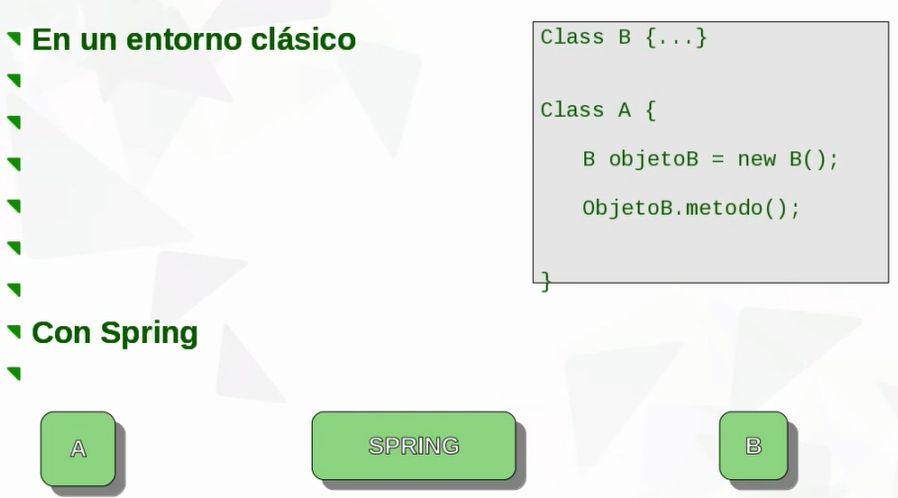
1. [Inversión de control](http://campus.dicotraining.com/mod/book/view.php?id=15)

Es la característica principal, el paradigma de [inversión de control](http://campus.dicotraining.com/mod/book/view.php?id=15) popularizado por [Martin Fowler](http://martinfowler.com/bliki/InversionOfControl.html), es también conocido como "Principio de Hollywood", por su famosa frase "no nos llame, nosotros le llamaremos", que resume claramente el propósito.

En un entorno orientado a objetos tradicional la llamada entre objetos se hace de manera directa, un objeto A invoca a un objeto B, sin embargo en este paradigma aparece un tercero (en este caso Springframework) que se encarga de realizar las llamadas entre objetos. Así cuando A quiere un objeto B, no lo llama directamente, sino que lo hace a través del intermediario que es Spring.



Las ventajas de este sistema es que tanto A como B no tiene porqué saber el uno del otro y se cargarían bajo demanda y por tanto no disponen de una **asociación fuerte**, lo que hace que exista un **bajo acoplamiento y mayor facilidad de reutilización y actualización.**

### Creación de objetos en spring

En spring para crear un objeto es necesario tener un fichero xml de configuración de donde el contenedor obtendrá la información necesaria para instanciar los objetos así como sus dependencias y ciclo de vida. En el xml de definición se crearan los beans del contexto que harán referencia a clases bean o POJOs (Plain Old Java Object).

Un elemento bean en el xml de configuración contiene la información para que el contenedor sepa crear el bean, detalles del ciclo de vida y las dependencias.

Todos los beans deben disponer de un identificador, es decir una  definición de alto nivel. Esto se realiza con el atributo ID="nombreDadoAlBean".

<bean id="nombreDadoAlBean"/>

Pero IDREF tienen limitaciones, por lo que para escribir un ID correcto es necesario atender a una serie de reglas:

1. debe comenzar con una letra seguida de caracteres alfanuméricos
2. o bien puede comenzar con un guión bajo \_ sin espacios en blanco
3. también puede comenzar con el signo dos puntos (:). Sin embargo se desaconseja esto ya que puede dar lugar a equívocos pues también se usa con los xmlnamespaces.

  Hay ocasiones en las que estas reglas son difíciles de aplicar, por ejemplo en una aplicación web donde el bean debe de ser accedido como parte de la url en ese caso tendríamos que poner algo como:

<bean id="/miBean">

Pero eso no es admitido por IDREF y por tanto el parser fallaría al pasarle el XML. En este caso, podemos hacer uso del atributo name, que permite crear tantos nombre como queramos separados por comas y no tiene las restricciones del ID, por lo que puede comenzar por cualquier caracter.

Otro escenario es cuando sobre el mismo bean, varios recursos hacen uso del mismo, pero con nombres diferentes, por ejemplo distintos equipos de trabajo, o configuraciones. En estos casos también tiene sentido usar el atributo name:

<bean id="miBean" name="/miBeanEnURL,dataSource1,baseDeDatosOracle" />

Básicamente el modelo de desarrollo basado en POJOs debe cumplir una serie de reglas:

* Son clases Java simples, autodescriptivas, esto implica no poner nombres extraños sólo conocidos por el programador
* Deben implementar el interfaz Serializable
* Su campos de clase debe ser privados
* Sus métodos de acceso debe ser públicos y hay de lectura y de escritura los métodos de lectura comenzarán por get y los escritura por set
* Deben utilizar nomenclatura [**camelCase**](http://es.wikipedia.org/wiki/CamelCase)
* Debe contener al menos un constructor vacío

A continuación se muestra un ejemplo:

*import java.util.Serializable*

*public Class Usuario implements Serializable {*

*private String nombre;*

*public String getNombre(){*

*this.nombre=nombre;*

*}*

*public void setNombre(String nuevoNombre){*

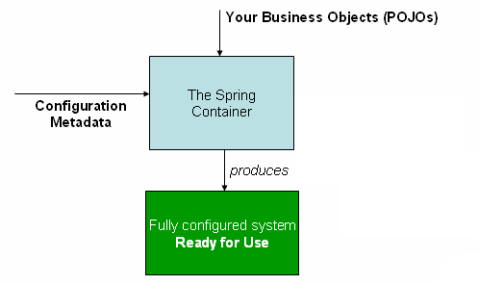
*this.nombre=nuevoNombre;*

*}*

*}*

Para referenciar nuestro bean con el POJO dentro de la etiqueta bean tenemos la propiedad class donde indicaremos la clase a la que dicho bean representa.

Con estas dos entradas, Spring levantará un **contexto de aplicación** ([ApplicationContext](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/context/ApplicationContext.html" \o "ApplicationContext" \t "_blank)). Este contexto es un interfaz cuya implementación se encargará de instanciar de nuestros beans, configurarlos y conectarlos:

[](http://csilgo.files.wordpress.com/2012/04/container-magic.png)

Ejemplo: **Proyectos**

**+ CosasDeBeans**

**+ HerenciaPropiedades**

**Herencia Propiedades** (Proyecto HerenciaPropiedades)

Teniendo dos clases sin herencia se puede poner por configuración que exista una herencia omas bien herencia de propiedades,

Puedo crear un vean que no existe abstracto (AlternateBean.xml), heredara conjunto de propiedades. La factoría por debajo hace el setMessage2 y coge el valor aunque no exista la clase.

Si esta creado el bean como abstract no podrás crear objeto y por tanto nunca se podrá crear una instancia de él.

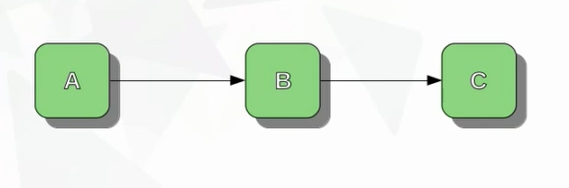
**Ejemplo: Proyecto HerenciaPropiedades**

**Ejercicio**: Generar un servicio que muestre los datos de una persona heredando el género de un bean abstracto que de la propiedad de masculino o femenino dependiendo del género.

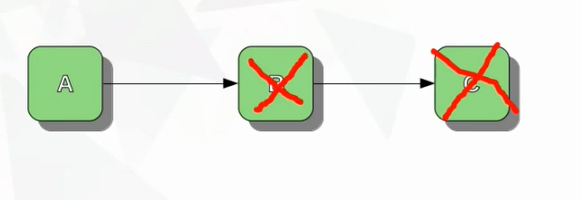
1. [Inyección de dependencias](http://campus.dicotraining.com/mod/book/view.php?id=26)

Como extensión de la inversión de control, disponemos de la [inyección de dependencias](http://campus.dicotraining.com/mod/book/view.php?id=26), así si el objeto A quiere al objeto B, pero con unos datos particulares y no como un objeto vacío, Spring puede "inyectar" valores en las propiedades públicas del objeto B, llenándolo de datos y así pasar el objeto B al objeto A tal y como este último lo necesita. Para realizar este trabajo se hace uso del [API de Reflexión](http://docs.oracle.com/javase/tutorial/reflect/index.html), y de una serie de buenas prácticas como es el desarrollo de clases bajo el modelo [POJO](http://es.wikipedia.org/wiki/Plain_Old_Java_Object) (Plain Old Java Object u objeto java de toda la vida).

Por ejemplo en la conexión a una base de datos



Si C esta caído, no tiene sentido que B exista para dar datos a A, por tanto B y C no se crearan.



### Inicialización de objetos

Spring no garantiza el orden de ejecución de los beans declarados en el xml de configuración. Así si un bean tarda más que otro en iniciarse y el segundo necesita del primero para completar su estado, pueden ocurrir algunas inconsistencias en el trabajo con dichos objetos. Sólo porque aparezca antes en la declaración de la configuración no es garantía suficiente para que Spring cree un objeto antes que otro.

Tenemos en cuenta  que además de la inicialización predeterminada por constructor vacío, luego es necesario satisfacer todas las dependencias de ese objeto, bien por métodos set, o como argumentos del constructor. Nuevamente Spring no puede garantizar que la [inyección de dependencias](http://campus.dicotraining.com/mod/book/view.php?id=26) se realice de forma ordenada y tal y como aparece en el código, puesto que un objeto referenciado a ser inyectado puede tardar en instanciarse más que otro en otra propiedad. Esto nuevamente puede dar lugar a algunas inconsistencias en la creación de los objetos.

Sabemos que un objeto no puede ser gestionado por spring completamente hasta que se han satisfecho todas sus dependencias y si alguna de tales dependencias tiene a su vez otras, primero serán inicializadas las últimas para ser inyectadas en el segundo para una vez esté completo ser inyectado en el primero. Luego entonces se trata de la creación de pequeños workflows o cadenas de objetos. También sabemos que no todas las propiedades deben ser iniciadas forzosamente si no son declaradas en el archivo de configuración. ¿Pero cómo podemos manejar una situación a nuestro gusto?

Spring nos proporciona algunos argumentos para controlar este workflow.

* Por ejemplo, a veces necesitamos que un bean sea cargado antes que otro, para ello existe una propiedad llamada **depends-on**:

<bean id="oracle-jdbc-driver" class="oracle.jdbc.driver.OracleDriver "/>

<bean id="accesoAbaseDeDatos" depends-on="oracle-jdbc-driver" class="..." >

...

</bean>

El ejemplo anterior es el caso donde un Driver siempre debe estar cargado antes de realizar una conexión, podemos resolverlo con **depends-on.**

En otras ocasiones puede no interesarnos crear una instancia de un objeto desde el principio si no únicamente realizarlo cuando se invoca. Para realizar esta tarea tenemos la propiedad lazy-init que por defecto tiene un valor de false, pero poniéndole el valor a true no instanciara el objeto hasta que no se le invoque

<bean id=*"lazy"* class=*"com.curso.MyLazy"* **lazy-init =*”true”*/>**

Para que un objeto no se instancie desde el principio con propiedad lazy-init que por defecto es false si se pone valor true no se instanciará hasta que no se crea.

### Inyección por propiedades

Disponer de métodos setXXX permite hacer que las clases sean modificables, o comúnmente mutables, mientras que si eliminamos todos los métodos set de una clase está sería inmodificable o inmutable.

Si una clase dispone de métodos setXXX, podemos introducir los valores esperados por esos métodos, eso es lo que se llama [inyección de dependencias](http://campus.dicotraining.com/mod/book/view.php?id=26). Spring realizará una comprobación de tipos, así si lo que se espera como parámetro es un Integer no se puede introducir un Boolean, y producirá una excepción, también Spring realizará la conversión automáticamente siempre que pueda, así, si lo esperado es un String y se pasa un int, spring lo convertirá a cadena de caracteres y a la inversa si es posible.

Desde el fichero de configuración XML esto se realiza añadiendo nodos hijos a la etiqueta <bean>, llamados <property> donde se especifica el nombre de la propiedad sin la partícula set. Por ejemplo sobre la clase Persona definida anteriormente:

<bean id="miPersona" class="es.adama.spring.Persona">

<property name="nombre" value="David" />

</bean>

A veces en una propiedad tenemos una asociación a otro objeto, entonces en lugar de usar el atributo **value**, usaremos el atributo **ref**, así:

<bean id="miPersona" class="es.adama.spring.Persona">

<property name="direcciones" >

<ref bean="objetoDirecciones"/>

</property>

</bean>

Value permite pasar literales, e incluso tipos que spring se encarga de convertir dinámicamente, siempre que pueda. Con Ref podemos pasar otros beans que estén bajo la ejecución de Spring y dispone de varios argumentos:

* <ref bean="">
* <ref local="">
* <ref parent="">

Los atributos local, bean o parent son mutuamente exclusivos y deben referenciar al ID de otro bean.

* Local hacer referencia a otro bean que debe existir en el mismo XML y a su ID, no al atributo name.
* Bean, puede estar en el mismo fichero, en otro fragmento XML y en la misma factoría o en otra parte.
* Parent: el bean debe de venir desde una factoría padre, se usa raramente y solo en casos donde puede existir un conflicto de nombres entre factoría padre e hija.

### Pros

* hace más fácil manejar valores por defecto u opcionales
* Las propiedades son heredadas por las subclases (mientras no sean privadas)
* Requieren menos discusión a nivel de código que los argumentos de los constructores
* Requieren menor trabajo de documentación javadoc ya que no existe duplicación
* En tiempo de ejecución se pueden usar para búsquedas y comparaciones vía reflexión (nombres visibles)
* Permiten establecer y recuperar el estado actual (con métodos get)
* Pueden ser mutables (a través de métodos set)
* Esto permite cambiar la dependencia

### Contras

* Puede ocurrir una inconsistencia del estado si una propiedad no ha sido inicializada
* No hay manera de especificar el orden de llamada de los métodos set
* Puede usarse un método ínit()

### Inyección por constructor

Las clases pueden ser inicializadas por propiedades, pero también pueden disponer de un constructor con argumentos y Spring puede inicializar los objetos inyectando las dependencias a través del constructor.

 NOTA: Hay que recordar que si creamos un constructor con argumentos, obligatoriamente debemos tener también un constructor vacío.

En este caso disponemos de la etiqueta:

*<constructor-arg>*

y del argumento index en caso de que existan varios argumentos en un constructor:

<beans>

<bean id="prediccionServicio" class="es.adama.spring.PrediccionServicioImpl">

<constructor-arg index="0">

<ref local="prediccionDao"/>

</constructor-arg>

<property name="numeroMaximoDeIntentos"><value>2</value></property>

</bean>

<bean id="prediccionDao" class="es.adama.spring.PrediccionDaoImpl">

</bean>

</beans>

Como se puede apreciar en el ejemplo anterior la inyección por constructor y por propiedad no son excluyentes.

### Pros

* Las configuraciones pasadas mediante constructores no son mutables
* A menos que también expongan sus propiedades
* Proporciona mejores mecanismos para la inmutabilidad
* Proporcionando las propiedades como finales
* Asegura que un objeto válido ha sido construido antes de usarlo

### Contras

* Constructor induce a tener varios constructores
* A veces con llamadas internas a otro
* Tantas variaciones, tantos argumentos
* Se puede convertir en inmantenible
* Los constructores no son heredados.
* Esto implica a crear constructores similares en las subclases o constructores “tontos” para llamar a las superclases
* Los nombres de los constructores a veces no intuyen su funcionalidad

En general parece que por constructor es mejor para escenarios de inicialización simple y con constructores con pocos argumentos, mientras que por JavaBean parece más manejable y se usa con menos trabajo.

### Ciclo de vida

Los beans gestionados por Spring tienen un ciclo de vida y sirve para que spring pueda invocar en el orden adecuado los distintos beans y satisfacer sus dependencias.

El ciclo de vida es el siguiente:

1. El contenedor inicializa la definición al inicio del contenedor
2. Cada propiedad es un valor actual o una referencia a otro bean
3. El contenedor realizará tantas validaciones como sean necesarias
4. El parser lanzará un excepción si no puede validar contra el DTD o el XMLSchema
5. Spring puede lanzar una excepción si la definición no es válida lógicamente. Por ejemplo dos propiedades mutuamente exclusivas
6. Si una dependencia no puede ser satisfecha se recibirá un error sólo cuando se necesite inyectar dicha dependencia
7. Las propiedades o argumentos que se refieren a otro bean forzarán al contenedor a crear y obtener ese bean primero
8. Cada propiedad o argumento debe poder ser convertido de cualquier tipo al tipo que el método esté esperando
9. La configuración XML disponen de elementos para especificar colecciones
10. Spring puede usar reflexión para ver los argumentos o propiedades que tienen el bean y construir una lista de dependencias (autowiring)

Esto implica que si un bean debe ser inyectado en otro bean, Spring creará una cadena y no instanciará la cadena completa si alguno de los beans no puede ser creado o satisfecho alguna de sus propiedades correctamente.

**Ejemplo 1**. Proyecto InyeccionDependencias, con IoDSetBeans.xml se pasa el valor como propiedad a través del set, mientras que en IoDConsBeans.xml se pasa como argumento del constructor.

Paso de listas, colecciones, Map, **Ejemplo2**. InyeccionColecciones

Anotaciones

Es una forma de añadir [metadatos](http://es.wikipedia.org/wiki/Metadatos) al código fuente Java que están disponibles para la aplicación en [tiempo de ejecución](http://es.wikipedia.org/wiki/Tiempo_de_ejecuci%C3%B3n).

Las Anotaciones permiten al programador declarar en su código fuente cómo debe comportarse el software.

Añadir metadatos en tiempo de ejecución puede causar sobrecarga adicional de memoria.

**Ejercicio vaca**, pasar listado de vacas al establo para mostrarlas.

Ver SpEL

Y

Manejador de eventos de contexto(Proyecto ContextEvents)

La factoría registra los listeners y los invoca automáticamente cuando suceden los eventos

1. Programación Orientada a Aspectos (AOP)

AOP es un paradigma de programación que busca poder tratar de extraer como módulos aquel **código que resuelve problemas transversales** a los componentes de una aplicación.

Para ver dónde podemos aplicar AOP, debemos identificar esas funcionalidades genéricas que se utilizan en muchos puntos diferentes. Algunas de las más comunes son: logs, transacciones, seguridad, cachés, manejo de errores, monitorización…

Un ejemplo sencillo, imaginemos que para todos los métodos del servicio PatientServiceImpl se requiere que el usuario tenga un determinado rol:

|  |  |
| --- | --- |
|  | public class PatientServiceImpl implements PatientService{  ...     public Patient findById(Integer patientId) {     if(!hasPermission(SecurityContext.getPrincipal())) {       throw new AccessDeniedException();     }     return patientDao.findById(patientId);   }   public List findByRoom(Integer roomId) {     if(!hasPermission(SecurityContext.getPrincipal())) {       throw new AccessDeniedException();     }     return patientDao.findByRoom(roomId);   }    } |

Este código tiene dos problemas:

* **Mezcla y acopla conceptos**que son diferentes: los métodos findById y findByRoom deben preocuparse de encontrar los datos, no de gestionar la seguridad. Esto es lo que conocemos como “**code tangling**” (enredo de código).
* La solución a un mismo problema aparece**repetida varias veces en diferentes partes de la aplicación**: el código que comprueba el rol de usuario está repetido en diferentes puntos. Esto es lo que conocemos como “**code scattering**” (dispersión de código). En una aplicación pequeña puede que esto no suponga un gran problema pero a medida que nuestra aplicación crece, es muy costoso mantener código disperso cuya funcionalidad, además, está entremezclada con otras.

AOP es una solución muy elegante para eliminar estos problemas que sigue en estos tres pasos:

1. Implementa la lógica de negocio de tu aplicación.
2. Implementa aspectos que resuelvan los problemas transversales a tu aplicación.
3. Enlaza estos aspectos en los puntos en los que sean necesarios.

Existen ciertos conceptos de AOP que utilizaremos mucho y que conviene conocer:

* **Join Point:**un punto en la ejecución de un programa (llamada a un método, asignación…)
* **Pointcut:**expresión que selecciona uno o más Join Points.
* **Advice:**código que queremos que se ejecute cuando un Join Point es seleccionado por un Poincut.
* **Aspect:**módulo que encapsula pointcuts y advice.

Existen diferentes formas de trabajar con aspectos en Spring AOP  ([mediante XML](http://www.mkyong.com/spring3/spring-aop-aspectj-in-xml-configuration-example/), mediante anotaciones AspectJ, utilizando configuración por Java, utilizando configuración por XML, con CGLib, etc…)  Nosotros nos centraremos en Spring AOP con anotaciones AspectJ, utilizando proxies que envuelven a las clases y ejecutan los aspectos. Hay que tener en cuenta que, por defecto, Spring AOP sólo permite definir aspectos sobre clases que implementen algún interfaz ya que para ello [utiliza proxies dinámicos de JDK](http://docs.oracle.com/javase/1.3/docs/guide/reflection/proxy.html), para hacerlo sobre otras clases debemos [utilizar CGLib y configurar la propiedad proxy-target-class a true.](http://static.springsource.org/spring/docs/3.0.0.M3/reference/html/ch08s06.html)

### Creación de un aspecto

Siguiendo con el ejemplo del hospital, veamos cómo podemos aplicar diferentes aspectos a nuestra aplicación. El primer objetivo que nos marcaremos será escribir un log cada vez que busquemos los pacientes de una habitación. Lo primero que haremos será permitir el uso de aspectos mediante la etiqueta: <aop:aspectj-autoproxy/>.

El siguiente paso será definir nuestro aspecto. Veamos una posible implementación:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Aspect  public class MonitoringAspect {      @Before("execution(\* com.hospital.services.PatientService.findByRoom(\*))")    public void findingPatients() {       System.out.println("Buscando pacientes en habitación...");    }    } |

En este caso, el poincut está expresando “método findByRoom del servicio PatientService” mientras que el advice sería “escribe Buscando pacientes en habitación…”, ambos conceptos formarían nuestro aspecto. Más adelante veremos las diferentes formas de definir pointcuts.

Una vez implementado, lo añadiremos al contexto como cualquier otro bean y le diremos a Spring que lo registre como aspecto utilizando la etiqueta anterior:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ...  <aop:aspectj-autoproxy>    <aop:include name="monitoringAspect"/>  </aop:aspectj-autoproxy>  <bean id="monitoringAspect" class="com.hospital.aspects.MonitoringAspect"/>  ... |

A partir de ahora, cada vez que alguna clase de nuestra aplicación invoque al servicio de pacientes para buscar los pacientes asignados a una habitación obtendremos un mensaje.

El siguiente paso sería obtener información acerca del punto de ejecución de la aplicación (JoinPoint). Para ello, pasaremos un parámetro de este tipo a nuestro Advice, que, a partir de ese momento, tendrá acceso a información relativa al punto de ejecución de la aplicación, por ejemplo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Before("execution(\* com.hospital.services.PatientService.findByRoom(\*))")  public void findingPatientsWithJoinPoint(JoinPoint joinPoint) {    String name = joinPoint.getSignature().toShortString();    Object roomId = joinPoint.getArgs()[0];    System.out.println(name + " buscando pacientes en habitación "+roomId);  } |

### Definición de Pointcuts

Para definir pointcuts en Spring AOP utilizaremos la notación de AspectJ y seguirán el siguiente patrón: execution(<patrón>) Para que un método sea “interceptado” por un aspecto, deberá cumplir el patrón que indiquemos. Además, podemos componer pointcuts utilizando && (and), || (or) y ! (not).

Los patrones los definiremos siguiendo esta estructura:

**[Modificadores] TipoRetorno [Clase] NombreMétodo ([Parámetros]) [throws TipoExcepción]**

Veámoslo con el pointcut anterior:  \* (cualquier tipo de retorno)com.hospital.services.PatientService (interfaz PatientService del paquete com.hospital.services) .findByRoom(método findByRoom) (\*) (aceptando un parámetro de cualquier tipo).

**Hay que recordar que los métodos que vayan a ser seleccionados por un pointcunt deben ser visibles (públicos).**Algunos ejemplos más:

* **execution(void send\*(String))**: cualquier método visible que comience por send, tome un String como único parámetro y cuyo tipo de retorno sea void.
* ***execution(\* send(\*))***: cualquier método visible llamado send que tome como parámetro un parámetro de cualquier tipo.
* **execution(\* send(int, ..))**: cualquier método visible llamado send que tome al menos un parámetro de tipo int. En este caso “..” indica 0 o más.
* **execution(void org.ejemplo.MessageServiceImpl.\*(..))**: cualquier método visible de la clase org.ejemplo.MessageServiceImpl que tenga como tipo de retorno void.
* **execution(void org.ejemplo.MessageService+.send(\*))**: cualquier método visible con nombre send de las clases del tipo org.ejemplo.MesssageService, incluyendo hijos e implementaciones, que reciban un único parámetro de cualquier tipo y tengan void como tipo de retorno.
* **execution(@javax.annotation.security.PermitAll void send\*(..))**: cualquier método visible que comience por send y que esté anotado con la anotación @PermitAll.
* **execution(\* org.ejemplo.\*.impl.\*.\*(..))**: cualquier método visible de cualquier clase de cualquier paquete impl situada dos escalones por debajo de org.ejemplo en la jerarquía de paquetes.
* **execution(\* org.ejemplo..impl.\*.\*(..))**: cualquier método visible de cualquier clase de cualquier paquete impl situada cualquier nivel por debajo por debajo de org.ejemplo en la jerarquía de paquetes. En este caso “..” indica que puede haber 0 o más directorios en la jerarquía de paquetes.

A la hora de definir pointcuts, también podremos hacer uso de los Named Pointcuts, donde la definición de estos se separa y se nombra para que posteriormente pueda ser reutilizada, combinada y externalizada (una buena práctica si trabajamos con mucho Pointcuts es definirlos todos en una única clase externa):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class MyPointCuts {      @Pointcut("execution(\* com.hospital..\*Dao.\*(..))")    public void daoMethods() {}      @Pointcut("execution(\* com.hospital..\*Service.\*(..))")    public void daoMethods() {}    }    public class MyAspect {      @Before("org.example.MyPointCuts.daoMethods")    public void myAdviceForDaos() {     ...    }      @After("org.example.MyPointCuts.daoMethods        ||org.example.MyPointCuts.serviceMethods")    public void myAdviceForDaosAndServices() {      ...    }    } |

### Implementación de Advices

Veamos cómo se implementan los advices que ejecutan el código deseado para nuestro aspecto. Lo primero que tenemos que decidir es cuándo queremos que se ejecute este código, tenemos varias opciones:

* **@Before:**es la que utilizamos en el ejemplo anterior. El advice se ejecutará **antes** de la ejecución del método interceptado. **Si el advice lanza una excepción NO se ejecutará el método.**
* **@AfterReturning:** el advice se ejecutará **después de que el método haya terminado con éxito**. Opcionalmente podremos acceder al objeto devuelto por el método, utilizando la propiedad returning (implica que el método debe devolver un objeto de este tipo).

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @AfterReturning(value="execution(\* com..PatientService.find\*(..))",  returning="patients")  public void returningPatientsWithJoinPoint(JoinPoint joinPoint, List patients) {    String name = joinPoint.getSignature().toShortString();    Object roomId = joinPoint.getArgs()[0];    System.out.println(name + " devolviento "+patients.size()+  " pacientes en habitación "+roomId);  } |

* **@AfterThrowing:**el advice se ejecutará **después de que el método lance una excepción del tipo indicado**. Esto no impedirá que la excepción se siga propagando aunque, opcionalmente, podremos lanzar una excepción de un tipo diferente.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @AfterThrowing(value="execution(\* com..PatientService.find (..))",  throwing="incorrectResultSizeDataAccessException")  public void returningPatientsWithJoinPoint(JoinPoint joinPoint,  IncorrectResultSizeDataAccessException incorrectResultSizeDataAccessException) {    String name = joinPoint.getSignature().toShortString();    Object roomId = joinPoint.getArgs()[0];    System.out.println("@AfterThrowing -> " + name  + " ocurrió un error al buscar pacientes en habitación "+roomId);  } |

* **@After:** el advice se ejecuta independientemente de si el método terminó con éxito o lanzó una excepción.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @After(value="execution(\* com..PatientService.find\*(..))")  public void returningWithJoinPoint(JoinPoint joinPoint) {    String name = joinPoint.getSignature().toShortString();    Object roomId = joinPoint.getArgs()[0];    System.out.println("@After -> " + name  + " terminó de buscar pacientes en habitación "+roomId);  } |

* **@Around:**este advice es un poco diferente a los demás, puesto que lo que hace es “rodear” la ejecución del método y, por lo tanto, seremos nosotros los encargados de decidir cuando la ejecución debe continuar y devolver el objeto esperado o lanzar una excepción. Para ello, recibiremos un el JoinPoint como tipo ProceedingJoinPoint y haremos uso del método proceed(). En este caso, sí que podremos interceptar una excepción y hacer que no se propague.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Around(value="execution(\* com.hospital.\*.PatientDao+.findByRoom(\*))")  public List aroundWithJoinPoint(ProceedingJoinPoint joinPoint)  throws Throwable {    String name = joinPoint.getSignature().toShortString();    Object roomId = joinPoint.getArgs()[0];    System.out.println("@Around -> " + name  + " va a buscar pacientes en la habitación "+roomId);    List patients = (List) joinPoint.proceed();    System.out.println("@Around -> " + name  + " terminó de buscar en la habitación "+roomId);    return patients;  } |

Por último, también tenemos la posibilidad de acceder a la información de contexto utilizando la notación del Advice en lugar de recibir directamente el objeto de tipo JoinPoint, lo que nos evita tener que hacer casting a los tipos de objetos que esperamos (JointPoint trabaja con Object):

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Before("execution(\* com.hospital.\*.PatientService.findByRoom(\*))  && target(patientService) && args(roomId)")  public void findingPatientsWithJoinPoint(PatientService patientService,  Integer roomId) {    String name = patientService.toString();    System.out.println("@Before (selección contexto) -> " + name +  " buscando pacientes en habitación "+roomId);  } |