

Resumen PAV 1ⁱ

2006-mar-27

Este resumen no pretende ser un curso de Java, sino que está pensado para ser leído después de haber asistido a clase. Tampoco resume *todos* los temas tratados en clase.

Programación orientada a objetos¹

Introducción

Los objetos son claves para entender la programación orientada a objetos. Mirando a nuestro alrededor podemos ver algunos objetos del mundo real: un escritorio, una ventana, un perro, un televisor...

Los objetos reales comparten dos características: tienen un *estado* y tienen un *comportamiento*. Por ejemplo, los perros tienen un estado (un nombre, un color, están hambrientos) y comportamiento (ladran, mueven la cola). Un automóvil tiene estado (su velocidad actual, la cantidad de gasolina en el tanque, su cilindrada) y un comportamiento (frena, acelera, enciende el motor).

Los objetos de software se modelan tomando como base los objetos del mundo real en el sentido de que también tienen un estado y un comportamiento. Un objeto de software mantiene su estado en una o más variables², e implementa su comportamiento usando métodos³.

Los objetos del mundo real pueden representarse utilizando objetos de software. Por ejemplo, podríamos representar a un perro del mundo real como objetos de software para usarlo en nuestro juego “el pitbull asesino”; o podríamos representar una bicicleta real usando con objetos de software para usarlo en un programa que controla una bicicleta electromecánica para hacer gimnasia.

Un ejemplo más: un *evento* es un objeto común que se usa en los sistemas operativos con interfaz gráfica (ventanas, botones, etc.) para representar la acción de un usuario al presionar un botón del

1 Algunas ideas y ejemplos de este apartado han sido tomadas de <http://java.sun.com/docs/books/tutorial/java/concepts/object.html>

2 Una variable es un elemento de datos al que se asocia un nombre (un identificador).

3 Un método es una función -una subrutina, un procedimiento... en este contexto podemos usar estos términos como sinónimos- asociada a un objeto.

mouse o una tecla del teclado.

Todo lo que el objeto de software sabe (su estado) y puede hacer (su comportamiento) está expresado en las variable y métodos de ese objeto.

El proceso de abstracción

Todos los lenguajes proporcionan abstracciones. Se podría decir que la complejidad de los problemas que podemos resolver con un determinado lenguaje de programación están en relación al tipo y calidad de la abstracción que permiten. Con “tipo” nos referimos aquí a *qué es lo que estamos abstrayendo* con el lenguaje.

El *Assembly language*, el lenguaje ensamblador (o lenguaje de máquina) que permite programar usando las instrucciones del CPU es una pequeña abstracción del CPU. Los lenguajes que siguieron, como el FORTRAN, BASIC y C, representaron una gran mejora respecto al lenguaje de máquina, pero tienen la limitación de que exigen resolver el problema que tengamos entre manos en términos de la estructura del computador en vez de en términos del problema mismo.

El enfoque de la programación orientada a objetos intenta proporcionar las herramientas necesarias para que el programador resuelva el problema en términos del problema mismo, independientemente de la arquitectura del computador en el que se implemente la solución. Dicho de otro modo, los lenguajes orientados a objetos permiten crear nuevos tipos de datos (clases), de modo que cuando uno describe el problema en el lenguaje de programación está describiendo el problema usando los términos del problema mismo.

Características de los lenguajes orientados a objetos

Alan Kay⁴ resumió las cinco características básicas de SmallTalk (uno de los primeros lenguajes orientados a objetos):

1. Todo es un objeto. Como hemos visto, un objeto es una agrupación de los datos y los métodos que actúan sobre los datos. En teoría, se debería poder tomar cualquier elemento conceptual del problema que estamos tratando de resolver (edificios, perros, bicicletas) y representarlo como un objeto en nuestro programa.
2. Un programa es un grupo de objetos que se dicen unos a otros qué tienen que hacer enviándose mensajes. *Enviar un mensaje* es otro modo de decir *llamar un método* del objeto.
3. Cada objeto tiene su propia memoria, y está compuesta por objetos. La *memoria* del objeto es su estado. Como *todo* es un objeto (cfr. n. 1), también los atributos (las variables) más primitivas (int, float, etc.) son objetos. Por tanto, la memoria está compuesta por objetos. Pero el objeto que compone la memoria también puede ser un objeto más complejo, con sus propios atributos y métodos.
4. Cada objeto tiene un tipo. Dicho en jerga de programación, cada objeto es la *instancia* (*instance*) de una clase. Aquí introducimos una distinción importante: con *clase* nos referiremos a la definición de un objeto (su plantilla, por decirlo de algún modo), mientras que

4 cfr. http://en.wikipedia.org/wiki/Alan_Kay

al decir *objeto* nos estamos refiriendo a un objeto creado en el programa en base a esa definición⁵. Volviendo al *tipo*, como una clase describe el conjunto de objetos que tienen idénticas características (atributos) y comportamientos (métodos), en sentido amplio una clase es un *tipo de datos*, de modo similar a como el tipo de datos *int* define la característica y comportamiento que tienen los números enteros.

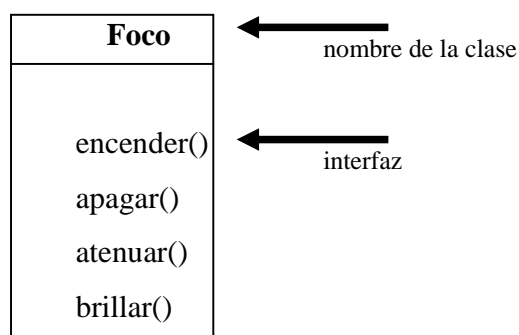
5. Todos los objetos de un tipo particular pueden recibir los mismos mensajes. En otras palabras, se pueden llamar los mismos métodos en todos los objetos creados a partir de una determinada clase⁶.

Booch expresa estas mismas ideas de un modo más sucinto: *un objeto tiene estado, comportamiento e identidad*. Un objeto tiene estado (sus atributos, que le permiten almacenar su estado), métodos (que le permiten comportarse de un modo determinado) y se distingue de otros objetos creados en el programa.

La interfaz

El comportamiento de una clase se define por su interfaz (*interface*). La interfaz determina los mensajes que se pueden enviar a un determinado objeto (traduciendo, los métodos que puede usar de ese objeto) para que realice una acción determinada.

Por ejemplo, un foco puede tener métodos para encenderlo, apagarlo, atenuar o aumentar su intensidad. Los métodos que publica la clase Foco se denominan su interfaz. Gráficamente, podría representarse así:



La clase puede tener otros métodos privados, que el programador de la clase ha necesitado para implementar los métodos que hace públicos. Pero desde el punto de vista del usuario de la clase, esos métodos no existen. Este enfoque al diseño de objetos, exponiendo sólo los métodos que se necesitan para que el objeto cumpla su cometido, se llama *information hiding* (ocultar la información).

Encapsulación e *information hiding*

Hay que distinguir *information hiding* (ocultar información) y encapsulación:

Information hiding, es una *estrategia de diseño de software*. Se refiere a ocultar al usuario de una

5 La clase son los planos para la construcción del objeto. Así como los planos para construir una bicicleta y la bicicleta son distintos, la clase y el objeto distintos.

6 Como veremos más adelante, cuando hablemos de *herencia*, esta afirmación es más rica de lo que parece a primera vista. Un objeto de clase círculo derivado de la clase figura es también una figura y, por tanto, puede recibir los mismos mensajes que la clase figura.

clase los detalles del diseño de la clase. El usuario utiliza el objeto a través de una interfaz bien definida. Con interfaz nos referimos aquí a los métodos y atributos que la clase hace públicos, es decir, los servicios que ofrece.

La encapsulación se refiere al agrupamiento de los datos y los métodos que actúan sobre los datos en un mismo objeto. Estos métodos y datos pueden estar ocultos, pero no necesariamente lo están.

Es decir, la encapsulación no implica necesariamente *information hiding*⁷.

La idea de *information hiding* fue introducida por David Parnas en 1972. El motivo para ocultar la información sobre cómo se implementa un determinado comportamiento de un objeto es que más adelante se pueda modificar el modo en que se implementa el método, sin que afecte a los programas que usan los métodos públicos de esa clase. Esto implica, se entiende, que los métodos y atributos que la clase publica son prácticamente inalterables.

Más vueltas sobre la interfaz

*An interface is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface*⁸. La interfaz es un contrato que especifica una colección de métodos y declaraciones constantes. Cuando una clase implementa una interfase, se compromete a implementar todos los métodos declarados en la interfase.

Código

Cómo se define una clase en Java

Las clases se definen con la palabra reservada `class`:

```
class Ejemplo {  
  
}
```

Las clases pueden tener métodos y atributos:

```
class NumeroComplejo {  
    private int x;  
    private int y;  
    public int sumaXY(int a, int b) {  
        return x + y;  
    }  
}
```

7 Hay que decir que muchos autores tratan encapsulación e *information hiding* como sinónimos, cuando *strictu sensu* no lo son. Para una discusión sobre el particular, cfr http://www.javaworld.com/javaworld/jw-05-2001/jw-0518-encapsulation_p.html

8 <http://java.sun.com/docs/books/tutorial/java/concepts/index.html>

```
}
```

Los atributos (también llamados variables o campos) se declaran con la siguiente sintaxis:

[cualificador de acceso] [modificador] [tipo] [nombre];

Por ejemplo:

```
private int x;  
public string nombre;  
private static int y;
```

Los métodos se declaran de modo semejante:

[cualificador de acceso] [modificador] [tipo] [nombre] (argumentos)

Ejemplo:

```
public void9 lanzaMisil(int numeroDeMisil) {  
    //10 Aquí iría el código que implementa el método  
}
```

Los argumentos de un método representan la información que necesita el método para realizar su función. Como todo en Java, los argumentos también son objetos. Por tanto, cuando definimos un método:

- debemos declarar el *tipo* de objeto de cada argumento (decir a qué clase pertenece);
- debemos dar a cada argumento un identificador (un nombre), que será usado dentro del método para hacer referencia al argumento en cuestión.

Firmas (signatures) y sobrecarga (overload)

Se denomina *signature* a la combinación nombre-del-método + tipo-de-los-argumentos-del-método. Dentro de una misma clase no pueden haber dos firmas iguales (no pueden haber dos métodos con el mismo nombre y con el mismo tipo de argumentos, aunque los nombres de los argumentos sean distintos). Java utiliza esta combinación para saber qué método de la clase debe ejecutar.

Cabe resaltar que el tipo que devuelve el método no es parte de la firma. La razón es que si dos métodos tuvieran el mismo nombre y la misma secuencia de tipos de argumentos, aunque devolviesen tipos de datos distintos, el compilador no podría saber a cuál de los dos está llamando el programa.

9 void, como todos sabemos, quiere decir “vacío”, e indica que el método no devuelve ningún valor. Desde otra perspectiva: un método puede devolver cualquier tipo de objeto (int, string, etc.) usando la palabra clave **return**. Pero si el método no devuelve ningún objeto, hay que indicarlo con la palabra clave **void**.

Por tanto, el método no puede ser parte del lado derecho de una asignación: si definimos un método `float raizCuadrada(float n) { (...) }`, podemos decir `a = 5 + raizCuadrada(16);`. En cambio, si decimos `void lanzaMisil(int numeroDeMisil) { (...) }`, no podemos decir `a = 5 + lanzaMisil(4);` pues `lanzaMisil` no devuelve ningún valor y, por tanto, no se puede sumar a 5.

10 De pasada comentamos que en Java los comentarios al código se especifican del mismo modo que en C++: `/* un comentario que puede abarcar varias líneas */` o `// un comentario de una sola línea`

Algunos ejemplos:

El siguiente código producirá un error de compilación, pues las firmas de los dos métodos `initGraphics` son idénticas (`initGraphics(int, int)`):

```
class GameEngine {
    int initGraphics(int width, int height) { return 0; }
    void initGraphics(int ancho, int alto) { }
}
```

El error es: `initGraphics(int,int) is already defined in GameEngine`.

Este código, en cambio, es válido, pues los métodos tienen firmas distintas:

```
class GameEngine {
    int initGraphics(int width, int height) { return 011; }
    int initGraphics(int width, int height, int backgroundColor) { return 0; }
}
```

El siguiente código muestra cómo Java distingue a qué método debe llamar en base al tipo de datos con que se llama al método:

```
public class Overloaded {
    // Este método tiene un argumento int
    void unMetodo(int a) {
        System.out.println("Estoy dentro de unMetodo(int). El argumento es "
+ a);
    }

    // Este método tiene un argumento de tipo string
    void unMetodo(String a) {
        System.out.println("Estoy dentro de unMetodo(String). El argumento es
" + a);
    }

    public static void main(String[] args) {
        Overloaded o = new Overloaded();
        // Java llama a métodos distintos dependiendo del tipo de
        // datos del argumento:
        o.unMetodo(6); // llamará a unMetodo(int)
        o.unMetodo("Hola"); // llamará a unMetodo(string)
    }
}
```

Salida¹²:

```
Estoy dentro de unMetodo(int). El argumento es 6
Estoy dentro de unMetodo(String). El argumento es Hola
```

Como es de suponer, si llamo a un método que no haya sido definido (`unMetodo(int, String)`, por ejemplo), el compilador dará un error.

Cuando en una clase existen varios métodos con el mismo nombre pero con firmas distintas, se

¹¹ Como estamos diciendo que el método devuelve un `int`, necesariamente tenemos que devolver un valor para que el código compile. Por eso escribimos `return 0`.

¹² Intente compilar y ejecutar este programa.

dice que el método está *sobrecargado* (*overloaded*). La sobrecarga de métodos es una característica muy potente de Java.

Los archivos de clases

El código fuente de las clases de Java se guarda en un archivo en un directorio de la PC. Sin embargo, Java impone una serie de restricciones sobre el contenido de los archivos de código fuente:

- el compilador Java permite una sola clase pública por archivo. En el mismo archivo pueden haber otras clases no públicas (que probablemente serán usadas por la clase pública).
- el archivo debe tener la extensión .java.
- si la clase es parte de un paquete, el nombre del archivo debe coincidir con el nombre de la clase pública (respetando mayúsculas y minúsculas).

Un ejemplo. Supongamos que queremos definir una clase pública Misil:

Archivo: Misil.java

```
/*
 * Misil.java
 */
public class Misil {
    private int potencia; // Potencia en megatones
    private int autorizacion; // Nivel de privilegio requerido
                                // Para autorizar el lanzamiento

    public void lanzaMisil(int nivelAutorizacion) {
        ConexionAPentagonito cn = new ConexionAPentagonito();

        if (cn.verificaAutorizacion(nivelAutorizacion) == 0) {
            lanzarMisil();
        } else {
            cn.informaIntentoNoAutorizado();
        }
    }

    /* lanzarMisil:
     * es el método privado que realiza realmente el
     * lanzamiento. No puede ser llamado directamente por otras
     * clases, para que no lancen el misil sin que se verifique
     * la autorización.
     */

    private void lanzarMisil() { /* (...) aquí lanzo el misil */ }
} // fin de la clase Misil

/*
 * class ConexionAPentagonito
 * es una clase no pública, que sólo puede ser usada por las otras clases
 * del paquete
 */
class ConexionAPentagonito {
```

```

        public void verificaAutorizacion(int nivel) {
            /* (..) */
        }
        public void informaIntentoNoAutorizado() {
            /* (...) */
        }
    } // fin clase ConexionAPentagonito
---
Aquí termina el archivo Misil.java

```

Al conjunto formado por la clase pública Misil y su clase ConexionAPentagonito se le conoce como *unidad de compilación (compilation unit)*. El compilador convierte cada unidad de compilación en un archivo de extensión **.class**.

Creación de objetos

Como hemos visto arriba, la definición de una clase no es más que una plantilla. Por eso, para que un objeto empiece a existir en nuestro programa y podamos usarlo, necesitamos crearlo.

En Java hay dos modos de crear objetos:

a) Creación de objetos en tiempo de ejecución

Para crear una instancia (*instance*) de una clase (crear un objeto a partir de una clase), se usa el operador **new**:

```

class Test {
    int x;

    public void Hola(String s) { System.out.println(s); }
}

public class Programa {
    public static void main(String[] args) {
        Test test = new Test();
        test.Hola("Prueba de new");
    }
}

```

b) Creación de objetos en tiempo de compilación

En lenguajes como C, el programador puede hacer que el compilador reserve espacio para una variable en el momento en que el programa es compilado. Cuando el programa se ejecute, antes de que llegue a la función de entrada (main), ya estará reservado el espacio para esas variables. Por ejemplo, consideremos el siguiente código C:

```

int x;
int main(int argc, char * argv[]) {
    x = 9;
    printf("Valor de x: %d", x);
}

```

Aunque la línea **x=9**; se ejecute recién cuando el programa corra, sin embargo el compilador ha separado el espacio suficiente en memoria para un int. En el instante en que el programa empieza a correr la variable x ya está lista para recibir valores.

En Java *todo es un objeto*, y no hay código que esté fuera de una clase. Por tanto, sólo es posible decir `int x` dentro de una clase, y las clases, como sabemos, son sólo una plantilla: no existen hasta que no son creadas.

Estríctamente hablando, en Java esto sólo sucede cuando una variable o un método se declara de tipo estático (`static`). Podríamos pensar que cuando declaramos una variable dentro de nuestra clase estamos creándola en tiempo de compilación:

```
class Test {
    int x;    // se podría pensar equivocadamente que se
              // reserva espacio para x al compilar el prog.
}
```

pero no es así, y el motivo es muy sencillo: hasta que no se cree explícitamente con el operador **new**, no existe ningún objeto de la clase `Test`. Si la clase `Test` no se crea, no pasa de ser una plantilla. El compilador toma nota y hasta nos felicita por nuestra creatividad, pero no crea ningún objeto en memoria.

Si queremos que el método o el atributo exista *antes* de que se haya creado una instancia de la clase, necesitamos decírselo así al compilador usando la palabra **static**.

```
class Test {
    static int x;
}
```

Ahora `x` existe independientemente de si la clase se crea o no con **new**. Pero este modo de hacer tiene consecuencias: cuando una variable se declara `static`, la variable se vuelve única para todas las instancias de la clase `Test`. Todos los objetos de tipo `Test` tienen la misma variable `x`, y si cambio `x` en uno de los objetos `x` cambia de valor en todos los demás (porque, repetimos, sólo hay una variable `x`, que es común a todos ellos).

Los métodos también pueden ser declarados **static**. Cuando un método se declara estático, puede llamarse independientemente de si se ha creado un objeto a partir de la clase o no. Pero, por eso mismo, los métodos estáticos tienen la restricción de que sólo pueden usar otros objetos o métodos estáticos, o crear nuevos objetos.

```
public class Test {
    public static int x;
    public static void imprime() {
        int a = operacionA(x);
        a = operacionB(a);
        System.out.println("Resultado: " + a);
    }
    // Estos métodos, llamados por imprime, son
    // estáticos para que imprime los pueda llamar, pues
    // los métodos estáticos sólo pueden llamar métodos
    // estáticos
    static int operacionA(int a) { return a*a; }
    static int operacionB(int a) { return a+a; }
}
```

Conviene notar algunas cosas de este último ejemplo. Esta vez, además de declarar `imprime` como método estático, lo hemos declarado **public**. De lo contrario nadie podría usar el método

`imprime` hasta que no se cree un objeto usando `new`, lo cuál, en este caso, haría inútil la declaración `static`¹³.

Para llamar un método estático sin haber creado un objeto de esa clase, se usa el nombre de la clase como objeto base. Así, en nuestro caso:

```
Test.imprime();    --> Resultado:  014
```

En cambio,

```
Test.x = 5;
Test.imprime();    --> Resultado: 50.
```

Otro ejemplo: el siguiente código usa una variable estática para contar cuántos objetos de tipo `Test` se han creado¹⁵.

```
class Test {
    static int n = 0;    // una variable estática, común a todos los
    objetos              // de clase Test. La inicializamos explícitamente a 0

    public void imprime() {
        n++;    // incrementamos la cuenta
        System.out.println("Ahora hay " + n + " objetos de clase Test");
    }
}

public class Programa {
    public static void main(String[] args) {
        Test a;    // un arreglo de 10 elementos
        // Creamos 10 objetos de clase Test
        for (int i=0; i<10; i++) {
            a = new Test();
            a.imprime();
        }
    }
}
```

El programa produce la siguiente salida¹⁶:

```
Ahora hay 1 objetos de clase Test
Ahora hay 2 objetos de clase Test
Ahora hay 3 objetos de clase Test
Ahora hay 4 objetos de clase Test
Ahora hay 5 objetos de clase Test
Ahora hay 6 objetos de clase Test
```

13 Esto no quiere decir que no puedan haber métodos estáticos que no sean públicos (de hecho los hay, y sirven para ser usados por otros métodos estáticos).

14 El resultado es 0, pues si no se inicializa explícitamente la variable `x`, Java la inicializa a 0.

15 En “técnico”, cuántas *instancias* de `Test` hay en el programa.

16 ¿Está seguro? Abra BlueJ y pruébelo.

```
Ahora hay 7 objetos de clase Test
Ahora hay 8 objetos de clase Test
Ahora hay 9 objetos de clase Test
Ahora hay 10 objetos de clase Test
```

El método main

El método `main` es un método estático especial. Es el punto de inicio del programa. Si se piensa un poco, se entiende que tenga que ser un método estático, pues cuando el programa empieza a correr, no existe aún ningún objeto. Necesitamos, por tanto, llamar a un método estático, que exista antes de que se cree ningún objeto, y ese método es `main`.

Siguiendo la misma lógica, el método `main` tiene que ser público, pues de lo contrario, ¿cómo podríamos ejecutarlo desde fuera de la clase que lo contiene?

El método `main` recibe, como argumentos, un arreglo de tipo `String` con los parámetros que se han pasado al programa al ejecutarlo.

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

El nombre `args` es arbitrario: podemos llamarlo como queramos.

Al declarar `main` como un método `void`, estamos diciendo que el programa no devuelve ningún tipo de valor. Pero, como sospechábamos y nunca nos atrevimos a preguntar, `main` puede devolver un tipo:

```
public static int main(String[] args) {
    return 1;
}
```

Para la mayoría de nuestros programas, nos bastará usar la primera forma.

Constructores

Un constructor es un método especial que es ejecutado automáticamente cuando se crea el objeto usando el operador `new`. Si el constructor no se declara explícitamente, Java proporciona automáticamente un *constructor por defecto*, que sólo reserva el espacio necesario en memoria.

En el siguiente ejemplo, se usa el constructor por defecto:

```
// importamos java.util.Date:
import java.util.*;

class TestConstructor {
    // Definimos un atributo fecha: el new Fecha() se
    // ejecutará recién cuando se cree el objeto TestConstructor
    Date fecha = new Date();

    public static void main(String[] args) {
        // Creamos la instancia de TestConstructor, usando
        // el constructor por defecto.
        TestConstructor test = new TestConstructor();
        test.printFecha();
    }
}
```

```

        void printFecha() { System.out.println("Fecha: " + fecha); }
    }

```

Algunos comentarios sobre el ejemplo:

- el método main del código de arriba usa el operador **new** para crear una instancia de la clase que lo contiene al propio método main. Aunque a primera vista puede parecer que esto es un contrasentido, *es importante no confundir el método main, que es el método en el que se inicia el programa*, con el constructor, que es el método que se llama cuando se crea un objeto.
- Estamos llamando al constructor por defecto, pues no hemos definido explícitamente ningún constructor.

Declaración del constructor

Para definir un constructor:

- se usa un metodo del mismo nombre de la clase que lo contiene
- no se especifica el tipo que devuelve el constructor, porque el constructor no devuelve ningún tipo. *Tampoco* se usa la palabra clave void.
- una clase puede tener varios constructores, siempre y cuando sus *signatures* sean distintas.

Algunos ejemplos:

En este ejemplo, el programa que contiene a main crea una clase distinta a la propia.

```

class Persona {
    private String nombre;
    private String apellido;
    private String DNI;

    /* Constructores */
    Persona(String _nombre, String _apellido, String _DNI) {
        nombre = _nombre;
        apellido = _apellido;
        DNI = _DNI;
    }

    /*
    * Persona(): este constructor crea la clase con
    ** los campos en blanco, llamando al constructor de arriba
    */

    Persona() {
        this("", "", "");
    }

    /* un método útil para mostrar la información */
    public String toString() {
        String result = "Persona\n" +
            "-----\n" +
            "Nombre: " + nombre + "\n" +
            "Apellido: " + apellido + "\n" +
            "DNI: " + DNI + "\n";

        return result;
    }
}

/* la clase que contiene a main */
public class Programa2 {

```

```

    public static void main(String[] args) {
        Persona persona = new Persona("Fulando", "de Tal", "09213213");
        System.out.println(persona);
    }
}

```

Hay varias cosas que comentar en este ejemplo:

a) Los constructores: la clase `Persona` tiene dos constructores, que se diferencian por su firma (tienen argumentos distintos). Pero lo interesante es cómo estamos implementando el constructor de `Persona()`:

```

    Persona() {
        this("", "", "");
    }

```

La palabra reservada `this` quiere decir en Java “el objeto que está actualmente en uso”. Dicho de otro modo, decir `this("", "", "")` es equivalente a llamar al constructor `Persona("", "", "")`, simplemente que el constructor sólo se puede llamar directamente usando el método `new`.

b) El método `toString()`: al definir el método `toString`, nos hemos adelantado al tema de herencia, que veremos más adelante. Pero podemos ir tocando algunos conceptos.

En Java todos los objetos tienen un método `toString()`, que es llamado cuando se esperaría una variable de retorno de tipo `String`. Las clases pueden redefinir el método `String` y devolver una representación de sí mismas, de modo que en cualquier momento se pueda decir `System.out.println(objeto)` y `println` llama automáticamente al método `toString()` de `objeto`.

Otros usos de `this`

La palabra reservada `this` se puede usar cuando se quiere distinguir entre un método o atributo de la propia clase y un método o atributo que está en otro ámbito. Por ejemplo:

```

class Prueba {
    private int x;
    private int y;

    // El constructor
    Prueba (int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

El constructor `Prueba` usa como argumentos dos variables `int` que tienen los mismos identificadores que los atributos de la clase. Por tanto, necesitamos decir claramente a qué `x` e `y` nos estamos refiriendo: a la de la clase, o al argumento del constructor. Para eso usamos el keyword `this`.

i 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.