Interfaces y polimorfismo

Polimorfismo

El polimorfismo es uno de los pilares de la programación orientada a objetos. Como la palabra lo indica, el polimorfismo es la capacidad de un objeto de funcionar de diversas formas; la capacidad de un mismo objeto de comportarse como otro. Si pensamos nuevamente en el ejemplo de los perros, vimos que un Perro se puede comportar como un Rottweiler, como un Husky o como un Yorkie. Esto se debe a que los tres perros pasan la prueba "es un" Perro: sabemos que son Perros porque hacen, al menos, todo lo que la clase Perro indica que hacen. Si alguno quiere agregar funcionalidad, está bien. Por ejemplo, Husky podría agregar tirarDelTrineo(), pero las operaciones de Perro están ahí. Entonces, un Husky es un Perro.

El polimorfismo no solo se da cuando usamos directamente un objeto que hereda de otro. Sucede lo mismo si el objeto polimórfico es el recibido por un método como parámetro.

```
public class Niño {

   molestarPerro(Perro p) {
      System.out.println("Voy a molestar al perro");
      System.out.println(p.ladrar());
   }

}
```

```
public class Test {
  public static void main(String [] args) {
    Nene n = new Nene();
    Perro d = new Rottweiler();
    Perro c = new Yorkie();
    n.molestarPerro(d);  //se escuchara el ladrido de un
rottweiler
    n.molestarPerro(c);//se escuchara el ladrido de un yorkie
  }
```

```
}
```

Hay algunas ventajas de usar código polimórfico. Aquí, podemos estar seguros de que modificaciones futuras que agreguen nuevas subclases (Beagle, Dóberman, Pitbull, etc.) no deberían afectar el código de la clase Niño ni su funcionamiento. Es decir, agregar subclases de Perro no "rompe" el diseño ni el código existente, todo sigue compilando y corriendo correctamente. Si el código usa Perros, es decir, cualquier objeto que cumpla con la regla es-un Perro, todo funcionará correctamente. Siempre que nuevas razas de Perros que agreguemos al sistema del Niño extiendan de Perro, funcionarán de forma correcta.

Contratos

Hasta ahora, usamos herencia para tener este “comportamiento polimórfico”. Para ver en acción el polimorfismo dependemos de “la buena conducta” del programador. Es decir, dependemos de que quien escriba el código conozca exactamente qué métodos debe sobrescribir y cómo hacerlo, conozca sus firmas, tipos de retorno, excepciones que lanzan, todo.

Dicho esto, si quien debe sobrescribir los métodos no conociera qué métodos sobrescribir, sus firmas, etc., podría heredar de Perro, pero no necesariamente sobrescribir los métodos correctos. Quizás decida agregar algunos, quizá decida sobrecargar los heredados o quizá, por pura casualidad, sobrescribir los métodos correctos. En este caso, no necesariamente podemos hacer uso del polimorfismo. Por ejemplo, si el programador no tuviera información sobre qué métodos debe sobrescribir y por tal motivo no sobrescribirá el ladrar(), nuestro sistema Niño quedaría bastante aburrido: cada perro que el Niño moleste solo dirá “GUAU” y nada más, no notaremos cuando el niño molesta a un perro o a otro.

¿Qué herramientas tenemos para combatir esta situación? Una forma de dar información al programador sobre los métodos que debe sobrescribir para hacer uso del polimorfismo y así introducir nuevos perros al sistema sería mediante la documentación. Este enfoque sería un poco ingenuo, dado que no necesariamente el programador va a leer la documentación. Puede que la lea y no la respete. En cualquier caso, no hay nada que obligue al programador a seguir esas reglas, nada obliga a sobrescribir métodos específicos de Perro para que nuestro sistema Niño funcione correctamente. Podríamos incluir alguna sanción al programador si no cumple con las reglas (¿descontarle algo del sueldo quizás?). Sin embargo, lo que nos interesa es el sistema, ¡posiblemente en tiempo de ejecución!

Es razonable asumir, entonces, que debería existir algún mecanismo que obligue al programador a sobrescribir ciertos métodos específicos, respetando sus firmas, sin depender de tantos factores externos para que nuestro sistema soporte el polimorfismo. Este mecanismo existe y es tan estricto que, de no cumplirse la reglas, hará que el código no compile correctamente. La obligación será dada por los contratos. Para establecerlos, podemos usar clases abstractas o interfaces.

Si sobreescribimos los métodos de la superclase, lo hacemos respetando la firma del o los métodos. Si respetamos la firma de los métodos, entonces respetamos el comportamiento heredado tal cual está, es decir, respetamos el "contrato" impuesto por la superclase. Entonces, **el polimorfismo solo es posible respetando los contratos**.

Como vimos, para asegurarnos de que los contratos sean respetados podemos definir **comportamiento en abstracto**. Esto lo logramos indicando en una clase qué comportamiento hay que llevar a cabo y diferimos a las subclases el cómo se llevará a cabo. Es decir, el contrato impuesto dirá qué hacer, pero cómo se hace no es relevante. En Java, definimos **contratos** mediante clases abstractas, un mecanismo que ya analizamos, y mediante interfaces que analizaremos a continuación.

Clases abstractas

Este concepto ya debería ser familiar, pero a modo de repaso diremos que con ellas podemos definir o “declarar” solo comportamiento para asegurar un contrato entre clases y así hacer uso del polimorfismo**. La intención de una clase abstracta es únicamente declarar comportamiento**. De esta manera, la clase abstracta no se puede instanciar, solo las hijas concretas de esa clase serán las que puedan instanciarse. **Las clases abstractas dictan “qué hay que hacer” y las clases hijas concretas dirán “cómo hay que hacerlo”**. De esta manera, podemos usar como referencia la clase abstracta, "atándole" una instancia de una clase concreta a esta y hacer uso efectivo del polimorfismo.

El problema con los contratos y la herencia

Definir comportamiento abstracto mediante clases abstractas es correcto, pero es importante notar que quita flexibilidad al dejar el comportamiento "atrapado" en una jerarquía. ¿Qué ocurriría si quisiéramos definir comportamiento abstracto pero fuera de una jerarquía? ¿Qué pasaría si quisiéramos definir comportamiento que es tan abstracto que no es parte de la clase Perro ni de una supuesta superclase clase Animal?

Tomemos un método completamente fuera de esta jerarquía: **vestir**. Hoy en día, nuestros peludos amigos son parte de la familia. Es muy común que la gente le compre "ropa" a sus pichichos. Si quisiéramos representar esta realidad en nuestro código, deberíamos incluir el método vestir() en alguna parte de la jerarquía, ¿verdad?

Una solución sería incluir el método vestir() en la clase Perro, pero todas las subclases de perro podrían tener el comportamiento "vestir"... y no todos a los perros los visten. Algunos perros pastores manejan el ganado en el campo; no sería conveniente que tuvieran una vestimenta. Es decir, algunos perros podrían tener el comportamiento vestir() y otros no.

De esta forma, una alternativa sería hacer el método vestir() abstracto e implementarlo solo en donde corresponda. Aun así, cuando llegue a una clase concreta en mi jerarquía de Perros me voy a ver obligado a implementar un método que a lo mejor un perro no debe llevar a cabo. En este punto, comenzamos a tener problemas. Para solucionar el problema, podría dejar la implementación vacía. Sin embargo, como siempre en la programación orientada a objetos, lo que queremos es representar fielmente la realidad. Entonces, si una clase tiene una operación que no hace nada, es como un comportamiento inútil; no debería tener esa operación siquiera. Alternativamente, se podría colocar el método vestir() solo donde corresponda, es decir, solo en los perros que pueda vestir (¿un Pug quizá? ¿Un Beagle?), pero se corre el riesgo de empezar a duplicar código, a repetir una y otra vez una misma forma de vestir a ciertos perros (los pequeños pueden vestirse de una forma, los más altos de otra, pero habrá similitudes entre sí). Adicionalmente, pierdo la posibilidad de utilizar polimorfismo, dado que no hay una superclase que defina que todos los perros deban vestirse.

Compliquemos un poco el panorama: ¿qué ocurriría si tenemos dos superclases? Supongamos una clase Perro y otra Mascota. Mascota podría tener el método vestir() y Perro los otros métodos (ladrar, jugar, etc.). Algunos lenguajes tienen la capacidad de hacer que una clase herede de varias clases a la vez, pero esto trae aparejada cierta complejidad. Para evitar esto, los creadores de Java hicieron que solo soporte la herencia simple.

Agreguemos otro supuesto ahora. No solo a los perros les podemos comprar algo de ropa. Los gatos también llevan ropa. Algunos les compran algún accesorio a sus Loros o Guacamayos. Salgamos un poco de las mascotas y pensemos en algo mucho más cerca… ¡nosotros también nos vestimos! Es decir, **el comportamiento vestir() no necesariamente es exclusivo de los perros**. En un sistema de mascotas, donde la gente interactúa con sus mascotas, podemos observar Aves, Perros, Gatos y Personas con el comportamiento “vestir()”. Queda muy claro que el mismo comportamiento **está compartido por variedad**

de clases, que pertenecen a jerarquías distintas. Entonces, si quiero utilizar el comportamiento polimórfico vestir() tanto en Perros como en Personas, ¿cómo hago? ¡No tengo un ancestro en común!

Solución al problema. Las interfaces

Las interfaces son muy similares a las clases abstractas: se definen con la palabra clave "interface" en vez de "class". Por defecto, **todos sus métodos son abstractos**, por lo cual no es necesario la palabra "abstract" y, al igual que las clases abstractas, los métodos no definen un cuerpo. Asimismo, y al igual que las clases abstractas y por la misma razón, las interfaces no se pueden instanciar.

Para utilizar las interfaces se usa la palabra clave "implements" en lugar de "extends". Al igual que con las clases abstractas, aquellas clases que “implementan” una interfaz quedan obligadas a darle cuerpo a los métodos definidos en la interfaz. De lo contrario, obtendremos un error de compilación. ¿Les suena familiar? ¡Es un contrato!

Veamos un pequeño ejemplo:

```
public interface Perro {
   String ladrar();
}
```

```
public Rottweiler implements Perro {
  //implements me va a obligar a cumplir el contrato
  public String ladrar() {
    return “WOF WOF”;
  }
}
```

Y al momento de ver esta clase en acción:

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

```
    //Perro x = new Perro(); // no es posible. Perro es una
interfaz
    Perro p = new Rottweiler();
    System.out.println(p.ladrar());
  }
}
```

Si observamos detenidamente, la línea Perro p = new Rottweiler() debería resultar muy familiar. Efectivamente, la interfaz Perro dice qué deben hacer los Perros. Es decir, cualquier clase que haga al menos lo que un Perro hace entonces supera la prueba es-un. Rottweiler hace lo que un Perro: ladrar(), y eso mismo nos aseguramos mediante el "implements". Entonces, es susceptible de ser usado del lado derecho del "=".

La pregunta que corresponde hacerse ahora es: ¿para qué sirve todo esto? **Lo que permiten las interfaces es independizarse de las jerarquías**. Las interfaces permiten agregar comportamiento a una clase, pero que no se obtenga desde un nivel superior en la jerarquía. No se "hereda desde arriba", sino que se "enchufa" lateralmente a la jerarquía. Incluso podríamos hasta mezclar ambos mecanismos. Veamos un ejemplo concreto.

Hagamos de Perro una clase nuevamente, podríamos hacerla abstracta, pero mantengamos las cosas sencillas:

```
public class Perro {
  public String ladrar() {
    return "WOF";
  }
}
```

Hagamos una interfaz para todos aquellos que se puedan vestir: Perros, Gatos, Personas, Caballos.

```
public interface Elegante {
   String vestir();
}
```

```
public class Yorkie extends Perro implements Elegante{
  public String ladrar() {
```

```
    return "WIF WIF";
  }
  public String vestir() {
    return "abrigo para perro chico";
  }
}
```

Hasta aquí, no parece nada novedoso, pero analicemos lo siguiente:

```
public class Persona implements Elegante{
  public String vestir() {
    return "vestimenta y sombrero";
  }
}
```

Y aquí viene lo interesante:

```
public class Test {
  public static void main(String[] args) {
    Elegante e = new Yorkie();
    System.out.println(e); // abrigo para perro chico
    e = new Persona();
    System.out.println(e); // vestimenta y sombrero
  }
}
```

Más aún, veamos cómo podemos "mezclar" Perros con Personas, siempre y cuando sean elegantes:

```
public class TestArray {
  public static void main(String[] args) {
    Elegante[] arr = new Elegante[3];
    arr[0] = new Yorkie();
    arr[1] = new Persona();
```

```
    arr[2] = new Yorkie();
    for(Elegante e : arr) {
      System.out.println(e.vestir()); // todos se visten!
    }

  }
}
```

Mandamos a vestirse a objetos que, incluso, no tienen un ancestro en común, no comparten jerarquía, no se heredan entre sí. Pero es la interfaz Elegante la que las hace cumplir con el contrato y, si lo cumplen, entonces tienen el método vestir() y se puede ejecutar sobre ese objeto. Gracias al Dynamic Binding, se ejecutará el método que corresponda: el de Yorkie o el de Persona.

Polimorfismo en Java

Según lo visto hasta aquí, las clases abstractas pueden definir un contrato y, por tanto, ser utilizadas para el polimorfismo. Las interfaces representan comportamiento puro, solo definen qué hacer, incluso, sin depender de ninguna jerarquía.

Entonces, dado que en una clase abstracta podemos mezclar comportamiento abstracto con métodos concretos, no se puede garantizar la definición completa de un contrato. Por esto, decimos que **en Java solamente hay polimorfismo si utilizamos interfaces**.

¿Crees que puedas reemplazar alguna de las clases abstractas que incorporaste anteriormente por una interfaz? ¡Te invitamos a hacer el intento!