PREGRADO

UNIDAD 1 | OVERVIEW

# C# OOP FEATURES

**SI730 | Aplicaciones Web**

Al finalizar la unidad de aprendizaje, el estudiante describe el proceso de software realizado aplicando el paradigma orientado a objetos combinado con aspectos de la programación funcional, utilizando el lenguaje C# y frameworks de actualidad, para desarrollar aplicaciones web básicas en un ambiente de desarrollo colaborativo.

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

## Intro

C# es un lenguaje que permite escribir código bajo un enfoque orientado a objetos.

El object-oriented approach cobra relevancia al momento de desarrollar aplicaciones dado que brinda la posibilidad de reutilizar aplicaciones o partes de las mismas.

Los conceptos de OOP permiten que el código sea limpio y mantenible.

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

## class Keyword

Se define una clase con la instrucción **class**. La clase puede tener miembros. Se delimita con llaves.

```
public class Student
{
    private string _firstName;
    private string _lastName;

    public string GetFullName()
    {
        return _firstName + ' ' + _lastName;
    }
}
```

# Declarar e instanciar objetos

Se utiliza **new** Keyword.

```
class Program
{
    static void Main(string[] args)
    {
        Student student = new Student();
    }
}
```

## Constructores

El compilador genera uno por defecto en caso no se declare.

Constructor por defecto  (sin parámetros):

```
public class Student
{
    private string _firstName;
    private string _lastName;

    public Student()
    {
        _firstName = string.Empty;
        _lastName = string.Empty;
    }

    public string GetFullName()
    {
        return _firstName + ' ' + _lastName;
    }
}
```

# Constructores

Puede existir más de un constructor en la misma clase (overloading).

```
public class Student
{
    private string _firtName;
    private string _lastName;

    public Student()
    {
        _firstName = string.Empty;
        _lastName = string.Empty;
    }

    public Student(string firstName, string lastName)
    {
        _firstName = firstName;
        _lastName = lastName;
    }

    public string GetFullName()
    {
        return _firstName + ' ' + _lastName;
    }
}
```

# Partial classes

**partial** keyword permite descomponer la definición de una clase.

```
partial class Student
{
    private string _firstName;
    private string _lastName;

    public Student()
    {
        _firstName = string.Empty;
        _lastName = string.Empty;
    }
}
```

```
partial class Student
{
    public Student(string firstName,
                   string lastName)
    {
        _firstName = firstName;
        _lastName = lastName;
    }

    public string GetFullName()
    {
        return _firstName + ' ' +
          _lastName;
    }
}
```

Útil para elevar legibilidad y mantenibilidad.

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

# Properties

Encapsula acceso a un private field. Soporta bloques **get** y **set**.

```
public class Student
{
    private string _firstName;
    private string _lastName;

    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }

    public string LastName
    {
        get { return _lastName; }
        set { _lastName = value; }
    }
```

```
    public Student(string firstName,
                   string lastName)
    {
        _firstName = firstName;
        _lastName = lastName;
    }

    public string GetFullName()
    {
        return _firstName + ' ' +
         _lastName;
    }
}
```

Soporta Read-Only (sólo get) / Write-Only (sólo set) properties.

## Properties

Cuando no se requiere lógica adicional, puede utilizarse **auto-implemented properties**.

```
public string FirstName { get; set; }
public string LastName { get; set; }
```

El compilador crea para cada property un private field.

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

## Static methods

Métodos independientes de una instancia en particular, se asocian a la clase directamente. La definición incluye el **static** keyword.

```
int givenNumber = 4;
Console.WriteLine(Math.Sqrt(givenNumber));
```

## Constants

Utiliza el **const** keyword.

```
public const int Months = 12;
public const int Weeks = 52;
public const int Days = 365;
```

## Static class

Sólo puede contener static members. Puede tener un constructor por defecto, pero static.

```
public static class GivenValueHolder
{
    private static int _givenValue;

    static GivenValueHolder ()
    {
        _givenValue = 128;
    }
}
```

# Extension methods

Extienden un tipo existente con métodos estáticos adicionales.

```
public static class StringExtender
{
    public static string FirstLetterUpperCase(this string word)
    {
        char letter = Char.ToUpper(word[0]);
        string remaining = word.Substring(1);
        return letter + remaining;
    }
}

class Program
{
    static void Main(string[] args)
    {
        string word = "developer"
                      .FirstLetterUpperCase();

        Console.WriteLine(word);
        Console.ReadKey();
    }
}
```

Se crean dentro de una static class y se agrega al primer parámetro el prefijo **this**.

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

## Anonymous classes

Clase sin nombre, cuyas instancias son objetos anónimos.

Resulta útil por ejemplo en query expressions.

Se crea con **new** y llaves.

```
var anAnonymousObject = new { Name = "nesto", Age = 32 };
```

## Nullable types

C# ofrece el modificador **?** para indicar que un variable de un tipo soporta valor nulo.

La propiedad **HasValue** es verdadera si el elemento tiene un valor.

La propiedad **Value** brinda acceso al valor almacenado.

```
int? number = null;
number = 234; //it can receive a value with =

if(number.HasValue)
{
    Console.WriteLine(number.Value);
}
else
{
     Console.WriteLine("number is null");
}
```

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

## Structures

Son **value types**, mientras que las clases son reference types.

Pueden tener fields, methods y constructors.

Se declaran con el **struct** keyword.

```
public struct Time
{
    private int _hours, _minutes, _seconds;

    public Time(int hours, int minutes, int seconds)
    {
        _hours = hours;
        _minutes = minutes;
        _seconds = seconds;
    }

    public void PrintTime()
    {
        Console.WriteLine($"Hours: {_hours}, Minutes: {_minutes}, Seconds: {_seconds}");
    }
}
```

# Structures

Acceso a structures.

```
static void Main(string[] args)
{
    Time time = new Time(3, 30, 25);
    time.PrintTime();
    Console.ReadKey();
}
```

## Structures

No se puede declarar un constructor por defecto (sin parámetros).

Se puede inicializar fields vía un constructor, pero se debe inicializar todos los fields en dicho constructor.

La instancia de una estructura reside en el **stack**, mientras que la instancia de una clase reside en **heap memory**.

La estructura debe ser pequeña, simple e inmutable, de otro modo debe optarse por una clase.

# Enumerations

Permiten agrupar constantes relacionadas a un mismo concepto.

Asigna un valor numérico a cada elemento (0, 1, ...) por defecto integer.

```
public enum DaysInWeek
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

static void Main(string[] args)
{
    DaysInWeek monday = DaysInWeek.Monday;

    Console.WriteLine(monday); // Monday
    Console.WriteLine((int)monday); // 0
    Console.ReadKey();
}
```

# Enumerations

Puede modificarse el valor numérico asociado, o ajustar el tipo.

```
public enum DaysInWeek
{
    Monday=1,
    Tuesday,
    Wednesday,
    Thursday, Friday,
    Saturday,
    Sunday
}
```

```
public short DaysInWeek: short
{
    Monday=10,
    Tuesday=20,
    Wednesday=30,
    Thursday=40,
    Friday=50,
    Saturday=60,
    Sunday=70
}
```

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

# Inheritance

C# soporta herencia simple, aplicando el símbolo :

```
public class Writer
{
    public void Write()
    {
        Console.WriteLine("Writing to a file");
    }
}
public class XMLWriter: Writer
{
    public void FormatXMLFile()
    {
        Console.WriteLine("Formating XML file");
    }
}
public class JSONWriter: Writer
{
    public void FormatJSONFile()
    {
        Console.WriteLine("Formating JSON file");
    }
}
```

# Constructor

Se accede al constructor del padre con **:base**.

```
public class Writer
{
    public string FileName { get; set; }

    public Writer(string fileName)
    {
        FileName = fileName;
    }

    public void Write()
    {
        Console.WriteLine("Writing to a file");
    }
}
public class XMLWriter: Writer
{
    public XMLWriter(string fileName)
        :base(fileName)
    {
    }

    public void FormatXMLFile()
    {
        Console.WriteLine("Formating XML file");
    }
}
```

```
public class JSONWriter: Writer
{
    public JSONWriter(string fileName)
        :base(fileName)
    {
    }

    public void FormatJSONFile()
    {
        Console.WriteLine("Formating JSON file");
    }
}
class Program
{
    static void Main(string[] args)
    {
        XMLWriter xmlWriter = new XMLWriter("xmlFileName");
        xmlWriter.FormatXMLFile();
        xmlWriter.Write();
        Console.WriteLine(xmlWriter.FileName);

        JSONWriter jsonWriter = new JSONWriter("jsonFileName");
        jsonWriter.FormatJSONFile();
        jsonWriter.Write();
        Console.WriteLine(jsonWriter.FileName);
    }
}
```

# Typecast

Se utiliza el **as** keyword.

```
XMLWriter xml = new XMLWriter("any name");
Writer writer = xml; //writer references to xml

XMLWriter newWriter = writer as XMLWriter;
newWriter.FormatXMLFile(); //valid because writer was xml
```

# Method hiding

Ocultar método de clase padre con otra versión usando **new** keyword.

```
public class Writer
{
    public string FileName { get; set; }

    public Writer(string fileName)
    {
        FileName = fileName;
    }

    public void Write()
    {
        Console.WriteLine("Writing to a file");
    }

    public void SetName()
    {
        Console.WriteLine("Setting name in the base Writer class");
    }
}

public class XMLWriter: Writer
{
    public XMLWriter(string fileName)
        :base(fileName)
    {
    }

    public void FormatXMLFile()
    {
        Console.WriteLine("Formating XML file");
    }

    public new void SetName()
    {
        Console.WriteLine("Setting name in the XMLWriter class");
    }
}
```

# Method refinement

Se permite refinar un método en la clase padre con **virtual** keyword.

```
public class Writer
{
    public string FileName { get; set; }

    public Writer(string fileName)
    {
        FileName = fileName;
    }

    public void Write()
    {
        Console.WriteLine("Writing to a file");
    }

    public void SetName()
    {
        Console.WriteLine("Setting name in the base Writer class");
    }

    public virtual void CalculateFileSize()
    {
        Console.WriteLine("Calculating file size in a Writer class");
    }
}
```

# Method refinement

Se refina un método en la clase descendiente con **override** keyword.

```
public class XMLWriter: Writer
{
    public XMLWriter(string fileName)
        :base(fileName)
    {
    }

    public void FormatXMLFile()
    {
        Console.WriteLine("Formating XML file");
    }

    public new void SetName()
    {
        Console.WriteLine("Setting name in the XMLWriter class");
    }

    public override void CalculateFileSize()
    {
        Console.WriteLine("Calculating file size in the XMLWriter class");
    }
}
```

# Method refinement

En el método refinado se accede a método original con **base** keyword.

```
public class XMLWriter: Writer
{
    ...
    public override void CalculateFileSize()
    {
        base.CalculateFileSize();
        Console.WriteLine("Calculating file size in the XMLWriter class");
    }
}
```

## Polymorphic method rules

**Virtual methods** no pueden ser privados.

**Overriden methods** no pueden ser privados, una clase derivada no puede cambiar el nivel de protección de un método que hereda.

**Signatures** en **virtual & overriden methods** deben ser idénticas.

Sólo podemos aplicar **override** sobre un **virtual method**.

Implementar sin usar **override** implica **hiding** de método en el padre.

De desear ello debe usarse **new**.

**Overriden** methods son **virtual** implícitamente.

# AGENDA

INTRO

CLASSES & CONSTRUCTORS

PROPERTIES

STATIC MEMBERS, CONSTANTS, METHODS

ANONYMOUS & NULLABLE TYPES

STRUCTURES & ENUMERATIONS

INHERITANCE

INTERFACES, ABSTRACT & SEALED CLASSES

## Definir interfaz

Se define con **interface** keyword, especificando **members** sin implementación.

```
public interface IWriter
{
    void WriteFile();
}
```

## Implementar interfaz

Se declara la clase o estructura incluyendo el signo de herencia : y la interfaz, implementando todos los members.

```
public class XmlWritter: IWriter
{
    public void WriteFile()
    {
        Console.WriteLine("Writing file in the XmlWriter class.");
    }
}

public class JsonWriter: IWriter
{
    public void WriteFile()
    {
        Console.WriteLine("Writing file in the JsonWritter class.");
    }
}
```

## Implementar interfaz

En caso de herencia e implementación de interfaz, se declara primero la herencia.

```
public interface IWriter
{
    void WriteFile();
}
```

```
public class FileBase
{
    public virtual void SetName()
    {
        Console.WriteLine("Setting name in the base Writer class.");
    }
}
```

# Implementar interfaz

En caso de herencia e implementación de interfaz, se declara primero la herencia.

```
public class XmlWritter: FileBase, IWriter
{
    public void WriteFile()
    {
        Console.WriteLine("Writing file in the XmlWriter class.");
    }

    public override void SetName()
    {
        Console.WriteLine("Setting name in the XmlWriter class.");
    }
}

public class JsonWriter: FileBase, IWriter
{
    public void WriteFile()
    {
        Console.WriteLine("Writing file in the JsonWritter class.");
    }

    public override void SetName()
    {
        Console.WriteLine("Setting name in the JsonWriter class.");
    }
}
```

## Referencia

Se define un objeto usando una **interface variable**.

Si se declara un objeto de una interfaz, sólo puede acceder a los members de la interfaz.

```
XmlWriter writer = new XmlWriter();
writer.SetName(); //overridden method from a base class
writer.WriteFile(); //method from an interface

IWriter writer = new XmlWriter();
writer.WriteFile(); //method from an interface
writer.SetName(); //error the SetName method is not part of the IWriter interface
```

# Decouple classes

Las interfaces son útiles para evitar acoplamiento.

```
public class FileWriter
{
    private readonly IWriter _writer;

    public FileWriter(IWriter writer)
    {
        _writer = writer;
    }

    public void Write()
    {
        _writer.WriteFile();
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        XmlWriter xmlWriter = new XmlWriter();
        JsonWriter jsonWriter =
          new JsonWriter();

        FileWriter fileWriter =
          new FileWriter(xmlWriter);
        fileWriter.Write();

        fileWriter =
          new FileWriter(jsonWriter);
        fileWriter.Write();

        Console.ReadKey();
    }
}
```

# Abstract classes

Se utiliza **abstract** keyword.

```
public abstract class AbstractPerformer
{
    public abstract void Perform();
}
```

## Abstract classes

Se implementa el **abstract method** en clase derivada con **override** keyword.

```
public override void Perform ()
{
    //method implementation
}
```

## Sealed classes

Son clases con las cuales no se puede declarar relación de herencia. Se declaran utilizando **sealed** keyword.

```
public sealed class LeafClass
{
}
```

# RESUMEN

## Recordemos

C# ofrece soporte para el paradigma orientado a objetos.

Permite declarar clases que contienen members.

Las clases soportan single-inheritance, pero pueden implementar múltiples interfaces.

Soporta method hiding, overriding.

Soporte abstract & sealed classes.

# REFERENCIAS

## Para profundizar


Object-Oriented programming (C#)
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/object-oriented-programming

C# 101 Video Series
https://channel9.msdn.com/Series/CSharp-101/?WT.mc_id=Educationalcsharp-c9-scottha


# PREGRADO

**Ingeniería de Software**

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

**UPC**
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

***exígete, innova***