

Este resumen ha sido elaborado para ser leído después de haber asistido a clase: pretende fijar y profundizar en los conceptos ya explicados.

# Polimorfismo

## Introducción

Polimorfismo es la capacidad de que diferentes objetos de una misma jerarquía tengan comportamientos<sup>1</sup> distintos aunque reciban el mismo mensaje<sup>2</sup>.

La idea detrás del polimorfismo es poder diseñar nuestro código teniendo en cuenta que todos las clases que derivan de una clase base heredan los métodos de la clase base.

Consideremos el siguiente ejemplo<sup>3</sup>, que define una clase abstracta `Animal`. `Animal` tiene un método `habla()` que es reemplazado (*override*) en las subclases `Perro` y `Gato`.

```
abstract public class Animal {
    private String nombre;
    public Animal(String nombre) { this.nombre = nombre; }
    public String getNombre() { return nombre; }
    abstract public void habla();
}

public class Perro extends Animal {
    public Perro(String nombre) { super(nombre); }
    public void habla() {
        System.out.println(getNombre()+" : Warf, Guau, etc.");
    }
}

public class Gato extends Animal {
    public Gato(String nombre) { super(nombre); }
    public void habla() {
        System.out.println(getNombre()+" : Miau, Pffff, etc.");
    }
}

public class TestAnimal {
    public static void main(String[] args) {
        // un array de Animal
        Animal[] animales = {
            new Perro("Kujo"),
            new Perro("Fido"),
        }
    }
}
```

---

1 Traducción propia. Definición original: "Polymorphism is the ability of different objects in a class hierarchy to exhibit unique behavior in response to the same message" (*Standard C++ Bible*. AL STEVENS, CLAYTON WALNUM. IDG Books, 2000, p. 440)

2 Recordar que enviar el mensaje `play()` a un objeto `test` quiere decir llamar al método `play()` del objeto `test`. Es decir, `test.play()`;

3 Si no se entiende por qué en `Gato` y `Perro` tenemos que definir constructores que llamen al constructor de la superclase, será bueno repasar el capítulo de herencia.

```

        new Gato("Garfield"),
        new Gato("Patán")
    };
    for( Animal a : animales)
        a.habla();
    } // end main
} // end class

```

La salida:

```

Kujo: Warf, Guau, etc.
Fido: Warf, Guau, etc.
Garfield: Miau, Pffff, etc.
Patán: Miau, Pffff, etc.

```

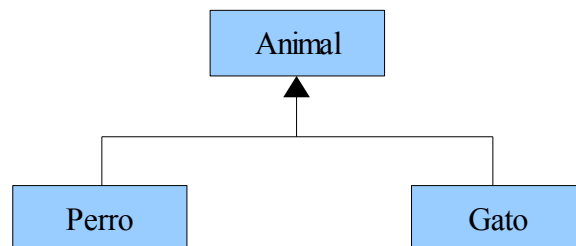
El método main de la clase TestAnimal crea varios objetos Perro y Gato, y los guarda en un arreglo<sup>4</sup>. Luego el for itera los elementos del arreglo, y llama al método habla() de cada elemento.

En este caso, el for se ha diseñado teniendo en cuenta solamente que la clase base tiene un método abstracto habla() y que, por tanto, todas las clases que deriven de ella tendrán un método habla(). TestAnimal funciona con Perro y Gato, pero también funciona con cualquier objeto que derive de Animal.

Cuando decimos a.habla(), estamos enviando el mismo mensaje a objetos distintos de la misma jeraquía de clases (Perro, Gato: depende de que objeto sea a en esa iteración), y cada objeto manifiesta su comportamiento de modo distinto (ladrando o mauyando, siguiendo la analogía).

## Upcasting

Cuando estudiamos herencia, vimos cómo un objeto puede usarse como objeto de su propio tipo o como un objeto del tipo de su clase base. Esto se llama *upcasting*. *Upcasting* es tomar la referencia a un objeto y usarla como una referencia de su clase base. Se llama así porque normalmente se dibujan los árboles de objetos con la clase base arriba.



Siguiendo el ejemplo anterior, cuando decimos Animal a = new Perro("Fido"); estamos haciendo un *upcast*, pues estamos usando un objeto Perro como si fuera del tipo de su clase base.

<sup>4</sup> Puedo almacenar objetos Perro y Gato en un arreglo de Animal porque Perro y Gato son del tipo de su subclase, Animal.

# Binding<sup>5</sup>

## Late binding

Para estar seguros de que estamos entendiendo cómo funciona el polimorfismo, volvamos a considerar este pedazo de código del ejemplo anterior. La línea `a.habla()`, ¿a qué método `habla()` llama? ¿Al de `Perro`, al de `Gato` o al de `Animal`?

```
Animal[] animales = {  
    new Perro("Kujo"),  
    new Perro("Fido"),  
    new Gato("Garfield"),  
    new Gato("Patán")  
};  
for( Animal a : animales)  
    a.habla();
```

La respuesta es “depende”. **Depende de qué tipo de objeto tenga la variable `a` en el momento en que se ejecuta esa línea de código.** Si `a` tiene un objeto `Perro`<sup>6</sup>, entonces llamará a `habla()` de `Perro`; si se trata de un objeto `Gato`, llamará a `habla()` de `Gato`.

Pero, ¿cómo sabe el compilador a cuál de los distintos `habla()` se refiere el `a.habla()` del `for`? No lo puede saber cuando el programa se compila, pues no está claro qué tipo de objeto tendrá `a`<sup>7</sup>: necesita esperar a que el programa **se ejecute**. Más en concreto, necesita esperar a que el `for` se ejecute. En cada iteración del loop, cuando el `for` haya asignado a la variable `a` uno de los elementos del arreglo `animales`, en ese exacto momento y no antes, Java puede determinar qué tipo de objeto es `a`. Si `a` es de tipo `Perro`, llamará a `habla()` de `Perro`; si es de tipo `Gato`, llamará a `habla()` de `Gato`.

Este modo que tiene Java de determinar a qué objeto en concreto debe enviar el mensaje<sup>8</sup> se denomina **late binding**. *Binding* quiere decir, en este contexto, la acción por la cual Java asocia la llamada a un método con un método perteneciente a un objeto concreto. Y se dice *late*<sup>9</sup> porque ocurre durante la ejecución del programa, no durante la compilación.

### En una parte del programa

```
Animal a = new Perro();  
a.habla();
```

binding

### En otra parte del programa

```
class Perro extends Animal {  
    public void habla() { ... }  
}  
  
class Gato extends Animal {  
    public void habla() { ... }  
}
```

Al *late binding* también se le suele denominar *dynamic binding* o *run-time binding*.

<sup>5</sup> “Enlazamiento”, pero seguro hay una mejor traducción.

<sup>6</sup> Más propiamente, habría que decir “si `a` tiene una referencia a un objeto `Perro`”.

<sup>7</sup> Y para hacer las cosas más difíciles para el compilador, `Animal` es abstract, de modo que ni siquiera puede suponer que siendo `a` de tipo `Animal`, se llamará al `habla()` de `Animal`.

<sup>8</sup> Es decir, al método de qué objeto debe llamar.

<sup>9</sup> “tardío”.

## Early binding

Los lenguajes que no están orientados a objetos no tienen *late binding*, por la sencilla razón de que no son polimórficos. En C, siempre que llamamos a un método, el compilador puede determinar a qué método (función) nos estamos refiriendo. Sólo hay una posibilidad porque no hay clases. El C++, en cambio, que es un lenguaje orientado a objetos, permite usar polimorfismo.

Se llama *early binding* a la asociación **durante la compilación del programa** de la llamada a un método y el objeto al que pertenece el método. (vrs. *late binding*, que ocurre durante la ejecución del programa).

En Java todo el *binding* es *late binding*. Siempre que se llama a un método, la determinación de a qué objeto pertenece se lleva a cabo durante la ejecución del programa. Sólo hay tres excepciones: los métodos estáticos (`static`), los métodos finales<sup>10</sup> (`final`) y los métodos privados (`private`). El motivo es que en estos tres casos, el compilador puede saber perfectamente a qué objeto nos estamos refiriendo<sup>11</sup> sin esperar a que el programa se ejecute.

Desde otro punto de vista, un modo de “apagar” el *late binding* (o, lo que es lo mismo, forzar al compilador a usar *early binding*) de un método es declararlo `static`, `final` o `private`. No siempre será posible. Pero esto permite que el compilador genere código ligeramente más eficiente (no tendrá que invertir tiempo durante la ejecución del programa en determinar a qué método tiene que llamar). De todos modos, la realidad es que el tiempo ganado es insignificante<sup>12</sup>.

Conclusión, si decidimos declarar un método `static`, `final` o `private`, que sea por una decisión de diseño, no de eficiencia.

## Algunos ejemplos de polimorfismo

### Ejemplo: Figura

El ejemplo clásico para explicar polimorfismo es el de las figuras (“shapes”). Es apropiado porque una figura es un concepto abstracto, que no existe realmente. Existen figuras concretas: círculos, cuadrados, triángulos, etc. Y es fácil acordarse de que todos los triángulos son figuras, pero no todas las figuras son triángulos.

Definamos una clase abstracta `Figura`, que tenga dos métodos: `draw()` e `erase()`. Los descendientes de `Figura` heredan y harán un `override` de estos métodos, para especializar la clase.

Esquemáticamente se vería así:

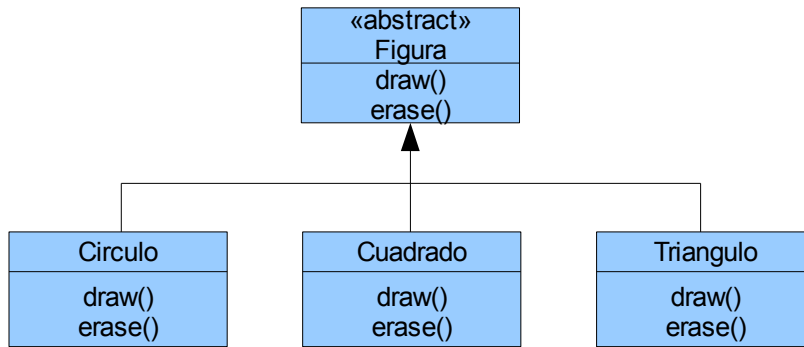
---

10 Esto ya incluye a todos los métodos declarados `private`, pues todos los métodos `private` son implícitamente `final`.

Normalmente se declara un método `final` cuando, siendo público, no queremos que pueda ser reemplazado (`overridden`) en algún subclase (porque implementa una funcionalidad importante para la clase que no debe cambiar, por ejemplo).

11 En el caso de métodos `static`, el método es común a toda la clase, y si se intenta reemplazar el objeto en una clase derivada, el método no es reemplazado, sino que coexisten los dos. En el caso de `final`, sucede algo parecido: al evitar que el método de la clase base pueda ser reemplazado en las subclases, no hay duda de que si usamos el método `final` nos estamos refiriendo al método tal como está definido en la clase base. Por último, los métodos `private` no existen en las subclases, de modo que sólo puede ser usado por otros métodos de la clase base.

12 Además declarar un método `final`, o `private` o `static` nos forzará a cambiar otras partes del programa.



Una posibilidad de implementación de esta jerarquía de clases sería la siguiente<sup>13</sup>:

```

public abstract class Figura
{
    public abstract void draw();
    public abstract void erase();
}

public class Circulo extends Figura
{
    public void draw() {
        System.out.println("Draw de Circulo");
    }
    public void erase() {
        System.out.println("Erase de Circulo");
    }
}

public class Cuadrado extends Figura
{
    public void draw() {
        System.out.println("Draw de Cuadrado");
    }
    public void erase() {
        System.out.println("Erase de Cuadrado");
    }
}

public class Triangulo extends Figura
{
    public void draw() {
        System.out.println("Draw de Triangulo"); }
    public void erase() {
        System.out.println("Erase de Triangulo");
    }
}
  
```

Si escribimos lo siguiente:

```

Figura f = new Circulo();
f.draw();
  
```

<sup>13</sup> No haría falta decirlo a estas alturas, pero si se desea compilar las clases (cosa que está vivamente aconsejada), no hay que olvidar que cada clase pública va en una unidad de compilación independiente. En otras palabras, cada clase pública va en su propio archivo, que tiene como nombre el nombre de la clase y la extensión .java. (mejor no dar nada por supuesto... en BlueJ una unidad de compilación y, por tanto, un archivo, son esas cajas color naranja que aparecen en la pantalla).

¿cuál de los métodos `draw()` se ejecuta?

(piensa unos segundos antes de seguir leyendo)

Se ejecuta `draw()` de la clase `Circulo`. Y el motivo es que es una llamada polimórfica. En el momento de compilación, Java no sabe a qué método `draw()` nos estamos refiriendo, pues `f` es de tipo `Figura`<sup>14</sup>. En el momento de ejecución, Java determina qué tipo de objeto ha sido asignado a `f`, y en base a eso determina a qué subclase pertenece el método `draw()` que debe llamar (*late binding*): en este caso, al `draw()` de `Circulo`.

Puede parecer que realmente el polimorfismo no tiene mucho sentido. Por último, si nosotros mismos estamos asignando a `f` un objeto de tipo `Circulo` (`new Circulo()`), ¿para qué declaramos que `f` es de tipo `Figura`? ¿No sería mejor declarar `f` como una variable de tipo `Circulo`?

El asunto es que no siempre podremos hacerlo. Y no siempre nos interesará hacerlo. Gracias al polimorfismo, podemos escribir código que trabaja en base a los métodos que se han definido en la clase base (la interfaz, en sentido amplio), que funciona con todos los objetos creados a partir de clases derivadas de nuestra clase base (`Circulo`, `Triangulo`, `Cuadrado`, derivados de `Figura`)... y que funcionará con cualquier objeto derivado de la clase base que se le ocurra a alguien crear. Es decir, el polimorfismo nos permite escribir código que funcionará con clases que no han sido escritas todavía<sup>15</sup>.

Para demostrar esta idea, escribamos un pequeño programa que genere figuras al azar, de modo que nosotros mismos no seamos capaces de saber qué tipo de objeto se creará.

Primero, una clase que genera un objeto descendiente de `Figura` aleatorio cada vez que se llama a su método `next()`:

```
import java.util.Random;
public class GeneradorDeFiguras
{
    private Random rand = new Random();

    /**
     * devuelve aleatoriamente un objeto
     * Circulo, Cuadrado o Triangulo.
     */
    public Figura next() {
        // generamos un numero aleatorio entre 0 y 2
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Circulo();
            case 1: return new Cuadrado();
            case 2: return new Triangulo();
        } // end switch
    } // end next
} // end class GeneradorDeFiguras
```

Una clase `Test`, que cree un buen número de objetos aleatorios y llame al método `draw()` de

---

14 Si `f` fuera de tipo `Circulo` entonces no habría duda, ¿verdad? Bueno, quizá nosotros no tendríamos duda, pero Java no se fiaría: quizá a alguien en otra parte del programa se le ha ocurrido derivar una subclase usando `Circulo` como clase base, y nos está pasando un objeto de esa subclase como si fuera un `Circulo`... Recordemos que sólo se usa *early binding* para los métodos `static`, `final` y `private`.

15 Más realista aún, con clases que están escribiendo otras personas de nuestro equipo que se encargan de desarrollar otras partes del mismo proyecto que nosotros.

cada uno:

```
public class Test
{
    public static void main(String[] args) {
        final int numFiguras = 100;
        Figura[] fig = new Figura[numFiguras];
        GeneradorDeFiguras factory16 = new GeneradorDeFiguras();

        for(int i=0; i<fig.length; i++) {
            fig[i] = factory.next();
        }

        for(Figura f : fig) {
            f.draw();
            f.erase();
        }
    }
}
```

## Downcasting

¿Qué sucede si definimos en una de las subclases de *Figura* un método nuevo, que no exista en *Figura*, e intentamos llamarlo desde una variable del tipo de la clase base? El código no funcionará (incluso no compilará), pues sólo se pueden llamar métodos que estén definidos en la clase base.

Añadamos un método *noFunciona()* a la clase *Triangulo*:

```
public class Triangulo extends Figura
{
    public void draw() {
        System.out.println("Draw de Triangulo"); }
    public void erase() {
        System.out.println("Erase de Triangulo");
    }
    public void noFunciona() {
        System.out.println("no funcionará");
    }
}
```

Si declaramos *Figura f = new Triangulo()*, sabemos que si decimos *f.draw()*; se llamará al *draw()* de *Triangulo*. En cambio, *f.noFunciona()*; dará un error, pues *Figura* no tiene definido un método *noFunciona()*.

Hay situaciones, sin embargo, en la que podemos necesitar llamar al método *noFunciona()* de *Triangulo*, aunque la variable que contiene la referencia al objeto sea de tipo *Figura*. El modo de hacer esto es hacer un ***downcast***<sup>17</sup>. No es otra cosa que soplarle al oído al compilador que aunque la variable *f* ha sido declarado de tipo *figura*, realmente contiene un objeto *Triangulo*: no te preocupes, puedes llamar a *noFunciona()* sin problemas.

Explicamos a continuación la sintaxis del *downcast*. Sea *Figura f = new Triangulo()*. Para invocar al método *noFunciona()*, decimos:

---

<sup>16</sup>

<sup>17</sup> Se le llama *downcast* porque implica descender en la jeraquía de la clase: desde la clase base hasta la subclase que corresponda.

```
((Triangulo) f).noFunciona()
```

La expresión `((Triangulo) f)` le señala al compilador que `f` es de tipo `Triangulo`. Y sobre este objeto resultante, de tipo `Triangulo`, llamamos al método `noFunciona()`. Tenemos que usar un par de paréntesis adicionales para que el compilador evalúe primero `((Triangulo) f)`, y recién después de saber que `f` es de tipo `Triangulo` intente llamar a `noFunciona()`. Si omitimos la paréntesis más exteriores, la expresión sería `(Triangulo) f.noFunciona()`. El compilador evaluaría primero `f.noFunciona()`, y luego intentaría convertirlo en un objeto (de tipo `Triangulo`)<sup>18</sup>.

Al hacer un *upcast* se pierde información acerca del objeto de la subclase: las diferencias específicas que tenía con su clase base quedan “ocultas”, y sólo permanecen expuestos los métodos que estaban ya declarados en la clase base (hayan sido reemplazados o no). En este sentido, el *upcast* es siempre seguro, porque la clase base nunca tiene más métodos que las clases que derivan de ella. En cambio, lo contrario no siempre se cumple: puede ser que una subclase sólo reemplace métodos de la clase base, pero nadie nos garantiza que no añadirá otros métodos más (que, perdón por la insistencia, no están en la clase base).

Cuando hacemos un *downcast*, antes de llamar al método sobre el objeto de la subclase, Java siempre verifica que el objeto que recibe es efectivamente del tipo que nosotros hemos dicho (en el caso del ejemplo, que `f` es de tipo `Triangulo` y no `Cuadrado`, `Circulo` o `Figura`). Si no es del tipo esperado, se genera un `ClassCastException`<sup>19</sup>.

Un uso bastante común del *downcast* es recibir de un método un objeto de tipo `Object`, y convertirlo al tipo de objeto adecuado. Por ejemplo, cuando se almacena un objeto en un `ArrayList`, el `ArrayList` lo trata como si fuera un objeto de tipo `Object` (el más general).

Podemos toquetear un poco la clase `Test` del ejemplo anterior, de modo que use `ArrayList`<sup>20</sup>:

```
import java.util.ArrayList;
public class TestConArrayList
{
    public static void main(String[] args) {
        final int numFiguras = 36;
        1 → ArrayList fig = new ArrayList();
        GeneradorDeFiguras factory = new GeneradorDeFiguras();

        for(int i=0; i<numFiguras; i++) {
            fig.add(factory.next()); 2 ←
        }

        3 → for(Object f : fig) {
            ((Figura) f).draw(); 4 ←
            ((Figura) f).erase();
        }
    }
}
```

18 Lo cual generaría un error de compilación, pues mientras que el compilador piense que `f` es de tipo `Figura`, no sabrá encontrar el método `noFunciona()`, pues no está declarado en `Figura`.

19 Esta no es una excepción que haya que atrapar con `try/catch`. Pero en el momento de la compilación Java nos advertirá que estamos usando código poco claro.

20 `ArrayList` es una clase de `java.util` que funciona como un array, pero que crece conforme se añaden elementos a la lista (los arrays, en cambio, una vez creados, no pueden cambiar de tamaño). Además del constructor `ArrayList()`, tiene, entre otros, el método `add(Object o)`, que añade un elemento a la lista. Y cuando devuelve uno de los objetos almacenados, lo hace como un objeto de tipo `Object`. Más adelante estudiaremos en detalle esta clase.

```
    }  
}
```

Explicuemos los puntos saltantes:



(1) Declaramos la variable `fig` de tipo `ArrayList`, y le asignamos una referencia a un objeto de tipo `ArrayList`, que hemos creado usando `new ArrayList()`. Ahora `fig` es una clase que puede almacenar objetos como si fuera un array, pero no tenemos que preocuparnos del tamaño del arreglo.

(2) Creamos `numFiguras` objetos aleatorios (`Circulo`, `Triangulo`, `Cuadrado`), y los añadimos al `ArrayList` usando `fig.add(Objeto o)`. Aquí lo interesante es que `ArrayList` acepta los elementos como objetos de tipo `Object`: no se entera de que estamos almacenando un objeto descendiente de `Figura`<sup>21</sup>.

(3) Iteramos uno a uno los objetos del `ArrayList fig`. El objeto visitado en cada iteración se guarda en la variable `f`. No podemos definir `f` como de tipo `Figura`, pues el `ArrayList` devuelve elementos de tipo `Object`. El código no compilaría.


(4) Hacemos un `downcast` para indicarle al compilador que `f` es de tipo `Figura`, y llamamos `draw()` e `erase()` del objeto resultante. (habíamos almacenado en el `ArrayList` objetos descendientes de `Figura`, pero realmente no sabemos si el objeto que tiene `f` en esta iteración concreta del `for` es de tipo `Circulo`, `Cuadrado`, `Triangulo`... tampoco importa mucho: voy a llamar `draw()` e `erase()`, que están definidos en la clase base, y gracias al polimorfismo Java se encarga de encontrar el método correcto)

También podríamos haber escrito el `for` del (3) de la siguiente manera:

```
for(int i=0; i<fig.size()22; i++) {  
    Figura f = (Figura) fig.get(i)23;   
     f.draw();  
    f.erase();  
}
```

En este caso, hacemos el *downcast* en (5), cuando asignamos el resultado de `f.get(i)`, que es de tipo `Object`, a `Figura f`, y luego usamos la llamada polimórfica en (6).

Aunque no es parte del curso, ni se ha visto en clase, vale la pena comentar que la versión 1.5 de Java introduce el uso de *Generics*<sup>24</sup>. Dicho rápido y sin profundizar más en el asunto, usando generics podemos definir el `ArrayList` del código anterior especificando qué tipo de objeto se va a almacenar. Esto nos evita tener que usar un *downcast* cuando recuperamos un elemento del `ArrayList`. En nuestro caso, quiere decir que el `ArrayList fig` guardaría y devolvería objetos de tipo `Figura`, en vez de objetos genéricos de tipo `Object`.

```
import java.util.ArrayList;  
public class TestConArrayListUsandoGenerics  
{  
    public static void main(String[] args) {  
  
        final int numFiguras = 36;  
  
         ArrayList<Figura> fig = new ArrayList<Figura>();  
  
        GeneradorDeFiguras factory = new GeneradorDeFiguras();
```

21 Y esto funciona gracias al polimorfismo.

22 `int size()` devuelve el número de elementos del `ArrayList`.

23 `Object get(int i)` devuelve el elemento `i` del `ArrayList`.

24 Además, cualquier chiquillo de 11 años la agarraría al vuelo.

```

        for(int i=0; i<numFiguras; i++) {
            fig.add(factory.next());
        }

        for(Figura f : fig) {
            f.draw();
            f.erase();
        }
    }
}

```

Como se ve en (1), declaro el ArrayList indicando entre < > el tipo de objeto que pienso almacenar, uso la misma sintaxis en el constructor:

```
ArrayList<Figura> fig = new ArrayList<Figura>();
```

La ventaja es evidente: en el for puedo declarar f como objeto de tipo fig que es, y puedo llamar f.draw() y f.erase() sin tener que hacer un *downcast*<sup>25</sup>.

### **Ejemplo: Instrumento**

Consideremos el siguiente ejemplo<sup>26</sup>: tenemos una clase abstracta Instrumento, que define dos métodos abstractos: play(Nota n) y ajustar(). De esta clase base derivan dos subclases de Instrumentos: DeViento y DePercusion, y de estas dos algunos instrumentos: Flauta y Oboe de DeViento; Tambor de DePercusion. Todas estas subclases implementan play(Nota n) y ajustar(), y además el método toString() que devuelve el nombre de la clase. Para representar Nota n, usaremos un enum<sup>27</sup>.

```

/* Nota.java */
public enum Nota
{
    DO, RE, MI, FA, SOL, LA, SI
}

/* Instrumento.java */
public abstract class Instrumento
{
    abstract public void play(Nota n);
}

```

25 Deja también de aparecer la advertencia que mostraba el compilador con el código sin *generics*: “TestConArrayList.java uses unchecked or unsafe operations. Recompile with -Xlint:unchecked for details.”. Esta advertencia sólo a partir de Java 1.5, precisamente por la introducción del uso de *generics*.

26 El ejemplo original es de *Thinking in Java, 4<sup>th</sup> Edition*. BRUCE ECKEL. 2006. Prentice Hall, p. 287 y ss. Ha sido modificado para explicar los puntos que nos interesan.

27 enum: *enumerated type*. Se trata de un tipo de datos que contiene un conjunto cerrado de valores nominales. Podemos tratar esos nombres como si fueran componentes normales de un programa. Si declaramos

```

public enum Dia {
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}

```

podemos usar en nuestros programas Dia.Lunes, Dia.Martes, etc. como si fueran valores. Además se cumple que Dia.Lunes < Dia.Martes < Dia.Miercoles ... Podemos decir: Dia d = Dia.LUNES; System.out.println(d); y obtenemos LUNES.

No vamos a explicar más aquí que esto. El enum nos ahorra estar definiendo constantes relacionadas entre sí, como tendríamos que hacer usando #define en C: LUNES = 0; MARTES = LUNES + 1; MIERCOLES = MARTES + 2, etc.

```

        public abstract void ajustar();
    }

    /* DeViento.java */
    public class DeViento extends Instrumento
    {
        public void play(Nota n) {
            System.out.println("DeViento.play(): " + n);
        }
        public void ajustar() {
            System.out.println("DeViento.ajustar()");
        }
        public String toString() { return "DeViento"; }
    }

    class Flauta extends DeViento {
        public void play(Nota n) {
            System.out.println("Flauta.play(): " + n);
        }
        public void ajustar() {
            System.out.println("Flauta.ajustar()");
        }
        public String toString() { return "Flauta"; }
    }

    class Oboe extends DeViento {
        public void play(Nota n) {
            System.out.println("Oboe.play(): " + n);
        }
        public void ajustar() {
            System.out.println("Oboe.ajustar()");
        }
        public String toString() { return "Oboe"; }
    }

    /* DePercusion.java */
    public class DePercusion extends Instrumento {
        public void play(Nota n) {
            System.out.println("DePercusion.play(): " + n);
        }
        public void ajustar() {
            System.out.println("DePercusion.ajustar()");
        }
        public String toString() { return "DePercusion"; }
    }

    class Tambor extends DePercusion {
        public void play(Nota n) {
            System.out.println("Tambor.play(): " + n);
        }
        public void ajustar() {
            System.out.println("Tambor.ajustar()");
        }
        public String toString() { return "Tambor"; }
    }

    /* Musica.java */
    public class Musica
    {
        static void afinar(Instrumento i) {
            i.play(Nota.DO);
        }
    }

```

```

static void afinarTodos(Instrumento[] instrumentos) { ← 2
    for (Instrumento i: instrumentos) {
        afinar(i);
    }
}

public static void main(String[] args) {
    3 → Instrumento[] orquesta = {
        new DePercusion(),
        new DeViento(),
        new Flauta(),
        new Oboe(),
        new Tambor()
    };
    afinarTodos(orquesta);
}
}

```

Lo interesante sucede en la clase Musica. Musica define dos métodos: `afinar` (1) y `afinarTodos` (2), que reciben como parámetro un objeto de tipo `Instrumento`. (Un arreglo de `Instrumentos`, en el caso de `afinarTodos`).

`afinar` no sabe exactamente qué subclase de `Instrumento` recibirá como parámetro cuando se ejecute el programa: sólo sabe que ese parámetro deriva de `Instrumento` y, por tanto, tendrá dos métodos: `play(Nota n)` y `ajustar()`. Gracias al polimorfismo, `ajustar` puede llamar al método `play` del objeto `i`, y dejar que Java se encargue de durante la ejecución del programa de llamar al método `play` apropiado<sup>28</sup>.

Algo similar sucede con `afinarTodos`: los elementos del arreglo `Instrumento[] instrumentos` serán distintos objetos de los descendientes de `Instrumento`. Por tanto, sabemos que tienen dos métodos (quizá otros más, pero esos dos seguro): `play(Nota n)` y `ajustar()`.

En `main` declaramos un arreglo de objetos de tipo `Instrumento` (3), donde puedo guardar objetos que desciendan de `Instrumento`. Inicializamos el arreglo directamente usando `new Flauta()`, `new Oboe()`, etc., y paso el arreglo como parámetro de `afinaTodos`.

## Refactoring

El ejemplo de los instrumentos tiene bastante código redundante. Se ha escrito así para mostrar cómo funciona el polimorfismo<sup>29</sup>. Pero como no queremos que alguien vaya a pensar que es así como enseñan a diseñar programas en la Universidad de Piura, vamos a mejorar nuestro código. Este proceso se llama *refactoring*.

Podemos notar<sup>30</sup>, por ejemplo, que en cada clase estamos repitiendo la implementación de `play` y de `ajustar`. Si pudiéramos escribir `play` y `ajustar` de modo que automáticamente sepa imprimir el nombre de su clase, podríamos pasar estos dos métodos a la clase base, y sacarlos de las subclases. Esto se puede hacer usando `this`:

```
public abstract class Instrumento
```

28 Aunque debería estar claro a estas alturas... si `afinar` recibe un objeto `Flauta`, entonces llamará a `play()` de `Flauta`.

29 Si los métodos de cada instrumento hicieran algo realmente útil, en vez de imprimir solamente un mensaje, el ejemplo sería más interesante. Pero complicaríamos la explicación.

30 Sobre todo si, como está tan recomendado, estamos copiando los ejemplos en BlueJ y probamos a compilarlos.

```

{
    public void play(Nota n) {
        System.out.println(this31+".play(): " + n);
    }
    public void ajustar() {
        System.out.println(this+".ajustar()");
    }
}

public class DeViento extends Instrumento
{
    public String toString() { return "DeViento"; }
}

class Flauta extends DeViento {
    public String toString() { return "Flauta"; }
}

class Oboe extends DeViento {
    public String toString() { return "Oboe"; }
}

public class DePercusion extends Instrumento {
    public String toString() { return "DePercusion"; }
}

class Tambor extends DePercusion {
    public String toString() { return "Tambor"; }
}

```

Musica queda igual. Por ahora dejaremos el código así, cuando veamos *Interfaces* haremos algunos cambios más.

## Common Pitfalls: algunos errores frecuentes<sup>32</sup>

### ***Override de métodos privados***

Quizá esperaríamos que el siguiente código imprima “public f()” en la consola:

```

public class OverrideDeMetodosPrivados
{
    private void f() { System.out.println("private f()"); }

    public static void main(String[] args) {
        OverrideDeMetodosPrivados op = new SubClase();
        op.f();
    }
}

class SubClase extends OverrideDeMetodosPrivados {
    public void f() { System.out.println("public f()"); }
}

```

---

31 `System.out.println(this)` imprime el nombre de la clase a la que se refiere `this`.

32 Seguimos en parte el esquema de BRUCE ECKEL, *op.cit.*, p. 290 y ss.

Lo que imprime el programa es “private f()”. Como sabemos, los métodos privados no se pueden reemplazar. Por eso, en la SubClase, no estamos reemplazando f(), sino que estamos creando un método f() nuevo. Para la subclase, el f() de la clase base no existe, no lo puede ver: es private.

Si creamos en una clase distinta de OverrideDeMetodosPrivados un nuevo objeto de tipo OverrideDeMetodosPrivados diciendo, por ejemplo, OverrideDeMetodosPrivados op2 = new OverrideDeMetodosPrivados(); ¿podemos decir op2.f()? La respuesta es **no**, y el motivo es que f() es private. f() sólo existe dentro de la clase que define f(), y sólo puede ser usado por los métodos de esa clase. Por eso es que main en OverrideDeMetodosPrivados sí puede acceder a f(), como se muestra en el ejemplo anterior.

Veamos si tenemos el concepto claro ¿Qué sucedería si main estuviera definido en SubClase?

```
class SubClase extends OverrideDeMetodosPrivados {
    public void f() { System.out.println("public f()"); }
    public static void main(String[] args) {
        OverrideDeMetodosPrivados op = new SubClase();
        op.f();
    }
}
```

El código del párrafo anterior no compila: el compilador nos dirá que f() de OverrideDeMetodosPrivados es private. ¿Cómo se me ocurre llamarlo?

El error que estamos ejemplificando radica, sobre todo, en olvidarse que el f() de la clase base es private, y pensar que en la subclase estamos haciendo un *override*. El código que se mostró primero compila porque estamos definiendo un nuevo f().

Este error es más frecuente de lo que parece, al punto que Java 1.5 introduce un “decorador” @Override para indicarle al compilador que queremos hacer un override. En nuestro código, se vería así:

```
public class OverrideDeMetodosPrivados
{
    private void f() { System.out.println("private f()"); }
    public static void main(String[] args) {
        OverrideDeMetodosPrivados op = new SubClase();
        op.f();
    }
}

class SubClase extends OverrideDeMetodosPrivados {
    @Override
    public void f() { System.out.println("public f()"); }
}
```

Una vez añadido @Override, antes de public f(), el programa ya no compila: el compilador nos advierte que no estamos haciendo un *override*: *method does not override a method from its superclass*.

## Acceso a variables

Caemos en esta trampa cuando pensamos que como el acceso a los métodos en Java es

polimórfico, entonces también el acceso a las variables de la clase es polimórfico, **y no es así**. Si accedemos a una variable directamente, el acceso se resuelve usando *early binding*, en tiempo de compilación.

Consideremos el siguiente ejemplo:

```
class SuperClase
{
    public int i = 0;
    public int getI() { return i; }
}

class SubClase extends SuperClase {
    public int i = 1;
    public int getI() { return i; }
    public int getSuperI() { return super.i; }
}

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

        // upcast
        SuperClase sup = new SubClase();
        1 → System.out.println("sup.i = " + sup.i
                               + ", sup.getI() = " + sup.getI());

        // sin upcast
        SubClase sub = new SubClase();
        2 → System.out.println("sub.i = " + sub.i
                               + ", sub.getI() = " + sub.getI()
                               + ", sub.getSuperI() = " + sub.getSuperI());
    }
}
```

Si corremos el programa, se obtiene el siguiente resultado:

```
1 → sup.i = 0, sup.getI() = 1
2 → sub.i = 1, sub.getI() = 1, sub.getSuperI() = 0
```

(1) Cuando usamos un *upcast* para asignar un objeto de tipo `SubClase` a una variable de tipo `SuperClase`, el acceso a las variables directo estos objetos se resuelve en tiempo de compilación y, por tanto, no es un acceso polimórfico. En el ejemplo, existen en la práctica dos variables `i` distintas: la de `SuperClase` y la de `SubClase`. Cuando desde `sup` tratamos de acceder a `i`, obtenemos `i` de `SuperClase`.

(2) En cambio, cuando el objeto de tipo `SubClase` se asigna a una variable de tipo `SubClase`, la variable que obtenemos `i` que obtenemos si tratamos de acceder directamente a `i` es la de la `SubClase`.

Hasta aquí la teoría. Ahora, miremos de nuevo el código que hemos escrito. ¿Hay algo raro?

Lo raro es que a alguien se le ocurra definir nuevamente `int i` en `SubClase`, cuando ya existe un `i` del mismo tipo y público en `SuperClase`. ¿Por qué no usar el `i` de la `SuperClase`? ¿Supersticioso(a)? O, por último, si realmente necesitamos otra variable `int` distinta de la de `SuperClase`, ¿por qué no le ponemos otro nombre (`j`, por decir)?

En otras palabras, es difícil encontrar un motivo válido para intentar hacer un *override* de una variable pública (entre otras razones, porque no se produce el *override*)<sup>33</sup>.

---

33 Hablando de personas que no razonan con mucha claridad, decía Babbage: *On two occasions I have been asked [by*

## Override de métodos estáticos

Ya hemos visto en el capítulo sobre herencia, que los métodos estáticos no pueden ser reemplazados. Cuando se declara en una subclase un método con la misma firma que un método de la superclase, los dos métodos coexisten.

En las siguientes expresiones, ¿cuál de los dos métodos se llamará? Depende del **tipo** de la variable que usemos.

```
SuperClase sp = new SubClase();    sc.staticF(); ← 1
SubClase sb = new SubClase(); sb.staticF(); ← 2
```

(1) llama a `staticF()` de `SuperClase`, mientras que (2) llama a `staticF` de `SubClase`.

No hay polimorfismo para métodos estáticos, pues las llamadas a métodos estáticos se resuelven en tiempo de compilación (*early binding*). Escribir clases que redeclaren métodos estáticos de su clase base está vivamente desaconsejado, porque se presta a confusión y no parece que responda a ningún esquema de diseño razonable.

## Polimorfismo, constructores y orden de inicialización

Los constructores no son polimórficos<sup>34</sup>. Son métodos especiales, que se llaman siempre que se crea un nuevo objeto de una clase (una nueva *instancia* de una clase). Para poder escribir programas que manejen jerarquías complejas y llamadas polimórficas, es importante entender a fondo cómo se comportan los constructores y el orden en el que se produce la inicialización de las distintas partes de la clase.

El orden que se sigue en la creación de una clase es el siguiente:

1. Se reserva espacio para las variables declaradas en la clase del objeto<sup>35</sup> (también llamadas *member variables*, variables-miembro) y se inicializan a **cero** (independientemente de que le hayamos asignado en la declaración un valor o no)<sup>36</sup>. Cero quiere decir que los tipos primitivos (`int`, `float`, etc.) son **0**, y las variables que almacenarán referencias a una clase (`String`, etc.) se inicializan a **null** (que indica que no tienen nada almacenado).
2. Se llama al constructor de la clase base<sup>37</sup> que se especifique en el constructor de nuestra clase. Si no se especifica un constructor de la superclase usando `super`, Java intenta llamar al constructor por defecto usando `super()`. (En este proceso se repite el n.1 y 2 para la superclase, hasta llegar al constructor de `Object`, “madre de todas las clases madre”).
3. Se inicializan las variables-miembro según el código de la clase<sup>38</sup>. Se sigue el orden en el

---

*members of Parliament*], 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question. Charles Babbage (1791-1871) fue un matemático inglés, filósofo, ingeniero mecánico y proto-computer-scientist, y el origen de la idea de la computadora programable. (cfr. [http://en.wikipedia.org/wiki/Charles\\_Babbage](http://en.wikipedia.org/wiki/Charles_Babbage))

34 En realidad, los constructores son métodos implícitamente estáticos. Pero si este dato lo confunde, óbvio por ahora.

35 Nos referimos a las variables declaradas en el cuerpo de la clase, no dentro de los métodos de la clase.

36 Esto quiere decir que si

```
class Test {
    int i = 5;
```

}, entonces se reserva espacio para `i`, y el espacio se inicializa a `0`. El `5` se asignará a `i` más adelante.

37 Recordemos que todas las clases descienden finalmente de `Object`. Si nuestra clase no tiene una superclase explícitamente declarada usando `extends`, entonces el constructor de la superclase que se llama es el constructor de `Object`.

38 Ahora sí, asignamos `5` a `i`.

que están declaradas.

4. Se ejecuta el código del constructor de nuestra clase.

Veamos un ejemplo. Además de imaginarte la deliciosa butifarra, ¿puedes deducir qué se imprimirá en la consola?

```
public class Alimento { ←5
    public Alimento() {
        System.out.println("Alimento"); ←6
    } ←7
}
3 → public class Pan extends Alimento {
    4 → public Pan() { ←8
        System.out.println("Pan"); ←8
    }
}
public class Jamon extends Alimento {
    public Jamon() {
        System.out.println("Jamón");
    }
}
public class SalsaCriolla extends Alimento {
    public SalsaCriolla() {
        System.out.println("SalsaCriolla");
    }
}
public class Lechuga extends Alimento {
    public Lechuga() {
        System.out.println("Lechuga");
    }
}

public class Butifarra {
    private Pan pan = new Pan(); ←2
    9 → private Jamon jamon = new Jamon();
    private Lechuga lechuga = new Lechuga();
    private SalsaCriolla salsaCriolla = new SalsaCriolla();

    1a → public Butifarra() {
        System.out.println("Butifarra"); ←10
    }

    1 → public static void main(String[] args) {
        new Butifarra();
    }
}
```

Llamada implícita a super()

La salida del programa es la siguiente:

```
Alimento
Pan
Alimento
Jamón
Alimento
Lechuga
Alimento
SalsaCriolla
```

El programa empieza a correr en `main`. `main` es un método estático, de modo que no se ha creado todavía ningún objeto hasta que Java encuentra la línea `new Butifarra()`:

1 ➡ Se reserva espacio en memoria para el nuevo objeto y para todas sus variables miembro. Las variables miembro de `Butifarra` son `pan`, `jamón`, `lechuga` y `salsaCriolla`. Se inicializan las variables miembro a cero (en este caso, a `null`).

1a ➡ (se llama al constructor de la superclase de `Butifarra`. Pero `Butifarra` descende sólo de `Object`, de modo que, a efectos de lo que estamos viendo, no sucede nada).

2 ➡ Empieza la inicialización de las variables miembro según se especifica en la clase. Se llama a `new Pan()`, que es el valor que se quiere asignar a `pan`. Al encontrar la expresión `new Pan()`, el compilador salta a la declaración de `Pan`.

3 ➡ En la clase `Pan`, el compilador separa espacio en memoria para un nuevo objeto de tipo `Pan`, reserva espacio para sus variables miembro e inicializa sus valores a cero (pero `Pan` no tiene variables miembro, así que no pasa nada).

4 ➡ Entramos al constructor de `Pan`. Como `Pan()` no usa `super`, el compilador automáticamente inserta un `super()`, y se llama al constructor de la clase base de `Pan`, es decir, de `Alimento`.

5 ➡ El compilador salta a la declaración de `Alimento`. Reserva espacio para las variables miembro y las inicializa a cero (en este caso, no hay ninguna).

6 ➡ Entramos al constructor de `Alimento`. Como `Alimento()` no llama explícitamente a ningún constructor de su superclase (`Object`, en este caso), el compilador llama automáticamente al constructor de la superclase (a efectos de nuestro problema, no pasa nada).

7 ➡ Se ejecuta (¡por fin!) el código del constructor de `Alimento`: se imprime “`Alimento`” en la consola. Termina el constructor de `Alimento` y volvemos al constructor de `Pan`.

8 ➡ En el constructor de `Pan` se ejecuta el código que imprime “`Pan`” a la consola.

9 ➡ Terminado el constructor de `Pan`, el objeto es asignado a la variable `pan` y el código continúa en la siguiente línea, con la creación de un objeto `Jamón`. Se repite el proceso de los nn. 3 a 8: saltamos al constructor de `Jamón`; se inicializan a cero las variables-miembro de la clase `Jamón` (no tiene); se llama al constructor de `Alimento`, la superclase de `Jamón`; y, se imprime `Alimento` en la consola. El código retorna al constructor de `Jamón`, se asignan los valores a las variables miembro de `Jamón` (en este caso no las tiene) y luego se ejecuta el código del constructor de `Jamón`: se imprime “`Jamón`” en la pantalla.

El proceso sigue hasta que todas las variables-miembro están inicializadas: aparecen en la pantalla `Alimento`, `Lechuga`, `Alimento`, `SalsaCriolla`.

10 ➡ Finalmente, se ejecuta el código del constructor de `Butifarra`, y se imprime “`Butifarra`” en la pantalla.

Cabe preguntarse por qué se sigue este orden en la creación de un objeto. Por supuesto, hay una lógica detrás de este modo de hacer las cosas:

- Cuando se ejecuta el código del constructor de una clase, el constructor da por sentado que las variables-miembro de la clase ya hayan sido inicializadas y están listas para usarse. Por eso es necesario inicializarlas antes de que se llame al constructor.
- Del mismo modo, cuando empieza a ejecutarse el constructor de una subclase, éste da por supuesto que las variables y métodos de su superclase ya están inicializados y funcionando.

Por eso es necesario inicializar la parte del objeto que corresponde a la superclase llamando a su constructor antes de que se ejecute el código del constructor de la subclase.

## Comportamiento polimórfico de métodos dentro de constructores

Está vivamente recomendado que los métodos que se llamen dentro de los constructores sean `final` o `private`, de modo que no puedan cambiar en las clases derivadas.

Ahora que sabemos en qué orden se llama a los constructores de la clase base y de las subclases se nos plantea un dilema interesante. ¿Qué sucede si el constructor de nuestra clase base llama a un método que luego es reemplazado en la subclase?

Consideremos el siguiente código:

```
public class Figura
{
    void draw() { System.out.println("Figura.draw()"); }

    4 → public Figura() {
        System.out.println("Figura() antes de draw()");
        5 → draw();
        System.out.println("Figura() después de draw()"); 7
    }
}

public class Circulo extends Figura
{
    1 → private int radio = 1; 8
    3 → Circulo(int r) {
        radio = r; 9
        System.out.println("Circulo.Circulo(), radio=" + radio);
    }

    6 → public void draw() {
        System.out.println("Circulo.draw(), radio=" + radio);
    }
}

public class Test
{
    public static void main(String[] args) {
        1 → new Circulo(5);
    }
}
```

¿Cuál es la salida del programa? No debería sorprendernos:

```
Figura() antes de draw()
Circulo.draw(), radio=0
Figura() después de draw()
Circulo.Circulo(), radio=5
```

Sigamos rápidamente la ejecución del programa:

- 1 → El código empieza a ejecutarse. Java va a crear un nuevo objeto `Circulo`.
- 2 → Se reserva espacio para las variables miembros de `Circulo`, y se les asigna el valor 0.

En este momento, `radio = 0`.

3 ➡ Empieza la ejecución del constructor de `Circulo`.

4 ➡ Antes de ejecutarse la primera línea del constructor, se llama al constructor de la clase base `Figura` usando `super()` (al no llamar nosotros explícitamente al constructor de la clase base, Java lo hace automáticamente). Se ejecuta la primera línea del constructor, que imprime “`Figura()` antes de `draw()`”.

5 ➡ Éste es el punto importante. Se llama al método `draw()`. ¿Cuál de los dos métodos `draw()` se ejecuta? ¿El de `Figura` o el de `Circulo`? **Se ejecuta el de `Circulo`**, pues el de `Figura` ha sido reemplazado al hacer el *override*.

6 ➡ Se ejecuta `draw()` de `Circulo` y, sorprendentemente, se imprime el valor de `radio` que es 0 en este momento. No olvidemos que todavía no se ha asignado ni el valor inicial a las variable-miembro (`int radio = 1`) ni se le ha asignado el valor 5 que estamos pasando como parámetro al constructor de `Circulo`.

7 ➡ Salimos de `draw()` y se sigue ejecutando el código del constructor de `Figura`. Se imprime en la pantalla “`Figura()` después de `draw()`”. Salimos del constructor de `Figura`.

8 ➡ Antes de empezar a ejecutar el código del constructor de `Circulo`, se asignan inicializan las variables miembro a los valores que especifique el programa, si es que se especifica algún valor. En este caso, se asigna 1 a `radio`.

9 ➡ Ahora sí, empieza a ejecutarse el código del constructor de `Circulo`. Se asigna a `radio` el valor del parámetro `r` (5 en este caso), y se imprime “`Circulo.Circulo(), radio=5`”.

Situaciones como la del código anterior pueden llevar a errores (*bugs*) sumamente difíciles de detectar. ¿Cuál es la moraleja de la historia? Los métodos que son llamados desde el constructor para inicializar una clase son normalmente cruciales para que la clase se inicialice correctamente. No podemos correr el riesgo de que un gracioso cree una clase derivada y los reemplace. Por tanto, siempre que sea posible (la mayor parte de veces lo será) deben declararse `private` o `final`.

Comentarios, correcciones y sugerencias: Roberto Zoia (roberto.zoia@gmail.com)

Salvo que alguien considere que hay partes que ya tienen un copyright más restrictivo: This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.