COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

Introducción a C#

Visual Studio

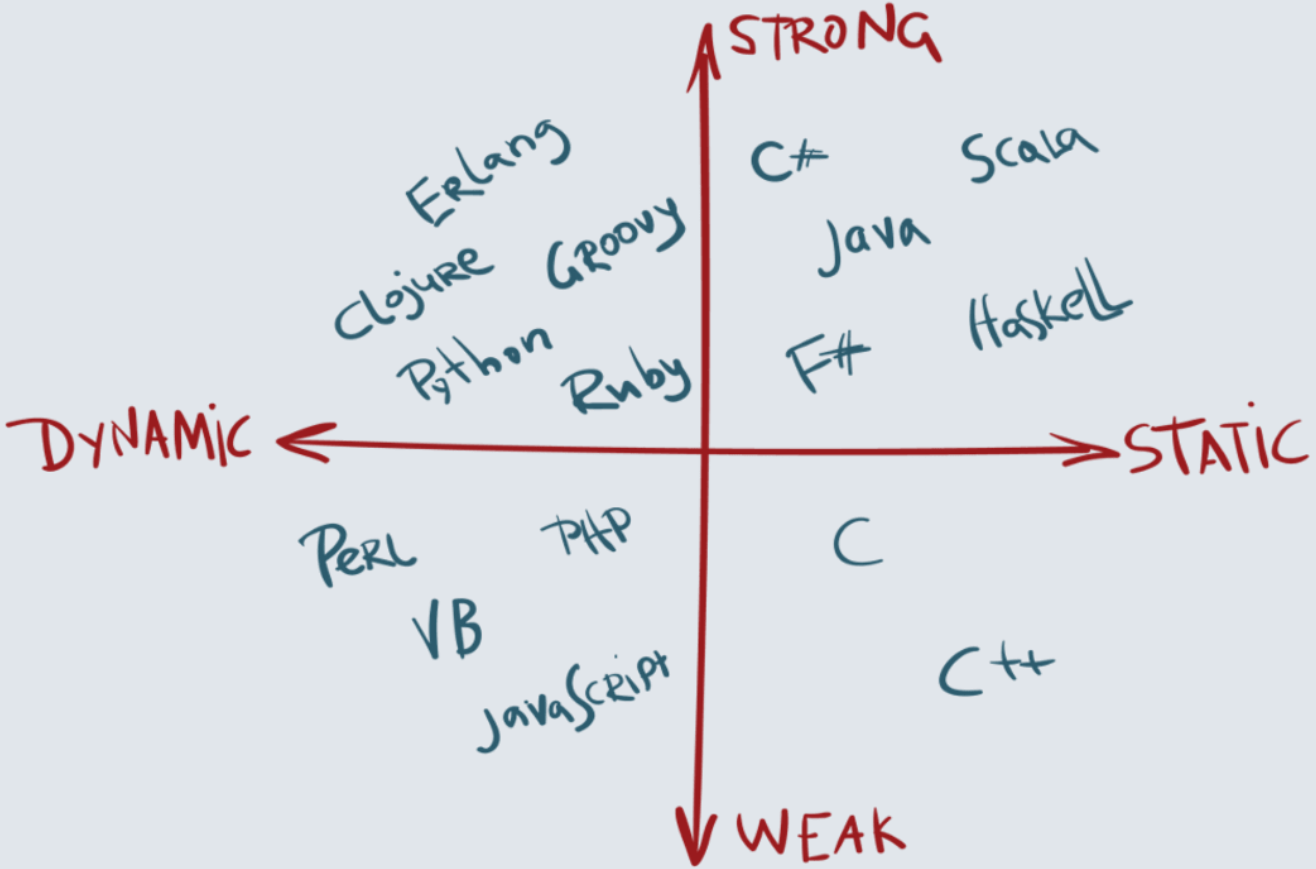
- Es un **IDE (Integrated Development Environment)** desarrollado por Microsoft
- Es el **IDE por defecto de Unity**
- Tiene soporte nativo para desarrollar en C# (además de muchos otros lenguajes)
- Se utiliza para desarrollar aplicaciones (pc y móviles), páginas web, servicios web…

C#

- C# es un lenguaje de programación desarrollado por Microsoft en el año 2000 como parte de su plataforma .NET
- Lenguaje de alto nivel
- Orientado a objetos (**POO**)
- **Tipado fuerte y estático**
- Gestión de memoria mediante un recolector de basura (**Garbage Collector**)
- **Todos los tipos heredan de Object**
- Desarrollo de aplicaciones de escritorio, aplicaciones web, aplicaciones móviles (Xamarin), videojuegos (Unity)…

Tipos de Tipado

- **Tipado estático:**
  - Tipo está **ligado a la variable**
  - Comprobación en tiempo de **compilación**
- **Tipado dinámico:**
  - Tipo está **ligado al valor**
  - Comprobación en tiempo de **ejecución**
- **Tipado débil:**
  - **Conversiones implícitas**
  - Posible pérdida de datos
- **Tipado fuerte:**
  - **Conversiones explícitas**
  - **Conversiones implícitas** cuando no se pierden datos


STRONG
WEAK
DYNAMIC
STATIC
Erlang
Clojure
Groovy
Python
Ruby
C#
Scala
Java
F#
Haskell
Perl
PHP
VB
JavaScript
C
C++

C# vs Python

- **Tipado Estático**
- Lenguaje **Compilado**
  - Se traduce a lenguaje de máquina antes de ejecutarse
- **Sintaxis rígida**
  - “ { } ” para definir bloques de código
  - “ ; ” al final de las declaraciones

- **Tipado Dinámico**
- Lenguaje **Interpretado**
  - Se ejecuta línea por línea por un intérprete en tiempo real
- **Sintaxis concisa**
  - **Indentación** para definir bloques de código
  - No necesita “ ; ” al final de las declaraciones

Hello World - C#

- Nuevo proyecto en Visual Studio
- Plantilla: Aplicación de consola

```
class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Herramientas

- Sentencias del estilo de **using System.Collections**
- Las herramientas en C# están organizadas bajo **espacios de nombres** (**namespaces**)
- Más información

- Los **módulos** son equivalentes a los espacios de nombres en C#
- Los **paquetes** son colecciones de módulos relacionados que se organizan en directorios
- Los módulos y paquetes son importados con la palabra clave **import**

Tipos de datos

- Toda aplicación almacena y manipula datos en la memoria de la máquina
- Hay dos clases de tipos de datos:
  - **Tipos de valor**:
    - Tipos simples, enums y structs
  - **Tipos de referencia**:
    - Clases, interfaces y delegados
- Más información, Tutorial Unity

- Hay dos tipos de datos:
  - **Datos Inmutables**:
    - **No pueden modificarse** después de su creación.
    - int, float, str, etc.
  - **Datos Mutables**:
    - **Pueden modificarse** después de su creación.
    - list, dict, class, etc.

Variables

- **Tipado necesario**
- Las **variables pueden** modificar su valor
- Las **constantes no pueden** modificar su valor
- Nombrar a las variables con **camelCase**

```
int intNum = 5;
const int constNum = 3;

intNum = 7; // OK
constNum = 10; // NO OK
```

- En Python no existen las constantes
- Nombrar a las variables con **snake_case**

```
int_num = 5;
CONST_NUM = 3; # By convention

int_num = 7; # OK
CONST_NUM = 10; # Bad practice
```

Operadores

- Algunos operadores típicos
  - **a.b**: acceso al miembro b de a (si a tiene miembros)
  - **new**: creación de instancias (No existe en Python)
  - **a[b]**: accede al elemento de a que tenga índice b (para elementos indexables)
  - **-a**: negación numérica
  - **!a**: negación lógica (not en Python)
  - **a + b**: suma (con números)
  - **a - b**: resta (con números)
  - **a * b**: multiplicación (con números)
  - **a / b**: división (con números)
- Pueden clasificarse según su tipo:
  - **Operadores unarios**: !a
  - **Operadores binarios**: a + b
  - **Operador ternario**: **a ? b : c**

  (Python: value_if_true if condition else value_if_false)

Operadores

- De igual forma que en álgebra, los operadores tienen diferente orden de precedencia.
  - Operadores principales
  - Unarios
  - Multiplicativos
  - Suma
  - Desplazamiento
  - Comprobación de tipos y relacionales
  - Igualdad
  - Operadores lógicos: AND > XOR > OR
  - Operadores condicionales: AND > OR
  - Asignaciones y lambda
- Más información

Tipos de valor (simple, enum, struct)

- Contienen directamente los datos que almacenan
- Valor por defecto al ser creados
- Si asignamos una variable de tipo de valor a otra, estamos copiando su valor

```
int a = 5;
int b = a;
a = 10;
Console.WriteLine($"A: {a}, B: {b}"); // A: 10, B: 5
```

- Más información, Valores por defecto

Tipos de valor simple

Tipo	Descripción	Tamaño (bytes)	Rango
int	Números enteros	4	-2.147.483.648 a 2.147.483.647
long	Números enteros (mayor rango)	8	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
float	Números decimales	4	De $\pm 1{,}5 \times 10^{-45}$ a $\pm 3{,}4 \times 10^{38}$
double	Números decimales (mayor precisión y rango)	8	De $\pm 5{,}0 \times 10^{-324}$ a $\pm 1{,}7 \times 10^{308}$
bool	Valores verdaderos o falsos	1	true / false

Tipos de valor – **Enumeraciones**

- Definen un conjunto de valores constantes con nombres descriptivos

- Valor entero (int)
- Comienzan por defecto en 0 y se incrementan en 1 en los sucesivos

```
enum Day { Mon, Tue, Wed,
    Thu, Fri, Sat, Sun }
```

- Puede modificarse el comportamiento.

```
enum Day { Mon = 1, Tue, Wed,
    Thu, Fri = 8, Sat, Sun }
```

- Necesaria librería externa enum

```
from enum import Enum

class Day(Enum):
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6
    Sun = 7
```

Tipos de valor – **Estructuras**

- Agrupar variables que están relacionadas

- Para definir una estructura: struct

```
struct Point {
    public float x;
    public float y;
}
```

```
Point p1 = new Point();
p1.x = 0;
p1.y = 5;
```

- Más información

- No existe una palabra clave struct
- Se utiliza una clase simple

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
point = Point(5, 10)
```

Principios Programación Orientada a Objetos (POO)

- **Abstracción**:
  - Simplificar objetos reales en modelos manejables
  - Representación de datos y procesos abstractos
- **Encapsulación**:
  - Ocultar detalles internos
  - Exponer solo la funcionalidad necesaria
- **Herencia**:
  - Crear nuevas clases basadas en clases existentes
  - Heredar características y comportamientos
- **Polimorfismo**:
  - Tratar objetos de diferentes clases de manera uniforme
  - Métodos con el mismo nombre, pero comportamientos específicos

Ventajas Programación Orientada a Objetos (POO)

- **Modularidad:**
  - Código organizado en clases independientes
- **Reutilización:**
  - Herencia para extender y reutilizar código
- **Mantenibilidad:**
  - Encapsulación y modularidad facilitan corrección de errores
- **Escalabilidad:**
  - Añadir nuevas funcionalidades fácilmente
- **Flexibilidad:**
  - Polimorfismo permite usar objetos de manera intercambiable

Tipos de referencia (clases, interfaces y delegados)

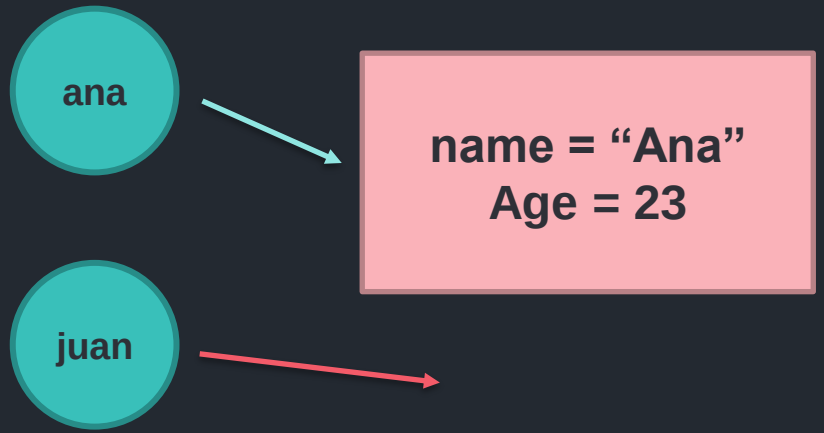
- Almacenan una referencia al lugar de la memoria donde se encuentran los datos de ese elemento
- Dos variables pueden hacer referencia a un mismo objeto

```
Number x = new Number(2);
Number y = x;
Console.WriteLine($"X: {x.num}, Y: {y.num}"); // X: 2, Y: 2
x.num = 7;
Console.WriteLine($"X: {x.num}, Y: {y.num}"); // X: 7, Y: 7
```

Tipos de referencia - **Null**

- null: referencia nula (referencia “a nada”)

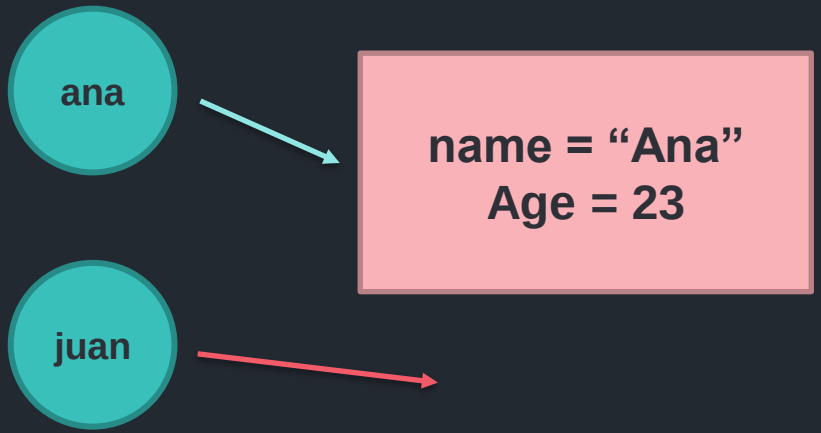
```
Person ana = new Person("Ana", 23);
Person juan = null;
int ageAna = ana.age; // 23
int ageJuan = juan.age; // System.NullReferenceException
```



Tipos de referencia - **None**

- None: referencia nula en Python

```
ana = Person("Ana", 0)
juan = None
edadAna = ana.age  # 0
# AttributeError: 'NoneType' object has no attribute 'age'
edadJuan = juan.age
```



Tipos de referencia – **Clases**

- Representan entidades o conceptos
- Principio de **Abstracción**
- Nombrar con **PascalCase** (UpperCamelCase)
- Definen un conjunto de miembros
  - **Variables**: datos que almacenan el estado
  - **Métodos**: acciones que puede realizar

```
public class Person
{
    public string name;
    public int age;
}
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Tipos de referencia – **Clases**

- Variables cuyo tipo sea una clase se les llama **objetos**
- **Instanciar:**
  - "Crear un objeto"
  - Operador new
  - Se reserva una zona de memoria con espacio suficiente para ese objeto
  - La variable creada guarda la ubicación de ese espacio de memoria

```
Person ana = new Person("Ana", 23);
```

- " . " para acceder a los métodos y variables

```
int ageAna = ana.age;
```

```
ana = Person("Ana", 23)
```

Tipos de referencia – clases - **Visibilidad**

- Los miembros del objeto deben ser visibles para poder acceder a ellos
- Modificadores de acceso:
  - **public**:
    - Acceso sin restricción
  - **protected**:
    - Acceso está limitado al tipo contenedor o tipos derivados
  - **private**:
    - Accesible solo para el tipo contenedor
- Principio de **Encapsulación**
- La accesibilidad se pone antes del tipo de dato que estamos definiendo

```
public int a;
protected int b;
private int c;
```

Tipos de referencia – clases - **Visibilidad**

- No existen los modificadores de acceso
- Convención de nombres:
  - Público: minúsculas
  - Protegido: guion bajo como prefijo
  - Privado: doble guion bajo como prefijo

```
int a;
int _b;
int __c;
```

Tipos de referencia – clases - **Visibilidad**

- Si no especificamos el nivel de accesibilidad, se utilizará el nivel de accesibilidad por defecto para ese tipo

Miembros de	Accesibilidad por defecto	Accesibilidades permitidas
enum	public	
class	private	public protected private
interface	public	
struct	private	public protected private

Tipos de referencia – clases - **Constructores**

- Pueden tener varios constructores (con diferentes parámetros)
- Constructor por defecto sin parámetros, que inicializará todos los atributos a sus valores por defecto
- Más información

```
public class Person
{
    public string name;
    public int age;

    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

Tipos de referencia – clases - **Constructores**

- **No** puedes definir varios constructores con diferentes parámetros
- Puede simular la sobrecarga de constructores proporcionando valores predeterminados para algunos de los argumentos del constructor

```
class Person:
    def __init__(self, name, age = 0):
        self.name = name
        self.age = age
```

Tipos de referencia – clases - **this**

- La palabra reservada this (self en Python) es una referencia al propio objeto desde donde se utiliza
- Utilizando this podemos distinguir entre:
  - Variable local (la que tenemos como parámetro de la función)
  - Variable de la instancia (la que tenemos definida como atributo)
- Más información

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
```

```
class Person:
    def __init__(self, name, age = 0):
        self.name = name
        self.age = age
```

Tipos de referencia – clases - **Métodos**

- Contiene una serie de instrucciones

```csharp
int Sum(int a, int b) {
    return a + b;
}
```

```python
def sumar(a: int, b: int) -> int:
    return a + b
```

- 2 partes:
  - **Cabecera o firma**:
    - Especifica el tipo de dato que devuelve el método `int`
    - Da un nombre al método en **PascalCase** `Sum`
    - Señala el orden, tipo y nombre de sus parámetros `(int a, int b)`
  - **Cuerpo:**
    - Define las instrucciones que se ejecutarán al llamar al método

      `return a + b;`

Tipos de referencia – clases - **Métodos**

- Cuando le pasamos un **tipo de valor** a un método como argumento, pasamos una copia de ese tipo
- Todas las modificaciones que hagamos a esa variable dentro del método no se propagarán fuera de él

```
void ModifyNumber(int num) {
    num++;
    Console.WriteLine($"Value of num within the method, once modified: {num}"); // 2
}
```

```
void Main() {
    var num = 1;
    Console.WriteLine($"Value of num before calling the method: {num}"); // 1
    ModifyNumber(num);
    Console.WriteLine($"Value of num outside the method, once modified: {num}"); // 1
}
```

Tipos de referencia – clases - **Métodos**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

```python
def modify_number(num):
    num += 1
    print(f"Value of num within the function, once modified: {num}") #2
```

```python
def main():
    num = 1
    print(f"Value of num before calling the function: {num}") #1
    modify_number(num)
    print(f"Value of num outside the function, once modified: {num}") #1
```

Tipos de referencia – clases - **Métodos**

- Si lo que pasamos al método es una variable de **tipo de referencia**, le estamos pasando la referencia al objeto, por lo que los cambios que hagamos dentro del método se efectuarán sobre el objeto original (por tanto, se propagarán fuera del método)

```
void IncreaseAge(Person p) {
    p.age++; //13
}
```

```
void Main() {
    Person miguel = new Person("Miguel", 12);
    Console.WriteLine($"Miguel's age before: {miguel.age}"); // 12
    IncreaseAge(miguel);
    Console.WriteLine($"Miguel's age after: {miguel.age}"); // 13
}
```

Tipos de referencia – clases - **Métodos**

```python
def increase_age(p):
    p.age += 1
```

```python
def main():
    miguel = Person("Miguel", 12)
    print(f"Miguel's age before: {miguel.age}")  # 12
    increase_age(miguel)
    print(f"Miguel's age after: {miguel.age}")  # 13
```

Tipos de referencia – clases – **Getters y Setters**

- Controlan el acceso a los campos de una clase
- Acceso a los datos de una clase de manera controlada y segura
- **Getter**:
  - Método que se utiliza para obtener el valor de un campo privado
  - Cabecera: "**public Datatype GetFieldName()**"
- **Setter**:
  - Método que se utiliza para establecer (o modificar) el valor de un campo privado
  - Cabecera: "**public void SetFieldName(DataType value)**"

Tipos de referencia – clases – **Getters y Setters**

```
class Person
{
    private string name;

    public string GetName()
    {
        return name;
    }

    public void SetName(string newName)
    {
        if (!string.IsNullOrEmpty(newName))
        {
            name = newName;
        }
    }
}
```

Tipos de referencia – clases – **Getters y Setters**

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

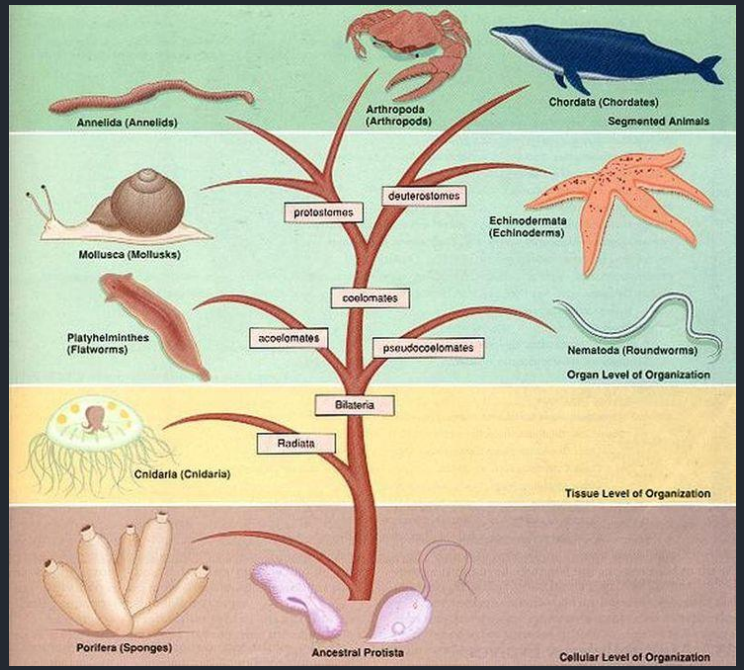
```python
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, new_name):
        if new_name:
            self._name = new_name
```

Tipos de referencia – clases - **Herencia**

- Heredar permite crear clases nuevas que extiendan o modifiquen (especialicen) el comportamiento de la clase que se está extendiendo (conocida como **clase padre** o **base**)
- Esto evita copy-pastes y otras conductas inapropiadas que empeoran la calidad de nuestro código
- Principio de **Herencia**
- Más información



Tipos de referencia – clases - **Herencia**

- La forma de crear una **clase hija** o **derivada** de la clase Person es:

```
class Worker : Person {
    // Attributes added by the derived class
    public string job;
    public int grossSalary;
}
```

- Una clase derivada:
  - Tiene todos los atributos y métodos de su clase padre (excepto constructores)
  - Puede tener nuevos miembros que no tenga su clase padre
  - Puede modificar el comportamiento de los métodos de su clase padre (con restricciones)

Tipos de referencia – clases - **Herencia**

- Aunque las clases derivadas no hereden directamente los constructores de sus clases padre, podemos acceder a ellos de la siguiente manera:

```
class Worker : Person {
    public string job;
    public int grossSalary;

    public Worker(string name, string job, int gSalary) : base(name) {
        this.job = job;
        this.grossSalary = gSalary;
    }
}
```

Tipos de referencia – clases - **Herencia**

```python
class Person:
    def __init__(self, name):
        self.name = name

class Worker(Person):
    def __init__(self, name, job, gross_salary):
        super().__init__(name)
        self.job = job
        self.gross_salary = gross_salary
```

Tipos de referencia – clases – herencia - **Abstract**

- Los métodos abstractos carecen de implementación (únicamente tienen cabecera)
- Debemos implementar “obligatoriamente” los métodos abstractos en las clases hijas
- Una clase que contenga métodos abstractos debe de estar marcada como abstracta
- No pueden crearse instancias de clases abstractas

Tipos de referencia – clases – herencia - **Abstract**

```
abstract class Person {
    public string name;
    public int age;

    public abstract string Greet();
}
```

```
class Worker : Person {
    public string job;
    public int salaryGross;

    public override string Greeting() {
        return "Hi, I'm a worker!";
    }
}
```

Tipos de referencia – clases – herencia - **Abstract**

- Necesaria librería externa “abc”

```python
from abc import ABC, abstractmethod

class Person(ABC):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @abstractmethod
    def greet(self):
        pass

class Worker(Person):
    def __init__(self, name, age, job, salary_gross):
        super().__init__(name, age)
        self.job = job
        self.salary_gross = salary_gross

    def greet(self):
        return "Hi, I'm a worker!"
```

Tipos de referencia – clases - **Polimorfismo**

- Polimorfismo quiere decir “muchas formas”
- Principio de **Polimorfismo** en POO, implica dos cosas:
  - Podemos declarar un objeto con el tipo de una clase base y asignarle un objeto de una clase derivada. Esto permite que, en tiempo de ejecución, el objeto no sea del tipo exacto declarado
  - Las clases base pueden tener métodos virtuales o abstractos que son sobrescritos por las clases derivadas. Cuando se llama a estos métodos en tiempo de ejecución, se invoca la versión del método de la clase derivada a la que pertenece la instancia
- Más información

Tipos de referencia – clases - **Polimorfismo**

```
Person p = new Person("Juan", 25);
Debug.Log(p.Saludar()); // Hello, I'm Juan
p = new Trabajador("Antonio", 30, "doctor", 48000);
Debug.Log(p.Saludar()); // Hello, I'm Antonio, and my job is doctor
```

```
p = Person("Juan", 25)
print(p.Greet())  # Hello, I'm Juan
p = Worker("Antonio", 30, "doctor", 48000)
print(p.Greet())  # Hello, I'm Antonio, and my job is doctor
```

- Para poder reemplazar un método de la clase base, tiene que estar marcado como **virtual** o **abstract** en la clase padre y como **override** en la clase derivada

```
class Person
{
    public virtual string Greet() {
        return "Hello, I'm a person!";
    }
}
```

```
class Worker : Person {
    public override string Greet() {
        return "Hello, I'm a Worker!";
    }
}
```

- Si queremos extender (sin sustituir) un método de la clase base, podemos hacerlo accediendo a base desde la clase hija

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

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

Tipos de referencia – clases – Polimorfismo - **Sobreescritura**

```python
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, I'm {self.name}"

class Worker(Person):
    def __init__(self, name, job, gross_salary):
        super().__init__(name)
        self.job = job
        self.gross_salary = gross_salary

    def greet(self):
        return f"{super().greet()}, and my job is {self.job}"
```

Control de flujo – **if-else**

- Más información

```csharp
if (number > 0)
{
    Console.WriteLine("The number is positive.");
}
else if (number < 0)
{
    Console.WriteLine("The number is negative.");
}
else
{
    Console.WriteLine("The number is zero.");
}
```

```python
if number > 0:
    print("The number is positive.")
elif number < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Control de flujo – **switch**

- Más información

```csharp
switch (day)
{
    case 1:
        Console.WriteLine("Monday");
        break;
    case 2:
        Console.WriteLine("Tuesday");
        break;
    case 3:
        Console.WriteLine("Wednesday");
        break;
    case 4:
        Console.WriteLine("Thursday");
        break;
    case 5:
        Console.WriteLine("Friday");
        break;
    default:
        Console.WriteLine("Weekend");
        break;
}
```

```python
match day:
    case 1:
        print("Monday")
    case 2:
        print("Tuesday")
    case 3:
        print("Wednesday")
    case 4:
        print("Thursday")
    case 5:
        print("Friday")
    case _:
        print("Weekend")
```

Control de flujo – bucles – **while** y **do while**

- while:

```csharp
while (count < 5)
{
    Console.WriteLine($"Count: {count}");
    count++;
}
```

```python
while count < 5:
    print(f"Count: {count}")
    count += 1
```

- do while:

```csharp
do
{
    Console.WriteLine($"Count: {count}");
    count++;
} while (count < 5);
```

```python
while True:
    print(f"Count: {count}")
    count += 1
    if count >= 5:
        break
```

- Más información

Control de flujo – bucles – **for**

- for:

```csharp
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Count: {i}");
}
```

```python
for i in range(5):
    print(f"Count: {i}")
```

- Más información

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