C# classes

Lecture 7
CS 638 Web Programming

Lecture overview
- Organization: namespaces and assemblies
- Class members – fields, methods, properties
  - Access modifiers (protection levels)
  - Static versus instance
- Inheritance
  - Polymorphism, overriding methods
- Value and reference types
  - Parameter passing

C# namespaces
- Used to avoid name conflicts (many classes with same name)
- Potential for name conflicts much higher in large projects or projects using many external libraries
- Every class has to be inside a namespace
- Namespaces can be nested
- Classes from the vast library offered by the .NET framework structured in vast hierarchy of namespaces
- Namespaces orthogonal to the structure of the source code
- There can be multiple namespaces in a single source file, a namespace can span multiple source files
- The "using somenamespace;" directive gives him convenience of not having to use fully qualified names for all classes
- May lead to name conflicts, compiler detects ambiguities

Some standard namespaces
- System contains classes that implement basic functionalities like mathematical operations, data conversions etc.
- System.IO contains classes used for file I/O operations.
- System.Collections.Generic contains classes that implement collections of objects such as lists, hashtable etc. using C# generics
- System.Text contains classes that manipulate strings and text
- System.Diagnostics contains classes used in profiling and debugging your application

The structure of applications
- Each C# project is compiled to an assembly
- Can be an executable file or a dynamic link library (DLL) containing the MSIL code
- Assemblies also contain a lot of useful metadata (e.g. version number)
- Can use classes from an external assembly by adding a reference to it in your project
  - Must add explicit references to other projects within solution to use the classes defined there
- Classes loaded when used
Classes
- At the heart of object oriented programming
- Application structure mirrors real world objects
- Related methods and data encapsulated in object
- Objects with the same structure are of same type
- A class is a blueprint for all things of that type
- Instance of a class is a thing, an object
- Classes have three main types of members
  - Methods (functions in other languages)
  - Fields (the data, sometimes called member variables)
  - Properties (accessed like fields, but actually methods)

Example of class and object
```csharp
public class AClass
{
}
```
```csharp
AClass aClass = new AClass();
```

Fields
- Syntax for declaring fields similar to that for local variables
- Typically fields are not "public"
- Public fields
  - break data encapsulation
  - cause loss of control of the class
  - easier for the lazy or hurried
- Use properties for public faces to internal variables (can also make them read-only)

C# properties and accessors
- A property looks just like a field outside the class
- Just like a field, a property has a type associated with it
- Accessors are methods for reading or writing the property – each field may have get and set accessor
- Class author writes body of accessors
- get accessor must return value of same type as property
- set accessor receives implicit parameter "value"
- By controlling the protection level of the accessors (or omitting one of them) the class author can control who can read and who can write property

Property & accessor example
```csharp
class TrackerMaths
{
private int successCounter;
private int failureCounter;
private int overallCount;
public int thing
{
        get
        {
                // return thing; //leads to stack overflow due to infinite recursion
                return thing;
        }
        set
        {
                failureCounter = 1; //leads to stack overflow due to infinite recursion
                thing = value;
        }
}
public void PrintCounter()
{
        Console.WriteLine("Number of success = " + successCounter + ", number of failures = " + failureCounter + ", total = " + overallCount);
}
}
```

Access modifiers (protection levels for class members)
- Class members and classes can have one of the following protection levels
  - public – accessible to everyone
  - private – accessible only inside class
  - protected – accessible for descendants
  - internal – accessible within the same assembly
- Default protection levels
  - Class members, struct members – private
  - Classes, structs, enums – internal
  - Enum members, interface members – public
Methods

- Must exist in a surrounding class or struct
- They have access to private members of the class
- Typically they are public
- "Global" methods done as static public methods
- Each method has name, return type, and 0 or more typed arguments
- The void return type indicates that the method does not return anything
- Overloading: two methods can have the same name, but differ in number or type of the arguments
- The various overloaded methods have separate bodies

Method overloading example

```csharp
class Program
{
    static void Main(string[] args)
    {
        LookUpPerson();
        LookUpPerson("first name", "last name");
        LookUpPerson("Social Security Number");
        Console.WriteLine("Looking up by first and last name.");
    }
}
```

Constructors

- Constructors are special methods
- same name as class
- no return type
- used for initialization of a class instance
- may be overloaded
- If no constructor specified, compiler generates one that initializes members to default values

Static versus instance

- All instances of a class share certain traits – but have individual copies
- Rexx and Fido are Dogs but have different names
- Name is a trait shared by all instances of the class Dog but each instance of Dog has its own copy
- This is the default
- A trait present in all instances of a class and physically shared by all instances is a "static" trait
- Can be methods or fields
- Must be fully named using enclosing class

The "this" keyword

- Refers to the current instance (the object whose method is executed)
- Used to qualify access to members of the current instance
- Typically used for disambiguating a member variable from a method parameter of the same name
- Cannot be used in static methods
- Cannot be used to qualify access to static methods
- Use class name instead

Other qualifiers for fields

- const
  - Compile time constant
- readonly
  - May be initialized at compile time or in a constructor
  - Neither can be changed after its value has been initialized
  - Use them when they apply – they help find some bugs (and they give the compiler more opportunities to optimize the code)
Lecture overview

- Organization: namespaces and assemblies
- Class members – fields, methods, properties
  - Access modifiers (protection levels)
  - Static versus instance
- Inheritance
  - Polymorphism, overriding methods
- Value and reference types
- Parameter passing

Inheritance

```csharp
public class Person
{
    public string name = "";
    static public readonly string status = "OK"
    public Person(string name)
    {
        this.name = name;
    }
    public class Log : Person
    {
        public Log(string name) : base(name)
        {
        }
    }
}
```

Why use inheritance?

- Code reuse
  - The derived class has all members of the base class
- Polymorphism
  - An object belonging to the derived class can be used where the program expects an object from the base class
  - Some methods of the derived class may behave differently than the same methods in base class
  - Methods in different derived classes may differ
  - Polymorphism means that at run time the environment picks the method to run based on actual type of object

Polymorphic virtual methods

- In base class, use keyword "virtual" for methods you want to behave in a polymorphic way
- In derived class, use keyword "override" for methods that implement polymorphic behavior
  - Can use the "base.method()" syntax to call the named method in the base class
  - Use keyword "abstract" for polymorphic methods for which the base class does not define a body
  - If any method in class abstract, class must be abstract
  - Non-abstract derived class overrides abstract methods
  - Abstract classes cannot be instantiated

Non-polymorphic methods

- By default methods are not polymorphic
- Derived classes may re-define such methods using the "new" keyword
- Demo shows difference between the behavior of the two types of methods

Lecture overview

- Organization: namespaces and assemblies
- Class members – fields, methods, properties
  - Access modifiers (protection levels)
  - Static versus instance
- Inheritance
  - Polymorphism, overriding methods
- Value and reference types
  - Parameter passing
Value and reference types

- Value types (ints, doubles, chars, structs)
- Variables of value types directly contain their data
- Reference types (strings, objects)
- Variables of reference types store references
- Two variables may point to the same object
- The "new" operator used to create an object
- Objects stored on heap and when there are no more live references to them they are discarded by the automatic garbage collector

All types descend from object

- All value and reference types are derived (indirectly or directly) from the object class
- object has 4 methods:
  - object.Equals(object other)
  - object.GetType()
  - object.ToString()
  - object.GetHashCode()

```csharp
int x = 1;
x.ToString() => "1"
1.ToString() => "1"
```

Parameters

- All parameters passed by value by default
- To pass by reference use "ref" keyword
- Changes to the parameter inside method visible after it returns
- To return more than one result, use "out" keyword
- "ref" and "out" must be present in both method definition and method invocation