Class and Interface

.NET

Class *is the most fundamental of C#'s types. A* ***class*** *is a data structure that combines state (fields) and actions (methods) into a single unit.*
Classes *support inheritance and polymorphism. A* ***Class*** *is a blueprint for a* ***Class*** *Object.*

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects

Class

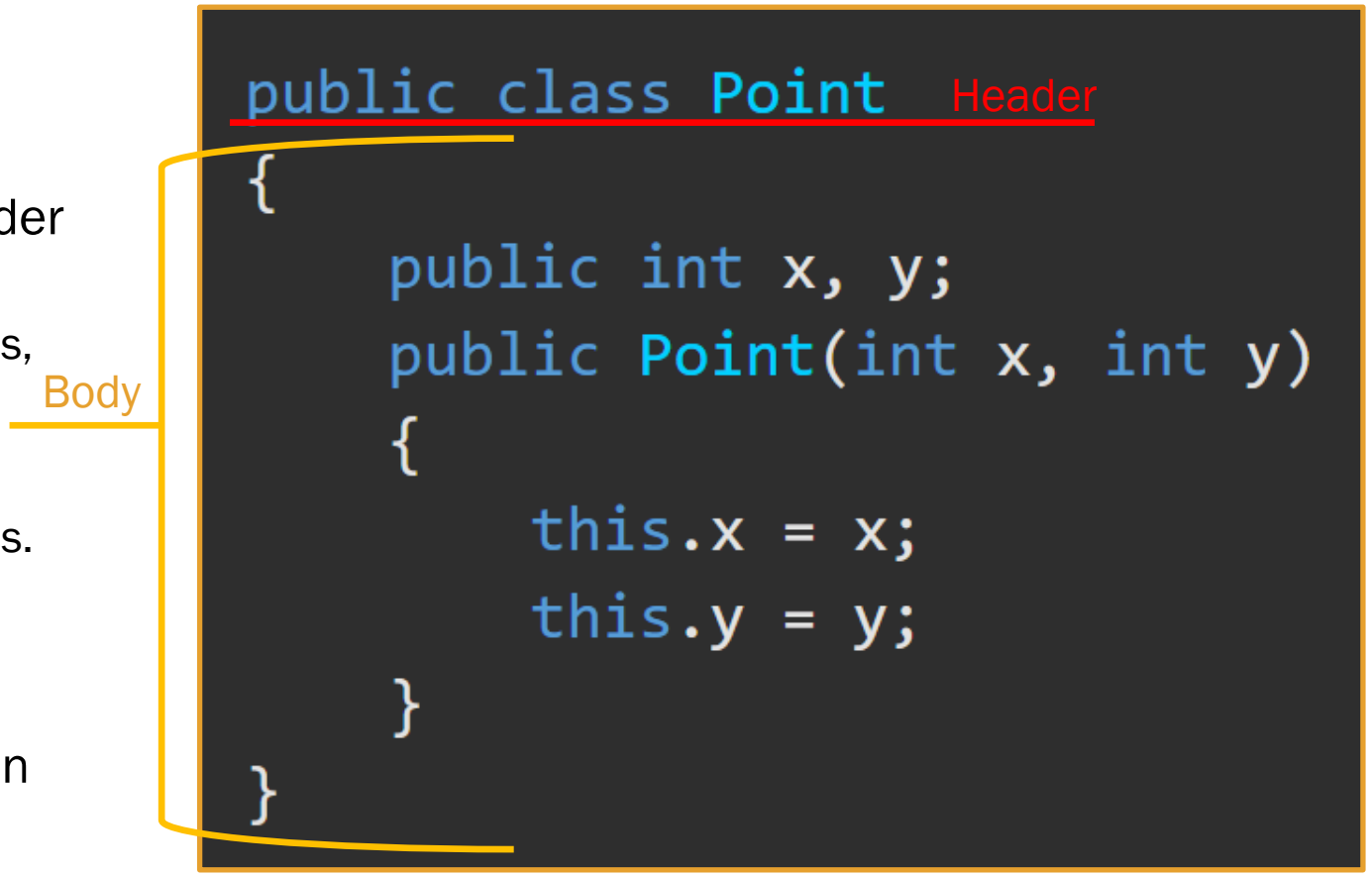
https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects

Classes are defined using class declarations.

A class declaration starts with a header that specifies

- the attributes and modifiers of the class,
- the name of the class,
- the base class (if given), and
- the interfaces implemented by the class.

The header is followed by the class ***body***, which consists of a list of member declarations written between curleyBrackets { }.



Class – Instance Instantiation

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects

Instances of classes are created using the ***new*** operator, which

- allocates memory for a new instance,
- invokes a constructor to initialize the instance
- returns a reference to the instance.

The memory occupied by an object is automatically reclaimed by the ***Garbage Collector*** when the object is out of scope (is no longer reachable).

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

Class - Members

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects

Members of a class are:

- Constructors - To initialize instances of the class
- Constants - Constant values
- Fields – Variables
- Methods – Computations/actions that can be performed
- Properties – Fields combined with the actions associated with reading/writing them
- Types - Nested types declared by the class

Class members can be:

- static - belong to classes. Envoked with: ClassName.MethodName();
- instance - belong to ***instances*** of classes. Envoked with: InstanceName.MethodName();

Accessibility of Classes

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

- ***Classes*** and ***structs*** declared directly in a ***namespace*** (not nested in another class or struct) can be either ***public*** or ***internal***.
- ***Derived*** classes can't have greater accessibility than their base class.
- ***Internal*** is default if no access modifier is specified.

Class – Member Accessibility

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types#classes-and-objects
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

---

Access Modifiers control the regions of a program that can access a class member.

- private - This class only.
- protected - accessed by code in the same class, or in a derived class.
- private protected - accessed by the same class or derived classes only when in the same assembly.
- internal – accessed by any code in the same assembly (.exe, .dll).
- protected internal - accessed by any code in the same assembly or from within a derived class in a different assembly.
- public - Access isn't limited.

Class – Local Variables

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#method-body-and-local-variables

Local variables are declared inside the body of a method. They must have a ***type*** name and a variable name. All variables must be given a default value.

- int == 0;
- string == “”;

```
using System;
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }
}
```

Class – Methods

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#methods

There are two categories of methods:

- ***Static*** – accessed directly through the class
- ***Instance*** – accessed though instances of a class.

Methods have a ***Method Signature*** which consists of:

- The name of the method,
- The ***type*** parameters (if needed),
- Parameter names.

*The signature of a method doesn't include the return type.

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

```
// Methods
public void Add(T item)
{
    if (count == Capacity) Capacity = count * 2;
    items[count] = item;
    count++;
    OnChanged();
}
```

Class – Static and Instance Methods

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#static-and-instance-methods

static method –

- declared with a ***static*** modifier.
- doesn't operate on a specific class instance.
- Only accessed through the class name. (Ex. MyClassName.MyStaticMethod())
- Cannot use this.

instance method –

- declared with any modifier other than ***static***.
- operates on a specific class ***instance*** only.
- can access both ***static*** and ***instance*** members.
- Can use this.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity()
    {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo()
    {
        return serialNo;
    }
    public static int GetNextSerialNo()
    {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value)
    {
        nextSerialNo = value;
    }
}
```

Value and ref Parameters

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

Parameters are used to receive variables from method calls.

There are five types of method parameters:

```
static void Divide(int x, int y,
{
    result = x / y;
    remainder = x % y;
}
```

1. value parameter

- a copy of the argument passed. Changes don't affect the original argument. Can be options by specifying a default value.

```
using System;
class RefExample
{
    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void SwapExample()
    {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine($"{i} {j}");    // Outputs "2 1"
    }
}
```

2. reference parameter

- declared with the 'ref' modifier. Used for passing value arguments by reference. The argument must be a variable with a definite value. Changes take place on the original value.

out and params parameters

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params

---

- **3. output parameter** –
  - declared with the out modifier.
    - Used for passing arguments by reference.
    - An explicitly assigned value is not allowed before the method call.
- **4. parameter array** –
  - permits an 'N' number of arguments to be passed to a method.
  - Declared with the params modifier.
  - Must be the last parameter and be a 1-D array.
  - Write() and WriteLine() methods use parameter arrays.

```
using System;
class OutExample
{
    static void Divide(int x, int y, out int result, out int remainder)
    {
        result = x / y;
        remainder = x % y;
    }
    public static void OutUsage()
    {
        Divide(10, 3, out int res, out int rem);
        Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
    }
}
```

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

in parameter

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier

---

The in keyword causes arguments to be passed by reference but ensures the argument is not modified. It makes the formal parameter an alias for the argument, which must be a variable.

It is like the ref or out keywords, except that in arguments cannot be modified by the called method. Whereas ref arguments may be modified, out arguments must be modified by the called method, and those modifications are observable in the calling context.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

Class – Method Overloading

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#method-overloading

Method overloading

- permits multiple methods in the same class to have the same name
- Methods must each have unique ***parameter*** lists.
- The compiler uses ‘overload resolution’ to determine the specific method to invoke.
- ‘Overload resolution’ finds the one method that best matches the arguments or reports an error if none is found.
- A method can be selected by explicitly ***casting*** the arguments to the exact ***parameter*** types.

```
using System;
class OverloadingExample
{
    static void F()
    {
        Console.WriteLine("F()");
    }
    static void F(object x)
    {
        Console.WriteLine("F(object)");
    }
    static void F(int x)
    {
        Console.WriteLine("F(int)");
    }
    static void F(double x)
    {
        Console.WriteLine("F(double)");
    }
    static void F<T>(T x)
    {
        Console.WriteLine("F<T>(T)");
    }
    static void F(double x, double y)
    {
        Console.WriteLine("F(double, double)");
    }
    public static void UsageExample()
    {
        F();            // Invokes F()
        F(1);           // Invokes F(int)
        F(1.0);         // Invokes F(double)
        F("abc");       // Invokes F<string>(string)
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<int>(int)
        F(1, 1);        // Invokes F(double, double)
    }
}
```

Optional Parameters and Default Parameter Values

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs

---

A parameter can be ***optional***. Any call must provide arguments for all required parameters but can omit arguments for ***optional*** parameters.

Each ***optional*** parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

Optional parameters are at the end of the parameter list after all required parameters. The caller must provide arguments for all required parameters before any optional parameters.

Interface

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/interfaces
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface

---

- An ***interface*** defines a ***contract*** that can be implemented by classes and structs.
- An ***interface*** can contain methods, properties, events.
- An interface usually only specifies the members that must be implemented by classes or structs that implement the interface.
- ***Interface*** implementation is NOT inheritance. It is intended to express a "can do" relationship between an interface and its implementing type.
- ***Interfaces*** are used to simulate ***multiple inheritance***.
- An ***interface*** CAN have *default interface methods*. These methods have implementations and are not required to be implemented by the implementing class. They are useful for when methods are added to an interface after other classes have implemented it already.

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Interface

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/interfaces

Interfaces may employ multiple inheritance.

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

Classes and structs can implement multiple interfaces.

```
interface IDataBound
{
    void Bind(Binder b);
}
public class EditBox: IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

Are Paint() and Bind() defined?

Class – Type Parameters

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#type-parameters

Class ***Type*** Parameters:

- are used to define a ***generic*** class type.
- follow the class name and are inside **< >**.
- are used to define the ***types*** that the members of the class act on.

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

Class – Base (inherited) Classes

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/classes-and-objects#base-classes

A class declaration specifies ***inheritance*** of a ***base*** class by following the class name (and type parameters) with...

: [baseClassName]

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
public class Point3D: Point
{
    public int z;
    public Point3D(int x, int y, int z) :
        base(x, y)
    {
        this.z = z;
    }
}
```

Record

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record