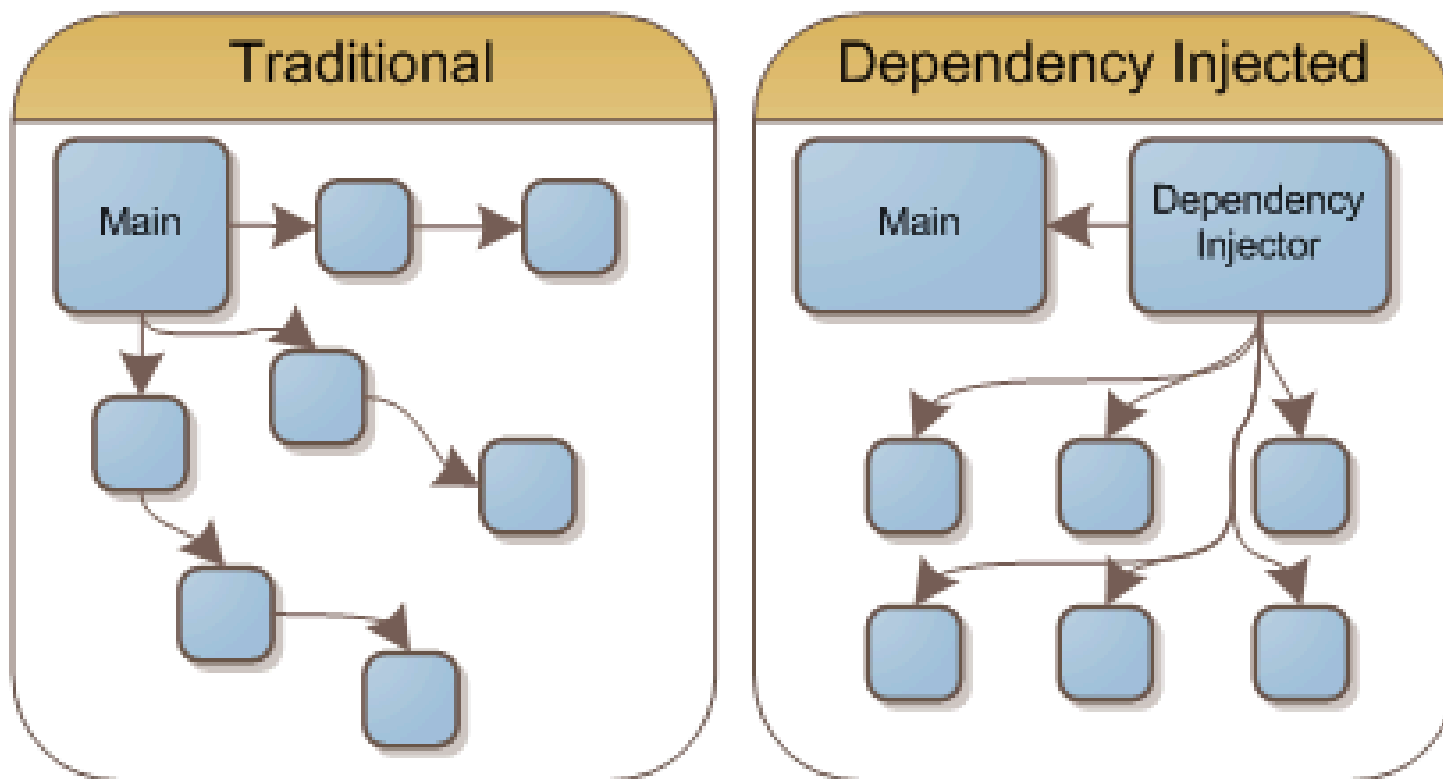


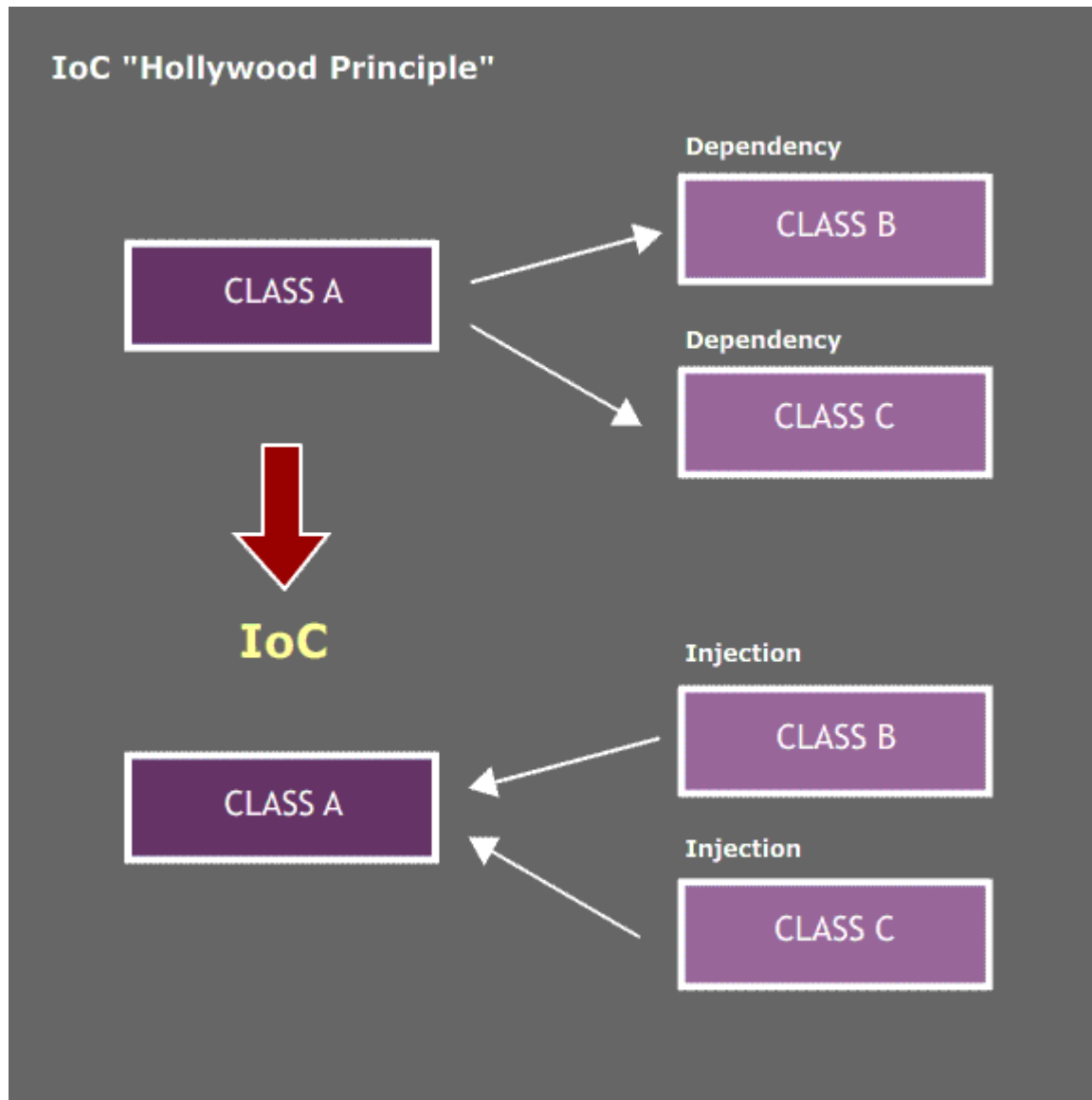
Contenedor de Inyección de Dependencias en Contexto

# **CONTEXT AND DEPENDENCY INJECTION (CDI)**

# Inyección de dependencias e inversión del control

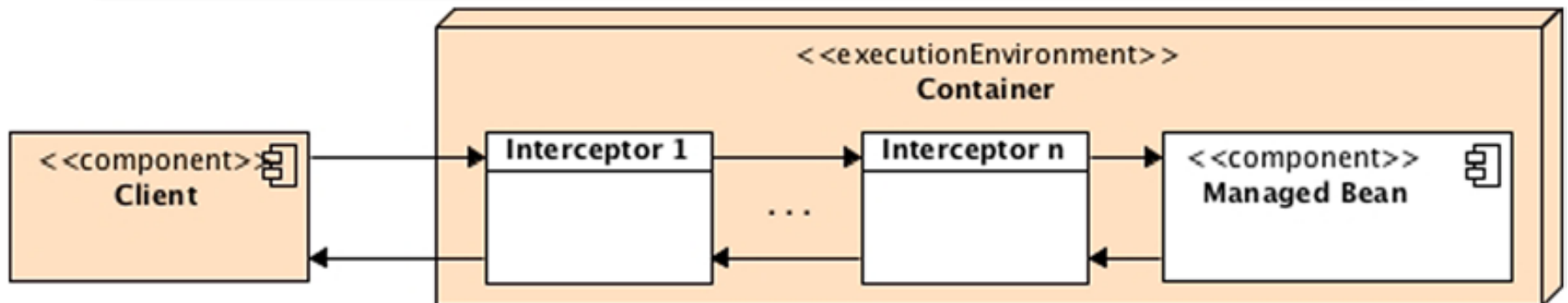


# Inyección de dependencias e inversión del control



# Interception

- La mayoría de las aplicaciones tienen código común repetido a través de components. Esto se puede solucionar con interceptors y AOP.
- Es el punto de entrada para AOP.
- Es un paradigma de programación que separa “concerns” que atraviesan toda la aplicación en componentes de código separados que se pueden utilizar para ser aplicados en todos los lugares donde sean necesario.
- Lo “concerns” pueden ser técnicos (por ejemplo dejar un log) o de negocio (realizar chequeos adicionales en determinadas circunstancias).



## CDI

- En la primer versión de J2EE se introdujo de manera poco flexible el concepto de IoC.
  - El contenedor toma el control del código de negocio, y provee servicios como transacciones, seguridad y otros.
  - Estos servicios estaban disponibles solo de manera interna en el contenedor pero no podían ser aprovechados por el desarrollador.
- En JavaEE5, se incorporó la posibilidad de “inyectarlos” en el código propio mediante anotaciones, pero con restricciones:
  - Solo se podían inyectar algunos recursos específicos administrados por el contenedor y en clases administradas por el contenedor.
- En JavaEE6 se introdujo un API completa para poder lograr que cualquier clase sea administrada por un motor de inyección de dependencia y lograr que estas clases tengan uso extendido a lo largo de todas las API's

## Origen

- En 2006 inspirados por frameworks ya existentes como Spring, Guice y fundamentalmente Seam, Gavin King se transformó en el líder de la especificación JSF 299 llamada “Web Beans”.
  - El objetivo era hacer que los EJB sean “back-beans” de JSF.
  - Luego se renombró a Context and Dependency Injection, y se basó en la JSR 330 (Dependency Injection for Java 1.0 que trajo la anotación @Inject y que fue liderada por Rod Johnson).
  - Estas dos especificaciones se complementan y no se puede usar una sin la otra en JavaEE.
- “Dependency Injection for Java” define un conjunto de anotaciones usadas para inyección.
- CDI le da semántica y servicios como manejo de eventos, decoradores, interceptors y permite extensibilidad.

## CDI

- CDI está basado en el concepto de “bajo acople y fuerte tipado”. Esto significa que si bien un bean CDI está debilmente acoplado con las clases que lo utilizan, se basa en una definición de tipo.
- El desacople se puede incrementar incluso usando algunas características que potencian a CDI como interceptores, decoradores y eventos.
- Una de las principales ventajas que se logró en JavaEE7, al estandarizar CDI como la tecnología base para declarar beans de capa web y de otras capas, es unificar los ámbitos de ciclo de vida de los beans.

## Ordenando Conceptos.. beans beans y beans

- Java SE tiene JavaBeans, Java EE posee Enterprise JavaBeans.
- Y además Java EE tiene otros componentes como Servlets,
- SOAP web services, RESTful web services, entidades y obviamente Managed Beans CDI.
- Y por supuesto POJOs.
  - POJOs son clases Java planas que se ejecutan en la JVM
  - JavaBeans son solo POJOs que siguen determinados patrones (convenciones de nombrado, constructor por defecto, setters y getters)
  - El resto de los componentes de JavaEE también siguen sus patrones definidos y son ejecutados en un contenedor que les da (les inyecta) servicios (transacciones, pooling, seguridad, etc).



## Que son los Managed Beans

- Managed Beans son objetos planos, que soportan un conjunto básico de servicios, sobre los cuales se puede extender y conectar con toda la potencia de JavaEE.
  - Fundamentalmente soportan inyección de recursos, administración del ciclo de vida, e interceptación.
- Fueron introducidos en Java EE 6.
- Se pueden integrar fácilmente con el resto de la plataforma:
  - Podemos transformar un EJB en un Managed Bean, el cual sería un bean con servicios extras y a su vez ser un back bean de JSF.
- Además estos objetos soportan interacción con EL, calificadores, y control completo del ciclo de vida con estado.

Potencialmente cualquier clase Java que tenga un constructor por defecto y se ejecute dentro del servidor (web o ejb) es un bean CDI

## Objetos que pueden ser inyectados

- (Casi) cualquier clase Java
- EJB Session beans
- Recursos Java EE : data sources, Java Message Service topics, queues, connection factories
- Persistence contexts (JPA EntityManager objects)
- Campos @Producer
- Objetos retornados por métodos productores.
- Referencias a servicios web
- Interfaces remotas.

# Implementaciones

- La implementación de referencia de CDI es Weld, un proyecto open source de JBoss.
- Otras implementaciones importantes son
  - Apache OpenWebBeans
  - CanDi (de Caucho).

## Descubriendo Beans

- Cuando se despliega una aplicación JavaEE, CDI busca beans dentro de un “empaquetado de archivos de beans”.
  - Un empaquetado de archivos de bean es cualquier modulo que contiene beans que puedan ser manejados e inyectados por CDI.
  - Existen 2 tipos: implícitos y explícitos.
- Un archivo de beans explícito es un archivo que contiene un descriptor “beans.xml” que puede ser un archivo vacío, o contener el número de la versión (1.1 en JavaEE7) y el modo de descubrimiento (atributo bean-discovery-mode).
  - El modo de descubrimiento puede ser :
    - all: todos los tipos en un archivo de empaquetado son beans.
    - annotated: solo los que tienen las anotaciones de beans serán considerados para inyección.
    - none: deshabilita CDI.

# Descubriendo Beans - Ejemplos

archivo vacio

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee">  
</beans>
```

1. **<beans**
2. xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5. http://xmlns.jcp.org/xml/ns/javaee/**beans\_1\_1.xsd**"
6. **bean-discovery-mode="all">**
7. **</beans>**

## Descubriendo Beans

- Un archivo de beans implícito es un archivo que contiene beans anotados con un ámbito, y no tiene archivo “bean.xml” o tiene un archivo “bean.xml” con el atributo `bean-discovery-mode` con el valor “annotated”.
- CDI solo puede manejar e inyectar beans anotados con un ámbito en un paquete de archivos implícitos.
- Para una aplicación web, el descriptor `beans.xml` si está presente debe estar en el directorio `WEB-INF`.
- Para módulos JAR de EJB de existir debe estar en el directorio `META-INF`.

- Las clases con la anotación *@Vetoed* no serán administradas por CDI

```
@Vetoed
public class SimpleGreeting implements Greeting {
    ...
}
```

- beans.xml* puede definir *exclude filters* para excluir algunas clases de ser administradas por CDI.

```
<beans ...>
<scan>
<exclude name="...">
...
</exclude>
<exclude name="org.sample.beans.*">
<if-class-available name="org.sample.beans.SimpleGreeting"/>
</exclude>
</scan>
</beans>
```

## Anatomía de un bean CDI

- De acuerdo a la especificación CDI 1.1, el contenedor trata a un bean como un bean CDI si satisface las siguientes condiciones
  - NO es una inner class estática.
  - Es una clase concreta o está anotada con `@Decorator`
  - Tiene un constructor por defecto sin parámetros o declara un constructor anotado con `@Inject`.
- Luego un bean puede tener opcionalmente
  - Un ámbito
  - Un nombre EL
  - un conjunto de interceptors y operaciones opcionales de administración del ciclo de vida.

**Cambio radical respecto a JavaEE6 !**



## Puntos de Inyección

- Una instancia de un bean puede ser inyectada en un atributo, en un método o en un constructor usando `@Inject`.

```
1. public interface Greeting {  
2.     public String greet(String name);  
3. }  
4. public class SimpleGreeting implements Greeting {  
5.     public String greet(String name) {  
6.         return "Hello" + name;  
7.     }  
8. }  
9. @Stateless  
10. public class GreetingService {  
11.     @Inject Greeting greeting;  
12.     public String greet(String name) {  
13.         return greeting.greet(name);  
14.     }  
15. }
```

En JavaEE6 tenia que tener obligatoriamente ámbito y ser `@Named`. Hoy solo es necesario si lo usamos en JSF.

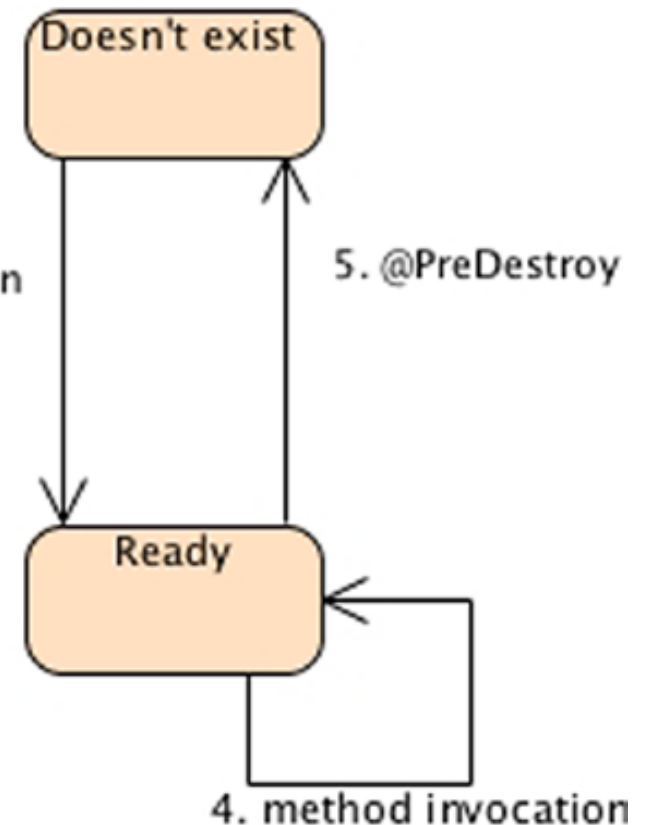
inyección de un bean

## Proceso de inicialización

1. El constructor por defecto o el que esté anotado con `@Inject` se ejecuta.
2. Luego se inicializan todos los campos anotado con `@Inject`.
3. Luego todos los métodos anotados con `@Inject`
4. Luego se ejecuta el método `@PostConstruct` en caso de que esté presente.

Como no podemos crear instancias con el `new`, se lo pedimos al contenedor mediante la inyección, y le decimos que antes de inyectarlo ejecute el método anotado con `@PostConstruct`

1. new instance
2. dependency injection
3. `@PostConstruct`



## Otros puntos de inyección

- Inyección en un método setter

```
@Inject  
public setGreeting(Greeting greeting) {  
    this.greeting = greeting;  
}
```

- Inyección en un constructor

```
@Inject  
public SimpleGreeting(Greeting greeting) {  
    this.greeting = greeting;  
}
```

## Inyección de recursos del servidor – **Nuevo en JavaEE7 !**

- La inyección de recursos originalmente surge en Java EE 5 con las anotaciones
  - @Resource
  - @PersistentUnit
  - @EJB.
- No obstante esto esta limitado a inyectar estos recursos en algunos componenetes.
- **Con CDI 1.1 se puede inyectar usando la anotación @Inject cualquier elemento en cualquier clase. Y debe ser de uso preferido por sobre los anteriores**

# Inyección por defecto

- Si se define una interface (Greeting) y una única implementación (GreetingSimple), es obvio que sucede cuando realizamos la inyección en un atributo del tipo Greeting, se creará una instancia de la única clase que la implementa.
- Esto se denomina inyección por defecto.
- Si no se le aplica ninguna anotación o “calificador a un bean” se asume que el bean está anotado con `@javax.enterprise.inject.Default`.

```
1. public interface Greeting {  
2.     public String greet(String name);  
3. }  
4. public class SimpleGreeting  
   implements Greeting {  
5.     public String greet(String name) {  
6.         return "Hello" + name;  
7.     }  
8. }
```

Si se define un bean sin calificador automáticamente se le asigna el calificador `@Default`

```
1. public class FancyGreeting implements  
   Greeting {  
2.     public String greet(String name) {  
3.         return "Nice to meet you, hello" + name;  
4.     }  
5. }
```

# Inyección por defecto

1. @Stateless

2. public class GreetingService {

3. @Inject Greeting greeting;

4. public String greet(String name) {

5. return greeting.greet(name);

6. }

7. }

Cual se inyecta???

- Una solución es decir que SimpleGreeting es el bean por defecto y FancyGreeting es una alternativa con un calificador.
- Crear el calificador:
  1. `@Qualifier`
  2. `@Retention(RUNTIME)`
  3. `@Target({METHOD, FIELD, PARAMETER, TYPE})`
  4. **public** `@interface Fancy` {
  5. }

```

1. public interface Greeting {
2.     public String greet(String name);
3. }
4. @Default
5. public class SimpleGreeting
   implements Greeting {
6.     public String greet(String name) {
7.         return "Hello" + name;
8.     }
9. }
```

```

1. @Fancy
2. public class FancyGreeting implements
   Greeting {
3.     public String greet(String name) {
4.         return "Nice to meet you, hello" + name;
5.     }
6. }
```

# Uso del calificador

```
1. @Stateless
2. public class GreetingService {
3.     @Inject @Fancy Greeting greeting;

4.     @Inject Greeting greeting2;

5.     public String sayHello(String name) {
6.         return greeting.greet(name);
7.     }

8. }
```



## Ámbitos de CDI

- **CDI** es claramente un framework de Inyección de Dependencias. Pero también de manejo de contexto (de hecho la C en CDI).
- Cada objeto administrado por CDI tiene un ámbito bien definido y un ciclo de vida atado a un contexto específico.
- En Java el ámbito de un POJO es simple, se crea una instancia de una clase, con new, se lo utiliza, y luego cuando se la deja de apuntar el GC la limpia.
- Con CDI un bean es vinculado a un contexto y esta instancia permanece “viva” hasta que el bean es destruido por el contenedor en el momento en que considera apropiado.
- No tenemos forma de quitar un bean de un contexto. Simplemente lo dejamos de usar.

## Ámbitos de CDI

- Los ámbitos en la capa web están bien definidos
  - application – session – request
- En la capa de servicio no es necesaria esa diferenciación pero CDI los trae para que puedan ser usados según la necesidad.
- Los ámbitos que define CDI son
  - @ApplicationScoped
  - @SessionScoped
  - @RequestScoped
  - @ConversationScoped
  - @Dependent

## Ámbitos de CDI

- *Application scope* (@ApplicationScoped):
  - la instancia mantendrá su estado (el valor de sus atributos) durante todo el tiempo que dure la aplicación.
  - Habrá una sola instancia de dicho bean para toda la aplicación.
  - Los datos son compartidos por todas las instancias de servidor que accedan a un bean de este ámbito
  - Recomendado para clases de utilidades o para almacenar datos compartidos, siempre que tengamos cuidado con el acceso concurrente.

## Ámbitos de CDI

- *Session scope* (@SessionScoped): es un ámbito que se extiende a través de múltiples requerimientos HTTP, todos asociados a la misma sesión de un usuario.
  - El bean es creado durante la sesión, y finaliza su ciclo de vida cuando la sesión culmina.
- Se usa este ámbito para objetos que son necesarios a lo largo de toda la sesión del usuario.
- No abusar de este ámbito porque puede causar problemas de memoria.
- *Request scope* (@RequestScoped): Corresponde a un objeto que se destruye cuando termina un requerimiento HTTP y que se crea cuando se necesita en el marco de cada petición HTTP.

## Ámbitos de CDI

- El problema con los ámbitos anteriores es que muchas veces necesitamos tener una instancia de un objeto cuyos valores perduren más que el tiempo de un request pero no tanto como una sesión.
- *Conversation scope* (`@ConversationScoped`): es un ámbito que se extiende a través de múltiples invocaciones y con límites que se pueden definir programáticamente según las necesidades del desarrollador.
- *Dependent pseudo-scope* (`@Dependent`): El ciclo de vida se ata al del bean en el que es inyectado. Es el ámbito por defecto que usa CDI.

## Ámbitos de CDI

- Todos los bean de sesión o de conversación, pueden necesitar en algún momento, guardar el estado del bean que se encuentra en memoria en disco para reusar la instancia para otro bean. Es por ello que deben implementar la interface “Serializable” para poder resolver esta situación.
  - @SessionScoped
  - public class CompraBean **implements Serializable** {...}
- Los beans “Dependent” nunca son compartidos entre diferentes clientes, o usados en diferentes puntos de inyección, como si lo son los de sesión, aplicación o conversación.
  - Dependen de un bean en particular.
  - Un bean “dependant” es instanciado cuando el bean del que depende es creado y destruido cuando el mismo es destruido.

## Ejemplo

- Ejemplo anotando un bean el scope “@Dependent”
  1. @Dependent @ThirteenDigits
  2. public class IsbnGenerator implements NumberGenerator {...}
- Como el ámbito por defecto es “dependendent” podemos usar solo un calificador y dejarlo por defecto, por lo que el código anterior sería equivalente al que se muestra a continuación :
  1. @ThirteenDigits
  2. public class IsbnGenerator implements NumberGenerator {...}
- Los ámbitos pueden “mezclarse” sin problemas. Esto es, un bean de sesión puede contener uno de conversación o viceversa, en un request podemos acceder a datos de una sesión o conversación.
- Todo lo administra CDI.

## Análisis del ámbito de Conversación

- El ámbito de conversación es diferente que los demás.
  - Guarda el estado asociado con el usuario y se expande en múltiples request, y es demarcado programáticamente por la aplicación.
- Un ámbito de conversación puede ser usado cuando en un proceso tenemos en claro cuando comienza y termina un proceso y cual es el flujo que sigue.
  - Ejemplo típico: un wizard o un formulario de compra que lleva varios pasos, y en el paso N+1 se necesita la información del paso anterior.
    - Permite que cada “pantalla” tenga su propio back-bean.
- Los objetos vinculados a un ámbito de conversación no son finalizados automáticamente sino a través del API `javax.enterprise.context.Conversation`.



## Ejemplo de conversacion

- Creación de un usuario en una aplicación web, que consta de 3 pasos.
  - Primer paso: información de login.
  - Segundo paso: datos personales.
  - Tercer paso, confirmar datos y crear la cuenta.

**1. @ConversationScoped**

- 2. public class CustomerCreatorWizard implements Serializable {
- 3. private Login login;
- 4. private Account account;

**5. @Inject**

- 6. private CustomerService customerService;

**7. @Inject**

**8. private Conversation conversation;**

- 9. public void saveLogin() {
- 10. **conversation.begin();**
- 11. login = new Login();
- 12. // Sets login properties
- 13. }

```
1. public void saveAccount() {  
2.     account = new Account();  
3.     // Sets account properties  
4. }  
5. public void createCustomer() {  
6.     Customer customer = new Customer();  
7.     customer.setLogin(login);  
8.     customer.setAccount(account);  
9.     customerService.createCustomer(customer);  
10.    conversation.end();  
11. }  
12. }
```

## ***Conversation API***

- **Method Description**

- void begin(): marca el inicio de la conversación actual.
- void begin(String id): marca el inicio de una conversación específica
- void end(): termina una conversación
- String getId(): obtiene el identificador de la conversacion
- long getTimeout(): obtiene el timeout
- void setTimeout(long millis):setea el timeout, en donde si la conversación no está active n millis se finaliza
- boolean isTransient(): permite determinar si la conversación fue previamente iniciada o no.

## CDI y Expression Language

- Una de las claves de CDI es que permite traer los beneficios de la capa transaccional a la capa web.
- Una de las ventajas de CDI es que el acople no depende del nombre que le pongamos a nuestras clases sino de los calificadores que usemos. Esto lo reduce fuertemente.
- El problema es que no lo podemos usar directamente en facelets mediante EL.
- Por defecto CDI no le asigna nombre a los componentes por lo que si deseamos darle un nombre para poder emplearlos en EL debemos utilizar un calificador que trae incorporado CDI que se denomina `@Named`.
  - Esto permite que el nombre que le asignemos sea resuelto por EL.

## Ejemplo

1. `@Named` ← nombre por defecto → “bookService”

```
2. public class BookService {
3.     private String title, description;
4.     private Float price;
5.     private Book book;
6.     @Inject @ThirteenDigits
7.     private NumberGenerator numberGenerator;
8.     public String createBook() {
9.         book = new Book(title, price, description);
10.        book.setIsbn(numberGenerator.generateNumber());
11.        return "customer.xhtml";
12.    }
13. }
```

```
<h:commandButton value="Send email" action="#{bookService.createBook}"/>
```

## CDI y Expression Language

- El calificador `@Named` permite acceder a un bean a través del nombre.
  - Si deseamos cambiar el nombre por defecto de un bean podemos pasar dicho valor como parámetro a la anotación `@Named`

```
@Named("myService")  
public class BookService {...}
```



```
<h:commandButton value="Send email"  
action="#{myService.createBook}"/>
```

## Obligatoriedad del archivo beans.xml

- El archivo beans.xml **no es obligatorio** de incluir en nuestras aplicaciones empaquetadas para usar CDI.
- CDI se activará solo, aunque no esté el archivo beans.xml, si anotamos a una clase con una anotación de ámbito.
- De lo contrario, si incluimos el archivo “beans.xml” y como parámetro de descubrimiento le indicamos “all” casi todas las clases no estáticas, serán administradas por CDI y podrán ser inyectadas.



## CDI - Alternativas

- En CDI si una interface es implementada por 2 beans, si deseamos inyectar una instancia de alguno de los dos beans, no era posible a menos que los diferenciemos.
- Para diferenciarlos creamos nuestras propias anotaciones, que se denominan “Qualifiers” .
- No obstante algunas veces necesitamos que se inyecte una instancia u otra, dependiendo de condiciones del contexto de ejecución.
  - Por ejemplo en un entorno de testing nos interesan que se genere un bean con datos mock y en uno de producción un bean “real”.
- Para estos escenarios usar alternativas.

## CDI - Alternativas

- Un bean anotado con el calificador *“javax.enterprise.inject.Alternative”* es una alternativa.
- Por defecto las alternativas están deshabilitadas y deben ser habilitadas en el archivo beans.xml para hacerlas disponibles para instanciación e inyección.

```
1. public interface Operacion {  
2.     public int operar(int a, int b);  
3. }
```

```
1. @OperarProducto  
2. public class OperacionProducto  
   implements Operacion, Serializable{  
3.     @Override  
4.     public int operar(int a, int b) {  
5.         return a*b;  
6.     }  
7. }
```

```
1. @OperarProducto @Alternative  
2. public class OperacionProductoMock  
   implements Operacion, Serializable{  
3.     @Override  
4.     public int operar(int a, int b) {  
5.         return 6;  
6.     }  
7. }
```

## CDI - Alternativas

```
1. public interface Operacion {  
2.     public int operar(int a, int b);  
3. }
```

```
1. public class OperacionSuma implements  
   Operacion, Serializable{  
2.     @Override  
3.     public int operar(int a, int b) {  
4.         return a+b;  
5.     }  
6. }
```

```
1. @Alternative  
2. public class OperacionSumaMock  
   implements Operacion, Serializable{  
3.     @Override  
4.     public int operar(int a, int b) {  
5.         return 100;  
6.     }  
7. }
```

Tenemos 4 implementaciones de la interface Operacion, OperacionSuma (default), OperacionProducto (marcada con un calificador) y OperacionSumaMock (marcada como alternativa) y OperacionProductoMock (marcada como alternativa y con el calificador producto)

## CDI - Alternativas

- Dado el siguiente código:
  1. `@Inject`
  2. `private Operacion operacion1;`
  3. `@Inject @OperarProducto`
  4. `private Operacion operacion2;`
  5. `public String operacion1(){`
  6. `this.resultado = this.operacion1.operar(operador1, operador2);`
  7. `}`
  8. `public String operacion2(){`
  9. `this.resultado = this.operacion2.operar(operador1, operador2);`
  10. `}`
- ¿Que implementación se usará en cada caso?
  - La implementación original, es decir `OperacionSuma` y `OperacionProducto`.

## CDI Alternativas

- Para usar la configuración alternativa debemos indicar, para cual implementación deseamos que se inyecte una alternativa (en beans.xml)

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5.         http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
6.     bean-discovery-mode="all">
7.     <alternatives>
8.         <class>ar.edu.utn.frsf.ofa.javaee7.tp.ejemplo.ejemplo10.OperacionProductoMock</class>
9.     </alternatives>
10. </beans>
```

Usará en este caso OperaciónSuma por defecto y OperacionProductoMock cuando la inyección esté acompañada del calificador @OperacionProducto

## CDI Alternativas

- La anotación `@Alternative` permite empaquetar muchas implementaciones de una interface, en distintos beans, con el mismo calificador y habilitarlos selectivamente al cambiar el descriptor de despliegue, el cual se puede configurar en función del entorno donde esté desplegado.
  - Podemos armar distintos “.war” para entorno de testing, de desarrollo, de integración, de producción, de capacitación
- Obtenemos **polimorfismo de despliegue** (el **mismo código** se ejecuta de **distinta forma** dependiendo **donde se despliegue**).

## CDI Productores

- Hasta aquí hemos visto que podemos inyectar una instancia de una clase pojo, que CDI administre y transforme en un bean CDI.
  - Es decir mediante CDI podemos inyectar una instancia de un bean nuestro en otro bean.
- ¿Y si queremos inyectar otro tipo de datos, como Primitivos, Strings, Date, Arreglos, u otro tipo de datos que no son automáticamente administrados por CDI?
  - Todas estas clases están empaquetadas en un archivo del JRE llamado “rt.jar” que no posee un “beans.xml”
- Recurriríamos para esto a una característica de CDI que son los atributos productores y los métodos productores.

## CDI Productores

- Un atributo productor, es el modo más sencillo y representa un campo de un bean que genera un objeto.
  - No podemos aplicarle lógica.
  - Son útiles para inyectar recursos JEE (datasources, JMS, WS, etc)
- Un método productor genera un objeto que puede ser inyectado.
- Situaciones típicas de uso:
  - Cuando deseamos inyectar en un cliente un objeto que no es en si mismo un bean.
  - Cuando el tipo concreto de objeto que queremos inyectar **varía en tiempo de ejecución**.
  - Cuando el objeto que vamos a inyectar requiere de alguna inicialización personalizada que no se puede realizar en el constructor.

Tantos los atributos como los métodos productores son anotados `@Produces` (`javax.enterprise.inject.Produces`).



## Ejemplo productor CDI

```
1. public class NumberProducer {  
2.     @Produces @ThirteenDigits  
3.     private String prefix13digits = "13-";  
4.     @Produces @ThirteenDigits  
5.     private int editorNumber = 11;  
6.     @Produces @Random  
7.     public double random() {  
8.         return Math.abs(new Random().nextInt());  
9.     }  
10. }
```

productor

```
@Qualifier  
@Retention(RUNTIME)  
@Target({FIELD, TYPE, METHOD})  
public @interface ThirteenDigits { }
```

```
@Qualifier  
@Retention(RUNTIME)  
@Target({FIELD, TYPE, METHOD})  
public @interface Random { }
```

```
1. @ThirteenDigits  
2. public class IsbnGenerator implements  
   NumberGenerator {  
3.     @Inject @ThirteenDigits  
4.     private String prefix;  
5.     @Inject @ThirteenDigits  
6.     private int editorNumber;  
7.     @Inject @Random  
8.     private double postfix;  
9.     public String generateNumber() {  
10.         return prefix + editorNumber + postfix;  
11.     }  
12. }
```

## Productor y ámbito

- Los métodos productores deben ser vistos como beans independientes por lo que por ejemplo tienen su propio ámbito (no heredan el del bean al que pertenecen !!! ).
- En CDI el ámbito por defecto de un bean es “@Dependent” por lo que se le crea una nueva instancia por defecto.
  - Anteriormente existía la anotación @New que obligaba siempre a producir una nueva instancia pero ha sido deprecada.
- Si deseamos que el productor use un bean de otro ámbito podemos indicarlo.

```
1. @Produces
2. @SessionScoped
3. public MessageSender getEmailMessageSender(
4.     EmailMessageSender emailMessageSender){
5.     return emailMessageSender;
6. }
```

Tendremos una sola  
instancia de  
MessageSender por sesión  
HTTP

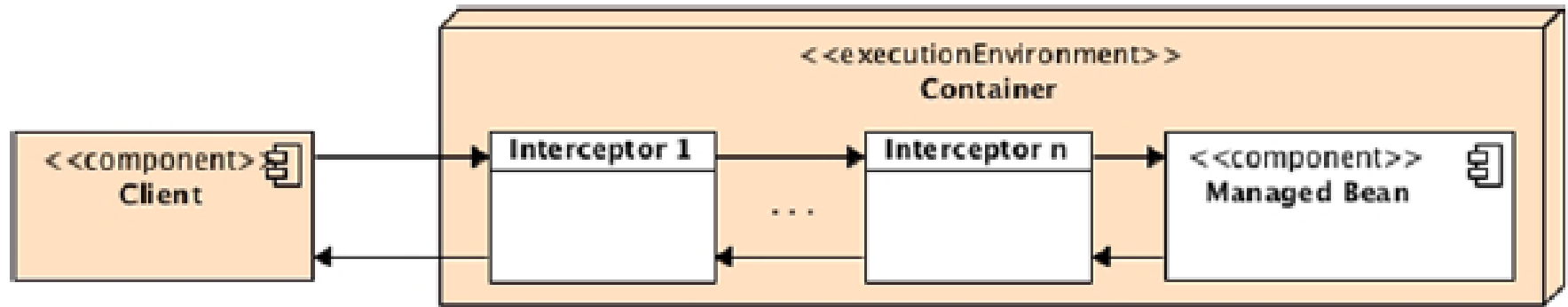
## API de InjectionPoint

- Cuando tenemos métodos productores, podemos hacer que los mismos reciban un parámetro, denominado InjectionPoint que nos da acceso a la siguiente información:
  - Type getType() el tipo requerido a inyectar
  - Set<Annotation> getQualifiers() los calificadores del punto de inyección (es decir del atributo que recibirá el objeto inyectado).
  - Bean<?> getBean(): obtiene el bean en donde se inyectará
  - Member getMember() obtiene el campo en donde se inyectará

```
1. public class LoggingProducer {  
2.     @Produces  
3.     private Logger createLogger(InjectionPoint injectionPoint) {  
4.         return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName());  
5.     }  
6. }
```

## CDI – Interceptores

- Los interceptors son una característica que nos permite utilizar AOP en una aplicación JavaEE, y dan la posibilidad de implementar “cross-cutting concerns” (\*) en nuestros beans.
  - (\*) Conceptos transversales a toda la aplicación. Ejemplos típicos son “login”, auditoria, seguridad desde la perspectiva de la lógica de negocio.



- Desde Java EE 6, los interceptores evolucionaron en una especificación separada (antes formaban parte de la especificación EJB) JSR 318
- Por lo tanto se pueden aplicar tanto a EJB, como a beans CDI y a SOAP, y RESTful
- web services.

## CDI – Interceptores

- Mediante los interceptores podemos, atrapar la solicitud de ejecuci[on de una determinada lógica en el momento antes de que se ejecute y realizar sobre los datos o la cadena de invocación el proceso que deseemos.
- Existe 4 tipos de interceptores
  - *Constructor-level interceptors*: asociados con el constructor de la clase destino(@AroundConstruct)
  - *Method-level interceptors*: asociados a un método (@AroundInvoke)
  - *Timeout method interceptors*: asociados a un timeout sobre un método @AroundTimeout
  - *Life-cycle callback interceptors*: se ejecutan luego de instanciar un objeto o antes de destruirlo (@PostConstruct o @PreDestroy).

## Interceptores del ciclo de vida

- Como el ciclo de vida de un bean CDI nosotros nunca invocaremos el new en una instancia, ni la setearemos directamente en null, o invocaremos a un método de destrucción sino que son tareas que le delegamos al contenedor.
- El problema es que si necesitamos ejecutar una lógica particular luego que una instancia está creada pero antes que sea usada, algo que típicamente hacemos en el constructor ahora no será posible.
  - Podríamos sobrescribir el constructor, pero en este punto no todos los campos estarían inyectados por lo que no estaría listo el proceso de inicialización.
- Tenemos 2 interceptores que se ejecutan en estos eventos:
  - @PostConstruct, se ejecuta antes de entregar la instancia inyectada pero después que fue inicializada.
  - @PreDestroy se ejecuta antes de destruir una instancia

## Ejemplo

- `public class CustomerService {`
- `@Inject`
- `private EntityManager em;`
- `@PostConstruct`
- `public void init() {`
- `// ...`
- `}`
- `public void createCustomer(Customer customer) {`
- `em.persist(customer);`
- `}`
- `public Customer findCustomerById(Long id) {`
- `return em.find(Customer.class, id);`
- `}`
- `}`

# Interceptores para todos los métodos de una clase

- Hay muchas formas de definir la intercepción.
  - La mas sencilla es agregar un interceptor de nivel de método, de ciclo de vida o timeout a un bean.
  - Si anotamos un método con “AroundInvoke” haremos que todas las llamadas a los demás métodos de una clase, sean procesados luego de invocarse el método anotado.
  - El ámbito del interceptor, es decir donde el interceptor es válido, es solo dentro de la instancia del bean en si misma.
  - Solo ún método de una clase puede tener la anotación AroundInvoke
  - Signatura
    1. *@AroundInvoke*
    2. *Object <METHOD>(InvocationContext ic) throws Exception;*

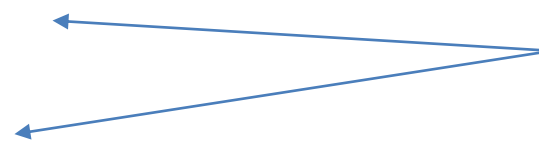


## Ejemplo

### 1. @Transactional

```
2. public class CustomerService {  
3.     @Inject private EntityManager em;  
4.     @Inject private Logger logger;  
5.     public void createCustomer(Customer customer) {  
6.         em.persist(customer);  
7.     }  
8.     public Customer findCustomerById(Long id) {  
9.         return em.find(Customer.class, id);  
10.    }
```

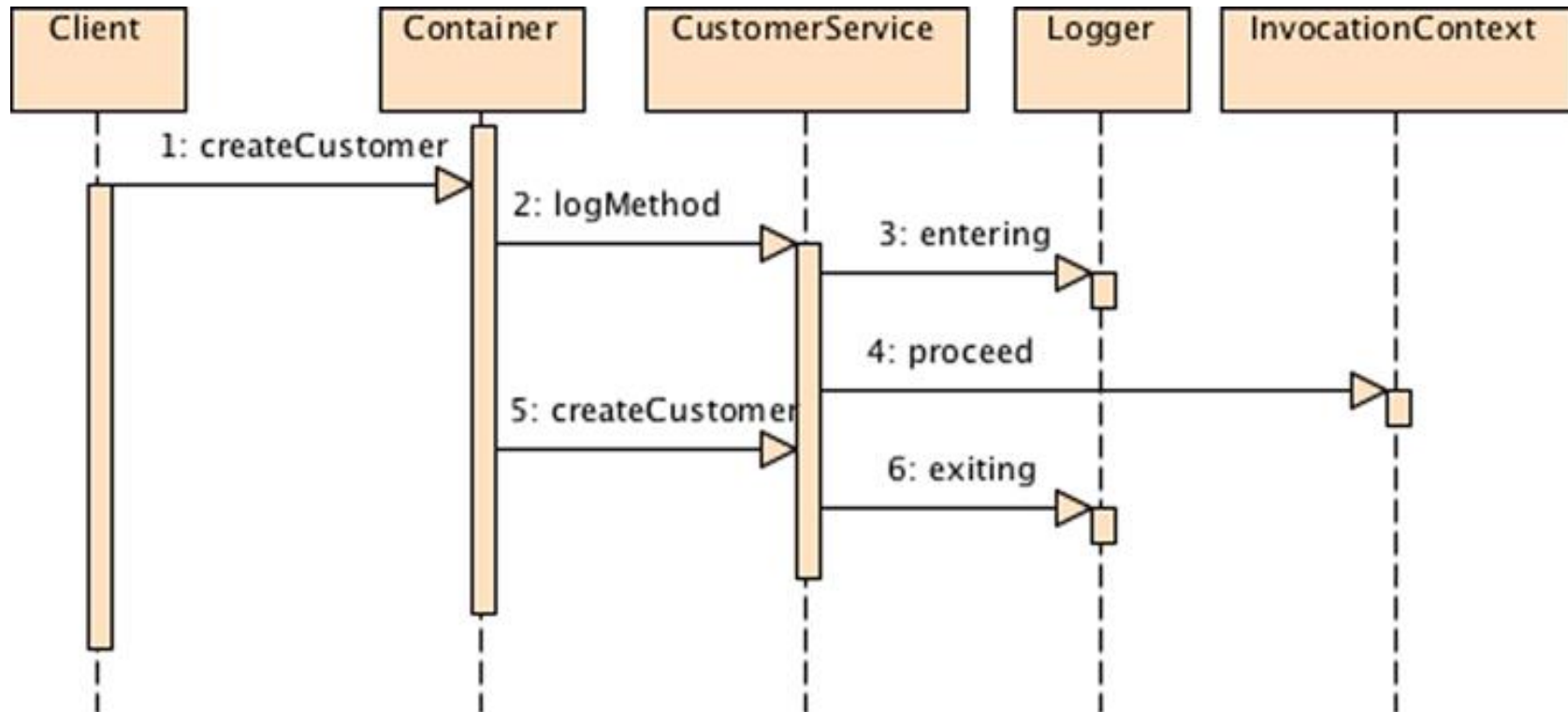
Cada vez que se ejecuten estos métodos antes se ejecutará el interceptor



### 11. @AroundInvoke

```
12. private Object logMethod(InvocationContext ic) throws Exception {  
13.     logger.entering(ic.getTarget().toString(), ic.getMethod().getName());  
14.     try { return ic.proceed(); }  
15.     finally { logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());  
16.     }  
17. }  
18. }
```

# Ejemplo de interacción del Interceptor



## Funcionalidades de la interface *InvocationContext*

1. `getContextData`: permite pasar datos en el mismo contexto de invocacion.
2. `getConstructo`: retorna el constructor de la clase donde fue invocado el interceptor..
3. `getMethod`: retorna el método por el cual el interceptor fue invocado.
4. `getParameters`: retorna los parámetros del método
5. `getTarget`: retorna la instancia a la que pertenece el método interceptado.
6. `getTimer`: retorna el timer asociado a un `@Timeout`.
7. `proceed`: hace que se ejecute el próximo interceptor en la cadena..
8. `setParameters`: permite modificar alguno de los parámetros.

# Interceptores

- El ejemplo anterior si bien es útil no nos permite aprovechar toda la potencia
  - Lo ideal sería que una clase única tenga todo el comportamiento que deseamos, y luego pueda ser declarado como interceptor de multiples métodos en distintas clases.
  - El login es un ejemplo clásico de esta necesidad de comportamiento. Con lo visto anteriormente en cada clase que deseemos login necesitaríamos declarar el método anterior cuando lo ideal es reutilizar el mismo método implementado solo una vez.
- Para esto CDI permite desarrollar un interceptor en una clase separada e instruir a un contenedor para que lo aplique en un bean específico o en algunos métodos específicos de un bean.
- La anotación `@Interceptors` permite definir en un bean, o en métodos de un bean que clase o clases serán interceptors.

## Ejemplo interceptor genérico

```
1. public class LoggingInterceptor {
2.     @Inject
3.     private Logger logger;
4.     @AroundConstruct
5.     private void init(InvocationContext ic) throws Exception {
6.         logger.fine("Entering constructor");
7.         try { ic.proceed(); }
8.         finally { logger.fine("Exiting constructor"); }
9.     }
10.    @AroundInvoke
11.    public Object logMethod(InvocationContext ic) throws Exception {
12.        logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
13.        try { return ic.proceed(); }
14.        finally { logger.exiting(ic.getTarget().toString(), ic.getMethod().getName()); }
15.    }
16. }
```

Cada vez que se cree una instancia del interceptor se ejecuta este método. A diferencia de `PostConstruct` recibe un argumento, el contexto de invocación

## Cliente del interceptor

```
1. @Transactional
2. public class CustomerService {
3.     @Inject
4.     private EntityManager em;
5.     @Interceptors(LoggingInterceptor.class)
6.     public void createCustomer(Customer customer) {...}
7.     public Customer findCustomerById(Long id) {...}
8. }
```

- Podemos aplicar el interceptor a nivel de un método en particular

- Si desamos aplicar el interceptor a todos los métodos lo definimos a nivel de clase

```
1. @Transactional
2. @Interceptors(LoggingInterceptor.class)
3. public class CustomerService {
4.     @Inject
5.     private EntityManager em;
6.     public void createCustomer(Customer customer) {...}
7.     public Customer findCustomerById(Long id) {...}
8. }
```

## Excluir interceptores

- Si anotamos una clase con un interceptor y deseamos que algún método particular no sea interceptado lo podemos hacer con la siguiente anotación:

1. *@Transactional*
2. *@Interceptors(LoggingInterceptor.class)*
3. *public class CustomerService {*
4. *public void createCustomer(Customer customer) {...}*
5. *public Customer findCustomerById(Long id) {...}*
6. ***@ExcludeClassInterceptors***
7. *public Customer updateCustomer(Customer customer) { ... }*
8. *}*

## Encadenar Interceptores

- La anotación `@Interceptors` permite vincular más de un interceptor como una lista de clases separadas por coma encerradas en llaves `{ }`
  - Cuando se definen múltiples interceptores todos se ejecutarán, en el orden en que están listados teniendo siempre prioridad los de clase y luego los de método

```
1. @Interceptors({I1.class, I2.class})
2. public class CustomerService {
3.     public void createCustomer(Customer customer) {...}
4.     @Interceptors({I3.class, I4.class})
5.     public Customer findCustomerById(Long id) {...}
6.     public void removeCustomer(Customer customer) {...}
7.     @ExcludeClassInterceptors
8.     public Customer updateCustomer(Customer customer) {...}
9. }
```

{ I1/I2 }

{ I1/I2/I3/I4 }

{ }



## Interceptores como anotaciones.

- Como vimos hasta aquí la definición de interceptores, con `@Interceptors`, rompemos un poco el concepto de bajo acople de tipo que impone CDI.
  - CDI sigue como principio que las clases se vinculen por interfaces, y si hay implementaciones particulares, el vinculo se realice a través de anotaciones particulares y no con el tipo de dato específico.
  - Pero el esquema visto hasta aquí quiebra este principio
- CDI agregó el concepto de interceptor binding para poder inyectar interceptores mediante anotaciones.
- Se define una anotación que definirá un interceptor con otra anotación `@InterceptorBinding`.

- `@InterceptorBinding`
- `@Target({METHOD, TYPE})`
- `@Retention(RUNTIME)`
- `public @interface Loggable { }`

1. @Interceptor
2. **@Loggable**
3. public class LoggingInterceptor {
4. @Inject
5. private Logger logger;
6. @AroundInvoke
7. public Object logMethod(InvocationContext ic) throws Exception {
8. logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
9. try {
10. return ic.proceed();
11. } finally {
12. logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
13. }
14. }
15. }

## Ejemplo de interceptor

- Ejemplo de aplicar a un método
  1. @Transactional
  2. public class CustomerService {
  3. @Loggable
  4. public void createCustomer(Customer customer) {...}
  5. public Customer findCustomerById(Long id) {...}
  6. }
- Ejemplo de aplicar a toda la clase
  1. @Transactional
  2. @Loggable
  3. public class CustomerService {
  4. public void createCustomer(Customer customer) {...}
  5. public Customer findCustomerById(Long id) {...}
  6. }

## Activar interceptor binding en CDI

- Por defecto esta funcionalidad está desactivada. para activarla hay indicarlo en el descriptor beans.xml
  1. `<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"`
  2. `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
  3. `xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee „{`
  4. `http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"`
  5. `version="1.1" bean-discovery-mode="all">`
  6. `<interceptors>`
  7. `<class>ar.edu.utn.frsf.curso.LoggingInterceptor</class>`
  8. `</interceptors>`
  9. `</beans>`

## Multiples interceptores.

- Si en una clase usamos múltiples interceptores de esta manera, no tenemos forma de definir el orden. Para ello lo que podemos es asignar a cada interceptor una prioridad y luego el contenedor los aplicará en el orden en que la prioridad lo defina.
- A menor prioridad antes será llamado un interceptor.
- `@Interceptor`
- `@Loggable`
- `@Priority(200)`
- `public class LoggingInterceptor {`
- `.....`
- `}`

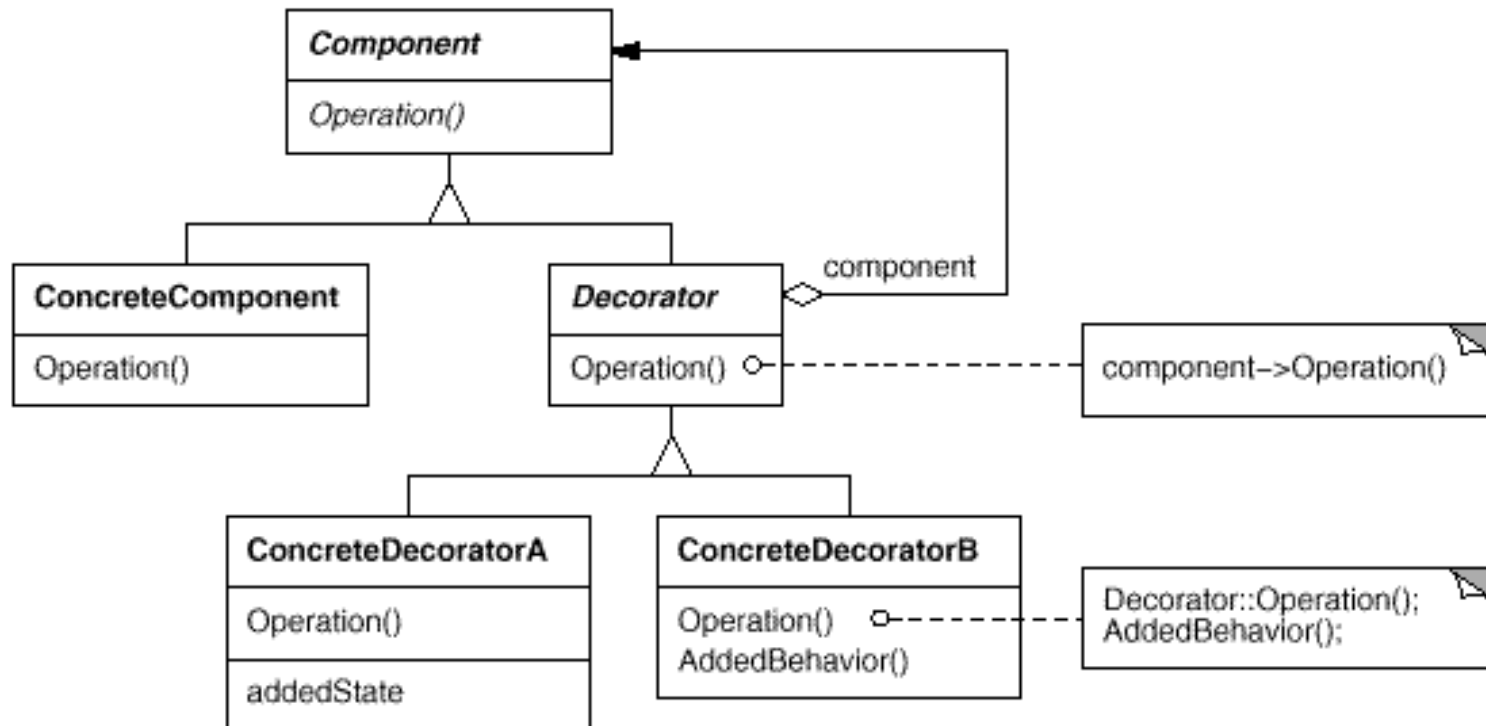
## Prioridades

- Existen las siguientes constantes:
- PLATFORM\_BEFORE = 0: definidos por Java EE
- LIBRARY\_BEFORE = 1000: rango para interceptores de librerías,
- APPLICATION = 2000: rango para nuestra aplicación
- LIBRARY\_AFTER = 3000: interceptores que se ejecutarán al finalizar la cadena de intercepción
- PLATFORM\_AFTER = 4000: se ejecutarán al finalizar la cadena de intercepción y son parte de Java EE.

## Decoradores

- Mientras los interceptores realizan tareas transversales a todas las clases, no tienen real conocimiento de la semántica de la lógica que interpreta.
  - Y si así deseáramos que la tuviesen tendríamos módulos muy complejos.
- Si deseamos interceptores puntuales asociados a conceptos de una lógica de negocio específica CDI brinda “decoradores”
  - El patrón decorator es un patrón del GOF muy común en el ADOO.
- La idea es tomar una clase y “envolverla con otra” (decorarla).
  - Así siempre antes de ejecutar la lógica de un método hay que atravesar la lógica que lo decora o envuelve.
  - No tienen como fin resolver conceptos técnicos sino de negocio puntual.

# GOF Decorator





## Ejemplo

```
1. public interface Traductor {  
2.     public String traducir(String i);  
3. }
```

```
1. public class TraducirIngles implements Traductor {  
2.     @Override  
3.     public String traducir(String i){  
4.         if(i.equalsIgnoreCase("hola")) return "Hello";  
5.         else return "palabra no encontrada en el diccionario";  
6.     }  
7. }
```

```
1. @Decorator  
2. @Priority(Interceptor.Priority.APPLICATION+1)  
3. public class TraductorDecorator implements Traductor{  
4.     @Inject @Delegate @Default  
5.     private Traductor trd;  
6.     @Override  
7.     public String traducir(String i){  
8.         return "[traduccion desde ingles]" + trd.traducir(i);  
9.     }  
10. }
```

```
1.  @Named
2.  @SessionScoped
3.  public class TraductorController implements Serializable{
4.      private String palabra;
5.      private String palabraTraducida;
6.      @Inject    private Traductor traductor;
7.      public String traducir(){
8.          this.palabraTraducida = this.traductor.traducir(palabra);
9.          return null;
10. }
```

Seleccione una opcion | **OPERAR** | **TRADUCIR** |

Palabra a traducir

[traduccion desde ingles]Hello

## Decoradores en cadena

- Si tengo más de un decorador por clase, entonces puedo ordenarlos mediante la anotación `@Priority`.
- Los decoradores por defecto no están habilitados en CDI. Tengo dos formas de habilitarlos:
  - Usar la anotación `@Priority`
  - Explícitamente en el archivo `beans.xml`

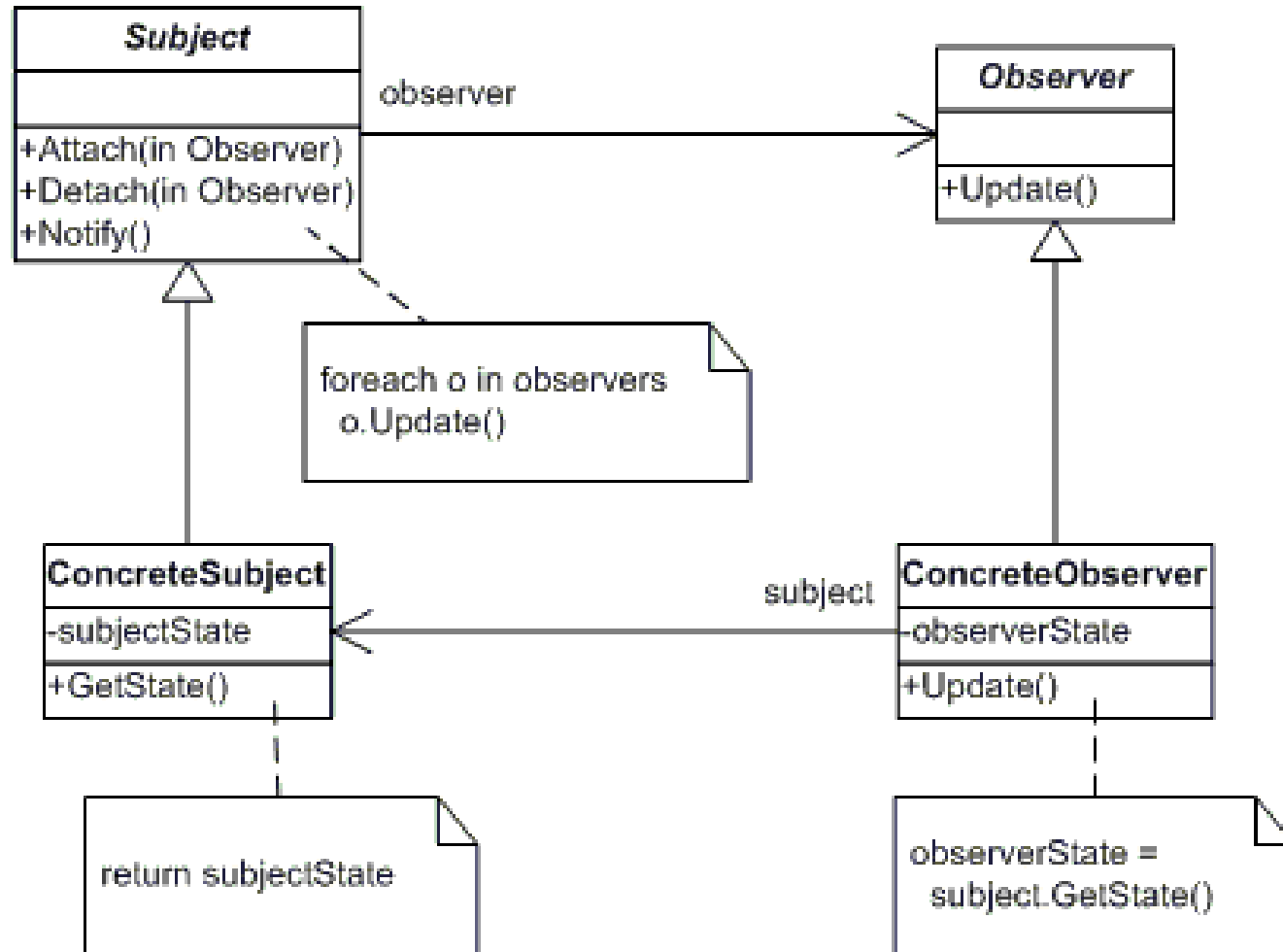
```
1. <beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="
4.     http://xmlns.jcp.org/xml/ns/javaee
5.     http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
6.   bean-discovery-mode="annotated">
7.   <decorators>
8.     <class>org.sample.MyDecorator</class>
9.   </decorators>
10. </beans>
```

si en una aplicación hay interceptores y decoradores, los interceptores se aplican primero

## Eventos

- Los conceptos vistos hasta aquí son muy potentes:
  - DI, Alternativas, interceptores y decoradores son todos manejados por el contenedor y permiten tener bajo acople agregando comportamiento adicional para variar en tiempo de despliegue o de ejecución el comportamiento de un sistema.
- Los eventos son una característica de CDI que va un paso más allá, y permite que los beans interactúen entre ellos sin tener la más mínima dependencia en tiempo de compilación.
- Un bean puede definir un evento, otro bean puede generarlo y un tercero puede manejarlo.
  - Los beans pueden estar en paquetes separados incluso en capas separadas de la aplicación.
- Sigue otro patron del GoF → Observer

# GOF Observer



## Evento

- Los productores de eventos los lanzan usando la interface `javax.enterprise.event.Event`.
  - Lo hacen a través del método “`fire()`”, y pasando el objeto evento como parámetro. Esto no depende del observador.
- Ejemplo: cada vez que se cree un libro en un inventario de libros, se lanzará un evento libro agregado. Este evento será capturado por un oyente y realizará la lógica que considere necesaria.
- Un productor de eventos lanza un evento usando la interface `Event<K>`
- Los eventos son observados por los subscriptores
  - Por evento puede haber más de uno.

- Un método es observador si recibe un parámetro de un tipo observado y le agrega la anotación @Observes.
  - Productor: @Inject @Any **Event**<**Algo**> event;
  - Observer: **void** onEvento(**@Observes** **Algo** event) {
  - Alternativamente podría tener calificadores.
- Un método observer es notificado de un evento si tiene la anotación @Observes y coincide el tipo de dato que observa y el calificador.

## Ejemplo – Lanzar un evento

```
1. public class BookService {  
2. @Inject  
3. private NumberGenerator numberGenerator;  
4. @Inject  
5. private Event<Book> bookAddedEvent;  
6. public Book createBook(String title, Float price, String description) {  
7. Book book = new Book(title, price, description);  
8. book.setIsbn(numberGenerator.generateNumber());  
9. bookAddedEvent.fire(book);  
10. return book;  
11. }  
12. }
```

```
1. public class InventoryService {  
2. @Inject  
3. private Logger logger;  
4. List<Book> inventory = new ArrayList<>();  
5. public void addBook(@Observes Book book) {  
6. logger.info("Adding book " + book.getTitle() + " to  
   inventory");  
7. inventory.add(book);  
8. }  
9. }
```



# Asignando Calificadores a un evento

```
1. public class BookService {
2.     @Inject
3.     private NumberGenerator numberGenerator;
4.     @Inject @Added private Event<Book> bookAddedEvent;
5.     @Inject @Removed private Event<Book> bookRemovedEvent;
6.     public Book createBook(String title, Float price, String description) {
7.         Book book = new Book(title, price, description);
8.         book.setIsbn(numberGenerator.generateNumber());
9.         bookAddedEvent.fire(book);
10.    return book;
11. }
12. public void deleteBook(Book book) {
13.     bookRemovedEvent.fire(book);
14. }
15. }
```

```
1. @Qualifier
2. @Retention(RUNTIME)
3. @Target({FIELD, TYPE, METHOD})
4. public @interface Added{ }
```

```
1. @Qualifier
2. @Retention(RUNTIME)
3. @Target({FIELD, TYPE, METHOD})
4. public @interface Removed{ }
```

## Eventos con calificadores.

```
1. public class InventoryService {  
2.     @Inject  
3.     private Logger logger;  
4.     List<Book> inventory = new ArrayList<>();  
5.     public void addBook(@Observes @Added Book book) {  
6.         logger.warning("Adding book " + book.getTitle() + " to inventory");  
7.         inventory.add(book);  
8.     }  
9.     public void removeBook(@Observes @Removed Book book) {  
10.        logger.warning("Removing book " + book.getTitle() + " to inventory");  
11.        inventory.remove(book);  
12.    }  
13. }
```

## Beans Predefinidos CDI

- Los siguientes recursos JavaEE pueden ser inyectados en cualquier bean CDI
  - `UserTransaction` → `@Resource UserTransaction transaction;`
  - `Principal` → `@Resource Principal principal;`
  - `Validator` → `@Resource Validator validator;`
  - `ValidatorFactory` → `@Resource ValidatorFactory factory;`
  - `HttpServletRequest` → `@Inject HttpServletRequest req;`
  - `HttpSession` → `@Inject HttpSession session;`
  - `ServletContext` → `@Inject ServletContext context;`