POO: Polimorfismo en Java

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

Imprimido por: Gentile Fabián
Día: domingo, 19 de abril de 2020, 18:58

Tabla de contenidos

1. Introducción

Así llegamos a la *última pata* de las ***4 patas de la Programación Orientada a Objetos***: el ***Polimorfismo***.

Conocido también como ***ligadura dinámica***, ***ligadura tardía*** (***late binding***), el concepto fundamental del ***Polimorfismo*** es "mismo mensaje, distinta implementación" y consiste en implementar o reemplazar el método de una clase con otro método llamado exactamente igual pero escrito en otra clase (derivada de la primera).

Para que se produzca esta ligadura tardía (que efectivamente se produce **en *tiempo de ejecución***, cuando el objeto es asignado a la variable) se requieren algunas cosas:

- Que las clase implicadas tengan declarado (y directa o indirectamente implementado) un método con ***exactamente la misma firma***.
- Que la variable que guarde el objeto sea de la clase de ***mayor jerarquía*** entre las todas las implicadas (la que todos los objetos posibles tengan en común).
- Que no se trate de un método *de clase* (***static***).
- Que no se trate de un método privado (porque no se puede heredar).
- Que no haya sido declarado como ***final***, porque no podrá ser definido.

Si se cumplen estos requisitos, usando **siempre la misma *llamada* o *invocación*** y **sea cual sea el objeto asignado**, **la variable declarada expondrá** a través de su *capa de abstracción* o *API* (*application programming interface*, o *interfaz de acceso programático*) ***distinta implementación***. Así, cuando una variable que contiene una instancia propia o de cualquiera de sus subclases invoca a un método, la versión del método que se ejecutará no será necesariamente la que figura en la clase de la variable declarada, sino la existente en la clase de la instancia referida por la variable. Dicho de otra manera, **la versión del método que será ejecutada depende de la clase del objeto referenciado, no de la variable que lo referencia**.

> **TIP**: **El *Principio de Sustitución de Liskov***
>
> El ***Principio de Sustitución de Liskov*** fue acuñado por **Barbara Liskov** en 1987 durante una conferencia sobre *Jerarquía y Abstracción de datos*. Este principio dice que al sobreescribir un método se debe asegurar que éste mantenga el espíritu original del método, sin alterar ni dejar de cumplir la premisa o responsabilidad declarada inicialmente para el mismo. Según este principio, **una clase derivada no únicamente *es*, sino que *debe comportarse* como la clase base**. Por ejemplo, si hay un método *abrirPuerta()*, una clase derivada no debería hacer que su *abrirPuerta()* abra las ventanas, o prenda la radio. A nivel declarativo y de programación puede estar perfecto, pero no sería correcto. Puede tener otra forma de abrir la puerta, pero no hacer otra cosa.

2. Mecanismo interno de referenciación de métodos en la Ligadura Tardía

¿Cómo se produce la *ligadura tardía y dinámica* que posibilita el polimorfismo?

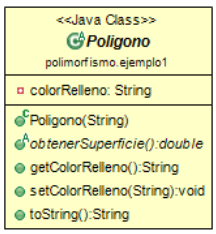
Gracias a los *punteros* o *referencias*: cuando se crea una variable el lenguaje registra en una lista interna cada método declarado para esa clase y espera a que se "guarde" un objeto en la variable a través de una asignación. Cuando se produce la asignación del objeto, se recorre internamente la lista de métodos de la declarados en la variable que pueden ser sobreescritos y se compara con los métodos disponibles para el objeto (propios o heredados). Si la *firma* coincide, el método encontrado en el objeto asignado se asociará a la declaración existente y ***sobreescribirá la referencia*** y ***sustituirá el enlace al método original***, si éste estaba implementado.

3. Polimorfismo, "contratos" y métodos abstractos

Volviendo al ejemplo de los polígonos visto en ***Relaciones Jerárquicas en Java***: ¿cómo haremos para calcular la superficie de distintos polígonos?

Cada polígono tiene su propia forma de calcular su superficie, porque esto depende de su forma y de sus propiedades. De hecho, la clase **Poligono** debe ser ***abstracta***, pues aunque todos los polígonos tengan una superficie para calcular, este cálculo no se puede hacer si no se tiene conocimiento preciso de las propiedades del polígono en cuestión. Entonces, la declaración del método *obtenerSuperficie()* en **Poligono** es necesaria para "establecer un *contrato*" con sus subclases, aún cuando es imposible implementarlo. Declarando este método como ***abstracto*** obligamos a todas las subclases a que lo ***implementen***. De no ser así, las subclases también serían abstractas porque contendrían al menos un método *no implementado* (les "faltaría una parte").

La clase **Poligono** tiene algunos atributos comunes con todos los demás polígonos (por ejemplo el color de relleno) pero sólo puede declarar el método *obtenerSuperficie()*, no implementarlo.

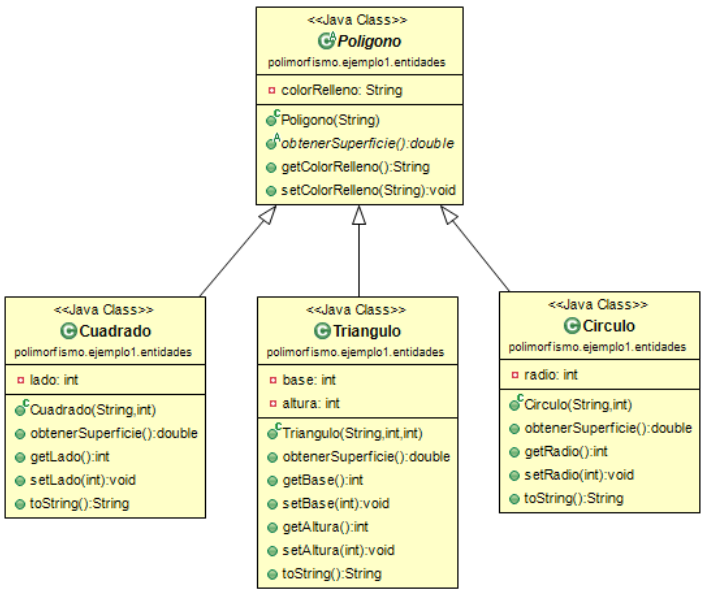


¿Es necesario que esta clase declare el método abstracto? ¿No podrían declararlo directamente cada una de las clases derivadas, ya que a pesar de todo van a tener que implementarlo?

Sí, es necesario. Si el método no es declarado, cualquier variable no la clase **Poligono** no *sabrá* que existe un método con esa firma, y el mismo será inaccesible salvo a través de ***downcasting***. Pero al haber declarado el método en **Poligono** (que es la *clase base* en el ***árbol de jerarquías*** de este ejemplo) no sólo obliga a que las clases derivadas implementen el método ó sean abstractas, sino que el método **ya estará disponible** para acceder a él **sin importar cuál subclase sea la instanciada** sin necesidad de casting alguno.

A partir de **Poligono** "ideal" podemos crear nuevas clases que representen distintos polígonos concretos. Para completar el ejemplo vamos a implementar las clases **Cuadrado** (para calcular su superficie nos hará falta saber el tamaño de uno de sus lados), **Triangulo** (necesitaremos conocer base y altura) y **Circulo** (nos alcanza con el radio).

La definición de **Circulo**, **Triangulo** y **Cuadrado** quedaría así:



Hasta ahora, nada novedoso en cuanto a lo que ya sabemos de Herencia: cada uno de los polígonos definidos hereda el atributo y los métodos relacionados con el color de relleno, y además debieron implementar los setters y getters de sus propios atributos, cada uno por separado, además del método *obtenerSuperficie()*.

Pero veamos qué sucede cuando creamos una colección de polígonos y le agregamos un polígono de cada tipo en un programa como el que sigue (todo el ejemplo completo está disponible en el Aula Virtual como un proyecto de Eclipse, así que no agregamos el código entero aquí):

```
package polimorfismo.ejemplo1;
import java.util.ArrayList;
import polimorfismo.entidades.Circulo;
import polimorfismo.entidades.Cuadrado;
import polimorfismo.entidades.Poligono;
import polimorfismo.entidades.Triangulo;
public class Ejemplo1 {
	public static void main(String[] args) {
		ArrayList<Poligono> poligonos = new ArrayList<>();
		poligonos.add(new Cuadrado("#FF0000", 50));
		poligonos.add(new Triangulo("#00FF00", 50, 50));
		poligonos.add(new Circulo("#0000FF", 25));
		// recorremos la coleccion mostrando cada poligono
		// y su superficie.
		for (Poligono poligono : poligonos) {
			System.out.printf("%s Superficie=%f\n", poligono, poligono.obtenerSuperficie());
		}
	}
}
```

Así, la ejecución del programa dará como resultado:

```
Cuadrado [lado=50, colorRelleno=#FF0000] Superficie=2500.000000
Triangulo [base=50, altura=50, colorRelleno=#00FF00] Superficie=1250.000000
Circulo [radio=25, colorRelleno=#0000FF] Superficie=1963.495408
```

Sin darnos cuenta, hemos usado polimorfismo casi desde el comienzo, cuando empezamos a escribir el método *toString()* en nuestras clases: este método está declarado e implementado inicialmente en la clase *Object*, y si nosotros no sobreescribimos la referencia, al mostrar un objeto ejecutará la versión de *toString()* original.

Recordemos qué mostraría el programa si a las clases que creamos les sacamos la propia implementación de *toString()* y sólo hacemos un " syso " de cada objeto:

```
polimorfismo.ejemplo1.entidades.Cuadrado@15db9742
polimorfismo.ejemplo1.entidades.Triangulo@6d06d69c
polimorfismo.ejemplo1.entidades.Circulo@7852e922
```

La clase *Object* puede mostrar algunas características del objeto instanciado (el nombre completo de su clase y un número en formato hexadecimal que identifica a la instancia) pero no más, porque esto es casi todo lo que sabe de él. Lo que es propio de la instancia permanece oculto.

Por lo pronto, implementemos el método *toString()* en nuestra clase base (Poligono). Sólo podemos referenciarnos a los atributos y métodos registrados en esta clase:

```
@Override
public String toString() {
	return String.format("Poligono [colorRelleno=%s, superficie=%s]", colorRelleno, obtenerSuperficie());
}
```

Al ejecutar el programa, obtenemos:

```
Poligono [colorRelleno=#FF0000, superficie=2500.0]
Poligono [colorRelleno=#00FF00, superficie=1250.0]
Poligono [colorRelleno=#0000FF, superficie=1963.4954084936207]
```

Esto está algo mejor, pero perdimos algo de información respecto a la versión anterior (el tipo de polígono) y más respecto al ejemplo inicial, donde veíamos los datos propios de cada polígono. Veamos qué podemos hacer al respecto.

Por ejemplo, podemos implementar nuevamente los métodos *toString()* de cada clase de polígono, pero esta vez incluyendo los datos que genera el *toString()* de Poligono :

```
@Override
public String toString() {
	return String.format("Circulo [radio=%s, %s]", radio, super.toString());
}

@Override
public String toString() {
	return String.format("Cuadrado [lado=%s, %s]", lado, super.toString());
}

@Override
public String toString() {
	return String.format("Triangulo [base=%s, altura=%s, %s]", base, altura, super.toString());
}
```

Dos detalles a resaltar de esta implementación:

- Sólo se hicieron cambios mínimos en las clases derivadas, el resto permanece inalterado.
- Los *toString()* de las clases derivadas llaman y usan el *toString()* de Poligono . Esto deja en evidencia que *sobreescribir* un método es, en realidad, implementar una nueva versión del método que será asociada a la clase correspondiente *sin destruir* la versión anterior. Ambas permanecen vivas, aunque sólo una (la de la instancia, más específica) es la publicada en la *API* de la variable.

Veamos qué es lo que genera este cambio de implementación. Al ejecutar el programa ahora obtenemos:

```
Cuadrado [lado=50, Poligono [colorRelleno=#FF0000, superficie=2500.0]]
Triangulo [base=50, altura=50, Poligono [colorRelleno=#00FF00, superficie=1250.0]]
Circulo [radio=25, Poligono [colorRelleno=#0000FF, superficie=1963.4954084936207]]
```

Es una mejoría en cuanto a la versión anterior pues ya podemos ver todos los datos de cada instancia. Pero tiene el inconveniente (mínimo) de mostrar el resultado del *toString()* de Poligono "anidado" dentro del *toString()* de cada una de las derivadas.

Hay una alternativa un poco más "elegante", y es hacer que el *toString()* de Poligono "aprenda" a mostrar toda la información de sus derivadas.

Empecemos por el nombre de la clase. Si la clase *Object* puede mostrar el nombre de la clase y el paquete de un objeto instanciado cualquiera, debe haber alguna forma de que lo hagamos nosotros. Esto se logra usando el método *getClass()* , presente en todos los objetos y que devuelve la clase de cada uno.

El siguiente ejemplo muestra el código necesario para que Poligono sepa mostrar tanto el nombre de clase como el paquete de la instancia, sumándolos a los datos que ya mostraba:

```
@Override
public String toString() {
	return String.format("%s [clase=%s, colorRelleno=%s, superficie=%s]",
			getClass().getSimpleName(), getClass().getName(), colorRelleno, obtenerSuperficie());
}
```

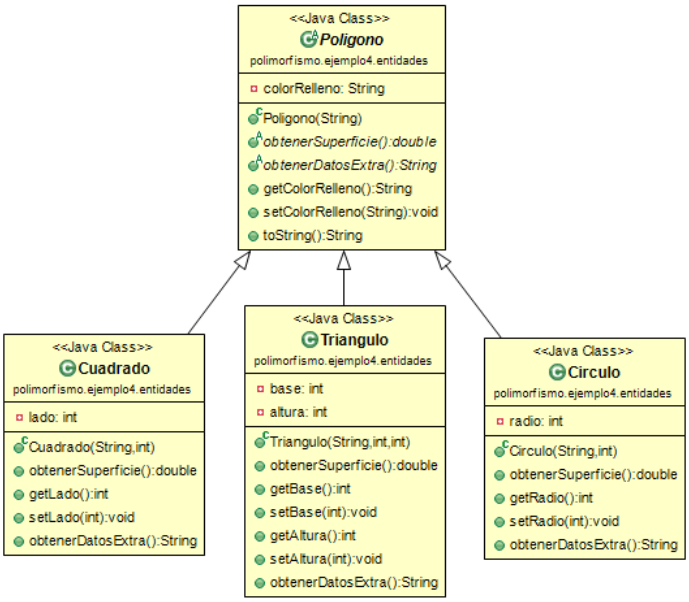
Al correr el programa ahora obtenemos:

```
Cuadrado [clase=polimorfismo.ejemplo2.entidades.Cuadrado, colorRelleno=#FF0000, superficie=2500.0]
Triangulo [clase=polimorfismo.ejemplo2.entidades.Triangulo, colorRelleno=#00FF00, superficie=1250.0]
Circulo [clase=polimorfismo.ejemplo2.entidades.Circulo, colorRelleno=#0000FF, superficie=1963.4954084936207]
```

Sólo nos faltaría obtener los datos propios de cada uno de los objetos.

Si pudimos obtener la superficie del polígono aplicando polimorfismo, podemos obtener estos datos (como string, al menos) usando la misma técnica.

Declaramos en Poligono un nuevo método abstracto *obtenerDatosExtra()* , el que debe ser ser implementado en cada clase derivada para que devuelva en un string esos datos propios que la clase Poligono desconoce. Este método será invocado desde el *toString()* de Poligono , tal como hacemos con *obtenerSuperficie()* .



Los cambios en las clases son:

```
package polimorfismo.ejemplo4.entidades;

public abstract class Poligono {
	...
	// Metodo adicional para obtener los datos propios de cada clase
	// derivada, "invisibles" desde Poligono.
	public abstract String obtenerDatosExtra();
	...
	@Override
	public String toString() {
		return String.format("%s [clase=%s, colorRelleno=%s, superficie=%s, %s]", getClass().getSimpleName(),
			getClass().getName(), colorRelleno, obtenerSuperficie(), obtenerDatosExtra());
	}
}

public class Circulo extends Poligono {
	private int radio;
	...
	@Override
	public String obtenerDatosExtra() {
		return String.format("radio=%s", radio);
	}}

public class Cuadrado extends Poligono {
	private int lado;
	...
	@Override
	public String obtenerDatosExtra() {
		return String.format("lado=%s", lado);
	}
}

public class Triangulo extends Poligono {
	private int base;
	private int altura;
	...
	@Override
	public String obtenerDatosExtra() {
		return String.format("base=%s, altura=%s", base, altura);
	}
}
```

Y con estos cambios, el resultado final es:

```
Cuadrado [clase=polimorfismo.ejemplo4.entidades.Cuadrado, colorRelleno=#FF0000, superficie=2500.0, lado=50]
Triangulo [clase=polimorfismo.ejemplo4.entidades.Triangulo, colorRelleno=#00FF00, superficie=1250.0, base=50,
altura=50]
Circulo [clase=polimorfismo.ejemplo4.entidades.Circulo, colorRelleno=#0000FF, superficie=1963.4954084936207,
radio=25]
```

4. Conclusión

Íntimamente asociado a la Herencia y a la sobreescritura de métodos, el Polimorfismo brinda una forma eficaz de conseguir flexibilidad de implementación en nuestros programas. Su base es el enlace dinámico de métodos en tiempo de ejecución, el cual permite que una variable vea como propias versiones de métodos declarados en ella pero cuya referencia ha sido reemplazada por la de una clase derivada.

Sólo nos resta ver el caso extremo de abstracción: las Interfaces .