Agenda

IDs, Lebenszyklus

DB-Identität

IDs, Composite IDs

ID-Generatoren

Lebenszyklus

Callbacks

Das selbe, das gleiche?

- **Object identity**
  - Prüfung der Referenzgleichheit
  - Prüfung mittels ==
- **Object equality**
  - Prüfung der Inhaltsgleichheit
  - Prüfung durch Methode equals(Object o)
- **Database identity**
  - Prüfung on Primary Key identisch ist

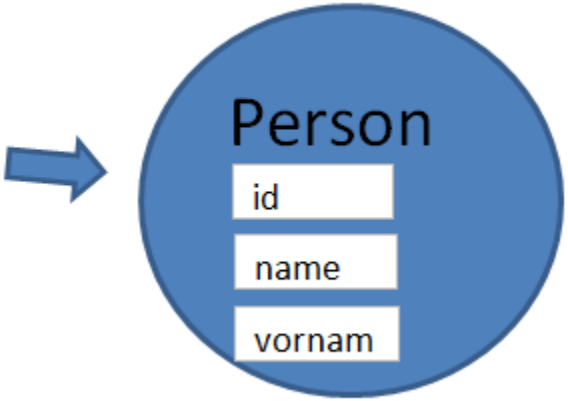
Identität eines Objektes herausbekommen

- Primary Key (bei ID-Generator) wird von System gesetzt
- ID nach persist(..) abfragbar
  - ggf. erst nach Flush/Commit

```
public int create(String position) {
   Spieler s = new Spieler();
   s.setPosition(position);
   em.persist(s);
   return s.getId();
}
```

Identität eines Objektes (Hibernate)

- Abfragen der ID über
  - p.getId()
  - session.getIdentifier(Object o)



equals() und hashCode()

- Kommen **Detached Objects zum Einsatz**
  - Entitäten, die in unterschiedlichen Sessions leben
- Muss equals() und hashCode() überschrieben werden

- 1: Implementierungs-Idee:
  - DB-ID vergleichen
  - möglicherweise Probleme bei Hash-basierten Containern
- wenn equals() überschrieben wird, muß auch hashCode() überschrieben werden

equals() und hashCode() mit der ID


```
public boolean equals(Object o) {
    if  (o == this)   {  return true; }
    if   (!(o instanceof Person))   {  return false; }
    Person p =  (Person) o;
    return   (id = null ? o.id = null  :  id.equals(o.id));
}
public int hashCode() {
    int r = 17;
    r = 37 * r +  (id != null ? id.hashCode()   : 0);
    return r;
}
```

Probleme bei hashCode()

- Naive Lösung führt bei Verwendung von Sets oder Maps in zu-n-Beziehungen und generierten Ids zu Problemen
  - durch save() generierte ID ändert den Hashcode der Entität
- fachlichen Schlüssel zu verwenden ist hierbei besser, z. B.:
  - firstName
  - lastName
  - email
  - birthday
- => siehe auch natural-id in Hibernate

equals() und hashCode() mit fachlichem Schlüssel


```
public boolean equals(Object o) {
      . . .
      Person p =  (Person) o;
      boolean rv = true;
      rv = rv &&  (name == null ? o.name == null :
                                  name.equals(o.name));
      // weitere Attribute  . . .
      return rv;
}

public int hashCode(){
   int r = 17;
   r = 37 * r +  (name != null ? name.hashCode()   :  0) ;
   // weitere Attribute ...
   return r;
}
```

Natural-Id (Hibernate)

- künstliche Schlüssel als DB-ID bevorzugt
  - String oder Long-Zahl, z. B. Kundennummer
- Property oder Kombination mehrerer Properties
  - müssen unique und not-null sein
  - sollten unveränderbar (immutable) sein
  - zum Vergleichen in equals () und hashCode ()
- Hibernate generiert automatisch
  - Unique-Key
  - Nullable Constraints

```
<natural-id>
      <property . . . />
      <many-to-one . . . />
</natural-id>
```

Agenda

IDs, Lebenszyklus

DB-Identität

IDs, Composite IDs

ID-Generatoren

Lebenszyklus

Callbacks

Konfiguration von Identifiern

- Ort von @Id bestimmt, ob System einen Attribut- oder Property-Access durchführt
- **Attribut:**

```
@Entity
public class Spieler implements Serializable  {
      @Id
      private int spielerID;
}
```

- **getter:**

```
@Entity
public class Spieler implements Serializable {
       @Id
       public int getSpielerID{
              return spielerID;
       }
```

Konfiguration von Identifiern (Hibernate)


```
public class Person {
   private Long id;
   public Long getId() {
        return this.id;
   }
   private void setId(Long id) {
        this.id = id;
   }
}
```

```
<class name="Person" table="PERS0N">
   <id name="person_id"  type="int" column="per_id"
   unsaved-value="null|any|none|undefined|id_value">
        <generator ... />
   </id>
</class>
```

Komponenten als Composite IDs

- **zusammengesetzter Primary Key**
- Benutzter definierte PK Klasse muß
  - Serializable implementieren
  - equals/hashCode überschreiben
  - kein ID-Generator möglich => Anwendung muß IDs erzeugen
- Foreign-Key-Beziehungen benötigen alle Spalten des Composite-Keys
- **zwei Varianten**
  - @Embeddedld
  - @ldClass

Eigene PK-Klassen mit @Embeddedld

- Persistente Klasse deklariert ld als **@EmbeddedId**
- PK-Klasse enthält **@Embeddable**

```
@Entity
public class Person {
   @EmbeddedId private PersonPK pk;   ...
}

@Embeddable
public class PersonPK ...
```

Eigene PK-Klassen mit @IdClass

- Persistente Klasse gibt durch **@IdClass** ihre PK-Klasse an
- Jedes der (passend benannten) Attribute muss als **@Id** gekennzeichnet sein

```
@Entity
@IdClass(PersonPK.class)
public class Person {
    @Id
    private String name;
    @Id
    private String vorname;
```

Eigene PK-Klassen mit @IdClass(2)

- Die PK-Klasse muss
  - entsprechende Properties enthalten
  - serialisierbar sein
  - die Methoden equals und hashCode implementieren

```
@Embeddable
public class PersonPK implements Serializable {
    @Override
    public boolean equals(Object   arg)   {   ... }
    @Override
    public int hashCode()   { ...  i
    . . .
```

Composite Id Hibernate Mapping


```
<class name="Person">
   <composite-id name="id" class="PersonId">
       <key-property name="firstName"/>
       <key-property name="lastName"/>
       <key-property name="birthday"/>
   </composite-id>
   . . .
</class>


public class PersonId {
   private String firstName;
   private String lastName;
   private Date birthday;
   . . .
}
```

Agenda

IDs, Lebenszyklus

DB-Identität

IDs, Composite IDs

ID-Generatoren

Lebenszyklus

Callbacks

ID-Generatoren

- Wer ist für Generierung von IDs zuständig?
  - Anwendung
  - Datenbank
  - Eigene Komponente

ID Generatoren

- In JPA auswählbar:
  - AUTO - Abhängig von der Datenbank (Native in Hibernate)
  - TABLE - ID wird aus einer Tabelle gelesen
  - IDENTITY - ID wird von Datenbank vergeben
  - SEQUENCE - Sequenzen (z. B. Oracle)

ID Generatoren


```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY )
private Long id;

@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,
  generator="course_seq")
@SequenceGenerator(
    name="course_seq",
    sequenceName="course_sequence",
    allocationSize=20
))
public Integer getId()  {   ... }
```

Konfiguration von Generatoren in Hibern


```
<id name="person_id" type="int" column="per_id">
   <generator class="increment"/>
</id>
<id name="person_id" type="int" column="per_id">
   <generator class="hilo">
        <param name="table">hi_value</param>
        <param name="column">next_value</param>
        <param name="max_lo">100</param>
   </generator>
</id>
```

Verschiedene Generatoren

Strategie	Parameter	Beschreibung
native	–	Wählt entsprechend der darunterliegenden Datenbank eine Strategie (identity, sequence oder hilo). Welche Strategie für eine Datenbank gewählt wird, ist in den Dialekten definiert.
uuid	separator	Gibt einen String mit Länge von 32 ausschliesslich hexadezimalen Zeichen zurück. Optional kann ein Separator zwischen den UUID-Komponenten mit generiert werden.
hilo	table column max lo	Dieser Generator nutzt einen Hi/Lo-Algorithmus, um numerische IDs (Long, long, int, ...) zu erzeugen. Optional können die Spaltennamen für die Hi/Lo- Tabelle angegeben werden (Defaultwerte: hibernate_unique_key und next_hi). Mit max_lo kann die Anzahl der IDs bestimmt werden, die erzeugt werden, bevor wieder ein Datenbankzugriff erfolgt, um den Max-Value zu erhöhen. Der Generator kann nicht mit einer eigenen Connection oder eine über JTA (Java Transaction API) erhaltene Connection verwendet werden, da Hibernate in der Lage sein muss, den "hi"-Value in einer neuen Transaktion zu holen.
seqhilo	sequence max_lo parameters	Dieser Generator kombiniert einen Hi/Lo-Algorithmus mit einer darunterliegenden Sequence, die die Hi-Values generiert. Die Datenbank muss Sequences unterstüzen, wie zum Beispiel Oracle und PostgreSQL. Der Parameter parameters wird dem Create Sequence Statement hinzugefügt, beispielsweise "INCREMENT BY 1 START WITH 1 MAXVALUE 100 NOCACHE". Mit sequence kann ein Name für die Sequence vergeben werden, Default ist "hibernate_sequence".

Verschiedene Generatoren (2)

Strategie	Parameter	Beschreibung
identity	–	Dieser Generator unterstützt Identity Columns bzw. autoincrement, die es beispielsweise in MySQL, HSQLDB, DB2 und MS SQL Server gibt.
select	key (required)	Die ID wird über ein Select mit einem eindeutigen key erhalten. Der Primärschlüssel wird von der Datenbank, zum Beispiel mit einem Trigger, vergeben.
sequence	sequence parameters	Dieser Generator unterstützt Sequences, die es beispielsweise in PostgreSQL, Oracle und Firebird gibt. Die ID kann vom Type Long, long, int etc. sein. Der Parameter parameters wird dem Create Sequence Statement hinzugefügt, beispielsweise "INCREMENT BY 1 START WITH 1 MAXVALUE 100 NOCACHE". Mit sequence kann ein Name für die Sequence vergeben werden, Default ist "hibernate_sequence".
assigned	–	Bei assigned muss die ID selbst gesetzt werden, vor dem Aufruf von save(). Nützlich bei natürlichen Keys. Das ist zugleich auch die Default-Strategie, falls die Annotation @GeneratedValue nicht angegeben wurde.
increment	–	Der maximale Primärschlüssel-Wert einer Tabelle wird beim Start der Anwendung gelesen und dann jedes Mal erhöht, wenn ein Insert erfolgt. Sollte nicht in einer Cluster-Umgebung benutzt werden.
foreign	property (required)	Benutzt die ID eines assoziierten Objekts, welches beim Parameter property angegeben werden muss. Wird gewöhnlich in Verbindung mit 1-zu-1-Beziehungen verwendet.
guid	–	Nutzt die von MS SQL Server oder MySQL generierte GUID.

Agenda

IDs, Lebenszyklus

DB-Identität

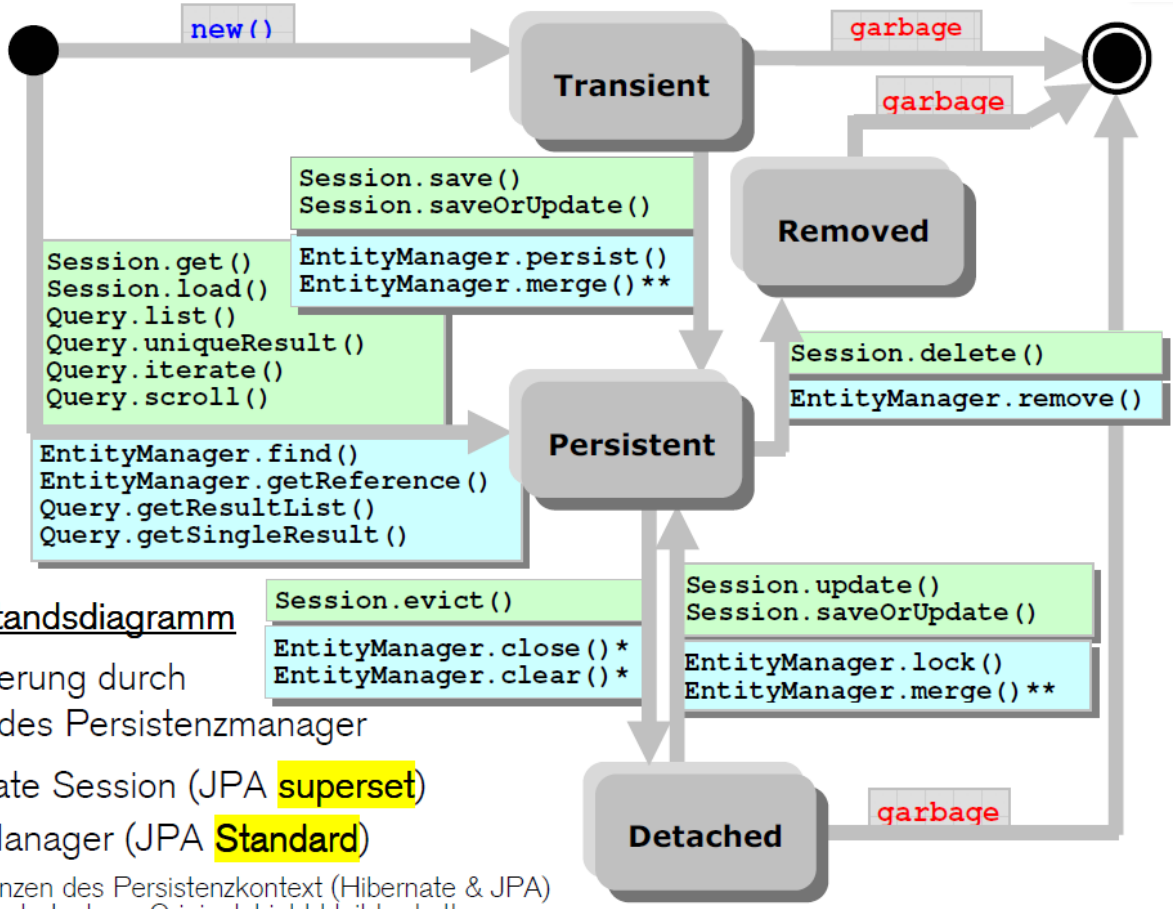
IDs, Composite IDs

ID-Generatoren

Lebenszyklus

Callbacks

Objektszustände JPA



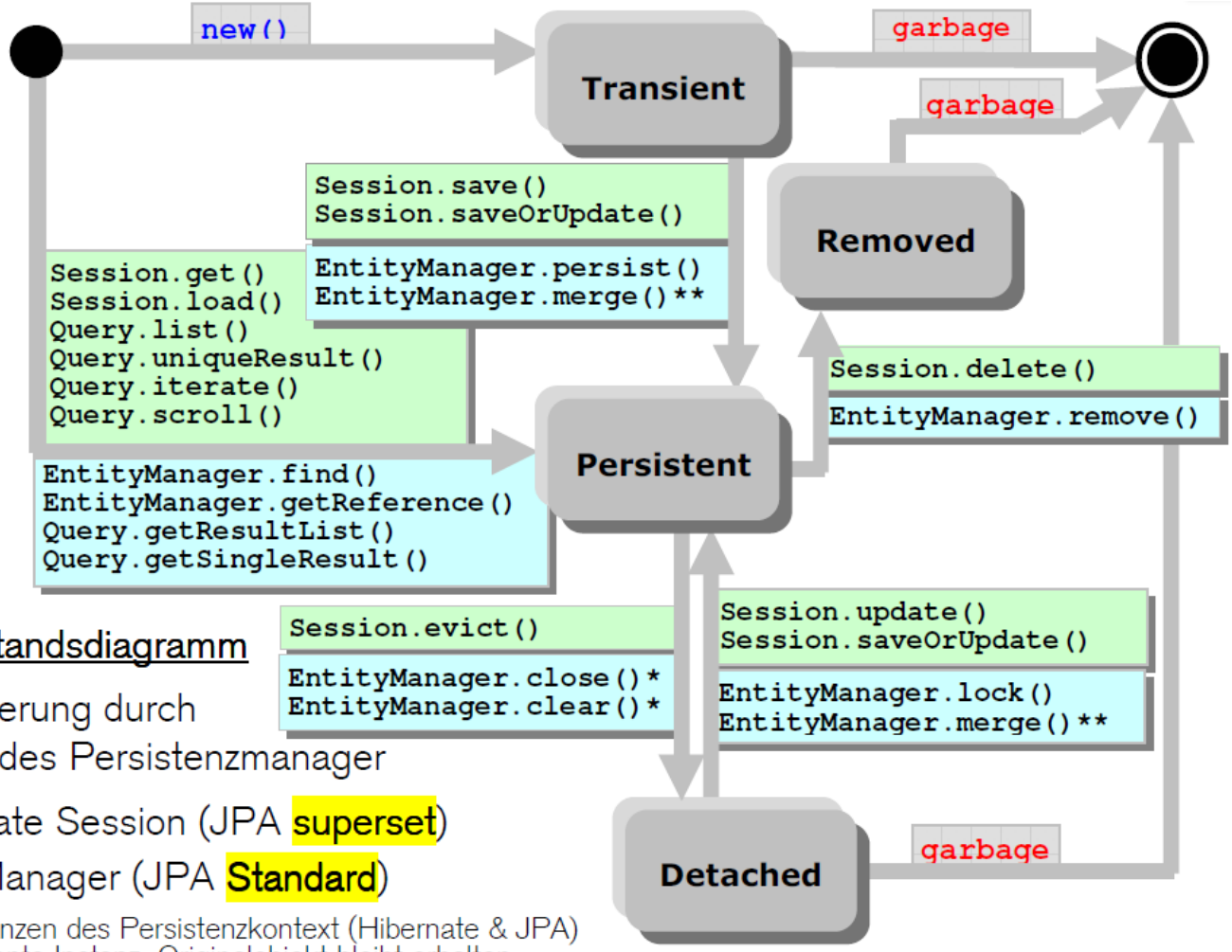
Objekt - Zustandsdiagramm

Zustandsänderung durch Operationen des Persistenzmanager

- ☐ Hibernate Session (JPA superset)
- ☐ EntityManager (JPA Standard)

* betrifft alle Instanzen des Persistenzkontext (Hibernate & JPA)
** erzeugt persistente Instanz, Originalobjekt bleibt erhalten

Objektszustände Hibernate



Objekt - Zustandsdiagramm

Zustandsänderung durch Operationen des Persistenzmanager

- ☐ Hibernate Session (JPA superset)
- ☐ EntityManager (JPA Standard)

* betrifft alle Instanzen des Persistenzkontext (Hibernate & JPA)
** erzeugt persistente Instanz, Originalobjekt bleibt erhalten

Agenda

IDs, Lebenszyklus

DB-Identität

IDs, Composite IDs

ID-Generatoren

Lebenszyklus

Callbacks

Callback-Methoden in JPA

- Entität enthält eigene Callback-Methoden
- oder Verwendung von Entity Listeners (POJOs)
  - **@PrePersist**
    - Aufgerufen, bevor der Entity-Manager persistiert.
  - **@PreRemove**
    - Aufgerufen, bevor der Entity-Manager das Objekt entfernt.
  - **@PostPersist**
    - Aufgerufen, nach dem der Entity-Manager das Objekt wirklich mit einem INSERT persistent gemacht hat.
  - **@PostRemove**
    - Aufgerufen, nach dem der Entity-Manager das Objekt entfernte.
  - **@PreUpdate**
    - Vor einem Datenbank UPDATE aufgerufen.
  - **@PostUpdate**
    - Nach einem Datenbank UPDATE aufgerufen.
  - **@PostLoad**
    - Aufgerufen, wenn die Entity-Bean in den Persistence-Context geladen oder refreshed wurde.

Entity-Listeners und Callback-Methoden

- Anwendungen sind zum Beispiel:
  - Eigene Methoden können Objektzustände initialisieren (etwa das Alter einer Person auf der Basis des Geburtsdatums).
  - Konsistenzbedingungen können Zustände prüfen und das Speichern verhindern (wenn etwa eine Person mit negativen Alter gespeichert werden soll).

Anwendungen der Annotationen

- Nutzungsvarianten:
  - Die Callback-Methoden sind an der Entity festgemacht.
  - Eine eigene Entity-Listener Klasse definiert die Callback-Methoden.
    - Wird mit @EntityListeners an die Entity-Bean gehängt.

Das Alter als abgeleitetes Attribut

- Nehmen wir an, eine Entity-Bean soll ein Attribut age für das Alter anbieten.
- Das Attribut soll aber aus dem Geburtsdatum abgeleitet sein.
- Teil 1: Das Geburtsdatum dateOfBirth ist ein normales persistentes Attribut, age aber nicht.

```
@Entity( access=AccessType.FIELD )
public class Person {
  @Id public Integer id;
  public String name;
  public Calendar dateOfBirth;
  @Transient public int age;
```

@PostLoad

- Teil 2: Die Variable age soll dann gesetzt werden, wenn die Entity-Bean geladen oder refreshed wird.

```
@PostLoad
void calculateAge() {
  Calendar birth = new GregorianCalendar();
  birth.setTime( dateOfBirth );
  Calendar now = Calendar.getInstance();
  int adjust = 0;
  if ( now.get( Calendar.DAY_OF_YEAR ) -
       birth.get( Calendar.DAY_OF_YEAR ) < 0 )
    adjust = -1;
  age = now.get( Calendar.YEAR ) –
        birth.get( Calendar.YEAR ) + adjust;
}
```

@EntityListeners anwenden

- Nehmen wir an, eine Entity-Bean hat eine Funktion setLastUpdate(long millis), die immer vor jedem Update oder Persistieren aufgerufen werden soll.
  - Damit möchte den den letzten Zeitpunkt erfragen können.
- Man schreibe einen externen Entity-Listener und hänge ihn mit @EntityListeners an die Entity-Bean!

```
public class LastUpdateListener {
  @PreUpdate
  @PrePersist
  public void setLastUpdate( LastUpdateAware o ) {
    o.setLastUpdate( System.currentTimeMillis() );
  }
}
```

Konsistenz prüfen

- Eine setXXX()-Methode sollte niemals eine Bereichs- oder Konsistenzprüfung durchführen und aufgrund von Fehlern eine Ausnahme auslösen.
  - Was ist, wenn etwa das Alter hintereinander
    setAge(-111); setAge(34);
    initialisiert wird? Das Ergebnis stimmt.
- Um vor der Ablage in der Datenbank zu testen, ob die Werte oder Beziehungen korrekt sind, kann man eine Methode mit @PrePersist, @PreUpdate markieren, die im Fehlerfall eine IllegalArgumentException auslöst.

Ausnahme bei falschem Alter


```
@Entity class Person
{
 ...
 @PrePersist
 @PreUpdate
 void testAge()
 {
  if ( p.getAge() < 0 || p.getAge() > 120 )
    throw new IllegalArgumentException(
                    "Wrong age: " + p.getAge() );
 }
```

Eingriffsmöglichkeiten in Hibernate

- Einklinken von Funktionalität in Hibernate Lifecycle
- Möglichkeit generische Funktionalität einzubetten
- Interceptor-Interface
  - CallbackInterface ohne dass persistente Objekte dieses Implementieren müssen
- Hibernate Event-System

Auszug Interceptor


```
public class MyInterceptor implements Interceptor {
public boolean onLoad(...)  throws CallbackException {
. . .
public boolean onSave(...)  throws CallbackException {
. . .
}
public boolean preFlush(...) throws CallbackException {
. . .
}
```

Anmeldung Interceptor

- Globales Anmelden:

```
new Configuration().setInterceptor(new MyInterceptor());
```

- Lokales Anmelden auf Session-Ebene:

```
Session session = sf.openSession(new MyInterceptor());
```

Aufgabe

- Demo 5: Callback-Interfaces