Lesson 1: The Basic Plug-in

In this lesson you will create your very first basic Autodesk Revit plug-in for copying groups selected by the user to a specified location.

Video: A Demonstration of Lesson 1 Steps to Create your First Plug-in

Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

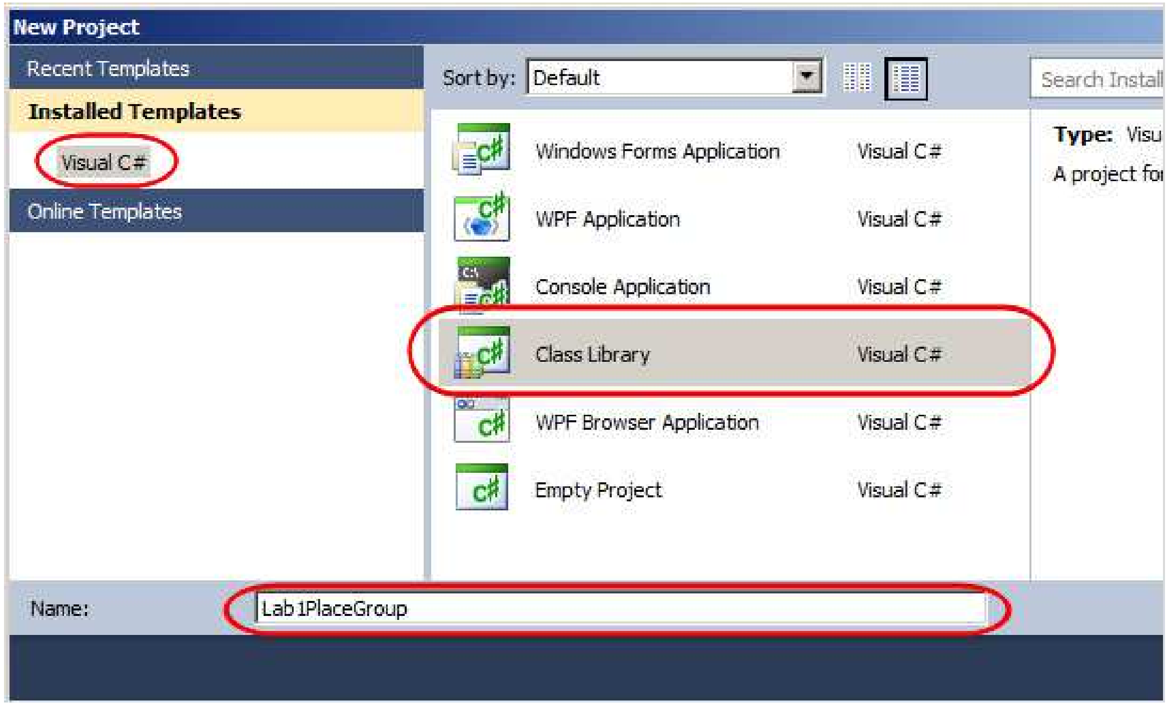
Lesson Downloads

- lesson1_revit_2014_projects.zip (zip - 20216Kb)
- lesson1_revit_2013_projects.zip (zip - 28884Kb)
- lesson1_revit_2012_and_earlier_project_files.zip (zip - 7283Kb)

Steps to Create Your First Plug-in

1. **Launch the Visual C# Express development environment:**
Open Visual C# 2010 Express using the Windows **Start** menu, selecting **All Programs**, and then **Microsoft Visual Studio 2010 Express** and then the **Microsoft Visual C# 2010 Express** submenu-item.

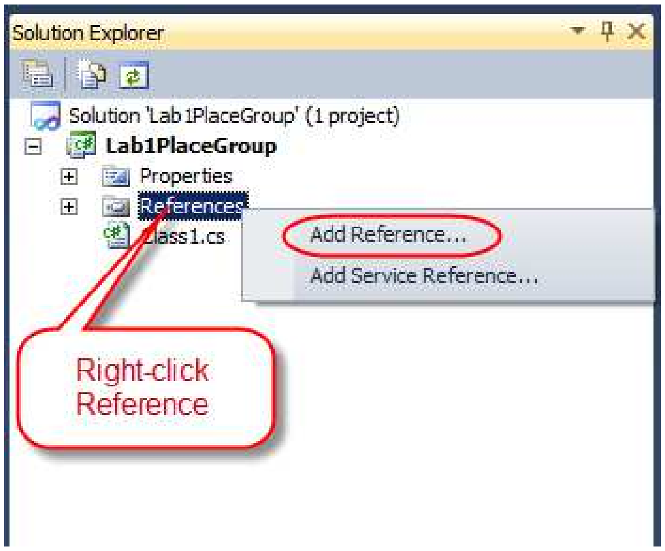
2. **Create a class library project:**
Inside Visual C# Express, on the **File** menu, click **New Project**. In the **Installed Templates** tab in the left-hand window, click **Visual C#**. In the middle window, click **Class Library**. Enter **Lab1PlaceGroup** in the **Name** box. And then click **OK**.



Visual C# Express will create a default code project for you and display the code in the code window.

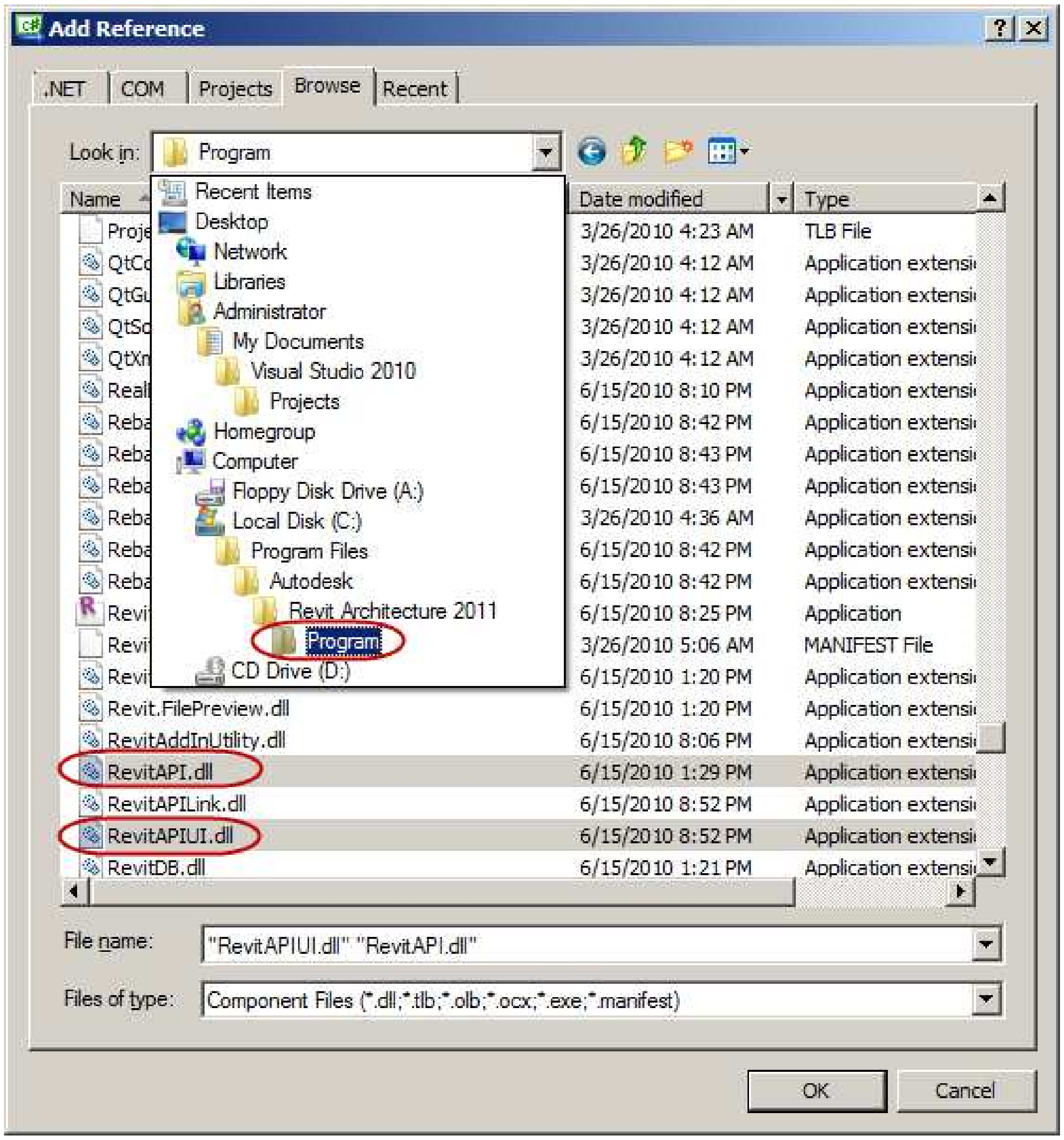
3. **Save the project:**
On the **File** menu, click **Save All**. In the display window type ***C:\test*** in the **Location** box, and then click Save.

4. **Add references:**
In the **Solution Explorer** window on the right-hand side of the Visual C# Express window, right-click **References** and click **Add Reference…**



5. Click the **Browse** tab in the **Add Reference** dialog and browse to the Revit product installation sub-folder. (The sub-folder path depends on where you have installed Revit Architecture 201x. The default path is *C:\Program Files\Autodesk\Revit Architecture 201x\Program**).

** The path may vary depending on the flavour of Autodesk Revit you are using.*



Look in: Program

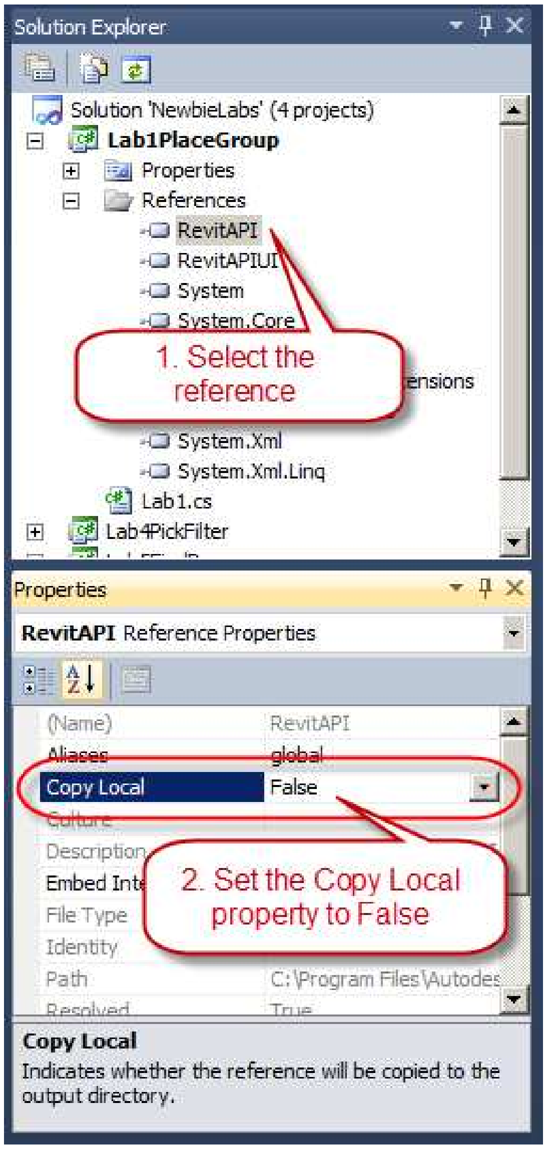
Name	Date modified	Type
Proje	3/26/2010 4:23 AM	TLB File
QtCo	3/26/2010 4:12 AM	Application extensi
QtGu	3/26/2010 4:12 AM	Application extensi
QtSo	3/26/2010 4:12 AM	Application extensi
QtXn	3/26/2010 4:12 AM	Application extensi
Reali	6/15/2010 8:10 PM	Application extensi
Reba	6/15/2010 8:42 PM	Application extensi
Reba	6/15/2010 8:43 PM	Application extensi
Reba	6/15/2010 8:43 PM	Application extensi
Reba	3/26/2010 4:36 AM	Application extensi
Reba	6/15/2010 8:42 PM	Application extensi
Reba	6/15/2010 8:42 PM	Application extensi
Revi	6/15/2010 8:25 PM	Application
Revi	3/26/2010 5:06 AM	MANIFEST File
Revi	6/15/2010 1:20 PM	Application extensi
Revit.FilePreview.dll	6/15/2010 1:20 PM	Application extensi
RevitAddInUtility.dll	6/15/2010 8:06 PM	Application extensi
RevitAPI.dll	6/15/2010 1:29 PM	Application extensi
RevitAPILink.dll	6/15/2010 8:52 PM	Application extensi
RevitAPIUI.dll	6/15/2010 8:52 PM	Application extensi
RevitDB.dll	6/15/2010 1:21 PM	Application extensi

File name: "RevitAPIUI.dll" "RevitAPI.dll"

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

You will add two reference files from this folder. Select **RevitAPI.dll**, hold the Ctrl key and select **RevitAPIUI.dll**, and then click **OK**. Now the two interface DLL files are referenced in your project. All the Revit APIs are exposed by these interface files and your project can use all of those available APIs from them.

6. **Set the referenced files' Copy Local property value:**



(Name)	RevitAPI
Aliases	global
Copy Local	False
Culture	
Description	
Embed Inte	
File Type	
Identity	
Path	C:\Program Files\Autodes
Resolved	True

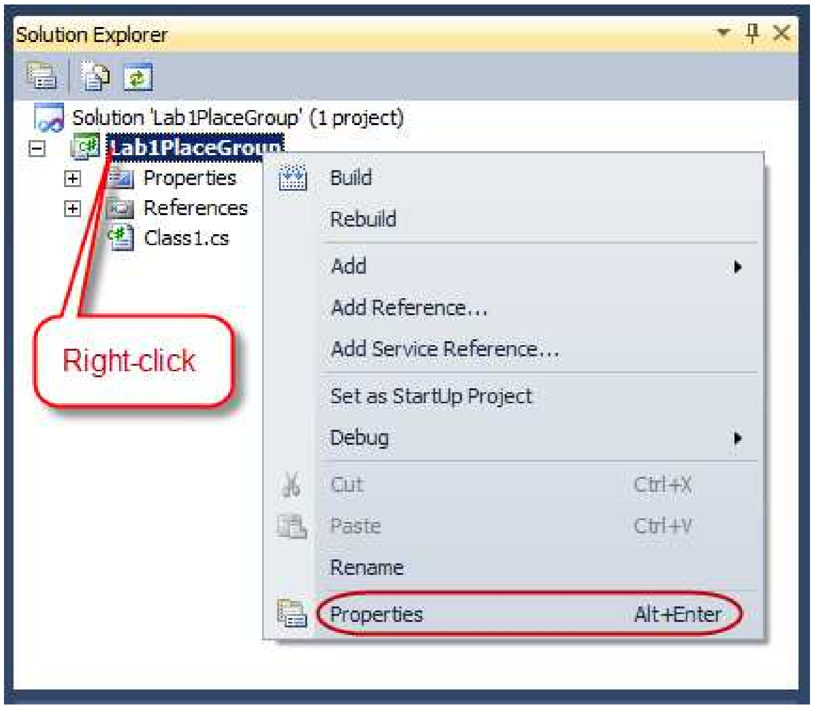
Copy Local
Indicates whether the reference will be copied to the output directory.

In the **Solution Explorer** window you saw in step 5, click **RevitAPI** under **Reference** node. In the **Properties** window, click **Copy Local** property, and then click the drop-down list, select **False**. Repeat the same steps to change **RevitAPIUI**'s **Copy Local** property value to **False**.

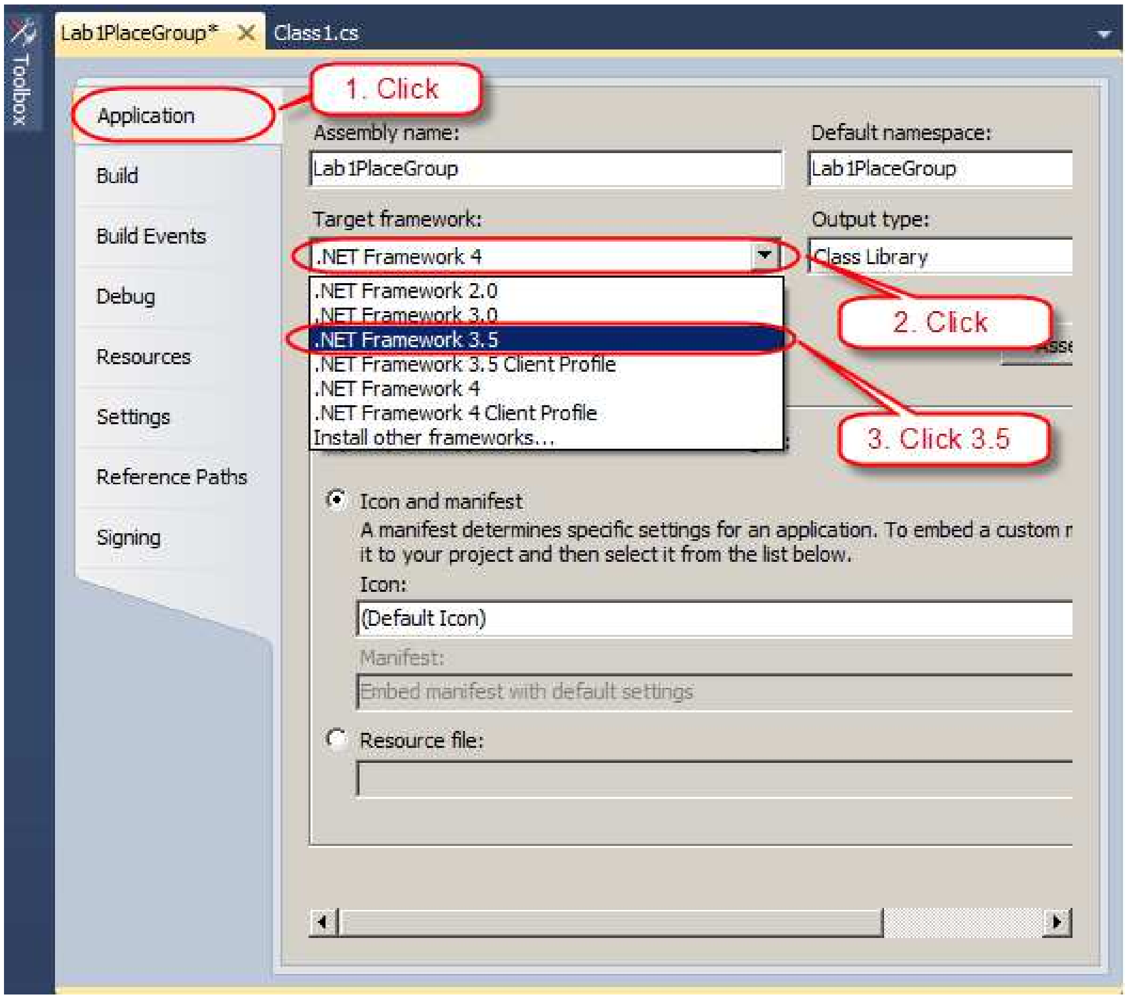
7. **Set target .NET Framework:**

Attention: Autodesk Revit 2011 supports the use of .NET Framework 3.5. Autodesk Revit 2012 and higher supports the use of .NET Framework 4.0, which Visual C# 2010 Express uses by default. The following step is needed in order to use the correct version. If the Revit Architecture version that you are using supports .NET Framework 4.0, you can skip step 7, 8 and 9.

In the **Solution Explorer** window, right-click **Lab1PlaceGroup** project, and click **Property**.



8. In the displaying Project Property form, click **Application** tab on the left-hand side of the window, click the drop-down list below **Target framework**, and then click the **.NET Framework 3.5** option in the list.

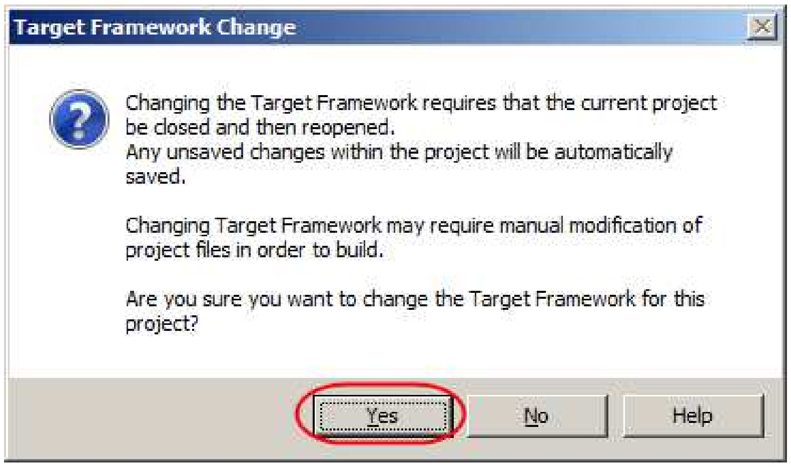


Assembly name: Lab1PlaceGroup
Default namespace: Lab1PlaceGroup
Target framework: .NET Framework 4
Output type: Class Library

.NET Framework 2.0
.NET Framework 3.0
.NET Framework 3.5
.NET Framework 3.5 Client Profile
.NET Framework 4
.NET Framework 4 Client Profile
Install other frameworks...

Icon and manifest
A manifest determines specific settings for an application. To embed a custom r
it to your project and then select it from the list below.
Icon:
(Default Icon)
Manifest:
Embed manifest with default settings
Resource file:

9. The following message box appears to ask your confirmation. Click **Yes** to confirm the change.



Target Framework Change

Changing the Target Framework requires that the current project be closed and then reopened.
Any unsaved changes within the project will be automatically saved.

Changing Target Framework may require manual modification of project files in order to build.

Are you sure you want to change the Target Framework for this project?

10. **Add the code:**
Double click **Class1.cs** in the **Solution Explorer** window to show the code-editing window. Delete everything in this window and then type the following C# code. To get the full experience of developing with Visual C# Express – including the use of features such as IntelliSense – we recommend you type the code from this guide rather than copying and pasting it. That said, if constrained for time you can also copy and paste into the Visual C# Express code window: although this reduces the experience you gain from working with the code directly.

```
using System;
using System.Collections.Generic;
using System.Linq;

using Autodesk.Revit.DB;
using Autodesk.Revit.DB.Architecture;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;

[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {
    //Get application and document objects
    UIApplication uiApp = commandData.Application;
    Document doc = uiApp.ActiveUIDocument.Document;

    //Define a Reference object to accept the pick result.
    Reference pickedRef = null;

    //Pick a group
    Selection sel = uiApp.ActiveUIDocument.Selection;
    pickedRef = sel.PickObject(ObjectType.Element, "Please select a group");
    Element elem = pickedRef.Element;
    Group group = elem as Group;

    //Pick a point
    XYZ point = sel.PickPoint("Please pick a point to place group");

    //Place the group
    Transaction trans = new Transaction(doc);
```

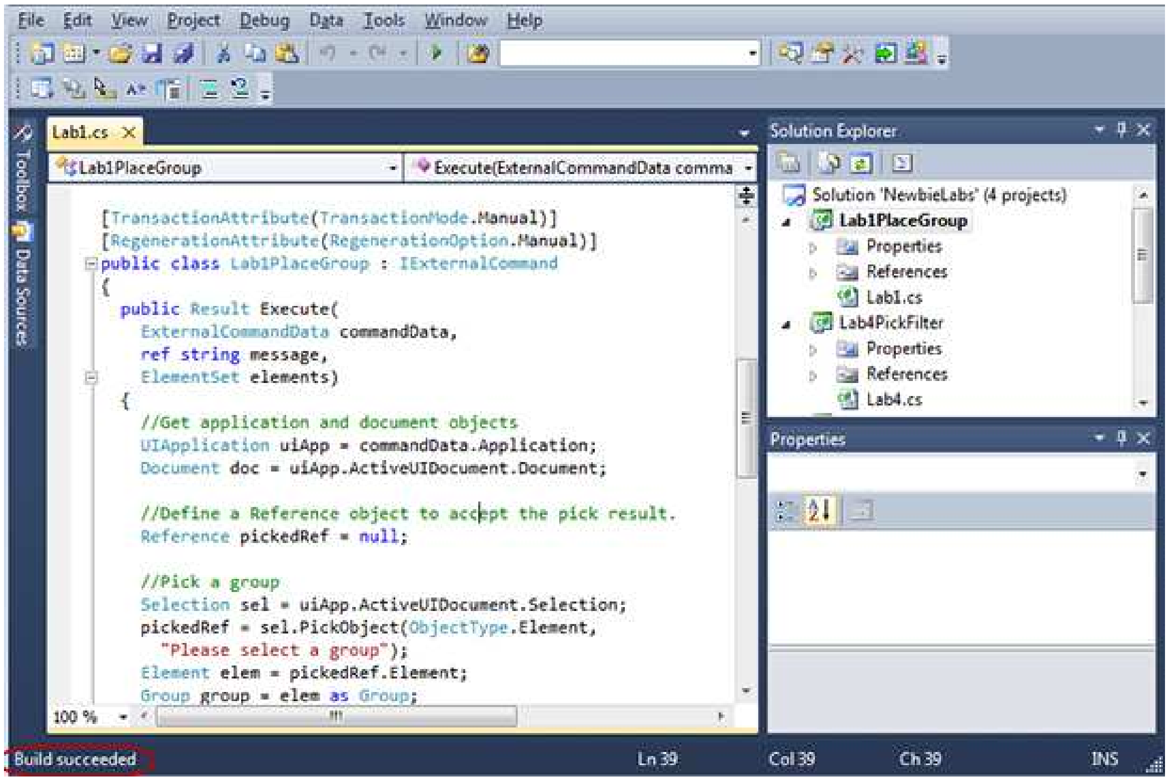
```
                trans.Start("Lab");
                doc.Create.PlaceGroup(point, group.GroupType);
                trans.Commit();

                return Result.Succeeded;
            }
        }
```

Don't worry about the details of the code for now, you'll come back to this shortly in the next couple of lessons.

11. **Save the file:**
On the **File** menu, click **Save All**.

12. **Build the project:**
The code you have written is in human readable form. To make the code readable by a computer, you will need to translate it or “build” it.

Inside Visual C# Express, in the **Debug** menu, click **Build Solution** to compile and build your plug-in. **Build Success** message shows in status bar of the Visual C# Express window if the code is successfully built.



```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {
    //Get application and document objects
    UIApplication uiApp = commandData.Application;
    Document doc = uiApp.ActiveUIDocument.Document;

    //Define a Reference object to accept the pick result.
    Reference pickedRef = null;

    //Pick a group
    Selection sel = uiApp.ActiveUIDocument.Selection;
    pickedRef = sel.PickObject(ObjectType.Element,
      "Please select a group");
    Element elem = pickedRef.Element;
    Group group = elem as Group;
```

```
Build succeeded          Ln 39     Col 39     Ch 39          INS
```

Note: If you are working with Revit 2012 API, you will see a warning stating that 'Autodesk.Revit.DB.Reference.Element' is obsolete. At this point, don't worry about this warning. We will address what the code needs to be changed to, in Lesson 3.

If you are working with Revit 2013 and higher API, you will need to replace the following line of code:

```
Element elem = pickedRef.Element;
```

with the following:

```
Element elem = doc.GetElement(pickedRef);
```

That’s it! You have just written your first plug-in for Autodesk Revit.

Before you actually work with the plug-in in Revit, you will need to do one more step, which is to write an AddIn manifest.

Writing an AddIn Manifest

An AddIn manifest is a file located in a specific location checked by Revit when the application starts. The manifest includes information used by Revit to load and run the plug-in.

1. **Add the manifest code:**
Start **Notepad.exe** from the Windows **Start** menu. Copy and paste the following plug-in load settings to the Notepad editor.

```
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="Command">
<Assembly>
  C:\test\Lab1PlaceGroup\Lab1PlaceGroup\bin\Release\Lab1PlaceGroup.dll
</Assembly>
    <ClientId>502fe383-2648-4e98-adf8-5e6047f9dc34</ClientId>
    <FullClassName>Lab1PlaceGroup</FullClassName>
    <Text>Lab1PlaceGroup</Text>
    <VendorId>ADSK</VendorId>
    <VisibilityMode>AlwaysVisible</VisibilityMode>
  </AddIn>
</RevitAddIns>
```

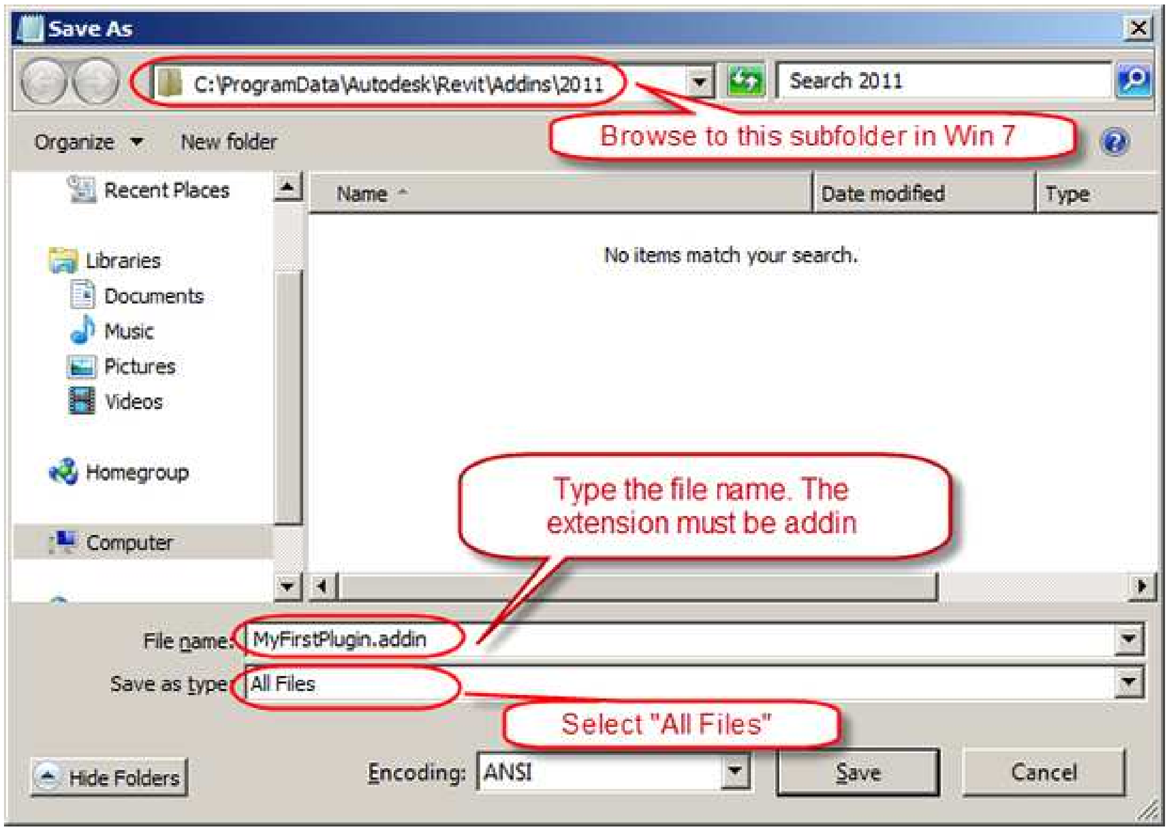
Depending on what version you are using you may need to change the path here to match your Lab1PlaceGroup.dll location on your computer:
C:\test\Lab1PlaceGroup\Lab1PlaceGroup\bin\Release\Lab1PlaceGroup.dll

11. **Save the file:**
On **Notepad's File** menu, click **Save.** Enter **MyFirstPlugin.addin** in the **File name** box. Change **Save as type** to the **All Files** option (the file name is up to you; however the file extension must be ".addin"). Browse to the following subfolder, and then click the **Save** button.

- For Windows XP* - *C:\Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2011\*
- For Windows Vista/Windows 7* - *C:\ProgramData\Autodesk\Revit\Addins\2011\* (The ProgramData folder is hidden by default)

** The path may vary depending on the flavour of Autodesk Revit you are using.*

For example, here is the setting in Save As dialog in Windows 7 for Revit Architecture 2011.



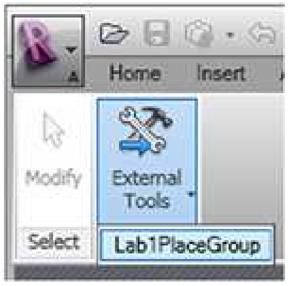
Let's run the plug-in to see what it does in Revit.

Running the Plug-in

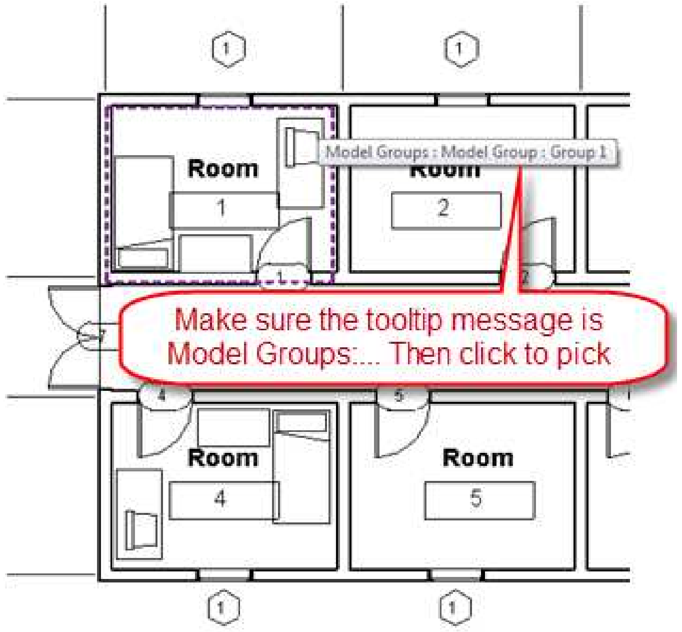
1. Start Autodesk Revit Architecture.

2. Open the Hotel.rvt Project file.

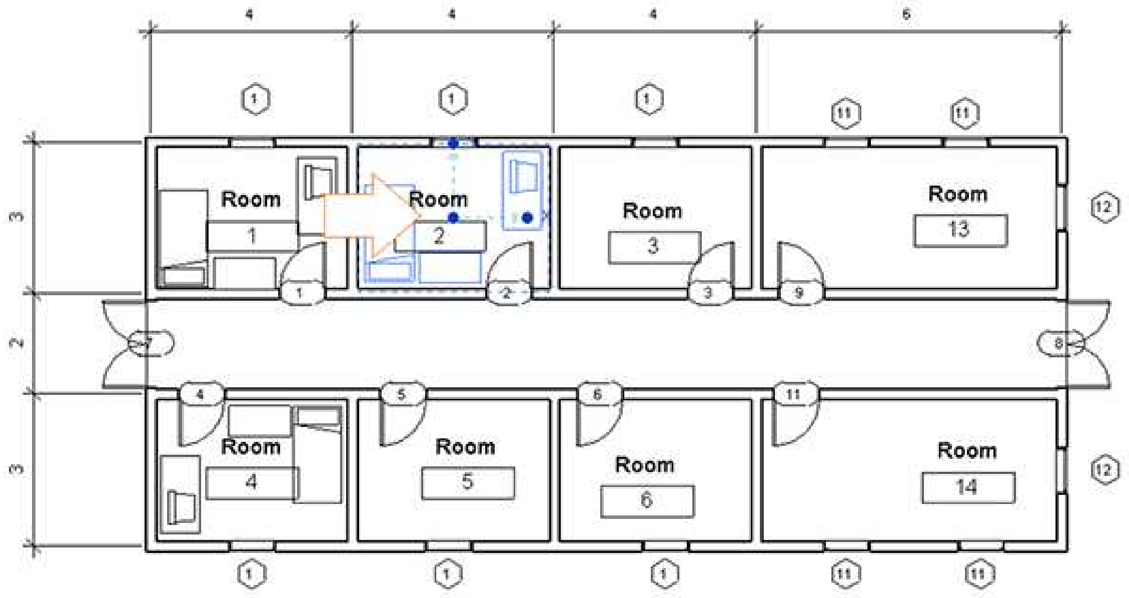
3. **Load your plug-in into Revit and allow the plug-in to communicate with Revit:**
Inside Revit on the **Add-Ins** ribbon tab, click the **External Tools** drop-down list, then click **Lab1PlaceGroup**. This will start your plug-in.



4. **Work with the plug-in:**
Move the cursor over **Room1** in the Revit building model. When the cursor is hovering over the furniture group, its bounding box should be highlighted as per the below picture, with a tooltip showing **Model Groups : Model Group : Group 1**. Click to select this furniture group. (Note: when highlighted the room looks very similar to the group. Please carefully select the group according to the message in the tooltip. If the room is selected, you will not see the expected result after the following step.)


Model Groups : Model Group : Group 1
Make sure the tooltip message is Model Groups:... Then click to pick
Room
1
Room
2
Room
4
Room
5

5. Pick a point in another room, for example in **Room 2**. You should see the group copied to this location. The center of the new group is the point you selected.



Congratulations! You have just written your first plug-in for Autodesk Revit. You will be reviewing the code in detail in Lesson 3.

Before you move on to the next lessons, let us go back to some of the things we skipped over earlier, starting with basics concept about programming, and the benefits it can bring to your day-to-day work.

Additional Topics

Introduction to Programming

The C# code you have just executed to copy a group is only 30 lines long. Here you see a small amount of code working in a similar way to the internal Revit command, **Create Similar**. Software programming allows you to capture the logic of a particular functionality once and then reap the benefits over and over again, every time you want to perform this functionality.

What is Programming?

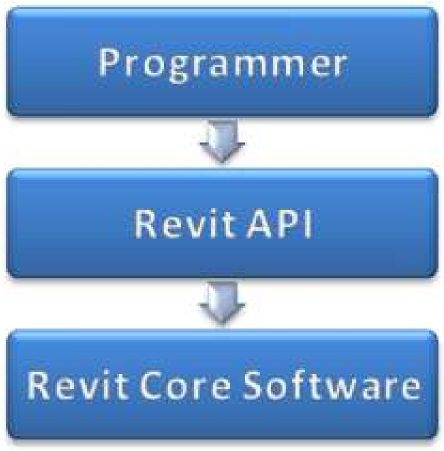
A simple answer to this question is: Computer programming is the process of creating a sequence of instructions to tell the computer to do something. You can look at your program as a sequence of instructions. During the course of the upcoming lessons, you will look at the various lines and blocks of code and look at them all in the context of being instructions for a computer.

If you were to explain what computers are to a young child, you might say: a computer is a tool which follows instructions you provide. Programming is one way of giving instructions to the computer. Internally, a computer sees these instructions encoded as a series of numbers (also called machine code). The human-readable instructions you saw at the beginning of this lesson is called source code and the computer converts these instructions to machine code which it can then read and execute. A sequence of such instructions (or code), written to perform a specific task, is called a program and a collection of such programs and related data is called a software. Autodesk Revit is one such software product.

Source code can be written in different languages, just as humans use different languages to communicate between ourselves. The language you will be using in this guide is called C# (pronounced “C-Sharp”).

What is an API?

API is the acronym for Application Programming Interface: the way a software programmer can communicate with a software product. For instance, the Revit API is the way programmers can work with Revit, and it establishes what functionality a software programmer can use within Revit. Such as the Revit API allows you to write instructions for Revit to execute one after the other.



To put this slightly differently: commercial software companies, such as Autodesk, often distribute a set of libraries which you can use in your own program to interact with a particular software product, such as Autodesk Revit, and extend its functionality. This set of libraries is known as the software product’s API.

The type of program you write to interact with a software product and extend its functionality will depend upon how the API has been designed and what has been exposed (through APIs) for you to work with.

What is a Plug-in?

A software plug-in is a type of program module (or file) that adds functionality to a software product, usually in the form of a command automating a task or some customization of the product’s behavior. When you talk about a plug-in for Revit – and you will also hear the term Add-In used for this product – we mean a module containing code that makes use of the Revit API. Revit loads such plug-ins and uses them to adjust its behavior under certain conditions, such as when a particular command is executed by the user of the plug-in.

Lesson 2: Programming Overview

Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

In the previous lesson, you saw how you can increase productivity in Autodesk Revit by implementing a plug-in built from a small amount of C# code.

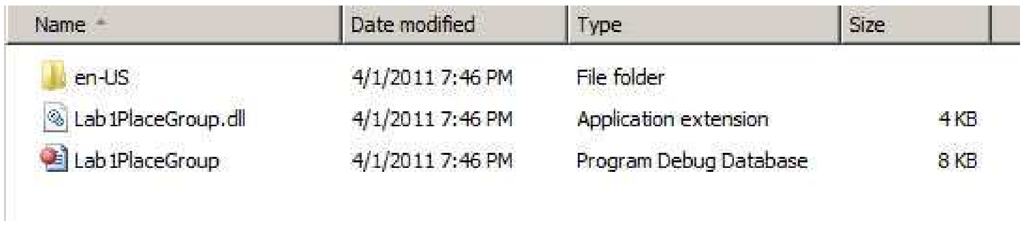
You have heard the term .NET from Lesson 1 with reference to programming with Revit. .NET is a technology that enables communication between software, if you are interested in learning more, you will find information in the Additional Topics section here.

You will now look more closely at what happened when you built and executed the code in the previous lesson.

What does it mean to "build" code?

The code you typed into Visual C# Express in Lesson 1 was a set of human-readable instructions (source code) that needed to be converted into code that could be understood and executed by the computer. The "build" you performed did just that: it packaged up the resulting executable code into a DLL (Dynamic-Link Library) that can be loaded into Autodesk Revit.

The following screenshot shows the output in DLL form along with the associated program debug database (which provides additional information when troubleshooting the DLL), once you have built the solution from Lesson 1 using Visual C# Express. The path to which the DLL gets compiled is specified in the Visual C# Express project settings and is set, by default, to the bin sub-folder of the Visual C# Express project folder.



Name	Date modified	Type	Size
en-US	4/1/2011 7:46 PM	File folder	
Lab1PlaceGroup.dll	4/1/2011 7:46 PM	Application extension	4 KB
Lab1PlaceGroup	4/1/2011 7:46 PM	Program Debug Database	8 KB

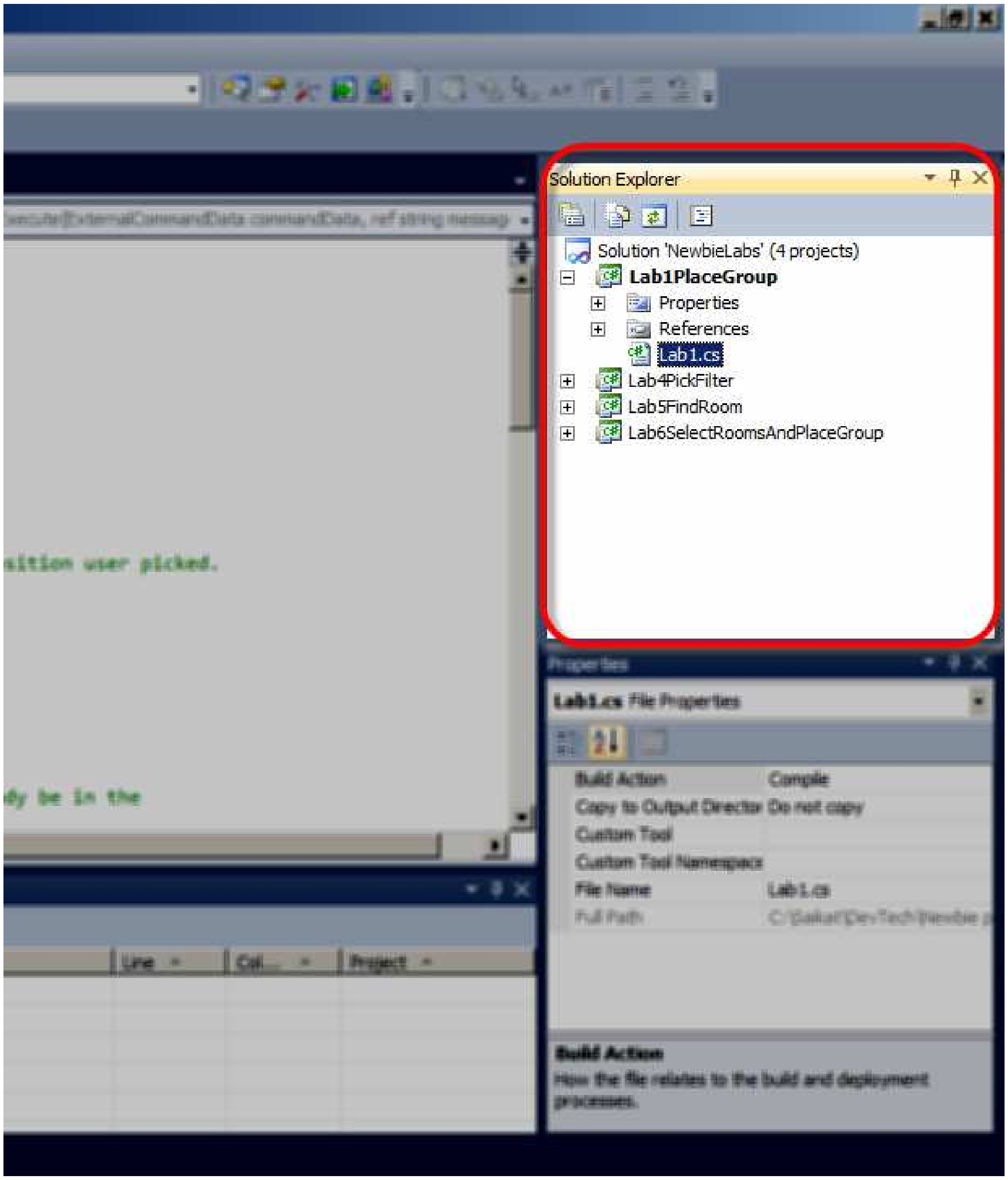
Choosing a Programming Language and Development Tool

Just as humans use different languages to communicate, you have various language options available to you when creating a Revit plug-in: for the purposes of this guide we have chosen C#, a strong general-purpose programming language that is popular with Revit developers.

There are a number of tools available for developing C# code. They range from open source tools such as SharpDevelop to Microsoft's flagship, professional development environment, Visual Studio. In your case you will be using Visual C# Express, a free version of Visual Studio focused on building C# applications.

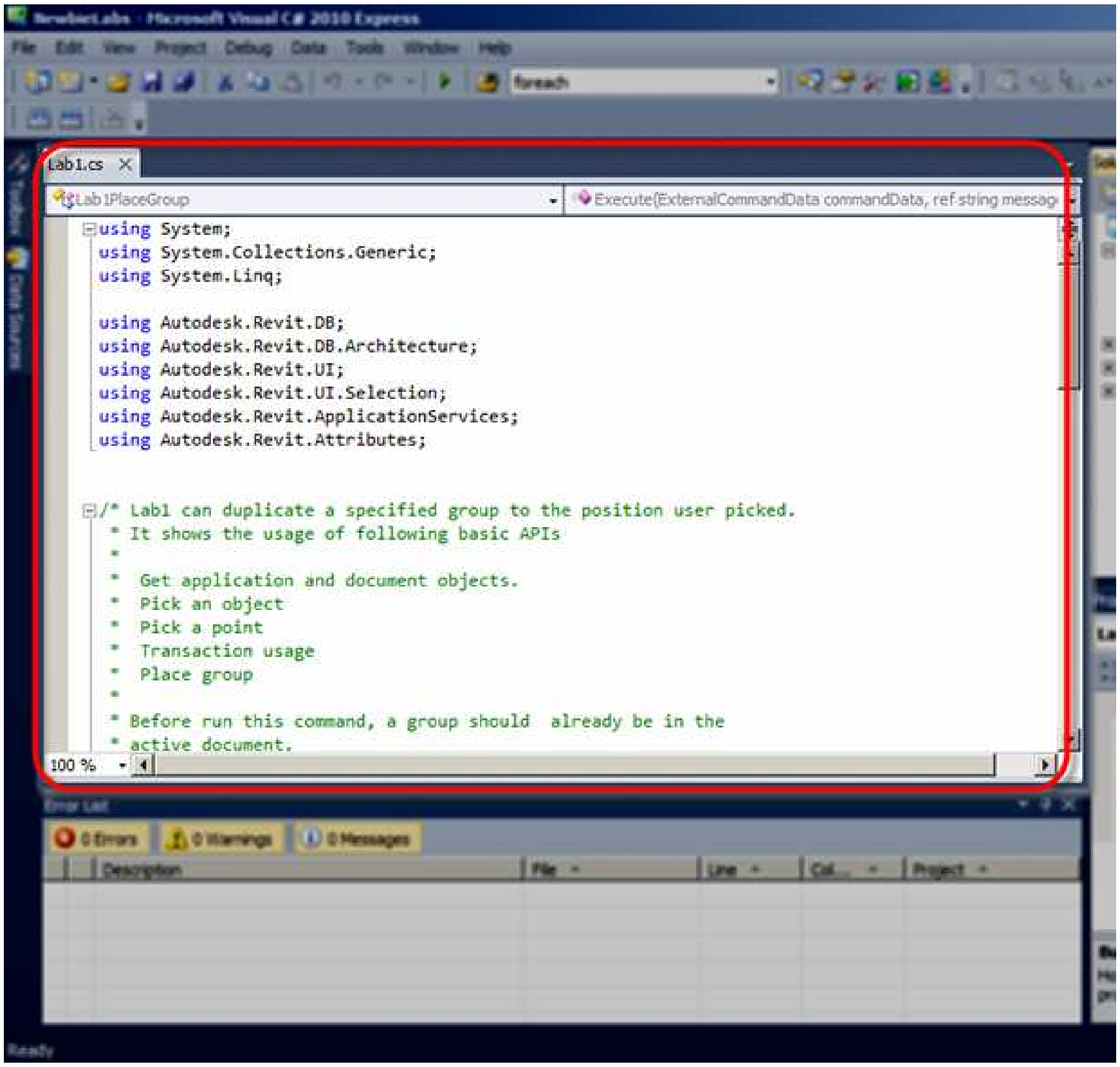
Visual C# Express is an Integrated Development Environment (IDE) because it is composed of various tools, menus and toolbars which ease the creation and management of your code.

The project system in Visual C# Express comprises of Solution and Project files as well as Project Items, the individual files belonging to projects. A solution is a container for one or more projects. Each project can in turn be considered a container for project items – such as source files, icons, etc. – most of which get compiled into the resultant executable file (EXE or DLL). Visual C# Express provides a Solution Explorer that organizes and displays the contents of the loaded solution in a tree-view format:



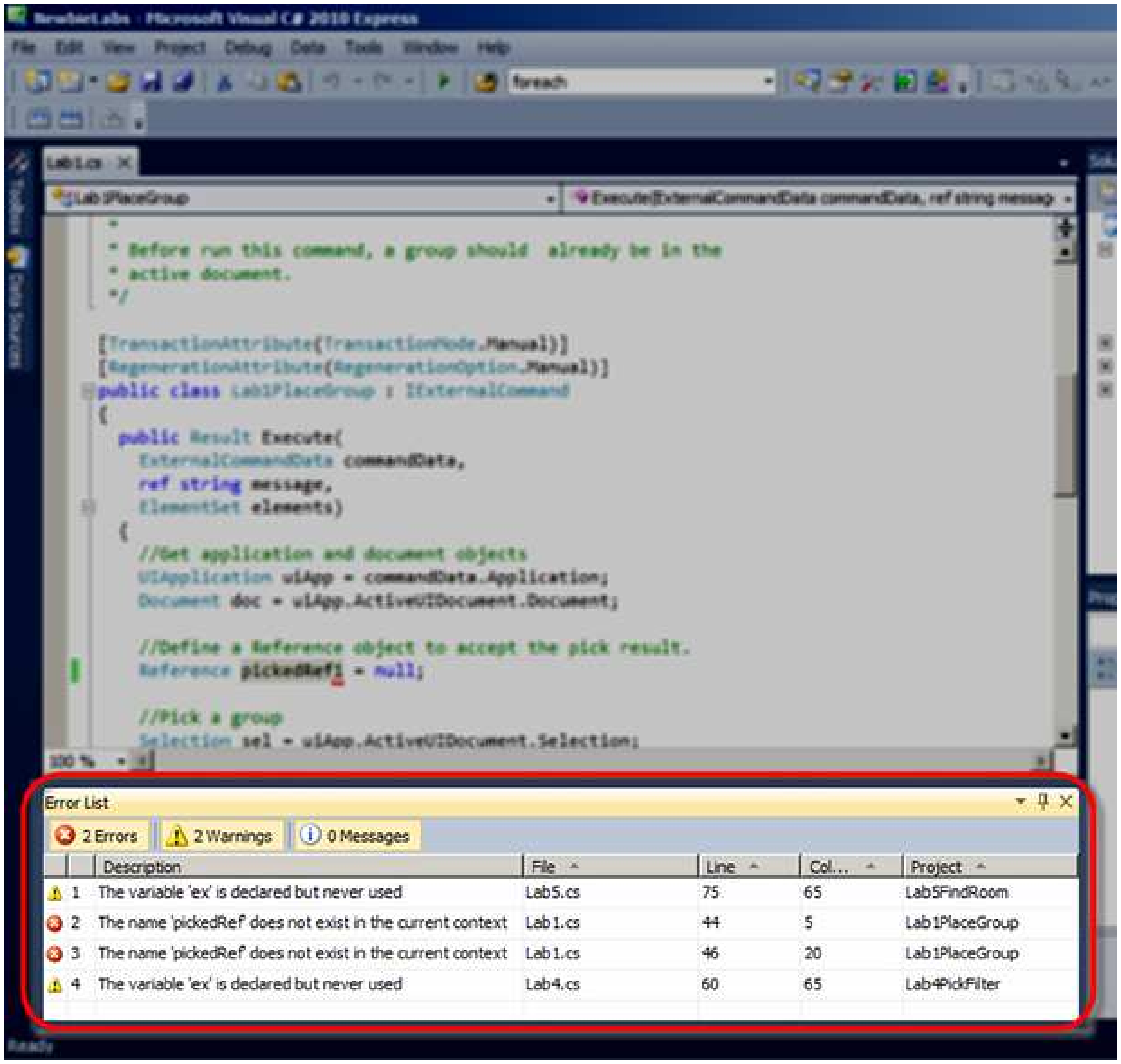
The Visual C# Express interface also contains a text editor and interface designer. These are displayed in the main window depending on the type of file being edited. The text editor is where you will enter the C# code for your Revit plug-in. This editor provides advanced features such as IntelliSense and collapsible code sections along with the more classic text-editing features such as bookmarks and the display of line numbers.

IntelliSense is an extremely valuable feature of the Visual Studio family that greatly improves programmer productivity: it automatically provides suggestions for the code being written based on the objects available and the letters that are being typed.


NewbieLabs - Microsoft Visual C# 2010 Express
File Edit View Project Debug Data Tools Window Help
foreach
Lab1.cs
Lab1PlaceGroup
Execute(ExternalCommandData commandData, ref string message
100 %
Error List
0 Errors
0 Warnings
0 Messages
Description
File
Line
Col...
Project
Ready

```
using System;
using System.Collections.Generic;
using System.Linq;

using Autodesk.Revit.DB;
using Autodesk.Revit.DB.Architecture;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;


/* Lab1 can duplicate a specified group to the position user picked.
 * It shows the usage of following basic APIs
 *
 *  Get application and document objects.
 *  Pick an object
 *  Pick a point
 *  Transaction usage
 *  Place group
 *
 * Before run this command, a group should  already be in the
 * active document.
```

Clearly one of the key features of Visual C# Express is its ability to build C# code into an executable file. During the build process, the language compiler performs various checks and analyses on the code. One such check is to ensure the code conforms to the syntactical rules of the C# language. The compiler also performs various other checks, such as whether a variable has been appropriately defined or not. Detected errors are reported via the Error List window, typically found at the bottom of the main window.



```
     *
     * Before run this command, a group should  already be in the
     * active document.
     */

    [TransactionAttribute(TransactionMode.Manual)]
    [RegenerationAttribute(RegenerationOption.Manual)]
    public class Lab1PlaceGroup : IExternalCommand
    {
      public Result Execute(
        ExternalCommandData commandData,
        ref string message,
        ElementSet elements)
      {
        //Get application and document objects
        UIApplication uiApp = commandData.Application;
        Document doc = uiApp.ActiveUIDocument.Document;

        //Define a Reference object to accept the pick result.
        Reference pickedRef1 = null;

        //Pick a group
        Selection sel = uiApp.ActiveUIDocument.Selection;
```

Error List

2 Errors | 2 Warnings | 0 Messages

		Description	File	Line	Col...	Project
⚠	1	The variable 'ex' is declared but never used	Lab5.cs	75	65	Lab5FindRoom
⊗	2	The name 'pickedRef' does not exist in the current context	Lab1.cs	44	5	Lab1PlaceGroup
⊗	3	The name 'pickedRef' does not exist in the current context	Lab1.cs	46	20	Lab1PlaceGroup
⚠	4	The variable 'ex' is declared but never used	Lab4.cs	60	65	Lab4PickFilter

Reviewing your use of Visual C# Express

In this section, you will review the steps performed using Visual C# Express from the previous lesson. However, we will put them in the context of what you have just learned about programming in general and building your code.

1. In the first step, you simply launched Visual C# Express.

2. You then created a new C# project of type Class Library.

   Since the development language used for this guide is C#, you are working with Visual C# Express, and therefore you see **Visual C#** under **Installed Templates** portion of the **New Project** dialog.

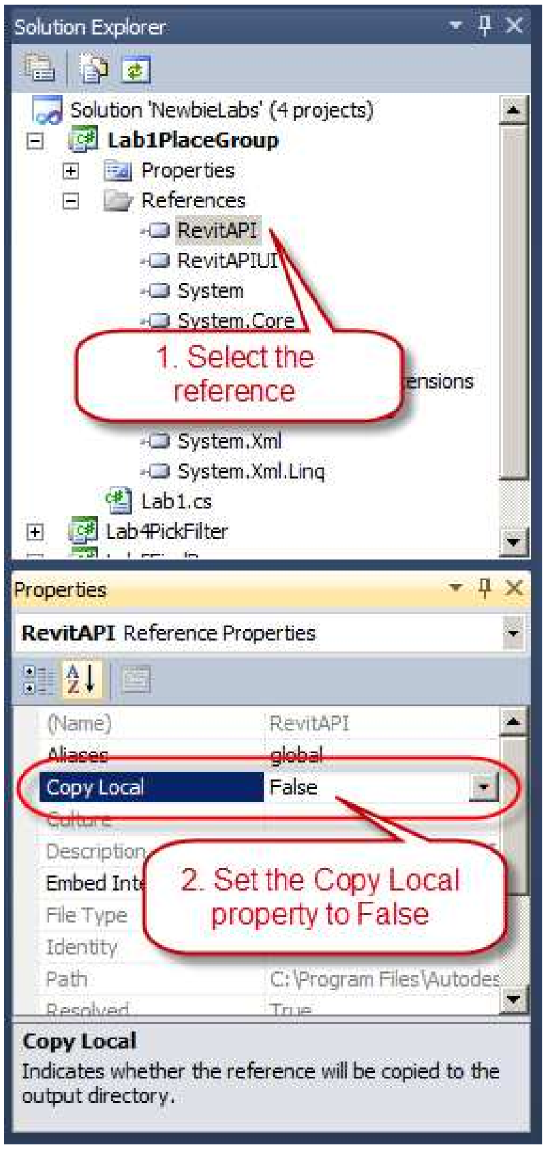
   In the middle section of this dialog, you saw various types of applications that can be created; you selected the template according to the type of application you wish to create.

   For plug-ins to load into Revit, they need to be Class Library assemblies (DLLs). It's for this reason, in the second step, that you selected the **Class Library** template. The name you entered is used to identify the project within the solution.

3. Your blank project was created, containing a few standard project references to core .NET components along with a blank C# class file. It's this file that gets displayed in the text editor window.

4. Saving the solution created physical files representing the contents of your class library project on the computer's hard drive, allowing you to open and edit it at another time in the future.

5. This blank project, as created by Visual C# Express, did not automatically make use of the Revit API. For it to do so, you added project references to the interface DLLs in Revit describing its API, *RevitAPI.dll* and *RevitAPIUI.dll*.



(Name)	RevitAPI
Aliases	global
Copy Local	False
Culture	
Description	
Embed Inte	
File Type	
Identity	
Path	C:\Program Files\Autodes
Resolved	True

Copy Local
Indicates whether the reference will be copied to the output directory.

When using the Revit API, it is usual to add project references to the two separate interface DLLs making up the API: one deals with core product functionality, the other with the product's user interface. You must link your project to these files to be able to work with Revit API.

RevitAPI.dll contains the APIs to access the Revit application, documents, elements, parameters, etc.

RevitAPIUI.dll contains the APIs related to manipulation and customization of the Revit user interface, including command, selections and dialogs

Having added your project references, it's important that you set one of their properties appropriately.

By default, Visual C# Express adds project references with its **Copy Local** property set to **True**. This means that the referenced DLLs will get copied to the project's output folder when it is built. In your case you wanted to change this setting to **False**, to make sure the DLLs did not get copied along with your assembly DLL.

To change this setting for the DLL, you selected the Revit API DLL (located under the **References** folder in the **Solution Explorer** on the right side of the Visual Studio interface) which populated the DLL reference's properties in the **Properties** panel below. In this properties panel, you saw a property called **Copy Local**. You clicked on it to change the value from *True* to *False* using the drop down list. Then you repeated these steps for the other DLL as well.

The reason for doing this is twofold: firstly, it's unnecessary to copy these files – they consume disk space and take time to copy – but, more importantly, the Common Language Runtime CLR can get confused about which copy of each DLL needs to be loaded if they exist in multiple places. Making sure the DLLs do not get copied with the project output means the CLR will correctly find the ones in the Revit folder.

Next you added C# code using the Revit API into your project. In other words providing Revit with instructions on how to perform the functionality of copying a user-selected group from one place to another.

While developing code, it's a good idea to build the solution from time to time, to check whether errors have been introduced in the code. The code does not necessarily have to be complete or functional when building the solution. This approach can help avoid potentially lengthy troubleshooting once the code is complete, and has the side benefit of automatically saving any edited source files before the build starts.

To build a solution inside Visual C# Express, select **Build Solution** from the **Debug** pull-down menu.

Tip: As a shortcut, you can use the Function key 'F6' to directly build the solution without accessing the menu.

If the build process was successful, you would see a **Build Succeeded** status in the bottom left corner of the Visual C# Express interface.

A quick recap: in this lesson you took a brief look at what happens when you build a project, as well as some background information on C# and Visual C# Express. You reviewed the steps you had taken in the previous lesson to build your basic Revit plug-in, putting it in the context of your learning about programming.

Additional Topics

What is .NET?

The .NET Framework is a software framework that sits on top of the Microsoft® Windows® operating system* and provides the underlying platform, libraries and services for all .NET applications. The services generally include memory management, garbage collection, common type system, class libraries, etc.

** Subsets of .NET are also available on other operating systems, whether via the open source Mono project or via Microsoft® Silverlight®, but these are not topics for this guide: you will focus solely on the use of .NET in the context of Microsoft Windows.*

What does the .NET Framework Contain?

The framework contains two main components:

1. **Common Language Runtime (CLR)** – This is the agent (or execution engine) in the .NET Framework responsible for managing the execution of code. Which is why code written to target this runtime is also known as managed code. All managed code runs under the supervision of the CLR, but what does this mean? The CLR manages code by providing core services such as memory management (which includes automatically releasing the computer's memory for reuse on other tasks when it is no longer needed), error (or exception) handling, managing the use of multiple threads of execution and ensuring rules around the use of different types of object are adhered to. The CLR is really the foundation of the .NET Framework.

2. **.NET Framework Class Library** – As the name suggests, this is a library or collection of object types that can be used from your own code when developing .NET applications. These .NET applications are targeted for Windows (whether command-prompt based or with a graphical user interface), the web or mobile devices. This library is available to all languages using the .NET Framework.

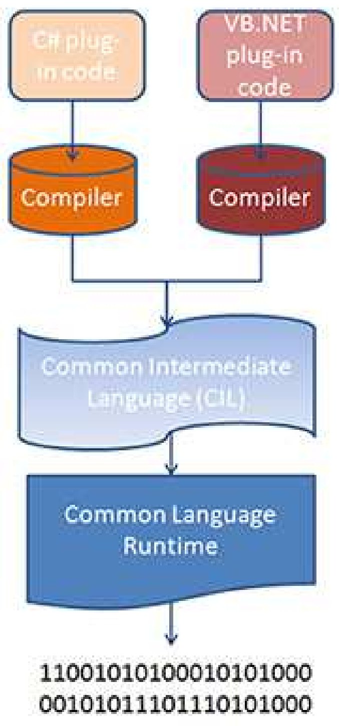
As mentioned above, the CLR improves code robustness by making sure the executing code conforms to a common type system (CTS). The CTS ensures that all .NET (or managed) code – irrespective of the language – uses a similar set of object types and can work together in the same environment. It is this feature that makes it possible for you to write applications in the development language of your choice and yet make use of components/code written by programmers using other .NET languages.

Building Executables

When you built your code into a DLL, it is compiled to Common Intermediate Language (CIL – also known as MSIL) code using a language-specific compiler. CIL is a CPU-independent set of instructions that can be executed by the CLR on Windows operating systems. CIL is typically portable across 32- and 64-bit systems and even – to some extent – to non-Windows operating systems. The CIL code generated from your C# source code is then packaged into a .NET assembly. Such an assembly is a library of CIL code stored in Portable Executable (PE) format (which contains both the CIL and its associated metadata). Assemblies can either be process assemblies (EXEs) or library assemblies (DLLs).

During the course of this guide, Revit plug-ins are compiled into library assembly files (DLLs) which are then loaded and executed from within Revit's memory space.

Running Executables



During execution of the .NET assembly, CIL (residing in the assembly) is passed through the CLR's just-in-time (JIT) compiler to generate native (or machine) code. JIT compilation of the CIL to native code occurs when the application is executed. As not all of the code is required during execution, the JIT compiler only converts the CIL when it is needed, thus saving time and memory. It also stores any generated code in memory, making it available for subsequent use without the need to recompile.

In the last step of this process, the native code gets executed by the computer's processor.

If you would like more details on the process of building .NET applications, please refer to the MSDN Library.

Lesson 3: A First Look at Code

Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

In this lesson, you will take your first close look at the Autodesk Revit API. You'll spend time looking at the C# code you typed into your plug-in during Lesson 1, understanding each of the lines of code in turn.

After reviewing the code you will find more detailed information on what is meant by Object-Oriented Programming and the concept of classes in the Additional Topics section here.

A Closer Look at the Lesson 1 Code

Keyword
When you look at the code, one of the first things you'll notice is the coloring: for readability Visual C# Express changes the color of certain words in the code. Words highlighted in blue are known as keywords, which can only be used in certain situations. You would not be able to declare a variable named **using**, for instance: as it is reserved to mean something very specific.

```
using System;
using System.Collections.Generic;
using System.Linq;

using Autodesk.Revit.DB;
using Autodesk.Revit.DB.Architecture;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
```

Namespaces
The first few lines of code start with the **using** keyword. When working with different library assemblies in your C# project, it's quite common for classes to share the same name – there might be multiple versions of the Point class in different libraries, for instance. To avoid naming conflicts, .NET provides you with the concept of namespaces. A namespace is a way to organize classes by grouping them together in a logical way and to help you identity the class you wish to use.

To give you an example, let's say there are two boys named John living in the same neighborhood. One possible way of uniquely identifying one of the Johns might be to use his last name. But if both Johns have the same last name, such as Smith, this is no longer enough to uniquely identify them. You now need to use an additional identifier, such as the school at which they study. If this was put into .NET syntax, you might use SchoolA.Smith.John for one and SchoolB.Smith.John for the other.

It's in this way that you're able to identify the use of specific classes in C#. The using keyword gives you direct access to the classes contained in the included namespace, which saves the effort of having to enter the full namespace when typing the names of classes. It also causes the namespace's various classes to be added to the list presented by the IntelliSense feature.

The using keyword is a directive telling the C# compiler to provide access to the contents of a specified namespace from the code in this source file.

Attributes

Let's look at the following block of code:

```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{

}
```

We'll start by talking about the first two lines inside the square brackets.

```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
```

These two lines are attributes that help control the transaction and regeneration behavior of your command.

1. Transaction Attribute – a transaction attribute declares the way the transaction should work. It could be either Manual or Automatic. Here, you have set the attribute to Manual.

   So what is a transaction? Transactions are objects that capture the changes to the Revit model. Changes to the Revit model can only be performed when there is an active transaction to do so. Transactions can either be committed – which means that their changes are written/recorded into the model – or rolled-back – which means their changes are undone.

   The value of TransactionMode.Manual for the TransactionAttribute requests that Revit not create a transaction automatically (besides the one Revit creates for any external command to help roll back any changes performed should the external command fail). This setting also implies that you are responsible for creating, naming, committing or aborting your transactions.

2. Regeneration Attribute – **this attribute applies only to Revit 2011 products**. A regeneration attribute sets the timing for Revit to draw graphical objects to the computer screen. It could be either Manual or Automatic. Here, you set have the attribute to Manual.

   RegenerationOption.Manual requests that Revit not regenerate (or refresh) the model on the screen after each change made to it. Instead you tell Revit when the model needs to be drawn or regenerated during the command execution process. If you set this as Automatic, Revit will try to draw every time something is drawn.

IExternalCommand

The line after the declaration of our attributes declares the class.

```
public class Lab1PlaceGroup : IExternalCommand
{

}
```

A class can be thought of as a type that can be used to describe a representation of a thing or an individual object. A class can be considered the template that describes the details of an object and is used to create individual objects. In this way classes can help define the common characteristics of objects of their type: the objects' attributes (properties) and behaviors (methods). For more details on understanding classes see Additional Topics.

The **public** keyword states that the class should be accessible from other classes in this plug-in as well in other Visual Studio projects which reference the plug-in DLL containing this class. The name **Lab1PlaceGroup** is the name of your class.

Next in the declaration statement, after the class name, you see a colon followed by **IExternalCommand**. Placing a colon after your class name followed by IExternalCommand tells the C# compiler that you want your class to implement an interface called IExternalCommand (which – if stated with the complete namespace – is actually Autodesk.Revit.UI.IExternalCommand).

From the Revit plug-in perspective, this denotes the class as the implementation of an external command: Revit will be able to use this class to execute a command when it gets triggered from the product's user interface. In simpler words, by declaring a class that implements IExternalCommand, Revit can recognize this as a command that you have defined. Interfaces are like classes except that they contain unimplemented methods. Interface names start with an "I", as is the case with IExternalCommand.

As your new class Lab1PlaceGroup implements the IExternalCommand interface from the Revit API, it needs to provide implementations for all the unimplemented methods defined by this interface. A common question when working with interfaces is this: once I've decided to implement an interface, how do I know what methods need to be implemented?

Thankfully Visual C# Express comes to the rescue, helping create the basic "skeleton" implementation of the method(s) that need to be implemented in the interface. If you right-click on the IExternalCommand word in the Code Editor window in Visual C# Express, you see an option in the context menu entitled **Implement Interface**. If you click on this further another sub-menu is provided, **Implement Interface Explicitly**. Clicking on this creates a blank method implementation for you to fill in.

The class in a Revit plug-in that implements this interface is known as the entry point for that plug-in: it's the class that Revit will attempt to find and call the **Execute()** method upon. Putting it another way, when a Revit user clicks on a command in the Revit user interface listed under the **External Tools** drop-down button on the **Add-Ins** tab, the code in the Execute() method is run (executed) from the corresponding class which implements this IExternalCommand interface.

```
[TransactionAttribute(TransactionMode.Manual)]
```

```
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {

  }

}
```

Any block of code in a class which performs a particular task (or action) is called a method. The method declaration starts with the word **public** in this case. You already know what public implies. For more details on understanding methods see Additional Topics.

This method returns a **Result** (in fact an Autodesk.Revit.UI.Result) rather than being declared void (i.e. not returning anything). The Result you returned from the Execute() method will tell Revit whether the command execution has succeeded, failed or been cancelled. If the command does not succeed, any changes it made will be reversed (Revit will cause the transaction that was used to make them to be rolled back).

The Execute() method has three parameters: **commandData, message** and **elements**. Let's take a closer look at what each of these parameters refer to:

1. **commandData** is of type **ExternalCommandData** and provides you with API access to the Revit application. The application object in turn provides you with access to the document that is active in the user interface and its corresponding database. All Revit data (including that of the model) is accessed via this commandData parameter.

2. **message** is a **string** parameter with the additional ref keyword, which means it can be modified within the method implementation. This parameter can be set in the external command when the command fails or is cancelled. When this message gets set – and the Execute() method returns a failure or cancellation result – an error dialog is displayed by Revit with this message text included.

3. **elements** is a parameter of type **ElementSet** which allows you to choose elements to be highlighted on screen should the external command fail or be cancelled.

Let's now look at the code inside in the Execute() method. This is the actual set of instructions which uses the Revit API to perform certain tasks when your command is executed. In Lesson 1 you saw how the Revit API can be used to add an external command which asks the user to select a group and a target location before going ahead and copying the group to that location.

Now let's look at the code, line-by-line:

```
    //Get application and document objects
    UIApplication uiApp = commandData.Application;
```

In the first line, you use the commandData parameter that was passed into the Execute() method to access the **Application** property of this object, which provides you with access to the Revit application. For more details on understanding properties and reviewing the main Revit API classes and the correlation between them, see the Additional Topics.

Variables

To be able to use the Application property you just retrieved from the **commandData** parameter, you created a container variable for the object named **uiApp** of type **UIApplication**. You then assigned the value of **commandData.Application** to it for later use in your program. When you're declaring a variable you create a named location for a value which can be accessed later on within the same block of code. Variables can be named according to your wishes, as long as the name is unique in that code-block and is not a reserved word (such as the "using" keyword mentioned earlier).

```
    Document doc = uiApp.ActiveUIDocument.Document;
```

The uiApp variable (which contains the Revit Application object) provides access to the active document in the Revit user interface via the ActiveUIDocument property. In the above line of code – in just one line – you directly accessed the database of the active document (this database is represented by the Document class). You stored this Document object in a variable named **doc.**

Object Selection

Let's look at how you prompted users to select Groups using the API.

```
   //Define a Reference object to accept the pick result.
    Reference pickedRef = null;
```

You started by creating an empty variable named **pickedRef** of type **Reference** and set its initial value to be **null** (which literally means nothing). By doing this you created an empty container in which to subsequently store a Reference object. Reference is a class which can contain elements from a Revit model associated with valid geometry.

```
    //Pick a group
    Selection sel = uiApp.ActiveUIDocument.Selection;
    pickedRef = sel.PickObject(ObjectType.Element, "Please select a group");
    Element elem = pickedRef.Element;
    Group group = elem as Group;
```

Next you accessed the current user selection using the API. The user selection from the user interface is represented by the **Selection** property on the **ActiveUIDocument** object: you placed this Selection object into a variable named **sel** of type Selection. This Selection object provides you with a method named **PickObject().** As the method's name suggests, it shifts focus to the user interface and prompts the user to select an object. The parameters of this method allow you to specify the type of element you want the user to select (you can specify if you are expecting users to select a face, an element, an edge, etc.) along with the message the user will see in the lower left corner of the Revit user interface while the plug-in waits for the selection to occur.

As the selected **Group** object has geometry data associated with it, it was safe to place it in the **pickedRef** variable you declared previously. You then used the reference's **Element** property to gain access to the reference's associated element: in this case you assigned its value to a variable named **elem**, of type Element. As you are expecting the elem object to be of type **Group**, in the last line of the above code snippet you performed a "cast", allowing us to treat the elem variable as a **Group** via the variable named group.

Note: If you are working with Revit 2012 API, the following line of code might throw a warning stating that 'Autodesk.Revit.DB.Reference.Element' is obsolete.

```
Element elem = pickedRef.Element;
```

To get rid of this warning in Revit 2012, please replace the line of code from above with the following –

```
Element elem = doc.GetElement(pickedRef);
```

Note: If you are working with Revit 2013 and higher API, the above mentioned change will need to be done to be able to build successfully and run the code.

Casting

In the manufacturing world, the term casting refers to the act of setting a given material into a mold to shape it into an object of a particular form. Similarly, in the programming world, casting means the act of trying to set a value of one type into another. Casting asks the language compiler to consider a value in a different way. In the last line of your code snippet, you are essentially casting the **Element** (which is actually a Group selected by the user) specifically into the **Group** type. The **as** operator in C# will cause the compiler to check the actual type of the object being cast: if it is incompatible with the target type, the value returned by the operator will be null.

Next you asked the user to pick a target point for the selected group to be copied to.

```
   //Pick a point
    XYZ point = sel.PickPoint("Please pick a point to place group");
```

For this, you once again made use of the variable named **sel**, of type **Selection** – the one you used previously to gain access to the PickObject() method – to call the **PickPoint()** method. The parameter for the **PickPoint()** method is simply the message to be shown in the bottom left corner of the screen while the plug-in waits for the user to select a point. The point selected was returned as an object of type **XYZ**, which you placed in a variable named **point**.

```
    //Place the group
    Transaction trans = new Transaction(doc);
    trans.Start("Lab");
    doc.Create.PlaceGroup(point, group.GroupType);
    trans.Commit();
```

We have already mentioned Transactions in the context of the Revit API. As you have set the transaction attribute to manual, Revit expects your plug-in to create and manage its own **Transaction** objects to make changes to the model. You needed to create your own transaction object to encapsulate the creation of your new group. You started by declaring a Transaction variable named **trans** to which you assigned a new instance of the Transaction class, created using the new keyword. The **constructor** (a special method used to create a new instance of a class) of the Transaction class expects the active document's database object to be passed in as a parameter, so that the Transaction knows with which document it will be associated. You then needed to start the transaction by calling its **Start()** method, allowing you to make changes to the Revit model. In this case, we named the transaction "Lab", however it can be any string value you choose.

The aim of this initial plug-in is to place a selected group at a location selected by the user. To perform this task, you used the **PlaceGroup()** method from the active document's database object under the creation-related methods made accessible via its **Create** property. This Create property makes it possible to add new instances of elements – such as Groups – to the Revit model. The PlaceGroup() method, as expected, required you to pass in the location at which you wanted to place your group, as well as the type (used in the context of Revit, rather than C#) of the group selected by the user.

Finally, you committed the transaction object using the **Commit()** method. This ensured the changes encapsulated by the transaction were successfully written to the Revit model.

```
    return Result.Succeeded;
```

As you may recall, the Execute() method – the entry point for your Revit plug-in – expects a **Result** to be returned. It is this Result which informs Revit whether the command completed successfully, whether it failed or was cancelled. At this point, assuming all went well with the code, you passed back a Result of **Succeeded.**

Well done – you have made it to the end of a lesson containing a great deal of information!

You covered a range of new topics in this lesson, including some basics of object oriented programming – defining what class is, what an object is – you looked at some fundamentals of the C# language and also took a first close look at the framework code needed by a Revit plug-in. You were then introduced the main classes of the Revit API and saw them being used in the code you created in Lesson 1. Finally, you looked at a number of different classes and methods from the Revit API which enabled you to work with user-selected objects and points and also helped you to create new instances of groups.

Additional Topics

Object Oriented Programming

In the previous lesson, you learned about how a program is a sequence of instructions which tells the computer how to perform one or more tasks. Simple programs tend to consist of a sequence or list of instructions operating on variables – data representing what's happening in the program as it executes. But, with increasing complexity, such linear sequences of instructions (an approach also often referred to as procedural programming) can be difficult to manage and ends up not being well-suited for the task for which the code was written.

To help address the complexity problem, Object-Oriented Programming (OOP) was developed to allow source code to be structured differently, enabling componentization and reuse. OOP encourages you to develop discrete, reusable units of programming logic which are aimed at modeling objects from the problem domain and the relationships between them. A good example of this is a map. A map of a city can be considered as a model or a simplified representation of the city itself, providing a useful abstraction of the information you need to get around it.

The key to good object-oriented design is to model the elements of the problem domain as individual objects displaying the relevant behavior of the originals and the relationships between them.

Classes

A class can be thought of as a type which can be used to describe a representation of a thing or an individual object. A class can be considered the template that describes the details of an object and is used to create individual objects. In this way classes can help define the common characteristics of objects of their type: the objects' attributes (properties) and behaviors (methods).

An object is an instance of a class. These are the building blocks of Object-Oriented Programming and can be thought of as variables - often quite complex ones – which encapsulate both data and behavior.

To provide an example, the relationship between a class and an object is similar to the concept of family and family instances in Revit. You could consider a wall system family as a class. It is in this file that the basic parameters of a wall are described. When you create an instance of this wall system family inside the model, it can be thought of as an object. The wall family is therefore the blueprint or template for all walls in the building model. Each wall instance has the set of parameters defined in the template but may have quite different parameter values: they might have different fire-ratings and locations, for instance.

How do you define a class in C#?

The following code snippet shows an example of a simple class declaration in C#. While describing this, you will also start looking more closely at the syntax of the C# language.

```
public class Point
{
  private int x, y;

  public void setLocation(int x, int y)
  {
    this.x = x;
    this.y = y;

    // Do some calculations next
    //
  }
}
```

The first word, **public**, sets the accessibility of the class. The use of public means that any programmer can gain access to this class in their own application. Other options would be **internal** (if only to be used within this project) or **private** (if only to be used in the context in which it is declared).

The next word, **class**, defines what follows as a class declaration. The next word, **Point**, defines the name of your class.

On the next line you have a "{". These are called braces and come in sets of two: for every opening brace ({) you must have a matching closing brace (}). The compiler will complain, otherwise, and often with a quite obscure error (as it attempts to interpret the following code in the wrong context). These braces help create blocks of code. In this case, the start and end braces signify the start and end of the code defining your class.

Variable Declarations

The next line defines two data members: these are member variables to be used in your class and help contain data which might be used in different portions of the code in the class. Data members allow you to share data between multiple blocks of code in the class. You define data members in one place and can then use them in different places.

```
  private int x, y;
```

The line starts with the information on the scope of access for these data members, which in this case is **private**. This means these variables can only be used and accessed within the class.

The word **int** refers to the "integer" type, which tells the compiler that you want to reserve some space to store some whole numbers. The most common types handled by C# are floating point (i.e. decimal) numbers, integer numbers and text (known as strings). x and y are names given to help identify the data members. C# code is made up of statements, each of which ends with a semi-colon (;).

Methods

The following block of code is called a method:

```
  public void setLocation(int x, int y)
  {
    this.x = x;
    this.y = y;



    // Do some calculations next
    //
  }
```

Any block of code in a class which performs a particular task (or action) is called a method. The method declaration starts with the word **public** in this case. You already know what public implies.

The next word is **void**. This is where you specify what the method is expected to return after performing the task(s) inside it. For example, if a method performs some calculations, it will be expected to return the result of those calculations. It's at this point that you specify the type of this result – in this case "void", which is the same as saying it doesn't return anything. This means that when you call the method you cannot assign its results to a variable, as there are no results being returned.

The word **setLocation** is the name you use to identify – and make use of – this method.

The portion of the statement between the brackets () contains the method's parameters. A parameter (or set of parameters) is the input provided to a method for it to use and work on. Parameters can also be used to return additional data to the caller of the method, but in this case you're keeping things simple and only specifying that data will be passed into it.

The next line has an opening brace which means this is the start of the block of code defining the implementation of the method. Inside the method, you can see the following line:

```
    this.x = x;
    this.y = y;
```

The word **this** identifies the current instance of your class. In this line, you are accessing the member variable x of this instance and so use the statement **this.x**. The "=" is called the assignment operator and means that the container on the left side of the operator will have its value changed to that specified on the right side. In this case, you are setting the value of the x member of the current instance of your Point class (using this) to the value passed into the method via the parameter x.

The next statement follows the same principle, but setting the y data member.

Lines starting with "//" are ignored by C#. This is one way of adding comments to your code –to describe its implementation in plain language – or if you have portions of the code which should be ignored during compilation.

Finally, the closing brace signifies the end of the block of code for this method's implementation.

If you look back at the code for the class, you'll see a subsequent brace. This denotes the end of the class (and has a different indentation to make it more readable).

Now you have a simple class defined using C#. The next question is typically…

How do you create objects of this class?

Creating Class Objects

As discussed before, an object is an instance of a class. The code included below shows how to create an object of a particular class. These lines would typically be contained in the implementation of another class (almost all code in a C# program is contained in classes – we'll look at how this code gets called later on).

```
Point pt = new Point();
pt.setLocation(10, 10);
```

In the first statement, you define a new variable named **pt** which is defined to be of type **Point** and use the assignment operator to set its initial value to be a new instance of the Point class, created using the **new** keyword. The second statement shows how to access the setLocation() method, passing in some parameter values (for x and y).

Properties

A core feature of C# and other .NET languages is the ability for classes to have properties. A property is simply a way of allowing access to an item of member data. It is good practice to keep member variables (or data variables) private to that class: properties allow you to expose this data in a controlled way (as you can then change the underlying class implementation without it impacting the way your class is used).

Extending the code snippet, let's see how a property can be added to a same class.

```
public class Point
{
  private int x, y;

  public void setLocation(int x, int y)
  {
    this.x = x;
    this.y = y;

    // Do some calculations next
    //
  }

  public int X
  {
    set
    {
      this.x = value;
    }
    get
    {
      return this.x;
    }
  }
}
```

The above code, in **bold**, shows how you can expose a property from your class, whether for use in your own project or consumption by other programmers in their projects.

The property declaration starts with the much-discussed **public** keyword. The next word specifies that the property is of integer type and is named **X**.

The **get** part of the property declaration describes what happens when someone reads the value of the property and the **set** part does the same for when someone writes to or sets the value of the property.

You have already seen the use of **this.x** to access a data member of the current class instance (or object), but the **value** keyword in the set part is something new: this provides you with the information being passed into the property, a bit like the parameters of a method. During set you simply take this value and assign it to your internal data member.

When someone attempts to access your property for a particular Point object, the get part of the property executes and the **return** statement passes back the value of your internal data member.

Let's now see how this property can be used:

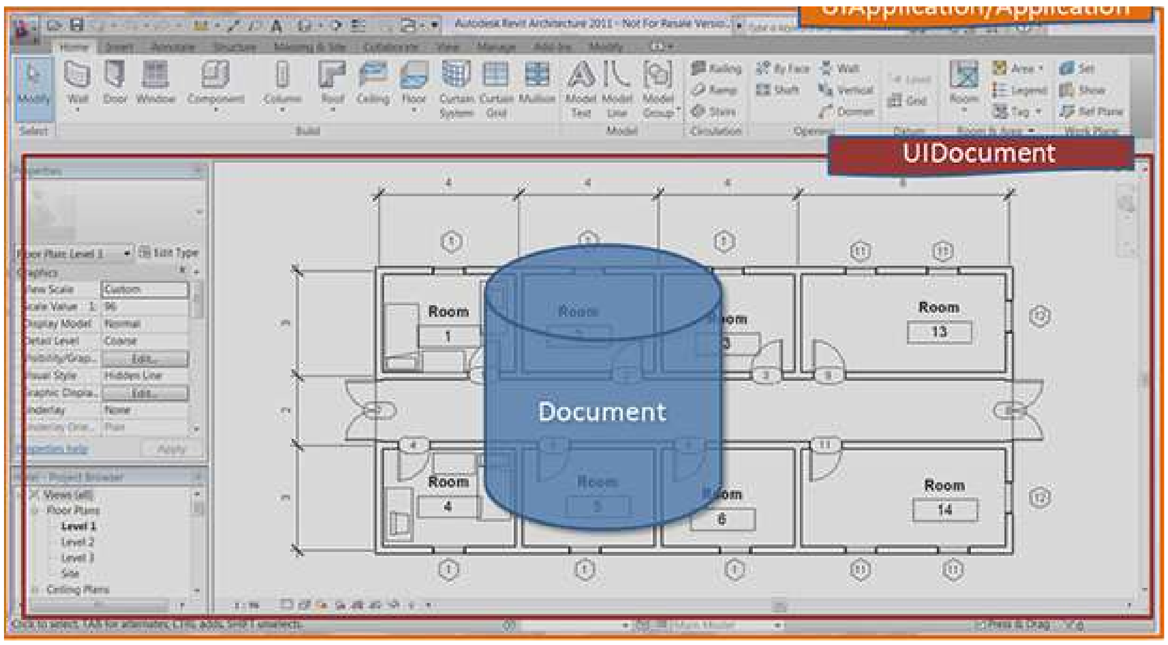
```
  Point pt = new Point();
  pt.setLocation(10, 10);
// getting the value of X property
int xCoord = pt.X;
// setting the value of X property
pt.X = 20;
```

You start by creating a new **Point** object, named **pt**, as you saw previously.

The lines in **bold** show you how to read (or get) the **X** property of your **pt** object, assigning its value to a variable **xCoord**, and then how to write (or set) it to a new value **20**.

Autodesk Revit Application and Document Class

Let's take a look at some of the main Revit API classes and the correlation between them. The following image shows the main Revit API classes shown in the context of the Revit product's user interface.



Here are these classes listed with their complete namespaces and a view of how they can be accessed as properties of other classes (i.e. how you would navigate from one to the other in your code).

Here's a brief description of what each of these classes represents:

Autodesk.Revit.UI.UIApplication:
Provides access to the active session of the Revit application and its user interface. Allows customization of ribbon panels, registration for events and access to the active document.

Autodesk.Revit.ApplicationServices.Application:
Provides access to the Revit application, to documents, options and other application-wide data and settings.

Autodesk.Revit.UI.UIDocument:
Provides access to the currently active project and the user interface for the document. Provides access to user selections and the ability to prompt users to pick points, select elements, etc.

Autodesk.Revit.DB.Document:
Provides access to all database-related information for the active project in Revit, such as the elements, views, etc.

Lesson 4: Debugging your code

New

Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

Learn how to use the Visual Studio debugger to step through your code line-by-line to follow the program execution; and to watch the values of variables as they are changed by your code.

It's rare that you will write a program that runs perfectly the first time. Even if you haven't made any mistakes that prevent Visual Studio from building your plug-in, it's quite likely that you made a mistake in your program logic that causes it to produce different results from the ones you expected. Errors in your code are called bugs, and the process of stepping through your code examining the program execution flow and variable values is called debugging.

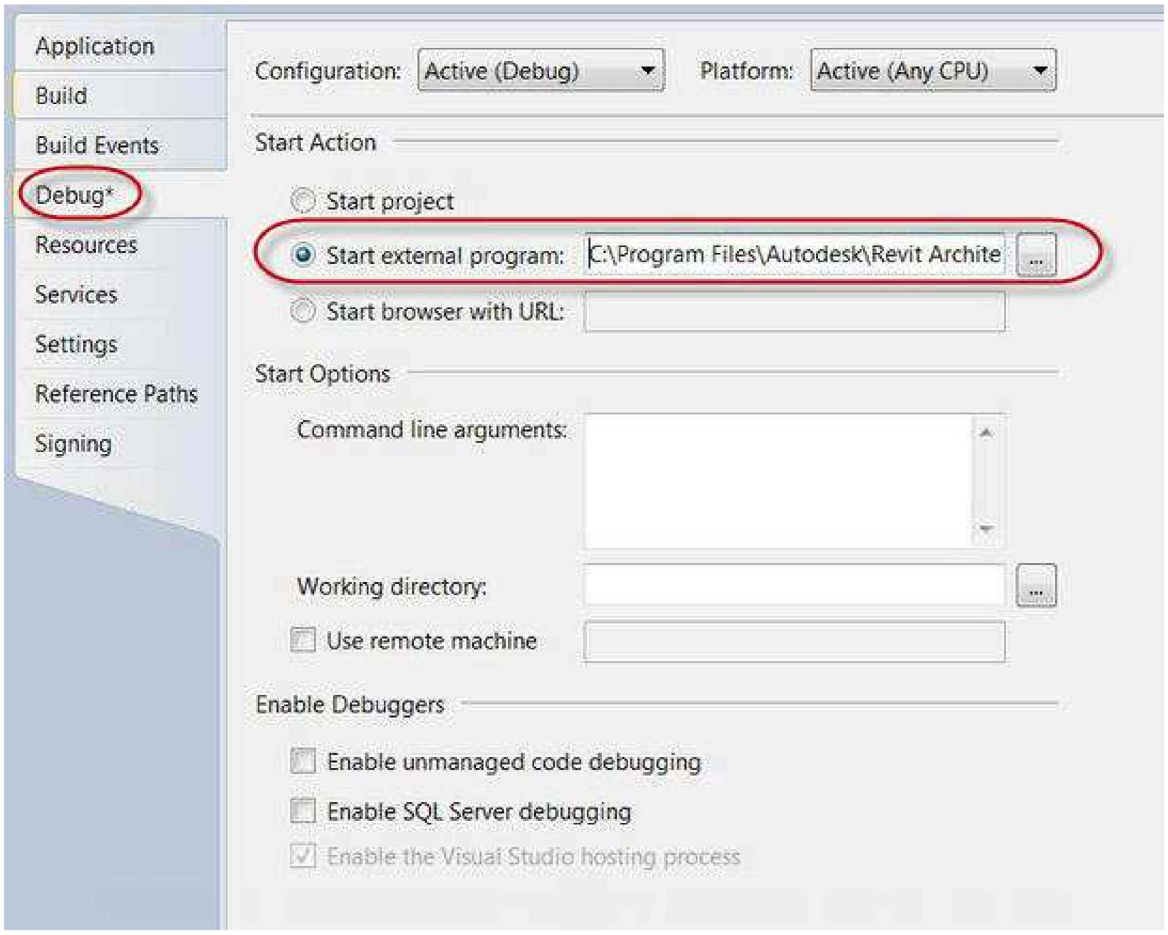
To debug your plug-in code, you're going to have to launch Revit from your Visual Studio Express debugger and load your plug-in into Revit. However, please do note that Visual C# Express and Visual VB.NET Express editions do not allow you to setup your project to debug a DLL, as you need to do. In order to be able to debug using Visual C# Express, we have to make some manual edits to the project files by implementing the following steps:

1. Locate the C# project file (*.csproj) in your project folder, which in this case, would be the Lab1PlaceGroup.csproj.
2. **Open the file:**
   Start **Notepad.exe** from the Windows **Start** menu. On **Notepad's File** menu, click **Open.** Navigate to the file and open it.
3. Look for the first <PropertyGroup> tag. (Note - you will see this tag in multiple times).
4. At the end of this tag, i.e. right before the close tag </PropertyGroup>, insert the two lines with the <StartAction> and <StartProgram> node (highlighted in bold). The *Lab1PlaceGroup.csproj* file should look like this:

```
  <PropertyGroup>
    . . .
    <StartAction>Program</StartAction>
    <StartProgram>C:\Program Files\Autodesk\Revit Architecture 
201x\Program\Revit.exe</StartProgram>
  </PropertyGroup>
```

(Edit the filepath for Revit.exe if you've installed it in a non-default location).

This is easy in the professional versions of Visual Studio – the Properties settings allow you to set the executable to launch when you start the debugger:



Configuration: Active (Debug)   Platform: Active (Any CPU)

Start Action

Start project

Start external program: C:\Program Files\Autodesk\Revit Archite

Start browser with URL:

Start Options

Command line arguments:

Working directory:

Use remote machine

Enable Debuggers

Enable unmanaged code debugging

Enable SQL Server debugging

Enable the Visual Studio hosting process

However, **this functionality isn't exposed through the Express IDE** and is why we perform the additional step of adding the settings to the .csproj file.

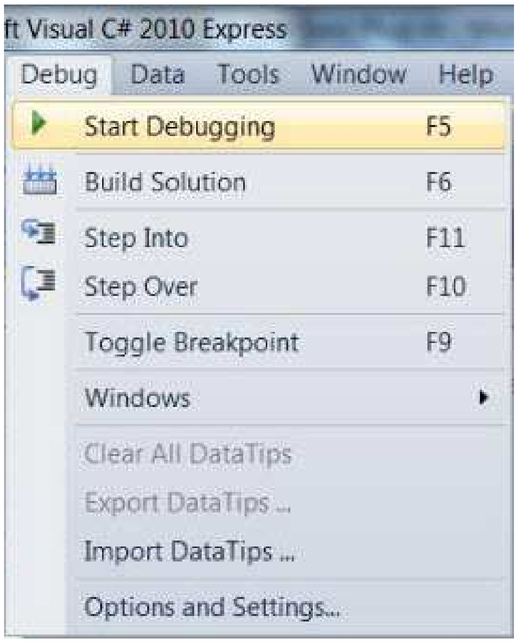
Launching the debugger

If you closed the Visual C# 2010 Express IDE, launch it again now and open the project you saved in Lesson 1 (called the Lab1PlaceGroup).

Before you launch the debugger, please ensure that the AddIn manifest file for this lab has been created and exists at the location described in the *Writing an AddIn Manifest* section in Lesson 1: The Basic Plug-in. Since you will now be debugging the plug-in code, Visual C# Express has created a 'debug' version of your .NET plug-in DLL. And so for Revit to read this 'debug' DLL instead of the 'release' DLL that was created in Lesson 1, we need to change the path of the DLL in the manifest to the following (highlighted in bold) -

```
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>
      C:\test\Lab1PlaceGroup\Lab1PlaceGroup\bin\Debug\Lab1.dll
    </Assembly>
    <ClientId>502fe383-2648-4e98-adf8-5e6047f9dc34</ClientId>
    <FullClassName>Lab1PlaceGroup</FullClassName>
    <Text>Lab1PlaceGroup</Text>
    <VendorId>ADSK</VendorId>
    <VisibilityMode>AlwaysVisible</VisibilityMode>
  </AddIn>
</RevitAddIns>
```

To start your debugging session, simply open the Debug menu and select the Start Debugging option, or just hit F5. This will launch Revit from your debugger. The flavor of Revit that gets launched will depend on the path you have provided in the tag in the .csproj file that we edited manually at the beginning of this lesson.



You have to launch Revit in this way so that your debugger can hook into the Revit process to monitor the execution of the code you compiled into your plug-in DLL. Because we've placed the AddIn manifest in the correct location, Revit will automatically load our plug-in.

If you have the Visual C# Express Immediate window open when you do this, you will see a lot of text scrolling up that window. Some of that text looks a little scary because it includes the word 'error'. Don't worry about that. This text is informational only – it doesn't mean there's a problem with Revit or with your plug-in.

Remember that the DLL you loaded in Lesson 1 was in the bin\Release *folder. When you are building a 'final' version of your plug-in DLL that you want to give to your users and customers, Visual Studio will build a **release** version. Visual Studio makes various optimizations to the compiled code in a release build so that it will run faster and take up less memory. When you want to debug your code, Visual Studio will create a **debug** version of your plug-in DLL. The debug version isn't optimized for speed/memory, and also includes additional information that the debugger uses to tell you about what is happening when the code runs.*

Note: If you are using Visual Studio 2010 Professional (and not the Express Edition), there is an alternative way of attaching the Visual Studio debugger to running instance of Revit. Please read more on this in this article: How to: Attach to a Running Process.

Now, open the Hotel.rvt Project file.

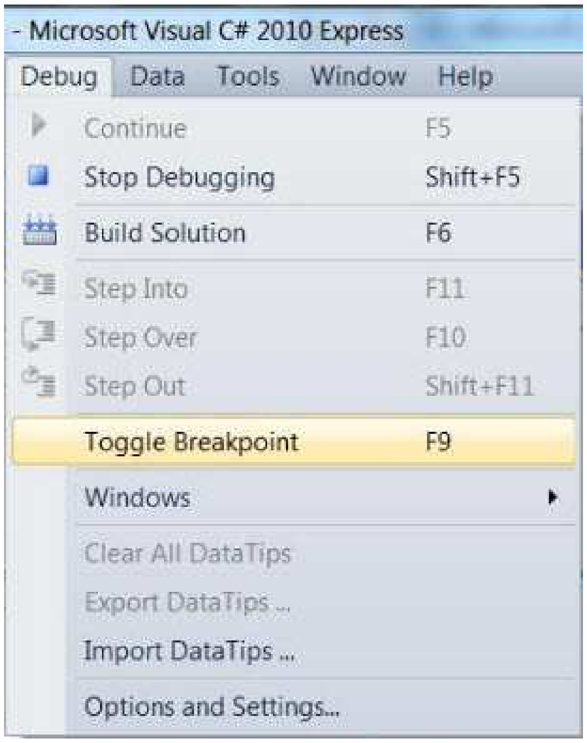
Breakpoints

Your plug-in DLL is now ready to debug. But before you run your **Lab1PlaceGroup** command, you have to tell the debugger you want it to stop when it is executing your code. You do this using a **breakpoint**.

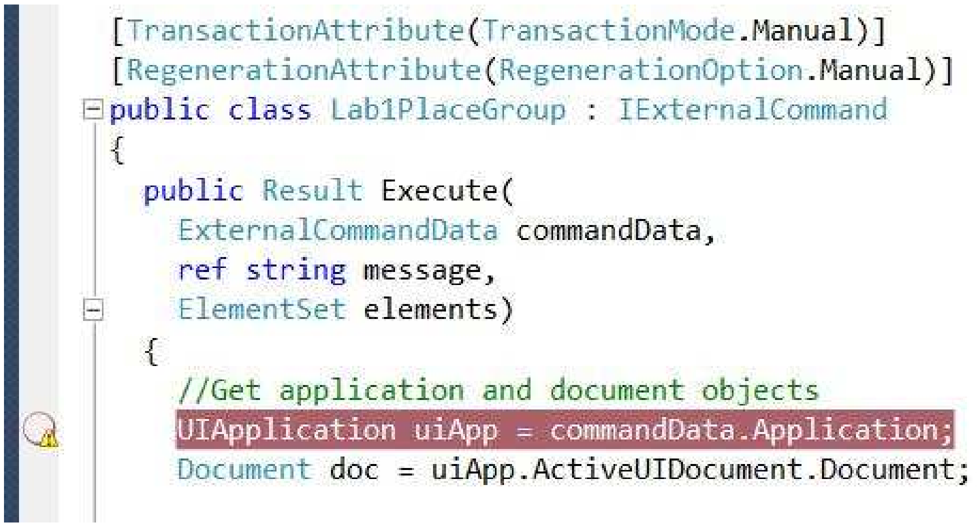
In Visual C# Express, double-click on Lab1.cs in the Solution Explorer to display your code and click anywhere in the line:

```
    UIApplication uiApp = commandData.Application;
```

Then select **Toggle Breakpoint** from the **Debug** menu (or hit **F9**).



That line will now be highlighted in red and have a red circle in the margin next to it to indicate that you've set a breakpoint for this line:



```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {
    //Get application and document objects
    UIApplication uiApp = commandData.Application;
    Document doc = uiApp.ActiveUIDocument.Document;
```

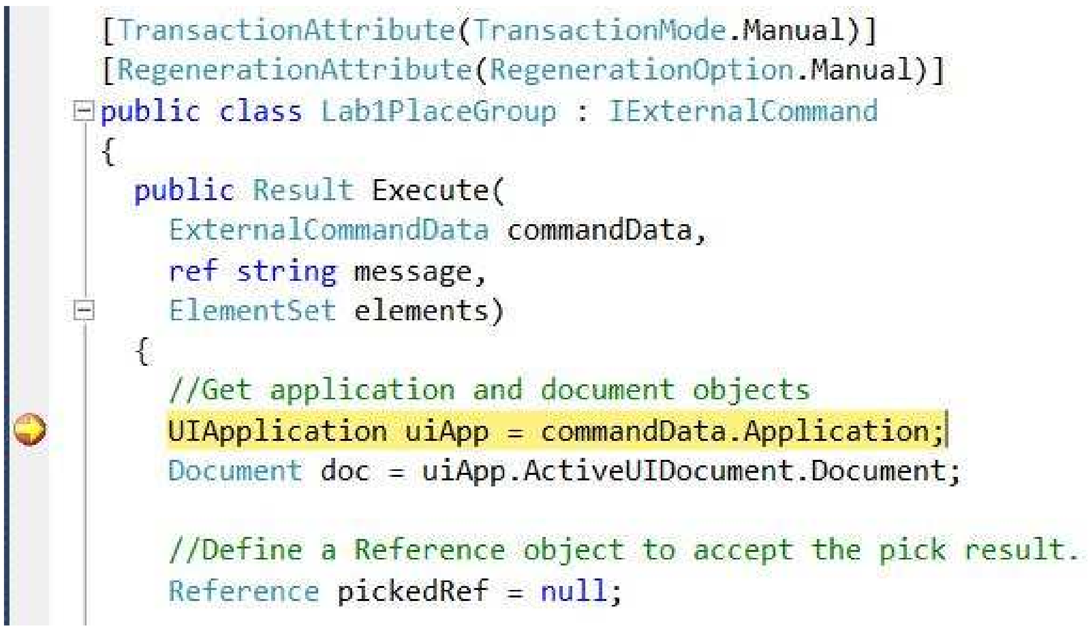
Set a breakpoint for the **PickPoint** function in the same way:

```
XYZ point = sel.PickPoint("Please pick a pont to place group");
```

When Revit calls these methods in your code, the debugger will stop at these lines and wait for you to tell it what to do.

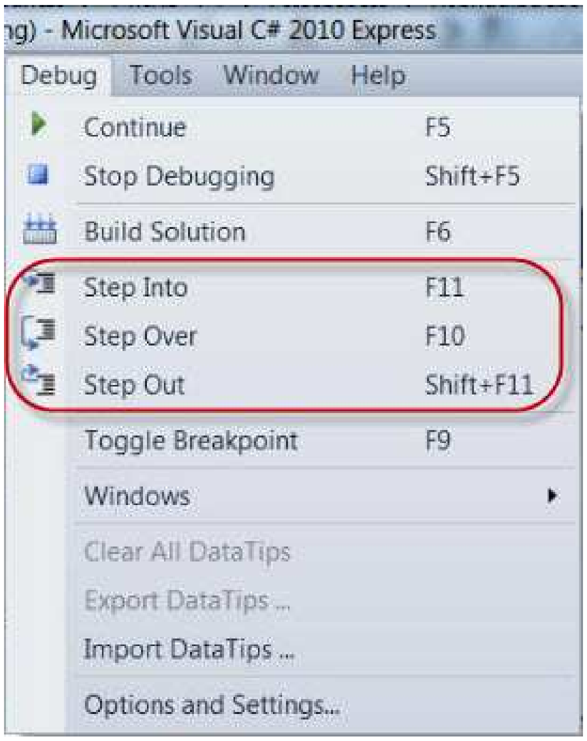
Stepping through your code

Now it's time to invoke your command. Inside Revit on the **Add-Ins** ribbon tab, click the **External Tools** drop-down list, then click **Lab1PlaceGroup**. This will start your plug-in in Revit and Visual C# Express should take control and become your foreground application. (If it doesn't, click on its icon in your Windows taskbar to activate it.) The debugger is now stopped waiting for you with the line of code it's about to execute highlighted in yellow and with a little yellow arrow in the margin next to it.

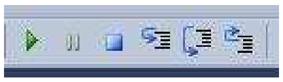


```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {
    //Get application and document objects
    UIApplication uiApp = commandData.Application;
    Document doc = uiApp.ActiveUIDocument.Document;

    //Define a Reference object to accept the pick result.
    Reference pickedRef = null;
```

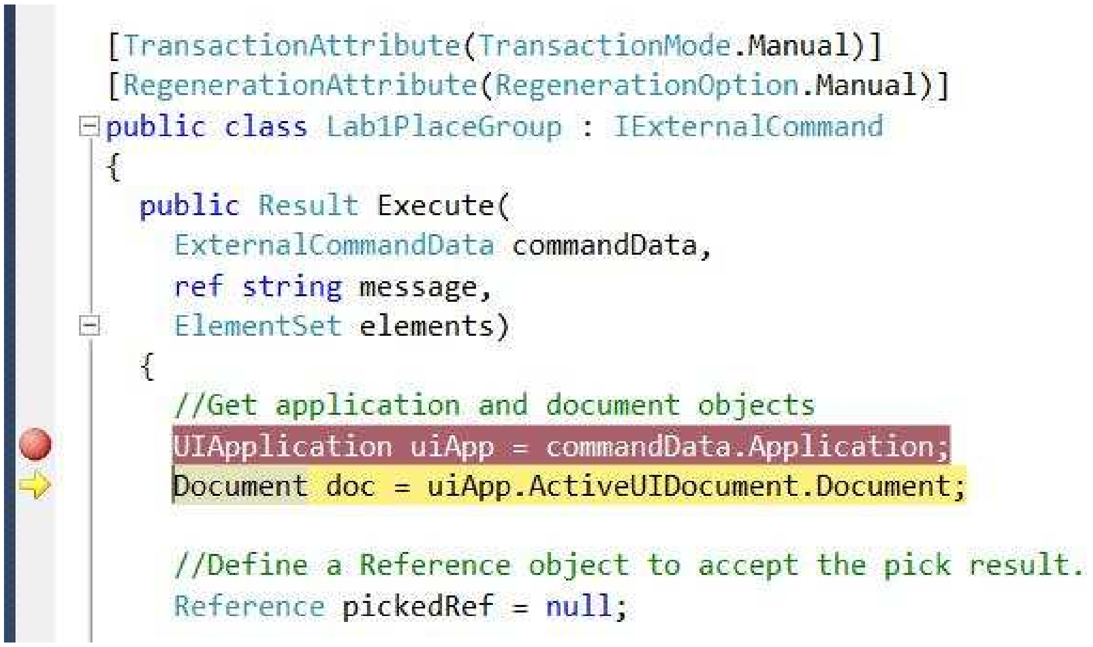
Now you're ready to step through your code. The Visual C# Express debug menu gives you three ways to step through your code: **Step Into**; **Step Over**; and **Step Out**. You'll mostly be using *Step Over* – this executes the next line of code (the line highlighted in yellow) in the debugger, and then moves to the next line. If the line of code to be executed is a method call, then Step Over executes the entire method. If you also want to execute the called method a line at a time, you can use *Step Into*; and you can use *Step Out* to move back up (out of the method) to the code you were originally debugging.



As well as in the *Debug* menu, you should also see *Step Into*, *Step Over* and *Step Out* icons on a toolbar, and each also has a corresponding hot key (**F8**, **Shift+F8**, and **Ctrl+Shift+F8**).

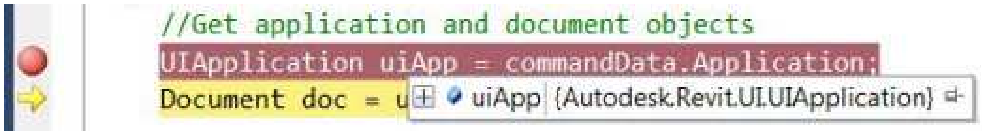


Click on the *Step Over* icon on the toolbar now. The debugger moves to the next line of code (it ignores comments).



```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {
    //Get application and document objects
    UIApplication uiApp = commandData.Application;
    Document doc = uiApp.ActiveUIDocument.Document;

    //Define a Reference object to accept the pick result.
    Reference pickedRef = null;
```

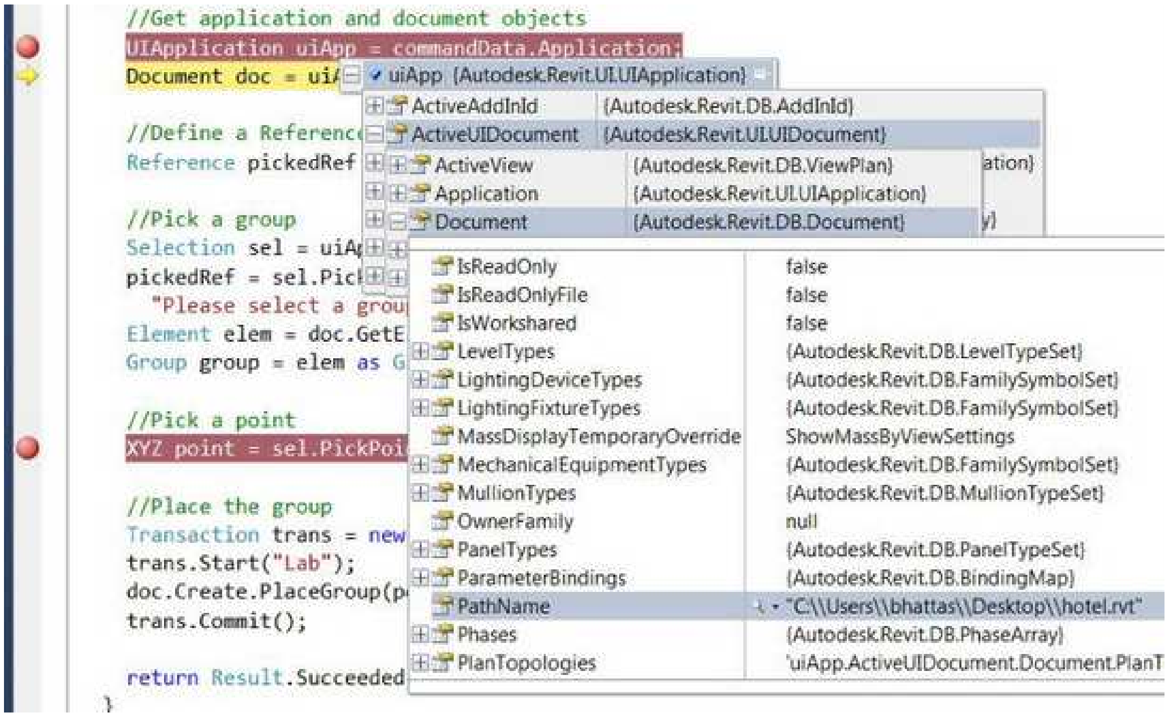
You can now hover the mouse over the text **uiApp** and a tooltip is displayed showing the value of the variable.



```
//Get application and document objects
UIApplication uiApp = commandData.Application;
Document doc = u
```

uiApp {Autodesk.Revit.UI.UIApplication}

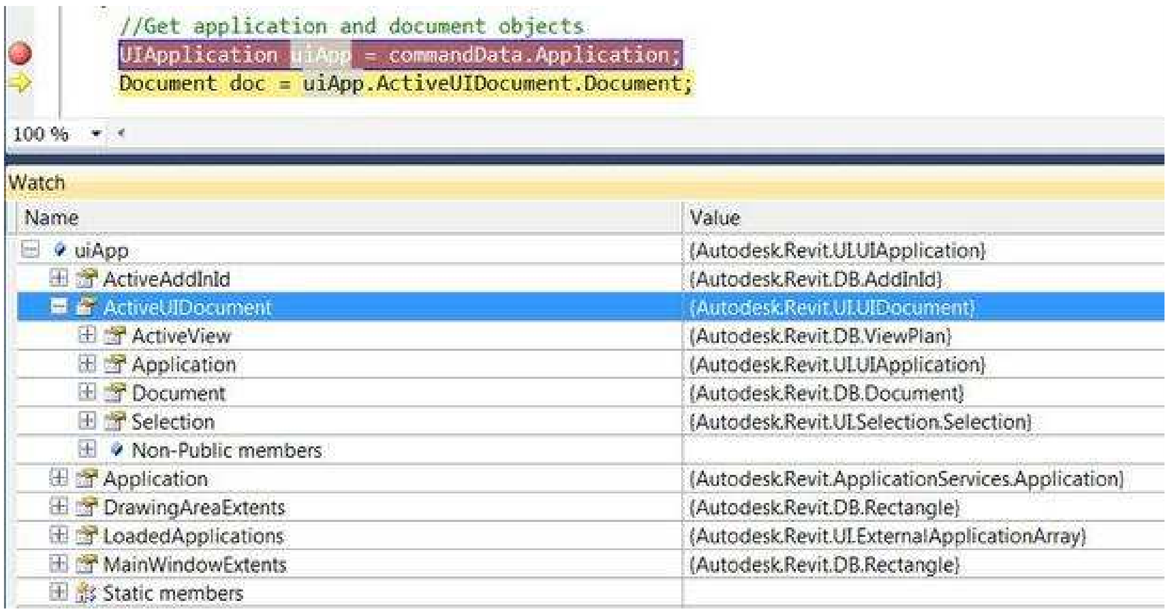
You can click on the **+** sign to drill down to the various properties and check their values.



```
//Get application and document objects
UIApplication uiApp = commandData.Application;
Document doc = ui

//Define a Referenc
Reference pickedRef

//Pick a group
Selection sel = uiA
pickedRef = sel.Pic
  "Please select a grou
Element elem = doc.GetE
Group group = elem as G

//Pick a point
XYZ point = sel.PickPoi

//Place the group
Transaction trans = new
trans.Start("Lab");
doc.Create.PlaceGroup(p
trans.Commit();

return Result.Succeeded
}
```

```
uiApp {Autodesk.Revit.UI.UIApplication}
  ActiveAddInId             {Autodesk.Revit.DB.AddInId}
  ActiveUIDocument          {Autodesk.Revit.UI.UIDocument}
    ActiveView              {Autodesk.Revit.DB.ViewPlan}
    Application             {Autodesk.Revit.UI.UIApplication}
    Document                {Autodesk.Revit.DB.Document}
      IsReadOnly                      false
      IsReadOnlyFile                  false
      IsWorkshared                    false
      LevelTypes                      {Autodesk.Revit.DB.LevelTypeSet}
      LightingDeviceTypes             {Autodesk.Revit.DB.FamilySymbolSet}
      LightingFixtureTypes            {Autodesk.Revit.DB.FamilySymbolSet}
      MassDisplayTemporaryOverride    ShowMassByViewSettings
      MechanicalEquipmentTypes        {Autodesk.Revit.DB.FamilySymbolSet}
      MullionTypes                    {Autodesk.Revit.DB.MullionTypeSet}
      OwnerFamily                     null
      PanelTypes                      {Autodesk.Revit.DB.PanelTypeSet}
      ParameterBindings               {Autodesk.Revit.DB.BindingMap}
      PathName                        "C:\\Users\\bhattas\\Desktop\\hotel.rvt"
      Phases                          {Autodesk.Revit.DB.PhaseArray}
      PlanTopologies                  'uiApp.ActiveUIDocument.Document.PlanT
```

To display variable values, you can also right click on the variable in your code and select **Add Watch**. This displays the variable and its value in the Watch window at the bottom of your IDE.

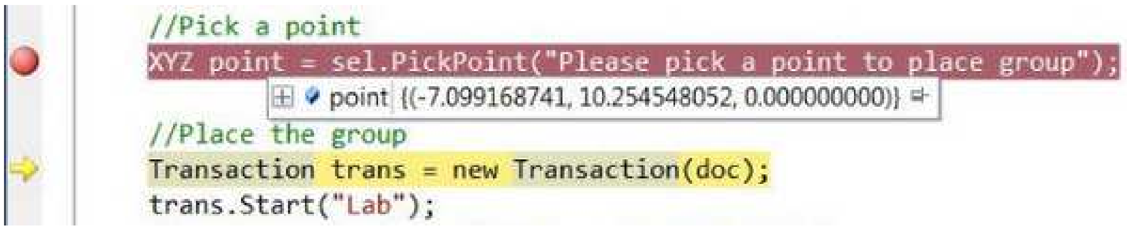


```
//Get application and document objects
UIApplication uiApp = commandData.Application;
Document doc = uiApp.ActiveUIDocument.Document;
```

Name	Value
uiApp	{Autodesk.Revit.UI.UIApplication}
ActiveAddInId	{Autodesk.Revit.DB.AddInId}
ActiveUIDocument	{Autodesk.Revit.UI.UIDocument}
ActiveView	{Autodesk.Revit.DB.ViewPlan}
Application	{Autodesk.Revit.UI.UIApplication}
Document	{Autodesk.Revit.DB.Document}
Selection	{Autodesk.Revit.UI.Selection.Selection}
Non-Public members	
Application	{Autodesk.Revit.ApplicationServices.Application}
DrawingAreaExtents	{Autodesk.Revit.DB.Rectangle}
LoadedApplications	{Autodesk.Revit.UI.ExternalApplicationArray}
MainWindowExtents	{Autodesk.Revit.DB.Rectangle}
Static members	

Keep stepping through the code, hovering over variable values and properties to see how they change as the code executes. If at any point you want to stop line-by-line debugging, just hit *F5* to **Continue** to the next breakpoint you had set – which in our case was set to the **sel.PickPoint()** line.

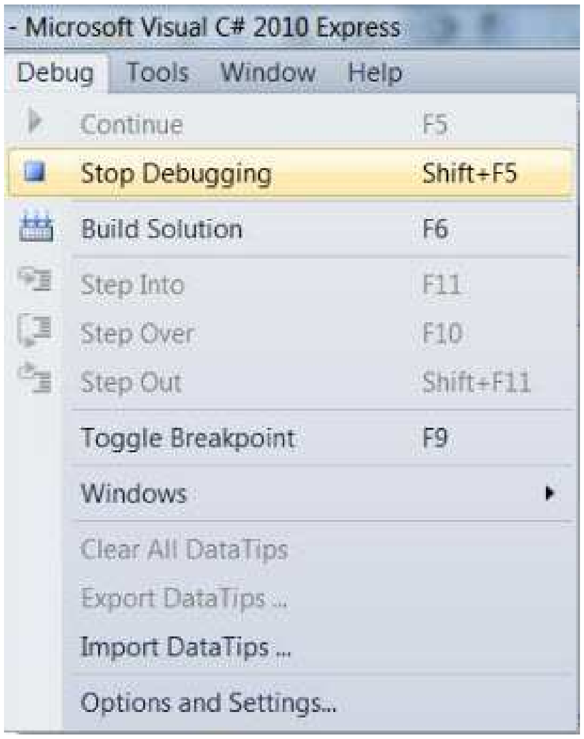
Stepping through this line by clicking *F10*, makes Revit the foreground application and prompts you to select the point where you want the Furniture group to be copied to. Let us select the approximate center of the adjacent room now. As soon as the point is selected, Visual C# Express now becomes the foreground application with the next line of code highlighted in yellow with the debugger waiting for us to either check the value of the selected point or continue line by line debugging or just jump to the next breakpoint by clicking *F5.* At this point, we can check the XYZ value of the selected point by just hovering the mouse over the point variable (which is the container/variable we created to store the target point for the furniture group to be copied to).



```
//Pick a point
XYZ point = sel.PickPoint("Please pick a point to place group");
    point {(-7.099168741, 10.254548052, 0.000000000)}
//Place the group
Transaction trans = new Transaction(doc);
trans.Start("Lab");
```

If you hit *F5* now, you will find that the debugger steps through all the remaining code, and makes Revit the foreground application again with the copy operation successfully completed.

Once you've finished experimenting, select **Stop Debugging** from the Debug menu to close Revit and end your debugging session.



Congratulations! You now know how to debug a project in Visual C# Express.

Lesson 5: Simple Selection of a Group

Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

Lesson Downloads

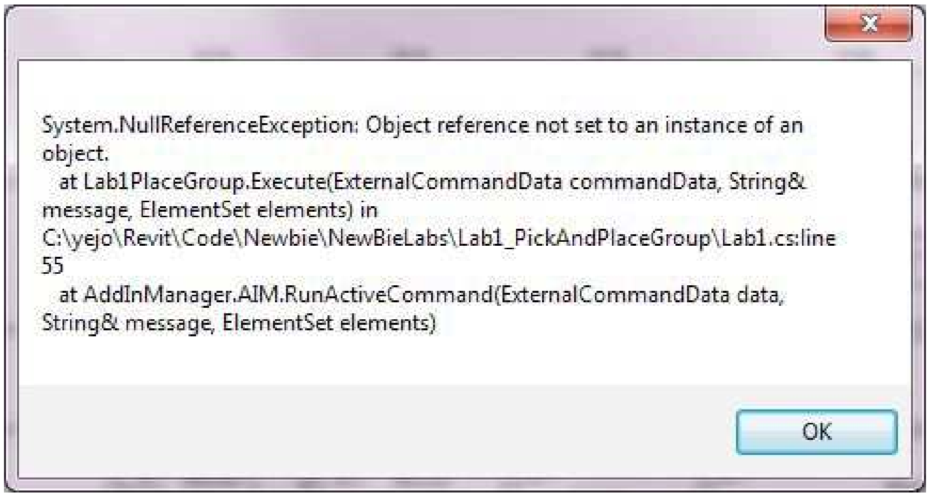
- lesson5_revit_2014_projects.zip (zip - 5660Kb)
- lesson5_revit_2013_projects.zip (zip - 32720Kb)
- lesson5_revit_2012_and_earlier_project_files.zip (zip - 7143Kb)

Throughout the next few lessons, you will be extending the original code to include additional features and to improve robustness. In this lesson, you will improve your selection functionality, by making it easier for the user to select a group and make sure your plug-in anticipates the possibility of something unexpected happening.

Planning out the New Functionality

When you launch the external command defined by the plug-in you created in Lesson 1, the status bar in Revit shows the prompt string, Please select a group. With the current implementation there are two scenarios that could prove problematic for your plug-in: first, the user may select a non-group element. Second, the user may click or press a wrong button or key during the input process. Neither of these possibilities have currently been anticipated by your original code. You will now add code to your plug-in to check for these error conditions and deal with them gracefully. First, let's see what happens in these two situations and how best to deal with them:

1. **Selecting a non-group element**:
While selecting a group, the user's cursor hovers over Room 1. In this case, the room, its bounding walls and the group itself could potentially each be highlighted and selected. However, your command expects the selected item to be a group. If the room or one of its bounding walls is selected, an error dialog, shown below, will be displayed before your command fails.

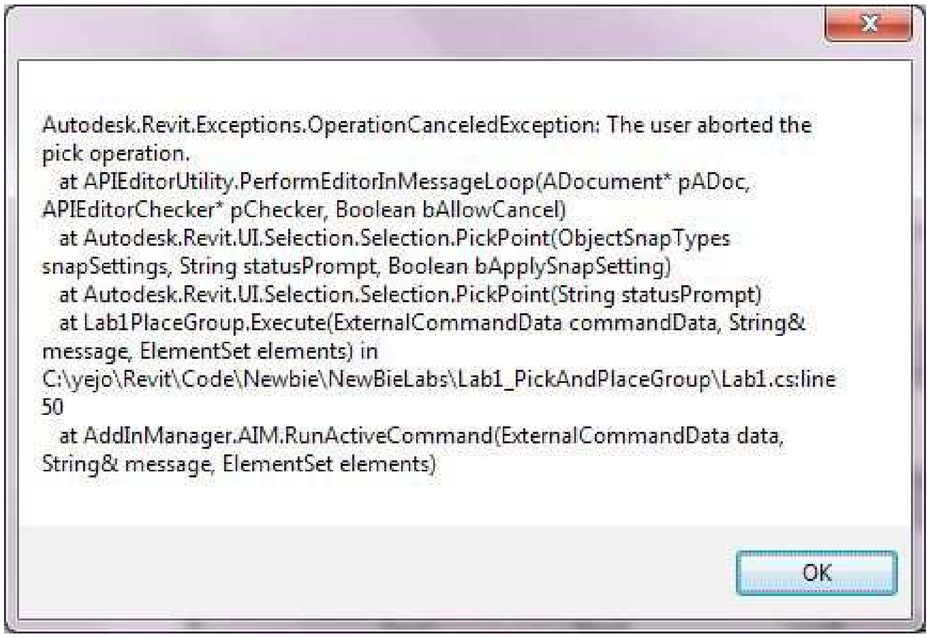


```
System.NullReferenceException: Object reference not set to an instance of an
object.
   at Lab1PlaceGroup.Execute(ExternalCommandData commandData, String&
message, ElementSet elements) in
C:\yejo\Revit\Code\Newbie\NewBieLabs\Lab1_PickAndPlaceGroup\Lab1.cs:line
55
   at AddInManager.AIM.RunActiveCommand(ExternalCommandData data,
String& message, ElementSet elements)
```

To avoid incorrectly selecting an element other than a group while using your command, the user would have to move the cursor very carefully until the group is highlighted and then click to select it. The command should manage this situation more cleanly: while the message presented to the user gives information about the problem in the code, this is not at all helpful for the user of the plug-in.

Plug-in Solution
To reduce the chances of the user incorrectly selecting a non-group element, you will add a selection filter. A selection filter limits the types of object being selected during the user interaction. When you ask the user to select a group, only groups will be highlighted when hovered over in the Revit user interface. This will help guide users to select a group, even if they haven't seen the prompt displayed on the status bar.

2. **Unexpected mouse clicks or button presses**:
While selecting a group or picking a point, it's possible for users to click any mouse button or hit any key on the keyboard. This is currently a problem for your plug-in, as both **PickObject()** and **PickPoint()** only expect the user to left-click the mouse during selection. If the user right-clicks the mouse, or hits the "Esc" key to cancel this command, this following confusing error message will be displayed:



```
Autodesk.Revit.Exceptions.OperationCanceledException: The user aborted the
pick operation.
   at APIEditorUtility.PerformEditorInMessageLoop(ADocument* pADoc,
APIEditorChecker* pChecker, Boolean bAllowCancel)
   at Autodesk.Revit.UI.Selection.Selection.PickPoint(ObjectSnapTypes
snapSettings, String statusPrompt, Boolean bApplySnapSetting)
   at Autodesk.Revit.UI.Selection.Selection.PickPoint(String statusPrompt)
   at Lab1PlaceGroup.Execute(ExternalCommandData commandData, String&
message, ElementSet elements) in
C:\yejo\Revit\Code\Newbie\NewBieLabs\Lab1_PickAndPlaceGroup\Lab1.cs:line
50
   at AddInManager.AIM.RunActiveCommand(ExternalCommandData data,
String& message, ElementSet elements)
```

Plug-in Solution
To prevent this error message from being displayed, you will add code to catch the exception and handle it. If the exception is caused by the user right-clicking or hitting the “Esc” key, you will simply terminate the current command. For other exceptions, you will present the error message to the user via the standard Revit error dialog.

Although the above should be enough to prevent these particular errors from being generated, you will also add an exception handling mechanism to deal with other possible errors, should they occur.

Coding the New Functionality

For clarity and better organization of the completed source code that we provide as an attachment for each lesson, we have changed the class names to match the lesson and the functionality we are working with.

In particular, in this lesson, the downloaded attachment’s class name is changed from “Lab1PlaceGroup” to “Lab5PickFilter”, as shown below:

public class Lab5PickFilter : IExternalCommand

{

Keep in mind that this change is not required, and that the detailed instructions we provide below assume you are always working from your original Lab1PlaceGroup code. That is why this name remains listed below and does not match the source code provided - whichever name you choose for your class name is fine.

You’ll start by adding your selection filter to highlight groups during the PickObject() call. You’ll then add an exception handler to avoid the obscure error message, should the user still select something other than a group.

1. In Visual C# Express, re-open the project from Lesson 1.

2. **Define a selection filter for groups only**:
Type the following code fragment after the class **Lab1PlaceGroup** definition code (after the closed curly bracket for the Lab1PlaceGroup), feel free to check the complete code to ensure your location is correct.

```
/// Filter to constrain picking to model groups. Only model groups
/// are highlighted and can be selected when cursor is hovering.
public class GroupPickFilter : ISelectionFilter
{
  public bool AllowElement(Element e)
  {
    return (e.Category.Id.IntegerValue.Equals(
      (int)BuiltInCategory.OST_IOSModelGroups));
  }
  public bool AllowReference(Reference r, XYZ p)
```

```
      {
        return false;
      }
    }
```

In the code window, replace this line:

```
pickedRef = sel.PickObject(ObjectType.Element, "Please select a group");
```

With the following two lines:

```
GroupPickFilter selFilter = new GroupPickFilter();
pickedRef = sel.PickObject(ObjectType.Element, selFilter,
  "Please select a group");
```

We'll come back to the explanation of the code later.

3. **Create error handling**:
Enter the following **bold** code in the **Execute()** method of the class **Lab1PlaceGroup** after the line that gets the document object and before the line asking the user to pick a group.

```
Document doc = uiApp.ActiveUIDocument.Document;

try
{
 //…Move most of code in Execute method to here.
}

//If the user right-clicks or presses Esc, handle the exception
catch (Autodesk.Revit.Exceptions.OperationCanceledException)
{
  return Result.Cancelled;
}
//Catch other errors
catch (Exception ex)
{
  message = ex.Message;
  return Result.Failed;
}
```

Then move the following block of code to be enclosed by the **try**'s two braces.

```
//Define a Reference object to accept the pick result.
Reference pickedRef = null;

//Pick a group using a filter.
//So only group can be selected.
Selection sel = uiApp.ActiveUIDocument.Selection;
GroupPickFilter selFilter = new GroupPickFilter();
pickedRef = sel.PickObject(ObjectType.Element, selFilter,
  "Please select a group");
Element elem = pickedRef.Element;
Group group = elem as Group;

//pick a point
XYZ point = sel.PickPoint("Please pick a point");

//place the group
Transaction trans = new Transaction(doc);
trans.Start("Lab");
doc.Create.PlaceGroup(point, group.GroupType);
trans.Commit();
```

All exceptions that might occur during execution of the above code will be caught and handled gracefully by this exception handling mechanism.

After the above two steps, you have completed your code for this lesson. The complete code for this lesson is also provided for download at the top of this lesson. It can be useful to see the complete code to compare your results and ensure they are correct.

22. **Save the file**:
On the **File** menu, click **Save All**.

23. **Build the project**:
Note: If Revit Architecture is already running, please close it.

Inside Visual C# Express, in the **Debug** menu, click **Build Solution** to compile and build your plug-in. If the code builds successfully, you will see the **Build succeeded** message in the status bar of Visual C# Express.

Running the Plug-in

1. Start Autodesk Revit Architecture.

2. Open the Hotel.rvt Project file.

3. Start the command as you did in Lesson 1. (In the Revit **Add-Ins** ribbon tab, click the **External Tools** drop-down list, and then **Lab1PlaceGroup** to start executing your plug-in's command.)

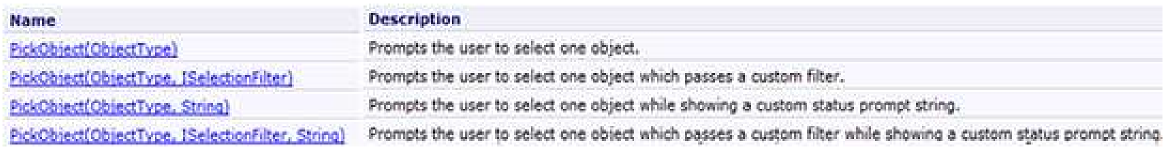
4. Move the cursor over **Room1**. Irrespective of the geometry being hovered over, only the furniture group should be highlighted. Click to select the furniture group.

5. You are now asked to select a point – and it is expected you will do so by left-clicking. To test your exception handler, right-click instead. The command should end without an error message. Similarly, instead of right-clicking, if you hit the "Esc" key, the command should also end cleanly.

A Closer Look at the Code

Object Selection

You called the **PickObject()** method to prompt the user to select a group. This method has four different forms as shown in the image below.



Name	Description
PickObject(ObjectType)	Prompts the user to select one object.
PickObject(ObjectType, ISelectionFilter)	Prompts the user to select one object which passes a custom filter.
PickObject(ObjectType, String)	Prompts the user to select one object while showing a custom status prompt string.
PickObject(ObjectType, ISelectionFilter, String)	Prompts the user to select one object which passes a custom filter while showing a custom status prompt string.

Each form has specific functionality and has a specific parameter signature. The **ObjectType** argument specifies the type of element that you would like selected – this is common across all four forms.

In Object-Oriented Programming, a method with the same name can take different numbers and types of parameters. This concept is known as overloading. In your original code, you called PickObject() this way:

```
    pickedRef = sel.PickObject(ObjectType.Element, "Please select a group");
```

This corresponds to the third overload in the above list:

```
public Reference PickObject(ObjectType objectType, string statusPrompt);
```

This overload takes two arguments: an object type and a prompt string. As the argument names indicate, the prompt string will be displayed in the status bar. The user will be allowed to select an element of the specified **ObjectType**.

In the code you wrote in this lesson, a selection filter is needed to constrain the range of types that can be selected. You did this by using the fourth overload in the list:

```
public Reference PickObject(ObjectType objectType, ISelectionFilter pSelFilter,
  string statusPrompt);
```
This overload expects you to pass in three arguments: an object type, a selection filter, and a prompt string. The second argument should be an instance of the class that implements the **ISelectionFilter** interface. We introduced the concept of an interface in Lesson 3, when talking about the IExternalCommand interface. In this lesson you created the GroupPickFilter class which implements the ISelectionFilter interface to tell Revit which elements and references can be selected.

The ISelectionFilter interface has two methods to be implemented: **AllowElement()** and **AllowReference()**: you needed to implement both of these in your GroupPickFilter class. The function parameter signatures of AllowElement() and AllowReference() are specified in the Revit API. The signatures of these two methods are shown in the following code.

Here is the skeleton for the new GroupPickFilter class:

```
public class GroupPickFilter : ISelectionFilter
{
  public bool AllowElement(Element e)
  {
    //insert your code here
  }
  public bool AllowReference(Reference r, XYZ p)
  {
    //insert your code here
  }
}
```
During the selection process, when the cursor is hovering over an element, this element will be passed into the AllowElement() method. The AllowElement() method allows you to examine the element and determine whether it should be highlighted or not. If you return true from this method, the element can be highlighted and selected. If you return false, the element can be neither highlighted nor selected.

It’s in the implementation of AllowElement() that you decided whether or not the element should be selectable by checking its category. This is the line of code that checks if the given element’s category type is of “model group”:

```
    return (e.Category.Id.IntegerValue.Equals(
       (int)BuiltInCategory.OST_IOSModelGroups));
```
e.Category.Id.IntegerValue – gets the integer value of the category ID from the element e passed in as a parameter to AllowElement().

BuiltInCategory.OST_IOSModelGroups – refers to a number identifying the built-in “model group” category, which we retrieve from the BuiltInCategory collection of fixed integer values.

The BuiltInCategory collection is called an enumeration and is declared using the enum keyword in the C# language. An enumeration is a way of associating a human-readable string of characters with an underlying integer: you can think of an enum as a list of integers, each of which has an associated string that can be used instead of the number itself. OST_IOSModelGroups is a particular value in this BuiltInCategory enumeration. In this case it actually refers to the integer value -2000095, although you never really need to know this and it would be bad programming practice to use this number directly in your code.

Since members of an enum are actually integers, you performed a cast to convert BuildingCategory.OST_IOSModelGroups into an integer to be able to compare it with the category ID value. If the category ID of the element passed into AllowElement() equals the model group’s category ID, then this element is a group and so should be selectable.

As AllowElement() returns a Boolean value (which is either “true” or “false”), it was possible to simplify the code into a single line in your method implementation:

```
  public bool AllowElement(Element e)
  {
       return (e.Category.Id.IntegerValue.Equals(
     (int)BuiltInCategory.OST_IOSModelGroups));
  }
```

The second method that you needed to implement is AllowReference(). During the selection process, if the cursor is hovering over a reference, this reference will be passed into the AllowReference() method. Once again, if you return true from this method, then the reference can be highlighted and selected. If you return false, the reference can be neither highlighted nor selected. The AllowReference() method also needs to be implemented for the ISelectionFilter to be complete, but as you don't care about references in your plug-in, you simply return false from it:

```
public bool AllowReference(Reference r, XYZ p)
{
  return false;
}
```

Error Handling

When a program needs to interact with its user but doesn't anticipate something the user does, it could lead to the program crashing or terminating unexpectedly. .NET provides an exception handling mechanism that can "catch" these – and other – errors in execution. You started by placing your code within the braces of a try block. This tells C# that you wanted the CLR (Command Language Runtime) to watch for exceptions (which may be errors or other atypical events) and report them to the subsequent **catch** block. You can think of the try statement as something that checks for exceptions, and, as soon as it finds one, stops executing the current block of code and throws it outside of itself to be caught below. The catch block for the type of exception that is thrown will handle it: it's possible to have multiple catch blocks if different behaviors are needed for certain exceptions.

In this lesson, you added two catch blocks. The first is related to the case where the user might hit "Esc" or right-click their mouse during selection of a group or point: this causes an **OperationCanceledException** exception to be thrown by the PickObject() or PickPoint() method. Using the following catch block, you simply returned **Result.Cancelled** from your command implementation, as you believe the logical behavior for the user to see at this point is the command to be cancelled:

```
  //If user right click or press ESC button, handle the exception
  catch (Autodesk.Revit.Exceptions.OperationCanceledException)
  {
    return Result.Cancelled;
  }
```

There are other exceptions that might occur during execution of your code, such as when you added a new group to the model via the **PlaceGroup()** method (although the possibility of this failing is very low). Such errors tend to happen for reasons outside of your control, such as the computer being low on memory. To make sure all such possibilities are considered, a second, more general catch block will handle any .NET exception. The argument of this catch block is **System.Exception** (which is shortened to Exception as you are **using** the System namespace in your code), which is the base class for all exceptions in .NET (you could cast any .NET exception to this type, should you wish). When this more general catch block executes, you want the command to fail: you place the message contained in your exception (which you access via its Message property) in the message parameter to the Execute() method and then return **Result.Failed**. This tells Revit the command has failed and causes it to display the exception information in its standard error dialog.

```
  //Catch other error.
  catch (Exception ex)
  {
    message = ex.Message;
    return Result.Failed;
  }
```

Well done! You have just written a more robust plug-in by improving the selection functionality and anticipating the possibility of something unexpected happening with the use of your plug-in, which will, in turn, improve the user experience. While implementing these new functionalities you discovered more programming concepts: overloading, enumerations and using a try-catch block for error handling.

Lesson 6: Working with Room Geometry

Over the next two lessons you will modify your command to copy the group to the same relative position in other rooms selected by the user. In this lesson you'll focus on the first part of this, to copy your selected group to a location relative to the center of the room it is in. You'll then extend that in the next lesson to allow selection of multiple rooms, copying the group to the same relative position in each room.

Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

Lesson Downloads

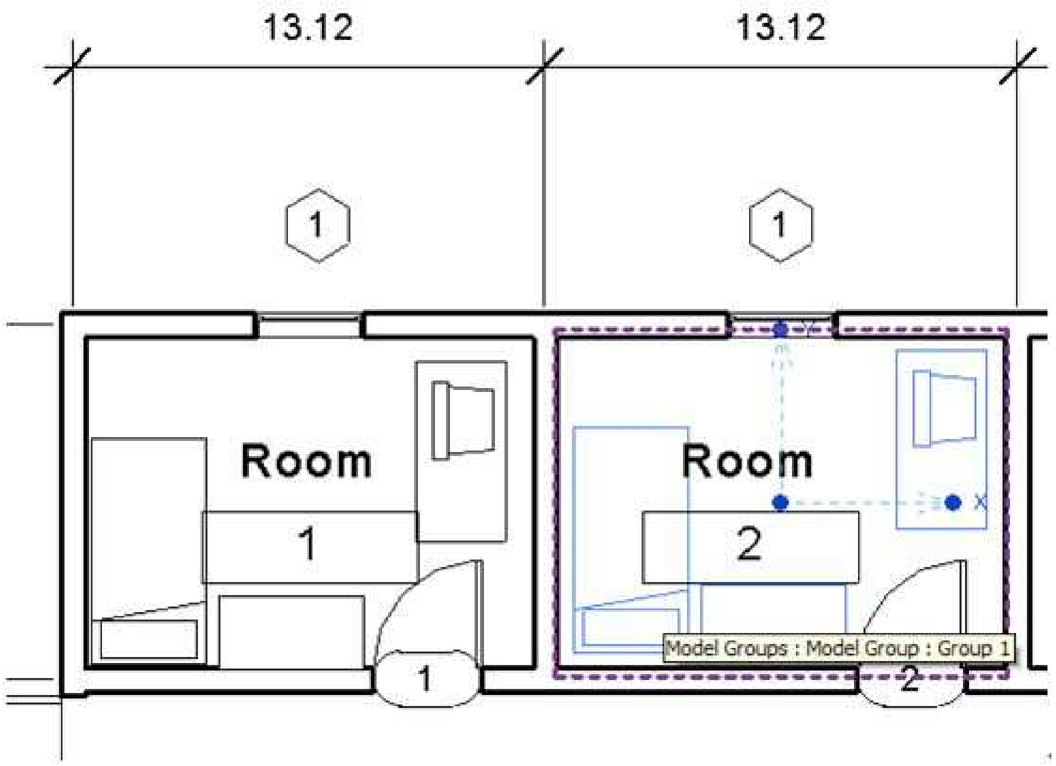
lesson6_revit_2014_projects.zip (zip - 5668Kb)

lesson6_revit_2013_projects.zip (zip - 21635Kb)

lesson6_revit_2012_and_earlier_project_files.zip (zip - 7148Kb)

Planning out the New Functionality

When the **PlaceGroup()** method inserts a group, it places the center of the group at the target location. For the purposes of this lesson, you're going to specify this location as being a fixed displacement from the center of the room containing the original group. You'll end up with a new group at the right location in the target room by setting its displacement to the difference in position between the source and target rooms.



A big part of this process will be to find the room containing a particular group. For the sake of simplicity, you will assume that the selected group is completely enclosed by a room: it should not span two or more rooms. You will then look for the room that contains the group's center point. You will calculate the center point of that room and the copied group's target location at a fixed displacement from it. Finally, you will place the copy of the selected group at this location.

Here are the tasks you implemented in the previous lessons:

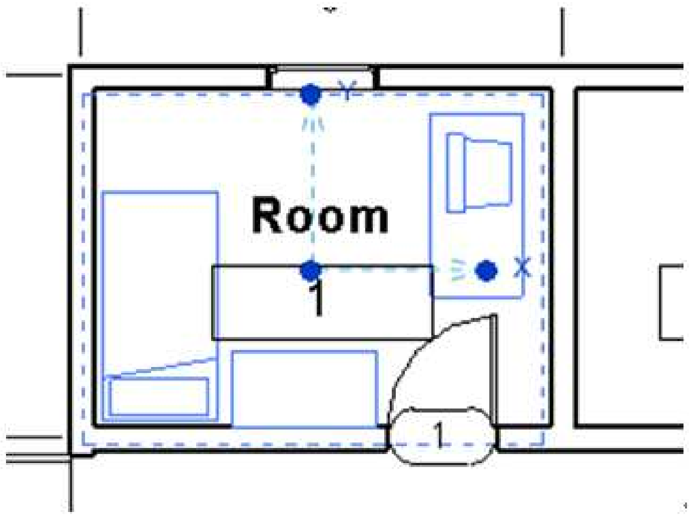
a. Prompt the user to select the group to be copied
b. Prompt the user to select the target location for this copy
c. Place the copy of the group at the target location

To implement the proposed enhancements, there are new tasks (in **bold**) to be added to the list above. Task b in the above list is no longer needed, as the group will now be placed using different logic.

a. Prompt the user to select the group to be copied
b. **Calculate the center of the selected group**
c. **Find the room that contains the center of the group**

d. **Calculate the center of the room**
e. **Display the x, y and z coordinate of the center of the room in a dialog box**
f. **Calculate the target group location based on the room's center**
g. Place the copy of the group at the target location (**some modification needed**)

As you can see from the above list, you're going to need to calculate the center point for both the Group and the Room objects. To do so, you will calculate the center point of their respective bounding boxes. As an example, in the following picture the dashed blue line is the bounding box of a selected group, while the blue dot in the middle is its center point.



Both the **Group** and the **Room** classes provide a **BoundingBox** property. They do so because they are derived from the **Element** class, and the Element class has this property. The Group and Room classes are said to **inherit** this property from their parent class. Because of this shared ancestry, you can implement a function which works for both the Group and the Room objects (and for any Element object, for that matter).

An Element's BoundingBox property returns a **BoundingBoxXYZ** object containing the minimum and maximum points of its geometry. You can calculate the center point from these two points by taking the point halfway between them.

To get the room that contains the selected group, you first need to get all rooms in the model and then go through each one, checking whether it contains the selected group's center point. You will calculate the center point of the room containing the group in the same way as you did for the group itself.

Coding the New Functionality

For clarity and better organization of the completed source code that we provide as an attachment for each lesson, we have changed the class names to match the lesson and the functionality we are working with.

In particular, in this lesson, the downloaded attachment's class name is changed from "Lab1PlaceGroup" to "Lab6FindRoom", as shown below:

public class Lab6FindRoom : IExternalCommand

{

Keep in mind that this change is not required, and that the detailed instructions we provide below assume you are always working from your original Lab1PlaceGroup code. This is why this name remains listed below and does not match the source code provided - whichever name you choose for your class name is fine.

1. Re-open the C# project that you have created in Lesson 5 in Visual C# Express, if you have closed it.

2. **b. Calculate the center of the selected group**:
Type the following code fragment inside the class **Lab1PlaceGroup**, making sure it is outside the **Execute()** method. The code defines a new method, **GetElementCenter(),** which takes an **Element** as a parameter and returns its center.

```
/// Return the center of an element based on its BoundingBox.
public XYZ GetElementCenter(Element elem)
{
    BoundingBoxXYZ bounding = elem.get_BoundingBox(null);
    XYZ center = (bounding.Max + bounding.Min) * 0.5;
    return center;
  }
```

In the Execute() method, after the line where you get the selected group, type the lines of code highlighted below in **bold**. The new statement calls your new GetElementCenter() method to get the center point of the selected group.

```
Group group = elem as Group;

// Get the group's center point
XYZ origin = GetElementCenter(group);
```

10. **c. Find the room that contains the center of the group**

Type the following code fragment inside the command class, **Lab1PlaceGroup**, making sure it is outside any existing method implementations. This code defines a new **GetRoomOfGroup()** method, which takes a Document and a point as parameters and returns the Room in which the specified point lies.

```
  /// Return the room in which the given point is located
  Room GetRoomOfGroup(Document doc, XYZ point)
  {
    FilteredElementCollector collector =
      new FilteredElementCollector(doc);
    collector.OfCategory(BuiltInCategory.OST_Rooms);
    Room room = null;
    foreach (Element elem in collector)
    {
      room = elem as Room;
      if (room != null)
      {
        // Decide if this point is in the picked room
        if (room.IsPointInRoom(point))
        {
          break;
        }
      }
    }
    return room;
}
```

Back in the Execute() method, after the line, **GetElementCenter(),** which you added in the last step, type the lines of code highlighted below in **bold**. The new statement calls your new GetRoomOfGroup() method to find the room containing the center of the selected group.

```
// Get the group's center point
XYZ origin = GetElementCenter(group);

// Get the room that the picked group is located in
Room room = GetRoomOfGroup(doc, origin);
```

4. **d. Calculate the center of the room and e. Display the x, y and z coordinate of the center of the room in a dialog box***:*

Type the following code fragment inside the command class, once again making sure the code is outside any existing methods. The code defines a new **GetRoomCenter()** method, which takes a Room and – as the name suggests – returns its center point. You use the previously defined GetElementCenter() to calculate this, but you modify the Z coordinate of the point you return to make sure it's on the floor of the room.

```
  /// Return a room's center point coordinates.
  /// Z value is equal to the bottom of the room
  public XYZ GetRoomCenter(Room room)
  {
    // Get the room center point.
    XYZ boundCenter = GetElementCenter(room);
    LocationPoint locPt = (LocationPoint)room.Location;
    XYZ roomCenter =
13.         new XYZ(boundCenter.X, boundCenter.Y, locPt.Point.Z);
14.       return roomCenter;
15.     }
```

In the Execute() method, after the statement which finds the room containing the center point of your group, type in the lines of code highlighted in **bold**, below. The code gets the room's center point and displays it to the user via a task dialog (a type of dialog that uses the Autodesk Revit user interface style).

```
// Get the room that the picked group is located in
Room room = GetRoomOfGroup(doc, origin);

// Get the room's center point
XYZ sourceCenter = GetRoomCenter(room);

string coords =
  "X = " + sourceCenter.X.ToString() + "\r\n" +
  "Y = " + sourceCenter.Y.ToString() + "\r\n" +
  "Z = " + sourceCenter.Z.ToString();

TaskDialog.Show("Source room Center", coords);
```

The first argument of **TaskDialog.Show()** is the name of which should appear in the title bar at the top of the dialog.

16. Remove or comment out (using two forward slashes: "//") the following line, which was your former step b. Your new group will be placed relative to the center of the original group's room, so you don't need the user to select anything else, at this stage.

```
17.         //pick a point
18.         XYZ point = sel.PickPoint("Please pick a point");
```

19. **f. Calculate the target group location based on the room's center and g. Place the copy of the group at the target location:**
Remove or comment out the current **PlaceGroup()** call in the Execute() method and replace it with the following lines in **bold**. Your new group will be placed at a displacement of (13.12, 0, 0) in feet from the center point of the original group's room (13.12 feet is the width of the two rooms and therefore the horizontal distance between their center points). As both **sourceCenter** and **new XYZ(13.12,2,0)** are of type XYZ, they can be added together to get the new location coordinates.

```
20.         doc.Create.PlaceGroup(point, group.GroupType);
21.
22.         // Calculate the new group's position
23.         XYZ groupLocation = sourceCenter + new XYZ(13.12, 0, 0);
24.         doc.Create.PlaceGroup(groupLocation, group.GroupType);
```

This completes your code for this lesson. The complete code for this lesson is also provided for download at the top of this lesson. It can be useful to see the complete code to compare your results and ensure they are correct.

25. **Save the file:**
On the **File** menu, click **Save All**.

26. **Build the project:**
Note: If Revit Architecture is already running, please close it.

Inside Visual C# Express, in the **Debug** menu, click **Build Solution** to compile and build your plug-in. If the code builds successfully, you will see the **Build succeeded** message in the status bar of Visual C# Express.

Running the Plug-in

The steps to run the command are similar to those used in prior lessons.

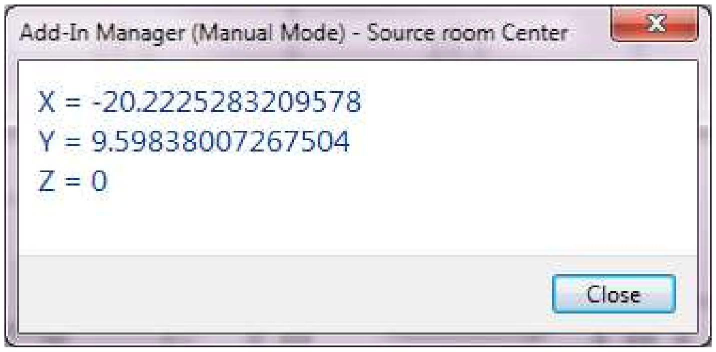
1. Start Autodesk Revit Architecture.

2. Open the Hotel.rvt Project file.

3. Start the command **Lab1PlaceGroup**.

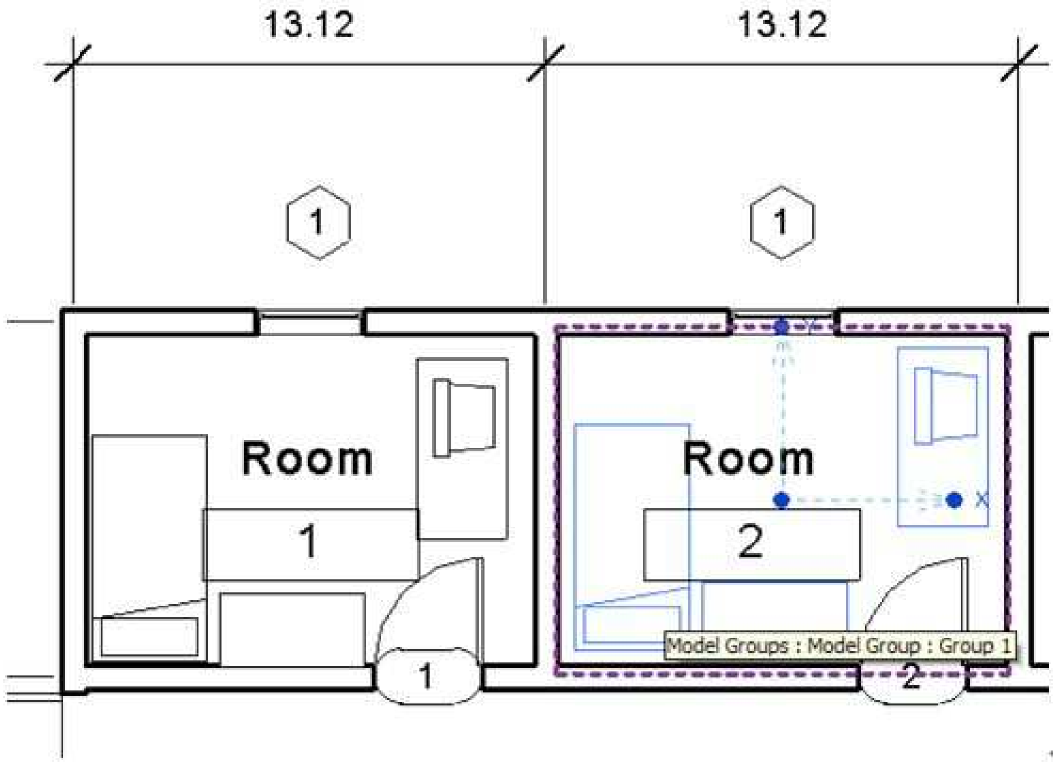
4. Select the group in Room 1, as you did in the previous lesson.

You will see the following task dialog showing the coordinates of the room's center.



```
X = -20.2225283209578
Y = 9.59838007267504
Z = 0
```

Click the Close button. Following this, a new group should be inserted into Room 2. Because the displacement of (13.12,0,0) is the vector from the center of Room 1 to the center of Room 2, the group appears to be copied from Room 1 to the same relative location in Room 2.



A Closer Look at the Code

Let's now take a closer look at the code. You'll start by looking at your new methods, before looking at the changes to the command's Execute() method.

You defined the **GetElementCenter()** method as follows:

```
public XYZ GetElementCenter(Element elem)
{
  BoundingBoxXYZ bounding = elem.get_BoundingBox(null);
  XYZ center = (bounding.Max + bounding.Min) * 0.5;
  return center;
}
```

In the implementation of the GetElementCenter() method, you started by accessing the **BoundingBox** property of the Element passed in, storing its value in a variable named **bounding**.

```
BoundingBoxXYZ bounding = elem.get_BoundingBox(null);
```

The BoundingBox property is slightly unusual in that it takes a parameter: the view for which the bounding box is to be calculated. If this parameter is null, the property returns the bounding box of the model geometry. If a property of a class takes one or more parameters, the **get_** prefix is needed before the property name to read the property value. This prefix isn't needed if the property doesn't take any parameters: you can just use the property name.

The returned **BoundingBoxXYZ** contains the coordinates of the minimum and maximum extents of the Element's geometry. The center point is calculated by taking the average (or mid-point) of these two points. For the sake of clarity, you stored this in another variable named **center**.

```
    XYZ center = (bounding.Max + bounding.Min) * 0.5;
```

Finally, you returned this center point from the function.

You defined **GetRoomOfGroup()** method as follow:

```
  Room GetRoomOfGroup(Document doc, XYZ point)
  {
    FilteredElementCollector collector =
      new FilteredElementCollector(doc);
    collector.OfCategory(BuiltInCategory.OST_Rooms);
    Room room = null;
    foreach (Element elem in collector)
    {
      room = elem as Room;
      if (room != null)
      {
        // Decide if this point is in the picked room
        if (room.IsPointInRoom(point))
        {
          break;
        }
      }
    }
    return room;
  }
```

Let's now take a closer look at the implementation of the **GetRoomOfGroup()** method. In this method, you started by retrieving all the rooms in the document, going through them to find the room that contains the group. The **FilteredElementCollector** class helped you with this task: it collects elements of a certain type from the document provided. That's why you needed to pass a document parameter to the GetRoomOfGroup() method, so it can be used there.

```
    FilteredElementCollector collector = new FilteredElementCollector(doc);
```

The **collector** object is now used to filter the elements in the document. In the next step you added a filter requesting that only rooms be collected.

```
    collector.OfCategory(BuiltInCategory.OST_Rooms);
```

You added your category filter to the collector using the **OfCategory()** method. Once the filter was applied, the collector only provided access to rooms. The FilteredElementCollector class provides several methods to add filters (and multiple methods can be applied at the same time for more complex requirements). More information on adding filters is provided later in this lesson in the Additional Topics section.

You then iterated through each room in the collector using a **foreach** expression. The code between the braces is repeatedly executed on each of the elements found by the collector. While you know these elements will be rooms, at this stage you accessed them as generic elements, as that's how the FilteredElementCollector provides access to them.

```
    foreach (Element elem in collector)
    {
      //code between braces pair executed repetitively.
    }
```

The **elem** variable represents the current element in the collector. So when the code in the body of the foreach statement gets executed for the first time, the elem variable contains the first room. When the code in the body of the foreach statement is executed again, this time the elem variable contains the second room. And so on until you have executed the code against each of the rooms collected from the document.

As mentioned before, the collector provides you with access to each element it contains an instance of the generic **Element** class. As you know your elements will actually be instances of the Room class, you needed to cast them to that type before being able to access the functionality you needed from them (i.e. the **IsPointInRoom()** method).

```
      room = elem as Room;
```

As you may recall, the **as** keyword first checks the actual type of the object before performing the type conversion: if the object is not of type Room, the variable will be set to null. Even though you fully expected the collector only to return rooms, it's still good practice to double-check that the room variable contains a valid room, just in case.

```
      if (room != null)
```

The above **if** statement performs a **conditional** operation. If the condition provided between the brackets evaluates to true, the subsequent code block gets executed. An optional **else** clause can be used to execute different code when the condition evaluates to false (although this particular statement does not have one). The **if** statement is a very important programming concept, so we have provided more information on its use in the Additional Topics section.

As implied by its name, the IsPointInRoom() method judged whether the specified point is inside the boundaries of the room. If it was inside, the method returned **true**, otherwise **false**. You called IsPointInRoom() on each room: as soon as it returned true for a particular room, you knew you had found the one that contained your point and so you did not need to check the others. You then used a **break** statement to escape the iteration, even though there may well have been rooms that had not yet been checked. The break statement stops execution of code in the enclosing loop (in this case the foreach) and starts executing the code following it.

```
        if (room.IsPointInRoom(point))
        {
            break;
        }
```

On completion of the loop, the room variable either contains the room in which the point was found – if IsPointInRoom() succeeded for it – or the last room in the list of rooms, otherwise. In either case, the contents of this variable gets returned as the result of the GetRoomOfGroup() method.

```
    return room;
```

You defined **GetRoomCenter()** method as follows:

```
  public XYZ GetRoomCenter(Room room)
  {
    // Get the room center point.
    XYZ boundCenter = GetElementCenter(room);
    LocationPoint locPt = (LocationPoint)room.Location;
    XYZ roomCenter =
      new XYZ(boundCenter.X, boundCenter.Y, locPt.Point.Z);
    return roomCenter;
  }
```

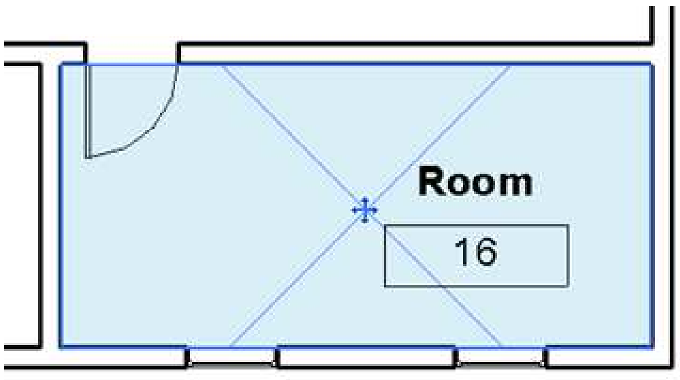
Let's now take a closer look at the implementation of the GetRoomCenter() method. This function is internally going to make use – once again – of your GetElementCenter() method, modifying the returned point to make it at the same level as the floor of the room.

The first line of actual code in the implementation of the GetRoomCenter() stores the center point of your Room's geometry in a variable, named **boundCenter:**

```
    // Get the room center point.
    XYZ boundCenter = GetElementCenter(room);
```

In order to make sure this point is adjusted to be at the elevation of the Room, you needed to access the Room's location point. The location point of a Room is always on its floor, so you determined its elevation by checking the location's Z coordinate.

The location of a particular room is at the intersection of the lines of the cross displayed by Revit when the room is selected.



You accessed this point via the Room's **Location** property. As a Room is located at a particular point, the value returned from the **Location** property is actually a **LocationPoint** object. But the Location property – as it is shared with other Elements – returns a more generic Location object. To get access to the point information stored in the property, you therefore needed to cast its value to be a LocationPoint. You stored the result of type conversion – i.e. the room's location – in a variable of type LocationPoint.

```
    LocationPoint locPt = (LocationPoint)room.Location;
```

This cast does not use the as keyword, as you saw before. The as keyword checks the type of an object before casting it to the target type. The above code does not perform this check and is appropriate for cases where you know the underlying type of the object. As you know that instances of the Room class always return a LocationPoint from their Location property, it's quite safe to use it here. If the property somehow could not be treated as a LocationPoint (hypothetically speaking), this operation would cause an **InvalidCastException** to be thrown.

To get the modified point to return, you took the X and Y coordinates from your Room's center point and the Z coordinate from its LocationPoint and used these three values to create a new XYZ point.

```
    XYZ roomCenter =
          new XYZ(boundCenter.X, boundCenter.Y, locPt.Point.Z);
```

Finally, you returned the adjusted point from the **GetRoomCenter()** method.

That's it for your new methods. Let's now take a look at the code you inserted into your command's Execute() method.

Now that you have your methods in place to do the actual calculations, it's a fairly simple matter of calling them, one after the other. You can see a progression in the data that is effectively passed from one to the other.

```
          // Get the group's center point
          XYZ origin = GetElementCenter(group);

          // Get the room that the picked group is located in
          Room room = GetRoomOfGroup(doc, origin);

          // Get the room's center point
          XYZ sourceCenter = GetRoomCenter(room);
```

You started by getting the center of your group using **GetElementCenter()**, placing it in the **origin** variable of type XYZ. This was passed into the GetRoomOfGroup() method – along with the active document – to help with the room collection process, which returned the containing Room object to be stored in the room variable. This was then passed to the GetRoomCenter() method to determine the center of this room, which was stored in the **sourceCenter** variable.

The sourceCenter variable of type XYZ – containing the center of the Room where the Group selected by the user resides – then needed to be displayed to the user in a dialog box.

Each coordinate – X, Y and Z – of the XYZ class is of type double. This is the type you use to represent a *double-precision, floating-point* number. It's not important to worry about what the terms in italics actually mean: suffice it to say that a variable of type double uses 64-bits of data to hold an extremely accurate decimal number (to an accuracy of 15-16 digits).

To show these values in a dialog box, they need to be converted to a string. **sourceCenter.X** returns the point's X coordinate as a double, which can then be converted to a string type using the **ToString()** method.

The + operator concatenates (joins together) two strings. The "\r\n" string represents a line break (it's a combination of a *carriage return* (\r) followed by a *linefeed* (\n), two characters that hark back to the days of the typewriter). This allowed you to split your string across multiple lines in the dialog box.

```
string coords =
    "X = " + sourceCenter.X.ToString() + "\r\n" +
    "Y = " + sourceCenter.Y.ToString() + "\r\n" +
    "Z = " + sourceCenter.Z.ToString();
```

The **TaskDialog** class allows you to display a standardized dialog box inside the Revit user interface. It's possible to add code to customize the dialog's controls prior to its display, but in this example you just used the **Show()** method to display your message. The first parameter is the dialog's title and the second is the main message.

```
TaskDialog.Show("Source room Center", coords);
```

This wraps up the coding part of Lesson 6, before you move on to the next lesson, we will discuss a couple topics more in depth: Filtering with FilteredElementCollector and the if Statement.

Additional Topics

Filtering with FilteredElementCollector

In this lesson, you requested the contents of the Revit model to be filtered using the **OfCategory()** method:

```
collector.OfCategory(BuiltInCategory.OST_Rooms);
```

Element access and filtering is extremely important to Revit programming. The **FilteredElementCollector** class provides several methods to add filters. The **OfClass()** method adds a class filter, so that the collected elements are all instances of the specified class. The **OfCategoryId()** method adds a filter such that the elements collected will all have the specified category identifiers. The combined filter can be added by the **WherePassed()** method. The use of some or all of these methods allows you to extract or filter out elements for a wide range of scenarios. This helps avoid iterating through very large sets of elements – often a very time-consuming and resource-intensive process. You can find more information about filtering in "Chapter 6 Filtering" of the Revit API Developer Guide in the Revit SDK.

The if Statement

The ability to execute or evaluate different blocks of code in different scenarios is a core part of all programming languages. In C# this is most commonly achieved using the **if** statement.

The if statement allows you to execute statements (or sets of statements) conditionally based on the value of an expression which evaluates to a Boolean (i.e. true or false).

In the following example, a Boolean variable, named **flagCheck**, is set to true and then checked in the following if statement. The output is: The flag is set to true.

```
bool flagCheck = true;
if (flagCheck)
{
    Console.WriteLine("The flag is set to true.");
}
else
{
    Console.WriteLine("The flag is set to false.");
}
```

The expression in the parentheses is evaluated to be true. Then the **Console.WriteLine("The Boolean flag is set to true.");** statement is executed. After executing the if statement, control is transferred to the next statement. The else clause is not executed in this example.

If you wish to execute more than one statement, multiple statements can be conditionally executed by including them in blocks using {}.

Lesson 7: My Final Plug-in

Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

Lesson Downloads

- lesson7_revit_2014_projects.zip (zip - 5669Kb)
- lesson7_revit_2013_projects.zip (zip - 21633Kb)
- lesson7_revit_2012_and_earlier_project_files.zip (zip - 7126Kb)

This is the last hands-on lesson of this guide, during which you will put the finishing touches on your plug-in's functionality, making it usable in a real-world scenario.

In the last lesson, you used the Autodesk Revit API to calculate and display the center of the room containing the user-selected group. You then duplicated the group and placed it at the center of another room. During this lesson you'll complete the plug-in's functionality, extending it to allow the user to select multiple rooms to which the group will be copied.

Planning the New Functionality

Let's start with a quick review of the main tasks covered during the previous lesson, looking at where and how to add this lesson's tasks to the sequence.

a. Prompt the user to select the group to be copied
b. Calculate the center of the selected group
c. Find the room that contains the center of the group
d. Calculate the center point of the room
e. Display the x, y and z coordinate of the center of the room in a dialog box
f. Calculate the target group location based on the room's center
g. Place the copy of the group at the target location

Your main task for this lesson is to prompt the user to select target rooms, information that is needed to place the copies of the selected group. There are a couple of choices here: one way to do this would be to keep the user selection related functionality together at the beginning, making sure that the user has entered all the required information before you start your real work. Another would be to have the group information calculated first and then you add your new tasks of selecting multiple rooms. Here we will choose the latter option. As you really don't need to display the room coordinates to the user in the final plug-in, you'll remove that task and add your new tasks after that (in **bold**, below).

a. Prompt the user to select the group to be copied
b. Calculate the center of the selected group
c. Find the room that contains the center of the group
d. Calculate the center point of the room
e. **Prompt the user to select the rooms to which the group will be copied**
f. **Place the group at the center of each of the selected rooms**

Now that you have the overall plan in place, let's look at the steps needed to implement this new functionality.

Coding the New Functionality

For clarity and better organization of the completed source code that we provide as an attachment for each lesson, we have changed the class names to match the lesson and the functionality we are working with.

In particular, in this lesson, the downloaded attachment's class name is changed from "Lab1PlaceGroup" to "Lab7SelectedRoomsAndPlaceGroups", as shown below:

public class Lab7SelectedRoomsAndPlaceGroups : IExternalCommand

{

Keep in mind that this change is not required, and that the detailed instructions we provide below assume you are always working from your original Lab1PlaceGroup code. This is why this name remains listed below and does not match the source code provided - whichever name you choose for your class name is fine.

1. Re-open the C# project that you have created in Lesson 5 in Visual C# Express, if you have closed it.

2. **e. Prompt the user to select the rooms to which the group will be copied:**

Start by adding a new type of filter class outside the implementation of the **Lab1PlaceGroup** class.

```
/// Filter to constrain picking to rooms
public class RoomPickFilter : ISelectionFilter
{
  public bool AllowElement(Element e)
  {
    return (e.Category.Id.IntegerValue.Equals(
      (int)BuiltInCategory.OST_Rooms));
  }

  public bool AllowReference(Reference r, XYZ p)
  {
    return false;
  }
}
```

You'll make use of this filter in the implementation of your **Execute()** method.

You no longer need to report the source room's center point to the user, so you can either remove or comment out the following lines of code. To comment out a block of code you can enclose the block in /* */, such as this:

```
/*
string coords =
  "X = " + sourceCenter.X.ToString() + "\r\n" +
  "Y = " + sourceCenter.Y.ToString() + "\r\n" +
  "Z = " + sourceCenter.Z.ToString();
*/
```

Beneath that add the code to prompt the user to select rooms, making use of your new **RoomPickFilter** class.

```
// Ask the user to pick target rooms
RoomPickFilter roomPickFilter = new RoomPickFilter();
IList<Reference> rooms =
  sel.PickObjects(
    ObjectType.Element,
    roomPickFilter,
    "Select target rooms for duplicate furniture group");
```

16. **f. Place the group at the center point of each of the selected room:**

You'll start this task with the implementation of a new method inside the **Lab1PlaceGroup** class. Add the following to Lab1PlaceGroup class:

```
/// Copy the group to each of the provided rooms. The position
/// at which the group should be placed is based on the target
/// room's center point: it should have the same offset from
/// this point as the original had from the center of its room
public void PlaceFurnitureInRooms(
  Document doc,
  IList<Reference> rooms,
  XYZ sourceCenter,
  GroupType gt,
  XYZ groupOrigin)
{
  XYZ offset = groupOrigin - sourceCenter;
  XYZ offsetXY = new XYZ(offset.X, offset.Y, 0);

  foreach (Reference r in rooms)
```

```
          {
            Room roomTarget = r.Element as Room;
            if (roomTarget != null)
            {
              XYZ roomCenter = GetRoomCenter(roomTarget);
              Group group =
                doc.Create.PlaceGroup(roomCenter + offsetXY, gt);
            }
          }
    }
```

Note: If you are working with Revit 2013 and higher API, the following line of code in the code snippet above

```
Room roomTarget = r.Element as Room;
```

needs to be replaced with the following:

```
Room roomTarget = doc.GetElement(r) as Room;
```

Now you need to edit your Execute() method to make use of this method to place copies of the selected group in each of the selected rooms.

You no longer need the next two lines of code. You can either remove them or comment them out:

```
XYZ groupLocation = sourceCenter + new XYZ(13.12, 0, 0);
doc.Create.PlaceGroup(groupLocation, group.GroupType );
```

Add the following statement, in place of the above, which calls your new method:

```
PlaceFurnitureInRooms(
  doc, rooms, sourceCenter,
  group.GroupType, origin);
```

The resulting code fragment should now look like this:

```
// Place furniture in each of the rooms
Transaction trans = new Transaction(doc);
trans.Start("Lab");
PlaceFurnitureInRooms(
  doc, rooms, sourceCenter,
  group.GroupType, origin);
trans.Commit();
```

This completes the code for this lesson and for your plug-in. The complete code for this lesson is also provided for download at the top of this lesson. It can be useful to see the complete code to compare your results and ensure they are correct.

41. **Save the file:**
On the **File** menu, click **Save All**.

42. **Build the project:**
Note: If Revit Architecture is already running, please close it.

Inside Visual C# Express, in the **Debug** menu, click **Build Solution** to compile and build your plug-in. If the code builds successfully, you will see the **Build succeeded** message in the status bar of Visual C# Express.

Running the Plug-in

The steps to run the command are similar to those used in prior lessons.

1. Start Autodesk Revit Architecture.
2. Open the Hotel.rvt Project file.
3. Start the command **Lab1PlaceGroup**.
4. Select the group in Room 1, as you did in the previous lesson.
5. Select multiple rooms to where you would like the selected group to be copied.

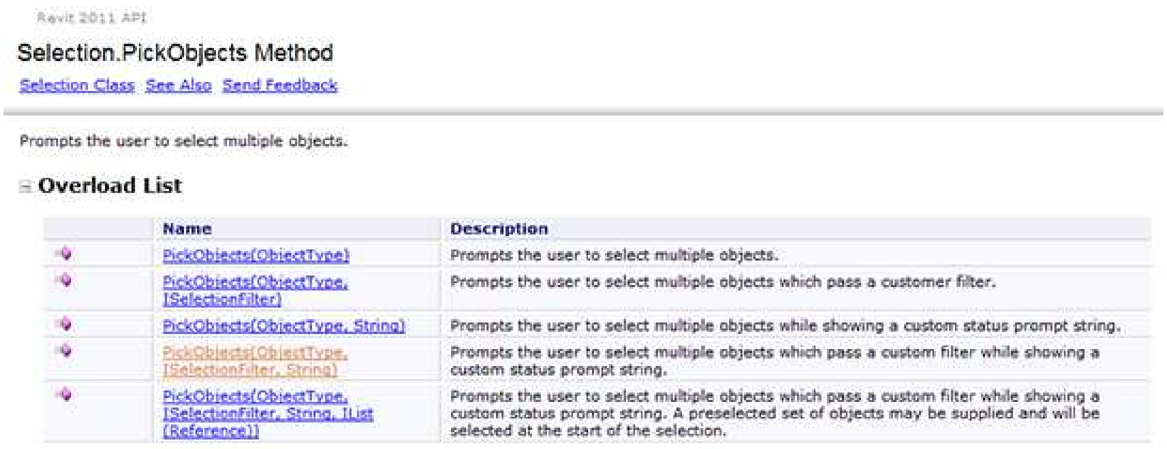
The selected group should be copied to the same relative position in each of the selected rooms.

A Closer Look at the Code

Let's now take a closer look at the implementation details for the tasks you have added during this lesson. The first of these tasks deals with asking the user to select a set of rooms.

```
IList<Reference> rooms =
  sel.PickObjects(
    ObjectType.Element,
    roomPickFilter,
    "Select target rooms for duplicate furniture group");
```

You have already used **PickObject()** and **PickPoint()** in previous lessons, to select a single object (i.e. a Group) and a point, respectively. Based on the naming convention of these methods, let's now discover what other selection methods are available to you. If you open the RevitAPI.chm file (the Revit API Help Documentation) and search with the keyword *Pick*, you will find a list of methods, one of which is called **PickObjects()**. As you can see from the description of this method, it is exactly what you're looking for:



Revit 2011 API

Selection.PickObjects Method

Selection Class See Also Send Feedback

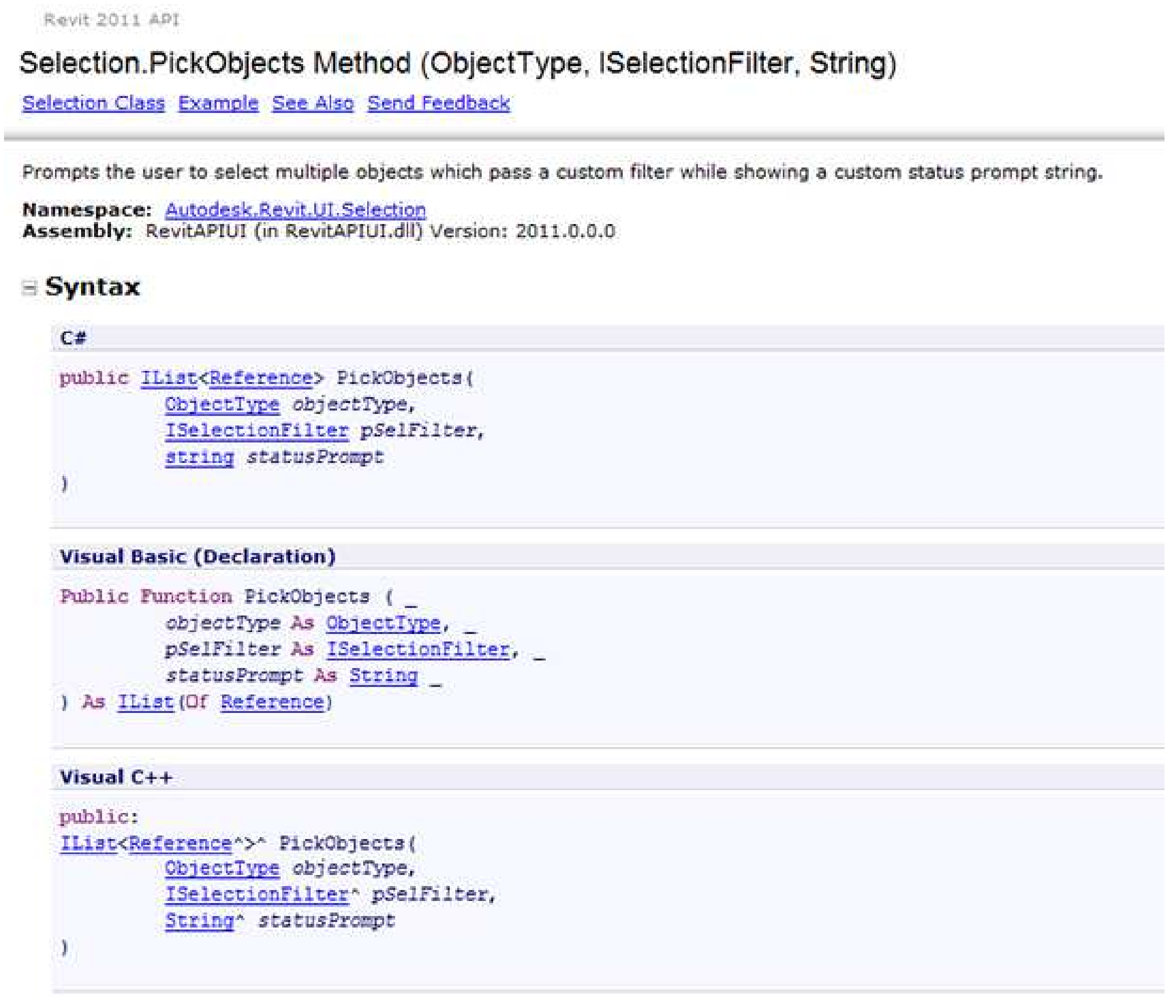
Prompts the user to select multiple objects.

Overload List

	Name	Description
	PickObjects(ObjectType)	Prompts the user to select multiple objects.
	PickObjects(ObjectType, ISelectionFilter)	Prompts the user to select multiple objects which pass a customer filter.
	PickObjects(ObjectType, String)	Prompts the user to select multiple objects while showing a custom status prompt string.
	PickObjects(ObjectType, ISelectionFilter, String)	Prompts the user to select multiple objects which pass a custom filter while showing a custom status prompt string.
	PickObjects(ObjectType, ISelectionFilter, String, IList(Reference))	Prompts the user to select multiple objects which pass a custom filter while showing a custom status prompt string. A preselected set of objects may be supplied and will be selected at the start of the selection.

As you can see, there are five different forms of PickObjects() (you may recall from looking at the PickObject() method that this is known as function overloading). As with your use of PickObject(), it's the fourth version that appears to be of most interest, as it allows us to use a selection filter and to display a prompt string to the user.

Click on the link for the fourth method to show you the details of this overload:



Revit 2011 API

Selection.PickObjects Method (ObjectType, ISelectionFilter, String)

Selection Class Example See Also Send Feedback

Prompts the user to select multiple objects which pass a custom filter while showing a custom status prompt string.

Namespace: Autodesk.Revit.UI.Selection
Assembly: RevitAPIUI (in RevitAPIUI.dll) Version: 2011.0.0.0

Syntax

C#

```
public IList<Reference> PickObjects(
	ObjectType objectType,
	ISelectionFilter pSelFilter,
	string statusPrompt
)
```

Visual Basic (Declaration)

```
Public Function PickObjects ( _
	objectType As ObjectType, _
	pSelFilter As ISelectionFilter, _
	statusPrompt As String _
) As IList(Of Reference)
```

Visual C++

```
public:
IList<Reference^>^ PickObjects(
	ObjectType objectType,
	ISelectionFilter^ pSelFilter,
	String^ statusPrompt
)
```

As you can see from the method's signature, it returns an IList of Reference objects. IList is a generic list class defined in the **System.Collections.Generic** namespace which can be specialized to contain a specific type of object – in this case instances of the Revit API's Reference class. You therefore declared a variable of type IList to which you assigned the results of the call to the PickObjects() method.

Before going ahead and calling this method, you needed to implement another selection filter, similar to the one you created previously for Group selection.

Here's the definition of your selection filter class limiting selection to Room objects:

```
public class RoomPickFilter : ISelectionFilter
{
  public bool AllowElement(Element e)
  {
    return (e.Category.Id.IntegerValue.Equals(
      (int)BuiltInCategory.OST_Rooms));
  }

  public bool AllowReference(Reference r, XYZ p)
  {
    return false;
  }
}
```

This class is very similar to the one you created previously: the changes are with the name – this one is called **RoomPickFilter** – and the category ID upon which to filter (**OST_Rooms**).

Back in the Execute() method, you created a RoomPickFilter before calling the PickObjects() method.

```
   RoomPickFilter roomPickFilter = new RoomPickFilter();
```

In the next statement, which is spread over multiple lines, you declared your reference list and assigned the results of the PickObjects() call to it:

```
IList<Reference> rooms =
  sel.PickObjects(
    ObjectType.Element,
    roomPickFilter,
    "Select target rooms for duplicate furniture group");
```

The first line contains the left-hand side of the assignment, specifying the name (rooms) and the type (IList<Reference>) of your variable. The subsequent lines call the function, passing the required parameters to it.

The second task in this lesson was to place a new group in each of the selected rooms. This involved looping through each of the rooms, calculating the center point and placing a new group relative to that point.

Here's the new method performing that task:

```
public void PlaceFurnitureInRooms(
  Document doc,
  IList<Reference> rooms,
  XYZ sourceCenter,
  GroupType gt,
  XYZ groupOrigin)
{
  XYZ offset = groupOrigin - sourceCenter;
  XYZ offsetXY = new XYZ(offset.X, offset.Y, 0);

  foreach (Reference r in rooms)
  {
    Room roomTarget = r.Element as Room;
    if (roomTarget != null)
    {
      XYZ roomCenter = GetRoomCenter(roomTarget);
      Group group =
        doc.Create.PlaceGroup(roomCenter + offsetXY, gt);
    }
  }
}
```

In the lines directly after the method's signature – where you defined the parameters being passed in – you defined two variables, **offset** and **offsetXY**. The offset variable holds the difference between the group's origin and the center of the room in which the group is located. This gives you an offset you can use to position each of the new groups relative to its target room's center. As you want the group to maintain the same level as the target room, you only care about X and Y offsets. This is the offset that you applied when placing your new groups.

The next statement was your **foreach** loop. You took each item from the list of rooms, in turn, and did something with it.

Inside the foreach loop, you first performed a cast on each Reference from the list, to be able to treat it as a Room. You did this using the "as" keyword, which – as you know – will return null if the object is not actually a Room. It's for this reason that you then used an if statement to make sure you actually have a Room to work with before proceeding (i.e. if it is not null, as "!=" means "is not equal to").

Assuming you have a valid Room, you then used the **GetRoomCenter()** method from the previous lesson to get the room's center, stored it in the **roomCenter** variable, and placed the group at the appropriate offset from the center of the target room (roomCenter + offsetXY).

With your new **PlaceFurnitureInRoom()** method implemented, it remained simply to update the **Execute()** method to make use of it.

Instead of your previous call to PlaceGroup(), you changed the code to call PlaceFurnitureInRoom(), passing in the required parameters.

```
// Place furniture in each of the rooms
Transaction trans = new Transaction(doc);
```

```
    trans.Start("Lab");
    PlaceFurnitureInRooms(
      doc, rooms, sourceCenter,
      group.GroupType, origin);
    trans.Commit();
```

This brings you to the end of this lesson and to the practical section of this guide.

Congratulations! You have just completed all the lessons in this guide. You have completed the journey from being a Revit product user to learning the basics of programming, getting a first look at the Revit API and creating your first real-world plug-in for Revit.

All that remains is to take a look at some of the resources you will find useful during your onwards journey with the Autodesk Revit API. We wish you all the very best!

Lesson 8: Learning More

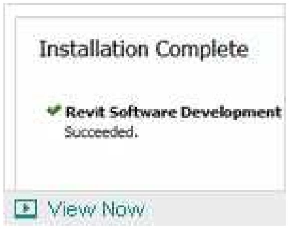
Provide Feedback: Please provide feedback about this Revit Training or this lesson via email: myfirstplugin@autodesk.com

This final lesson is intended to provide you with additional information to increase your productivity when creating Autodesk Revit plug-ins. You will look at information on API-related help, documentation, samples and – most importantly – where to go next to continue your learning around the Autodesk Revit API.

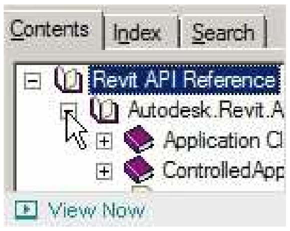
Let's first look at the API resources that can be installed along with the Autodesk Revit product onto your computer:

Resources on your System

Autodesk Revit Software Development Kit (SDK)
The Revit SDK contains a great deal of useful information regarding the Revit API. The three main categories of content are: documentation, samples and tools.



Video: Installing Autodesk Revit SDK



Video: Exploring the Autodesk Revit SDK

Documentation

The SDK contains two Microsoft Word documents, one on getting started using the Revit API and another which lists the changes to the Revit API when compared with the last version. It also contains an Adobe PDF document called the Revit API Developer Guide, which serves as an introduction to various API concepts on a chapter-by-chapter basis. A complementary file, which is extremely important to anyone working with the Revit API, is the Revit API Help Documentation, found in RevitAPI.CHM file. You can search this file for a particular class, method or property, read their descriptions and take a look at associated sample code (if any is provided).

Samples

The SDK contains a comprehensive list of API samples which cover almost all the major Revit API topics. One useful tip is to open the main "Samples" solution file in Visual C# Express and search for the API method you wish to work with: if it exists in the samples, you can very quickly see how the API is used and – if it makes sense – re-use the code in your plug-in. The SDK also contains Visual Studio for Applications (VSTA) samples that show you how to take advantage of VSTA to put together quick prototypes to test particular Revit API functionality. Revit 2013 replaces VSTA with SharpDevelop and these SharpDevelop samples can be located in a folder named 'Macro Samples' in the Revit 2013 SDK.

Tools

1. **Autodesk Revit LookUp** is a tool that helps analyze the currently running instance of Revit, the documents loaded in it and the elements (including content) contained in the project. In other words, it's a great tool for taking the lid off the Revit database and understanding what's going on. The tool comes with complete source code which can also be used to

understand how to access and analyze model information programmatically. It's most commonly used as a standalone tool for inspecting the Revit model, though: you certainly don't need the source code for it to be useful.

2. **Add-In Manager** is a tool that helps add external commands and applications to Revit without having to manually edit manifest files (as you did in lesson 1). The tool has its own dialog that allows you to browse to the folder containing the plug-in DLL, to specify the command name and to provide a plug-in description. Using this information, you can load commands directly into Revit or save the plug-in information to the manifest file for future loading. This also helps avoid typing errors when working with manifest files that can lead to unexpected results when attempting to load a plug-in.

Resources Online

Autodesk Revit Developer Center

The Autodesk Revit Developer Center is a great resource for those working with the Autodesk Revit API. As well as listing recent API additions and improvements, it aggregates the various other resources available to you. Highlights of this webpage include:

1. The latest available download for Revit SDK

2. DevTVs, self-paced video tutorials which show you how to create your own plug-ins using the Revit API. These can be viewed online or downloaded for later viewing at your convenience.

3. Webcast recordings for the Revit API. They cover the APIs of Autodesk Revit Architecture, Autodesk Revit Structure, Autodesk Revit MEP as well as the Autodesk Revit Family API. These courses introduce basic Revit programming concepts and are a great way to learn more about using the Revit API: view the list.

The Building Coder Blog

The Building Coder is a popular blog containing information related to the Revit API, including white papers, discussions on the use of various APIs and the answers to many commonly-asked API questions.

ADN AEC DevBlog

ADN AEC DevBlog is a new resource for software developers working with Revit, Navisworks and other AEC and BIM technologies from Autodesk.

Other Online Resources

1. Autodesk Revit product downloads, these can be installed and used without activation for 30 days from the day of installation
   - Autodesk Revit Architecture
   - Autodesk Revit Structure
   - Autodesk Revit MEP

2. The Autodesk Discussion Group dedicated to the Revit API

3. API class content:
   - Autodesk Developer Camp 2012 *(zip -320 Mb)*
   - Autodesk Developer Camp 2010 *(zip -161 Mb)*

4. Autodesk Developer Network (ADN) members can use this online resource to access additional technical content, to submit specific questions related to the APIs to Autodesk products, and to access released and beta software for development purposes. If you are serious about creating plug-ins with the Revit API – even if for internal use within your company – and are interested in finding out more about what ADN can offer, please visit: http://www.autodesk.com/joinadn

This brings you to the end of this guide. We hope this has proven to be a helpful beginning to your journey with the Revit API. We wish you all the very best in your future API endeavors.