# Create Your First Autocad Plugin

Today, you’re going to learn how to create a useful plugin for Autocad, using C#. I’ll try to explain things so that if you’re new to programming, you should be able to understand what’s going on. Learning how to program requires some effort, but that shouldn’t stop you from jumping in.

Before we get started, there are a few pre-requisites you’ll need.

* Visual Studio 2019 Community edition (free). You can get that here: <https://visualstudio.microsoft.com/downloads/>
* Autocad ObjectARX SDK (free) for the version of Autocad you have installed. You can get that here: <https://www.autodesk.com/developer-network/platform-technologies/autocad/objectarx-license-download>

## About Me

Ben Rand has been using AutoCAD software since Release 12. He learned to program using LISP in AutoCAD, worked his way up through VBA to VB.NET, and now spends most of his days programming in C# (occasionally still in AutoCAD!). He has worked in the Industrial Engineering field for more than 18 years as a CAD Manager, developer and IT Director. In 2013, he was the Top DAUG overall winner at AU, and he served as a mentor for the AutoCAD Mentor All-Star team. Ben has been presenting at AU since 2015, and was honored to appear on the Top-Rated Speaker list in 2017 and 2018, and is a Pluralsight Author. Ben is the proud father of four children and enjoys reading and playing a variety of sports including pickleball, volleyball, and tennis. In 2018, Ben was a member of a USTA men's league team that placed 1st in the entire country.

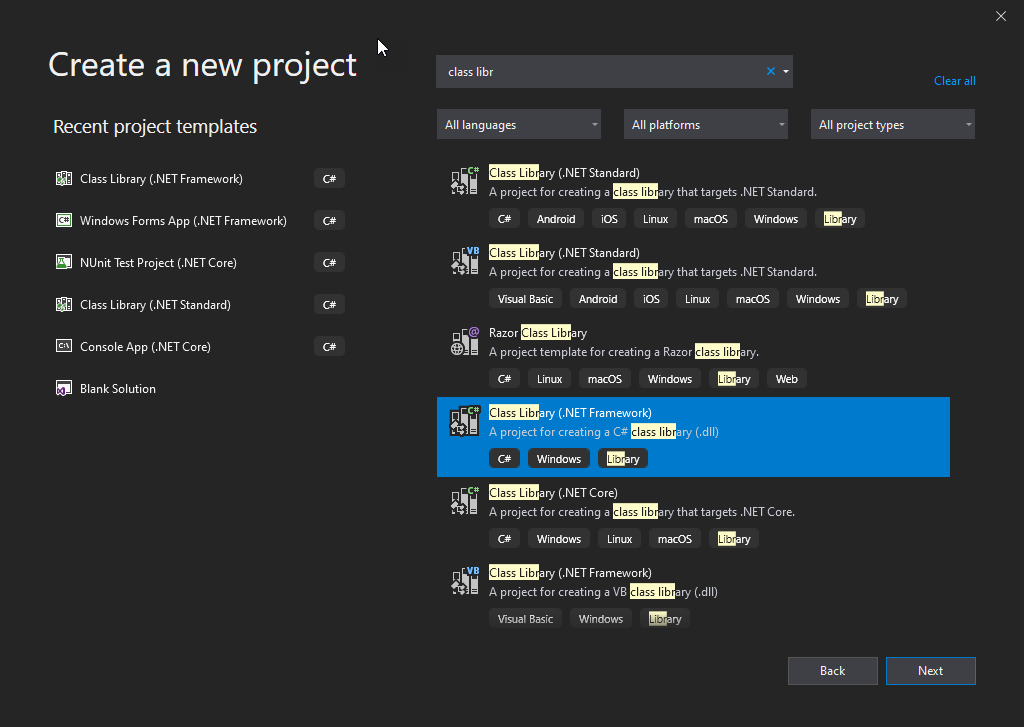
## Our scenario

Let’s give ourselves some context. You work for a local city government, and regularly deal with individual properties, or parcels. You’d like to make common tasks like drawing and quantifying parcels a little bit easier.

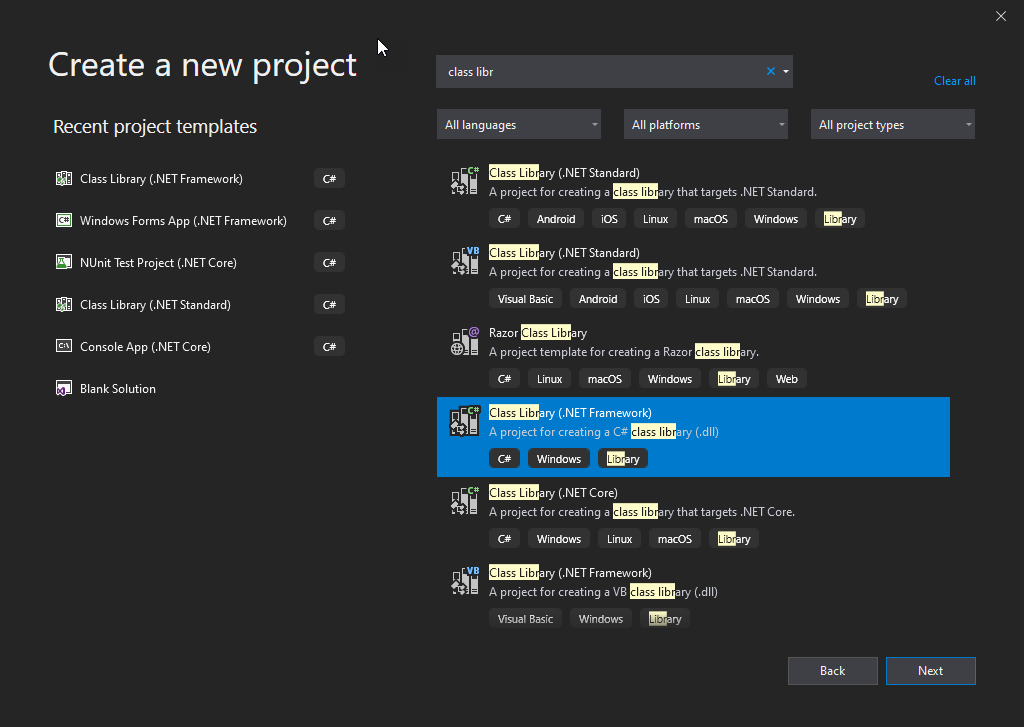
## Create a solution

The first thing you need to do is to create a new *solution* in Visual Studio (VS). When you start VS, it will ask, “What do you want to do?”

1. Click **Create a new project**.
2. In the **Search for templates** box, enter **class library**.
3. Select **Class Library (.NET Framework)** then click **Next**. Note that you’ll want the C# version to follow along with this class, although all of this is easily done in VB.NET as well.
4. Click **Next**.



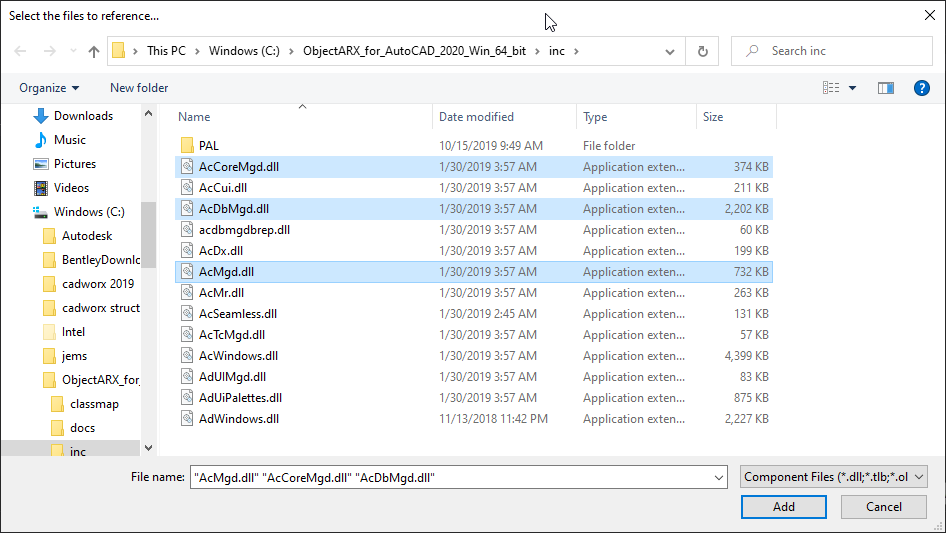
1. Enter a project name of **Parcels**.
2. Click **Create**.



1. In **Solution Explorer**, delete **Class1.cs**.

We need to add some references to various Autocad libraries, so that we can interact with them.

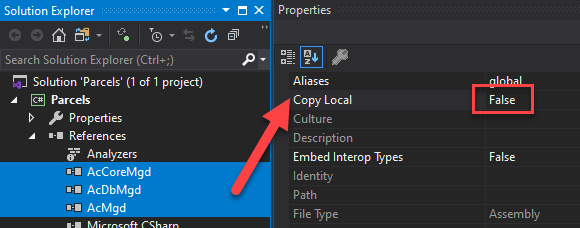
1. Right-click **References**,then choose **Add Reference**.
2. Click **Browse**,then locate the folder where you extracted/installed the ObjectARX SDK, then navigate to the **inc** folder, as shown below.



1. Hold the CTRL key and select **AcCoreMgd.dll**, **AcDbMgd.dll** and **AcMgd.dll**, then click **Add** and **OK**.

These are the most common libraries you need for interacting with Autocad and a drawing database.

1. In Solution Explorer, hold SHIFT and select each of the libraries you just added, then right-click and choose **Properties**.
2. Change **Copy Local** to **False**.



When your plugin is loaded into Autocad, these libraries will already have been loaded by Autocad itself, so there’s no need to copy them when you build your solution. We need them as references, though, so that we can program against them. This is required for any new Autocad plugin solution. After you’ve been through it a few times, it’ll take only a few seconds.

## Hello World

Every programming tutorial starts with a Hello World example.

1. Right-click the **Parcels** project node (not the solution at the top!) and choose **Add Class**.
2. Name the class **Commands** and click **Add**.

A *class* is a template for creating objects, which hold information and can perform operations when our program is running. Classes can define *fields* and *properties*, which hold information, and *methods*, which do things, like reading data from a text file, drawing a line in a drawing, or storing information in a database.

It’s a good idea to keep classes rather small and focused on a single job. The fields, properties and methods within that class should all be focused on doing that job. The methods should also be as short as possible, again focused on one particular task.

1. Change the Commands class code to read as follows:

using Autodesk.AutoCAD.Runtime;

using System;

using System.Linq;

namespace Parcels

{

public class Commands

{

[CommandMethod("PS\_Hello")]

public void Hello()

{

var document = Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocument;

var editor = document.Editor;

editor.WriteMessage("\nHello World!");

}

}

}

At the top of the file you see several **using** statements, such as

using Autodesk.AutoCAD.Runtime;

A *namespace* is a mechanism for organizing code, kind of like a folder helps us organize files. These give us access to classes within each of those namespaces. In the example above, we now have access to classes within the Autodesk.AutoCAD.Runtime namespace.

Next you see a **namespace** statement:

namespace Parcels

This indicates that the code in this file belongs to the Parcels namespace.

Squiggly brackets { } are used in C# at the beginning of each code block. Code blocks include namespaces, classes, methods, and within certain control structures such as **if**, **foreach**, **while** and **switch**.

Next we identify the class name:

public class **Commands**

In addition to identifying the class name, I have added the *accessibility modifier* **public** to indicate that this class is visible anywhere in our solution. We’ll keep things simple for now and make most classes and methods **public**.

Next we get to a **method** declaration (and yes, I’m skipping a line on purpose for just a moment):

public void Hello()

This is a *void* method which indicates that it does something, but does not return any value. We’ll see other methods later which return a value, in which case we’ll replace *void* with the type of value the method returns. This method has no *parameters,* so we just include an empty set of parentheses at the end of the statement.

Sitting on top of the method declaration you see:

[CommandMethod(“PS\_Hello”)]

This is known as an *attribute*, which adds metadata to the method. In this case, it’s telling Autocad that when the user types PS\_HELLO at the command prompt, run the Hello() method. Where does CommandMethod come from? The Autodesk.AutoCAD.Runtime namespace, which is why we needed the using statement at the very top of the file.

Our first statement inside the Hello() method is a doozy:

var document = Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocument;

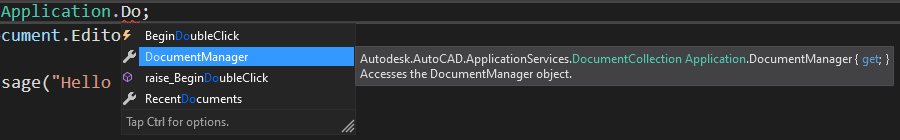
Here we’re establishing a **variable** (a placeholder) named ***document***. This is going to represent a **Document** object within AutoCAD. In other words, a drawing. Technically I should have identified the variable type as follows:

**Document document = Autodesk.AutoCAD…***truncated for brevity*

The **var** keyword tells C# to work out the variable type itself. I like it because it’s less typing and gets the job done. Note the really long chain of objects we have to navigate through to get there. In my opinion, this is one of the more intimidating aspects of programming in Autocad, trying to figure out where everything is.

The **Autodesk.Autocad** namespace makes some sense, since we’re programming in Autocad. We could surmise that there’s an **Autodesk.Revit** namespace floating out there in some other libraries. From there **ApplicationServices.Application** represents the application itself. Think of that like the Autocad program window (maybe not entirely accurate, but good enough for a start).

As you type in VS, *intellisense* kicks in and reveals available namespaces and classes, helping you (somewhat) through the discovery process. Below is a screenshot of this in action.



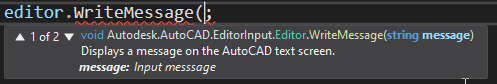
Autocad can have multiple drawings open at a time, which are managed by **DocumentManager**, and from there we can get the **MdiActiveDocument**, which represents the active drawing in Autocad when we run our command.

Once we have a variable representing the active document, we assign its **Editor** property to a variable named **editor**. This object enables us to interact with the drawing in many of the same ways we do in the drawing window and command prompt. For example, we can select objects in the drawing, write messages and much more.

Finally, we get to this statement:

editor.WriteMessage(“Hello World!”);

The **WriteMessage()** method has a required *string* argument (the message we want to output). Once again, intellisense guides us as we type.



This statement writes “Hello World!” to the command prompt. The “\n” feeds a newline character in first, so that “Hello World!” will appear on an empty line.

### Setting up for debugging

It’s time to run our program! Before we can do that, we need to setup one last thing. Class libraries (files with a **.dll** extension) are intended to be hosted within an executable program, in our case AutoCAD. So, we need to tell VS that when we **debug** our application, we want a host application to start first. We’ll then load our class library into the application, and run our new command.

1. Right-click the **Parcels** project in Solution Explorer, then choose **Properties**.
2. Select the **Debug** tab on the left.
3. Under **Start Action**, select **Start External program**, then click **Browse** and locate **acad.exe** on your system. Mine is at **C:\Program Files\Autodesk\AutoCAD 2020**.
4. Under **Start options** enter **/nologo**. This helps Autocad start up quicker.

Note that you can use any of your usual startup switches, such as **/p** followed by a profile name if you want Autocad to start in a specific profile.

1. Close the **Properties** tab, then click **Save All** on the toolbar.

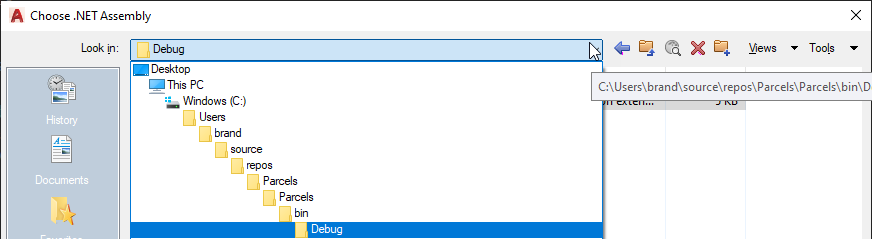
### Debugging in Autocad

At long last, we’re ready to run (debug) our program.

1. Click **Start** or press F5 to begin a debugging session.

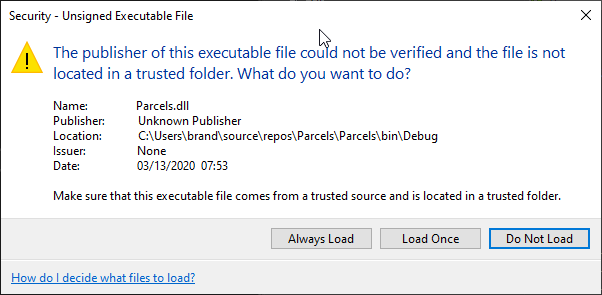
VS builds your application, then launches AutoCAD.

1. Start a new drawing, then type **NETLOAD** at the command prompt and press Enter.
2. Browse to your solution folder to locate the **Parcels.dll**. Mine is located at **C:\Users\*username*\source\repos\Parcels\Parcels\bin\Debug**.



C:\Users\*username*\source\repos is the default folder for VS solutions. The firstParcels folder is the solution folder. The second Parcels folder is the project folder. The bin folder is where binary (compiled) files are stored. And finally, the Debug folder is where our particular build is going. When you are ready to “ship” a final product, you should switch your build to Release which will send the binary files to the Release folder instead.

1. Click **Open** to load the library.
2. At the Security prompt, choose **Always Load**.



1. At the command prompt, type **PS\_HELLO** and press Enter.

You should see Hello World! at the command prompt.



1. Close Autocad to end your debugging session and return to VS.

## Layer Creator

Time for something more interesting. Let’s add a command to create and activate a layer in our drawing. In the **Commands** class, insert the following code after the closing bracket of the Hello() method, and before the last two brackets which close the class and namespace blocks.

[CommandMethod("PS\_CreateParcelLayer")]

public void CreateLayer()

{

var creator = new ParcelLayer();

creator.Create();

}

At this point, most tutorials would just jam all the code for creating a layer in this method, but that’s a really good way to start you on the path of writing *spaghetti code*, which means code that’s doing too much. Your **Commands** class would grow into an uncontrollable mess, there’d be copied and pasted code left and right, dogs and cats living together, sheer chaos. At the risk of adding a few more classes, I’d rather get you off on the right foot.

In my opinion, the Commands class should identify the commands we want accessible within Autocad, instantiate an object or two responsible for doing whatever action we want done, and then get out of the way.

Here the guts of the CreateLayer() method are pretty straightforward. We instantiate a new object of type **ParcelLayer** using the **new ParcelLayer()** statement, and assign this newly created object to a variable named **creator**.

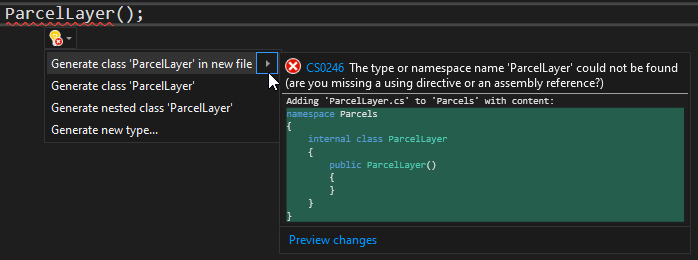
var creator = new ParcelLayer();

We then run a method belonging to that class, named **Create()**.

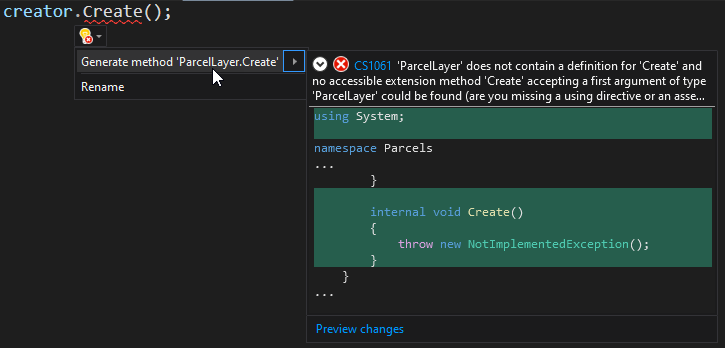
creator.Create();

At this point, no ParcelLayer class exists, hence the red squiggly line you see in VS. We need to add this class.

1. Hover your cursor over ParcelLayer, then press CTRL+. (period on your keyboard). Or use the little Quick Action icon that appears below the token.
2. Select **Generate class ‘ParcelLayer’** in new file.



1. Notice that a new file, **ParcelLayer.cs** appears in Solution Explorer.
2. Hover your cursor over **Create()**, the use the Quick Action icon and choose **Generate method ‘ParcelLayer.Create’**.



VS should open the **ParcelLayer** class file at this point, but if not, double-click it in the Solution Explorer window. Here’s the code you should see.

internal class ParcelLayer

{

public ParcelLayer()

{

}

public void Create()

{

throw new NotImplementedException();

}

}

The *internal* access modifier means that this class is visible only within the **Parcels** project, but not to other projects you may eventually add to the solution. For our purposes it doesn’t really matter, but in practice, I don’t use the *internal* modifier much.

Next we see an empty *constructor*.

public ParcelLayer()

{

}

A constructor is a special type of method, named the same as the class and with no return value (not even **void**). Constructors are used to start a new object in a specific state. Frequently we’ll use a constructor with parameters as a means of passing initial data or other objects into the object being created. If your constructor is truly empty, like it is here, you don’t need it at all as VS will just infer one. To keep my code clean, I would probably erase this, and only add a constructor back in when I need to do something more substantial, which we’ll get to later.

1. Within the **Create()** method, replace the throw new NotImplementedException(); line with the following code:

var layerName = "Parcels";

var document = Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocument;

var database = document.Database;

using(var transaction = database.TransactionManager.StartTransaction())

{

var layerTable = (LayerTable)transaction.GetObject(database.LayerTableId, OpenMode.ForRead);

LayerTableRecord layer;

if (layerTable.Has(layerName) == false)

{

layer = new LayerTableRecord

{

Name = layerName,

Color = Color.FromColorIndex(ColorMethod.ByAci, 161)

};

layerTable.UpgradeOpen();

layerTable.Add(layer);

transaction.AddNewlyCreatedDBObject(layer, true);

}

database.Clayer = layerTable[layerName];

transaction.Commit();

};

After typing this code, you’re probably going to have lots of red squigglies. Place your cursor in any of those spots in the code, then press CTRL+. to add in the missing using statements:

using Autodesk.AutoCAD.Colors;

using Autodesk.AutoCAD.DatabaseServices;

The keyword **using** is used a couple of different ways in C#. Here, at the beginning of a file, it is a *directive*, which allows you to use types from a namespace without having to fully qualify them. For example, on the line where we say:

Color = Color.FromColorIndex ...

Looks like this when fully qualified:

Color = Autodesk.AutoCAD.Colors.Color.FromColorIndex ...

That’s a lot to type, especially when we have to use it over and over again, so we give ourselves a nice little shortcut by making use of the using directive. We’ll talk more about **using** as a *statement* in a minute.

Lots to unpack in this method. First, we assign “Parcels” to a variable named **layerName**. We’ll use that in several places throughout the code.

Next, we grab the active document, and its Database. An Autocad DWG file is actually a big database which happens to display most of its data graphically. Any time we want to draw, or access layers, styles, etc., we’ll be working with the Database class.

Note that this is the second time we’ve needed to get to the MdiActiveDocument object, which is going to get VERY tedious. We’ll make this easier a little later, but I wanted to point it out now as I really hate stuff that makes my life more difficult, and typing

Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocument

more than once definitely qualifies.

Next, we see our old friend, **using**, but this time it is used as a *statement*. Many objects in C# use an *interface* called **IDisposable** which ensures that objects release important resources. For example, if you are writing to a file, you want to be sure that the file is properly closed following a write operation.

Within the **using** statement arguments, we start a new **Transaction**.

using(var transaction = database.TransactionManager.StartTransaction())

{

//do some stuff here

transaction.Commit();

};

A transaction is a critical mechanism Autocad uses to safeguard access to the database. It ensures that things get saved properly if everything goes well, in which case we can call **transaction.Commit()** at the very end of the using block. If something goes wrong, a **transaction.Abort()** statement is issued, and nothing that happened within the using block is persisted to the database.

Because you are going to use this A LOT when programming with Autocad, we’ll introduce a helper method later on to make working with transactions a lot easier. But for now, let’s continue with our explanation of the Create() method.

Typically, when you add a new layer in Autocad, you open the Layer Manager palette, which displays all the layers and their various settings (properties!). We can access the same information via the **LayerTable** object in the drawing’s Database. We access objects in the database via Transaction’s **GetObject()** method.

var layerTable = (LayerTable)transaction.GetObject(database.LayerTableId, OpenMode.ForRead);

Each item in the database has an **ObjectId** which we have to supply to GetObject(). Some items in the database are readily accessible via a special “named” property, like **LayerTableId**. Using the ObjectId for the LayerTable object, GetObject() retrieves that object from the database. We *cast* the returned object to its specific type by adding **(LayerTable)** in front of the transaction keyword. We could also do this by placing **as LayerTable** just after the closing parenthesis as shown below:

var layerTable = transaction.GetObject(database.LayerTableId, OpenMode.ForRead) as LayerTable;

And finally, we’re opening the LayerTable in read-only mode, which is recommended practice. We’ll change it to read-write mode later when we actually need to change something in the database.

LayerTableRecord layer;

Each layer in the LayerTable is represented by a LayerTableRecord, so we’ll setup a variable for this. We can’t use the **var** keyword here as we’re not assigning a value to it yet, so the compiler can’t infer what type the object will be.

When we run our command, it is possible that the “Parcels” layer already exists, in which case we don’t need to create it again. So we’ll check to see if the layer exists first.

if (layerTable.Has(layerName) == false)

{

layer = new LayerTableRecord

{

Name = layerName,

Color = Color.FromColorIndex(ColorMethod.ByAci, 161)

};

};

if…else… structures are highly important in programming. They give our programs the ability to make decisions, or *branch*. An if statement evaluates some *conditional* and if true, branches to do something. Otherwise (the conditional is false), it does something else.

Note the double equals == in the conditional. When we use a single = we’re saying, “Assign whatever’s on the right side to the variable on the left side.” But when we use == we’re asking “Is the thing on the right equal to the thing on the left?”

If the layer table does not have a layer matching **layerName** (i.e. “Parcels”), then we want to do something—create a new layer and set its color.

Here we’re using *object initialization syntax* to initialize the newly created layer object’s name and its color. There are many options for assigning colors, but if you’re sticking with the standard 256 color palette, you’ll typically use the **FromColorIndex()** method of the **Color** object:

Color.FromColorIndex(ColorMethod.ByAci, 161)

Which sets the color to 161.

Note that you could have written the code above like this as well:

layer = new LayerTableRecord();

layer.Name = layerName;

layer.Color = Color.FromColorIndex(ColorMethod.ByAci, 161);

Both do the same thing; I just prefer using object initializer syntax.

Also note that the else clause is optional, we don’t always have to do something else. Assuming the layer already exists, we don’t need to do anything.

This next part was very strange to me when starting to work with transactions. While we’ve created a new layer in *memory*, we haven’t actually added it to the LayerTable in our drawing. Before we can do this, we need to upgrade that object to read-write mode, which we do using the **UpgradeOpen()** method.

layerTable.UpgradeOpen();

We can then add the layer object to it:

layerTable.Add(layer);

And finally, we have to add the new object to our transaction object.

transaction.AddNewlyCreatedDBObject(layer, true);

After the end of the **if** block, we’ll set the database’s **CLayer** property (the current layer) to the ObjectId of the layer matching *layerName* (“Parcels”). **layerTable** represents a collection of LayerTableRecord objects, and we can access a specific object in a collection using square brackets **[ ]** with an index value (here, a string containing the layer’s name).

database.Clayer = layerTable[layerName];

And finally (critically!), we need to commit our transaction. Otherwise nothing is persisted.

transaction.Commit();

Follow the steps in Debugging in Autocad earlier in this document, using the command **PS\_CREATEPARCELLAYER** in step 6 to test your program. The “Parcels” layer should be added to your drawing and set current.

## Refactoring to the Active class

I mentioned before that there are some really common things we need access to when programming in Autocad, such as the MdiActiveDocument, Editor, and transactions. We can make our life a lot simpler by creating a “helper” class.

1. Add a new class file to your project named **Active.cs**.
2. Replace the default code with the following.

using Autodesk.AutoCAD.ApplicationServices;

using Autodesk.AutoCAD.DatabaseServices;

using Autodesk.AutoCAD.EditorInput;

using System;

namespace Parcels

{

public static class Active

{

public static Document Document => Application.DocumentManager.MdiActiveDocument;

public static Editor Editor => Document.Editor;

public static Database Database => Document.Database;

public static void UsingTransaction(Action<Transaction> action)

{

using (var transaction = Active.Database.TransactionManager.StartTransaction())

{

action(transaction);

transaction.Commit();

}

}

}

}

Note: going forward, I’ll omit the **namespace Parcels { }** portion of code for new classes or interfaces to save space.

This class has a different access modifier we haven’t seen yet: **static**. Static classes are kind of special in that you cannot instantiate them (via the **new** keyword), and they can only contain static methods and properties. Here, we identify three properties which give us direct, easy access to the current Document, its Editor, and its Database. So, wherever you need access to MdiActiveDocument, you can just do this:

var doc = Active.Document;

The **UsingTransaction()** method requires a little more explanation. Recall when we first discussed transactions that we’re always going to have the following “boilerplate” code (I’ve substituted our new static property **Active.Database** for the original *database* variable):

using(var transaction = Active.Database.TransactionManager.StartTransaction())

{

//do some stuff here

transaction.Commit();

};

If we could somehow substitute in “random” code in place of the comment **//do some stuff here**, it would make our lives quite a bit simpler. And we can do just that by passing in an **Action<Transaction>** argument to the **Active.UsingTransaction()** method. This method spins up a transaction, executes whatever code is passed in via the *action* argument, then commits and closes the transaction. We’ll see exactly how to call this in the next part of our lesson.

## Summarizing Parcels

One of the tasks we’re frequently asked to do in our imaginary parcel job is summarizing how many parcels are in a specific area, and what the combined area is. We’ll define a “parcel” as being a closed polyline, on the “Parcels” layer, and ignore all other polylines that might wind up in our selection set.

Let’s first setup the command in our Commands class.

[CommandMethod("PS\_CountParcels")]

public void CountParcels()

{

var cmd = new ParcelCounter();

var summary = cmd.Count();

Active.Editor.WriteMessage($"Found {summary} parcels.");

}

First we instantiate a ParcelCounter object and then call its Count() method. This method will return some sort of summary object which we assign to the **summary** variable.

We then use the Autocad Editor (via our handy-dandy Active class!) to write out the number of parcels found.

Create the ParcelCounter class as before by placing your cursor anywhere inside ParcelCounter, then using CTRL+. and choosing Declare Class.

1. Add the following code:

public int Count()

{

var count = 0;

var options = new PromptSelectionOptions();

options.MessageForAdding = "Add parcels";

options.MessageForRemoval = "Remove parcels";

var filter = new SelectionFilter(new TypedValue[]

{

new TypedValue((int)DxfCode.Start, "LWPOLYLINE"),

new TypedValue((int)DxfCode.LayerName, "Parcels")

});

var result = Active.Editor.GetSelection(options, filter);

if (result.Status == PromptStatus.OK)

{

Active.UsingTransaction(tr =>

{

foreach (var objectId in result.Value.GetObjectIds())

{

var polyline = (Polyline)tr.GetObject(objectId, OpenMode.ForRead);

if (polyline.Closed)

{

count++;

}

}

});

}

return count;

}

1. Use CTRL+. shortcut to add using statements for **Autodesk.AutoCAD.DatabaseServices** and **Autodesk.AutoCAD.EditorInput** to resolve a couple of red squigglies.

This is a rather lengthy method that essentially does two things. First, it sets up some selection options and filters, then prompts the user to select objects in the drawing. Then it uses a transaction to iterate through those objects to count them.

While it is very common to write code this way, your methods will quickly grow to be unruly messes with all kinds of responsibilities. Let’s refactor this by cutting all the code from **var options** to **var result** and placing them in their own method as shown below. Note that we’ll change **var result =** to **return**.

1. Add the following code:

private PromptSelectionResult SelectParcels()

{

var options = new PromptSelectionOptions();

options.MessageForAdding = "Add parcels";

options.MessageForRemoval = "Remove parcels";

var filter = new SelectionFilter(new TypedValue[]

{

new TypedValue((int)DxfCode.Start, "LWPOLYLINE"),

new TypedValue((int)DxfCode.LayerName, "Parcels")

});

return Active.Editor.GetSelection(options, filter);

}

1. Back in **Count()** add a line after **var count = 0;** as follows:

var result = SelectParcels();

Let’s discuss **SelectParcels()**.

private PromptSelectionResult SelectParcels()

This method is marked with the **private** access modifier, which means that it cannot be seen or accessed from outside this class. It returns a **PromptSelectionResult** which indicates whether the selection operation was successful or not (the user might not have selected anything, or hit ESC). If successful, we can also access the objects selected by the user.

The **Editor** object has several methods we can use to get a selection of objects, which all use a similar pattern. They all take some sort of Options object (like **PromptSelectionOptions**), and some also allow a **SelectionFilter** object. Here, the *options* variable sets up the prompts the user sees.

var options = new PromptSelectionOptions();

options.MessageForAdding = "Add parcels";

options.MessageForRemoval = "Remove parcels";

The *filter* variables sets up some filters so that only “valid” objects are selected.

var filter = new SelectionFilter(new TypedValue[]

{

new TypedValue((int)DxfCode.Start, "LWPOLYLINE"),

new TypedValue((int)DxfCode.LayerName, "Parcels")

});

A **SelectionFilter** uses an array of **TypedValue** objects. If you’ve ever done any LISP coding, TypedValue objects are essentially “dotted pairs” that you see when you use the following LISP command, and select a polyline:

(entget (car (entsel)))

((-1 . <Entity name: 299753b1500>) (0 . "LWPOLYLINE") (330 . <Entity name: 299753b71f0>) (5 . "210") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "Parcels") (100 . "AcDbPolyline") (90 . 3) (70 . 0) (43 . 0.0) (38 . 0.0) (39 . 0.0) (10 17.2384 8.79722) (40 . 0.0) (41 . 0.0) (42 . 0.0) (91 . 0) (10 28.128 12.8708) (40 . 0.0) (41 . 0.0) (42 . 0.0) (91 . 0) (10 32.9505 5.84313) (40 . 0.0) (41 . 0.0) (42 . 0.0) (91 . 0) (210 0.0 0.0 1.0))

Earlier, we decided that the definition of a “parcel” was a closed polyline on the “Parcels” layer. So, the two things we need to filter on are the object type (dotted pair 0) and the layer (dotted pair 8). Autocad’s library uses an **enum** named **DxfCode** to represent each of the DXFCODE values. While the following pairs of statements are equivalent, use of the DxfCode enum is preferred because it is more readable.

new TypedValue((int)DxfCode.Start, "LWPOLYLINE"),

new TypedValue((int)DxfCode.LayerName, "Parcels")

new TypedValue(0, "LWPOLYLINE"),

new TypedValue(8, "Parcels")

With our PromptSelectOptions and SelectionFilter objects ready to go, we can pass them into Editor’s **GetSelection()** method, and return the results.

return Active.Editor.GetSelection(options, filter);

We can now return to a discussion of the Count() method.

var result = SelectParcels();

if (result.Status == PromptStatus.OK)

{

We assign the return value of SelectParcels to the *result* variable. We then check to see if its **Status** equals **PromptStatus.OK**. PromptStatus is another enum from the Autocad library. It is important to do this check because the user might have cancelled out of the command, or made an invalid selection, in which case there’s no point in trying to do anything else.

Active.UsingTransaction(tr =>

{

Next, we use our **Active.UsingTransaction** method. The Action parameter requires a placeholder for a transaction, which we’ll identify using **tr**. The **=>** symbol is called a lambda operator. I read somewhere that this was called the “goes to” operator, which kind of stuck for me. So “tr (a transaction) goes to (the following code).” The code that we want to happen inside the transaction is as follows:

foreach (var objectId in result.Value.GetObjectIds())

{

var polyline = (Polyline)tr.GetObject(objectId, OpenMode.ForRead);

if (polyline.Closed)

{

count++;

}

}

**foreach** is a new type of construct for us. It allows us to *loop* over a collection of objects. In this case, a PromptSelectionResult object contains a **Value** property, from which we can use **GetObjectIds()** to get the ObjectIds of the objects the user selected.

We then use the Transaction’s **GetObject()** method on each *objectId*, casting the returned object to a Polyline and storing it in the *polyline* variable. At this point, we can check if the polyline is **Closed**. If so, increase the *count* variable using the **++** operator. ++ increments an integer variable by 1. You could also do this as follows:

count = count + 1;

Incidentally, we could have setup our SelectionFilter to only select Closed polylines. But that involved even more strange DxfCodes, and I wanted you to see how to accomplish this in code. Plus I think it’s just more readable.

Calling the Count() method gives us a count of selected parcels. The *caller* of this method, Commands.CountParcels(), receives an integer value back, and outputs a summary message to the command prompt, via Active.Editor.WriteMessage().

A simple count of objects seems kind of boring. We could do that via the Properties panel in Autocad. But what about the sum of the areas of all the parcels? If you try that in the Properties panel, you just get “\*Varies\*” which is not helpful. You could try using the AREA command, but that’s kind of a pain. Let’s refactor our Count method to provide more data about the parcels selected.

### Refactoring the Count() method

It would be nice if our method could provide not only a count of selected parcels, but also the total area of the selected parcels. Let’s create a class to store this information.

1. Add a new class to the Parcels project named **ParcelSummary**.
2. Add the following code:

public class ParcelSummary

{

public int Count { get; set; }

public double Area { get; set; }

}

This simple class contains two properties, Count and Area. Change the **ParcelCounter.Count()** method to return a ParcelSummary object, replacing the *count* (integer) variable with a *summary* (ParcelSummary) variable. Changes are highlighted below:

public ParcelSummary Count()

{

var summary = new ParcelSummary();

var result = SelectParcels();

if (result.Status == PromptStatus.OK)

{

Active.UsingTransaction(tr =>

{

foreach (var objectId in result.Value.GetObjectIds())

{

var polyline = (Polyline)tr.GetObject(objectId, OpenMode.ForRead);

if (polyline.Closed)

{

summary.Count++;

summary.Area += polyline.Area;

}

}

});

}

return summary;

}

Each time the program finds a closed polyline in the selected objects, the *summary* object’s Count property will be incremented, and its Area property will have the polyline’s area added to it. The **+=** operator adds a value to an existing value.

Unfortunately, our message results are going to be a little weird at this point, because C# won’t know what to display for a ParcelCounter object. We could override a built-in method called **ToString()** but I have bigger plans.

### Outputting results in different ways

While writing to the command prompt is one way for us to report information, it’s pretty limited. We might want to eventually write to a text file, to Excel, or to a database. But how can we do this in a way that is flexible?

First, we should probably come up with some sort of “writer” class that can take in a ParcelCounter object, and “write” the data in a certain way, and to a specific destination.

A good programming technique for classes that need to interact with one another is to program to *abstractions*, not to concrete implementations. Programming to **Editor.WriteMessage()** is an example of programming to a concrete implementation. There’s only one place the output can go (Autocad’s command prompt).

*Interfaces* are a type of abstraction used to define “contracts”—specifying method signatures and/or properties without providing an actual implementation. We can then write a class that implements an interface, meaning that it has to provide an implementation of the methods and properties identified by the interface.

1. Right-click the **Parcels** project, then choose **Add > New Item**.
2. Select **Interface** from the list of templates, then type the name **IMessageWriter**, then click **Add**.
3. Copy the following code into the IMessageWriter.cs file.

public interface IMessageWriter

{

void WriteMessage(string message);

}

Note that there is no executable code in an interface. The convention for naming interfaces in C# is to prefix the name with an I (an “eye” not a “one” or an “ell”). This contract specifies a single void method, named **WriteMessage()** which takes a string argument [the message we want “written” out].

Let’s add another interface called **IParcelSummaryWriter**.

1. Use steps 1-3 above, but name the file IParcelSummaryWriter.
2. Add the following code:

public interface IParcelSummaryWriter

{

void Write(ParcelSummary summary, IMessageWriter writer);

}

This interface defines a single method, Write, which requires a ParcelSummary object, and an IMessageWriter object.

We now need an implementation of each of our interfaces. Let’s start by implementing a simple IMessageWriter, which will write to the Autocad command prompt.

1. Add a new class named **AutocadMessageWriter** to the Parcels project.
2. Add the following code:

public class AutocadMessageWriter : IMessageWriter

{

public void WriteMessage(string message)

{

Active.Editor.WriteMessage(message);

}

}

Now let’s implement IParcelSummaryWriter.

1. Add a new class to the Parcels project named **ParcelSummaryWriter**.
2. Add the following code:

public class ParcelSummaryWriter : IParcelSummaryWriter

{

public void Write(ParcelSummary summary, IMessageWriter writer)

{

var message = $"Found {summary.Count} parcels.";

message += $"\nCombined area: {summary.Area:N2}";

writer.WriteMessage(message);

}

}

By programming to the *IMessageWriter* abstraction, the ParcelSummaryWriter class doesn’t need to know *where or how* the messages will be written to. It just knows that *messageWriter* must have a WriteMessage() method that it can call. A *message* is composed, and then sent to the *messageWriter*.

Let’s go back to the **Commands.CountParcels()** method, and hook all this together.

[CommandMethod("PS\_CountParcels")]

public void CountParcels()

{

...

//omitted for brevity

var writer = new AutocadMessageWriter();

var summarizer = new ParcelSummaryWriter();

summarizer.Write(summary, writer);

}

We instantiate an AutocadMessageWriter object, then a ParcelSummaryWriter object. Then we call the Write() method, passing in the *summary* object returned by ParcelCounter.Count(), and the *writer* object.

If you debug this, open a drawing with some closed polylines on the Parcels layer, and run PS\_CountParcels, you’ll see the summary output to the command prompt.

Let’s create another IMessageWriter, that writes to a text file.

1. Add a new class file to your project called **TextMessageWriter**.
2. Add the following code:

public class TextMessageWriter : IMessageWriter

{

readonly string \_filePath;

public TextMessageWriter(string filePath)

{

\_filePath = filePath;

}

public void WriteMessage(string message)

{

bool shouldAppend = true;

using (var streamWriter = new StreamWriter(\_filePath, shouldAppend))

{

streamWriter.WriteLine(message);

}

}

}

This class also implements IMessageWriter. Note that to instantiate this class, the caller needs to specify a file path where we want the file to be written. This is passed in via the class constructor, and stored in a **readonly** field for later use. The **readonly** modifier means that once the variable is assigned a value, it cannot be changed later.

This **WriteMessage** method uses a StreamWriter (part of the .NET framework) to write data to a file. Note that we’re choosing to *append* to the file using one of StreamWriter’s several constructor overloads.

How do we use this? Pretty simple, actually. Back in CountParcels(), make the following changes:

[CommandMethod("PS\_CountParcels")]

public void CountParcels()

{

var cmd = new ParcelCounter();

var summary = cmd.Count();

var acadWriter = new AutocadMessageWriter();

var myDocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

var filePath = Path.Combine(myDocuments, "ParcelSummary.txt");

var textWriter = new TextMessageWriter(filePath);

var summarizer = new ParcelSummaryWriter();

summarizer.Write(summary, acadWriter);

summarizer.Write(summary, textWriter);

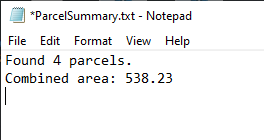
}

NOTE: Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) retrieves the path to the currently logged in user’s My Documents folder. Path.Combine combines portions of a file path (i.e. the path to My Documents with the file name “Parcels.txt”).

Because ParcelSummaryWriter.Write() relies on an *abstraction* (IMessageWriter) and not a concrete implementation, we can easily substitute the type of *writer* without having to change ParcelSummaryWriter. Notice how we call **summarizer.Write** twice, passing in a different IMessageWriter object (acadWriter, then txtWriter) each time.

At this point, you can debug and test your code. Use the **PS\_CREATEPARCELLAYER** command to create a Parcels layer in an empty drawing. Draw a few open and closed polylines on that layer. For good measure, add a few open and closed polylines on another layer, then use the **PS\_COUNTPARCELS** command to see the summary. Note that you should now get output to two different locations: the command prompt and to a text file in your My Documents folder!

Opening the file, we see something like the following:



While it makes sense to have a rather terse summary at the command prompt in Autocad, this TXT file is going to be a little hard to decipher if we use this tool on more than one drawing, or at different times. The beauty of having separate IMessageWriter implementations is that we can tailor the output in each implementation. Open **TextMessageWriter** and make the following changes to the WriteMessage method:

public void WriteMessage(string message)

{

bool shouldAppend = true;

using (var streamWriter = new StreamWriter(\_filePath, shouldAppend))

{

streamWriter.WriteLine($"{Active.Document.Name}");

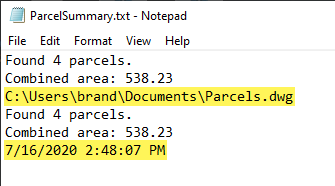
streamWriter.WriteLine(message);

streamWriter.WriteLine(DateTime.Now.ToString());

}

}

We’re now sandwiching the original message between the document’s name, and the date/time the command was run. Debug again and re-run the PS\_CountParcels command, then reopen the ParcelSummary.txt file when finished. You should see something like the following:



# Bonus Material (Time Permitting)

What if we now wanted to include an option for writing out to an HTML file for viewing on the Web? Turns out that most of the plumbing is already in place.

First, we need a new IParcelSummaryWriter. Add a new class and name it **HtmlParcelSummaryWriter**, with the following code. All we’re doing here is creating a long string in HTML format, with some of the data from our ParcelSummary object sprinkled in.

public class HtmlParcelSummaryWriter : IParcelSummaryWriter

{

public void Write(ParcelSummary summary, IMessageWriter writer)

{

var message =

$@"<!DOCTYPE HTML>

<html>

<head>

<title>Parcel Summary</title>

</head>

<body>

<h1>Parcel Summary for {Active.Document.Name}</h1>

<ul>

<li>Found {summary.Count} parcels.</li>

<li>Combined area: {summary.Area:N2}</li>

</ul>

</body>

</html>";

writer.WriteMessage(message);

}

}

We need to write out to a text file, similar to our TextMessageWriter. However, our HtmlParcelSummaryWriter formats the message exactly the way we need it to be a valid web page. Our TextMessageWriter sandwiches the message with a couple of extra things that will make the resulting Html file invalid. We could copy and paste all the code from TextMessageWriter into a new class, but DUPLICATE CODE IS BAD! Let’s see how we can leverage our existing code with an important Object-Oriented programming technique called *inheritance*.

Inheritance allows classes to inherit code from ancestors, similar to the way you inherited characteristics of your parents.

Open the **TestMessageWriter** class. Insert a new line above the very last bracket (the end of the namespace block), and add the following code.

public class HtmlMessageWriter : TextMessageWriter

{

public HtmlMessageWriter(string filePath) : base(filePath)

{ }

}

This class inherits from TextMessageWriter. Its constructors takes in a filePath, and passes that to TextMessageWriter’s constructor. Next, we need to modify TextMessageWriter. The WriteMessage method currently looks like this:

public void WriteMessage(string message)

{

bool shouldAppend = true;

using (var streamWriter = new StreamWriter(\_filePath, appendToFile))

{

streamWriter.WriteLine(Active.Document.Name);

streamWriter.WriteLine(message);

streamWriter.WriteLine(DateTime.Now.ToString());

}

}

The three lines I’ve highlighted are a problem. First, appending to the file is fine for TextMessageWriter, which acts as a logging system. But it’s not so good for our HtmlMessageWriter which will simply rewrite the file every time. The other two lines prepend and append additional messages, which we want in the case of our text file, but don’t want for our Html file. Let’s take care of the file appending issue first.

We can move the **shouldAppend** variable to a field in the class. If we then mark it as *protected* instead of *private*, any descendants of TextMessageWriter will have access to it. We’ll set this variable to **true** or **false** in the different constructors. In TextMessageWriter, replace the previous constructor with the following code:

protected bool \_shouldAppend;

public TextMessageWriter(string filePath)

{

\_filePath = filePath;

\_shouldAppend = true;

}

Change the constructor of HtmlMessageWriter as follows:

public HtmlMessageWriter(string filePath) : base(filePath)

{

\_shouldAppend = false;

}

Next, we’ll modify TextMessageWriter.WriteMessage to use the new \_shouldAppend field, and also to call some “hooks.” Hooks are methods that we can override in descendant classes to provide modifiable behavior. Replace the WriteMessage method with the following code, including two new methods, Before and After:

public void WriteMessage(string message)

{

using (var streamWriter = new StreamWriter(\_filePath, \_shouldAppend))

{

Before(streamWriter);

streamWriter.WriteLine(message);

After(streamWriter);

}

}

protected virtual void Before(StreamWriter streamWriter)

{

streamWriter.WriteLine(Active.Document.Name);

}

protected virtual void After(StreamWriter streamWriter)

{

streamWriter.WriteLine(DateTime.Now.ToString());

}

When WriteMessage is called, it sets up a StreamWriter to write to a text file, then calls Before, passing along the same StreamWriter object, then writes the message, then calls After, again passing along the same StreamWriter object. The keyword *protected* makes the methods visible to descendents, and the keyword *virtual* makes them overridable. So we can change the behavior of these two methods in any descendant classes.

In HtmlMessageWriter, add the following code:

protected override void Before(StreamWriter streamWriter)

{ }

protected override void After(StreamWriter streamWriter)

{ }

These methods now override (or replace) the default behavior found in TextMessageWriter. In this case, the methods are empty—we don’t want to do anything in the Before or After methods when writing to an Html file.

Let’s go wire this up in our CountParcels method and run it! In the Commands class, change CountParcels to the following:

[CommandMethod("PS\_CountParcels")]

public void CountParcels()

{

var myDocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

var filePath = Path.Combine(myDocuments, "ParcelSummary.txt");

var acadWriter = new AutocadMessageWriter();

var textWriter = new TextMessageWriter(filePath);

var htmlWriter = new HtmlMessageWriter(Path.Combine(myDocuments, "ParcelSummary.html"));

var cmd = new ParcelCounter();

var summary = cmd.Count();

IParcelSummaryWriter summarizer = new ParcelSummaryWriter();

summarizer.Write(summary, acadWriter);

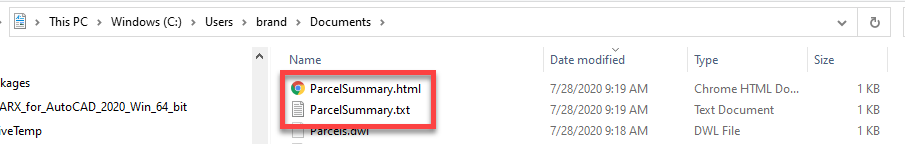
summarizer.Write(summary, textWriter);

summarizer = new HtmlParcelSummaryWriter();

summarizer.Write(summary, htmlWriter);

}

Note that we’re instantiating an HtmlMessageWriter, as well as an HtmlParcelSummaryWriter. Note that because *summarizer* is declared as an *IParcelSummaryWriter*, we can reuse the same variable to represent both a ParcelSummaryWriter and an HtmlParcelSummaryWriter object at different lines in the code. Debug the code and run PS\_CountParcels again. If you look in your My Documents folder, you should see the following:



Open up the ParcelSummary.html file and you’ll see a very basic web page that looks like this:



So with a little bit of inheritance thrown into the mix, we now get two completely different text file outputs, one as a TXT file, the other as HTML!

# Summary

In summary, we learned how to create a plugin for Autocad, using Visual Studio and the C# language. We saw how to add references to key Autocad DLLs so that we could program to the Autocad API.

We learned about namespaces, classes, properties and methods. We learned some programming constructs like if/else and foreach to control the flow of our program.

Within the Autocad API, we saw how to use the CommandMethod attribute to flag certain methods for Autocad so that we could use those commands at the command prompt, just like native Autocad commands. We learned how to use Transactions to query information from a drawing database, and how to add objects (like new layers) to the database. We created a special helper class, Active, to simplify access to the active Document, Database and Editor. We also created a boilerplate helper method, UsingTransaction, which takes in arbitrary code as an Action parameter, and does all the necessary Transaction setup and Commit code for us. This kind of method greatly simplifies very common code we would otherwise have to write over and over again.

We learned the importance of splitting code up into separate, focused classes and methods to keep our code readable, and easy to maintain. We learned about coding to abstractions instead of concrete implementations. This leads to more flexible, adaptable code.

I hope you’ve enjoyed this class and will start experimenting on your own! Please feel free to reach out if you have questions. You can reach me via email at **ben@leadensky.com**.

# Resources

Some good resources for your continued learning include the following:

Autocad .NET forum: <https://forums.autodesk.com/t5/net/bd-p/152>

AutoCAD VisualBasic forum: <https://forums.autodesk.com/t5/visual-basic-customization/bd-p/33>

Civil 3D Customization forum: <https://forums.autodesk.com/t5/civil-3d-customization/bd-p/190>

Revit API forum: <https://forums.autodesk.com/t5/revit-api-forum/bd-p/160>

Autodesk University Software Development courses: <https://www.autodesk.com/autodesk-university/au-online?query=%22Software%20Development%22>

My AU courses on Clean Code:

<https://www.autodesk.com/autodesk-university/class/Clean-Code-Tips-Writing-Clear-and-Concise-Code-2018>

<https://www.autodesk.com/autodesk-university/class/Clean-Code-2-More-Tips-Writing-Clear-and-Concise-Code-2019>

And my AU course on using Entity Framework with Autocad:

<https://www.autodesk.com/autodesk-university/class/Exploring-Entity-Framework-AutoCAD-Development-And-Beyond-2015>