**C# OOP essentials**

**Part I – C# essentials**

**Chapter 1 – Basic terminology**

Very much like JAVA, C# is a strictly object-oriented language, so all the codes will be contained within class files.

A C# program consists of the following parts:

* Namespace declaration
* A class
* Class methods
* Class attributes
* A Main method
* Statements and Expressions

A complete example of C# class is shown below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

// namespace is like a folder in which contains all the classes

namespace MyCircle

{

class Circle // define a class, default modifier for class is internal

{

// define a Class Property/Attributes or Member Variable

private int \_circleDiameter;

// constructors. Constructors have the same name as the class // a default constructor

public Circle()

{

// Do nothing

}

// we can also make a constructor by ourselves - this is called orverloading a constructor

// adding a access modifier such as private in front of a constructor // is a common mistake

public Circle(int defaultCircleDiameter)

{

if (defaultCircleDiameter >= 0)

{

\_circleDiameter = defaultCircleDiameter;

}

else

{

\_circleDiameter = 0;

}

}

// Class Property Accessor

public int Diameter

{

set { \_circleDiameter = value; }

get { return \_circleDiameter; }

}

// Method. In defining a method, we should always take care of 1. the access modifier, in this case it's public; 2. the return type, in this case it's void.

public void DrawCircle(Graphics drawingArea, int x, int y)

{

drawingArea.DrawEllipse(new Pen(System.Drawing.Color.Red), x, y, \_circleDiameter, \_circleDiameter);

}

}

}

**Forms and Controls**

A Form is a container object for holding other controls that will allow the user to interact with the application. In Studio, the Toolbox contains the controls that may be placed on a Form. A form is a class in C#, so the naming convention is the Pascal case. Whenever you create or add new Forms and add controls, name the Forms and controls before you begin working with them and making modifications.

Double-clicking a Button in the Visual Editor automatically opens the C# Code Editor, and places your cursor inside an event handler for the Button. This event handler is the code that will execute or "fire" when the Button is clicked.

A form is a class, containing all the properties, methods and events. A button in a form is a method in the corresponding class.

**Namespace**

Namespace enables you to manage your code without worrying about class name collisions. Namespace is like a folder in which contains all the classes.

**Classes**

A class is a blue print for an object. A class may contain components such as properties, methods, and events.

A class, just like a blueprint, is not an actual object, but a definition for an object. In order to create an actual object, you have to create an **instance** of the class, and hte instance is called an **object**.

**Class Properties / Attributes / Member Variables**

Variables are attributes / properties or data members of a class. They are used for storing data. In the example, the Circle class has one member variable named \_circleDiameter.

**Class Property Accessors**

Use get and set to access the class properties. For properties, we normally use getters and setters to get and set the value of the property.

Newcomers to OOP often ask why it’s necessary to go through so much work to get and set properties. Couldn’t you just create public instance variables that the user could read and write to directly? The answer lies in one of the fundamental tenets of OOP: **encapsulation**. Encapsulation means that the client code does not have direct access to the data. When working with the data, the client code must use clearly defined properties and methods accessed through an instance of the class.

**Constructors**

A constructor class method is a method that is used when you first create an instance of an object.

eg: myCircle = new Circle();

You'll notice that Circle() includes parentheses. Does that mean that we're calling a

method of the Circle class? Yes, we are! We're calling a constructor! Every C# class must have a constructor, and if one is not explicitly written in the code, a default constructor is provided when the project is built. A default constructor contains no parameters, which matches the format of the Circle(): no parameters are included within the parentheses. Constructors have the same name as the class they're in.

Class constructor is the same as initialize method in Python. The only difference is:

In Python, we use \_\_init\_\_ function

In C#, the class constructor method is named the same as the class.

The class constructor method is named the same as the class. When an object instance of a class is instantiated by client code, the constructor method is executed.

**Methods**

In defining a method, we should always take care of 1. the access modifier, in the example it's public, which means that this method may be used by any other code within the same assembly, or another assembly that references it; 2. the return type, in the exmaple it's void.

Form is a class, and the buttons are considered methods in the class. Remember that Studio will generate the event handler method for a button automatically if you double-click on the button.

**Main method**

Main method is the entry point for all C# programs. The program execution starts at the Main method.

**Overloading Methods**

You overload methods in a class by defining multiple methods that have the same name but contain different signatures. A method signature is a combination of the name of the method and its parameter type list. The compiler will determine which method to execute by examining the parameter type list passed in by the client. Basically, overloading method means we define two or more methods with the same name. However, the parameters pass into the method should be different. A common technique in OOP is to overload the constructor of the class.

**Naming conventions**

* Forms: Pascal case, descriptive, commonly describing the purpose of the Form (MyCircle)
* Controls: Camel case, descriptive, ending with a declaration of the type of control (circleDiameterTextBox, createCircleButton)
* Classes: Pascal case, descriptive (Circle)
* Class properties (variables): Camel case, prefixed with underscore, descriptive (\_circleDiameter)
* Class methods: Pascal case, descriptive
* Method variables: Camel case, descriptive (diameter)
* Filenames: Match the contents (MyCircle.cs, Circle.cs)

**Part II – OOP essentials**

**Chapter 3 – The Three Pillars of OOP**

**1. Inheritance /ɪnˈherɪtəns/**

Classes can inherit from each other, which means the inheriting/derived class gets all of the behavior of the inherited class, also known as the base class.

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string GetFullName()

{

return FirstName + " " + LastName;

}

}

public class **Employee : Person**

{

public decimal Salary { get; set; }

}

The trick is in the Employee : Person part. That part basically says “**Employee inherits from Person**”. In this case we can call Person a **base class** or superclass and Employee a **subclass**. Another common way of saying it is that **Employee extends Person**.

**Override:** if we want to replace a method in base class, then we can override the method in base class and redefine a method in the inheriting class. We use **virtual** keyword in the base class and **override** keyword in the subclass.

Marking classes as **sealed** prevents these classes from being inherited.

Unfortunately, in C#, one class can only inherits one other class. Unlike C++, C# cannot support one class inherits multiple classes. An interface defines a contract that can be implemented by classes and structs. What if we want to inherit multi classes. Then we can use interface, as discussed in the next chapter.

**2. Encapsulation**

Encapsulation is the process of hiding the internal workings of our classes. That means we specify a public specification, used by consumers of our class, while the actual work is hidden away. The advantage is that a class can change how it works without needing to change its consumers.

In C# we have four **access modifiers** keywords which enable five ways of controlling code visibility:

* **Private**—only visible to the containing class. Private is the default modifier for everyting except for class.
* **Protected**—only visible to the containing class and inheritors.
* **Internal**—only visible to classes in the same assembly. Assembly could be understood as project. Internal is the default modifier for class. That means that a class, by defaut, is visable and accessible within the same namespace.
* **Protected internal**—only visible to the containing class and inheritors in the same assembly.
* **Public**—visible to everyone.

**3. Polymorphism /pɒlɪ'mɔːfɪz(ə)m/**

We’ve seen inheritance and that we can alter the behavior of a type through inheritance. Our Person class had a GetFullName method which was altered in the subclass Employee. We’ve also seen that whenever, at run-time, an object of type Person is expected we can throw in any subclass of Person, like Employee. This is called polymorphism. In the following example the PrintFullName method takes an object of type Person, but it prints “Rossel, Sander” because the parameter that’s passed into the method is actually of subtype Employee, which **overrides** the functionality of GetFullName.

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public **virtual** string GetFullName()

{

return FirstName + " " + LastName;

}

}

public class Employee : Person

{

public decimal Salary { get; set; }

public sealed **override** string GetFullName()

{

return LastName + ", " + FirstName;

}

}

class Program

{

static void Main(string[] args)

{

Person p = new Employee();

p.FirstName = "Sander";

p.LastName = "Rossel";

Console.WriteLine(p.GetFullName());

Console.ReadKey();

}

}

The Three Pillars of OOP are the foundation of object-oriented programming. They haven’t been implemented for nothing and they do solve real problems. It’s crucial that you know these features by heart and practice them in your daily code. **Think about encapsulation every time you create a class or method. Use inheritance when necessary, but don’t forget it brings extra complexity to the table as well. Be very wary of polymorphism**, know which code will run when you inherit classes and override methods.

**Chapter 4 – Interface**

An interface can contain methods, properties, events, and indexers, just like a class. But an interface doesn't provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ **multiple inheritance**. In the following example, the interface IComboBox inherits from both ITextBox and IListBox.

interface IControl

{

void Paint();

}

interface ITextBox: IControl

{

void SetText(string text);

}

interface IListBox: IControl

{

void SetItems(string[] items);

}

interface IComboBox: ITextBox, IListBox {}

Classes and structs can implement multiple interfaces. In the following example, the class EditBox implements both IControl and IDataBound

interface IDataBound

{

void Bind(Binder b);

}

public class EditBox: IControl, IDataBound

{

public void Paint() { }

public void Bind(Binder b) { }

}

**Chapter 4 – Asynchronos ⁄eɪˈsɪŋkrənəs⁄ Programming**

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/>

So our program, a collection of statements packaged in an EXE after it's complied, we take a program like that, load it into memory, and then we want to execute the statements in that program and we typically going to start in C# with static void main. Now, the thing that's running that code, we usually can call it a thread for now, so we have the operating system running our hardware within that operating system.

Asynchronos programing means more than one of the code will be excuting. We want to use Async when we don’t want to wait for my code to be excuted. That means we need more than one thread. So async programing requires **multi-threading**.

Thread safety when two threads can run through a piece of code safely at the same time, without interfering with each other.

The work still needs doing, so what we typically find is that, we can do things in parallel when it makes sense to do things at the same time especially when we have multiple processors, we have subsystems that can do more than one thing at a time, but it might not always be faster.

When we talk about blocking, or blocking I/O, or blocking calls, it just means that we're making a synchronous call. So we call into a method, our thread goes off, executes the statements in that method and comes back when it's done. So from my perspective as the caller, I'm blocked, I'm waiting. Method A calling method B, method A is blocked while method B is executing. Non-blocking thus, is what we're after in this course. **So method A calls method B, method A continues to run, cuz method B will deal with running it's code some other way, almost certainly on another thread.**

Below is an example of using Acycronos programing for I/O bound. You may need to download some data from a web service when a button is pressed, but don’t want to block the UI thread. It can be accomplished simply like this.

private readonly HttpClient \_httpClient = new HttpClient();

downloadButton.Clicked **+= async (o, e) =>**

{

// This line will yield control to the UI as the request

// from the web service is happening.

// The UI thread is now free to perform other work.

var stringData = **await** \_httpClient.GetStringAsync(URL);

DoSomethingWithData(stringData);

};

And that’s it! The code expresses the intent (downloading some data asynchronously) without getting bogged

down in interacting with Task objects.

**Chapter 4 – Delegate**

A delegate type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages. Unlike function pointers, delegates are object-oriented and type-safe.

The following example declares and uses a delegate type named Function.

using System;

**delegate double Function(double x);**

class Multiplier

{

double factor;

public Multiplier(double factor)

{

this.factor = factor;

}

public double Multiply(double x)

{

return x \* factor;

}

}

class DelegateExample

{

static double Square(double x)

{

return x \* x;

}

static double[] Apply(double[] a, **Function f**)

{

double[] result = new double[a.Length];

for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);

return result;

}

static void Main()

{

double[] a = {0.0, 0.5, 1.0};

double[] squares = Apply(a, Square);

double[] sines = Apply(a, Math.Sin);

Multiplier m = new Multiplier(2.0);

double[] doubles = Apply(a, m.Multiply);

}

}

Delgate is a key word in C#. Using delgate, we get some methods, which helps us to use asyc code. Thread.Sleep(n) means block the current thread for at least the number of timeslices (or thread quantums) that can occur within n milliseconds.

**Chapter 6 – Design Patterns**

**Singleton Pattern**

You are building an application in C#. You need a class that has only one instance, and you need to provide a global point of access to the instance. You want to be sure that your solution is efficient and that it takes advantage of the Microsoft .NET common language runtime features. You may also want to make sure that your solution is thread safe.

<https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650316(v=pandp.10)>