COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

C# Avanzado

Tipos de referencia – clases - **propiedades**

- Una **propiedad** es un tipo de método especial que sustituye a los Getters y Setters
- Nombrar con **PascalCase** (UpperCamelCase)
- Permiten **leer**, **escribir** o **calcular** valores de campos privados, de forma que su implementación siga siendo privada
- Para devolver el valor de una propiedad se usa el descriptor de acceso **get**, mientras que para asignarle un valor se usa el **set**
- Dentro del cuerpo del **set**, tenemos acceso a la palabra clave **value**, que contiene el valor que se quiere asignar
- Si no requieren un código personalizado, pueden implementarse automáticamente
- Más información

Tipos de referencia – clases - **propiedades**

```
class Fruit {
    private float _kilos; // Variable in memory

    public float Kilos { // Reading and writing
        get {
            return _kilos;
        } set {
            _kilos = value;
        }
    }

    public float KilosAlt { // Reading and writing
        get => _kilos;
        set => _kilos = value;
    }

    public float KilosReadOnly => _kilos; // Read Only

    public float Pounds { // Computed/calculated property
        get {
            return _kilos * 2.204f;
        } set {
            _kilos = value / 2.204f;
        }
    }

    public string Name { get; private set; } // Public reading and private writing
}
```

Tipos de referencia – clases - **propiedades**

```
class Program
{
    static void Main()
    {
        Fruit apple = new Fruit();
        apple._kilos = 7; // Error
        apple.Kilos = 7; // We give the value 7 to _kilos
        apple.KilosAlt = 8; // We give the value 8 to _kilos
        apple.KilosReadOnly = 9; // Error
        apple.Pounds = 15; // We give the value 6.805808 to _kilos
        apple.Name = "Apple"; // Error
    }
}
```

Tipos de referencia – clases - **propiedades**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
class Fruit:
    def __init__(self):
        self.__kilos = 0.0

    @property
    def kilos(self):
        return self.__kilos

    @kilos.setter
    def kilos(self, value):
        self.__kilos = value

    @property
    def pounds(self):
        return self.__kilos * 2.204

    @pounds.setter
    def pounds(self, value):
        self.__kilos = value / 2.204
```

Tipos de referencia – clases - **static**

- El modificador **static** sirve para declarar miembros estáticos
- Se utiliza en la programación para indicar que algo pertenece a una clase en lugar de a una instancia específica de esa clase
- Los miembros estáticos pertenecen al propio tipo en el que se declaran, en lugar de pertenecer a una instancia concreta
- Podemos declarar como **static** clases, métodos, variables, propiedades…
- Más información

Tipos de referencia – clases - **static**

```
using System;

public class Logger
{
    // Static method to log messages to the console
    public static void Log(string message)
    {
        Console.WriteLine($"[LOG] {DateTime.Now}: {message}");
    }
}

class Program
{
    static void Main()
    {
        Logger.Log("Processing data...");
    }
}
```

Tipos de referencia – clases - **static**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
class Logger:
    @staticmethod
    def log(message):
        print(f"[LOG] {message}")

def main():
    # Use the static Logger class to log messages
    Logger.log("Application started.")

if __name__ == "__main__":
    main()
```

Tipos – **casting** (Conversión de tipos)

- C# es un lenguaje fuertemente tipado, esto quiere decir, que una vez declaro una variable con un tipo, no le puedo cambiar el tipo ni puedo asignarle directamente un valor de otro tipo (a no ser que éste sea convertible de forma implícita al tipo de mi variable)

```
int a = 15;
a = "Hola"; // Compilation error
```

- C# implementa conversiones implícitas para tipos numéricos siempre y cuando no vayamos a perder información al pasar de un tipo a otro

```
float b = a;
```

Tipos – **casting** (Conversión de tipos)

- En alguna ocasión, puedo necesitar convertir de forma explícita el tipo de una variable, a esta operación se le conoce como **casting**

```
int c = 15;
c = (int)15.5; // OK, although we lose information
```

- Pueden implementarse conversores personalizados, para realizar más conversiones que las que nos proporciona C#
- Más información, operadores de conversión

Tipos – **casting** (Conversión de tipos)

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
a = 15
a = "Hola" # OK

b = a

c = 15
c = int(15.5)  # OK, although we lose information
```

Tipos – **casting** (Conversión de tipos)

- Hay que tener cuidado al realizar las conversiones explícitas ya que el compilador no nos protege del todo ante imprevistos y es posible que intentemos realizar una conversión al tipo incorrecto, esto producirá un error en tiempo de ejecución

```
class Unemployed  : Person {
    public int monthsUnemployed;

    public Unemployed(string name, int age, int monthsUnemployed) : base(name, age) {
        this.monthsUnemployed = monthsUnemployed;
    }

    public override string Greet() {
        return $"{base.Greet()}, and I have been {monthsUnemployed} months on unemployment";
    }
}
```

```
Person p = new Unemployed("John", 25, 12);
Console.WriteLine(((Worker)p).Greet()); // System.InvalidCastException: Specified cast is not valid.
```

Tipos – **tipado implícito**

- Las variables locales pueden declararse de forma implícita como **var**, sin especificar de qué tipo son, siempre y cuando se inicialicen en el momento de declararlas
- Variables locales son aquellas que no son miembros de una clase, struct etc. Son aquellas que solo existen en el ámbito en el que se declaran (el "trozo de código" en el que se crean)
- Al estar la inicialización al lado de la declaración, el compilador es capaz de determinar con exactitud el tipo asociado a esa variable

```
var a = 10;
```

- La variable creada de este modo tendrá su tipo definido, igual que si la hubiéramos definido de forma explícita
- **No confundir tipado implícito con tipado débil**
- Más información, otro enlace

Operador **is**

- El operador **is** en C# nos permite fácilmente comprobar si la clase preguntada es del tipo indicado.
- Es una forma segura de realizar un casteo de un tipo a otro antes de ejecutarlo.
- Asignamos el nombre de la variable después del tipo para poder operar con él.

```
for (int i = 0; i < peopleAtWork.Count; i++)
{
    Person person = peopleAtWork[i];
    person.Greet();

    if (person is Worker worker)
    {
        moneyEarned += worker.Work();
    }
}
```

Tipos de referencia - **Interfaces**

- Una interfaz define una funcionalidad que puede ser implementada por una clase o estructura
  - Las structs **no pueden heredar** de clases u otros structs, pero sí que pueden **implementar** interfaces
  - Una interfaz no puede contener constantes, variables, operadores, constructores, destructores ni tipos
  - Podemos considerarlas como clases abstractas que **únicamente definen métodos**
- Se pueden **implementar** varias interfaces
  - Esto es importante porque C# no permite heredar simultáneamente de varias clases
- Para implementar un miembro de una interfaz, su implementación debe ser **pública**, no estática, y con la misma cabecera que la definida en la interfaz

Tipos de referencia - **Interfaces**

- Por convención, los nombres de interfaces empiezan por “**I**”
- Si una clase implementa una interfaz, debe proporcionar definiciones para todos los miembros especificados por esa interfaz
- Si una clase base implementa una interfaz, sus clases derivadas heredarán esa implementación
- Podemos definir un objeto especificando como tipo una interfaz sin problema, aunque, de igual modo que con las clases abstractas, no podremos instanciar directamente un objeto de tipo interfaz
- Python no tiene interfaces como en C#, pero admite herencia múltiple
- Más información

Tipos de referencia - **Interfaces**

```
interface IGreeter {
    string Greet();
}

class Person: IGreeter {
    public string name;
    public int age;

    public virtual string Greet() {
        return "Hello, I'm " + name;
    }
}

class Worker : Person {
    public string job;
    public int salaryGrossSalary;

    public override string Greet() {
        return base.greet() + ", and my job is " + job;
    }
}
```

Colecciones

- Las colecciones nos permiten agrupar elementos para poder tratar con conjuntos de datos
- Podemos agrupar los elementos de la siguiente manera:
  - Mediante **matrices**, para trabajar con un número fijo de elementos fuertemente tipados.
  - Mediante colecciones que nos permiten trabajar con conjuntos de tamaño variable (mayor flexibilidad)
    - Es preferible utilizar colecciones que agrupen elementos del mismo tipo para mantener la coherencia y la seguridad del código, aunque es posible crear colecciones con diferentes tipos
- Más información

Colecciones - **Arrays** (Vectores o matrices unidimensionales)

- Permiten almacenar varias variables del mismo tipo en una estructura de datos (y en un bloque de memoria continuo)
- Debemos de especificar el tamaño del array (el número de elementos que puede almacenar) en el momento de su creación
  - Se inicializan todos sus elementos con sus valores por defecto

```
int[] numbers = new int[5]; // Array of 5 integers
```

- Podemos especificar con que elementos queremos inicializar el array

```
int[] numbers = {7, 5, 9, 3, 1};
```

- Podemos acceder a un elemento concreto mediante su índice (empezando por el 0)

```
var firstNumber = numbers[0]; // First element of the array
```

- Más información

Colecciones - **Arrays**

- En Python, no es necesario especificar de antemano el tamaño de la lista, y puedes inicializarla directamente con valores

```python
numbers = [7, 5, 9, 3, 1]  # List of integers

first_number = numbers[0] # Accessing the first element of the list
print(first_number)
```

Colecciones - **Matrices Multidimensionales**

- Iguales a los arrays, pero con filas y columnas
- Todas las filas tienen el mismo tamaño y todas las columnas tienen el mismo tamaño (matrices cuadradas)
- La inicialización es muy similar a la de los vectores

```
var words = new string[3, 5]; // Matrix of 3x5 strings
string[,] wordPairs = { {"hola", "hello"}, {"adiós", "bye"} }; // 2x2
```

- Lo mismo con el acceso a elementos, aunque esta vez hay que usar un índice para cada dimensión de la matriz

```
var hello = wordPairs[0,1]; // "hello"
var bye = wordPairs[1,1]; // "bye"
```

- Más información

Colecciones - **Matrices Multidimensionales**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
# Matrix of 3x5 strings
words = [["" for _ in range(5)] for _ in range(3)]

# 2x2 matrix of word pairs
wordPairs = [["hola", "hello"], ["adiós", "bye"]]

# Accessing elements in the matrix
hello = wordPairs[1][0]  # "hello"
bye = wordPairs[1][1]    # "bye"
```

Colecciones - **Matrices Escalonadas**

- Son matrices en las que sus elementos son otras matrices
- Esto permite crear matrices con filas de diferentes tamaños

```
int[][] staggered = new int[3][];
staggered[0] = new int[] { 1, 2, 3 };
staggered[1] = new int[] { 4 };
staggered[2] = new int[] { 5,6 };

int[][] staggeredAlt = { new int []{1,2}, new int[]{}, new int[]{3}};
```

- El acceso a elementos resulta bastante familiar

```
var numberFour = staggered[1][0];
```

- Más información

Colecciones - **Matrices Escalonadas**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
staggered = [None] * 3
staggered[0] = [1, 2, 3]
staggered[1] = [4]
staggered[2] = [5, 6]

staggeredAlt = [[1, 2], [], [3]]

numberFour = staggered[1][0]
```

Colecciones - **Listas**

- Permite almacenar varias variables del mismo tipo en una estructura de datos cuyo tamaño puede variar
- Se encuentra bajo el namespace **System.Collections.Generic**

```
List<Person> people = new List<Person>(); // Create empty list
var alba = new Person("Alba", 22);
people.Add(alba); // Add a person to the end of the list
people.Add(new Person("Ana", 23)); // Create and add another person
var ana = people[1];
Console.WriteLine(people.Count); // 2
people.Remove(alba); // Remove alba from the list
Console.WriteLine(people.Count); // 1
people.RemoveAt(0); // Remove the first element of the list (ana)
Console.WriteLine(people.Count); // 0
```

- Más información

Colecciones - **Listas**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
people = [] # Create empty list
alba = Person("Alba", 22)
people.append(alba) # Add a person to the end of the list
people.append(Person("Ana", 23)) # Create and add another person
ana = people[1]
print(len(people)) # 2
people.remove(alba) # Remove alba from the list
print(len(people)) # 1
people.pop(0) # Remove the first element of the list (ana)
print(len(people)) # 0
```

Colecciones - **Diccionarios**

- Es una colección de pares clave, valor
- Se encuentra bajo el namespace **System.Collections.Generic**
- Proporciona un acceso muy rápido a los elementos (valores) por clave
- Podemos tener valores repetidos, pero no claves repetidas

```
// The keys will be of type string and the values int.
Dictionary<string, int> numbers = new Dictionary<string, int>();
numeros["zero"] = 0;
numbers["seven"] = 7;
Console.WriteLine(numeros.Count);
foreach (var value in numeros.Values) {
    Console.WriteLine(value);
}
```

- Más información

Colecciones - **Diccionarios**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
# Create an empty dictionary
numbers = {}

# Add key-value pairs to the dictionary
numbers["zero"] = 0
numbers["seven"] = 7

# Get the count of key-value pairs in the dictionary
print(len(numbers))

# Iterate through the values in the dictionary
for value in numbers.values():
    print(value)
```

Diferencias colecciones C# - Python

Concepto	C#	Python
Mutable, dinámica, ordenada	List<T>	list
Mutable, tamaño fijo, ordenada	T[]	-
Inmutable, tamaño fijo, ordenada	Tuple<T1,T2,…>	tuple
Valores únicos, sin orden	HashSet<T>	Set
Acceso por índice	List<T>,T[]	List, tuple

Colecciones – **Bucle foreach**

- Es una variación del bucle **for** que nos permite repetir un conjunto de sentencias por cada elemento que esté contenido en una colección
- No es recomendable modificar la colección que estamos recorriendo
- Dentro de cada iteración del bucle, tendremos acceso al elemento de la colección con el que estemos trabajando

```
var people = new List<Person>();
people.Add(new Person("Juan", 25));
people.Add(new Person("Jaime", 23));
people.Add(new Person("Ana", 27));

foreach (var person in people) {
    Console.WriteLine($"Name: {person.name}, age: {person.age}");
}
```

- Más información

Colecciones – **Bucle foreach**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
people = [
    Person("Juan", 25),
    Person("Jaime", 23),
    Person("Ana", 27)
]

for person in people:
    print(f"Name: {person.name}, age: {person.age}")
```

Clases Genéricas

- Encapsulan funcionalidad que no es específica para un tipo concreto
- Las listas y diccionarios se implementan mediante clases genéricas
- Pueden aplicarse restricciones, para que nuestras clases genéricas no puedan aplicarse a cualquier tipo

```
public class Box<T>
{
    private T content;
    public void SetContent(T newContent)
    {
        content = newContent;
    }
}
```

```
Box<int> box = new Box<int>();
box.SetContent(10);
```

- Más información

Clases Genéricas

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
from typing import TypeVar, Generic

T = TypeVar('T')

class Box(Generic[T]):
    def __init__(self):
        self._content = None
    def set_content(self, new_content: T):
        self._content = new_content
```

```python
box = Box[int]()
box.set_content(10)
```

Enlaces de interés

- Documentación C#: https://learn.microsoft.com/es-es/dotnet/csharp/
- Documentación Unity: https://docs.unity3d.com/Manual/index.html
- Documentación Scripting Unity: https://docs.unity3d.com/ScriptReference/index.html