

Programación Orientada a Objetos

Introducción

- Los problemas suelen tener **varias soluciones** posibles.
- En programación existen **diversas metodologías** que nos ayudan a enfrentar un problema.
- Cada metodología tiene **diversos lenguajes** que las soportan.
 - Algunos lenguajes soportan varias metodologías.

Metodología	Lenguaje
Estructurada	Fortran, C, Pascal, Basic
Orientada a objetos (OOP)	C++, Java, Python
Orientada a eventos	VisualBasic

Programación Orientada a Objetos

Definición:

La **Programación Orientada a Objetos (OOP)** es un **método** de programación en el cual los programas se organizan en colecciones cooperativas de **objetos**, cada uno de los cuales representa una **instancia** de alguna **clase**, y cuyas clases son, todas ellas, miembros de una **jerarquía de clases** unidas mediante **relaciones de herencia**.

Comentarios:

- Usamos **objetos en lugar de algoritmos** como bloque fundamental
- Cada objeto es una **instancia** de una clase
- Las clases están relacionadas entre sí por relaciones tan complejas como la **herencia**

Ventajas de la POO

- Proximidad de los conceptos modelados respecto a objetos del **mundo real**
- Facilita la **reutilización** de código
 - Y por tanto el **mantenimiento** del mismo
- Se pueden usar **conceptos comunes** durante las fases de **análisis**, **diseño** e **implementación**
- Disipa las barreras entre el **qué** y el **cómo**

Desventajas de la POO

- Mayor **complejidad** a la hora de entender el flujo de datos
 - Pérdida de linealidad
- Requiere de un lenguaje de modelización de problemas más elaborado:
 - *Unified Modelling Language* (UML)
 - **Representaciones gráficas** más complicadas

Conceptos de la OOP

Conceptos básicos

- Objeto
- Clase

Características de la OOP

- Abstracción:
- Encapsulamiento:
- Modularidad:
- Jerarquía

Otros conceptos OOP

- Tipos
- Persistencia

Tipos de relaciones

- Asociación
- Herencia
- Agregación
- Instanciación

Representaciones gráficas

- Diagramas estáticos (de clases, de objetos...)
- Diagramas dinámicos (de interacción...)

Objeto y Clase

Un **objeto** es algo de lo que hablamos y que podemos manipular

- **Existen** en el mundo real (o en nuestro entendimiento del mismo)

Objeto:Clase
Atributo1=valor Atributo2=valor ...

Una **clase** describe los objetos del mismo tipo

- Todos los objetos son **instancias** de una clase
- Describe las **propiedades** y el **comportamiento** de un tipo de objetos

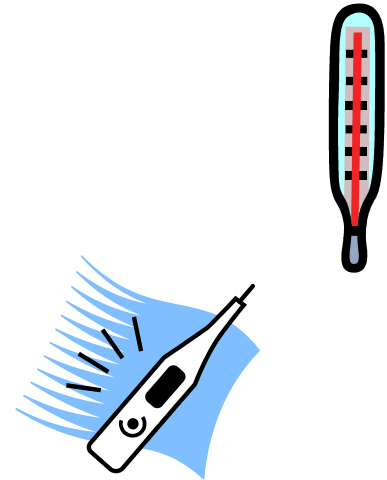
Clase
Atributos
Operaciones

Conceptos OOP: Abstracción

- Nos permite trabajar con la **complejidad del mundo real**
 - Resaltando los aspectos relevantes de los objetos de una clase
 - Ocultando los detalles particulares de cada objeto
- Separaremos el **comportamiento** de la **implementación**
- Es más importante **saber qué se hace** en lugar de **cómo se hace**:

Un sensor de temperatura

- Se define porque...
 - mide la temperatura
 - nos muestra su valor
 - se puede calibrar...
- No sabemos... (no nos importa)
 - cómo mide la temperatura
 - de qué está hecho
 - cómo se calibra

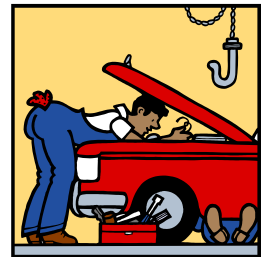


Conceptos OOP: Abstracción

- La abstracción **no es única**:

Un coche puede ser...

- Una cosa con ruedas, motor, volante y pedales (conductor)
- Algo capaz de transportar personas (taxista)
- Una caja que se mueve (simulador de tráfico)
- Conjunto de piezas (fabricante)



Conceptos OOP: Encapsulamiento

- Ninguna parte de un sistema complejo debe depender de los detalles internos de otra.
- Complementa a la abstracción
- Se consigue:
 - Separando la interfaz de su implementación
 - Ocultando la información interna de un objeto
 - Escondiendo la estructura e implementación de los métodos (algoritmos).
 - Exponiendo solo la forma de interactuar con el objeto

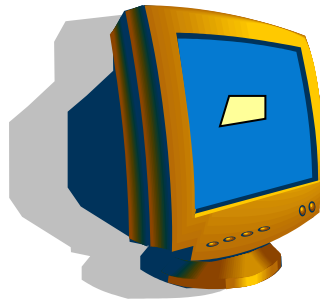
Conceptos OOP: Encapsulamiento

Ejemplo: Un paralelogramo



Vemos que se puede...

- Construir con:
 - 4 puntos (y restricciones)
 - 1 punto y 2 vectores
 - 1 punto, 1 vector, 1 ángulo y 1 lado
- Transformaciones:
 - Escalado
 - Rotación
 - Desplazamiento
- Dibujar



No vemos...

- Como está representado internamente
 - 4 puntos?
 - 1 punto y 2 vectores?
 - ...
- Como se modifica su escala
 - Guardando el factor?
 - Escalando en el momento?
- Idem para rotación, traslación, etc...

Conceptos OOP: Modularidad

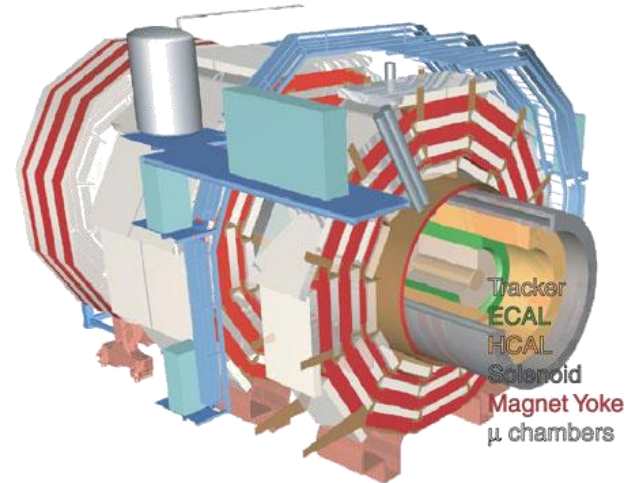
- Consiste en separar el sistema en bloques poco ligados entre sí: módulos.
 - Organización del código
- Es una especie de encapsulamiento de más alto nivel.
 - El C++ no lo impone aunque lo soporta (namespace)
 - El Java es más formal (packages)
- Difícil pero muy importante en sistemas grandes.
 - Suele aplicarse refinando el sistema en sucesivas iteraciones
 - Cada módulo debe definir una interfaz clara

Conceptos OOP: Modularidad

Ejemplo: Simulación detector de AAEE

Puede dividirse en los siguientes módulos...

1. **Geometría:** Describe el detector físicamente (forma, materiales, tamaño)
2. **Partículas:** Las partículas cuyas interacciones nos interesan
3. **Procesos:** Aquí enlazamos la información del detector (materia) con las propiedades de las partículas.
4. ...
 - Podríamos dividir el módulo de procesos en *procesos electromagnéticos*, *procesos hadrónicos*, ...
 - Lo mismo podríamos hacerlo con las partículas: *leptones*, *hadrones*, ...



Conceptos POO: Jerarquía

- Es una **clasificación** u ordenamiento de las abstracciones
- Hay dos jerarquías fundamentales:
 - **Estructura de clases:**
 - Jerarquía “*es un/a*”
 - Relaciones de **herencia**
 - **Estructura de objetos:**
 - Jerarquía “*parte de*”
 - Relaciones de **agregación**
 - Está implementada de manera genérica en la estructura de clases

Conceptos OOP: Jerarquía

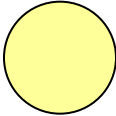
Ejemplo: Figuras planas y diagramas

- Una **figura plana** es:
 - Algo con una posición en el plano
 - Escalable
 - Rotable
- Un **gráfico** es algo que se puede dibujar en 2D
- Un **diagrama** es un conjunto de cuadrados y círculos

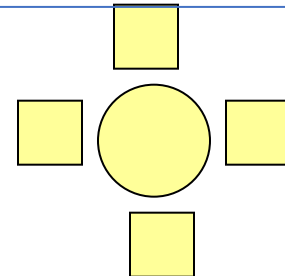
Herencia simple

- Un cuadrado *es una* figura
- Un círculo *es una* figura

Herencia múltiple

 *es una* figura
es un gráfico

Agregación



Conceptos OOP: Tipo

- Es el **reforzamiento** del concepto de clase
- Objetos de tipo diferente no pueden ser intercambiados
- El C++ y el Java son lenguajes fuertemente “tipeados”
- Ayuda a corregir errores en tiempo de compilación
 - Mejor que en tiempo de ejecución

Conceptos OOP: Persistencia

- Propiedad de un objeto de **trascender** en el tiempo y en el espacio a su creador (programa que lo generó)
- No se trata de **almacenar** sólo el estado de un objeto sino **toda la clase** (incluido su comportamiento)
- No está directamente soportado por el C++
 - Existen librerías y sistemas completos (OODBMS) que facilitan la tarea
 - Frameworks (entornos) como ROOT lo soportan parcialmente (reflex)
- El concepto de **serialización** del Java está directamente relacionado con la persistencia

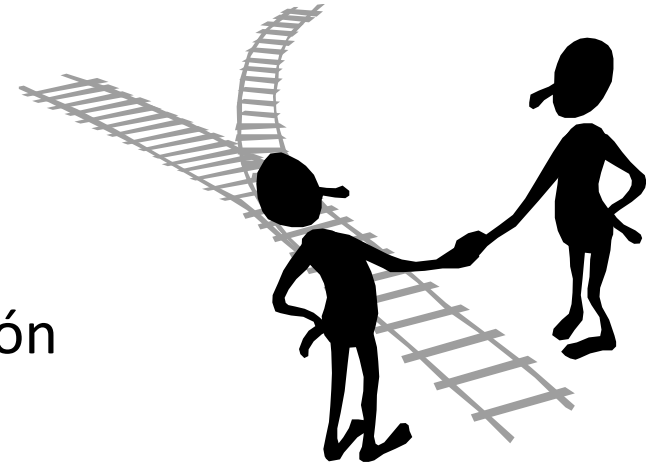
Relaciones

- Están presentes en cualquier sistema
- Definen como se producen los intercambios de información y datos
- También ayudan a comprender las propiedades de unas clases a partir de las propiedades de otras
- Existen 4 tipos de relaciones:
 - Asociación
 - Herencia
 - Agregación
 - Instanciación

Relación de Asociación

- Relación más **general**
- Denota una **dependencia semántica**
- Es **bidireccional**
- **Primer paso** para determinar una relación más compleja

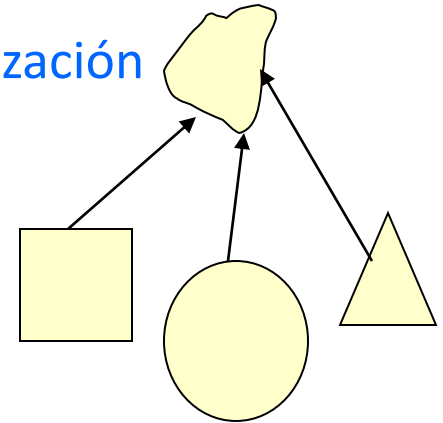
Ejemplo: Relación entre un producto y una venta. Cualquier venta está asociada a un producto, pero no es, ni forma parte de, ni posee ningún producto... al menos en una primera aproximación.



- **Cardinalidad:** multiplicidad a cada lado
 - Uno a uno: Venta-Transacción
 - Uno a muchos: Producto-Venta
 - Muchos a muchos: Comprador-Vendedor

Relación de Herencia

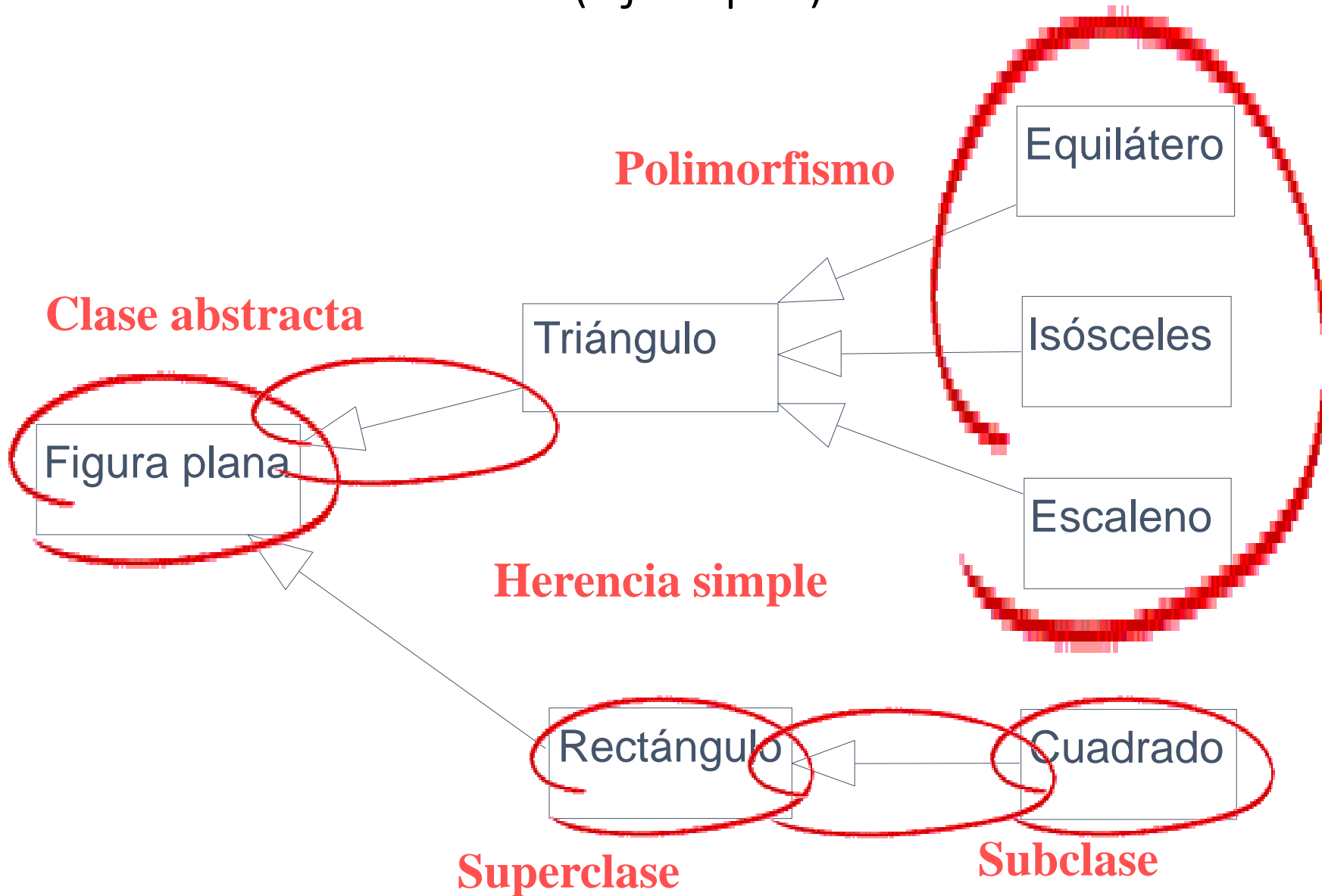
- ¡Relación **característica** de la OOP!
- Puede expresar tanto **especialización** como **generalización**
- Evita definir repetidas veces las **características comunes** a varias clases
- Una de las clases **comparte** la **estructura** y/o el **comportamiento** de otra(s) clase(s).
- También se denomina relación “*es un/a*” (*is a*)



Relación de Herencia (vocabulario)

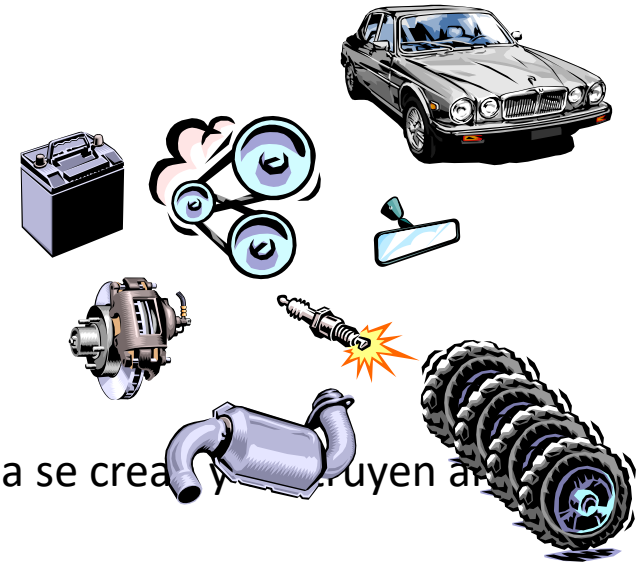
- **Clase base o superclase**: clase de la cual se hereda
- **Clase derivada o subclase**: clase que hereda
- **Herencia simple**: Hereda de una sola clase
- **Herencia múltiple**: Hereda de varias clases
 - Java solo la soporta parcialmente
 - Presenta diversos problemas (¿qué hacer cuando se hereda más de una vez de la misma clase?)
- **Clase abstracta**: La que no lleva, ni puede llevar, ningún objeto asociado
- **Polimorfismo**: Posibilidad de usar indistintamente todos los objetos de un clase y derivadas.

Relación de Herencia (ejemplo)



Relación de Agregación

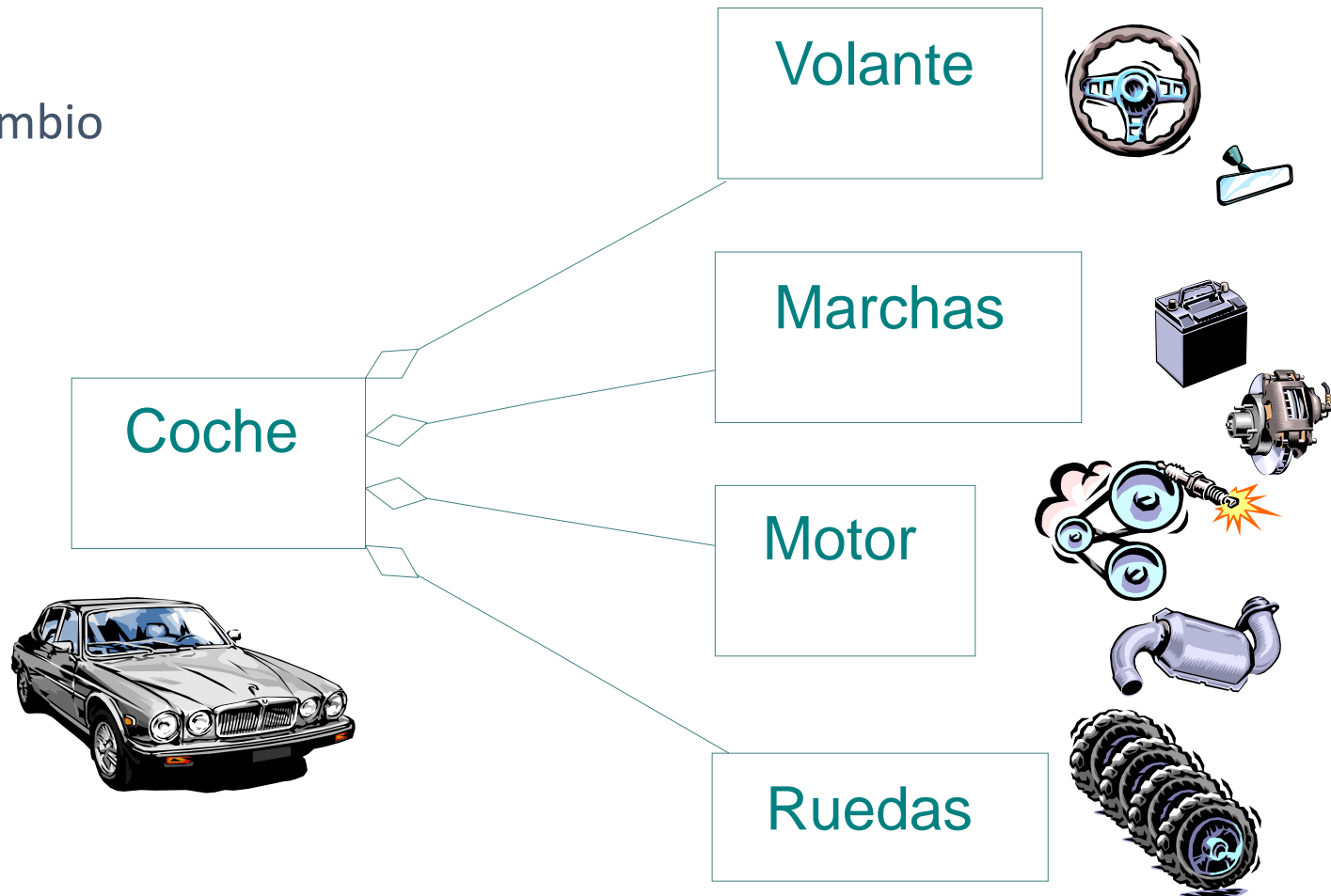
- Una **clase contiene** a otra **clase**
 - Ésta “es parte de” aquélla.
- También se denomina relación “*es parte de*” (has a)
- Una clase puede contener a otra:
 - **Por valor**: Cuando los objetos de la clase contenida se crean y destruyen al mismo tiempo que los de la clase continente
 - **Por referencia**: Cuando no necesariamente ocurre lo anterior



Relación de Agregación

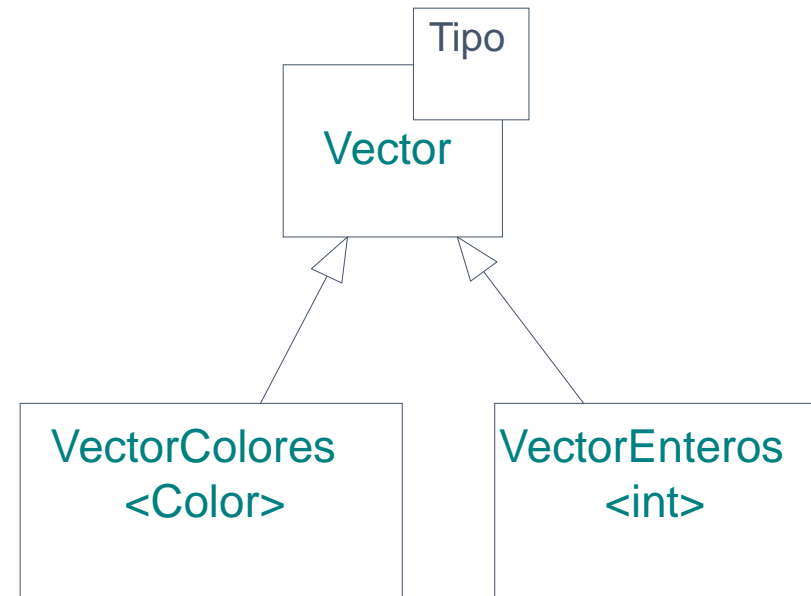
Un **coche** está hecho de

- Volante
- Palanca de cambio
- Motor
- Ruedas

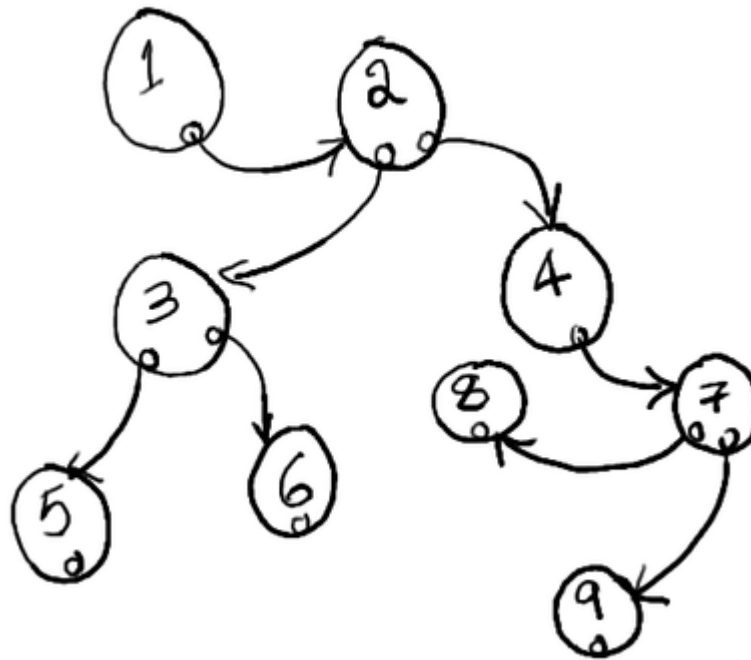


Relación de Instanciación

- En determinados casos una clase (p.ej. un vector) puede implementarse **independientemente del tipo** (real, complejo, color...) de alguno de sus atributos:
 - Definimos una **clase parametrizada** o *template* (plantilla)
 - Para cada uno de los tipos que necesitamos definimos una nueva clase \Rightarrow **Instanciación**



PROGRAMACION ORIENTADA A OBJETOS CON PYTHON

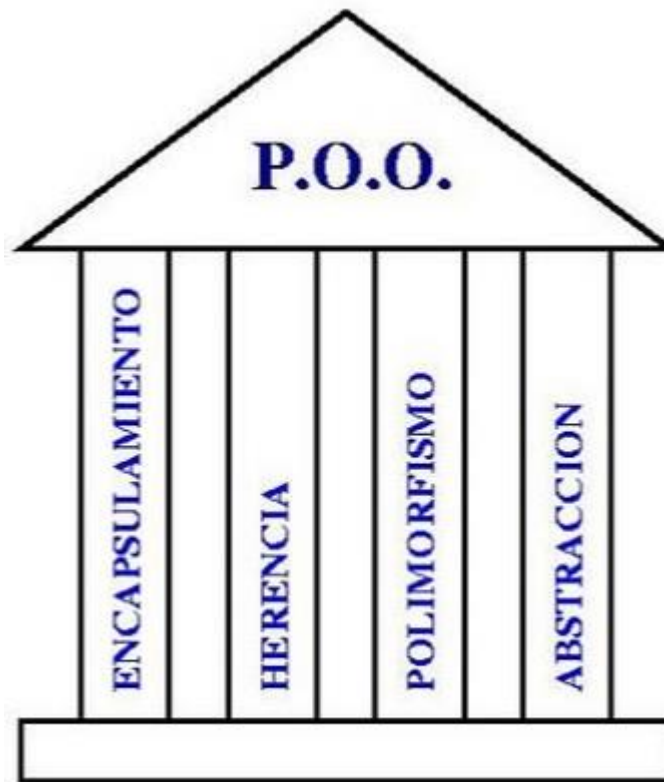


CONTENIDO

- Introducción
- Formas de Pensar
- Algunos Paradigmas Habituales
- Programación Multiparadigma
- Tipos de Datos
- Clases y Objetos
- Fisonomía de una Clase
- Un Paseo entre Objetos
- Acceso individualizado a Objetos: self
- Método de inicialización __init__
- Propiedades y Atributos
- Herencia y derivación de clases
- Jerarquías de Clases
- Encapsulación y Grados de Privacidad
- Resumen Final

INTRODUCCION

La programación orientada a objetos (POO, o OOP por sus siglas en inglés), uno de los paradigmas de programación estructurada más importante hoy en día.



FORMAS DE PENSAR

Los paradigmas de programación son herramientas conceptuales para analizar, representar y abordar los problemas, presentando sistematizaciones alternativas o complementarias para pasar del espacio de los problemas al de las implementaciones de una solución.

ALGUNOS PARADIGMAS HABITUALES

- La programación Modular
- La programación Procedural
- La programación Estructurada
- La programación Imperativa
- La programación Declarativa
- La programación Funcional
- La programación Orientada a Objetos

PROGRAMACION MULTIPARADIGMA

Los paradigmas de programación son idealizaciones, y, como tales, no siempre se presentan de forma totalmente 'pura', ni siempre resultan incompatibles entre sí. Cuando se entremezclan diversos paradigmas se produce lo que se conoce como programación multiparadigma.

El uso de distintos modelos, según se adapten mejor a las diversas partes de un problema o a nuestra forma de pensamiento, resulta también más natural y permite expresar de forma más clara y concisa nuestras ideas.

TIPOS DE DATOS

Una forma de almacenar y representar una fecha en un programa podría ser la siguiente:

d = 14

m = "Noviembre"

a = 2006

def dime_fecha(dia, mes, anho):

return "%i de %s de %i del calendario gregoriano" % (dia, mes, anho)

print dime_fecha(d, m, a)

para obtener la siguiente salida:

"14 de Noviembre de 2006"

En este ejemplo, para representar y manipular lo que conceptualmente entendemos como una fecha, se utilizan las variables *d*, *m* y *a* como almacenes de datos, y un procedimiento o función, de nombre ***dime_fecha***, para realizar la representación de la fecha almacenada.

CLASES Y OBJETOS

- Al enunciar algunos tipos de paradigmas hemos visto que la programación orientada a objetos define los programas en términos de “clases de objetos” que se comunican entre sí mediante el envío de mensajes.
- Las clases surgen de la generalización de los tipos de datos y permiten una representación más directa de los conceptos necesarios para la modelización de un problema permitiendo definir nuevos tipos al usuario.
- Las clases permiten agrupar en un nuevo tipo los datos y las funcionalidades asociadas a dichos datos, favoreciendo la separación entre los detalles de la implementación de las propiedades esenciales para su uso.
- Una clase define propiedades y comportamiento que se muestran en los entes llamados objetos (o instancias de una clase).

FISONOMIA DE UNA CLASE

En python, el esquema para la definición de una clase es el siguiente:

```
class NombreClase:
```

```
<instrucción_1>
```

```
...
```

```
<instrucción_n>
```

en donde **class** es una palabra reservada que indica la declaración de una clase, **NombreClase** una etiqueta que da nombre a la clase y, los dos puntos, que señalan el inicio del bloque de instrucciones de la clase.

UN PASEO ENTRE OBJETOS

Veremos algunas características de los objetos:

- Identidad. Los objetos se diferencian entre sí, de forma que dos objetos, creados a partir de la misma clase y con los mismos parámetros de inicialización, son entes distintos.
- Definen su comportamiento (y operar sobre sus datos) a través de métodos, equivalentes a funciones.
- Definen o reflejan su estado (datos) a través de propiedades o atributos, que pueden ser tipos concretos u otros objetos.

Hemos visto cómo usar atributos de clase para compartir datos entre todos los objetos de una misma clase, también nos interesa conocer cómo utilizar atributos comunes a una clase pero que puedan tomar valores distintos para cada uno de los objetos.

Retomamos ahora el misterioso parámetro ***self*** que vimos estaba presente como primer parámetro de todos los métodos de una clase. En su momento ya comentamos que ***self*** contiene una referencia al objeto que recibe la señal (el objeto cuyo método es llamado), por lo que podemos usar esa referencia para acceder al espacio de nombres del objeto, es decir, a sus atributos individuales.

EL METODO DE INICIALIZACION `__init__`

Hemos visto ya cómo definir y usar atributos de clase y de instancia. Pero en python no es necesaria la declaración de variables, por lo que cualquier atributo al que se realice una asignación en el código se convierte en un atributo de instancia:

```
>>> a = F()
```

```
>>> a.i = 6
```

```
>>> a.atr = 3
```

```
>>> print a.atr
```

```
3
```

```
>>> a.__dict__
```

```
{ 'i': 6, 'atr': 3 }
```

EL METODO DE INICIALIZACION `__init__`

Para poder inicializar los atributos de una instancia existe un método especial que permite actuar sobre la inicialización del objeto. Dicho método se denomina `__init__` (con dos guiones bajos al principio y al final) y admite cualquier número de parámetros, siendo el primero la referencia al objeto que es inicializado (**`self`**, por convención), que nos permite realizar las asignaciones a atributos de la instancia.

```
class Clase:
```

```
def __init__(self, x=2):
```

```
self.x = x
```

```
self.a = x**2
```

```
self.b = x**3
```

```
self.c = 999
```

```
def dime_datos(self):
```

```
return "Con x=%i obtenemos: a=%i, b=%i, c=%i" % (self.x, self.a, self.b, self.c)
```

```
a1 = Clase(2)
```

```
a1.dime_datos() # Salida Con x=2 obtenemos: a=4, b=8, c=999
```

```
a2 = Clase(3)
```

```
a2.dime_datos() # Salida Con x=3 obtenemos: a=4, b=27, c=999
```

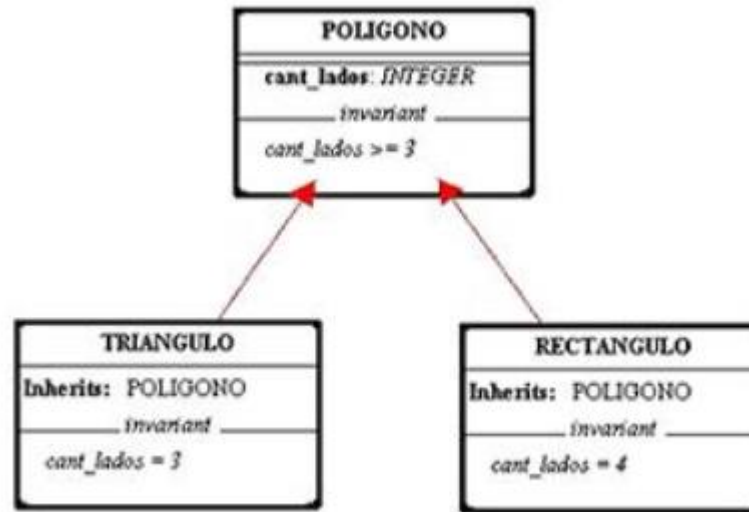
PROPIEDADES Y ATRIBUTOS

Aunque en la terminología general de la programación orientada a objetos atributo y propiedad se pueden utilizar como sinónimos, en python se particulariza el uso de propiedades para un tipo especial de atributos cuyo acceso se produce a través de llamadas a funciones.

```
class ClaseC(object):  
def __init__(self):  
self.__b = 0  
def __get_b(self):  
return self.__b  
def __set_b(self, valor):  
if valor > 10:  
self.__b = 0  
else:  
self.__b = valor  
b = property(__get_b, __set_b, 'Propiedad b')
```

PROPIEDADES Y ATRIBUTOS

```
c1 = ClaseC()
print c1.b
# b es 0
c1.b = 5
print c1.b
# b es 5
c2 = ClaseC()
print c2.b
# b es 0
c2.b = 12
print c2.b
# b es 0
```



En Python, para poder usar propiedades en una clase es necesario hacerla derivar de la clase **object** de ahí que aparezca al lado del nombre de la clase ClaseC.

La signatura de la función **property**, que define una propiedad de la clase es la siguiente:

nombre_propiedad = property(get_f, set_f, del_f, doc) donde **get_f** es la función llamada cuando se produce la lectura del atributo; **set_f** la función llamada cuando se produce una asignación al atributo; **del_f** la función llamada cuando se elimina el atributo, y **doc** es una cadena de documentación de la propiedad.

HERENCIA Y DERIVACION DE CLASES

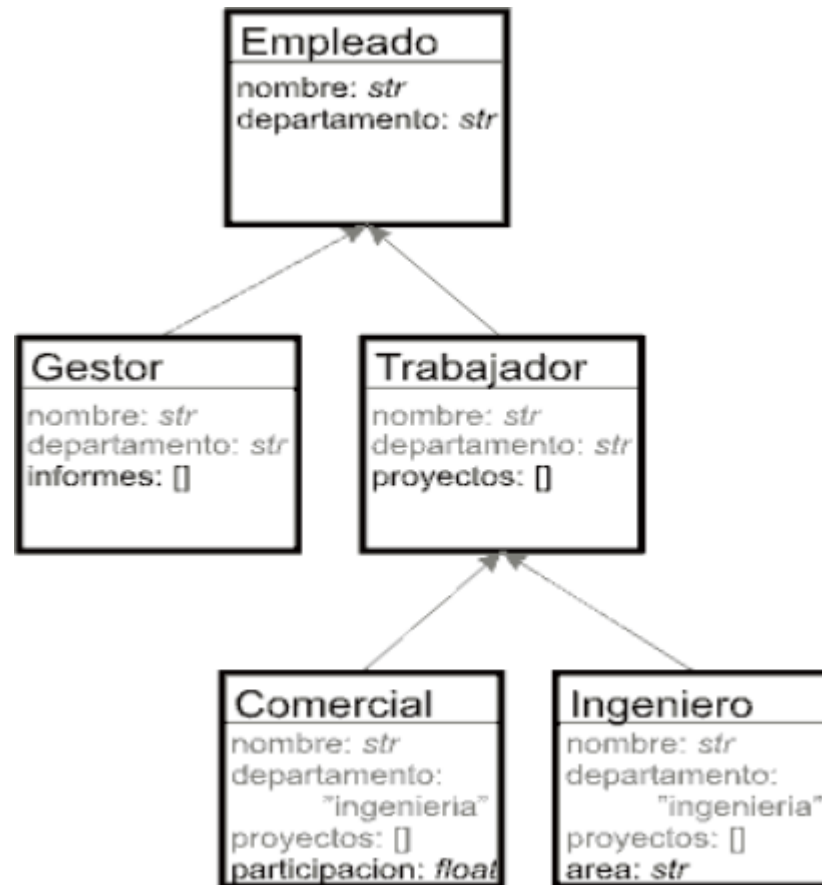
El uso de clases hace más adecuada la representación de conceptos en nuestros programas. Además, es posible generar una clase nueva a partir de otra, de la que recibe su comportamiento y estado (métodos y atributos), adaptándolos o ampliándolos según sea necesario.

Esto facilita la reutilización del código, puesto que se pueden implementar los comportamientos y datos básicos en una clase base y especializarlos en las clases derivadas.

En el lenguaje python, para expresar que una clase deriva, descende o es heredera de otra u otras clases se añade tras el nombre, en la declaración de la clase, una tupla con los nombres de las clases base.

JERARQUIAS DE CLASES

La herencia permite establecer relaciones entre clases, y, estas relaciones de pertenencia pueden ser a varios niveles, y ramificarse en lo que se denominan jerarquías de clases.



ENCAPSULACION Y GRADOS DE PRIVACIDAD

Uno de los principios que guían la POO, heredados de la programación modular, es el de encapsulación. Hemos visto cómo se puede agrupar comportamiento y datos gracias al uso de objetos, pero hasta el momento, tanto los atributos como los métodos que se definen en las clases correspondientes se convierten en métodos y atributos visibles para los usuarios de la clase (la API de la clase).

Python utiliza para ello convenciones a la hora de nombrar métodos y atributos, de forma que se señale su carácter privado, para su exclusión del espacio de nombres, o para indicar una función especial, normalmente asociada a funcionalidades estándar del lenguaje.

`_nombre`

Los nombres que comienzan con un único guión bajo indican de forma débil un uso interno. Además, estos nombres no se incorporan en el espacio de nombres de un módulo al importarlo con "***from ... import ****".

`__nombre`

Los nombres que empiezan por dos guiones bajos indican su uso privado en la clase.

RESUMEN FINAL

La programación orientada a objetos enuncia la posibilidad de escribir un programa como un conjunto de clases de objetos capaces de almacenar su estado, y que interactúan entre sí a través del envío de mensajes.

Las técnicas más importantes utilizadas en la programación orientada a objetos son:

Abstracción: Los objetos pueden realizar tareas, interactuar con otros objetos, o modificar e informar sobre su estado sin necesidad de comunicar cómo se realizan dichas acciones.

Encapsulación (u ocultación de la información): los objetos impiden la modificación de su estado interno o la llamada a métodos internos por parte de otros objetos, y solamente se relacionan a través de una interfaz clara que define cómo se relacionan con otros objetos.

Polimorfismo: comportamientos distintos pueden estar asociados al mismo nombre

Herencia: los objetos se relacionan con otros estableciendo jerarquías, y es posible que unos objetos hereden las propiedades y métodos de otros objetos, extendiendo su comportamiento y/o especializándolo. Los objetos se agrupan así en clases que forman jerarquías.

Las clases definen el comportamiento y estado disponible que se concreta en los objetos.

Los objetos se caracterizan por:

- Tener identidad. Se diferencian entre sí.
- Definir su comportamiento a través de métodos.
- Definir o reflejar su estado a través de propiedades y atributos.