Diario de un Principiante

Episodio 2: Cómo empezar a programar

por Leo Suarez

Por fin empiezo a programar en Java.

¿Ya te has dado un atracón de POO?, ¿Has recopilado toda la documentación necesaria? Pues si es así vamos a ponernos manos a la masa que es lo que nos interesa a todos... sumergirnos en el apasionante mundo de Java ; -). Eso sí, a partir de ahora debemos tener abiertas en el navegador que utilices las especificaciones HTML de la API de Java, o tener uno de esos entornos de desarrollo modernos que te sugiere los métodos de una clase según los escribes.

En este apartado siempre interesa empezar por los típicos programas del estilo Hola Mundo aunque cuando tengas más soltura lo que te recomiendo es que te plantees solucionar algún problema, es decir, programarlo y en los tutoriales (un mazo de ejemplos en el directorio .../java/demo/ y en los tutoriales on-line de Sun) vayas buscando los ejemplos que se asemejen a las clases que quieres realizar. Eso te va a servir de referencia ya que tienes una base de la que partir, te familiarizará con las clases que incorpora la API de Java, sus métodos, etc...

En este artículo nos vamos a centrar en cómo empezar a desarrollar una clase, adquirir un e stilo de programación y saber qué son y para qué son las palabras reservadas que casi siempre aparecen en cualquier clase hecha en Java.

package.

Una cosa que siempre cuesta al principio es programar orientado a objetos, esto es, emplear los beneficios de la POO ya que lo que normalmente hacemos es usar los métodos y clases que Java nos ofrece pero con la tendencia de programación estructurada. La principal ventaja de la POO es que cada objeto (clase) hace su cometido y si otra clase necesita algo de ella pues que se lo pida.

Para adoptar este hábito lo más rápido posible, aconsejo utilizar la sentencia **package** que lo que hace es agrupar las clases bajo un mismo paquete. Esto también nos ayudará a hacer un diseño de la aplicación bastante limpio, pues agruparemos las clases en paquetes según la finalidad que tengan, quedando nuestra aplicación mucho más definida y estructurada si posteriormente queremos hacer modificaciones.

IMPORTANTE: Las clases que pertenecen a un paquete HAY que ponerlas en un directorio del mismo nombre. Por ejemplo, si tenemos las clases A, B, C, D que pertenecen al paquete "misejemplos", entonces, deberán estar alojadas en el directorio "misejemplos" para que el núcleo de Java no nos de un error al compilar la aplicación.

Una última cosa a tener en cuenta es que, aunque no es obligatorio, debemos poner cada clase que creemos en ficheros diferentes y en la primera línea de cada uno poner el paquete al que pertenecen, **empaquetar** no significa poner todas las clases de una aplicación en un disquete, ni meter todas las clases de un paquete en un fichero, empaquetar es referenciar cada uno de los ficheros (con una única clase) al paquete al que pertenecen por la palabra reservada package.

import

Con esta sentencia **importamos** las clases de los paquetes que necesitamos para nuestro código, las de la API de Java y las de creación propia.

Pero, ¿es obligatorio importar las clases? Pues no, pero para que el compilador no te diera el error de que no ha encontrado la clase **tal** tendrías que poner todo el path de los paquetes de las clases que emplees , por ejemplo, para crear un objeto Frame habría que hacer:

```
java.awt.Frame frame = new java.awt.Frame();
```

No te parece un peñazo? ;-), así que usemos import y nos bastará con:

```
import java.awt.*;
. . .
Frame frame = new Frame();
```

Puede parecer que con una clase solo no merece la pena, pero creeme que tu emplearás más de una clase del JDK y que entonces merecerá mucho la pena.

Algo que es muy habitual encontrar en los fuentes de java que consultamos es que importan el paquete entero (todas sus clases) mediante el comodín *, por ejemplo, import javax.swing.* pero si bien es lo más cómodo te recomiendo que cargues solo las clases que necesitas, por experiencia propia te aseguro que las asimilas antes pues te ves obligado a importarlas.

encapsulación.

Esta es una de las propiedades de la POO y hace referencia a la visibilidad que le concedemos a una clase, un método o una variable, es decir, define su accesibilidad. Si te has mirado cualquier tutorial sobre la POO habrás visto que existen diferentes tipos de restricciones pero para que no te hagas un lío a la hora de definir las restricciones de una clase y de sus métodos y atributos ten en cuenta la siguiente distinción:

- la parte pública: accesible para el resto de las clases. Para ello se usa la sentencia public.
- la parte privada: A ella sólo pueden acceder los métodos de la clase. Para ello se usa la sentencia private o protected.
- la clase principal SIEMPRE tiene que ser pública, de lo contrario si la importamos el compilador nos cantará un error.
- todos los atributos (variables) que no necesiten ser vistos por otras clases se declararán como private o a lo sumo como protected.

class

Su propio nombre indica que lo que viene a continuación de ella es la clase que vamos a crear, no tiene más historia así que pasamos a otra cosa.

extends

Aquí aparece otra propiedad de la POO, el concepto de herencia. Mediante esta palabra reservada heredamos los métodos y atributos de la clase madre de la que heredamos. Así si heredamos de la clase Frame heredamos todos sus métodos pero además los de la jerarquía de clases que están por encima de Frame que han ido heredando unas de otras.

Para que entiendas esto, consulta la clase Frame de las especificaciones y verás al principio de la página el árbol de herencias de dicha clase. Detrás de la relación de métodos que tiene esta clase verás todos los métodos a los que puede acceder una clase que herede de Frame.

Por definición, la herencia no es múltiple, es decir, solo se puede **extender** a única clase pero esta limitación la podemos saltar implementando las interfaces que deseemos de las que proporciona el JDK como se verá en capítulos posteriores.

Por último, decir que las clases heredadas las podemos redefinir a nuestro gusto para que se ajusten a nuestras necesidades o ampliarlas.

super

Esta palabra está íntimamente relacionada con los conceptos de herencia y de encapsulación. Aunque una subclase incluye todos los miembros de su super-clase, ésta no puede acceder a aquellos miembros de la super-clase que se hayan declarado como private.

Cuando la superclase de la que heredamos guarda sus datos miembro como privados, la clase heredera no los podrá inicializar, es por ello que cuando una subclase necesita referirse a su inmediata superclase utilizamos la palabra reservada super. Su uso puede tener dos objetivos:

1. Llamar al constructor de la superclase.
2. Acceder a un método o variable de la clase madre por haberlo escondido la subclase, por ejemplo, la subclase define una variable con el mismo nombre que la superclase.

this

Esta palabra se emplea dentro de cualquier método para referirse al objeto sobre el que estamos trabajando, es decir, this es siempre una referencia al objeto sobre el cual el método se invocó. Un caso en el que se aprecia bastante bien la utilidad de esta palabra es cuando resolvemos un conflicto de nombres idénticos, pero la realidad es que la mayoría de las veces, su uso es redundante pues nos lo podríamos ahorrar. No obstante, su empleo clarifica aun más el código y mi recomendación es que siempre se utilice.

Poniéndolo todo junto.

En el siguiente ejemplo se intenta recoger todo lo aprendido en este artículo, es un ejemplo muy sencillo que calcula volúmenes.

```
/*
fichero Volumen.java en directorio cap2. Obviamente, lo tenemos que tener
recogido en el classpath pues si no el núcleo de java nos
 cantaría el error "java.lang.NoClassDefFound cap2/Volumen".
*/

package cap2;

public class Volumen {

  private double x;
  private double y;
  private double z;

  public Volumen(double x, double y, double z) {

    //al usar this resolvemos el conflicto de nombres
    this.x = x;
    this.y = y;
    this.z = z;
  }

  public double calculaVol() {
    return this.x * this.y * this.z;
    /*
     en cambio, su uso aquí es redundante, podríamos haber
     hecho simplemente x*y*z
    */
  }
}

//fichero Demo.java

import cap2.Volumen;

class Demo extends Volumen {

  Demo(double x, double y, double z) {

    super(x,y,z); //llamamos al constructor de la clase Volumen
    /*
     inicializamos las variables de esta clase.Observa, que al ser declaradas
     como private las variables en la clase Volumen, ésta
     es la única manera de acceder a ellas.
    */
  }

  private void printVol() {
    System.out.println("El volumen total es : " + this.calculaVol());
  }

  public static void main(String args[]) {
    Demo volumen = new Demo(20,30,20);
    volumen.printVol();
  }
}
```

Leo Suarez está actualmente realizando el Proyecto Fin de Carrera en Java para obtener el título de Ingeniero Superior de Telecomunicaciones por la Universidad de Las Palmas de Gran Canaria.
Cuando no está proyectando o escribiendo para javaHispano aprovecha para disfrutar con la novia y los amigos del estupendo clima de esa maravillosa isla.
Para cualquier duda o tirón de orejas, e-mail a: leo@javahispano.com