



Advanced Java technologies: JBoss

Část 2.

Contexts and Dependency Injection (CDI)

Enterprise JavaBeans

March 2017

Matěj Novotný

@Inject

@SessionScoped

Contexts and Dependency Injection for the Java EE platform (CDI)

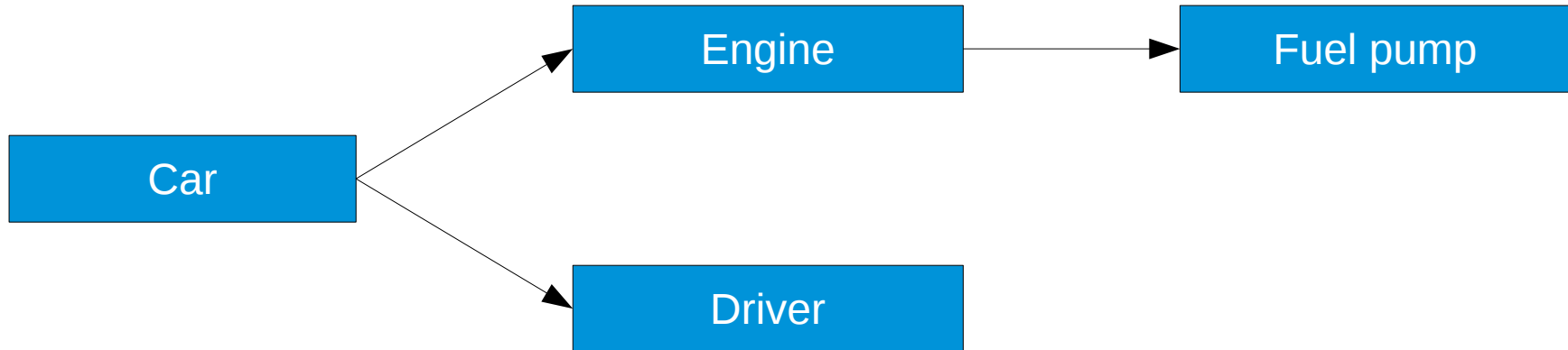
- Java EE specifikace
 - Od Java EE 6 (2009)
- Několik implementací
 - **Weld** (JBoss)
 - **OpenWebBeans** (Apache)
 - **CanDI** (Resin)
- Aktuální verze 1.2
- CDI 2.0 release bude (asi) v dubnu
 - Verze 2.0 momentálně ve fázi PFD

Contexts and Dependency Injection for the Java EE platform (CDI)

- Definuje komponentový model
 - komponenty spravuje server
 - komponentám jsou poskytované služby
 - správa životního cyklu
 - správa závislostí
 - etc.
- Integrace v rámci Java EE

Dependency Injection (@Inject)

- Inversion of control



Loose coupling (Volné propojení)

- Rozšiřitelnost a změny aplikace
- Testování



Strong typing

- Java (rozhraní, třídy, anotace)
- Refactoring
- Kompilátor

Charakteristika CDI Beany

- Třída
- Není abstraktní
- Má vhodný konstruktork
 - Bezparametrický
 - @Inject (viz. dále)
- Obsahuje “**bean defining annotation**”
 - Scope, @Interceptor, @Decorator
- V případě, že využívá proxy
 - Není final
 - Nemá final metody

- Session Beany (EJB) jsou zároveň CDI Beany

Vlastnosti CDI Beany

- Scope (viz. dále)
- Množina typů (Bean type closure)
 - Všechny nadtypy a všechny (i nepřímo) implementované rozhraní
- Množina kvalifikátorů (Qualifiers – viz. dále)
- Volitelně
 - Jméno (@Named)
 - Množina stereotypů (Stereotypes)
 - Množina interceptor vazeb (Interceptor bindings)

Příklad CDI Beany

```
@RequestScoped
public class Car {

    @Inject
    private Driver driver;
    @Inject
    private Engine engine;

    public void driveTo(Location location) {
        engine.start();
        // TODO
    }
}
```

Příklad CDI Beany

```
@SessionScoped
@Named
@Stateful
public class LoginManager implements Serializable {

    private static final long serialVersionUID = 1L;

    @Inject
    private EntityManager em;

    @Inject
    private UserManager userManager;

    private User currentUser;
```

Životní cyklus

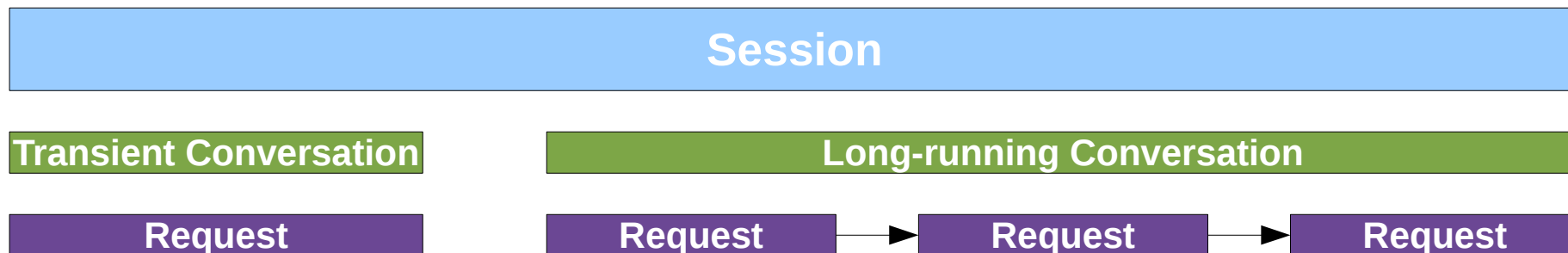
- Inversion of control – o životní cyklus se stará kontejner
- **Vytvoření nové instance** (viz. constructor injection)
- Field injection (naplnění atributů objektu)
- @PostConstruct / @Inject initializer
- **Uložení do kontextu** (scope)
- @PreDestroy / @Disposer method (viz. dále)

Kontexty - Scopes

- Servlet
 - Request Scope - @RequestScoped
 - Conversation Scope - @ConversationScoped
 - Session Scope - @SessionScoped
 - Application Scope - @ApplicationScoped
- Dependent Scope - @Dependent
 - Vždy se vytváří nová instance
 - Její životní cyklus je svázaný s Beanou která ji požaduje
- EE
 - @TransactionScoped (JTA)
 - @ViewScoped / @FlowScoped (JSF)

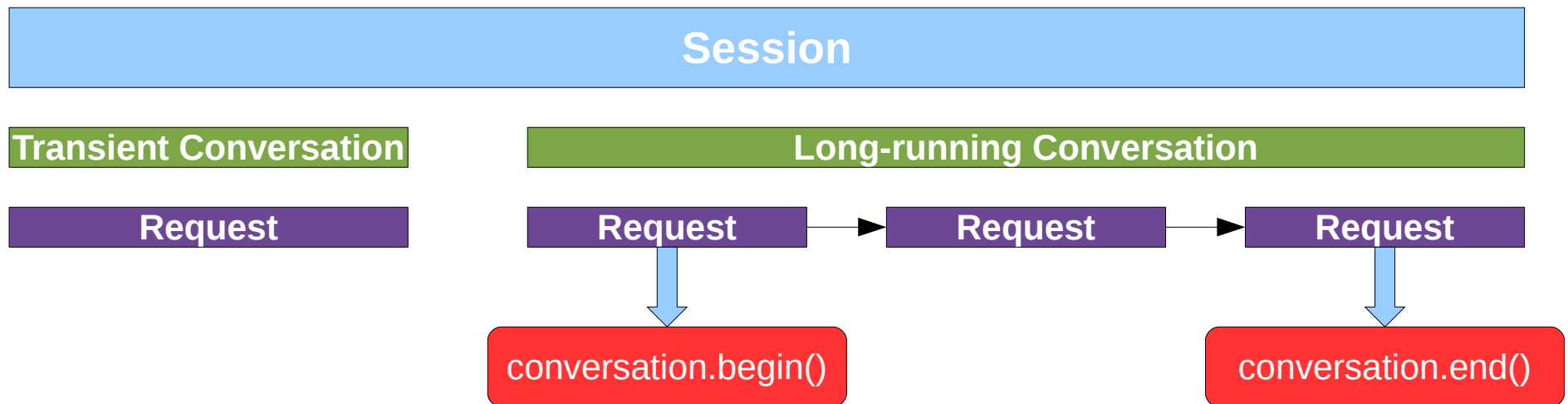
Conversation scope

- Uchovává stav sekvence po sobě jdoucích logicky souvisejících requestů
- Samotná konverzace má 2 stavy
 - **Transient** – jeden Servlet request
 - **Long-running** – sekvence Servlet requestů



Ovládaní konverzací

```
public class NewAuctionWizzard {  
  
    @Inject  
    private Conversation conversation;  
  
}
```



Conversation API

```
public interface Conversation
{
    public void begin();

    public void begin(String id);

    public void end();

    public String getId();

    public long getTimeout();

    public void setTimeout(long milliseconds);

    public boolean isTransient();
}
```

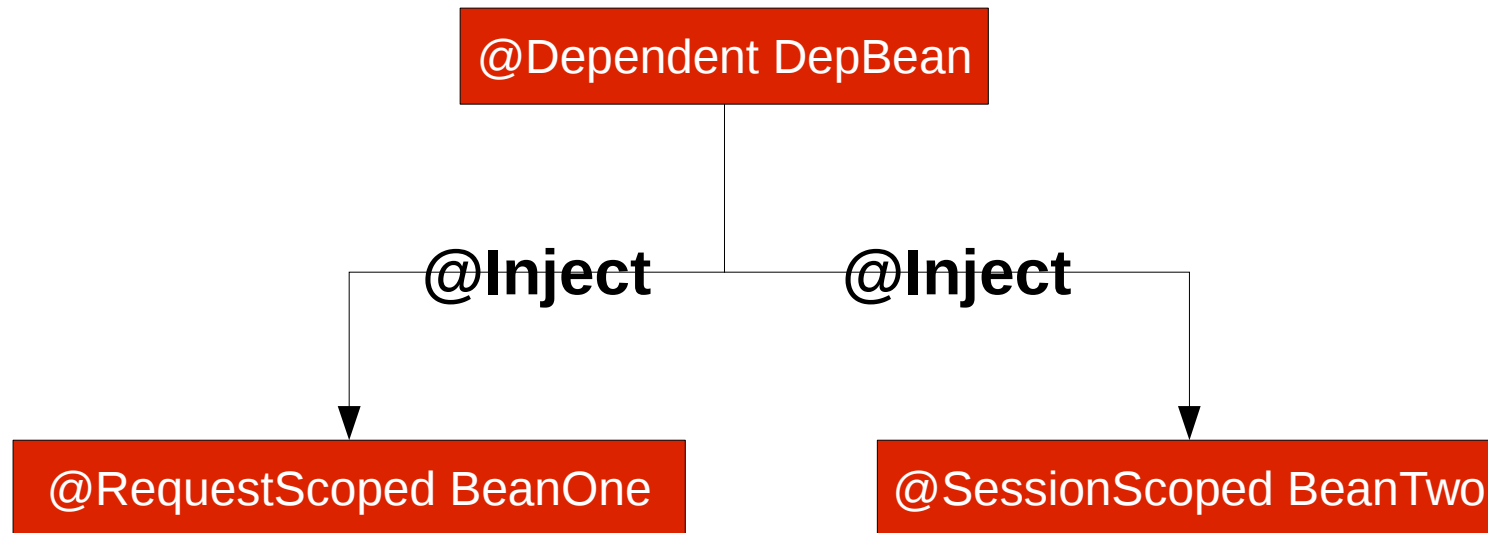
Propagace konverzací

- String identifier
 - Set by application
 - Generated by container
- V rámci session
- Propagace pomocí **cid** parametru
 - Automatická propagace při odesílání JSF formuláře
 - Manuální při plain Servlet request

/cdi-seminar/registration1.jsf?cid=1

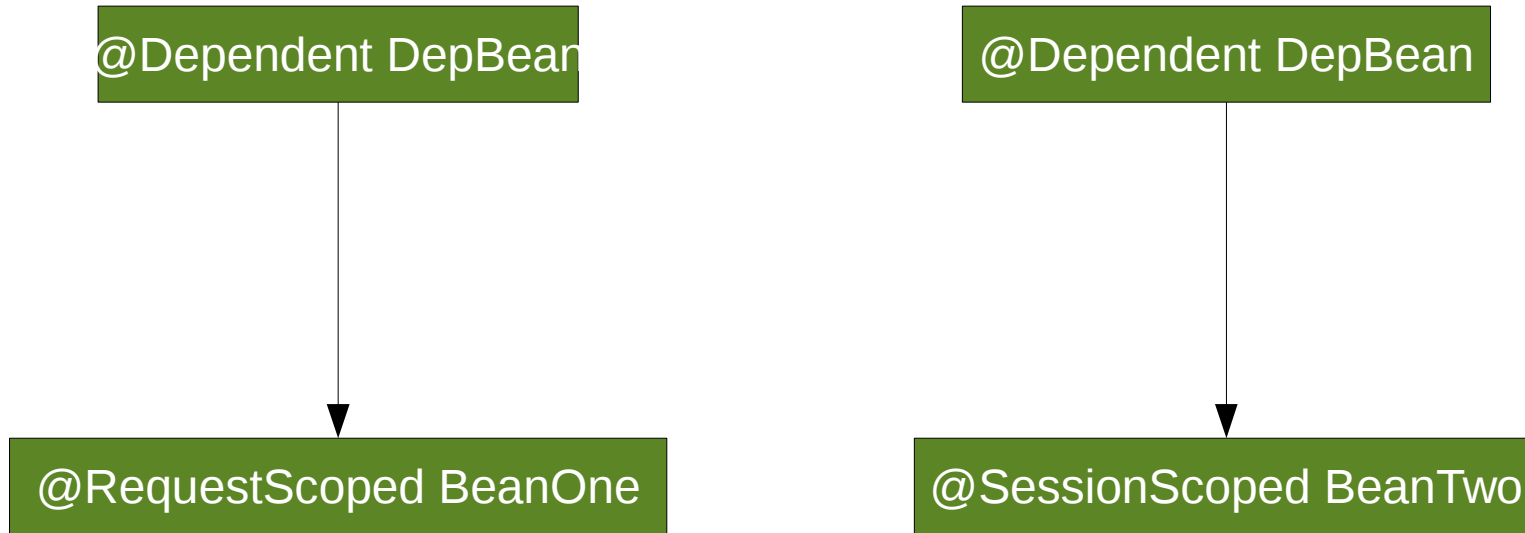
Dependent Pseudo Scope

- Na úrovni tříd



Dependent Pseudo Scope

- Na úrovni objektů



Passivation

- Velikost paměti potřebné pro uložení sessions roste lineárně s počtem existujících sessions
- Session a Conversation – passivating scopes
- Odkladání neaktivních sessions na sekundární úložiště
- Replikace sessions
 - Load balancing
 - Failover
- Každá @SessionScoped a @ConversationScoped komponenta musí být serializovatelná
 - `java.io.Serializable`
 - `java.io.Externalizable`

Dependency Injection

- Mechanismus získávání závislostí
- Spouští se explicitně pomocí anotace @Inject

```
@RequestScoped  
public class Car {  
  
    @Inject  
    private Driver driver;  
}
```



Pravidla DI - Typesafe resolution

- Výběr na základě
 - Typů
 - Kvalifikátorů (Qualifiers)
 - Slouží primárně na odlišení více instancí daného typu
- Na dependency injection se použije objekt daného typu
 - Z kontextu (pokud objekt existuje v kontextu)
 - Nový
 - vytvoří jej container
 - při `@Dependent` se vytváří vždy nový objekt

Qualifiers

```
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
public @interface LoggedIn {  
  
}
```

```
@Inject  
@LoggedIn  
private User user;
```

Pravidla DI - Typesafe resolution

- Injection point X vyžaduje

- Typ (Z)
- Množinu kvalifikátorů

- Bean Y poskytuje


- Množinu typů
- Množinu kvalifikátorů

- Bean Y je vhodným kandidátem pokud

- Vyžadovaný typ se nachází v množině poskytovaných typů
- Všechny požadované kvalifikátory se nachází v množině poskytovaných kvalifikátorů

```
public class Vehicle
    @Wild
    public class Cat
    @Domestic
    public class Dog

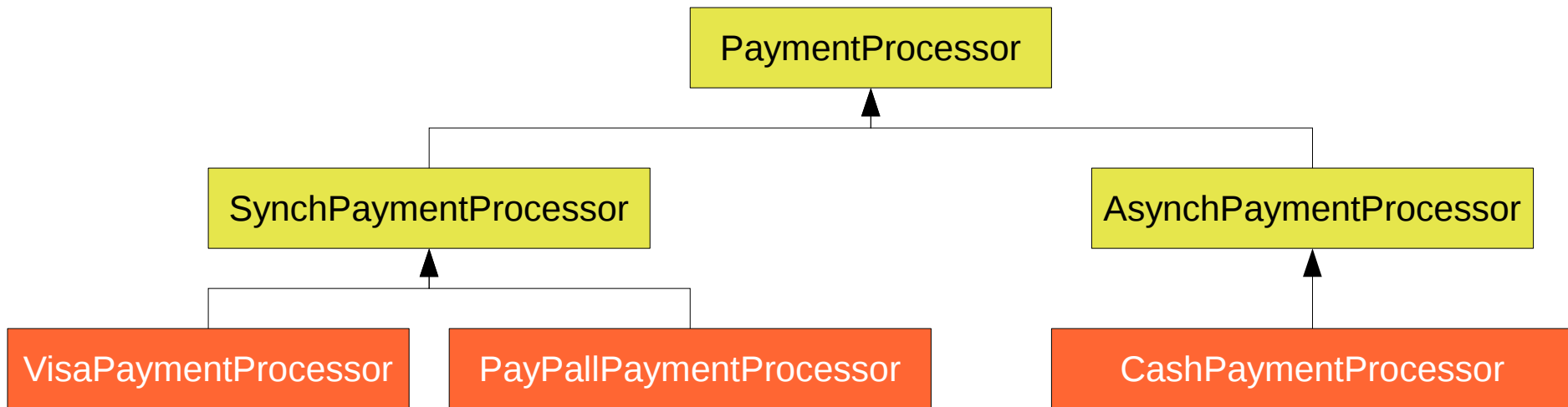
@Inject
@Wild
Animal animal;
```



Poskytované typy (Bean types)

- Tranzitivní uzávěr rozšiřovaných a implementovaných typů
- `java.lang.Object` je vždy typem
- `java.io.Serializable` není nikdy typem

Příklad typové rezoluce (typy)



```
@Inject  
private PaymentProcessor pp;
```

```
@Inject  
private SynchronPaymentProcessor spp;
```

```
@Inject  
private VisaPaymentProcessor vpp;
```

Příklad typové rezoluce (kvalifikátory)

```
public interface Animal
```

```
public class Sheep
```

```
@Domestic  
public class Dog
```

```
@Wild  
@Hungry  
public class Lion
```

```
@Wild  
public class Elephant
```

```
@Domestic  
@Lazy  
public class Cat
```

```
@Inject  
@Any  
private Animal animal;
```

```
@Inject  
@Domestic  
private Animal domestic;
```

```
@Inject  
@Wild @Lazy  
private Animal lazyWildAnimal;
```



Speciální kvalifikátory

- @Default – implicitní kvalifikátor v případě, že daná komponenta / injection point nedefinuje jiný kvalifikátor
- @Any – implicitní kvalifikátor každé komponenty
- @Named

```
public class Shelter {
```

```
    @Inject  
    Animal animal1;
```

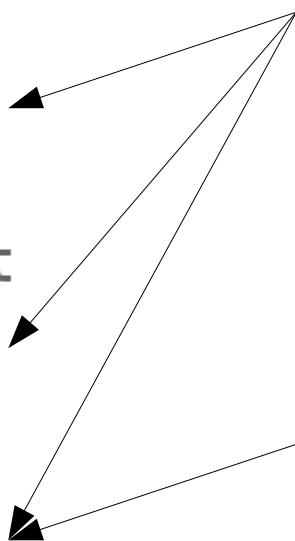
```
    @Inject @Default  
    Animal animal2;
```

```
    @Inject @Any  
    Animal animal3;
```

```
}
```

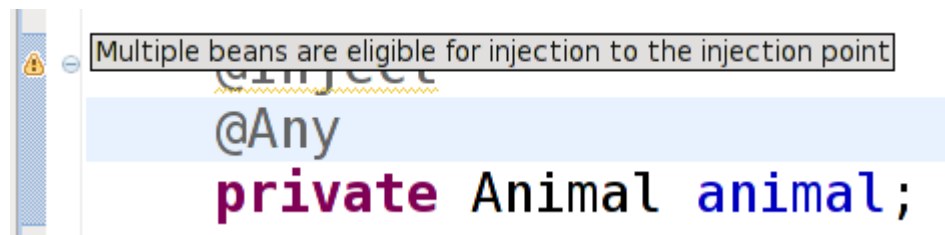
```
public class Dog  
    implements Animal {  
}
```

```
@Domestic  
public class Cat  
    implements Animal {  
}
```



Pravidla DI - Typesafe resolution

- 1 injection point = právě jedna vhodná Bean
- Kontrola při deploy (nebo tooling)



The screenshot shows an IDE window with a warning message: "Multiple beans are eligible for injection to the injection point". Below the message, the code is displayed with a light blue highlight on the line containing the annotation `@Any`. The code snippet is:

```
private Animal animal;
```

Typy DI

- Field injection
- Constructor
 - Immutable objects
- Initializer methods
- Producer method/Disposer/Observer

```
@Inject  
private PaymentProcessor pp;
```

```
@Inject  
public PaymentManager(PaymentProcessor pp) {  
    this.pp = pp;  
}
```

```
@Inject  
public void init(PaymentProcessor pp) {  
    this.pp = pp;  
}
```

Producer method

- Factory - pro případy kdy volání konstrukturu třídy nestačí na vytvoření objektu
- Rozšířená kontrola
 - Výsledek JPA dotazu
 - Náhodné číslo
 - ...
- Konkrétní metoda (není abstraktní)
 - Umístěná na CDI Bean
 - Tranzitivní uzávěr typu návratové hodnoty = typ objektu
 - Kvalifikátory, Scope a jméno objektu se definují na metodě

Producer method

```
@ApplicationScoped
public class RandomNumberGenerator {

    private final java.util.Random random = new java.util.Random();

    @Produces
    @Random
    public int generateRandomInt() {
        return random.nextInt();
    }
}
```

Producer method

```
@Produces
@Named
@CurrentAuction
public Auction getCurrentAuction() {
    if (currentAuction != null &&
        !em.contains(currentAuction)) {
        currentAuction = em.merge(currentAuction);
    }
    return currentAuction;
}
```


Producer field

- Zjednodušení producer metody

```
@Produces  
@Zero  
private final int zero = 0;
```

Disposer method

- Některé objekty vyžadují explicitní zničení
- @PreDestroy nefunguje na produkováných objektech

```
public void close(@Disposes Connection connection) {  
    try  
    {  
        connection.close();  
    }  
    catch (SQLException e)  
    {  
        e.printStackTrace();  
    }  
}
```

Alternatives

- @Alternative
- Nahrazení komponenty jinou na základě prostředí
 - Konfigurace
 - Testování
- Alternativa musí být explicitně aktivovaná (enabled) pomocí anotace **@Priority**
- Rozšíření **Typesafe resolution** algoritmu
 1. Výběr vhodných kandidátů na základě typu a kvalifikátorů
 2. V případě, že existuje víc jak jeden kandidát jsou eliminováni všichni kandidáti kromě aktivovaných alternatives
 3. V případě, že všichni kandidáti mají definovanou prioritu, je vybrán ten s nejvyšší prioritou

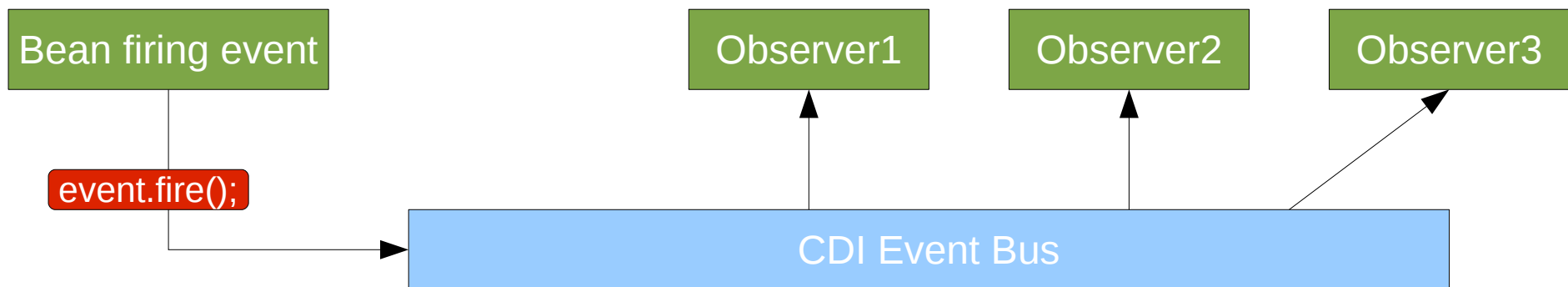
Integrace s prezentační vrstvou

- Přístup k CDI komponentám pomocí jména
- @Named

```
<h:form>
  <h:inputText value="#{student.firstName}" />
  <h:inputText value="#{student.surname}" />
  <h:commandButton id="register"
    action="#{registrator.register}" value="Register" />
</h:form>
```

Události (Events)

- Komunikace pomocí odesílání zpráv
- Oddělení producentů zpráv od jejich konzumentů
- Loose coupling - další observery můžeme přidat později bez zásahu do producenta událostí
- Synchronní (Asynchronní k dispozici v CDI 2.0)
- Event = message (payload)



Události (Events) - Odesílání zprávy

- In-built Event bean

```
public class RegistrationManager {  
    @Inject  
    @Registered  
    private Event<User> userEvent;  
}
```

- Vyvolání události

```
public void register()  
{  
    em.persist(user);  
    userEvent.fire(user);  
}
```

Vlastnosti zprávy

- Typ zprávy
- Qualifiers

```
public class RegistrationManager {  
    @Inject  
    @Registered  
    private Event<User> userEvent;  
}
```

Události (Events) - Doručení zprávy

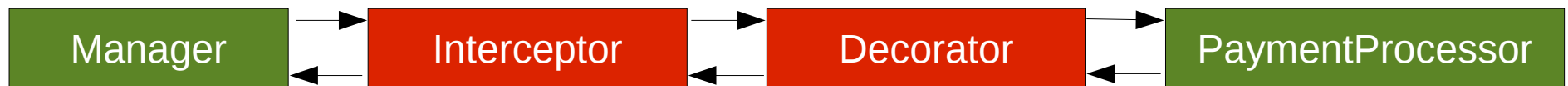
- Observer method
- Pravidla typové rezoluce stejné jako pro DI

```
public void log(@Observes User user) {  
    System.out.println("Hello " + user.getName());  
}
```

```
public void log(@Observes @Registered User user) {  
    System.out.println("Hello " + user.getName());  
}
```


Dekorátory a Interceptory

- Aspect-oriented programming
- Odchytávání volání metod
- Izolování kódu, který bychom jinak opakovali v metodách
- Změna chování v závislosti na prostředí

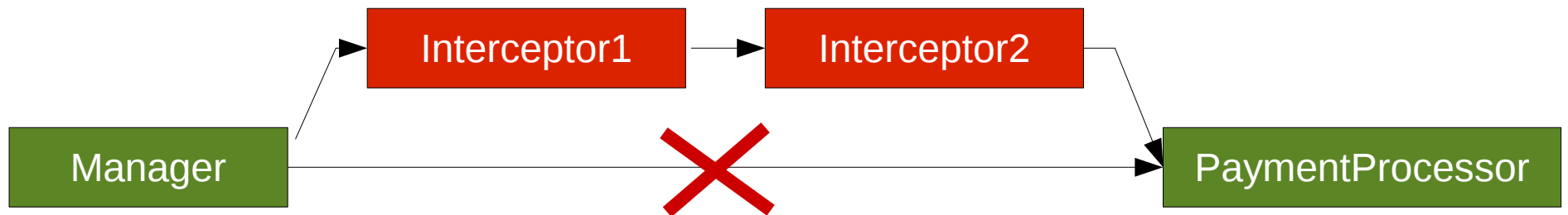


Interceptory

- Oddělení technických detailů od logiky aplikace
- Cross-cutting concerns (logování, zabezpečení, caching)
- Můžu odchyťávat
 - Volání metod (@AroundInvoke)
 - Životní cyklus
 - @AroundConstruct
 - @PostConstruct
 - @PreDestroy

Interceptory

```
public class PaymentManager {  
  
    @Inject  
    private PaymentProcessor pp;  
  
    pp.foo();  
}
```



Interceptory

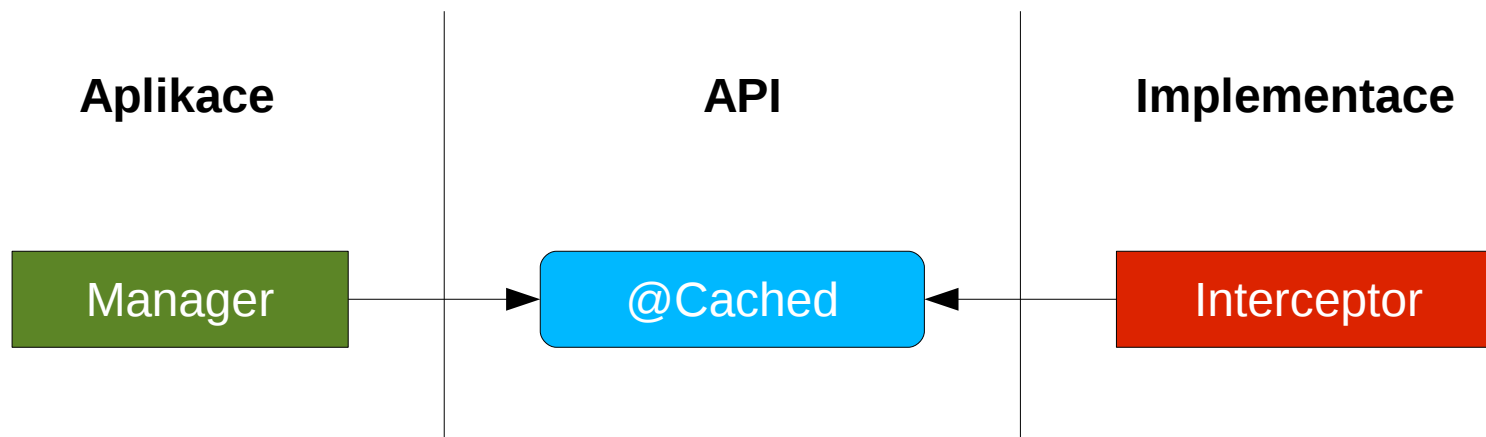
```
@Interceptor
@Cached
@Priority(Interceptor.Priority.APPLICATION)
public class CachingInterceptor {

    @AroundInvoke
    Object intercept(InvocationContext ctx) throws Exception {
        // do something before the call
        Object result = ctx.proceed();
        // do something after the call
        return result;
    }
}
```

Interceptor binding

- Anotace reprezentující funkcionalitu poskytovanou interceptorem (loose coupling)

```
@InterceptorBinding  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Cached {  
  
}
```



Decorator

- “Typovaný” interceptor – “pozná” sémantiku volání které obaluje
- Není nutné anotovat odchyťávaný objekt
- Implementuje přímo metody rozhraní které obaluje
 - Nemusí implementovat všechny metody rozhraní (abstraktní třída)
- Speciální injection point (**@Delegate**)

Decorator

```
public interface Account {  
    void withdraw(BigDecimal amount);  
}
```

@Dependent

```
public class AccountImpl implements Account {  
  
    private BigDecimal balance;  
  
    public AccountImpl(BigDecimal initialBalance) {  
        this.balance = initialBalance;  
    }  
  
    public void withdraw(BigDecimal amount) {  
        balance.subtract(amount);  
    }  
}
```

Decorator

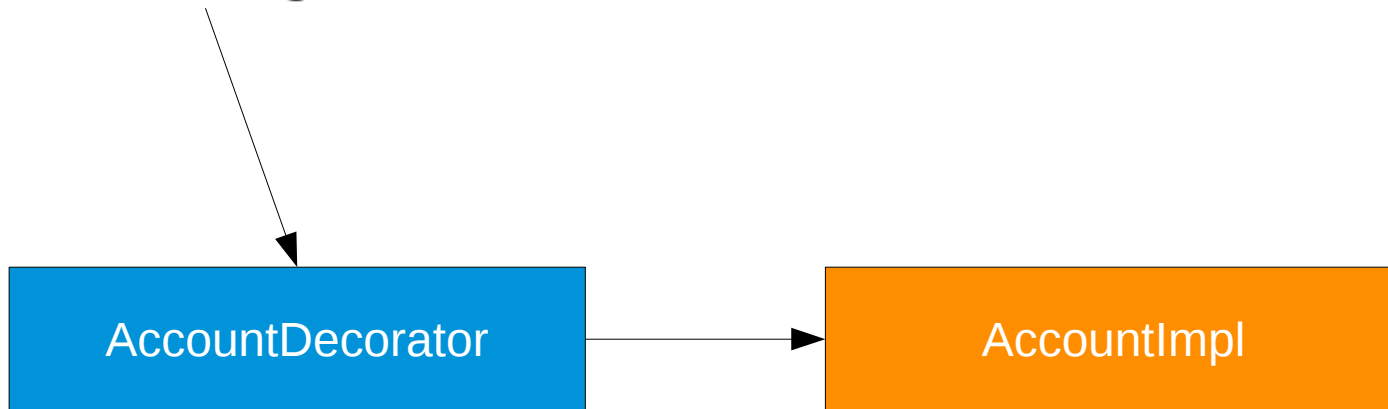
```
@Inject  
private Account account;  
account.withdraw(BigDecimal.TEN);
```



AccountImpl

Decorator

```
@Inject  
private Account account;  
account.withdraw(BigDecimal.TEN);
```



Decorator

```
public class AccountDecorator implements Account {

    private final Account delegate;

    public AccountDecorator(Account delegate) {
        this.delegate = delegate;
    }

    @Override
    public void withdraw(BigDecimal amount) {
        if (amount.compareTo(LIMIT) > 0 &&
            !isAuthorizedForLargeTransactions()) {
            throw new SecurityException("Amount too high!");
        }
        delegate.withdraw(amount);
    }
}
```

Decorator

```
@Decorator
@Priority(Interceptor.Priority.APPLICATION)
public class AccountDecorator implements Account {

    private final Account delegate;

    @Inject
    public AccountDecorator(@Delegate @Any Account delegate) {
        this.delegate = delegate;
    }

    @Override
    public void withdraw(BigDecimal amount) {
        if (amount.compareTo(LIMIT) > 0 &&
            !isAuthorizedForLargeTransactions()) {
            throw new SecurityException("Amount too high!");
        }
        delegate.withdraw(amount);
    }
}
```

Aktivace a pořadí

- @Priority(2000) - magic number
- Interceptors, Decorators, Alternatives

Interceptor.PRIORITY.

0 - 999	PLATFORM_BEFORE
1000 - 1999	LIBRARY_BEFORE
2000 - 2999	APPLICATION
3000 - 3999	LIBRARY_AFTER
4000 - 4999	PLATFORM_AFTER

- Transactional interceptor (200)
- Bean Validation interceptor (4800)

Stereotypy

- Obrana před “annotation hell”
- Anotace sdružující
 - Scope
 - Interceptor bindings
 - @Named
- Vestavěný stereotyp **@Model**

```
@Named
@RequestScoped
@Documented
@Stereotype
@Target( { TYPE, METHOD, FIELD } )
@Retention(RUNTIME)
public @interface Model
{
}
```

Integrace

- Beans
 - @Inject **HttpServletRequest**
 - @Inject **UserTransaction**
 - @Inject **javax.security.Principal**
- Events
 - **@Initialized(SessionScoped.class) HttpSession**
- Scopes
 - **@TransactionScoped**
 - **@ViewScoped / @FlowScoped**

Integrace - non-contextual components

- Servlets / Filters
- JPA Entity listeners
- JSF converters / validators
- JAX-WS / JAX-RS / WebSocket endpoints

- @Inject (the component itself cannot be injected anywhere)
- Firing events (no observing support)
- Interceptors

Contexts and Dependency Injection

- Vlastnosti CDI komponent
- Životní cyklus (Scopes)
- Dependency Injection
- Události (Events)
- Dekorátory a Interceptory (Decorators and Interceptors)
- Integrace

Tipy na závěr

- Nezapomenout **scope annotation**
- Pozor na správné FQCN!
 - javax.inject.Inject
 - javax.enterprise.inject.Produces
 - javax.enterprise.context.*Scoped
 - javax.enterprise.event.Event
 - javax.enterprise.event.Observes
 - javax.enterprise.inject.spi.Extension;

Enterprise JavaBeans

- Komponentový model poskytující služby:
 - Resource injection
 - Transactions
 - **Security** (separátní přednáška)
 - Remote method invocation (RMI) - @Remote
 - Naming and directory services - JNDI
 - **Messaging** (JMS)
 - **Asynchronous invocations**
 - **Scheduling** (Timer service)
 - Concurrency control

Enterprise JavaBeans

- Session beans
 - Stateless
 - Stateful
 - Singleton
- Message-driven beans

Enterprise JavaBeans 3.1

- Zjednodušení existujícího komponentového modelu
 - Rozhraní (views) nejsou povinná
 - Session beans jsou automaticky CDI komponentami
 - Scopes (Stateful session bean)
 - Dependency injection
 - Events
 - Interceptory / Dekorátory
 - Možnost používat v jednoduché webové aplikaci (.war)
- **@Singleton** session bean
- Asynchronní volání metod

Singleton session bean

- Jediná instance sdílená v rámci celé aplikace
 - V rámci jedné JVM
- V porovnání s **@ApplicationScoped** navíc
 - Inicializovaná při startu aplikace (**@Startup**)
 - Podporuje paralelní přístup
 - @Lock(LockType.READ)
 - @Lock(LockType.WRITE)
 - @AccessTimeout
 - @ConcurrencyManagement
- DEMO na cvičení

Asynchronní volání metod

- Abstrakce od přímého použití vláken, exekutorů a thread pools
- Použití
 - Asynchronní vykonávání dlouhotrvajících úloh
 - Odesílání potvrzujícího e-mailu
 - Návrátová hodnota **void**
 - Paralelizace výpočtu
 - Návrátová hodnota **Future<V>** - **AsyncResult<V>**
- **@Asynchronous**
- DEMO na cvičení

Enterprise JavaBeans 3.2

- Převážně kosmetické změny
 - `@Stateful(passivationCapable = false)`
 - `@PostConstruct/@PreDestroy` volitelně transakční
 - `TimerService.getAllTimers()`
 - Všechny Timers pro daný modul

Odkazy

- <http://cdi-spec.org/>
- <http://docs.jboss.org/cdi/spec/1.2/>
- <http://docs.oracle.com/javaee/7/tutorial/doc/>

CDI 2.0 - přehled novinek

- **SE Bootstrap**
- Manuální aktivace kontextu
- Řazení observerů
- **Asynchronní události**
- Configurators SPI
- Interceptory pro producery
- A další...

CDI 2.0 - SE Bootstrap

- CDI je možné použít v Java SE
- Start/stop CDI kontejneru
 - Extends AutoCloseable
- Volitelné, co je na classpath
 - Tzv. synthetic bean archive
- Dostupné scopes - Application, Dependent (, Request)
- beans.xml je vyžadováno
- Krátké demo na cvičeních

CDI 2.0 - Asynchronní události

- Opět z rozhraní Event, pomocí metody `fireAsync(...)`
- Observer musí být specificky monitorovat asynchronní události
 - `@ObservesAsync @MyQualifier Payload payload`
- Nelze mít observer pro sync i async události zároveň
- Demo na cvičeních

	<code>callMe(@Observes payload)</code>	<code>callMe(@ObservesAsync payload)</code>
<code>event.fire(payload)</code>	Sync. call	Not notified
<code>event.fireAsync(payload)</code>	Not notified	Async call



Questions?

manovotn@redhat.com | www.redhat.com