

Sovelto

Bean Validation

Valmiit annotaatiot

- `@AssertFalse`, `@AssertTrue`: Elementin tulee olla `false/true`
- `@DecimalMax`, `@DecimalMin`: Elementin tulee olla arvoltaan tietyn kokoinen; tuetut tyypit: `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` (sekä primitiivityyppien wrapper-luokat); `null` on validi
- `@Digits`: Kokonaisluvun ja desimaaliosan numeroiden lukumäärät
- `@Future`, `@Past`: Päivämääräarvo, joka on tulevaisuudessa
- `@Max`, `@Min`: Sama kuin `@DecimalXXX`, mutta ei merkkijonoille
- `@NotNull`, `@Null`: `null`-arvojen käsittely
- `@Pattern`: Säännöllinen lauseke (`null` on validi)
- `@Size`: Elementin koko; tuetut tyypit: `String`, `Collection`, `Map`, taulukko (`null` on validi)

Annotaatioiden käyttö

- Annotaation voi liittää joko instanssimuuttujaan tai propertyyn (tarkemmin: `get`-metodiin); omien annotaatioiden yhteydessä myös luokkaan
 - Jos käyttää molempia, niin validointi tapahtuu kahteen kertaan
 - Validointi tapahtuu suoraan kentän kautta, mikäli se on annotoitu ja vastaavasti kutsumalla metodia, mikäli annotaatio on metodissa
 - Bean Validation 1.1:ssä (Java EE 7) vaatimukset voi kohdistaa myös metodiin (eli sen paluuarvoon) tai metodin parametreihin
- Elementin näkyvyydellä ei ole merkitystä
- Staattisia jäseniä ei voi validoida

Annotaatioiden käyttö

- Jos luokkia periytetään, niin myös annotaatiot periytyvät
 - Lapsiluokka voi myös lisätä jäsenille uusia validointeja
- Oliograafi voidaan validoida merkitsemällä olioviittaukselle annotaatio `@Valid`
 - Esimerkki (kokoelman tapauksessa kaikki alkiot validoidaan):

```
public class Tilaus {  
    ...  
    @NotNull  
    @Valid  
    private List<Tilausrivi> tilausrivit;  
    ...  
}
```

- Oliograafien tapauksessa `null`-arvoja ei huomioida

Ohjelmallinen validointi

- Ohjelmallinen validointi tehdään `Validator`-olion avulla
 - Säieturvallinen
 - Uudelleenkäytettävä
- Paluuarvona on `Set` virheolioita; kaiken ollessa kunnossa kokoelma on tyhjä
- Esimerkki koko olion validoinnista:

```
validatorFactory validoijatehdas =  
    validation.buildDefaultValidatorFactory();  
validator validoija = validoijatehdas.getValidator();  
Asiakas asiakas = new Asiakas();  
asiakas.setNimi("Malli");  
asiakas.setSalasana("foo");  
Set<ConstraintViolation<Asiakas>> virheet =  
    validoija.validate(asiakas);
```

Ohjelmallinen validointi

- Esimerkki yksittäisen kentän validoinnista:

```
Set<ConstraintViolation<Asiakas>> virheet =  
    validoija.validateProperty(asiakas, "salasana");
```

- Esimerkki tietyn arvon oikeellisuuden testaamisesta:

```
Set<ConstraintViolation<Asiakas>> virheet =  
    validoija.validateValue(Asiakas.class,  
        "salasana", "abc");
```

- Huom.: Näissä tapauksissa @Valid-annotaatiota ei huomioida

Virheilmoitusten muokkaaminen

- Mahdollinen lokalisointi tehdään tiedostoihin `/validationMessages_xx.properties`; esimerkki:

```
javax.validation.constraints.Size.message=\n    Pituuden tulee olla välillä {min} - {max}
```

- Oma, kiinteä virheilmoitus on myös mahdollista määritellä annotaation parametrilla `message`

```
@NotNull(message = "Tieto ei voi puuttua")
```

- Tai lokalisointiavain voidaan vaihtaa:

```
@NotNull(message = "{{fi.sovello.tarkistukset.einull}}")
```

Validointiryhmät

- Validointiryhmät mahdollistavat validointien rajoittamisen vain haluttuihin eri tilanteissa
- Ryhmät kuvataan tyhjiä rajapinnoilla; esimerkki:

```
public interface Perustietotarkistukset { }
```

- Ryhmä määritetään annotaation parametrilla `groups`:

```
@NotNull(groups = {Default.class, Perustietotarkistukset.class})  
@Size(min = 6, max = 20,  
      groups = {Default.class, Perustietotarkistukset.class})  
private String salasana;
```

- Oletusarvoinen ryhmä (jos muuta ei määritellä):
`javax.validation.groups.Default`

Validointiryhmät

- Esimerkki ryhmän käytöstä ohjelmallisessa validoinnissa:

```
Set<ConstraintViolation<Asiakas>> virheet =  
    validoija.validate(  
        asiakas, Perustietotarkistukset.class);
```

- Oletusarvoisesti ryhmätarkistukset tapahtuvat satunnaisessa järjestyksessä (jos ryhmiä on useita); tämä voidaan määrätä annotaatiolla @GroupSequence:

```
@GroupSequence({Perustietotarkistukset.class, Default.class})  
public interface JarjestetytTarkistukset { }
```

Validointiryhmät

- Annotaatiolla `@GroupSequence` voidaan myös määrittää luokkakohtaisesti uusi oletusarvo sille, mitä `Default.class` tarkoittaa validoitaessa kyseinen luokka
 - Esimerkki:

```
@GroupSequence({Asiakas.class, Perustietotarkistukset.class})  
public class Asiakas implements Serializable {  
    private static final long serialVersionUID = 1L;  
    ...  
  
    @Size(min = 2, max = 50, groups = Perustietotarkistukset.class)  
    private String nimi1;
```

```
Set<ConstraintViolation<Asiakas>> virheet =  
    validoija.validate(asiakas);
```

Myös tämä
validoidaan,
vaikka kutsussa
on vain
`Default.class`

- Huom.: Luokan oma nimi tulee olla listassa

Oman rajoitusehdon luonti

- Oman rajoitusehdon luonnissa on kolme vaihetta:
 - Annotaation luonti
 - Validoijan logiikan kirjoittaminen
 - Oletusarvoisen virheilmoituksen luonti
- Esimerkki annotaatiosta:

```
public enum Aakkoskokotyyppi { PIENI, SUURI }
```

```
@Target({ ElementType.METHOD, ElementType.FIELD,  
          ElementType.ANNOTATION_TYPE })  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Constraint(validatedBy = Aakkoskokovalidaattori.class)  
public @interface Aakkoskoko {  
    String message() default "{fi.sovelto.tarkistukset.aakkoskoko}";  
    Class<?>[] groups() default {};  
    Class<? extends Payload>[] payload() default {};  
    Aakkoskokotyyppi haluttuKoko();  
}
```

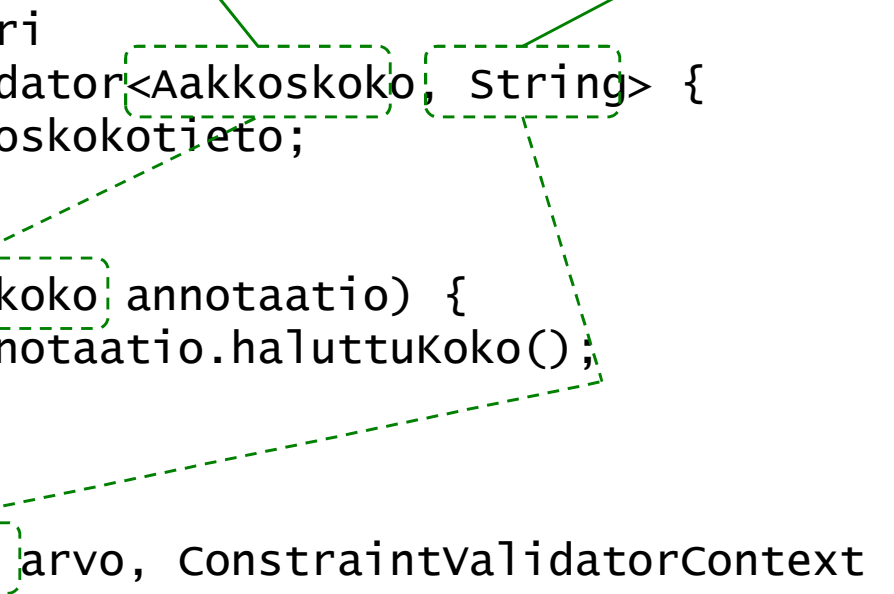
Oman rajoitusehdon luonti

- Oma logiikkaluokka:

```
public class Aakkoskokovalidaattori
    implements ConstraintValidator<Aakkoskoko, String> {
    private Aakkoskokotyyppi aakkoskokotieto;

    @Override
    public void initialize(Aakkoskoko annotaatio) {
        this.aakkoskokotieto = annotaatio.haluttukoko();
    }

    @Override
    public boolean isValid(String arvo, ConstraintValidatorContext cvc) {
        if (arvo == null) {
            return true;
        }
        if (aakkoskokotieto == Aakkoskokotyyppi.PIENI) {
            return arvo.equals(arvo.toLowerCase());
        } else {
            return arvo.equals(arvo.toUpperCase());
        }
    }
}
```



Oman rajoitusehdon luonti

- Lokalisointitiedosto `ValidationMessages.properties`:

```
fi.sovello.tarkistukset.aakkoskoko=\nAakkosten koon tulee olla {haluttuKoko}
```

- Validoijan käyttö koodissa:

```
@Aakkoskoko(haluttuKoko = Aakkoskokotyyppi.PIENI)\nprivate String extranetTunnus;
```

Oman rajoitusehdon luonti

- Omia rajoitusehtoja voi luoda myös yhdistämällä valmiista annotaatioista oma tyyppinsä:

```
@NotNull
@Size(min = 6, max = 20)
@Aakkoskoko(haluttuKoko = Aakkoskokotyyppi.PIENI)
@Target({ ElementType.METHOD, ElementType.FIELD,
          ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Extranettunnus {
    String message() default "{fi.sovelto.tarkistukset.extranet}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

- Tässä tapauksessa jokainen validoija tulostaa oman virheilmoituksensa, annotaatiolla `@ReportAsSingleViolation` käytetään annotaation omaa

Luokkakohtainen annotaatio

- Luokkakohtaista annotaatiota voi käyttää esimerkiksi riippuvuuksien tarkistamiseen; esimerkki:

```
@Target({ ElementType.Type, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Constraint(validatedBy=Autokapasiteettivalidaattori.class)
public @interface Autokapasiteettitarkistus {
    String message() default
        "{fi.sovelto.tarkistukset.autokapasiteetti}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

```
@Autokapasiteettitarkistus
public class Auto {
    @Min(1)
    private int istuinlkm;
    private List<String> henkilonimet = new ArrayList<String>();
    ...
}
```

Luokkakohmainen annotaatio

- Esimerkki jatkuu...:

```
public class Autokapasiteettivalidaattori
    implements ConstraintValidator<Autokapasiteettitarkistus, Auto> {

    @Override
    public void initialize(Autokapasiteettitarkistus annotaatio) { }

    @Override
    public boolean isValid(Auto auto, ConstraintValidatorContext c) {
        if (auto == null) {
            return true;
        }
        if (auto.getHenkilonimet().size() > auto.getIstuinlkm()) {
            return false;
        } else {
            return true;
        }
    }
}
```


JSF-integraatio

- JSF 2.x:n myötä myös Bean Validation -tekniikka on tuettu backing beanien validoinnissa
- Esimerkki:

```
@ManagedBean(name = "validointiBB")
@RequestScoped
public class ValidointiBB implements Serializable {
    private static final long serialVersionUID = 1L;
    @NotNull
    @Size(min = 2)
    private String lahiosoite;
    ...
}
```

- Jos sivuun liittyen halutaan validoida vain tietyt validointiryhmät, voidaan tämä määritellä `f:validateBean`-elementin `validationGroups`-attribuutilla

JPA-integraatio

- JPA 2:n yhteydessä validointi tapahtuu oletusarvoisesti aina seuraavissa tilanteissa (tarkkaan ottaen heti näihin liittyvän elinkaarimetodin jälkeen):
 - `@PrePersist`
 - `@PreUpdate`
 - `@PreRemove`
- Validointia voi hallita seuraavilla avaimilla:
`javax.persistence.validation.group.pre-persist`,
`javax.persistence.validation.group.pre-update` ja
`javax.persistence.validation.group.pre-remove` ja
`javax.persistence.validation.mode` (viimeisimmällä arvot `auto` (oletus), `callback` (luonti, luku ja poisto) sekä `none`)
- On tosin hyvä miettiä, kuuluuko kannan sisällön validointi Java-koodiin

Bean Validation: Best practices

- Käytä tekniikkaa etupäässä käyttäjän syötteen varmistamiseen
- Vaikka lähes kaikki validoinnit on tehtävissä säännöllisillä lausekkeilla, on ylläpidettävyyden vuoksi parempi tehdä erillisiä validoijia

Sovelto

CDI:n perusteet

Mikä on CDI?

- *Contexts and Dependency Injection for the Java EE Platform*
- Context = Määrittelyalue, esimerkiksi pyyntö (*request*) tai istunto (*session*)
- Dependency Injection = Riippuvuuksien injektointi, tuttu esimerkiksi Spring-sovelluskehiksestä
- Miksi? Miksi Java EE 5:n vastaavat piirteet eivät ole riittäneet?
Tärkeimpiä syitä:
 - Injektoinnit ja muut palvelut saadaan käyttöön missä tahansa luokassa
 - Injektoida voi mitä vain, aiemmin tämä oli hyvinkin rajattua
 - Toteutusten vaihtaminen helposti esimerkiksi testauksessa
 - Uusia palveluja, kuten tapahtumat ja decoratorit

Mikä on bean?

- Aiemmin termillä ei ole ollut varsinaista yleistä määritelmää Java EE:n yhteydessä; määritelmät ovat olleet tekniikkakohtaisia
 - EJB
 - JSF:n managed bean
 - Spring:n bean
- Managed Bean -spesifikaatio toimii uutena määritelmänä
 - Ei juuri ohjelmointirajoituksia (eli POJO)
 - Peruspalvelut, kuten resurssien injektointi, elinkaarimetodien annotointi ja interceptorit
- Muut spesifikaatiot (kuten EJB tai CDI) laajentavat tätä

Managed beanin vaatimukset

- Parametriton konstruktori tai konstruktori, jolla on annotaatio `@javax.inject.Inject`
- Java EE:n 6-versiossa beanin tulee olla JAR-paketissa, jossa on mukana tiedosto `META-INF/beans.xml` (WAR: `WEB-INF/beans.xml`)
 - Tiedosto voi olla täysin tyhjä; mikäli tiedostossa on määrittelyksiä, kannattaa editoinnin helpottamiseksi hyödyntää seuraavaa XML Schema -määritystä:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                          http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

- CDI 1.1:n Schema (EE 7:ssä tiedoston voi jättää halutessaan pois):

```
<beans xmlns="http://java.sun.com/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">
</beans>
```

Injektointi

- Luokka voi saada injektoinnilla viittauksen toiseen beaniin kolmella tavalla:

- Kenttä: `@Inject private Injektoitava viittaus1;`

- Konstruktori (voi olla vain yksi injektoiva konstruktori):

```
@Inject  
public Injektoiva(Injektoitava viittaus2) {  
    this.viittaus2 = viittaus2;  
}
```

- Asetusmetodi (voi olla useita):

```
@Inject  
private void asetaKohde(Injektoitava viittaus3) {  
    this.viittaus3 = viittaus3;  
}
```


Beanin tyyppi

- Oikea kohde löydetään seuraavilla tiedoilla:
 - Tyyppi (luokka tai rajapinta, EJB-komponenteilla vain local-näkymä)
 - Joukko määrittäjiä (*qualifier*, käytännössä oma annotaatio)
- Esimerkki tyypistä: Seuraavalla komponentilla on tyypit `Tili`, `Lokitettava` ja `java.lang.Object` (luokan nimi ja kantaluokka eivät ole tyyppejä, koska EJB-komponentin luokkatyyppi ei näy ulkopuolelle):

```
@Stateless  
public class TiliEJB extends Pankkisovellus  
    implements Tili, Lokitettava { }
```

Beanin määrittäjä

- Koska usea bean voi toteuttaa saman rajapinnan, pelkkä tyyppi ei yleensä riitä oikean injektointikohteen löytämiseen
- Tämän vuoksi kohteet voidaan tarkentaa tekemällä omia annotaatioita, jotka edelleen on merkitty annotaatiolla `javax.inject.Qualifier`
- Esimerkki omasta määrittäjästä (muiden annotaatioiden paketti on `java.lang.annotation`):

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER,
        ElementType.FIELD})
public @interface Kirjakauppakanta { }
```

Beanin määrittäjä

- Injektointiviittaus:

```
@Inject @Kirjakauppakanta private EntityManager em;
```

- Esimerkki metodista, joka kertoo *tuottavansa* kyseisellä annotaatiolla merkittyjä olioita:

```
@Produces @Kirjakauppakanta EntityManager luoEM() {  
    return omaEntityManagerFactory.createEntityManager();  
}
```

- Tämä ei ole välttämätön; jos tuottajaa ei ole merkitty, instantioi container olion itse
- Tuottajana voi olla myös kenttä

Sisäänrakennetut ja useat määrittäjät

- Jos injektoitava bean tai injektointikohta ei sisällä varsinaista määrittäjää, container käyttää oletusmäärittäjää `@Default`
- Jos injektointikohdan halutaan hyväksyvän mikä tahansa määrittäjätyyppi, voidaan käyttää määrittäjää `@Any`
 - Kaikilla beaneilla on aina kyseinen määrittäjä
- Jos beanilla tai injektointikohdassa on useita määrittäjiä, toisella osapuolella tulee olla kaikki samat määrittäjät

Beanin nimi

- Beaneihin viitataan varsin usein EL-kielellä web-sivuilta (JSP tai facelet)
- Tätä varten bean tai jopa metodi voidaan nimetä annotaatiolla `@javax.inject.Named`
 - Oletusarvoisesti nimi on luokan nimi ensimmäinen kirjain muutettuna pientä kirjaimen alkuisiksi, metodin tapauksessa bean propertyn nimi
 - Nimi voidaan määritellä itse annotaation `value`-attribuutilla
- Esimerkki nimeämisestä ja viittauksesta facelet-sivulla:

```
@Produces @Named @Kirjautunut Asiakas getNykyinenKayttaja() {  
    return kayttaja;  
}
```

```
Kirjautunut nimellä #{nykyinenKayttaja.email}
```

Java EE -resurssien injektointi

- Java EE -ympäristön oliot voidaan injektoida itse tehtyjen määrittäjäannotaatioiden avulla, kunhan niille tehdään erilliset tuottajakentät
- Termi Java EE -olion palauttavalle tuottajakentälle: resurssi (*resource*)
- Esimerkki:

```
@Produces @PersistenceContext @Kayttajatietokanta  
EntityManager kayttajakanta;
```

- Käyttö EJB-komponentissa:

```
@Stateless  
public class TilaushallintaEJB implements Tilaushallinta {  
    @Inject @Kayttajatietokanta EntityManager em;  
    ...  
}
```

Java EE -resurssien injektointi

- Käytännössä myös metodi osaa toimia tuottajana, vaikka spesifikaatio käsittelee vain kentän tapausta:

```
public class Kayttajatietokantatuottaja {  
    @PersistenceContext private EntityManager kayttajakanta;  
  
    @Produces @Kayttajatietokanta  
    public EntityManager haekayttajakanta() {  
        return kayttajakanta;  
    }  
}
```

Injektointien ohjelmallinen käsittely

- Jossakin tilanteissa injektoitavan olion tulee päästä käsiksi injektoinnin kohteeseen
- Tämä onnistuu `InjectionPoint`-olion avulla; esimerkki:

```
@Produces Log LuoLokittaja(InjectionPoint injektointikohta) {  
    return LoggerFactory.getLog(  
        injektointikohta.getMember().getDeclaringClass().getName());  
}
```

- Commons Logging -kirjaston lokikäsittelijän injektointi onnistuu tämän jälkeen helposti:

```
@Inject private Log loki;
```


Stereotyypit

- Stereotyypit mahdollistavat määrittelyjen (määrittelyalue sekä yksi tai useampi määrittäjä) yhdistämisen yhdeksi annotaatioksi
- Esimerkki:

```
@Stereotype
@RequestScoped
@Named
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Action { }
```
- Erityisesti JSF-käyttöä varten on tehty valmis stereotyyppi `@Model`, joka on yllä olevan esimerkin kaltainen (eli `@Named` ja `@RequestScoped`)

Stereotyytit

- Stereotyyppi voi sisältää myös @Named-, @Alternative- ja/tai interceptor-annotaation
 - Lisäksi stereotyyppi voi sisältää muita stereotyyppejä
 - @Typed-annotaatiota ei saa käyttää
- Beaniin voi liittyä useitakin stereotyyppejä
 - Voidaan liittää joko luokkaan tai tuottajametodiin/-kenttään
 - Huom.: Usean määrittelyalueen määrittäminen aiheuttaa asennuksenaikaisen virheen
- Bean voi tarvittaessa ohittaa stereotyyppin asetuksia yksinkertaisesti esimerkiksi määrittämällä oman määrittelyalueen

Määrittäjät: Attribuutit

- Määrittäjällä voi olla attribuutteja, joita käytetään injektoinnin määrittämiseen aivan kuten pelkkiä määrittäjätyyppejäkin (jos jotakin attribuuttia ei haluta käyttää tähän, merkitään se annotaatiolla `@javax.enterprise.util.Nonbinding`):

```
@Qualifier
@Retention(RUNTIME)
@Target({ METHOD, FIELD, PARAMETER, TYPE })
public @interface Maksutapa {
    Maksutapatyyppi value();
    @Nonbinding String kommentti() default "";
}
```

- (Esimerkin enum-tyyppi:)

```
public enum Maksutapatyyppi {
    LUOTTOKORTTI, PANKKIKORTTI, KATEINEN
}
```

Määrittäjät: Attribuutit

- Malli määrittäjän käytöstä:

```
@Inject @Maksutapa(Maksutapatyyppi.KATEINEN)  
private Maksukasitteliija kasitteliija;
```

Beanin tyypin rajoittaminen *

- Tyyppi voidaan rajata annotaatiolla:

```
@Typed(Kauppa.class)
public class Kirjakauppa extends BusinessSovellus
    implements Kauppa<Kirja> { }
```

- Esimerkissä tyyppejä ovat Kauppa<Kirja> ja java.lang.Object

Vaihtoehdot (*alternative*)

- Vaihtoehdot mahdollistavat injektoitavien luokkien vaihtamisen toisiksi esimerkiksi osana yksikkötestausta
- Esimerkki vaihtoehtoisesta toteutuksesta:

```
@Alternative @SessionScoped @Named  
public class KirjautuminenStub implements Kirjautuminen {  
    ...  
}
```

- Vaihtoehdot otetaan käyttöön tiedostossa `beans.xml`:

```
<alternatives>  
  <class>  
    fi.sovelto.cdi.kirjautumisdemo.KirjautuminenStub  
  </class>  
</alternatives>
```

Vaihtoehdot (*alternative*) *

- Pieni ongelma: Jos alkuperäisellä luokalla on määrittäjä, jota vaihtoehdolla ei ole ja kyseistä määrittäjää käytetään injektiossa, käyttää CDI alkuperäistä luokkaa
 - Vaikka vaihtoehto sisältäisi kaikki määrittäjät, on mahdollista että alkuperäinen luokka tekee joko tuottaja- tai kuuntelijametodin, joka aiheuttaa sen instantioinnin
- Ongelman korjaa erikoistuminen (*specialisation*):
 - Vaihtoehtoluokka periyttää alkuperäisen
 - Vaihtoehtoluokalle lisätään annotaatio `@Specializes`
 - Vaihtoehtoluokka ei saa sisältää `@Named`-annotaatiota

```
@RequestScoped @SpecTiedot @Alternative @Specializes  
public class KayttajatiedotStub extends Kayttajatiedot {  
    ...  
}
```

Stereotyypit ja vaihtoehdot

- Jos stereotyypistä tehdään vaihtoehtotiedon (@Alternative) sisältävä, saadaan kyseisen stereotyypin sisältävät vaihtoehtoiset luokat käyttöön XML-tiedostossa seuraavasti:

```
@Alternative
@Stereotype
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Stub { }
```

```
<alternatives>
  <stereotype>
    fi.sovelto.cdi.stereotyyppi.Stub
  </stereotype>
</alternatives>
```


Beanin nimi

- Beaneihin viitataan varsin usein EL-kielellä web-sivuilta (JSP tai facelet)
- Tätä varten bean tai jopa metodi voidaan nimetä annotaatiolla `@javax.inject.Named`
 - Oletusarvoisesti nimi on luokan nimi ensimmäinen kirjain muutettuna pientä kirjainta, metodin tapauksessa bean propertyn nimi
 - Nimi voidaan määritellä itse annotaation `value`-attribuutilla
- Esimerkki nimeämisestä ja viittauksesta facelet-sivulla:

```
@Produces @Named @Kirjautunut Asiakas getNykyinenKayttaja() {  
    return kayttaja;  
}
```

```
Kirjautunut nimellä #{nykyinenKayttaja.email}
```

Tuetut määrittelyalueet

- Määrittelyalueannotaatioiden paketti:
`javax.enterprise.context`
- `@RequestScoped`: Servlet, JSF, WS, EJB remote & asynkr. & timeout
- `@SessionScoped`: Servlet, JSF
- `@ApplicationScoped`: Servlet, JSF, WS, EJB remote & asynkroninen & timeout
- `@ConversationScoped`: JSF
 - Lähes sama kuin istunto, mutta liittyy selaimessa yhteen välilehteen ja alkaa ja loppuu vain ohjelmallisesti
- Oletusarvo on *dependent pseudoscope* (`@Dependent`) eli sama kuin injektoivalla komponentilla
- Huom.: Web-näkyvyysalueissa (istunto ja conversation) olevan beanin tulee olla serialisoituva

Tuetut määrittelyalueet

- @New luo aina uuden olion:

```
@Inject @New private Maksukasitteliija maksukasitteliija;
```

- Huom.: JPA-entiteeteille ei tule antaa muuta määrittelyaluetta kuin @Dependent, koska muutoin JPA:n EntityManager:n toiminta sotkeutuu

Conversation-määrittelyalue

- Esimerkki; huomaa @Conversation-injektointi:

```
@ConversationScoped @Named
public class Keskusteludemo implements Serializable {
    @Inject private Conversation keskustelu;
    private String data;

    public String aloita() {
        data = "alustettu";
        keskustelu.begin();
        return "toka";
    }

    public String lopeta() {
        keskustelu.end();
        return "valmis";
    }

    public String getData() { return data; }
}
```

Conversation-määrittelyalue

- Mikäli keskustelutiedon halutaan välittyvän muiden kuin JSF-pyyntöjen kautta, tulee pyynnöissä kuljettaa mukana parametri nimeltä `cid`:

```
<h:link outcome="kolmas" value="viimeistelee">  
  <f:param name="cid" value="#{conversation.id}"/>  
</h:link>
```

- Tämän avulla tieto keskustelusta voidaan välittää myös redirect-ohjausten mukana
- Aikakatkaistu (ohjeellinen) asetetaan metodilla `Conversation.setTimeout(millisek)`

Singleton *

- `@javax.inject.Singleton` on eräessä mielessä myös määrittelyalue (termi: *pseudo-scope*)
- Komponentit viittaavat suoraan tällä annotoituun olioon, joten serialisoinnin kanssa tulee olla huolellinen
 - Käytetään `transient`-viittausta
 - Ohjelmoidaan singleton-beaniin serialisoinnin metodit `writeReplace()` ja `readResolve()`
 - Muutetaan viittauksen tyyppiä `Instance<Luokka>`
- Neljäs, usein paras vaihtoehto on käyttää `@ApplicationScoped`-määrittelyaluetta

Sovelto

Tapahtumat

Tapahtumat

- Tapahtumat (*event*) ovat voimallinen uusi piirre, jotka mahdollistavat toimintojen ja niiden kuuntelijoiden täydellisen erottamisen (myös käännösaikaisena)
 - Tapahtumia on mahdollista suodattaa
 - Tapahtumat voidaan käsitellä heti tai vasta transaktion suorittamisen yhteydessä
- Paketti: `javax.enterprise.event`

Tapahtuman lähettäminen

- Tapahtumille tehdään yleensä tapahtumatyypin määrittävä annotaatio:

```
@Qualifier
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
public @interface Paivitetty { }
```

- Tapahtuman käynnistäminen:

```
@Stateless
public class TilaushallintaEJB implements Tilaushallinta {
    @Inject @Any Event<String> merkkijonoEiMaaritteitaEvent;
    @Inject @Paivitetty Event<String> merkkijonoPaivitettyEvent;

    public List<Tilaus> haeTilaukset() {
        merkkijonoEiMaaritteitaEvent.fire("Hei maailma!");
        merkkijonoPaivitettyEvent.fire("Hei maailma!!");
        return ...;
    }
}
```

- Tapahtumista ensin mainittu (@Any) välitetään kuuntelijoille, joilla ei ole mitään määritteitä

Tapahtuman kuuntelija

- Esimerkki tapahtuman kuuntelijasta:

```
import javax.enterprise.event.Observes;

public class Tapahtumakuuntelija {
    private final Log loki = LogFactory.getLog(getClass());

    public void tapahtumaTapahtuiKaikki(
        @Observes String merkkijono) {
        loki.info(merkkijono);
    }

    public void tapahtumaTapahtuiPaivitetty(
        @Observes @Paivitetty String merkkijono) {
        loki.info(merkkijono);
    }
}
```

- Kuuntelijametodilla voi olla muitakin injektoitavia parametreja, esimerkiksi CDI:n versiossa 1.1 tapahtumasta kertova EventMetadata-olio

Kuuntelijametodin toiminnasta

- Kuuntelu on oletusarvoisesti synkronista eli suoritus tapahtuu lähettäjäsäikeessä
- Jos jokin kuuntelijametodeista heittää poikkeuksen, muita kuuntelijametodeja ei kutsuta ja `fire()`-metodi heittää poikkeuksen
- Oletusarvoisesti CDI instantioi kuuntelijan, jos yhtäkään ei ole olemassa
 - Tämä voidaan estää annotaation `@Observes` parametrilla `notifyObserver = IF_EXISTS`; esimerkki:

```
public void tapahtumaTapahtuiPaivitettyJosOlemassa(  
    @Observes(notifyObserver = Reception.IF_EXISTS)  
    @Paivitetty String merkkijono) {  
    loki.info(merkkijono);  
}
```

- Huom.: Beanilla tulee tällöin olla jokin muu määrittelyalue kuin `@Dependent`

Lähetettävän olion määrittäjä

- Lähetettävän olion määrittäjä voidaan määrätä myös ohjelmallisesti; esimerkki:

```
merkkijonoKaikilleEvent.select(  
    new AnnotationLiteral<Paivitetty>(){}).fire("Hei maailma!!!");
```

- Jos määrittäjiä on useita, tulee tapahtuma kaikille metodeille joilla on yksikin määrittäjäistä (tai ei yhtään, mikä käsittelee kaikki tapahtumat)
- Määrittäjällä voi olla myös parametreja; tätä ei käsitellä kurssilla

Tapahtumat ja transaktiot

- Transaktionaalisten kuuntelijoiden tyypit (määritellään `@Observes`-annotaation `during`-attribuutilla):
 - `IN_PROGRESS`: Kuuntelijoita kutsutaan välittömästi (oletus)
 - `AFTER_SUCCESS`: Transaktion after completion -vaihe, mutta vain onnistuessa
 - `AFTER_FAILURE`: Transaktion after completion -vaihe, mutta virhetilanteessa
 - `AFTER_COMPLETION`
 - `BEFORE_COMPLETION`

Määrittämisalueiden (*context*) valmiit tapahtumat

- CDI 1.1:n myötä containerien tulisi automaattisesti lähettää seuraavat tapahtumat:
 - `@Initialized(RequestScoped.class)` ja `@Destroyed(RequestScoped.class)`
 - `@Initialized(ConversationScoped.class)` ja `@Destroyed(ConversationScoped.class)`
 - `@Initialized(SessionScoped.class)` ja `@Destroyed(SessionScoped.class)`
 - `@Initialized(ApplicationScoped.class)` ja `@Destroyed(ApplicationScoped.class)`

Sovelto

Interceptorit ja decoratorit

Interceptorit

- Interceptorit ovat jossain määrin servlettien filttereitä vastaava tekniikka, joka mahdollistaa metodikutsun esi- ja jälkikäsitteilyn
 - Interceptorit ovat ohjelmalliselta toiminnaltaan samoja kuin EJB 3.x -tekniikan interceptorit (tekniikat on määritelty samassa, erillisessä spesifikaatiossa)
 - Käyttöönotto ja järjestyksen hallinta poikkeaa kuitenkin EJB:n vastaavasta, sillä niitä hallitaan laajemmalla määrällä annotaatioita sekä lisäksi XML-tiedostoilla
- Interceptoreita voi määritellä:
 - Business-metodeille
 - Elinkaaren callback-metodeille
 - EJB:n timeout-metodille
- Paketti: `javax.interceptor`

Interceptorin nimeävä annotaatio

- Esimerkki interceptorin nimen määrittävästä annotaatiosta:

```
@InterceptorBinding  
@Target( { ElementType.TYPE, ElementType.METHOD })  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Lokittava { }
```

Interceptorit

- Esimerkki annotaatiota käyttävästä interceptorista:

```
@Interceptor @Lokittava
public class LokiInterceptor {
    private final Log loki = LogFactory.getLog(getClass());

    @AroundInvoke
    public Object lokita(InvocationContext kutsukonteksti)
        throws Exception {
        loki.info("lokita()");
        Object paluuarvo = kutsukonteksti.proceed();
        return paluuarvo;
    }
}
```

- Käyttö metodissa (myös koko luokka voitaisiin annotoida):

```
@Lokittava
public List<Tilaus> haeTilaukset() { ... }
```

- Jos interceptor-luokkaa ei mallin mukaisesti annotoida `@Interceptor`-annotaatiolla, tulee se yhdistää komponentteihin perinteiseen EJB 3.x:n tyyliin joko `@Interceptors`-annotaatiolla (myöhempi sivu), tiedoston `ejb-jar.xml` avulla tai globaalina `@Priority`-annotaatiolla (CDI 1.1)

Interceptor kuvaustiedostossa

- Interceptor tulee erikseen ottaa käyttöön tiedostossa `beans.xml`, muutoin sitä ei huomioida
 - Tiedoston käyttö mahdollistaa interceptoreiden helpon vaihtamisen eri tarkoituksissa (build-prosessissa voidaan helposti vaihtaa tiedostoa tilanteen mukaisesti)
 - Interceptoreiden järjestys määräytyy täällä tehtyjen esittelyiden mukaisesti
- Esimerkki:

```
<interceptors>  
  <class>fi.sovelto.cdi.interceptor.LokiInterceptor</class>  
</interceptors>
```

Suorituksen tiedon käsittely

- Kuuntelija pääsee käsiksi metodikutsun dataan kontekstioliion metodilla `Object[] getParameters()`
 - Tämä on muokattavissa
- Kokonaan uusi sisältö voidaan antaa metodilla `setParameters(Object[])`
- Esimerkki, jossa sisältöä muokataan suoraan:

```
Object[] tiedot = kutsukonteksti.getParameters();
for (Object tieto: tiedot) {
    loki.info("tarkistusmetodi() tyyppi: " +
        tieto.getClass().getName());
    loki.info("tarkistusmetodi() arvo: " + tieto);
    if (((Long) tieto).longValue() == 11) {
        tiedot[i] = 123;
    }
}
```

- Huomaa, että myöhemmin käsiteltävä decorator-tekniikka esittelee helpomman syntaksin parametrien käsittelylle

Interceptorin liittämisen EJB 3.x -syntaksi *

- Kuuntelijat liitetään EJB 3.x -komponenttiin @Interceptors-annotaatiolla
 - Joko koko luokalle tai metodikohtaisesti
- Esimerkki metodiin liittämisestä:

```
@Interceptors({ LokiInterceptor.class })
public boolean päivitaHenkilo(Henkilo henkilo) {
    ...
}
```

Muita määrittämiä (EJB) *

- Asennustiedoston `ejb-jar.xml` kautta voidaan määrittää kaikkiin sovelluksen EJB-komponentteihin kohdistuvia interceptoreita
 - Termi: Oletusarvoinen (*default*) interceptor
- Liittämisen hallinnan annotaatiot:
 - `@ExcludeClassInterceptors`: Voidaan liittää metodiin, jolloin luokkaan kohdistuvia interceptoreita ei kohdisteta kyseiseen metodiin
 - `@ExcludeDefaultInterceptors`: Oletusarvoiset interceptorit voidaan poistaa joko luokalta tai metodilta
- Suoritusjärjestys:
 - Interceptorit suoritetaan siinä järjestyksessä kuin ne esitellään
 - Oletusarvoiset suoritetaan ennen luokka- ja metodikohtaisia
 - Järjestyksen voi tarvittaessa määrittää kuvaustiedoston kautta

Interceptorit: Elinkaarimetodin käsittely

- Esimerkki elinkaarimetodia käsittelevästä interceptorista:

```
@Interceptor @Lokittava
public class ElinkaariInterceptor {
    private final Log loki = LogFactory.getLog(getClass());

    @AroundInvoke
    public void lokita(InvocationContext kutsukonteksti) {
        loki.info("lokita()");
        try {
            kutsukonteksti.proceed();
        } catch (Exception ex) {
            loki.error("VIRHE", ex);
        }
    }
}
```

- Huomaa, että tämäkin interceptor-luokka tulee esitellä tiedostossa `beans.xml`

Interceptorin kohdistamisesta metodiin

- Interceptorille on mahdollista antaa useita määrittäjäannotaatioita
- Tällöin käytävällä tyypillä tulee esiintyä näistä kaikki, jotta interceptor kohdistetaan metodiin
 - Osan voi esitellä luokalle ja osan metodeille

Interceptorin nimeävä annotaatio: Attribuutit *

- Interceptorin nimen määräävälle annotaatiolle voidaan määrittää myös attribuutteja:

```
@InterceptorBinding
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Lokittava {
    Lokitustyyppi value() default Lokitustyyppi.TIEDOTUS;
}
```

- (Esimerkin enum-tyyppi:)

```
public enum Lokitustyyppi {
    VIRHE, VAROITUS, TIEDOTUS, DEBUG
}
```

Interceptorit: Attribuutit

- Esimerkki interceptorista:

```
@Interceptor @Lokittava(Lokitustyyppi.VAROITUS)
public class VaroitusLokiInterceptor {
    private final Log loki = LogFactory.getLog(getClass());

    @AroundInvoke(Lokitustyyppi.VAROITUS)
    public Object lokita(InvocationContext kutsukonteksti)
        throws Exception {
        loki.warn("lokita()");
        Object paluuarvo = kutsukonteksti.proceed();
        return paluuarvo;
    }
}
```

- Käyttö metodissa:

```
@Lokittava(Lokitustyyppi.VAROITUS)
public List<Tilaus> haeTilaukset() { ... }
```

Decoratorit

- Interceptorin ongelmana on, että pääsy metodikutsun dataan on kömpelöä ja virheherkkää (metodin muuttaminen rikkoo interceptorin, mutta kääntäjä ei huomaa asiaa)
- Decorator on uusi, toiminnaltaan interceptorin kaltainen tekniikka, jossa päästään suoraan käsiksi haluttuun rajapintaan
- Luokalla voi olla sekä decoratoreita että interceptoreita; tällöin interceptorit suoritetaan ennen decoratoreita
- Toimintamalli: Decorator toteuttaa saman rajapinnan kuin sen koristama komponentti
 - Rajapinta on siis vaatimus, decorator ei toimi luokkatyyppien kanssa
 - Lisäksi: Decorator ei toimi EJB-komponenttien remote-rajapinnan kanssa
- Jos koristeltava komponentti on EJB-komponentti, tulee myös decoratorin ohjelmoinnissa noudattaa EJB:n rajoituksia (esimerkiksi synkronointia ei saa tehdä)

Decoratorit

- Komponentin rajapinta (esimerkissä EJB):

```
@Local public interface Tilaushallinta {  
    public List<Tilaus> haeTilaukset();  
}
```

- Decorator:

```
@javax.decorator.Decorator  
public abstract class SuuriTilaushallintaEJBDecorator  
    implements Tilaushallinta {  
    private final Log loki = LogFactory.getLog(getClass());  
    @Inject @Delegate @Any Tilaushallinta tilaushallinta;  
  
    @Override  
    public List<Tilaus> haeTilaukset() {  
        List<Tilaus> tilaukset = tilaushallinta.haeTilaukset();  
        if (tilaukset.size() > 25) {  
            loki.warn("Suuri tilausten lkm: " + tilaukset.size());  
        }  
        return tilaukset;  
    }  
}
```

Decoratorit

- Myös decorator tulee esitellä tiedostossa `beans.xml`, jotta se otetaan käyttöön:

```
<decorators>
  <class>
    fi.sovelto.cdi.decorator.SuuriTilaushallintaEJBDecorator
  </class>
</decorators>
```

CDI: Best practices

- Injektointi
 - Käytä pääasiassa tavallista, muuttumatonta injektointia
 - Käytä tuottajia vain tilanteessa, jossa ohjelmallista käsittelyä todella tarvitaan
- Käytä määrittäjäannotaatioita enum-tyyppien kanssa annotaatioiden määrän vähentämiseksi
 - Materiaalin sivu "Määrittäjät: Attribuutit"
- Interceptorit ja decoratorit: Käyttötilanteet:
 - Interceptorit: Järjestelmätason toiminnot, jotka eivät liity business-logiikkaan
 - Decoratorit: Lähellä business-logiikkaa
- Nimeäminen (@Named): Nimeä vain komponentit (tai piirteet), joihin tulee pystyä viittaamaan JSF:n esitystasolta EL-kielellä

CDI: Best practices

- Mieti rajapintojen käyttöä EJB-komponenteilla, jos aitoa tarvetta toteutuksen vaihtamiselle ei ole