Classes and Objects

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for instances of the class, also known as **objects**. Classes support inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

New classes are created using `class` declarations. A `class` declaration starts with a header. The header specifies:

- The attributes and modifiers of the `class`
- The name of the `class`
- The base `class` (when inheriting from a base class)
- The interfaces implemented by the `class`.

The header is followed by the class body, which consists of a list of member declarations written between the delimiters { and }. The following code shows a declaration of a simple class named **Point**:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

Instances of classes are created using the **new** operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two **Point** objects and store references to those objects in two variables:

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

The memory occupied by an object is automatically reclaimed when the object is no longer reachable. It's neither necessary nor possible to explicitly deallocate objects in C#.

Base classes (Inheritance)

A class declaration may specify a base class. Follow the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type object. In the following example, the base class of Point3D is Point. From the first example, the base class of Point is object:

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
```

```
        Z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains almost all members of its base class. A class doesn't inherit the instance and static constructors, and the finalizer. A derived class can add new members to those members it inherits, but it can't remove the definition of an inherited member. In the previous example, **Point3D** inherits the **X** and **Y** members from **Point**, and every **Point3D** instance contains three properties, **X**, **Y**, and **Z**.

An implicit conversion exists from a class type to any of its base class types. A variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type Point can reference either a **Point** or a **Point3D**:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Types of classes in C#:

1. Abstract Class
2. Sealed Class
3. Partial Class

Abstract Class

The `abstract` modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as abstract must be implemented by non-abstract classes that derive from the abstract class.

Classes can be declared as abstract by putting the keyword abstract before the class definition. For example:

```
public abstract class A
{
    // Class members here.
}
```

An `abstract` class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword abstract before the return type of the method. For example:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Abstract methods have the following features:

- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.
- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. The implementation is provided by a method override, which is a member of a non-abstract class.
- It is an error to use the static or virtual modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the abstract modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

An abstract class must provide implementation for all interface members. An abstract class that implements an interface might map the interface methods onto abstract methods.

Example:

In this example, the `class` **Square** must provide an implementation of **GetArea** because it derives from **Shape**:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
```

```
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Sealed Class

The `sealed` keyword enables you to prevent the inheritance of a class or certain class members that were previously marked virtual. Classes can be declared as sealed by putting the keyword `sealed` before the class definition. For example:

```
public sealed class D
{
    // Class members here.
}
```

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the sealed keyword before the override keyword in the class member declaration. For example:

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

Example:

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

In the previous example, you might try to inherit from the sealed class by using the following statement:

```
class MyDerivedC: SealedClass {} // Error
```

The result is an error message:

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Partial Class

It is possible to split the definition of a class, a struct, an interface or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled. There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.

The `partial` keyword indicates that other parts of the `class`, `struct`, or `interface` can be defined in the namespace. All the parts must use the `partial` keyword. All the parts must be available at compile time to form the final type. All the parts must have the **same accessibility**, such as public, private, and so on.

The following are merged from all the partial-type definitions:

- XML comments
- interfaces
- generic-type parameter attributes
- class attributes
- members

There are several rules to follow when you are working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with partial. For example, the following class declarations generate an error:

  ```
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- The partial modifier can only appear immediately before the keywords class, struct, or interface.
- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

  ```
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  ```

```
{
    partial class NestedClass { }
}
```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.
- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.
- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
  public, private, protected, internal, abstract, sealed, base class, new modifier (nested parts), generic constraints

Example:

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```