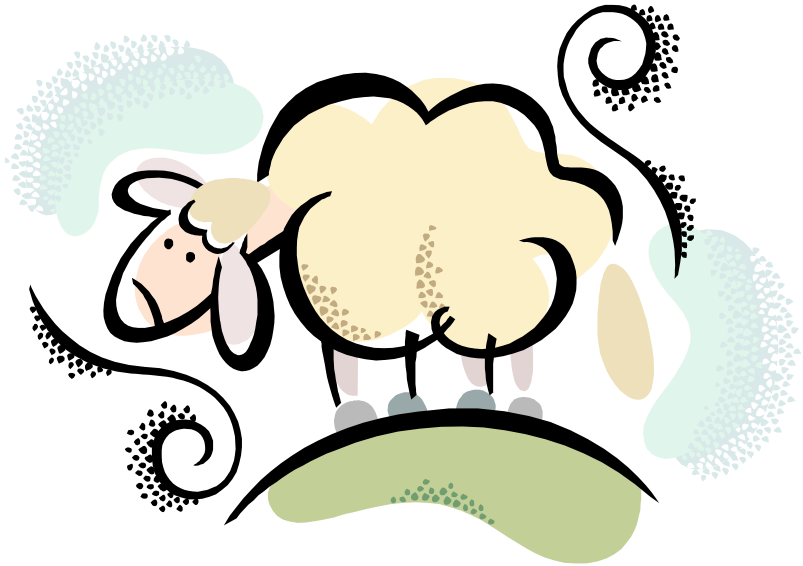


C# Programming I:

Concepts of Object-Oriented Programming

Legal Stuff

This presentation as well as other presentations and documents found on the sheepsqueezers.com website may contain quoted material from outside sources such as books, articles and websites. It is our intention to diligently reference all outside sources. Occasionally, though, a reference may be missed. No copyright infringement whatsoever is intended, and all outside source materials are copyright of their respective author(s).

.NET Lecture Series

C# Programming I: Concepts of OOP	*C# Programming II: Beginning C#*	*C# Programming III: Advanced C#*

C# Programming IV-1: System Namespace	*C# Programming IV-2: System.Collections Namespace*	*C# Programming IV-3: System.Collections. Generic Namespace*	*C# Programming IV-4A: System.Data Namespace*	*C# Programming IV-4B: System.Data.Odbc Namespace*
C# Programming IV-4C: System.Data.OleDb Namespace	*C# Programming IV-4D: Oracle.DataAccess.Client Namespace*	*C# Programming IV-4E: System.Data.SqlClient Namespace*	*C# Programming IV-4F: System.Data.SqlTypes Namespace*	*C# Programming IV-5: System.Drawing/(2D) Namespace*
C# Programming IV-6: System.IO Namespace	*C# Programming IV-7: System.Numerics*	*C# Programming IV-8: System.Text and System.Text. RegularExpressions Namespaces*	*C# Programming V: Introduction to LINQ*	

C# Self-Inflicted Project #1 Address Cleaning	*C# Self-Inflicted Project #2 Large Intersection Problem*

Charting Our Course

- What is Object-Oriented Programming (OOP)?
- What is an Object? What is a Class?
- What is Encapsulation?
- What is Inheritance?
- What is Polymorphism?
- What is an Interface?
- What is a Constructor?
- Real-World Example?

What is Object-Oriented Programming (OOP)?

On a day-to day basis, most programmers create procedures and/or functions in order to complete the tasks assigned to them. This type of programming is called *procedural programming* and has been around since the late 1960s. Procedural programming is one of many *programming paradigms* that includes *structured programming*, *imperative programming*, *functional programming*, *declarative programming*, and so on, as well as *object-oriented programming*, the subject of this presentation. These programming paradigms enforce (or, maybe, suggest) how the programmer approach a particular programming problem. Not all programming tasks should be approached using the familiar procedural programming paradigm: breaking up a programming task into several subroutines/functions and then calling them one after the other in a MAIN routine, say.

In *Object-Oriented Programming* (OOP), the programmer does not, at first, think about all of the subroutines, functions and variables involved in completing the programming task, but about the *things* being acting upon *by those subroutines, functions and variables*. In effect, you are thinking at a 30,000 foot level overlooking the problem rather than looking directly at the problem through a microscope. To me, this is exciting because it is another way to think about a programming problem rather than *same-old same-old*. OOP has been around since the 1960, but don't hold that against it! ☺

During this presentation, we will refer to the lowly automobile several times during our journey through OOP Land because everyone is familiar with the automobile, so there should be no misunderstandings (unlike if I talked about, say, HVAC).

What is an Object? What is a Class?

On the previous slide, I talked about *things* being acted upon by those subroutines, functions and variables. In OOP Land, these *things* are called *objects*. An *object* is, for example, a person, a place, a Windows Dialog box, a concept, etc.

For example, when you last went to purchase a car, what features did you want that car to have? Two-doors or four-doors? Four- or Six-cylinders? Blue, Brown, Green, or other color? In OOP Land, these features are called *attributes* (or *properties*) and replace the lowly variable of yore, at least, in name only. They serve the same purpose as variables, though, as holders of specific information about your car: `NumberOfDoors`, `NumberOfCylinders`, `ExteriorColor`, and so on.

Did you want your car's engine to start? Did you want the air conditioner to turn on? Did you want the radio to play? These actions are called *methods* and are analogous to subroutines and functions. These methods perform some action on your car such as `Engine(on|off)`, `AirCon(on|off)`, `Radio(on|off)`, etc.

In OOP Land, you define your object, programmatically, using a *class*. A *class* is the programmatic definition of your object and houses all of the attributes and methods that make up your object. Now, there are several different ways to *picture* a class and it will remind some of you of a database Entity-Relationship (ER) diagram. That is, a class is a rectangle with its attributes above a line and methods below the line. An example is on the next slide.

What is an Object? What is a Class?

Automobile

```
NumberOfDoors
NumberOfCylinders
ExteriorColor
------------------
Engine(true|false),
AirCon(true|false)
Radio(true|false)
```

An Automobile *class*, in diagram form, is made up of three attributes, appearing above the line, and three methods appearing below the line.

```
public class Automobile {

 //attributes
 public Int32 NumberOfDoors;
 public Int32 NumberOfCylinders;
 public Int32 NumberOfColors;

 //methods
 public Boolean Engine(Boolean bToggle) {
  //code to start or stop the engine
 }
 public Boolean AirCon(Boolean bToggle) {
  //code to start or stop the AC
 }
 public Boolean Radio(Boolean bToggle) {
  //code to start or stop the radio
 }

}
```

Here, the Automobile class has been taken from diagram-to-C# code.

What is an Object? What is a Class?

As mentioned above, a *class* is the programmatic definition of your object and houses all of the attributes and methods that make up your object. But, a class does NOT represent an object like a car that you, the programmer, can act upon. A class is the *definition* of an object. As you saw on the last slide, the code for the class just *defines* the object's attributes and methods, but does not give the programmer any real thingy to work with. It is the act of *instantiating the class* that creates a programmable thingy. That is, an *instance of a class* is an object that the programmer can act upon in code.

In order to create one or more objects, you can use code similar to the following:

```
Automobile auto1 = new Automobile();
Automobile auto2 = new Automobile();
```

At this point, you have two `Automobile` objects, `auto1` and `auto2`. If you want to start their engines, you can call the `Engine()` method:

```
auto1.Engine(true);
auto2.Engine(true);
```

What is Encapsulation?

As you saw in the code on the previous slides, I used the `public` keyword to define my Automobile class's attributes and methods. This means that anyone can turn the engine on and off as well as change the color of the car. Liken this to anyone being able to obtain your salary information or social security number. Not good!! In OOP Land, *encapsulation* refers to the ability to restrict access to the attributes and methods of a class. Information, such as salary, can be set as `private` so that only code internal to the class can access that information.

There are several *access modifiers* available in C#:

1. `public` – this indicates that an attribute or method is accessible by anyone who has access to an instantiated object.
2. `private` – this indicates that an attribute or method is accessible only by code defined within the class itself. Even if you have access to an instantiated object, you cannot access private attributes or methods.
3. `protected` – this is a combination of `public` and `private`. That is, a `protected` attribute or method can be accessed from within the class itself, or from a class *derived from your class*. We will talk more about this later.

Now, let's change our class to have private attributes:

```
public class Automobile {

 //attributes
 private Int32 NumberOfDoors;
 private Int32 NumberOfCylinders;
 private Int32 NumberOfColors;
```

Now, these three attributes can only be accessed from within the class code.

What is Inheritance?

On the previous slides, we discussed our Automobile class. There may be times when the class you are trying to define seems to work better defined as two separate classes working together. So far, we've been talking about cars, but what happens if we throw *trucks* into the mix? Are cars and trucks the same *thing*? You might respond: some things are the same and some things are not. For example, a car and a truck both have a number of cylinders, and a number of doors, but only a truck has Weight Class such as Light Duty, Medium Duty and Heavy Duty. This leads us to defining two classes: the first class contains all of the attributes common to both a car and truck; the second class – the `Truck` class – is *derived from* the first class, but includes additional attributes such as `WeightClass`. Let's see how this works:

```
public class Vehicle {

 //attributes
 public Int32 NumberOfDoors;
 public Int32 NumberOfCylinders;
 public Int32 NumberOfColors;

 //methods
 public Boolean Engine(Boolean bToggle) {
  //code to start or stop the engine
 }
 public Boolean AirCon(Boolean bToggle) {
  //code to start or stop the AC
 }
 public Boolean Radio(Boolean bToggle) {
  //code to start or stop the radio
 }

}
```

```
public class Truck : Vehicle {

 //additional attribute(s)
 public Int32 WeightClass;

}
```

As you see above, our `Truck` class is *derived from* the `Vehicle` class (indicated by the colon) and we go on to define an additional attribute called `WeightClass`. All of the attributes and methods are available from within the `Truck` class. You do not need to instantiate using the class `Vehicle`, just `Truck`. The `Vehicle` class is referred to as the *base class*.

What is Polymorphism?

In the great tradition of sports car racing, the following phrase can usually be heard before a race:

Gentlemen, start your engines!

But, depending on the car, this phrase should actually be something like this:

Gentlemen, for those of you with a modern car, push the button to start your engine. But, for those of you with slightly older cars, insert your key and turn to start your engine. For those of you who have vintage cars, get out of your vehicle and turn the crank to start your engine!

Clearly, the second phrase is a lot less exciting than the first – I can imagine most of the spectators walking away by the end of it – but it does bring us to our next OOP topic: *polymorphism*. Using procedural programming, you would have to code each one of the three ways to start an engine: `PushButton()`, `TurnKey()`, `TurnCrank()`. Then, you would have to call the correct procedure depending on which car you are starting.

But, referring to the first phrase, the announcer assumes that the car's owner knows how to start its engine without being explicitly told. In the object-oriented programming paradigm, you would only ever need to call a `StartEngine()` method to start the car's engine. But, how would this method know which procedure to perform to start the engine? This is where *polymorphism* comes into play. The `StartEngine()` method would take a single parameter: a car object. But, depending on whether that car object is a vintage car, a modern car or a very modern car, the `StartEngine()` method would execute the correct car starting procedure.

What is Polymorphism?

In terms of coding, you would have three classes: `VintageCar`, `NotSoModernCar` and `ModernCar`. Each of these three classes can use the `Vehicle` class as a base class. And each of these three classes will implement their very own `StartEngine()` method. It is the responsibility of .NET to execute the correct method based on the object supplied to it. Let's see that in code.

```
public class Vehicle {

 //attributes
 private Int32 NumberOfDoors;
 private Int32 NumberOfCylinders;
 private Int32 NumberOfColors;

 public virtual Boolean StartEngine(Boolean bToggle) {
  //virtual method to start or stop the engine
 }

}

public class VintageCar : Vehicle {

 public override Boolean StartEngine(Boolean bToggle) {
  //overridden code to start or stop a Vintage engine
 }

}
```

Note that the keyword *virtual* indicates that the derived class has the option to override the base class's method (indicated by the keyword *override*).

…continued on next slide…

What is Polymorphism?


```
public class NotSoModernCar : Vehicle {

 public override Boolean StartEngine(Boolean bToggle) {
  //overridden code to start or stop a not so modern engine
 }

}

public class ModernCar : Vehicle {

 public override Boolean StartEngine(Boolean bToggle) {
  //overridden code to start or stop a modern engine
 }

}
```

With these three classes defined, each deriving from the `Vehicle` class and overriding its `StartEngine()` method, let's create three cars all lined up and ready to start their engines!

```
//Create three cars (zero-based array gives three entries: 0, 1 and 2)
public Vehicle[] cars = new Vechicle[2];

//Define each of the three car objects
cars[0] = new VintageCar(); //turn crank to start engine
cars[1] = new NotSoModernCar(); //turn key to start engine
cars[2] = new ModernCar(); //push button to start engine

//Start the engines
for (int i=0; i<3; i++) {
 cars[i].StartEngine(true); //Since each car object is defined using a different class,
                            //the appropriate StartEngine() method will be called.
}
```

What is Polymorphism?

Note that you do not need to create a class with a virtual method and then override it in derived classes in order to use polymorphism. You can just create several methods all with the same name in your class AS LONG AS your parameters are different, either in number of data type. When you call the method, it is the parameters/datatypes (together called the *signature*) which determine which method is called.

What is an Interface?

As we've seen on the previous slides, we can derived a new class from an existing base class. For example, we derived the `VintageCar` class from the `Vehicle` class. You might want to know if you can derive from two or more base classes when creating your derived class. The answer is Yes, and yet at the same time, No. The reason is that some object-oriented languages allow for *multiple inheritance* and some allow for *single inheritance*. C# allows for single inheritance only, whereas C++ allows for multiple inheritance. So, the answer is No.

But, not to worry, you can fake it out using *interfaces*. So, the answer is Yes. An interface defines the attributes that are available and the methods that can be executed, but contains no code to execute other than the code used to create the definitions. (This is different, for the most part, from how you define a class!) Once your class *implements the interface*, there is a contractual arrangement between the two and you are guaranteed that the class actual does implement the interface because the compiler ensures that it does.

You may think that this sounds strange and is esoteric. In fact, interfaces are used quite a lot in .NET. For example, when you create an array using the `ArrayList` class, you also get the `IEnumerable` interface thrown in for free. The `IEnumerable` interface exposes the `GetEnumerator()` method which allows you to iterate through an array using the `foreach` syntax. Not Earth-shattering, but useful.

You can have one or more interfaces implemented by your class along with a base class. You do not have to have a base class, though, in order to implement one or more interfaces.

What is an Interface?

As an example, let's create two interfaces: `ISpeed` and `IManeuver`:

```
interface ISpeed {
 Boolean ChangeSpeedTo(Int32 iSpeed);
}

interface IManeuver {
 Boolean Turn(Int32 iDegrees);
}
```

Next, let's implement these two interfaces in our `Vehicle` class:

```
public class Vehicle : ISpeed, IManeuver {

 //attributes
 private Int32 NumberOfDoors;
 private Int32 NumberOfCylinders;
 private Int32 NumberOfColors;

 public virtual Boolean StartEngine(Boolean bToggle) {
  //virtual method to start or stop the engine
 }

 public Boolean ChangeSpeed(Int32 iSpeed) {
  //code changing of vehicle's speed here!
 }

 public Boolean Turn(Int32 iDegrees) {
  //code turning of vehicle here!
 }

}
```

What is a Constructor?

One nice feature is the ability to set your attributes when you create a new object. Think: initialization. In order to do this, you need to create a constructor; that is, a method with the same name as the class which accepts incoming parameters used to set your attributes within the class itself. Here is our `Vehicle` class with a constructor used to set the three attributes:

```
public class Vehicle : ISpeed, IManeuver {

 //attributes
 private Int32 NumberOfDoors;
 private Int32 NumberOfCylinders;
 private Int32 NumberOfColors;

 public Vehicle(Int32 pNbrDoor,Int32 pNbrCyl,Int32 pNbrClr) {
  NumberofDoors = pNbrDoor;
  NumberOfCylinders = pNbrCyl;
  NumberOfColors = pNbrClr;
 }

 //…additional code removed for clarity…
}
```

Here's how you would use the constructor when you create a new object:

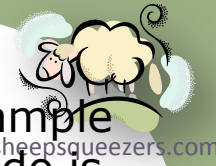
What is a Constructor?


```
Vehicle car = new Vehicle(4,6,254);
```

In the code above, we are creating an instance of the `Vehicle` class, called `car`, and initializing the number of doors to 4, the number cylinders to 6 and the color to 254 (whatever that might mean).

There's a lot more to OOP than what we've talked about so far. The number of rules you have to know as to when you can and cannot do something is daunting. See Microsoft's MSDN website for more information about the .NET platform.

Real-World Example?



Talking about cars and trucks is great, but I'd like to create a real-world example that can be used throughout this series of C# lectures. So, what I'd like to do is create an address cleaning program. Not only will it help us learn C#, but it will be (hopefully) useful when it's done.

Our class will be called `AddressClean` and its functionality should include the following:

- Read data in from a variety of sources (text files, databases, etc.)
- Clean the addresses and separate them into individual fields such as house number, street name, street type(street, drive, boulevard, etc.), apartment number, suite number, etc.
- Create a final cleaned address
- Write out a final cleaned address file (to a text file, database, etc.)

See the *Self-Inflicted C# Project #1: Address Cleaning* in the Documents section of the Sheepsqueezers website (www.sheepsqueezers.com).

Real-World Example?

Here is a first crack at our address cleaning class. As I learn more, I'll modify this class during the course of this lecture series.

```
class AddressClean {

 //Constructor
 public AddressClean() {
  //initialize properties, etc.
 }

 //Read in the data from a variety of sources.
 //Attempt to use polymorphism here so that there
 //is only one ReadData method.
 public Boolean ReadData(text file object???) {
  //this method reads from text files
 }
 public Boolean ReadData(database object???) {
  //this method reads from a database table
 }

 //Clean the addresses
 public Boolean CleanAddress() {
  //code to clean addresses
 }

 //Clean the cities
 public Boolean CleanCity() {
  //code to clean cities
 }

 private Boolean ReadZipCodes() {
  //this method reads in a list of "master" zipcodes from somewhere...have no clue yet!
 }
```

…continued on next slide…

Real-World Example?


```
//Clean the zipcode
public Boolean CleanZipcode() {

 //Read in master zipcodes first
 ReadZipCodes()

 //code to clean zipcodes based on the master zipcodes

}

//Create final cleaned address field
public Boolean CreateCleanAddress() {
 //code to create one field containing the cleaned standardized address.
 //This field excludes the city and state and zip.
}

//Write out final data to file
public Boolean WriteData(text object???) {
 //this method writes to text file
}

//Write out final data to database table
public Boolean WriteData(database object????) {
 //this method writes to database table
}

}
```

References

Click the book titles below to read more about these books on Amazon.com's website.

- Introducing Microsoft LINQ, Paolo Pialorsi and Marco Russo, Microsoft Press, ISBN:9780735623910
- LINQ Pocket Reference, Joseph Albahari and Ben Albahari, O'Reilly Press, ISBN:9780596519247
- Inside C#, Tom Archer and Andrew Whitechapel, Microsoft Press, ISBN:0735616485
- C# 4.0 In a Nutshell, O'Reilly Press, Joseph Albahari and Ben Albahari, ISBN:9780596800956
- The Object Primer, Scott W. Ambler, Cambridge Press, ISBN:0521540186
- CLR via C#, Jeffrey Richter, Microsoft Press, ISBN:9780735621633

Support sheepsqueezers.com

If you found this information helpful, please consider supporting sheepsqueezers.com. There are several ways to support our site:

- ☐ Buy me a cup of coffee by clicking on the following link and donate to my PayPal account: Buy Me A Cup Of Coffee?.
- ☐ Visit my Amazon.com Wish list at the following link and purchase an item: http://amzn.com/w/3OBK1K4EIWIR6

Please let me know if this document was useful by e-mailing me at `comments@sheepsqueezers.com`.