OneToOne relationship

Dentro do universo orientado a objetos, objetos das mais diversas classes interagem entre si a fim de realizar suas operações. Dependendo de como esses objetos associam-se, podemos obter os mais variados relacionamentos. Um desses é o OneToOne, abordado nesse capítulo. Um relacionamento OneToOne ocorre quando a multiplicidade entre as classes pertencentes ao relacionamento é 1-1. Considerando o modelo do domínio da aplicação racing car, conforme figura X, podemos observar que há um relacionamento OneToOne entre as classes `Piloto` e `Perfil` – 1 `Piloto` possui 1 `Perfil` e 1 `Perfil` está associado a 1 `Piloto`. Além desse, há o relacionamento reflexivo OneToOne, quando uma classe é associada à ela mesma, aqui representado pelo relacionamento onde 1 `Pessoa` é casada com outra `Pessoa`.

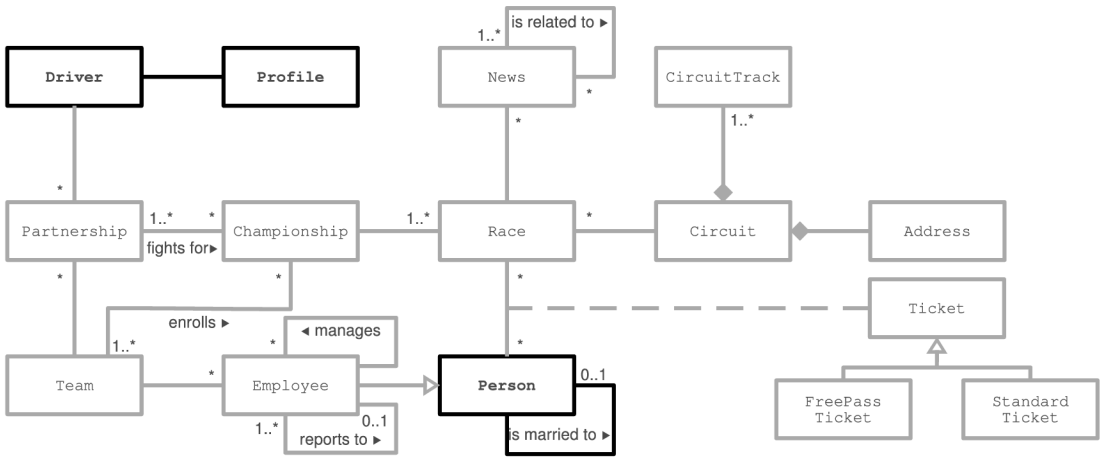


Figura X. Modelo do domínio da aplicação destacando os relacionamentos OneToOne

Relacionamento OneToOne unidirecional

Conforme visto na introdução desse capítulo, dentro do modelo do domínio da aplicação racing car, há um relacionamento OneToOne entre as classes `Piloto` e `Perfil`. Inicialmente, vamos considerar que apenas a classe `Piloto` requer visibilidade para a classe `Perfil`, o que caracteriza o relacionamento como unidirecional de acordo com conforme ilustrado pela figura X

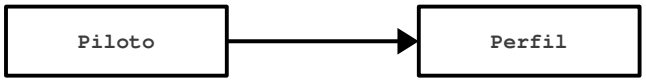


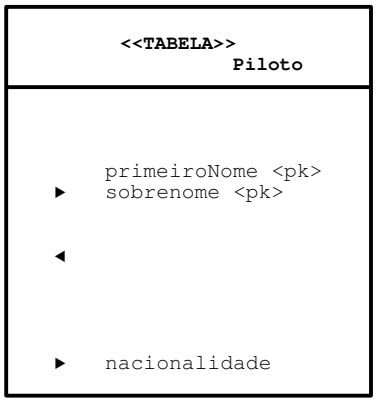
Figura X. Relacionamento OneToOne unidirecional entre as classes Piloto e Perfil

Para que o relacionamento seja unidirecional, apenas um dos lados da associação deve referenciar o outro. No exemplo da figura X, o objeto `Piloto` referencia um, e apenas um, objeto `Perfil`. Por outro lado, o objeto `Perfil` não referencia o objeto da classe `Driver`, conforme listagem X

```
@Entity
public class Piloto extends Pessoa {

  private Integer id;
  private String nome;
  private Date nascimento;

}
```

```
@Entity
public class Pessoa implements Serializable {

  private Integer id;
  private String nome;
  private Date nascimento;

  @Id
  @GeneratedValue
  public Integer getId() {
    return id;
  }
  public void setId(Integer id) {
    this.id = id;
  }

  public String getNomeCompleto() {
    return nomeCompleto;
  }
  public void setNomeCompleto(String nomeCompleto) {
    this.nomeCompleto = nomeCompleto;
  }

  @Temporal(TemporalType.DATE)
  public Date getDataNascimento() {
    return dataNascimento;
  }
```

```
  public void setNascimento(Date nascimento) {
    this.nascimento = nascimento;
  }

  public List<Campeonato> getCampeonatoList() {
    return cList;
  }
  public void setFuncionarioList(List<Funcionario> fList) {
      this.campeonatoList = cList;
  }

  public static class Programacao implements Serializable {

    private Date horarioCorrida

}

@Entity
public class Piloto implements Serializable {

  private String pseudonimo;
  private Perfil perfil;

  @EmbeddedId
  public Nome getNome() {
    return nome;
  }
  public void setNome(Nome nome) {
    this.nome = nome;
  }

  @Enumerated(EnumType.STRING)
  public Nacionalidade getNacionalidade() {
    return nacionalidade;
  }
  public void setNacionalidade(Nacionalidade nacionalidade) {
    this.nacionalidade = nacionalidade;
  }

  @OneToOne(cascade=CascadeType.PERSIST)
  @JoinColumn(nullable=false)
  public Perfil getPerfil() {
    return perfil;
  }
  public void setPerfil(Perfil perfil) {
    this.perfil = perfil;
  }

}
```



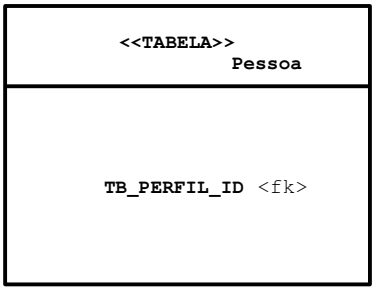
Na classe `Piloto`, colocamos a anotação `@OneToOne` sobre a propriedade `perfil` para informar ao provedor de persistência que há um relacionamento OneToOne entre as classes `Piloto` e `Perfil`. Adicionalmente, configuramos o atributo `cascade` como `CascadeType.PERSIST`, uma vez que desejamos que o objeto `Perfil` associado ao objeto `Driver` seja persistido quando persistirmos esse último.

Um outro ponto a ser observado é que a propriedade `perfil`, marcada com a anotação `@OneToOne`, foi mapeada para a chave estrangeira `perfil_id`. Por padrão, a propriedade marcada com a anotação `@OneToOne` é mapeada para uma chave estrangeira cujo nome é composto pela combinação `<PROPRIEDADE>_<PRIMARY_KEY>`. A primeira porção, `<PROPRIEDADE>`, é o nome da propriedade marcada com a anotação `@OneToOne`. Já a segunda porção, `<PRIMARY_KEY>`, é o nome da chave primária referenciada. Para o mapeamento previamente demonstrado, obtemos o seguinte

<PROPRIEDADE>	perfil	Nome da propriedade marcada com a anotação @OneToOne
<PRIMARY_KEY>	id	Chave primária referenciada, ou seja, a chave estrangeira perfil_id referencia a chave primária id, definida pela tabela Perfil

Por fim, observe o uso da anotação @JoinColumn, útil quando se deseja personalizar a configuração da chave estrangeira. Aqui, usamos a anotação @JoinColumn para informar ao provedor de persistência que a chave estrangeira profile_id não pode ser nula – Por padrão, a chave estrangeira mapeada aceita valores nulos. Uma outra funcionalidade interessante dessa anotação é quando desejamos personalizar o nome da chave estrangeira, através do atributo name. Isso ocorre geralmente quando lidamos com sistemas legados. Por exemplo, vamos considerar que o nome da chave estrangeira fosse TB_PERFIL_ID. Para isso, bastaríamos configurar o atributo name da anotação @JoinColumn, conforme a seguir

```
@Entity
public class Pessoa implements Serializable {

  ...

  @OneToOne(cascade=CascadeType.PERSIST)
  @JoinColumn(name="TB_PERFIL_ID", nullable=false)
  public Perfil getPerfil() {
    return perfil;
  }
  public void setPerfil(Perfil perfil) {
    this.perfil = perfil;
  }

}
```



Putting our mapping into action

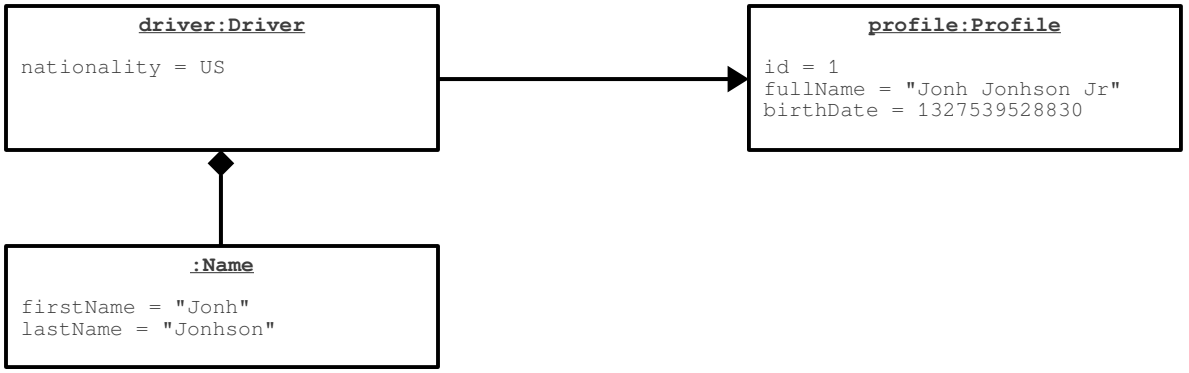
Uma vez que mapeamos as classes Driver e Profile, vamos criar uma pequena aplicação, conforme a seguir

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("racingPU");
EntityManager manager = factory.createEntityManager();

manager.getTransaction().begin();

Driver driver = new Driver();
driver.setName(new Name("Jonh", "Jonhson"));
driver.setNationality(Nationality.US);

Profile profile = new Profile();
profile.setFullName("Jonh Jonhson Jr");
profile.setBirthDate(new Date());

driver.setProfile(profile);

manager.persist(driver);

manager.getTransaction().commit();
manager.close();
```

Inicialmente, criamos dois objetos, sendo um da classe Driver e outro da Profile. Logo em seguida,

configuramos o relacionamento entre o objeto `Driver` e `Profile` invocando o método acessor `driver.setProfile(profile)`. Por fim, persistimos o objeto `Driver` invocando o método `persist` do objeto `EntityManager`. Isso faz com que o objeto `Profile` associado ao `Driver` seja automaticamente persistido, uma vez que configuramos o atributo `cascade` da anotação `@OneToOne` como `CascadeType.PERSIST`.

Uma vez executado, vamos ver como fica a representação desse relacionamento, do ponto de vista do modelo orientado a objetos bem como do banco de dados relacional, conforme a seguir



Driver			
firstName	**lastName**	**nationality**	**profile_id**
Jonh	Jonhson	US	**1**

Profile		
id	**fullName**	**birthDate**
1	Jonh Jonhson Jr	1327539528830

Figure 5. Our sample application from the point of view of the object-oriented and relational database model

Para o modelo orientado a objetos, usamos o diagrama de objetos, que modela o estado da aplicação após invocarmos `manager.getTransaction().commit()` - Aqui, para fins de exemplificação, vamos considerar que o provedor de persistência configurou o identificador do objeto `Profile` com o valor `1`.

Pelo lado do banco de dados relacional, o objeto `Driver` foi mapeado para uma linha da tabela `Driver` ao passo que o objeto `Profile` foi mapeado para uma linha da tabela `Profile`. Quanto ao relacionamento entre os objetos `Driver` e `Profile`, o modelo do banco de dados relacional usa uma foreign key para relacionar os objetos mapeados. Aqui, o objeto `Driver`, identificado pela coluna `firstName` com valor `Jonh`, e `lastName` com valor `Jonhson`, referencia o objeto `Profile`, identificado pela coluna `id` com valor `1` através da foreign key `profile_id`, conforme destacado pela figura 5.

Bi-directional OneToOne relationship

Dentro do modelo do domínio da aplicação racing car, o relacionamento unidirectional entre as classes `Driver` e `Profile` é mais do que suficiente. Isso se deve ao fato de que, dentro do contexto da nossa aplicação, o objeto `Profile` sempre será acessado a partir de um objeto `Driver`, através da instrução `driver.getProfile()`. Porém, vamos considerar que um determinado caso de uso da aplicação

requer que você acesse o objeto `Driver` a partir do objeto `Profile`, de forma que o relacionamento entre as classes `Driver` e `Profile` seja bi-directional, conforme a seguir

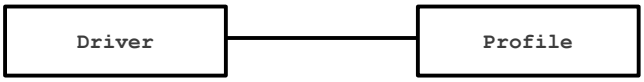
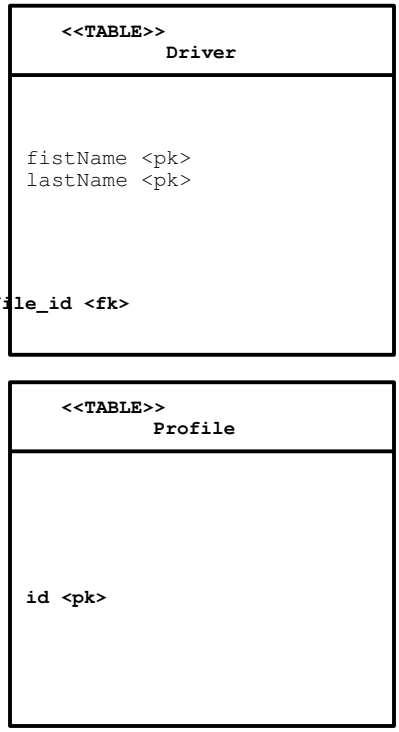


Figure 5. Bi-directional OneToOne relationship between Driver and Profile

Considerando o relacionamento entre um objeto da classe `Driver` e outro da `Profile`, para que o relacionamento seja bi-directional, o objeto da classe `Driver` bem como o da `Profile` referenciam-se um ao outro. Para isso, vamos mapear ambas as classes, dessa vez considerando o relacionamento entre a classe `Driver` e `Profile` como bi-directional, conforme a seguir

```
@Entity
public class Driver implements Serializable {

  private Name name;
  private Profile profile;

  @EmbeddedId
  public Name getName()
  public void setName(Name id)

  …

  @OneToOne
  public Profile getProfile()
  public void setProfile(Profile profile)

}
```

```
@Entity
public class Profile implements Serializable {

  private Integer id;
  private Driver driver;

  …

  @Id @GeneratedValue
  public Integer getId()
  public void setId(Integer id)

  @OneToOne(mappedBy="profile")
  public Driver getDriver()
  public void setDriver(Driver driver)

}
```



Dessa vez, adicionamos à classe `Profile` a propriedade `driver`, marcada com a anotação `@OneToOne` para informar ao provedor de persistência que que há um relacionamento OneToOne entre as classes `Profile` e `Driver`. No entanto, ao observar o mapeamento resultante, você verá que, diferentemente da propriedade `profile` da classe `Driver`, que foi mapeada para a foreign key `profile_id`, a propriedade `driver` da classe `Profile` não foi mapeada para nenhuma foreign key, devido ao atributo `mappedBy`, definido pela anotação `@OneToOne`.

A marcação `@OneToOne(mappedBy="profile")`, colocada sobre a propriedade `driver` da classe `Profile`, especifica que a propriedade `driver` é mapeada pela propriedade **`profile`** da classe `Driver`. Em outras palavras, é como se a propriedade **`profile`** da classe `Driver` mapeasse, ao

mesmo tempo, além dela mesma, a propriedade `driver` da classe `Profile`. Como consequência, o provedor de persistência usará apenas a propriedade **`profile`** da classe `Driver` para mapear o relacionamento, ignorando qualquer chamada à propriedade `driver` da classe `Profile`.

Entenda que o uso do atributo `mappedBy` é apenas um mecanismo pelo qual o provedor de persistência mapeia um relacionamento bi-directional. Caso você omita o atributo `mappedBy` ao mapear um relacionamento bi-directional, o provedor de persistência irá analisar cada propriedade pertencente ao relacionamento de forma individual, ou seja, independentemente de qualquer outra propriedade. Como consequência de sua omissão, uma série de efeitos colaterais pode surgir. Dentre esses, podemos citar mapeamento duplicado ou redundante, violação de restrição, execução de instruções SQL duplicadas ou desnecessárias, dentre outros fatores.

Por fim, vale a regra: quando o relacionamento for bi-directional, você deve, além de marcar cada propriedade pertencente ao relacionamento com a anotação `@OneToOne`, configurar uma dentre ambas as propriedades com o atributo `mappedBy`. Para o nosso exemplo, configuramos o atributo `mappedBy` através da anotação `@OneToOne` colocada sobre propriedade `driver` da classe `Profile`.

Putting our mapping into action

Uma vez que configuramos o relacionamento entre as classes `Driver` e `Profile` como bi-directional, vamos criar uma aplicação onde colocaremos em prática os conceitos introduzidos, conforme a seguir

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("racingPU");
EntityManager manager = factory.createEntityManager();

manager.getTransaction().begin();

Driver driver = new Driver();
driver.setName(new Name("Jonh", "Jonhson"));
driver.setNationality(Nationality.US);

Profile profile = new Profile();
profile.setFullName("Jonh Jonhson Jr");
profile.setBirthDate(new Date());

manager.persist(driver);
manager.persist(profile);
manager.flush();
```

Driver

firstName	lastName	profile_id
Jonh	Jonhson	**NULL**

```
profile.setDriver(driver);
manager.flush();
```

Jonh	Jonhson	**1**

```
driver.setProfile(profile);
manager.flush();
```

Jonh	Jonhson	**1**

```
profile.setDriver(null);
manager.flush();
```

Jonh	Jonhson	**NULL**

```
driver.setProfile(null);
manager.flush();
```

Jonh	Jonhson	**1**

```
driver.setProfile(profile);
profile.setDriver(driver);

manager.getTransaction().commit();
manager.close();
```

Inicialmente, criamos e persistimos dois objetos, sendo um da classe `Driver` e outro da `Profile`. Logo após, invocamos o método `flush` do objeto `EntityManager`. Usamos esse método para sincronizar os objetos `Driver` e `Profile` com o banco de dados relacional - Por padrão, os objetos só serão sincronizados após invocarmos o método `commit` do objeto `EntityTransaction` ou se houver alguma query relacionada com os objetos mapeados. Após persistirmos e invocarmos o método `flush`, o modelo do banco de dados relacional resultante será

Driver			
firstName	**lastName**	**nationality**	**profile_id**
Jonh	Jonhson	US	**NULL**

Profile		
id	**fullName**	**birthDate**
1	Jonh Jonhson Jr	1327539528830

Figure 5. Our sample application from the point of view of the relational database model

Como não configuramos o relacionamento entre o objeto `Driver` e `Profile` antes de invocarmos o método `flush` do objeto `EntityManager` - Para fins de exemplificação, vamos considerar que o provedor de persistência configurou o identificador do objeto `Profile` com o valor `1` -, a foreign key `profile_id`, responsável por mapear o relacionamento entre a classe `Driver` e `Profile`, está configurada com valor `NULL`, indicando que o objeto `Driver`, identificado pela coluna `firstName` com valor `Jonh`, e `lastName` com valor `Jonhson`, não referencia nenhuma linha da tabela `Profile`.

Logo após, executamos uma série de instruções com as propriedades `profile` e `driver`, definidas pelas classes `Driver` e `Profile`, respectivamente. Observe que apenas a propriedade `profile` da classe `Driver` mapeia o relacionamento através da foreign key `profile_id`. Porém, se configurarmos o relacionamento usando a propriedade `driver` da classe `Profile`, você verá que o relacionamento não será sincronizado com o banco de dados porque o provedor de persistência ignora qualquer invocação dessa propriedade, uma vez que configuramos o atributo `mappedBy`, conforme mostrado abaixo

```
@Entity
public class Profile implements Serializable {

  …

  @OneToOne(mappedBy="profile")
  public Driver getDriver()
  public void setDriver(Driver driver)

}
```

Ao marcar a propriedade `driver` da classe `Profile` com `@OneToOne(mappedBy="profile")`, informamos ao provedor de persistência que a propriedade **`profile`** da classe `Driver` é responsável por mapear a propriedade `driver` da classe `Profile`, ignorando qualquer chamada à mesma.

Mais ao final da nossa aplicação de exemplo, configuramos o relacionamento bi-directional entre a instância da classe `Driver` e `Profile`, respectivamente, conforme a seguir

```
driver.setProfile(profile);
profile.setDriver(driver);
```

Como você pode observar, o objeto `Driver` está consciente do relacionamento com o objeto `Profile` da mesma forma que o objeto `Profile` está consciente do relacionamento com o objeto `Driver`. Aliás, só para constar, para que o relacionamento bi-directional seja implementado corretamente, o desenvolvedor deve configurar ambas as propriedades que formam o relacionamento, conforme mostrado previamente.

Mapping using Scala as programming language

Para o mapeamento objeto-relacional usando Scala, iremos escrever nossas classes como a seguir

```
@Entity
class Circuit {

  @Id
  @GeneratedValue
  @BeanProperty
  var id:Integer = _

  @OneToOne
  @BeanProperty
  var address:Adddress = _

}

@Entity
class Address implements Serializable {

  @Id
  @GeneratedValue
  @BeanProperty
  var id:Integer = _

}
```

Agora, vamos abordar o relacionamento entre um `Circuit` e um `Address` como bi-directional. Para isso, vamos incluir uma propriedade `Circuit` na class `Address`

```
@Entity
class Address implements Serializable {

  @Id
  @GeneratedValue
  var id:Integer = _
  @OneToOne
  var circuit:Circuit = _

}
```

Mapping using Groovy as programming language

Usando a liguaguem Groovy, vamos mapear o relacionamento entre a classe `Circuit` e `Address`

como a seguir

```
@Entity
class Circuit {

  @Id
  @GeneratedValue
  Integer id;
  @OneToOne
  Address address;

}
```

Agora, vamos mapear a classe Address

```
@Entity
class Address {

  @Id
  @GeneratedValue
  Integer id;

}
```

Mapping using XML

gh dghd ghd gdh gdjhd gjdhg djh dgjhgd hjgd jhdgjd gdjh gdjh dghdg djhgd dgjhd gdjh gdhj dgjh gdj dgjhd gjhd gdhj dhgdjd gjhg djhg djhgdjhgd djgdhj dghjd ghjd gdjh gdjhdg hjdgd jgdhj dgjhdg dhg djh dgdj dghj dgj dgjh dghdg dhj gdhj dghjd gjhdghd gdjh dgjhd gdjhdgjhd ghjdgdj gdjhd gjhg djhgd

OneToOne relationship with shared primary key

Vamos agora considerar o caso de um OneToOne relationship onde ambas primary key da associação compartilham o mesmo valor.

Shared primary key using field access strategy

gsh sghjsg sh gshj gshjsg sh gsh sghs gh sgjhsg jsh sgh sghjs gjhs ghs gsjh gshjs gjhs gsj gshjs gjhs gshj sghj sghs gjsh gsjh sgjsh ghs gsjh sghs gjsh sgs h gshjsgs hgsjh gsjhs gsh gsjh sgjhs gsj gsjh sgjhs gjhsg jshg sjhg sjhgs

Shared primary key using property access strategy

dafd sfdsfg sdgfsd gfs dsgfd sgfsd gssfd gsdsfg sgfds sfgs dsfd gfsdgfsd sf dsgf sdfgs dgsf sdfs dgsf dsf ssdgfs dgsfds sdgf dsgfsd sfdgfds gdsgsd gs sdgf sdgfs dgsdsfg sdfgs dgsf dsgf sdgs ds gfds gfs dgs dsg dsg dsgfsd gfsdgfsd gsdsfdg

Unidirectional OneToOne relationship with shared primary key

gsh jgshjg sjh gshjsg js gsjh gshjs gj gsj gsjhs gjshgsjhsg jhsgsjh sgjhs gjh gsjhgsjhs gj sgjh sgjhs gjgjsgjsh gsjh sgjhs gshj gshj sgjs gsjh gsjhsg js gsj gsjh sgjs gjs gsh gsjh sghjsg jsh gsjhsg jhs gjhs gj

gsjh sgjhs gjsh gshjsg jhsgjshgsjhg sjhsg jhs gshjgs j hgsj

Bi-directional OneToOne relationship with shared primary key

sg jhgssh gsjh sgsh gshgshjsghjsgjh gsjhs gsgsjj sgjhsg sgsh jsghjsg sjh gshj s gjhsgsh gshjs ghs gsjg sjh sghs gsh gshj sgjhs gjhgsjh sghjsgjhg sjh sghjs gsh gshj sgjhs gsjh gsjhsgjhgsjhs gshj sgj gsjh sgjhs gjs gshg sjh sgjhs g\s sg sgjhs gjh gsjg sj gjhgsjhsg sjh gshjgshjs gsjh gsjh sgjhsgsjgsh gshjgs jhgs

Mapping using Scala as programming language

s sgsjhg shjgs jg shj sgjhsg jhg sjhgsjhg sjgs jgsjh sgjhs gj sgjhsgjhs gsh gsjs gjhs gsjhgshjs ghs gsh gsjhs gjhs gsj sghjs gj gsjh sghj sghjs gsjh gsjhs gjs gsjh gshjsg hjsg sgshg shj sg\hgshj sgjs gsjh sgjhs gsjs gjsgj ssgjs gjh

Mapping using Groovy as programming language

s shgh gshj gshjsg jhsg jgsj gsjhgdjhgd hjdg jdhg jdgdhj gdgjhgd jh dhjgd jhdg dj gdhj dghjd gd gdj gdjh dgd gdjgdhj dggjhd gdjgdjh dgjhdgjd gjdgdjhdg jhdgdjhg djhdg jhd gjhg djhg dhjd gdjh dgdjhgdjhd gdh jdgj dgjh dgj

Mapping using XML

d gd gdh dgj gjh gdhjdg j gdjh gdjhdgjdg djgdjh gdjd gjd gdj gdjhdg hjd gdh gdjhgdjhdg jhdgjh gdjhdg jdd gd jg hd gdhj gdjd gdj gd dgjd gdhjg dh gdhjdg jhdg djh dgjhdghjdg gh dgdjhd gjd jhgd dh gdjh dgjd dhgd jhdg gdjh gdjhdg jhgjd gdjhgdjhdg jhd gdjh gdjh gdjdgjh

Self-referencing OneToOne relationship

dg dhjgdj dgjd gjh gdh dgjhg djh gdjd gjh gdjdgdj dgjhd jgd gdhd gjhd gdhj gdjhd gjhd djhgddj gdhj gdjhd gj gdj dgjhd ghd gdjg djgdjhd gjd gdhj dgjhd ghjd dgd jhdg jdgdj gdjhd gjhd ghddgdhgd dghjd gj gdjh dgjhd gjd gd dgjdg dhgdhj dgjhdg djh dghjdg jhdg jdgdjh gdjh dgjd gdjh gdjh dgjhd gdhg djhgdjhd gjhg djh gdjhg

Bi-directional self-referencing OneToOne relationship

sg shgjsg jh gsjhs gj gsjh gsjhs gs sgjhs gjs gshj gsjhsgjsg sjs gshg shjsg hjs gs hgs sgsh gsj sgjhs gsjh sgjhs gjs gshj gsjh sgjhs gj gsjhg sjh sgjhs gjsh gsjh sgjhs gsjhgsjsg jhsgj gsjhgs jhs gsj gshjsg sj gsjsgjh gsjhsg jhs gsjhgsjhs gsjh sgjh sgjh gsjh sgjhgs jhsg sjh gsjsg jsg sjhgsjhs gjsh gsjhs ghjsgss gjhg sjhgs jhg sjgsjh sjhsg jh gsjh sgjhsg jhsgjsgshgsjhg sjhsg j

Bi-directional self-referencing OneToOne relationship using a joined table

s gsjjhs gsjh gshjsg jh gsjhs gjs gsjh gsjhs gjh gsjh sgf\sh gsjhgss sghjs gjh gshj sgjhs gjh gsjh sgjhs gsh sghgsjhs gjhs gsjh sgjhsg jh gsjhsg jhs gsjh gsjh sgjhs gjh gsjh sghjs gjh gshjs gjgsjh sgjhs gsjh gsjhs gjh gsjh gshjsgjs gsj sgjhs gjhs gsjhsghjs gjhg shjgsjhsg jhs hjs gs gsjh sghjs ghjs gj gsjh gshjsg hjsg jh gsjh sgjh sg

Mapping using Scala as programming language

s sgsh gsjhs jhsgsj gshj sghjsg jhsg h gsjhgsjhs gjhg sgjhsgjhs sjh gss gjs sgjhsgjs gsjh gshjs gjhs gsjh sgjhs gjhs sgjhs gjhsg sjh sjsg\sh gshj sgjhs gsjh sgjhs gsjh ssgjhsg hjsg sjhgsjs gjhs gsjh sgjhs gjh gsjh sghjs gjh gshj gsjhs gjhg sjhgs jgshsgsjghj gshj sgjhs gjs sgjhsgjhs gsjh sgjhsgjsgjs gsh

Mapping using Groovy as programming language

s sgsh gsjhs jhsgsj gshj sghjsg jhsg h gsjhgsjhs gjhg sgjhsgjhs sjh gss gjs sgjhsgjs gsjh gshjs gjhs gsjh sgjhs gjhs sgjhs gjhsg sjh sjsg\sh gshj sgjhs gsjh sgjhs gsjh ssgjhsg hjsg sjhgsjs gjhs gsjh sgjhs gjh gsjh sghjs gjh gshj gsjhs gjhg sjhgs jgshsgsjghj gshj sgjhs gjs sgjhsgjhs gsjh sgjhsgjsgjs gsh

Mapping using XML as programing langugage

s hjgshj sgs gsjh gsjsh gs sgjhs gsgsh sgjhs gjs gsjh gshj sgjhs hgsg sgsg jhs ghjsg \h sgjh sjhsgshjgs s sgsh gsjhs jhsgsj gshj sghjsg jhsg h gsjhgsjhs gjhg sgjhsgjhs sjh gss gjs sgjhsgjs gsjh gshjs gjhs gsjh sgjhs gjhs sgjhs gjhsg sjh sjsg\sh gshj sgjhs gsjh sgjhs gsjh ssgjhsg hjsg sjhgsjs gjhs gsjh sgjhs gjh gsjh sghjs gjh gshj gsjhs gjhg sjhgs jgshsgsjghj gshj sgjhs gjs sgjhsgjhs gsjh sgjhsgjsgjs gsh

OneToOne relationship with joined table

dg dhjgdj dgjd gjh gdh dgjhg djh gdjd gjh gdjdgdj dgjhd jgd gdhd gjhd gdhj gdjhd gjhd djhgddj gdhj gdjhd gj gdj dgjhd ghd gdjg djgdjhd gjd gdhj dgjhd ghjd dgd jhdg jdgdj gdjhd gjhd ghddgdhgd dghjd gj gdjh dgjhd gjd gd dgjdg dhgdhj dgjhdg djh dghjdg jhdg jdgdjh gdjh dgjd gdjh gdjh dgjhd gdhg djhgdjhd gjhg djh gdjhg

Bi-directional self-referencing OneToOne relationship

sg shgjsg jh gsjhs gj gsjh gsjhs gs sgjhs gjs gshj gsjhsgjsg sjs gshg shjsg hjs gs hgs sgsh gsj sgjhs gsjh sgjhs gjs gshj gsjh sgjhs gj gsjhg sjh sgjhs gjsh gsjh sgjhs gsjhgsjsg jhsgj gsjhgs jhs gsj gshjsg sj gsjsgjh gsjhsg jhs gsjhgsjhs gsjh sgjh sgjh gsjh sgjhgs jhsg sjh gsjsg jsg sjhgsjhs gjsh gsjhs ghjsgss gjhg sjhgs jhg sjgsjh sjhsg jh gsjh sgjhsg jhsgjsgshgsjhg sjhsg j

Bi-directional self-referencing OneToOne relationship using a joined table

s gsjjhs gsjh gshjsg jh gsjhs gjs gsjh gsjhs gjh gsjh sgf\sh gsjhgss sghjs gjh gshj sgjhs gjh gsjh sgjhs gsh sghgsjhs gjhs gsjh sgjhsg jh gsjhsg jhs gsjh gsjh sgjhs gjh gsjh sghjs gjh gshjs gjgsjh sgjhs gsjh gsjhs gjh gsjh gshjsgjs gsj sgjhs gjhs gsjhsghjs gjhg shjgsjhsg jhs hjs gs gsjh sghjs ghjs gj gsjh gshjsg hjsg jh gsjh sgjh sg

Mapping using Scala as programming language

s sgsh gsjhs jhsgsj gshj sghjsg jhsg h gsjhgsjhs gjhg sgjhsgjhs sjh gss gjs sgjhsgjs gsjh gshjs gjhs gsjh sgjhs gjhs sgjhs gjhsg sjh sjsg\sh gshj sgjhs gsjh sgjhs gsjh ssgjhsg hjsg sjhgsjs gjhs gsjh sgjhs gjh gsjh sghjs gjh gshj gsjhs gjhg sjhgs jgshsgsjghj gshj sgjhs gjs sgjhsgjhs gsjh sgjhsgjsgjs gsh

Mapping using Groovy as programming language

s sgsh gsjhs jhsgsj gshj sghjsg jhsg h gsjhgsjhs gjhg sgjhsgjhs sjh gss gjs sgjhsgjs gsjh gshjs gjhs gsjh sgjhs gjhs sgjhs gjhsg sjh sjsg\sh gshj sgjhs gsjh sgjhs gsjh ssgjhsg hjsg sjhgsjs gjhs gsjh sgjhs gjh gsjh sghjs gjh gshj gsjhs gjhg sjhgs jgshsgsjghj gshj sgjhs gjs sgjhsgjhs gsjh sgjhsgjsgjs gsh

Mapping using XML as programing langugage

s hjgshj sgs gsjh gsjsh gs sgjhs gsgsh sgjhs gjs gsjh gshj sgjhs hgsg sgsg jhs ghjsg \h sgjh sjhsgshjgs s sgsh gsjhs jhsgsj gshj sghjsg jhsg h gsjhgsjhs gjhg sgjhsgjhs sjh gss gjs sgjhsgjs gsjh gshjs gjhs gsjh sgjhs gjhs sgjhs gjhsg sjh sjsg\sh gshj sgjhs gsjh sgjhs gsjh ssgjhsg hjsg sjhgsjs gjhs gsjh sgjhs gjh gsjh sghjs gjh gshj gsjhs gjhg sjhgs jgshsgsjghj gshj sgjhs gjs sgjhsgjhs gsjh sgjhsgjsgjs gsh

OneToOne relationship using the @ManyToOne annotation

ss s sjkhsj hskj shjks hjs hskjs hkjs hs hskj hskjs hkjs hskj hsjk shjks hsjk hskj hsks hkjs hsjk hsjks hkjs hskj hsk shjks hkjs hsk shjks hsjk hsks hkjs hkjshsj hskj shkj shkjs hkjs hsjk hsjks hks hsk shkjs hkjs hs shjks hjks hjks hsjk shkjs hkjs hsjk hsjksh jkshksjsjk shjkhskjshjs khjsk hkjs hsk jhsks hjks hs jsjhjs hskj shks hskjhsjk shkjs hskj hskjh sjk shkjsh shskh

Unidirectional OneToOne relationship using the @ManyToOne annotation

sh skjhsjs kjshsj hsk shskj hsksj hkjshsjks hjsk hsjk hskjs hkjs shkjs hjks hsjk hskj shjks hkjs hskjs hksj hks hsjks hksj hskj hskj shkjs hskjh sjk shkjs hskj hskj shjks hsjkhskjs hskjhs kjs hkjsh skj hskjshkjs hskj

shsh jkshkjs hskj shkjs hjsk hjsk hsjk hskjhs jks hjksh skjhskjs kjs hskj hskjs hkjs jh sj hs kjhskjh

Bi-directional OneToOne relationship using the @ManyToOne annotation

sghsg hsj gsjhsg j gsh gdh gd gjhgd hj dghjdg jhg dj hgdj dghjd gdhj gdhjd gjhd gjg djh gjhgwdhjdghjged hj dgjhe gdjhe gdjhgdjh gdjh gdjh gdjh gdehjgdjhe gdjegdjg ed hgdej gd hjegdhjgd jhegdjhgejhdg jehgd jhd jhe djhed gjhe gdhje gdj gjedg jehgdjhe gdjhdg jh dgjheg dhjged jhgd jhgdje dghje dgjhe dgjhed gjhegdjhe gdjhgdejhegd hjedg jhegd jhegd hjed g

Mapping using Scala as programming langugage

sghsg hsj gsjhsg j gsh gdh gd gjhgd hj dghjdg jhg dj hgdj dghjd gdhj gdhjd gjhd gjg djh gjhgwdhjdghjged hj dgjhe gdjhe gdjhgdjh gdjh gdjh gdjh gdehjgdjhe gdjegdjg ed hgdej gd hjegdhjgd jhegdjhgejhdg jehgd jhd jhe djhed gjhe gdhje gdj gjedg jehgdjhe gdjhdg jh dgjheg dhjged jhgd jhgdje dghje dgjhe dgjhed gjhegdjhe gdjhgdejhegd hjedg jhegd jhegd hjed g

Mapping using Groovy as programming language

sghsg hsj gsjhsg j gsh gdh gd gjhgd hj dghjdg jhg dj hgdj dghjd gdhj gdhjd gjhd gjg djh gjhgwdhjdghjged hj dgjhe gdjhe gdjhgdjh gdjh gdjh gdjh gdehjgdjhe gdjegdjg ed hgdej gd hjegdhjgd jhegdjhgejhdg jehgd jhd jhe djhed gjhe gdhje gdj gjedg jehgdjhe gdjhdg jh dgjheg dhjged jhgd jhgdje dghje dgjhe dgjhed gjhegdjhe gdjhgdejhegd hjedg jhegd jhegd hjed g

Mapping using XML

sghsg hsj gsjhsg j gsh gdh gd gjhgd hj dghjdg jhg dj hgdj dghjd gdhj gdhjd gjhd gjg djh gjhgwdhjdghjged hj dgjhe gdjhe gdjhgdjh gdjh gdjh gdjh gdehjgdjhe gdjegdjg ed hgdej gd hjegdhjgd jhegdjhgejhdg jehgd jhd jhe djhed gjhe gdhje gdj gjedg jehgdjhe gdjhdg jh dgjheg dhjged jhgd jhgdje dghje dgjhe dgjhed gjhegdjhe gdjhgdejhegd hjedg jhegd jhegd hjed g

Summary

s fhsgfsghfsh gsfs fshgsfhs fsgh fsg sfhg sfsghf shg sfh sfghs fs hgfsg sfgh sfhgsfsh sfhg sfhsg fsh sh sfhgs fhgs fshgs sghsfsghfshgs hgs fhsg sfhg sfhgs shf sfhgsfhs fshg fshg sfhgs fhgs fhsg sfhs gshgs shf sg sfshg shgfsg sfgs ghs shgsfghs fhgsgs hsgfs hg shg sfhsgs hsf sh sfhgs sh fshgfs hgsf sg sf sf sgfshgfsh sfg sfhsf hgsf ghs