



COMPSCI 280 S1 2011 Enterprise Software Development

Programming Fundamentals
Objects



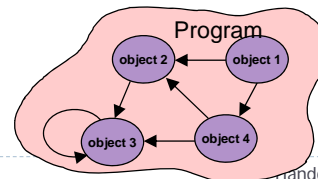
Agenda & Reading

- ▶ **Agenda:**
 - ▶ Understanding Object-Oriented Programming
 - ▶ Classes
 - ▶ Objects
 - ▶ Fields & Properties
 - ▶ Constructors & Destructors
 - ▶ Methods
 - ▶ Passing mechanism
 - ▶ Parameters Arrays
 - ▶ Overloading Methods
 - ▶ Static Classes, Fields & Methods
 - ▶ Structs Vs Classes
- ▶ **Recommended Reading:**
 - ▶ <http://msdn.microsoft.com/en-us/library/dd460654.aspx>
 - ▶ Microsoft Visual C# 2008 Step by Step
 - ▶ Chapter 3, 7, 8, & 9
- ▶ **Hands-On Lab:**
 - ▶ Lecture08Lab



Object-Oriented Programming

- ▶ Object-oriented development assumes that a system is a collection of objects that interact to accomplish tasks.
 - ▶ An object is a thing that has attributes and behaviors
 - ▶ Objects interact by sending messages to each other, asking another object to invoke, or carry out, one of its methods
- ▶ A problem is viewed as a collection of interacting objects with certain features, or attributes, and certain behaviours
- ▶ **Object-Oriented Modeling**
 - ▶ Classes are made of fields, properties, methods, and events
 - ▶ Fields and properties represent information that an object contains.
 - ▶ Methods represent actions that an object can perform.



Classes

```
public class MyPoint
{
}

```

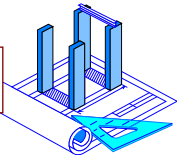
- ▶ The class defines what all objects of the class represent:
 - ▶ The data contained in the object
 - ▶ Usually implemented as fields (variables) or properties
 - ▶ The behavior (methods) and processing
 - ▶ Usually implemented as methods
- ▶ Classes serve as templates or blueprints for creating objects
- ▶ Examples:

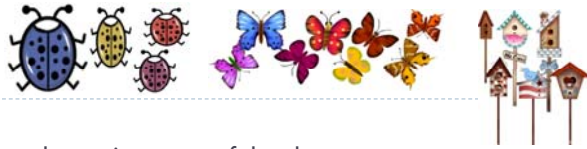
Class Name: Automobile
Data:
 amount of fuel _____
 speed _____
 license plate _____
Methods (actions):
 increaseSpeed:
How: Press on gas pedal.
 stop:
How: Press on brake pedal.

First Instantiation:
Object name: patsCar
amount of fuel: 10 gallons
speed: 55 miles per hour
license plate: "135 XJK"

Second Instantiation:
Object name: suesCar
amount of fuel: 14 gallons
speed: 0 miles per hour
license plate: "SUES CAR"

Objects that are instantiations of the class





▶ Objects

- ▶ Objects can be referred to as instances of the class
- ▶ When an object is created for the class, it is common to say the class is instantiated
- ▶ There may be many objects (instances) of a particular type (class)
- ▶ All objects of a given class contain different values for their properties and fields
- ▶ Objects send and receive messages to invoke actions
 - ▶ Invoking a method means to call the method, i.e. execute the method

▶ Declaring an Object

```
MyPoint p;
```

- ▶ Note: This declaration only declares the object, but it does not create an instance of the object. It has a value of Nothing.

▶ Creating an Object

```
p = new MyPoint(10,20);
```

- ▶ Use the new keyword to create the object
 - ▶ Allocates memory for the new object, calls the constructor and initializes the values of the new object
 - ▶ Returns a reference to the object it created

```
MyPoint p2 = new MyPoint(20,30);
```

```
p1.x = 100;
Console.WriteLine(p1.x);
```

▶ Using an Object

- ▶ Properties/methods defined in a class are now accessed through instances of the class.
 - ▶ You can use the dot operator to manipulate or inspect its variables/properties or invoke its methods.

▶ Cleaning up Unused Objects

- ▶ The environment deletes objects when it determines that they are no longer being used (i.e. no more references to that object). This process is called garbage collection.
- ▶ The garbage collector periodically frees the memory used by objects that are no longer referenced.

▶ We can also write our own classes that define specific objects that we need

- ▶ You need to analyze the characteristics and behaviors that your object needs

▶ A class definition has two parts: a class declaration and a class body.

▶ Class declaration

- ▶ Example:

```
public class MyPoint
{
}
```

▶ Class body

- ▶ It contains
 - ▶ constructors for initializing new objects,
 - ▶ declarations for the variables that provide the state,
 - ▶ properties for the variables that provide more control on the state &
 - ▶ methods to implement the behavior of the class and its objects.

Attributes represent the internal "state" of a given instance of this class.

▶ Fields

- ▶ Fields store the data a class needs to fulfill its design
- ▶ Fields should be declared as Private.

```
class MyPoint {
    private int _x;
    private int _y;
}
```

▶ Properties

- ▶ Properties are retrieved and set like fields, but are implemented using property **Get** and property **Set methods**, which provide more control on how values are set or returned.
 - ▶ It helps isolate your data and allows you to validate values before they are assigned or retrieved.
 - ▶ Property has Public access
 - ▶ The **value** keyword is used to define the value being assigned by the set method.
 - ▶ Property can be read-only (with the Get portion only) or write-only (with the Set portion only)

```
public int X {
    get {
        return _x;
    }
    set {
        if (value > 0)
            _x = value;
    }
}
```

```
public int Y {
    get {
        return _y;
    }
}
```

Read-only property



Fields Vs Property

- ▶ **Use property when:**
 - ▶ You need to control when and how a value is set or retrieved.
 - ▶ The property has a well-defined set of values that need to be validated.
 - ▶ Setting the value causes some perceptible change in the object's state, such as an `IsVisible` property.
 - ▶ Setting the property causes changes to other internal variables or to the values of other properties.
 - ▶ A set of steps must be performed before the property can be set or retrieved.
- ▶ **Use field when:**
 - ▶ The value is of a self-validating type.
 - ▶ Any value in the range supported by the data type is valid. This is true of many properties of type `Single` or `Double`.
 - ▶ The property is a `String` data type, and there is no constraint on the size or value of the string.



Constructors & Destructor

Default Constructor

- ▶ **Constructors**
 - ▶ Constructors control the creation of objects.
 - ▶ An instance of a class, an object, is created by using the `new` keyword. The code in the constructor is executed.
 - ▶ Constructor is an ideal location for any initialization tasks setting the initial values of variables
 - ▶ The system implicitly creates a default constructor at run time if you do not explicitly define a constructor for a class.
 - ▶ You can define parameterized constructors, specify the names and data types of arguments
- ▶ **Destructors**
 - ▶ Destructors control the destruction of objects
 - ▶ We ignore destructors and rely on .NET garbage collection
 - ▶ Not guaranteed to be called at a specific time, but guaranteed to be called before shutdown
 - ▶ Objects are destroyed through a garbage collection mechanism
 - ▶ Destructors cannot be called. They are invoked automatically.
 - ▶ Empty destructors should not be used for performance reason. i.e. Don't need to write an empty destructor

```
public MyPoint() {
    ...
}
```

```
public MyPoint(int newX, int newY) {
    ...
}
```

```
~MyPoint() // destructor
{
    // cleanup statements...
}
```



Methods

```
public string GetStatus() {
    ...
    return ...;
}
```

- ▶ Methods are made up of a block of code that performs one specific action
- ▶ Methods can affect the values of properties
- ▶ A method definition has two parts: a method declaration and a method body
 - ▶ Methods are declared within a class by specifying the access level, the return value, the name of the method, and any method parameters.
 - ▶ The method parameter specifies the type and name of each parameter
 - ▶ The return type indicates the type of value that the methods sends back to the calling location
 - A method that does not return a value has a void return type
 - ▶ A method body is where all the action takes place. It contains the instructions that implement the method
 - ▶ The `return` statement specifies the value to be returned
 - Its expression must conform to the return type



Passing mechanism

- ▶ A method may modify the programming element underlying the argument in the calling code. It depends on the following two factors:
 - ▶ The argument is being passed by value or by reference .
 - ▶ The argument data type is a value type or a reference type.
 - ▶ The default is to pass arguments by value.
- ▶ **Passing parameters by reference**
 - ▶ Change the value of the parameters and have that change persist.
 - ▶ Use the `ref` or `out` keyword
 - ▶ Note: `ref` requires that the variable be **initialized** before being passed.
 - ▶ Note: `out` arguments need not be initialized prior to being passed but calling method is required to **assign a value before** the method returns.
- ▶ You should choose the passing mechanism carefully for each argument



Passing Value-Type Parameters

- ▶ Case 1: value Type + Pass by value
 - ▶ Passing a copy of the variable to the method
 - ▶ The method cannot modify the variable itself

```
int x1 = 1;
UpdateIntByValue(x1);
Console.WriteLine(x1);
```

```
public static void UpdateIntByValue(int v){
    v *= 2;
}
```

1 (unchanged)

- ▶ Case 2: value Type + Passy by reference (ref/out)
 - ▶ Reference is passed into the method
 - ▶ Both the method definition and the calling method must explicitly use the ref/out keyword.
 - ▶ The method can modify the variable itself

```
int x2=1, x3=1;
UpdateIntByRef(ref x2);
UpdateIntByOut(out x3);
```

```
public static void UpdateIntByRef(ref int v) {
    v *= 3;
}
```

```
public static void UpdateIntByOut(out int v) {
    v = 4;
}
```

3 (changed) x2 must be initialised.

4 changed

required to assign a value before the method returns



Passing Reference-Type

- ▶ Case 3: Reference Type + Pass by value
 - ▶ The method cannot change the variable but can change members of the instance to which it points
 - ▶ A) The method can change the **members**

```
int[] y1 = { 1, 2, 3 };
UpdateRefTypeByValue(y1);
Console.WriteLine(y1[0]);
```

```
public static void UpdateRefTypeByValue(int[] x){
    x[0] = 10;
}
```

=10

- ▶ B) The method cannot change the variable
 - ▶ Example: you cannot assign a new array to the variable

```
int[] y2 = { 1, 2, 3 };
ReplaceRefTypeByValue(y2);
Console.WriteLine(y2[0]);
```

```
public static void ReplaceRefTypeByValue(int[] x) {
    x = new int[] { 2, 3 };
}
```

=1

Cannot change the entire array



Passing Reference Type

- ▶ Case 4: Reference Type + Pass by Reference
 - ▶ The method can change both the variable and members of the instance to which it points
 - ▶ A) The method can change the members

```
int[] y3 = { 1, 2, 3 };
UpdateRefTypeByRef(ref y3);
Console.WriteLine(y3[0]);
```

```
public static void UpdateRefTypeByRef(ref int[] x){
    x[0] = 10;
}
```

=10

change the member

- ▶ B) The method can also change the variable
 - ▶ Example: you can assign a new array to the variable

```
int[] y4 = { 1, 2, 3 };
ReplaceRefTypeByRef(ref y4);
Console.WriteLine(y4[0]);
```

```
public static void ReplaceRefTypeByRef(ref int[] x){
    x = new int[] {2,3};
}
```

=2

change the entire array



Parameters Arrays

- ▶ When you need an indefinite number of arguments, you can declare a parameter array
 - ▶ Use the param keyword in a method declaration
- ▶ Rules:
 - ▶ A method can have only one parameter array (one-dimensional), and it must be the last argument in the procedure definition.
 - ▶ The parameter array must be passed by value.

```
public static void PrintChars(params char[] chars) {
    foreach (char c in chars)
        Console.Write(c + " ");
}
```

- ▶ To call the method:
 - ▶ No parameter — that is, you can omit the argument
 - ▶ By a list of an indefinite number of arguments
 - ▶ By an array with the same element type

```
PrintChars();
PrintChars('a');
PrintChars('a', 'b');
PrintChars(new char[] { '1', '2' });
```



The this keyword

- ▶ The this keyword refers to the current instance of the class.
- ▶ It is useful if you have name clashes between the parameter name and your internal variable

```
public Employee(string name)
{
    this.name = name;
    this.alias = alias;
}
```

Refers to the argument from the method call

- ▶ To pass an object as a parameter to other methods

```
CalcTax(this);
```

- ▶ To invoke another constructor in the same object from a constructor

```
public MyPoint(): this(0,0) {
    ...
}
```



Static Classes

- ▶ Note: Methods and properties only available after creating an instance of the class
- ▶ Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class
 - ▶ Static methods can be invoked directly from the class (not from a specific instance of a class)
- ▶ Rules:
 - ▶ Static classes contain static members.
 - ▶ Static classes cannot be instantiated

```
static class CompanyInfo
{
    public static string GetCompanyName() { return "CompanyName"; }
    public static string GetCompanyAddress() { return "CompanyAddress"; }
    //...
}
```



Static Fields & Methods

- ▶ Static Fields
 - ▶ Note: An instance field has one copy for each object of the class.
 - ▶ A static field has one copy for all objects of a class.
 - ▶ They share a value across all instances of a class.
 - ▶ It is useful if we want to total or count the number of objects for a class
- ▶ Static methods
 - ▶ Static methods and properties can only access static fields

```
class MyCounter {
    public static int Count;
    public int value;
    public MyCounter() {
        Count += 1;
    }
    public MyCounter(int i) {
        value = i;
        Count += 1;
    }
}
```

```
MyCounter obj1 = new MyCounter(1);
MyCounter obj2 = new MyCounter(2);
Console.WriteLine(MyCounter.Count);
```



Overloading

- ▶ Overloading is the creation of more than one method, instance constructor, or property in a class with the same name but different argument types.
- ▶ At run time, system calls the correct method based on the data types of the parameters you specify.

```
public static void Display(char theChar) {
    Console.WriteLine("The char");
}
public static void Display(int theInteger) {
    Console.WriteLine("The Integer");
}
public static void Display(double theDouble) {
    Console.WriteLine("The Double");
}
```

- ▶ Note: methods cannot be overloaded if one method takes a ref argument and the other takes an out argument

```
public static void Display(int theInt) {}
public static void Display(ref int theInt) {}
```

OK!

```
public static void Display(ref char theChar) {}
public static void Display(out char theChar) {}
```

ERROR!



Partial Classes

- ▶ Use the **partial** keyword modifier to split the definition of a class or a struct, or an interface over two or more source files.
 - ▶ Don't need to recreate the original source file
 - ▶ Rules:
 - ▶ All partial-type definitions meant to be parts of the same type must be modified with **partial**
 - ▶ The partial modifier can only appear immediately before the keywords **class**, **struct**, or **interface**
 - ▶ All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module

```
public partial class Employee{
    public void DoWork() {
    }
}
```

```
public partial class Employee {
    public void GoToLunch() {
    }
}
```



Structs Vs Classes

- ▶ The structs in C# seems to similar classes. But they are two entirely different aspects of the language:
 - ▶ The classes are reference types while a struct is a value type in C#.
 - ▶ The objects of class types are always created on heap while the objects of struct types are always created on the stack.
- ▶ Summary:

struct	class
Value Type	Reference Type
Can't initialize a filed inside a struct	OK
No destructor	destructor
No inheritance	inheritance

```
public class Employee{
    int x=10;
    public void DoWork(){
    }
}

public struct MyStruct {
    int x; //no initialization
}
```