Inheritance

The basic idea behind classical inheritance is that new classes can be created using existing classes as a starting point.

***Inheritance** is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.*

The existing class that will serve as the basis for the new class is termed a base class, superclass, or parent class. The role of a base class is to define all the common data and members for the classes that extend it. The extending classes are formally termed derived or child classes.

Ordinarily, inheritance is used to express an *"is a" relationship* between a base class and one or more derived classes, where the derived classes are specialized versions of the base class; the derived class is a type of the base class. For example, the Publication class represents a publication of any kind, and the Book and Magazine classes represent specific types of publications. Derived classes have access to each public member defined within the parent class.

Although constructors are typically defined as public, a derived class never inherits the constructors of a parent class. Constructors are used to construct only the class that they are defined within, although they can be called by a derived class through constructor chaining.

Always remember that inheritance preserves encapsulation.

```
public class Car
{
        public string Model { get; set; }
        public int Speed;
        public int Year { get; set; }
}
public class Mercedes : Car
{
        public string carClass {get; set; }
}

static void Main(string[] args)
{
        Car myCar = new Car();
        myCar.Model = "Smth.";
}
```

It is important to keep in mind that C# demands that a **given class have exactly one direct base class**. It is not possible to create a class type that directly derives from two or more base classes (this technique, which is supported in unmanaged C++, is known as multiple inheritance, or simply MI). But it is permissible directly to derive from multiple interfaces.

sealed keyword

C# supplies another keyword, sealed, that prevents inheritance from occurring. **When you mark a class as sealed, the compiler will not allow you to derive from this type.**

```
sealed class Car
{
}      // The Car class cannot be extended!
```

If you try to derive from this sealed class, you will receive a compile-time error.

Recall, C# structures are always implicitly sealed. therefore, you can never derive one structure from another structure, a class from a structure, or a structure from a class. structures can be used to model only stand-alone, atomic, user-defined data types. if you want to leverage the is-a relationship, you must use classes.

To help optimize the creation of a derived class, you will do well to implement your subclass constructors to explicitly call an appropriate custom base class constructor, rather than the default. In this way, you are able to reduce the number of calls to inherited initialization members (which saves processing time).

```
public class Car
{
        public string Name { get; set; }
        public int Speed;
        public int Year { get; set; }
        public Car()  { }
        public Car(string name, int year)
        {
                Name = name;
                Year = year;
        }
        public Car(string name, int speed, int year) : this(name, year)
        {
                Speed = speed;
        }
}
public class Mercedes : Car
{
        public string carClass { get; set; }
        public Mercedes(string name, int speed, int year, string carClass)
        : base(name, speed, year)
        {
                this.carClass = carClass;
        }
}
```

The base keyword is hanging off the constructor signature which always indicates a derived constructor is passing data to the immediate parent constructor. In this situation, we are explicitly calling the three-parameter constructor defined by Car base class.

*Use the **base** keyword whenever a subclass wants to access a public or protected member defined by a parent class. Use of this keyword is not limited to constructor logic.*

Finally, recall that once you add a custom constructor to a class definition, the default constructor is silently removed. Therefore, be sure to redefine the default constructor for the Car type.

When a base class defines protected data or **protected** members, it establishes a set of items that can be accessed directly by any descendant. *The benefit of defining protected members in a base class is that derived types no longer have to access the data indirectly using public methods or properties.* The possible downfall, of course, is that when a derived type has direct access to its parent's internal data, it is possible to accidentally bypass existing business rules found within public properties. So, we cannot call the protected member of base class from the instances of base or derived classes.

Although protected field data can break encapsulation, it is quite safe (and useful) to define protected methods. When building class hierarchies, it is common to define a set of methods that are only for use by derived types and are not intended for use by the outside world.

Nested Type Definitions

- A public nested type can be used by anybody
- A private nested type can only be used by members of the containing class.
- Nested types allow you to gain complete control over the access level of the inner type because they may be declared privately (recall that non-nested classes cannot be declared using the private keyword).
- Because a nested type is a member of the containing class, it can access private members of the containing class.
- Often, a nested type is useful only as a helper for the outer class and is not intended for use by the outside world.