

LENGUAJES DE PROGRAMACION O.O



Profesor: Luis Hernando García

Email:

lugarcia@ingenierias.univalle.edu.co

**Desarrollo de Software Orientado a Objetos
Universidad del Valle
Cali Colombia**

CONTENIDO



- Fundamentos de la P.O.O
 - Encapsulado
 - Herencia
 - Polimorfismo
- Lenguajes basados en Clases
- Lenguajes basados en Objetos
- Referencias

FUNDAMENTOS DE LA P.O.O



Los lenguajes de P.O.O proporcionan mecanismos que ayudan a implementar el modelo Orientado a Objeto. Estos mecanismos se llaman:

- **Abstracción**
- **Encapsulado,**
- **Herencia y**
- **Polimorfismo.**

ABSTRACCIÓN



- La abstracción como idea fundamental

"La civilización avanza extendiendo la cantidad de operaciones importantes que puede hacer sin pensar en ellas " Alfred N. Whitehead (1911)

La complejidad de los problemas que podemos resolver está muy relacionada con la capacidad de abstracción que empleamos en su resolución.

Abstracción: visión simplificada de una realidad de la que sólo consideramos determinados aspectos esenciales.

ENCAPSULADO

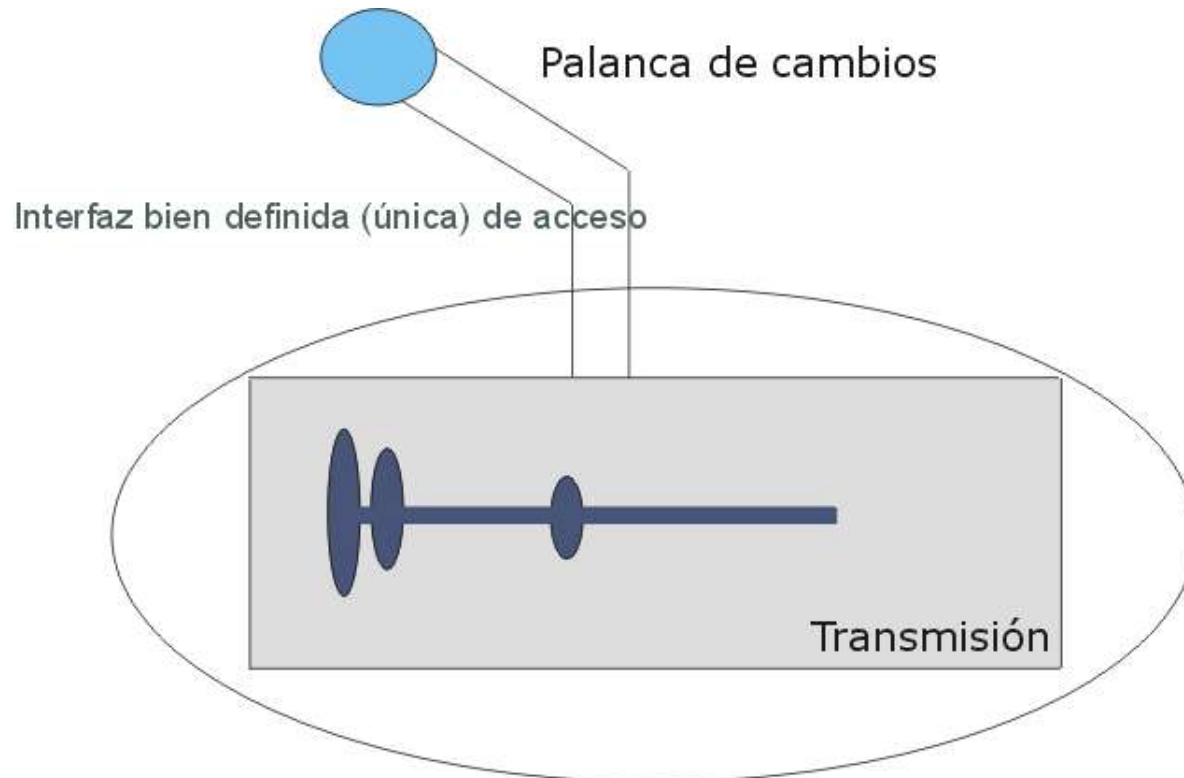


El encapsulado es el mecanismo que permite juntar el **código y los datos** que manipula, y que mantiene ambos alejados de posibles interferencias y usos indebidos.

- El acceso al código se realiza de forma controlada a través de una interfaz bien definida.
- Ejemplo en la realidad, la transmisión de un vehículo.

...ENCAPSULADO

- Ejemplo de la transmisión de un vehículo.



Implementación Encapsulada
Desde el punto de vista del conductor
no importa la implementación

...ENCAPSULADO



- Los L.P.O.O proveen regularmente un mecanismo para manejar el encapsulamiento a través de un concepto conocido como **niveles de visibilidad**. Algunos de ellos son:
 - **Público**, un elemento con este nivel de acceso es visible desde cualquier parte del programa, no existe ninguna restricción de visibilidad.
 - **Privado**, el elemento solo es accesible desde dentro de la clase donde se define

...ENCAPSULADO



- **Privado Protegido**, es accesible por la propia clase en que se define, y cualquiera de las clases derivadas, sin importar donde estén ubicadas
- **Amigable**, un atributo friendly es visible desde la clase en que se define y en cualquier otra clase del mismo paquete. Una subclase en otro paquete no tendrá acceso a dicho atributo.
- **Protegido**, solo es accesible por la clase en que se define, por sus subclases, estén donde estén, y por otras clases del mismo paquete (subsistema).



Siguiente: Herencia

HERENCIA



La herencia es el proceso mediante el cual un objeto adquiere las propiedades de otro. Importancia:

- Clasificación jerárquica
- Si no se usara la jerarquía cada objeto debería definir todas sus características.
- La herencia permite a un objeto ser una instancia específica de un caso más general.
- Ejemplos: *Animal*-Mamífero-Vaca, *Persona*-Estudiante-Egresado.

...HERENCIA



- Puede existir una clase "raíz" en la jerarquía de la cual heredan las demás directa o indirectamente.
- Incluye todas las características comunes a todas las clases

Eiffel: clase ANY

Smalltalk: clase Object

Java: clase Object

C++: no existe

...HERENCIA

Si B hereda de A entonces B incorpora la **estructura** (atributos) y **comportamiento** (métodos) de la clase A, pero puede incluir **adaptaciones**:

- B puede **añadir** nuevos **atributos**

- B puede **añadir** nuevos **métodos**

- B puede **REDEFINIR métodos**

• Refinamiento: Extender el uso original

• Reemplazo: Mejorar la implementación

- B puede **renombrar** atributos o métodos

- B puede implementar un método diferido en A

...

Adaptaciones dependientes del lenguaje

(Redefinición disponible en cualquier LOO)

El proceso de herencia es **transitivo**

A



B hereda de A

C hereda de B y A

B



B y C son descendientes (subclases) de A

B es un descendiente directo de A

C es un descendiente indirecto de A

C

TERMINOLOGÍA

B hereda de A

B es descendiente de A (Eiffel)

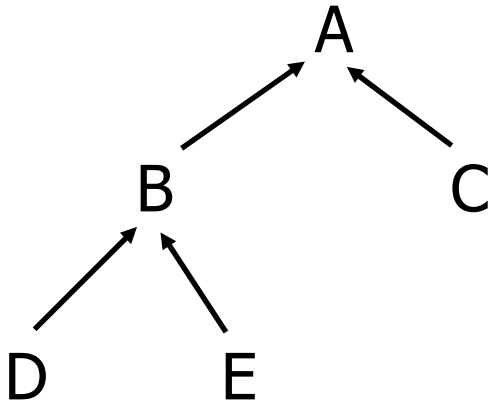
A es un ascendiente de B (Eiffel)

B es subclase de A (Smalltalk, Java)

A es superclase de B (Smalltalk, Java)

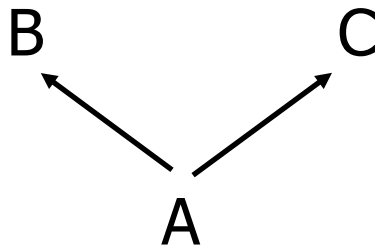
B es una clase derivada de A (C++)

Tipo de Herencia



- **Herencia simple**

- Una clase puede heredar de una única clase.
- Ejemplo: Smalltalk, Java



- **Herencia múltiple**

- Una clase puede heredar de varias clases.
- Clases forman un grafo dirigido aciclico
- Ejemplos: Eiffel, C++

¿Cómo detectar la herencia durante el diseño?

■ Generalización (Factorización)

Se detectan clases con un comportamiento común (p.e. *Libro* y *Revista*)

■ Especialización (Abstracción)

Se detecta que una clase es un caso especial de otra (p.e. *Rectangulo* de *Poligono*)

No hay receta mágica para crear buenas jerarquías

Ejemplo: Polígonos y Rectángulos

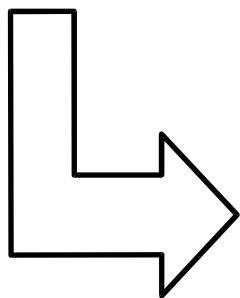
- Tenemos la clase **Poligono** y necesitamos representar rectángulos:

¿Debemos crear la clase *Rectangulo* partiendo de cero?

Podemos aprovechar la existencia de similitudes y particularidades entre ambas clases

Polígonos y Rectángulos

- Un rectángulo tiene muchas de las características de un polígono (rotar, trasladar, vértices,..)
- Pero tiene características especiales (diagonal) y propiedades especiales (4 lados, ángulos rectos)
- Algunas características de polígono pueden implementarse más eficientemente



```
class Rectangulo inherit
```

```
    Poligono
```

```
feature
```

```
    ...Características específicas para rectángulos
```

```
end
```



Siguiente: Polimorfismo

POLIMORFISMO



El polimorfismo (del griego **muchas formas**) es una característica que le permite a una interfaz ser utilizada por una clase general de acciones. La acción que se realiza depende del contexto.

- “Una interfaz, múltiples métodos”

...POLIMORFISMO

- **Es restringido por la herencia**
- Importante para escribir código genérico
- Sea las declaraciones:

ox: X; rutina1 (oy:Y)

- En un lenguaje con **monomorfismo** (Pascal, Ada, ..) en t.e. **ox** y **oy** denotarán valores de los tipos **X** e **Y**, respectivamente.
- En un lenguaje con **polimorfismo** (Eiffel, C++, ..) en t.e. **ox** y **oy** podrán estar asociados a objetos de varios tipos diferentes

TIPO DE POLIMORFISMO

■ Real

- Paramétrico, posibilidad de que una misma operación sintáctica trabaje uniformemente sobre un rango determinado de tipos.
- Inclusión:
 - | Basado en la herencia
 - | una función y varias interpretaciones diferentes

■ Aparente

- Sobrecarga
 - Varias funciones todas con el mismo nombre

...TIPO DE POLIMORFISMO



■ Ejemplo polimorfismo paramétrico:

```
long (l:lista): integer
  if l=nil then long:=1
  else long:= 1 + long(cola(l))
```

...TIPO DE POLIMORFISMO (sobrecarga)



- Es el nombre de la función lo que es polimórfico
- Se resuelve en tiempo de compilación, según la signatura de la rutina.
- No es necesario que exista similitud semántica.
- En los lenguajes OO puede existir sobrecarga
 - dentro de una clase
 - entre clases no relacionadas

...TIPO DE POLIMORFISMO (sobrecarga)

■ Ejemplo sobrecarga de un constructor:

```
class caja{
    double ancho;
    double alto;
    double prof;
    string pertenecen = "luish"
    caja (double w, double h, double d){
        ancho=w;    alto=h;    prof=d;
    }
    caja (double len){//Caso de un cubo
        ancho=alto=prof= len;
    }
}
```

...TIPO DE POLIMORFISMO (sobrecarga)

■ ...Ejemplo sobrecarga de un constructor:

```
calcularVolumen () {  
    return ancho * alto * prof;  
}  
} // END: class
```



Siguiente: Lenguajes basados en Clases

LENGUAJES BASADOS EN CLASES

- Clases y objetos:

- Clases como descripción de la estructura de objetos.

- Ejemplo:

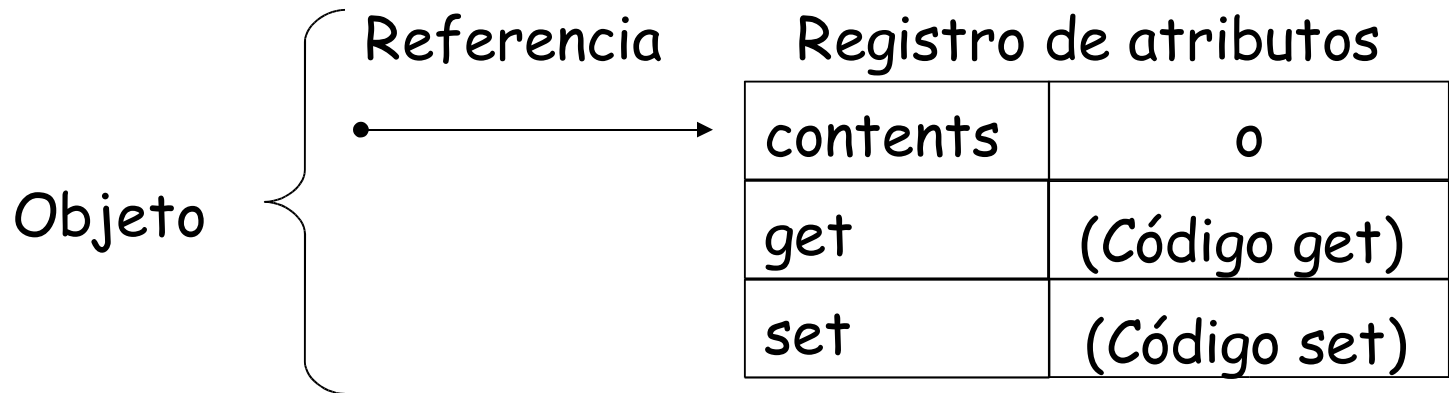
```
Class cell is
  var contents: Integer:=0;
  method get( ): Integer is ;
    return self.contents;
  end;
  method set (n:integer) is
    self.contents:=n;
  end;
end;
```

- **-self**: se refiere al objeto mismo, como el **this** en Java.

...LENGUAJES BASADOS EN CLASES

▪ ...Clases y objetos:

- `new c`: crea un objeto a partir de la clase `c`.



- Dos `new c`, crea dos objetos diferentes
⇒ referencias a diferentes registros de atributos.

Modelo de embebimiento (Embedding Model)

- `o=new c` es un "objeto de clase `c`" o una "instancia de la clase `c`".

...LENGUAJES BASADOS EN CLASES



▪ Subclases y Herencia

- ❑ Una subclase describe la estructura de un conjunto de objetos, pero relativa a una superclase:
 - Extensiones.
 - Cambios.
- ❑ Campos de una superclase, son replicados en la subclase. Nuevos campos pueden ser añadidos.
- ❑ Métodos de una superclase, son ó:
 - Replicados (por defecto).
 - Sobreescritos por métodos con el mismo nombre y misma signatura (tipos de los argumentos).

...LENGUAJES BASADOS EN CLASES

▪ ...Subclases y Herencia

□ Herencia: compartir atributos (campos y métodos) entre una clase y sus subclases.

➤ Los métodos que no se sobrescriben se heredan.

➤ Los valores iniciales de los atributos se heredan.

□ "c' hereda de c" es lo mismo que "c' es subclase de c"?

➤ Una subclase de una clase sin campos puede sobrescribir todos sus métodos y terminar "heredando nada".

➤ Existen mecanismos para compartir código que no están basados en la relación de subclases: procedimientos, por ejemplo.

...LENGUAJES BASADOS EN CLASES

- ...Subclases y Herencia

- ¿Y `self`?

- Sin subclases: se refiere a un objeto de la clase.
 - Con subclases: si `c'` es subclase de `c`, y un método de `c` hace referencia a `self`, en `c'`, `self` es visto como un objeto de `c'` y no de `c`.

⇒ `self` tiene acceso a los métodos redefinidos en `c'` y no a los originales de `c`.

...LENGUAJES BASADOS EN CLASES



- ...Subclases y Herencia

- Si se desea invocar un método como fue definido en `c`, se usa `super`. Sin embargo `self` sigue referenciando el objeto actual de la subclase.

- ⇒ Cualquier ocurrencia de `self` dentro de un método `m` es interpretada con respecto a la subclase actual, y no con respecto a ninguna superclase.

...LENGUAJES BASADOS EN CLASES

- Especialización de métodos:

- Hasta ahora se pueden reescribir métodos siempre y cuando los argumentos tengan el mismo tipo.

- Consideremos el ejemplo siguiente:

```
class c is
  method m(x:A):B is...end;
  method m1(x1:A1):B1 is...end;
end;
subclass c' of c is
  override m(x:A'):B' is...end;
end;
```



Siguiente: Lenguajes basados en Objetos

LENGUAJES BASADOS EN OBJETOS



▪ Introducción

- ❑ Relativamente nuevos, con desarrollos menos acelerados.
- ❑ Motivación: Lenguajes O.O. más simples y más flexibles que los tradicionales basados en clases.
- ❑ Gran flexibilidad con tipos: Un gran reto.
- ❑ Potencialidad del concepto de Objeto como base de la potencialidad del lenguaje.
- ❑ Pocos conceptos básicos (Objeto, despacho dinámico, tipos de objetos, subtipaje)
 - ⇒ Más fácil comprender el paradigma

...LENGUAJES BASADOS EN OBJETOS

▪ Objetos sin clases

□ No clases ⇒ nuevos mecanismos de construcción de objetos.

➤ Interfaces: **ObjectType**

➤ Implementación: **Object**

□ Ejemplo: **ObjectType Cell is**

```
var contents: Integer;
```

```
method get():Integer;
```

```
method set(n:Integer);
```

```
end;
```

Object cell:Cell is

```
var contents:Integer:=0;
```

```
method get():Integer is return self.contents end;
```

```
method set(n:Integer) is self.contents:=n end;
```

```
end;
```

...LENGUAJES BASADOS EN OBJETOS

▪ Prototipos y clones

□ Es difícil e inconveniente anticipar todas las formas en que los objetos pueden ser parametrizados.

□ Prototipaje: Concepto para resolver este problema.

➤ Creación de "objetos prototipo".

➤ Creación de nuevos objetos a partir de prototipos.

➤ Adecuación posterior de los nuevos objetos.

} Lenguajes
basados en
prototipos

...LENGUAJES BASADOS EN OBJETOS

▪ ... Prototipos y clones

□ Clonación: Mecanismo básico para implementar el concepto.

➤ Producir una copia de la estructura del objeto, compartiendo los valores de los atributos, pero con estado independiente.

➤ `var cellClone:Cell:= clone cellInstance;`

➤ Es como `new` pero opera sobre objetos y no sobre

□ ^{clases} Todo objeto puede ser a su vez un prototipo.

□ Un objeto, en el rol de prototipo:

➤ Actúa como una clase para los objetos clonados a partir de

➤ él. Actúa como una superclase para los prototipos clonados a partir de él.

...LENGUAJES BASADOS EN OBJETOS

▪ ...Prototipos y clones

□ La clonación necesita de un mecanismo complementario de "mutación" para ser realmente útil:

➤ Modificación de campos:

```
cellClone.contents:= 3
```

➤ Modificación de métodos:

```
cellClone.get:=method():Integer is
    if self.contents < 0
    then return 0
    else return self.contents
end;
end;
```

⇒ Modificación dinámica de métodos es una característica

distintiva de los LPO basados en objetos

...LENGUAJES BASADOS EN OBJETOS

▪ Herencia

- Clonación \cong Herencia, Method Update \cong Sobreescritura.
- ¿Cómo heredar el comportamiento de un objeto, en uno de diferente forma?
 - Extensión de objetos con nuevos atributos
 - Construcción de objetos nuevos con atributos de objetos existentes.

...LENGUAJES BASADOS EN OBJETOS

▪ ...Herencia

□ Dos aspectos ortogonales:

➤ Obtener los atributos del objeto donante:

- ♦ Implícitamente.
- ♦ Explícitamente.

➤ Incorporar estos atributos al objeto huésped:

- ♦ Por embebimiento.
- ♦ Por delegación.

4 categorías
de L.P.O.O.
basados en
prototipos.

...LENGUAJES BASADOS EN OBJETOS

- Herencia por embebimiento

- Ejemplo de herencia implícita:

```
Object cell:Cell is
  var contents:Integer:=0;
  method get():Integer is
    return self.contents
  end;
  method set (n:Integer) is
    self.contents:=n
  end;
end;
```

```
Object reCellImp:ReCell extends cell is
  var backup:Integer:=0;
  override set(n:Integer) is
    self.backup:=self.contents;
    embed cell.set(n);
  end;
  method restore() is
    self.contents:=self.backup
  end;
end;
```

...LENGUAJES BASADOS EN OBJETOS



- ...Herencia por embebimiento

- Herencia implícita:

- **extends** declara quien es el donante.

- El donante debe ser conocido estáticamente!

- **override** reemplaza un método del donante en el huesped.

- **embed** se sigue necesitando.

...LENGUAJES BASADOS EN OBJETOS

▪ Herencia por delegación

□ Herencia implícita:

```
Object cell:Cell is
  var contents:Integer:=0;
  method get():Integer is
    return self.contents
  end;
  method set (n:Integer) is
    self.contents:=n
  end;
end;
```

```
Object reCellImp':ReCell child of cell
  var backup:Integer:=0;
  override set(n:Integer) is
    self.backup:=self.contents;
    delegate cell.set(n);
  end;
  method restore() is
    self.contents:=self.backup
  end;
```

➤ **delegate** cell.set(n): Ejecutar el método set de cell con **self** ligado al objeto actual.

➤ ¿Diferencia con **embed**?

embed obtiene el método del padre al momento de la creación del objeto. **Delegate** lo obtiene al momento de la invocación.

REFERENCIAS



- Ver documentos y referencias en el Sitio Web del curso:
- <http://eisc.univalle.edu.co/materias/dsoo/>

