Classes and Objects

Class

A **class** is a data structure or a blueprint that defines data and actions in a single unit. It defines dynamically created **instances** of the class, also known as **objects**. It is defined using the keyword class. The program below shows the declaration of a class:

```
public class Point
        //instance variables
        public int xPos, yPos;
        //constructor Point
        public Point (int x, int y) {
                this.xPos = x;
                this.yPos = y;
        }

        public void displayPosition() {
                Console.WriteLine("xPos = " + this.xPos + " yPos = " + this.yPos);
        }
}
```

Instances of classes are created using the **new** keyword which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following shows the general syntax for creating an instance of a class:

```
ClassName objectName = new ClassName(arguments_list);
```

arguments_list refers to the actual value passed on the constructor or method. For example, the statements below create two (2) objects and store references to those objects in two (2) variables:

```
Point p1 = new Point(15, 18);
Point p2 = new Point(10, 20);
```

A created object can directly access the members of the class that are declared as public using dot (.) operator. For example, the objects p1 and p2 can directly access the public instance variables xPos and yPos and can access its methods.

```
int positionX = p1.xPos;
p1.displayPosition();
```

Encapsulation

Encapsulation is the process of hiding or encapsulating data from the outside world. A **class** or **structure** can specify how accessible each of its members is to code outside of the class or structure. Methods and variables that are not intended to be used outside of the class or structure can be hidden to limit the potential for coding errors. Then the members of class or structure can only be accessed or updated by calling the respective methods through the object.

Encapsulation can be implemented by using the following:

- Defining the modifier of class data members, like instance variables, as **`private`** – This will make sure that data members of the class are not directly accessible from an object instance.
- Accessing the private data members through the class methods or by using properties – The properties use a pair of public mutator (set) and accessor (get) methods to manipulate private data members of a class. The following shows the general syntax of a property:

```
access_modifier datatype propertyName {
        get {
                //get accessor code
        }
        set {
                //set mutator code using value keyword
        }
}
```

The following class Point shows an example where its instance variables are encapsulated using the **`private`** keyword and on how to create property.

Example 1:

```
public class Point {
       //instance variables
       private int xPos, yPos;

       //constructor
       public Point() {
              this.xPos = 0;
              this.yPos = 0;
       }

       //Using properties (set and get methods) in get, use return, while in set, use the value keyword
       public int xPoint {
              get {
                     return this.xPos;
              }
              set {
                     this.xPos = value;
              }
       }

       public int yPoint {
              get {
                     return this.yPos;
              }
              set {
                     this.yPos = value;
              }
       }

}
```

In the given example, the **return** keyword is used in the **get** accessor to return the value of the property. The **value** keyword used in **set** mutator represents the value being assigned to the property. In the example, the property names **xPoint** and **yPoint** can be used in expressions and assignments. After using a property name, the respective get and set methods are automatically invoked. The following statements show how to use properties:

```
Point p1 = new Point();
//this following initialization automatically calls the set method
p1.xPoint = 20;
p1.yPoint = 10;
//accessing the property automatically calls the get method and return its value
int x = p1.xPoint;
Console.WriteLine("value of p1.xPoint is " + x);
Console.WriteLine("value of p1.yPoint is " + p1.yPoint);
```

Output:

```
value of p1.xPoint is 20
value of p1.yPoint is 10
```

In the given example, **xPoint** and **yPoint** are properties and have **get** and **set** methods. While using the statement **p1.xPoint = 20;**, the set method is called, and the value **20** is assigned to the parameter **value** keyword – from which it is assigned to the data member of **Point** class which is **xPos**. While using the **p1.xPos**, it calls the get method, returning the value in the data member of the class which is **xPos**.

The *this* Keyword

The ***this*** keyword refers to the object or the current instance of the class. This can be used within class members. The use of `this` keyword is the capability to pass a reference from the current object instance to a method. The following example demonstrates how the `this` keyword represents the class instance being referenced:

```
public class RectArea {
       private int width, height;p
       public void setValues(int w, int h) {
              this.width = w;
              this.height = h;
       }

       public int getRectArea() {
              return this.width * this.height;
       }
}
```

The `this` keyword helps resolve ambiguity when the name of the data members and parameters are the same.

Constructors

A **constructor** is a special method where its name is the same as the name of its type (class or structure). Constructors are used to ensure the initialization of instance variables when an object is created. The following is the general form for defining constructors:

```
access_modifier ClassName (parameter_list) {
       //statement(s) to initialize object
}
```

The method signature of a constructor only includes the modifier, name, and its parameter list. Constructors cannot return any value. The access modifier of a constructor should be set to `public` because it is called from outside of the class. The following class contains an example of constructor declaration:

```
public class Person {
       private string firstName;
       private string lastName;
       private int age;

       public Person() {
              this.firstName = "no name";
              this.lastName = "no name";
              this.age = 0;
       }
}
```

In the given example, the constructor `Person()` is declared and initializes the instance variables with values. The following statement shows how to create an object using the constructor `Person()`:

```
Person p1 = new Person();
```

The object **p1** is defined using the **Person()** constructor. Then, the constructor executes automatically and perform its statements to initializes the instance variables of object p1.

Overloading Constructors

Constructors can be overloaded like all methods. A class can have one (1) or more constructors with same name, as long as they have a different set of parameters. Constructors with parameters or overloaded constructors are useful in directly initializing instance variables and states with defined values to the objects at the time of creation. The following is an example of overloaded constructors and how to invoke them depending on the required arguments:

```
public class Person {
      private string firstName;
      private string lastName;
      private int age;

      public Person() {
            this.firstName = "no name";
            this.lastName = "no name";
            this.age = 0;
      }

      public Person(string fName, string lName) {
            this.firstName = fName;
            this.lastName = lName;
            this.age = 0;
      }

      public Person (string newfName, string newLName, int newAge) {
            this.firstName = newfName;
            this.lastName = newLName;
            this.age = newAge;
      }
}
```

Overloaded constructors can be used to initialize the instance variables of an object when defining a new object.

```
Person p1 = new Person();
Person p2 = new Person("Jack", "Paul");
Person p3 = new Person("Elizabeth", "Cruz", 25);
```

When an overload constructor is called in C#, the Common Language Runtime (CLR) automatically selects which constructor to execute depending on the defined set of arguments.

Namespace

A **namespace** provides a logical grouping to organize related classes, structures, interfaces, and other types. Namespaces help in controlling the scope of classes and method names in larger applications. A defined namespace can be imported or used in any other program to use the classes, its methods, and its other content readily to avoid repetitive statements. The following shows the general form of defining a namespace using the **namespace** keyword.

```
namespace name {
      //namespace members
}
```

The name is a defined name for namespace. The members of a namespace must be declared within a pair of curly braces. These members can be classes, structures, or interfaces. In the given example below, the class Person is a member of the namespace CustomNamespace.

```
using System;
//start of namespace CustomNamespace
namespace CustomNamespace
{
    public class Person
    {
        private string firstName;
        private string lastName;
        private int age;

        public Person()
        {
            this.firstName = "no name";
            this.lastName = "no name";
            this.age = 0;
        }
        public Person(string fName, string lName)
        {
            this.firstName = fName;
            this.lastName = lName;
            this.age = 0;
        }

        public Person(string newfName, string newLName, int newAge)
        {
            this.firstName = newfName;
            this.lastName = newLName;
            this.age = newAge;
        }

        public void displayInfo() {
            Console.WriteLine("First name: " + this.firstName);
            Console.WriteLine("Last name: " + this.lastName);
            Console.WriteLine("Age: " + this.age);
        }
    }
} //end of namespace CustomNamespace
```

The members of a namespace can be accessed using the **using** keyword and must appear at the top of the program. The following example shows how to import the namespace CustomNamespace:

```
using CustomNamespace;
```

When a namespace is imported, all of its public members can be accessed by the class where it is declared. The following example shows how to implement the using keyword on a class:

```
using System;
using CustomNamespace;

namespace ConsoleApp
{
    public class Program
    {
        static void Main(string[] args)
        {
            //The class Person is a member of the namespace CustomNamespace
            Person p1 = new Person();
            Person p2 = new Person("Jack", "Paul");
            Person p3 = new Person("Elizabeth", "Cruz", 25);
            p3.displayInfo();
            //The class Console is a member of the namespace System
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

Output:

```
First name: Elizabeth
Last name: Cruz
Age: 25
Press any key to exit...
```

REFERENCES:

Deitel, P. and Deitel, H. (2015). *Visual C# 2012 how to program* (5th Ed.). USA: Pearson Education, Inc.

Gaddis, T. (2016). *Starting out with visual C#* (4th Ed.). USA: Pearson Education, Inc.

Harwani, B. (2015). *Learning object-oriented programming in C# 5.0*. USA: Cengage Learning PTR.