



Programación y Laboratorio II

Clase 09

Polimorfismo y clases abstractas

Prof. Mauricio Cerizza

Repaso de herencia

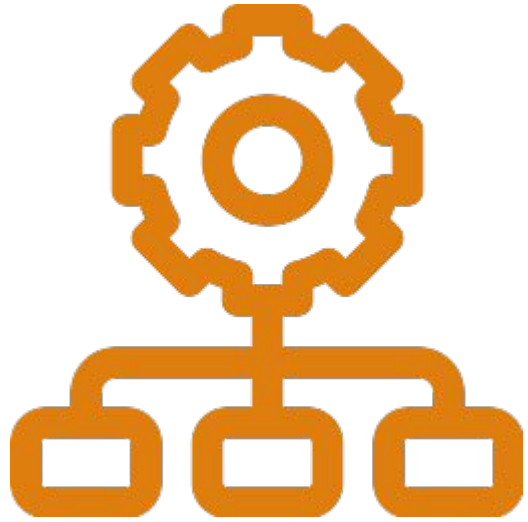
- Herencia
- Principio de sustitución de Liskov

Polimorfismo

- ¿Qué significa polimorfismo?
- Polimorfismo basado en herencia
- Herencia no-polimórfica en C#
- Herencia polimórfica en C#
- Sobrescribiendo métodos de equivalencia

Clases abstractas

- ¿Qué es una clase abstracta?
- Miembros abstractos



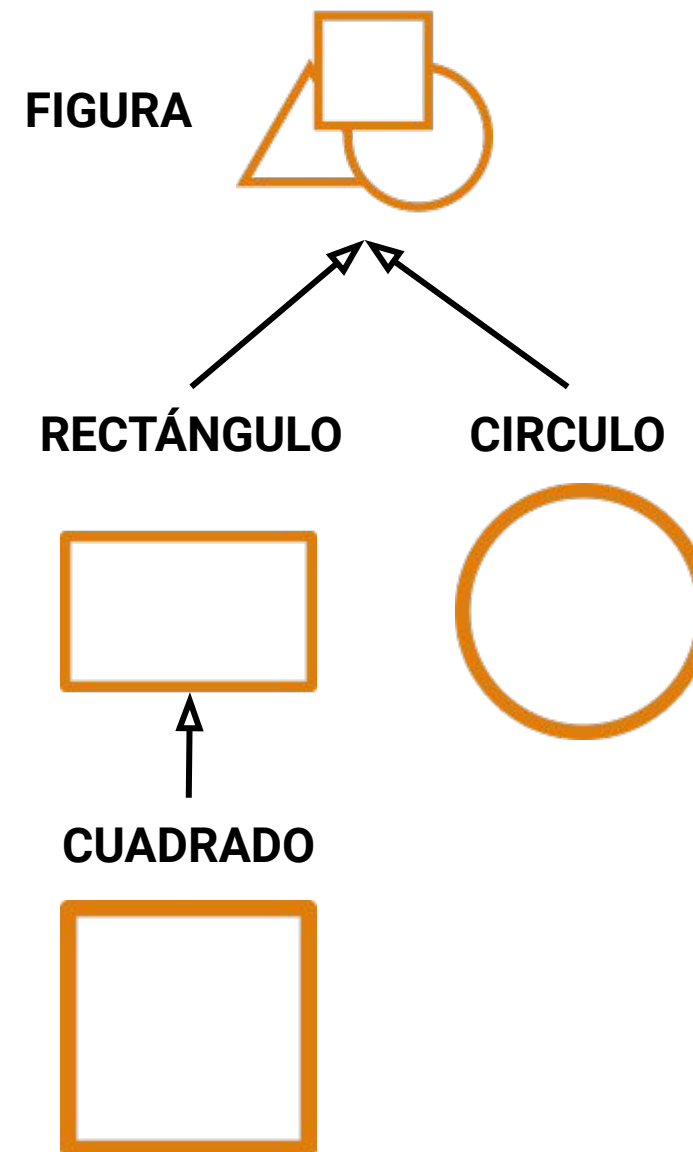
01.

Repaso de herencia

Herencia

- La herencia es la implementación de una **relación de generalización-especialización**.
- La herencia va de lo general a lo particular, factorizando el **comportamiento y características en común en las clases base**.
- Las clases derivadas (hijas) heredan (comparten) todos los miembros de la clase base, salvo los constructores.
- Por **transitividad**, una clase C que hereda de una clase B que a su vez hereda de una clase A, también hereda de la clase A.

$A \leftarrow B \leftarrow C$



Principio de Sustitución de Liskov

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

*Si S es un subtipo de T , entonces los objetos de tipo T en un programa de computadora pueden ser **sustituídos** por objetos de tipo S .*



T objeto = new **S**();

Animal animal = new **Perro**();

Persona persona = new **Alumno**();



02.

Polimorfismo



¿Qué significa **POLIMORFISMO**?

Palabra de origen griego que significa **muchas formas**.

Es la habilidad de los objetos de responder al mismo mensaje de distintas formas.

Los tipos derivados pueden **redefinir** la implementación de una operación en el tipo base.

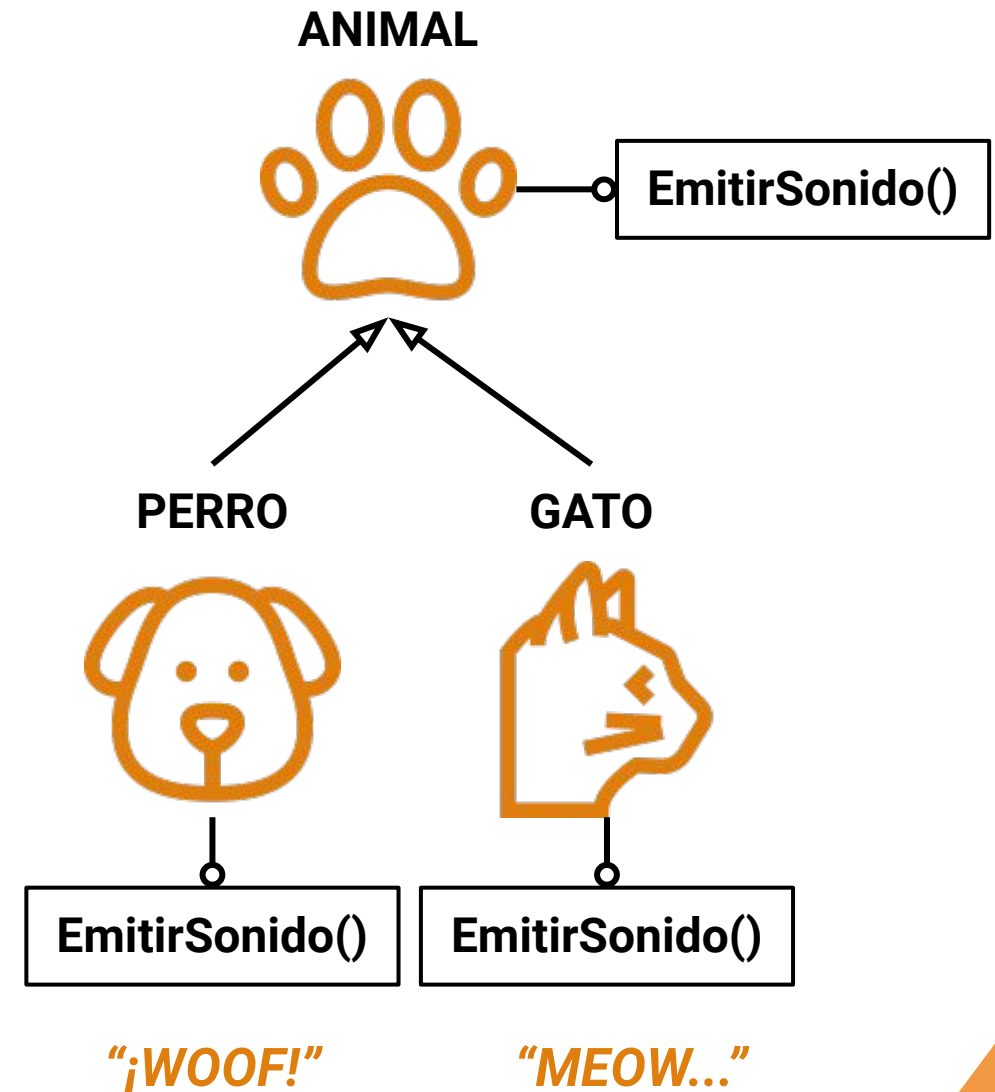
Polimorfismo basado en herencia

En un buen diseño orientado a objetos, un objeto debería poder responder a todas las preguntas importantes sobre sí mismo.

→ Un objeto debe ser **responsable** de sí mismo.

En herencia, todas las clases derivadas heredan la **interfaz** de su clase base.

Sin embargo, como cada clase derivada es una entidad semi-independiente, cada una podría requerir resolver la **respuesta al mismo mensaje de distinta forma**.



Polimorfismo basado en herencia

VENTANA



Abrir()



FIJAS



OSCILANTE



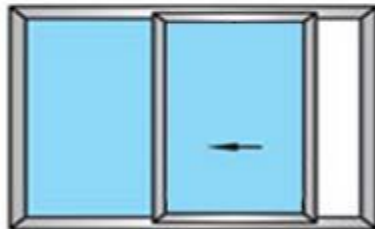
BATIENTE



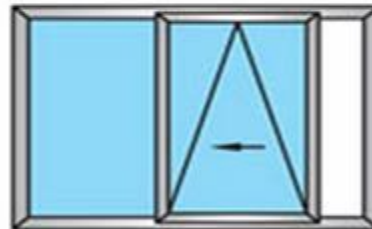
OSCILO-BATIENTE



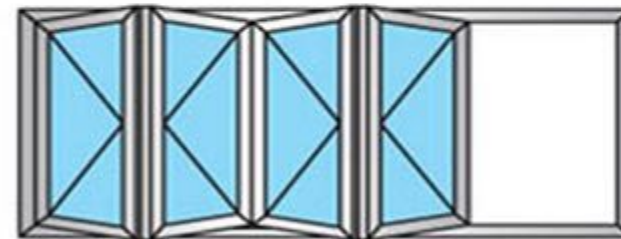
PIVOTANTE



CORREDERA

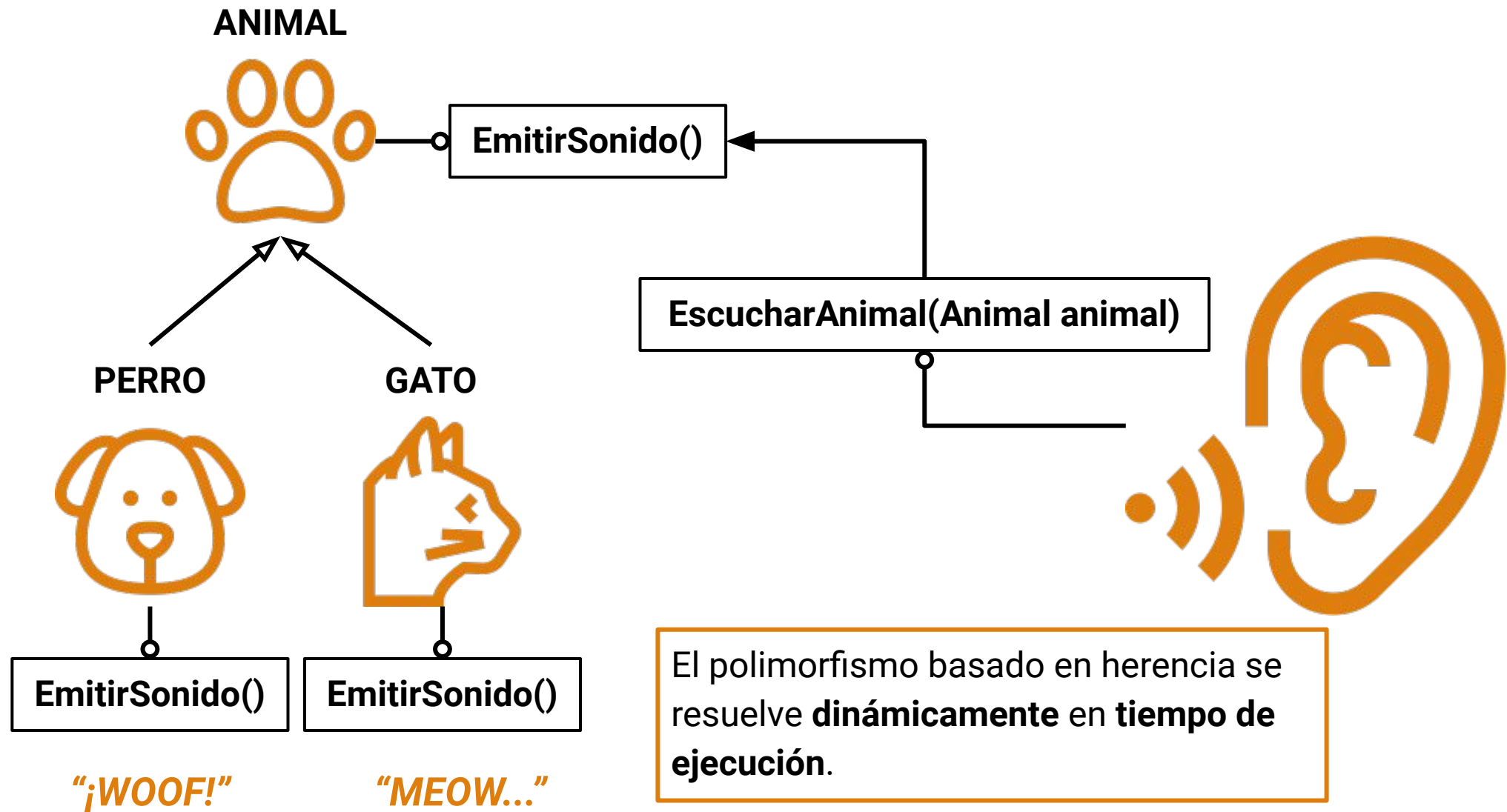


OSCILO-PARALELA



PLEGABLE

Polimorfismo basado en herencia



Herencia **no-polimórfica** en C#

La **herencia no-polimórfica** nos permite redefinir un método de la clase base pero **sin que se aplique polimorfismo**.

El runtime ejecutará la implementación correspondiente al **tipo de la referencia**, sin importar el tipo del objeto en memoria.

NEW → Palabra reservada para **invalidar / redefinir / sobrescribir** un método de la clase base de forma no-polimórfica.

```
1 public class Forma
2 {
3     public string Dibujar ()
4     {
5         return "Dibujando un...";
6     }
7 }
8
9 public class Rectangulo : Forma
10 {
11     public new string Dibujar ()
12     {
13         return base.Dibujar() + "rectángulo.";
14     }
15 }
16
17 public class Circulo : Forma
18 {
19     public new string Dibujar ()
20     {
21         return base.Dibujar() + "círculo.";
22     }
23 }
```

Herencia **polimórfica** en C#

La **herencia polimórfica** nos permite redefinir un método de la clase base **aplicando polimorfismo**.

El runtime ejecutará la implementación correspondiente al **tipo real del objeto en memoria**, independientemente del tipo de la referencia.

VIRTUAL → Palabra reservada para declarar un método que pueda ser **invalidado / redefinido / sobrescrito** por una clase derivada.





OVERRIDE → Palabra reservada para **invalidar / redefinir / sobrescribir** un método virtual de la clase base.



```
1 public class Forma
2 {
3     public virtual string Dibujar ()
4     {
5         return "Dibujando un...";
6     }
7 }
8
9 public class Rectangulo : Forma
10 {
11     public override string Dibujar ()
12     {
13         return base.Dibujar() + "rectángulo.";
14     }
15 }
16
17 public class Circulo : Forma
18 {
19     public override string Dibujar ()
20     {
21         return base.Dibujar() + "círculo.";
22     }
23 }
```

Sobrescribiendo métodos de equivalencia

```
1 public class Persona
2 {
3     private string nombre;
4     private char genero;
5     private string dni;
6
7     public static bool operator == (Persona persona, Persona otraPersona)
8     {
9         if (persona is null || otraPersona is null)
10        {
11            return false;
12        }
13
14        return persona.GetType() == otraPersona.GetType()
15            && persona.dni == otraPersona.dni
16            && persona.genero == otraPersona.genero;
17    }
18
19    public static bool operator != (Persona persona, Persona otraPersona)
20    {
21        return !(persona == otraPersona);
22    }
23 }
```

Sobrescribiendo métodos de equivalencia

Entire Solution  0 Errors  2 Warnings  0 of 6 Messages  Build + IntelliSense

	Code	Description
	CS0660	'Persona' defines operator == or operator != but does not override Object.Equals(object o)
	CS0661	'Persona' defines operator == or operator != but does not override Object.GetHashCode()

Sobrescribiendo métodos de equivalencia

De **object** heredamos los métodos **Equals** y **GetHashCode**.

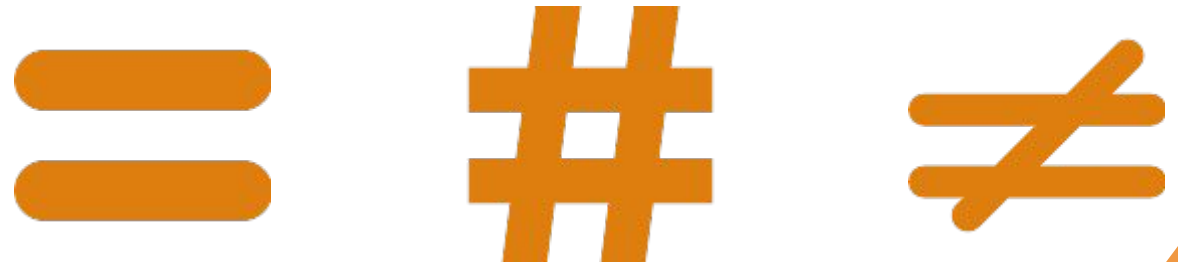
Por defecto: Dos objetos son iguales si tienen la misma dirección de memoria.

Ambos métodos pueden redefinirse en las clases derivadas con una nueva implementación.

Un **hashcode** es un valor numérico que se utiliza para identificar y comparar objetos, por ejemplo en las colecciones **HashTable** y **HashSet**.

Dos objetos iguales deberían retornar el mismo hashcode.

```
1 public override bool Equals(object obj)
2 {
3     Persona otraPersona = obj as Persona;
4
5     return otraPersona != null && this == otraPersona;
6 }
7
8 public override int GetHashCode()
9 {
10     return (dni, genero).GetHashCode();
11 }
```

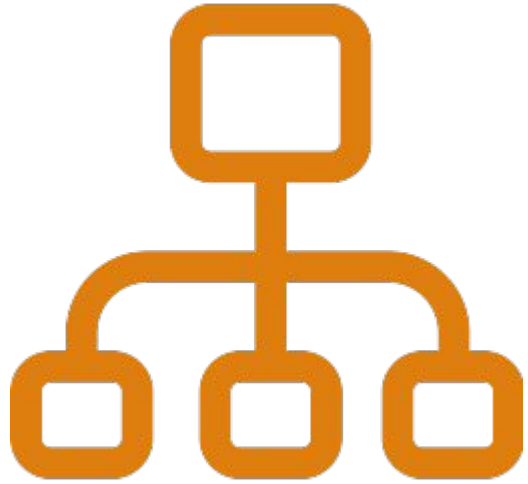


Ejercicios



- Ejercicio I01 - Sobre-sobrescribiendo esas advertencias - **PARTE I**

https://codeutnfra.github.io/programacion_2_laboratorio_2_apuntes/



03.

Clases abstractas



¿Qué es una CLASE ABSTRACTA?

Son clases que **no se pueden instanciar**.

Su propósito es modelar la base de una jerarquía de herencia.

Podemos decir que son clases incompletas cuyas piezas faltantes son aportadas por sus clases derivadas.

Miembros abstractos

Los **miembros abstractos** definen una operación pero no su implementación.

Serán las clases derivadas no-abstractas las que deban aportar una implementación al método.

Las clases abstractas son las únicas que pueden contener miembros abstractos.

Los miembros abstractos son implícitamente virtuales, el polimorfismo aplica de igual forma.

```
1 public abstract class Shape
2 {
3     public abstract void Paint(Graphics g, Rectangle r);
4 }
5
6 public class Ellipse : Shape
7 {
8     public override void Paint(Graphics g, Rectangle r)
9     {
10         g.DrawEllipse(r);
11     }
12 }
13
14 public class Box : Shape
15 {
16     public override void Paint(Graphics g, Rectangle r)
17     {
18         g.DrawRect(r);
19     }
20 }
```

Ejercicios



- Ejercicio I01 - Sobre-sobrescribiendo esas advertencias - **PARTE II**

https://codeutnfra.github.io/programacion_2_laboratorio_2_apuntes/

Ejercicios



- Ejercicio I02 - Calculadora de formas

https://codeutnfra.github.io/programacion_2_laboratorio_2_apuntes/