Object-Oriented Programming

Class Type

The most fundamental programming construct is the **class type**. Formally, a **class** is a user-defined type that is composed of field data (often called member variables) and members that operate on this data (such as constructors, properties, methods, events, and so forth).

The power of object-oriented languages, such as C#, is that by grouping data and related functionality in a unified class definition, you are able to model your software after entities in the real world.

A class is defined in C# using the **class** keyword. After you have defined a class type, you will need to consider the set of member variables that will be used to represent its state. For example, you might decide that cars maintain an int data type to represent the current speed and a string data type to represent the car's friendly pet name. After you have defined the set of member variables representing the state of the class, the next design step is to establish the members that model its behavior. For this example, the Car class will define one method named SpeedUp() and another named PrintState().

```
class Car
{
        // The 'state' of the Car.
        public string petName;
        public int currSpeed;

        // The functionality of the Car.
        public void PrintState()
        {
                Console.WriteLine($"{petName} is going {currSpeed}MPH.");
        }

        public void SpeedUp(int delta)
        {
                currSpeed += delta;
        }
}
```

Notice that these member variables are declared using the public access modifier. Public members of a class are directly accessible once an object of this type has been created. Recall the ***term object is used to describe an instance of a given class type created using the new keyword***.

```
Car myCar = new Car();
myCar.petName = "Fred";

// or
Car myCar;
myCar = new Car();
```

Field data of a class should seldom (if ever) be defined as public. to preserve the integrity of your state data, it is a far better design to define data as private (or possibly protected) and allow controlled access to the data via properties.

Objects must be allocated into memory using the new keyword. *If you do not use the new keyword and attempt to use your class variable in a subsequent code statement, you will receive a compiler error.*

Constructor

C# supports the use of *constructors*, which allow the state of an object to be established at the time of creation.

A constructor is a special method of a class that is called indirectly when creating an object using the new keyword. *However, unlike a "normal" method, constructors never have a return value (not even void) and are always named identically to the class they are constructing.*

Every C# class is provided with a "freebie" default constructor that you can redefine if need be. **Default constructor never takes arguments.**

After allocating the new object into memory, the default constructor ensures that all field data of the class is set to an appropriate default value.

As soon as you define a custom constructor with any number of parameters, the default constructor is silently removed from the class and is no longer available. Therefore, if you want to allow the object user to create an instance of your type with the default constructor, as well as your custom constructor, you must explicitly redefine the default.

Following class supports a total of three constructors:

```
public class Person
{
        public string Name { get; set; }
        public int Age { get; set; }

        public Person()
        {
        }

        public Person(string name) =>  Name = name;

        public Person(string name, int age)
        {
                Name = Name;
                Age = age;
        }
}
```

The third constructor is not a valid candidate, since expression bodied members are designed for one-line methods.

Keep in mind that what makes one constructor different from another (in the eyes of the C# compiler) is the number of and/or type of constructor arguments.

Remember that when you define a method of the same name that differs by the number or type of arguments, you have ***overloaded*** *the method !*

this Keyword

C# supplies a this keyword that provides access to the current class instance.

- One possible use of the this keyword is to resolve scope ambiguity, which can arise when an incoming parameter is named identically to a data field of the class.

```
public class Person
{
        public string Name { get; set; }
        public Person(string Name)
        {
                this.Name = Name;
        }
}
```

- Another use of the this keyword is to design a class using a technique termed constructor chaining. This design pattern is helpful when you have a class that defines multiple constructors. Given that constructors often validate the incoming arguments to enforce various business rules, it can be quite common to find redundant validation logic within a class's constructor set.

```
public class Person
{
        public string Name { get; set; }
        public int Age { get; set; }

        public Person(string name)
        {
                Name = name;
        }
        public Person(string name, int age):this(name)
        {
                Age = age;
        }
}
```

A cleaner approach is to designate the constructor that takes the greatest number of arguments as the “master constructor” and have its implementation perform the required validation logic. The remaining constructors can make use of the this keyword to forward the incoming arguments to the master constructor and provide any additional parameters as necessary.

On a final note, do know that once a constructor passes arguments to the designated master constructor (and that constructor has processed the data), the constructor invoked originally by the caller will finish executing any remaining code statements.