POO: Relaciones jerárquicas en Java

Sitio: Instituto de Tecnología ORT
Curso: Taller de Programación 1 - Plan 2 años 1°C 2020
Libro: POO: Relaciones jerárquicas en Java

Imprimido por: Gentile Fabián
Día: domingo, 19 de abril de 2020, 18:57

Tabla de contenidos

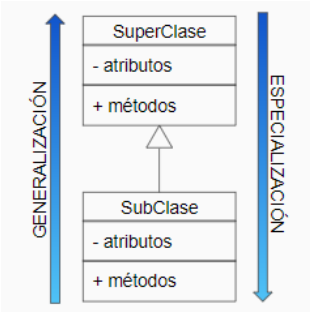
1. Relaciones jerárquicas en Java

Conocida como una de las *4 patas de la Programación Orientada a Objetos*, la Herencia es el mecanismo que permite construir una clase (*subclase* o *clase derivada*) a partir de otra (*superclase* o *clase base*). Gracias a este mecanismo, la subclase hereda los miembros (atributos y métodos) de la superclase.

En Java la herencia es siempre simple, es decir, una clase sólo puede tener una única superclase directa. Así, la herencia define una jerarquía de clases en la que cada clase tiene una única superclase y cero o más subclases. No obstante, cualquier clase puede tener —además de su superclase directa— múltiples superclases indirectas (todas sus *antecesoras*) hasta llegar a la clase *Object*, que en Java es la clase base por excelencia y de la que heredan todas las demás clases.

2. Generalización y Especialización

La relación que se da en entre los miembros del *Árbol Jerárquico* o *Árbol de Herencia* sitúa a las clases más genéricas en el tope de su estructura, mientras que las clases situadas en la parte inferior de la jerarquía son más especializadas que las que están en la parte superior. Cuando se *sube* por el árbol de herencia se habla de Generalización . El camino inverso es la Especialización .



3. Sintaxis

La sintaxis para definir una subclase en Java es:

```
class Subclase extends Superclase
```

Por ejemplo, la declaración para la creación de rectángulos y pentágonos a partir de un polígono *base* se definiría de la siguiente manera:

```
class Rectangulo extends Poligono { … }
class Pentagono extends Poligono { … }
```

En este caso, **Rectangulo** y **Pentagono** comparten el *mismo nivel de Jerarquía*, y ambas *heredan* directamente de **Poligono**.

Si en la definición de una clase no se especifica la cláusula extends se entiende que su superclase es la clase Object , raíz jerárquica de todas las clases en Java.

4. Miembros heredados

- Una subclase hereda todos los miembros de su superclase excepto los *constructores* .
- La herencia de un miembro no implica el acceso al mismo. Las reglas de acceso son:
  - Una subclase no tiene acceso directo a los miembros *private* de su superclase.
  - Un subclase sí tiene acceso directo a los miembros *public* y *protected* de su superclase.
  - Si una subclase pertenece al mismo paquete de su superclase, también tiene acceso a los miembros sin calificar (*default*).

	public	protected	private	default
Desde una subclase del mismo paquete	SI	SI	NO	SI
Desde una subclase de otro paquete	SI	SI	NO	NO

4.1. Acceso a atributos heredados

Ejemplo de acceso a los atributos miembro heredados con distinta visibilidad:

```
/*
* Definición de la superclase con atributos
* con distinto nivel de acceso.
*/
public class Superclase {
    public String atributoPublic = "atributoPublic";
    String atributoDefault = "atributoDefault";
    protected String atributoProtected = "atributoProtected";
    private String atributoPrivate = "atributoPrivate";
}
public class Subclase extends Superclase {
    public String atributoPrueba;
    public Subclase() {
        // Acceso permitido.
        atributoPrueba = atributoPublic;
        atributoPrueba = atributoDefault;
        atributoPrueba = atributoProtected;
        // Acceso no permitido. Se produce un error de compilación:
        // Variable atributoPrivate in Superclase not accessible from Subclase
        atributoPrueba = atributoPrivate;
    }
}
```

4.2. Acceso a métodos heredados

Ejemplo de acceso a los métodos miembro heredados con distinta visibilidad:

```
/*
* Definición de la superclase con métodos
* con distinto nivel de acceso.
*/
public class Superclase {
    public void metodoPublic() {...}
    void metodoDefault() {...}
    protected void metodoProtected() {...}
    private void metodoPrivate() {...}
}
public class Subclase extends Superclase {
    public void pruebaDeAccesoAMetodos() {
        // Acceso permitido.
        metodoPublic();
        metodoDefault();
        metodoProtected();
        // Acceso no permitido. Se produce un error de compilación:
        // No method matching metodoPrivate() found in Subclase
        metodoPrivate();
    }
}
```

4.3. Herencia y Accesibilidad

Accesibilidad y Principio de Encapsulamiento

Si una subclase tuviera acceso a los miembros privados , entonces siempre se podría acceder a los miembros privados de las clases con tan solo crear una subclase, y esta situación violaría el *principio de encapsulamiento* .

Acceso a miembros privados de una superclase

Sin embargo, una subclase puede acceder a los miembros privados de su superclase a través de métodos no privados de la superclase para acceso a sus miembros privados (*API* de acceso).

Composición de una instancia de subclase

La estructura interna de la *instancia* de una subclase se compone de los miembros que ella defina más los que hereda de su superclase.

Propagación de Herencia

Los miembros heredados por una subclase son también heredados por sus propias subclases, *salvo si esta restringe la accesibilidad* al miembro.

Atributos estáticos

Si una clase define un atributo *static* , se mantendrá una única *instancia* para todos los objetos de su misma clase y los de todas sus subclases.

5. Atributos con el mismo nombre

Una subclase puede acceder directamente a los atributos públicos , protegidos y default (si está en el mismo paquete) de su superclase. Pero si una subclase añade un atributo con un nombre coincidente con el nombre de algún miembro heredado y accesible, entonces el miembro heredado queda oculto para la subclase. Ya no podrá acceder directamente a él, aunque sí podrá hacerlo utilizando la palabra reservada *super* .

Ejemplo:

```
/* Definimos la clase ClaseA con un atributo x
* y dos métodos que acceden a él.
*/
public class ClaseA {
    protected int x = 1;
    public int getX() { return x; }
    public int get10X () { return 10 * x; }
}
/*
* Se define la clase ClaseB que hereda de la clase ClaseA
* y agrega un atributo con el mismo nombre (x).
* La clase ClaseB también define un método que accede
* al atributo.
*/
public class ClaseB extends ClaseA {
    private int x = 5;
    @Override
    public int getX() { return x; }
    public int getSuperX() { return super.x; }
}
/*
* Creamos una instancia de ClaseB para comprobar el acceso
* a los miembros propios y heredados.
public class Test {
    public static void main(String [] args) {
        ClaseB objetoClaseB = new ClaseB();
        // Accede al atributo x declarado en ClaseB (devuelve 5)
        System.out.println(objetoClaseB.getX());
        // Accede al atributo x declarado en ClaseA al usar super (devuelve 1)
        System.out.println(objetoClaseB.getSuperX());
        // Accede al atributo x de la ClaseA sin usar super (devuelve 10)
        System.out.println(objetoClaseB.get10X());
    }
}
```

5.1. Control de acceso sobre atributos

El control de acceso de los atributos que una subclase define con el mismo nombre que atributos heredados se puede modificar en cualquier sentido (restringirlos más o menos).

6. Redefinición de métodos

Redefinir un método heredado es volver a escribirlo con el mismo nombre y la misma declaración de parámetros , es decir *la misma firma* . En el ejemplo anterior, *getX()* está definido en ambas clases. Por lo tanto, se dice que *ClaseB.getX()* redefine o *sobreescribe* el método *ClaseA.getX()* .

Una aclaración muy importante en relación a la redefinición de métodos es que, a diferencia de otros lenguajes orientados a objetos, en Java la firma de un método no incluye en ella al tipo de dato de retorno .

6.1. Redefinición y sobrecarga de métodos

Se entiende por sobrecarga de un método a la definición de un método con el *mismo nombre* pero con *distinta cantidad y/o tipo de parámetros* . El caso más común es el de los constructores por defecto y parametrizados.

Cuando una subclase redefine un método de una superclase, el método de la superclase se oculta , pero no las sobrecargas que pudieran existir sobre dicho método en la superclase. Y si el método se redefine en la subclase con distinto número o tipo de parámetros, el método de la superclase no se oculta, sino que se comporta como una sobrecarga).

6.2. La cláusula @Override

Como todas las *anotaciones* que comienzan con *@*, *@Override* cumple un papel sutil pero importante en relación a la *sobreescritura de métodos*. Las anotaciones no son en sí instrucciones Java: no formarán parte del código binario, pero sirven para indicarle al compilador del lenguaje determinadas cosas más allá de él mismo. En este caso, la *anotación @Override* obliga al compilador a comprobar *en tiempo de compilación* que se intenta *sobreescribir* un método heredado.

Sabiendo que sólo se sobreescriben aquellos métodos que coinciden perfectamente en su declaración (con la misma firma), al agregar esta cláusula sobre la declaración del método nos aseguramos de que, efectivamente, el programa compile sólo cuando el método que se está intentando sobreescribir exista en la clase base. Si la firma del nuevo método no coincide con la del método que se intenta sobreescribir se dará un error de compilación. Por ejemplo, si en la clase derivada ClaseB hubiésemos declarado el método *obtenerX()* en vez de *getX()*, y hubiésemos agregado el *@Override*, se daría un error de compilación con el mensaje

"The method obtenerX() of type ClaseB must override or implement a supertype method".

Pero, ¿qué hubiese pasado en caso de no incluir la cláusula *@Override*?

Si el método de la clase derivada coincidiese en firma con el de la superclase (si ambos métodos se llamasen *getX()*) el método de la superclase hubiese sido sobreescrito. Pero *si no coincidieran* en firma, el segundo sería tomado como una sobrecarga y ambos coexistirían en la clase derivada.

De la misma forma, si un método es modificado en su declaración en cualquier nivel del árbol jerárquico, la inclusión de la cláusula *@Override* permitirá evitar accidentes y errores de funcionalidad producidos por el cambio en la firma. De no incluir esta cláusula, se daría el mismo efecto que el explicado en el párrafo anterior: ambos método co-existirían y podrían causar efectos indeseados en la funcionalidad de las clases que deriven de la modificada.

Este es *uno de los errores más difíciles de resolver* ya que el programa falla en su funcionalidad, pero no necesariamente se dará un error de ejecución. Generalmente, si no se realiza una prueba exhaustiva del árbol jerárquico que lo haga visible, el cambio puede llegar a pasar desapercibido y así el error puede subsistir por mucho tiempo sin que nadie se dé cuenta a simple vista.

6.3. Herencia de métodos abstractos

Si una subclase hereda un *método abstracto* tiene que redefinirlo e implementarlo, o bien ser también *abstracta* .

No se pueden instanciar objetos de clases que no estén completamente implementadas. Entonces, si una clase ha implementado todos sus métodos propios pero no ha implementado un *método abstracto heredado* , será tan abstracta como su antecesora .

Aunque se vea como una molestia, es en realidad un punto fuerte de la POO . Volveremos a hablar de esto más adelante, cuando veamos en detalle cómo funciona el *Polimorfismo* .

6.4. Acceso a métodos heredados que han sido redefinidos

Al igual que para acceder a atributos de la superclase homónimos de atributos propios, utilizando la palabra reservada *super* se puede acceder a un método de la superclase que ha sido redefinido en la subclase.

Una subclase que redefina un método heredado sólo tiene acceso a su propia versión y a la de su superclase directa.

6.5. Control de acceso a métodos

No se puede redefinir un método en una subclase y hacer que el control de acceso sea más restrictivo que en la superclase.
En caso de intentarlo se dará un error de compilación con el mensaje

"Cannot reduce the visibility of the inherited method from SuperClase"

- Si un método en la superclase es *public* , solo puede redefinirse como *public* .
- Si un método en la superclase es *protected* , puede redefinirse como *protected* o como *public* .
- Si un método en la superclase es *private* , no tiene sentido hablar de redefinición ya que sólo es accesible desde su propia clase.

7. Herencia y Constructores

Dado que los constructores no se heredan hace falta darles un tratamiento especial.

Los constructores no son *invocables* en cualquier momento. Sólo hay dos formas de hacerlo:

- En forma *implícita*, cuando se hace un *new*.
- En forma implícita o explícita, siempre como primera instrucción dentro de un constructor, y siempre llamándolo con el operador *this* (si se intenta invocar a *otro constructor de la misma clase*) o a través del operador *super* (cuando se intenta invocar a *un constructor de la clase inmediatamente superior* en el árbol de herencia).

Cuando hay al menos dos clases encadenadas en una línea de jerarquía cada *constructor por defecto* invoca implícitamente al constructor por defecto de su superclase. Así los llamados se encadenarán hasta alcanzar a la clase Object. Desde ahí los constructores son ejecutados en cadena hasta llegar otra vez al constructor de la clase instanciada.

El siguiente ejemplo demuestra el llamado implícito entre los constructores. Declarando dos clases (ClaseA y ClaseB, donde ClaseB desciende de ClaseA) se crea una instancia de ClaseB y se muestran los valores de sus atributos:

```
public class ClaseA {
    private int valorDeA;
    public ClaseA() {
        System.out.println("inicializando la clase A...");
        setValorDeA(1);
    }
    public int getValorDeA() {
        return valorDeA;
    }
    private void setValorDeA(int valorDeA) {
        this.valorDeA = valorDeA;
    }
}

public class ClaseB extends ClaseA {
    private int valorDeB;
    public ClaseB() {
        System.out.println("inicializando la clase B...");
        setValorDeB(2);
    }
    public int getValorDeB() {
        return valorDeB;
    }
    private void setValorDeB(int valorDeB) {
        this.valorDeB = valorDeB;
    }
}

public class Test {
    public static void main(String[] args) {
        ClaseB objeto = new ClaseB();
        System.out.println(objeto.getValorDeA());
        System.out.println(objeto.getValorDeB());
    }
}
```

El resultado de la ejecución del programa desde la clase Test emitirá por consola las siguientes líneas:

```
inicializando la clase A...
inicializando la clase B...
1
2
```

Esto demuestra que, aunque la "invocación" fue a través de una instancia de ClaseB, primero se ejecutó el constructor de ClaseA (incluso antes de ejecutar la primera instrucción del constructor de esta última) y recién entonces se siguió con el de ClaseB (asumimos que antes se ejecutó el de Object, pero eso no podemos verlo).

Además, vemos que los atributos inicializados en cada constructor tienen los valores esperados.

7.1. Constructores parametrizados en Herencia

Pero, ¿qué pasa cuando los constructores son *parametrizados* ?

En el siguiente ejemplo se creará y luego mostrará una instancia de TrianguloRectangulo , que hereda directamente de Triangulo y a su vez de Poligono :

```
public class Poligono {
    public Poligono () {
        System.out.println("paso por el constructor de Poligono...");
    }
}

public class Triangulo extends Poligono{
    private int base;
    private int altura;
    public Triangulo(int base, int altura) {
        this.setBase(base);
        this.setAltura(altura);
        System.out.println("terminé de inicializar el Triangulo...");
    }
    public int getBase() {
        return base;
    }
    public void setBase(int base) {
        this.base = base;
    }
    public int getAltura() {
        return altura;
    }
    public void setAltura(int altura) {
        this.altura = altura;
    }
    @Override
    public String toString() {
        return "Triangulo [base=" + base + ", altura=" + altura + "]";
    }
}

public class TrianguloRectangulo extends Triangulo {
    public TrianguloRectangulo(int base, int altura) {
        super(base, altura);
        System.out.println("terminé de inicializar el TrianguloRectangulo...");
    }
}
```

Al ejecutar el programa, veremos que aún sin ser invocado explícitamente , el constructor de Poligono fue ejecutado en primer lugar (siempre luego de Object). Luego se ejecutó el de Triangulo (invocado por el *super()* en el constructor de TrianguloRectangulo) y finalmente se completó la ejecución de el constructor de esta última clase:

```
paso por el constructor de Poligono...
terminé de inicializar el Triangulo...
terminé de inicializar el TrianguloRectangulo...
Triangulo [base=4, altura=6]
```

Aprovechamos aquí el mecanismo de la herencia para utilizar el *toString()* implementado en Triangulo para mostrar el valor de los atributos *base* y *altura* de la instancia.

TIP: Nunca se puede invocar a *super()* si no es como primera instrucción en el constructor. Ni siquiera es posible ponerlo dentro de un *if*. Es la única forma de asegurar de que los atributos se llenarán siempre desde la clase base hasta la instanciada y así también asegurar la *integridad de los datos* .

8. Referencias a superclases y subclases

Una *referencia* no es más que una variable, atributo o parámetro de "tipo clase", cuyo contenido siempre será un objeto o *null* .

¿Cómo juega esto cuando tenemos clases relacionadas entre sí a través de una *línea de jerarquía* ?

Solemos declarar a las variables del mismo tipo del valor que vamos a asignarle, pero esto no solo no es estrictamente necesario, sino que impide que se produzca un mecanismo riquísimo de la POO: el *polimorfismo* , y hablaremos en detalle sobre este mecanismo más adelante.

Una *referencia* a un objeto de una superclase puede apuntar a objetos de cualquiera de sus subclases. Volviendo a uno de los ejemplos iniciales, podemos declarar figuras geométricas (polígonos) que "guarden" objetos de diferente clase, pero siempre derivadas de Poligono :

```
Poligono figura1 = new Rectangulo();
Poligono figura2 = new Pentagono();
```

En estos casos se produce una *conversión ascendente implícita* , sin importar si la relación entre superclase y subclase es directa o indirecta. Por ejemplo, si existiese la clase TrianguloEquilatero que "colgase" del árbol de herencia debajo de Triangulo (la que a su vez hereda de Poligono), podríamos hacer:

```
Poligono figura3 = new TrianguloEquilatero();
```

La conversión entre clases implica que la *interfaz* o *API* visible no es realmente la del objeto instanciado , sino que es la de la variable declarada . Por lo tanto s ólo se tendrá acceso a los miembros públicos de la clase declarada .

8.1. Conversión explícita o casting

¿Qué pasa cuando necesitamos que una variable de tipo subclase reciba el valor de otra de un tipo con mayor jerarquía?

Esto es posible, con algunas restricciones, a través del mecanismo de *enmascaramiento* o *casting*, y más precisamente del *downcasting* (*conversión explícita descendente*).

Cuando se hace *downcasting* se está forzando al lenguaje a *ver* la instancia referida *como si fuese de la clase de la máscara*. De no hacer esto, como la variable originalmente declarada no es válida para la asignación, se daría el siguiente error:

"Explicit cast needed to convert Superclase to Subclase"

Sigamos con los ejemplos:

```
Poligono figura = new TrianguloEquilatero();
TrianguloEquilatero trianguloE = (TrianguloEquilatero) figura;
```

La variable *trianguloE* puede recibir, previo *downcasting*, la referencia guardada en *figura3* sólo porque el objeto al que se tiene acceso (el objeto guardado en *figura3*) es una instancia de su misma subclase.

Antes de seguir con los siguientes ejemplos, y para entender por qué es esto posible, revisemos esta regla:

> Si *Derivada* es subclase de *Ancestro*, *Derivada* es *Ancestro*, pero *Ancestro* no es *Derivada*.

Esto es porque las *generalizaciones* no cumplen con todos los requisitos que hacen falta para ser la clase derivada: le faltarán atributos y/o métodos y/o habrá cambio en la visibilidad de sus miembros, y por lo tanto desde la variable no se puede acceder a los miembros propios de la subclase.

Sin embargo, las *especializaciones* sí cumplen con todos los requisitos de sus ancestros, porque heredan todos sus miembros.

Ahora revisemos las siguientes declaraciones:

```
// La variable figura es de tipo Poligono
// pero guarda una instancia de TrianguloEquilatero.
Triangulo triangulo1 = (Triangulo) figura;
// La variable trianguloE guarda la instancia de un TrianguloEquilatero.
Triangulo triangulo2 = trianguloE;
```

Para comprobar si estas declaraciones son posibles, preguntémonos si lo que tenemos es del tipo de la variable declarada.

- Aunque la variable *figura no es* Triangulo, el objeto contenido en ella sí lo es, porque TrianguloEquilatero desciende de y cumple todos los requisitos para ser Triangulo. Como el contenido es válido para la variable *triangulo1* el *downcasting* es posible.
- La variable *triangulo2* no requiere casting de ninguna clase pues *trianguloE ya es* un Triangulo (porque TrianguloEquilatero desciende de Triangulo).

Pero veamos qué sucede si hacemos al revés:

```
Triangulo unTriangulo = new Triangulo();
// ERROR DE COMPILACION:
// Se produce una Excepción del tipo ClassCastException
TrianguloEquilatero trianguloE = (TrianguloEquilatero) unTriangulo;
```

La asignación anterior es imposible porque Triangulo *no es* TrianguloEquilatero.

8.2. Acceso a los métodos de la instancia

Cuando se accede a un objeto de una subclase por medio de una referencia a su superclase, sólo se pueden invocar los métodos de la superclase. Es decir que los métodos que se pueden invocar sobre un objeto dependen del tipo de variable declarado para referenciarlo, y no de la clase del objeto . Por eso, si queremos acceder a un método que no está declarado en la *interfaz* de la variable, se producirá un error :

```
public class Superclase {
    public void metodoSup1() {...}
    public void metodoSup2() {...}
}
public class Subclase extends Superclase {
    public void metodoSub1() {...}
    public void metodoSub2() {...}
}
public class MainClass {
    public static void main(String [] args) {
        // Creamos una variable de una clase pero le asignamos
        // un objeto de una de sus subclases.
        Superclase referenciaSup = new Subclase();
        // Podemos acceder a los métodos declarados en
        // la superclase.
        referenciaSup.metodoSup1();
        referenciaSup.metodoSup2();
        // Pero no podemos acceder a los métodos de la
        // subclase porque no figuran entre los declarados
        // en la superclase.
        referenciaSup.metodoSub1();
        referenciaSup.metodoSub2();
    }
}
```

El error de compilación que se produce para los dos últimos casos es

"Method metodoSubclase not found in Superclase"

8.3. Acceso a métodos y downcasting

Tal como sucede en la asignación, se puede recurrir al *downcasting* para acceder a un método que existe pero que queda "oculto" al no existir en la *API* de la variable declarada.

Supongamos que la clase Triangulo tiene un método llamado *calcularHipotenusa()*. Éste sería accesible si el triángulo hubiese sido asignado a una variable de su tipo. Pero como fue guardado en un Poligono, el método quedó oculto (existe, pero no se puede acceder a él pues no figura en la "lista de miembros publicados").

Para volverlo visible recurriremos al *downcasting*. Los paréntesis "externos" son usados para que la conversión se limite a *figura*, y no al resultado que retorna el método *calcularHipotenusa():*

```
Poligono figura = new Triangulo();
...
...
float hipotenusa = ((Triangulo) figura).calcularHipotenusa();
```

9. Conclusión

Bajo el paradigma de la Programación Orientada a Objetos , la Herencia es uno de los mecanismos más utilizados crear nuevas clases partiendo de una clase o de una jerarquía de clases preexistente. Este mecanismo permite crear nuevas piezas de código sin alterar estas clases previas, evitando así el rediseño, la reescritura y posterior re-verificación de la parte ya implementada y hace que la nueva clase obtenga todo el comportamiento y, explícita o implícitamente, los atributos de las clases de las cuales hereda (la o las que están sobre ella en la línea de clases o jerarquía de herencia).

De la misma manera, y al heredar el comportamiento y las cualidades de las clases previas en el *árbol jerárquico* , cualquier cambio en el diseño y/o comportamiento en una clase impactará en todas las clases que dependen de ella.

La sobreescritura de métodos , completamente asociada a la POO y a la Herencia, no sólo permiten que nuevas clases implementen métodos abstractos definidos en clases de las que herede, sino que además permite sobreescribir y redefinir comportamientos preexistentes, dando lugar a uno de los efectos más interesantes de la Herencia: el Polimorfismo .