codecademy

Learn C#: Interfaces and Inheritance

C# Interface

In C#, an *interface* contains definitions for a group of related functionalities that a class can implement. Interfaces are useful because they guarantee how a class behaves. This, along with the fact that a class can implement multiple interfaces, helps organize and modularize components of software.
It is best practice to start the name of an interface with "I".

```
interface IAutomobile
{
  string LicensePlate { get; }
  double Speed { get; }
  int Wheels { get; }
}

// The IAutomobile interface has three
properties. Any class that implements this
interface must have these three
properties.

public interface IAccount
{
  void PayInFunds ( decimal amount );
  bool WithdrawFunds ( decimal amount );
  decimal GetBalance ();
}

// The IAccount interface has three
methods to implement.

public class CustomerAccount : IAccount
{ }

// This CustomerAccount class is labeled
with : IAccount, which means that it will
implement that interface.
```

C# Inheritance

In C#, *inheritance* is the process by which one class inherits the members of another class. The class that inherits is called a *subclass* or *derived* class. The other class is called a *superclass*, or a *base* class.
When you define a class that inherits from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class can thereby reuse the code in the base class without having to re-implement it. In the derived class, you can add more members. In this manner, the derived class extends the functionality of the base class.

```
public class Honeymoon : TripPlanner
{ }

// Similar to an interface, inheritance
also uses the colon syntax to denote
a class inherited super class. In this
case, Honeymoon class inherits from
TripPlanner class.

// A derived class can only inherit from
one base class, but inheritance is
transitive. That base class may inherit
from another class, and so on, which
creates a class hierachy.
```

C# protected Keyword

In C#, a protected member can be accessed by the current class and any class that inherits from it. This is designated by the `protected` access modifier.

```
public class BankAccount
{
  protected decimal balance = 0;
}

public class StudentAccount : BankAccount
{
}

// In this example, the BankAccount
(superclass) and StudentAccount (subclass)
have access to the balance field. Any
other class does not.
```

C# override/virtual Keywords

In C#, a derived class (subclass) can modify the behavior of an inherited method. The method in the derived class must be labeled `override` and the method in the base class (superclass) must be labeled `virtual`.
The `virtual` and `override` keywords are useful for two reasons:

1. Since the compiler treats “regular” and virtual methods differently, they must be marked as such.
2. This avoids the “hiding” of inherited methods, which helps developers understand the intention of the code.

```
class BaseClass
{
  public virtual void Method1()
  {
    Console.WriteLine("Base - Method1");
  }
}

class DerivedClass : BaseClass
{
  public override void Method1()
  {
    Console.WriteLine("Derived
- Method1");   }
}
```

C# abstract Keyword

In C#, the `abstract` modifier indicates that the thing being modified has a missing or incomplete implementation. It must be implemented completely by a derived, non-abstract class.
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.
If at least one member of a class is `abstract`, the containing class must also be marked `abstract`.
The complete implementation of an `abstract` member must be marked with `override`.

```
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;
}

// In this example, GetArea() is an
abstract method within the abstract Shape
class. It is implemented by the derived
class Square.
```