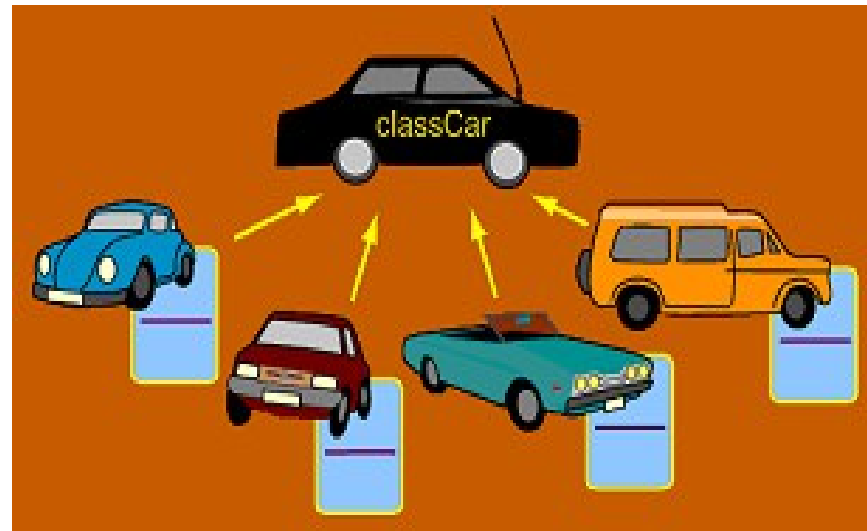


Tema 3



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	2
3.2	Representación en UML de los mecanismos de reutilización	1
3.3	Polimorfismo	2

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. Herencia en los lenguajes de programación.
4. Herencia y ocultamiento de información.
5. Pseudovariable super.
6. Redefinición de métodos.

Contenidos



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

1. Mecanismos de reutilización

Clase

Herencia

Composición

Interfaz

Clase abstracta

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 madre/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 solo 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"**.

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

3. Herencia en los lenguajes de programación

Según la herencia, 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 estructuras 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 {...}`

4. 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.

4. La herencia y el ocultamiento de información

<i>Variables y métodos declarados en una clase como:</i>	<i>Son visibles en:</i>			
	Java		Ruby	
	Subclase		El propio objeto en la subclase (self)	Cualquier objeto de la subclase
	En el mismo paquete	En distinto paquete		
private	-	-	✓	-
package	✓	-	-	-
protected	✓	✓	✓	✓
public	✓	✓	✓	✓

Recuerda que en Ruby los atributos siempre son privados y los métodos pueden ser private, protected o public

4. 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;}  
  
    Object hablar() { return "bla bla bla";}  
}
```

4. 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

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

```
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;
    }
}
```



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

4. La herencia y el ocultamiento de información: Ruby



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

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

  def self.get_num_personas
    @@num_personas
  end

  def hablar
    'bla bla bla'
  end

  protected :nombre, :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
```

Resuelve las mismas
cuestiones que en Java




4. La herencia y el ocultamiento de información: Java

Clase Profesor:

```
public String getNombreyDNI() {  
    String s1="mi nombre es "+ this.getNombre();  
    String s2="mi DNI es"+ this.getDni();  
    return s1+s2;  
}
```

//probar lo siguiente en la Clase Profesor y en otra clase que no herede de Persona

```
public static void main(String[] args) {  
    Profesor profe= new Profesor("44444","Pedro", "PDOO",8);  
    System.out.println(profe.nombre);  
    Profesor.getNumPersonas();  
    Persona.getNumPersonas();  
    System.out.println(Persona.numPersonas);  
    System.out.println(Profesor.numPersonas);  
    profe.getNombre();  
    profe.getDni();  
    profe.setDni("123");  
    profe.nombre="Luis";  
}
```



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



¿Dan error de compilación o ejecución estas instrucciones?

4. La herencia y el ocultamiento de información: Ruby

Clase Profesor:

```
def datos_personales_to_s
  s1="mi nombre es "+ self.nombre
  s2="mi DNI es"+ dni
  return s1+s2
end
```

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



probar el siguiente código en la Clase Profesor y en cualquier otra clase

```
profe= Profesor.new("44444","Pedro", "PDOO",8)
puts profe.nombre
profe.nombre="Luis"
Profesor.get_num_personas
Persona.get_num_personas
puts profe.dni
profe.dni="123"
```

¿Dan error de compilación o ejecución estas instrucciones?



4. La herencia y el ocultamiento de información

- Las variables de clase son variables **compartidas** desde la clase donde han sido definidas hacia abajo de toda la jerarquía de herencia.
- En Java, si una variable de clase se redefine en una clase asignando un nuevo valor, solo sus subclasses verán el nuevo valor.

4. La herencia y el ocultamiento de información



Variable de
ámbito de
clase,
compartida
por las
subclases

Redefinición
de la
variable de
clase

```
public class Persona {  
    static String planeta="Tierra";  
    public static String getPlaneta() { return planeta;}  
}  
public class Estudiante extends Persona {}  
public class Astronauta extends Persona {  
    static String planeta="Marte";  
    public static String getPlaneta() { return planeta;}  
}  
System.out.println(Persona.getPlaneta()); // Tierra  
System.out.println(Estudiante.getPlaneta()); // Tierra  
System.out.println(Astronauta.getPlaneta()); // Marte →  
Saldría Tierra si no se hubiese redefinido el get
```



4. La herencia y el ocultamiento de información



```
class Persona
  @@planeta= "Tierra"
  def self.get_planeta
    @@planeta
  end
end
class Estudiante < Persona
end
class Astronauta < Persona
  @@planeta="Marte"
end

puts Persona.get_planeta # Marte
puts Estudiante.get_planeta # Marte
puts Astronauta.get_planeta # Marte
```

Variable de
ámbito de
clase,
compartida por
todas las
clases y
subclases

Redefinición de
la variable de
clase



5. 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.

5. 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.

6. 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 signatura 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.

6. 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. En una clase no puede haber dos métodos con igual nombre.

6. Redefinición de métodos: Java

- En Java se anota con `@Override` a los métodos que redefinen a 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
 - **Tipo 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).

6. Redefinición de métodos: Java



```
public class Persona{  
    ...  
    Object hablar(){ return "bla bla bla";}  
}
```

```
public class Profesor extends Persona{  
    // Redefinición del método hablar  
    @override  
    public String hablar(){  
        return "Estimados" + super.hablar();  
        // Se invoca hablar() de Persona  
    }  
}
```

Visibilidad package y tipo devuelto Object que se redefine en Profesor como: public y String

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



¿Cuál sería el resultado de ejecutar el siguiente código?

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



6. Redefinición de métodos: Ruby



```
class Persona
  def hablar
    "bla bla bla"
  end
end
```

```
class Profesor < Persona
  # Redefinición del método hablar
  def hablar
    "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)
puts profesor.hablar
persona = Persona.new('1111','a')
puts persona.hablar
```



6. Redefinición de métodos: Ruby



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

```
class Profesor < Persona
...
def hablar(*interlocutores)
  puts 'estimados'
  super
end
...
end
```

Lista de
argumentos de
tamaño variable
que
internamente al
método se
gestiona como
un Array

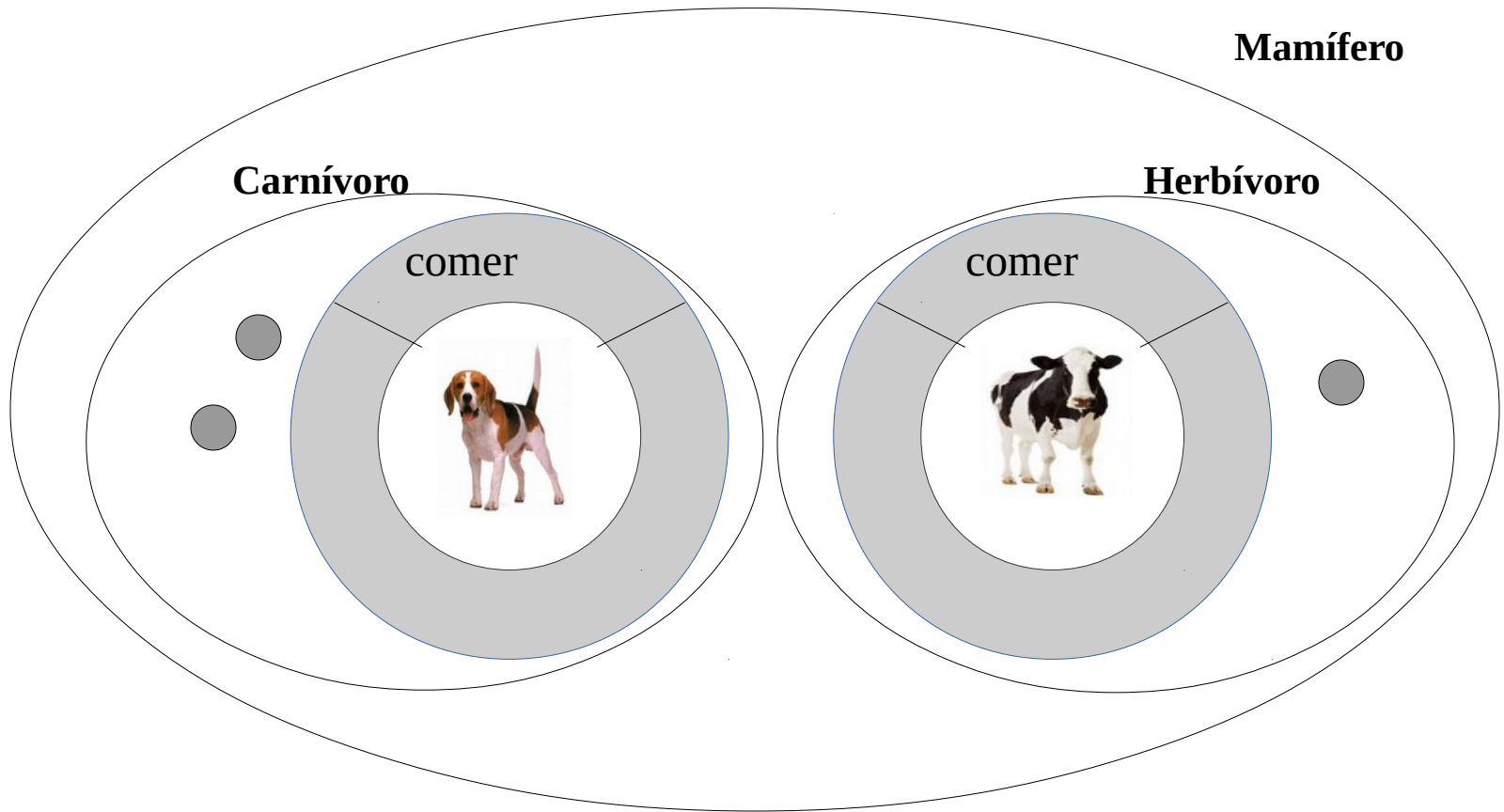
¿Cuál sería el resultado de ejecutar el siguiente código?:

```
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(interlocutores[1])* en el método *hablar* de *Profesor*?

7. 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.

7. 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.

7. 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**.
- **Ruby no soporta clases abstractas**.

7. Clases abstractas: Java



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

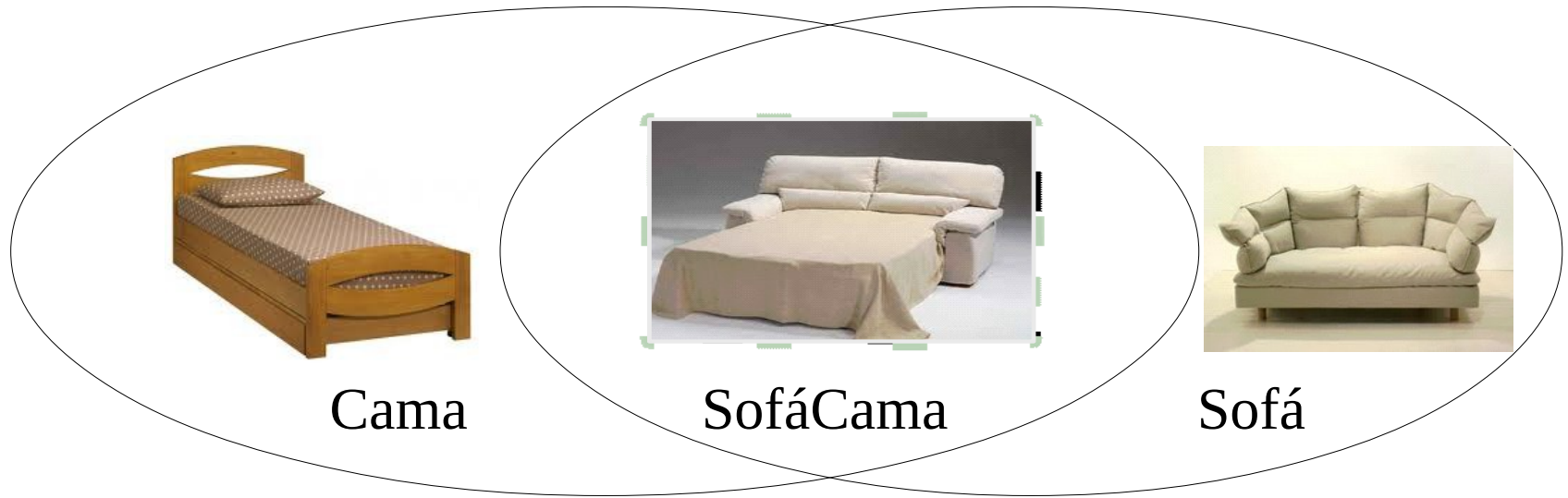
```
public abstract class Mamifero{  
    public abstract void comer();}
```

```
public class Vaca extends Mamifero{  
    @Override  
    public void comer() {System.out.println("como hierba");}}
```

```
public class Perro extends Mamifero{  
    @Override  
    public void comer() {System.out.println("como carne");}}
```

- Una clase que hereda de una clase abstracta y no implementa alguno de los métodos abstractos es también una clase abstracta.
- Puede haber una relación de herencia entre clases abstractas.

8. Herencia múltiple



9. Herencia múltiple

- Se tiene **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 (anfibiaos, 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.

8. Herencia múltiple. Ejemplo en C++



```

Class Estudiante {

public:
    Estudiante(string i,string c, int cu);
    String getId(){return id;}
    String getCarrera(){return carrera;}
    int getCurso(){return curso;}
    string estudiar() {...}
    float calculaSueldo() {...}

protected:
    string id;
    string carrera;
    int curso; };

Estudiante::Estudiante(string i, string c,int cu) {
    id=i; carrera=c; curso=cu;}
}
  
```

```

Class Trabajador {

public:
    Trabajador(string i, string e, string o);
    string getId(){return id+emp;}
    string getEmpresa(){return empresa;}
    string getOficio(){return oficio;}
    float calculaSueldo() {...}

protected:
    string id;
    string empresa;
    string oficio; };

Trabajador::Trabajador(string i, string e, string o) {
    id=i; empresa=e; oficio=o;}
}
  
```



```

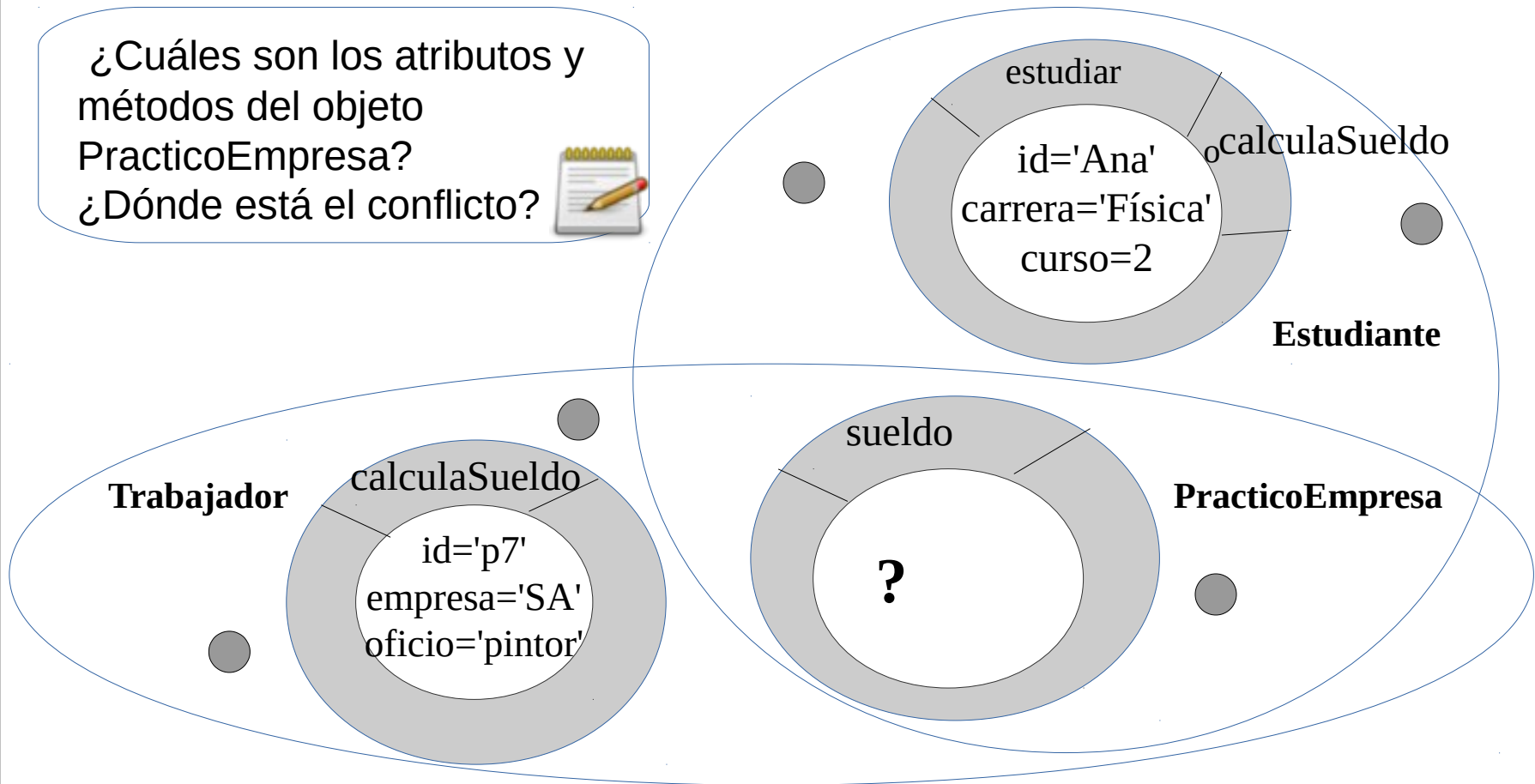
Class PracticoEmpresa:
public Estudiante,
public Trabajador {
protected:
    string tipo; }

PracticoEmpresa::PracticoEmpresa(string i, string c, int cu,string e, string o, string t) : Estudiante(i,c,cu), Trabajador(i,e,o)
{tipo=t;}
  
```

8. 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 y métodos del objeto PracticoEmpresa?
¿Dónde está el conflicto?



8. Herencia múltiple: Conflicto de nombres

Soluciones

- Indicar que se hereda sólo uno de ellos y decir cuál. En lenguajes como Python se sigue el orden de la declaración. En C++ se indica explícitamente.

p. ej. : “se hereda id de Estudiante, no de Trabajador”

- Crear un atributo nuevo en la subclase (¿para qué heredar entonces?)

p. ej. : “idPE es un atributo nuevo para PracticoEmpresa y así no usar id”

- 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 id de Trabajador por idTrabaj”

8. Herencia múltiple: Conflicto de nombres

Ejemplos en C++

```
main() {  
    PracticoEmpresa *pe= new PracticoEmpresa("Juan", "informatica", 2, "MWWSA", "programador");  
    cout pe->getId(); // conflicto de nombres  
    cout pe->Estudiante::getId();  
    // solución con invocación explícita  
}
```

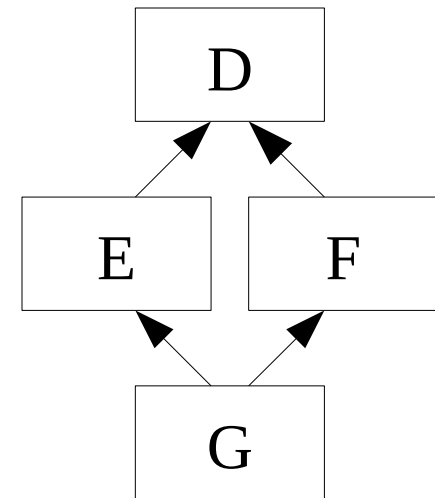
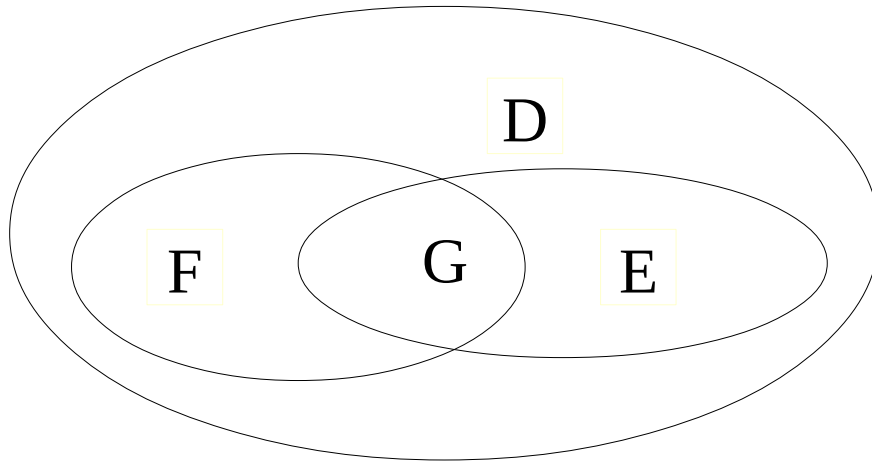
```
-----  
Class PracticoEmpresa: public Estudiante, public Trabajador {  
    public:  
        float getSueldo() { return Trabajador::getSueldo() }  
}
```

// solución en definición de método de subclase indicando a qué método de la superclase se invoca.

8. 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 (ver sección 10), o concretar por dónde se hereda D.

8. Herencia múltiple: mal uso



Tarta



TartadeManzana



Manzana

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



9. Herencia vs. Composición



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

Herencia



Terminar de definir
estas clases



```
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;  
}
```

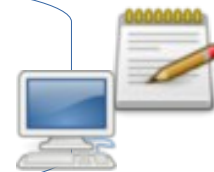
9. Herencia vs. Composición



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;}  
}
```

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

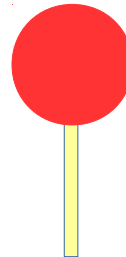


9. Herencia vs. Composición



```
class Circulo {  
    private double radio;  
    private String color;  
    public Circulo(double radio, String color){...}  
    public double superficie(){  
        return Math.PI*radio*radio;  
    }  
}  
  
// opción 1: herencia  
class Piruleta extends Circulo{  
    // características de la piruleta;  
}  
  
// opción 2: composición  
class Piruleta{  
    private Circulo r;  
    // otras características de la piruleta;  
}
```

Termina de definir estas clases



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



9. 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 por la 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.

9. 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 Piruleta y el Circulo).

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

10. El concepto de interfaz

- Una **interfaz define** un determinado protocolo de **comportamiento** (T. Budd p.88) y permite reutilizar la especificación de dicho comportamiento. Será una clase la que lo implemente.
- Ruby no provee sintaxis para declarar interfaces, pero Java sí.
- 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.

10. Concepto de interfaz: Java



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

10. 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

10. El concepto de Interfaz: Java



Pueden implementarse algunos métodos en las interfaces: métodos **default** y métodos **static**

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

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

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

Uso: MiSaludo v = new MiSaludo();
 v.diHola(); // hello
 v.diAdios(); // ciao

10. El concepto de Interfaz: Java



Ejemplo de **redefinición** de un método default

```
public interface Saludo {  
    public void diHola();  
    public default void diAdios(){ System.out.println("ciao");}  
}  
  
public class MiSaludo implements Saludo {  
    @Override  
    public void diHola(){ System.out.println("hello"); }  
    @Override  
    public void diAdios(){ System.out.println("good bye"); }  
}
```

La clase MiSaludo redefine el método diAdios() para cambiar su funcionalidad

Uso:

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

10. El concepto de Interfaz: Java



- Métodos **static**: Código estático en la interfaz, no se redefine.

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

/*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)*/

```
}
```

Uso:

```
public class MiSaludo implements Saludo{...}  
MiSaludo v = new MiSaludo();  
Saludo.diAdios(); // ciao  
MiSaludo.diAdios(); // ERROR
```

11. Simulando herencia múltiple: Java

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

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

11. Simulando herencia múltiple: Java



```
public interface FiguraGrafica {  
    public final int grosorBorde=2;  
    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 abstract void girar();  
    public void setNumLados(int numL){numLados = numL;}  
    public int getNumLados(){return numLados;}  
}
```



11. 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){...}}
```



¿Qué ocurre con girar()?
¿Cómo se soluciona?

Uso: Rectangulo r= new Rectangulo(2, 3.5);
r.pintarBorde("negro");
double perimetro = r.perimetro();
double area = r.area();

r.pintarBorde("azul");
r.colorear("verde");
r.girar(90);

11. Simulando herencia múltiple: Java

En Java se presenta **conflicto de nombres con los métodos default** cuando una clase implementa varias interfaces, o cuando una interfaz hereda de varias interfaces, con miembros del mismo nombre.

- Para evitarlo, hay varias soluciones haciendo uso de la redefinición.
 1. Proporcionar una nueva implementación
 2. Elegir una de las implementaciones
 3. Creando un método abstracto

11. Simulando herencia múltiple: Java



Solución 1: Redefinir con una nueva implementación

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

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

```
public class CBienEducado implements Despedida, Saludo{  
    @Override  
    public void diAdios() { System.out.println("nos vemos"); }  
}  
  
public interface IBienEducado extends Despedida, Saludo{  
    @Override  
    public void diAdios() { System.out.println("nos vemos"); }  
}
```

Uso: CBienEducado v = new CBienEducado();
v.diAdios(); //nos vemos

11. Simulando herencia múltiple: Java



Solución 2: Redefinir, eligiendo una de las implementaciones

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

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

```
public class CBienEducado implements Despedida, Saludo{  
    @Override  
    public void diAdios() { Despedida.super.diAdios(); }  
}  
public interface IBienEducado extends Despedida, Saludo{  
    @Override  
    public void diAdios() { Despedida.super.diAdios(); }  
}
```

Uso: CBienEducado v = new CBienEducado();
 v.diAdios(); //bye bye

11. Simulando herencia múltiple: Java

- **Solución 3:** Redefinir, creando un método abstracto.

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

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

```
public abstract class CBuenaEducacion implements Despedida, Saludo{  
    @Override  
    public abstract void diAdios();  
}
```

```
public interface IBuenaEducacion extends Despedida, Saludo{  
    @Override  
    public void diAdios();  
}
```

11. Simulando herencia múltiple: Ruby



Para **simular herencia múltiple** en Ruby hay una forma posible, llamada comúnmente “mixin”, que consiste en :

- Heredar de una clase e
- Incluir un módulo dentro de la clase

11. Simulando herencia múltiple: Ruby



```
module Cortadora
  $POTENCIA = 8
  def cortar ... end
end
```

```
class Tractor
  attr_accessor :marca
  def initialize(m)
    @marca=m
  end
  def moveuse ... end
end
```

```
require_relative "cortadora.rb"
class SierraElectrica
  include Cortadora
  def initialize(l)
    @longitud=l
  end
end
```

```
require_relative "cortadora.rb"
require_relative "tractor.rb"
class CortaCésped < Tractor
  include Cortadora
  def initialize(m,c)
    super(m)
    @colector=c
  end
  def recoger_césped
    ...
  end
end
```



Un CortaCésped es un tractor que contiene una cortadora.
Una SierraElectrica también incluye una cortadora.

Uso:

```
miCortador= CortaCésped.new(3333, "c3")
miCortador.recoger_césped
miCortador.moveuse
miCortador.cortar
```



```
miSierra= SierraElectrica.new(8)
miSierra.cortar
```

12. Clases parametrizables



- Una clase parametrizable presenta un **alto grado de reutilización** pero con limitaciones.
- **Encapsulan** operaciones válidas para diferentes tipos de datos, generalizando los tipos y sus operaciones.
- Su **uso más frecuente** es cuando sus atributos están formados por varios objetos del mismo tipo, siendo éste un parámetro que tomará valor cuando usamos la clase parametrizable. Por ejemplo, suelen usarse para dar valor al tipo de elementos de una colección: *ArrayList<Persona>* y para reutilizar las operaciones que permiten su manipulación: añadir, borrar, consultar, etc, ya que conceptualmente son iguales, aunque cambie el tipo de dato sobre el que se apliquen.
- En Ruby no existe este concepto.

12. Clases parametrizables

```
public class Tienda<T> {  
    String nombre;  
    ArrayList<T> stock;  
    float ganancias;  
  
    public Tienda(String n){  
        nombre=n;  
        stock= new ArrayList();  
        ganancias=0;  
    }  
    public void comprarAProveedor(T objeto, float precioCoste){  
        stock.add(objeto);  
        ganancias=ganancias-precioCoste;  
    }  
    public void venderACliente(T objeto, float pvp){  
        stock.remove(objeto);  
        ganancias=ganancias+pvp;  
    }  
    public float getGanancias(){  
        return ganancias;  
    }  
}
```



Ejemplo de
declaración de una
clase parametrizable.

12. Clases parametrizables



```
Tienda<Fruta> t1= new Tienda("Frutería Roja");  
Fruta melon= new Fruta();  
t1.comprarAProveedor(melon, 10);  
t1.venderACliente(melon, 12);  
System.out.println(t1.getGanancias());
```

```
Tienda<Bicicleta> t2= new Tienda("Ciclos");  
Bicicleta bici1= new Bicicleta();  
t2.comprarAProveedor(bici1, 1000);  
t2.venderACliente(melon, 5);  
t2.venderACliente(bici1, 1200);  
System.out.println(t2.getGanancias());
```

Ejemplos de uso de una clase parametrizable. Se pueden crear tiendas de objetos diferentes

Observa y prueba la clase parametrizable Tienda. Crea una nueva clase a partir de ella para gestionar intervalos de Edad.



13. Formas de establecer relaciones de herencia

(T. Budd pag. 170-174) (con* formas no recomendadas)

- **Especialización:** las clases hijas cambian algún comportamiento de la clase padre.
- **Especificación:** la clase padre declara, pero no implanta, cuál debe ser el comportamiento de las clases hijas, que serán las encargadas de implantarlo.
- ***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.

13. 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?

