Object-Oriented Programming and Classes

- Both structures and classes can be used to create objects.
- Most of the following discussion applies to both structures and classes except that structures cannot use inheritance – see the note on structures.

Object-Oriented Programming (OOP) Concepts

- *Traditional programming* is linear. An application starts at one point and ends at another. These applications are designed to solve specific problems; they generally do not adapt to new situations nor can they be easily modified to deal with future circumstances.
- Although traditional programming is simpler, *object-oriented programming* (OPP) is a better approach to developing software by providing more versatility for modifying code and more opportunity for reusing code. More work is required up front but it has the benefit of extensibility. OPP is data-oriented. The building block of OPP is objects which contain logically cohesive units of code. Object-Oriented Programming consists of two kinds of development: creating objects and writing programs that use objects.

Terminology

- **Class** or **Structure**: a template used to define the characteristics and behaviors of an object(s)

- **Object**: an instance (or copy) of a class or structure

  Analogy Note: the class/structure is like the blueprint or design of a house; the object is an actual instance of that house; there could be many houses built based on that one blueprint.

- **Fields**: the variables that provide *direct* uncontrolled access to an object(s) data

- **Properties**: the attributes or characteristics that provide *indirect* controlled access to an object(s) data

- **Methods**: the behaviors (actions) of an object(s)

- **Constructor**: special method(s) used to initialize an object – usually assigning values to the fields

- **State**: all the data stored in an object instance makes up the state of that object

- **User-defined type:** a class or structure written by a programmer (also known as an **Abstract Data Type**)

- **Members**: everything declared or defined at the class/structure level (fields, constructors, properties, methods, etc.)

- **Instance variables:** non-static fields

- **Class variables:** static fields

- **Instantiation:** the act of creating an object from a class/structure (typically by using the new operator)

- **Instance Member:** a copy of a method and/or variable used by an object (each object will have their own copies)

- **Helper Methods**: private methods that are called by other methods in the same class to help perform a given task

Pillars of OOP

1. **Information Hiding**: Keeping data private and allowing that data to be altered in ways that you can control.

   Note: When you build an application and use objects you know how to interact with the object(s) by calling well-written public methods or properties, but you are unaware of the low-level details of how the methods and properties actually work.

2. **Encapsulation**: logically grouping data and/or methods into cohesive units of code
   - classes and structures that will be instantiated into objects

3. **Interfaces:** a list of methods and properties that a class must define. In this way several implementing classes will have a common interface.

4. **Inheritance**: Allows you to create a class that inherits the characteristics and behaviors of another already existing class. The new class is known as the **derived** class and the existing or parent class is known as the **base** class. This concept cannot be applied to structures.

   Benefits
   - better code organization
   - eliminates code redundancy
   - easier to understand
   - facilitates code re-use (Why re-invent the wheel?)
   - reduces errors and debugging time (inherited classes are already tested)

5. **Containment:** A class/structure declares an instance of another class/structure. One class or structure contains another.

6. **Polymorphism (many forms)**: Method execution is determined at run-time (flexible) instead of compile-time (efficient). Multiple classes can be written that implement the same interface or derived classes that contain the same method that is implemented differently. Programs are written to execute a specific interface to an object – thus allowing the programs to interface with future objects not yet created. The lifespan of these applications is longer because they can adapt to new code (objects) without having to be re-worked.

   Ways to implement
   - Method Overriding: one class overrides a method in another class thru Inheritance
   - Virtual Methods: a base class dictates the public interface (methods) that all subclasses must implement.
   - Interfaces: all classes must implement the same properties and/or methods

Class Concepts

Class Scope

internal (default): class is assessable within its namespace only
public: class is assessable to all namespaces

Notes:
- The majority of classes are **public** so that object instances can be used in any program you need to write. You may choose to declare a class internal because that class will only be used in the current program, OR you may declare a class internal because it serves as a helper class to other public classes.
- You cannot declare a class to be private, protected, or protected internal.

Other Class Modifiers

abstract: class cannot be declared – only extended
sealed: class cannot be extended – only declared

Notes:
- **Abstract** classes serve as base classes and cannot be declared. Base classes define the general characteristics, behaviours, and actions of all other classes within the same hierarchy. Abstract classes can contain abstract methods. Abstract methods must be defined by all derived classes. Abstract methods ensure that all objects have the same method signature(s). This is one way to enable polymorphic programming – programs that can communicate with any object that has the abstract method defined. As new classes are added to the hierarchy, existing programs require no re-work since the new objects will have the same interface or abstract method.
- **Sealed** classes can only be declared and used as is. This ensures the programmer that the methods, properties etc cannot be changed thru inheritance.

Examples:

```
public class RegularClass { … }
class InternalClass { … } OR internal class InternalClass { … }
public abstract class BaseClass { … }
public sealed class DerivedClass { … }
```

Class Members:

Each class is to be cohesive by defining data and methods that make sense for that class. This is called encapsulation. The public properties and methods provide intuitive access to the object's interface. This is called information hiding.

Member Scope Modifiers:

private (default): can be accessed within the class it is declared
public: can be accessed by any class in any namespace
protected: can be accessed by any derived class from any namespace thru inheritance
internal: can be access by any class within the same namespace
protected internal: can be access by any class within the same namespace
OR any derived class from any namespace thru inheritance

Notes:

- The above modifiers are used when declaring variables, properties, and methods within a class. Modifiers are never used when declaring local variables. Local variables are by default private to the block of code in which they are declared.
- When defining your object classes the data (fields) are almost always declared private or protected to prevent direct access from programs. The properties and methods are declared public to make them available to the programs that use the objects. Internal members (variables or methods) are usually declared as part of a program you are writing rather than an object you are defining.

Other Member Modifiers:

static: class members shared by all object instances; the value of static variables can be changed
const: class variables shared by all object instances; the value is a constant and must be assigned when it is declared; the value cannot be changed
readonly: the value can be set once after the declaration but cannot be changed afterwards

Note:

- The **static** modifier is used to declare variables or fields and methods that belong to the class. When this modifier is not used the variables or fields and methods belong to the object instance(s). The **const** and **readonly** modifiers are used on variable declarations only. The static and readonly modifiers can be used by themselves or combined. The const modifier is never combined with the other two since const variables are already static and readonly.

Instance Field Examples:

```
private int number;
private readonly string code;
protected DateTime date;
```

Class Field Examples:

```
private static int count;
public readonly static string CompanyName;
public const decimal GST = 0.06M;
```

The **CompanyName** variable is shared text data and can be access from any class. Once the value is assigned (by a constructor) it cannot change. There is only one copy in memory regardless of how many objects are declared.

The Typical Class Structure for Defining Objects may contain:

- Instance data or fields
- Class data or fields (static)
- Constructors (usually overloaded)
- Instance Property
- Class Property (static)
- Instance Methods
- Class Methods (static)

Polymorphic Methods:

abstract: declared inside abstract classes only; derived classes must implement these methods because they are missing
a method body.
virtual: derived classes can choose to implement these methods; **ToString** and **Equals** are examples of virtual methods
that programmers usually define with new implementations

Abstract and virtual methods are declared in base classes and then overridden in derived classes. You use the override keyword to define both types of methods in a derived class.

Base Class Polymorphic Method Example:

```
public abstract double CalculatePay();
     //no body


public virtual void ClassInfo()
{
        …//original method body
}
```

Derived Class Polymorphic Method Example:

```
public override double CalculatePay()
{
        …//method implementation
}
public override void ClassInfo()
{
        …//new method body
```

Note:
- Constructors, property methods, and other methods are usually declared public when defining objects, however, the other scope modifiers can be used for specific purposes. For example a private method may support the object’s class code, but is not meant to be called directly by a program that uses the object.
- Regular methods, properties, and constructors can be any scope; however, virtual and abstract methods cannot be private since private methods cannot be overridden.

Creating & Using Classes

- Classes are used to encapsulate related data (fields) and the actions (methods) that can be performed on the fields.
- Classes can serve two purposes:
  - Template Class: A nonstatic class that is used to define objects.
  - Static Class: A class defined with the keyword *static*. All members of a static class are also defined as static. These classes are used just to group related fields and methods together without the need to create objects. The *Math* class is a static class that groups math related fields (E and PI) and methods (Pow and Tan) together.
- All classes inherit from the **Object** class – this class provides some methods that the programmer can use or override in their classes.

Adding a Class to a Project

- Classes can be added to the file that defines you form – but this could make for a long file.
- Typically, you add a new file to your project that will define some or all of the classes the programmer will create.
- To add a class as a separate file:
  - Go to the **Project** menu and select either **Add Class** or **Add New Item** – both selections will bring up the **Add New Item** dialog window.
  - In the **Add New Item** dialog, select either **Class** (for files that will contain a single class) or **Code File** (for files that will contain multiple classes and/or structures). Give the file an appropriate name.
  - After the file is added, it will show up in the solution explorer window. Classes and/or structures can now be added.
  - *Note:* If the file or class has already been coded but is not yet part of the current project, select the **Add Existing Item**.

Defining and Creating a Class

- Classes are defined and created much like structures – see the note on Structures. They can contain everything a structure can contain – fields, properties, methods, and constructors.
- An instance of a class (an object) is created by using the *new* keyword followed by a call to a constructor.
- Defining a class differs from defining a structure in 4 ways:
  - The keyword **class** is used instead of **struct**.
  - Fields in a class can be initialized when declared (only static/constant fields can be initialized in a structure).
  - The default parameterless constructor can be overridden in a class (in a structure it can't).
  - Since classes can use the concept of *inheritance*, members of a class can use the additional access modifiers *protected* and *internal protected* – see the note on class concepts and OOP.
- The keywords *abstract* and *sealed* apply to classes that will be inherited so they will be discussed in the notes on inheritance.

General Form:

```
class ClassName    // if no access modifier specified, classes are internal by default
{
    // declare class level variables (fields) and constants

    // declare constructor(s) – optional but necessary if you want the private fields to be initialized with non-default values when the class
    is instantiated

    // declare properties – recommended for private fields

    // declare methods – optional
}
```

- *Note:* To create a *static* class, add the keyword *static* before the *class* keyword. All fields, properties, and methods will also require the use of the keyword *static*. Static classes do not define constructors since no objects will ever be created from a static class!

Properties and Fields

- The same concepts discussed with structures apply with classes.
- The only difference is that all fields can be initialized when declared.

Constructors

- Constructors are special methods that are typically used to initialize the private fields within a class.
- To create a constructor:
    - Start with the access modifier *public* so that any code from any project can create an instance of the class the constructor is in.
    - Constructors do not return data – ever – so there is no data type listed after the access modifier.
    - The constructor name is *always* the same name as the class it is in.
    - Constructors have a parameter list like other methods – zero, one, or more parameter can be listed.
    - Constructors can be overloaded just like other methods – they have the same name but a different parameter list.

General Form for a Constructor:

```
public ClassName( parameter list )
{
    // initialize appropriate fields
    // based on the parameters in the constructor header
}
```

- Like structures, C# provide a default parameterless constructor that will initialize all nonstatic fields to a default value:
    - Numerical Types: default to zero
    - Booleans: default to false
    - Object Types: default to *null* (for a string, null is like the empty quotes – "")
- Unlike structures, classes can **override** the default parameterless constructor – just create your own parameterless constructor and C#'s default will be replaced.

Example: Overriding C#'s Default Constructor

```
public ClassName()
{
    // initialize appropriate fields to default values that make sense for this class
}
```

Methods

- Add methods to a class just as you did earlier for your form classes – all previous method concepts discussed apply to any classes created by the programmer.
- Most methods will be *public*, very few methods will be declared *private*. Those methods declared *private* can only be accessed by the class they are defined in so they are typically known as *helper methods*.
- The keywords *abstract* and *virtual* apply to methods that will be inherited so they will be discussed in the notes on inheritance

Overriding ToString( ) and the *this* Reference

- All classes inherit from Microsoft's base class called *Object* – this leads to a number of powerful things you can do with classes – most of which will be covered in the notes on inheritance.
- The Object has a couple of methods that can be *overridden* – the most commonly overridden *Object* class method is **ToString( )**.
  - The ToString( ) method from the Object class returns a string containing the name of the namespace followed by the name of the class – which is usually not very useful.
  - By *overriding* the ToString( ) method, the programmer can return a string that contains something more relevant for a particular class.
  - To *override* an inherited method, you must use the keyword **override** – see example below.

- The keyword **this** is frequently used inside a class and refers the current instance of the class. The *this* keyword is used only in nonstatic (template) classes – for static classes, the name of the class is used instead.
  - Usually, the use of the *this* keyword is optional – some programmers like to use it when access any member of the class they are currently coding.
  - The *this* keyword is required when a field and a method level variable exist that have the same name. When inside the method, using the variable name alone accesses the method level variable. To access the class level variable (field) from within the method, use the keyword *this* followed by the dot operator, followed by the variable name.

*Example: Using **this** and Overriding the Inherited ToString( ) Method*

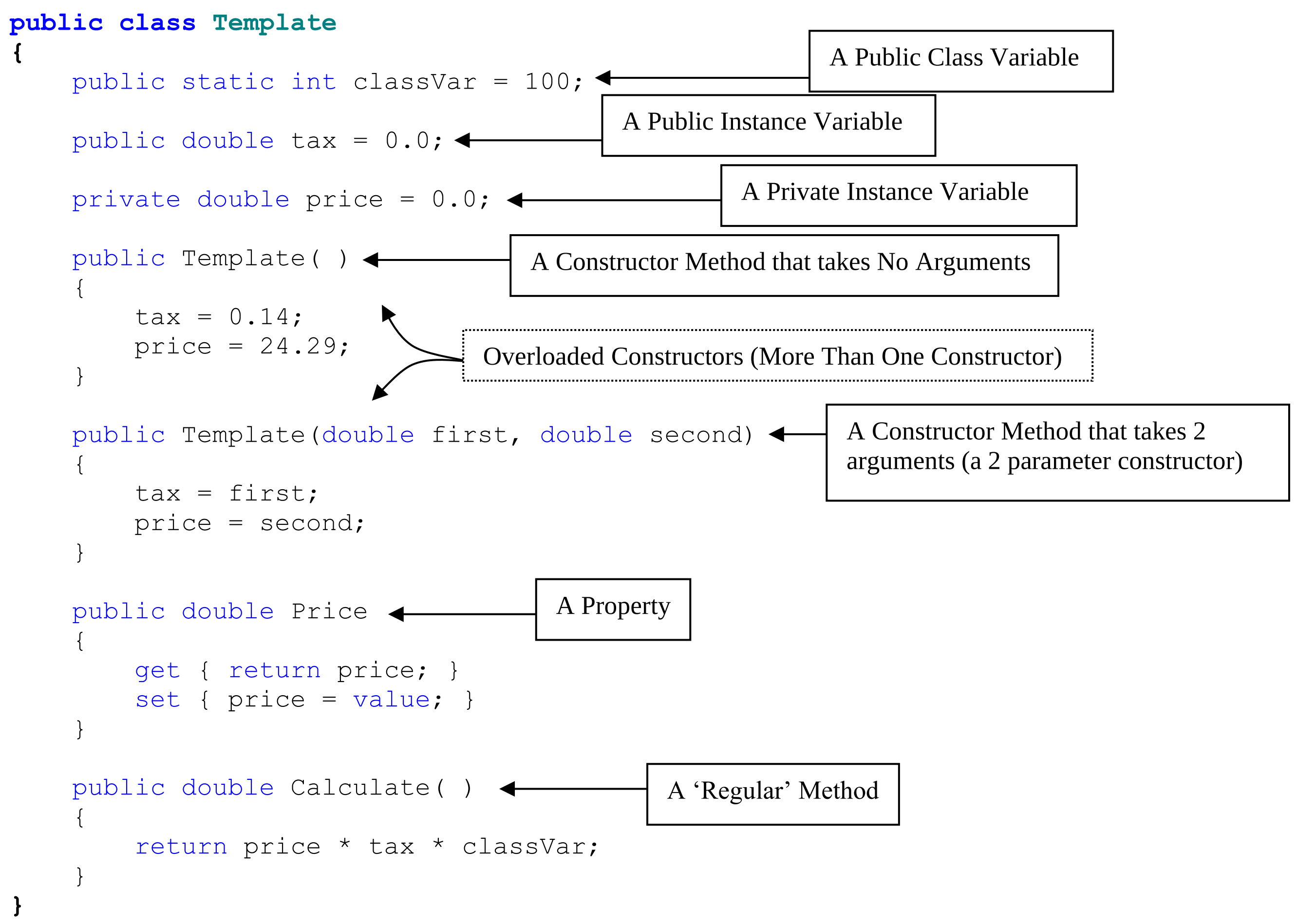
```
public class Instructor
{
    private string firstName;
    private string lastName;

    // overloaded constructors
    public Instructor()
    {
        // using this is optional – no other variables share these names from this location
        this.firstName = "Unknown";
        this.lastName = "Unknown";
    }
    public Instructor(string firstName, string lastName)
    {
        // notice the names of the parameter variables – parameter variables are method level variables
        // using this is not an option if you want to access the class level variables (fields) of the same name
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // this method originally came from the Object class – which all classes inherit from
    // the use of the override keyword – along with the same name and parameter list – overrides (replaces) the original
    public override string ToString()
    {
        // instead of returning NamespaceName.ClassName, the ToString( ) method will return the full name of the instructor
        // using this is optional – no other variables share these names from this location
        return this.firstName + " " + this.lastName;
    }
}

… Creating and Using the Instructor Class …
// this is an event handler method inside a form class
private void Display_Click(object sender, EventArgs e)
{
    // create 2 Instructor objects by calling both constructors
    Instructor teacher1 = new Instructor();   // using the parameterless constructor
    Instructor teacher2 = new Instructor("Cliff", "Patrick");   // using the 2 parameter constructor

    // call the overridden ToString() method for both Instructor objects
    txtTeacher1.Text = teacher1.ToString();   // displays:  Unknown Unknown
    txtTeacher2.Text = teacher2.ToString();   // displays:  Cliff Patrick
}
```

Note: It is important to test your classes…

- Create an object using every constructor – for each constructor, create objects using both good and bad data
- Test each property and method – again, use both good and bad data
- Your classes should be able to elegantly handle bad data situations!

Basic Class Examples

Example 1 – Identify Template Class Members and Use Them



```
public class Template
{
    public static int classVar = 100;

    public double tax = 0.0;

    private double price = 0.0;

    public Template( )
    {
        tax = 0.14;
        price = 24.29;
    }

    public Template(double first, double second)
    {
        tax = first;
        price = second;
    }

    public double Price
    {
        get { return price; }
        set { price = value; }
    }

    public double Calculate( )
    {
        return price * tax * classVar;
    }
}
```

A Public Class Variable
A Public Instance Variable
A Private Instance Variable
A Constructor Method that takes No Arguments
Overloaded Constructors (More Than One Constructor)
A Constructor Method that takes 2 arguments (a 2 parameter constructor)
A Property
A 'Regular' Method

******************************** **Console Test** ***************************************

```
static void Main(string[] args){
   // create 2 Template objects
   Template temp1 = new Template();    Template temp2 = new Template(0.07, 2.99);

   // display the value of each variable for the first object
   Console.WriteLine(Template.classVar);
   Console.WriteLine(temp1.tax);
   Console.WriteLine(temp1.Price);

   // display the value of each variable for the second object
   Console.WriteLine(Template.classVar);
   Console.WriteLine(temp2.tax);
   Console.WriteLine(temp2.Price);

   // call the Calculate method for each object
   Console.WriteLine(temp1.Calculate());
   Console.WriteLine(temp2.Calculate());
}
```

Output:

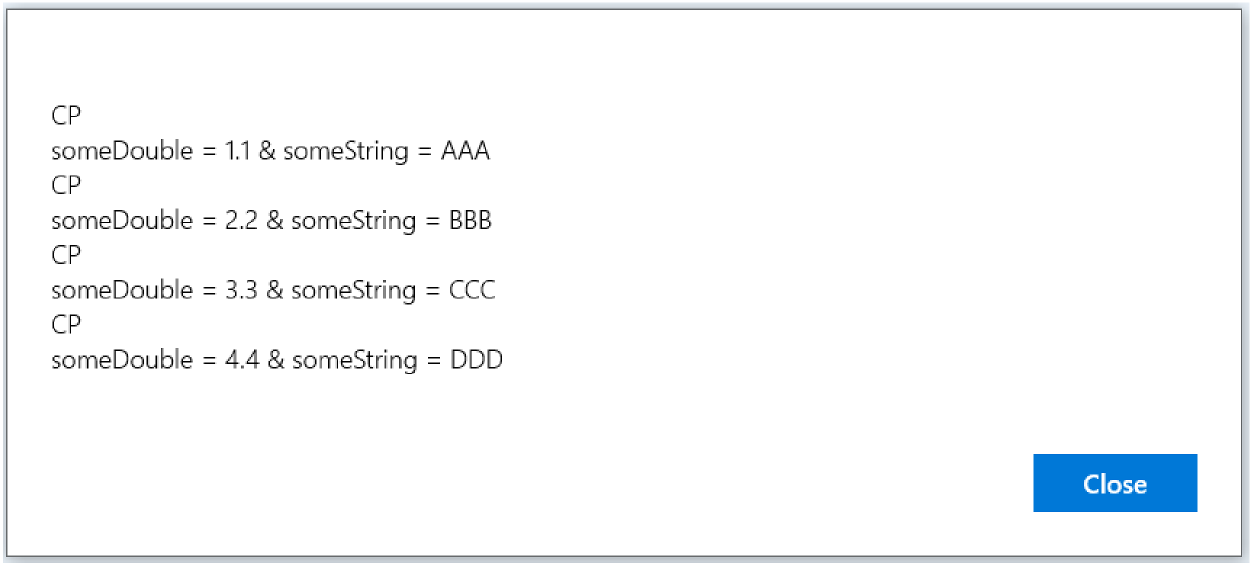
```
100
0.14
24.29
100
0.07
2.99
340.06
20.93
```

Example 2 – Focus on Template Class Constructors

```
public class Template
{
    // instance variables
    private double someDouble;
    private string someString;
    private char first, last;

    // 4 overloaded constructors
    public Template( )
    {
        someDouble = 1.1;
        someString = "AAA";
        Initialize( );  // a call to the private method below
    }
    public Template(double anotherDouble)
    {
        someDouble = anotherDouble;
        someString = "BBB";
        Initialize( );
    }
    public Template(string someString)
    {
        someDouble = 3.3;
        this.someString = someString;
        Initialize( );
    }
    public Template(double someDouble, string someString)
    {
        this.someDouble = someDouble;
        this.someString = someString;
        Initialize( );
    }

    // a private 'helper' method
    private void Initialize( )
    {
        first = 'C';
        last = 'P';
    }

    // a public method
    public string Display()
    {
        return "\n" + first + last + "\nsomeDouble = " +
                  someDouble + " & someString = " + someString;
    }
}
```

************************************** UWP Test **

```
private void Button_Click(object sender, EventArgs e)
{
   // declare 4 instances of a Template object
   Template temp1 = new Template();
   Template temp2 = new Template(2.2);
   Template temp3 = new Template("CCC");
   Template temp4 = new Template(4.4, "DDD");

   // call the Display() method for each object
   string display = temp1.Display() + "\n" + temp2.Display() + "\n" +
                    temp3.Display() + "\n" + temp4.Display();
   MessageDialog msg = new MessageDialog(display);
   msg.ShowAsync();
}
```



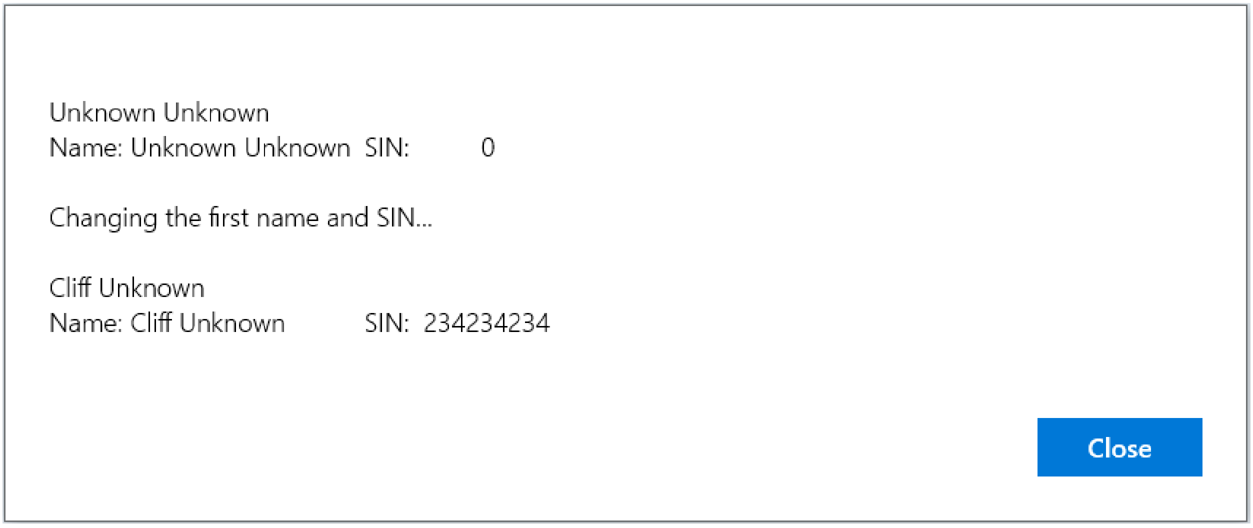
```
CP
someDouble = 1.1 & someString = AAA
CP
someDouble = 2.2 & someString = BBB
CP
someDouble = 3.3 & someString = CCC
CP
someDouble = 4.4 & someString = DDD
```

Example 3 – Focus on Template Class Properties

```
namespace Properties
{
   class Template
   {
      // instance variables
      private string first = "Unknown";
      private string last = "Unknown";
      private int sin;

      // properties
      public string FirstName
      {
         // read-write property
         get { return first; }
         set
         {
            if(value.Length > 0)
               first = value;
            else
               first = "unknown";
         }
      }
      public string LastName
      {
         // read-only property
         get { return last; }
      }
      public int Sin
      {
         // write-only property
         set
         {
            if(value  > 0)
               sin = value;
            else
               sin = 111111111;
         }
      }
      // method(s)
      public string GetInfo( )
      {
         return String.Format("Name: {0} {1}\tSIN: {2,10}\n",
                              FirstName, LastName, sin);
      }

    }// class
   }// namespace
```

************************* UWP Test ***************************

```
private void Button_Click(object sender, EventArgs e)
{
   // create a Template object - using the default constructor
   Template temp = new Template();

   // Display -> Change -> Display again - the user's information
   string display = temp.FirstName + " " + temp.LastName + "\n" + temp.GetInfo();

   display += "\nChanging the first name and SIN...\n\n";
   temp.FirstName = "Cliff";
   temp.Sin = 234234234;

   display += temp.FirstName + " " + temp.LastName + "\n" + temp.GetInfo();
   MessageDialog msg = new MessageDialog(display);
   msg.ShowAsync();

}
```



```
Unknown Unknown
Name: Unknown Unknown  SIN:        0

Changing the first name and SIN...

Cliff Unknown
Name: Cliff Unknown           SIN:  234234234
```