## 1.1. Trabajo Práctico s/Inyección de Dependencias

### 1.1.1. Introducción

Los objetos se componen de atributos. Muchos de estos atributos son, en sí mismo, objetos; que se componen de otros atributos.

Por ejemplo: un objeto auto se compone, entre otras cosas, de 1 motor, 4 ruedas y 4 butacas. Un objeto rueda se compone, entre otras cosas, de 1 cubierta, 1 llanta y 4 bulones y así sucesivamente

Para instanciar un objeto auto primero tendremos que instanciar todos los objetos de los cuales depende el auto. Y para instanciar a cada uno de estos objetos, primero tendremos que instanciar a cada uno de los objetos de los que éstos, a su vez, dependen.

### 1.1.2. Políticas de instanciación de los objetos dependientes

Para instanciar un objeto que depende de otros objetos existen diversas opciones.

1. Que el mismo objeto sea el que se ocupe de instanciar, uno por uno, los objetos de los cuales depende. Ya sea mediante un *factory method* o, directamente, instanciándolos manualmente.
2. Que seamos nosotros, como programadores, los que instanciemos a esos objetos y se los pasemos como parámetros en el constructor o mediante los métodos de acceso.
   Se dice que los objetos que generan una dependencia fuerte se deben pasar por constructor. Por ejemplo, el caso del objeto motor para el objeto auto; pues, el auto no puede funcionar sin el motor.
   En cambio, las dependencias débiles deben pasarse mediante los métodos de acceso. Este sería el caso del objeto matafuego para el objeto auto. Un auto perfectamente puede funcionar sin un matafuego; luego, opcionalmente, podemos adquirir un matafuego y tenerlo en el auto.

### 1.1.3. Motor de inyección de dependencias

Cuando hablamos de un motor de inyección de dependencias pensamos en una herramienta que realice, por nosotros, la tarea de instanciar y asignar los objetos dependientes, los objetos de los cuáles dependen los objetos dependientes y así sucesivamente. El *framework* String realiza esta tarea.

Por ejemplo, en el siguiente código vemos cómo la clase `Factory` (que será nuestro motor de inyección de dependencias) funciona como una factoría de objetos a la cual, en este caso, le pedimos una instancia de la clase `Auto`.

```
public static void main()
{
   Auto a = Factory.getObject(Auto.class);
}
```

Pero `Factory` no sólo instanciará a `Auto`. También instanciará todos los atributos de los cuales depende la clase `Auto` y todos los atributos de los cuales dependen los atributos de los que depende la clase auto; y así instanciará e inyectará todo el árbol de dependencia de los objetos.

Veamos el código de la clase `Auto`.

```
package demo;

@Component
public class Auto
{
   @Injected
   private Motor motor;

   @Injected(count=4)
   private Rueda[] ruedas;

   @Injected(count=4)
   private List<Butaca> butacas;

   @Injected(implementation=AutostereoSonyImple.class)
   private Autostereo autostereo;

   // ...
}
```

Vemos que con la *annotation* `@Injected` le indicamos a `Factory` cuáles son los atributos que debe instanciar e inyectar en la variable que se encuentra justo debajo de la anotación.

A su vez, `@Injected` tiene parámetros (opcionales) que podemos utilizar para indicar pautas sobre la instanciación de los objetos dependientes.

- `count`, indica cuántas instancias de la clase anotada se deben crear y asignar. Este atributo sólo es válido al aplicarlo sobre *array*s o listas. En el caso de aplicarlo sobre un atributo que es de tipo `List` (que es una *interface*) se podrá agregar el atributo `implementation` para indicar qué implementación concreta queremos utilizar; por *default* será `ArrayList`.
- `implementation`, sólo es válido si se aplica sobre un atributo cuyo tipo es una *interface*; entonces `implementation` debe indicar cuál es la implementación concreta que `Factory` debe instanciar e inyectar.
  Si el tipo de dato de un atributo es una *interface* y existe una única implementación de la misma, entonces podremos omitir el parámetro `implementation`. Si hubiera más de una implementación y dicho parámetro no está especificado entonces `Factory` arrojará una *exception* indicando que debemos especificar qué implementación debe utilizar.
- `singleton`, permite indicar que se debe mantener una única instancia de la clase que queremos inyectar en el atributo anotado.

En el código de `Auto` vemos que la clase está anotada con `@Component`. `Factory` sólo inyectará objetos cuyas clases tengan esta indicación.

Observemos que, en el caso particular de `Auto` y así como está planteado el ejemplo, no es necesario que la clase esté anotada con `@Component`. Pues `Factory` no está inyectando en ningún lado una instancia de esta clase.