Programación Orientada a Objetos (POO)

Sitio: Centros - Cádiz
Curso: Programación
Libro: Programación Orientada a Objetos (POO)

Imprimido por: Barroso López, Carlos
Día: martes, 21 de mayo de 2024, 23:37

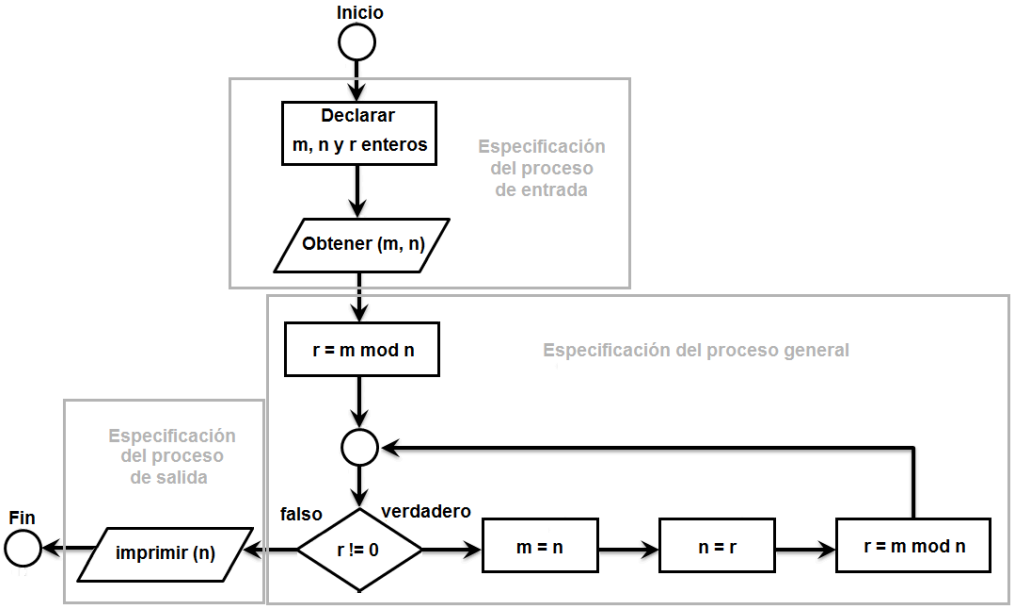
Tabla de contenidos

1. POO: Abstracción y Encapsulamiento

1. Enfoque OO

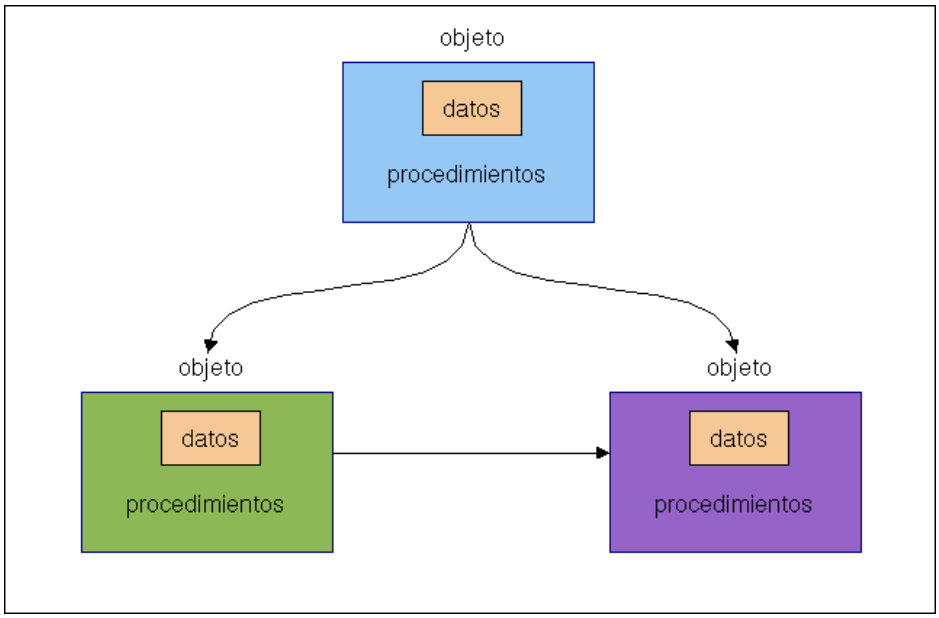
En la **Programación Estructurada**:

Los datos y las operaciones que se realizan sobre ellos, se definen y utilizan de forma separada. Además, la comunicación entre las diferentes partes o módulos de un programa se realiza mediante el uso de parámetros.



En la **POO**:

Los datos y las operaciones se agrupan formando entidades denominadas **objetos**, las cuales representan elementos que se desean modelizar y programar, de este modo existe una relación entre los datos y sus funcionalidades.



2. Abstracción de datos y Encapsulamiento

2.1. Abstracción de datos

La abstracción de datos consiste en ocultar los detalles de la implementación de un objeto, únicamente importa cómo utilizarlo.

Por tanto, en un objeto se pueden distinguir dos partes:

a. **Una parte privada**

Oculta al programa que lo utiliza, la cual debe estar protegida y es irrelevante para su uso.

Se compone tanto por la **implementación de las operaciones** como por los **datos** (información interna necesaria para la implementación).

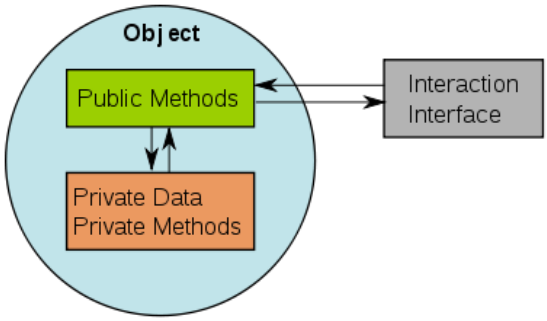
b. **Una parte pública** (*interfaz*)

Representa el comportamiento del objeto y permite interactuar con él por medio de **mensajes**.

2.2. Encapsulamiento

El encapsulamiento de un objeto garantiza que sólo se pueda acceder a él a través de su interfaz y por medio del envío de mensajes.

La encapsulación consiste en hacer que los datos sean modificados únicamente por las funciones destinadas a tal efecto, para asegurar que los datos conserven un estado válido y consistente, y evitar su modificación no contralada con el consiguiente problema de generación de inconsistencias.



3. Lenguajes OO

La tendencia hacia la POO, ha hecho que lenguajes tradicionales evolucionen para dar soporte a la OO, como por ejemplo **C++**.

Por otro lado, existen lenguajes diseñados exclusivamente para la POO, como son Smalltalk o **Java**.

2. Objetos y Clases

1. Objeto

Es un **tipo de dato abstracto** (TDA) que modeliza una entidad de la cual se conoce su comportamiento, pero no los detalles de su representación interna.

Características de un objeto:

i. **Identidad**

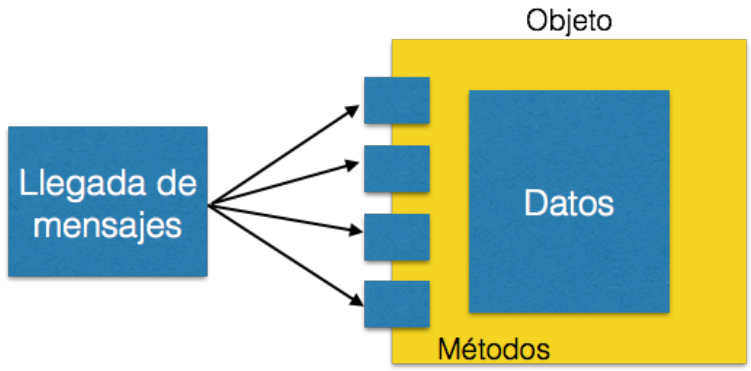
La identidad o identificación es la propiedad que permite diferenciar a un objeto de otro. Generalmente esta propiedad viene determinada por su **identificador**.

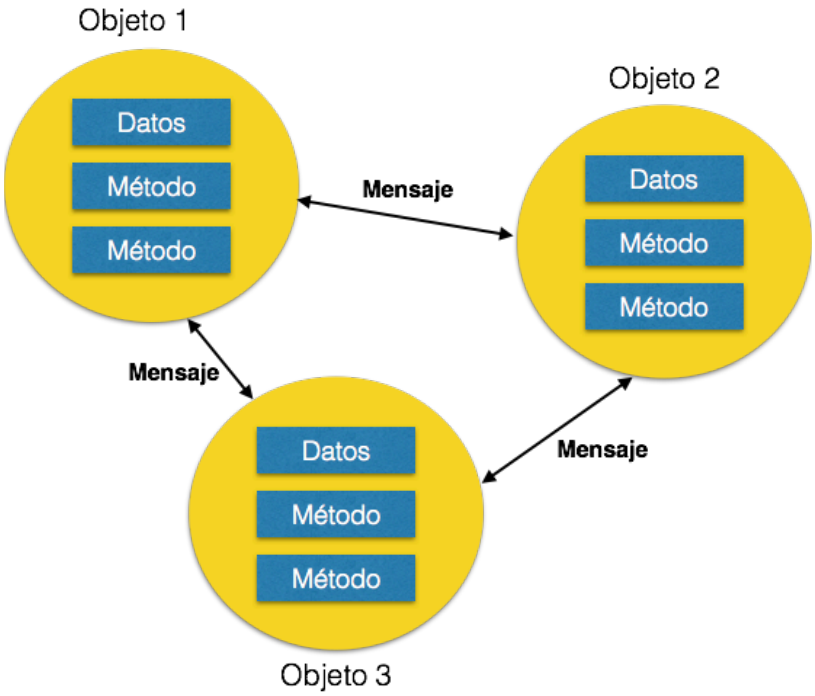
ii. **Comportamiento**

El comportamiento de un objeto está relacionado con su funcionalidad, y determina las operaciones que éste puede realizar (**métodos**) como respuesta de los **mensajes recibidos** por parte de otros objetos.

iii. **Estado**

El estado de un objeto se refiere al **conjunto de atributos** y sus valores en un instante determinado. El comportamiento de un objeto puede modificar su estado.





2. Clase

Una clase es un **modelo o plantilla** a partir del cual se crean los objetos de dicha clase que sean necesarios para el programa.

Define los atributos y los métodos para operar con dichos datos. A los elementos declarados en una clase, tanto atributos como métodos, se les denominan **miembros** de dicha clase.

2.1. Declaración

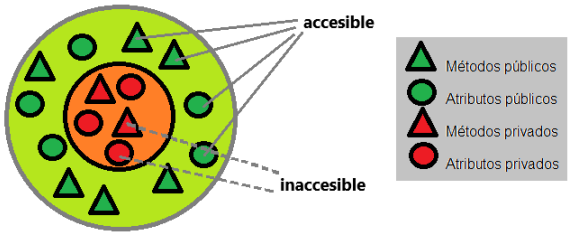
La sintaxis típica de una clase es:

```
class Nombre {
    // Variables miembro o atributos (habitualmente privadas)
    atributo_1; // Pueden almacenar datos simples, estructurados u otros objetos
    atributo_2;
    atributo_3;
     // Funciones miembro o métodos (habitualmente públicas)
    metodo_1();
    metodo_2();
}
```

2.2. Nivel de acceso

Los miembros de una clase, según su nivel de acceso, pueden ser:

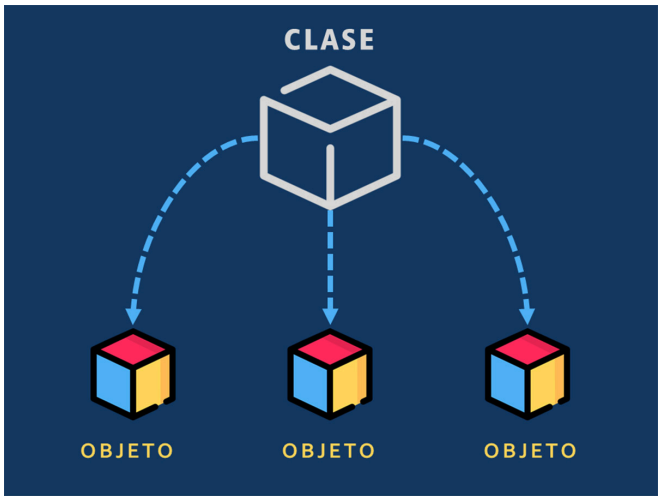
- **Privados**: sólo son accesibles desde la propia clase. Es el nivel por defecto.
- **Públicos**: accesibles desde fuera de la clase.



2.3. Instanciación

Las clases no son utilizables directamente, requieren de una instanciación, es decir, la creación de un objeto de dicha clase, el cual dispone de memoria reservada y valores propios.

Cada objeto creado a partir de la clase se denomina **instancia de la clase**.



2.4. Métodos especiales

i. **Constructores y Destructores**

Todas las clases disponen de métodos para crear y destruir objetos llamados constructores y destructores. Si no son definidos, el compilador crea uno **por defecto**.

Constructor

Se llama de forma automática al crear un objeto, asignando un valor inicial a sus atributos.

Se pueden definir diferentes constructores sobrecargando dicho método, ejecutando uno u otro según los parámetros recibidos.

Destructor

Se encarga de eliminar un objeto de memoria. A diferencia del constructor, es único y no recibe ningún parámetro.

ii. **Get y Set**

Los atributos de un objeto suelen ser privados debido a su encapsulamiento.

Por tanto, para permitir el acceso (consulta y modificación) a dichos atributos de forma controlada, se realiza habitualmente mediante pares de métodos denominados: get y set.

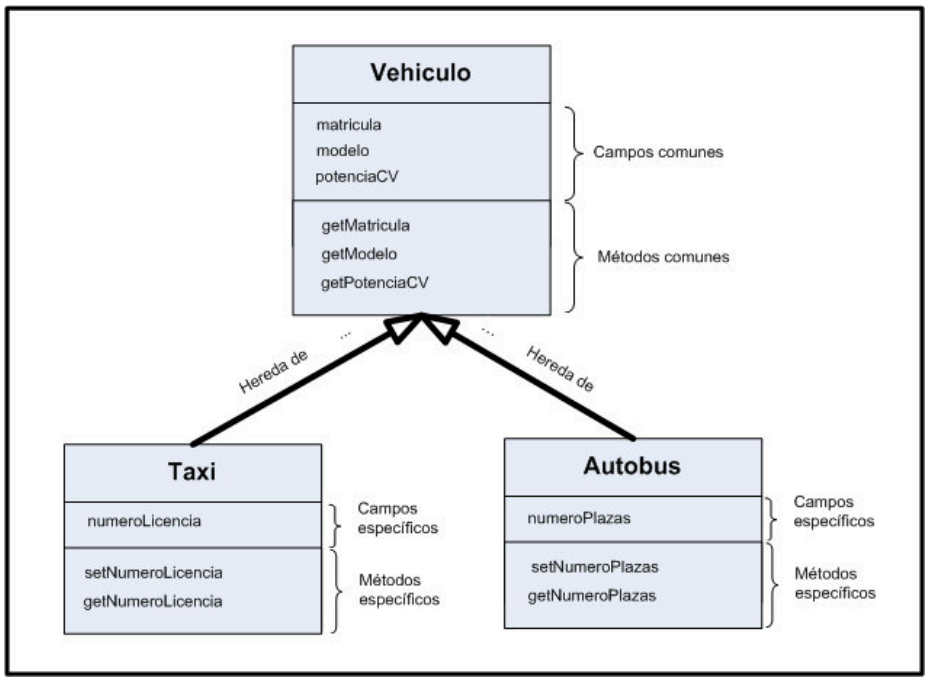

```
public class Perro
{
     private int Peso;
     public int GetPeso()
     {
          return Peso;
     }
     public void SetPeso(int newPeso)
     {
          if (newPeso > 0)
          {
               Peso = newPeso;
          }
     }
}
```

3. Herencia

1. Definición

La herencia es un mecanismo de la POO que permite **definir una nueva clase a partir de otras ya existentes** extendiendo su funcionalidad.

La nueva clase adquiere los miembros (atributos y métodos) de la clases que hereda, pudiendo añadir los suyos propios o redefinir los heredados.



2. Ventajas y desventajas

Ventajas

- La principal ventaja de este mecanismo es que facilita la **reutilización de código**.

Los atributos y métodos comunes, implementados en la clase de la que se hereda, no se tienen que volver a implementar en las clases que heredan, únicamente será necesario escribir el código de la parte específica que requieran.

Desventajas

Si la jerarquía de clases es demasiado compleja, el programador puede tener problemas para comprender el funcionamiento de un programa.

Además puede volverse más complejo detectar y resolver errores de programación.

3. Tipos de relaciones

El mecanismo de herencia puede representar distintos tipos de relaciones entre las clases, como por ejemplo:

- Especialización/Generalización
- Extensión de funcionalidades
- Plantilla con elementos comunes
- Etc.

3.1. Jerarquía de clases

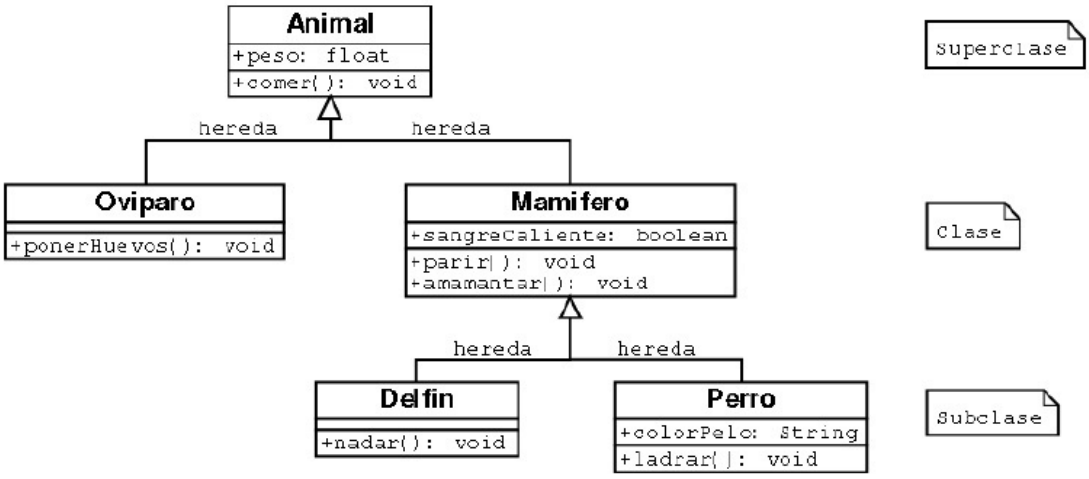
La definición de clases utilizando el mecanismo de la herencia hace que se establezca entre ellas una relación jerárquica, conocida como *jerarquía de clases*.

1. Clase base y clase derivada

Dentro de una jerarquía de clases, éstas se pueden clasificar en dos tipos:

- **Clase base** (o *superclase*): clase de la que se hereda.
- **Clase derivada** (o *subclase*): clase que hereda.

Ejemplo:

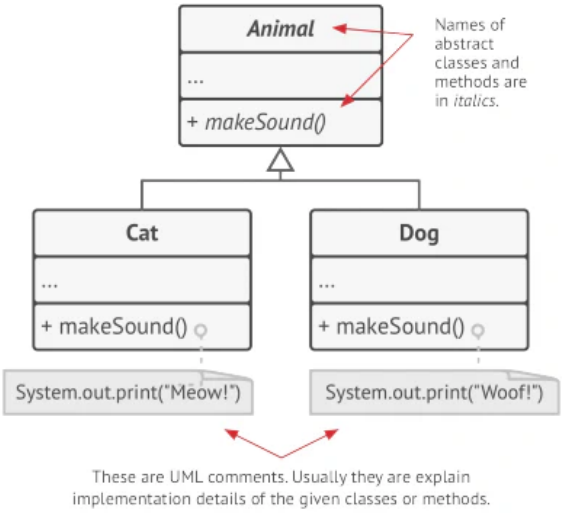


2. Clases abstractas

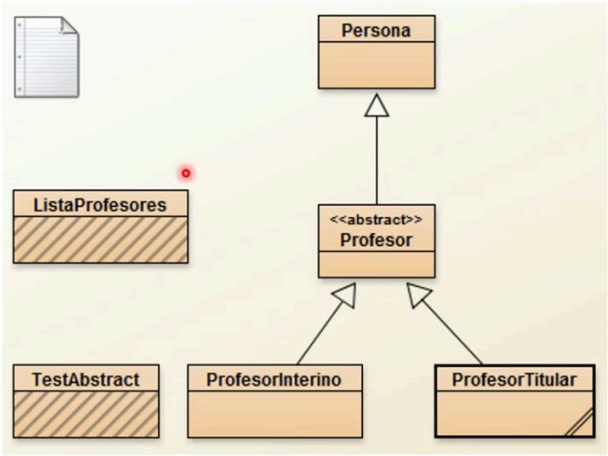
Una clase abstracta es una clase cuya implementación no está completa y de la que **no se pueden crear instancias** (objetos). Sirve sólo como base para crear otras clases derivadas.

Este mecanismo permite definir comportamientos (métodos) cuya implementación se realiza en las clases derivadas. Por tanto, se obliga a las clases derivadas a tener que implementarlos.

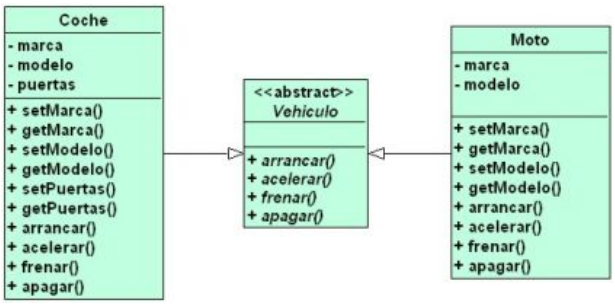
Representaciones:



EJEMPLO



Ejemplo:



3.2. Nivel de acceso

Para proteger y separar la implementación interna de un objeto de su interfaz pública (**encapsulamiento**), las clases disponen de una serie de niveles de acceso que se pueden establecer a la hora de definir sus miembros, e incluso la propia clase.

1. Nivel de acceso de los miembros de una clase

La siguiente tabla muestra si es o no accesible un miembro de una clase base según su nivel de acceso:

Nivel de acceso / Tipo de clase	*Clase base*	*Subclase*	*Otras*
public	Sí	Sí	Sí
private	Sí	No	No
protected	Sí	Sí	No

- Nivel de acceso **público**: permite el acceso al miembro desde cualquier clase.
- Nivel **privado**: sólo permite el acceso desde la propia clase donde está definido.
- Nivel **protegido**: permite el acceso desde la propia clase y desde las clases derivadas, pero no desde el resto de clases.

2. Nivel de acceso de una clase

El nivel de acceso de una clase afecta a la hora de heredar sus miembros las clases derivadas y poder ser heredados por otras clases derivadas de estas últimas.

Dos niveles:

Público

Sus miembros públicos y protegidos mantienen su nivel de acceso para las clases derivadas.

Privado

Sus miembros públicos y protegidos pasan a ser privados respecto a las clases derivadas que los heredan.

Acceso no permitido al atributo edad de la clase Animal.

3.3. Tipos de herencia

Existen dos tipos básicos de herencia:

1. Herencia simple

Una clase puede **heredar** características y comportamientos **sólo de una superclase**. En cambio, una superclase sí puede tener una o varias clases derivadas.

Las clases forman una jerarquía en forma de árbol.

Ej.: Java, C#

2. Herencia múltiple

Una clase puede **heredar** características y comportamientos **de más de una superclase**.

Las clases forman una jerarquía en forma de grafo dirigido no cíclico.

Ej.: C++, Python

Nota: Algunos lenguajes que no soportan la herencia múltiple implementan mecanismos que permiten simularla. Ejemplo: Las ***interfaces*** de Java.

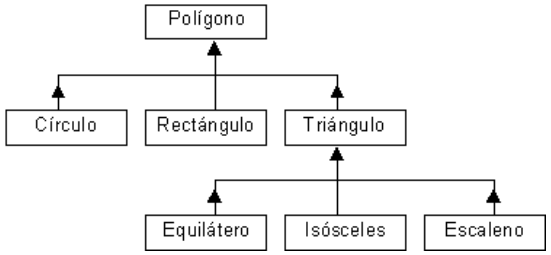
2.1. Problema de ambigüedades

En la herencia múltiple pueden aparecer **ambigüedades** que provoquen la duplicidad de algunos miembros o conflictos en la resolución de nombres.

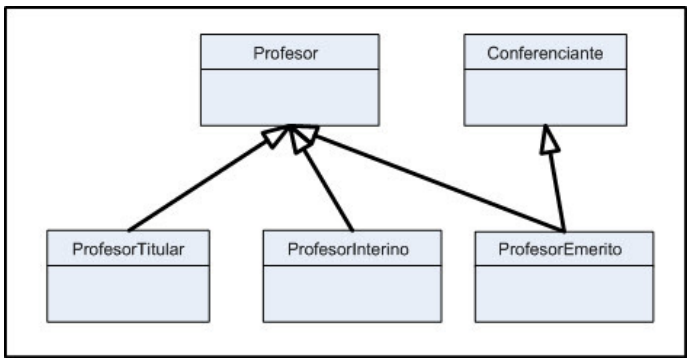
Cada lenguaje de programación trata estos problemas de diferente forma. Por ejemplo, C++ requiere que el programador establezca de qué clase base vendrá la característica a usar.

```
Ej.: Trabajador::Persona.edad
```

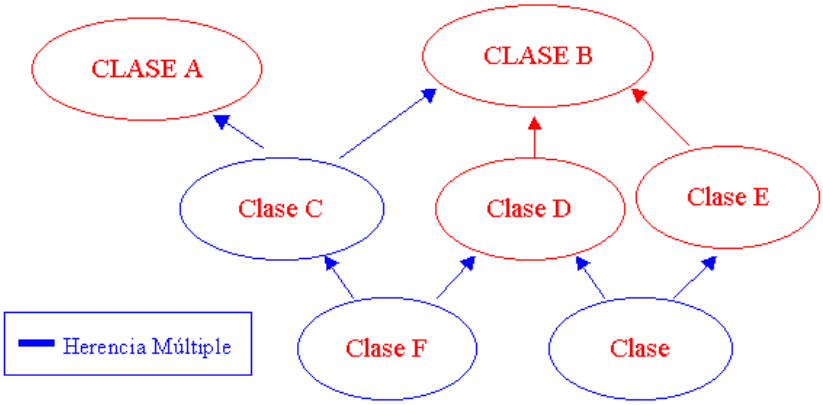
3. Ejemplos



Herencia simple



Herencia simple y múltiple (clase ProfesorEmerito)



Herencia simple y múltiple

4. Polimorfismo

1. Definición

El polimorfismo es la propiedad por la que es posible **enviar mensajes sintácticamente iguales a objetos de diferente tipo** (diferente clase), **respondiendo** cada uno **de manera diferente**.

Esta característica permite que, sin alterar el código existente, se puedan incorporar nuevas funcionalidades. Por tanto, la interfaz sintáctica se mantiene inalterada pero cambia el comportamiento en función de qué objeto se esté utilizando en cada momento.

2. Características

- El polimorfismo se resuelve en **tiempo de ejecución**, en función del tipo de objeto que recibe el mensaje, se ejecutará un método u otro.
- Dado que un grupo de objetos polimórficos generalmente contendrán otros métodos (que otros objetos en dicho grupo no contienen), lo correcto es decir que dichos objetos se pueden utilizar de modo polimórfico para un cierto conjunto de mensajes.

3. Mecanismos

Para poder utilizar un grupo de objetos diferentes de manera polimórfica es necesario que:

a. **Tengan implementado de manera particular un mismo método (o grupo de métodos).** De esta manera, es posible enviarles un mismo mensaje (o grupo de mensajes).

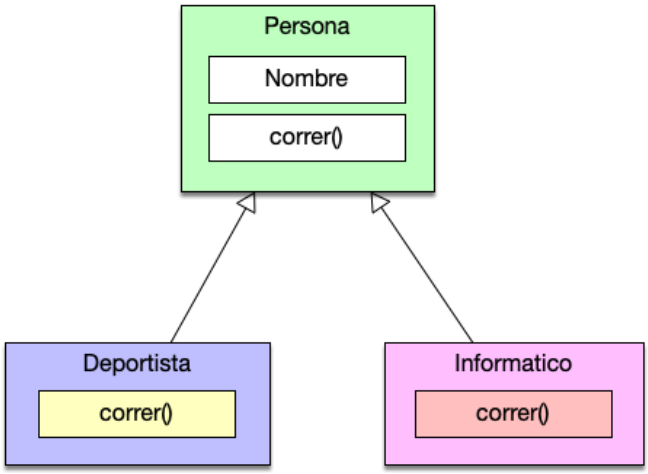
b. **Puedan ser referenciados mediante un mismo tipo de dato**. De este modo el código es independiente del tipo de objeto utilizado, cambiando sólo su funcionamiento.

Los lenguajes habitualmente proporcionan dos mecanismos para implementar el polimorfismo:

3.1. Mediante herencia y uso de métodos abstractos

Si los objetos pertenecen a una misma jerarquía de clases, se puede hacer uso de **métodos abstractos** en la superclase que tienen en común. Esto obligará a que todas sus clases derivadas implementen de manera particular dichos métodos.

Por tanto, todos los objetos de dicha jerarquía de clases tendrán implementados estos métodos, y dicha implementación será diferente según a la clase a la que pertenezcan.



Ejemplo:

Dos clases: Pez y Ave, que heredan de la superclase Animal.

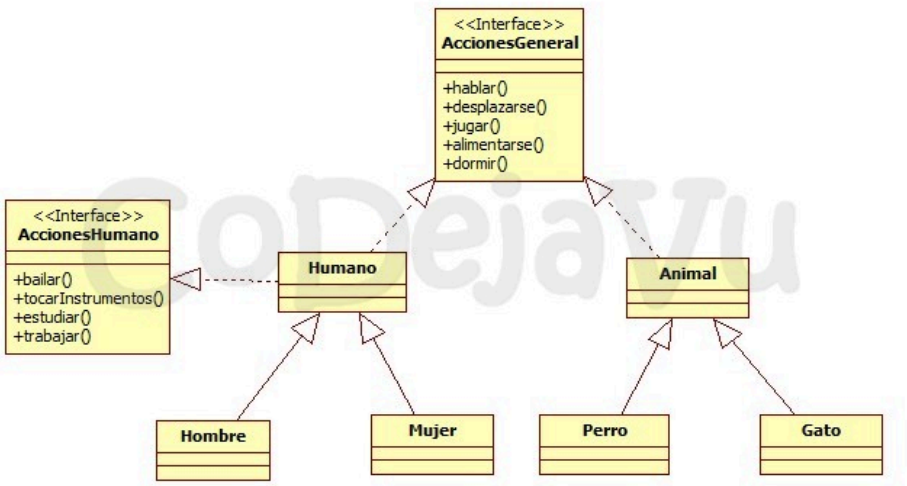
La clase Animal tiene el método abstracto "mover()" que se implementa de forma distinta en cada una de las subclases (peces y aves se mueven de forma distinta).

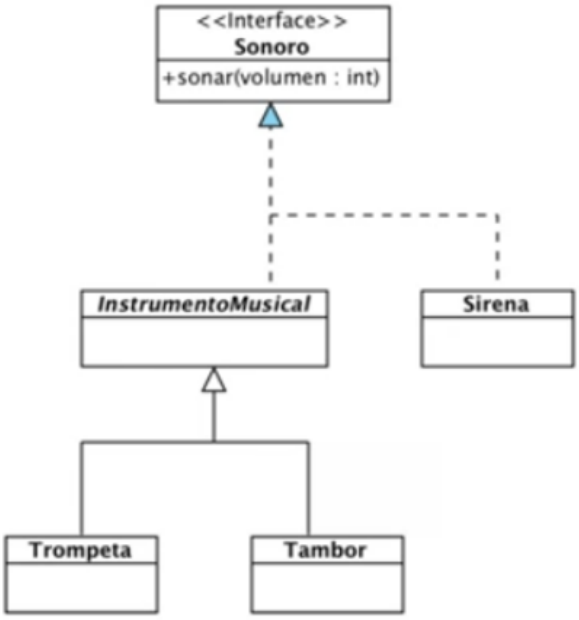
Esto hace posible enviar el mensaje "mover" a un grupo de objetos Pez y Ave, haciendo así un uso polimórfico de dichos objetos respecto del mensaje "mover".

3.2. Mediante interfaces

Si los objetos no pertenecen a una misma jerarquía de clases, o bien no se quiere hacer uso de clases abstractas, lenguajes como C++ o Java permiten hacer uso de interfaces.

Una interfaz sirve para definir una serie de métodos que deberán ser implementados por todas las clases que utilicen (implementen) dicha interfaz. Se puede considerar a la interfaz como un "plantilla" que debe seguir cualquier clase que la implemente.





5. Ejemplos

A continuación, se muestran varios ejemplos de herencia y polimorfismo en C++ y Java.

5.1. Herencia y polimorfismo en C++

```
#include <iostream>
#include <cmath>
using namespace std;

class Figura {  // Clase Figura
 protected:
  float base;
  float altura;
 public:
  void captura();
  virtual float perimetro()=0;
  virtual float area()=0;
};

class Rectangulo: public Figura {  // Clase Rectangulo, que hereda de la clase Figura
 public:
  void imprime();
  float perimetro(){return 2*(base+altura);}
  float area(){return base*altura;}
};

class Triangulo: public Figura {  // Clase Triangulo, que hereda de la clase Figura
 public:
  void muestra();
  float perimetro(){return 2*sqrt(pow(altura,2)+pow((base/2),2))+base;} //Usando pitágoras
  float area(){return (base*altura)/2;}
};

void Figura::captura()  // Implementación del método captura() de la clase Figura
{
  cout << "CALCULO DEL AREA Y PERIMETRO DE UN TRIANGULO ISÓSCELES Y UN RECTANGULO:" << endl;
  cout << "escribe la altura: ";
  cin >> altura;
  cout << "escribe la base: ";
  cin >> base;
  cout << "EL PERIMETRO ES: " << perimetro() << endl;  // USO POLIMÓRFICO DE LOS MÉTODOS perimetro()
  cout << "EL AREA ES: " << area() << endl;            // y area()
  system("pause");

}
```

```
main () {
    Rectangulo rec;
    Triangulo tri

    rec.captura();
    tri.captura();
}
```

5.2. Herencia y polimorfismo en Java

```
public abstract class Persona {

  private String nombre;

  public String getNombre() {
    return nombre;
  }

  public void setNombre(String nombre) {
    this.nombre = nombre;
  }

  public Persona(String nombre) {
    super();
    this.nombre = nombre;
  }

  public abstract int correr() ;
}
```

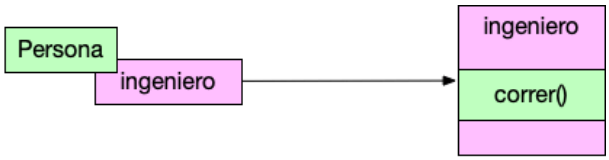
```
public class Deportista extends Persona {

  public Deportista(String nombre) {
    super(nombre);

  }

  @Override
  public int correr() {

    return 7;
  }

}
```

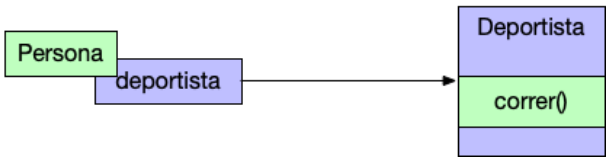
```
public class Ingeniero extends Persona{

  public Ingeniero(String nombre) {
    super(nombre);

  }

  @Override
  public int correr() {

    return 3;
  }

}
```

```
public class Principal {

  public static void main(String[] args) {


    Persona i= new Ingeniero("pedro");
    Persona d= new Deportista("gema");
    System.out.println(i.correr());
    System.out.println(d.correr());


  }

}
```

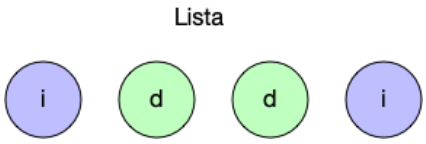
Se utiliza una variable de tipo Persona para referenciar a un Ingeniero y a un Deportista.

Esto es correcto dado que que ambos objetos son del tipo Persona y ambos sobrescriben el método correr() que la clase Persona tiene declarado como abstracto. Este mecanismo es esencial para poder tratar a todos los objetos de manera homogénea.





¿Cómo implementarías un método que calcule la velocidad media de un conjunto de objetos de tipo Ingeniero y Deportista?



5.3. Interfaces y polimorfismo en Java

```
// Interfaz Vehiculo
public interface Vehiculo {
    public void arrancar();  // siempre son públicos, aunque se omita
    public void detener();
}
// Clase Coche, implementa Vehículo
public class Coche implements Vehiculo {
    public void arrancar() {
        System.out.println("arrancando motor...");
    }
    public void detener() {
        System.out.println("deteniendo motor...");
    }
}
// Clase Camión, implementa Vehículo
class Camion implements Vehiculo {
    public void arrancar() {
        System.out.println("arrancando el motor del camión...");
    }
    public void detener() {
        System.out.println("deteniendo el motor del camión...");
    }
}
// Clase Arrancador, únicamente tienen un método estático para llamar al método arrancar()
// del objeto de tipo Vehículo pasado por parámetro.
class Arrancador {
    // método estático, se puede llamar sin instanciar la clase
    public static void arrancarMotor(Vehiculo vehiculo) {  // Ojo: Vehiculo (tipo), vehiculo (variable)
        vehiculo.arrancar();  // POLIMORFISMO
    }
}
// Clase que contiene el programa principal (main)
class Principal {
  public static void main(String[] args) {
    Vehiculo tesla = new Coche();
    Vehiculo tata = new Camion();
    Arrancador.arrancarMotor(tesla); // arrancando motor...
    Arrancador.arrancarMotor(tata); // arrancar el motor del camión...
   }
}
```

Reiniciar tour para usuario en esta página