



Enterprise Java Developer



Programación Orientada a Objetos

Herencia

Eric Gustavo Coronel Castillo
gcoronelc@gmail.com
gcoronelc.blogspot.com

Ricardo Walter Marcelo Villalobos
ricardomarcelo@hotmail.com

ÍNDICE

- Objetivo
- Introducción
- Definición
- Características
- Diseño
- Implementación
- Herencia y Constructores
- Acceso Protegido
- Redefinición
- Modificador final
- Clases Abstractas
- Clases Parcialmente Abstractas
- Proyecto Ejemplo

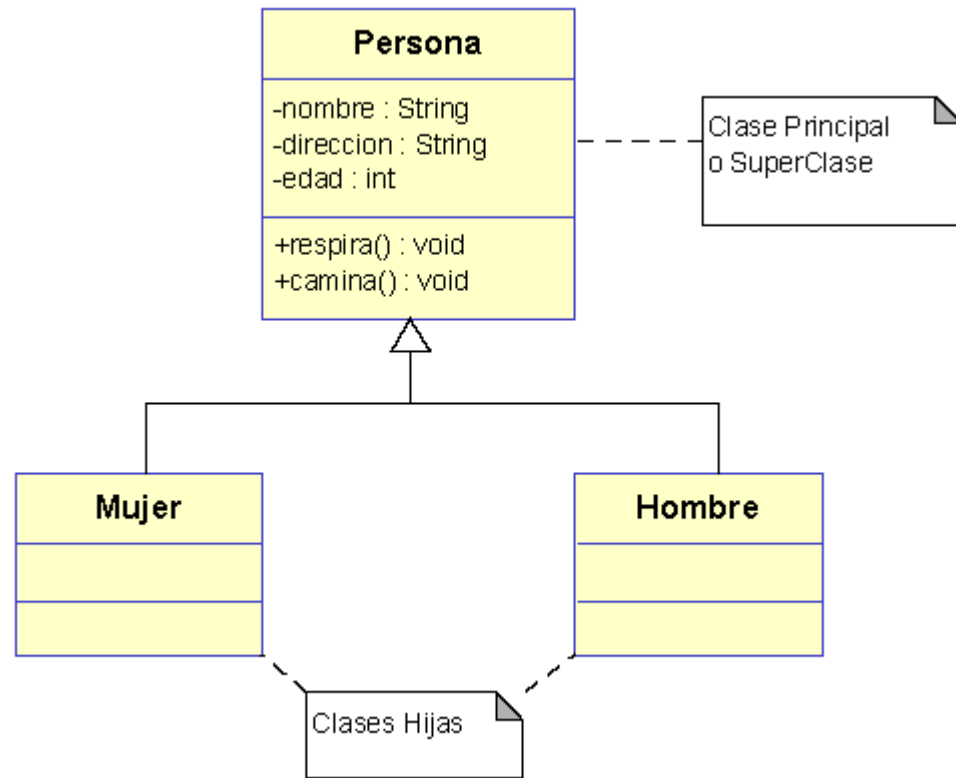
OBJETIVO

Aplicar la herencia para:

- Reutilizar código.
- Extender la funcionalidad de clases (Especialización).
- Aprovechar el poliformismo.

De esta manera:

- Mejoramos la productividad.
- Disminuimos el esfuerzo de mantenimiento.
- Aumentamos la fiabilidad y eficiencia.



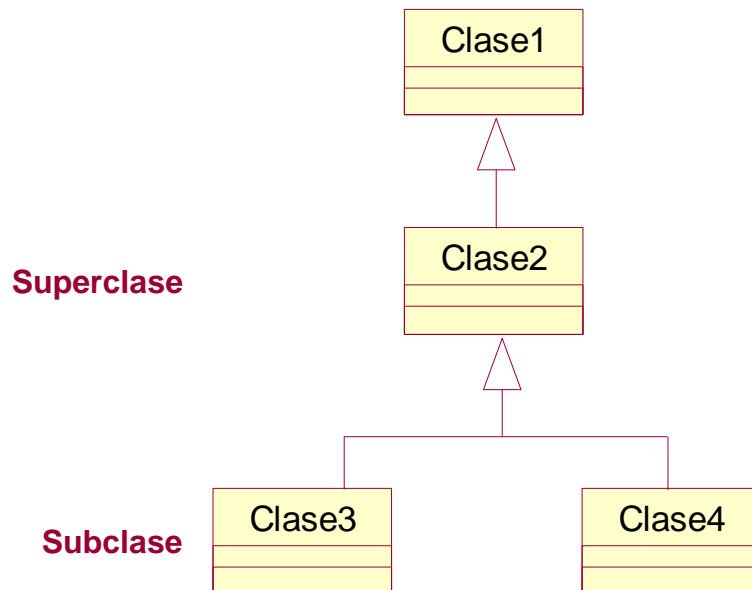
INTRODUCCIÓN

- Las clases no son suficientes para conseguir los objetivos de:
 - **REUTILIZACIÓN:** Necesidad de mecanismos para generar código genérico:
 - Capturar aspectos comunes en grupos de estructuras similares
 - Independencia de la representación e implementación
 - Variación en estructuras de datos y algoritmos
 - **EXTENSIBILIDAD:** Necesidad de mecanismos para favorecer:
 - “Principio abierto-cerrado” y “Principio Elección Única”
 - Estructuras polimórficas.

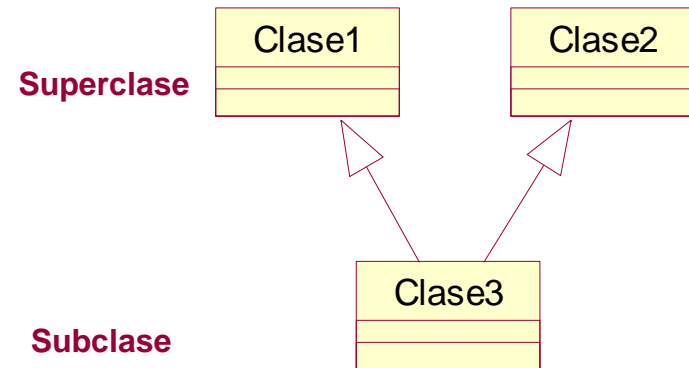
DEFINICIÓN

- La herencia es el mecanismo mediante el cual podemos definir una clase (**Subclase**) en función de otra ya existe (**Superclase**).
- Las subclases heredan los atributos y operaciones de sus superclases.
- Existen dos tipos de herencia (simple y múltiple)

Herencia Simple



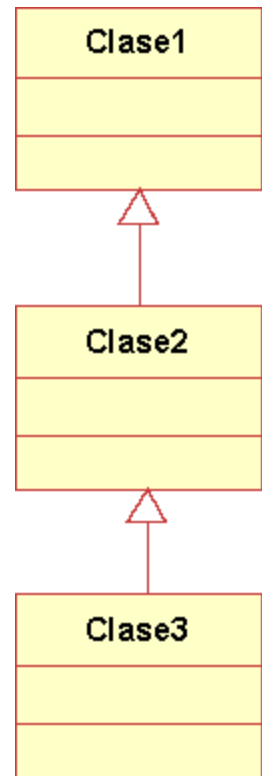
Herencia Múltiple



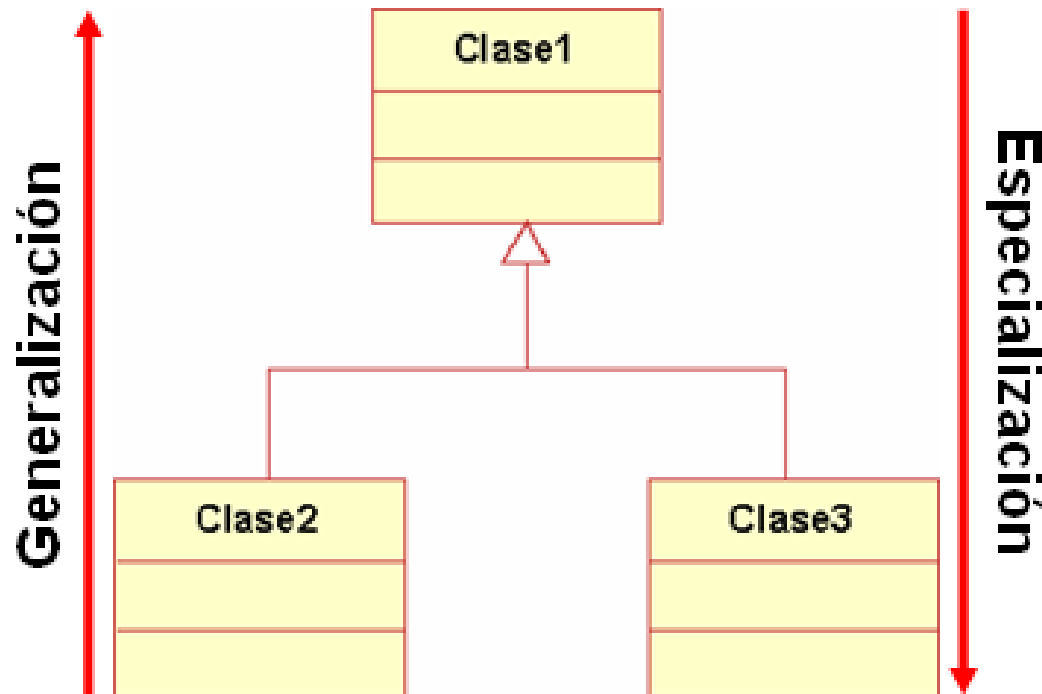
No se puede implementar la Herencia múltiple en Java.

CARACTERÍSTICAS

- Si **Clase2** hereda de **Clase1**, entonces **Clase2** incorpora la estructura (atributos) y comportamiento (métodos) de **Clase1**, pero puede incluir adaptaciones:
 - Clase2 puede añadir nuevos atributos.
 - Clase2 puede añadir nuevos métodos.
 - Clase2 puede redefinir métodos heredados (refinar o reemplazar).
- La herencia es transitiva
 - Clase2 hereda de Clase1
 - Clase1 es la superclase y Clase2 la subclase
 - Clase3 hereda de Clase2 y Clase1
 - Clase2 y Clase3 son subclases de Clase1
 - Clase2 es un descendiente directo de Clase1
 - Clase3 es un descendiente indirecto de Clase1



DISEÑO

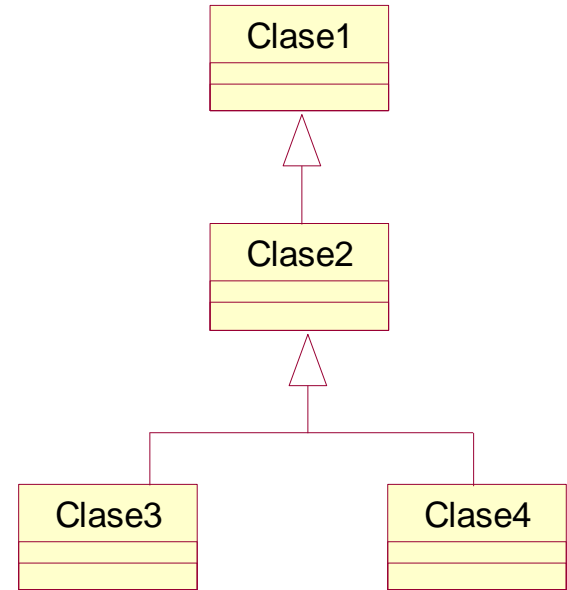


No hay receta mágica para crear buenas jerarquías de herencia.

- **Generalización (Factorización):** Se detectan dos clases con características comunes y se crea una clase padre con esas características.
 - Ejemplo: Libro, Revista → Publicación
- **Especialización:** Se detecta que una clase es un caso especial de otra.
 - Ejemplo: Rectángulo es un tipo de Polígono.

IMPLEMENTACIÓN

```
public class Clase1 {  
  
}  
  
public class Clase2 extends Clase1 {  
  
}  
  
public class Clase3 extends Clase2 {  
  
}  
  
public class Clase4 extends Clase2 {  
  
}
```



Recuerde usar:

this: referencia a métodos del objeto actual.

super: referencia a métodos de la superclase.

HERENCIA Y CONSTRUCTORES

- En Java, los constructores no se heredan.
- Java permite invocar a los constructores de la clase padre dentro de un constructor utilizando la llamada **super(...)**.
- Cuando se aplica herencia, la llamada a un constructor de la clase padre es obligatoria.
- Debe ser la primera sentencia del código del constructor.
- Si se omite la llamada, el compilador asume que la primera llamada es **super()**.

```
public class Clase2 extend Clase1 {  
    public Clase2() {  
        super();  
        ...  
    }  
}
```

ACCESO PROTEGIDO

- Una subclase hereda todos los miembros definidos en la superclase, pero no puede acceder a los miembros privados.
- Para permitir que un método de la subclase pueda acceder a un miembro (variable/método) de la superclase, éste tiene que declararse como **protected**.
 - **private**: visible sólo en la clase donde se define.
 - **Sin modificador (por defecto)**: visible a las clases del mismo paquete.
 - **protected**: características visibles en las subclases y al resto de clases del paquete.
 - **public**: visible a todas las clases.

REDEFINICIÓN

- La redefinición reconcilia la reutilización con la extensibilidad.
- Las **variables** no se pueden redefinir, sólo se ocultan
 - Si la clase hija define una variable con el mismo nombre que un variable de la clase padre, éste no está accesible.
 - La variable de la superclase todavía existe pero no se puede acceder
- Un **método** de la subclase con la misma firma (nombre y parámetros) que un método de la superclase lo está redefiniendo.
 - Si se cambia el tipo de los parámetros se está sobrecargando el método original.
- Si un método redefinido refina el comportamiento del método original puede necesitar hacer referencia a este comportamiento.
 - **super:** se utiliza para invocar a un método de la clase padre:
 - **super.metodo (...) ;**

MODIFICADOR *final*

- Aplicado a una variable lo convierte en una constante.

```
protected final String NOMBRE= "Gustavo Coronel" ;
```

- Aplicado a un método impide su redefinición en una clase hija.

```
public final int suma( int a, int b ) { ... }
```

- Aplicado a una clase indica que no se puede heredar.

```
public final class Clase1 {  
    ...  
}
```

CLASES ABSTRACTAS

- Una clase abstracta define un tipo, como cualquier otra clase.
- Sin embargo, no se pueden construir objetos de una clase abstracta.
- Los constructores sólo tienen sentido para ser utilizados en las subclases.
- ❖ Especifica una funcionalidad que es común a un conjunto de subclases aunque no es completa.
- ❖ Justificación de una clase abstracta:
 - Declara o hereda métodos abstractos.
 - Representa un concepto abstracto para el que no tiene sentido crear objetos.

<i>Clase1</i>
<i>+ metodo1()</i> <i>+ metodo2()</i>

```
public abstract class Clase1 {  
  
    public abstract void metodo1();  
    public abstract void metodo2();  
  
}
```

CLASES PARCIALMENTE ABSTRACTAS

- Contienen métodos abstractos y concretos.
- Los métodos concretos pueden hacer uso de los métodos abstractos.
- Importante mecanismo para incluir código genérico.
- Incluyen comportamiento abstracto común a todos los descendientes.

```
public abstract class Clase1 {  
  
    public abstract void metodo1();  
    public abstract void metodo2();  
  
    public void metodo3() {  
        ...  
    }  
    public void metodo4() {  
        ...  
    }  
  
}
```

<i>Clase1</i>
+ <i>metodo1()</i> + <i>metodo2()</i> + <i>metodo3()</i> + <i>metodo4()</i>

OPERADOR `instanceof`

- Comprueba si el tipo de una variable es compatible con un tipo dado.
 - Es de ese tipo o alguna de sus subclases
- Si no se hace la comprobación, en el caso de que fallara el casting (en tiempo de ejecución) se abortaría el programa.
- No es lo mismo hacer la comprobación con `instanceof` que con el método `getClass` heredado de la clase.

```
if ( variable instanceof Clase ) {  
  
    // Script  
  
}
```

PROYECTO EJEMPLO

- El restaurante "El Buen Sabor" necesita implementar una aplicación que permita a sus empleados calcular los datos que se deben registrar en el comprobante de pago.
- Los conceptos que se manejan cuando se trata de una factura son los siguientes:

– Consumo	100.00
– Impuesto	19.00
– Total	119.00
– Servicio (10%)	11.90
– Total General	130.90
- Cuando se trata de una boleta son los siguientes:

– Total	119.00
– Servicio (10%)	11.90
– Total General	130.90
- Diseñe y desarrolle la aplicación que automatice el requerimiento solicitado por el restaurante.
- Se sabe que el dato que debe proporcionar el empleado es el **Total**.

BIBLIOGRAFÍA

