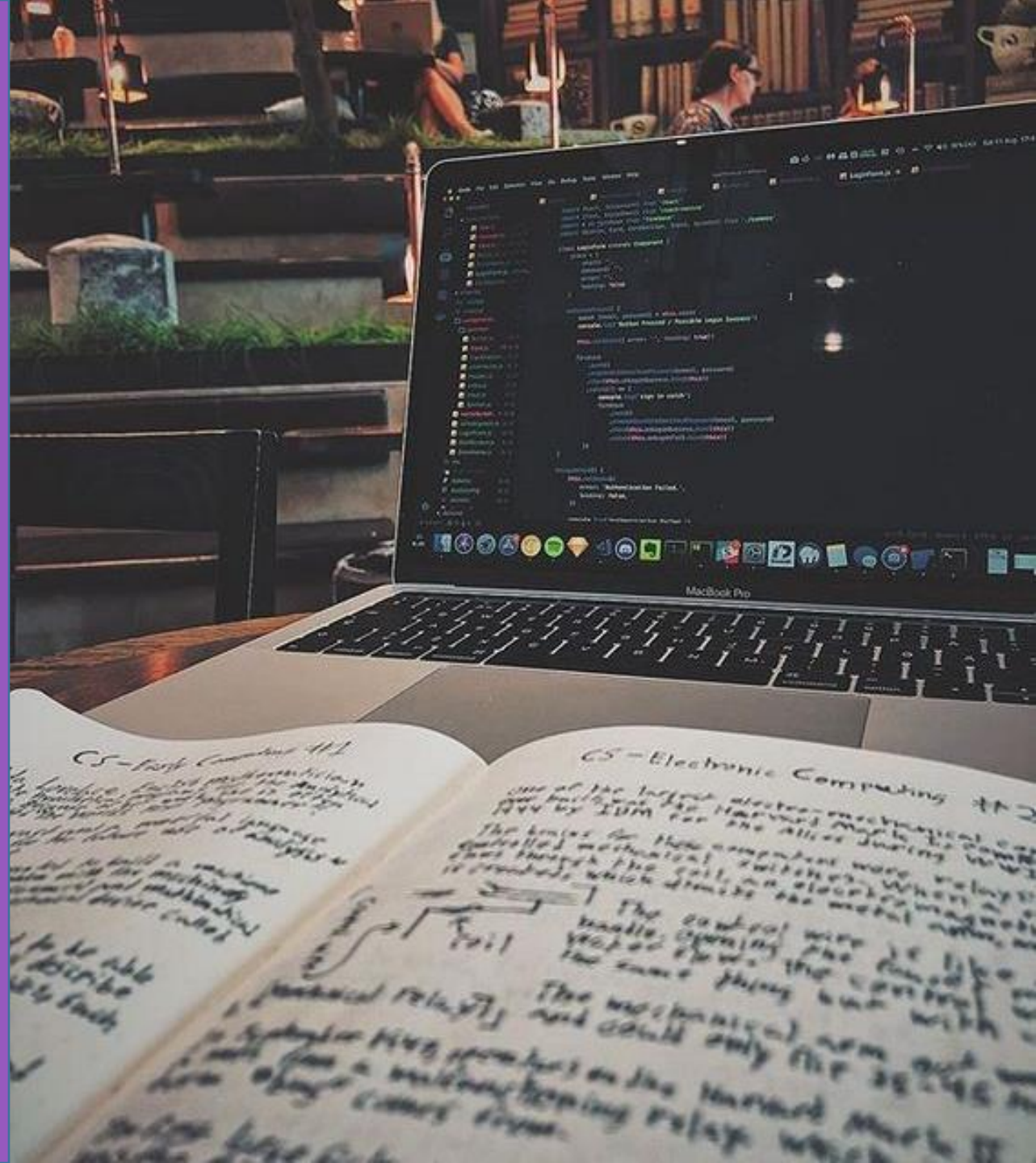
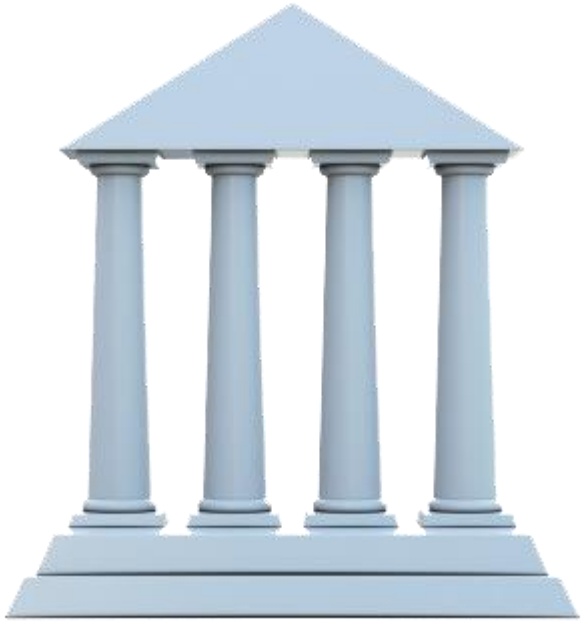
POO

- Herencia
- Polimorfismo
- Sobreescritura de métodos
- Errores de abstracción



Pilares de la POO



Abstracción

Encapsulamiento

Herencia

Polimorfismo

Herencia

¿Que es herencia en la POO?

Es un tipo de relación donde un objeto es creado a partir de otros objetos ya existentes, obteniendo todas las características (métodos y atributos) de la clase base.

Normalmente, nos referimos a este tipo de relación con la expresión: "Es un"

Ej: Auto es un Vehiculo

Ej: Vaca es un Animal

Beneficios

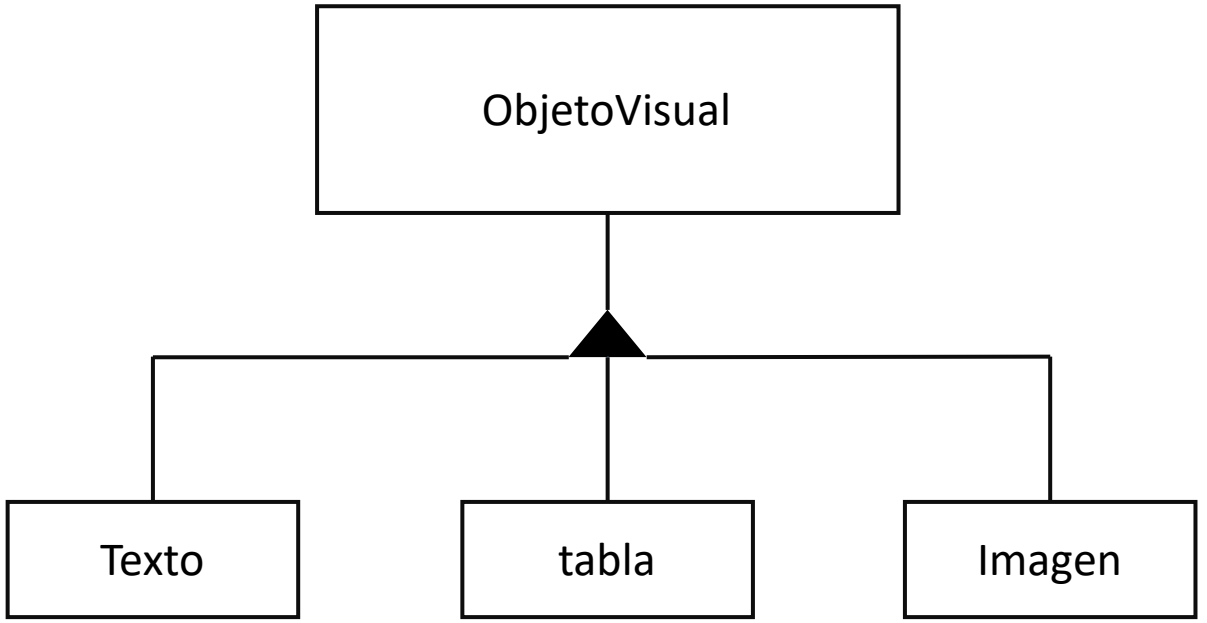
- Reutilización
- Polimorfismo

Herencia

Caso de ejemplo de herencia

Supongamos una herencia para un sistema que permita la edición de contenidos gráficos, como puede ser por ejemplo: CorelDraw, PowerPoint o Adobe Illustrator

La herencia quedaría planteada en los términos de diagrama de la izquierda donde **ObjetoVisual** es el objeto base del que herendan las clases **Texto**, **Tabla** e **Imagen**.



Herencia

Herencia en C#

En C# la herencia se especifica de la *Clase Base* a la *Clase derivada* utilizando el operador ":" (dos puntos).

Class **ClaseDerivada** : **ClaseBase**
{
[…] // procesos de la clase derivada
}

En el siguiente ejemplo se puede ver:
La clase **Texto** hereda de la clase **ObjetoVisual**
La clase **Imagen** hereda de la clase **ObjetoVisual**

También podríamos leer:
Texto **es un** ObjetoVisual
Imagen **es un** ObjetoVisual

```
public class ObjetoVisual
{
     public int X { get; set; }
     public int Y { get; set; }
     public ObjetoVisual()
     {
          […]
     }
}

public class Texto : ObjetoVisual
// Hereda de clase ObjetoVisual
{
     public Texto() :base()
     {
          […]
     }
}

public class Imagen : ObjetoVisual
// Hereda de clase ObjetoVisual
{
     public Imagen() :base()
     {
          […]
     }
}
                    […]
ObjetoVisual Figura = new Texto();
Figura.X = 10;
```

Polimorfismo

¿Que es el polimorfismo en la POO?

El polimorfismo suele considerarse el cuarto pilar de la programación orientada a objetos, después de la encapsulación y la herencia. Polimorfismo es una palabra griega que significa "con muchas formas" y tiene dos aspectos diferentes:

- En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos y colecciones o matrices.

- Las clases base pueden definir e implementar *métodos* virtuales, y las clases derivadas pueden invalidarlos, lo que significa que pueden proporcionar su propia definición e implementación

Polimorfismo

Caso de estudio

En a la clase Figura se crea un método virtual llamado Dibujar (marcada con virtual).

Este método puede ser sobrescrito (override) en cada clase derivada para dibujar la forma determinada que la clase que corresponda.

```
public class ObjetoVisual
{
      public int X { get; set; }
      public int Y { get; set; }
      public ObjetoVisual ()
      {
            […]
      }
      public virtual void Dibujar()
      {
            Console.WriteLine(“método para dibujar”);
      }
}

public class Texto : ObjetoVisual
// hereda de clase figura
{
      Public override void Dibujar()
      {
            Console.WriteLine(“muestro texto”);
      }
}

public class Imagen : ObjetoVisual
// hereda de clase figura
{
      public override void Dibujar()
      {
            Console.WriteLine(“muestro Imagen”);
      }
}
```

Encapsulamiento

Modificador de acceso

Los principales modificadores de acceso son los siguiente:

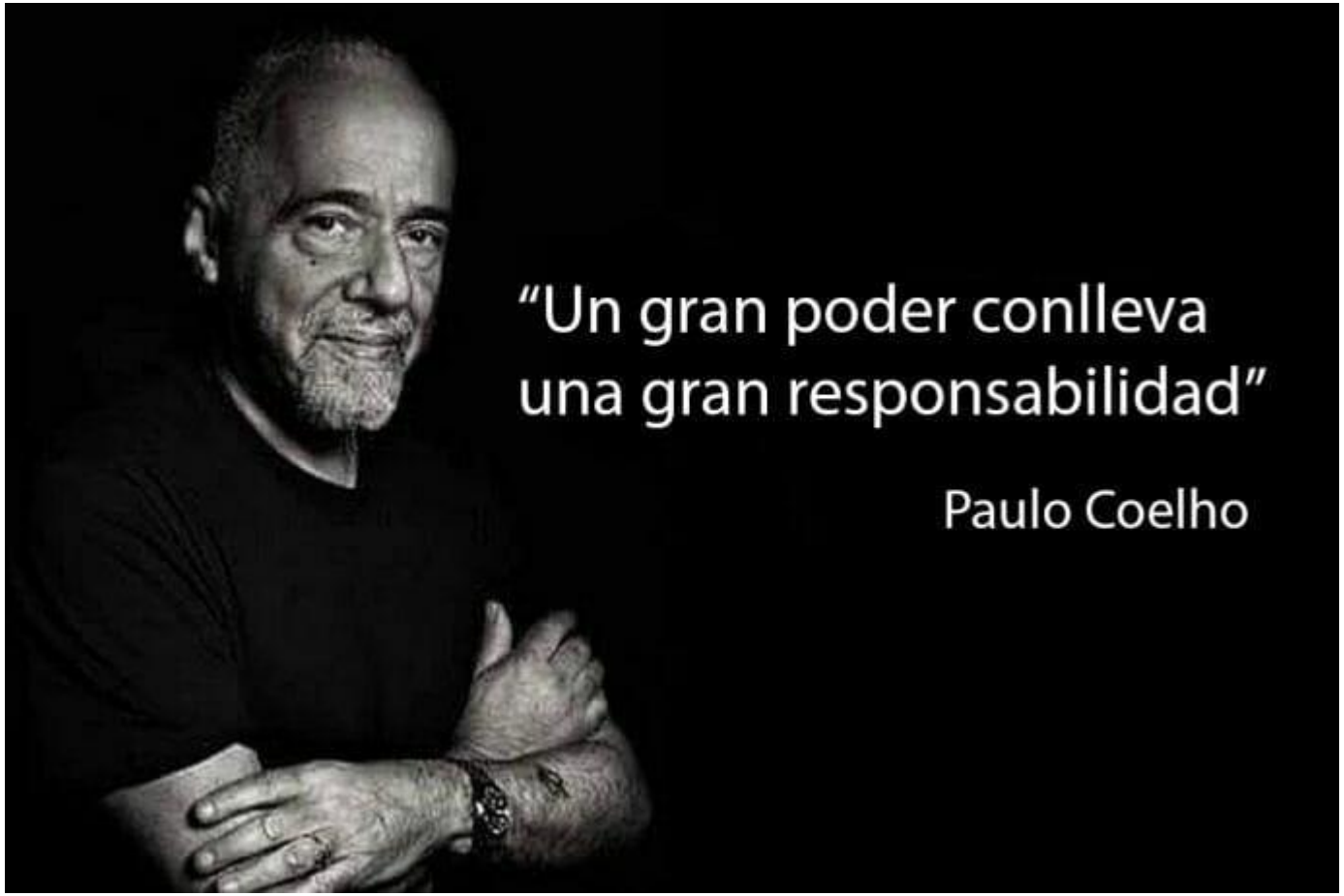
public : Accesible desde cualquier lugar del programa.

private: Accesible desde la propia clase.

protected: Accesible desde la propia clase y desde las clases derivadas.

Nota:
Existen otros modificadores de acceso (internal, protected internal, private protected), pero solo nos centraremos en los anteriores.

Problemas de abstracciones



Problemas de abstracciones

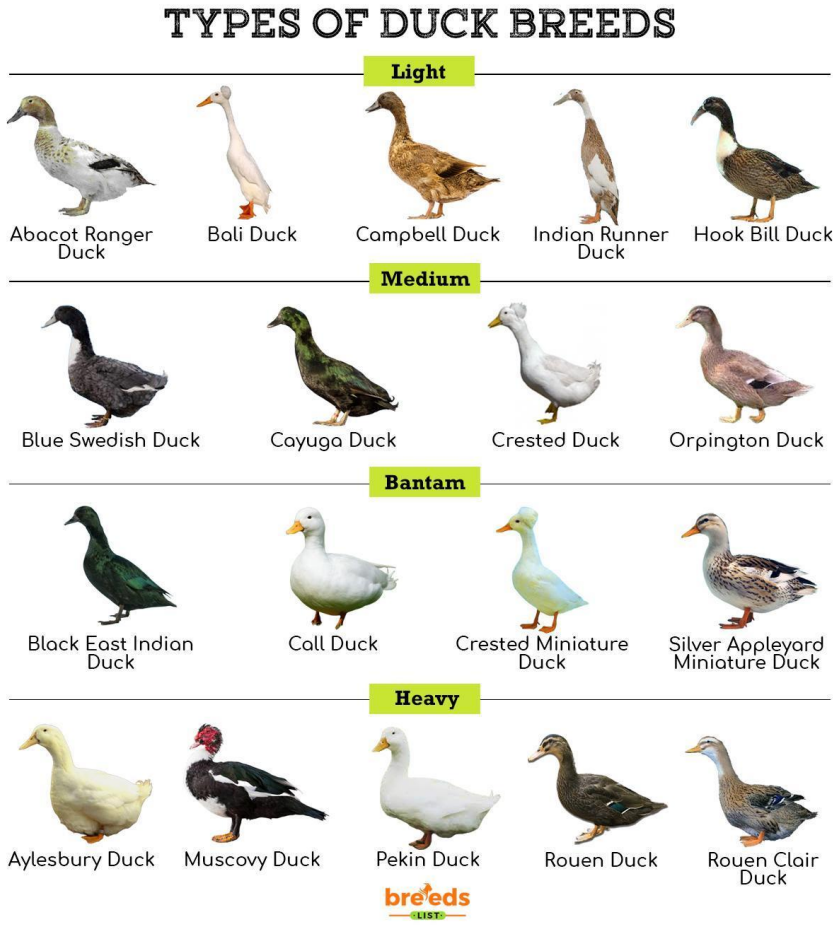
Liskov Substitution Principle

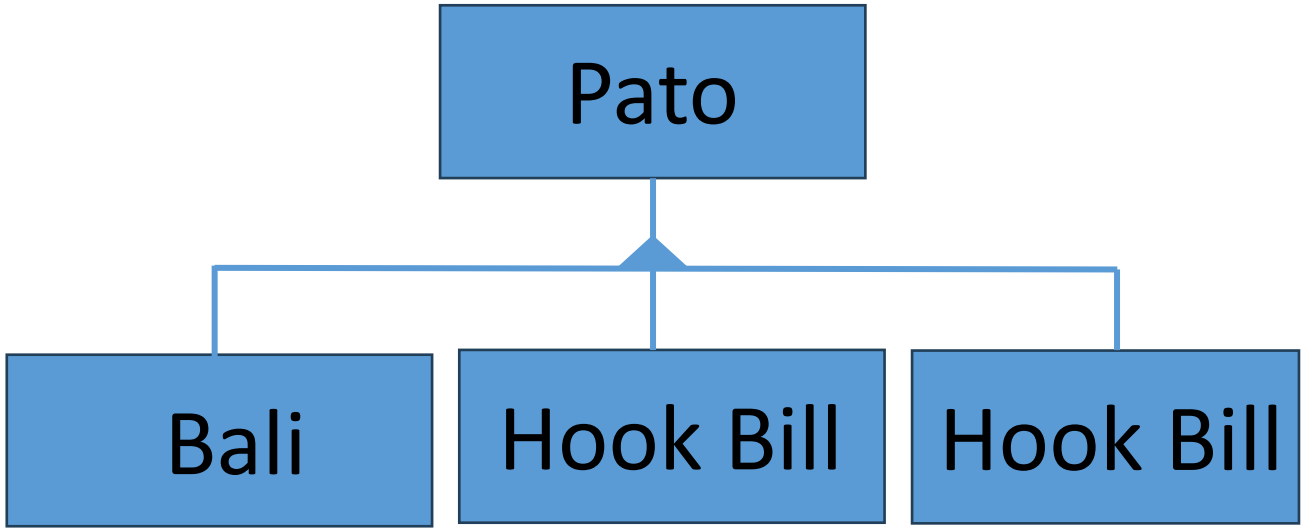
Los objetos de la clase derivada deben ser sustituibles por los objetos de la clase base/padre. Esto significa que los objetos de la clase derivada deben comportarse de una manera coherente con las promesas hechas en el contrato de la clase base.

En lenguaje más formal: si **S** es un subtipo de **T**, entonces los objetos de tipo **T** en un programa de computadora pueden ser sustituidos por objetos de tipo **S** (es decir, los objetos de tipo S pueden sustituir objetos de tipo T), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.).

Problemas de abstracciones

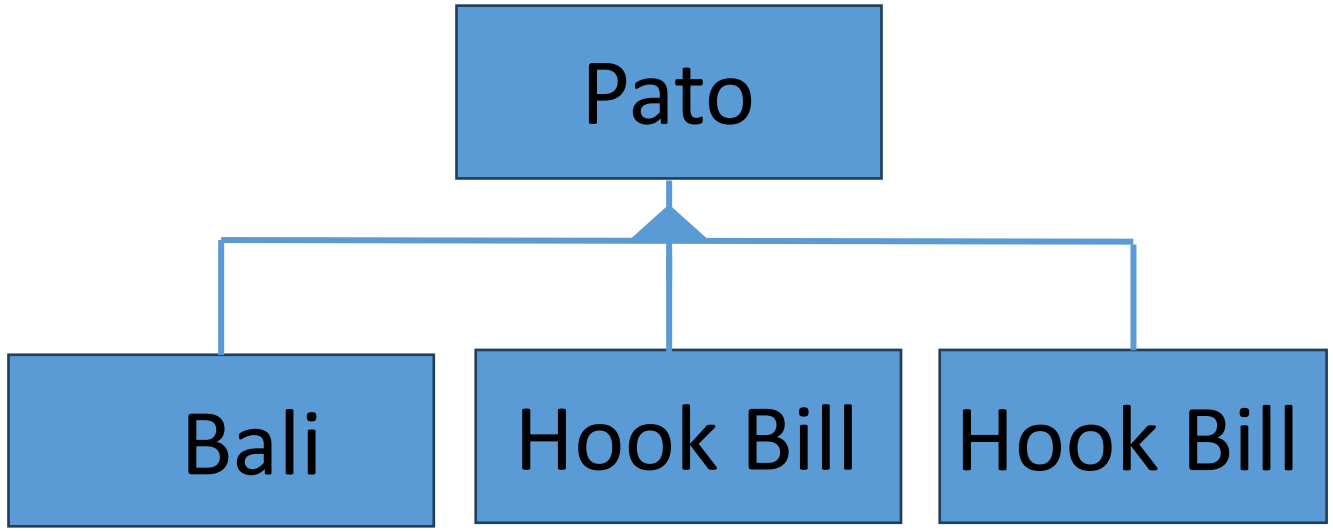
Liskov Substitution Principle





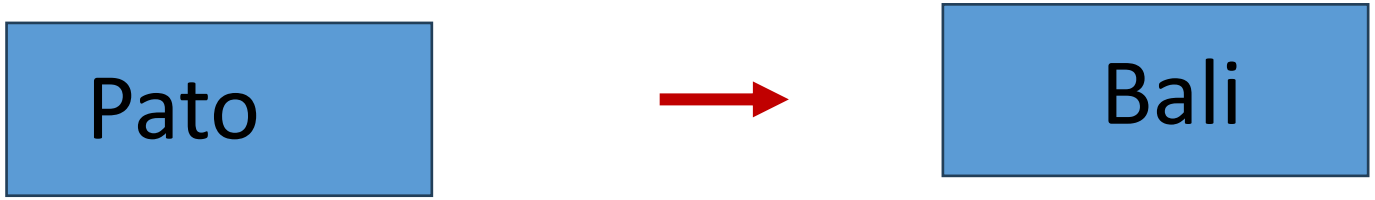
Problemas de abstracciones

Liskov Substitution Principle



Problemas de abstracciones

Liskov Substitution Principle




```
class Pato
{
    public virtual void Quack(){
    }

    public virtual void Volar(){
    }
}
```

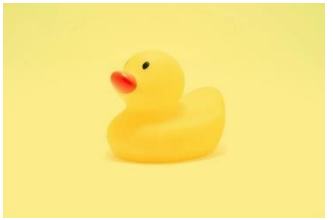
```
class PatoBali : Pato
{
    public override void Quack(){
        Console.WriteLine("Quack quack!");
    }

    public override void Volar(){
        Console.WriteLine(“Volando como un campeón.");
    }
}
```

Problemas de abstracciones

Liskov Substitution Principle

```
class PatoBali : Pato
{
    public override void Quack(){
        Console.WriteLine("Quack quack!");
    }

    public override void Volar(){
        Console.WriteLine(“Volando.");
    }
}
```

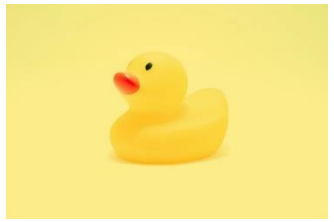
≠



```
class PatoJuguete : Pato
{
    public override void Quack() {
        Console.WriteLine(""Squeak squeak!"!");
    }

    public override void Volar() {
        Console.WriteLine(“NO VUELO.");
    }
    public override void CargarPilas(){
        Console.WriteLine(“Cargando Pilas.");
    }
}
```

Problemas de abstracciones

Liskov Substitution Principle



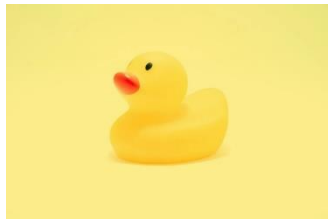
```
class PatoJuguete : Pato
{
    public override void Quack() {
        Console.WriteLine(""Squeak squeak!"!");
    }

    public override void Volar() {
        Console.WriteLine(“NO VUELO.");
    }
    public override void CargarPilas(){
        Console.WriteLine(“Cargando Pilas.");
    }
}
```

¿Cómo CARGO LAS PILAS?

¿Puedo usar Volar?

Problemas de abstracciones

Liskov Substitution Principle



¿Cómo CARGO LAS PILAS?

```
class PatoJuguete : Pato
{
    public override void Quack() {
        Console.WriteLine("Squeak squeak!!");
    }

    public override void Volar() {
        Console.WriteLine(“NO VUELO.");
    }
    public override void CargarPilas(){
        Console.WriteLine(“Cargando Pilas.");
    }
}
```

```
Pato patito = new PatoJuguete();
patito.Quack();
patito.CargarPilas();
```

¡NO LO PUEDO USAR!

¡NO HACE NADA AQUÍ!

Problemas de abstracciones

Liskov Substitution Principle

La clave para evitar el "Problema de la Abstracción Incorrecta" es crear abstracciones que se ajusten de manera natural y coherente a la lógica del problema que estás resolviendo. En algunos casos, puede ser más apropiado crear abstracciones separadas para casos diferentes, en lugar de forzar una única jerarquía de clases. En el ejemplo de los patos, podría ser más claro y sencillo tratar los patos de goma como objetos independientes sin intentar forzarlos a encajar en la misma jerarquía que los patos reales.

"Los objetos de una superclase deben poder ser reemplazados por objetos de sus subclases sin romper la aplicación."

Ejercicio: Simulando construcción de vehículos

Suponga que se requiere modelar objetos vehículos considerando:

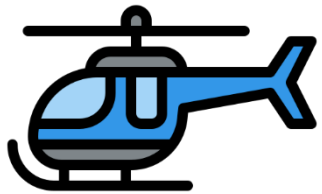
- Un Avión
- Un Auto
- Una Bicicleta

Considere los siguientes métodos:

- Motor(string sonido),
- Conducir()
- ApagarMotor(string sonido)
- Volar()

Ejercicio: Simulando construcción de vehículos

- EncenderMotor(string sonido),
- Conducir()
- ApagarMotor(string sonido)
- Volar()



Tiene motor	**No Tiene Motor**	**Tiene Motor**
Conduce	Se puede Conducir	Se Puede conducir
No Puede volar	No puede volar	Puede volar

Recordando

Para referirnos a las relaciones de las clases diremos que:

- La **Herencia** nos dice *es un*.
- La **Composición** nos dice *es parte de*.
- La **Agregación** nos dice *tiene un.*

"Los objetos de una superclase deben poder ser reemplazados por objetos de sus subclases sin romper la aplicación."