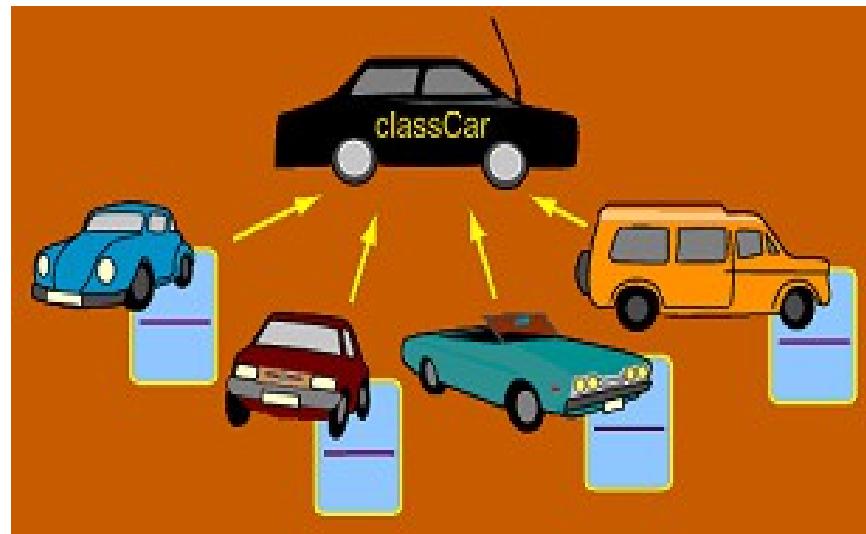


# Tema 3



## Reutilización y polimorfismo

# Objetivos generales



- Concienciarse de las posibilidades de reutilización que ofrece la OO y en especial la herencia.
- Conocer las propiedades principales de la herencia, como la de transitividad en la jerarquía de clases y la redefinición de métodos.
- Reconocer los distintos tipos de herencia que se pueden encontrar en los lenguajes de programación orientada a objetos.
- Conocer la sintaxis básica para definir estructuras de herencia en Java y en Ruby.
- Entender el uso de interfaces y clases genéricas como mecanismos complementarios de reutilización.
- Comprender el concepto de polimorfismo y sus posibilidades.
- Ser capaces de usar el polimorfismo para enriquecer programas en Java y en Ruby.
- Entender las nuevas posibilidades que las interfaces abren en relación a la simulación de la herencia múltiple y el polimorfismo, en Java y en Ruby.



# Contenidos

Lección	Título	Nº sesiones (horas)
3.1	Mecanismos de reutilización de código	3
3.2	Representación en UML de los mecanismos de reutilización	1
3.3	Polimorfismo	2

[http://groups.diigo.com/group/pdoo\\_ugr](http://groups.diigo.com/group/pdoo_ugr)



# Lección 3.1

## Mecanismos de reutilización de código

# Objetivos de aprendizaje



- Conocer los distintos mecanismos de reutilización de código de la POO.
- Comprender la herencia como una relación entre clases (es-un).
- Entender la diferencia entre herencia y composición.
- Identificar las posibilidades de la herencia múltiple, frente a la herencia simple.
- Entender la utilidad de las clases abstractas.
- Entender el concepto de interfaz y su utilidad.
- Comprender el concepto de clase parametrizada y su utilidad.

# Contenidos



1. Mecanismos de reutilización.
2. Definición y propiedades de la herencia.
3. Formas de establecer relaciones de herencia.
4. Herencia en las lenguajes de programación.
5. Herencia y ocultamiento de información.
6. Pseudovariable super.
7. Redefinición de métodos.

# Contenidos



8. Clase Abstracta.
9. Herencia múltiple.
10. Herencia vs composición.
11. El concepto de interfaz.
12. Simulando la herencia múltiple.
13. Clases parametrizables.

# 1. Mecanismos de reutilización

---

*Herencia*

*Interfaz*

*Clase parametrizable*

## 2. Definición y propiedades de la herencia

---

- Mecanismo que permite **derivar clases** (descendiente, subclase, clase hija o clase derivada) a partir de las clases existentes (ancestro, superclase, clase padre o clase base).
- Las clases padre e hija comparten un código común que se define en la clase padre y es heredado por la clase hija. Es decir, la clase hija **reutiliza** todo lo definido en la clase padre. (T.Budd. Pag. 162 y 163):
  - **Reutilización de código:** Cuando la clase hija hereda el comportamiento de la clase padre, sin modificar la forma de llevar a cabo ese comportamiento.
  - **Reutilización del concepto:** Cuando la clase hija hereda el comportamiento de la clase padre, pero modifica la forma de llevar a cabo ese comportamiento.

## 2. Definición y propiedades de la herencia

---

- La clase hija a la vez es (B.Meyer. Pag. 166-169):
  - Desde el punto de vista de la clase como un módulo, una **extensión** de la clase padre añadiendo atributos y/o métodos.
  - Desde el punto de vista de la clase como un tipo, una **especialización** o **restricción** de los individuos que forman parte de la clase hija, siendo éstos un subconjunto de la clase padre.
- La herencia, desde el punto de vista del número de ancestros, puede ser:
  - **Simple**: una clase sólo puede tener un ancestro.
  - **Múltiple**: una clase puede tener más de un ancestro.

## 2. Definición y propiedades de la herencia

- Test de **especialización**: “es un”.

Para establecer una relación de herencia entre la clase A (padre) y la clase B (hija), se debe cumplir que "Un B es un A". Es decir, todas las características aplicables a los individuos de la clase A, deben ser aplicables también a los de la clase B.

Ejemplos:

“Un reloj de pared es un reloj”

~~“Un coche es un motor”~~

- La herencia es **transitiva**:

Si (C hereda de B y B hereda de A)

entonces C hereda de A

- Una clase hereda de su superclase las variables y métodos de instancia.

### 3. Formas de establecer relaciones de herencia

---

(T. Budd pag. 170-174) (en rojo formas no recomendadas)

- **Especialización:** las clases hijas cambian algún comportamiento de la clase padre.
- **Especificación:** la clase padre define cuál debe ser el comportamiento de las clases hijas.
- **Construcción:** Se utiliza una clase para construir otra sin cumplir con la regla “un A es-un B”.
- **Extensión:** la clase hija no cambia el comportamiento de la clase padre, pero sí que lo amplía.

### 3. Formas de establecer relaciones de herencia

- **Limitación**: la clase hija restringe el comportamiento del padre, anulando alguno de sus métodos, mediante envío de mensajes de error.
- **Generalización**: varias clases comparten propiedades, se crea una nueva clase (padre de las anteriores), con los aspectos comunes entre ellas.
- **Combinación**: una clase hija hereda de varios padres (herencia múltiple).

¿Cuándo se está reutilizando código y cuándo se está reutilizando concepto en estas formas de herencia?



# 4. Herencia en los lenguajes de programación

Existen dos **tipos de lenguajes OO**:

- Con una **jerarquía de herencia única**: todas las clases heredan al menos de una superclase salvo la clase que actúa como raíz de la jerarquía (p.ej. en Java y Ruby se trata de la clase Object).
- Con posibilidad de **varias jerarquías de herencia**: puede haber varias clases sin superclase (p.ej. C++).

Definición de herencia en los lenguajes de programación:

Java:       **class** Clase\_hija **extends** Clase\_padre{ ... }

Ruby:       **class** Clase\_hija < Clase\_padre ...

Python:      **class** Clase\_hija (Clase\_padre1,...): ...

C++:        **class** Clase\_hija : public Clase\_padre1,... {...}

PHP:        **class** Clase\_hija **extends** Clase\_padre {...}

## 5. La herencia y el ocultamiento de información

---

En general los especificadores de acceso (`private`, `protected` y `public`) de los miembros definidos en una clase (variables y atributos) afectan a sus subclases de la siguiente forma:

- Los miembros definidos como `private` en una clase **no son accesibles** en las subclases.
- Los miembros definidos como `protected` en una clase **son accesibles** desde las subclases.
- Los miembros definidos como `public` en una clase **son accesibles** desde las subclases, pues lo son desde cualquier clase.

# 5. La herencia y el ocultamiento de información

## Java:

- Los miembros definidos como **package** en una clase **son accesibles** desde las clases y subclases que estén en el mismo paquete.
- Los miembros definidos como **protected** son accesibles desde **todas las subclases** estén donde estén y **desde las clases que estén en el mismo paquete**.

## Ruby:

- Las variables de instancia por defecto son accedidas desde cualquier subclase.
- Los métodos de instancia son públicos, su visibilidad puede cambiarse a private o protected en cualquier punto de la jerarquía de herencia y este cambio afecta a todas sus subclases.

```
class MiClase
  attr_accessor :mc
  def initialize
    @mc= "mi clase"
  end

  protected :mc
end
```

```
class MiSubclase < MiClase
  public :mc
end
```

```
miSubclase = MiSubclase.new
puts miSubclase.mc      # mi clase
```

```
miClase = MiClase.new
puts miClase.mc          # NoMethodError: protected method ..
```

## 5. La herencia y el ocultamiento de información: Java

```
public class Persona {  
  
    private static int numPersonas = 0;  
    protected String dni;  
    private String nombre;  
  
    public Persona(String d, String nom) {  
        this.setDni(d);  
        this.setNombre(nom);  
        NumPersonas +=1;}  
  
    static int getNumPersonas(){return numPersonas;}  
  
    protected String getNombre(){ return this.nombre;}  
    private String getDni(){ return this.dni; }  
    protected void setNombre(String nom) {this.nombre=nom; }  
    void setDni(String d){ this.dni=d; }  
    public void hablar(){ System.out.println("bla bla bla");}  
}
```



## 5. La herencia y el ocultamiento de información: Java

```
public class Profesor extends Persona{  
    String asignatura;  
    int experiencia;  
    public Profesor (String d, String nom, String asig, int exp)  
{  
        super(d,nom);  
        this.asignatura=asig;  
        this.experiencia=exp;  
    }  
}
```



Se ejecuta el constructor de la superclase

```
public class Alumno extends Persona {  
    string carrera;  
    int curso;  
    public Alumno(String d, String nom, String carr, int cur){  
        super(d,nom);  
        this.carrera=carr;  
        this.curso=cur;  
    }  
}
```



¿Cuál es el estado y la funcionalidad de los objetos de las clases Profesor y Alumno?

¿Qué atributos y métodos de Persona no son heredados por Profesor y Alumno?

## 5. La herencia y el ocultamiento de información: Java



```
Profesor profe= new Profesor("44444","Pedro", "PDOO",8);  
profe.nombre;  
Profesor.getNumPersonas();  
Persona.getNumPersonas();  
Persona.numPersonas;  
Profesor.numPersonas;  
profe.getNombre();  
profe.getDni();  
profe.setDni();
```

¿Dan error de compilación o ejecución las instrucciones anteriores?



¿Qué atributos y métodos de Persona no son accesibles desde Profesor y Alumno?

## 5. La herencia y el ocultamiento de información: Ruby

```
class Persona
  @@numPersonas = 0
  attr_accessor :nombre, :dni

  def initialize(d,nom)
    @dni = d
    @nombre= nom
    @@numPersonas += 1
  end

  def self.getNumPersonas
    @@numPersonas
  end

  def hablar
    puts 'bla bla bla'
  end

  protected :nombre
  private :dni
end
```



```
class Profesor < Persona
  def initialize(d,nom,asig,exp)
    super(d,nom)
    @asignatura = asig
    @experiencia = exp
  end
end
```

```
class Alumno < Persona
  def initialize(d,n,car,cur)
    super(d,n)
    @carrera = car
    @curso = cur
  end
end
```

Modificar la visibilidad de los métodos y resolver las mismas cuestiones que en Java



## 5. La herencia y el ocultamiento de información

---

- **Respecto al Ámbito:** Las **variables de clase** se **comparten** o **se heredan**, dependiendo del lenguaje.
  - En Java y Ruby, son variables **compartidas** desde la clase donde han sido definidas hacia abajo de toda la jerarquía de herencia.
  - En Smalltalk se **heredan**, de tal forma que una modificación de la variable de clase sólo se verá en la clase en la que es modificada y en sus subclases.

## 5. La herencia y el ocultamiento de información

```
public class Persona {  
    static String planeta="Tierra";  
    public static String getPlaneta(){return planeta;}  
}  
  
public class Astronauta extends Persona{  
    public static void setPlaneta(String otroPlaneta){  
        planeta=otroPlaneta;  
    }  
}  
  
System.out.println(Persona.getPlaneta()); // Tierra  
Astronauta.setPlaneta("Marte");  
System.out.println(Astronauta.getPlaneta()); // Marte  
System.out.println(Persona.getPlaneta()); // Marte
```

Variable de ámbito de clase,  
compartida por las subclases



# 5. La herencia y el ocultamiento de información

```
class Persona
    @@planeta= "Tierra"
    def self.getPlaneta
        return @@planeta
    end
end

class Astronauta <Persona
    def self.setPlaneta(otroPlaneta)
        @@planeta=otroPlaneta
    end
end

puts Persona.getPlaneta() # Tierra
Astronauta.setPlaneta("Marte")
puts Astronauta.getPlaneta() # Marte
puts Persona.getPlaneta() # Marte
```



Variable de  
ámbito de  
clase, también  
compartida por  
las subclases

## 6.Pseudovariable super

---

- En general la pseudovariable *super*, al igual que self o this, referencia al objeto receptor de un mensaje.
- Dentro de un método de una clase usamos la pseudovariable *super* para llamar a un método de la superclase y no el definido en la clase con ese nombre.

Ejemplo de uso de *super* en Java:

*super.borrar()* invoca al método *borrar()* implementado en la superclase y no al método *borrar()* implementado en la propia clase.

- La pseudovariable super está muy relacionada con la redefinición de métodos, es por lo que la veremos con más detalle en el siguiente apartado.

## 6. Pseudovariable super: Ruby

- Cuando se utiliza super en un método de una clase, sólo se puede invocar al método con el mismo nombre de la superclase, por eso no es necesario indicar el nombre del método.
- super puede ser usado, dentro de un método de una clase, de las tres formas siguientes:

**super** → Invoca el método de la superclase con los mismos argumentos con los que se llamó al de la clase.

**super()** → Invoca al método de la superclase sin ningún argumento.

**super(con\_argumentos)** → Invoca al método de la superclase con los argumentos que se indiquen.

# 7. Redefinición de métodos

---

- Una subclase **redefine** o **sobreescribe (override)** un método de una superclase cuando cambia su implementación, anulando el comportamiento heredado pero manteniendo la misma firma o cabecera.
- Un **método redefinido** en una clase **anula el heredado** de sus ancestros.
- Cuando se envía un mensaje a un objeto, se busca el método a ejecutar en la clase a la que pertenece. Si no se encuentra, se busca en la superclase, y así sucesivamente hasta que se encuentre en alguno de sus ancestros. Se **ejecuta la última redefinición** del método que se encuentre en esa rama de herencia.

# 7. Redefinición de métodos

- Una subclase **redefine** métodos heredados de la superclase para:
  - Cambiar un comportamiento.
  - Ocultar un comportamiento. (no es buena práctica)
  - Extender un comportamiento.
- **Redefinición Vs. Sobrecarga:** No se debe confundir la redefinición (*override*) con la sobrecarga (*overloading*) de métodos.
  - Una clase **sobrecarga** un método cuando tiene el mismo nombre que otro, pero tiene distintos argumentos (número y/o tipo).
  - En **Ruby no existe la sobrecarga**: un método sustituye a otro definido anteriormente con igual nombre, aunque tenga distintos argumentos y sea heredado de una superclase.

# 7. Redefinición de métodos: Java

- Desde la versión 5 de Java, se puede anotar con `@override` a los métodos que redefinen los de la superclase.

## **@override**

```
public String toString(){...}
```

Útil para localizar errores en el nombre del método: el compilador devuelve error si el método no redefine ningún método heredado.

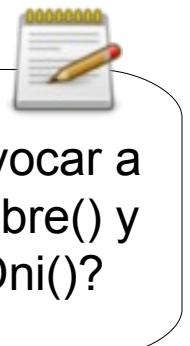
- La cabecera del método que se redefine puede cambiar:
  - Especificador de acceso: Mayor accesibilidad  
Ej. `protected` en clase padre → `public` en clase hija
  - Valor de retorno: Se permiten “tipos covariantes de retorno” según regla de compatibilidad de tipos en OO  
Ej. `Object` en clase padre → `String` en clase hija
- No se puede redefinir un método declarado como `final` (esto también ocurre en otros lenguajes).

# 7. Redefinición de métodos: Java

```
public class Profesor extends Persona{  
    //Todo lo definido anteriormente en la clase Profesor  
    // Nueva funcionalidad de Profesor  
    public void darClase(){  
        System.out.println("dando clase");  
    }  
    // Redefinición del método hablar  
    @Override  
    public void hablar(){  
        System.out.println( "Estimados");  
        super.hablar(); // Se invoca hablar() de Persona  
    }  
}
```



¿Se podría invocar a  
super.getNombre() y  
a super.getDni()?



**Cuál sería el resultado de ejecutar el siguiente código:**

```
Persona persona = new Persona("1111","p");  
Profesor profesor = new Profesor("2222","pro","asig1",5);  
persona.hablar();  
profesor.hablar();
```



# 7. Redefinición de métodos: Ruby



```
class Profesor < Persona  
  #Todo lo definido anteriormente en la clase Profesor  
  #Nueva funcionalidad de Profesor  
  def darClase  
    puts 'dando clase'  
  end  
  # Redefinición de hablar() de Persona  
  def hablar  
    puts 'estimados'  
    super  
  end  
end  
...
```

¿A qué método se invoca con super?



Cuál sería el resultado de ejecutar el siguiente código:

```
profesor = Profesor.new('2222','p','asig1',4)  
profesor.hablar  
persona = Persona.new('1111','a')  
persona.hablar
```



# 7. Redefinición de métodos: Ruby

```
..  
def hablar (*interlocutores)  
  if(interlocutores != nil)  
    puts interlocutores.to_s  
  end  
  puts 'bla bla bla'  
end  
..
```

En la clase *Persona* definimos *hablar* con argumentos



En *Profesor* redefinimos *hablar*

```
..  
def hablar(*interlocutores)  
  puts 'estimados'  
  super  
end  
..
```

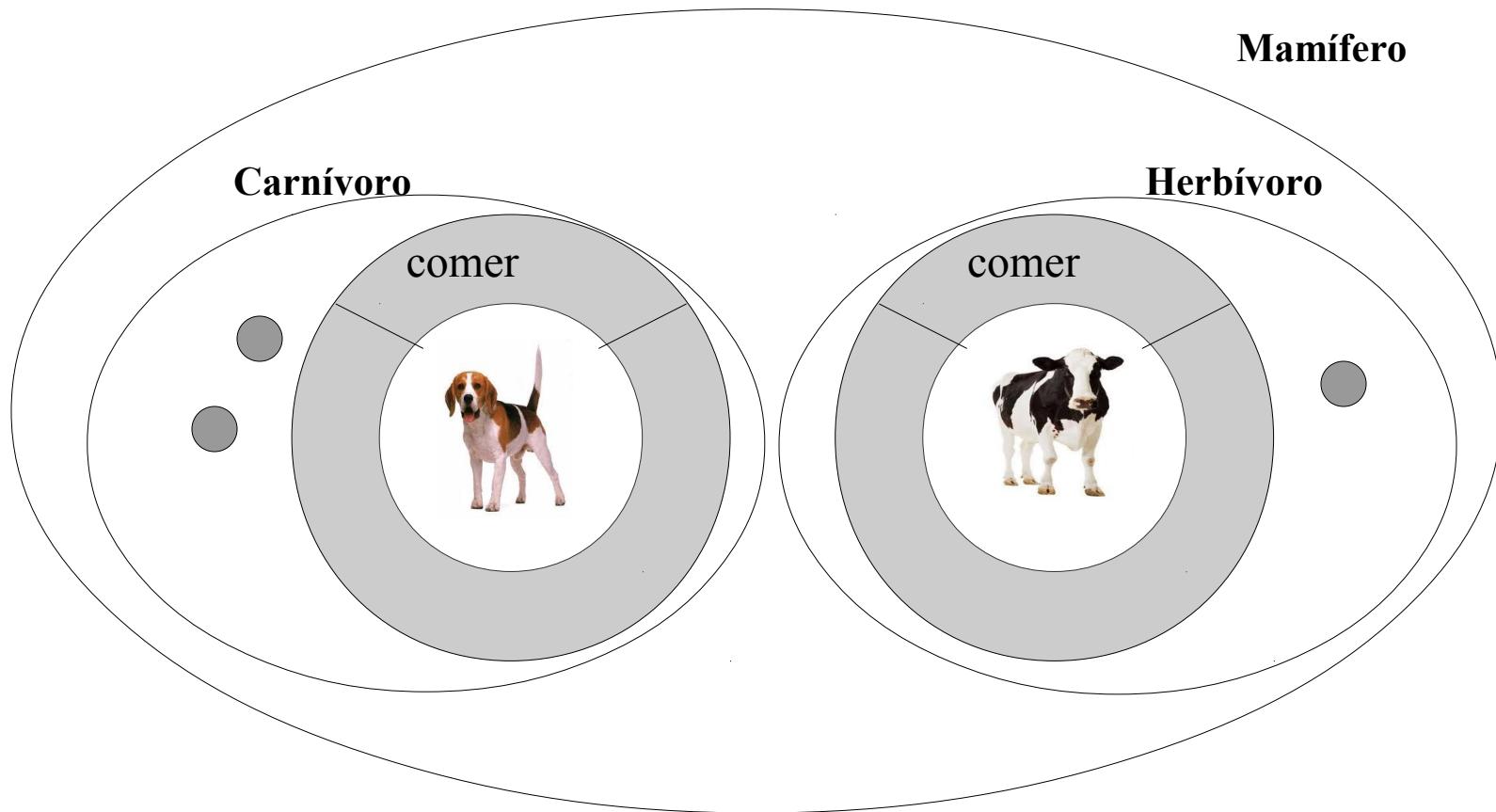
Cuál sería el resultado de ejecutar el siguiente código:

```
profesor = Profesor.new('2222','a','asig',5)  
profesor.hablar('ana','juan','pepe')
```



¿Qué ocurre cuando ejecutamos el trozo de código anterior, si en vez de *super* tenemos *super()* o *super('juan')*, en el método *hablar* de *Profesor*?

# 8. Clases abstractas



*Mamífero* es una clase abstracta, sin instancias, que tiene definida la funcionalidad *comer*, pero no se puede concretar cómo comen todos los mamíferos, se indica en cada una de sus subclases.

## 8. Clases abstractas

---

- Una clase es abstracta cuando **tiene definida su funcionalidad pero no la tiene completamente implementada**, debido a que no se conoce todo su estado o la forma de llevarla a cabo.
- **Ayudan** durante el **diseño**, cuando se sabe lo que pueden realizar los objetos de esa clase pero no cómo, es decir se tiene la especificación pero aún se desconoce cuál será la implementación.
- Por lo anterior se dice que **proporcionan módulos reutilizables** de alto nivel y capturan comportamientos comunes.

## 8. Clases abstractas

---

- Las clases abstractas **no pueden ser instanciadas**.
- Las clases abstractas **contienen métodos abstractos**: con nombre, tipo de retorno y lista de parámetros, pero sin implementar.
- Una clase abstracta **debe tener subclases**, las cuales implementan sus métodos abstractos. Esas subclases pueden encontrarse en cualquier nivel de la jerarquía de herencia que parta de la clase abstracta.
- Una clase, cuando no es abstracta, se dice que es **concreta**.

# 8. Clases abstractas: Java



- Una clase abstracta en Java es aquella que tiene al menos un método no implementado, aunque declarado

```
public abstract class Mamifero{  
    public abstract void comer();}  
  
public class Herbivoro extends Mamifero{  
    @override  
    public void comer() {System.out.println("como hierba");}  
  
public class Carnivoro extends Mamifero{  
    @override  
    public void comer(){System.out.println("como carne");}
```

- Si una clase que hereda de una clase abstracta y no implementa alguno de los métodos abstractos se convierte también en clase abstracta.
- Puede haber una relación de herencia entre clases abstractas.

# 8. Clases abstractas: Ruby

- Ruby no soporta las clases abstractas, aunque hay una forma de **simularlo**: declarando el método *new* como privado en la clase que queremos que se comporte como abstracta, y declararlo como público en sus subclases concretas.

```
class Empleado
  attr_accessor :nombre, :dni, :director
  def initialize(nom,dni)
    @nombre = nom
    @dni = dni
    @director = nil
  end
  private_class_method :new
end
```



```
class EmpleadoAsalariado < Empleado
  attr_accessor :salario
  def initialize(nom,dni,sal)
    super(nom,dni)
    @salario=sal
  end
  public_class_method :new
end
```

El siguiente trozo de código  
emp1 = Empleado.new('pepe','ppp')  
da el siguiente error:  
private method `new' called for ModeloEmpresa::Empleado:Class

# 8. Clases abstractas: Ruby

- Para simular los métodos abstractos:

```
class Empleado
# ...
def calcularNomina
    raise NotImplementedError.new
end
#...
end
```



El método puede redefinirse en la subclase

```
class EmpleadoAsalariado < Empleado
# ...
def calcularNomina
    @salario+250
end
#...
end
```

# 9. Herencia múltiple



Cama



SofáCama



Sofá

# 9. Herencia múltiple

---

- Se tienen **Herencia múltiple** cuando una **clase puede tener más de una superclase**.
- La subclase hereda todas las variables y métodos de sus superclases.
- Permite modelar problemas en los que un objeto tiene propiedades según criterios diferentes. Por ejemplo:
  - Vehículos clasificados según el permiso de conducir.
  - Vehículos clasificados según el medio (anfibios, terrestres y aéreos).
- Las estructuras de herencia múltiple son **más flexibles** pero más difíciles de manejar y de entender.
- Presenta **problemas de implementación**: conflicto de nombres y herencia repetida.
- C++ y Python tienen herencia múltiple, Java y Ruby no.

# 9. Herencia Múltiple: Conflicto de nombres

Si una clase B hereda desde más de una clase A1, A2, ..., An, se pueden dar conflictos de nombres en B si al menos dos de sus superclases definen miembros con el mismo nombre.

¿Cuáles son los atributos del objeto PracticoEmpresa?  
¿Dónde está el conflicto?



Trabajador

sueldo

emp='SA'  
nom='pp'  
of='pintor'

sueldo

?

PracticoEmpresa

estudiar  
Nom='Ana'  
carr='Física'  
curso=2

Estudiante

# 9. Herencia múltiple: Conflicto de nombres

## Soluciones

- Indicar que se hereda sólo uno de ellos y decir cuál. En lenguajes como Phyton se sigue el orden de la declaración.

*p. ej. : “se hereda nom de Estudiante, no de Trabajador”*
- Crear un atributo nuevo en la subclase, sin heredar ninguno (¿para qué heredar entonces?)

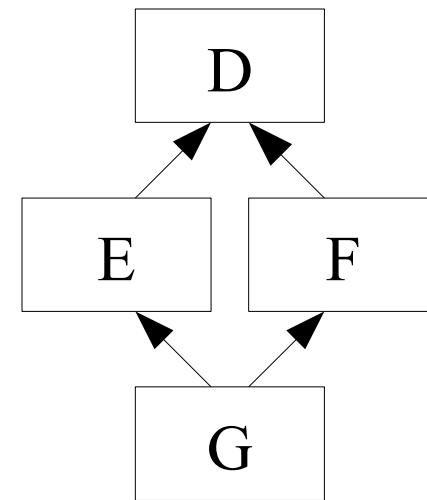
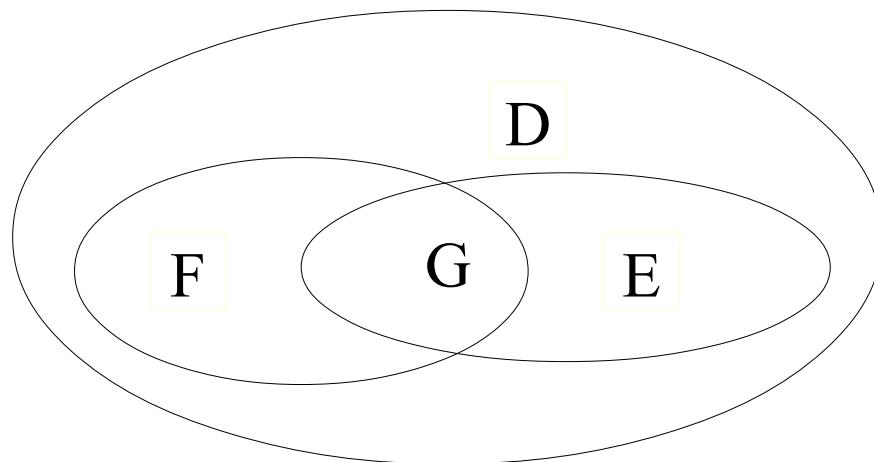
*p. ej. : “nombrePE es un atributo nuevo para PracticoEmpresa y así no se hereda nom”*
- Cambiar el nombre del atributo en alguna de las superclases: mala solución ya que las superclases pueden tener objetos o ser reutilizadas por otras clases.

*p. ej. : “se cambia nom de Trabajador por nomTrabaj”*

# 9. Herencia múltiple: Herencia repetida

## El problema del diamante

- Es una consecuencia de la herencia múltiple cuando **en el árbol de herencia hay un padre común**, ejemplo:



- G hereda de D de forma repetida: a través de F y a través de E, presentándose el conflicto de nombres.
- **Solución:** anular una de las dos herencias y usar composición/agregación en su lugar, o concretar por dónde se hereda D.

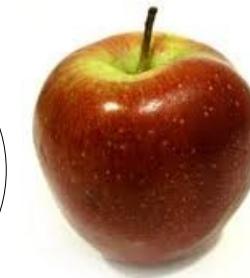
# 9. Herencia múltiple: mal uso



Tarta



TartadeManzana



Manzana

¿Por qué este ejemplo es incorrecto?  
¿Cómo debería haberse diseñado?



# 10. Herencia vs Composición



```
class Empleado {  
    private String Nombre;  
    private double sueldo;  
    public Empleado(String nom, double suel){..}  
    public calcularNomina(){  
        return (sueldo - calcularRetenciones())}  
}
```

Herencia

```
class Director extends Empleado{  
    private String despacho;  
    private double incentivos;//% que se incrementa su nómina  
    public Director(String nom, double s, String d, double i)  
    {    super(nom,s);  
        ...  
    }  
    public double calcularNomina(){  
        return super.calcularNomina() * incentivos;}  
}
```

Terminar de definir  
estas clases



# 10. Herencia vs Composición



```
class Director {  
    private Empleado emp;  
    private String despacho;  
    private double incentivos;// % que se incrementa su nómina  
    public Director(String n, double s, String d, double i)  
    {      emp = new Empleado (n,s);  
          despacho = d;  
          incentivos = i;  
    }  
    public String getNombre(){  
        return emp.getNombre();  
    }  
    public double calcularNomina(){  
        return emp.calcularNomina() * incentivos;  
    }  
}
```

Composición

¿Cuál es la solución más apropiada en este caso? ¿la herencia a la composición?



# 10. Herencia vs Composición



```
class Rectangulo {  
    private double ancho;  
    private double alto;  
    public Rectangulo(double an, double al){  
        ...  
    }  
    public doble superficie(){  
        return ancho*alto;  
    }  
}  
  
class Ventana extends Rectangulo{  
    // características de la ventana;  
}  
  
class Ventana{  
    private Rectangulo r;  
}
```

Otro ejemplo.  
Terminar de definir estas clases



¿Cuál es la solución más  
apropiada en  
este caso: la herencia o la  
composición?



# 10. Herencia vs Composición

---

## Características de la **herencia**:

- La herencia proporciona una reutilización de caja blanca, es decir los aspectos internos de la superclase son conocidos subclase, rompiendo el principio de encapsulación.
- Es estática y se define en compilación, no se permiten cambios de estas estructuras en ejecución.
- Facilita la incorporación de nuevos conceptos a una estructura ya desarrollada, proporcionando:
  - Alto grado de reutilización.
  - Facilidad para la extensión.

# 10. Herencia vs Composición

---

Características de la **composición**:

- Proporciona una reutilización de caja negra, es decir no se conocen los aspectos internos de los componentes, solo la interfaz de acceso a los métodos (cabecera con argumentos).
- Es dinámica, estas estructuras pueden ser cambiadas en ejecución.
- Tiene un grado de reutilización de código menor que la herencia.
- La composición también la podemos ver como una delegación de funciones (ver el ejemplo de la Ventana y el Rectangulo).

**Conclusión:** No son contrapuestas, hay que usarlas cuando el concepto que estemos modelando lo implique.

# 11. El concepto de interfaz

---

- Una **interfaz define** un determinado protocolo de **comportamiento sin implementarlo** (T. Budd p.88) y permite reutilizar la especificación de dicho comportamiento. Será una clase la que lo implemente.
- Al definir una interfaz se **define un tipo**, pudiéndose declarar variables de ese tipo. Dichas variables pueden referenciar a objetos que sean instancias de clases que implementan la interfaz.
- Una **clase** puede implementar más de una **interfaz** y una interfaz puede ser implementada por varias clases (**relación de realización entre interfaces y clases**).
- Entre las interfaces se pueden establecer jerarquías de herencia (**relación de herencia entre interfaces**).
- Una interfaz puede **heredar de una o varias interfaces**.

# 11. Concepto de interfaz: Java



```
[public] interface MilInterface {  
    // Conjunto de operaciones que proporciona  
    // el protocolo de esa interfaz, por ejemplo:  
    [public] [abstract] [void|Tipo] metodo (*[Tipo arg]);  
    ...  
    // en una interfaz también se pueden definir constantes  
    // de ámbito global, por ejemplo:  
    [public] [static] [final] Tipo NOMBRE_VARIABLE = valor;  
    ...  
}
```

# 11. Concepto de interfaz: Java



## Relación de realización entre interfaces y clases

```
[public] class MiClase implements MilInterface,... {  
    //Definición del estado de los objetos de la clase  
    //Implementación de los métodos definidos en MilInterface  
    [public] [void|Tipo] metodo (*[Tipo arg]) {...}  
  
    ...  
    //Implementación de los demás métodos de la clase  
  
    ...  
}
```

## Relación de herencia entre interfaces

```
[public] interface OtraInterface extends MilInterface,... {  
    // Definición del protocolo propio de esta interface  
}
```

La interfaz también puede redefinir métodos de la interfaz de la que hereda

# 11. El concepto de Interfaz: Java



Interfaces en Java para manejar las colecciones: `java.util.*`, p. ej.  
la interfaz List: <http://docs.oracle.com/javase/6/docs/api/java/util/List.html>

Si observamos la clase `ArrayList`:

- **Hereda de** `AbstractList` y ésta de `AbstractCollection`
- **Implementa** `List`

Consultar el protocolo definido en las interfaces `Collection`, `List`, `Set`, `SortedSet`, `Map`, `SortedMap` y sus relaciones de **herencia**.



Consultar las clases `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `LinkedHashSet`, `HashMap`, `TreeMap` y `LinkedHashMap` sus relaciones de **herencia** y sus relaciones de **realización** con las interfaces indicadas.

# 11. El concepto de Interfaz: Java



En Java 8 se ha introducido la posibilidad de implementar algunos métodos en las interfaces. Esto puede hacerse de dos formas que tienen dos propósitos diferentes:

- Métodos **default**: Código por defecto para un método.

```
public interface Saludo {  
    public void diHola();  
    public default void diAdios() {System.out.println("chao");}  
}
```

```
public class MiSaludo implements Saludo {  
    @Override  
    public void diHola() {System.out.println("hello");}  
}
```

/\*La clase no tiene que implementar el método diAdios() porque ya existe un código por defecto en la interfaz\*/

```
MiSaludo v = new MiSaludo();  
v.diHola(); //hello  
v.diAdios(); //chao
```

# 11. El concepto de Interfaz: Java



Métodos **default**: Código por defecto para un método.  
**Redefinición.**

```
public interface Saludo {  
    public void diHola();  
    public default void diAdios() {System.out.println("chao");}  
}  
  
public class MiSaludo implements Saludo {  
    @Override  
    public void diHola() {System.out.println("hello");}  
    @Override  
    public void diAdios() {System.out.println("good bye");}  
}  
/*La clase puede redefinir el método diAdios()*/
```

```
MiSaludo v = new MiSaludo();  
v.diHola(); //hello  
v.diAdios(); //good bye
```

# 11. El concepto de Interfaz: Java



Métodos **default**: Código por defecto para un método.  
**Abstracto.**

```
public interface Saludo {  
    public void diHola();  
    public default void diAdios() {System.out.println("chao");}  
}  
  
public abstract class MiSaludo implements Saludo {  
    @Override  
    public void diHola() {System.out.println("hello");}  
    @Override  
    public abstract void diAdios();  
}  
  
/*La clase puede sustituir la implementación del método diAdios()  
haciéndolo abstracto*/
```

# 11. El concepto de Interfaz: Java



- Métodos **static**: Código estático en la interfaz.

```
public interface Saludo {  
    public static void diAdios() {System.out.println("chao");}  
    ... /*Los métodos estáticos pueden ser invocados dentro  
    de otros métodos static y default de la interfaz*/  
}  
  
/*Los métodos estáticos pueden ser también ejecutados  
directamente por la interfaz (similarmente a los métodos  
estáticos de las clases)*/
```

**Saludo.diAdios();** //Mostraría “chao” en la pantalla

# 11. El concepto de Interfaz: Java



Métodos **static**: Código estático en la interfaz.  
**No se puede redefinir**

```
public interface Saludo {  
    public static void diAdios() {System.out.println("chao");}  
    ...  
}  
public class MiSaludo implements Saludo {  
    @Override //hay que eliminarlo para que no genere un error  
    public void diAdios() {System.out.println("adios");}  
    ...  
}  
/*La redefinición de un método estatic en la clase daría error porque el  
método solo existe a nivel de la interfaz*/  
/*Si eliminamos la anotación @Override no da error, pero se trata de un  
nuevo método diAdios() en la clase*/
```

## 12. Simulando herencia múltiple: Java

---

Para **simular herencia múltiple** en Java se usan interfaces. Existen varias formas posibles:

- Una clase hereda de otra clase (que puede ser abstracta) e implementa una interfaz.
- Una clase implementa varias interfaces.
- Una clase implementa una interfaz que a su vez hereda de varias interfaces.

# 12. Simulando herencia múltiple: Java



```
public interface FiguraGrafica {  
    int grosorBorde=2; // variable global al paquete  
    public void pintarBorde(String color);  
    public void colorear(String color);  
}
```

```
public abstract class FiguraGeometrica{  
    private int numLados;  
    public FiguraGeometrica(int lados){  
        this.setNumLados(lados);  
    }  
    public abstract double perimetro();  
    public abstract double area();  
    public void setNumLados(int numL){numLados = numL;}  
    public int getNumLados(){return numLados;}  
}
```

# 12. Simulando herencia múltiple: Java



```
public class Rectangulo extends FiguraGeometrica  
                           implements FiguraGrafica{  
    private double ladoa, ladob;  
    public Rectangulo (double la, double lb){  
        super(4);  
        ladoa =la;  
        ladob =lb;  
    }  
    @Override  
    public double perimetro(){return ((ladoa*2)+(ladob*2));}  
    @Override  
    public double area(){return (ladoa*ladob);}  
    @Override  
    public void pintarBorde(String color){...}  
    @Override  
    public void colorear(String color){...}  
}
```

```
Rectangulo r= new Rectangulo(2, 3.5);  
r.pintarBorde("negro");  
r.colorear("verde");  
r.girar(90);  
r.pintarBorde("azul");  
double perimetro = r.perimetro();  
double area = r.area();
```

# 12. Simulando herencia múltiple: Java

- A partir de Java 8, cuando una clase implementa varias interfaces hay que tener en cuenta si existen métodos default con la misma cabecera. Hay varias soluciones.

```
public interface Saludo {  
    public default void diAdios() {  
        System.out.println("chao");  
    }  
}
```

```
public interface Despedida {  
    public default void diAdios() {  
        System.out.println("hasta pronto");  
    }  
}
```

Solución: sobreescibir

```
public class MiDespedida implements Despedida, Saludo{  
    @Override  
    public void diAdios() {  
        System.out.println("nos vemos");  
    }  
    /* La clase puede sobreescibir  
    el método para eliminar el problema de ambiguedad*/  
}
```

```
MiDespedida v = new MiDespedida();  
v.diAdios(); //nos vemos
```

# 12. Simulando herencia múltiple: Java

- Solución: elegir

```
public interface Saludo {  
    public default void diAdios() {  
        System.out.println("chao");  
    }  
}
```

```
public interface Despedida {  
    public default void diAdios() {  
        System.out.println("hasta pronto");  
    }  
}
```

```
public class MiDespedida implements Despedida, Saludo{
```

```
    @Override  
    public void diAdios() {  
        Despedida.super.diAdios();  
    }
```

```
/* La clase puede decir con cuál de las dos implementaciones se queda para  
eliminar el problema de ambigüedad*/  
}
```

```
MiDespedida v = new MiDespedida();  
v.diAdios(); //hasta pronto
```

# 12. Simulando herencia múltiple: Java

- Solución: abstracto

```
public interface Saludo {  
    public default void diAdios() {  
        System.out.println("chao");  
    }  
}
```

```
public interface Despedida {  
    public default void diAdios() {  
        System.out.println("hasta pronto");  
    }  
}
```

```
public abstract class MiDespedida implements Despedida, Saludo{  
    @Override  
    public abstract void diAdios();  
}  
/* La clase puede hacer el método abstracto y renunciar a ambas  
implementaciones*/
```

# 12. Simulando herencia múltiple: Java

- Cuando una interfaz hereda de dos interfaces que implementan dos métodos *default* con la misma cabecera existen las mismas alternativas:
  - La interfaz elige la implementación de una de las dos super-interfaces
  - La interfaz descarta ambas implementaciones:
    - Establece su propia implementación para el método
    - Hace abstracto el método

@Override

```
public abstract void diAdios();
```

O simplemente:

```
@Override
```

```
public void diAdios();
```

# 12. Simulando herencia múltiple: Java

- Cuando una clase hereda de otra clase e implementa una interfaz que presenta un método default con la misma cabecera que uno existente en la clase, vuelven a estar operativas las mismas soluciones:
  - La clase descarta ambas implementaciones:
    - Establece su propia implementación para el método
    - Hace abstracto el método
  - La clase elige la implementación de:
    - La interfaz @Override  
public void diAdios() {Interfaz.super.diAdios();}
    - Su superclase @Override  
public void diAdios() {super.diAdios();}



# 12. Simulando herencia múltiple: Ruby

**module** FiguraGrafica

```
$GROSORBORDE = 2 # constante global  
def pintarBorde(color)  
    raise NotImplementedError.new  
end  
def colorear(color)  
    raise NotImplementedError.new  
end  
end
```

Juega el papel de clase abstracta

**class** FiguraGeometrica

```
attr_accessor :numLados  
def initialize(lados)  
    @numLados=lados  
end  
private_class_method :new  
def perimetro  
    raise NotImplementedError.new  
end  
def area  
    raise NotImplementedError.new  
end  
end
```

Juega el papel de interfaz

# 12. Simulando herencia múltiple: Ruby

```
class Rectangulo < FiguraGeometrica
  include FiguraGrafica
  def initialize(la,lb)
    super(4)
    @ladoA, @ladoB = la,lb
  end
  public_class_method :new
  def perimetro
    @ladoA*2 + @ladoB*2
  end
  def area
    @ladoA*@ladoB
  end
  def pintarBorde(color)
    puts "Color = #{color}, Grosor = #{$GROSORBORDE}"
  end
  def colorear(color)
    puts "Soy de color #{color}"
  end
end
```



# 13. Clases parametrizables

---

- Una clase parametrizable presenta un alto grado de reutilización pero con limitaciones. Se usa cuando alguno/s de los atributos de la clases no tiene definido el tipo, siendo éste un parámetro que tomará valor cuando usamos la clase.
- Su uso más frecuente es cuando sus atributos están formados por varios objetos del mismo tipo, por ejemplo, las colecciones:  
ArrayList<Persona>



# 13. Clases parametrizables

Ejemplo:



```
class Intervalo<T> {  
    T inicio, fin;  
    public Intervalo (T ini, T fin)  
        { inicio=ini; fin=fin; }  
    //... implementación de funcionalidad del intervalo  
}  
  
// a y b variable de tipo Point e inicializadas  
Intervalo<Point> segmento= new Intervalo(a,b);  
  
// h1 y h2 variables de tipo Time e inicializadas  
Intervalo<Time> reunion = new Intervalo(h1,h2);  
  
// d1 y d2 variables de tipo Date e inicializadas  
Intervalo<Date> semanaSanta = new Intervalo(d1,d2);
```