Grundlæggende object/relational mapping (ORM)

Copyright © Lund & Bendsen A/S

Hvad handler denne sektion om?

- I denne sektion kigger vi nærmere på, hvordan man laver selve entity-klassen:
  - Krav til entity-klasser
  - Introduktion til primary keys
  - Mapping af basic types (fx Date, int, String)
- I senere sektioner vil vi se på hvordan man mapper associationer og arv, samt hvordan man laver sammensatte nøgler m.m.

Copyright © Lund & Bendsen A/S

Grundlæggende O/R-mapping

- Krav til entity-klasser
- Introduktion til primary keys
- Mapping af basic types

Copyright © Lund & Bendsen A/S

Hvad er en entity?

An entity is a lightweight persistent domain object.

- Lightweight: POJO
- Persistent: Gemmes i database
- Domain object: Afspejler et relevant koncept i vores forretningsdomæne

Copyright © Lund & Bendsen A/S

Krav til Entity bean klassen

- En entity er en klasse som er annoteret med @*Entity* (eller tilsvarende i en orm.xml-fil*)

- De vigtigste regler for entity-klasser er:
  - At klassen skal have en public eller protected constructor, som ikke tager nogle argumenter (man må gerne overloade med flere constructors)
  - At klassen skal have en primær nøgle
  - Hvis entity-instanser skal kunne sendes over netværket skal klassen implementere Serializable-interface'et

** Alle de metadata, som man kan angive vha. @-annotationer, kan evt. angives i XML-konfiguration i stedet. På dette kursus bruger vi kun annotationer.*

Field access vs. property access 1/3

- Der er 2 forskellige måder at angive en entity-klasses persistente felter på
- JPA kalder det for *access mode*
- De 2 access modes udgør:

  - **Field access** – fx:
    private String firstName;

  - **Property access** – fx:
    public String getFirstName( ){…}
    public void setFirstName(String name) {…}

Copyright © Lund & Bendsen A/S

Field access vs. property access 2/3

Hvordan håndterer persistence provideren de 2 access-typer?

- **Field access**
  - Persistence provideren aflæser felterne direkte
  - Persistence provideren sætter felterne direkte
- **Property access**
  - Persistence provideren aflæser vha. getXxx( )
  - Persistence provideren sætter vha. setXxx(…)

Copyright © Lund & Bendsen A/S

Field access vs. property access 3/3

Best practice er at bruge *field access* fordi:

- Property access gør, at man skal passe på med kun at læse/ændre felterne via get- og set-metoderne (og *ikke* fra andre metoder i klassen)
- Det er ikke usædvanligt, at man laver validering af parameteren i set-metoder.
  - Hvis man bruger property access vil valideringen også blive kørt, når JPA indlæser objektet fra basen (det tager tid og kan fejle hvis data i basen ikke er valide i forhold til set-metodens regler)
  - Hvis man bruger field access kan man stadig lave validering i set-metoden, men valideringen bliver *ikke* aktiveret, når JPA indlæser objektet fra databasen
- Property access er forbundet med en del ekstra regler i diverse forskellige situationer
- På dette kursus bruger vi konsekvent field access!

Copyright © Lund & Bendsen A/S

Hvordan angiver man access mode?

Hvordan ”fortæller” vi persistence provideren, hvilken access mode vi vil bruge?

- Hvis vi vil bruge **field access**
  - Skal vi sætte @*Id*-annotationen på vores id-felt
  - Fx @*Id Integer id;*
  - Så bruger JPA field-baseret access i hele klassen
- Hvis vi vil bruge **property access**
  - Skal vi sætte @*Id*-annotationen på id-get-metoden
  - Fx: @*Id public Integer getId( ){…}*
  - Så bruger JPA property-baseret access i hele klassen

Copyright © Lund & Bendsen A/S

Regler for brugen af field access

- Vælg field access ved at sætte @*Id*-annotationen på primary key-feltet
- Felterne må *kun* ændres af metoder, der er erklæret i samme klasse, som feltet (hvilket under alle omstændigheder er best practice)
- Felterne må *ikke* være public

Copyright © Lund & Bendsen A/S

Regler for brugen af property access

- Vælg property access ved at sætte @*Id*-annotationen på get-metoden til primary key-feltet
- Persistente properties skal følge JavaBean-navngivningsstandarden – dvs:
- Vil man lave en property firstName af typen String, så
  - Skal skrivemetoden hedde: void setFirstName(String n)
  - Skal læsemetoden hedde: String getFirstName( )
  - Hvis typen er boolean, så må læsemetoden starte med *is* i stedet for *get* – fx *boolean isMale( )*
- Property-metoderne skal være public eller protected

Copyright © Lund & Bendsen A/S

Kan man blande access typerne?

- Nogle persistence providers tillader, at man blander field access og property access
- Men vil man være sikker på, at ens entity-klasser er kompatible med alle persistence providers, så:
  - Skal alle felter i en given klasse bruge enten field access eller property access
  - Skal alle klasser i samme arvehierarki bruge enten field access eller property access

Copyright © Lund & Bendsen A/S

Eksempler

```
// Da klassen er Serializable kan den sendes over netværket
@Entity
public class Person implements Serializable {
        // Da vi ikke har lavet en constructor indsætter compileren en no-args public constructor for os
        // hvilket vil sige, at vi lever op til kravet om, at der skal være en no-arg constructor

        // Entity-klassen vil bruge field access, da vi har brugt field access på klassens primary key
        @Id private Integer personId;
        •••
}
```

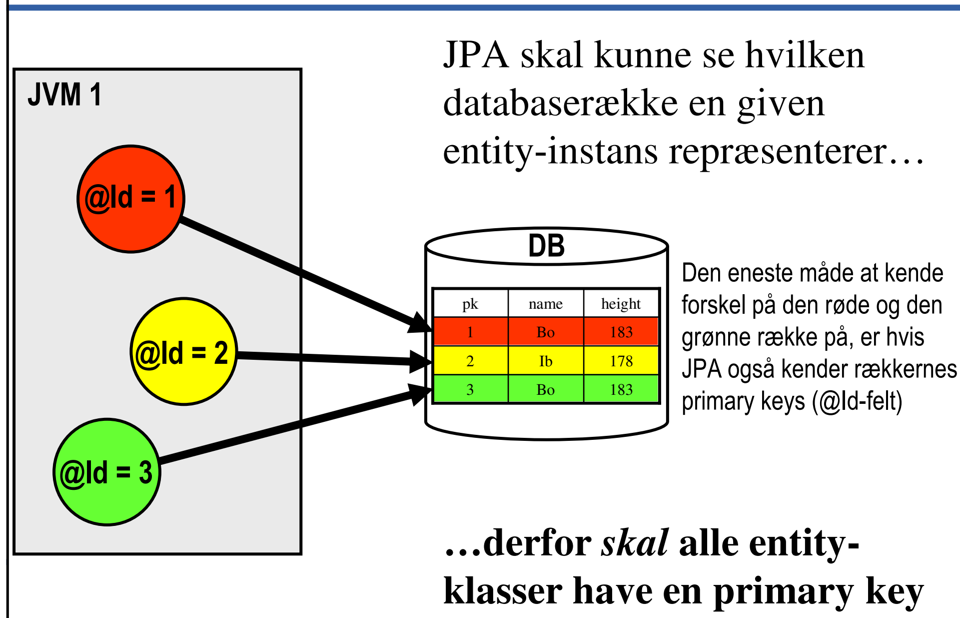
```
@Entity
public class Person {
        // Da vi overloader constructoren skal vi selv sørge for at indsætte en no-arg constructor
        protected Person( ) {...}
        public Person(String firstName) {…}
        // Entity-klassen vil bruge property access, da vi sætter @Id på primary key get-metoden
        @Id public Integer getPersonId ( ) {…}
        •••
}
```

Grundlæggende O/R-mapping

- Krav til entity-klasser
- Introduktion til primary keys
- Mapping af basic types

Copyright © Lund & Bendsen A/S

Alle entity-klasser skal have en @Id



Entity-objekter kan altså sammenlignes på 3 måder!

Gælder i alle Java-programmer

- ***Identical objects***: Er når to variable x og y begge refererer til objektet på en bestemt memory-adresse. *Testes vha. x = = y.*
- ***Equal objects***: Er når x og y refererer til objekter, som er ens. Det er applikationsspecifikt, hvad der menes med ens, idet det bestemmes af klassens equals-metode. *Testes vha. x.equals(y).*

Specielt for JPA-programmer

- ***Entity identity***: Er når x og y refererer til entity-objekter, som er af ens type (fx Person) og som har samme primary key-værdi (fx 342) – dvs 2 Java-objekter, som begge repræsenterer den samme række i databasen. *Testes vha. xkey.equals(ykey)*

Copyright © Lund & Bendsen A/S

Bemærkninger om primary key

- JPA stiller følgende krav mht. primary key
  - Primary key må ikke være null, når først objektet er persisteret (evt. gerne når det er helt nyt)
  - To instanser af samme type må aldrig have samme id (i så fald betragtes de som to kopier af samme entitet)
  - Værdien (værdierne) af nøglen, må aldrig ændre sig.
- Det frarådes at bruge nøgler, der har domæne-mæssig betydning (= best practice i databaser)
- Nøgler kan evt. være sammensat (det ser vi på i en anden slide-sektion)

Copyright © Lund & Bendsen A/S

Primary keys der mapper til ét felt

- Følgende datatyper er velegnede som PK:
  - Heltalstyper – fx int, Integer, long, Long eller java.math.BigInteger
  - Tekststrenge – dvs. java.lang.String
  - Tids-typer – dvs. java.sql.Date eller java.util.Date
- Skal annoteres med @*Id* - fx
  @Id private Integer personId;
- NB: Integer er som regel et godt valg
  - Fylder ikke ret meget i databasen
  - Effektiv at indexere og søge på i databasen
  - Kan autogenereres af alle JPA-implentationer
  - Kan være null indtil en entity har fået tildelt sin pk

Automatisk generering af PK-værdi

- Alle JPA-implementationer understøtter automatisk tildeling af en PK-værdi til entiteter (uanset hvilken base der er bag)
- Man kan aktivere automatisk tildeling af primary key vha. @GeneratedValue – fx:
  @GeneratedValue
  @Id private Integer personId;

Copyright © Lund & Bendsen A/S

Best practice

- Når først en entity-instans er blevet persisteret til databasen, er det ikke tilladt at ændre dens primary key.
- Derfor er det best practice:
  - Ikke at lave en set-metode på @Id-feltet
  - At sætte @Id-feltet i en constructor i de tilfælde, hvor man ikke lader JPA lave automatisk generering af pk-værdier

Copyright © Lund & Bendsen A/S

Der er mere om primary keys…

- JPA understøtter flere forskellige strategier for hvordan primary keys auto-genereres.
  Det vender vi tilbage til senere.
- Det er muligt at lave sammensatte nøgler i JPA – dvs. nøgler der mapper til flere kolonner i databasen.
  Det vender vi også tilbage til senere.

Copyright © Lund & Bendsen A/S

Grundlæggende O/R-mapping

- Krav til entity-klasser
- Introduktion til primary keys
- Mapping af basic types

Copyright © Lund & Bendsen A/S

Hvad er JPA basic types?

- En JPA basic type er en datatype, der som regel mappes til 1 kolonne i databasen, og som ikke er en reference til andre entities (en foreign key)
- Nedenstående er eksempler på basic types:

```
@Entity public class Person {
    @Id private Long pk;            // Long er en basic type
    private String firstName;       // String er en basic type
    private int height;             // int er en basic type
}
```


Copyright © Lund & Bendsen A/S

JPA-understøttede basic types

- **Primitive Java-typer:** byte, short, int, long, boolean, char, float, double
- **Javas wrapper-klasser:** Byte, Short, Integer, Long, Boolean, Character, Float, Double
- **Byte og character arrays:** byte[], Byte[], char[], Character[]
- **Store taltyper:** java.math.BigInteger, java.math.BigDecimal
- **Tekststrenge:** java.lang.String
- **Javas temporale typer:** java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp
- **Enums:** Både dem der findes i Java-api'et og dem man selv har lavet.
- **Serializable objekter:** Bemærk dog, at disse typisk gemmes som serialiserede objekter – dvs. som en stribe bytes. M.a.o. kan man ikke søge på dem, og de giver ikke mening udenfor Java-programmer.

Copyright © Lund & Bendsen A/S

Best practice

- Skal man bruge primitive typer eller wrapper-typer til sine felter?
- Man bør sørge for, at hvis null er tilladt i databasen, så skal det også være tilladt i Java
- Det vil sige:
  - Har databasekolonnen en not-null constraint, så bør det tilsvarende Java-felt være en af Javas primitive typer (der ikke kan være null)
  - Må der indsættes null-værdier i kolonnen i databasen, så bør det tilsvarende Java-felt være en wrappertype (der også godt kan have værdien null)

Copyright © Lund & Bendsen A/S

Hvordan mapper man basic felter?

- Man behøver ikke gøre noget!
- JPA antager at alle basic type felter i en entity-klasse er persistente felter, der skal gemmes i databasen
- Man *må* gerne annotere sine felter med @*Basic*
  - Men det er der sjældent grund til
  - Med mindre… man vil angive ekstra informationer om feltet (avancerede ting, som vi vender tilbage til senere)
- Nogle af basic typerne *kræver* dog yderligere annotationer (det kigger vi på om lidt)

Copyright © Lund & Bendsen A/S

Hvad hvis man ikke vil have gemt et felt?

- Som udgangspunkt ser JPA alle felter i vores entity-klasser som persistente
- Hvis et felt ikke skal gemmes til basen kan man erklære det som transient:
  - Vha. keywordet transient – fx:
    transient boolean myMemoryOnlyField;
  - Eller vha. annotationen @Transient – fx:
    @Transient boolean myMemoryOnlyField;

Copyright © Lund & Bendsen A/S

Enumerations

- Som default gemmes en enum-værdi som et tal (dets ordinal fx *SUMMER.ordinal()*)
  enum Season {SPRING, SUMMER, FALL, WINTER}
  - Enum-værdien SPRING gemmes som talværdien 0
  - Enum-værdien SUMMER gemmes som talværdien 1
  - Enum-værdien FALL gemmes som talværdien 2
  - Enum-værdien WINTER gemmes som talværdien 3
- Kan gemmes som String vha.
  @Enumerated(EnumType.STRING)

Se eksempel næste slide

Copyright © Lund & Bendsen A/S

Eksempler med enums

```
public enum Season {SPRING, SUMMER, FALL, WINTER}
```

```
@Entity
public class Person {
        @Id private Integer pk;
        private Season favoriteSeason; // gemmes som tallet 0, 1, 2 eller 3
        •••
}
```

```
@Entity
public class Person {
        @Id private Integer pk;
        @Enumerated(EnumType.STRING)
        private Season favoriteSeason; // gemmes som teksten SPRING, SUMMER osv.
        •••
}
```


Copyright © Lund & Bendsen A/S

Mapping af java.util-tidstyperne

- Klasserne i java.sql behøver ikke ekstra mapping.
- Tidsklasserne i java.util (Date og Calendar) skal dog mappes til en dato, et klokkeslæt eller et tidspunkt
- Gøres vha. annotationen @*Temporal:*
  - *@Temporal(TemporalType.DATE)*
    svarer til java.sql.Date
  - *@Temporal(TemporalType.TIME)*
    svarer til java.sql.Time
  - *@Temporal(TemporalType.TIMESTAMP)*
    svarer til java.sql.TimeStamp

Se eksempel næste slide

Copyright © Lund & Bendsen A/S

Eksemplel med Date

```
@Entity
public class Person {
    @Id private Integer pk;
    @Temporal(TemporalType.DATE)
    private Date birthYear;
    •••
}
```


Copyright © Lund & Bendsen A/S

Mapping af store felter (CLOB & BLOB)

- Vil man mappe meget store felter til en CLOB eller en BLOB gøres det vha. @*Lob*
- @Lob kan bruges på basic felter af typerne:
  - byte[], Byte[] og Serializable (= BLOB)
  - char[], Character[] og String (= CLOB)
- Det er som regel en god idé at annotere @*Lob*-felter med:
  - *@Basic(fetch=FetchType.LAZY)*
  - *@Column(length = <field-lenght-in-db>)*

Se eksempel næste slide

Copyright © Lund & Bendsen A/S

Eksempel på brugen af stort felt

Læs ikke PDF-dokumentet ind før feltet tilgås af applikationen

```
@Entity
public class Document {
    @Id private Integer id;
    @Lob @Basic(fetch=FetchType.LAZY)
    @Column(length = 8 * 1024 * 1024)
    private byte[] docAsPdf;
    // ...
}
```

Sæt feltstørrelsen til noget andet end default

Copyright © Lund & Bendsen A/S

Øvelse

- Brug: ***JPA - Lab04 - Use basic types***
- Se evt.: ***JPA - Solution04 - Use basic types***
- Udvid Person-klassen med:
  - height (et kommatal – fx 183,34)
  - cpr (et String-felt)
  - sex (enum Sex{MALE, FEMALE})
- Prøv nogle af følgende ting, hvis du har tid:
  - birthDate vha. Calendar (en dato – fx 12/3-2004)
  - at lave manuel tildeling af pk (fx til cpr)
  - at lave et @Lob-felt kaldet photo (brug photo.jpg)
  - at lave en Serializable Address-klasse og lave et felt af denne type i Person (check hvordan Address-objekter bliver gemt i databasen)

Copyright © Lund & Bendsen A/S