

Programación Orientada a Objetos

Fundamentos de la POO



Fundamentos del enfoque orientado a objetos (EEO)

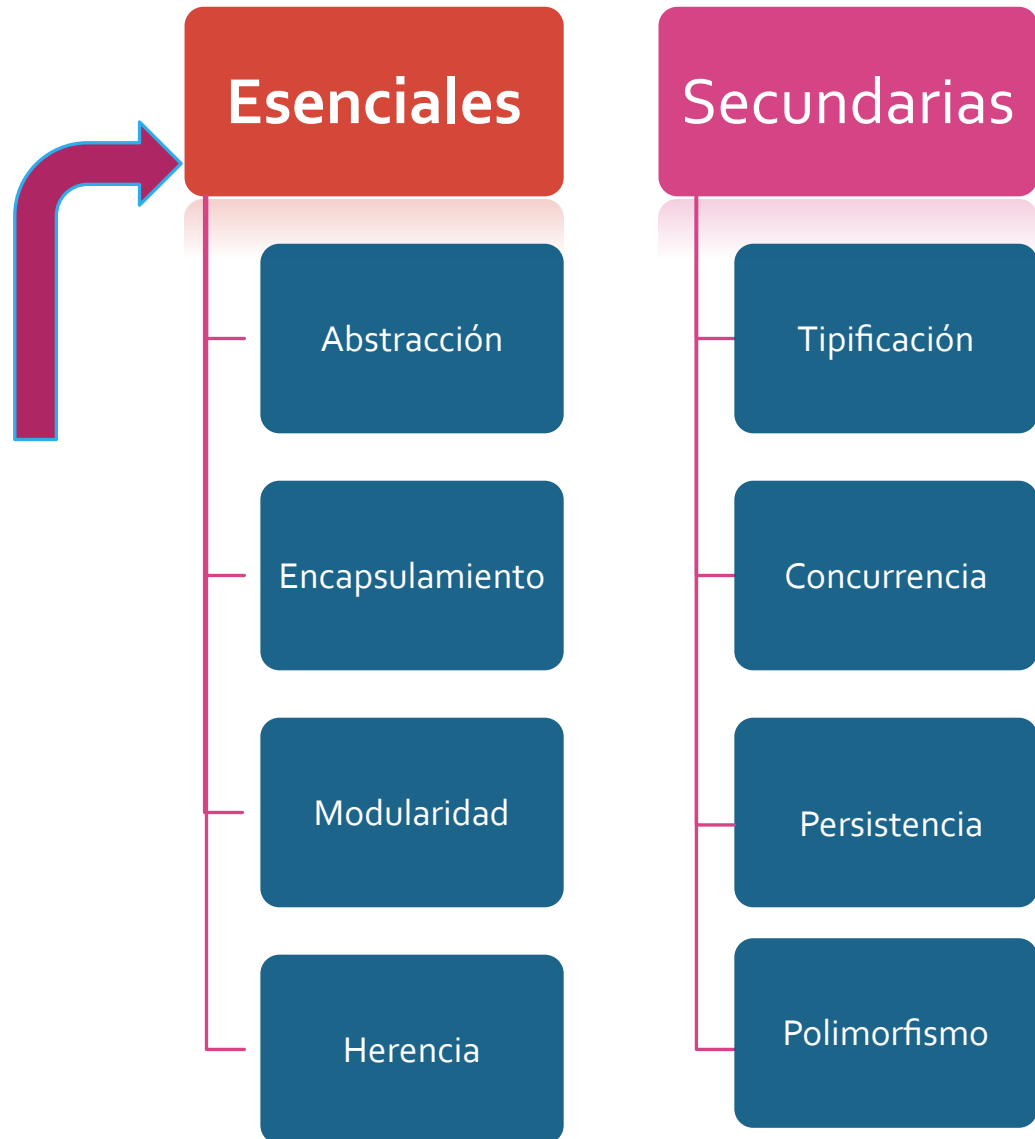
El **Enfoque Orientado a Objeto** se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos principios son:

- La Abstracción
- El Encapsulamiento
- La Modularidad
- Jerarquía (Herencia, composición y agregación).

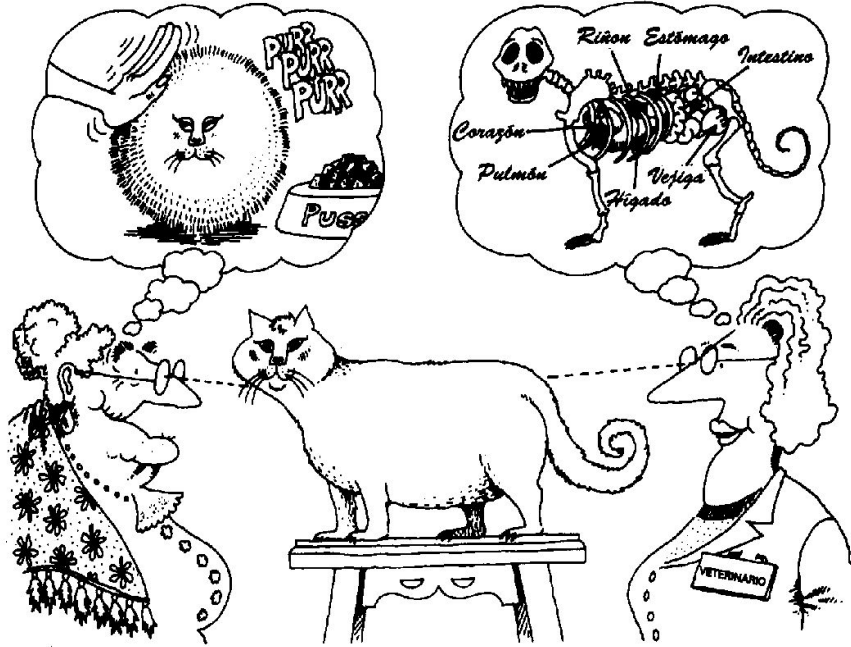
Otros elementos a destacar (aunque no fundamentales) en el EEO son:

- Polimorfismo
- Tipificación
- Concurrencia
- Persistencia.

Modelo de Objetos: Características



¿Qué es la abstracción?



Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

La abstracción se centra en las características esenciales de un objeto en relación a la perspectiva del observador.

Abstracción

Los mecanismos de abstracción son usados en el EOO para extraer y definir del medio a modelar, sus características y su comportamiento. Entre ellos podemos nombrar:

La **GENERALIZACIÓN**. Mecanismo de abstracción mediante el cual un conjunto de clases de objetos son agrupados en una clase de nivel superior (Superclase), donde las semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas.

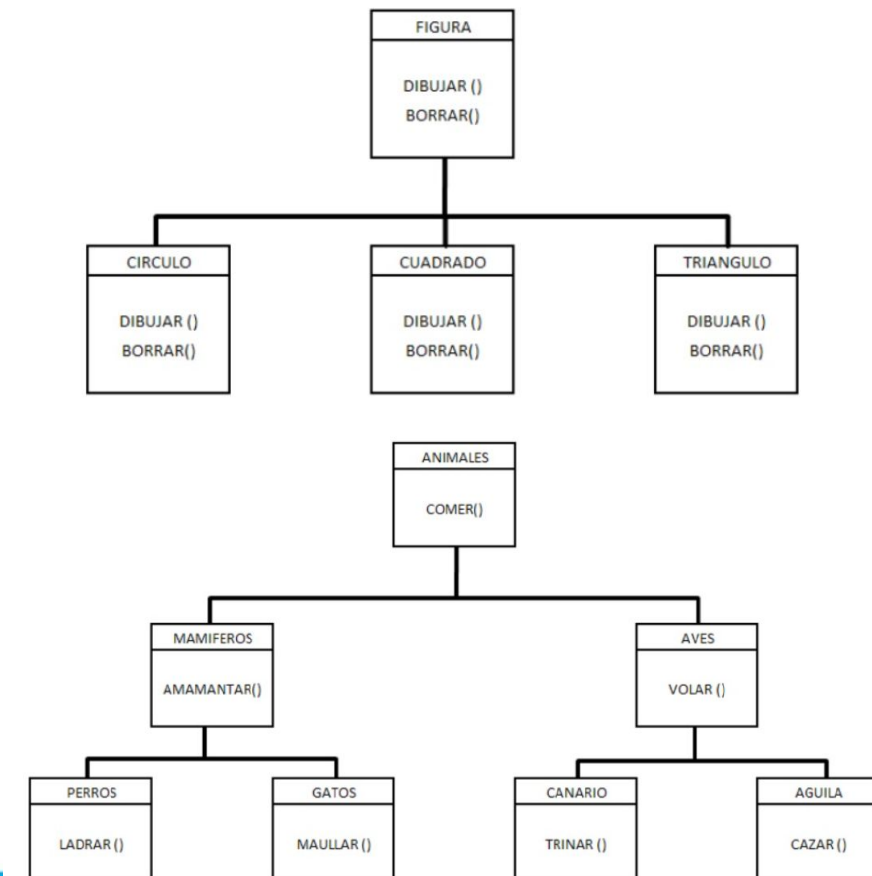
En consecuencia, a través de la generalización:

La **superclase** almacena datos generales de las subclases

Las **subclases** almacenan sólo datos particulares.

La **ESPECIALIZACIÓN** es lo contrario de la generalización:

La **clase Médico** es una especialización de la clase Persona, y a su vez, la **clase Pediatra** es una especialización de la superclase Médico.



Abstracción

Los mecanismos de abstracción son usados en el EOO para extraer y definir del medio a modelar, sus características y su comportamiento. Entre ellos podemos nombrar:

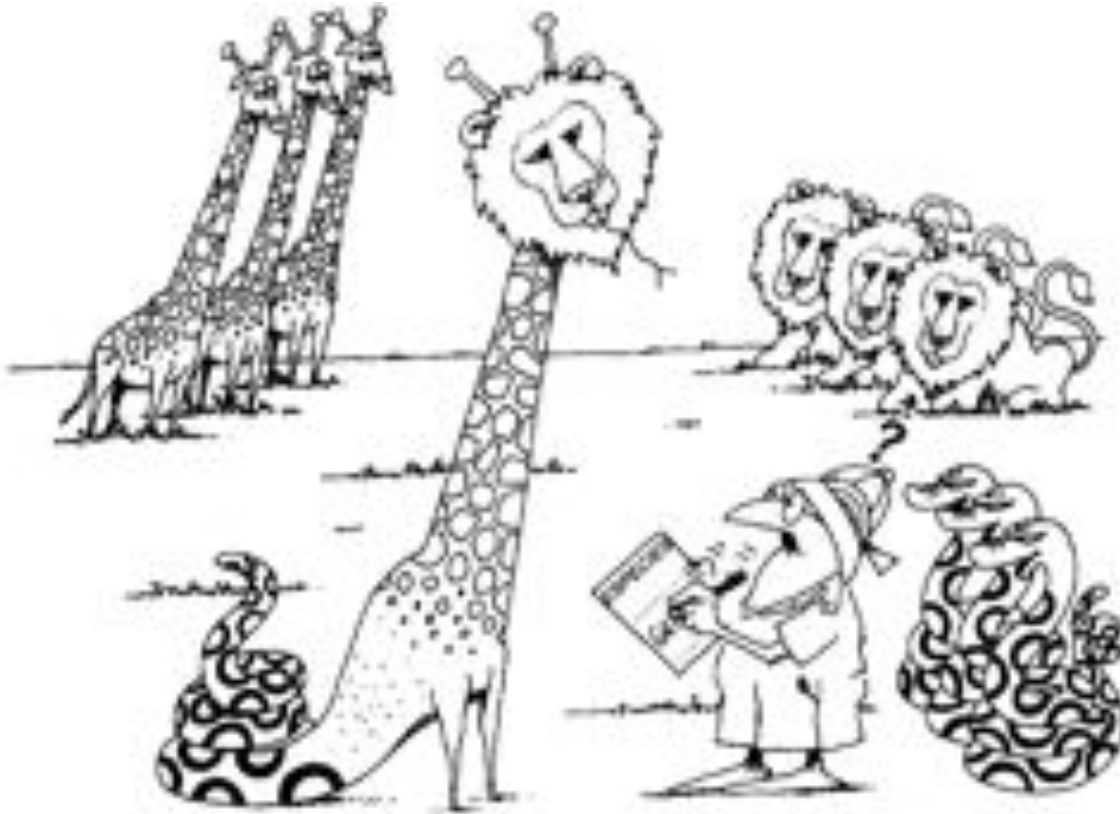
- La **AGREGACIÓN**. Mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo una computadora, por descomponerse en: la CPU, la ULA, la memoria y los dispositivos periféricos. El contrario de agregación es la descomposición.
- La **CLASIFICACIÓN**. Consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La ejemplificación es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular.

Clasificación



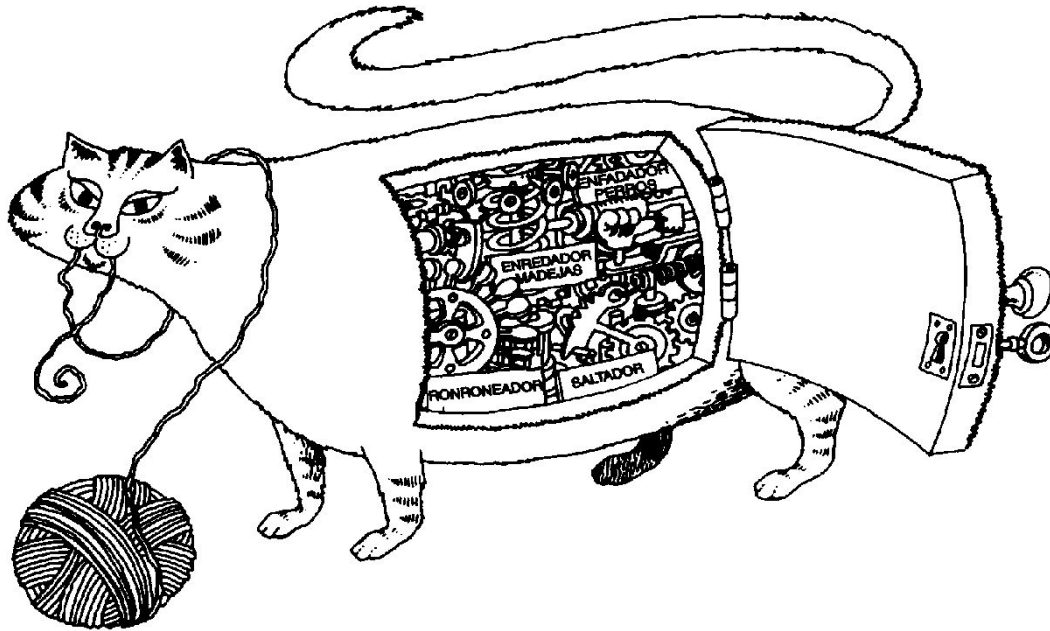
Las clases y objetos deberían estar al nivel de abstracción adecuado:
ni demasiado alto ni demasiado bajo

Clasificación



La clasificación es el medio por el que ordenamos, el conocimiento ubicado en las abstracciones.

¿Qué es el encapsulamiento?



El encapsulamiento en un sistema orientado a objeto se representa en cada clase u objeto, definiendo sus atributos y métodos con los siguientes **modos de acceso**:

- Público (+) Atributos o Métodos que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella.
- Privado (-) Atributos o Métodos que solo son accesibles dentro de la implementación de la clase.
- Protegido (#): Atributos o Métodos que son accesibles para la propia clase y sus clases hijas (subclases).

El encapsulamiento oculta los detalles de implementación de un objeto.

Ejemplo

```
class Persona{  
    private nombre:string;  
    private apellido:string;  
    private añoNac:number;  
    constructor (nombre:string,  
apellido:string) {  
        this.nombre = nombre;  
        this.apellido = apellido;  
    }  
  
    get Nombre():string {  
        return this.nombre;  
    }  
  
    get Apellido():string {  
        return this.apellido;  
    }  
    ...  
}
```

La clase Persona encapsula los atributos a fin de que, no tomen valores inconsistentes.

El encapsulamiento da lugar al Principio de Ocultamiento: sólo los métodos de una clase deberían tener acceso directo a los atributos de esa clase, para impedir que un atributo sea modificado en forma insegura, o no controlada por la propia clase.

Diferencia entre abstracción y encapsulamiento

ABSTRACCIÓN	ENCAPSULAMIENTO
Busca la solución en el diseño	Busca la solución en la implementación
Únicamente información relevante	Ocultación de código para protegerlo
Centrado en la ejecución	Centrado en la ejecución

¿Qué es la modularidad?



Es la propiedad que permite tener independencia entre las diferentes partes de un sistema. La modularidad consiste en dividir un programa en módulos o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros módulos. En un mismo módulo se suele colocar clases y objetos que guarden una estrecha relación. El sentido de modularidad está muy relacionado con el ocultamiento de información.

Empaqueta abstracciones en unidades discretas

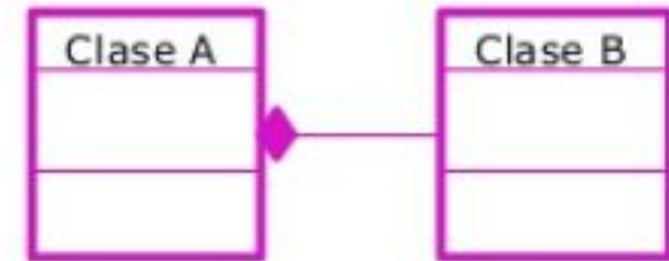
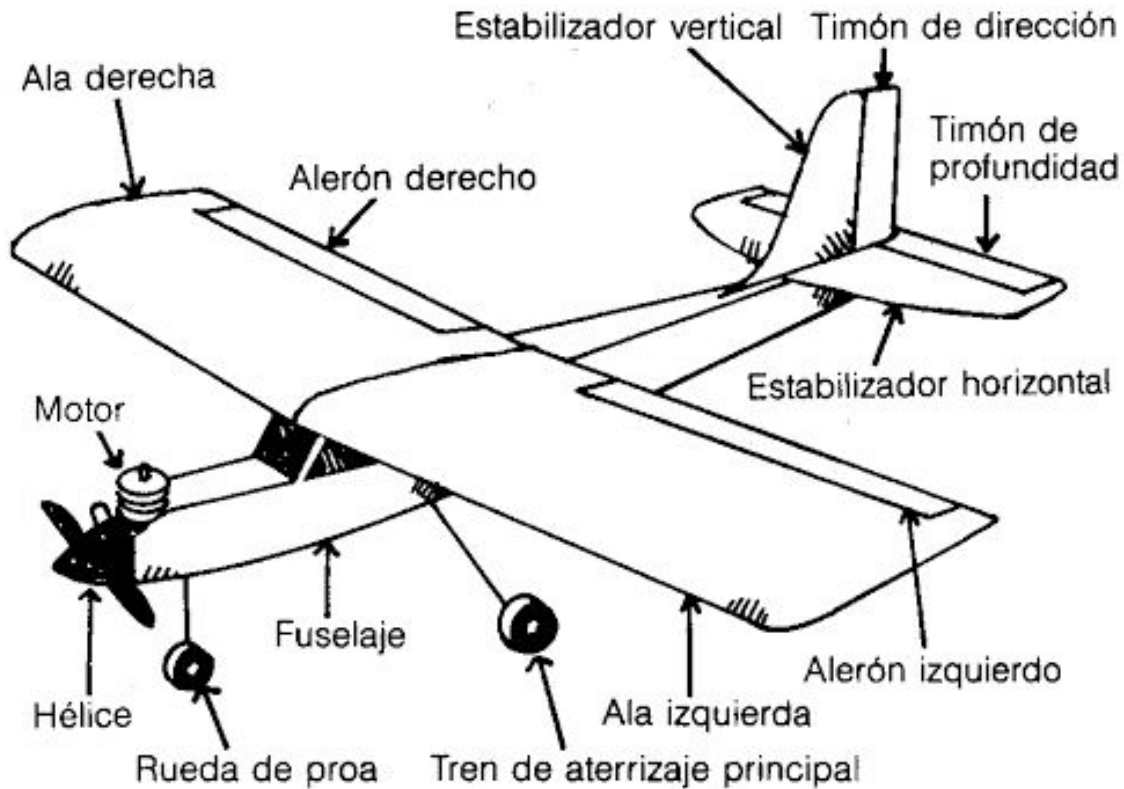
Jerarquía de Clases



La jerarquía es una clasificación u ordenación de abstracciones.

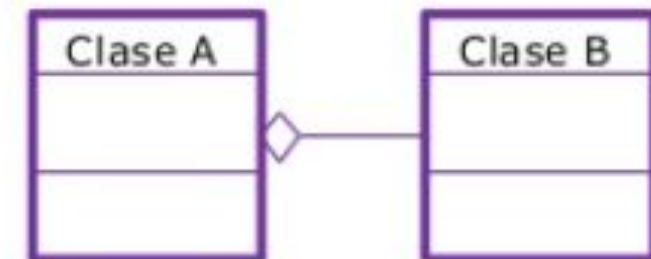
Las dos jerarquías más importantes en un sistema complejo son su estructura de clases (la jerarquía "de clases") y su estructura de objetos (la jerarquía "de partes").

Jerarquía de Partes: Agregación y Composición

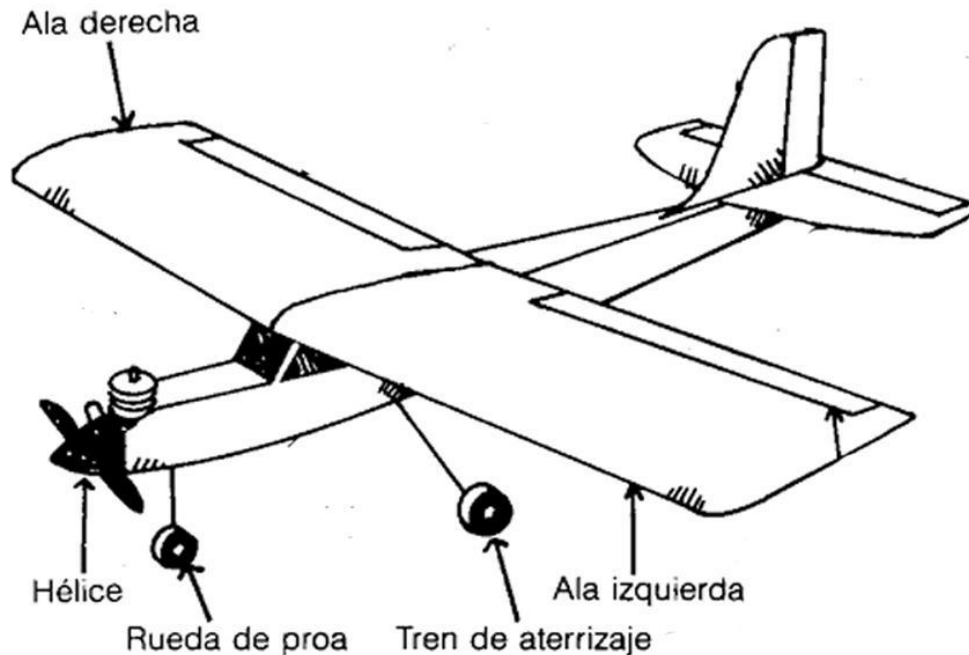


Comprende relaciones del tipo "es parte elemental de" o "tiene un" al realizar una descomposición (Composición)

Comprende relaciones del tipo "es parte de" al realizar una descomposición (Agregación)



Implementación de Clases. Caso de estudio: El Aeroplano



Aeroplano:

- 1 hélice frontal.
- Tren de aterrizaje fijo, tiene 3 neumáticos y 3 amortiguadores.
- 2 alas frontales y 3 de cola.
- La cubierta cuenta con sólo una cabina de vuelo, 1 tanques de combustible, 1 puerta de salida.

Implementación de Clases. Caso de estudio: El Aeroplano

```
class Turbina
{
    private numTurbinas:number = 0;
    public constructor( n :number)
    {
        this.numTurbinas = n;
    }
    public ToString()
    {
        return this.numTurbinas + " Turbina/s";
    }
}

class Helice
{
    private numHelices:number = 0;
    public constructor( n:number)
    {
        this.numHelices = n;
    }
    public ToString()
    {
        return this.numHelices + " hélice/s";
    }
}
```

```
class TrendeAterrizaje
{
    private numNeumaticos:number = 0;
    private numAmortiguadores:number= 0;
    private fijoRetractil:boolean = false;
    public constructor(a:number, b:number, c:boolean)
    {
        this.numNeumaticos = a;
        this.numAmortiguadores = b;
        this.fijoRetractil = c;
    }
    public ToString()
    {
        let mensaje:string = "Tren de Aterrizaje compuesto por: ";
        if (this.fijoRetractil)
        {
            mensaje += " con Retractil fijo, ";
        }
        mensaje += this.numNeumaticos + " neumáticos, " +
        this.numAmortiguadores + " amortiguadores ";
        return mensaje;
    }
}
```



Clases que forman las “partes”

Implementación de Clases. Caso de estudio: El Aeroplano

```
class Cubierta
{
    private cabinaTripulacion:boolean = false;
    private cabinaVuelo:boolean = false;
    private sistemaEmergencia:boolean = false;
    private numTanquesCombustible:number = 0;
    private numPuertasSalidas:number = 0;
    public constructor( pCabinaTripulacion:boolean, pCabinaVuelo:boolean,
pSistemaEmergencia:boolean, pTanquesCombustible:number,
pPuertasSalida:number)
    {
        this.cabinaTripulacion = pCabinaTripulacion;
        this.cabinaVuelo = pCabinaVuelo;
        this.sistemaEmergencia = pSistemaEmergencia;
        this.numTanquesCombustible = pTanquesCombustible;
        this.numPuertasSalidas = pPuertasSalida;
    }

    public ToString()
    {
        let mensaje = "Cubierta compuesta de: ";
        if (this.cabinaVuelo)
        {
            mensaje += " Cubierta de Vuelo, ";
        }
    }
}
```

```
class Alas
{
    private numAlasFrente:number = 0;
    private numAlasCola:number = 0;
    public constructor( mAlasFrente:number, nAlasCola:number)
    {
        this.numAlasFrente = mAlasFrente;
        this.numAlasCola = nAlasCola;
    }

    public ToString()
    {
        return "Alas Frontales: " + this.numAlasFrente + " Alas Posteriore: "
+ this.numAlasCola;
    }
}
```



Clases que forman las “partes”

Implementación de Clases. Caso de estudio: El Aeroplano

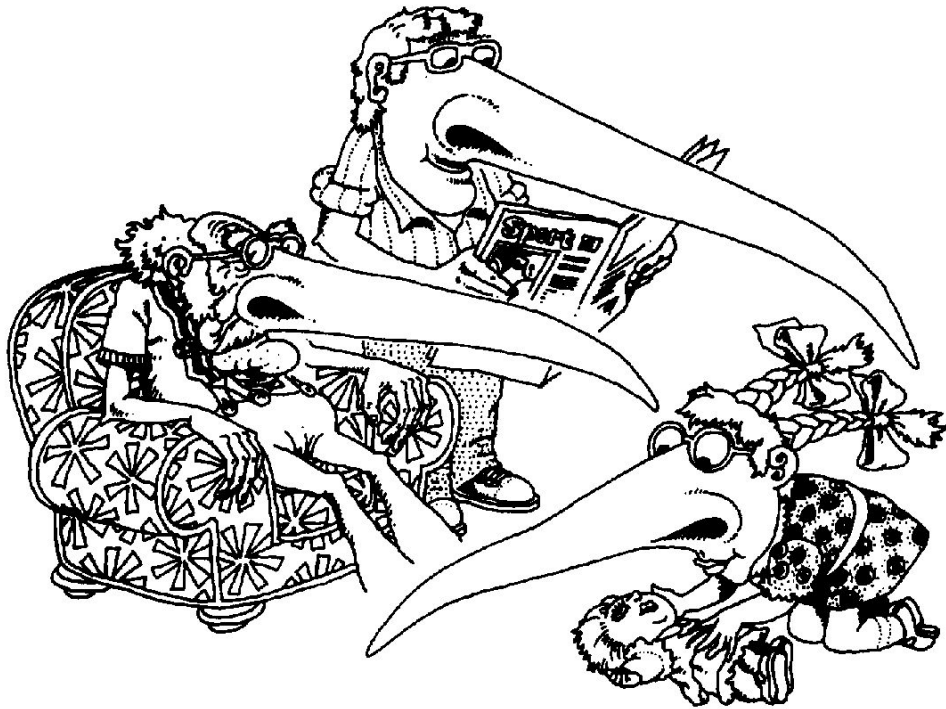
```
class Aeroplano
{
    private helice: Helice ;
    private trenAterrizaje:TrendeAterrizaje ;
    private alas: Alas;
    private cubierta:Cubierta ;

    public constructor
    ( phelice:Helice, pTrenAterrizaje:TrendeAterrizaje, pAlas:Alas, pCubierta:Cubierta)
    {
        this.helice = phelice;
        this.trenAterrizaje = pTrenAterrizaje;
        this.alas = pAlas;
        this.cubierta = pCubierta;
    }
    public ToString()
    {
        let mensaje = "Aeroplano compuesto por: ";
        mensaje += this.helice.ToString();
        mensaje += this.alas.ToString();
        mensaje += this.trenAterrizaje.ToString();
        mensaje += this.cubierta.ToString();
        return mensaje;
    }
}
```

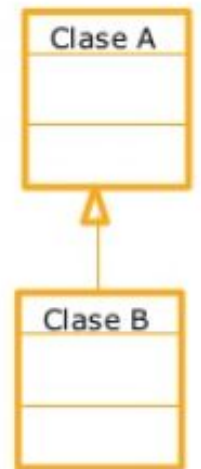
Clases que forma el “todo”

Descarga el fuente de las
clases [aquí](#).

Jerarquía de Clases: Herencia



Permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial), evitando repetición de código y permitiendo la reusabilidad.



La herencia es una herramienta que permite definir nuevas clases en base a otras clases existentes.

Para más info hacé clic [aquí](#)

Herencia. Ejemplo

```
class A
{
    public Method01()
    {
        console.log("Este método es público");
    }
}

class B extends A{
}

let b= new B();
b.Method01();
```

La clase B hereda "extends"
de A

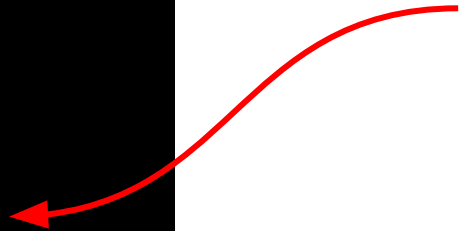
Los miembros públicos son visibles en
las clases derivadas y forman parte de
la interfaz pública de dichas clases.

Herencia. Miembros Privados. Ejemplo

```
class A
{
    protected _valueProtected = "Valor  
accedido por A";
    public Method01()
    {
        console.log("Este método es  
público");
    }
}

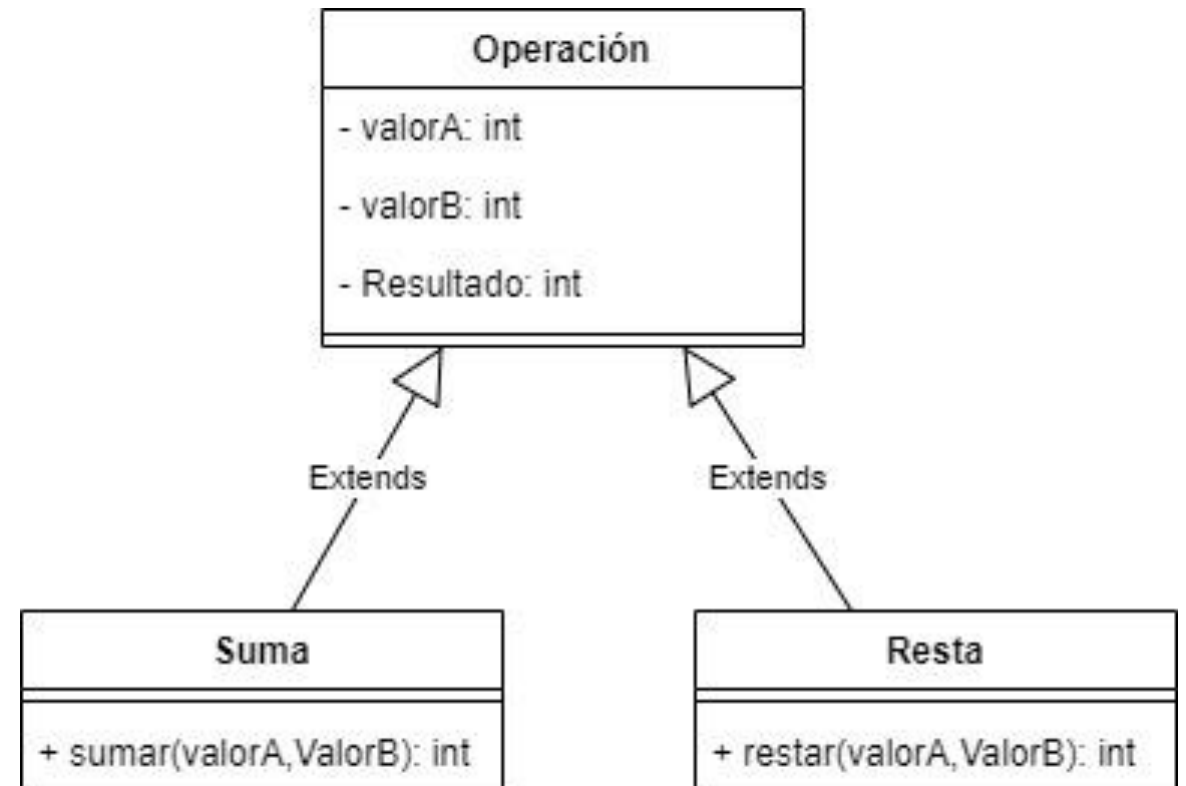
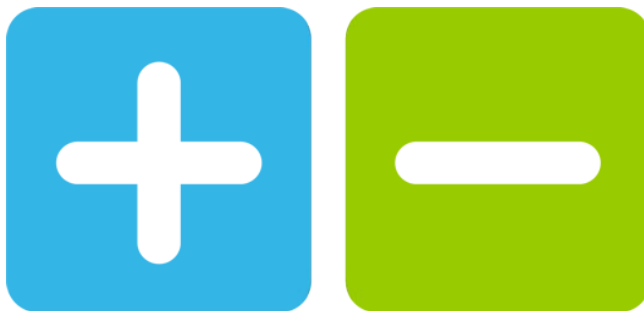
class B extends A{
    public Method02()
    {
        return this._valueProtected;
    }
}
```

Los miembros protected solo son visible en las clases derivadas que están anidadas en su clase base.



Implementación de Clases. Caso de estudio: Las operaciones

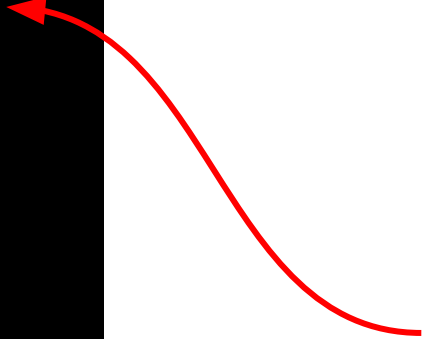
Necesitamos implementar dos clases que llamaremos **Suma** y **Resta**. Cada clase tiene como atributo dos valores, el resultado y las operaciones correspondientes a cada operación.



Implementación de Clases. Caso de estudio: Las operaciones

```
class Operacion{  
    protected _valorA:number;  
    protected _valorB:number;  
    protected _resultado:number;  
    constructor () {  
        this._valorA=0;  
        this._valorB=0;  
        this._resultado=0;  
    }  
  
    set ValorA(value:number) {  
        this._valorA=value;  
    }  
    set ValorB(value:number) {  
        this._valorB=value;  
    }  
  
    get Resultado():number {  
        return this._resultado;  
    }  
}
```

Clase Base o
SuperClase



Implementación de Clases. Caso de estudio: Las operaciones

```
class Suma extends Operacion
{
    Sumar ()
    {
        this._resultado=this._valorA+this._valorB;
    }
}
```

```
class Resta extends Operacion
{
    Restar ()
    {
        this._resultado=this._valorA-this._valorB;
    }
}
```



SubClases o
Clases Derivadas

Implementación de Clases. Caso de estudio: Las operaciones

```
let operacionS= new Suma();  
operacionS.ValorA=45;  
operacionS.ValorB=10;  
operacionS.Sumar();  
console.log("El resultado de la suma es " +  
operacionS.Resultado);
```

```
let operacionR= new Resta();  
operacionR.ValorA=45;  
operacionR.ValorB=10;  
operacionR.Restar();  
console.log("El resultado de la resta es " +  
operacionR.Resultado);
```

Modelo de Objetos: Características



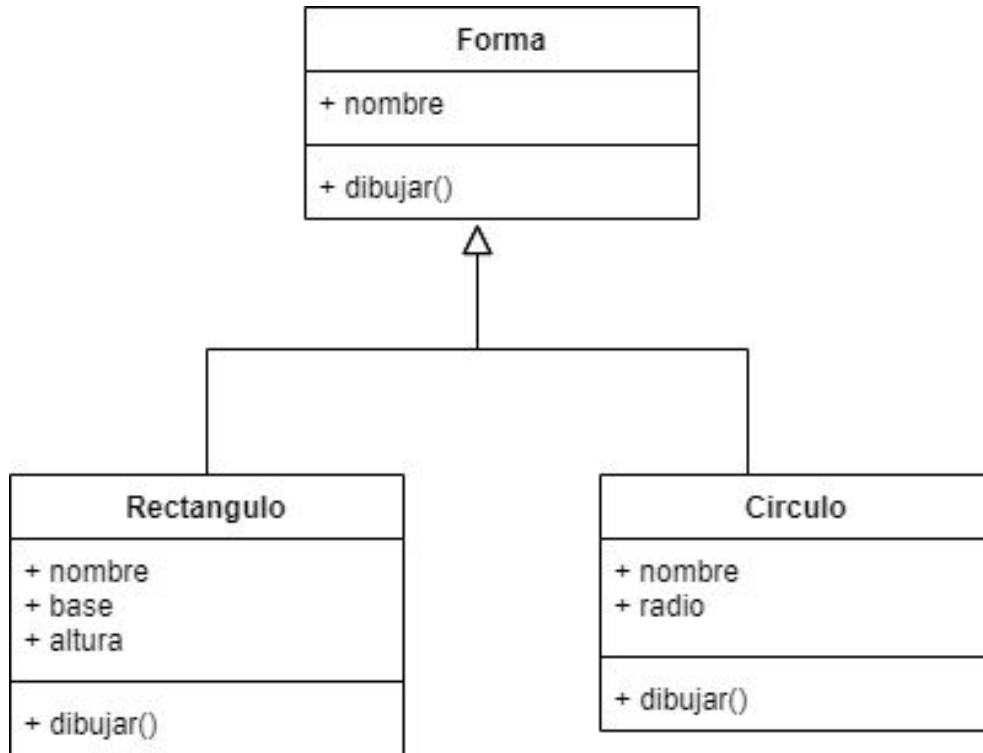
Polimorfismo



Clases diferentes
(polimórficas)
implementan métodos
con el mismo nombre.

Comportamientos diferentes, asociados a objetos distintos pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando.

Polimorfismo por Herencia



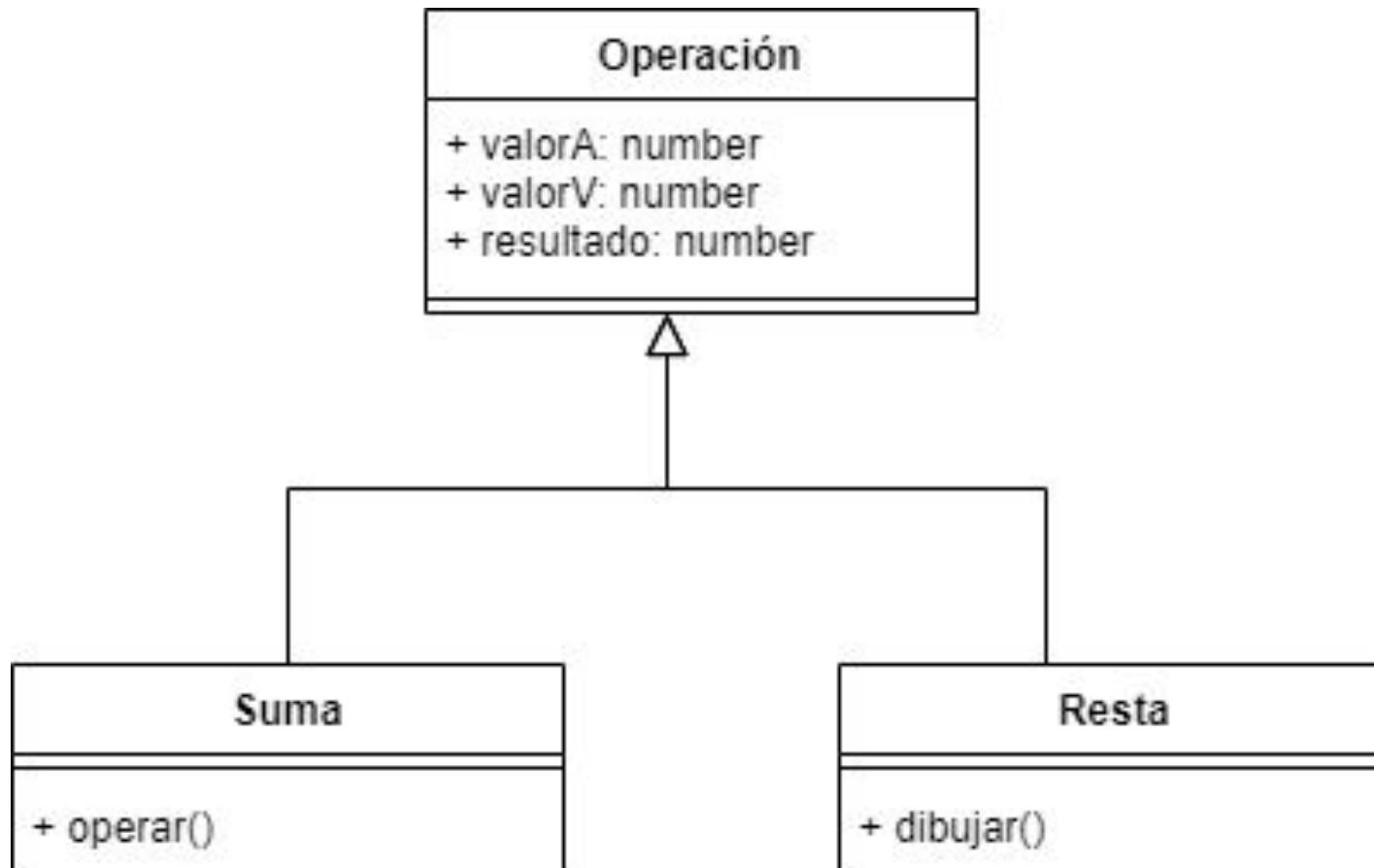
Para más información hacer clic [aquí](#)

Es una propiedad del EOO que permite que un método tenga múltiples implementaciones, que se seleccionan en base al tipo objeto indicado al solicitar la ejecución del método.

El polimorfismo operacional permite aplicar operaciones con igual nombre a diferentes clases o están relacionados en términos de inclusión. En este tipo de polimorfismo, los métodos son interpretados en el contexto del objeto particular, ya que los métodos con nombres comunes son implementados de diferente manera dependiendo de cada clase.

Polimorfismo por Herencia. Situación de Análisis

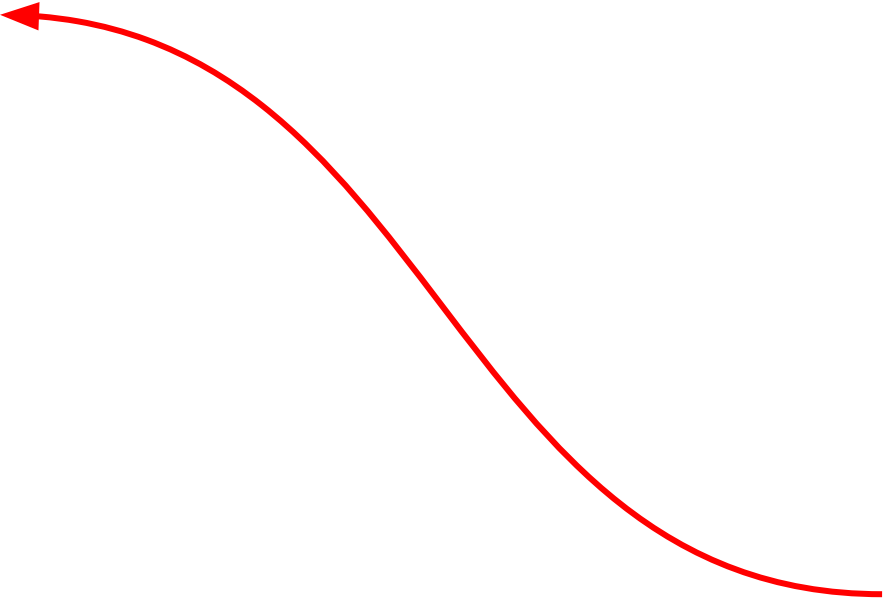
El ejemplo de las operaciones visto anteriormente ¿se puede resolver haciendo uso del polimorfismo?



Polimorfismo por Abstracción

Clase Abstracta. Clase que no es posible crear instancias. Frecuentemente, están implementadas parcialmente o no están implementadas. Forzosamente se ha de derivar si se desea crear objetos de la misma.

```
abstract class Operacion{  
    protected _valorA:number;  
    protected _valorB:number;  
    protected _resultado:number;  
    constructor() {  
        this._valorA=0;  
        this._valorB=0;  
        this._resultado=0;  
    }  
  
    abstract Operar():void;  
  
    set ValorA(value:number){  
        this._valorA=value;  
    }  
    set ValorB(value:number){  
        this._valorB=value  
    }  
  
    get Resultado():number {
```



Obligatoriamente la clase derivada deberá implementar este método.

Polimorfismo por Abstracción

```
class Suma extends Operacion
{
    Operar ()
    {
        this._resultado=this._valorA+this._valorB;
    }
}
```

```
class Resta extends Operacion
{
    Operar ()
    {
        this._resultado=this._valorA-this._valorB;
    }
}
```



SubClases o
Clases Derivadas

Polimorfismo por Interfaces

Interface. Es un CONTRATO. Define propiedades y métodos, pero no implementación.

```
interface IOperacion{  
    Operar(a:number,b:number):number;  
}
```

Interface



```
class Suma implements IOperacion{  
    Operar(a:number,b:number):number{  
        return a+b;  
    }  
}
```

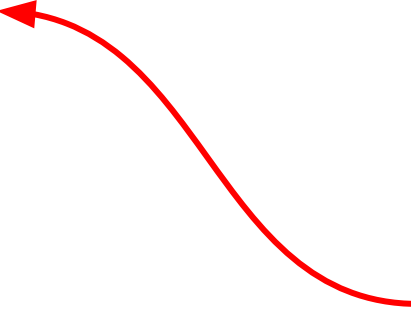
Implementación



Polimorfismo por Interfaces

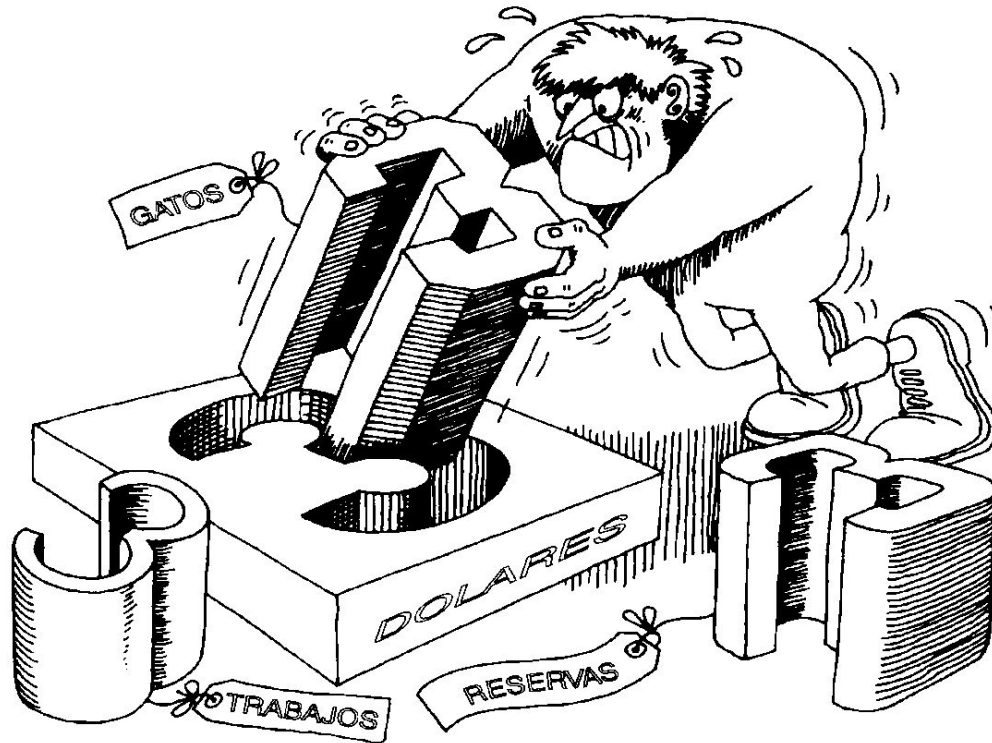
```
public class Suma : IOperacion
{
    public int Operar(int valor1, int valor2)
    {
        return valor1 + valor2;
    }
}
```

```
public class Reat : IOperacion
{
    public int Operar(int Valor1, int Valor2)
    {
        return Valor1 - Valor2;
    }
}
```



Clases que
implementan la
Interface

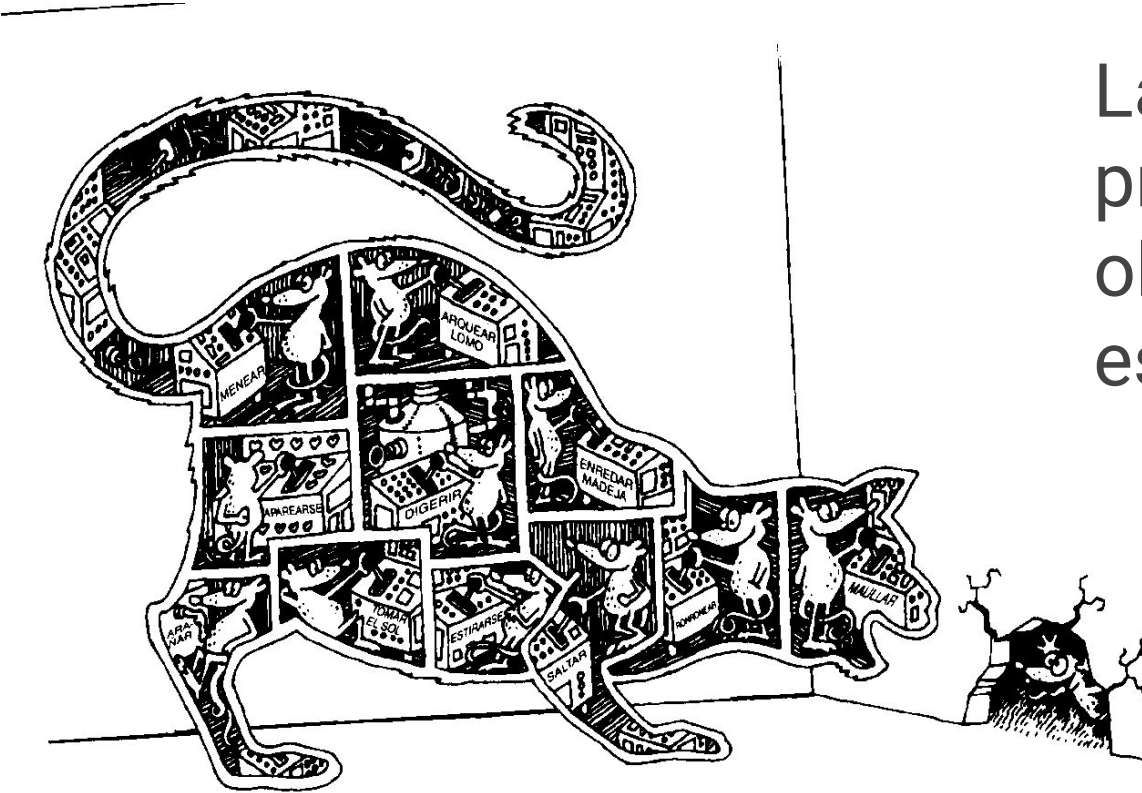
Tipificación



Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas.

Caracterización precisa de propiedades estructurales o de comportamiento que comparten ciertas entidades.

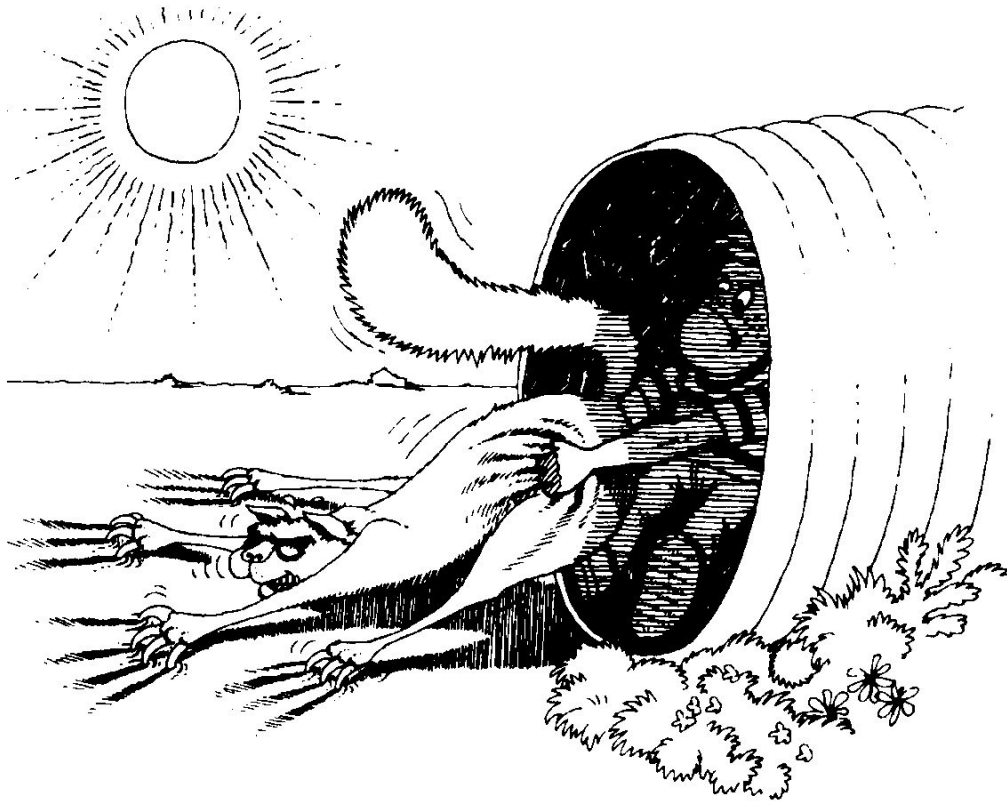
Concurrencia



La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.

La concurrencia permite a dos objetos actuar al mismo tiempo.

Persistencia



La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

Conserva el estado de un objeto en el tiempo y en el espacio.

REFERENCIAS

https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teor%C3%ADa/concepts.html

<https://docs.microsoft.com/>

Apuntes Programa Clip - Felipe Steffolani