

Relaciones entre clases en Java

Ricardo Pérez López

IES Doñana, curso 2020/2021



Generado el 21 de abril de 2021 a las 14:03:00

1. Asociaciones básicas
2. Generalización
3. Polimorfismo
4. Restricciones

1. Asociaciones básicas

1.1 Agregación

1.2 Composición

1.1. Agregación

Agregación

- ▶ La **agregación** se consigue haciendo que el objeto agregador contenga, entre sus variables de instancia, una referencia al objeto agregado.
- ▶ Para que sea agregación, la vida del objeto agregado **no** debe depender necesariamente del objeto agregador; es decir, que al destruirse el objeto agregador, eso no signifique que se tenga que destruir también al objeto agregado.
- ▶ Eso implica que puede haber en el programa varias referencias al objeto agregado, no sólo la que almacena el agregador.
- ▶ Lo habitual en la agregación es que la variable de instancia que almacene la referencia al objeto agregado se asigne, o bien directamente (si la variable tiene la suficiente visibilidad) o bien a través de un método que reciba la referencia y se la asigne a la variable de instancia.
- ▶ Ese método puede ser (y suele ser) un constructor de la clase.

► Ejemplo:

```
class Agregador {  
    private Agregado ag;  
  
    public Agregador(Agregado ag) {  
        setAg(ag);  
    }  
  
    public Agregado getAg() {  
        return ag;  
    }  
  
    public void setAg(Agregado ag) {  
        this.ag = ag;  
    }  
}
```

- En la agregación, es frecuente encontrarnos con métodos *getter* y *setter* para la variable de instancia que hace referencia al agregado.

1.2. Composición

Composición

- ▶ La **composición** se consigue haciendo que el objeto compuesto contenga, entre sus variables de instancia, una referencia al objeto componente.
- ▶ Para que sea composición, la vida del objeto componente **debe depender** necesariamente del objeto compuesto; es decir, que al destruirse el objeto compuesto, se debe destruir también al objeto componente.
- ▶ Eso implica que sólo puede haber en el programa una sola referencia al objeto componente, que es la que almacena el objeto compuesto.
- ▶ Por eso, lo habitual en la composición es que el objeto compuesto sea el responsable de crear al objeto componente.
- ▶ Normalmente, no hay *setters* para el componente y, en caso de haber *getters*, deberían devolver una copia del objeto componente, y no el objeto componente original.

2. Generalización

2.1 Declaración

2.2 Subtipado entre tipos referencia

2.3 Herencia

2.4 La clase `Object`

2.5 Visibilidad protegida

2.1. Declaración

Declaración

- ▶ Java es un lenguaje con generalización simple, por lo que una clase sólo puede ser subclase directa de una única clase.
- ▶ La relación de generalización directa entre dos clases se declara en la propia definición de la subclase usando la cláusula **extends**:

```
⟨clase⟩ ::= [public] [abstract | final] class ⟨subclase⟩ extends ⟨superclase⟩ {  
    ⟨miembro⟩*  
}
```

donde **⟨subclase⟩** y **⟨superclase⟩** son los nombres de la subclase directa y la superclase directa, respectivamente.

- ▶ Cuando no se especifica la superclase directa a la hora de definir una clase, el compilador sobreentiende que esa clase es subclase directa de la clase **Object**.

2.2. Subtipado entre tipos referencia

Subtipado entre tipos referencia

- ▶ A partir de ese momento, se introduce en el sistema de tipos una regla que dice que:

$\langle \textit{subclase} \rangle <_1 \langle \textit{superclase} \rangle$

- ▶ Por tanto, se puede decir que el tipo definido por la subclase es un subtipo del tipo definido por la superclase.

2.3. Herencia

Herencia

- ▶ Mediante el mecanismo de la herencia, una subclase hereda ciertos miembros de sus superclases (directas o indirectas), dependiendo de la visibilidad de esos miembros.
- ▶ Los miembros con visibilidad privada no se heredan.
- ▶ Los miembros con visibilidad pública se heredan siempre.
- ▶ Los miembros con visibilidad predeterminada se heredan si la subclase y la superclase pertenecen al mismo paquete.
- ▶ Los miembros con visibilidad protegida se heredan siempre, aunque la subclase y la superclase pertenezcan a paquetes distintos.

2.4. La clase Object

La clase Object

- ▶ La clase `Object` es la raíz de la jerarquía de clases en Java.
- ▶ Toda clase en Java es subclase (directa o indirecta) de `Object`.

2.5. Visibilidad protegida

Visibilidad protegida

- ▶ La visibilidad protegida está pensada para cuando queremos limitar la visibilidad de un miembro de una clase a sus posibles subclases.
- ▶ Pero no debemos olvidar que, en Java, un miembro con visibilidad protegida también puede ser accesible desde cualquier clase que pertenezca al mismo paquete que la clase donde está declarado el miembro.

3. Polimorfismo

3.1 El principio de sustitución de Liskov

3.2 Sobreescritura de métodos

3.1. El principio de sustitución de Liskov

El principio de sustitución de Liskov

- ▶ Recordemos que, por el principio de sustitución de Liskov, se puede usar una instancia de una clase allí donde se espere una instancia de una superclase suya.
- ▶ En un lenguaje de tipado estático como Java, el compilador comprueba que el método que se quiere invocar sobre un objeto es compatible con el tipo declarado para ese objeto.
- ▶ Eso quiere decir que el compilador tiene que determinar, en tiempo de compilación, si un objeto de ese tipo puede responder a la invocación de ese método.

- ▶ Ese tipo será:
 - El tipo estático de la variable si se está intentando invocar un método sobre una variable:

```
Trabajador t; // Declara la variable de tipo «Trabajador» (tipo estático)
t = new Docente(); // «t» contiene una referencia a un objeto
t.despedir(); // El método «despedir» debe ser compatible con «Trabajador»
```

- El tipo definido por su clase si se está invocando el método directamente sobre una instancia de una clase:

```
// El método «despedir» debe ser compatible con «Docente», porque un
// docente es un trabajador:
(new Docente()).despedir();
```

3.2. Sobreescritura de métodos

3.2.1 Sobreescritura y visibilidad

3.2.2 Sobreescritura de variables

3.2.3 `super`

3.2.4 Covarianza en el tipo de retorno

3.2.5 Invarianza en el tipo de los argumentos

3.2.6 Sobreescritura de `equals`

3.2.7 Sobreescritura de `hashCode`

Sobreescritura de métodos

- ▶ Los métodos heredados desde una superclase se pueden sobreescribir.
- ▶ **Sobreescribir o redefinir un método heredado** consiste en definir, en la subclase, un método (el método que *sobreescribe* o que *redefine*) con el mismo nombre y la misma lista de parámetros que el método heredado de la superclase (el método *sobreescrito* o *redefinido*).
- ▶ El tipo de retorno del método que redefine tiene que ser *compatible* con el tipo de retorno del método redefinido, cosa que estudiaremos luego con más profundidad.

Por ahora sólo diremos que si dos tipos de retorno son iguales, entonces son compatibles.

- ▶ En Java, se recomienda (pero no es obligatorio) que el método que redefine se defina usando el decorador `@Override`.
- ▶ Lo normal es que el método que redefine tenga **un cuerpo distinto** al del método redefinido (generalmente, para eso se redefine).

► Ejemplo:

```
class Base {  
    public String hola(String s) {  
        return "Hola, " + s + ", soy la clase Base";  
    }  
}  
  
class Derivada extends Base {  
    @Override  
    public String hola(String s) {  
        return "Hola, " + s + ", soy la clase Derivada";  
    }  
}
```

Sobreescritura y visibilidad

- **La subclase no puede reducir la visibilidad del método redefinido.** Por tanto, el método que redefine debe tener, al menos, la misma visibilidad que el método redefinido, pero nunca menos.

```
class Base {  
    protected String hola(String s) {  
        return "Hola, " + s + ", soy la clase Base";  
    }  
}  
  
class Derivada extends Base {  
    @Override  
    private String hola(String s) {    // Error: reduce la visibilidad  
        return "Hola, " + s + ", soy la clase Derivada";  
    }  
}
```

- Sí que se puede ampliar la visibilidad:

```
class Derivada extends Base {  
    @Override  
    private String hola(String s) {    // Esto sí se permite  
        return "Hola, " + s + ", soy la clase Derivada";  
    }  
}
```

- ▶ Esta restricción tiene sentido si recordamos que un dato de un subtipo debe poder usarse en cualquier sitio donde se espere un dato de un supertipo.
- ▶ Por ejemplo, un Mamífero es una subclase de Animal y, por tanto, debe poderse usar allí donde se espere un Animal.

Es decir: un Mamífero es un Animal.

- ▶ Si pudiéramos sobrecribir un método con otro método menos visible, entonces podríamos tener el problema de que el Mamífero no podría ser capaz de hacer todo lo que puede hacer un Animal.

Sobreescritura de variables

- ▶ Las variables (de instancia o estáticas) de una clase se pueden redefinir sin restricción alguna, simplemente declarando en la subclase una variable con el mismo nombre que la variable heredada de una superclase, sin importar el tipo.
- ▶ Por tanto, las dos variables (la definida en la subclase y la definida en la superclase) pueden tener el mismo nombre pero distinto tipo.
- ▶ Asimismo, las dos variables pueden tener cualquier visibilidad; es decir, la visibilidad de ambas variables es independiente una de la otra.
- ▶ A todos los efectos, son variables completamente diferentes.
- ▶ Es como si la variable de la subclase hiciese sombra a la de la superclase.

- En el siguiente ejemplo, las dos variables de instancia `x` son distintas e independientes:

```
class Base {  
    public int x;           // Esta es una variable...  
}  
  
class Derivada extends Base {  
    protected String x;    // ... y esta es otra distinta  
  
    public String getX() {  
        return x;          // Accede a la «x» de Base  
    }  
}  
  
public class Prueba {  
    public static void main(String[] args) {  
        Base b = new Base();  
        b.x = 4;             // Accede a la «x» de Base  
        Derivada d = new Derivada();  
        d.x = "Hola";        // Accede a la «x» de Derivada  
        System.out.println(d.getX()); // Imprime "Hola"  
    }  
}
```

super

- ▶ En Java, la palabra reservada **super** es una variable especial que se puede usar dentro de una clase y que hace referencia a un objeto de la superclase directa de la clase actual.
- ▶ Cuando se crea una instancia de una clase, automáticamente se crea de forma implícita una instancia de su superclase directa, a la que se tiene acceso a través de la variable **super**.
- ▶ A través de esa variable, se puede invocar métodos de la superclase directa o acceder a variables de instancia de la superclase directa.

- Acceder a una variable de instancia de la superclase directa:

```
class Animal {  
    String color = "blanco";  
}  
  
class Perro extends Animal {  
    String color = "negro";  
  
    void imprimirColor() {  
        System.out.println(color);           // imprime el color de Perro  
        System.out.println(super.color);     // imprime el color de Animal  
    }  
}  
  
public class Prueba {  
    public static void main(String args[]) {  
        Perro p = new Perro();  
        p.imprimirColor();  
    }  
}
```

- Imprime:

negro
blanco

- Ejecutar un método de instancia de la superclase directa:

```
class Animal {  
    void comer () {  
        System.out.println("Comiendo...");  
    }  
}  
  
class Dog extends Animal {  
    void comer() {  
        System.out.println("Comiendo pan...");  
    }  
    void ladrar() {  
        System.out.println("Ladrando...");  
    }  
    void hacer() {  
        super.comer();  
        ladrar();  
    }  
}  
  
public class Prueba {  
    public static void main(String args[]) {  
        Perro p = new Perro();  
        p.hacer();  
    }  
}
```

- Imprime:

Comiendo...
Ladrando...

► Invocar al constructor de la superclase directa:

```
class Animal {  
    Animal() {  
        System.out.println("Se ha creado un animal");  
    }  
}  
  
class Perro extends Animal {  
    Perro() {  
        super();  
        System.out.println("Se ha creado un perro");  
    }  
}  
  
public class Prueba {  
    public static void main(String args[]) {  
        Perro p = new Perro();  
    }  
}
```

► Imprime:

Se ha creado un animal
Se ha creado un perro

- El compilador introduce automáticamente una llamada a **super()** como primera sentencia del constructor de la subclase si éste no incluye ninguna llamada a **super()** o **this()**:

```
class Bicicleta {  
    Bicicleta() {  
        // ...  
    }  
}
```

→ compilador →

```
class Bicicleta {  
    Bicicleta() {  
        super();  
        // sentencias  
    }  
}
```

- Sabemos que, si una clase no tiene constructores, el compilador introduce un constructor por defecto. En ese caso, ese constructor por defecto también llamará a **super()** como primera sentencia:

```
class Bicicleta {  
}
```

→ compilador →

```
class Bicicleta {  
    Bicicleta() {  
        super();  
        // sentencias  
    }  
}
```

► Por ejemplo:

```
class Animal {  
    Animal(){  
        System.out.println("Se ha creado un animal");  
    }  
}  
  
class Perro extends Animal {  
    Perro() {  
        System.out.println("Se ha creado un perro");  
    }  
}  
  
public class Prueba {  
    public static void main(String args[]) {  
        Perro p = new Perro();  
    }  
}
```

► Imprime:

Se ha creado un animal

Se ha creado un perro

Covarianza en el tipo de retorno

- ▶ El tipo de retorno del método redefinido y del método que lo redefine deben ser **compatibles**.
- ▶ La compatibilidad entre el tipo de retorno de un método redefinido y el de un método que lo redefine, se define de la siguiente forma:
 - Si el tipo de retorno es un **tipo primitivo**, los dos tipos de retorno deben ser **el mismo**.

Por ejemplo: si el tipo de retorno del método redefinido es `int`, el del método que lo redefine también deberá ser `int`.

- Si el tipo de retorno es un **tipo referencia**, el tipo de retorno del método que redefine debe ser un **subtipo** del tipo de retorno del método redefinido.

Por ejemplo: si el tipo de retorno del método redefinido es `Number`, el tipo de retorno del método que lo redefine deberá ser un subtipo de `Number` (lo que incluye al propio `Number`).

- Esto se puede resumir diciendo:

En Java, la signatura de un método redefinido es **covariante** en su tipo de retorno con el del método que lo redefine.

- También se puede decir que el tipo de retorno puede cambiar en la misma dirección que la subclase.
- En consecuencia, el tipo de un método redefinido puede sustituirse en el método que lo redefine por otro tipo «más estrecho», es decir, por un subtipo del tipo original.

- ▶ Formalmente, la covarianza es una propiedad que puede tener un tipo compuesto a partir de otros.
- ▶ Sean A y B dos tipos simples en un sistema de tipos, y sean $T\langle A \rangle$ y $T\langle B \rangle$ dos tipos contruidos sobre A y B , respectivamente. Si $<:$ es una relación de orden que indica subtipado, y además $A <: B$, se dice que:
 - T es covariante si $T\langle A \rangle <: T\langle B \rangle$.
 - T es contravariante si $T\langle B \rangle <: T\langle A \rangle$.
 - T es invariante si no es covariante ni contravariante.
- ▶ T se denomina en este contexto un constructor de tipos.
- ▶ Nótese que si T se construye con más de un parámetro, puede ser covariante o contravariante de forma indistinta en cada uno de ellos.
- ▶ Si asumimos que la signatura de un método se puede representar como $S\langle \bar{P}, R \rangle$, entonces podemos decir que entre un método redefinido y el método que lo redefine se cumple que $S\langle \bar{P}, R_1 \rangle <: S\langle \bar{P}, R_2 \rangle$ si $R_1 <: R_2$.

Invarianza en el tipo de los argumentos

- ▶ En cambio, los tipos de los parámetros del método redefinido deben coincidir exactamente con los tipos de los parámetros del método que lo redefine, lo que se puede resumir diciendo:

La signatura de un método en Java es **invariante** en los tipos de sus parámetros.

- ▶ Si los tipos de los parámetros no son idénticos, entonces lo que se está haciendo es una sobrecarga en lugar de una redefinición.
- ▶ Por eso siempre es conveniente usar el decorador `@Override` para asegurarse de que no está ocurriendo ésto accidentalmente.

4. Restricciones

4.1 Clases y métodos abstractos

4.2 Clases y métodos finales

4.1. Clases y métodos abstractos

4.2. Clases y métodos finales