The Syntax and Terminologies

In this lesson, you will learn how to use inheritance and the terminologies related to it.

WE'LL COVER THE FOLLOWING

The Terminologies

Since we know that a new class is created *based* on an *existing* class in inheritance, we use the terminology below for the new class and the existing class:

- **Base Class (Mother Class or Superclass):** This class allows the *re-use* of its `non-private` members in another class.
- **Derived Class (Child Class or Subclass):** This class is the one that *inherits* from the superclass.

> 💡 A *child* class has **all non-private** characteristics of the *mother* class.

What Does a Child have?

An object of the child class can use:

- All members defined in the **child** class.
- All `non-private` members defined in the **mother** class.

> ⚠️ In C#, we can declare some classes non-inheritable using the keyword, `sealed`, like: `public sealed ClassName {...}`.

Implementation

In C#, to implement inheritance, we have to use the colon `:` operator. The generic syntax looks like the following:

```
SubClass : SuperClass{
//contents of SubClass
}
```

Let's jump back to our example of a `VendingMachine`. In a food vending machine, we have got several types of products e.g. beverages, chocolates, biscuits, etc. Now the noticeable thing here is that we always implement an inheritance relationship between classes when they have some of the attributes in common. For example, in the case of the products, all the products have names, prices, expiry dates, etc. Also, the *IS A* relationship stands valid in this case as we can say:

- A beverage *is a* product.
- A chocolate *is a* product.
- A biscuit *is a* product.

So, from the above discussion, we can conclude that if there is a base class named `Product`, we can derive the `Beverage`, `Chocolate`, `Biscuit`, etc. classes from it and can implement the specific attributes of these derived classes in them.

Let's derive and use the `Beverage` class in code:

```
// Base Class Product
class Product {

  // Private Fields: Common attributes of all type of products
  private string _name;
```

```
  private double _price;
  private string _expiryDate;



  // Parameterized Constructor
  public Product(string name, double price, string expiryDate) {
    this._name = name;
    this._price = price;
    this._expiryDate = expiryDate;

  }

  // public method to print details
  public void PrintDetails() {
    Console.WriteLine("Name: " + this._name);
    Console.WriteLine("Price: " + this._price);
    Console.WriteLine("Expiry Date: " + this._expiryDate);
  }

}

// Derived Class Beverage
class Beverage : Product {

  // Private fields : Fields specific to the derived class
  private double _litres;
  private string _flavor;

  // Parameterized Constructor
  public Beverage(string name, double price, string expiryDate, double litres, string flavor)
    : base(name, price, expiryDate) //calling parent class constructor
  {
      this._litres = litres;
      this._flavor = flavor;
  }

  public void BeverageDetails() {  //details of Beverage
    PrintDetails();         //calling inherited method from parent class
    // Printing fields of this class
    Console.WriteLine("Litres: " + this._litres);
    Console.WriteLine("Flavor: " + this._flavor);
  }

}

class Demo {

  public static void Main(string[] args) {
    Beverage cola = new Beverage("RC Cola", 0.7, "8/12/2019", 0.35, "Cola"); //creation of Be
    cola.BeverageDetails(); //calling method to print details
  }

}
```

In the code above, ignore the `base` keyword on **line 36** for now. We are simply passing the arguments to the parent constructor. You will get to know more

about this in the future.

Notice how we were able to utilize the fields of the base `Product` class by initializing them through the object of the derived `Beverage` class at **line 54**. The base class already has a method, `PrintDetails()`, for displaying the *name, price and expiry date*, we delegate display of that information to that method on **line 43**. This is code reuse, which is a useful property of inheritance.

> **Note:** In C#, a class can extend (inherit) from only one other class at a time and a class cannot extend itself.

Access Modifiers

In the chapter **Classes and Objects**, lesson: Access Modifiers, we discussed the basic and most used types of access modifiers, `private` and `public`. However, there are four remaining types of access modifiers that we have yet to discuss but are rarely used in our code.

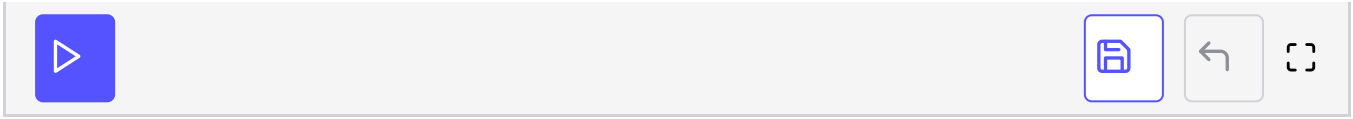
Let's discuss these briefly:

Protected

The accessibility of the members declared with the `protected` access modifier is limited to the class containing it and the classes which derived from this class.

```
class Product { // Base Product class
  // Protected field
  protected double price = 50;
  // Protected method
  protected void Print() {
    Console.WriteLine("Hi! I am the Print() method from the Product class");
  }
}

class Beverage : Product { // Derived Beverage class

    public void Access() {
      // Accessing the protected field
      Console.WriteLine("I can Access price: " + price);
      // Accessing the protected method
      Print();
    }
}

class Demo {
```

```
class Demo {

    public static void Main(string[] args) {
      var cola = new Beverage();
      cola.Access();
    }

}
```



In the code above, the `price` field is declared protected on **line 3** and is accessed in the derived class on **line 14**.

It is very rare that we will ever have to use the following types of access modifiers in our code when developing an application, as they don't guarantee good encapsulation of our classes. However, we will just take a quick look into these.

Internal

The accessibility of the class members declared as `internal` is limited to the current assembly only, i.e., they cannot be accessed from any other assembly in our code.

Protected Internal

The accessibility of the class members declared as `protected internal` is limited to any code in the assembly in which they are declared or from within a derived class in another assembly.

Private Protected

The accessibility of the class members declared as `private protected` is limited to the code in the same class or a derived class only inside the assembly which they are declared in.

Let's move on to the description of a very important C# keyword: `base`.