</> JavaBeans, OSGi, Hibernate, Spring

Jan Caspar,Aktualisiert 30. Mai 2019,v 1.0.0

Java Beans

Java Beans sind einfache Java Klassen die speziellen Konventionen folgen um in Java ein Komponentenmodell zu realisieren

- Ein Java Bean muss immer einen leeren Konstruktor haben, es kann mehrere Konstruktoren haben muss aber mindestens einen leeren besitzen.
- Ein Java Bean kann aus folgenden Konstrukten bestehen: Properties, Methods, Events für die Persistence eigenschaft sollte es Serializable erben.
- Ein Java Bean darf keine öffentlichen Felder haben. Öffentliche Felder müssen in Properties gewrappt werden.
- Property Getter & Setter müssen mit `T getProperty()` und **void** `setProperty(T val)` annotiert werden, die Ausnahme bilden boolean-Getter sie werden mit **boolean** `isProperty()` annotiert.
- Events müssen ebenfalls einem strikten Namensmuster folgen, das Anhängen eines Listeners muss in der Form **void** `addEventTypeListener(EventTypeListener ev)` und das Abhängen in der Form **void** `removeEventTypeListener(EventTypeListener ev)` geschrieben werden. Wobei EventType der Name des Events (ohne Postfix s.u.) sein muss.
  - Eventtypen selbst müssen per Konvention mit Event enden und sollten von `java.util.EventObject` erben.
  - EventListenerTypen sollten den auf Listener enden und von `java.util.EventListener` erben. Als Beispiel `SampleEvent` **extends** `EventObject` und `SampleListener` **extends** `EventListener`

Info

Jede Klasse die diesen Konventionen folgt ist eine JavaBean, es ist nicht notewendig von einer JavaBean Klasse oder einem Bean Interface zu erben. Jede Klasse muss sich lediglich an dieses Muster halten. JavaBeans sind **keine** Enterprise JavaBeans! Was nicht notwendig ist aber in den Folien auch behandelt wurde ist die **BeanInfo-**Klasse, diese wird angelegt um die Properties für visuelle Editoren (z.B. GUI Tools) zur Verfügungs zu stellen. Sie ermöglicht es für die Tools Metadaten zum Bean zur Verfügung zu stellen.

Beispiel Event Implementierung in Bean

```
public interface TimerListener
 extends EventListener { void expired(TimerEvent event); }
public class Timer {
 private Vector<TimerListener> listeners = new Vector<>();
 void fireEvent(TimerEvent te) {
  Vector<TimerListener> listenersClone = (Vector<TimerListener>) listeners.clone();
  for (TimerListener l: listenersClone) {
   l.expired(te);
  }
 }
 public void addTimerListener(TimerListener listener) {
  this.listeners.add(listener);
 }
 public void removeTimerListener(TimerListener listener) {
  this.listeners.remove(listener);
 }
}

//Benutzung
timer.addTimerListener(e —> { //do something with e });
```

OSGi BundleActivator

```
package com.sample.myservice
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
    @Override
    public void start(BundleContext ctx) throws Exception {
        System.out.println("Starting!");
        ctx.registerService(SampleFactory.class, new SampleFactory(), null);
    }
    @Override
    public void stop(BundleContext ctx) throws Exception {
        System.out.println("Stopping!");
    }
}
```

OSGi Manifest

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyService bundle
Bundle-SymbolicName: com.sample.myservice
Bundle-Version: 1.0.0
Bundle-Activator: com.sample.myservice.Activator
Import-Package: org.apache.commons.logging;version="1.0.4"
Export-Package: com.sample.myservice.api;version="1.0.0"

Bundle-SymbolicName A name that identifies the bundle uniquely.

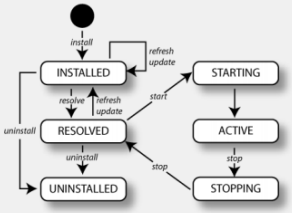
Bundle-Version This header describes the version of the bundle, and enables multiple versions of a bundle to be active concurrently in the same framework instance.

Bundle-Activator This header notifies the bundle of lifecycle changes.

Import-Package This header declares the external dependencies of the bundle that the OSGi Framework uses to resolve the bundle. Specific versions or version ranges for each package can be declared. In this example manifest file, the org.apache.commons.logging package is required at Version 1.0.4 or later.

Export-Package This header declares the packages that are visible outside the bundle. If a package is not declared in this header, it is visible only within the bundle.

OSGi Bundle Lifecycle



- Bundles haben einen definierten Lebenszyklus
- Zustandsänderungen können programmatisch oder durch einen Management Agent getriggert werden

OSGi Service Abfragen über den BundleContext:

```
ServiceReference serviceReference =
                ctx.getServiceReference(Logger.class.getName());
if(serviceReference != null) {
  Logger logger =
    (Logger) ctx.getService(serviceReference);
  if(Logger != null) { [...] }
}
```

OSGi ServiceTracker

- ServiceListener: Callbacks, wenn sich etwas ändert
- ServiceTracker: Verfolgen von Service Listener Events

```
public class LoggerServiceTracker extends ServiceTracker {
  public LoggerServiceTracker(BundleContext context) {
    super(context, Logger.class.getName(), null);
  }
  public Object addingService(ServiceReference reference) {
    Logger logger =
      (Logger) super.addingService(reference);
    // do something here...
    return logEventStore;
}
  public void removedService(ServiceReference reference,
    Object service) {
    // do something here...
    super.removedService(reference, service);
  }
}

public class Activator implements BundleActivator {
  private ServiceTracker _serviceTracker;
  public void start(BundleContext ctx) throws Exception {
    _serviceTracker =
      new LoggerServiceTracker(ctx);
    _serviceTracker.open();
  }
  public void stop(BundleContext ctx) throws Exception {
    _serviceTracker.close();
  }
}
```

OSGi vs. Java Module

- Das Java-Modulsystem stellt ein einfaches, allgemein einsetzbares Modulsystem dar.
  - ☑ einfachere Handhabung
  - ⊗ Keine Versionierung auf Modul- und Paketebene möglich.
  - ⊗ Import geschieht auf Modul- nicht auf Paketebene.
  - ⊗ Das Java-Modulsystem unterstützt keine dynamischen Module.
- OSGi definiert ein komplexes Modulsystem für spezielle Anwendungsszenarien.
  - ☑ In bestimmten Anwendungsszenarien (z. B. bei hochverfügbaren Systemen) ist auch in aktuellen Java-Versionen OSGi erforderlich.

Spring Application Context

Die Hauptaufgabe des Applikationskontexts:

- Verwalten des Lebenszyklus der Spring-Beans
- Aufbau des Bean-Grafen

Weitere Aufgaben

- Laden von Ressourcen (Filesystem, Klassenpfad etc.)
- Internationalisierung
- Feuern von Events

Erzeugen und Schließen eines Applikationskontexts

```
try (AbstractApplicationContext factory =
  new ClassPathXmlApplicationContext(
    "<path>/applicationContext.xml")) {
      // work with beans
}
```

Erzeugen des Context aus Java Config

```
AbstractApplicationContext ctx =
  new AnnotationConfigApplicationContext(
    Application.class);
```

Spring Bean Configuration

Bean in Java konfiguriert

```
@Configuration
public class AppConfig {
  @Bean
  public EmployeeDao employeeDao() {
    return new EmployeeDaoImpl();
  }
  @Bean(name="workLog")
  public WorkLogFacade getWorkLog() {
    return new WorklogImpl(employeeDao());
  }
}
```

@Configuration Definition einer Klasse, die als Bean-Factory fungiert.

@Bean Factory-Methode, die Bean-Instanz erzeugt.

Standardmäßig legt der Methodenname den Namen des Beans fest.

Spring DI XML

Setter Injection

```
<bean id="workLog" class="swt6.spring.WorkLogImpl">
  <property name="employeeDao" ref="employeeDao" />
</bean>
```

CTOR Injection

```
<bean id="workLog" class="swt6.spring.WorkLogImpl">
  <constructor-arg ref="employeeDao" />
</bean>
```

Spring DI @Autowired

- Mit @Autowired wird der annotierten Property ein Bean vom Typ der Property zugewiesen.
- Mit Qualifizierern wird die Auswahl eingeschränkt.
- Kein/mehrere passende(s) Beans *UnsatisfiedDependencyException*.
- Datenkomponenten und Setter-Methoden können annotiert werden.

Qualifier setzen

```
<context:annotation-config />
<bean id="employeeDaoJdbc" class="swt6.spring.EmployeeDaoJdbcImpl">
  <qualifier type="DefaultDao" />
</bean>
```

Qualifier benutzen

```
public class WorkLogImpl implements WorkLogFacade {
  @Autowired(required=true)
  @DefaultDao
  private EmployeeDao employeeDao;
}
```

Spring DI @Resource

- @Resource ist Java EE-Annotation.
- Beans werden über deren ID angesprochen.
- Wird bei @Resource kein Name angegeben, wird dafür der Name der annotierten Property verwendet.
- Ist die Suche per Namen erfolglos, wird per Typ gesucht.

Bean Config

```
<context:annotation-config />
 <bean id="employeeDaoJdbc" class="swt6.spring.EmployeeDaoJdbcImpl">
</bean>
```

Resource Verwendung

```
public class WorkLogImpl implements WorkLogFacade {
  @Resource(name="employeeDaoJdbc") // default for name is employeeDao
  private EmployeeDao employeeDao;
}
```

Spring DI @Inject

Spring interpretiert Annotationen des JSR 330 (DI for Java).

@Inject: Einer Datenkomponente wird ein typkompatibles Objekt zugewiesen.

@Named: Komponente mit dem angegebenen Namen wird injiziert.

Standardname: Bezeichner der Datenkomponenten

Bean Config

```
<context:annotation-config />
  <bean id="employeeDao" class="swt6.spring.EmployeeDaoJdbc">
</bean>
```

Resource Verwendung

```
public class WorkLogImpl implements WorkLogFacade {
  @Inject @Named("employeeDao")
  private EmployeeDao employeeDao;
}
```

Spring DI @Qualifier

Definition eines Qualifizierers mit @Qualifier (JSR 330):

Qualifizierers definieren

```
@Qualifier
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Dao {
  String technology() default "JPA"
}
```

Bean Config für Qualifizierer

```
<bean id="employeeDaoJdbc" class="swt6.spring.EmployeeDaoJdbcImpl">
  <qualifier type="Dao">
    <attribute key="technology" value="JDBC" />
  </qualifier>
</bean>
```

Verwendung des Qualifizierers

```
@Dao(technology="JDBC")
public class EmployeeDaoJdbc implements EmployeeDao { ... }
```

Injizieren anhand des Qualifizierers

```
public class WorkLogImpl implements WorkLogFacade {
  @Inject @Dao(technology="JDBC")
  private EmployeeDao employeeDao;
}
```

Spring @Configuration

@Configuration gibt an das diese Klasse verwendet wird um Spring zu konfigurieren (anstatt XML).

Spring @ComponentScan & @Component

@ComponentScan muss in Kombination mit @Configuration angegeben werden und ist gleichwertig mit der XML Notation <context:component-scan>. @ComponentScan bewirkt das Spring nach *@Component* annotierten Klassen sucht (auch nach Derivaten von @Component wie etwa *@Repository*, *@Controller* *@Service* oder auch *@Configuration*).
Dies geschieht in dem Package (unter allen Unterpackages von diesem) in dem @ComponentScan annotiert ist, sollte das Package in dem gesucht wird abweichen so kann man dies @ComponentScan mitgeben @ComponentScan(basePackages = { "com.sample.zeuch" }).
Es gibt weitere Filtermöglichkeiten wie etwa Regex um Compenents explizit auszunehmen oder implizit miteinzubeziehen. Alles was von @ComponentScan gefunden wurde wird automatisch in den ApplicationContext aufgenommen.
Die verschiedenen Arten von @Component beziehen sich auf die jeweilige Schicht in

- @Component ist eine generische Komponente ohne besondere Zuordnung
- @Service markiert eine Komponente für den „Service Layer“ - die Geschäftsschicht
- @Repository markiert eine Komponente für den „Persistence Layer“ - Die Datenschicht
- @Controller markiert einen Spring MVC Contoller - Er gehört zur Präsentationschicht
- @Configuration markiert das die Komponente eine Konfiguration ist
- @Aspect markiert eine Komponente als Aspect für AOP

Spring scannt den ClassPath nach diesen Annotation und nimmt sie automatisch durch @ComponentScan in den ApplicationContext auf.

Spring Transaktionen mit @Transactional

- Deklarative Transaktionen über AOP, sprich Sie sind als Cross-Cutting-Concern von der Geschäftslogik getrennt.
- Default Wert für readonly ist false
- @Transactional wird über AOP abgehandelt, das heißt @Transactional funktioniert nur bei Zugriff über den von Spring generierten Proxy. Eine selbst intazierte Version hat keine Transaktionsfunktionalität. Da die Annotation auch auf Methoden gemacht werden kann, ist darauf zu achten das klasseninterne Aufrufe auch **nicht** vom Proxy erfasst werden.

Aktivieren von @Transactional über XML

```
<tx:annotation—driven transaction—manager="txManager"/>
```

Konfiguration des Transaction Manager (DataSource)

```
<bean id="txManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref= "datasource" />
</bean>
```

Konfiguration des Transaction Manager (JPA)

```
<bean id="txManager"
  class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Beispiel DAO mit @Transactional (Methodenebene)

```
@Repository
public class TestDao {
    private JdbcTemplate jdbcTemplate;
    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    @Transactional
    public void insertData(String arg) {
        this.jdbcTemplate.update("...", arg);
    }
}
```

Spring + JPA @PersistenceContext

„*Expresses a dependency on a container-managed EntityManager and its associated persistence context.*“ - ofizelle JPA Doku.

Spring interpretiert die @*PersistenceContext* Annotation und injiziert automatisch einen EntityManager. Unter der Haube generiert der *PersistenceAnnotationBeanPostProcessor* automatisch einen JPA Entity Manager und injiziert diesen.
Über @*PersistenceContext*(unitName=“name“) lässt sich ein bestimmter Persistence Context aus der Config laden.

Beispiel generisches DAO mit @PersistenceContext

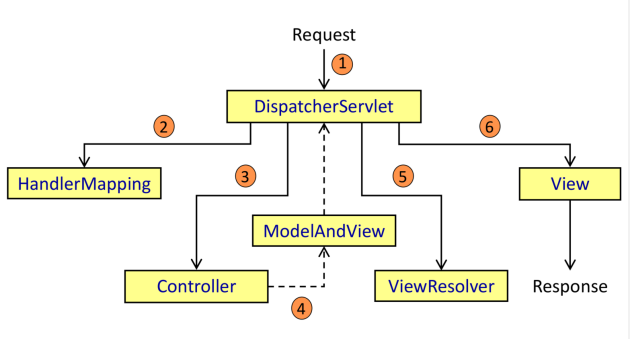
```
public abstract class AbstractJpaDAO< T extends Serializable > {

  @PersistenceContext
  EntityManager entityManager;

  private Class<T> clazz;
  public final void setClazz(Class<T> clazzToSet) {
    this.clazz = clazzToSet;
  }
  public List<T> findAll(){
    return entityManager
      .createQuery( "from_" + clazz.getName() )
      .getResultList();
  }
  public void create(T entity){ entityManager.persist( entity ); }
  public T update(T entity){ return entityManager.merge( entity ); }
  public void delete(T entity){ entityManager.remove(entity); }
}
```

Spring MVC

Spring MVC wird aktiviert in dem bei @Configuration zusätzlich die Annotation @EnableWebMvc angibt.
Spring MVC handelt alles Requests über einen globalen Dispatcher, dieser sucht in dem HandlerMapping (Route Table / Routing Table für nicht Spring Leute ...) nach einer definierten Route und delegiert dies dann an den passenden Controller weiter, dieser verarbeitet den Request und gibt ein View Objekt retour welches anschließend durch die vom ViewResolver definierte ViewEngine gerendert und zurückgegeben wird.



Beispiel MVC Controller aus Folien

```
@Controller
public class ViewLogbookEntriesController {
  @Autowired
  private WorkLogFacade workLog;

  @RequestMapping("/employees/{employeeId}/entries")
  public String listLogbookEntries (
    @PathVariable("employeeId") long employeeId, Model model) {
    Employee empl = this.workLog.findEmployeeById(employeeId);
    model.addAttribute("employee", empl);
    model.addAttribute("logbookEntries", empl.getLogbookEntries());
    return "logbookEntriesOfEmployee";
  }
}
```

Beispiel REST Controller

```
@RestController
public class QuoteController {
    private QuoteRepository repository;
    public QuoteController(@Autowired QuoteRepository repository){
        this.repository = repository;
    }

    @RequestMapping(value="/")
    public Iterable<Quote> index(Model m){
        return repository.findAll();
    }

    @RequestMapping(value="/quote/{id}")
    public Optional<Quote> getQuoteById(@PathVariable("id") long id){
        return repository.findById(id);
    }

    @PostMapping(value = "/quote/add")
    public Quote addQuote(@RequestBody Quote q){
        return repository.save(q);
    }
}
```

Spring Advices

Ein Advice ist eine Aktion auf einen Aspekte an einem JoinPoint ausgeführt wird. Dies kann vor dem Aufruf (before), nach dem Aufruf (after) oder bei beiden (arround) passieren. Das Hauptaufgabengebiet von Aspekten sind Cross Cutting Concerns wie logging, profiling, caching und transaction management.

Beispiel Aspect der vor jedem Methodenaufruf in @Repository loggt

```
@Component
@Aspect
public class LoggingAspect {
    private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

        @Pointcut("@target(org.springframework.stereotype.Repository)")
    public void repositoryMethods() {};

    @Before("repositoryMethods()")
    public void logMethodCall(JoinPoint jp) {
        String methodName = jp.getSignature().getName();
        logger.info("Before_" + methodName);
    }
}
```

Beispiel Aspect der die Ausführngszeit jeder @Repository Methode misst

```
@Aspect
@Component
public class PerformanceAspect {
    @Pointcut("within(@org.springframework.stereotype.Repository_*)")
    public void repositoryClassMethods() {};

    @Around("repositoryClassMethods()")
    public Object measureMethodExecutionTime(ProceedingJoinPoint joinPoint)
      throws Throwable {
        long start = System.nanoTime();
        Object returnValue = joinPoint.proceed();
        long end = System.nanoTime();
        String methodName = joinPoint.getSignature().getName();
        System.out.println(
          "Execution_of_" + methodName + "_took_" +
          TimeUnit.NANOSECONDS.toMillis(end — start) + "_ms");
        return returnValue;
    }
}
```

Pointscuts können auch über XML definiert werden

Aktivieren von @Transactional über XML

```
<aop:config>
    <aop:pointcut id="anyDaoMethod"
      expression="@target(org.springframework.stereotype.Repository)"/>
</aop:config>
```

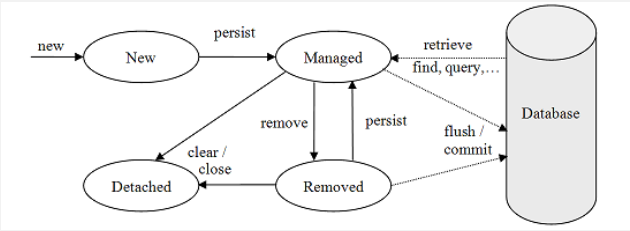
Es gibt folgenden Möglichkeiten in PointCuts einzugreifen

- @Before("pointCutName()") resultiert in einer Methode mit *(JoinPoint jp)* Argument
- @After("pointCutName()") resultiert in einer Methode mit *(JoinPoint jp)* Argument, egal ob eine Exception gefolgen ist oder nicht
- @AfterReturning("pointCutName()") resultiert in einer Methode mit *(JoinPoint jp, Object entity)* Argument, wobei Entity das von der Methode zurückgegeben Objekt ist.
- @AfterThrowing ("pointCutName()", throwing = e) resultiert in einer Methode mit *(JoinPoint jp, Exception e)* Argument, wird aufgeführt wenn eine Exception geflogen ist
- @Around("pointCutName()") resultiert in einer Methode mit *(ProceedingJoinPoint pjp)* Argument

JoinPoint
Der JoinPoint verfügt über folgende Methoden
getArgs() gibt die Argumente die übegeben wurden retour
getThis() gibt das Proxy Objekt retour
getTarget() gibt das eigentlich Objekte hinter dem Proxy retour
getSignature() gibt die Methodensignatur retour
Der **ProceedingJoinPoint** verfügt über eine Methode *.proceed()* welche ihn anweist, dass der Before Teil abgeschlossen ist.

JPA - Entity Manager

Entitäten haben verschiedene Zustände, dieser kann Transient (New), Persistent (Managed) oder Detached sein. Dieser kann sich durch Interaktion mit dem Entity Manager ändern.



Aufgaben des Persistenz-Managers

- Speichern, Laden und Aktualisieren von Objekten.
- Durchführung von Abfragen.
- Überwachung und Durchführung von Transaktionen.
- Verwaltung der gepufferten Objekte (Cache).

Funktionen des Entity Managers

persist(entity) Speichern einer Entität in DB. transient → persistent

merge(entity) transient, detached, persistent → persistent

remove(entity) Löschen aus der DB, persistent → transient

find(entityClass,id) Laden einer Entität mit ID.

getReference(entityClass,id) Laden einer Entität mit ID. Referenz kann ein Proxy-Objekt sein.

createQuery(jql) Abfrage in JPA-QL

JPA: Speichern von Entity

```
EntityManager em = emFactory.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(employee);
tx.commit();
em.close();
```

JPA: Laden von Entities

```
EntityManager em = emFactory.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
List<Employee> emplList =
  em.createQuery("select_e_from_Employee_e").getResultList();
for (Employee e : emplList) System.out.println(e);
tx.commit();
em.close();
```

JPA: Aktualisieren von Entities

```
EntityManager em = emFactory.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Employee empl = em.find(Employee.class, emplId);
empl.setFirstName("Hugo");
tx.commit();
em.close();
```

JPA Annotationen auf Klassen

@Entity muss jede Entität haben

@Embeddable Annotiert das diese Klassen in einer Entität eingebettet werden kann

@Table definiert den Table Namen in der Datenbank (Standard = Entitätenname)

JPA Annotationen auf Properties

@Id annotiert den Primärschlüssel, ist für jede Entität erforderlich

@GeneratedValue(strategy=) Schlüsselgenerierungsverfahren (Standard = Auto), Möglichkeiten: SEQUENCE, IDENTITY, TABLE (HI LO Alg.)

@SequenceGenerator Mit @SequenceGenerator bzw @**TableGenerator** kann die DB-Sequenz bzw. -Tabelle näher spezifiziert werden.

@EmbeddedId wird für zusammengesetzt Schlüssel benötigt - dieser wird als @Embeddable annotierte Klasse implementiert

@JoinColumn(name="fkId") verhindert das Fremdschlüssel in eigener Tabelle generiert werden

JPA Vererbungsstrategien

Vererbung kann auf drei Arten realisiert werden

- Tabelle pro konkreter Klasse
- Tabelle für gesamte Klassenhierarchie (TPH)
- Tabelle pro Klasse (TPC)

Tabelle pro konkreter Klasse

Die abstrake Klasse wird nicht @Entity annotiert sondern @MappedSuperClass

Tabelle pro konkreter Klasse

```
@MappedSuperClass
public abstract class Employee { ... }
@Entity
public class PermanentEmployee extends Employee { ... }
```

Tabelle für gesamte Klassenhierarchie

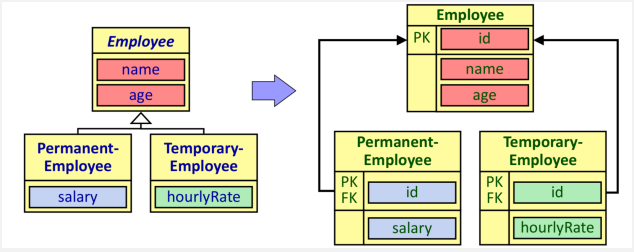
Es wird in der Tabelle ein Discrimantor festgelegt welcher es ermöglicht zu erkennen um welches Objekt es sich handelt.

In der Tabelle steht für Permanent P und Temporary T in der Spalte emType

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="emType",
  discriminatorType=DiscriminatorType.STRING)
public abstract class Employee { ... }

@DiscriminatorValue("P")
class PermanentEmployee { ... }
@DiscriminatorValue("T")
class TemporaryEmployee { ... }
```

Tabelle pro Klasse



Eine Tabelle pro Klasse

```
@Entity
@Inheritance(strategy=
  InheritanceType.JOINED)
public abstract class Employee { ... }

@Entity
public class PermanentEmployee
  extends Employee { ... }
```

JPA Assoziationen

FetchTypes

Eager lädt alles sofort mit

Lazy lädt bei ersten Zugriff Daten aus DB nach

Cascade

CascadeType.PERSIST: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed

CascadeType.MERGE: cascades the merge operation to associated entities if merge() is called or if the entity is managed

CascadeType.REMOVE: cascades the remove operation to associated entities if delete() is called

CascadeType.REFRESH: cascades the refresh operation to associated entities if refresh() is called

CascadeType.DETACH: cascades the detach operation to associated entities if detach() is called

CascadeType.ALL: all of the above

orphanRemoval=true entfernt automatisch verwaist Einträge

Mapping

@OneToOne 1 : 1 Beziehung

OneToOne mit Foreign Key

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private Address address;
}

@Entity
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;

    @OneToOne(mappedBy = "address")
    private User user;
}
```

@OneToMany 1 : N Beziehung

OneToMany auf objName Feld

```
@OneToMany(mappedBy = "objName",
  fetch = FetchType.Lazy,
  cascade = CascadeType.ALL)
private Set<Obj> objs = new HashSet<>();
```

@ManyToMany N : M Beziehung

ManyToMany mit Join Table

```
@ManyToMany(cascade = CascadeType.ALL,
  fetch = FetchType.LAZY)
@JoinTable(name ="OBJ1_OBJ2",
  joinColumns =
    @JoinColumn(name ="OBJ1_ID"),
  inverseJoinColumns =
    @JoinColumn(name = "OBJ2_ID"))
  private Set<Obj2> obj2s = new HashSet<>();
```

@ManyToOne N : 1 Beziehung

ManyToOne

```
@ManyToOne(fetch = FetchType.LAZY,
  cascade = CascadeType.ALL)
    private Obj obj;
```