Week 3 – Objects & the Rhino Common Library

	T	Overall topic	Coding / Geometry topic
1	10-Jan	*Intro*	*Grasshopper C# component*
2	17-Jan	*C# Language*	*C# language overview*
3	24-Jan	*Geometry - Classes & Rhino Common*	*Classes, inheritance, memory, Rhino common*
4	31-Jan	*Linear algebra & transforms*	*nD arrays, dot & cross products, transforms*
5	7-Feb	*Curves & surfaces 1*	*Parametric curve & surface geometry*
6	14-Feb	*Surface Algorithms - Recursion*	*Recursion algorithms*
7	21-Feb		
8	28-Feb	*Surface Algorithms - NURBS & Subdivision Surfaces*	*NURBS Surface geometry*
9	7-Mar		
10	14-Mar	*Surface Curvature & Developable Surfaces*	*Surface curvature*
11	21-Mar	*Triangulation & Vornoi*	*Triangulation, closet point, etc. algorithms*
12	28-Mar	*Solids & Voxel algorithms*	*Voxel arrrays*
13	4-Apr	*Physics based modeling*	*Particle physics systems*
14	11-Apr	*Web: standing up a server*	*Web services, javascript*
15	18-Apr	*Rhino3DM.JS*	*Rhino3DM*
16	25-Apr	*ThreeJS, Observable, etc.*	*Rhino3DM & Three.js*
17	2-May		

Part 1
A few additional C# language constructs

- Enum
- Struct
- Classes, instances & instance

Part 2
Rhino Common Geometry

- **Point3d**
- **Vectors**
- **Lightweight Curves**
- **Lightweight Surfaces**

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.

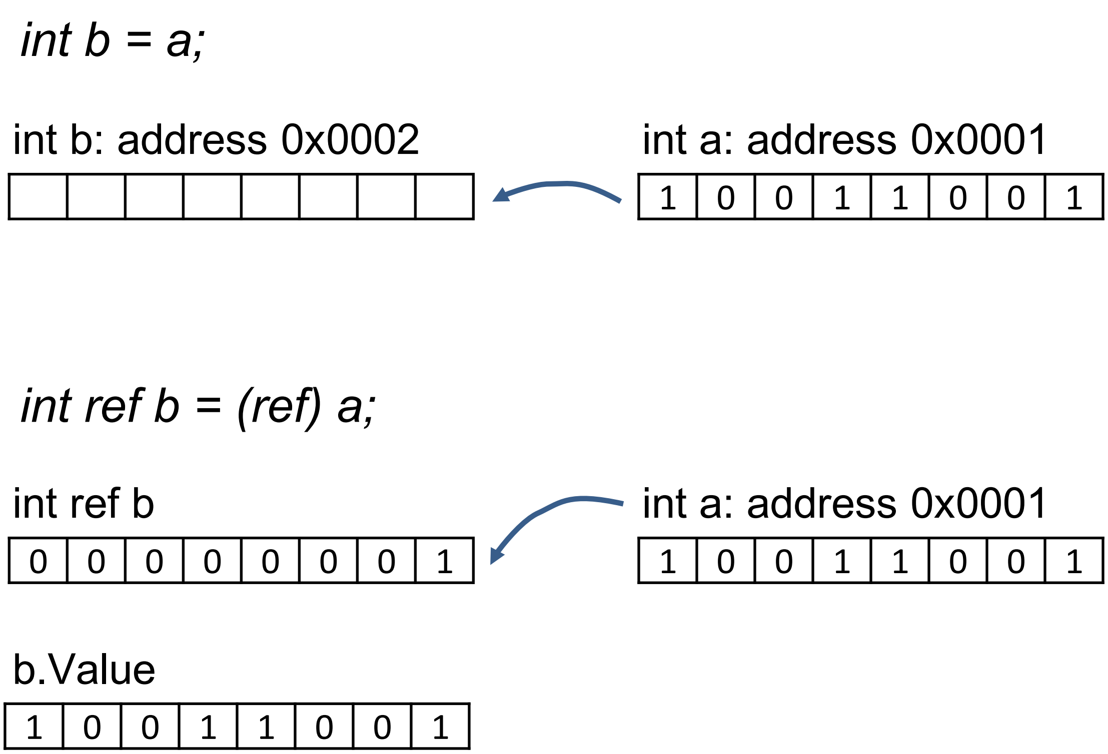
```
enum Traffic
{
  Red = 1,
  Yellow = 2,
  Green = 3
}
Traffic signal = Traffic.Yellow;
if( signal == Traffic.Red)
    Print("STOP");
else if(signal == Traffic.Yellow)
    Print("YIELD");
else if(signal == Traffic.Green)
    Print("GO");
else
    Print("This is not a traffic signal color!");
```

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

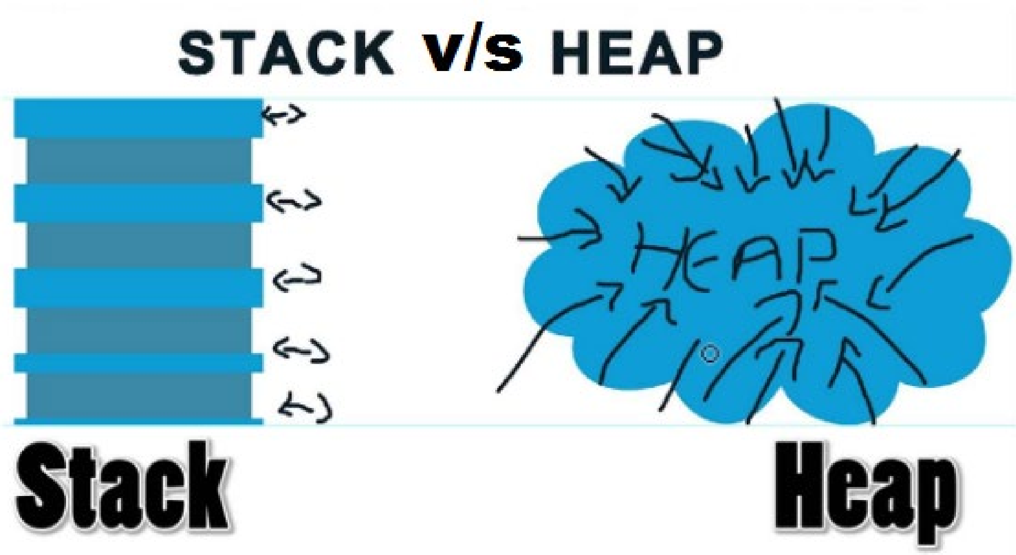
2_9_4: Value vs reference types

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.



How memory is managed



How memory is managed

>>

```
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
Utility functions

Members

  /**/
  private void RunScript(object x, object y, ref object A)
  {

    int RunScriptVariable = 1; // A variable in the scope of myFunction: Goes on the STACK
    int a = 1; // A variable in the scope of RunScript: Goes on the STACK

    Print("RunScript a: " + a);
    myFunction();

    A = a;

  }

  // <Custom additional code>

  // GLOBAL VARIABLES
  int globalVariable = 0;  // A Global variable: Goes on the HEAP
  int a = 0;  // A Global variable: Goes on the HEAP

  // A (Globally defined) Function
  void myFunction() {
    int myFunctionVariable = 2; // A variable in the scope of myFunction: Goes on the STACK
    int a = 2; // A variable in the scope of myFunction: Goes on the STACK
    Print("myFunction a: " + a);

  }

  // </Custom additional code>
}
```

Stack		Heap
Global / Script_Instance Scope		
globalVariable	0	
a	0	
RunScript() Scope		
RunScriptVariable	1	
a	1	
myFunction() Scope		
myFunctionVariable	2	
a	2	

User Defined Data Types - Structs

2_9_2: Structures

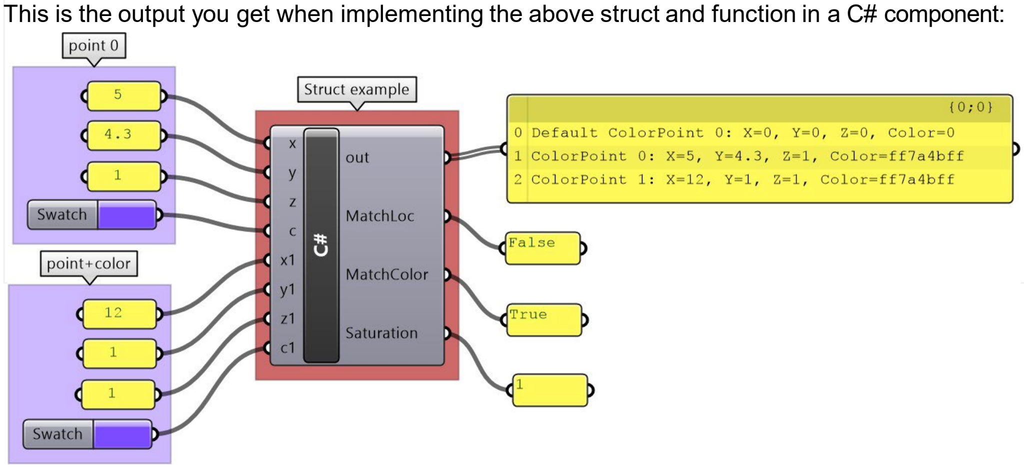
A structure is used to define a new ***value-type***. In C# programming , we use the keyword ***struct*** to define new structure. The following is an example of a simple structure that defines a number of variables (fields) to create a custom type of a colored 3D point. We use ***private*** access to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

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:



Procedural Programming

There are **data** and **operations** on data.

Data

- All data are of a type

```
int a;      // a variable
```

Operations

- take 1 or more data as inputs
- do something internally
- return data as output

```
add(a, 6);            // add two numbers, returns a number
```

Object oriented programming

Procedural Programming

There are **data** and **operations** on data.

Data

- All data are of a type

```
int a;          // a variable
```

Operations

- take 1 or more data as inputs
- do something internally
- return data as output

```
add(a, 6);                  // add two numbers, returns a number
```

Versus **Object Oriented Programming**

There are object **classes** and **instances**

```
MySpecialInteger a;       // a is an instance of the
                                                          //
MySpecialInteger class
```

There are object **state** valuables and **methods**

```
a.value;                  // returns the internal value of a

a.Add(b)      // asks a to do something
```

Data and operations are packaged together.

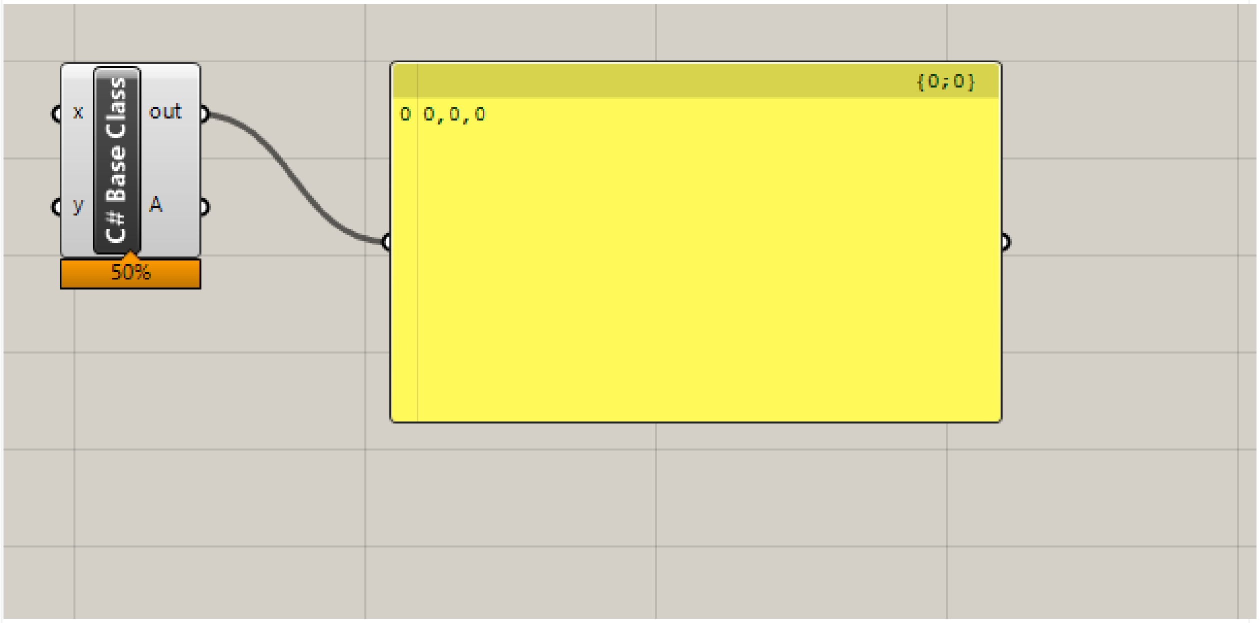
A dot (.) signals we are asking the object to do something, or "passing it a message"

Examples

```
myCar.drive();

Sally.drive(myCar);



BuildingElement myElement;

myElement.draw();

myElement.delete();
```

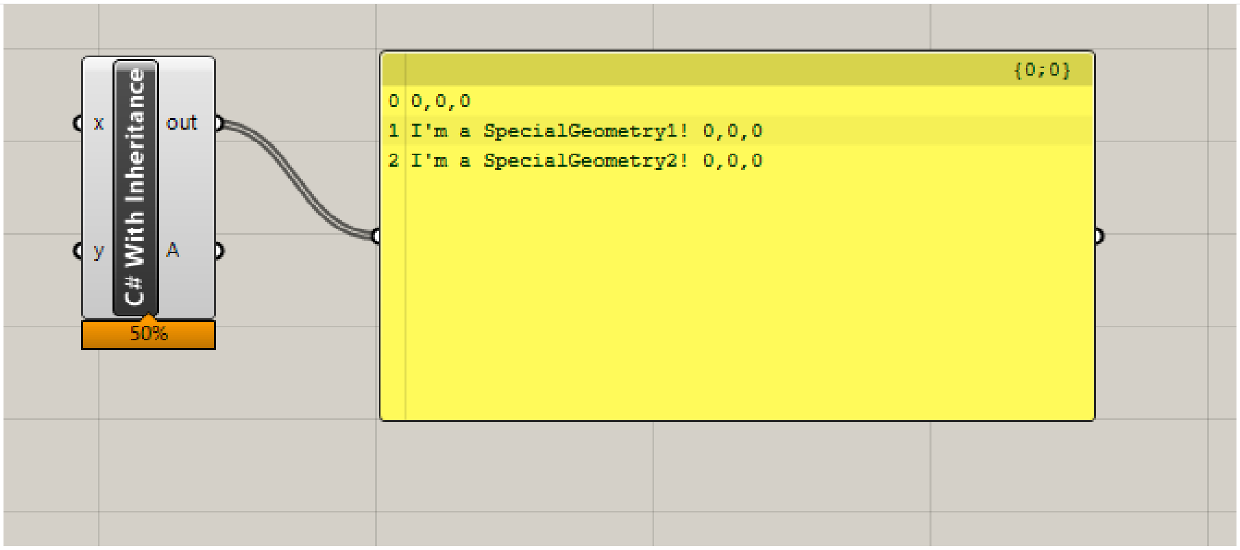
Simple C# example

>>




```
public class Script_Instance : GH_ScriptInstance
{
Utility functions

Members

  /**/
  private void RunScript(object x, object y, ref object A)
  {

    DP2Geometry myGeo = new DP2Geometry();
    myGeo.Print(this);

  }

  /**/

  class DP2Geometry {
    private Point3d location;

    public Point3d Location {
      get {return location;}
      set {location = value;}
    }


    public DP2Geometry() {
      Location = new Point3d(0, 0, 0);
    }

    public void Print(Script_Instance gh_script) {
      //Rhino.RhinoApp.WriteLine(location.ToString());
      gh_script.Print(location.ToString());
    }

  }
```

Simple C# example – with inheritance




```
private void RunScript(object x, object y, ref object A)
{

  DP2Geometry myGeo = new DP2Geometry();
  myGeo.Print(this);

  myGeo = new DP2SpecialGeometry1();
  myGeo.Print(this);

  myGeo = new DP2SpecialGeometry2();
  myGeo.Print(this);

}

// <Custom additional code>

class DP2Geometry {
  private Point3d location;

  public Point3d Location {
    get {return location;}
    set {location = value;}
  }


  public DP2Geometry() {
    Location = new Point3d(0, 0, 0);
  }

  public virtual void Print(Script_Instance gh_script) {
    //Rhino.RhinoApp.WriteLine(location.ToString());
    gh_script.Print(location.ToString());
  }

}

/////////// DP2SpecialGeometry1 Class ///////////
class DP2SpecialGeometry1 : DP2Geometry {

  public override void Print(Script_Instance gh_script) {
    //Rhino.RhinoApp.WriteLine(location.ToString());
    gh_script.Print("I'm a SpecialGeometry1! " + Location.ToString());
  }

}
```

Struct vs. Class

Structs are light versions of classes. Structs are value types and can be used to create objects that behave like built-in types.

Structs share many features with classes but with the following limitations as compared to classes.

Struct cannot have a default constructor (a constructor without parameters) or a destructor.

Structs are value types and are copied on assignment.

Structs are value types while classes are reference types.

Structs can be instantiated without using a new operator.

A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from System.ValueType, which inherits from System.Object.

Struct cannot be a base class. So, Struct types cannot abstract and are always implicitly sealed.

Abstract and sealed modifiers are not allowed and struct member cannot be protected or protected internals.

Function members in a struct cannot be abstract or virtual, and the override modifier is allowed only to the override methods inherited from System.ValueType.

Struct does not allow the instance field declarations to include variable initializers. But, static fields of a struct are allowed to include variable initializers.

A struct can implement interfaces.

A struct can be used as a nullable type and can be assigned a null value.

---

https://www.c-sharpcorner.com/blogs/difference-between-struct-and-class-in-c-sharp

Values vs reference types: structs & classes

Pass by value

Point3d a = {1,2,3};
Point3d b = a;

int b: address 0x0002

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0

int a: address 0x0001

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0

Pass by reference

Point3d a = {1,2,3};
Point3d ref b = (ref) a;
Print b.Value

int ref b

0	0	0	0	0	0	0	1

int a: address 0x0001

1	0	0	1	1	0	0	1

b.Value

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0

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

3_2: Geometry structures

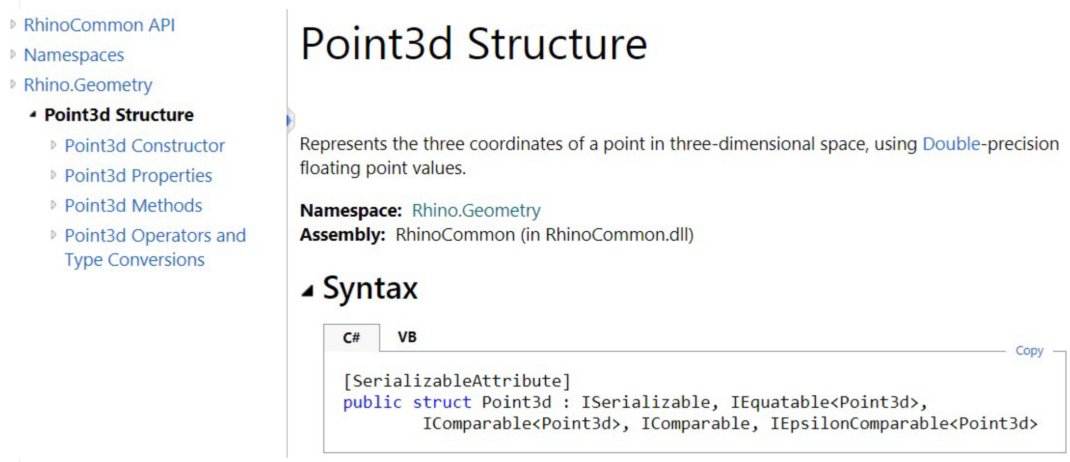
RhinoCommon defines basic geometry types using structures. We will dissect the ***Point3d*** structure and show how to read in the documentation and use it in a script. This should help you navigate and use other structures. Below is a list of the geometry structures.

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

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:

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]

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

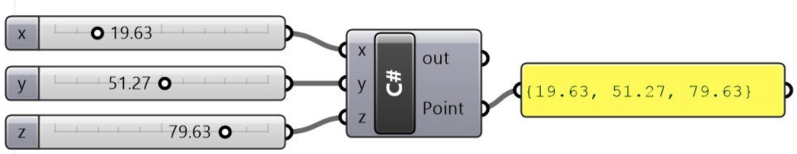
Point3d Constructors

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.




```
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;
}
```

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

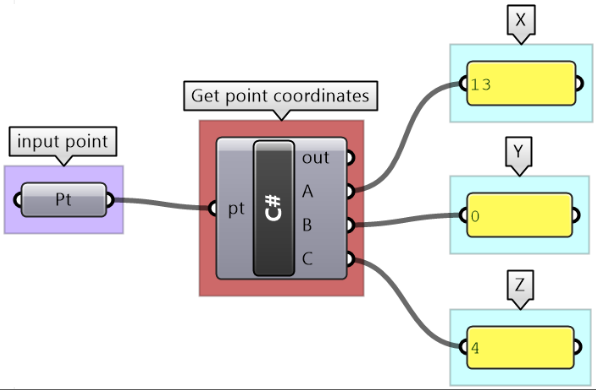
Point3d Properties

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:

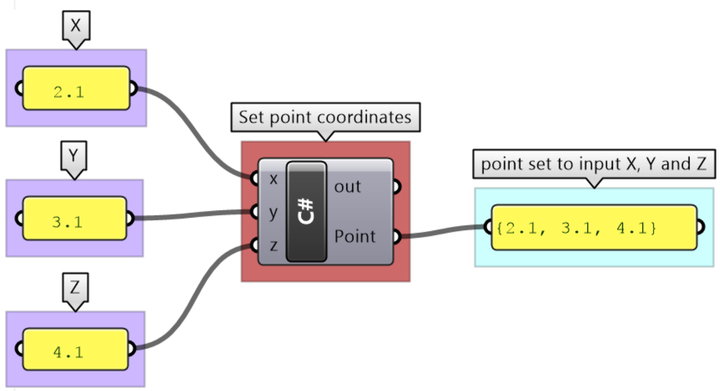
Properties

	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***.

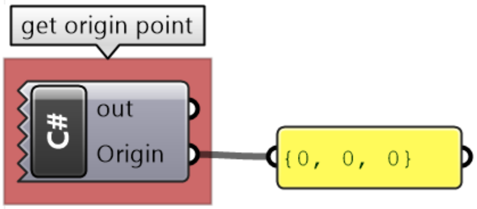



```
private void RunScript(Point3d pt, ref object A, ref object B, ref object C)
{
    //Assign the point coordinates to output
    A = pt.X;
    B = pt.Y;
    C = pt.Z;

}
```



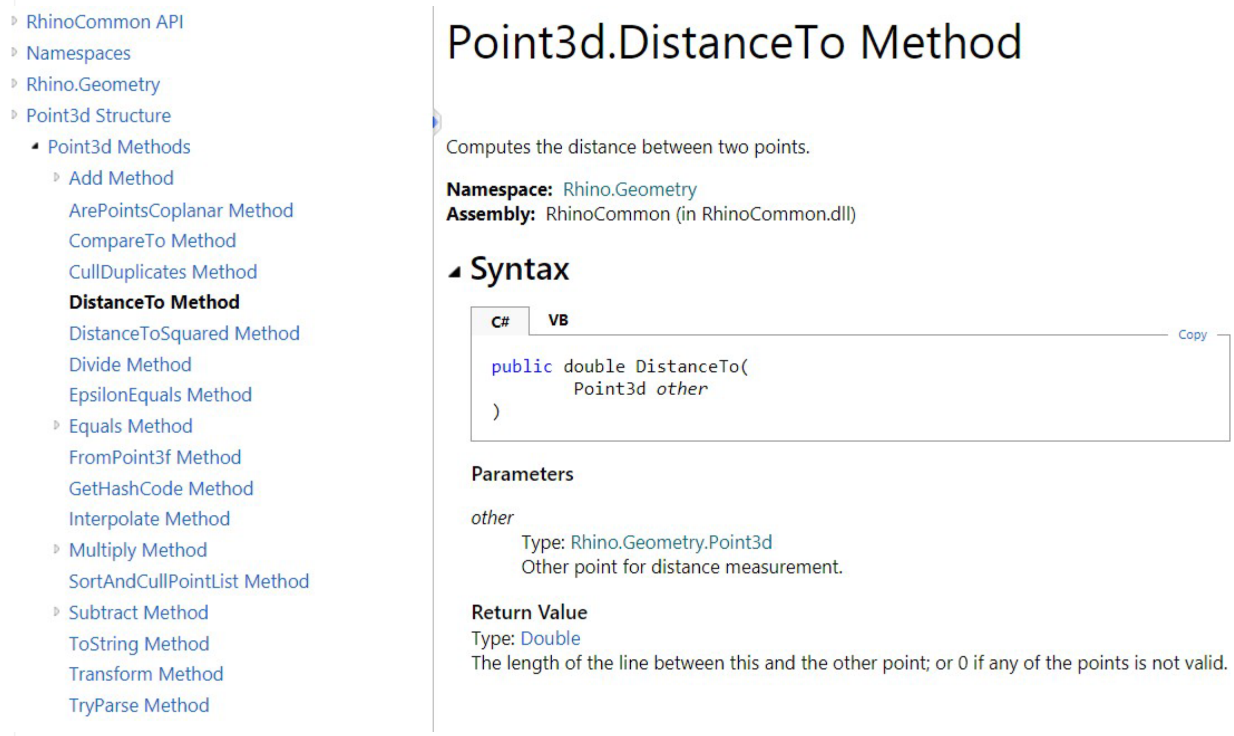

```
private void RunScript(double x, double y, double z, ref object Point)
{
    //Declare a new point
    Point3d newPoint = new Point3d(Point3d.Unset);

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

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

}
```

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.



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.DistanceTo Method

Computes the distance between two points.

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

Syntax

C# VB Copy

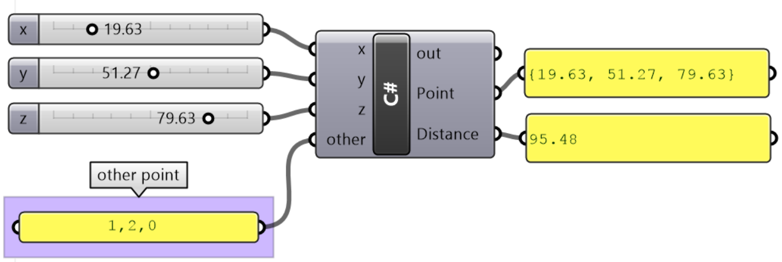
```
public double DistanceTo(
        Point3d other
)
```

Parameters

other
Type: Rhino.Geometry.Point3d
Other point for distance measurement.

Return Value
Type: Double
The length of the line between this and the other point; or 0 if any of the points is not valid.

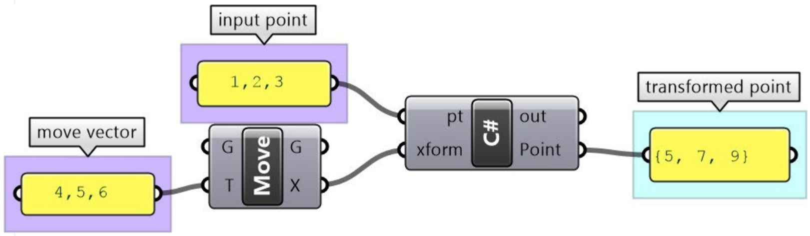
Here is how the ***DistanceTo*** method is used in an example.




```
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.

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;
```

```
}
```

- 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.

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.

- 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.ArePointsCoplanar Method

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

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

Syntax

C# VB Copy

```
public static bool ArePointsCoplanar(
	IEnumerable<Point3d> points,
	double tolerance
)
```

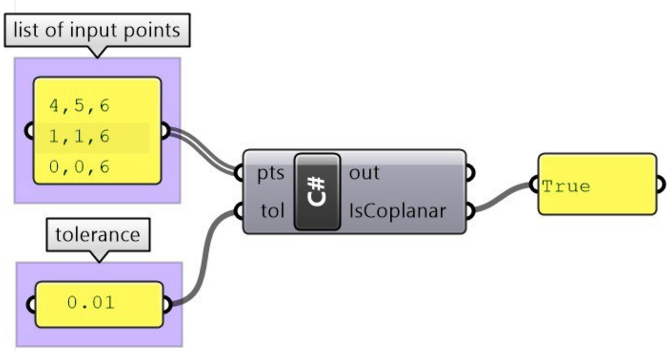
Parameters

points
Type: System.Collections.Generic.IEnumerable<Point3d>
A list, an array or any enumerable of Point3d.

tolerance
Type: System.Double
A tolerance value. A default might be RhinoMath.ZeroTolerance.

Return Value
Type: Boolean
true if points are on the same plane; false otherwise.

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***.

- RhinoCommon API
- Namespaces
- Rhino.Geometry
- Point3d Structure
- Point3d Operators and Type Conversions
  - Addition Operator
    - **Addition Operator (Point3d, Point3d)**
    - Addition Operator (Point3d, Vector3d)
    - Addition Operator (Point3d, Vector3f)
    - Addition Operator (Vector3d, Point3d)

Point3d.Addition Operator (Point3d, Point3d)

Sums two Point3d instances.

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

Syntax

C# VB Copy

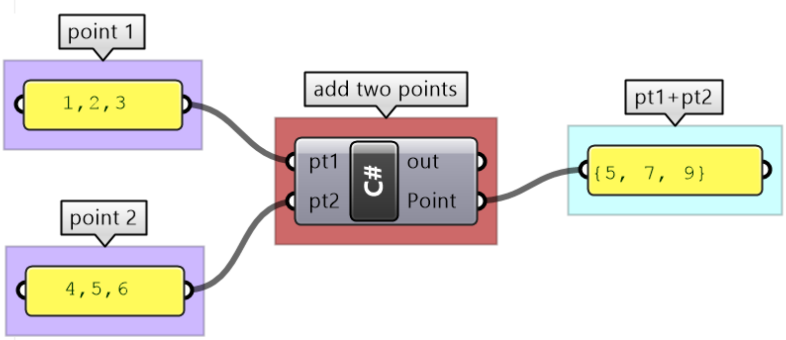
```
public static Point3d operator +(
        Point3d point1,
        Point3d point2
)
```

Parameters

point1
Type: Rhino.Geometry.Point3d
A point.
point2
Type: Rhino.Geometry.Point3d
A point.

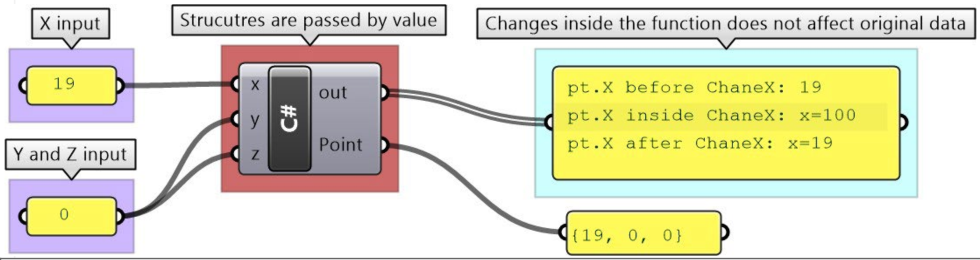
Return Value
Type: Point3d
A new point that results from the addition of point1 and point2.

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.

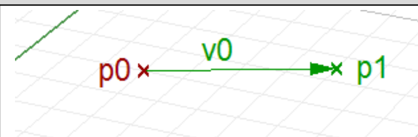
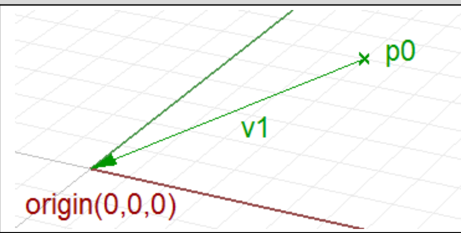
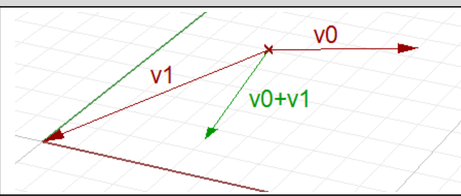
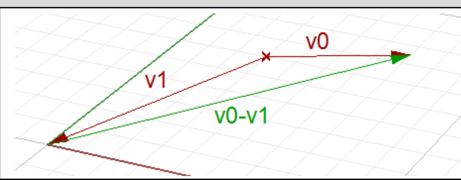
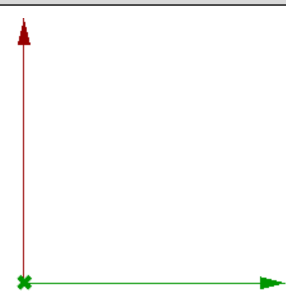
Class name	Member variables	Notes
Point2d	**X** as Double **Y** as Double	Used for parameter space points.
Point3d	**X** as Double **Y** as Double	Most commonly used to represent points in three dimensional coordinate space
Point4d	**X** as Double **Y** as Double **Z** as Double **W** as Double	Used 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
	Y as Double **Z** as Double	

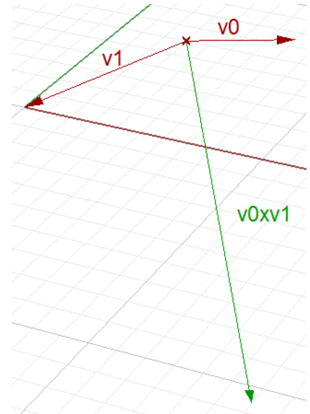
[12] For more detailed explanation of vectors and points, please refer to the "Essential Mathematics for Computational Design", a publication by McNeel.
https://www.rhino3d.com/download/rhino/6/essentialmathematics

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 factor = 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)	

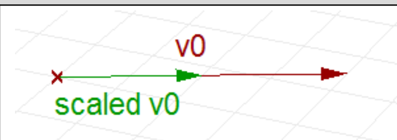
Vector cross product (result is a vector normal to the 2 input vectors)

```
Vector3d cross = new Vector3d(Vector3d.Unset);
cross = Vector3d.CrossProduct(v0, v1);
```



Scale a vector

```
Vector3d scaleV0 = new Vector3d(Vector3d.Unset);
scaleV0 = factor * v0;
```

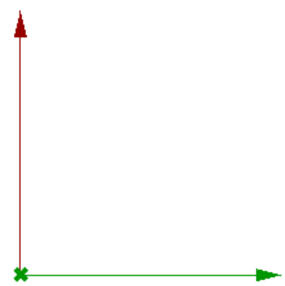


Get vector length

```
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

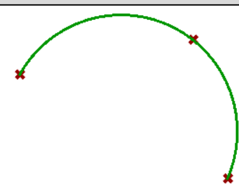
RhinoCommon defines basic types of curves such as lines and circles as structures and hence most of them are value types. The mathematical representation is easier to understand and is typically more light-weight. If needed, it is relatively easy to get the Nurbs approximation of these curves using the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

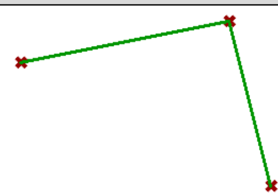
Lightweight Curves Types	
Line	Line between two points
Polyline	Polyline connecting a list of points (not value type)
Arc	Arc on a plane from center, radius, start and end angles
Circle	Circle on a plane from center point and radius
Ellipse	Defined by a plane and 2 radiuses

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

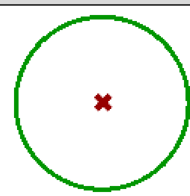
3_2_3: Lightweight curves

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

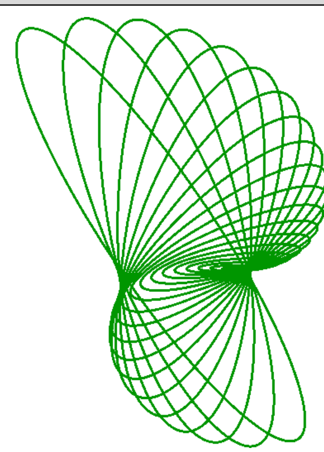
Declare and initialize 3 new points	
Point3d p0 = new Point3d(0, 0, 0); **Point3d** p1 = new Point3d(5, 1, 0); **Point3d** p2 = new Point3d(6, -3, 0);	
Create an instance of a ***Line***	
//Create an instance of a lightweight Line **Line** line = new **Line**(p0, p1);	
Create an instance of a ***Arc***	
//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);
}
	

3_2_4: Lightweight surfaces

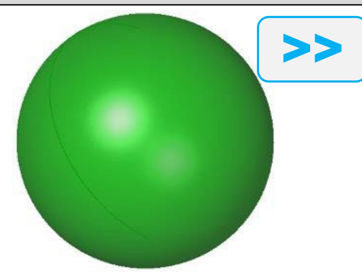
Just like with curves, ***RhinoCommon*** defines a number of lightweight surfaces that are defined as structures. They can be converted to Nurbs surfaces using the ***ToNurbsSurface()*** method. They include common surfaces such as cones and spheres. Here is a list of them:

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:

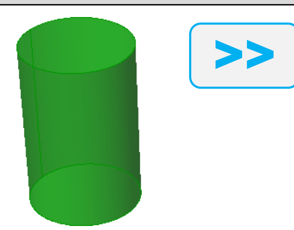
Create an instance of a ***Sphere***

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

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



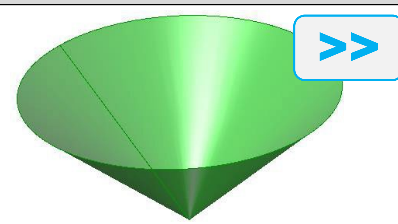
Create an instance of a ***Cylinder***

```
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***

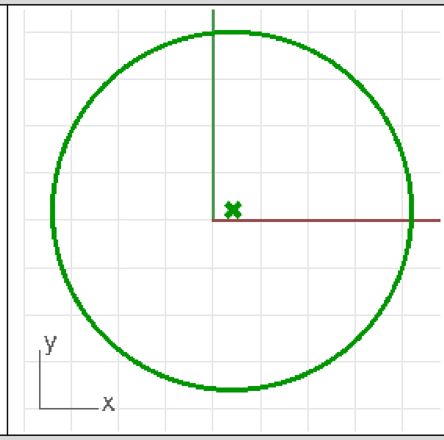
```
double minorRadius = 2.5;
double majorRadius = 7.5;

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.

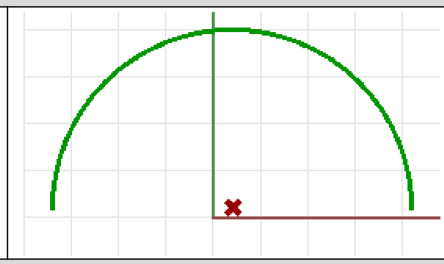
1- Create an instance of a circle on the xy-plane, center (2,1,0) and a random radius between 10 and 20

```
//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);
```



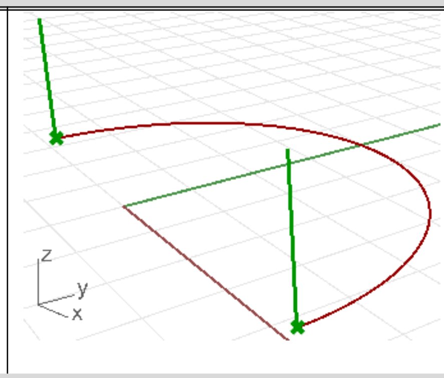
2- Create an instance of an arc from the circle and angle interval between 0 and Pi

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

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



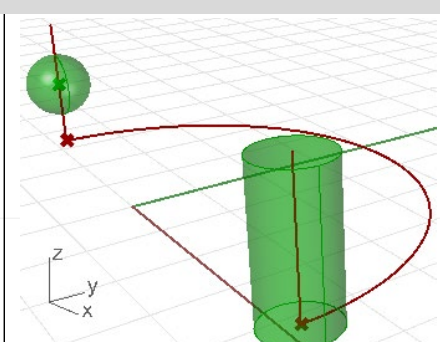
3- Extract the end points of the arc and create a vertical lines with length = 10 units

```
//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);
```



Assignment Week 03

Develop a composition of primitive and lightweight geometries that can be parametrically driven.

Minimum requirements:
At least 10 instances / elements
At least 3 different types / classes
At least 3 input parameters
Export geometry
Print (to out) some meaningful data along the way

Include screen captures of the resulting spatial configurations of the algorithm and the code used to generate your two versions. Paste these screen captures in this presentation, show two different states. Include your name on each slide.

Use this template to start the project.