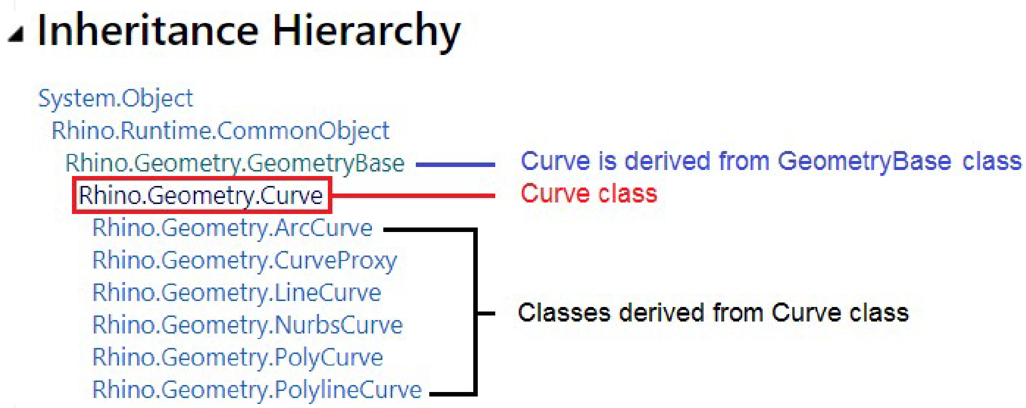
3_3: Geometry classes

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
      - OrdinateDimension
      - RadialDimension

      Classes derived from **Dimension** class
    - Leader
    - TextEntity

    Classes derived from **AnnotationBase** class
  - **Brep**
  - **BrepLoop**
  - **Curve** — Abstract Curve class
    - ArcCurve
    - CurveProxy
      - BrepEdge
      - BrepTrim
    - LineCurve
    - NurbsCurve
    - PolyCurve
    - PolylineCurve

    Classes derived from **Curve** base class
  - **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
    - RevSurface
    - SumSurface
    - SurfaceProxy
      - BrepFace

    Classes derived from **Surface** base class
  - **TextDot**

3_3_1: Curves

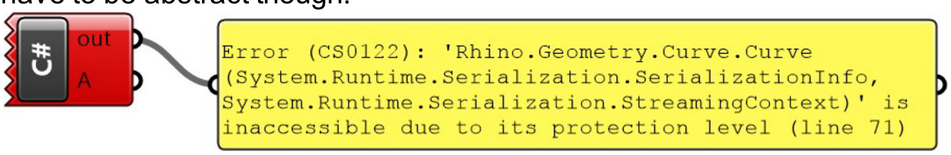
The ***RhinoCommon SDK*** has the ***abstract Rhino.Geometry.Curve*** class that provides a rich set of functionality across all curves. There are many classes derived from the parent ***Curve*** class and we will learn about how to create and manipulate them. The following is a list of the classes derived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

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.

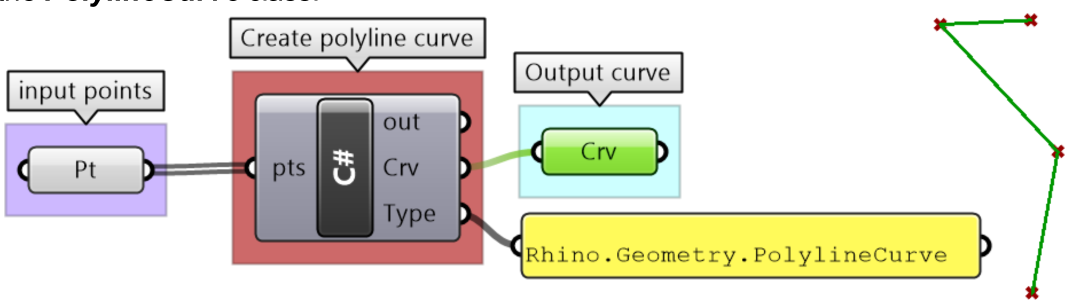
- **Abstract Classes**: The "***GeometryBase***" in ***RhinoCommon*** is one example of an abstract class. You cannot create an object or instantiate an instance of an abstract class. The purpose is to define common data and functionality that all derived classes can share.

- **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.



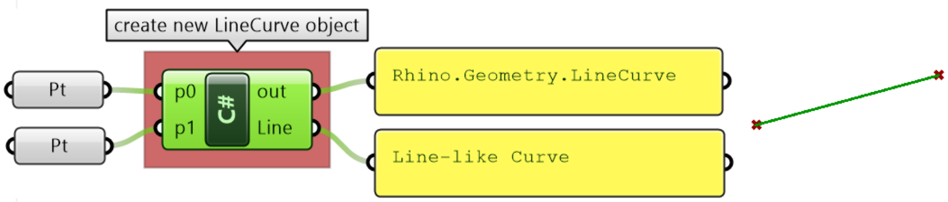

```
private void RunScript(ref object A)
{
    //ERROR: attempt to create an instance of the abstract "Curve" class
    Rhino.Geometry.Curve crv = new Rhino.Geometry.Curve();
}
```

- **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.



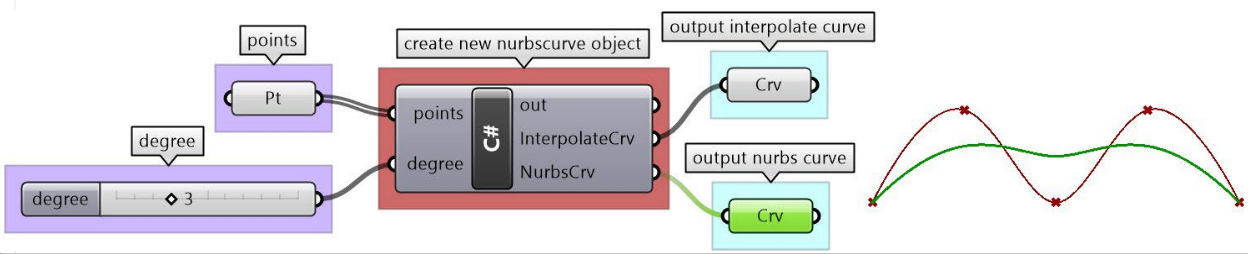

```
private void RunScript(List<Point3d> pts, int degree, ref object Crv, ref object Type)
{
    //Declare and create a new instance of a polyline curve from points
    var crv = new Rhino.Geometry.PolylineCurve(pts);
    //Assign curve to A output
    Crv = crv;
    //Assign curve type to B output
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

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

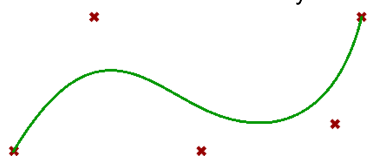
Class methods return values and sometimes those are new instances of objects. For example the ***Offset*** method in the ***Curve*** class returns a new array of curves that is the result of the offset.

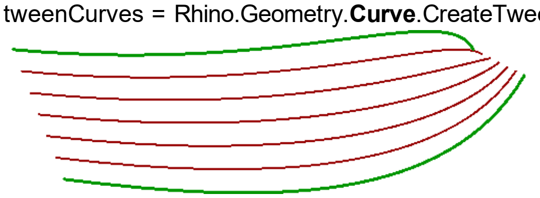
```
Curve[ ] offsetCurves = x.Offset( Plane.WorldXY, 1.4, 0.01, CurveOffsetCornerStyle.None );
```

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.

3_3_1: Curves

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

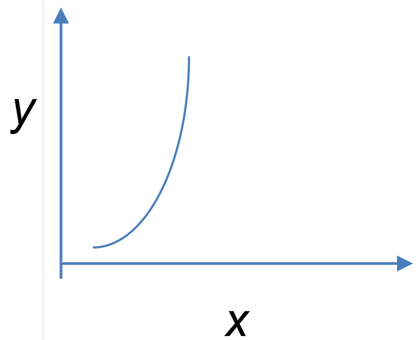
//Declare a variable of type Curve
Curve[ ] tweenCurves = null;
 	
//Create an array of tween curves
tweenCurves = Rhino.Geometry.**Curve**.CreateTweenCurves(curve0, curve1, count, 0.01); 

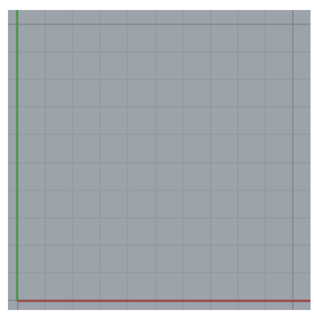
Curves

Explicit

y = f(x)   // y is a function of x

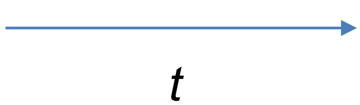
$y = x^2$



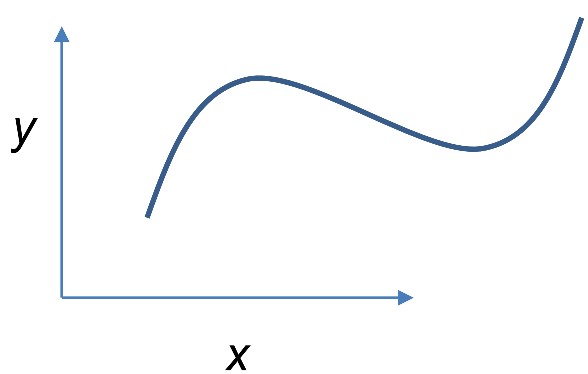


Implicit

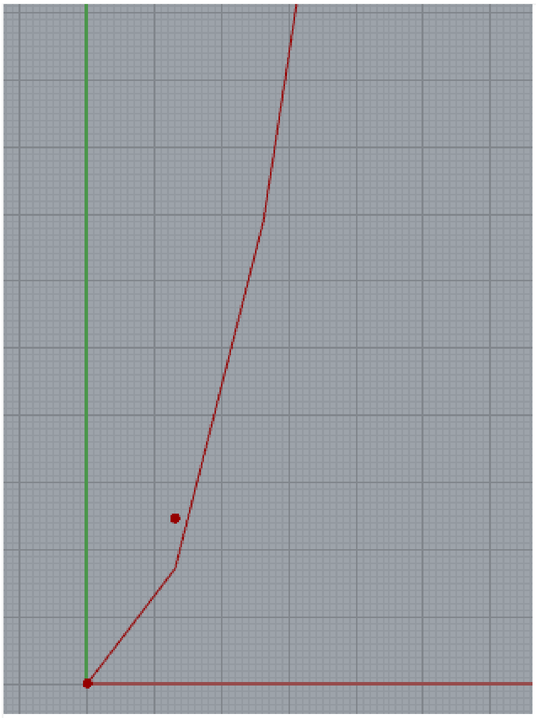
x = f(t)
y = f(t)

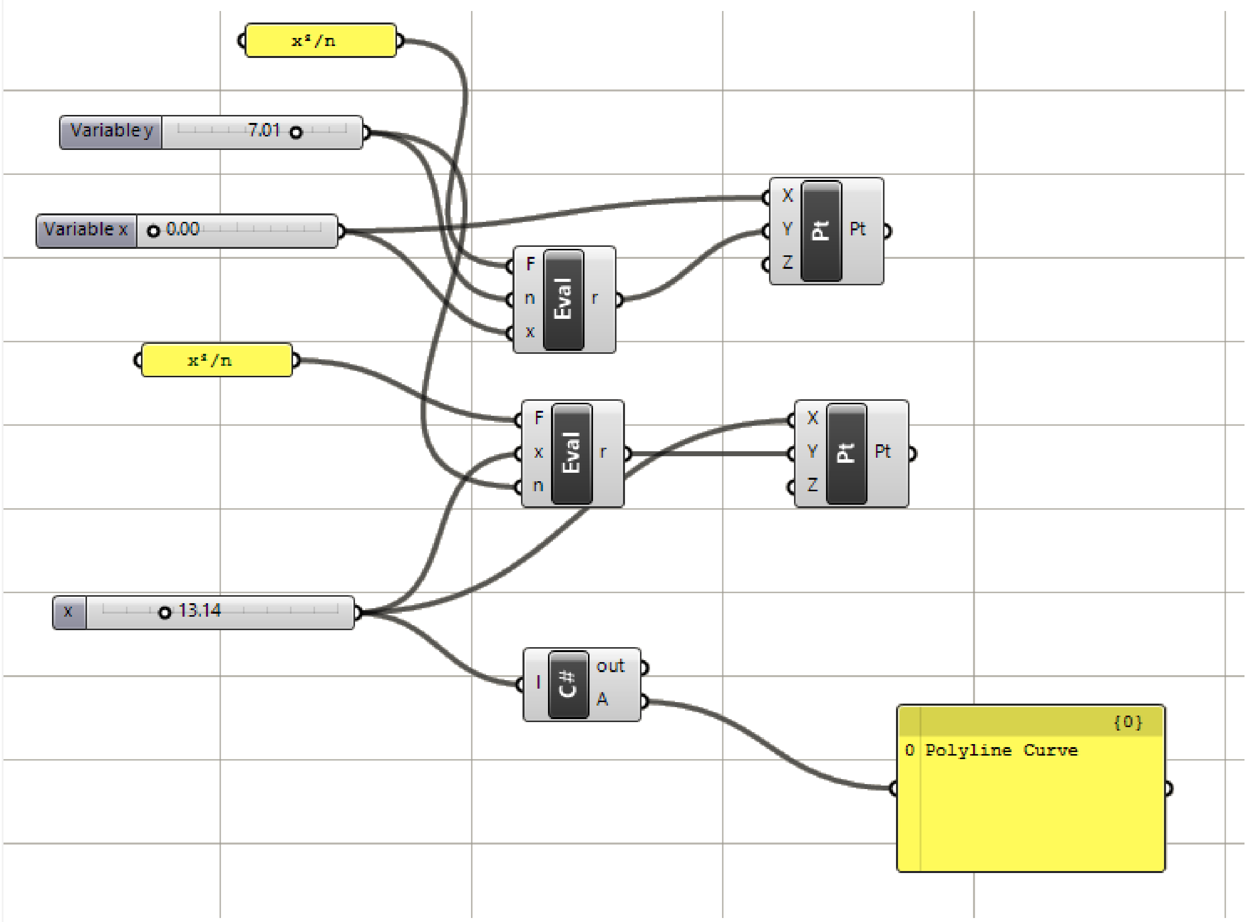


$p(t) = (x_t, y_t)$



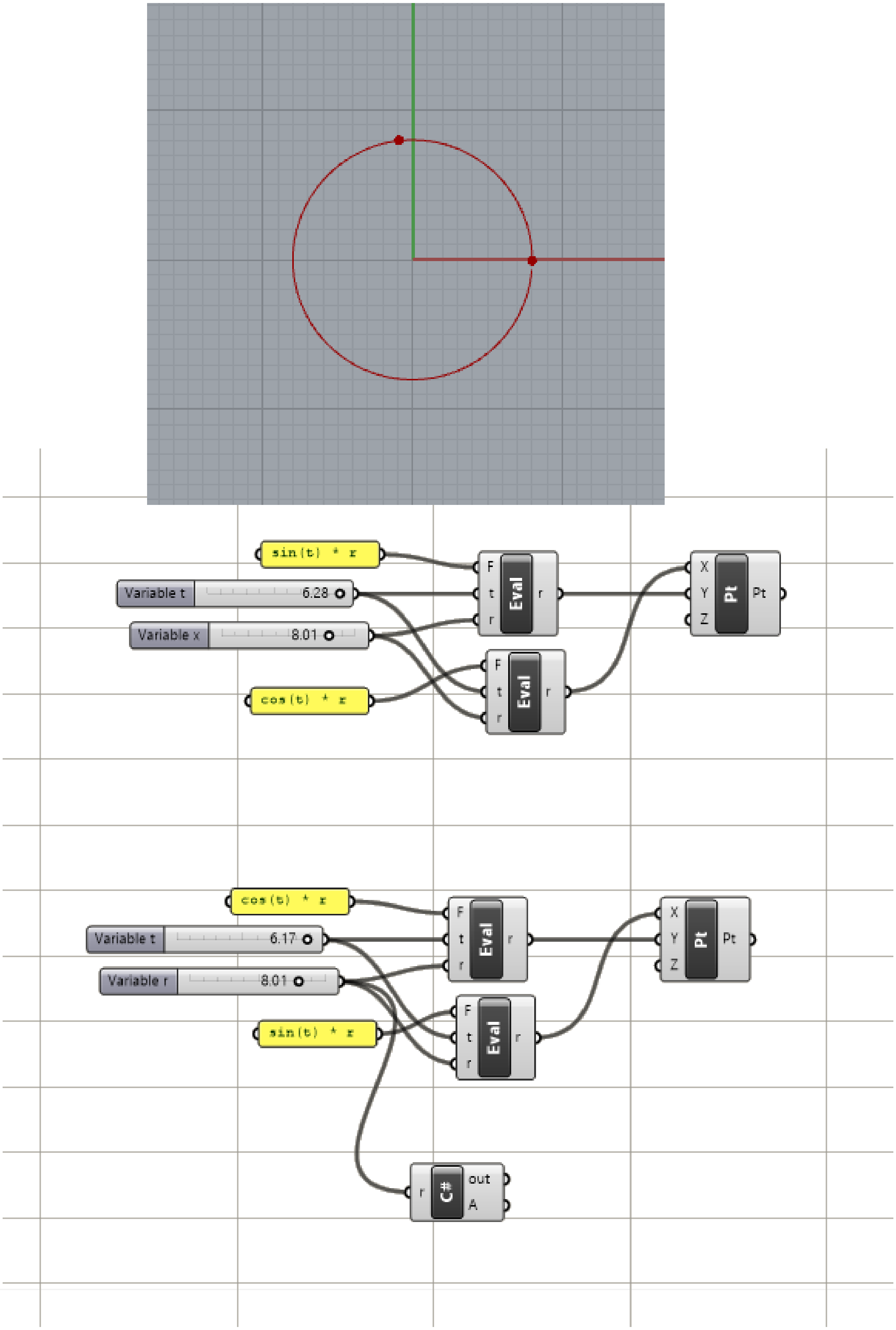
Explicit vs implicit curve functions



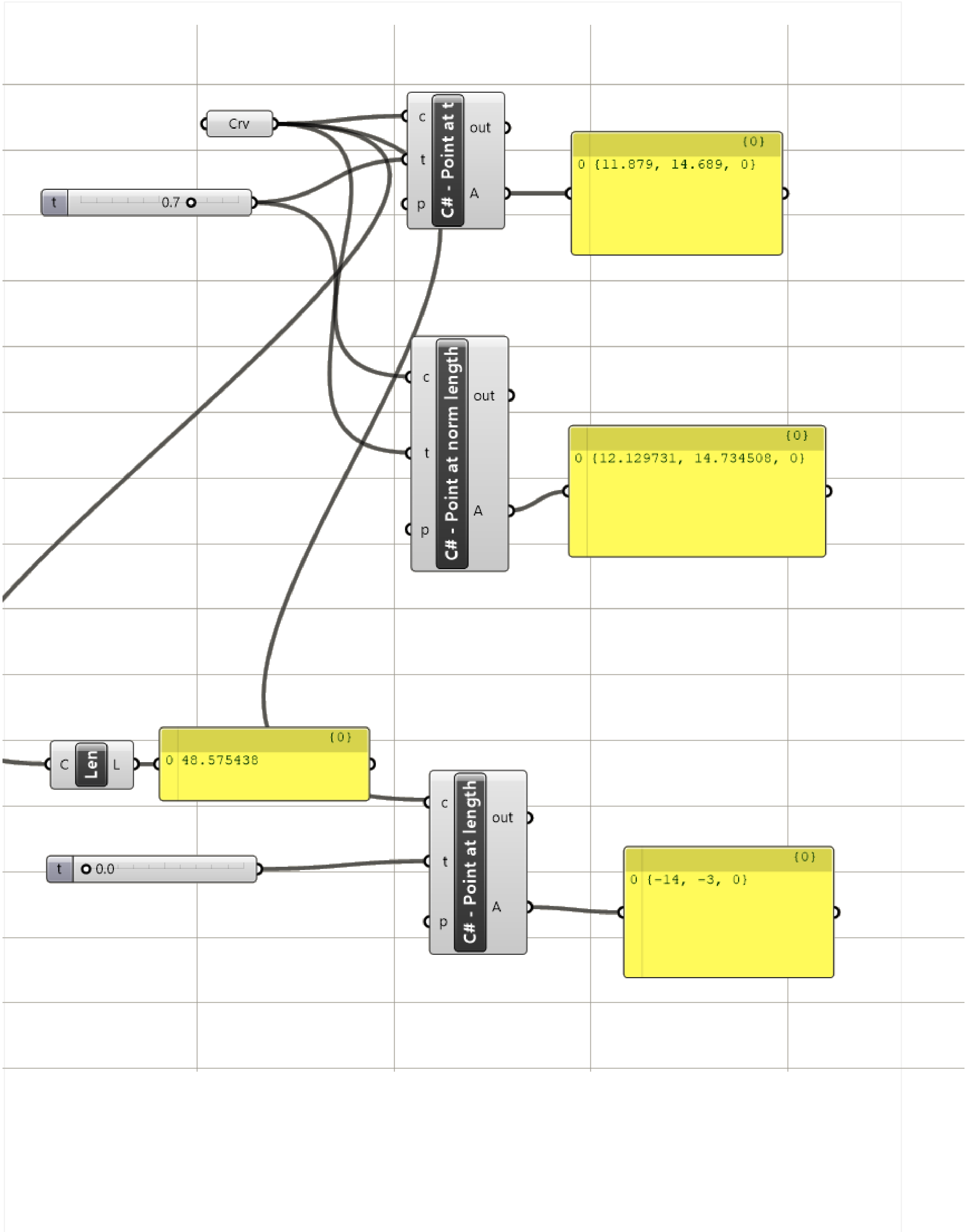


Explicit vs implicit curve functions - circle

>>

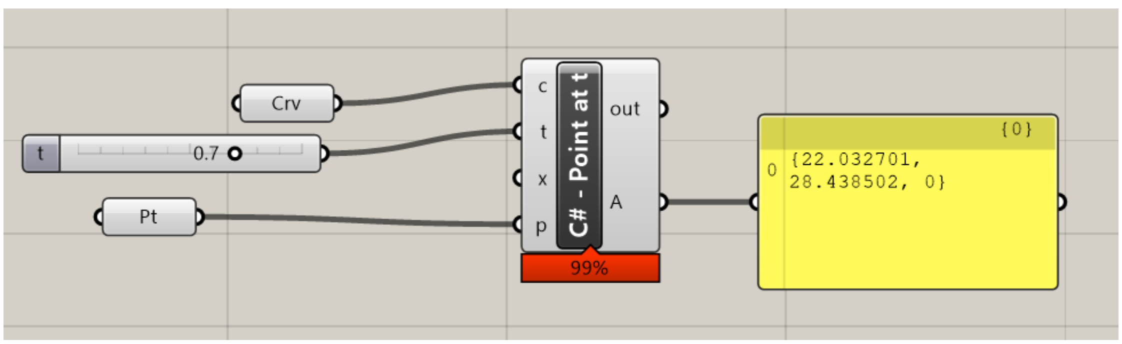


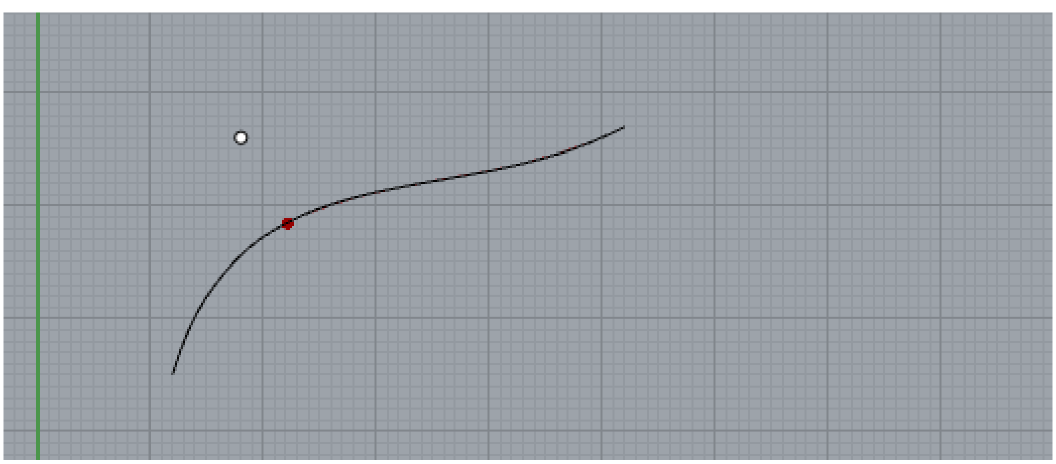
Point at parameter



Curve functions – closest point

```
private void RunScript(Curve c, double t, object x, Point3d p, ref object A)
{

  double tOut;
  c.ClosestPoint(p, out tOut);
  Point3d closeP = c.PointAt(tOut);
  A = closeP;
  Print(tOut.ToString());

}
```

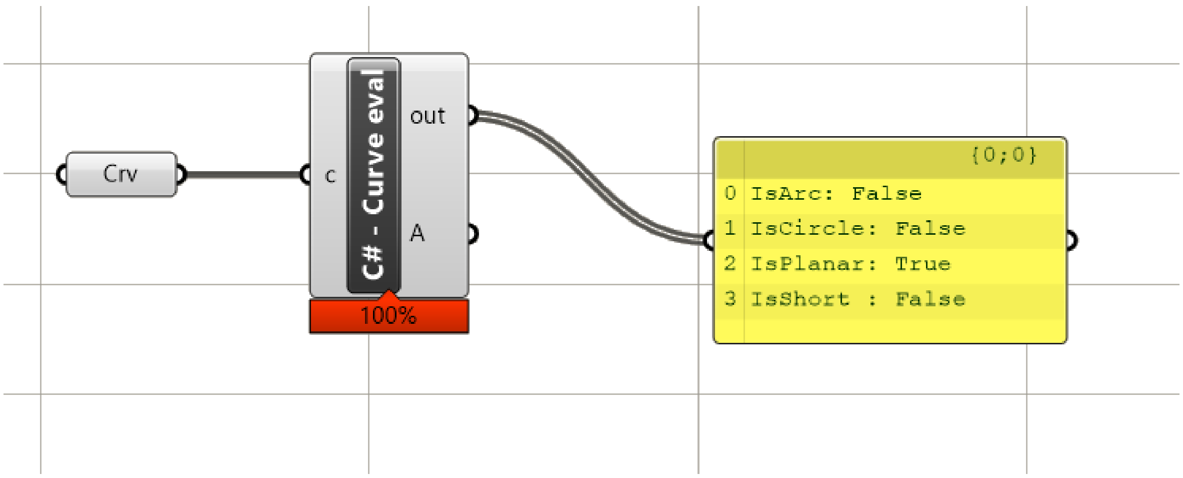


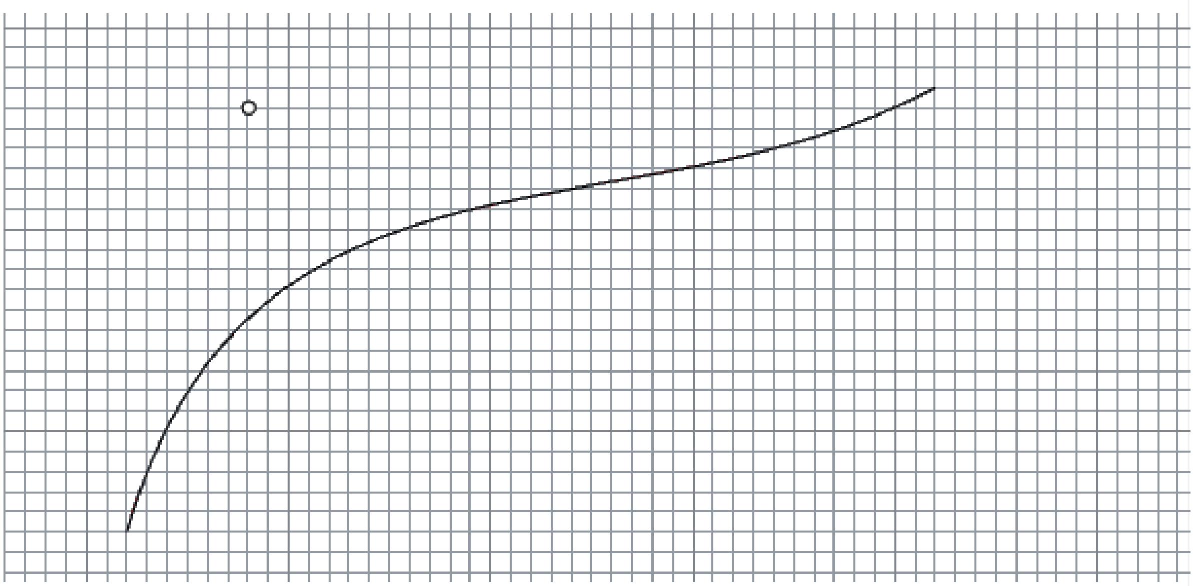


Curve functions – evaluate

>>

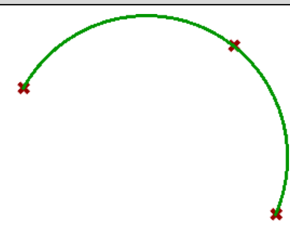
```
private void RunScript(Curve c, ref object A)
{

  Print("IsArc: " + c.IsArc());
  Print("IsCircle: " + c.IsCircle());
  Print("IsPlanar: " + c.IsPlanar());
  Print("IsShort : " + c.IsShort(1.0));

}
```



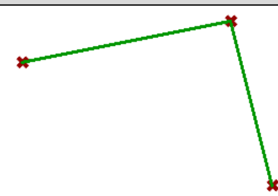


Another way to create new curves is to use the constructor of the curve with the ***new*** keyword. The following are examples to create different types of curves using the constructor or the ***Create*** method in the class. You can reference the ***RhinoCommon*** help for more details about the constructors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

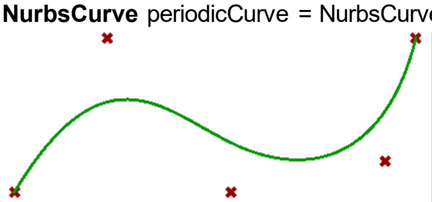
Create an instance of a ***PolylineCurve*** using the class constructor and ***new*** keyword	

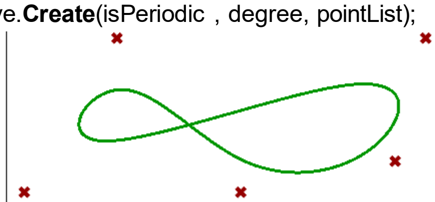
//Put the 3 points in a list
Point3d[ ] pointList = {p0, p1, p2};
 	
//Create an instance of an PolylineCurve
PolylineCurve polyline = new **PolylineCurve**(pointList);
	

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);

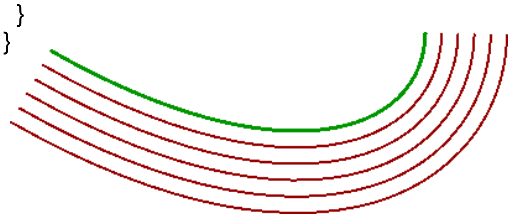




Curves can also be the return value of a method. For example offsetting a given curve creates one or more new curves. Also the surface ***IsoCurve*** method returns an instance of a curve.

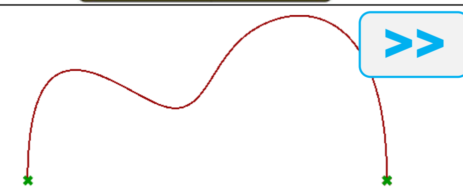
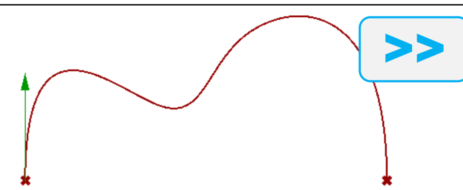
Multiple offset of curve

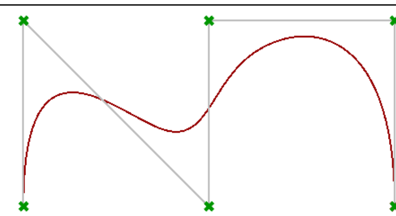
```
private void RunScript(Curve crv, int num, double dis, double tol, Plane plane)
{
  //Declare the list of curve
  List<Curve> crvs = new List<Curve>();
  Curve lastCurve = crv;
  for (int i = 1; i <= num; i++)
  {
    Curve[ ] curveArray = last_crv.Offset(plane, dis, tol, CurveOffsetCornerStyle.None);

    //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;
  }
}
```

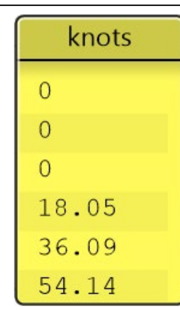


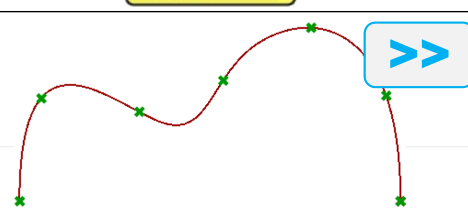
Curve methods:
Each class can define methods that help navigate the data of the class and extract some relevant information. For example, you might want to find the endpoints of a curve, find the tangent at some point, get a list of control points or divide the curve. The ***AutoComplete*** helps to 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.

Some of the ***Curve*** and ***NurbsCurve*** methods	
//Get the domain or interval of the curve **Interval** domain = crv.Domain;	
//Get the start and end points of a curve **Point3d** startPoint = crv.PointAtStart; **Point3d** endPoint = crv.PointAtEnd;	
//Get the tangent at start of a curve **Vector3d** startTangent = crv.TangentAtStart;	

//Get the control points of a NurbsCurve (nc)
List<**Point3d**> cpList = new List<Point3d>();
int count = nc.Points.Count;
 	
//Loop to get all cv points
for (int i = 0; i <= count - 1; i++) {
    ControlPoint cp = nc.Points[i];
    cpList.Add(cp.Location);
}
	

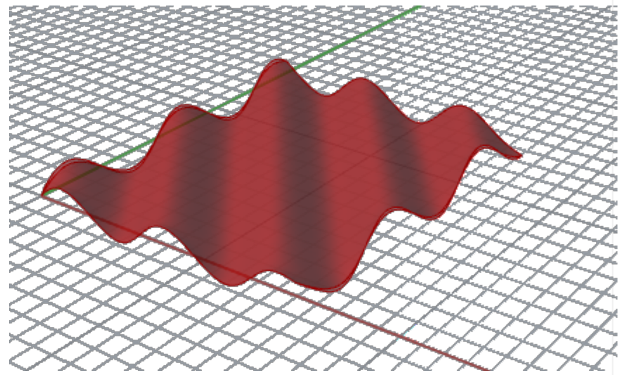
//Get the knot list of a NurbsCurve (nc)
List<double> knotList = new List<double>();
int count = nc.Points.Count;
 	
//Loop to get all knots values
for (int i = 0; i <= count - 1; i++) {
    double knot = nc.Knots[i];
    knotList.Add(knot);
}
	

//Divide curve (crv) by number (num)
//Declare an array of points
Point3d[ ] points = { };
 	
//Divide the curve by number
crv.DivideByCount(num, true, out points);
	

Surfaces

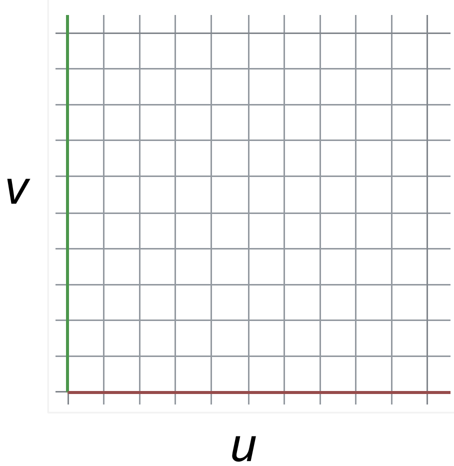
Explicit

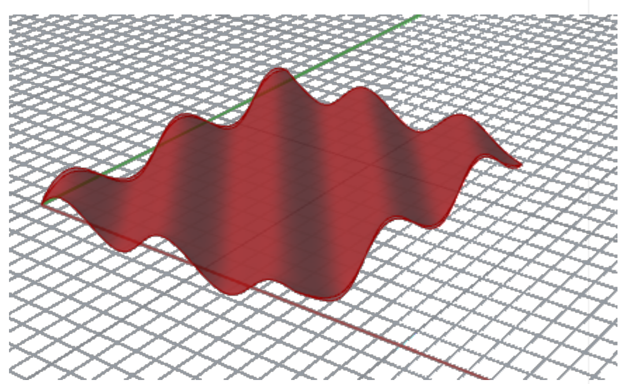
$z = f_s(x, y)$



Implicit

$x = f_x(u, v)$
$y = f_y(u, v)$
$z = f_z(u, v)$

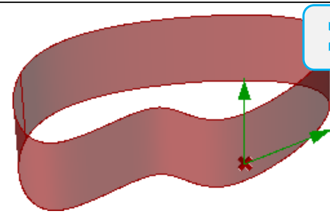




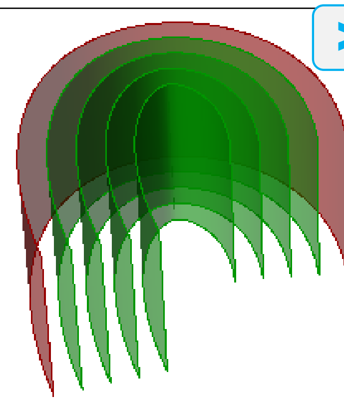
Examples of the ***Surface*** and ***NurbsSurface*** methods

```
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();
```

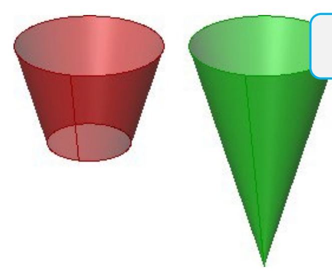
```
Surface srf = … //from input
double u = 0.5;
double v = 0.5;
//Declare and instantiate the evaluation point and derivative vectors
Point3d evalPt = new Point3d(Point3d.Unset);
Vector3d[ ] derivatives = {};

//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++) {t
    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;
}
```



```
//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.

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);`


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.

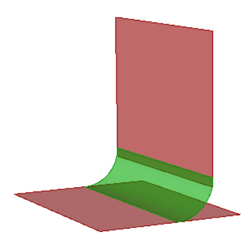
Create surface objects:

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.

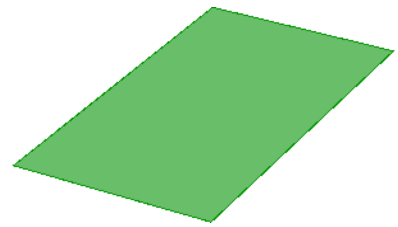
```
private void RunScript(Surface srfA, Surface srfB, double radius, double tol, ref object A)
{
   //Declare an array of surfaces
   Surface[ ] surfaces = {};

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



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.

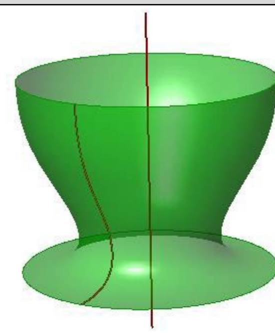
Create an instance of a ***PlaneSurface*** using the constructor and ***new*** keyword

```
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);
```



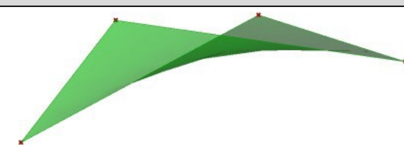
Create an instance of a ***RevSurface*** from a line and a profile curve

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

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



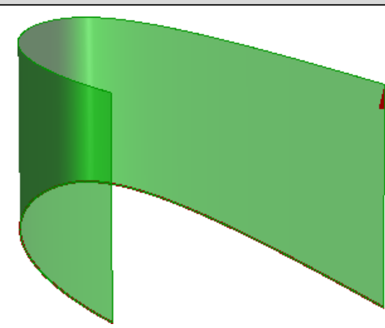
Create an instance of a ***NurbsSurface*** from a list of control points

```
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.