
*UNIVERSIDAD POLITÉCNICA DE MADRID*
*ESCUELA UNIVERSITARIA DE INFORMÁTICA*
*LENGUAJES, PROYECTOS Y SISTEMAS INFORMÁTICOS*
*Luis Fernández Muñoz*
*setillo@eui.upm.es*

# Programación Orientada a Objetos en Java

# Tema 4. HERENCIA

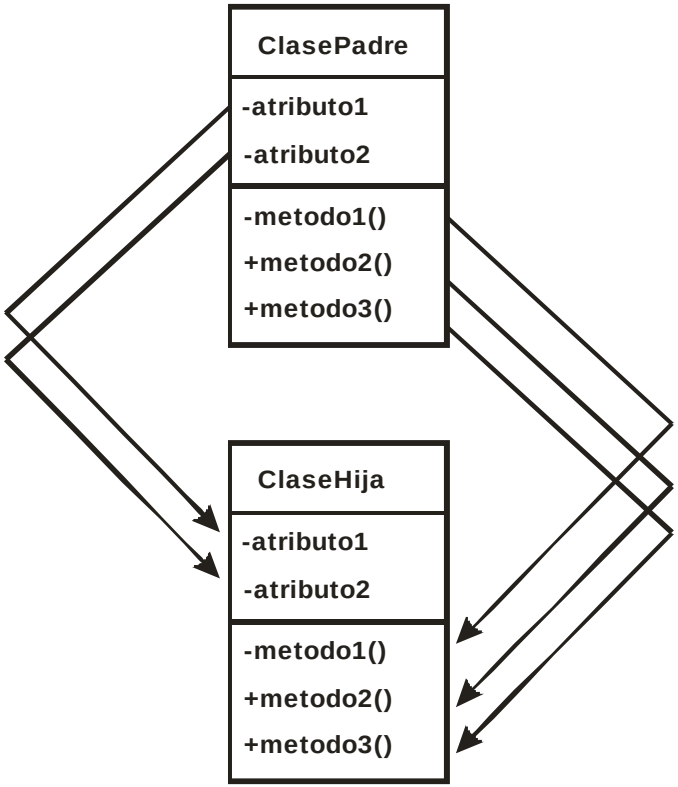
# ÍNDICE

# 1. RELACIÓN DE HERENCIA

**TRANSMISIÓN:**

• La herencia en todos los ámbitos (derecho, biología, …) tiene connotaciones de transmisión

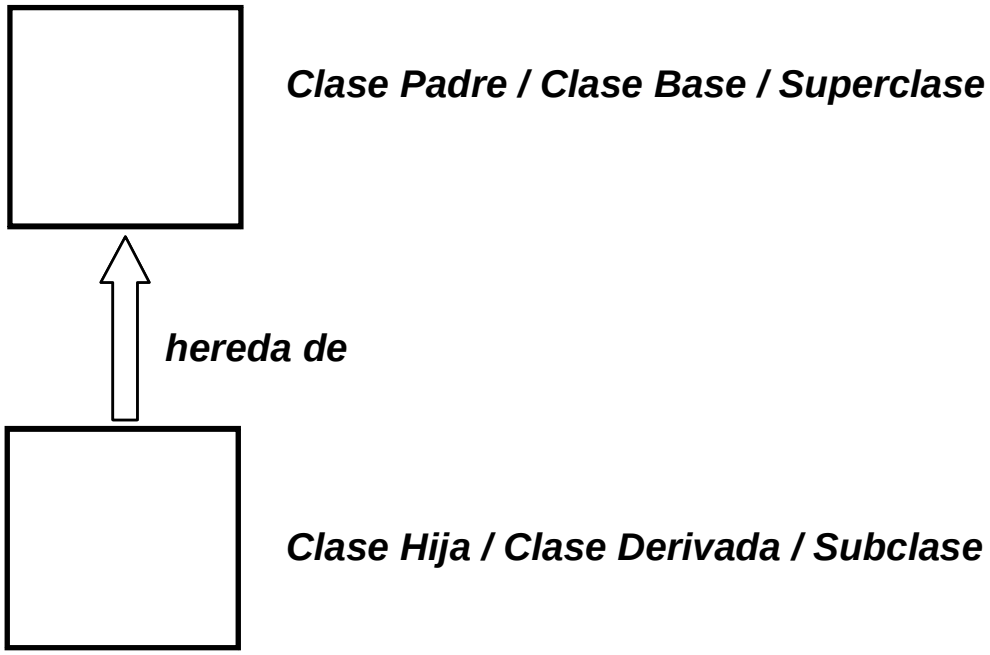
• En Programación Orientada a Objetos es la transmisión de la Vista Pública (métodos públicos) y de la Vista Privada (atributos, métodos privados y definición de los métodos) de una clase a otra.



# *1. RELACIÓN DE HERENCIA*

**Terminología:**

- terminología paralela a los árboles genealógicos (padre, hija, …);
- también clase base para la que transmite y clases derivadas a las que reciben la transmisión;
- superclase en desuso y subclases para las clases derivadas;



# *1. RELACIÓN DE HERENCIA*

**Colaboración entre objetos:**

• Las relaciones de **composición**, **asociación** y **dependencia** son relaciones binarias que devienen de la colaboración entre objetos.

• La relación de **herencia:**

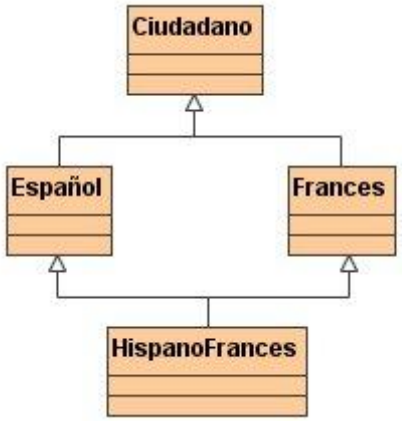
- es una relación binaria entre clases
- si existe una relación de herencia, no es necesario que exista una colaboración entre los objetos de sus clases aunque tampoco lo impide *(Ej.: la clase Persona hereda de la clase Animal; en una aplicación sobre la evolución de las especies, sus objetos no colaboran; en una aplicación para la gestión de una granja, sus objetos sí colaboran)*
- por tanto, los objetos de las clases de una relación de herencia son, a priori, independientes.

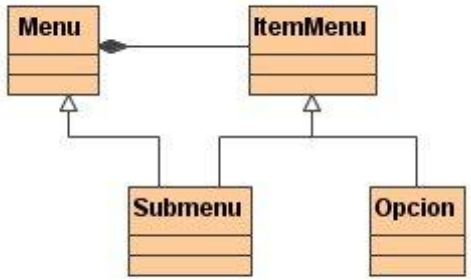
# 1. RELACIÓN DE HERENCIA

**Tipos de Relación de Herencia:**

- *Herencia simple:* cuando una clase derivada hereda de una única clase base.
- *Herencia múltiple:* cuando una clase derivada hereda de varias clases base.

*Ej. Un español es un ciudadano; un francés es un ciudadano; y un hispanofrancés es español y es francés.*
*Ej. Un menú tiene una lista de items; los items pueden ser opciones y submenús; y un submenú a su vez también es un menú.*

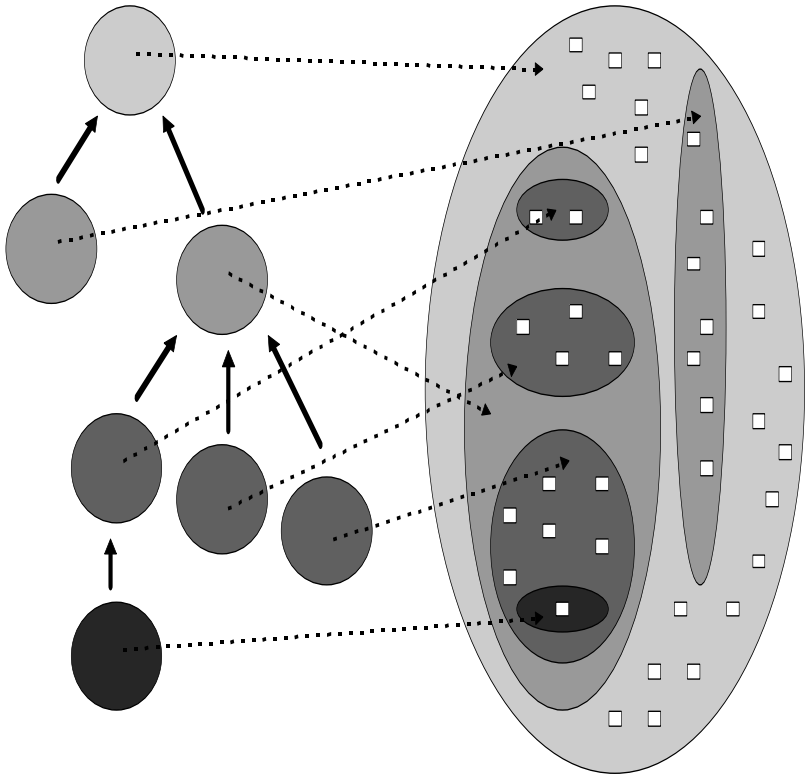


# 2. JERARQUÍAS DE CLASIFICACIÓN

**DEFINICIÓN:** una jerarquía por grado de clasificación es aquélla donde cada nodo (clases) de la jerarquía establece un dominio de elementos (conjuntos de objetos de la clase) incluido en el dominio de los nodos padre e incluye a los dominios de cada nodo hijo.

*Ej. Animal, Vertebrado, Invertebrado, … Persona, …*

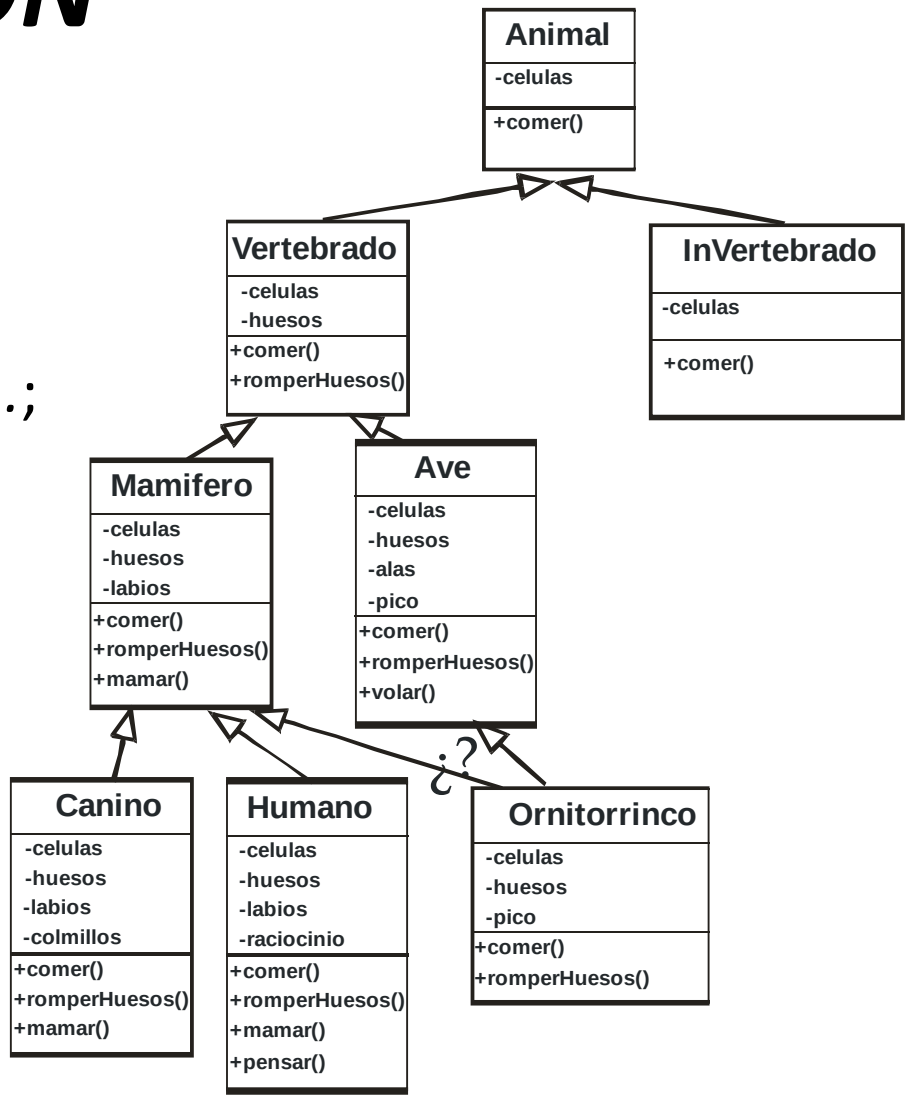


La relación de herencia permite establecer jerarquías por grado de clasificación

# 2. JERARQUÍAS DE CLASIFICACIÓN

**Características:**

- Eminentemente subjetivas. *Ej.: pacientes de un hospital: pública/privada, por especialidad, …*;
- Contemplan elementos que son difícilmente categorizables. *Ej. Ornitorrinco, pingüino, mula, …*
- Dificultad para establecer una clasificación “perfecta”;
- Esqueleto fundamental de un programa junto con la jerarquía de composición;



# 2. JERARQUÍAS DE CLASIFICACIÓN

**Reglas de Construcción:**

• *Regla de Generalización / Especialización:* cuando existen unas características específicas de un subconjunto de elementos de un determinado conjunto más amplio, que pese a que mantienen las características esenciales e identificativas del conjunto al que pertenecen, también son lo suficientemente relevantes como para ser rasgos distintivos de dicho subconjunto de elementos.

• *Regla ¿Es un? (ISA):* responder afirmativamente que un objeto de la clase hija es un objeto de la clase padre.

# 3. HERENCIA POR EXTENSIÓN

**SINTAXIS:**

```
class <claseDerivada> extends <claseBase> {
    …
}
```

```
Ej.: class Abuela {
       …
     }

     class Padre extends Abuela {
       …
     }

     class Hija extends Padre {
       …
     }
```

# 3. HERENCIA POR EXTENSIÓN

**Especialización por adición de atributos y/o métodos:**

```
class <claseDerivada> extends <claseBase> {
    <atributoAñadido>
    …
    <metodoAñadido>
    …
}
```

• *los atributos añadidos* en la clase hija tienen la mismas reglas sintácticas y semánticas que en una clase que no sea derivada;
• *los métodos añadidos* en la clase hija tienen las mismas reglas sintácticas y semánticas que en una clase que no sea derivada excepto que NO tienen acceso a los atributos y métodos privados transmitidos desde la clase padre;

# *3. HERENCIA POR EXTENSIÓN*

**Implicaciones sobre los objetos:**

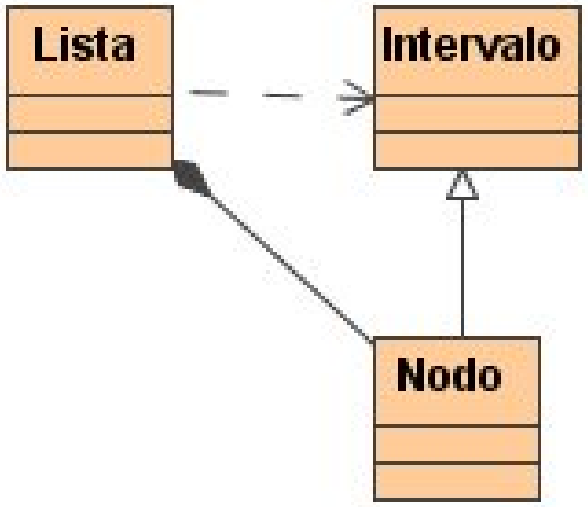
- los *objetos de la clase padre* NO sufren ninguna alteración por la presencia de clases derivadas
- los *objetos de la clase hija*:
  - tienen todos los atributos transmitidos desde la clase padre junto con los atributos añadidos en la clase hija;
  - responden a mensajes que corresponden con los métodos públicos transmitidos desde la clase padre junto con los métodos públicos añadidos en la clase derivada;

# 3. HERENCIA POR EXTENSIÓN

*POR TANTO: los objetos de la clase hija contienen los atributos privados transmitidos desde la clase padre aunque en la implantación de la clase derivada no se puede acceder a ellos*

*IMPLICACIÓN: contención del mantenimiento, dado que, si se modifica la implantación de la clase padre, no repercute sobre la implantación de la clase hija y se obtiene un mínimo acoplamiento entre ambas clases*

*Ej. Una lista de intervalos contiene nodos; un nodo es un intervalo enlazable a otros nodos;*



# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Intervalo {

       private double minimo;
       private double maximo;

       public double getMinimo() {
          return minimo;
       }

       public double getMaximo() {
          return maximo;
       }

       public void asignar(Intervalo intervalo) {
          this.minimo = intervalo.minimo;
          this.maximo = intervalo.maximo;
       }

       public boolean iguales(Intervalo intervalo) {
          return minimo == intervalo.minimo &&
                 maximo == intervalo.maximo;
       }
       …
```

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Nodo extends Intervalo {

       private Nodo anterior;
       private Nodo siguiente;

       public Nodo(Nodo anterior, Nodo siguiente) {
         this.setAnterior(anterior);
         this.setSiguiente(siguiente);
       }

       public Nodo getAnterior() {
         return anterior;
       }

       public Nodo getSiguiente() {
         return siguiente;
       }

       …
```

# 3. HERENCIA POR EXTENSIÓN

Ej.:

```
  public void setAnterior(Nodo anterior) {
    this.anterior = anterior;
    if (anterior != null) {
      anterior.siguiente = this;
    }
  }

  public void setSiguiente(Nodo siguiente) {
    this.siguiente = siguiente;
    if (siguiente != null) {
      siguiente.anterior = this;
    }
  }

  public Intervalo getIntervalo() {
    return new Intervalo(this.getMinimo(),
                         this.getMaximo());
  }
}
```

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Lista {

       private Nodo inicio;
       private Nodo fin;

       public Lista() {
         inicio = null;
         fin = null;
       }

       public boolean vacia() {
         return inicio == null;
       }

       public void insertarInicio(Intervalo intervalo) {
         inicio = new Nodo(null, inicio);
         inicio.asignar(intervalo);
         if (fin == null) {
           fin = inicio;
         }
       }
       …
```

# 3. HERENCIA POR EXTENSIÓN

Ej.:
```
public void insertarFin(Intervalo intervalo) {
  fin = new Nodo(fin, null);
  fin.asignar(intervalo);
  if (inicio == null) {
    inicio = fin;
  }
}

public boolean esta(Intervalo intervalo) {
  if (this.vacia()) {
    return false;
  } else {
    Nodo recorrido = inicio;
    while (recorrido.getSiguiente() != null &&
        !recorrido.getIntervalo().iguales(intervalo)) {
       recorrido = recorrido.getSiguiente();
    }
    return recorrido.getIntervalo().iguales(intervalo);
  }
}
…
```

# 3. HERENCIA POR EXTENSIÓN

Ej.:

```
  public Intervalo eliminarInicio() {
    Intervalo intervalo = inicio.getIntervalo();
    inicio = inicio.getSiguiente();
    if (inicio == null) {
      fin = null;
    } else {
      inicio.setAnterior(null);
    }
    return intervalo;
  }

  public Intervalo eliminarFin() {
    Intervalo intervalo = fin.getIntervalo();
    fin = fin.getAnterior();
    if (fin == null) {
      inicio = null;
    } else {
      fin.setSiguiente(null);
    }
    return intervalo;
  }
}
```

# 3. HERENCIA POR EXTENSIÓN

*PROBLEMA: la clase padre no transmite los métodos públicos necesarios para manipular los atributos privados transmitidos desde la clase padre en los métodos añadidos en la clase hija*
*(Ej. No existen o son privados los métodos "getMinimo", "getMaximo" y "asignar" de la clase Intervalo)*

**Visibilidad *public*:** Añadir dichos métodos públicos a la clase padre NO es solución puesto que rompe el principio de encapsulación ya que, para la implantación de una clase hija, los objetos de la clase padre dan a conocer más allá de lo que se les solicitaba previamente a la existencia de la clase derivada.

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Intervalo {

        private double minimo;
        private double maximo;

        public double getMinimo() {
           return minimo;
        }

        public double getMaximo() {
           return maximo;
        }

        public void asignar(Intervalo intervalo) {
            this.minimo = intervalo.minimo;
            this.maximo = intervalo.maximo;
        }
        …
```

# *3. HERENCIA POR EXTENSIÓN*

**Visibilidad *protected*:** los miembros (atributos y/o métodos) son accesibles en la implantación de la clase y en cualquier clase derivada.

Métodos ***protected get/set***: son métodos para obtener el valor y asignar un valor a los atributos de la clase que posibilitan cualquier manipulación por parte de la clase hija futura;

*IMPLICACIÓN: contención del mantenimiento dado que si se modifica la implantación de la clase padre no repercute sobre la implantación de la clase hija y se obtiene un mínimo acoplamiento entre ambas clases*

# 3. HERENCIA POR EXTENSIÓN

Ej.:
```
class Intervalo {

    private double minimo;
    private double maximo;

    protected double getMinimo() {
      return minimo;
    }

    protected double getMaximo() {
      return maximo;
    }

    protected void setMinimo(double minimo) {
      this.minimo = minimo;
    }

    protected void setMaximo(double maximo) {
      this.maximo = maximo;
    }

    protected void asignar(Intervalo intervalo) {
      this.setMinimo(intervalo.getMinimo());
      this.setMaximo(intervalo.getMaximo());
    }

    …
```

# 3. HERENCIA POR EXTENSIÓN

Ej.:
```
class Intervalo {

    private double puntoMedio;
    private double longitud;

    protected double getMinimo() {
      return puntoMedio – longitud / 2;
    }

    protected double getMaximo() {
      return puntoMedio + longitud / 2;
    }

    protected void setMinimo(double minimo) {
      this.longitud = this.getMaximo() - minimo;
      this.puntoMedio = minimo + longitud / 2;
    }

    protected void setMaximo(double maximo) {
      this.longitud = maximo - this.getMinimo();
      this.puntoMedio = maximo – longitud / 2;
    }

    protected void asignar(Intervalo intervalo) {
      this.setMinimo(intervalo.getMinimo());
      this.setMaximo(intervalo.getMaximo());
    }
    …
```

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Nodo extends Intervalo {
       …
       public Intervalo getIntervalo() {
         return new Intervalo(this.getMinimo(),
                              this.getMaximo());
       }

       public void setIntervalo(Intervalo intervalo) {
         this.asignar(intervalo);
       }
       …
     }
```

# *3. HERENCIA POR EXTENSIÓN*

Atributos ***protected***: dentro del cuerpo de los métodos de la clase derivada se tiene acceso a los atributos transmitidos, a los atributos añadidos, a los parámetros del método y a las declaraciones locales (*ley flexible de Demeter*)

*IMPLICACIÓN: desbordamiento del mantenimiento dado que si se modifica la implantación de la clase padre **SI** repercute sobre la implantación de la clase hija y se obtiene un máximo acoplamiento entre ambas clases*

```
Ej.: class Intervalo {
        protected double minimo;
        protected double maximo;

        …
     }
```

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Nodo extends Intervalo {
        …
        public Intervalo getIntervalo() {
           return new Intervalo(minimo, maximo);
        }
        …
     }
```

```
Ej.: class Nodo extends Intervalo {
        …
        public Intervalo getIntervalo() {
           return new Intervalo(puntoMedio – longitud / 2,
                                puntoMedio + longitud / 2);
        }
        …
     }
```

# 3. HERENCIA POR EXTENSIÓN

**Especialización por redefinición de métodos:**

```
class <claseDerivada> extends <claseBase> {
    <metodoRedefinido>
    …
}
```

• donde la cabecera del método es exactamente igual[1] a la cabecera del método de la clase padre, excepto su visibilidad, que puede ampliarse;

• la visibilidad del método en la clase padre no puede ser *private*.

• Sus implicaciones son:

  • se anula la transmisión del método de la clase padre;

  • los objetos de la clase padre responden al mensaje con el comportamiento dado en la clase padre;

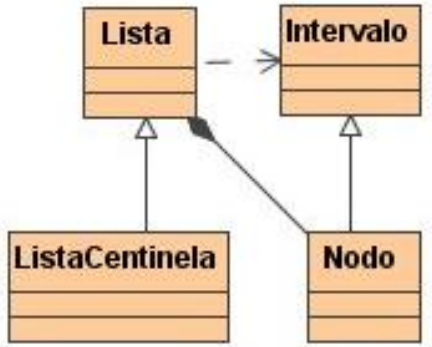
  • los objetos de la clase hija responden al mensaje con el comportamiento dado en la clase hija;



[1] *en caso contrario, sería sobrecarga y no redefinición*

# 3. HERENCIA POR EXTENSIÓN

*Ej. Una lista con centinela*
*es una lista que optimiza*
*la búsqueda de elementos*




```
Ej.: class ListaCentinela extends Lista {

       public boolean esta(Intervalo intervalo) {
         this.insertarFin(intervalo);
         Nodo centinela = this.getFin();
         Nodo recorrido = this.getInicio();
         while (!recorrido.getIntervalo().iguales(intervalo)) {
           recorrido = recorrido.getSiguiente();
         }
         boolean esta = recorrido != centinela;
         this.eliminarFin();
         return esta;
       }
     }
```

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Lista {

       private Nodo inicio;
       private Nodo fin;
       …
       protected Nodo getInicio() {
          return inicio;
       }

       protected Nodo getFin() {
          return fin;
       }

       protected void setInicio(Nodo inicio) {
          this.inicio = inicio;
       }

       protected void setFin(Nodo fin) {
          this.fin = fin;
       }
       …
    }
```

# 3. HERENCIA POR EXTENSIÓN

• **super**, en la implantación de cualquier clase derivada, es una referencia constante que guarda la dirección del objeto que recibe el mensaje correspondiente al método que se está redefiniendo, pero con el comportamiento de la clase padre; por tanto:

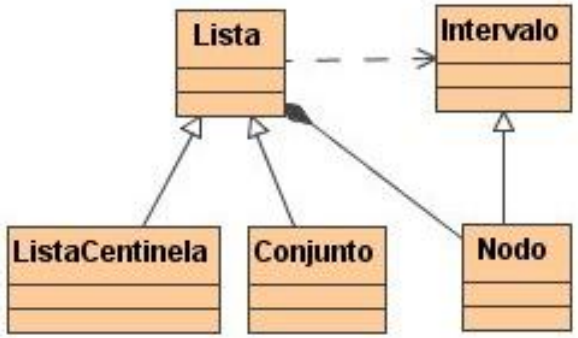
```
private final <ClasePadre> super;
```

**USO:**

- reutilización del método de la clase padre, anulado en la transmisión, desde la redefinición del método de la clase hija.

# 3. HERENCIA POR EXTENSIÓN

*Ej. Un conjunto es una lista que evita elementos repetidos*




```
Ej.: class Conjunto extends Lista {

       public void insertarInicio(Intervalo intervalo) {
          if (!this.esta(intervalo)) {
             super.insertarInicio(intervalo);
          }
       }

       public void insertarFin(Intervalo intervalo) {
          if (!this.esta(intervalo)) {
             super.insertarFin(intervalo);
          }
        }
       …
```

# 3. HERENCIA POR EXTENSIÓN

Ej.:
```
public Conjunto union(Conjunto conjunto) {
  Conjunto union = new Conjunto();
  Iterador iterador = this.iterador();
  while (iterador.hasNext())
    union.insertarInicio(iterador.next());
  iterador = conjunto.iterador();
  while (iterador.hasNext())
    union.insertarInicio(iterador.next());
  return union;
}

public Conjunto interseccion(Conjunto conjunto) {
  Conjunto interseccion = new Conjunto();
  Iterador iterador = this.iterador();
  while (iterador.hasNext()) {
    Intervalo intervalo = iterador.next();
    if (conjunto.esta(intervalo))
      interseccion.insertarInicio(intervalo);
  }
  return interseccion;
}
```

# 3. HERENCIA POR EXTENSIÓN

**Especialización de los constructores:**

```
class <claseDerivada> extends <claseBase> {
        …
        <visibilidad> <ClaseDerivada> (<parametros>) {
            super(<argumentos>);
            …
        }
        …
}
```

• donde debe ser la primera sentencia de los constructores de la clase derivada y sus argumentos deben coincidir en número y tipo con la lista de parámetros de algún constructor público o protegido de la clase padre

• se puede omitir para el caso del constructor de la clase padre con una lista vacía de parámetros

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Intervalo {

       private double minimo;
       private double maximo;

       public Intervalo (Intervalo intervalo) {
         this.minimo = intervalo.minimo;
         this.maximo = intervalo.maximo;
       }
       …
     }
```

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Nodo extends Intervalo {

       private Nodo anterior;
       private Nodo siguiente;

       public Nodo(Nodo anterior, Intervalo intervalo,
                   Nodo siguiente) {
         super(intervalo);
         this.setAnterior(anterior);
         this.setSiguiente(siguiente);
       }
       …
     }
```

# 3. HERENCIA POR EXTENSIÓN

```
Ej.: class Lista {

       …
       public void insertarInicio(Intervalo intervalo) {
         inicio = new Nodo(null, intervalo, inicio);
         if (fin == null) {
           fin = inicio;
         }
       }

       public void insertarFin(Intervalo intervalo) {
         fin = new Nodo(fin, intervalo, null);
         if (inicio == null) {
           inicio = fin;
         }
       }
       …
     }
```

# 3. HERENCIA POR EXTENSIÓN

**CLASE: Object**

Toda clase no derivada hereda de la clase predefinida **Object**, que proporciona un conjunto de métodos comunes a todas las clases, algunos de ellos susceptibles de ser redefinidos en cualquier clase de la manera oportuna.

```
public boolean equals(Object)
protected Object clone()
public String toString()
protected void finalize()
public int hashCode()
…
```

• además, proporciona otros métodos para la sincronización de programas concurrentes y para usar la reflexión, que se estudiarán en temas posteriores.

# 4. CLASES ABSTRACTAS

**Clases Concretas:** surgen de la descripción de los atributos y métodos que definen el comportamiento de un cierto conjunto de objetos homogéneos

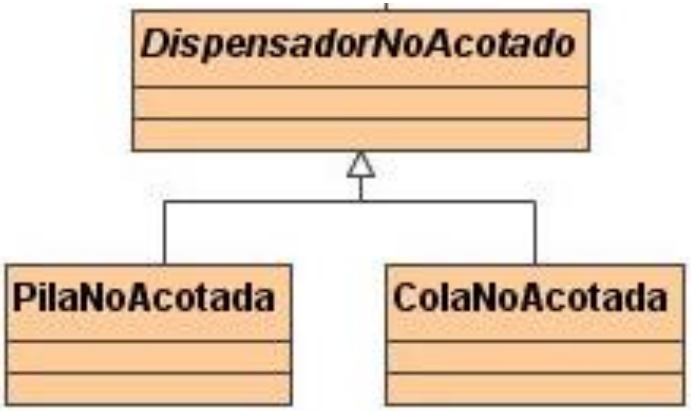
**Clases Abstractas:** son clases NO instanciables que surgen del factor común del código de otras clases con atributos comunes, métodos comunes y/o cabeceras de métodos comunes sin definición.

```
abstract class <claseAbstracta> {
        …
        [ abstract <cabeceraMetodo>; ]
        …
}
…
        … new <claseAbstracta>(…)
```

- donde los métodos abstractos no pueden ser *private.*

# 4. CLASES ABSTRACTAS

*Ej. En un dispensador NO acotado se pueden meter o sacar elementos sin limitación para el número de elementos; una pila es un dispensador con política LIFO; una cola es una dispensador con política FIFO*



# 4. CLASES ABSTRACTAS

```
Ej.: abstract class DispensadorNoAcotado {

       protected Nodo entrada;

       protected DispensadorNoAcotado(){
         entrada = null;
       }

       public void meter (Intervalo intervalo){
         entrada = new Nodo(null, intervalo, entrada);
       }

       public abstract Intervalo sacar();


       public boolean vacia(){
         return entrada == null;
       }
     }
```

# 4. CLASES ABSTRACTAS

```
Ej.: class PilaNoAcotada extends DispensadorNoAcotado {
       public Intervalo sacar() {
         Intervalo intervalo = entrada.getIntervalo();
         entrada = entrada.getSiguiente();
         return intervalo;
       }
     }

     class ColaNoAcotada extends DispensadorNoAcotado {
       private Nodo salida;
       public ColaNoAcotada() {
         salida = null;
       }
       …
```

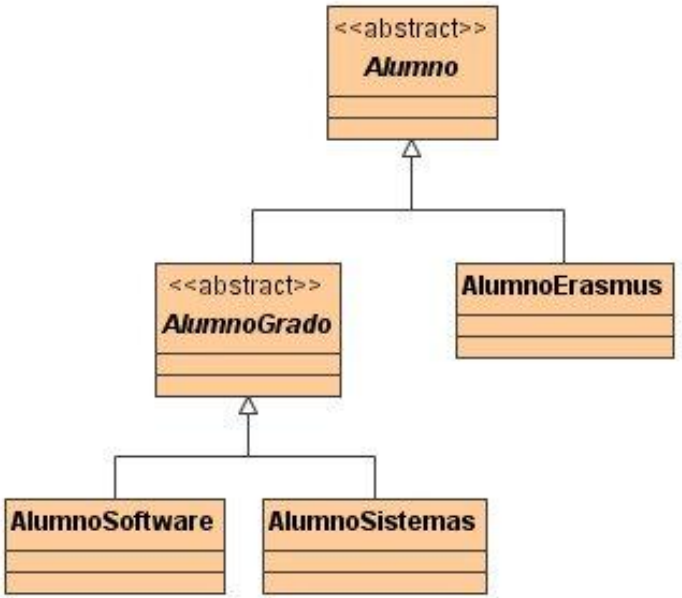
# 4. CLASES ABSTRACTAS

Ej.:
```
public void meter(Intervalo intervalo) {
  boolean vacia = this.vacia();
  super.meter(intervalo);
  if (vacia) {
    salida = entrada;
  }
}

public Intervalo sacar() {
  Intervalo intervalo = salida.getIntervalo();
  if (salida.getAnterior() == null) {
    entrada = null;
    salida = null;
  } else {
    salida = salida.getAnterior();
    salida.setSiguiente(null);
  }
  return intervalo;
}
}
```

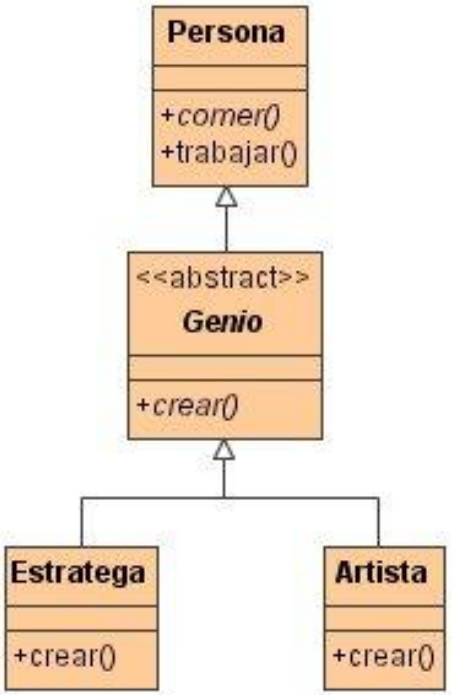
# 4. CLASES ABSTRACTAS

**Posibilidades:**

• una clase abstracta puede ser hija de otra clase abstracta porque se especializa (añadiendo atributos y/o métodos y/o redefiniendo métodos) pero NO redefine todos los métodos abstractos transmitidos y/o añade algún método abstracto;




• una clase abstracta puede ser hija de una clase concreta si en su especialización añade algún método abstracto



# 4. CLASES ABSTRACTAS

• un método no abstracto de una clase abstracta puede definirse apoyándose en métodos abstractos entendiendo que será un código que se transmite hasta clases concretas que redefinen los métodos abstractos;

```
Ej.: abstract class Solido {

       private double densidad;

       public double peso() {
         return densidad * this.volumen();
       }

       public abstract double volumen();
     }
```

# 4. CLASES ABSTRACTAS

```
Ej.: class Cubo extends Solido {

       private double lado;

       public double volumen() {
         return Math.pow(lado, 3);
       }
     }

     class Esfera extends Solido {

       private double radio;

       public double volumen() {
         return 4.0 / 3.0 * Math.PI * Math.pow(radio, 3);
       }
     }
```

# 5. HERENCIA POR IMPLEMENTACIÓN

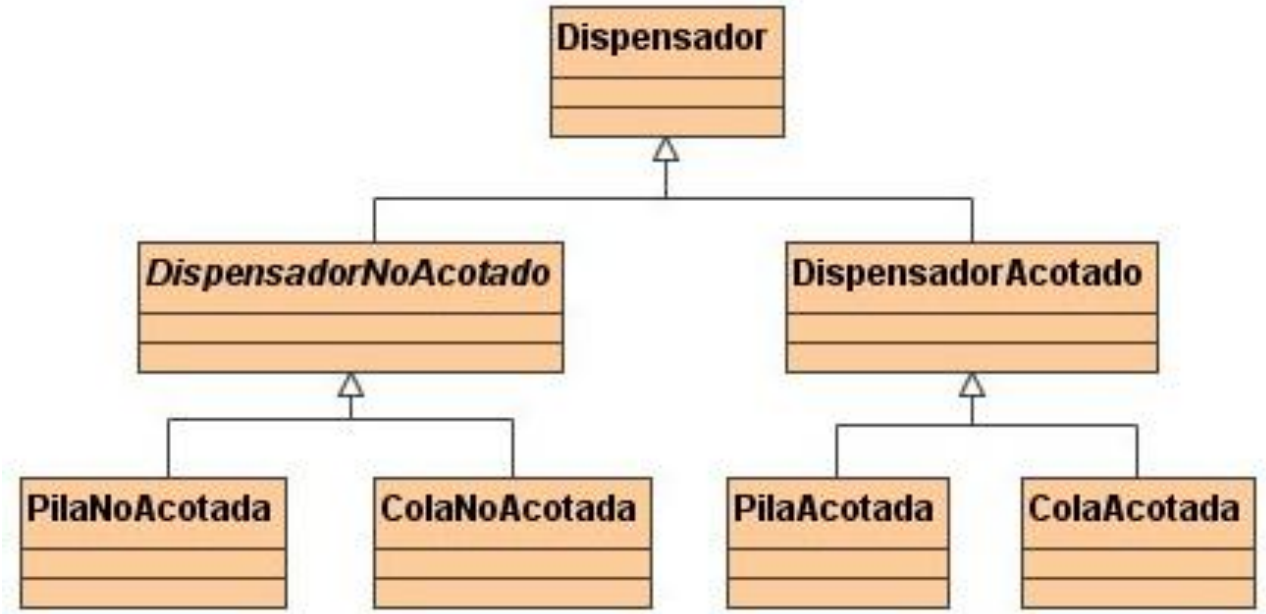
**Interfaces:** son clases abstractas puras que no contienen ningún atributo ni la definición de ningún método, sólo contienen métodos abstractos. Puede ser clases padre de clases u otros interfaces.

```
interface <interfazBase> {
        <cabeceraMetodo1>;
        …
        <cabeceraMetodoN>;
}

[abstract] class <claseDerivada> implements <interfazBase> {
        …
}

interface <interfazDerivado> extends <interfazBase> {
        …
}
```

- donde todos los métodos del interfaz son públicos.

# 5. HERENCIA POR IMPLEMENTACIÓN

*Ej. Un dispensador NO acotado es un dispensador implantado con listas dinámicas; un dispensador acotado es un dispensador implantado con vectores*



# 5. HERENCIA POR IMPLEMENTACIÓN

```
Ej.: interface Dispensador {

       void meter(Intervalo elemento);

       Intervalo sacar();

       boolean vacia();
     }

     abstract class DispensadorNoAcotado
                              implements Dispensador {
        …
       // expuesto anteriormente
     }
```

# 5. HERENCIA POR IMPLEMENTACIÓN

```
Ej.: abstract class DispensadorAcotado implements Dispensador {

      protected Intervalo[] elementos;
      protected int cuantos;
      protected int siguiente;

      protected DispensadorAcotado(int tamaño) {
        elementos = new Intervalo[tamaño];
        cuantos = 0;
        siguiente = 0;
      }

      public void meter(Intervalo intervalo) {
        cuantos++;
        elementos[siguiente] = intervalo;
        siguiente++;
      }
      …
```

# 5. HERENCIA POR IMPLEMENTACIÓN

Ej.:
```
  public boolean vacia() {
    return cuantos == 0;
  }

  public boolean llena() {
    return cuantos == elementos.length;
  }
}

class PilaAcotada extends DispensadorAcotado {

  public PilaAcotada(int tamaño) {
    super(tamaño);
  }

  public Intervalo sacar() {
    cuantos--;
    siguiente--;
    return elementos[siguiente];
  }
}
```

# 5. HERENCIA POR IMPLEMENTACIÓN

```
Ej.: class ColaAcotada extends DispensadorAcotado {

       private int inicio;

       public ColaAcotada(int tamaño) {
         super(tamaño);
         inicio = 0;
       }

       public void meter(Intervalo intervalo) {
         super.meter(intervalo);
         if (siguiente == elementos.length)
           siguiente = 0;
       }

       public Intervalo sacar (){
         cuantos--;
         Intervalo intervalo = elementos[inicio];
         inicio = (inicio + 1) % elementos.length;
         return intervalo;
       }
```

# 6. LIMITACIONES DE LA HERENCIA

- Un interfaz **NO** puede heredar de una clase de ninguna manera, pero **SI** puede heredar de otro interfaz por extensión
- Una clase **SI** puede heredar de una clase por extensión o de un interfaz por implementación
- La herencia por extensión **NO** disfruta de herencia múltiple, pero la herencia por implementación **SI** disfruta de herencia múltiple

```
class <claseDerivada>
      extends <claseBase>
      implements <interfaz1>, …, <interfazN> {
      …
}
interface <interfazDerivado>
      extends <interfaz1>, …, <interfazN> {
      …
}
```

# 6. LIMITACIONES DE LA HERENCIA

**Clases *final*:** no permiten ningún tipo de herencia posterior.

```
final class <clase> {
        …
}
```

**Metodos *final*:** no permiten ningún tipo de redefinición posterior.

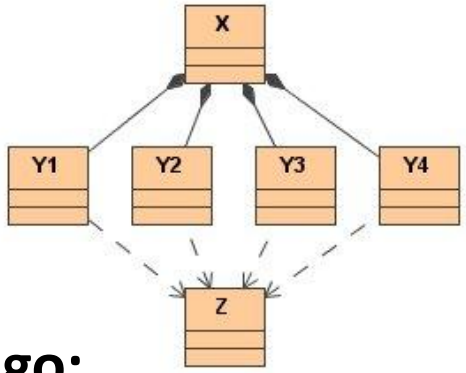
```
class <clase> {
        …
        final <metodo>
        …
}
```

- Los enumerados son siempre *final.*
- Un enumerado **NO** pueden heredar de una clase por extensión, pero **SI** puede heredar de un interfaz por implementación.

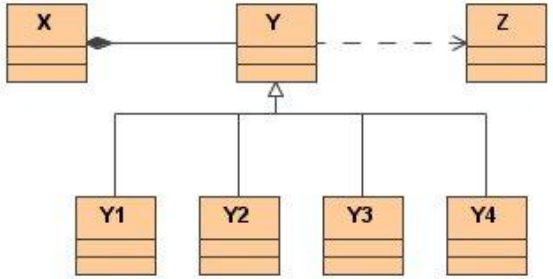
# 7. BENEFICIOS DE LA HERENCIA

**Integridad de la Arquitectura del Software:**

- La herencia favorece la comprensión de la arquitectura del software.
- La jerarquía de clasificación de las clases establece los niveles de generalización que reducen significativamente el número de clases al estudiar en un diseño.




**vs**




**Reusabilidad del código:**

- Utilización del código de la clase padre previamente escrito, probado y documentado.
- No es necesario duplicar código similar, todo el código común se “factoriza” en la clase padre.