**SINGLETON**

Patrón de diseño de aplicaciones que tiene como objetivo asegurar que solo hay una instancia u objeto por clase y un punto de acceso global a ella.

Es útil cuando se quiere tener acceso a un único recurso físico común, por ejemplo, un fichero abierto. También se utiliza cuando otras clases necesitan acceder a dato en concreto perteneciente a un objeto que debe ser único y global.

Ventajas:

* Mayor control sobre el acceso a instancias
* El espacio de nombres es más reducido, por lo que el código es más claro
* La herencia está optimizada al existir menor cantidad de objetos

Spring se basa en este patrón de diseño

**PROTOTYPE**

Patrón de diseño de aplicaciones, opuesto a Singleton, que permite que una clase tenga varios objetos o instancias, creados mediante clonación. Se crea un objeto modelo o prototipo a partir del cual se clonan los demás y se varían sus características.

Es útil cuando se necesita trabajar con varios objetos que tienen atributos repetidos o coumnes.

Ventajas:

* La aplicación puede crear y eliminar objetos en tiempo de ejecución
* Permite ahorrar tiempo y recursos puesto que la creación de objetos se basa en la clonación. Siempre es más óptimo clonar un objeto que crear uno de nuevo.

Spring puede trabajar con este principio de diseño a pesar de que utiliza Singleton por defecto.

**CICLO DE VIDA DE UN BEAN**

1. Se inicia el contenedor (Spring)
2. Crea una instancia del vean
3. Se inyectan las dependencias
4. El contenedor procesa el vean internamente
5. El contenedor deja el vean listo para usar, a nuestra disposición
6. Se apaga el contenedor

Dentro del ciclo de vida del bean surge la necesidad de realizar tareas antes de que el bean sea puesto a nuestra disposición y después de que el bean ya se ha utilizado. Se consigue con los métodos **INIT** (instrucciones que el bean relizará antes de estar listo) y **DESTROY** (instrucciones que se ejecutan después de ser utilizado):

1. Se inicia el contenedor (Spring)
2. Crea una instancia del bean
3. Se inyectan las dependencias
4. El contenedor procesa el bean internamente
5. Método INIT
6. El contenedor deja el vean listo para usar, a nuestra disposición
7. Se apaga el contenedor
8. Método DESTROY

En el método INIT se pueden cargar dependencias de otros beans, activar conexiones con otros servicios…

En el método DESTROY se puede liberar recursos, cerrar conexiones con otros servicios…

**JAVA ANNOTATIONS**

Son “etiquetas” que sirven para añadir metadatos a las clases. Su uso permite sustituir el uso de archivos XML. Se pueden procesar tanto en tiempo de compilación como en tiempo de ejecución. Facilitan el uso de beans y simplifica el código del archivo de configuración XML.

[Metadatos: Conjunto de datos que describen el contenido o la esencia de un objeto.]

Spring escanea, en segundo plano, todas las clases en busca de anotaciones (recursividad).

* **@Component(“nombreBean”)**: Crea el bean de una clase java.
* **@Autowired:** Para inyectar la dependencia en un constructor, método SETTER o cualquier otro método que llame a la interfaz de la dependencia o en un campo de clase que apunta a la dependencia. Hace que Spring escanee todo el proyecto en busca de clases que implementen la interfaz de la dependencia. No es necesaria esta anotación si solo hay un constructor en el bean que se pide a Spring.
* **@Qualifier(“nombreBean”)**: Para especificar el ID de un bean. Se usa cuando existen varias clases que implementen la interfaz de la dependencia (varios bean que pueden inyectar la dependencia). La anotación se coloca debajo de @Autowired.
* **@Scope(“nombrePatron”)**: Permite establecer el patrón de diseño (Singleton – Prototype) sin entrar en el código del HTML. El nombre de patron se introduce en minúsculas (singleton / prototype).
* **@PostConstruct**: Indica la parte de código o instrucciones que se ejecutarán después de la construcción del bean pero antes de su puesta a disposición oejecución (misma función que el método Init). No funciona con el scope prototype (Spring no maneja el ciclo de vida completo en el scope prototype).
* **@PreDestroy:** Indica la parte de código o instrucciones que se ejecutarán después de la ejecución del bean y antes de que este sea destruido (misma función que el método Destroy). No funciona con el scope prototype.
* **@Configuration:** Permite configurar el contenedor Spring con código Java en lugar de emplear un archivo de configuración XML. En este caso se genera un archivo de configuración .CLASS.
* **@ComponentScan(“paqueteAescanear”):** Se coloca a continuación de la anotación @Configuration para indicar el paquete que Spring debe escanear en busca de anotaciones.
* **@Bean:** Permite la creación o definición de un BEAN dentro del fichero de configuración .CLASS mediante código Java.
* **@PropertySource(“classpath:archivoPropiedades”):** Permite especificar la ruta del fichero externo de propiedades dentro del fichero de configuración .CLASS.
* **@Value(“${nombrePropiedad}”):** Permite inyectar el valor de las propiedades desde un fichero externo.

**ARCHIVO DE CONFIGURACIÓN XML**

Hay que configurar el archivo XML indicando que lo realice:

<context:component-scan base-package=*"paqueteAescanear"*>

</context:component-scan>

Cuando encuentra la anotación, Spring registra automáticamente el Bean en el contenedor, sin tener que escribir código manualmente en el XML.

**ARCHIVO DE CONFIGURACIÓN .CLASS**

Crear una nueva clase y emplear las anotaciónes **@Configuration** y **@ComponentScan**

@Configuration

@ComponentScan("paqueteAEscanear")

**public** **class** Config {

(…)

}

A su vez, en la clase MAIN hay que cargar el archivo de configuración:

**public** **static** **void** main(String[] args) {

AnnotationConfigApplicationContext contexto =

**new** AnnotationConfigApplicationContext(Config.**class**);

(…)

}