Object Oriented Programming in C#

Trusted partner for your Digital Journey

Atos | Syntel

Objectives

- Understanding the concept of a class.
- Learning different types of constructors in C#.
- Understanding structures in C# and differentiating it with a Class.
- Understanding Inheritance in C#.
- Understanding the concept of a Properties and Indexers.
- Understanding Polymorphism in C# which includes Function.
- Overloading and Function Overriding.
- Understanding the concept of a Abstract Class and Sealed Class.
- Learning Interfaces in C# and differentiating it with Abstract Class.
- Understanding Method Parameters in C#.

Classes

- A class is a user-defined type (UDT) that is composed of field data (*member variables*) and methods (member functions) that act on this data.
- In C# Classes can contain the following
  - Constructors and destructors
  - Fields and constants
  - Methods
  - Properties
  - Indexers
  - Events
  - Overloaded operators
  - Nested types (classes, structs, interfaces, enumerations and delegates)

Class Constructors

- A *constructor* is called automatically right after the creation of an object to initialize it.
- Constructors have the *same* name as their class names
- *Default* constructor: if no constructor is declared, a parameterless constructor can be declared
- A class can contain default constructor and *overloaded* constructors to provide multiple ways to initialise objects.
- *Static* constructor: similar to static method. It must be parameterless and must not have an access modifier (private, public).

Structures in C#

- Structs are similar to classes in that they represent data structures that can contain data members and function members.
- Unlike classes, structs are value types and do not require heap allocation.
- Structs are particularly useful for small data structures that have value semantics.
- The simple types provided by C#, such as int, double, and bool, are in fact all struct types.

Structures in C#

Classes and Structs Similarities:

- Both are user-defined types
- Both can implement multiple interfaces
- Both can contain
- Data
  - Fields, constants, events, arrays
- Functions
  - Methods, properties, indexers, operators, constructors
- Type definitions
  - Classes, structs, enums, interfaces, delegates

Class vs. Structure

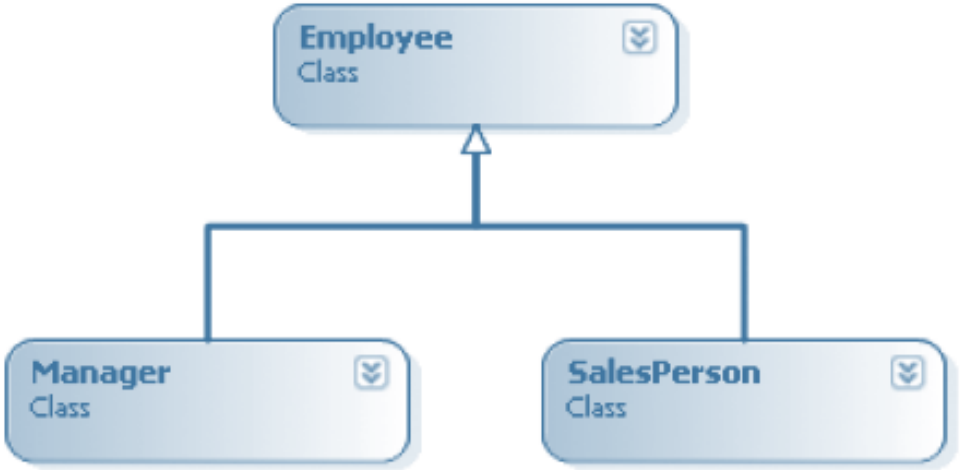
Class	Structure
Reference type	Value type
Can inherit from any non-sealed reference type	No inheritance (inherits only from System.ValueType)
Can have a destructor	No destructor
Can have user-defined parameterless constructor	No user-defined parameterless constructor

Class Inheritance

- Inheritance is a form of *software reusability* in which classes are created by reusing data and behaviours of an existing class with new capabilities.
- A class inheritance hierarchy begins with a *base* class that defines a set of common attributes and operations that it shares with *derived* classes.
- A derived class *inherits* the resources of the base class and *overrides* or *enhances* their functionality with new capabilities.
- The classes are separate, but related

Class Inheritance

- Inheritance is also called "is a" relationship



- A SalesPerson "is-a" Employee (as is a Manager)
  - **Base classes** (such as Employee) are used to define general characteristics that are common to all descendents
  - **Subclasses** (such as SalesPerson and Manager) extend this general functionality while adding more specific behaviors

Class Accessibility

- **public:** access is not restricted
- **private:** access is limited to the containing type
- **protected:** access is limited to the containing class or types derived from the containing class
- **internal**: access is limited to the current assembly
- **protected internal:** access is limited to the current assembly or types derived from the containing class

Properties

- Properties provide the opportunity to protect a field in a class by reading and writing to it through the property
- In other languages, this is often accomplished by programs implementing specialized getter and setter methods
- One or two code blocks - representing a get accessor and/or a set accessor
- The code block for the get accessor is executed when the property is read; the code block for the set accessor is executed when the property is assigned a new value

Properties

- A property without a set accessor is considered read-only.
- A property without a get accessor is considered write-only.
- A property that has both assessors is read-write.
- Properties have many uses:
  - they can validate data before allowing a change;
  - they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database;
  - they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Indexers

- Indexers are “smart arrays”.
- Indexers permit instances of a class or struct to be indexed in the same way as arrays.
- Indexers are similar to properties except that their accessors take parameters.
- Simple indexer-declaration is as follows:
  - Modifier type this [formal-index-parameter-list] {accessor-declarations}

Method Overriding

- Polymorphism provides a way for a subclass to customize how it implements a method defined by its base class.
- If a base class wishes to define a method that may be overridden by a subclass, it must specify the method as virtual.
- A subclass uses the override keyword to redefine a virtual method:

Abstract Class

- An abstract class is one that cannot be instantiated.
- It is intended to be used as a base class.
- May contain abstract and non-abstract function members.
- It cannot be sealed.

Abstract Methods

- *Abstract* methods *do not* have an implementation in the abstract base class and every concrete derived class *must override all* base-class abstract methods and properties using keyword *override*.
- Must belong to an abstract class
- Intended to be implemented in a derived class
- When a class has been defined as an abstract base class, it may define any number of *abstract members* (which is analogous to a C++ *pure virtual function*).
- Abstract methods can be used whenever you wish to define a method that *does not* supply a default implementation.

Sealed Class

- To prevent inheritance a sealed modifier is used to define a class.
- A sealed class is one that cannot be used as a base class.
- Sealed classes can’t be abstract.
- All structs are implicitly sealed.
- Many .NET Framework classes are sealed: String, StringBuilder, and so on
- Why seal a class?
  - To prevent unintended derivation.
  - Code optimization.
  - Virtual function calls can be resolved at compile-time.

Interfaces

- An interface defines a contract
- Interface is a purely abstract class; has only signatures, no implementation.
- May contain methods, properties, indexers and events

  (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly *public abstract* (*virtual*).
- Interface members must not be *static*.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.

Implementing Interfaces

- A class can inherit from a *single base class*, but can implement *multiple interfaces*.
- A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, indexer) must be *implemented* or *inherited* from a base class.
- Implemented interface methods must *not* be declared as *override*.
- Implemented interface methods can be declared as *virtual* or *abstract* (i.e. an interface can be implemented by an abstract class).

Interfaces – Explicit Implementation

- If two interfaces have the same method name, you can explicitly specify interface & method name to disambiguate their implementations.

Abstract Class & Interface

- Abstract classes typically do far more than define a group of abstract methods. They are free to define public, private and protected state data, as well as any number of concrete methods that can be accessed by the subclasses.
- Interfaces on the other hand, are pure protocol. Interfaces never define data types, and never provide a default implementation of the methods. Every member of an interface (whether is method or property) is automatically abstract. Given that C# support single inheritance , the interface-based protocol allows a given type to implement multiple interfaces and all implemented methods has to be public .

Method Parameters

- A *method* is a member that implements a computation or action that can be performed by an object or class. Methods are declared using *method-declaration*:

- [attributes] [method-modifiers] return-type method-name-identifier ([formal-parameter-list]){ [statements]}

- There are 3 kinds of parameters:
  - out
  - ref
  - params

Partial Classes

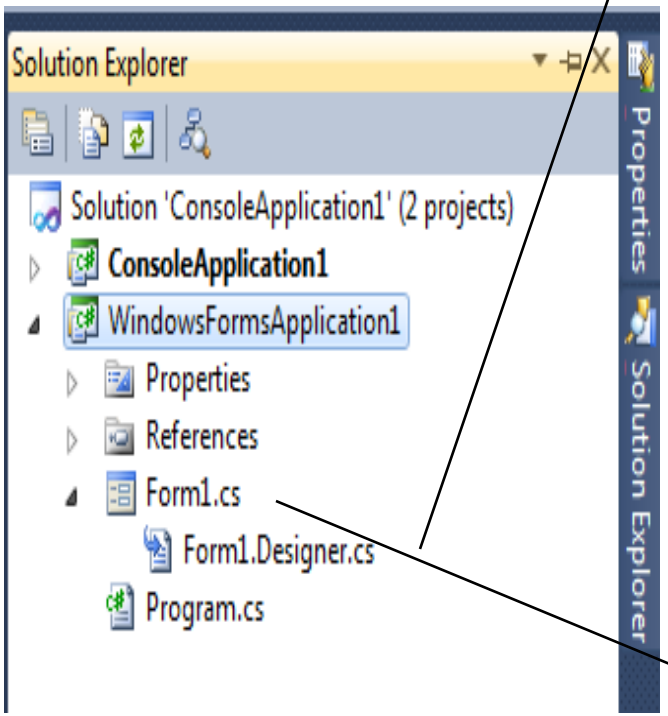
- Partial classes give you the ability to split a single class into more than one source code (.cs) file. Here's what a partial class looks like when it is split over two files:

//stored in file MyClass1.cs
 	
public partial class MyClass
 	
{
 	
public MethodA()
 	
{...}
 	
}
	//stored in file MyClass2.cs
	
	public partial class MyClass
	
	{
	
	public MethodB()
	
	{...}
	
	}

Concept of a Partial Class

- When you build the application, Visual Studio .NET tracks down each piece of MyClass and assembles it into a complete, compiled class with two methods, MethodA() and MethodB().

Concept of a Partial Class



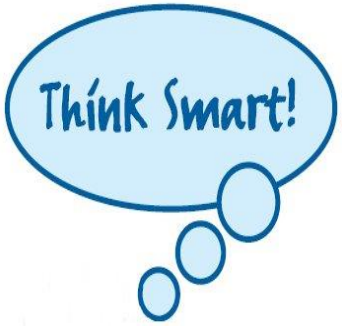

```
partial class Form1
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    /// <param name="disposing">true if managed resourc
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    Windows Form Designer generated code
}
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

Quick Recap

- Creating a class in C#
- Different Access Modifiers in C#
- Structures and its distinction from classes
- How to use inheritance in C#
- What are properties and Indexers and how to use them
- The concept of Function Overriding in C#
- The concept of an Abstract Class, Abstract Methods
- Sealed Classes
- Method Parameters (value, ref, out & params)
- Partial Classes

Test your memory

- How is class different from a structure?
- Why is class called as a “is-a” relationship?
- What are the different access specifiers in C#?
- What is a Sealed Class in C#?
- Can abstract class be sealed?
- How is abstract class different from an interface?
- How is function overriding implemented in C#?
- What are the different method parameters in C#?
- How is Structure different from Class?



Thank You

Atos | Syntel