C# Programming

Notes by JACPro

C# 1.x Notes

Classes

public class Customer

{

//fields

//properties

//methods

//events

}

Customer object2;

Declares type and namespace but will be null unless an object is assigned.

Customer object2 = new Customer();

Create and assign an object.

Classes are reference types.

Customer object3 = object2;

object2 and object3 refer to the same object, so changes to one will be reflected in the other.

Inheritance

You can only inherit from a single class.

public class Manager : Employee

{

//fields, properties, methods and events inherited from Employee

//new fields, properties, methods and events for Manager

}

Abstract Classes

The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

Abstract classes cannot be instantiated.

public abstract class A

{

//Class members here

}

Sealed Classes

Sealed classes cannot be used as base classes because they prevent derivation.

public sealed class D

{

//Class members here

}

A method, index, property or event can be declared as sealed in a derived class when overwriting a virtual member of the base class; this prevents the virtual aspect from being derived further.

public class D : C

{

public sealed override void DoWork() { }

}

Auto-Implementation

Automatic properties are used to make private variables visible outside the class by using properties with get; set; methods to expose private fields.

* Fields are normal member variables of a class
* Properties enable a class to publicly expose methods for getting and setting private fields. Because they provide a definition but no implementation, they can be used by interfaces (unlike fields).

public int SomeProperty { get; set;}

Automatic properties are used when no additional logic is required in the property accessors (i.e. using only standard get; set; accessors).

Automatic properties are shorthand for the following non-automatic property code:

private int \_someField;

public int SomeProperty

{

get { return \_someField; }

set { \_someField = value; }

}

The above code shows how you might implement a property that was declared in an interface (or how to declare and implement a property in a class).

Uses of properties include:

* Validating data before allowing a change
* Transparently exposing data in a class where that data is actually retrieved from some other source (e.g. a database)
* They can take an action when data is changed, such as raising an event or changing the value of other fields

Structs

A structure type is a value type that can encapsulate data and related functionality. Use the “struct” keyword to define a structure type.

public struct Coords

{

public Coords (double x, double y)

{

X = x;

Y = y;

}

public double X { get; }

public double Y { get; }

public override string ToString() =>$“({X}, {Y})”;

}

Typically, you use structure types to design small data-centric types that provide little or no behaviour.

You can also declare readonly structs using the readonly keyword.

public readonly struct Coords

{

**//can’t declare a parameterless constructor**

public Coords (double x, double y)

{

**//struct constructor must initialize all instance fields**

X = x;

Y = y;

}

**//can’t initialize instance fields or properties at declaration**

**//but can initialize static or const field or static property at declaration**

public double X { get; }

public double Y { get; }

public override string ToString() =>$“({X}, {Y})”;

}

Structs can’t inherit from other classes or structs and they can’t be inherited from. However, the can **implement interfaces**.

Instantiating Structs

Typically, structs are instantiated using the new keyword; in addition to explicitly declared constructors, structs have an implicit parameterless constructor, which produces the default value of the type.

If all instance fields are accessible, structs can also be instantiated without the “new” operator:

public struct Coords

{

public double x;

public double y;

}

public static void Main()

{

Coords p;

p.x = 3;

p.y = 4;

}

Interfaces

Interfaces contain definitions for a group of related functionalities that a **non-abstract** class or struct **must** implement. Interfaces may define **static** methods, which must have implementation. Since C# 8.0, interfaces can define default implementations for members. Interfaces **can’t** declare instance data (e.g. fields, auto-implemented properties and property-like events).

While C# doesn’t support multiple inheritance, it does allow you to implement an unrestricted number of interfaces, meaning a class can use behaviour from many different places. Additionally, structs are unable to inherit from classes or other structs, but they can implement interfaces, allowing structs to include behaviour from other sources.

Interfaces can be defined using the interface keyword, as follows:

interface IEquatable<T>

{

bool Equals (T obj);

}

By convention, interface names should start with a capital I.

Any methods declared by an interface must be implemented by any class or struct that implements that interface, with a signature matching the one specified by the interface. For example, a class that implements IEquatable **must** always have a method called Equals, which takes a parameter of type T and returns a Boolean value.

The definition of IEquatable<T> doesn’t provide an implementation for Equals; it **must** be implemented by the class or struct that implements the interface.

Interfaces **can** contain:

* Instance methods
* Properties
* Events
* Indexers
* Static Constructors
* Fields
* Constants
* Operators

Interfaces **can’t** contain:

* Instance fields
* Instance constructors
* Finalizers

By default, interface members are public.

To implement an interface member, the corresponding member of the class that implements it must be public, non-static, and have an identical name and signature to the interface member.

When implementing an interface, a class or struct **must** provide an implementation for every member which is declared but not implemented by the interface. However, if a base class implements an interface, any class derived from that base class will inherit the base class’ implementations for all interface members.

You can implement interfaces as follows:

public class Car : IEquatable<Car>

{

public string Make { get; set; }

public string Model { get; set; }

public string Year { get; set; }

//Implementation of the Equals method defined by IEquatable<T>

public bool Equals (Car car)

{

return (this.Make, this.Model, this.Year)

== (car.Make, car.Model, car.Year);

}

}

If implementing two interfaces that both contain members with the same signature, both interfaces will use the same implementation for those members.

public interface IControl

{

void Paint();

}

public interface ISurface

{

void Paint();

}

public class SampleClass : IControl, ISurface

{

public void Paint()

{

Console.Writeline(“Paint method in SampleClass”);

}

}

In the above code, calls to the paint method of SampleClass, IControl and ISurface will all call the same method. In order to implement different methods of the same name each interface, you can use explicit interface implementation.

public class SampleClass : IControl, ISurface

{

void IControl.Paint()

{

Console.Writeline(“Paint method for IControl”);

}

void ISurface.Paint()

{

Console.Writeline(“Paint method for ISurface”);

}

}

Class indexers and properties can define additional accessors for properties and indexers defined by an interface. For example, if an interface defines a property with only a get accessor, the class implementing the interface can go on to declare both get and set accessors. However, if the property uses explicit implementation (see above), the accessors **must** match.

Interfaces may also inherit from an unrestricted number of interfaces; derived interfaces will inherit all members from the base interfaces, including members inherited by those base interfaces. A class inheriting a derived interface can be implicitly converted to any of its base interfaces.

A class may implement an interface multiple times by implementing a base interface and an interface derived from it; however, a class may **only provide one** implementation of any given interface.

If an interface is inherited because the base class implements it, the implementation of the interface from the base class will be provided. However, any virtual interface members can be overwritten with new implementations if required.

When interfaces declare default implementations for methods, classes implementing those interfaces will inherit those implementations; implementations defined in interfaces are **virtual**, so they implementing class can override them with a new implementation.