Revit Family API Labs

Lab1 – Create Rectangular Column

March 2010 by M. Harada

Last updated, Date : April 04, 2021

C# version

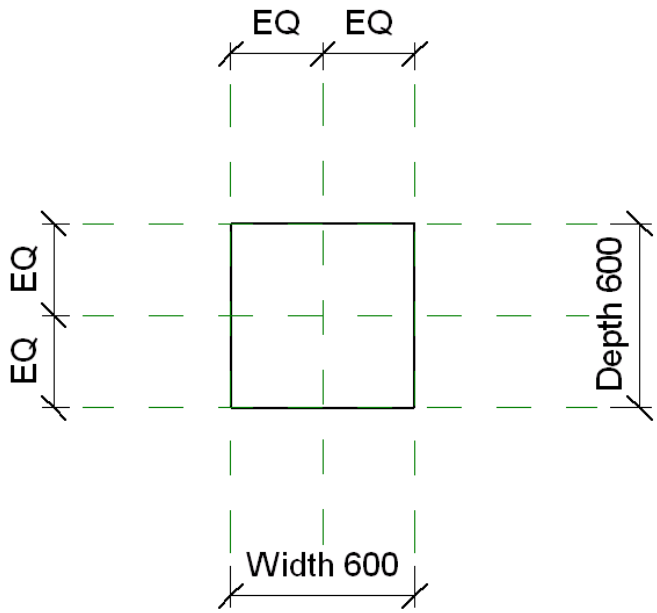
Objective: In this lab, we will learn the basics of family API. We'll learn how to:

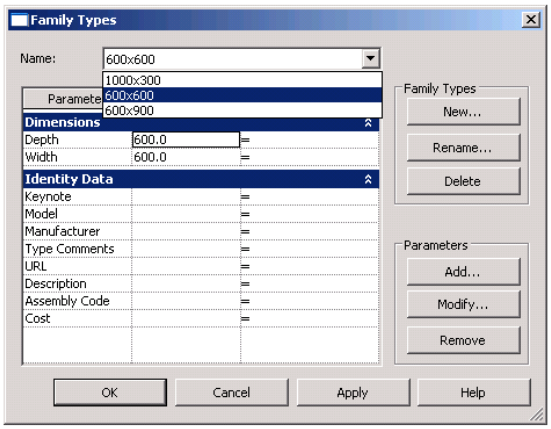
- check the family environment
- create a simple solid using extrusion
- set alignments
- add types

Tasks: We'll define a command that creates a column family which has a rectangular profile, and add three types with dimensional variations:

(0) Take "Metric Column.rft" as a template. We can assume the user has opened the correct template. But within the command, we check if the user has chosen a right template.
(1) Define a rectangular profile and create a simple box solid using extrusion.
(2) Add alignments between each face and corresponding reference plane.
(3) Define types with dimensional variations.

Figure 1 shows the image of rectangular columns that we are going to define in this lab.





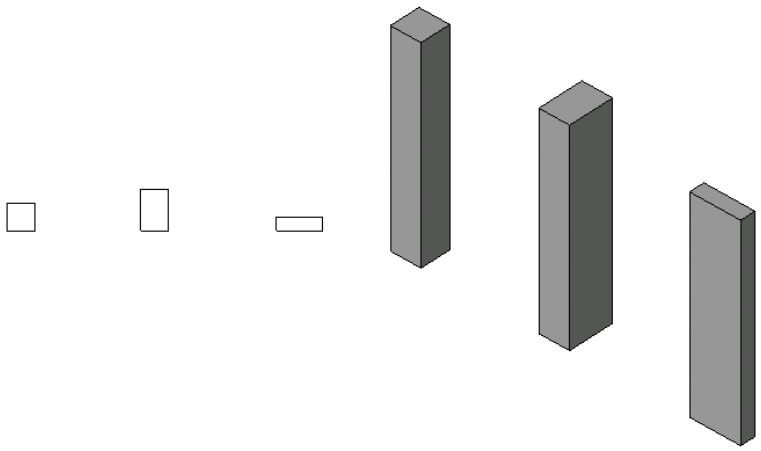


Figure 1. A column family with a rectangular profile we will be creating in Lab1.

The following is the breakdown of step by step instructions in this lab:

Note: the C# version uses an Util class to simplify the display of a MessageBox. The code is attached at the end of this document.

1. Define an External Command

1.1 Create a new Visual Studio project of your choice (C# or VB.NET). Add references. Define a new external command. Let's name them as follows:

- Solution name: **FamilyLabs**
- Project name: **FamilyLabsCS**
- File name: **1 FamilyCreateColumnRectangle.cs (or .vb)**
- Command class name: **RvtCmd_FamilyCreateColumnRectangle**

(You may choose to use any names you want here. When you do so, just remember what you are calling your own project, and substitute these names as needed while reading the instruction in this document.)

We will need the following reference at least:

- System
- System.Core (this is for LINQ query)
- System.Windows.Forms
- Revit API
- RevitAPIUI

The following is the list of namespaces you will need for this lab. Add them to the top of your .cs files.

```
using System;
using System.Collections.Generic;
using System.Linq; // in System.Core
using Autodesk.Revit;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.ApplicationServices;
```

1.2 Keep the top level object access for Revit application and document as member variables, e.g., `_rvtApp, _rvtDoc` respectively. The code should look like below:

```
[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Manual)]
[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
  class RvtCmd_FamilyCreateColumnRectangle : IExternalCommand
  {
    // member variables for top level access to the Revit database
    //
    Application _rvtApp;
    Document _rvtDoc;

    // command main
    //
    public Result Execute(
      ExternalCommandData commandData,
      ref string message,
      ElementSet elements )
    {
      // objects for the top level access
      //
      _rvtApp = commandData.Application.Application;
      _rvtDoc = commandData.Application.ActiveUIDocument.Document;


      return Result.Succeeded;
    }
  }
```

2. Check the Validity of Document Context

Our command with family API works only in the context of Family editor. We will first check the validity of the document context that we are current in. Let's define a function `isRightTemplate()` to check this. The function `isRightTemplate()` takes BuiltInCategory as an argument.

1.3 Add the follow function to the class:

```
    // ===========================================
    //   (0) check if we have a correct template
    // ===========================================
    bool isRightTemplate( BuiltInCategory targetCategory )
    {
      // This command works in the context of family editor only.
      //
      if( !_rvtDoc.IsFamilyDocument )
      {
        Util.ErrorMsg( "This command works only in the family editor." );
        return false;
      }

      // Check the template for an appropriate category here if needed.
      //
      Category cat = _rvtDoc.Settings.Categories.get_Item( targetCategory );
      if( _rvtDoc.OwnerFamily == null )
      {
        Util.ErrorMsg( "This command only works in the family context." );
        return false;
      }
      if( !cat.Id.Equals( _rvtDoc.OwnerFamily.FamilyCategory.Id ) )
      {
        Util.ErrorMsg( "Category of this family document does not match the
context required by this command." );
        return false;
      }

      // if we come here, we should have a right one.
      return true;
    }
```

This function checks:

- If we are in a family document. You can check `_rvtDoc.IsFamilyDocument` for this.
- If the template is for defining a column. The category of current document can be checked by looking at `_rvtDoc.OwnerFamily.FamilyCategory.Id`.

If the above are satisfied, the function returns true, otherwise false.

2.2 Call `isRightTemplate`() function from your main command function `Execute()`. If it is not a right template, the command will halt:

```
    public Result Execute(
      ExternalCommandData commandData,
      ref string message,
```

```
      ElementSet elements )
    {
      // objects for the top level access
      //
      _rvtApp = commandData.Application.Application;
      _rvtDoc = commandData.Application.ActiveUIDocument.Document;

      // (0) This command works in the context of family editor only.
      //     We also check if the template is for an appropriate category if
needed.
      //     Here we use a Column(i.e., Metric Column.rft) template.
      //     Although there is no specific checking about metric or imperial,
our lab only works in metric for now.
      //
      if( !isRightTemplate( BuiltInCategory.OST_Columns ) )
      {
        Util.ErrorMsg( "Please open Metric Column.rft" );
        return Result.Failed;
      }

      ...
```

Note: we are using a helper class Util to simplify the display of a message box. Please take a look at Appendix B for the actual code sample.

3. Create a Simple Solid with Extrusion

Now that we have a valid template, let's start creating a simple solid. Here we define a rectangular profile, extrude with a given height.

3.1 Add the follow function to the class. This code defines a profile with a simple rectangular shape:

```
    // ===========================================
    //   (1.1) create a simple rectangular profile
    // ===========================================
    CurveArrArray createProfileRectangle()
    {
      //
      // define a simple rectangular profile
      //
      //  3     2
      //   +---+
      //   |   | d    h = height
      //   +---+
      //  0     1
      //  4  w
      //

      // sizes (hard coded for simplicity)
      // note: these need to match reference plane. otherwise, alignment
won't work.
```

```
      // as an exercise, try changing those values and see how it behaves.
      //
      double w = mmToFeet(600.0);
      double d = mmToFeet(600.0);

      // define vertices
      //
      const int nVerts = 4; // the number of vertices

      XYZ[] pts = new XYZ[] {
        new XYZ(-w / 2.0, -d / 2.0, 0.0),
        new XYZ(w / 2.0, -d / 2.0, 0.0),
        new XYZ(w / 2.0, d / 2.0, 0.0),
        new XYZ(-w / 2.0, d / 2.0, 0.0),
        new XYZ(-w / 2.0, -d / 2.0, 0.0) };

      // define a loop. define individual edges and put them in a curveArray
      //
      CurveArray pLoop = _rvtApp.Create.NewCurveArray();
      for( int i = 0; i < nVerts; ++i )
      {
        Line line = Line.CreateBound( pts[i], pts[i + 1] );
        pLoop.Append( line );
      }

      // then, put the loop in the curveArrArray as a profile
      //
      CurveArrArray pProfile = _rvtApp.Create.NewCurveArrArray();
      pProfile.Append( pLoop );
      // if we come here, we have a profile now.

      return pProfile;
    }
```

There is one helper function we use in this function, i.e.,

```
mmToFeet()
```

Revit uses feet. You will need to convert unit whenever you define dimensions through API. The code for `mmToFeet()` is attached toward the end of this doc, the section, Appendix A. Please copy it and paste to the end of this class.

We are hard-coding the actual size of rectangle as well as vertices of rectangular shape for simplicity and for the readability of the code for our learning purpose. The size actually comes from the distance between the reference planes that are predefined in the column family template. If you are using a different template, you will need to adjust those values. A profile is defined as a CurveArrArray (or a collection of curve arrays).

This may be a good time to take closer look at your template. Open the template and observe the following:

- Dimensions used in the template.

- The names of the reference planes that are predefine for the template.
- Looks for these in Front view as well as Plan view.

What do you see there? You may want to note those as we'll reference them throughout the labs.

3.2 Using the profile we have just defined, we then create a solid from extrusion. Add the following function to your class code:

```
    // ========================================
    //   (1) create a simple solid by extrusion
    // ========================================
    Extrusion createSolid()
    {
      //
      // (1) define a simple rectangular profile
      //
      //  3     2
      //   +---+
      //   |   | d    h = height
      //   +---+
      //  0     1
      //  4  w
      //
      CurveArrArray pProfile = createProfileRectangle();
      //
      // (2) create a sketch plane
      //
      // we need to know the template. If you look at the template (Metric
Column.rft) and "Front" view,
      // you will see "Reference Plane" at "Lower Ref. Level". We are going
to create an extrusion there.
      // findElement() is a helper function that find an element of the given
type and name.  see below.
      //
      ReferencePlane pRefPlane = findElement( typeof( ReferencePlane ),
"Reference Plane" ) as ReferencePlane;
      SketchPlane pSketchPlane = SketchPlane.Create(_rvtDoc,
pRefPlane.Plane);

      // (3) height of the extrusion
      //
      // once again, you will need to know your template. unlike UI, the
alightment will not adjust the geometry.
      // You will need to have the exact location in order to set alignment.
      // Here we hard code for simplicity. 4000 is the distance between Lower
and Upper Ref. Level.
      // as an exercise, try changing those values and see how it behaves.
      //
      double dHeight = mmToFeet( 4000.0 );

      // (4) create an extrusion here. at this point. just an box, nothing
else.
      //
      bool bIsSolid = true;
```

```
      return _rvtDoc.FamilyCreate.NewExtrusion( bIsSolid, pProfile,
pSketchPlane, dHeight );
    }
```

There is the second helper function we use in this function, i.e.,

```
findElement()
```

This helper function finds an element of the given type and the name. You can use this, for example, to find a ReferencePlane, Level or View. The full code is attached at the end of this doc, the section, Appendix A. Please copy it and paste to the end of this class.

At the bottom of the above code, you see a call to a method,

```
_rvtDoc.FamilyCreate.NewExtrusion
```
().

This is the main method we use to define an extrusion. It takes Solid/Void flag, a profile, a sketch plane and a height as argument. We use one of the predefined reference planes to define a sketch plane (i.e., "Reference Plane" in the above code or in our template).

Once again, we have hard-coded the height information here. This comes from the template and is the distance between the lower and upper reference level. (Check this in the Front view of the template, for example.)

3.3 Call the function above from your main command function. `createSolid()` returns an object of type Extrusion:

```
    public IExternalCommand.Result Execute(
      ExternalCommandData commandData,
      ref string message,
      ElementSet elements )
    {
      ...

      // (0) This command works in the context of family editor only.
      ...

      if ( !isRightTemplate( BuiltInCategory.OST_Columns ) )
      {
        Util.ErrorMsg( "Please open Metric Column.rft" );
        return IExternalCommand.Result.Failed;
      }

      // (1) create a simple extrusion. just a simple box for now.
      Extrusion pSolid = createSolid();

      // We need to regenerate so that we can build on this new geometry
      _rvtDoc.Regenerate();


      ...
```

3.4. Your code is ready to build and run at this point to test if your solid is created correctly. If you would like, go ahead and see how your solid look like at this point.

You can create an .addin manifest file with the information like the following, and add it to the location that Revit would recognize. (I'm assuming that you are familiar with this by now.) Make necessary adjustment to match with your environment, of course. One thing you may notice is that we have set the visibility mode as "NotVisibleInProject". This is because our command is specifically designed to work in Family Editor mode and not in a Revit project.

```
<?xml version="1.0" encoding="utf-16" standalone="no"?>
<RevitAddIns>
 <AddIn Type="Command">
  <Assembly>C:\Revit SDK 202x\Family Labs\FamilyLabsCS\bin\Debug\FamilyLabsCS.dll</Assembly>
  <AddInId>98F162FC-90E3-411e-BEC8-75D403BBF11A</AddInId>
  <FullClassName>FamilyLabsCS.RvtCmd_FamilyCreateColumnRectangle</FullClassName>
  <Text>Family API 1 CS - Create Rectangular Column</Text>
  <Description>Family API lab 1 to create rectangular column</Description>
  <VisibilityMode>NotVisibleInProject</VisibilityMode>
  <AccessibilityClassName>Revit.Samples.SampleAccessibilityCheck </AccessibilityClassName>
  <VendorId>ADNP</VendorId>
  <VendorDescription>Autodesk, Inc. www.autodesk.com</VendorDescription>
 </AddIn>
</RevitAddIns>
```

Remember to start with Family Editor and use "Metric Column.rft" template.

After running a command, you may find the column you just create does not quite behave as you intended. This is as expected at this stage. A family is a parametric object. What we have just defined is an initial state of the model. We'll be adding more to it to make our column behave in parametric manner.

4. Add Alignments

The next step is to add alignment constraint between each face of the solid and corresponding reference planes. This is needed to make our column behaves in parametric manner; when we want our column to adjust its sizes when the user changes its dimensions.

4.1 Add the follow function to the class. This function adds six alignments: one for each of six faces of box-shape:

```
    // ====================================
    //   (2) add alignments
    // ====================================
    void addAlignments( Extrusion pBox )
    {
      //
      // (1) we want to constrain the upper face of the column to the "Upper
Ref Level"
```

```
      //

      // which direction are we looking at?
      //
      View pView = findElement( typeof( View ), "Front" ) as View;

      // find the upper ref level
      // findElement() is a helper function. see below.
      //
      Level upperLevel = findElement( typeof( Level ), "Upper Ref Level" ) as
Level;
      Reference ref1 = upperLevel.PlaneReference;

      // find the face of the box
      // findFace() is a helper function. see below.
      //
      PlanarFace upperFace = findFace( pBox, new XYZ( 0.0, 0.0, 1.0 ) ); //
find a face whose normal is z-up.
      Reference ref2 = upperFace.Reference;

      // create alignments
      //
      _rvtDoc.FamilyCreate.NewAlignment( pView, ref1, ref2 );

      //
      // (2) do the same for the lower level
      //

      // find the lower ref level
      // findElement() is a helper function. see below.
      //
      Level lowerLevel = findElement( typeof( Level ), "Lower Ref. Level" )
as Level;
      Reference ref3 = lowerLevel.PlaneReference;

      // find the face of the box
      // findFace() is a helper function. see below.
      PlanarFace lowerFace = findFace( pBox, new XYZ(0.0, 0.0, -1.0) ); //
find a face whose normal is z-down.
      Reference ref4 = lowerFace.Reference;

      // create alignments
      //
      _rvtDoc.FamilyCreate.NewAlignment( pView, ref3, ref4 );

      //
      // (3)  same idea for the Right/Left/Front/Back
      //
      // get the plan view
      // note: same name maybe used for different view types. either one
should work.
      View pViewPlan = findElement( typeof( ViewPlan ), "Lower Ref. Level" )
as View;

      // find reference planes
      ReferencePlane refRight = findElement( typeof( ReferencePlane ),
"Right" ) as ReferencePlane;
```

```
      ReferencePlane refLeft = findElement( typeof( ReferencePlane ),
"Left" ) as ReferencePlane;
      ReferencePlane refFront = findElement( typeof( ReferencePlane ),
"Front" ) as ReferencePlane;
      ReferencePlane refBack = findElement( typeof( ReferencePlane ),
"Back" ) as ReferencePlane;

      // find the face of the box
      PlanarFace faceRight = findFace( pBox, new XYZ( 1.0, 0.0, 0.0 ) );
      PlanarFace faceLeft = findFace( pBox, new XYZ( -1.0, 0.0, 0.0 ) );
      PlanarFace faceFront = findFace( pBox, new XYZ( 0.0, -1.0, 0.0 ) );
      PlanarFace faceBack = findFace( pBox, new XYZ( 0.0, 1.0, 0.0 ) );

      // create alignments
      //
      _rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refRight.Reference,
faceRight.Reference );
      _rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refLeft.Reference,
faceLeft.Reference );
      _rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refFront.Reference,
faceFront.Reference );
      _rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refBack.Reference,
faceBack.Reference );
    }
```

There is the third helper function we use in this function, i.e.,

```
findFace()
```

This helper function finds a planar face with the given normal from an extrusion solid. The full code is attached at the end of this doc, the section, Appendix A. Please copy it and paste to the end of this class.

Let's focus on the first portion of the code where we add an alignment between the top face of the solid and the reference plane "Upper Ref Level". Once you understand one, the rest should be more or less the same.

```
_rvtDoc.FamilyCreate.NewAlignment(pView, ref1, ref2)
```

This is the main method to create a new alignment. It takes a view, and two references as arguments. As we do in UI, to align the top face to the upper reference plane, we look at the model from a side. We use Front view here. findElement() and findFace() are helper functions.

One thing to note is that unlike UI, API method NewAlignment will not automatically calculate and adjust the geometry of the model. Here is the excerpt from the RevitAPI.chm file:

> *"These references must be already geometrically aligned (this function will not force them to become aligned)."*

You will need to make sure that they are at the same location before you call NewAlignment.

4.2 Call `addAlignments(pSolid)` from your main command execute:

```
public Result Execute(
  ExternalCommandData commandData,
  ref string message,
  ElementSet elements )
{
    ...

  // (1) create a simple extrusion. just a simple box for now.
  Extrusion pSolid = createSolid();

  // (2) add alignment
  addAlignments( pSolid );
  ...
```

4.3 Your code should build and run at this point. Your column should respond when you change a reference; for example, when you move the level.

5. Add Types

Let's add a couple of types now, for example, ones with dimensions corresponding to "Width" and "Depth" to:

- 600 x 900
- 1000 x 300
- 600 x 600

5.1 Add the follow functions to the class:

```
// ======================================
//   (3) add types
// ======================================
void addTypes()
{
  // addType(name, Width, Depth)
  //
  addType( "600x900", 600.0, 900.0 );
  addType( "1000x300", 1000.0, 300.0 );
  addType( "600x600", 600.0, 600.0 );
}

// add one type
//
void addType( string name, double w, double d )
{
  // get the family manager from the current doc
  FamilyManager pFamilyMgr = _rvtDoc.FamilyManager;

  // add new types with the given name
  //
  FamilyType type1 = pFamilyMgr.NewType( name );
```

```
      // look for 'Width' and 'Depth' parameters and set them to the given
value
      //
      // first 'Width'
      //
      FamilyParameter paramW = pFamilyMgr.get_Parameter( "Width" );
      double valW = mmToFeet( w );
      if ( paramW != null )
      {
        pFamilyMgr.Set( paramW, valW );
      }

      // same idea for 'Depth'
      //
      FamilyParameter paramD = pFamilyMgr.get_Parameter( "Depth" );
      double valD = mmToFeet( d );
      if ( paramD != null )
      {
        pFamilyMgr.Set( paramD, valD );
      }
    }
```

The key class that you need to get hold of is a Family Manager object of the given document. You can access it through:

```
_rvtDoc.FamilyManager
```

Once you have a family manager, you can create a new type using NewType method:

```
pFamilyMgr.NewType(name)
```

You can access to a parameter of your interest, using (e.g.):

```
pFamilyMgr.Parameter("Width")
```

Then sets its value, using:

```
pFamilyMgr.Set(paramW, valW)
```

The above code defines three types.

5.2 Call `addTypes()` from your main command function:

```
    public Result Execute(
      ExternalCommandData commandData,
      ref string message,
      ElementSet elements )
    {
      ...
      // (2) add alignment
      addAlignments(pSolid)
```

```
      // (3) add types
      addTypes();

      //  finally, return
      return Result.Succeeded;
    }
```

5.3 Your code should be ready to build and run.

6. Test Your Column

Your code is ready to build and run at this point to test if your column is created correctly.

Remember to start with Family Editor and use "Metric Column.rft" template.

After running a command, go to the type dialog, check to see if three types are created. Apply each of them, and see if your column changes its size accordingly.

In the next lab, we will modify the profile of the column and learn how to add reference planes, parameters, and dimensions.

Appendix A. Helper Functions Used in Lab1

In the Lab1, we use the following helper functions. Copy and paste from the code below to your code as required.

- findFace() - given a extrusion solid, find a planar face with the given normal.
- findElement() - find an element of the given type and the name. You can use this, for example to find Reference or Level with the given name.
- mmToFeet() - convert unit from millimeter to feet.

```
#region Helper Functions

    // ============================================================
    //   helper function: find a planar face with the given normal
    // ============================================================
    PlanarFace findFace(Extrusion pBox, XYZ normal)
    {
      // get the geometry object of the given element
      //
      Options op = new Options();
      op.ComputeReferences = true;
      IEnumerable<GeometryObject> geomObjs =
pBox.get_Geometry(op).AsEnumerable();

      // loop through the array and find a face with the given normal
      //
      foreach( GeometryObject geomObj in geomObjs )
      {
        if( geomObj is Solid ) // solid is what we are interested in.
```

```
        {
          Solid pSolid = geomObj as Solid;
          FaceArray faces = pSolid.Faces;
          foreach( Face pFace in faces )
          {
            PlanarFace pPlanarFace = pFace as PlanarFace;
            if( (pPlanarFace != null) &&
pPlanarFace.Normal.IsAlmostEqualTo( normal ) ) // we found the face
            {
              return pPlanarFace;
            }
          }
        }

        // will come back later as needed.
        //
        //else if (geomObj is Instance)
        //{
        //}
        //else if (geomObj is Curve)
        //{
        //}
        //else if (geomObj is Mesh)
        //{
        //}
      }

      // if we come here, we did not find any.
      return null;
    }

    // ===========================================================================
    //   helper function: find an element of the given type and the name.
    //   You can use this, for example, to find Reference or Level with the
given name.
    // ===========================================================================
    Element findElement(Type targetType, string targetName)
    {
      // get the elements of the given type
      //
      FilteredElementCollector collector = new
FilteredElementCollector(_rvtDoc);
      collector.WherePasses(new ElementClassFilter(targetType));

      // parse the collection for the given name
      // using LINQ query here.
      //
      var targetElems = from element in collector where
element.Name.Equals(targetName) select element;
      List<Element> elems = targetElems.ToList<Element>();

      if ( elems.Count > 0 ) {  // we should have only one with the given
name.
          return elems[0];
      }

      // cannot find it.
```

```
      return null;    }

    // ============================================
    //   helper function: convert millimeter to feet
    // ============================================
    double mmToFeet( double mmVal )
    {
      return mmVal / 304.8;
    }

#endregion // Helper Functions
```

Appendix B. Helper Class to Display a Message Box in C#

In the Labs, we use the following helper class to simplify the display of a Message Box in C#. Add a new .cs file, and copy and paste from the code below to your code as required.

```
#region Namespaces
using System;
using System.Diagnostics;
using WinForms = System.Windows.Forms;
#endregion // Namespaces

namespace FamilyLabsCS
{
  public class Util
  {
    #region Formatting and message handlers
    public const string Caption = "Revit Family API Labs";

    /// <summary>
    /// MessageBox wrapper for informational message.
    /// </summary>
    public static void InfoMsg( string msg )
    {
      Debug.WriteLine( msg );
      WinForms.MessageBox.Show( msg, Caption, WinForms.MessageBoxButtons.OK,
WinForms.MessageBoxIcon.Information );
    }

    /// <summary>
    /// MessageBox wrapper for error message.
    /// </summary>
    public static void ErrorMsg( string msg )
    {
      WinForms.MessageBox.Show( msg, Caption, WinForms.MessageBoxButtons.OK,
WinForms.MessageBoxIcon.Error );
    }
    #endregion // Formatting and message handlers
  }
}
```