Hibernate

sommaire

- principes de l'« object-relationnal mapping »
  - correspondances objet / relationnel
  - vocabulaire
- Java et les bases de données
- Hibernate
  - présentation
  - déploiement
  - implémentations
  - mapping par annotations
  - gestion transactionnelle

principes d’« object-relationnal mapping »

principes d'« object-relationnal mapping »

> établir une correspondance entre monde objet et monde relationnel

> permettre la persistance durable des objets

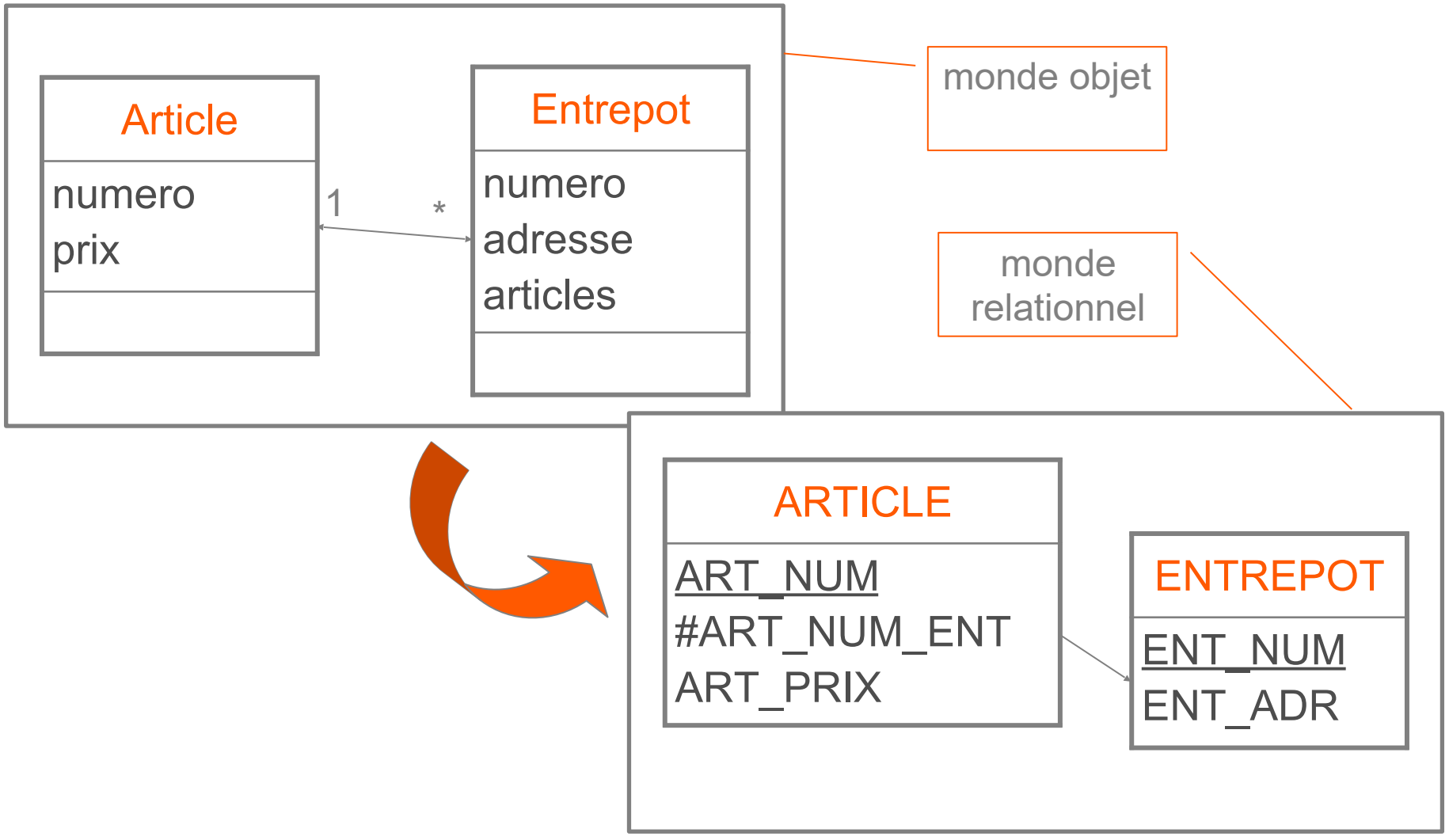
> masquer l'utilisation d'une base de données relationnelle
- diminuer la complexité
- permettre la réutilisation

> permettre la recherche d'informations

correspondance objet / relationnel

objet	relationnel
classe	table
attribut	colonne
instance	enregistrement
association	clé étrangère

correspondance objet / relationnel



correspondance objet / relationnel

> correspondance des associations
- n à 1 ➔ attribut simple
- 1 à n ➔ collection
- n à n ➔ collection

```
public class Article {
 private String numero;
 private Double prix;
 private Entrepot entrepot;
}
```

```
public class Entrepot {
 private String numero;
 private String adresse;
 private Set<Article> stock;
}
```

vocabulaire

> persistance
  - mécanisme de sauvegarde et de restauration des données

> classe persistante
  - classe dont les instances peuvent être rendues persistantes

> instance persistante
  - instance d'une classe persistante dont les données sont présentes dans la base de données

> instance transiente
  - instance d'une classe persistante dont les données sont absentes de la base de données

Java et les bases de données

Java et les bases de données

> API standard : JDBC
  - Java Database Connectivity
  - couche de bas niveau pour la connexion aux SGBD relationnels

> établit des connexions de manière générique
  - un Driver spécifique au SGBD doit être installé dans le classpath

> permet l'envoi de requêtes et la lecture des résultats
  - les requêtes SQL sont envoyées sous forme de String

> maîtrise nécessaire du SQL

> résultats renvoyés sous forme de liste d'enregistrements
  - les données doivent être transformées pour être utilisables dans l'application

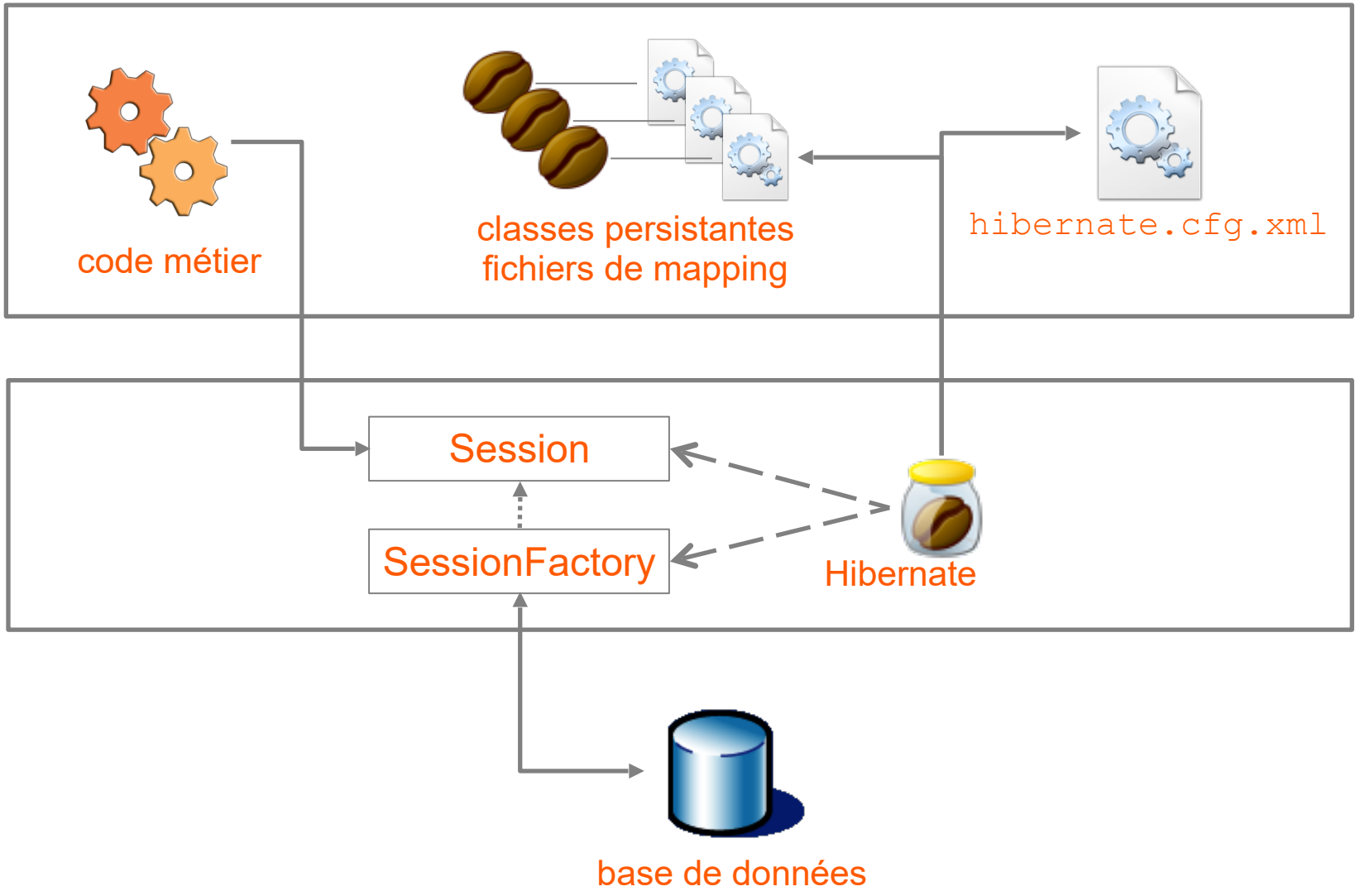
Hibernate

Hibernate

> Hibernate est un framework de mapping objet-relationnel
- couche supplémentaire au dessus de JDBC
- aucun SQL à produire
- génère dynamiquement les requêtes SQL
- aucune interaction directe avec la base de données
- l'application manipule uniquement des objets

> L'interaction avec la base est faite à travers une « Session »
- contrôle la connexion à la base
- fournit des méthodes d'accès aux données
  - lecture d'une instance
  - recherche
  - création et enregistrement
  - suppression

architecture d'un projet Hibernate



composants d'un projet Hibernate

> la bibliothèque Hibernate
  - un fichier JAR dans le classpath
    - (`WEB-INF/lib/ pour les webapps`)
  - les dépendances

> un fichier de configuration principal (`hibernate.cfg.xml`)
  - paramètres de connexion à la base
  - inclusion des fichiers de mapping

> un fichier de configuration pour chaque classe persistante
  - correspondance entre la table et la classe
  - relations avec les autres classes persistantes

configuration du framework

Fichier hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="factory">
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.username">admin</property>
        <property name="hibernate.connection.password">admin</property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/stocks
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQL5InnoDBDialect
        </property>

        <mapping resource="com/company/stock/model/Article.hbm.xml"/>
        <mapping resource="com/company/stock/model/Entrepot.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

déclaration du mapping

Fichier Article.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.company.stock.model">
  <class name="Article" table="ARTICLE">
    <id name="numero" column="ART_NUM">
      <generator class="assigned" />
    </id>
    <property name="prix" column="ART_PRIX" />
    …
</class>
</hibernate-mapping>
```

déclaration du mapping

Fichier Entrepot.hbm.xml

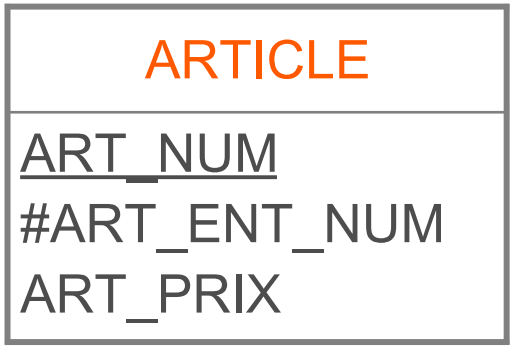
```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.company.stock.model">
  <class name="Entrepot">
    <id name="numero" column="ENT_NUM">
      <generator class="assigned" />
    </id>
    <property name="adresse" column="ENT_ADR" />
    …
</class>
</hibernate-mapping>
```

détail du mapping

```
<class name="Article" table="ARTICLE">
    <id name="numero" column="ART_NUM">
      <generator class="assigned" />
    </id>
    <property name="prix" column="ART_PRIX" />
  …
</class>
```

> id
  - clé primaire de la table
  - attribut correspondant dans la classe
  - méthode de génération de l'identifiant
    - assigné par l'application
    - généré automatiquement par la base
    - etc.

> property
  - colonne de la table (sauf clé étrangère)
  - attribut correspondant dans la classe



identifiant généré

> La clé primaire est un entier généré par Hibernate
  - Utilisation d'un générateur prédéfini
    - increment : la clé est incrémentée par Hibernate
    - identity : clé primaire auto-incrémentée par la base
    - sequence : la clé primaire est alimentée à partir d'une séquence

```
<class name="Article" table="ARTICLE">
    <id name="id" column="ART_ID">
      <generator class="identity" />
    </id>
    <property name="numero" column="ART_NUM" />
    <property name="prix" column="ART_PRIX" />
  …
</class>
```

ARTICLE
ART_ID
ART_NUM
#ART_ENT_NUM
ART_PRIX

obtenir une Session

- on obtient une Session à l'aide de la SessionFactory
- SessionFactory
  - lente à construire
    - une seule fois au lancement de l'application
  - lit et vérifie la configuration du framework
  - se connecte à la base
  - une fois construite, fournit des Sessions à la demande
- toujours refermer la Session

```
Configuration cfg = new Configuration().configure();

SessionFactory factory = cfg.buildSessionFactory();
// [...]
Session session = factory.openSession();
// [...]
session.close();
```

requêtage par identifiant

> Méthode Session.get()
  - 1er argument = type de l'entité à charger
  - 2ème argument = valeur de l'identifiant (clé primaire)

```
Article art = (Article) session.get(Article.class, "AZ123456");
```

La méthode get() retourne un seul objet

requêtage HQL

```
Query q = session.createQuery("from Article where
   prix = :p");
q.setDouble("p", 50D);
List<Article> resultats = q.list();
```

> expression d'une requête sur une classe
  - définition de paramètres nommés
  - le mot clé `select` est facultatif

mapping des associations

mapping des associations

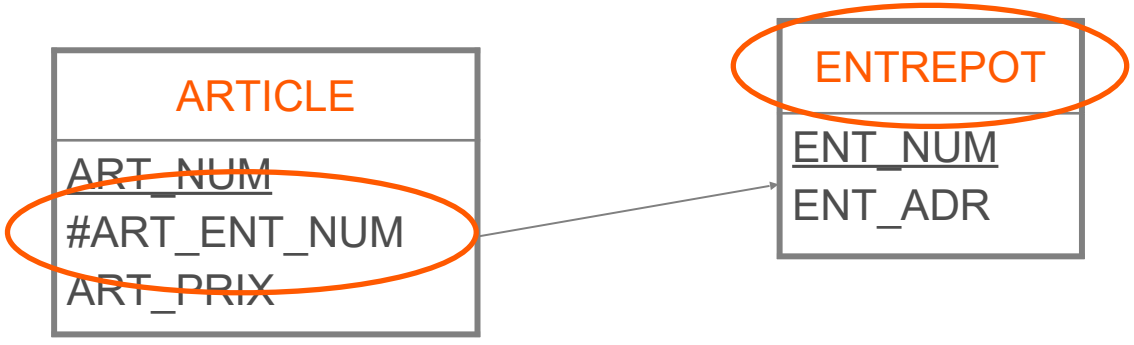
> plusieurs à 1
  - un attribut seul

> 1 à plusieurs
  - une collection reçoit les instances associées

> plusieurs à plusieurs
  - une collection à chaque bout de l'association

mapping « plusieurs à 1 »

> many-to-one
- attribut correspondant dans la classe mappée
- colonne qui porte la clé étrangère
- classe associée




```
<class name="Article" table="ARTICLE">
  <id name="numero" column="ART_NUM">
    <generator class="assigned" />
  </id>
  <property name="prix" column="ART_PRIX" />
  <many-to-one name="entrepot" column="ART_ENT_NUM"
               class=« com.company.stock.model.Entrepot"/>
 </class>
```

mapping « 1 à plusieurs »

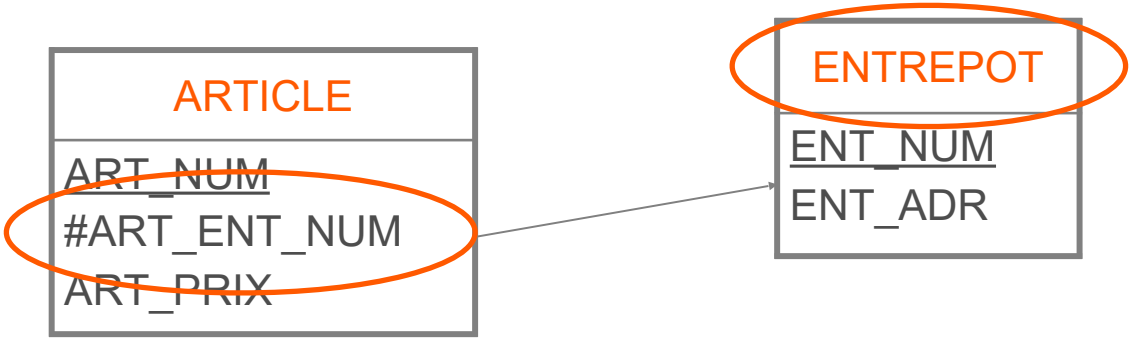
> plusieurs sémantiques d'association
  - ensemble (Set) : les instances associées sont retournées sans ordre particulier
  - liste (List) : les instances sont retournées dans un ordre défini par un attribut donné (index dans la liste)
  - dictionnaire (Map) : les instances sont retournées dans une Map ; un attribut sert de clé, l'instance de valeur

> set / list / map
  - nom de l'attribut dans la classe mappée
  - key
    - nom de la colonne dans la table associée vers la table mappée
  - one-to-many
    - nom de la classe associée

mapping « 1 à plusieurs »

> Exemple

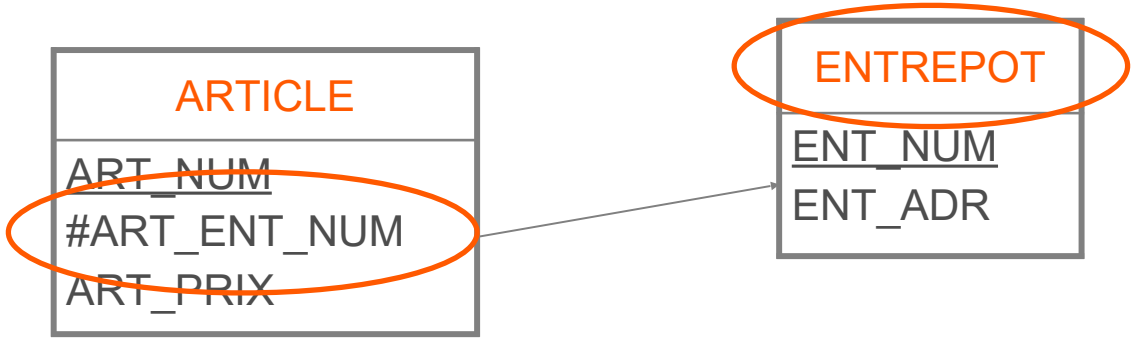
```
<class name="Entrepot">
   <id name="numero" column="ENT_NUM">
     <generator class="assigned" />
   </id>
   <property name="adresse" column="ENT_ADR" />
   <set name="stock">
     <key column="ART_ENT_NUM"/>
     <one-to-many class="com.company.stock.model.Article"/>
   </set>
 </class>
```



mapping « 1 à plusieurs » bidirectionnel

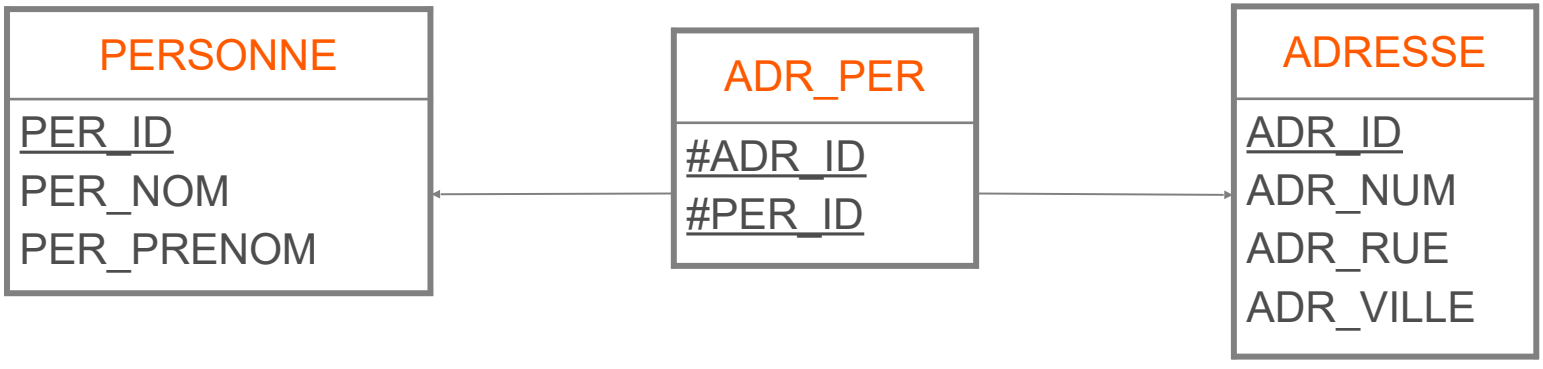
> La collection doit être marquée avec « inverse = ‘’true’’ »

```
<class name="Entrepot">
   <id name="numero" column="ENT_NUM">
     <generator class="assigned"/>
   </id>
   <property name="adresse" column="ENT_ADR" />
   <set name="stock" inverse="true" >
     <key column="ART_ENT_NUM"/>
     <one-to-many class="com.company.stock.model.Article"/>
   </set>
 </class>
```



mapping « plusieurs à plusieurs »

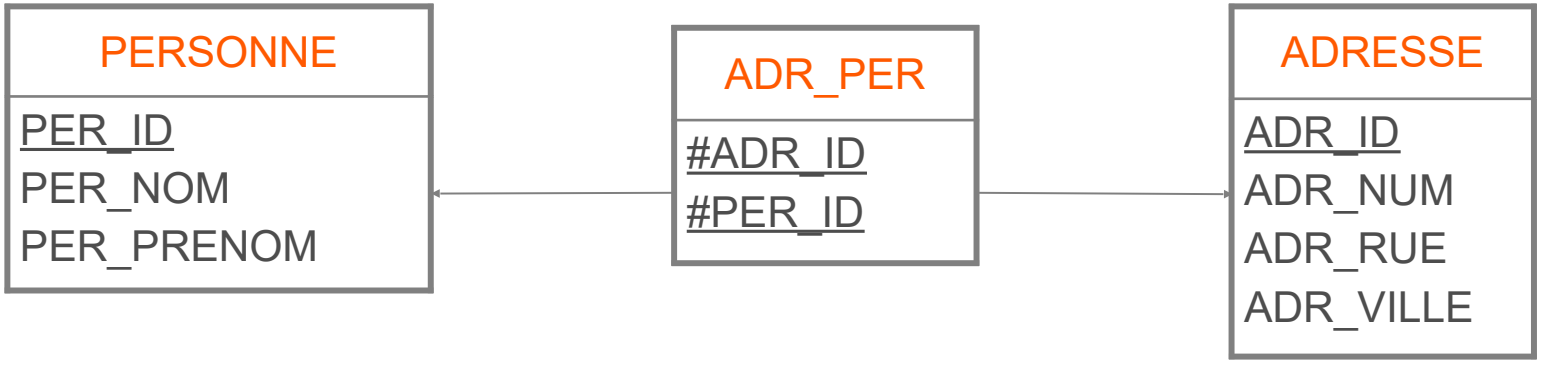
> sémantiques identiques à « 1 à plusieurs »
> une table d'association entre les tables mappées
> set / list / map
  - nom de la table d'association
  - key
    - nom de la colonne dans la table d'association vers la table mappée
  - many-to-many
    - nom de la classe associée
    - nom de la colonne vers la table de la classe associée



mapping « plusieurs à plusieurs »

> Exemple

```
<class name="Personne">
   <id name="id" column="PER_ID">
     <generator class="identity" />
   </id>
   <property name="nom" column="PER_NOM" />
   <property name="prenom" column="PER_PRENOM" />

   <set name="adresses" table="ADR_PER" >
     <key column="PER_ID"/>
     <many-to-many class="com.company.stock.model.Adresse"
                   column="ADR_ID"/>
   </set>
 </class>
```



mapping de l’héritage

mapping de l'héritage

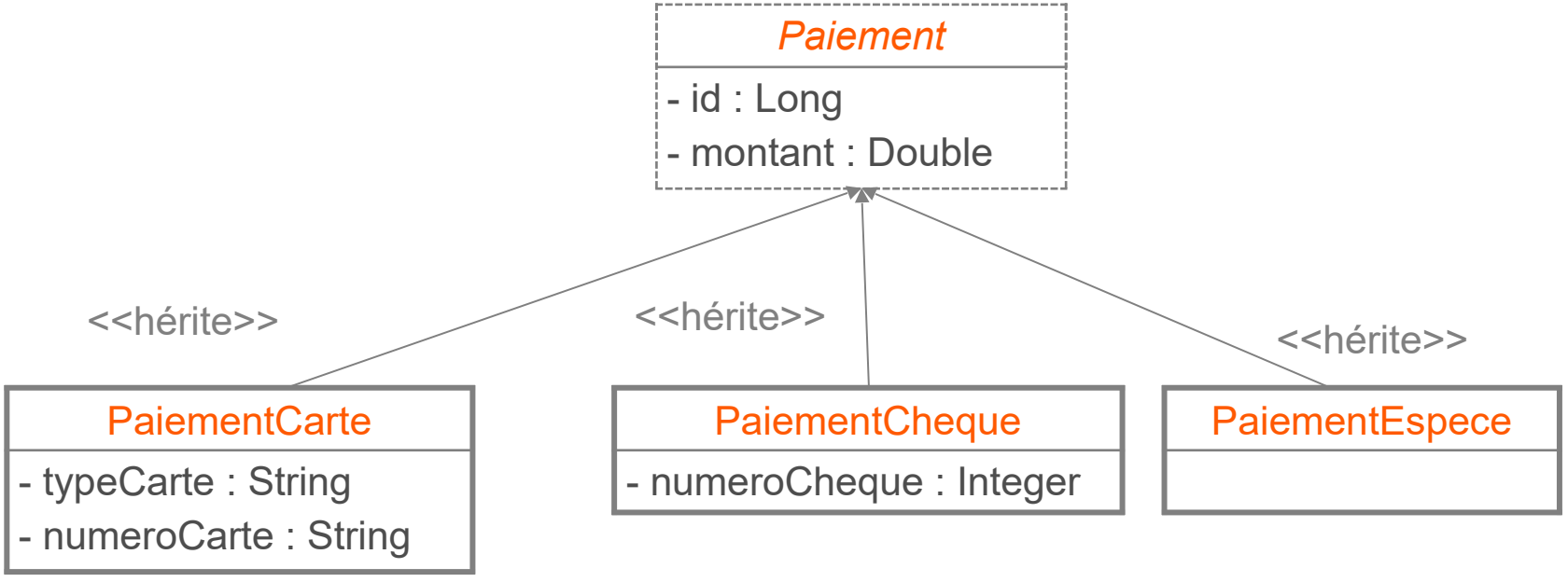
> 3 stratégies :
  - Une table par hiérarchie de classe
  - Une table par classe fille,
  - Une table par classe concrète.

> Possibilité de mixer les 3 stratégies

> Le choix de l'implémentation dépend des cas d'utilisation

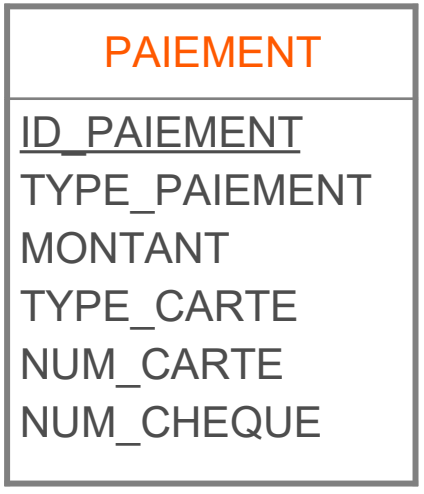
> Permet d'utiliser le polymorphisme pour sélectionner les objets
  - Lister les paiements d'un montant supérieur à 200 €
  - Lister les paiements par cash de plus de 500 €

mapping de l'héritage



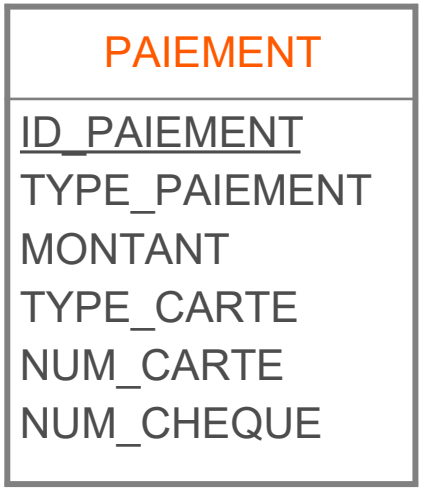
mapping : une table par hiérarchie de classe

> Toute la hiérarchie de classes est persistée dans une table unique
> Une colonne sert de discriminant
> Limitation
  - les colonnes des classes filles ne peuvent avoir de contrainte not null



mapping : une table par hiérarchie de classe

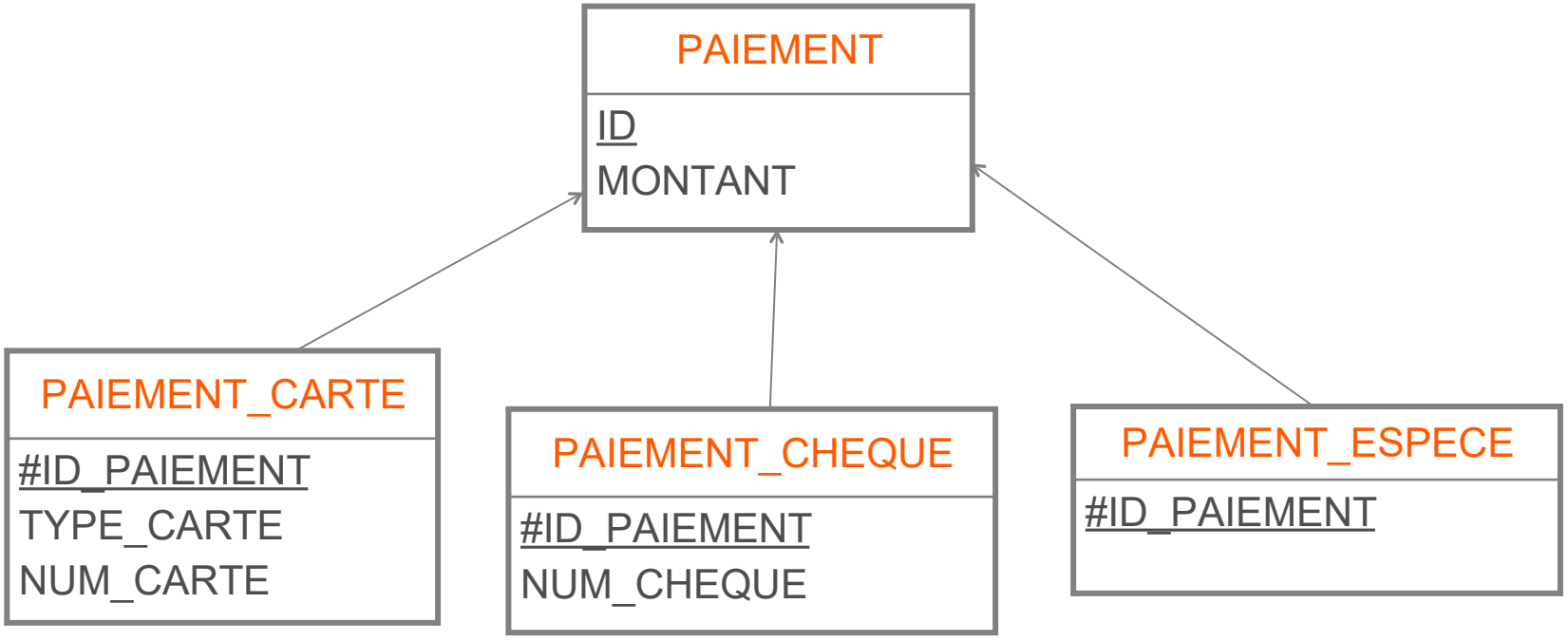
> Fichier de mapping




```
<class name="Paiement" table="PAIEMENT">
  <id name="id" type="long" column="ID_PAIEMENT">
    <generator class="identity"/>
  </id>
  <discriminator column="TYPE_PAIEMENT" type="string"/>
  <property name="montant" column="MONTANT"/>
  ...
  <subclass name="PaiementCarte"
            discriminator-value="CREDIT">
    <property name="typeCarte" column="TYPE_CARTE"/>
    ...
  </subclass>
  <subclass name="PaiementEspece"
            discriminator-value="ESPECE">
    ...
  </subclass>
  <subclass name="PaiementCheque"
            discriminator-value="CHEQUE">
    ...
  </subclass>
</class>
```

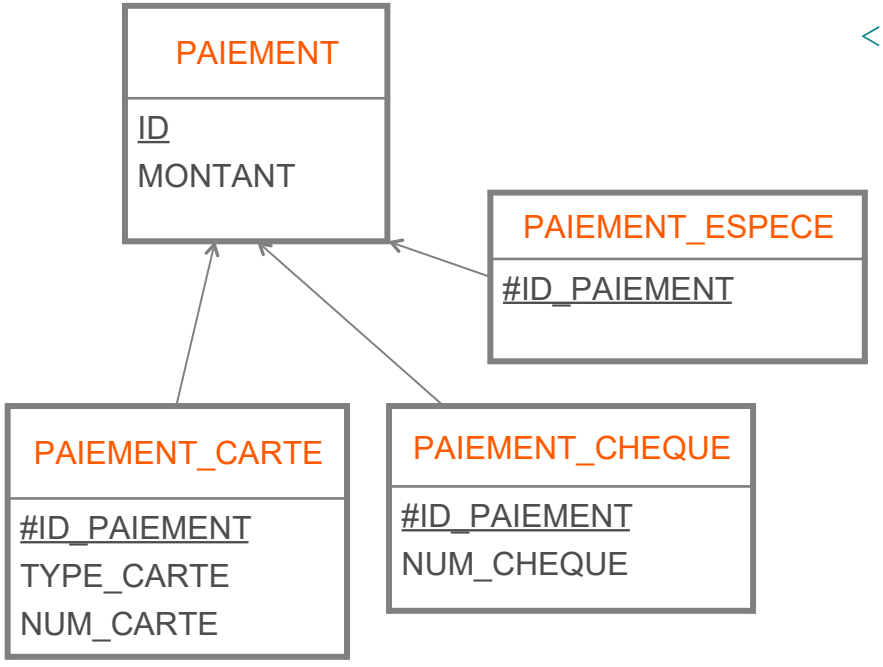
mapping : une table par classe fille

> Une table chaque classe de la hiérarchie
> Les clés primaires des tables filles référencent la table mère



mapping : une table par classe fille

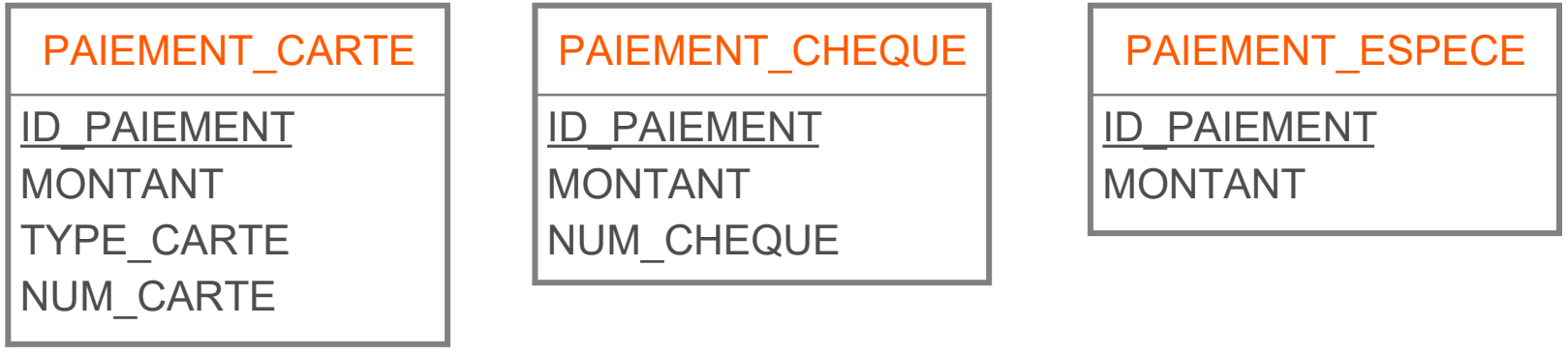
> Fichier de mapping




```
<class name="Paiement" table="PAIEMENT">
  <id name="id" column="ID">
    <generator class="identity"/>
  </id>
  <property name="montant" column="MONTANT"/>
  …
  <joined-subclass name="PaiementCarte"
                   table="PAIEMENT_CARTE">
    <key column="ID_PAIEMENT"/>
    <property name="typeCarte" column="TYPE_CARTE"/>
    …
  </joined-subclass>
  <joined-subclass name="PaiementEspece"
                   table="PAIEMENT_ESPECE">
    <key column="ID_PAIEMENT"/>
    …
  </joined-subclass>
  <joined-subclass name="PaiementCheque"
                   table="PAIEMENT_CHEQUE">
    <key column="ID_PAIEMENT"/>
    …
  </joined-subclass>
</class>
```

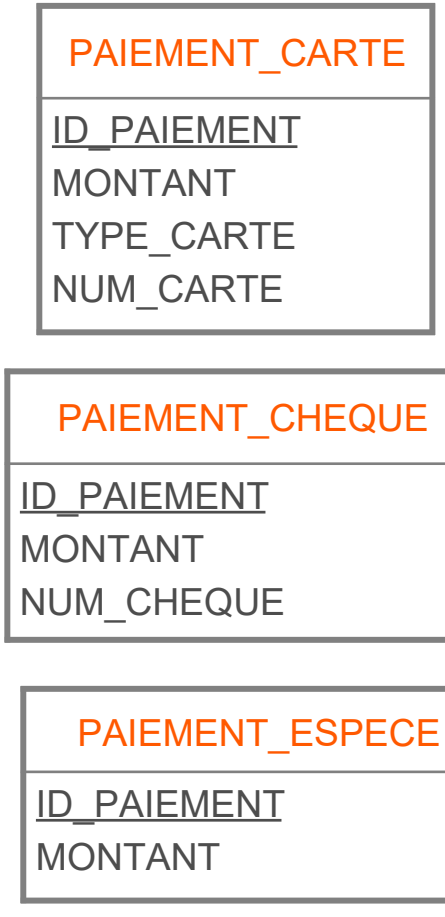
mapping : une table par classe concrète

> Autant de tables que de classes filles
> Les données de la classe mère sont contenues dans la table des classes filles
> Limitations :
  - Même nom de colonnes dans toutes les tables pour les propriétés héritées



mapping : une table par classe concrète

> Fichier de mapping




```
<class name="Paiement">
  <id name="id" type="long" column="ID_PAIEMENT">
    <generator class="identity"/>
  </id>
  <property name="montant" column="MONTANT"/>
  ...
  <union-subclass name="PaiementCarteCredit"
                  table="PAIEMENT_CREDIT">
    <property name="typeCarte"
              column="TYPE_CARTE"/>
    ...
  </union-subclass>
  <union-subclass name="PaiementEspece"
                  table="PAIEMENT_CASH">
    ...
  </union-subclass>
  <union-subclass name="PaiementCheque"
                  table="PAIEMENT_CHEQUE">
    ...
  </union-subclass>
</class>
```

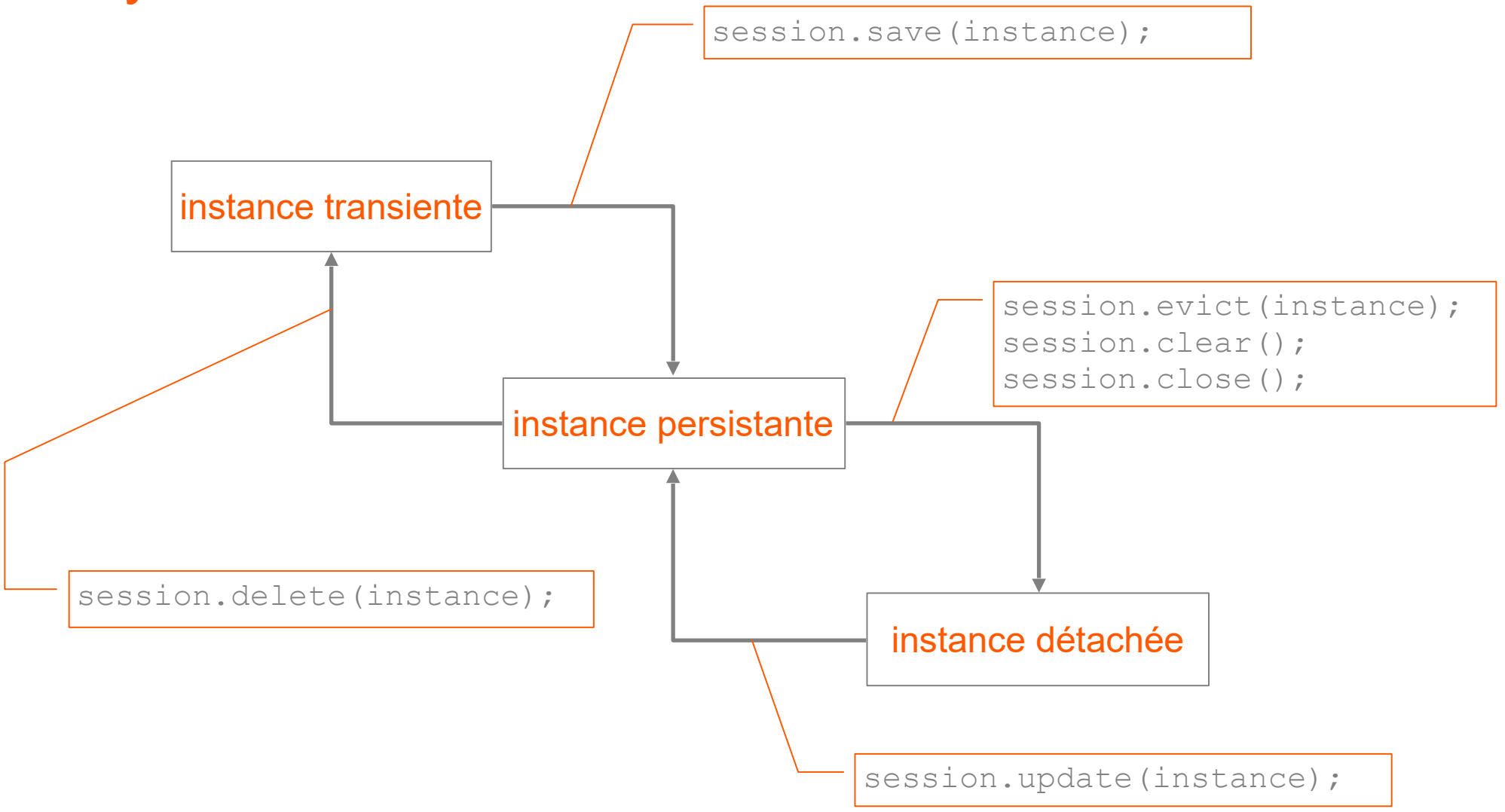
manipuler les objets persistants

état des objets

3 états possibles d'une instance de classe persistante

- transiente
  - objet nouvellement instancié
    - pas encore de données en base. Si le programme s'arrête, l'objet est perdu
  - objet supprimé en base
    - données en base supprimées, mais l'instance existe toujours
- détachée
  - objet persistant dont la Session a été fermée
- persistante
  - objet lu depuis la base
  - objet transient qui a été enregistré
  - objet détaché qui a été rattaché à une nouvelle Session

cycle de vie


session.save(instance);
instance transiente
session.evict(instance);
session.clear();
session.close();
instance persistante
session.delete(instance);
instance détachée
session.update(instance);

utiliser la Session

> charger une instance persistante
- `Object Session.get(Class clazz, Serializable id)`
  - charge l'instance de la classe `clazz` portant l'identifiant `id`
  - si aucun enregistrement ne porte cet identifiant, retourne `null`
- `Object Session.load(Class clazz, Serializable id)`
  - identique à `get`
  - lance une `HibernateException` plutôt que de renvoyer `null`

> rendre une instance persistante
- `void Session.save(Object instance)`
  - si l'instance est volatile : insertion
  - si l'instance est déjà persistante : mise à jour
  - inutile d'appeler `save` sur une instance déjà persistante, sauf cas exceptionnels

utiliser la Session

> rendre une instance transiente
  - `void Session.delete(Object instance)`
    - effectue un `delete` en base
    - l'objet Java n'est pas supprimé

> détacher une instance
  - `void Session.evict(Object instance)`
    - l'instance persistante est détachée de la Session
    - les modifications à l'instance ne seront plus sauvegardées

> rattacher une instance
  - `void Session.update(Object instance)`
    - l'instance est rattachée à la Session
    - l'instance est synchronisée avec la base de données
    - les modifications seront sauvegardées en base

requêtage

> rechercher des objets dans la base de données

- API Criteria
  - construction programmatique de requêtes
- langage HQL
  - transposition de SQL dans le monde objet
  - requêtage sur les classes, leurs attributs et leurs relations
- langage SQL natif
  - alternative à HQL
  - permet de bénéficier des spécificités du SGBD

requêtage par Criteria

```
Criteria critArt =
   session.createCriteria(Article.class);
critArt.add(Restrictions.eq("prix", 50D));
List<Article> resultats = critArt.list();
```

> construction dynamique d'une requête

- spécifier la classe des instances recherchées
- ajouter des restrictions
- voir la classe `org.hibernate.criterion.Restrictions` pour les critères disponibles

requêtage par HQL

```
Query q = session.createQuery("from Article where
   prix = :p");
q.setDouble("p", 50D);
List<Article> resultats = q.list();
```

> expression d'une requête sur une classe
  - définition de paramètres nommés
  - le mot clé `select` est facultatif

requêtage par SQL natif

```
SQLQuery q = session.createSQLQuery("select {a.*}
   from ARTICLE {a} where ART_PRIX = :p");
q.addEntity("a", Article.class);
q.setDouble("p", 50D);
List<Article> resultats = q.list();
```

> expression d'une requête sur une ou plusieurs tables
- retourne tout de même des objets
- préciser des alias pour chaque classe
- définition de paramètres nommés

configuration par annotations

principe de la configuration par annotations

> Les fichiers de mapping « .hbm.xml » sont remplacés par l'utilisation d'annotations dans les classes persistantes
  - @Entity
  - @Id
  - @Column
  - …
> Le fichier « hibernate.cfg.xml » ne déclare donc plus des configurations de mapping de type « resource », mais de type « class »

```
<hibernate-configuration>
  <session-factory>
    …
    <mapping class="com.company.stock.model.Article"/>
    <mapping class="com.company.stock.model.Entrepot"/>
        …
  </session-factory>
</hibernate-configuration>
```

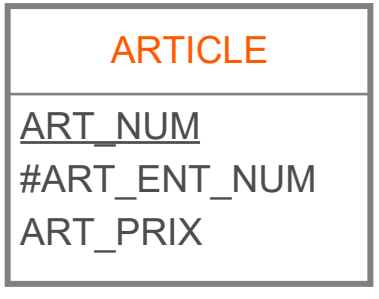
principe de la configuration par annotations

> Utilisation d'une extension de la classe Configuration
  - Classe AnnotationConfiguration
  - Permet d'interpréter les méta-données de mapping sous la forme d'annotations

```
Configuration cfg = new AnnotationConfiguration().configure();

SessionFactory sf = cfg.buildSessionFactory();
Session session = sf.openSession();
…
```

mapping d'une classe et ses propriétés

> Mapping d'une classe
  - @Entity
  - @Table
> Mapping des propriétés
  - @Id
  - @Column




```java
@Entity @Table(name="ARTICLE")
public class Article {

  @Id @Column(name="ART_NUM")
  private String numero;

  @Column(name="ART_PRIX")
  private double prix;

  // getters et setters
  …
```

identifiant généré

> L'annotation @GeneratedValue permet de spécifier une stratégie de génération de l'identifiant

```
@Entity @Table(name="ARTICLE")
public class Article {
  @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="ART_ID")
  private Long id;

  @Column(name="ART_NUM")
  private String numero;
  …
```

ARTICLE
ART_ID
ART_NUM
#ART_ENT_NUM
ART_PRIX

mapping d'une propriété temporelle

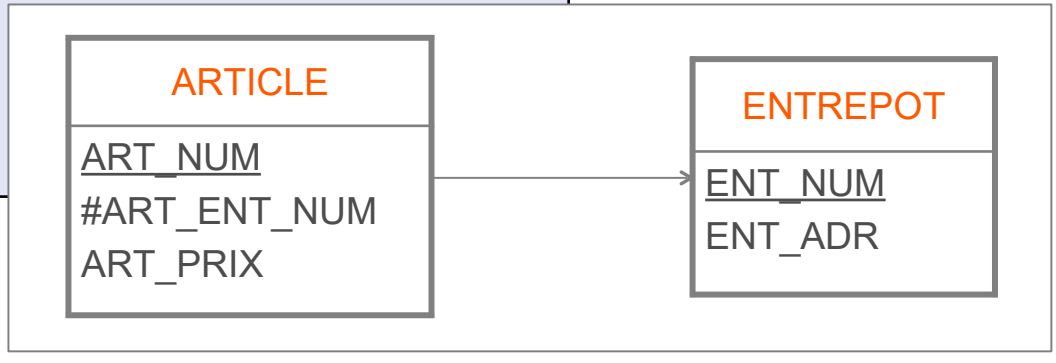
> Le mapping doit indiquer le type de la propriété temporelle à l'aide de l'annotation @Temporal
  - DATE
  - TIME
  - TIMESTAMP

```
@Entity @Table(name="ARTICLE")
public class Article {

  @Id @Column(name="ART_NUM")
  private String numero;

  @Column(name="ART_DATE_CREATION")
  @Temporal(TemporalType.DATE)
  private Date dateCreation;

  // getters et setters
  …
```

association many-to-one

> @ManyToOne définit le type de l'association
> @JoinColumn permet de préciser la foreign key

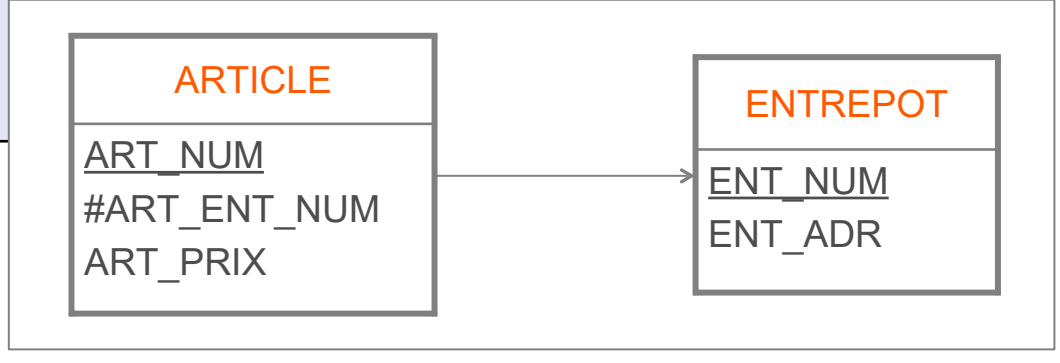
```
@Entity @Table(name="ARTICLE")
public class Article {
  @Id @Column(name="ART_NUM")
  private String numero;

  @Column(name="ART_PRIX")
  private double prix;

  @ManyToOne @JoinColumn(name="ART_ENT_NUM")
  private Entrepot entrepot;

  // getters et setters
  …
```



association one-to-many

> @OneToMany définit le type de l'association
> @JoinColumn permet de préciser la foreign key

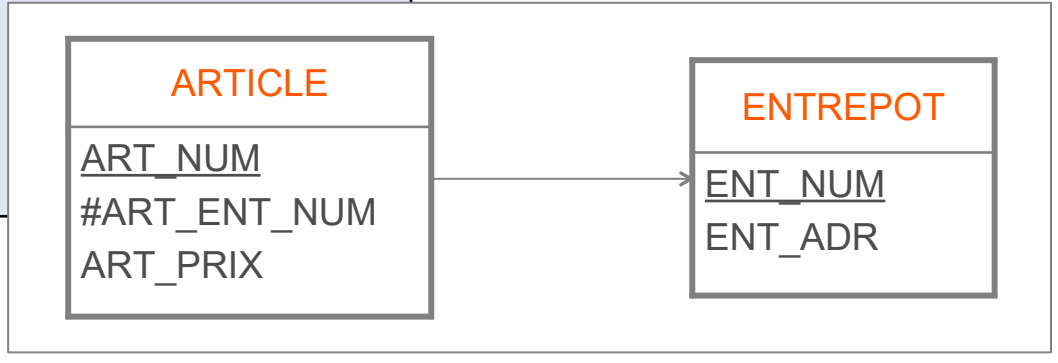
```
@Entity @Table(name="ENTREPOT")
public class Entrepot {
  @Id @Column(name="ENT_NUM")
  private String numero;

  @Column(name="ENT_ADR")
  private String adresse;

  @OneToMany @JoinColumn(name="ART_ENT_NUM")
  private Set<Article> stock;

  // getters et setters
  …
```



association many-to-one bidirectionnelle

> La foreign key n'est renseignée que d'un côté avec @JoinColumn
> L'attribut « mappedBy » permet à Hibernate de retrouver la foreign key sur le côté non renseigné

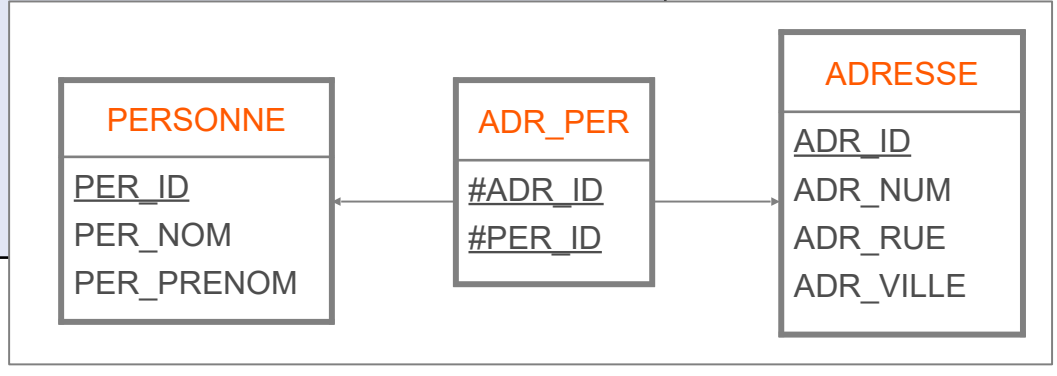
```
@Entity @Table(name="ENTREPOT")
public class Entrepot {
  @Id @Column(name="ENT_NUM")
  private String numero;

  @Column(name="ENT_ADR")
  private String adresse;

  @OneToMany(mappedBy="entrepot")
  private Set<Article> stock;

  // getters et setters
  …
```



association many-to-many

> @ManyToMany définit le type d'association
> @JoinTable permet de préciser la table d'association

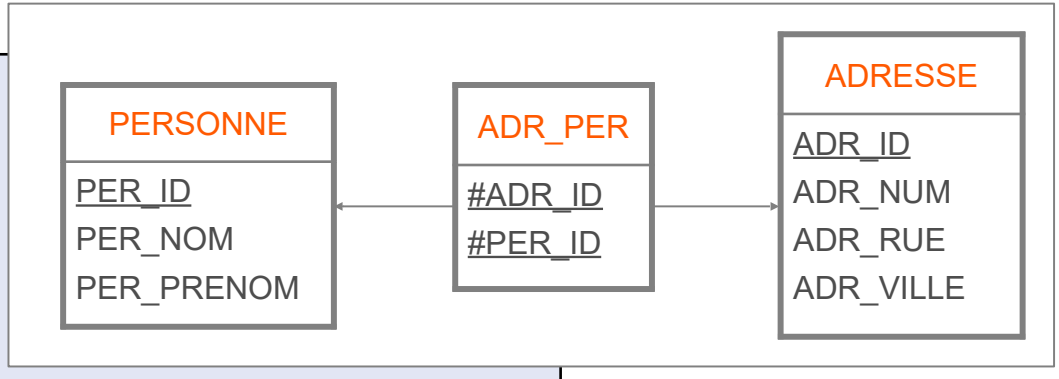
```
@Entity @Table(name="PERSONNE")
public class Personne {
  @Id @Column(name="PER_ID") private Long id;

  @Column(name="PER_NOM") private String nom;
  …
  @ManyToMany
  @JoinTable(name="ADR_PER",
            joinColumns={@JoinColumn(name="PER_ID")},
            inverseJoinColumns={@JoinColumn(name="ADR_ID")})
  private Set<Adresse> adresses;

  // getters et setters
  …
```



association many-to-many bidirectionnelle

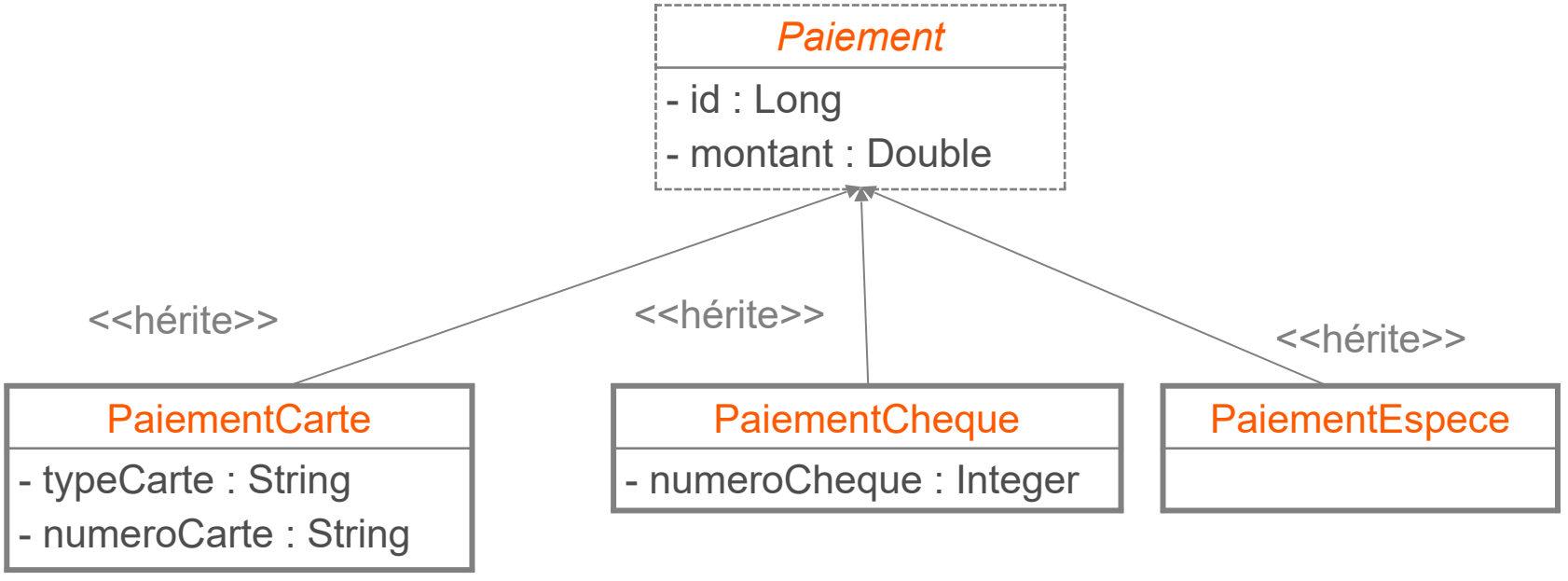
> La table d'association n'est renseignée que d'un côté avec @JoinTable
> L'attribut « mappedBy » permet à Hibernate de retrouver la foreign key sur le côté non renseigné

```
@Entity @Table(name="ADRESSE")
public class Adresse {

  @Id @Column(name="ADR_ID")
  private Long id;

  @Column(name="ADR_NUM")
  private Long numero;
  …

  @ManyToMany(mappedBy="adresses")
  private Set<Personne> personnes;

  // getters et setters
  …
```



mapping de l'héritage

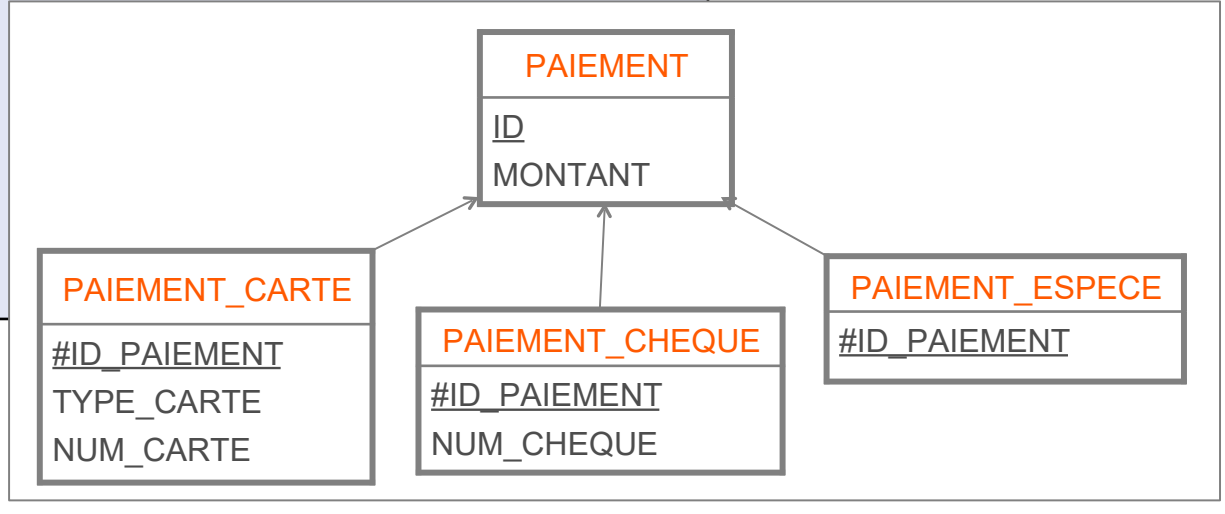
> Hiérarchie d'héritage



mapping de l'héritage

> Table par sous-classe avec jointure
  - La classe mère est annotée avec @Inheritance
  - Le type d'héritage est InheritanceType.JOINED

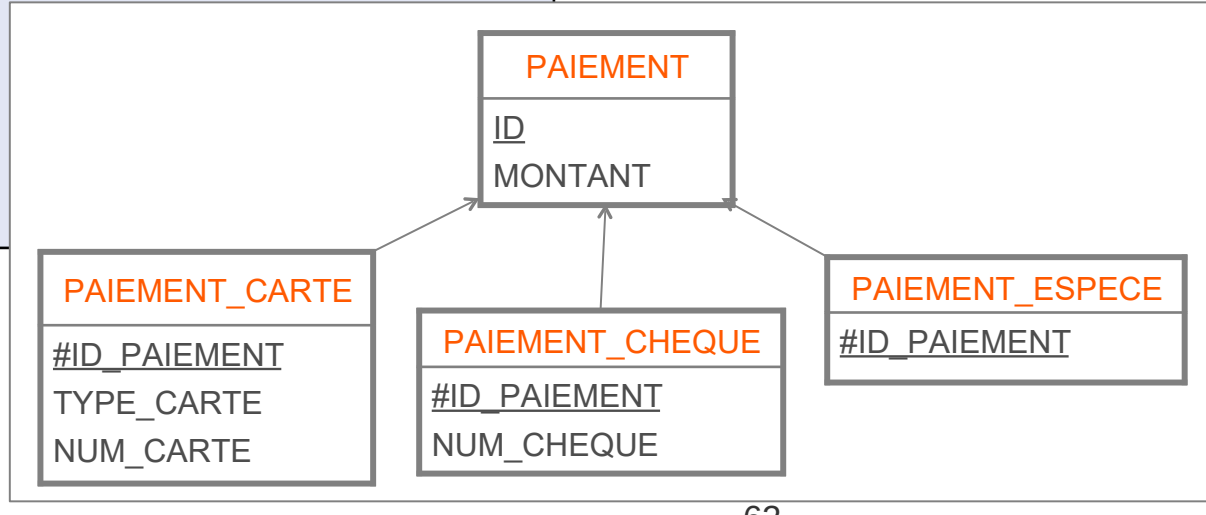
```
@Entity @Table(name="PAIEMENT")
@Inheritance(strategy=InheritanceType.JOINED)
public class Paiement {

  @Id @Column(name="ID")
  private Long id;

  @Column(name="MONTANT")
  private double montant;

  // getters et setters
  …
```



mapping de l'héritage

> Table par sous-classe avec jointure
  - Les classes filles sont annotées avec @PrimaryKeyJoinColumn pour définir la clé primaire + foreign key

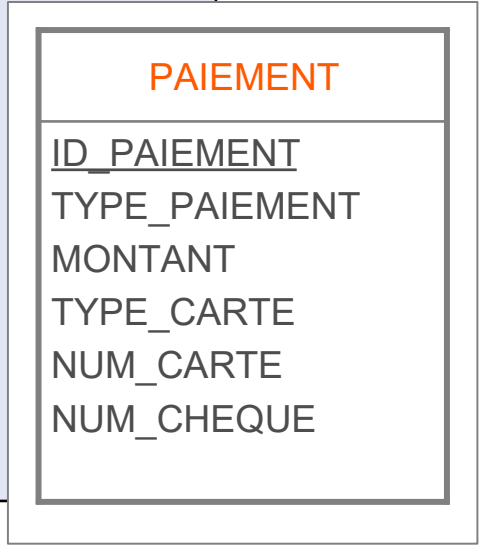
```
@Entity @Table(name="PAIEMENT_CARTE")
@PrimaryKeyJoinColumn(name="ID_PAIEMENT")
public class PaiementCarte extends Paiement {

        @Column(name="TYPE_CARTE")
        private String typeCarte;

        @Column(name="NUM_CARTE")
        private String numeroCarte;

  // getters et setters
  …
```



mapping de l'héritage

> Table par hierarchie de classe
  - La classe mère est annotée avec @Inheritance et @DiscriminatorColumn pour préciser la colonne de type
  - Le type d'héritage est InheritanceType.SINGLE_TABLE

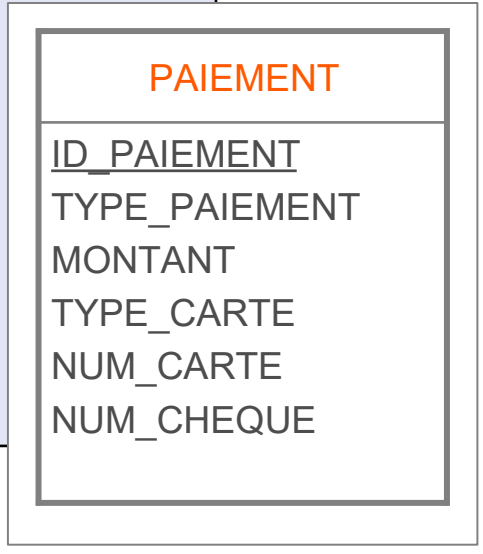
```
@Entity @Table(name="PAIEMENT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_PAIEMENT",
        discriminatorType=DiscriminatorType.STRING)
public class Paiement {

  @Id @Column(name="ID_PAIEMENT")
  private Long id;

  @Column(name="MONTANT")
  private double montant;

  // getters et setters
  …
```



mapping de l'héritage

> Table par hierarchie de classe
  - Les classes sont annotées avec @DiscriminatorValue pour définir les valeurs de type correspondant à chaque classe de la hierarchie
  - On ne répète pas @Table sur les classes filles

```
@Entity @DiscriminatorValue("PC")
public class PaiementCarte extends Paiement {

        @Column(name="TYPE_CARTE")
        private String typeCarte;

        @Column(name="NUM_CARTE")
        private String numeroCarte;

  // getters et setters
  …
```



gestion transactionnelle

notion de transaction

> Opération informatique cohérente composée de plusieurs tâches unitaires
  - soit toutes les opérations réussissent
  - soit l'ensemble est annulé

> Propriétés ACID
  - atomicité : tout ou rien
  - cohérence : les données ne sont pas laissées dans un état intermédiaire
  - isolation : les états intermédiaires sont invisibles aux autres transactions
  - durabilité : après une transaction, l'état est définitivement enregistré

gestion transactionnelle avec JDBC

> Gestionnaire de transaction : java.sql.Connection

- auto-commit=true
  - Transaction = une requête

- auto-commit=false
  - Les méthodes commit() et rollback() marquent la fin de la transaction

```
Connection connection = getConnection();// Début transaction
try {
  connection.setAutoCommit(false);
  process();// Traitement métier
  connection.commit();// Fin transaction
} catch (Exception e) {
  connection.rollback();// Fin transaction
} finally {
  try {
    connection.close();
  } catch (Exception e)
}
```

gestion transactionnelle avec JTA

> JTA

- Transactions en environnement distribué
- Exemple : conteneur EJB
- Objet de pilotage de la transaction → javax.transaction.UserTransaction

- La méthode begin() marque le début de la transaction
- Les méthodes commit() et rollback() marquent la fin de la transaction

```
UserTransaction tx = getUserTransaction();
try {
  tx.begin();// Début transaction
  process();// Traitement métier
  tx.commit();// Fin transaction
} catch (Exception e) {
  tx.rollback();// Fin transaction
}
```

gestion transactionnelle avec Hibernate

> Couche d'abstraction au-dessus de JDBC / JTA
  - Objet de pilotage de la transaction → org.hibernate.Transaction
  - Code portable

  - La méthode beginTransaction() marque le début de la transaction
  - Les méthodes commit() et rollback() marquent la fin de la transaction

```
Session session = getSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();// Début transaction
  process();// Traitement métier
  tx.commit();// Fin transaction
} catch (Exception e) {
   if (tx != null) tx.rollback();// Fin transaction
   throw e;
} finally {
  session.close();
}
```

déploiement dans une application web

> Utilisation d'une data source

- Hibernate ne gère plus la connection
- Les quatre propriétés Hibernate / JDBC doivent être remplacées par la propriété Hibernate / DataSource dans le fichier hibernate.cfg.xml

```
hibernate.connection.datasource = java:/comp/env/jdbc/MyDatasource
```

- Rappel : propriétés Hibernate / JDBC
  - hibernate.connection.driver_class
  - hibernate.connection.url
  - hibernate.connection.username
  - hibernate.connection.password

utiliser une session contextuelle

> Lors d'une transaction, la session doit être partagée entre
  - La couche de service métier pour la gestion transactionnelle
  - Les DAO pour le requêtage

> Principe → utiliser une session contextuelle

```
hibernate.current_session_context_class = thread
```

utiliser une session contextuelle

> Lorsqu'une session contextuelle est utilisée
  - La session est récupérée à l'aide de la méthode getCurrentSession() de la SessionFactory → pas openSession()
  - La méthode close() ne doit pas être invoquée, car la session est fermée automatiquement par Hibernate à la fin de la transaction

```
Session session = sessionFactory.getCurrentSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();// Début transaction

  process();// Traitement métier

  tx.commit();// Fin transaction
} catch (Exception e) {
   if (tx != null) tx.rollback();// Fin transaction
   throw e;
}
```