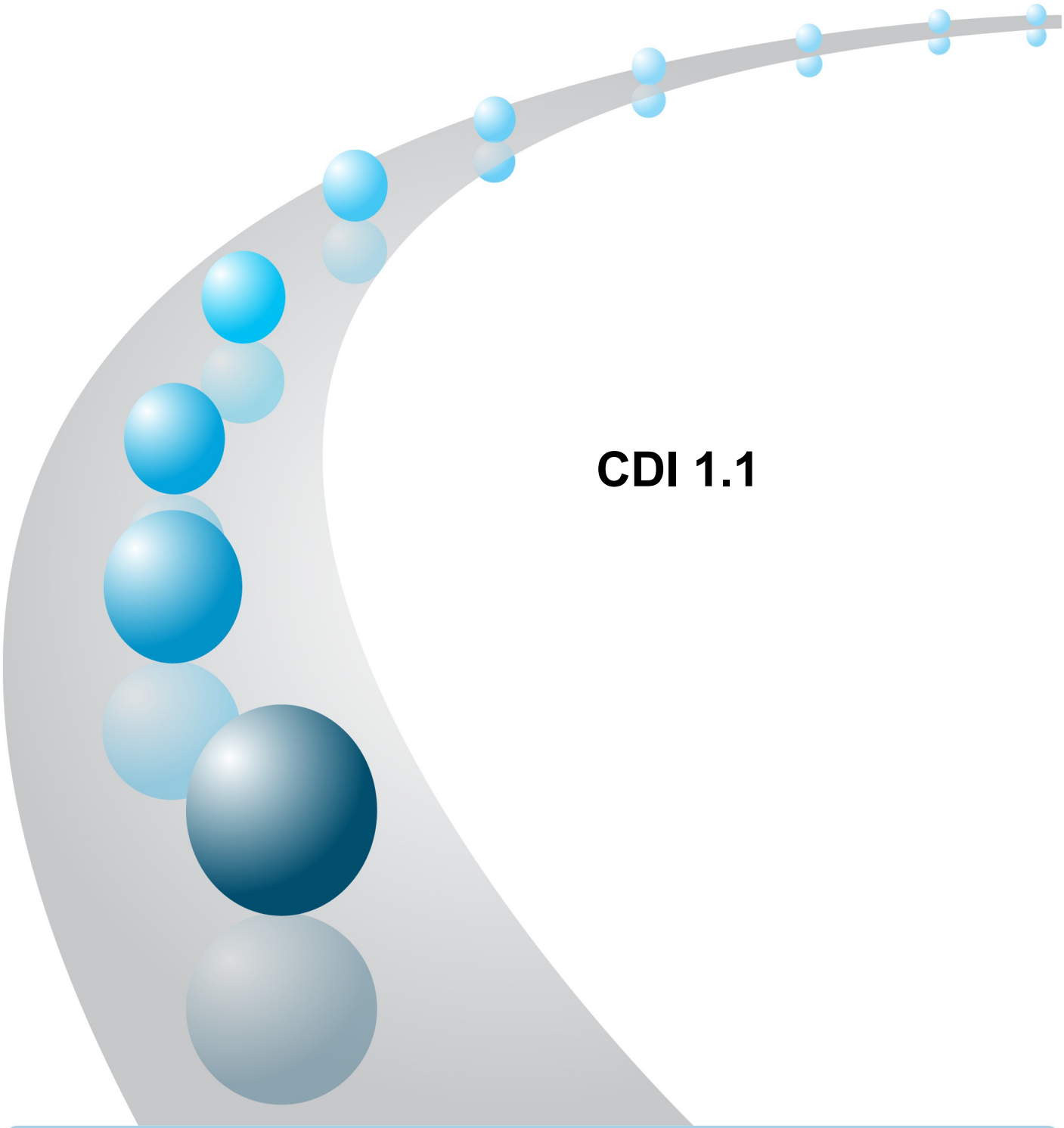
Noël Vaes

Java Trainer & Consultant



CDI 1.1

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Vrijwel alle namen van software- en hardwareproducten die in deze cursus worden genoemd, zijn tegelijkertijd ook handelsmerken en dienen dienovereenkomstig te worden behandeld.

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand of openbaar worden gemaakt in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door fotokopieën, opnamen of op enige andere manier, zonder voorafgaande schriftelijke toestemming van de auteur. De enige uitzondering die hierop bestaat, is dat eventuele programma's en door de gebruiker te typen voorbeelden mogen worden ingevoerd opgeslagen en uitgevoerd op een computersysteem, zolang deze voor privé-doeleinden worden gebruikt, en niet bestemd zijn voor reproductie of publicatie.

Correspondentie inzake overnemen of reproductie kunt u richten aan:

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

Ondanks alle aan de samenstelling van deze tekst bestede zorg, kan de auteur geen aansprakelijkheid aanvaarden voor eventuele schade die zou kunnen voortvloeien uit enige fout, die in deze uitgave zou kunnen voorkomen.

24/09/2016

Copyright© 2016 Noël Vaes

Inhoudsopgave

Hoofdstuk 1: Inleiding

CDI is de afkorting van *Contexts and Dependency Injection.* Voorheen was deze technologie ook gekend als *Web Beans.*

CDI is geen echt nieuwe technologie. Het is een standaardisatie van technologieën die buiten de JEE-standaard ontwikkeld werden. Het gaat onder andere om het concept van ***Inversion of Control (IOC)*** *en* ***Dependency Injection (DI)*** dat vooral in het Spring-*framework* toegepast wordt. Maar ook *frameworks* als *Google Guice* en *Seam* staan aan de wieg van CDI.

Daarnaast biedt CDI ook **contextueel beheer van de levenscyclus** van een object.

Een gedetailleerde beschrijving van beide concepten volgt later in de cursus maar we kunnen hier alvast een summiere introductie geven.

1.1. Dependency Injection

Bij object georiënteerde programmeertalen bestaat software uit een aantal objecten die gebruikmaken van elkaars diensten. Belangrijk hierbij is dat de objecten ontwikkeld worden volgens het principe ***Loose coupling and high cohesion.***

Objecten dienen een kerntaak te vervullen en zich niet bezig te houden met nevenactiviteiten. Dit maakt ze meer geschikt voor hergebruik in andere omstandigheden. We noemen dit *high cohesion.*

Voor de taken die niet tot de kerntaak van een object behoren dient het object beroep te doen op andere objecten. Er moet dus op de een of andere manier een koppeling zijn tussen verschillende objecten. Praktisch kan dit gerealiseerd worden doordat een object een referentie heeft naar een ander object en dit andere object zelf instantieert. Dit is echter een vaste koppeling (*tight coupling)* en biedt weinig flexibiliteit. Het is niet mogelijk het gekoppelde object later op een eenvoudige manier door iets anders te vervangen. De afhankelijkheid tussen de twee objecten is intrinsiek. We zouden dit evenwel kunnen omdraaien en deze afhankelijkheid van buiten aanreiken. Dit noemt men *Inversion of Control (IOC)*. De controle over de afhankelijkheid ligt dan niet langer binnen het object zelf maar wordt extern geregeld. Ze wordt van buitenaf toegewezen. Praktisch gebeurt dit door de referentie naar het andere object extern in te stellen. Dit noemt men *Dependency Injection (DI)*.

Indien we in de afhankelijkheid verder gebruikmaken van polymorfisme via interfaces of abstracte klassen is de koppeling nog losser. Dit alles resulteert in de vereiste *loose coupling*.

Frameworks als *Spring* zijn op dit principe van *Inversion of Control* gebaseerd aangevuld met *Aspect Oriented Programming* (AOP) om *cross cutting concerns* te integreren.

In JEE5 werden reeds een aantal van deze technieken opgenomen. Zo was het onder andere mogelijk allerlei *resources* in EJB's en webcomponenten te injecteren door middel van annotaties. Tevens konden EJB's onderling via eenvoudige annotaties geïnjecteerd worden. Het AOP-verhaal kreeg tevens een gedeeltelijk equivalent door het gebruik van *interceptors*.

In JEE6 worden deze concepten nog verder uitgewerkt in de CDI-specificatie. Dit is eigenlijk een ecosysteem waarbij *beans* gecreëerd en aaneengeregen (*wiring)* worden door middel van *Dependency Injection.* Deze *beans* slaan bovendien de brug tussen de *managed beans*

van *Java Service Faces (JSF)* in de presentatielaag en de EJB's in de *business*-laag. Dit is een hele mond vol en daarom zullen we in deze cursus de verschillende concepten stap voor stap introduceren en illustreren met praktische voorbeelden.

1.2. Contextual lifecycle management

CDI zorgt voor het instantiëren, aaneenrijgen en nadien opruimen van objecten of *beans*. De hele levenscyclus van een *bean* wordt dus geregeld door de CDI-container. Maar wanneer start deze levenscyclus en wanneer eindigt deze? Dat hangt af van de context waarin een object nodig is. Sommige objecten zijn nodig gedurende de hele tijd dat de applicatie actief is, andere slechts gedurende de interactie met een gebruiker.

CDI zorgt ervoor dat de levenscyclus van een *bean* doorlopen wordt naargelang de gevraagde omstandigheden of *context*. Men noemt dit daarom *contextual lifecycle management*.

Bij een interactie met de eindgebruiker kunnen er zowel globale *beans* als gebruikerspecifieke *beans* in het spel zijn. CDI zorgt er steeds voor dat deze vlekkeloos met elkaar kunnen samenwerken door de *beans* op het gepaste moment te instantiëren, te koppelen aan andere *beans* en ze nadien op te ruimen.

Hoofdstuk 2: Beans

2.1. Managed Beans

Binnen CDI wordt er steeds gesproken van *beans*. Het woord *bean* heeft binnen de Java-wereld echter al een hele geschiedenis achter de rug. Zo kennen we de oeroude *JavaBeans,* de *JSF managed beans* en de *Enterprise JavaBeans (EJB),* maar een uniforme en eenduidige definitie bestond niet echt. CDI tracht binnen het JEE-platform de definitie van een *bean* te consolideren in een duidelijk afgebakend begrip.

Het basisbegrip is de *managed bean*. Dit zijn gewoon Java-objecten waarvan de levenscyclus door een container beheerd *(managed)* wordt. Als we spreken over 'gewoon Java-object' dan wil dat zeggen dat deze objecten niet aan allerlei specifieke voorwaarden moeten voldoen. Dergelijke objecten worden wel eens aangeduid met het acroniem POJO: *Plain Old Java Object*.

Managed beans dienen de volgende mogelijkheden te hebben:

- *lifecycle callbacks.*
- *interceptors.*
- *resource injection.*

Van dit soort *managed beans* zijn er specialisaties die extra mogelijkheden toevoegen. Zo is een EJB een *managed bean* die tevens beveiliging en transactioneel gedrag toevoegt.

Een CDI-*bean* daarentegen is een *managed bean* die *contextual lifecycle management* toevoegt.

De voorwaarden voor een dergelijke CDI-*bean* zijn miniem: ieder object met een standaardconstructor wordt beschouwd als een *bean*, op een aantal uitzonderingen na dan.

Voor de volledigheid geven we hier de lijst van voorwaarden waaraan voldaan moet worden:

- De klasse moet een **constructor zonder argumenten** hebben ofwel een **constructor met geïnjecteerde argumenten**. Dit laatste zullen we later in de cursus behandelen.
- De klasse mag **geen geneste klasse** zijn, een *static inner class* mag wel.
- Het moet een **concrete klasse** zijn, met uitzondering van een *decorator* voorzien van de annotatie `@Decorator`.
- Het mag **geen EJB** zijn. Deze vormen namelijk een afzonderlijke categorie. We wijden hier overigens een afzonderlijke paragraaf aan.
- De klasse mag de interface `javax.enterprise.inject.sp.Extension` niet implementeren.

Geef toe, de voorwaarden zijn heel ruim en de meeste objecten die we kennen, komen in aanmerking.

2.2. Containers

Een *bean* is een object dat door een container beheerd wordt. Concreet wil dit zeggen dat de volledige levenscyclus door een containerapplicatie geregeld wordt: de instantiatie, initialisatie en opruiming.

Het is in dit programmeermodel niet langer de programmeur die een nieuw object maakt door de constructor op te roepen: objecten worden door de container gemaakt en ter beschikking gesteld van andere componenten.

Als we hier spreken over 'container' dan is dat doorgaans de JEE-applicatieserver: dit is namelijk de natuurlijke habitat van CDI-*beans.* Maar CDI is evenwel niet beperkt tot deze omgeving. Het is mogelijk binnen een gewone *standalone*-applicatie gebruik te maken van een CDI-container. Dit kan bijvoorbeeld door gebruik te maken van de referentie-implementatie *Weld.* We verwijzen hiervoor naar website van *Weld:* http://weld.cdi-spec.org.

Applicatieservers die voldoen aan de JEE6-standaard of hoger dienen een implementatie van CDI aan boord te hebben. En dit geldt zowel voor het *full profile* als het *web profile*.

In deze cursus laten we het gebruik van CDI buiten JEE buiten beschouwing. Wij zullen gebruikmaken van een JEE7-applicatieserver, meer concreet *WildFly 10.* Alle voorbeelden kunnen evenwel ook uitgevoerd worden op om het even welke andere JEE-server vanaf versie 6. Webcontainers zoals *Tomcat* voldoen niet volledig aan de JEE-standaard en hebben momenteel geen CDI aan boord. Ze kunnen evenwel wel met CDI uitgebreid worden. Hiervoor verwijzen we naar de uitgebreide documentatie van *Weld*.

Vermits CDI geïntegreerd is in een JEE-applicatieserver, dienen we hier geen extra bibliotheken toe te voegen. We moeten wel CDI activeren. Dit gebeurt door een bestand ***beans.xml*** toe te voegen aan een JAR- of WAR-bestand in de respectievelijke mappen ***META-INF*** of ***WEB-INF.***

Bij aanwezigheid van dit bestand zal CDI automatisch actief worden voor de *beans* die in het archief aanwezig zijn.

Sinds JEE7 is dit bestand niet altijd nodig. CDI wordt in JEE7 namelijk automatisch geactiveerd indien er *beans* gevonden worden met een *scope*-annotatie of bij aanwezigheid van EJB's.

Het XML-bestand ziet er als volgt uit:

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.1" bean-discovery-mode="all">
</beans>
```

Merk alvast het attribuut `bean-discovery-mode` op. We stellen de waarde hiervan in op `all` zodat alle objecten in aanmerking komen als *bean*. De voorkeurswaarde in CDI 1.1 (JEE7) is evenwel `annotated` zodat enkel *beans* die voorzien zijn van een annotatie in aanmerking komen. Eventueel kan men bepaalde klassen uitsluiten met de annotatie `@Vetoed`.

CDI maakt vooral gebruik van annotaties. Slechts in bepaalde gevallen zullen we extra gegevens toevoegen aan het XML-bestand. Doorgaans blijft dit evenwel leeg en dient het enkel om CDI te activeren voor een bepaald archief.

2.3. Mijn eerste CDI-*bean*

Tijd om de handen uit de mouwen te steken en onze eerste CDI-*bean* tot leven te wekken.

Ook hier maken we de zoveelste variant van de obligate "Hello World".

De *bean* ziet er als volgt uit:

```
public class HelloBean {
   public String sayHello() {
      return "Hello World";
   }
}
```

Niet bijzonders dus. Een gewone klasse met standaardconstructor en een publieke methode. Ook extra annotaties zijn hier niet nodig om deze klasse tot *bean* te maken.

We kunnen deze *bean* nu gebruiken in een webapplicatie met bijvoorbeeld een *servlet*:

```
@WebServlet("/Hello")
public class HelloServlet extends HttpServlet {
   @Inject
   private HelloBean bean;

   protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      response.setCharacterEncoding("UTF-8");
      Writer out = response.getWriter();
      out.write("<html><head><title>Hello</title></head><body>");
      out.write(bean.sayHello());
      out.write("</body></html>");
   }
}
```

De *servlet* heeft een instantie van de *bean* nodig om zijn werk te kunnen doen. In plaats van deze zelf te instantiëren, wordt dit overgelaten aan de CDI-container. Deze zal een instantie maken en in het veld injecteren. We geven dit aan met de annotatie `@Inject`.

We merken op dat een *servlet* zelf omwille van zijn bijzondere functie niet als *bean* beschouwd wordt, maar het is wel mogelijk *beans* te gebruiken binnen de *servlet*. Hetzelfde geldt voor andere componenten als *stateless session beans* en *message driven beans*. We komen hier later nog op terug.

Opdracht 1: Mijn eerste CDI-bean

In deze opdracht gaan we een webproject maken met een *servlet* die gebruikmaakt van een *bean*. We gaan er hierbij van uit dat *WildFly* en *Maven* reeds geïnstalleerd zijn.

- Maak een nieuw Maven-project met de volgende POM:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>eu.noelvaes</groupId>
   <artifactId>cdi</artifactId>
   <version>1.1</version>
   <packaging>war</packaging>

   <properties>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
```

```
      <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
   </properties>

   <build>
      <finalName>${project.artifactId}</finalName>
      <plugins>
         <plugin>
            <groupId>org.wildfly.plugins</groupId>
            <artifactId>wildfly-maven-plugin</artifactId>
            <version>1.1.0.Alpha11</version>
         </plugin>
      </plugins>
   </build>
   <dependencies>
      <dependency>
         <groupId>javax</groupId>
         <artifactId>javaee-api</artifactId>
         <version>7.0</version>
         <type>jar</type>
         <scope>provided</scope>
      </dependency>
   </dependencies>
</project>
```

- Voeg het bestand ***beans.xml*** toe in de map ***/webapp/WEB-INF***:

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.1" bean-discovery-mode="all">

</beans>
```

- Voeg de *bean*-klasse toe in een pakket naar keuze:

```
public class HelloBean {
   public String sayHello() {
      return "Hello World";
   }
}
```

- Voeg ten slotte de *servlet*-klasse toe:

```
@WebServlet("/Hello")
public class HelloServlet extends HttpServlet {
   @Inject
   private HelloBean bean;

   protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      response.setCharacterEncoding("UTF-8");
```

```
      Writer out = response.getWriter();
      out.write("<html><head><title>Hello</title></head>");
      out.write(bean.sayHello());
      out.write("</html>");
   }
}
```

- Stel de webapplicatie in werking met het volgende commando:

  ```
  mvn package wildfly:deploy
  ```

- Open de volgende URL: http://localhost:8080/cdi/Hello

2.4. Named beans en expression language

Zoals we in de vorige paragraaf aantoonden, is er geen extra configuratie nodig om een Java-object als CDI-*bean* te gebruiken. We kunnen deze met de annotatie `@Inject` injecteren in een *servlet*.

Beans kunnen ook gebruikt worden in JSP-pagina's en JSF-*facelets*. Hiervoor is het evenwel nodig de *beans* een unieke naam te geven. Vervolgens kunnen ze gebruikt worden door middel van *unified expression language*. Dit benoemen van een *bean* is geen verplichting om een object tot *bean* te maken, het is enkel vereist indien we de *bean* willen gebruiken in combinatie met *expression language (EL).*

We illustreren dit met onze `HelloBean`. Om deze rechtstreeks binnen een JSP-pagina te kunnen gebruiken, geven we hem eerst een naam met de annotatie `@Named`:

```
import javax.inject.*;

@Named("hello")
public class HelloBean {
   public String sayHello() {
      return "Hello World";
   }
}
```

Deze annotatie `@Named` kan optioneel voorzien worden van een waarde die dan de naam van de *bean* is. In dit voorbeeld is de naam van de *bean* daarom ***hello***.

Indien we deze waarde achterwege laten, wordt de verkorte klassennaam van de *bean* gebruikt waarbij de eerste hoofdletter naar een kleine letter wordt omgezet. In dit geval zou de naam dan ***helloBean*** zijn.

We kunnen vervolgens deze *bean* gebruiken binnen een JSP-pagina of JSF-*facelet* met behulp van *expression language*:

Hello.jsp

```
<html>
<head><title>Hello</title></head>
<body>
${hello.sayHello()}
</body>
</html>
```

We gebruiken in deze JSP-pagina een uitdrukking met onmiddellijke evaluatie (*immediate evaluation*). De uitdrukking start daarom met het $-teken.
Versie 2.1 van *unified expression language* maakt het mogelijk methoden van een object aan te roepen, hetgeen we hier ook doen in dit voorbeeld.

EL 2.1 maakt deel uit van JEE6/7 en kan dus zonder problemen aangewend worden.

Opdracht 2: Named Beans

In deze opdracht gebruiken we een *bean* binnen een JSP-pagina.

- Voorzie de *bean* van een naam met de annotatie `@Named`.
- Maak een JSP-pagina die de *bean* rechtstreeks gebruikt.

2.5. CDI-*beans* en *JavaServer Faces*

Binnen JEE is *JavaServer Faces* de aangewezen technologie om complexere webapplicaties te ontwikkelen. Bij JSF kennen we reeds het concept van *managed beans* die voorzien zijn van de annotatie `@ManagedBean` eventueel aangevuld met een extra annotatie voor de *scope*. Deze *managed beans* dienen onder andere als *backing bean* voor de JSF *-pagina's* in de vorm van JSP-pagina's of *facelets*.

Met de introductie van CDI kan iedere CDI-*bean* ook gebruikt worden binnen het JSF-*framework* en zijn de *managed beans* eigenlijk overbodig geworden; ze worden volledig vervangen door CDI-*beans,* die veel meer mogelijkheden hebben zoals in de loop van deze cursus zal blijken.

Het gebruik van CDI-*beans* in combinatie met JSF is enkel mogelijk binnen een volledige JEE-applicatieserver. Indien men JSF wil gebruiken binnen een webcontainer zoals *Tomcat*, dient men nog steeds gebruik te maken van de *JSF managed beans*.

We kunnen onze `HelloBean` als volgt gebruiken in een JSF-*facelet*:

Hello.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
   <title>Hello</title>
</h:head>
<h:body>
<h:outputText value="#{hello.sayHello()}" />
</h:body>
</html>
```

Hier maken we gebruik van een uitdrukking met uitgestelde evaluatie (*deferred evaluation*) zoals dat gebruikelijk is binnen JSF.

In het verder verloop van deze cursus zullen we steeds gebruikmaken van JSF om het CDI te illustreren. Daarin komen stelselmatig de verschillende mogelijkheden van CDI-*beans* binnen JSF aan bod.

2.6. De levenscyclus van een bean

Beans worden door de container ten gepaste tijde geïnstantieerd en opgeruimd. Wanneer dit gebeurt, hangt af van de context en dat zullen we in het volgende hoofdstuk uitvoerig toelichten.

In ieder geval doorloopt een *bean* een bepaalde levenscyclus met de volgende stappen:

1. Instantiatie.
2. Injectie van afhankelijkheden: eerst de velden daarna de *setters* (zie later).
3. Initialisatiemethoden met de annotatie `@PostConstruct`.
4. Gebruik van de *bean.*
5. Opruimmethoden met de annotatie `@PreDestroy`.

Deze levenscyclus wordt volledig door de container beheerd. Zoals vele andere componenten in een JEE-omgeving kunnen *beans* methoden hebben die voorzien worden van de annotaties `@PostConstruct` en `@PreDestroy`. Hierin kunnen extra activiteiten uitgevoerd worden voor de initialisatie en opruiming van de *bean*. Deze methoden mogen geen argumenten hebben en dienen `void` als *return*-type te hebben.
De `@PostConstruct`-methode wordt door de container automatisch opgeroepen na instantiatie en injectie van afhankelijkheden. In deze initialisatiemethode worden daarom typisch activiteiten verricht waarbij de afhankelijkheden vereist zijn, anders zou het ook gewoon in de constructor kunnen gebeuren. Een voorbeeld hiervan is het opbouwen van een connectie met een databank. Het opnieuw sluiten van de connectie gebeurt dan in de `@PreDestroy`-methode.

We kunnen onze `HelloBean` als volgt uitbreiden:

```
@Named("hello")
public class HelloBean {

   @PostConstruct
   public void init() {
      System.out.println("init()");
   }

   @PreDestroy
   public void destory() {
      System.out.println("destroy()");
   }

   public String sayHello() {
      return "Hello World";
   }
}
```

Opdracht 3: CDI en JSF

In deze opdracht configureren we onze webapplicatie voor gebruik van *JavaServer Faces*. Tevens voorzien we de *bean* van initialisatie- en opruimmethoden.

- Voeg in ***web.xml*** de volgende configuratie voor JSF toe:

```
<servlet>
   <servlet-name>Faces Servlet</servlet-name>
   <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

- Voeg de volgende *facelet* toe die gebruikmaakt van de *bean*.

Hello.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
   <title>Hello</title>
</h:head>
<h:body>
<h:outputText value="#{hello.sayHello()}" />
</h:body>
</html>
```

- Voorzie de `HelloBean` van initialisatie- en opruimmethoden.
- *Deploy* de applicatie en open de volgende URL:
  http://localhost:8080/cdi/Hello.faces
- Ga na wanneer de initialisatie- en opruimmethoden worden opgeroepen.

Hoofdstuk 3: Contexten en scopes

3.1. Inleiding

De levenscyclus van een *bean* wordt door de container beheerd. Dat wil zeggen dat de *bean* op een bepaald moment geïnstantieerd en geïnitialiseerd wordt en vervolgens ter beschikking wordt gesteld voor gebruik. Het moment waarop dit gebeurt, hangt af van de context waarin een *bean* gebruikt wordt. Vandaar dat men ook spreekt van *contextual lifecycle management*.

Het concept van een context wordt duidelijk als we het hebben over typische webapplicaties. Een gebruiker kan via zijn browser een verzoek sturen naar een webserver. De afhandeling van zo'n individueel verzoek gebeurt door een afzonderlijke *thread* binnen de webserver. Indien er tijdens deze afhandeling *beans* nodig zijn, dan is hun levensduur beperkt tot de tijdsduur van de afhandeling. Deze *beans* hebben daarom een ***request scope***. CDI zal dergelijke *beans* instantiëren en ter beschikking stellen gedurende de afhandeling. Nadat de *thread* zijn werk beëindigd heeft, worden de *beans* weer opgeruimd. De levensduur is dus zeer kort en deze *beans* kunnen enkel informatie bijhouden gedurende de afhandelingsperiode.

Een eindgebruiker kan evenwel meerdere verzoeken naar dezelfde server sturen, gespreid in de tijd. We noemen dit een sessie. Dergelijke losstaande verzoeken worden door de webserver samengevoegd tot een geheel. Gedurende de tijd dat de sessie actief is, kunnen er *beans* nodig zijn die informatie bevatten die geldig is voor een bepaalde gebruikerssessie. Meerdere verzoeken van dezelfde gebruiker kunnen dan gebruikmaken van deze *bean.* Zulke *beans* hebben een ***session scope***. Ze worden aangemaakt bij het begin van een sessie (of bij het eerste gebruik binnen een sessie) en worden opgeruimd zodra de sessie afloopt. Sessies worden doorgaans beëindigd na een bepaalde periode van inactiviteit of na uitdrukkelijk verzoek van de gebruiker (invalidatie van de sessie).

Gebruikerssessies zijn vaak langlopend. Telkens een gebruiker een verzoek stuurt naar de server, wordt de levensduur van de sessie verlengd. Bovendien kan een gebruiker meerdere tabbladen in de browser openen met inhoud van dezelfde server. Naargelang de browser maken deze tabbladen soms deel uit van dezelfde sessie.

Vaak is bepaalde informatie slechts tijdelijk nodig binnen een sessie en ook binnen hetzelfde tabblad. Het bijhouden van deze informatie gedurende de gehele sessie kan overbodig geheugengebruik tot gevolg hebben. Om die reden kan men *beans* voorzien van een ***conversation scope***. Deze *scope* is enkel bruikbaar in combinatie met JSF. De levensduur van een *bean* omvat dan verschillende verzoeken vanuit dezelfde browservenster. Het begin en het einde van de conversatie moeten evenwel expliciet aangegeven worden.

Ten slotte kunnen bepaalde *beans* nodig zijn voor de gehele applicatie. Ze kunnen informatie bevatten die door alle verzoeken en sessies gedeeld worden. Deze *beans* krijgen ***application scope*** of ***singleton scope***. Er is een verschil tussen beide dat later duidelijk zal worden.

Rest ons nog een laatste *scope*: de ***dependent scope***. Dit is tevens de standaard-*scope* en geeft aan dat de *scope* afhangt van degene die de *bean* gebruikt. De *bean* krijgt dezelfde *scope* als diens gebruiker.

In het onderstaande schema worden de verschillende *scopes* en de interactie tussen browser en server visueel weergegeven: