8. POLIMORFISMO

El polimorfismo se basa en gran medida en los conceptos aprendidos anteriormente, de hecho, es una de las principales aplicaciones de la herencia y supone el principal motivo de la existencia de las clases abstractas.

Pero antes de definir el polimorfismo, es necesario conocer un fenómeno fundamental sobre la asignación de objetos a variables.

8.1. Asignación de objetos a variables de su superclase

En Java, **es posible asignar un objeto de una clase a una variable de su superclase.** Esto es aplicable, incluso, cuando la superclase es una clase abstracta.

Por ejemplo, dada una variable de tipo Figura:

```
Figura f;
```

Es posible asignar a esta variable un objeto Triángulo:

```
f = new Triangulo(...);
```

A partir de aquí, **puede utilizarse esta variable para invocar a aquellos métodos del objeto que también estén definidos o declarados en la superclase (métodos de `Figura`),** pero no a aquéllos que sólo existan en la clase a la que pertenece el objeto (métodos propios de `Triangulo`).

Por ejemplo, puede utilizarse la variable f para invocar a los métodos `area()` y `getColor()` del objeto de tipo `Triangulo`, pero no para llamar a `getBase()` ni a `getAltura()`:

```
f.getColor(); //invoca a getColor() de Figura
f.area(); //invoca a area() de Triangulo, también definido en la superclase
f.getBase(); //error de compilación, invoca método de Triangulo
f.getAltura(); //error de compilación, invoca método de Triangulo
```

8.2. Definición de polimorfismo

El término polimorfismo significa que hay un nombre (variable, método o clase) y múltiples significados diferentes (distintas definiciones).

Al igual que se forma una jeraquía de clases, el hecho de que las clases definan tipos hace que la herencia dé lugar a una jerarquía de tipos. El tipo que se define mediante una subclase se dice que es un subtipo del tipo definido en su superclase.

El pilimorfismo implica que una variable tiene un tipo estático y un tipo dinámico: el tipo estático es el asociado en la declaración y el tipo dinámico es el correspondiente a la clase del objeto conectado en tiempo de ejecución. Las variables polimórficas tienen **un único tipo estático** y **varios tipos dinámicos**.

Una variable que apunta a un objeto de un supertipo, puede contener objetos de ese supertipo o de cualquiera de sus subtipos.

¿Qué utilidad puede tener asignar un objeto a una variable de su superclase para llamar a sus métodos, cuando eso mismo podemos hacerlo si le asignamos a una variable de su propia clase?

Para responder a esta pregunta, volvamos al ejemplo anterior de las clases `Figura`, `Triangulo` y `Circulo` e imaginemos que, además de éstas, tenemos también la clase `Rectangulo` como subclase de `Figura`. Según lo comentado en el apartado anterior, sería posible almacenar en una variable `Figura` cualquier objeto de sus subclases, es decir, objetos `Triangulo`, `Circulo` o `Rectangulo`:

```
Figura f; //variable de la superclase
f = new Triangulo (...);
f.area(); //Método área de Triangulo
f = new Circulo(...);
f.area(); //Método área de Circulo
f = new Rectangulo(...);
f.area(); //Método área de Rectángulo
```

De lo anterior se desprende un hecho muy interesante: **la misma instrucción `f.area()` permite llamar a distintos métodos `area()`, dependiendo del objeto almacenado en la variable `f`.**

En esto consiste precisamente el polimorfismo, que puede definirse de la siguiente manera:

"La posibilidad de utilizar una misma expresión para invocar a diferentes versiones de un mismo método, determinando en tiempo de ejecución la versión del método que se debe ejecutar".

La posibilidad de uso de variables polimórficas reduce la cantidad de código que es necesario escribir y facilita su reusabilidad. Por ejemplo, en vez de usar 3 bucles distintos...

```
for (Triangulo triangulo : tirangulos) { ... }
for (Circulo circulo : circulos) { ... }
for (Rectangulo rectangulo : rectangulos) { ... }
```

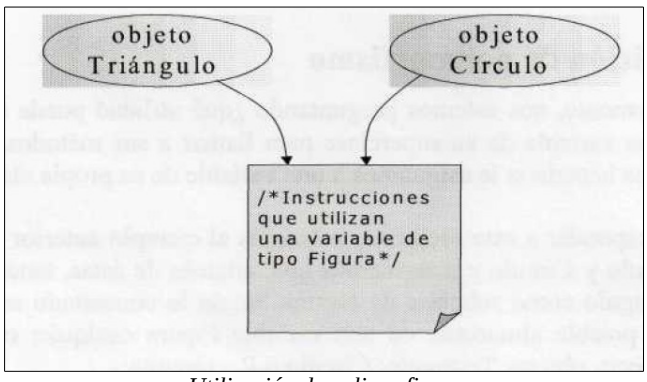
...se puede utilizar un único bucle (haciendo referencia a la superclase edirectamente n vez de a cada subclase individualmente):

```
for (Figura figura : figuras) { ... }
```

8.3. Ventajas de la utilización del polimorfismo

A partir de la definición de polimorfismo y del ejemplo presentado con las figuras, es evidente que la principal ventaja que éste ofrece es la **reutilización de código.** El hecho de que utilizando una variable de una clase pueda escribirse una única instrucción que sirva para invocar a diferentes versiones del mismo método, permitirá agrupar instrucciones de este tipo en un bloque de código para que pueda ser ejecutado con cualquier objeto de las subclases.

La utilidad de asignar un objeto a una variable de su superclase para invocar a los métodos de éste, la respuesta es rotunda: **reutilización de código.**



Utilización de polimorfismo

8.4. Determinación del tipo de variables con `instanceof`

La palabra clave instanceof (todo en minúsculas), sirve para verificar el tipo de una variable. La sintaxis y sus normas de uso son las siguientes:

```
if (objeto instanceof tipo) { ... } else { ... }
```

Solamente se pueden comparar instancias que tengan relación dentro de la jerarquía de tipos (en cualquier dirección), pero no objetos que pertenezcan a distintas jerarquías. Es decir, se pueden comparar `animales` con `mamiferos` pero no se pueden comparar `personas` con `colores`.

Solamente se puede utilizar instanceof asociado a una estructura condicional. No puede ser usado, por ejemplo, directamente en una impresión por pantalla.

```
Circulo circulo1 = new Circulo(3, “negro”);
if (circulo1 instanceof Circulo) { ... } // sintaxis válida
if (circulo1 instanceof Figura) { ... } // sintaxis válida
if (circulo1 instanceof Triangulo) { ... } // Error, ya que figura1 y
Triangulo no pertenecen a la misma jerarquía de tipos
if (circulo1 instanceof Vehiculo) { ... } // Error, ya que figura1 y
Vehiculo no pertenecen a la misma jerarquía de tipos
```

8.5. Conversiones entre clases

Existen dos tipos de conversión: implícita y explicita.

8.5.1. Conversión implícita:

Se puede asignar de forma implícita una referencia a un objeto de una subclase utilizando una variable declarada con el tipo de la superclase. Son tipos compatibles directamente. También se denominan *conversiones ascendentes* o *upcasting*.

```
Figura f;
Triangulo t = new Triangulo(6, 3, ”gris”);
f = t; // asignación válida, pq un triángulo siempre es una figura.
t = f; // error, porque una figura no siempre es un triangulo
```

La variable `f` de tipo `Figura` puede contener la referencia a un objeto de tipo `Triangulo` porque son tipos compatibles (un `Triangulo` siempre es una `Figura`). Pero ojo, sobre el objeto `f` no se pueden invocar los métodos propios de la clase `Triangulo` directamente, tan solo los métodos de la clase `Figura`.

La variable `t` de tipo `Triangulo` no puede contener la referencia a un objeto de tipo `Figura` porque no son tipos compatibles (una `Figura` no siempre es un `Triangulo`).

8.5.2. Conversión explícita:

Se puede forzar una conversión de un objeto de la superclase en un objeto de la subclase utilizando un **casting**:

```
Figura f;
Triangulo t = new Triangulo(6, 3, "gris");
Rectangulo r = new Rectangulo(4, 3, "amarillo");
f = t; // upcasting
t = (Triangulo) f; // casting
t = (Triangulo) r; // error, no se puede forzar la conversión entre tipos de distintas jerarquías (Triangulo y Rectangulo)
t = (Rectangulo) f; // error, no se puede hacer una asignación entre distintos tipos de jerarquías (Triangulo y Rectangulo)
```

Se puede forzar la conversión del objeto `f` en un objeto de tipo `Triangulo` para luego asignárselo a un objeto de tipo `Triangulo`. Pero no se puede forzar la conversión de un objeto de tipo `Rectangulo` a un objeto de tipo `Triangulo` porque pertenecen a distintas jerarquías de tipos. Tampoco se puede asignar un objeto de tipo `Rectangulo` a una variable de tipo `Triangulo` porque porque pertenecen a distintas jerarquías de tipos.

8.5.3. Enlace dinámico

En tiempo de diseño, antes de ejecutarse un programa, el compilador controla los tipos basándose en el tipo estático (declarado). Pero no conoce si en un momento dado una variable está apuntando a un subtipo.

Pero en tiempo de ejecución, Java está constantemente enlazando el método que corresponda en función del tipo dinámico al que apunta una variable. Si el método invocado no está definido en el tipo dinámico de que se trate, Java busca el método en la superclase para tratar de ejecuarlo; si no lo encuentra en la superclase, continúa subiendo niveles en la jerarquía hasta llegar a la clase Object. Si en la clase Object tampoco se encontrase un método que coincidiese, el programa produciría una excepción.

```
Figura f = new Rectangulo(7, 3, "marrón");
Triangulo t;
t = (Triangulo) f; // no se produce error al compilar, pero sí al ejecutar
```

El compilador no muestra ningún error al compilar el código anterior, pero al ejecutarlo se produce una excepción de tipo ClassCastException en la última línea, pues se está tratando de forzar una conversión (casting) de un objeto de tipo Rectangulo en un objeto de tipo Triangulo, y éstos no son tipos compatibles porque no pertenecen a la misma jerarquía de clases.

El compilador no muestra error porque realmente la tercera línea fuerza una conversión de un objeto de la clase Figura a un objeto de la clase Triangulo, lo cual es válido siempre y cuando el objeto representado por dicha Figura sea de tipo Triangulo. En este caso es un rectángulo y la excepción se muestra al producirse el enlace dinámico, que es cuando detecta que la conversión no es posible.