Lesson 3: A First Look at Code

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 in 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 topic here mentioned below.

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 Studio 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 System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.DB.Architecture;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
```

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:

```
[Transaction(TransactionMode.Manual)]
public class Class1 : IExternalCommand
{

}
```

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

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

This line is an attribute that helps control the transaction 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.

IExternalCommand

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

```
public class Class1 : 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 **Class1** 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 Class1 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 Studio 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 Studio, 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.

```
[Transaction(TransactionMode.Manual)]
    public class Class1 : 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
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).

```
//Get document objects
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 = doc.GetElement(pickedref);
   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.

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

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 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**.

My First Revit Plug-in Overview >> Lesson 1 >> Lesson 2 >> Lesson 3>> Lesson 4 >> Lesson 5 >> Lesson 6 >> Lesson 7 >> Lesson 8