative algorithms

Most of the generative algorithms require recursive functions that are only possible through scripting in Grasshopper. The following are four examples of generative solutions to generate the dragon curve, fractals, penrose tiling and game of live.

4_3_1: Dragon curve

Dragon curve




```
private void RunScript(string startString, string ruleX, string ruleY, int Num, double Length, ref object
DragonCurve)
{
  // declare string
  string dragonString = startString;
  // generate the string
  GrowString(ref Num, ref dragonString , ruleX, ruleY);
  //generate the points
  List<Point3d> dragonPoints = new List<Point3d>();;
  ParseDeagonString(dragonString, Length, ref dragonPoints);
  // create the curve
  PolylineCurve dragonCrv= new PolylineCurve(dragonPoints);
  // assign output
  DragonCurve = dragonCrv;
}
```

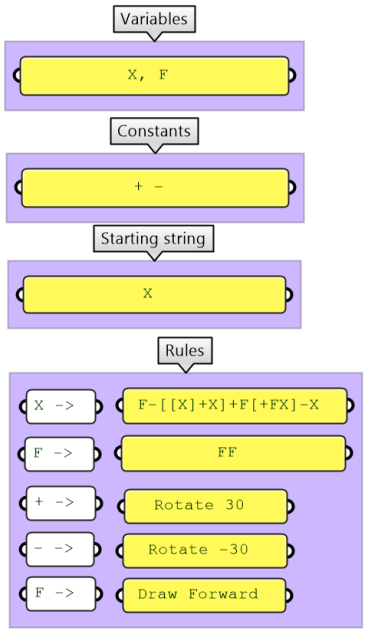
```
void GrowString(ref int Num, ref string finalString, string ruleX, string ruleY)
 {
  // decrement the count with each new execution of the grow function
  Num = Num - 1;
  char rule;
  // create new string
  string newString = "";
  for (int i = 0; i < finalString.Length ; i++)
  {
```

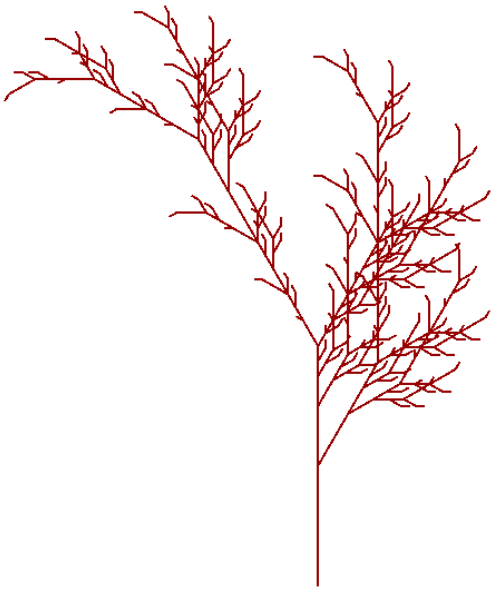
```
      rule = finalString[i];
      if (rule == 'X')
        newString = newString + ruleX;
      if (rule == 'Y')
        newString = newString + ruleY;
      if (rule == 'F' | rule == '+' | rule == '-')
        newString = newString + rule;
    }
    finalString = newString;
    // stopper condition
    if (Num == 0)
      return;
    // grow again
    GrowString(ref Num, ref finalString, ruleX, ruleY);
}
```

```
void ParseDeagonString(string dragonString, double Length, ref List<Point3d> dragonPoints)
{
    //parse instruction string to generate points
    //let base point be world origin
    Point3d pt = Point3d.Origin;
    dragonPoints .Add(pt);

    //drawing direction vector - strat along the x-axis
    //vector direction will be rotated depending on (+,-) instructions
    Vector3d V = new Vector3d(1.0, 0.0, 0.0);

    char rule;
    for(int i = 0 ; i < dragonString.Length;i++)
    {
      //always start for 1 and length 1 to get one char at a time
      rule = DragonString[i];
      //move Forward using direction vector
      if( rule == 'F')
      {
        pt = pt + (V * Length);
        dragonPoints.Add(pt);
      }
      //rotate Left
      if( rule == '+')
        V.Rotate(Math.PI / 2, Vector3d.ZAxis);
      //rotate Right
      if( rule == '-')
        V.Rotate(-Math.PI / 2, Vector3d.ZAxis);
    }
}
```

4_3_2: Fractal tree

Fractal tree





```
private void RunScript(string startString, string ruleX, string ruleY, int num, double length, ref object FractalLines)
{
    // declare string
    string fractalString = startString;
    // generate the string
    GrowString(ref num, ref dragonString, ruleX, ruleY);
    //generate the points
    List<Line> fractalLines = new List<Line>();
    ParsefractalString(fractalString, length, ref fractalLines );
    // assign output
    FractalLines = fractalLines ;
}
```

```
void GrowString(ref int num, ref string finalString, string ruleX, string ruleF)
{
    // Decrement the count with each new execution of the grow function
    num = num - 1;
    char rule;
    // Create new string
    string newString = "";
    for (int i = 0; i < finalString.Length ; i++)
    {
      rule = finalString[i];
      if (rule == 'X')
        newString = newString + ruleX;
      if (rule == 'F')
        newString = newString + ruleF;
      if (rule == '[' || rule == ']' || rule == '+' || rule == '-')
        newString = newString + rule;
    }
    finalString = newString;
    // Stopper condition
```

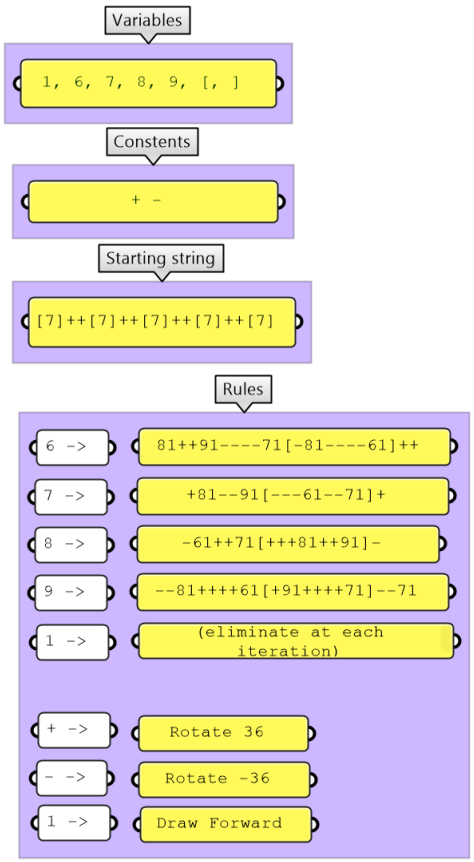
```
    if (num == 0)
      return;
    // Grow again
    GrowString(ref num, ref finalString, ruleX, ruleF);
}
```

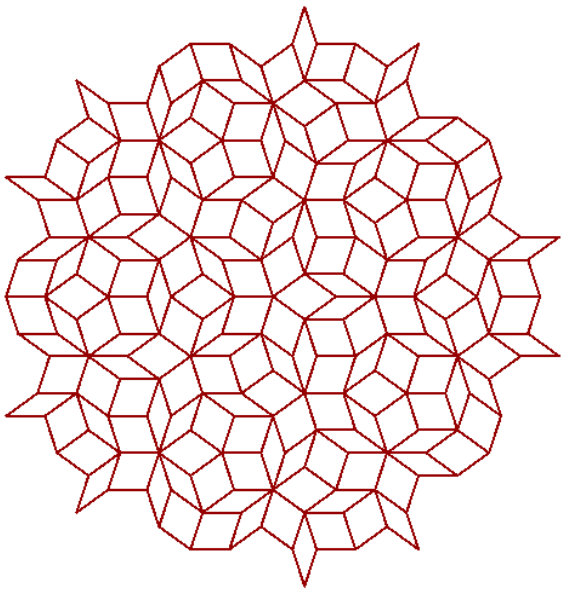
```
void ParsefractalString(string fractalString, double length, ref List<Line> fractalLines)
{
    //Parse instruction string to generate points
    //Let base point be world origin
    Point3d pt = Point3d.Origin;

    //Declare points array
    //Vector rotates with (+,-) instructions by 30 degrees
    List<Point3d> arrPoints = new List<Point3d>();

    //Draw forward direction
    //Vector direction will be rotated depending on (+,-) instructions
    Vector3d vec = new Vector3d(0.0, 1.0, 0.0);

    //Stacks of points and vectors
    List<Point3d> ptStack = new List<Point3d>();
    List<Vector3d> vStack = new List<Vector3d>();

    //Declare loop variables
    char rule;
    for(int i = 0 ; i < fractalString.Length; i++)
    {
      //Always start for 1 and length 1 to get one char at a time
      rule = fractalString[i];
      //Rotate Left
      if( rule == '+')
        vec.Rotate(Math.PI / 6, Vector3d.ZAxis);
      //Rotate Right
      if( rule == '-')
        vec.Rotate(-Math.PI / 6, Vector3d.ZAxis);
      //Draw Forward by direction
      if( rule == 'F')
      {
        //Add current points
        Point3d newPt1 = new Point3d(pt);
        arrPoints.Add(newPt1);
        //Calculate next point
        Point3d newPt2 = new Point3d(pt);
        newPt2 = newPt2 + (vec * length);
        //Add next point
        arrPoints.Add(newPt2);
        //Save new location
        pt = newPt2;
      }
      //Save point location
      if( rule == '[')
      {
        //Save current point and direction
        Point3d newPt = new Point3d(pt);
        ptStack.Add(newPt);

        Vector3d newV = new Vector3d(vec);
        vStack.Add(newV);
      }
      //Retrieve point and direction
```

```
      if( rule == ']')
      {
        pt = ptStack[ptStack.Count - 1];
        vec = vStack[vStack.Count - 1];
        //Remove from stack
        ptStack.RemoveAt(ptStack.Count - 1);
        vStack.RemoveAt(vStack.Count - 1);
      }
    }
    //Generate lines
    List<Line> allLines = new List<Line>();
    for(int i = 1; i < arrPoints.Count; i = i + 2)
    {
      Line line = new Line(arrPoints[i - 1], arrPoints[i]);
      allLines.Add(line);
    }
}
```

4_3_3: Penrose tiling

Penrose tiling





```
private void RunScript(string startString, string rule6, string rule7, string rule8, string rule9, int num, ref object
PenroseString)
 {
   // Declare string
   string finalString;
   finalString = startString;
```

```
  // Generate the string
  GrowString(ref num, ref finalString, rule6, rule7, rule8, rule9);
  // Return the string
  PenroseString = finalString;
}
```

```
void GrowString(ref int num, ref string finalString, string rule6, string rule7, string rule8, string rule9)
{
  // Decrement the count with each new execution of the grow function
  num = num - 1;
  char rule;

  // Create new string
  string newString = "";
  for (int i = 0; i < finalString.Length; i++)
  {
    rule = finalString[i];
    if (rule == '6')
      newString = newString + rule6;
    if (rule == '7')
      newString = newString + rule7;
    if (rule == '8')
      newString = newString + rule8;
    if (rule == '9')
      newString = newString + rule9;

    if (rule == '[' || rule == ']' || rule == '+' || rule == '-')
      newString = newString + rule;
  }
  finalString = newString;

  // Stopper condition
  if (num == 0)
    return;

  // Grow again
  GrowString(ref num, ref finalString, rule6, rule7, rule8, rule9);
}
```

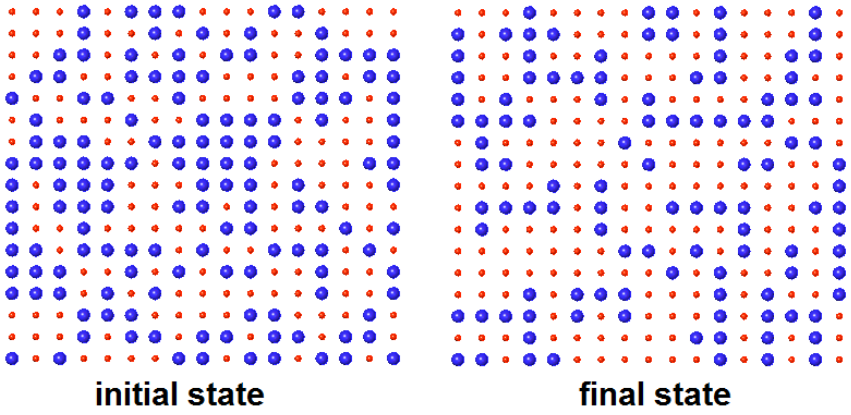
```
private void RunScript(string penroseString, double length, ref object PenroseLines)
{
  //Parse instruction string to generate points
  //Let base point be world origin
  Point3d pt = Point3d.Origin;

  //Declare points array
  //Vector rotates with (+,-) instructions by 36 degrees
  List<Point3d> arrPoints = new List<Point3d>();

  //Draw forward direction
  //Vector direction will be rotated depending on (+,-) instructions
  Vector3d vec = new Vector3d(1.0, 0.0, 0.0);

  //Stacks of points and vectors
  List<Point3d> ptStack = new List<Point3d>();
  List<Vector3d> vStack = new List<Vector3d>();

  //Declare loop variables
  char rule;
```

```
  for(int i = 0 ; i < penroseString.Length; i++)
  {
    //Always start for 1 and length 1 to get one char at a time
    rule = penroseString[i];
    //Rotate Left
    if( rule == '+')
      vec.Rotate(36 * (Math.PI / 180), Vector3d.ZAxis);
    //Rotate Right
    if( rule == '-')
      vec.Rotate(-36 * (Math.PI / 180), Vector3d.ZAxis);
    //Draw Forward by direction
    if( rule == '1')
    {
      //Add current points
      Point3d newPt1 = new Point3d(pt);
      arrPoints.Add(newPt1);
      //Calculate next point
      Point3d newPt2 = pt + (vec * length);
      //Add next point
      arrPoints.Add(newPt2);
      //Save new location
      pt = newPt2;
    }

    //Save point location
    if( rule == '[')
    {
      //Save current point and direction
      Point3d newPt = new Point3d(pt);
      ptStack.Add(newPt);

      Vector3d newVec = new Vector3d(vec);
      vStack.Add(newVec);
    }

    //Retrieve point and direction
    if( rule == ']')
    {
      pt = ptStack[ptStack.Count - 1];
      vec = vStack[vStack.Count - 1];

      //Remove from stack
      ptStack.RemoveAt(ptStack.Count - 1);
      vStack.RemoveAt(vStack.Count - 1);
    }
  }

  //Generate lines
  List<Line> allLines = new List<Line>();
  for(int i = 1; i < arrPoints.Count; i = i + 2)
  {
    Line line = new Line(arrPoints[i - 1], arrPoints[i]);
    allLines.Add(line);
  }

  PenroseLines = allLines;
}
```

4_3_4: Conway game of live

A cellular automaton consists of a regular grid of cells, each in one of a finite number of states, "On" and "Off" for example. The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighborhood (usually including the cell itself) is defined relative to the specified cell. For example, the neighborhood of a cell might be defined as the set of cells a distance of 2 or less from the cell. An initial state (time t=0) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood.
Check wikipedia for full details and examples.

Conway game of live




```
private void RunScript(Surface srf, int uNum, int vNum, int seed, ref object PointGrid, ref object StateGrid)
{
  if(uNum < 2)
    uNum = 2;
  if(vNum < 2)
    vNum = 2;

  double uStep = srf.Domain(0).Length / uNum;
  double vStep = srf.Domain(1).Length / vNum;
  double uMin = srf.Domain(0).Min;
  double vMin = srf.Domain(1).Min;

  //create a grid of points and a grid of states
  DataTree<Point3d> pointsTree = new DataTree<Point3d>();
  DataTree<int> statesTree = new DataTree<int>();
  int pathIndex = 0;
  Random rand = new Random(seed);
  for (int i = 0; i <= uNum; i++)
  {
    List<Point3d> ptList = new List<Point3d>();
    List<int> stateList = new List<int>();
    GH_Path path = new GH_Path(pathIndex);
    pathIndex = pathIndex + 1;
```

```
    for (int j = 0; j <= vNum; j++)
    {
      Point3d srfPt = srf.PointAt(uMin + i * uStep, vMin + j * vStep);
      ptList.Add(srfPt);
      int randState = rand.Next(0, 2);
      stateList.Add(randState);
    }
    pointsTree.AddRange(ptList, path);
    statesTree.AddRange(stateList, path);
  }

  PointGrid = pointsTree;
  StateGrid = statesTree;
}
```

```
private void RunScript(DataTree<int> grid, int gen, ref object OutGrid)
{
  //Get state at the defined generation
  for(int i = 0; i < gen; i++)
    grid = NewGeneration(grid);

  OutGrid = grid;
}
```

```
  public DataTree<int> NewGeneration(DataTree<int> inStates)
{
  int i, j, c, nc;
  List<int> prvBranch;
  List<int> nxtBranch;
  List<int> branch;
  DataTree<int> states = new DataTree<int>();
  states = inStates;

  for (i = 0; i <= states.Branches.Count - 1; i++)
  {
    branch = states.Branches[i];
    for (j = 0; j <= branch.Count - 1; j++)
    {
      c = branch[j];
      nc = 0;

      // Check neighbouring states
      // next
      nc = nc + branch[(j + 1 + branch.Count) % branch.Count];
      // prv
      nc = nc + branch[(j - 1 + branch.Count) % branch.Count];

      // top
      nxtBranch = states.Branches[(i + 1 + states.Branches.Count) % states.Branches.Count];
      nc = nc + nxtBranch[(j + 1 + nxtBranch.Count) % nxtBranch.Count];
      nc = nc + nxtBranch[(j + nxtBranch.Count) % nxtBranch.Count];
      nc = nc + nxtBranch[(j - 1 + nxtBranch.Count) % nxtBranch.Count];

      // bottom
      prvBranch = states.Branches[(i - 1 + states.Branches.Count) % states.Branches.Count];
      nc = nc + prvBranch[(j + 1 + prvBranch.Count) % prvBranch.Count];
      nc = nc + prvBranch[(j + prvBranch.Count) % prvBranch.Count];
      nc = nc + prvBranch[(j - 1 + prvBranch.Count) % prvBranch.Count];

      // set the new state
```

```
      if (c == 1)
      {
        if (nc < 2 | nc > 3)
          c = 0;
      }
      else if (c == 0)
      {
        if (nc == 3)
          c = 1;
      }
      branch[j] = c;
    }
  }
  return states;
}
```