C# OOP Cheat Sheet (1 of 4)

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is a programming paradigm that employs objects to encapsulate code. Objects consist of types and are called classes. A class is just a template for an object which is an instance of the class, which occupies memory. When we say that a class is instantiated, we mean that an object in memory has been created. Classes contain data and executable code. Everything in C# and .NET is an object. In the menu View ➤ Object Browser.

Programming Principles

DRY is an acronym for Don't Repeat Yourself. In OOP, encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof: (1) A language mechanism for restricting direct access to some of the object's components. (2) A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data. In OOP, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Simple Class Declaration

The simplest class declaration is:

```
class Foo { }
```

```
[ public | protected | internal | private ]
[ abstract | sealed | static ]
class class_name [:class/interfaces inherited from ]
```

A `class` is a data structure that can store data and execute code. It contains data members and function members. The members can be any combination of nine possible member types. A local variable is a variable declared inside a function member. On the Internet are the StyleCop Rules Documentation the ordering of members in classes. Note: Whenever you have a class, such as our Customer, and inside that class you have a **List** of objects of any type, you should always initialize that list to an empty list with object initializer syntax: { };

Static

Static classes are meant to be consumed without instantiating them. Static classes can be used to group members that are to be available throughout the program. A static class must have all members marked as static as well as the class itself. The class can have a static constructor, but it cannot have an instance constructor. Static classes are implicitly sealed, meaning you cannot inherit from a static class. A non-static instantiable class can have static members which exist and are accessible even if there are no instances of the class. A static field is shared by all the instances of the class, and all the instances access the same memory location when they access the static field. Static methods exist. Static function members cannot access instance members but can access other static members. Static members, like instance members, can also be accessed from outside the class using dot-syntax notation. Another option to access the member doesn't use any prefix at all, if you have included a using static declaration for the specific class to which that member belongs:

```
Using static System.Console;
```

Abstract Classes

A class declared as abstract can never be instantiated. Instead, only its concrete subclasses can be instantiated. Abstract classes can define abstract members which are like virtual members, except they don't provide a default implementation. That implementation must be provided by the subclass, unless that subclass is also declared abstract.

Sealed Classes

A sealed class cannot be used as the base class for any other class. You use the sealed keyword to protect your class from the prying methods of a subclass. Static classes are implicitly sealed.

Instance Constructors

For classes, the C# compiler automatically generates a parameterless public constructor if and only if you do not define any constructors. However, as soon as you define at least one constructor, the parameterless constructor is no longer automatically generated, so you may need to write it yourself.

Instance constructors execute when the object is first instantiated. When an object is destroyed the destructor is called. Memory is freed up at this time. Constructors are called with the new keyword.

Constructors can be static. A static constructor executes once per type, rather than once per instance. A type can define only one static constructor, and it must be parameterless and have the same name as the type.

```
public class Customer
{
    public string Name;  // in real world these are private
    public int Id;  // in real world these are private
    public Customer() { }  // constructor (same name as class)
    public Customer(int id) // constructor
    {
        this.Id = id;  // set Id property
    }
    public Customer(int id, string name) // constructor
    {
        this.Id = id;  // 'this' references current object Customer
        this.Name = name;  // here we set Name property
    }
}
class Program
{
    static void Main(string[] args)
    {
        // ERROR: not contain constructor that takes zero arguments
        // unless we create OUR OWN parameterless constructor (we did)
        var customer = new Customer();
        customer.Id = 7;
        customer.Name = "John";
        Console.WriteLine(customer.Id);
        Console.WriteLine(customer.Name);
    }
}
```

Fields

A field is a variable that belongs to a class. It can be of any type, either predefined or user-defined. A field initializer is part of the field declaration and consists of an equal sign followed by an expression that evaluates to a value. The initialization value must be determinable at compile time. If no initializer is used, the compiler sets the value of a field to a default value, determined by the type of the field. The default value for each type is 0 and is false for bool. The default for reference types is null. Readonly variable values are assigned t runtime.

```
class Order
{
    public int Id;
}
class Customer
{
    public int Id;
    public string Name;
    public readonly List<Order> Orders = new List<Order>();
    // Note: no parameterless constructor for Customer
    public Customer(int id)
    {   // a constructor
        this.Id = id;  // the keyword this is redundant
    }
    public Customer(int id, string name) : this(id)
    {   // a constructor
        this.Name = name; // the keyword this is redundant
    }
    public void DoSomething() { } // just an example method

}
class Program
{
    static void Main(string[] args)
    {
        var customer = new Customer(3, "Bob");
        customer.Orders.Add(new Order());
        customer.Orders.Add(new Order() { Id = 7 });
        Console.WriteLine("Customer Id: " + customer.Id + " Name: "
            + customer.Name);
        Console.WriteLine("Num orders: " + customer.Orders.Count);
        foreach (var ord in customer.Orders) { Console.WriteLine("O
rder Id: " + ord.Id); }
    }
}
```

Here is the Console output of he above program. Notice that the first order Id below is zero because zero is the default.

```
Customer Id: 3 Name: Bob
Number of orders: 2
Order Id: 0
Order Id: 7
```

Generally, you would use private fields with public properties to provide encapsulation.

Methods

A method is a named block of code that is a function member of a class. You can execute the code from somewhere else in the program by using the method's name, provided you have access to it. Below is the simplest way to write a method inside a class, which are usually named using PascalCase (first letter of each word is capitalized).

```
class NotAnything { void DoNothingMethod() { } }
```

You can also pass data into a method and receive data back as output. A block is a sequence of statements between curly braces. It may contain local variables (usually for local computations), flow-of-control statements, method invocations, nested blocks or other methods known as local functions.

```
[access modifier]
[static|virtual|override|new|sealed|abstract]
method name (parameter list) { body }
```

C# allows for **optional** parameters which you can either include or omit when invoking the method. To specify that, you must include a default value for that parameter in the method declaration. Value types require the default value to be determinable at compile time, and reference types only if the default value is null. The declaration order must be all required (if any) – all optional (if any) – all params (if any).

C# OOP Cheat Sheet (2 of 4)

Access Modifier	Description
public	Fully accessible. This is the implicit accessibility for members of an enum or interface.
private	Accessible only within the containing type. This is the default accessibility for members of a class or struct. Perhaps you have a method that is implementation detail that calculates something.
protected	Accessible only within the containing type or subclasses (derived classes). May be a sign of bad design.
internal	Accessible only from the same assembly. We create a separate class library and use internal. How? Right-click Solution ➤ Add ➤ New Project ➤ Class Library (DLL). We'll need to add a Reference (Project, Add Reference) and add using statement.
protected internal	Not used normally! Accessible only from the same assembly or any derived classes. The union of protected and internal.

virtual – method can be overridden in subclass.
override – overrides virtual method in base class.
new – hides non-virtual method in base class.
sealed – prevents derived class from inheriting.
abstract – must be implemented by subclass.
Below is an example of a method called MyMethod (()

```
public class MyClass
{
    public int MyMethod (int integer, string text)
    {
        return 0;
    }
}
```

Properties

A property is declared like a field, but with a get/set block added. Properties look like fields from the outside, but internally they contain logic, like methods do. You can set the values of a public field and a public property, no problem. Note that `-=` means subtract from self.

```
class Program
{
    static void Main(string[] args)
    {
        Item it = new Item();
        it.FieldPrice = 24.67M;
        it.PropertyPrice = 45.21M;
        Console.WriteLine(it.FieldPrice + " " + it.PropertyPrice);
        it.FieldPrice -= 1.00M;
        it.PropertyPrice -= 1.00M;
        Console.WriteLine(it.FieldPrice + " " + it.PropertyPrice);
    }
}
public class Item
{
    public decimal FieldPrice;
    public decimal PropertyPrice { get; set; }
}
```

Here is a public property Amount with its backing field, that can be simplified with auto implemented property with `{ get; set; }`.

```
private decimal _amount; // backing field
public decimal Amount   // public property
{
    get { return _amount; }
    set { _amount = value; } // notice the keyword value
}
```

The `get` and `set` denote property **accessors**. The `set` method could throw an exception if value was outside a valid range of values.

Object Initializer Syntax

C# 3.0 (.NET 3.5) introduced Object Initializer Syntax, a new way to initialize an object of a class or collection. Object initializers allow you to assign values to the fields or properties at the time of creating an object without invoking a constructor.

```
class Program
{
    public class Person
    {
        public int id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
    }
    static void Main(string[] args)
    {   // don't need to initialize all fields
        var p = new Person {FirstName = "J", LastName = "Smith"};
        Console.WriteLine("Last name is {0}", p.LastName);
        // OUTPUT: Last name is Smith
    }
}
```

If the Person class had a constructor that initializes the LastName field, and the same field in the object initializer was initialized, the assignment in the LastName field in the object Initializer wins.

Collection Initializer Syntax works similarly as follows in the Main():

```
var p1 = new Person { Id = 1, FirstName = "John" };
var listP = new List<Person>()
{  // the List is being initialized
    p1,
    new Person {Id = 2, FirstName="Jackie"},
    new Person {Id = 3 },
};
foreach (var p in listP)
{
    Console.WriteLine(p.Id + " " + p.Name);
}
var listEmpty = new List<Person> { }; // initialize to an Empty list
```

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are like properties but are accessed via an index argument rather than a property name. The string class has an indexer that lets you access each of its char values via an int Index.

```
string s = "hello";
Console.WriteLine(s[0]); // 'h' zero-based
Console.WriteLine(s[1]); // 'e'
Console.WriteLine(s[99]); // IndexOutOfRangeException
Console.WriteLine(s[-1]); // IndexOutOfRangeException
```

The index argument(s) can be of any type(s), unlike arrays. You can call indexers null-conditionally by inserting a question mark before the square bracket as shown below.

```
string str = null;
Console.WriteLine(str?[0]); // Writes nothing; no error.
Console.WriteLine(str[0]); // NullReferenceException
```

To write an indexer, define a property called `this`, specifying the arguments in square brackets.

```
class Sentence
{
    string[] words = "The quick brown fox".Split(); //field
    public Sentence() { }  // default constructor
    public Sentence(string str) // constructor
            { words = str.Split(); }
    public int Length  // property
            { get { return words.Length; } }
    public string this[int wordNum] // indexer
    {
        get { return words[wordNum]; }
        set { words[wordNum] = value; }
    }
}
static void Main(string[] args)
{
    string s = "hello world";
    Console.WriteLine(s[0]); // 'h' zero-based
    Console.WriteLine(s[5]); // ' '
    string str = null;
    Console.WriteLine(str?[0]); // Writes nothing; no error.
    // Console.WriteLine(str[0]); // NullReferenceException

    Sentence sen = new Sentence();
    Console.WriteLine(sen[1]); // quick
    sen[3] = "wildebeest";  // replace the 4th word
    Console.WriteLine(sen[3]); // wildebeest
    for (int i=0;i<sen.Length;i++) { Console.Write(sen[i] + "|"); }
    // now use our constructor to use our sentence
    Sentence sent = new Sentence("The sleeping black cat");
    Console.WriteLine(sent[1]);  // sleeping
}
```

You have your own class `Customer` with fields `FirstName` and `LastName`. Instantiate it as Cust1. Get the first name and last name with Cust1.FirstName and Cust1.LastName. Indexers allow you to do the same with Cust1[0] and Cust1[1] respectively. An indexer is a pair of get and set accessors inside the code block of `ReturnType this [ Type param1, ... ]`. The `set` and `get` blocks use `switch`. An indexer allows an instance of a class to be indexed like an array.

Inheritance

Inheritance is a type of relationship ("Is-A") between classes that allows one class to inherit members from the other (code reuse). A horse "is an" animal. Inheritance allows for polymorphic behaviour. In UML, the Animal is at the top with the Horse under it with an arrow pointing up to Animal. Another example of inheritance is where a Saving Bank Account and Chequing Bank Account inherit from a Bank Account.

```
public class BaseClass
{
    public int Amount { get; set; }
    public BaseClass() { Console.WriteLine("Base constr"); }
    public void BaseDo() { Console.WriteLine("Base's BaseDo."); }
    public virtual void Do() { Console.WriteLine("Base's Do"); }
}
public class SubClass : BaseClass
{
    public SubClass() { Console.WriteLine("Sub constr"); }
    public override void Do() { Console.WriteLine("Sub's Do");}
}
class Program
{
    static void Main(string[] args)
    {
        var bas = new BaseClass();
        var sub = new SubClass();
        sub.Amount = 1;  // Amount inherited from Base
        sub.Do(); // Sub's Do
    }
}
```

Output:

```
Base constr
Base constr
Sub constr
Sub's Do
```

Constructor Inheritance

When you instantiate a sub class, base class constructors are always executed first, then sub class constructors, as you can see in lines 2 and 3 from the output. Base class constructors are not inherited.

C# OOP Cheat Sheet (3 of 4)

Composition (aka Containment)

Composition is a type of relationship ("has -a") between two classes that allows one class to **contain** another. Inheritance is another type of relationship. Both methods give us code re-use. In our example, both the car and truck have an engine and the engine needs to send a message to the console. We use a private field in the **composite** class (car and truck) to achieve this. You use a member field to hold an object instance. Generally, inheritance results in a more tightly-couple relationship than composition and many developers prefer composition, but it depends on your project. Two things to remember: **private field** and **constructor**.

```
using System;
namespace CompositionGeneral
{
    class Car
    {
        private readonly Engine _engine;
        public Car(Engine engine)  // constructor
        {
            _engine = engine;
        }
        public void DriveCar()
        {
            float speed = 0.0F;
            _engine.EngineStatus("car starting engine");
            speed = 50.0F;


          _engine.EngineStatus($"speed of {speed} Km/hr");
            _engine.EngineStatus("car engine off");
        }
    }
    class Truck
    {
        private readonly Engine _engine;
        public Truck(Engine engine)  // constructor
        {
            _engine = engine;
        }
        public void DriveTruck() { //...
        }
    }
    class Engine  // the car and truck "Have An" engine
    {
        public void EngineStatus(string message)
        {
            Console.WriteLine("Engine status: " + message);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var e = new Engine();
            var sedan = new Car(e);
            sedan.DriveCar();
            var pickup = new Truck(new Engine());
            pickup.DriveTruck();

        }
    }
}
```

Instead of having the car and truck contain a concrete class, like Engine, what if we used an interface, like IEngine instead? Please see the section on **Interfaces & Extensibility** for an example of this.

Composition vs Inheritance

Designing classes needs to be done carefully. Be careful with designing your inheritance because it can result in large hierarchies that become fragile (difficult to modify due to tight coupling). You can always re-factor inheritance into composition. A horse and a fish are both animals, but they are quite different. Both eat and sleep (Animal class) but horses walk and fish swim. You could use composition and create a CanWalk and CanSwim class. The horse "has-a" CanWalk class. This is fine even though "has-a" may not make sense or sound correct in the real world. You don't want to put a Walk() method in your Animal class unless you are certain all of your animals now and in the future can walk. If you have that, using inheritance, you may need a sub-class of Animal called mammal, and re-compile and re-deploy your code. Also, with composition we get an extra benefit that's not possible with inheritance: **Interfaces**. We can replace our Animal class with an interface IAnimal. This is **dependency injection** and is covered later in the topic called Interfaces & Extensibility.

Method Overriding & Polymorphism

Method overriding is changing the implementation of an **inherited** method that came from the base class. Use the **`virtual`** keyword in the method of the base class and **`override`** in the derived class. `Virtual` is just gives us the opportunity to `override`. You don't have to override it. What is polymorphism? Poly means many and morph means form. Let's use an example with classes called `BaseClass`, `ChildRed` and `ChildBlue`.

```
class MyBaseClass
{
    public int CommonProperty { get; set; }
    public virtual void WriteMessage() { }
}
class ChildRed : MyBaseClass
{
    public new int CommonProperty { get; set; }
    public override void WriteMessage()
    {
        CommonProperty = 46;
        Console.WriteLine("Red " + CommonProperty);    }
}
class ChildGreen : MyBaseClass
{
    public override void WriteMessage()
    {
        Console.WriteLine("Green " + CommonProperty);
    }
}
class Display
{
    public void WriteMyMessages(List<MyBaseClass> baseclasses)
    {
        foreach (var bc in baseclasses)
        {
            bc.WriteMessage();
        }
    }
}
```

When we call `WriteMessage()` above we have polymorphic behavior. We have a list of different colors, but the implementation is different for each colour. Red and Green overrode the base class's method. Notice that the list is a list of the base class `MyBaseClass.`

```
class Program
{
    static void Main(string[] args)
    {
        var baseclasses = new List<MyBaseClass>();
        baseclasses.Add(new ChildRed() { CommonProperty = 1 });
        baseclasses.Add(new ChildGreen() { CommonProperty = 3 });
        var display = new Display();
        display.WriteMyMessages(baseclasses);
    }
}
```

You can assign a variable that is of a derived type to a variable of one of the base types. No casting is required for this. You can then call methods of the base class through this variable. This results in the implementation of the method in the derived class being called. You can cast a base type variable into a derived class variable and call the method of the derived class.

Interfaces

An interface is like a class, but it provides a specification (contract) rather than an implementation for its members. Interface members are all implicitly abstract. A class (or struct) can implement **multiple interfaces** but a **class can inherit from only a single class**, and a struct cannot inherit at all (aside from deriving from `System.ValueType`). The interface's members will be implemented by the classes and structs that implement the interface. By convention, interface names start with the capital letter "I". If a class implements an interface, it must implement **all** the members of that interface. An interface declaration can contain only declarations of the following kinds of nonstatic function members: Methods, Properties, Events or Indexers. Interfaces can inherit interfaces.

```
interface IInfo
{
    string GetName(); string GetAge();
}
class CA : IInfo
{   // declare that CA implements the interface IInfo
    public string Name;  public int Age;
    // implement two interface methods of IInfo:
    public string GetName() { return Name; }
    public string GetAge() { return Age.ToString(); }
}
class CB : IInfo
{   // declare that CB implements the interface
    public string First;  public string Last;
    public double PersonsAge;
    public string GetName() { return First + " " + Last; }
    public string GetAge() { return PersonsAge.ToString(); }
}
class Program
{   // pass objects as references to the interface
    static void PrintInfo(IInfo item)
    {
        Console.WriteLine("Name: {0}  Age: {1}",  item.GetName() ,
                                          item.GetAge() );
    }
    static void Main()
    {
        // instantiate using object initialization syntax
        CA a = new CA() { Name = "John Doe", Age = 35 };
        CB b = new CB() { First = "Jane", Last = "Smith",
                                           PersonsAge = 44.0 };
        // references to the objects are automatically
        // converted to references
        // to the interfaces they implement (in the code below)
        PrintInfo(a);
        PrintInfo(b);

        Type myType = typeof(Program);
        Console.Title = myType.Namespace;
    }
}
```

Output:

```
Name: John Doe  Age: 35
Name: Jane Smith  Age: 44
```

C# OOP Cheat Sheet (4 of 4)

Interfaces & Extensibility

We create a constructor and inject a dependency, which is called **dependency injection**. In the constructor we are specifying the dependencies of our class. The FilesProcessor is not directly dependent on the ConsoleLogger. It doesn't care who implements ILogger. It could be a DatabaseLogger that does it. Four parts in bold for easier reading.

```
public interface ILogger
{
    void LogError(string message);  // method
    void LogInfo(string message);  // method
}
public class ConsoleLogger : ILogger
{   // ConsoleLogger implements ILogger
    public void LogError(string message)
    {  // Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(message);
    }
    public void LogInfo(string message)
    {
        Console.WriteLine(message);
    }
}
public class FilesProcessor
{  // FilesProcessor is dependent on an interface.
    private readonly ILogger _logger;
    public FilesProcessor(ILogger logger) // constructor
    {
        _logger = logger;
    }
    public void Process()
    {
        try
        {   // we might employ the using keyword in the real world
            _logger.LogInfo($"Migrating started at {DateTime.Now}");
            _logger.LogInfo($"In middle of doing stuff...");
            int zero = 0;
            int myError = 1 / zero;
            _logger.LogInfo($"Migrating ended at {DateTime.Now}");
        }
        catch
        {
            _logger.LogError($"Opps! Error");
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        // Our logger to sends to console
        var filesProcessor = new FilesProcessor(new ConsoleLogger());
        filesProcessor.Process();
        Console.WriteLine("done program.");
    }
}
```

Output:

```
Migrating started at 2019-02-07 10:05:43 AM
In middle of doing stuff...
Opps! Error
done program.
```

We can **extend** it. We can create more loggers other than ConsoleLogger, such as DatabaseLogger, FileLogger, EmailLogger SMSLogger and so on. All we need to do is change the Main(). Note that you can have the FilesProcessor depend on more than one interface if needed.

Add a property to the **ILogger**. Must be a property - no fields allowed.

```
bool Colourize { get; set; } // property
```

ConsoleLogger implements Colourize this way:

```
public bool Colourize { get; set; } // no default
```

We changed the ConsoleLogger method this way:

```
public bool Colourize { get; } = true; // default is true
public void LogError(string message)
{
    if (Colourize) { Console.ForegroundColor = ConsoleColor.Red; }
    else { Console.ForegroundColor = ConsoleColor.White; }
    Console.WriteLine(message);
    …
}
```

The Main() program uses object initializer syntax this way:

```
var FilesProcessor = new FilesProcessor(new ConsoleLogger() {
Colourize = false }); // uses object initializer syntax
```

or this way since Colourize has a default value of true, initialization is not necessary if you want Colourize to be true.

```
var FilesProcessor = new FilesProcessor(new ConsoleLogger()});
```

Add Another Constructor (log to more than one location)

In the FilesProcessor:

```
private readonly ILogger _logger2;  // add this
public FilesProcessor(ILogger logger, ILogger logger2) // new ctor
{ _logger = logger; _logger2 = logger2;  }
```

In the Process() method add lines like this:

```
if (_logger2 != null) { _logger2.LogInfo($"Processing started at
{DateTime.Now}"); }
```

In the Main() you can now log **to the console and a file,** or just to the console or just to a file.

```
var filesProcessor = new FilesProcessor(new ConsoleLogger(), new
FileLogger("D:\\test\\InterfacesExtensibility4.txt"));
```

Interfaces & Testability

Using interfaces help with **unit testing**. We'll use the Microsoft Test Runner. You get a new journal entry and then post it passing the entry to the Post() method of the JournalPoster class. Posting the entry requires the services of the checker. We must use an interface for the checker.

```
class Program
{
    static void Main(string[] args)
    {
        var jp = new JournalPoster(new DrEqualsCrChecker());
        var je = new JournalEntry { DebitAmount = 120.50f,
                                    CreditAmount = -120.50f };
        jp.Post(je);
        Console.WriteLine("Posted? " + je.IsPosted);
        Console.WriteLine("Date posted: " +
            je.Posting.PostingDate.ToString("yyyy-MM-dd"));
        Console.WriteLine("Debit amount: {0:C}", je.DebitAmount);
        Console.WriteLine("Credit amount: {0:C}", je.CreditAmount);
        Console.WriteLine("JE Balance: {0:C}", je.Posting.Balance);
    }
}
```

Output:

```
Posted? True
Date posted: 2019-02-07
Debit amount: $120.50
Credit amount: -$120.50
JE Balance: $0.00
```

```
public class JournalEntry
{   // in the reaal world there is more than this
    public Posting Posting { get; set; }
    public float DebitAmount = 0f;
    public float CreditAmount = 0f;
    public DateTime DatePosted { get; set; }
    public bool IsPosted {
        get { return Posting != null; }
    }
}
public class Posting  {
    public float Balance { get; set; } // zero if Dr = Cr
    public DateTime PostingDate { get; set; }
}
public class JournalPoster
{   // this class does not even know about DrEqualsCrChecker
    private readonly IDrEqualsCrChecker _checker;
    public JournalPoster(IDrEqualsCrChecker checker)
        { _checker = checker; }
    public void Post(JournalEntry je)
    {
        if (je.IsPosted)
            throw new InvalidOperationException("Opps. Already posted!
");
        je.Posting = new Posting
        {
            Balance = _checker.CalcBalance(je),
            PostingDate = DateTime.Today.AddDays(1)
        };
    }
}
public interface IDrEqualsCrChecker
    { float CalcBalance(JournalEntry je); }
public class DrEqualsCrChecker : IDrEqualsCrChecker
{
    public float CalcBalance(JournalEntry je)
    {
        var balance = je.DebitAmount + je.CreditAmount;
        return balance;
    }
}
```

We need to test the JournalPoster's Post method. We need to isolate it so we can write code to test our code. Go to the Solution Explorer ➤ Right-Click Solution ➤ Add ➤ Project ➤ Visual C# ➤ Test ➤ Unit Test Project ➤ Name it after the Project and append **.UnitTests** to the name ➤ OK. You get the following by default. We'll change that.

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()  {  }
}
```

To test `JournalPoster()`, rename UnitTest1 to JournalPosterTests. Rename TestMethod1 following the naming convention of methodname_condition_expectation. Add a Reference to our Project in our UnitTest project; in that right-Click References ➤ Add Reference ➤ Projects ➤ click the check box of the project. Create a Fake debit equals credit checker because we don't want to pass the original one to the JournalPoster. Pass an always-working fake one to test JournalPoster.

```
[TestClass]
public class JournalPosterTests
{
    // need to add a Reference to our project
    [TestMethod]
    [ExpectedException(typeof(InvalidOperationException))]
    public void JournalPoster_JEIsAlreadyPosted_ThrowsAnException()
    {   // naming convention: methodname_condition_expection
        var JournalPoster = new JournalPoster(new FakeDrEqualsCrChecke
r());
        var je = new JournalEntry { Posting = new Posting() };
        JournalPoster.Post(je);
    }
    [TestMethod]
    public void JournalPoster_JEIsNotPosted_ShouldSetPostedPropertyOfJ
ournalEntry()
    {
        var JournalPoster = new JournalPoster(new FakeDrEqualsCrChecke
r());
        var je = new JournalEntry();
        JournalPoster.Post(je);
        Assert.IsTrue(je.IsPosted);
        Assert.AreEqual(1, je.Posting.Balance);
        Assert.AreEqual(DateTime.Today.AddDays(1), je.Posting.PostingD
ate);
    }
}
public class FakeDrEqualsCrChecker : IDrEqualsCrChecker
{   // methods defined in an interface must be public
    public float CalcBalance(JournalEntry je)
        { return 1; } // simple and it will works
}
```