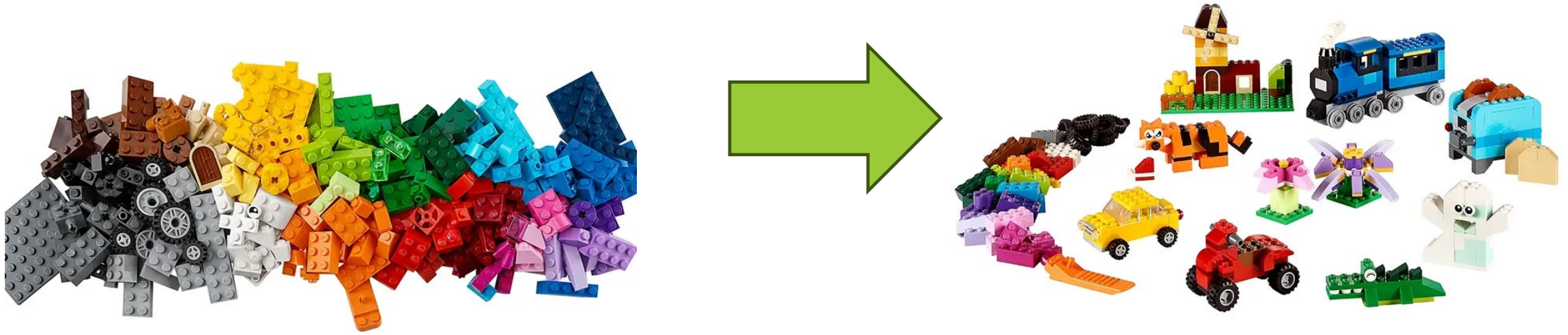
TEMA 3

3.1 RELACIÓN ENTRE CLASES

3.1 RELACIÓN ENTRE CLASES

Los objetos se unen para hacer uno mas grande y complejo



Ref: LEGO. (2015). *LEGO® Medium Creative Brick Box (Set 10696)*. Billund, Denmark: The LEGO Group.

3.1 RELACIÓN ENTRE CLASES

Día Internacional del Cocido: aprende a elaborar el auténtico cocido madrileño

INGREDIENTES

1 trozo de morcillo de ternera (de unos 300 g aproximadamente)
1 muslo de pollo
1 hueso de babilla
1 hueso de caña
1 punta de jamón
2 huesos de espinazo
300 g de garbanzos
1 puerro
3 zanahorias
½ repollo
2 chorizos
1 morcilla de cebolla (se cuece aparte)
1 trozo de tocino ibérico (150 g aproximadamente)
Fideos para la sopa
Sal

PREPARACIÓN

1. Poner los garbanzos a remojo la noche anterior.
2. En una olla grande echar los huesos, la punta de jamón, el morcillo, el pollo, las verduras (el repollo puede cocerse a parte para evitar que suelte mucho sabor) y un puñado de sal. Después cubrirlo todo con agua hasta arriba y dejar cocer durante al menos 2 horas a fuego lento.
3. Pasados los primeros 45 minutos de cocción se incorporan los garbanzos (mejor si van en una rejilla, para que sea más fácil retirar el cocido).
4. Mientras cocer la morcilla y el repollo en ollas aparte.
5. Cuando esté todo en su punto (comprobar con ayuda de un cuchillo o un tenedor), se retira el cocido del fuego y se va separando el caldo del resto de ingredientes.
6. En una olla aparte poner caldo y hacer la sopa en función de los comensales.
7. Mientras, se trocea la carne y las verduras y se sirven en un plato aparte. En otro se ponen los garbanzos. Cuando esté lista la sopa, hay quién prefiere echarse los garbanzos en ella, esto será en cuestión de gusto

Los objetos se unen y cambian su naturaleza dinámicamente para hacer uno mas grande y complejo

RTVE. *Día Internacional del Cocido: aprende a elaborar el auténtico cocido madrileño*. RTVE.es.

3.1 RELACIÓN ENTRE CLASES

Relaciones entre Clases

- Colaboración entre objetos (orientada al tiempo de ejecución)

  Estudian cómo colaboran los objetos mediante el paso de mensajes

  - ***Composición/Agregación***
  - ***Asociación***
  - ***Dependencia (uso)***
- Transmisión entre clases (orientada al tiempo de compilación)

  Estudian como una clase transmite a otra todos sus atributos y métodos

  - ***Herencia por extensión***
  - ***Herencia por implementación***

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

- Características de la colaboración entre objetos:
  - Visibilidad: determina si la comunicación entre dos objetos es "exclusiva", es decir, sólo hablan entre ellos (visibilidad privada) o si los objetos hablar con terceros (visibilidad pública).
  - Temporalidad: determina si la colaboración es temporal o duradera en términos de tiempo.
  - Versatilidad: determina si la colaboración es con un determinado objeto o con cualquier objeto genérico del mismo tipo
- Estas tres características ayudan a tipificar los tipos de relaciones.

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

- Relación de Composición/Agregación
  - Es la relación que se constituye entre el todo y las partes.
  - Hay una relación de composición/Agregación entre la **clase A**, el todo, y la **clase** B, la parte, si un objeto de la clase A **"tiene un"** objeto de la clase B.
  - Se trata de composición cuando la vida del objeto de clase contenida tiene que coincidir con la vida de la clase contenedoras. La supresión del todo implica la supresión de los componentes. Los componentes no pueden compartirse. **Composición Fuerte**.
  - Se trata de agregación cuando la vida del objeto de clase contenida no debe coincidir con la vida de la clase contenedora. La destrucción del todo no implica la destrucción de los componentes. Los componentes pueden compartirse. **Composición Débil**.

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

- Relación de **Composición**

El constructor crea las partes.

Características	Notación UML	Código Java

• Visibilidad: Privada
• Temporalidad: Alta
• Versatilidad: Baja
		`class Todo {`
		`  private Parte parte;`
		
		`  public Todo(){`
		`  this.parte = new Parte();`
		`  }`
		`}`
		
		`class Parte {`
		`}`

- **Implantación mediante atributos y creación en el constructor o ...**
  - referencias privadas con ciclo de vida igual al objeto

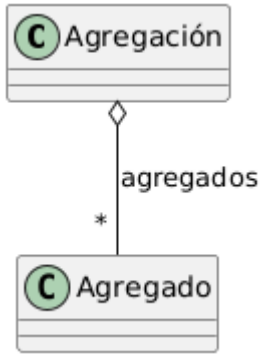
3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

- Relación **Agregación**

Crea la ***colección*** de objetos y, normalmente, no crea los objetos agregados sino que dispone de métodos para añadirlos a la colección

Características	Notación UML	Código Java

• Visibilidad: Pública
• Temporalidad: Alta/Media
• Versatilidad: Baja
		`class Agregación {`
		`  private List<Agregado> agregados;`
		
		`  public Agregación(){`
		`  this.agregados = new ArrayList<Agregado>();`
		`  }`
		
		`  public void add(Agregado agregado){`
		`  this.agregados.add(agregado);`
		`  }`
		
		`  public void remove(Agregado agregado){`
		`  this.agregados.remove(agregado);`
		`  }`
		`}`
		
		`class Agregado {`
		`}`

- **Implantación mediante atributos y métodos de inserción, borrrado o ...**
  - referencias privadas con ciclo de vida igual al objeto

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

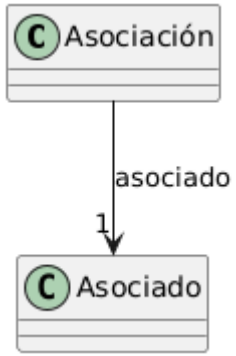
- Relación de Asociación
  - Este tipo de asociación perdura en el tiempo entre clases.
  - Esta relación afecta tanto a temas físicos como lógicos.
  - Una de ellas desempeña el rol de "Servidor" y la otra desempeña el rol de "Cliente"
  - Hay una relación de asociación entre la **clase A, el cliente**, y la **clase B, el servidor**, si un objeto de la clase A disfruta de los servicios de un objeto determinado de la clase B (mensajes lanzados) para llevar a cabo la responsabilidad del objeto de la clase A en diversos momentos creándose una dependencia del objeto servidor.
  - El objeto de clase **cliente** consume métodos de la clase **servidor**.
  - El constructor de clase **cliente** NO CREA el objeto de la clase **servidor** sino que se lo asigna de alguna forma.

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

- Relación de Asociación

Características	Notación UML	Código Java

• Visibilidad: Pública
• Temporalidad: Alta/Media
• Versatilidad: Baja
		`class Asociación {`
		`  private Asociado asociado;`
		
		`  public Asociación(Asociado asociado){`
		`  this.set(asociado);`
		`  }`
		
		`  public void set(Asociado asociado){`
		`  this.asociado = asociado;`
		`  }`
		`}`
		
		`class Asociado {`
		`}`

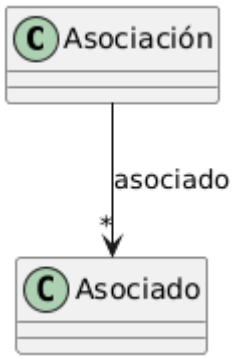
- **Implantación mediante atributos y constructor, métodos de asociación o ...**
  - referencias privadas con ciclo de vida igual al objeto

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

- Relación de Asociación

Características	Notación UML	Código Java

• Visibilidad: Pública
• Temporalidad: Alta/Media
• Versatilidad: Baja
		`class Asociación {`
		`  private List<Asociado> asociados;`
		`  public Asociación(List<Asociado> asociados){`
		`  this.asociados = asociados;`
		`  }`
		`  public void add(Asociado asociado){`
		`  this.asociados.add(asociado);`
		`  }`
		`  public void remove(Asociado asociado){`
		`  this.asociados.remove(asociado);`
		`  }`
		`}`
		`class Asociado {`
		`}`

- **Implantación mediante atributos y constructor, métodos de asociación o ...**
  - referencias privadas con ciclo de vida igual al objeto

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

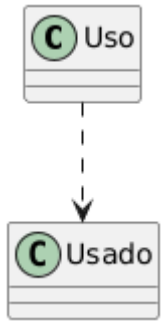
- Relación de Dependencia (uso)
  - Es una relación momentánea entre un cliente y un servidor.
  - Esta relación afecta tanto a temas físicos como lógicos.
  - Existe una relación de uso entre la **clase A, el cliente,** y la **clase B, el servidor**, si un objeto de la **clase A** disfruta de los servicios de un objeto de la **clase B** (mensajes lanzados) para llevar a cabo la responsabilidad del objeto de la clase A en un momento dado sin dependencias futuras.
  - ***No va a ser un atributo de la clase ya que no es información relevante de la misma.***
  - El objeto servidor se crea fundamentalmente dentro de un método.

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos

- Relación de Dependencia (uso)

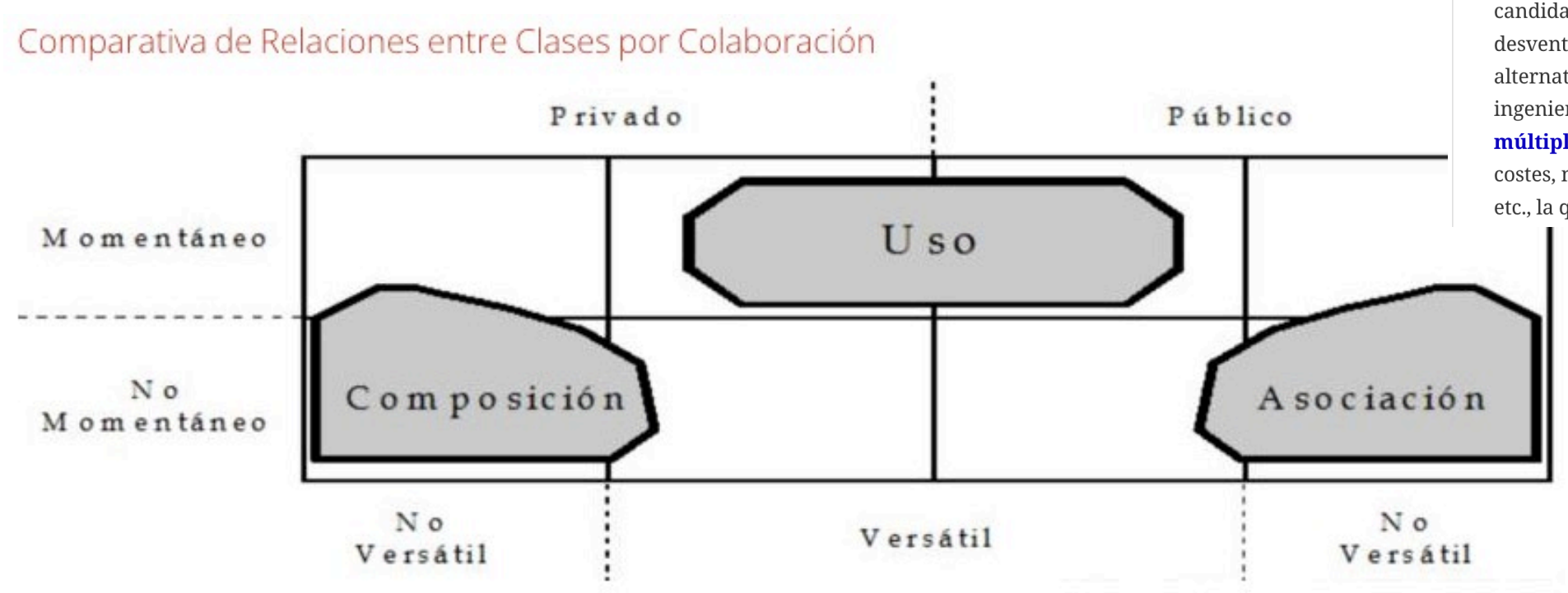
Características	Notación UML	Código Java

• Visibilidad: Pública y Privada
• Temporalidad: Baja
• Versatilidad: Alta
		`class Uso {`
		`  public void método(Usado parámetro){`
		`  Usado local = new Usado();`
		`  ...`
		`  }`
		`}`
		
		`class Usado {`
		`}`

- **Implantación mediante parámetros o variables locales de métodos o ...**
  - referencias con ciclo de vida igual a la ejecución del método

3.1 RELACIÓN ENTRE CLASES

Colaboración entre objetos



- **No existe para toda colaboración un relación ideal categórica**. Es muy frecuente que sean varias relaciones candidatas, cada una con sus ventajas y desventajas. Por tanto, al existir diversas alternativas, será una decisión de ingeniería, un **compromiso entre múltiples factores no cuantificables**: costes, modularidad, legibilidad, eficiencia, etc., la que determine la relación final.

¡El contexto influye!

3.1 RELACIÓN ENTRE CLASES

Herencia entre clases

extend/implements
en java

Pueden encontrarse los siguientes tipos:

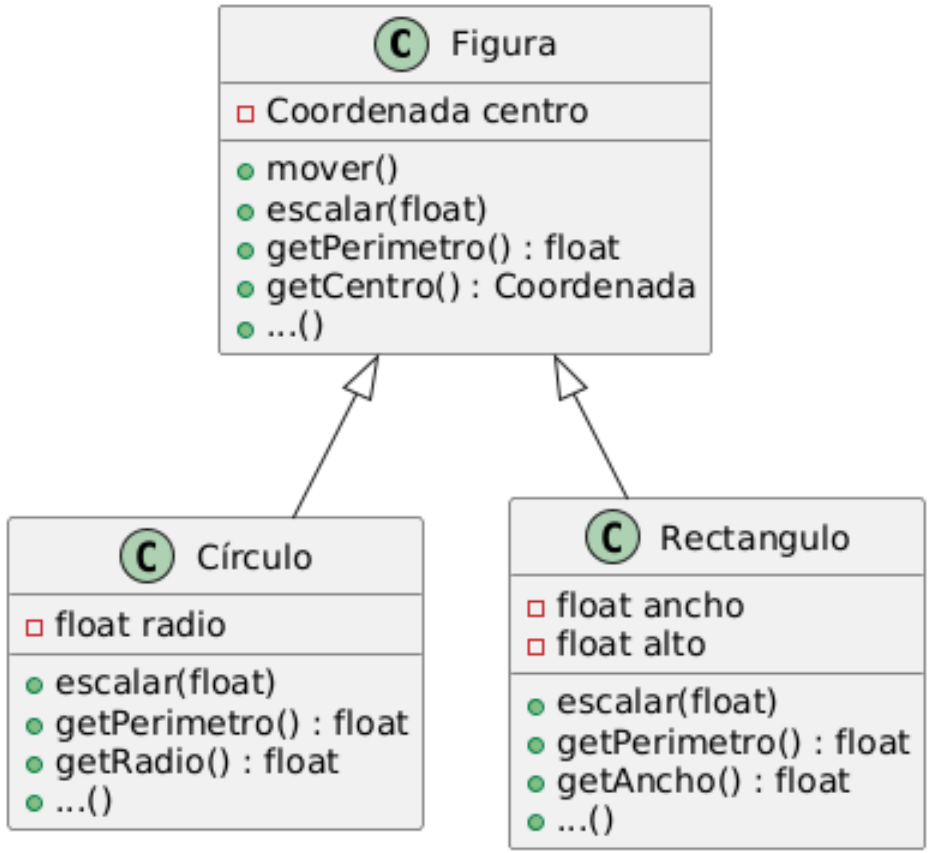
- Correctos
  - Por especialización: donde la clase descendiente implementa todas las operaciones de la clase base, añadiendo o redefiniendo partes especializadas. (extends)
  - Por extensión: donde la especialización transforma el concepto de la clase base a la clase derivada. (extends/implements)
- Incorrectos
  - Por limitación: donde la clase descendiente no implementa todas las operaciones de la clase base, completamente desaconsejada porque imposibilita el tratamiento polimórfico.
  - Por construcción: donde realmente es una relación de composición, completamente desaconsejada.

3.1 RELACIÓN ENTRE CLASES

Herencia entre clases

Por especialización: ☑

- La clase descendiente implementa todas las operaciones de la clase base, añadiendo o redefiniendo partes especializadas0 o la clase ascendiente unifica comportamientos y atributos comunes.
- Permite la incorporación o refinamiento de atributos o métodos específicos.
- Uso de @Override en java

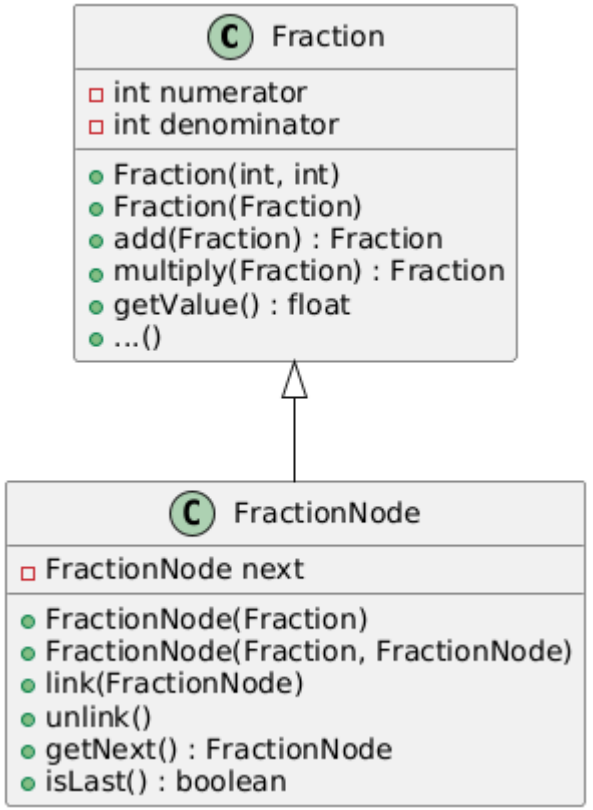


3.1 RELACIÓN ENTRE CLASES

Herencia entre clases

Por extensión: ☑

- donde la especialización transforma el concepto de la clase base a la clase derivada.
- La clase derivada cambia la naturaleza del objeto.
- Permite crear clases "no instanciables" que puede aglutinar parte de la funcionalidad pero cuyo fin no tiene por que ser su propia existencia.

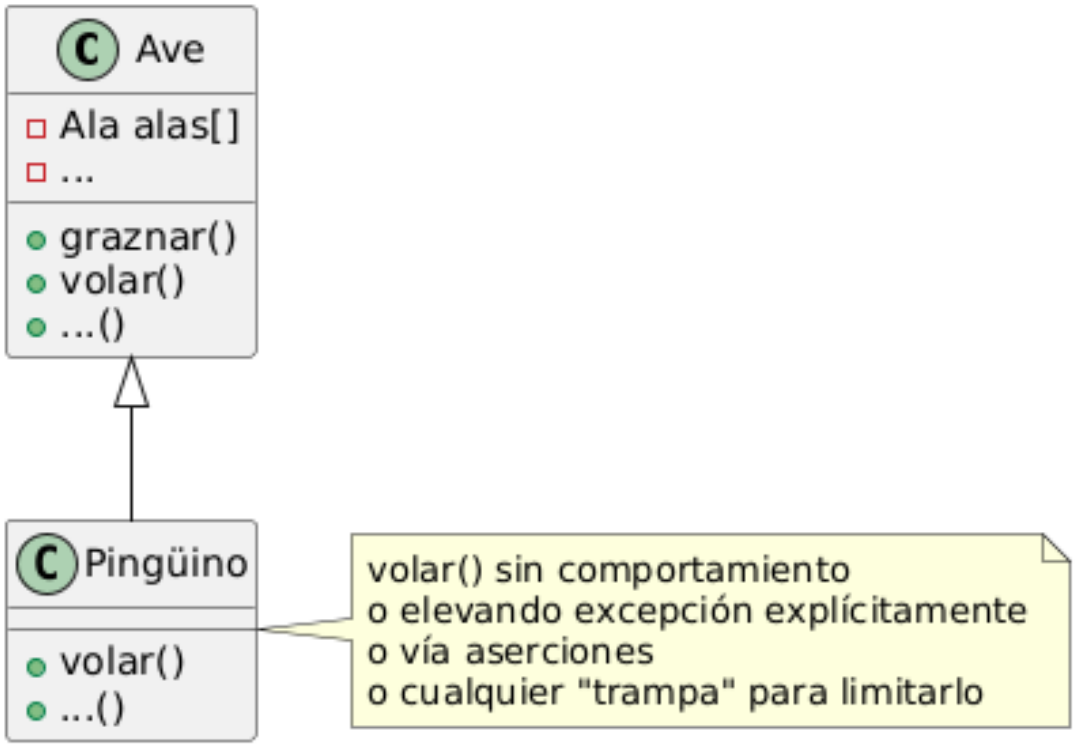


3.1 RELACIÓN ENTRE CLASES

Herencia entre clases

Por limitación: ❌

- donde la clase descendiente no implementa todas las operaciones de la clase base.
- Hace implementaciones vacías de métodos.
- Completamente desaconsejada porque imposibilita el tratamiento polimórfico.
- Representa un mal diseño.

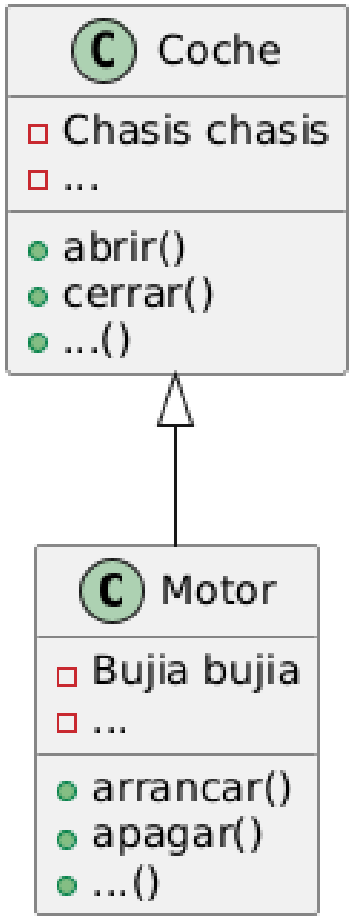


3.1 RELACIÓN ENTRE CLASES

Herencia entre clases

Por construcción: ❌

- donde realmente es una relación de composición.
- Está completamente desaconsejada.
- El todo hereda de la parte.



3.1 RELACIÓN ENTRE CLASES

Diferencia entre "ser parte de" o "se una herencia de"

Compartiva entre Herencia y Composición		
• **Relación**	• **Composición**	• **Herencia**
• **Concepto**	• **A tiene un B**	• **A es un B**, (acrónimo **ISA**)
• **Alternativa**	• **mientras que tener no es siempre ser**.	• **en muchos casos ser también es tener**.
• *Ejemplo*	• *un propietario de un coche es una persona pero no es un coche; un propietario de un coche tiene un coche*	• *un ingeniero del software es un ingeniero, o sea que en cada ingeniero del software hay un ingeniero, o sea, un ingeniero del software tiene un ingeniero*
• **Recomendación**	• **decantarse por la composición**	• Ante la duda, si la cardinalidad de la parte/base en cuestión puede ser mayor que 1, **decantarse por la composición**

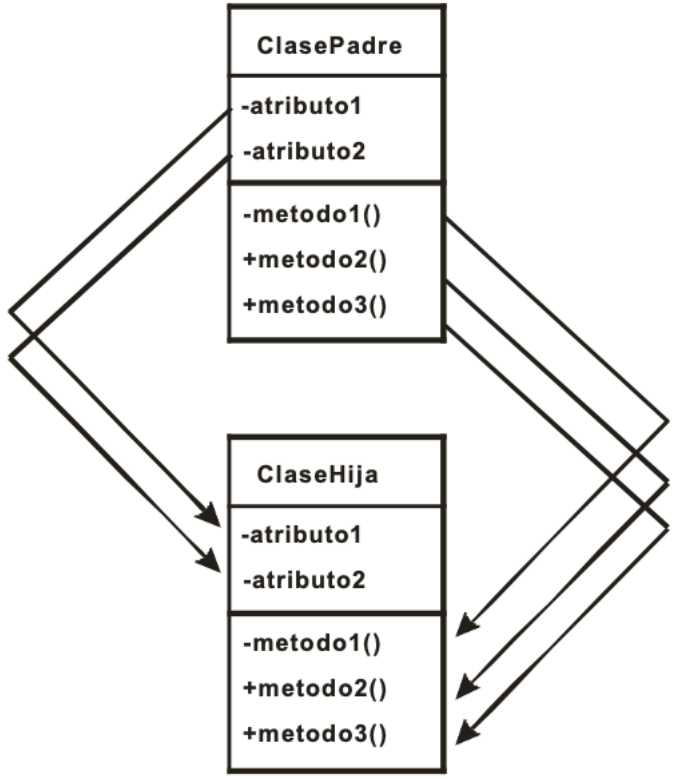
TEMA 3

3.2 HERENCIA POR EXTENSIÓN

3.2 HERENCIA POR EXTENSIÓN

Relación de herencia

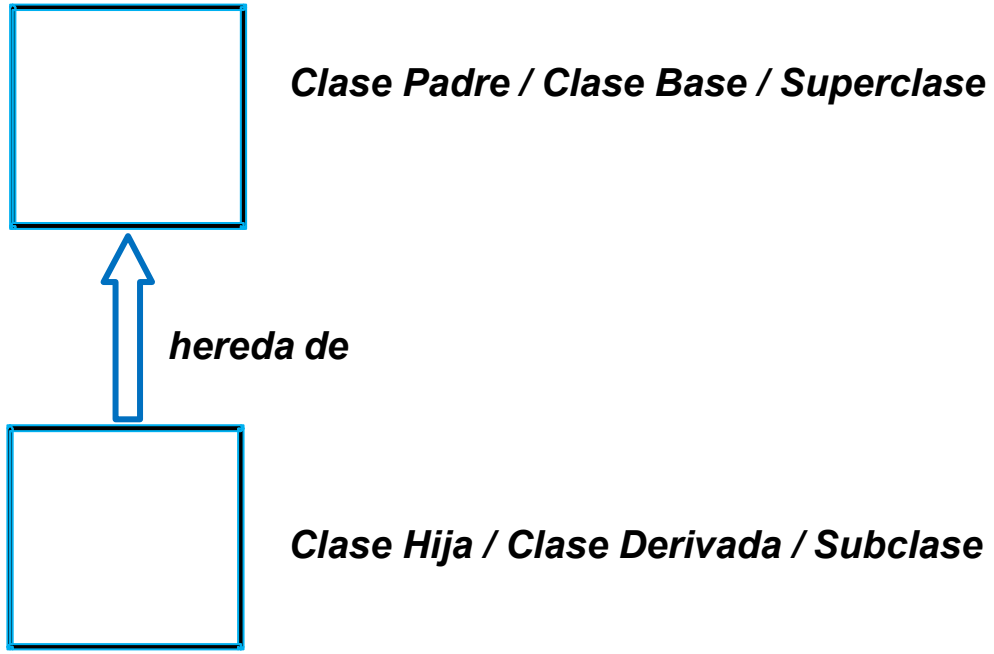
- TRANSMISIÓN:
  - La herencia en todos los ámbitos (derecho, biología, …) tiene connotaciones de transmisión
  - En Programación Orientada a Objetos es la transmisión de la Vista Pública (métodos públicos) y de la Vista Privada (atributos, métodos privados y definición de los métodos) de una clase a otra. Aunque los privados no son accesibles.



3.2 HERENCIA POR EXTENSIÓN

Relación de herencia

- Paralela a los árboles genealógicos (padre, hija, …).
  - Clase base para la que transmite y clases derivadas a las que reciben la transmisión (transitividad).
  - Superclase en desuso y subclases para las clases derivadas mas especificas.



3.2 HERENCIA POR EXTENSIÓN

Relación de herencia

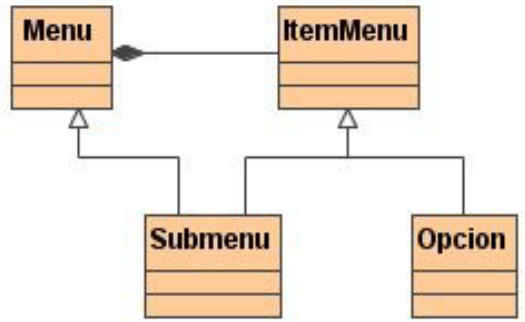
- La relación de herencia:
  - es una relación binaria entre clases
  - si existe una relación de herencia, **no es necesario que exista una colaboración** entre los objetos de sus clases aunque tampoco lo impide (Ej.: la clase Persona hereda de la clase Animal; en una aplicación sobre la evolución de 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**.

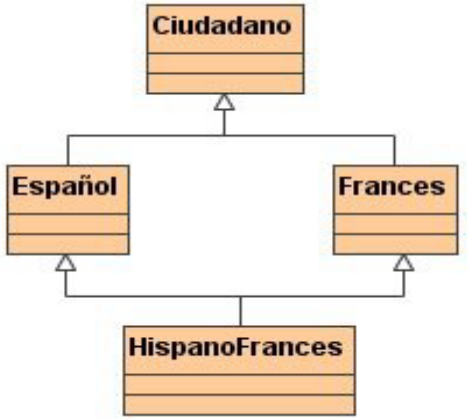
3.2 HERENCIA POR EXTENSIÓN

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



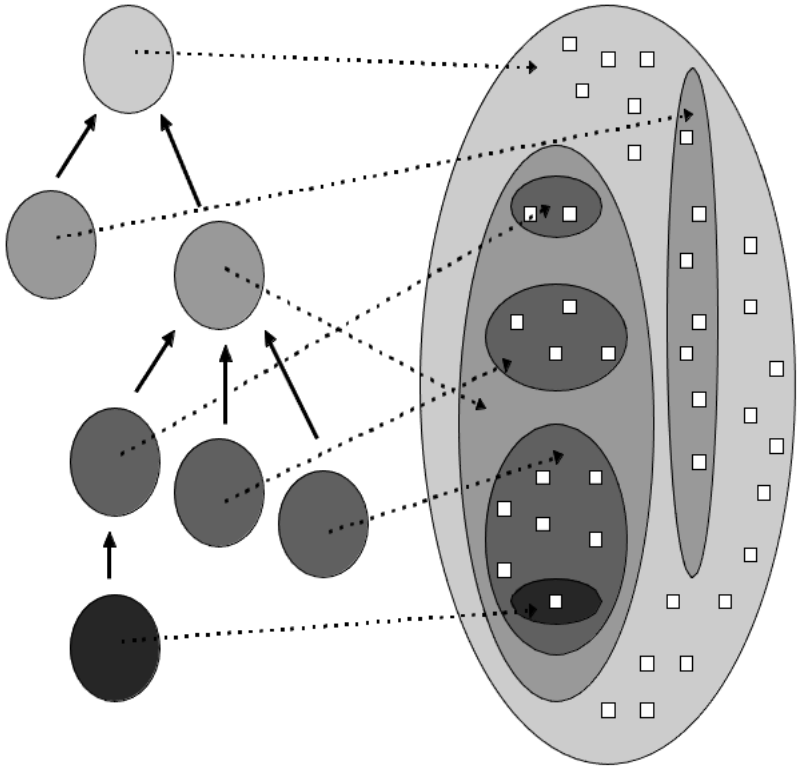


3.2 HERENCIA POR EXTENSIÓN

Relación de herencia

Las herencias generan jerarquías

- 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 herencia consigue clasificar los tipos de datos (abstracciones) por variedad.
- La relación de herencia permite establecer jerarquías por grado de clasificación.

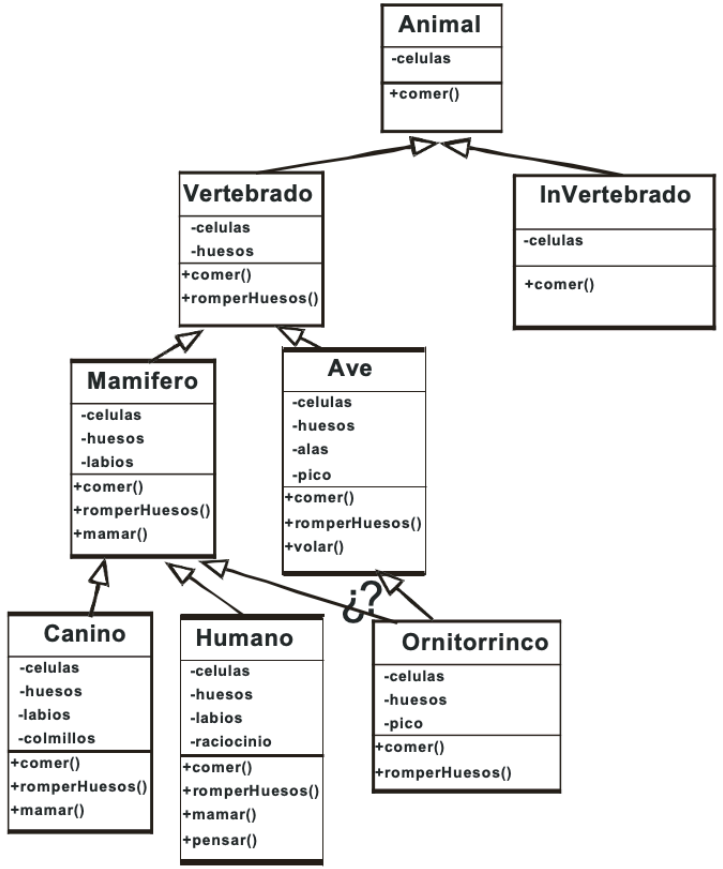


3.2 HERENCIA POR EXTENSIÓN

Relación de herencia

Jerarquías de clasificación

- 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.



3.2 HERENCIA POR EXTENSIÓN

Relación de herencia

Jerarquías de clasificación

- **Regla de Generalización:** Cuando la jerarquía nace de abstaer sobre un padre atributos, comportamientos o relaciones de hijos mas específicos. En este caso, el padre siempre será un elemento "decorativo" (***abstracto***) ya que se ha creado artificialmente.
- **Regla de Especialización**: Cuando la jerarquía nece de crear uno o mas hijos que especializan una clase ya existente. En este caso, el padre puede ser instanciable (se puede crear) siempre y cuando la naturaleza de la extensión no sea sustituir al propio padre en la jerarauia (actualización/ mejora)

3.2 HERENCIA POR EXTENSIÓN

Herencia por extensión en Java:

- Sintaxis:

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

- Ejemplo:

```
class Abuela {
    …
}
class Padre extends Abuela {
    …
}
class Hija extends Padre {
    …
}
```

3.2 HERENCIA POR EXTENSIÓN

Especialización por adición de atributos y/o métodos:

- 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.

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

3.2 HERENCIA POR EXTENSIÓN

Implicaciones sobre los objetos:

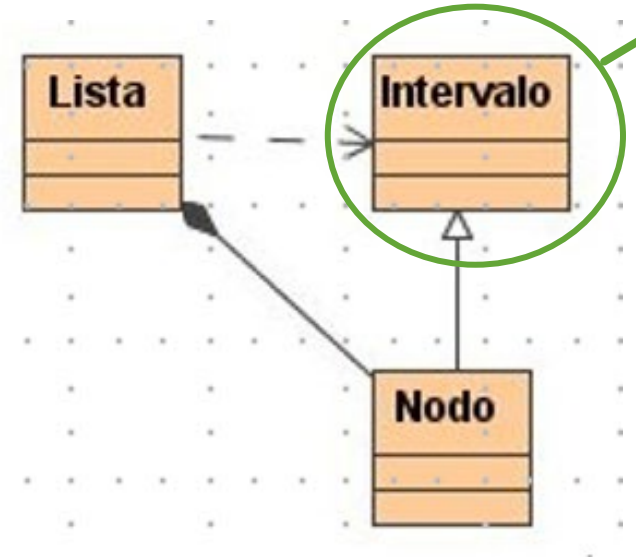
- los **objetos de la clase padre** NO sufren ninguna alteración por la presencia de clases derivadas.
- los **objetos de la clase hija**:
  - tienen todos los atributos transmitidos desde la clase padre junto con los atributos añadidos en la clase hija.
  - responden a mensajes que corresponden con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

3.2 HERENCIA POR EXTENSIÓN

- **NO** tienen acceso a los atributos y métodos **privados** transmitidos desde **la clase padre.**
- **NO** puede cambiar la naturaleza de los **públicos/protected** del padre.
- **PUEDE** crear un atributo con el mismo nombre que un publico, pero entonces lo **sobreescribe** en la clase (**tendrá el propio y el del padre**)
- **PUEDE** sobreescribir los métodos del padre (*sobrecarga*)
- **TIENEN** Los hijos tienen la naturaleza de los padres predecesores como parte de la herencia de **atributos, métodos y relaciones** (*polimorfismo*)

3.2 HERENCIA POR EXTENSIÓN

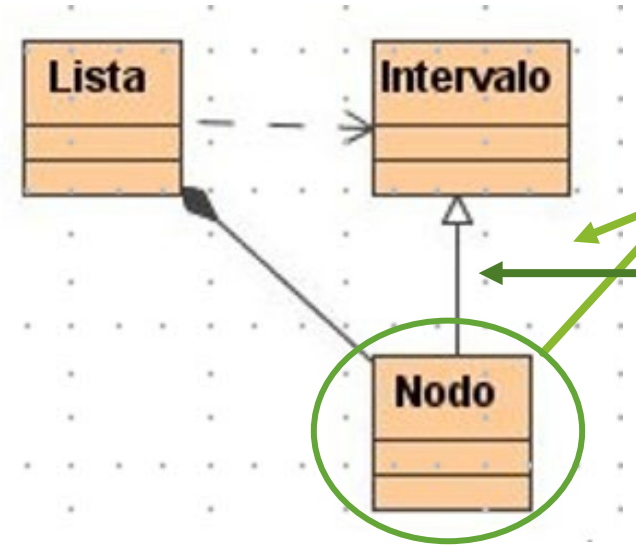
- Ejemplo: implementar




```
class Intervalo {
    private double minimo;
    private double 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;
    }
    …
}
```

3.2 HERENCIA POR EXTENSIÓN

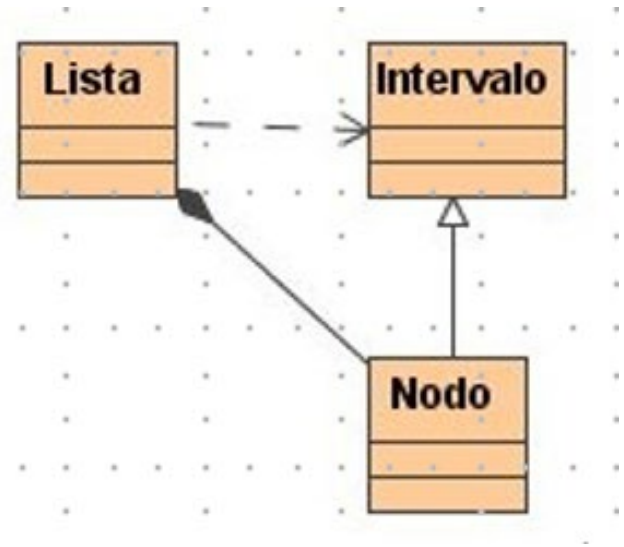
- Ejemplo: implementar




```
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;
    }
    …
```

3.2 HERENCIA POR EXTENSIÓN

- Ejemplo: implementar

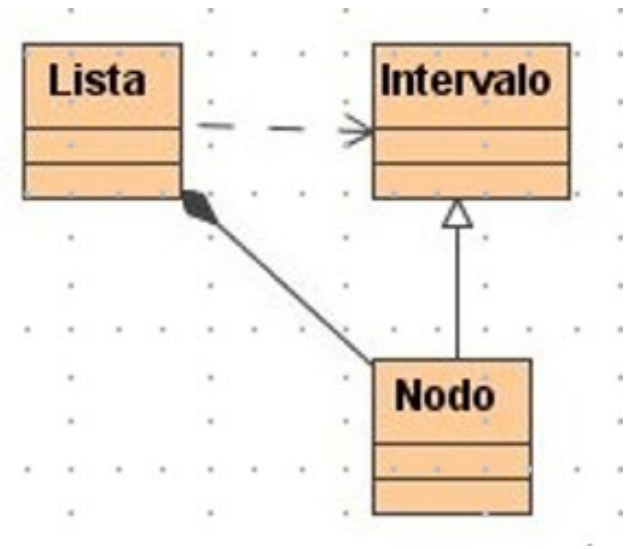



```
public void setAnterior(Nodo anterior) {
    this.anterior = anterior;
    if (anterior != null) {
        anterior.siguiente = this;
    }
}

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

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

    }
} // Class Nodo
```

No existen => definirlos

3.2 HERENCIA POR EXTENSIÓN

- Ejemplo: implementar




```
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;
    }
    …
```

Añadido por necesidad de la clase Nodo

3.2 HERENCIA POR EXTENSIÓN

PROBLEMA

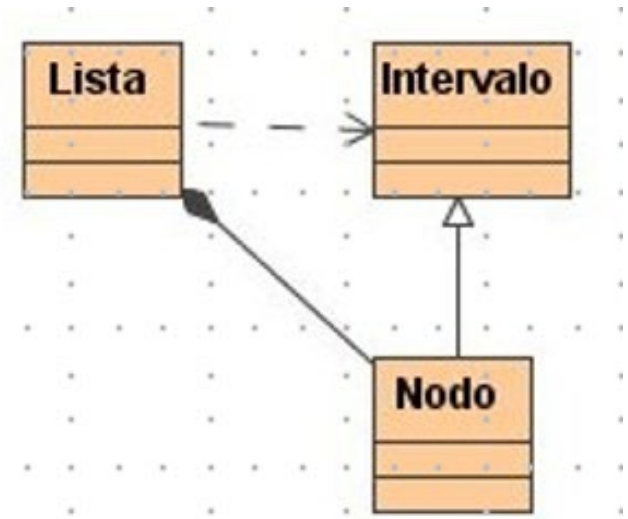
- La clase padre no transmite los métodos públicos necesarios para manipular los atributos privados transmitidos desde la clase padre en los métodos añadidos en la clase hija

  (Ej. No existen o son privados los métodos "getMinimo" y "getMaximo" 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 información de la que se les solicitaba previamente** a la existencia de la clase derivada.

3.2 HERENCIA POR EXTENSIÓN

- Ejemplo: implementar




```
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;
    }
    …
```

Añadido por necesidad de la clase Nodo

3.2 HERENCIA POR EXTENSIÓN

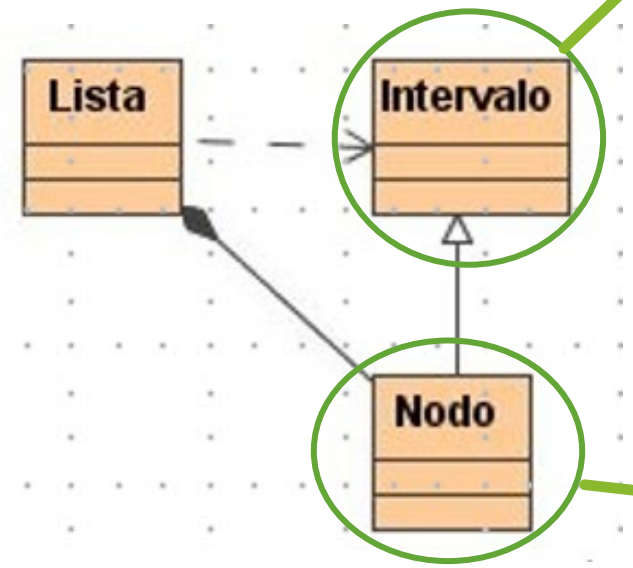
Solución 1:

- **Visibilidad protected**: los miembros (atributos y/o métodos) son accesibles en la implantación de la clase y en cualquier clase derivada.
  - **Atributos protected**: dentro del cuerpo de los métodos de la clase derivada se tiene acceso a los atributos heredados, 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

3.2 HERENCIA POR EXTENSIÓN

Solución 1:

- Ejemplo: Adaptar

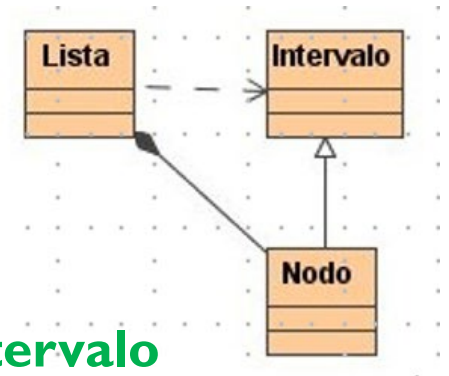



```
class Intervalo {
    protected double minimo;
    protected double maximo;

    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;
    }
    …
}

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

3.2 HERENCIA POR EXTENSIÓN

Solución 1:

- Supongamos que se cambia la forma de definir la clase intervalo y en lugar de venir determinado por el límite inferior y el límite superior viene determinada por los siguientes valores:
  - Punto medio del intervalo: valor double
  - Longitud del intervalo: valor double



¿Qué hay que cambiar? → **Clase Intervalo**
Clase Nodo

3.2 HERENCIA POR EXTENSIÓN

Solución I:

```
Ej.: class Intervalo {
        protected double puntomedio;
        protected double longitud;

        public Intervalo(double minimo, double maximo) {
          puntomedio = (minimo + maximo) / 2;
          longitud = maximo - minimo;
        }
        …
      }
```

¡Demasiados cambios!

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

3.2 HERENCIA POR EXTENSIÓN

Solución 2:

- **Métodos get/set protected**: 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.

3.2 HERENCIA POR EXTENSIÓN

```
class Intervalo {
  private double minimo;
  private double maximo;

  public Intervalo(double minimo, double maximo) {
    this.minimo = minimo;
    this.maximo = maximo;
  }
  protected double getMinimo() {
    return minimo;
  }
  protected double getMaximo() {
    return maximo;
  }
  …
}
```

```
class Intervalo {
  private double puntoMedio;
  private double longitud;

  public Intervalo(double minimo, double maximo) {
    puntoMedio = (minimo + maximo) / 2;
    longitud = maximo - minimo;
  }
  protected double getMinimo() {
    return puntoMedio - longitud / 2;
  }
  protected double getMaximo() {
    return puntoMedio + longitud / 2;
  }
  …
```

```
class Nodo extends Intervalo {
  …
  public Intervalo getIntervalo() {
    return new Intervalo(this.getMinimo(),   // NO CAMBIA
                         this.getMaximo());
  }
  …
}
```

3.2 HERENCIA POR EXTENSIÓN

El atributo super

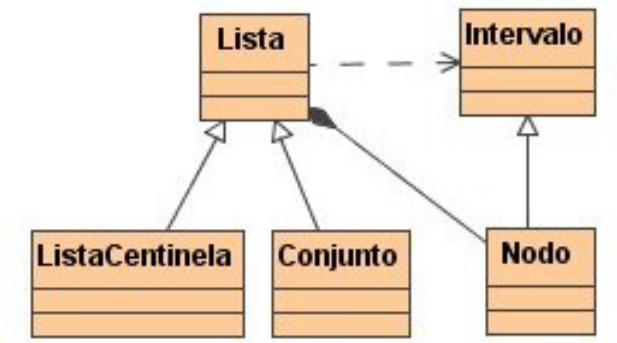
- **super**, Es una atributo por defecto en las clases derivadas para hacer referencia al padre. Tanto en los métodos como atributos.

- **Uso**:
  - Permite acceder de manera explicita a los atributos y métodos publicos/protected.
  - Permite asegurar el acceso y evita que, por reescritura se acceda a un valor no valido o método no valido.
  - Permite invocar manualmente métodos del padre en métodos del hijo de manera limpia y clara.

3.2 HERENCIA POR EXTENSIÓN

El atributo super

- Ejemplo: implementar




```
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);
        }
    }
```

...

3.2 HERENCIA POR EXTENSIÓN

El atributo super

- **Por especialización de constructores**
  - donde super 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.***

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

3.2 HERENCIA POR EXTENSIÓN

El atributo super

```
class Intervalo {
   private double minimo;
   private double maximo;

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

```
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);
   }
   …
}
```

3.2 HERENCIA POR EXTENSIÓN

El atributo super

```
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;
       }
   }
   …
}
```

3.2 HERENCIA POR EXTENSIÓN

Clase Object

- ***RECORDAR***. Toda clase no derivada hereda de la clase predefinida **Object *aunque no este explicitado en la cabecera***, por lo que, las clases derivadas, por herencia, siempre heredaran en ultima instancia de **Object** manteniendo el axioma en java de que todas las clases derivan de **Object** en ultima instancia.

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

…
```

3.2 HERENCIA POR EXTENSIÓN

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
  - Pueden tener cabeceras de métodos comunes con diferente implementación donde pueden definir los métodos como abstractos y nunca en **private**, para poder luego ser implementados de las clases hijas.
  - ***Estas clases nacen desde la herencia por abstracción solamente.***

3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

```
abstract class <claseAbstracta> {
   //Atributos comunes
   …
   [ abstract <cabeceraMetodo>; ]
   …
}
…
```

~~<claseAbstracta> objeto = new <claseAbstracta>(…)~~ 

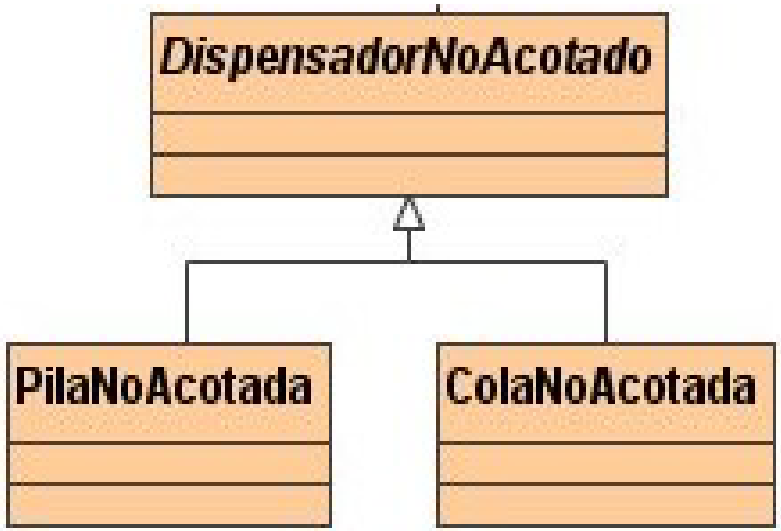
No instanciable

3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

Ejemplo

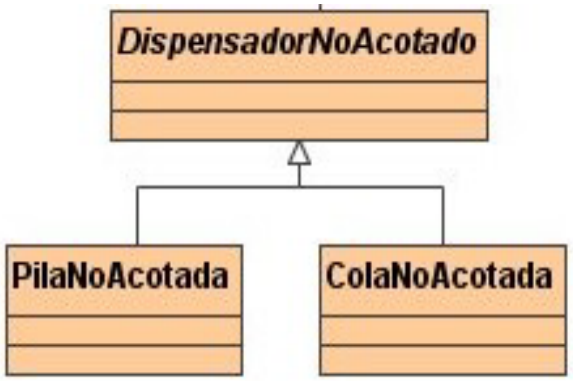
- En un dispensador NO acotado se pueden meter o sacar elementos sin limitación para el número de elementos;
  - una **pila** es un dispensador con política LIFO;
  - una **cola** es una dispensador con política FIFO



3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

- **Ejemplo**

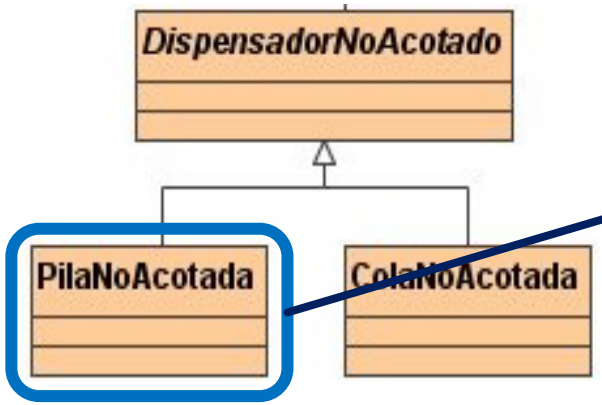



```
abstract class DispensadorNoAcotado {
    protected Nodo entrada;

    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;
    }
}
```

3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

- **Ejemplo**



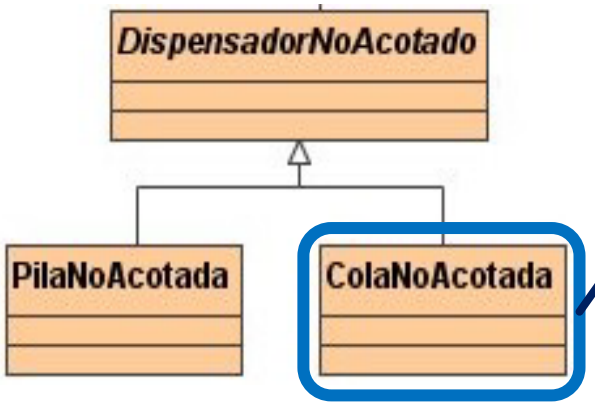

```
abstract class DispensadorNoAcotado {
   protected Nodo entrada;

   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;
   }
}
```

```
class PilaNoAcotada extends DispensadorNoAcotado {

   public Intervalo sacar() {
         Intervalo intervalo = entrada.getIntervalo();
         entrada = entrada.getSiguiente();
         return intervalo;
      }
}
```

3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

- **Ejemplo**



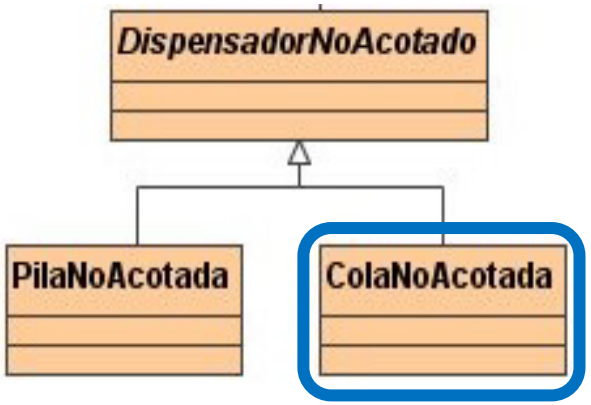

```
class ColaNoAcotada extends DispensadorNoAcotado {
    private Nodo salida;

    public ColaNoAcotada() {
        salida = null;
    }

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

```
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;
    }
}
```

3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

- **Ejemplo**



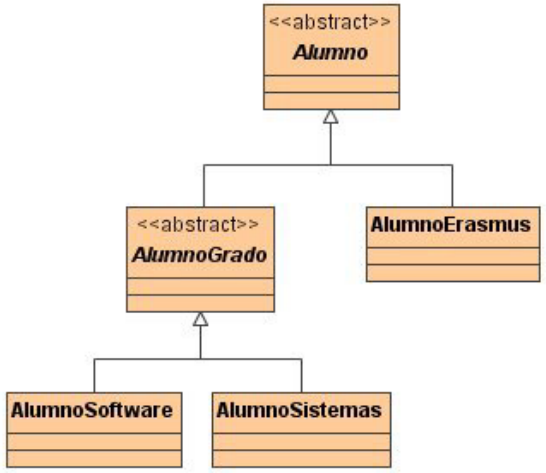

```
abstract class DispensadorNoAcotado {
   protected Nodo entrada;

   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;
   }
}
```

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

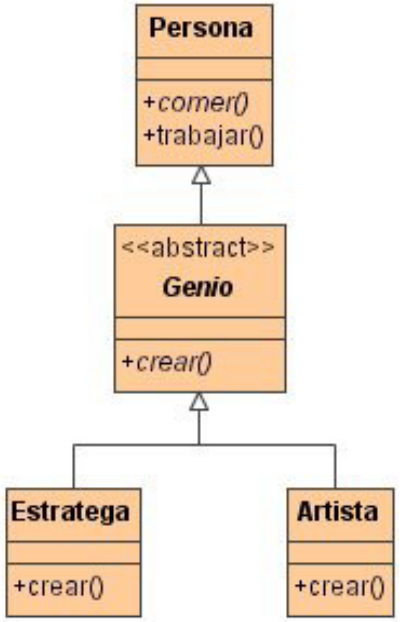
3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

- 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



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



3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

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

```
abstract class Solido {
   private double densidad;

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

   public abstract double volumen();
}
```

3.2 HERENCIA POR EXTENSIÓN

Clases abstractas

```
abstract class Solido {
   private double densidad;

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

   public abstract double volumen();
}
```

```
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);
   }
}
```

TEMA 3

3.3 HERENCIA POR IMPLEMENTACIÓN

3.3 HERENCIA POR IMPLEMENTACIÓN

Interfaces

- Son clases abstractas puras y por tanto nunca instanciables.
- Son clases padre de clases u otros interfaces.
- Todos los métodos y atributos de la interfaz son públicos.

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

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

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

3.3 HERENCIA POR IMPLEMENTACIÓN

Interfaces

- Todos los **atributos** definidos en una interface **serán siempre *public static final*, aunque no estén definidos los modificadores**.
- Todos los **métodos** de la interfaz son **públicos, aunque no estén definidos los modificadores..** Si se desea incluir **cuerpo al método**, **solo** se permitirá para los **métodos static.**

```
interface Configuracion {
    int MAX_USUARIOS = 100;  // es public static final por defecto
    static void info() { … } // es public por defecto.
}
```

3.3 HERENCIA POR IMPLEMENTACIÓN

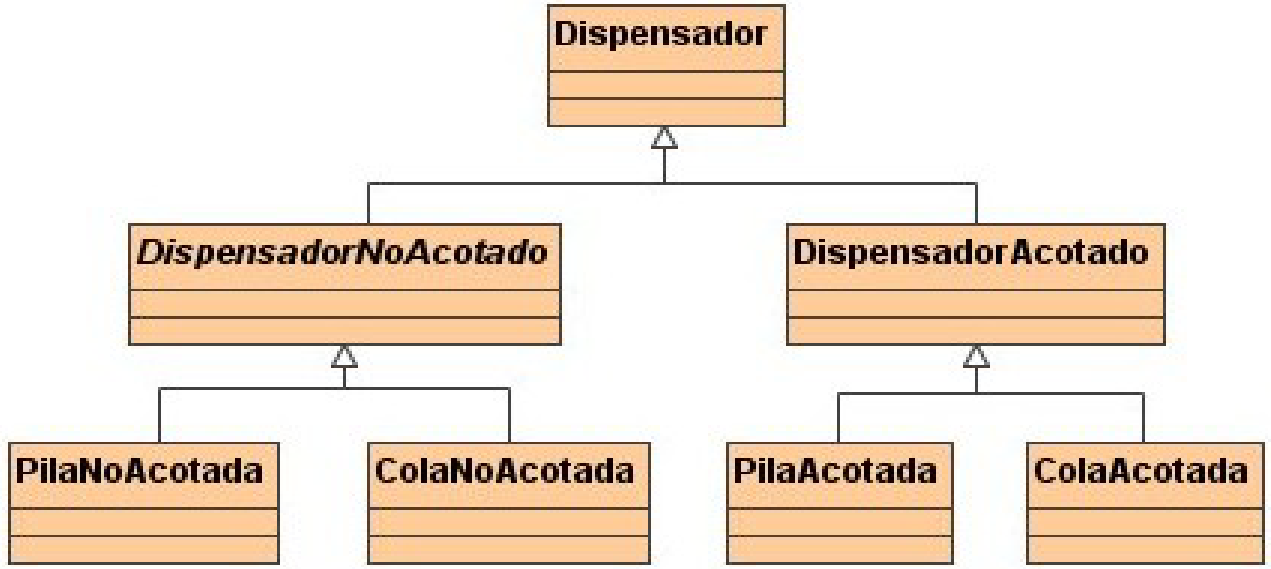
Interfaces

```
// Interfaz
interface MiInterfaz {
    int MAX = 3;
    void saludar(String nombre);
    static void info() { System.out.println("App con máx: " + MAX); }
}

// Clase que implementa
class Impl implements MiInterfaz {
    public void saludar(String n) { System.out.println("Hola " + n); }
}

// Clase principal
public class Main {
    public static void main(String[] a) {
        MiInterfaz.info();
        new Impl().saludar("Ana");
        System.out.println(MiInterfaz.MAX);
    }
}
```

3.3 HERENCIA POR IMPLEMENTACIÓN

Ejemplo Dispensadores

- Un dispensador NO acotado es un dispensador implantado con listas dinámicas;
- Un dispensador acotado es un dispensador implantado con vectores



3.3 HERENCIA POR IMPLEMENTACIÓN

Ejemplo Dispensadores

```
interface Dispensador {
   void meter(Intervalo elemento);
   Intervalo sacar();
   boolean vacia();
}

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

```
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++;
   }
   …
```

3.3 HERENCIA POR IMPLEMENTACIÓN

Ejemplo Dispensadores

```
    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];
    }
}
```

3.3 HERENCIA POR IMPLEMENTACIÓN

Ejemplo Dispensadores

```
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;
   }
}
```

3.3 HERENCIA POR IMPLEMENTACIÓN

Limitaciones de la herencia de extensión vs implementacion (en Java)

- Un interfaz **NO puede heredar de una clase (class)**, pero **SI puede heredar de otro interfaz (interface)** por extensión
- Una clase SI puede heredar de una clase por extensión o de un/varias interfaz por implementación
- La **herencia por extensión NO disfruta de herencia múltiple**, pero la **herencia por implementación SI disfruta de herencia múltiple**

```
interface <interfazDerivado>
   extends <interfaz1>, …, <interfazN>
{
   …
}
```

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

TEMA 3

3.4 POLIMORFISMO Y SOBRECARGA

3.4 POLIMORFISMO Y SOBRECARGA

Limitaciones de la herencia

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

```
final class <clase> {
    …
}
```

- Métodos final: no permiten ningún tipo de redefinición posterior.

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

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

3.4 POLIMORFISMO Y SOBRECARGA

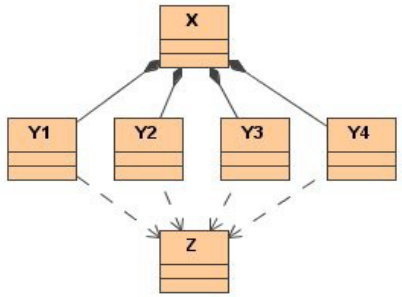
Limitaciones de la herencia

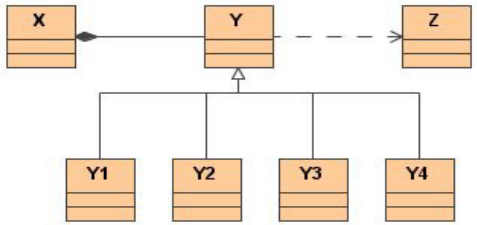
```
interface Accion { void ejecutar(); }

enum Opcion implements Accion {
    INICIAR { public void ejecutar(){System.out.println("Start");}},
    DETENER { public void ejecutar(){System.out.println("Stop");}}
}

public class Main {
    public static void main(String[] a){
        for(Opcion op: Opcion.values()) op.ejecutar();
    }
}
```

3.4 POLIMORFISMO Y SOBRECARGA

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.





- 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.

3.4. POLIMORFISMO Y SOBRECARGA

Polimorfismo

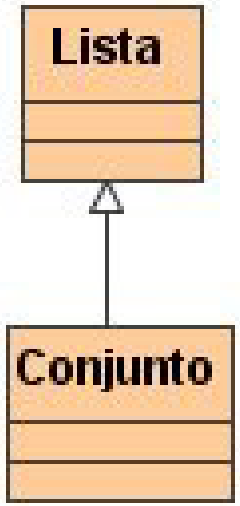
- Término de origen griego que significa "muchas formas".
  - Ej.: Una persona puede pagar con tarjeta o con efectivo
  - Ej.: Una empresa de transporte realiza ventas de billetes por ventanilla o a través de una máquina
  - Ej.: Un sistema operativo imprime a través de drivers de impresora para cada modelo
  - Ej.: Un navegador muestra textos, imágenes, videos, …, con muy diversos formatos

- Limitaciones en Programación Orientada a Objetos:
  - No se contempla que algo cambie de forma.
  - No se contempla que algo sea dos cosas a la vez.
  - Ej.: Una persona de ventanilla NO se convierte en máquina expendedora
  - Ej.: Una persona NO es a la vez una máquina expendedora.
  - Ej.: Simplemente, un billete lo puede vender una persona o una máquina expendedora en cada momento que se vende un billete

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo

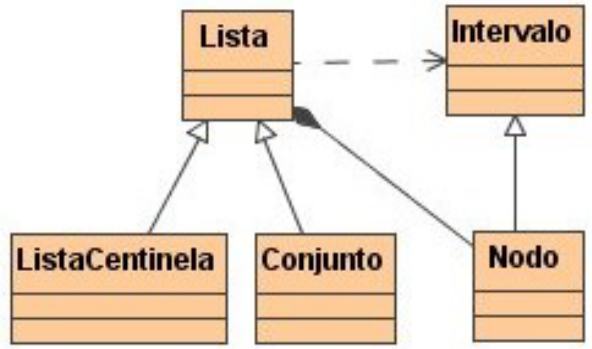
- El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, …).
- Por tanto:
  - **el polimorfismo exige la existencia de una jerarquía de clasificación.**
  - **las jerarquías de clasificación NO exigen tratamientos polimórficos**.



- Ej.: En un punto dado, existen listas, en otro punto, existen conjuntos y, en otro punto, pueden existir indiferentemente listas o conjuntos.

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo



ERROR: Incompatibilidad de tipos

```
Lista lista = new Lista();

Conjunto conjuntoMal   = new Lista();
Conjunto conjunto = new Conjunto();

Lista coleccion;
…
coleccion = new Lista();
…
coleccion = new Conjunto(); // POLIMORFISMO
```

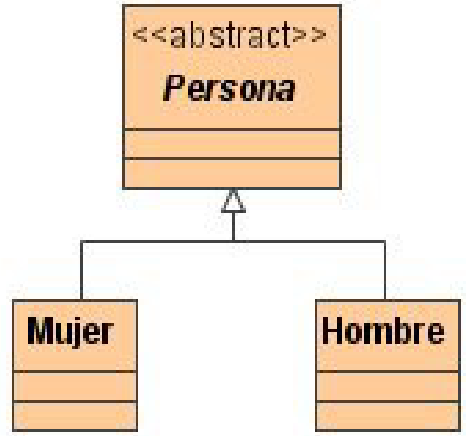
ERROR: Incompatibilidad de tipos

- *donde **un objeto se crea de una clase y siempre será de esa clase** mientras que la referencia a un objeto (como colección) puede contener una dirección de un objeto lista o de un objeto conjunto.*

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo

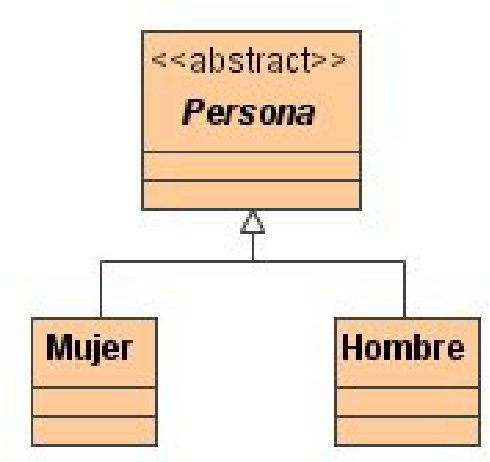
- Con la incorporación del polimorfismo, tiene sentido declarar referencias a clases abstractas con la intención de almacenar direcciones de objetos de clases concretas derivadas, **NO de clase abstractas que no son instanciables**;



- Ej.: En el mundo, existe la jerarquía de personas: mujeres y hombres. Pero en ciertos momentos y en ciertos países, no existe polimorfismo: el lugar de una mujer no lo puede ocupar un hombre y el de un hombre no lo puede ocupar una mujer. Mientras que, en otros momentos u otros países, el lugar de una persona es indiferentemente ocupado por una mujer o un hombre.

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo




```
Mujer mujer = new Mujer();
Mujer mujerMal = new Hombre(); // ERROR 1
Mujer mujerPeor = new Persona(); // ERROR 2
Hombre hombre = new Hombre();
Hombre hombreMal = new Mujer(); // ERROR 1
Hombre hombrePeor = new Persona(); // ERROR 2
Persona personaBien = new Mujer(); // POLIMORFISMO
Persona personaGuay = new Hombre();// POLIMORFISMO
Persona personaMal = new Persona(); // ERROR 2
```

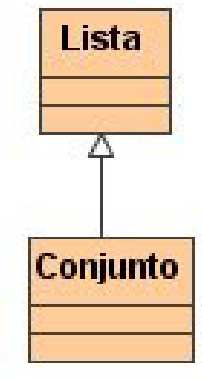
- *ERROR 1: Incompatibilidad de tipos*
- *ERROR 2: Instanciación de una clase abstracta*

- *donde no se contempla que una mujer se convierta en hombre o un hombre en mujer; ni se contempla que alguien sea mujer y hombre a la vez; lo único que se contempla es que una referencia a persona apunte a un objeto de la clase mujer u hombre.*

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo

- **Comportamiento**: cuando se lanza un mensaje a un objeto a través de una referencia polimórfica se ejecuta el método prescrito en la clase del objeto que recibe el mensaje.
- Ej.: En un punto dado, existen listas, en otro punto, existen conjuntos y, en otro punto, pueden existir indiferentemente listas o conjuntos sobre la variable colección que es de tipo Lista.

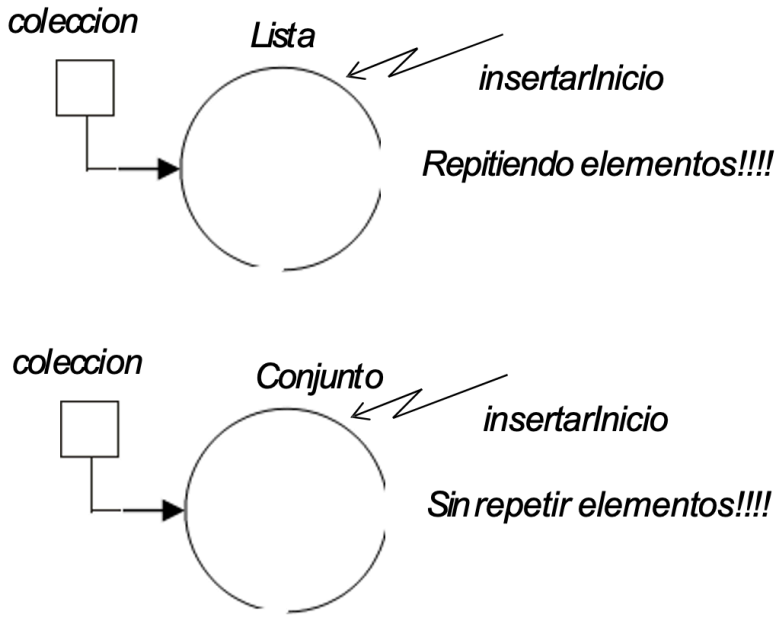



```
Lista coleccion;
```

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo (insertaInicio)

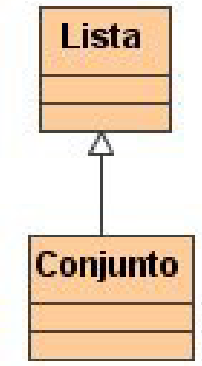
```
class Lista {
  void insertarInicio(String e){ System.out.println("Lista (!repetidos)"); }
}

class Conjunto extends Lista {
  void insertarInicio(String e){ System.out.println("Conjunto(!repetidos)"); }
  void interseccion(){ System.out.println("Intersección"); }
}

public class Main {
  public static void main(String[] a) {
    Lista coleccion = new Lista();
    coleccion.insertarInicio("A");
    coleccion = new Conjunto();
    coleccion.insertarInicio("B");
    // coleccion.interseccion(); // ❌
    new Conjunto().interseccion();     // ✓
  }
}
```



3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo

- **Limitación**: cuando se lanza un **mensaje** a un objeto a través de una referencia polimórfica, éste **debe estar contemplado en el interfaz o el padre** de la clase de la que se declaró la referencia sin contemplar los posibles métodos añadidos en la clase del objeto apuntado.
- Ej.: En un punto dado, existen listas, en otro punto, existen conjuntos y, en otro punto, pueden existir indiferentemente listas o conjuntos

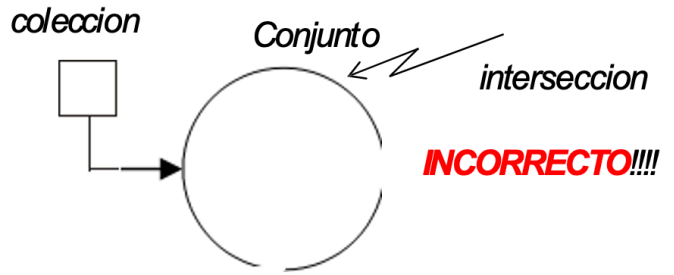


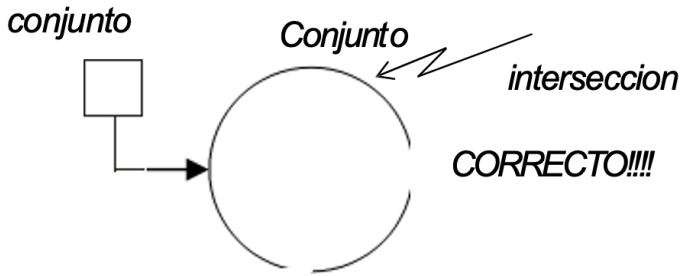

```
Lista coleccion;
```

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo (interseccion)

```
interface Coleccion { void insertarInicio(String e); }

class Lista implements Coleccion {
  public void insertarInicio(String e){System.out.println("Lista (!repetidos)");}
}

class Conjunto implements Coleccion {
  public void insertarInicio(String e){System.out.println("Conjunto(!repetidos)");}
  public void interseccion(){ System.out.println("Intersección"); }
}

public class Main {
  public static void main(String[] a) {
    Coleccion coleccion = new Lista();     coleccion.insertarInicio("A");
    coleccion = new Conjunto();            coleccion.insertarInicio("B");
    // coleccion.interseccion(); // ❌ no está en la interfaz
    new Conjunto().interseccion();         // ✓ correcto
  }
}
```





3.4 POLIMORFISMO Y SOBRECARGA

Variables Polimórficas

- **Enlace estático**: aquel enlace que se puede resolver analizando el código, o sea, en **tiempo de compilación**;
  - Ej.: una variable y su nombre, su tipo, …; una constante y su nombre, su valor, su tipo, …; una expresión y su número de operadores, su tipo,
- **Enlace dinámico**: aquel enlace que NO se puede resolver analizando el código sino que se resuelve en **tiempo de ejecución**;
  - Ej.: una variable y su valor, …; ***una expresión y su valor evaluado que se resuelve en ejecución debido al polimorfismo.***

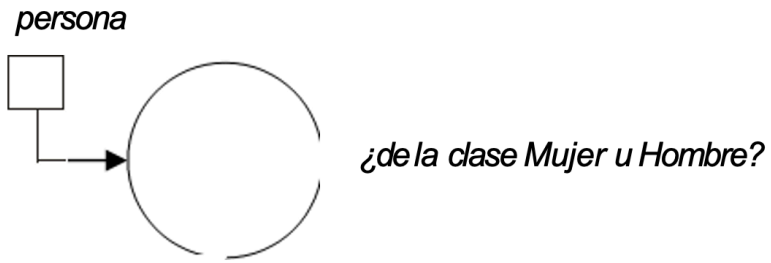
3.4 POLIMORFISMO Y SOBRECARGA

Variables Polimórficas

- **Enlace Polimórfico**: es un enlace dinámico entre una referencia y la clase de objeto apuntado por la referencia;

```
Lista coleccion;
…
… coleccion …
```


```
Persona persona;
…
… persona …
```

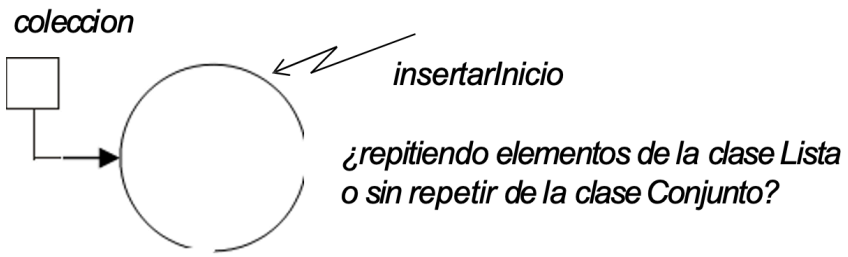


3.4 POLIMORFISMO Y SOBRECARGA

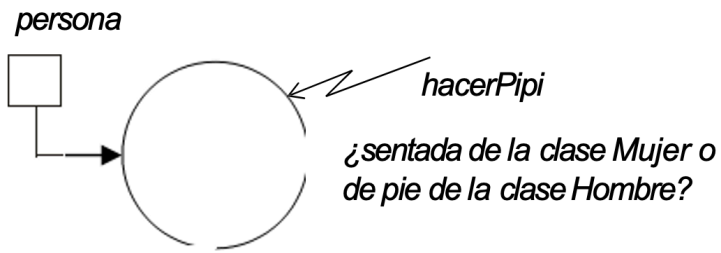
Variables Polimórficas

- **Enlace Polimórfico**: enlace dinámico entre un mensaje y el método que se ejecuta;

```
Lista coleccion;
…
… coleccion.insertarInicio …
```




```
Persona persona;
…
… persona.hacerPipi …
```



3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo vs Sobrecarga

- **Sobrecarga**: es un enlace estático entre entre un mensaje y el método que se ejecuta;

Ej.:

```
class A {
    public void m()
    public void m(A a)
    public void m(B b)
    public A m(B b, C c)
    public B m(A a, C c)
}

class B {
}

class C {
}
```

```
public class Main {
  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();

    a.m(b, c).m();
    // Selecciona m(B,C), devuelve un A -> sobre ese A se llama a m()

    // Ejemplo 2: asignación
    b = a.m(a, c);
    // Selecciona m(A,C), devuelve un B que se asigna a b
  }
```

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo + Sobrecarga

- **Sobrecarga**: es un enlace dinamico entre entre un mensaje y el método que se ejecuta;

```
Ej.:class A {

  public void m() { System.out.println("A.m()"); }
  public void m(A a) { System.out.println("m(A a)"); }
  public void m(B b) { System.out.println("m(B b)"); }
  public A m(B b, C c) {
    System.out.println("m(B,C) -> devuelve A");
    return new A();
  }
  public B m(A a, C c) {
    System.out.println("m(A,C) -> devuelve B");
    return new B();
  }
}
```

```
class B extends A {

  @Override
  public void m() {
   System.out.println("B.m()"); }
  }
}

class C extends A {

  @Override
  public void m() {
   System.out.println("C.m()"); }
}
```

3.4 POLIMORFISMO Y SOBRECARGA

Polimorfismo + Sobrecarga

- **Sobrecarga**: es un enlace dinamico entre entre un mensaje y el método que se ejecuta;

```
Ej.:public static void main(String[] args) {

    A a = new A();
    B b = new B();
    C c = new C();

    a.m();           // A.m()
    a.m(b);          // m(B b)  (sobrecarga, porque la firma es distinta)
    a.m(c);          // m(A a)  (c es un A, no hay m(C), así que usa m(A a))
    a.m(b,c).m();    // m(B,C) devuelve un A → sobre ese A se llama m() → A.m()
    b = a.m(a,c);    // m(A,C) devuelve un B, que redefine m(), pero aquí solo lo asignamos
    b.m();           // B.m() (sobrescritura, ejecuta versión de B)

}
```

3.4 POLIMORFISMO Y SOBRECARGA

Conversión de tipos

Conversión de Variables Dinamicas a Objetos:

- **Conversión ascendente** (*upcast*): cuando se transforma una dirección de un objeto de una clase a una dirección del objeto pero de una clase ascendente (padre, abuela, …) ;
- **Conversión descendente** (*downcast*): cuando se transforma una dirección de un objeto de una clase a una dirección del objeto pero de una clase derivada (hija, nieta, …) ;
- **Cualquier otra conversión produce error en tiempo de ejecución:** conversión de una dirección de un objeto de una clase a otra clase fuera de la rama ascendente o del árbol descendente: tíos, hermanos, clases ajenas a la jerarquía, …

3.4 POLIMORFISMO Y SOBRECARGA

Conversión de tipos

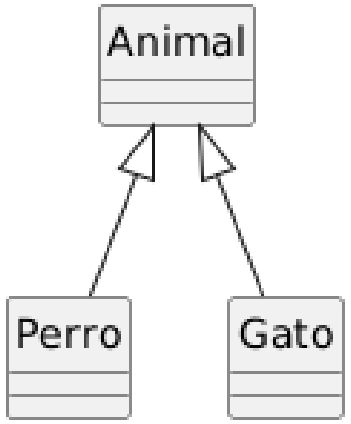
- **Conversión implícita**: por conversión ascendente cuando se asigna una dirección de un objeto de una clase a una referencia declarada a una clase ascendente;
- **Conversión explícita**: por conversión descendente a través del operador de conversión de tipos (cast) (<tipo> <variable>)

```
Animal a = new Perro();     // Implicita
a.sonido();                 // Guau (polimorfismo dinámico)

Perro p = (Perro) a;        // Explicita
p.ladrar();                 // Ladrando
```

3.4 POLIMORFISMO Y SOBRECARGA

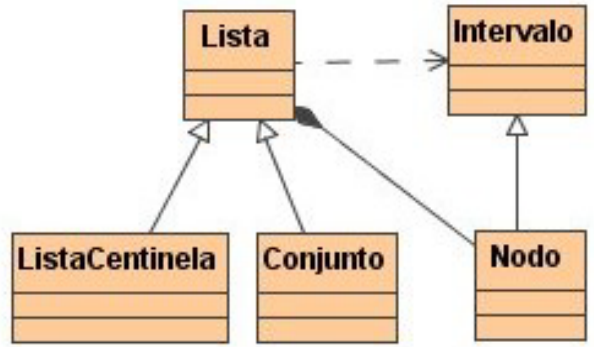
Conversión de tipos

```
class Animal { void sonido() { System.out.println("Animal"); } }

class Perro extends Animal {
     void sonido() { System.out.println("Guau"); }
     void ladrar(){ System.out.println("Ladrando"); }
}

class Gato extends Animal { void sonido() { System.out.println("Miau"); } }

public class Main {
  public static void main(String[] args) {

    Animal a = new Perro();    // Upcast
    a.sonido();                // Guau (polimorfismo dinámico)

    Perro p = (Perro) a;       // Downcast correcto
    p.ladrar();                // Ladrando

    Animal g = new Gato();
    // Perro p2 = (Perro) g;   // ❌ Runtime error: ClassCastException
  }
}
```



3.4 POLIMORFISMO Y SOBRECARGA

Conversión de tipos




```
Lista lista = new Lista();
Conjunto conjunto = new Conjunto()
Lista coleccion = new Conjunto();
Conjunto resultado;
resultado = conjunto.interseccion(coleccion); // ERROR
resultado = conjunto.interseccion((Conjunto) coleccion);
resultado = coleccion.interseccion(conjunto); // ERROR
resultado = ((Conjunto)coleccion).interseccion(conjunto);
((Conjunto) lista).interseccion(conjunto); // ERROR DE EJECUCIÓN
```

3.4 POLIMORFISMO Y SOBRECARGA

Beneficios del polimorfismo

Abstracción

- Entonces es el receptor del mensaje el que determina cómo se interpretará el mensaje y no lo hará el emisor. El emisor sólo necesita conocer qué comportamiento puede desarrollar el otro objeto, no qué clase de objeto cree que es y, por tanto, qué método realiza en cada instante el comportamiento.
- Esta es una herramienta extremadamente importante para permitirnos desarrollar sistemas flexibles. De esta manera, sólo tenemos especificado qué ocurre pero no cómo ocurrirá.
- Mediante esta forma de delegar qué ocurrirá, se obtiene un sistema flexible y resistente a las modificaciones. [Jacobson (creador de los Casos de Uso), 1992]
- ***No se puede preguntar por la clase de un objeto polimórfico!!!***

3.4 POLIMORFISMO Y SOBRECARGA

Beneficios del polimorfismo

Extensibilidad

- Emplear las consultas de tipo durante la ejecución para implantar un enunciado de conmutación – estructura de control de flujo CASE o IF-THEN-ELSE encadenados – en un campo de tipo destruye toda la modularidad de un programa y anula los objetivos de la programación orientada a objetos. También es propensa a errores; […]

- La experiencia demuestra que los programadores que se formaron con lenguajes como Pascal o C encuentran esta trampa muy difícil de resistir. Una razón es que este estilo requiere menos premeditación […]; en este contexto, semejante falta de premeditación muchas veces no es más que una chapuza." [Stroustrup (creador de C++), 1993]
- ***No se puede preguntar por la clase de un objeto polimórfico!!!***