Les Beans Entity et Java Persistence API

Pr. Youssef Saadi

Master Informatique Décisionnelle

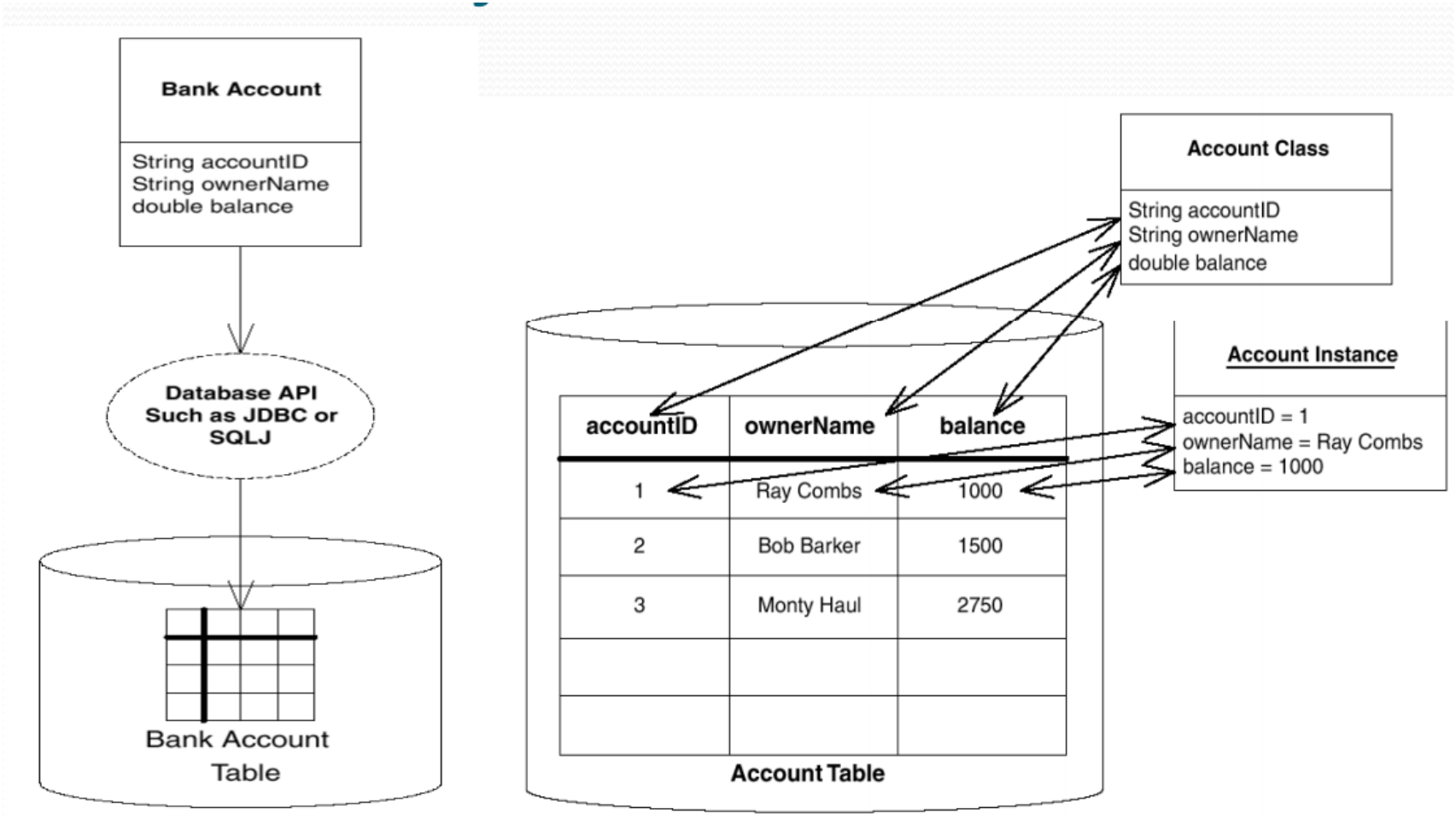
Faculté Des Sciences Et Techniques
Université Sultan Moulay Slimane Béni-Mellal

AU: 2019/2020

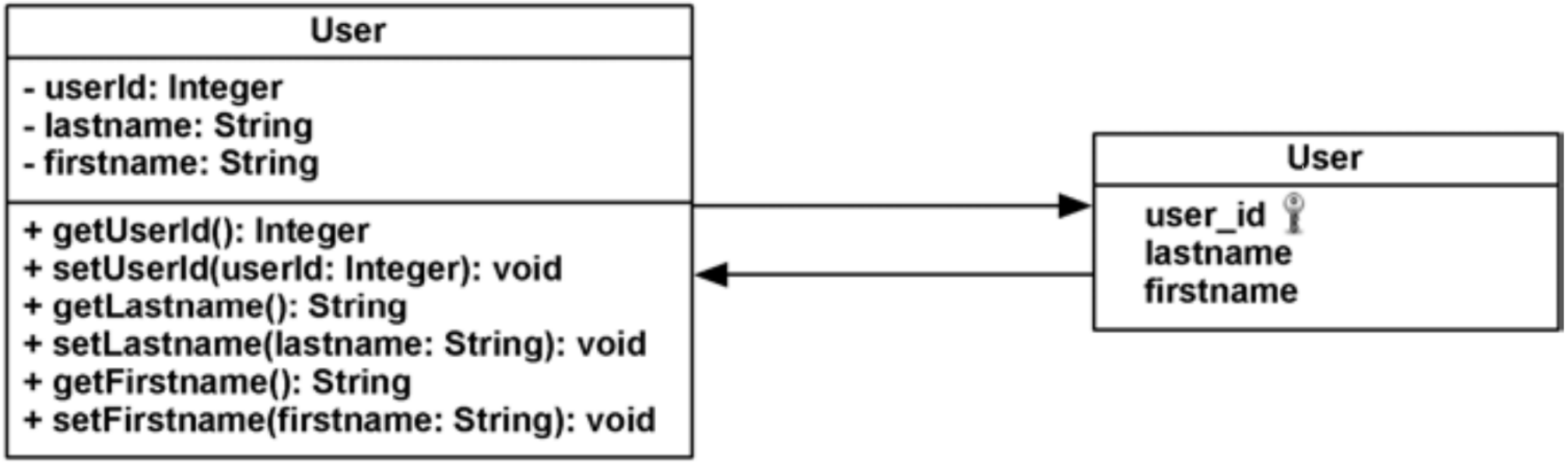
Entity Bean

- Un Entity Bean représente:
  - Des objets persistants stockés dans une base de données,
- Gestion via JPA2 + Session Beans
- JPA utilise la persistance par Mapping Objet/Relationnel:
  - On stocke l'état d'un objet dans une base de donnée.
  - *Exemple* : la classe Personne possède deux attributs ***nom*** et ***prenom***, on associe cette classe à *une table* qui possède deux colonnes : ***nom*** et ***prenom***.
- Les outils qui assureront la persistance (Toplink, Hibernate, EclipseLink,...) sont intégrés au serveur d'application et devront être compatibles avec la norme JPA.

Mapping O/R (ORM)



Mécanisme de persistance objet/relationnel



La liaison entre les données et l'application par un objet s'appelle le mapping (relier).

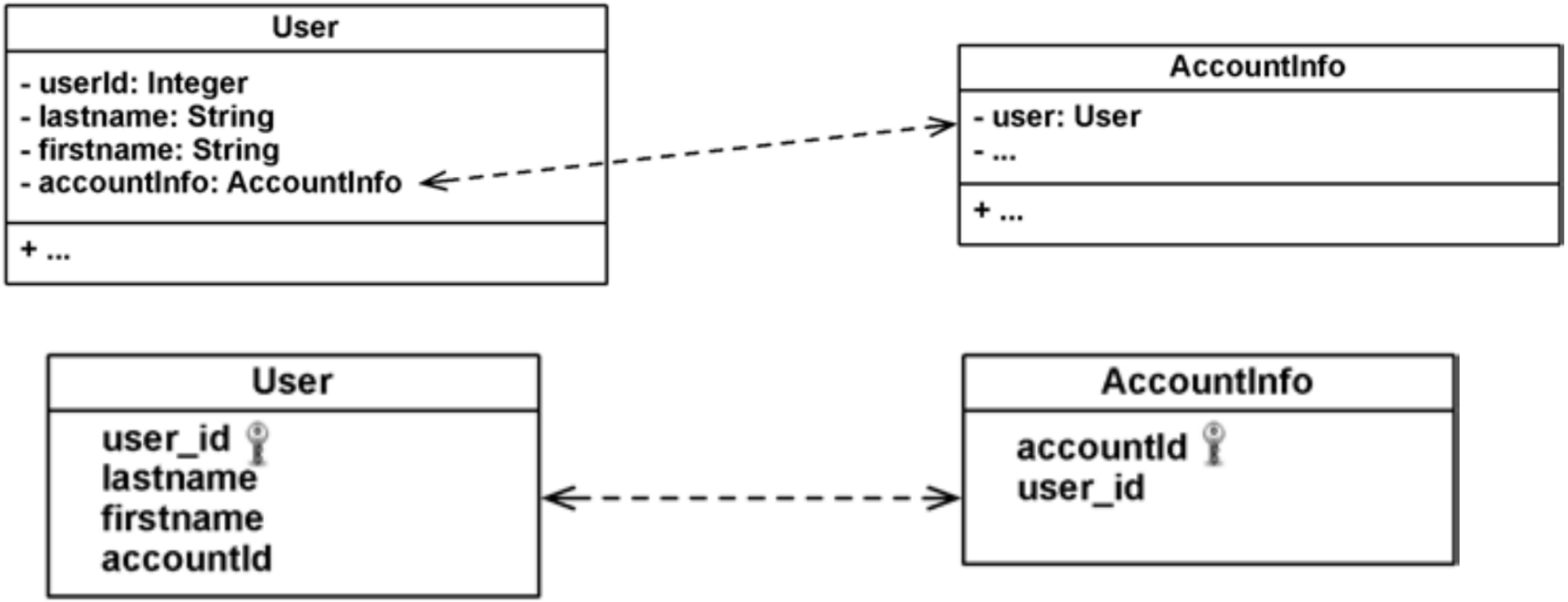
Session Bean / Entity Bean

- L'utilisation des Entity Beans permet de représenter une entité de l'application et non une fonctionnalité.
  - **Exemple**: User serait un Entity Bean, mais UserSubscription serait un Session Bean ; un utilisateur étant voué à rester persistant, alors que l'inscription de l'utilisateur est une opération.
- Contrairement aux Session Beans, les données des Entity Beans ont généralement une durée de vie longue ; elles sont enregistrées et stockées dans des systèmes de persistance (base de données).

Propriété d'un Entity Bean

- Chaque **propriété** de cet objet est liée à un **champ** de la table.
- Chaque **instance** de cet objet représente généralement un **enregistrement** de la table.
- Toutefois, il est possible qu'un Entity Bean soit **réparti** sur **plusieurs tables**.
- Les Entity Bean suivent le même principe que les tables relationnelles et doivent tous posséder un **identifiant** unique (clé primaire).
- Un **champ relationnel** est représenté, dans un Entity Bean, par une propriété dont le type est un autre Entity Bean. On parle **d'agrégation**, en programmation objet. À l'opposé, une table est liée à une autre table par une **clé étrangère**.

Liaison entre les classes User et AccountInfo



Types de relation entre les Entity Bean

- **One To One** (un à un) :
  - Si un utilisateur ne peut avoir qu'un seul et unique compte alors la relation entre l'utilisateur et son compte est de type « One To One ».
- **One To Many** (un à plusieurs) et **Many To One** (plusieurs à un) :
  - Un utilisateur peut avoir plusieurs portefeuilles alors qu'un portefeuille est détenu par un seul utilisateur. La relation entre Portefeuille et Utilisateur est de type « **Many To One** » et la relation entre Utilisateur et Portefeuille est de type « **One To Many** ».
- **Many To Many** (plusieurs à plusieurs) :
  - Un utilisateur a plusieurs loisirs(hobby) et un loisir peut être partagé avec plusieurs utilisateurs. Dans ce cas, la relation est de type « **Many To Many** » entre utilisateur et utilisateur.

Création d'un Entity Bean (EJB3)

- L'annotation **@Entity** précise au conteneur les classes à considérer en tant qu'Entity Bean.
- Cette annotation permet de définir, si besoin, le nom (unique dans une application) de l'entité via l'attribut **name**.
- La valeur par défaut utilisée est le nom de la classe de l'entité.

```
@Entity(name = "MyUser")
public class User {
//...
}
```

Création d'un Entity Bean (EJB3)

- Par défaut, un Entity Bean User est mappé sur la table « User ».
- Si pour certaines raisons, vous deviez le *mapper* sur une autre table, vous devrez alors utiliser l'annotation **@Table**. Cette annotation possède différents attributs :
  - **name** (requis) : définit le nom de la table à utiliser pour le *mapping*.
  - **catalog** (optionnel) : définit le catalogue utilisé.
  - **schema** (optionnel) : définit le schéma utilisé.
  - **uniqueConstraints** (optionnel) : définit les contraintes qui seront placées sur la table. Cet attribut est utilisé lorsque le conteneur génère les tables au déploiement et n'affecte en rien l'exécution même de l'entité.

```
@Entity
@Table(name = "User")
public class User {
//...
}
```

Les champs persistants

- N'importe quel champ (variable d'instance) non « **static** » et non « **transient** », d'un Entity Bean, est automatiquement considéré comme persistant par le conteneur.
- On peut considérer deux types d'annotations liés au mapping objet/relationnel :
  - les annotations liées aux **propriétés**,
  - les annotations liées aux **colonnes**.
- Ces deux types d'annotations, qui seront décrites de manière plus détaillée dans les parties suivantes, peuvent se placer de deux façons :
  - directement sur les champs (type « **FIELD** »),
  - sur les accesseurs (et plus précisément le ***getter***, type « **PROPERTY** »).

Exemple

Les annotations étant ici précisées sur les variables d'instance (façon « FIELD »), le conteneur injecte les valeurs directement dans celles-ci.

```
@Entity
...
public class User implements Serializable {

  // Déclaration de l'énumération des sexes (Masculin / Féminin)
  public enum SexType { MALE, FEMALE };

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int id;

  @Basic // optionnel
  private String lastName;

  private String firstName;

  @Column(unique=true)
  private String login;

  private String password;

  @Enumerated(value=EnumType.STRING)
  @Column(length=5)
  private SexType sex;

  @Temporal(TemporalType.DATE)
  private Date birthDate;
  //...
}
```

Exemple avec utilisation de la façon « PROPERTY »

```
@Entity
...
public class User implements Serializable {
  // Déclaration de l'énumération des sexes (Masculin / Fé
  public enum SexType { MALE, FEMALE };

  private int id;
  private String lastName;
  private String firstName;
  private String login;
  private String password;
  private SexType sex;
  private Date birthDate;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public int getId() { return id; }

  public void setId(int id) { this.id = id; }

  @Basic // optionnel
  public String getLastName() { return lastName; }

  public void setLastName(String lastName) { this.lastName = lastName; }

  public String getFirstName() { return firstName; }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  @Column(unique=true)
  public String getLogin() { return login; }

  public void setLogin(String login) { this.login = login;}

  public String getPassword() { return password; }

  public void setPassword(String password) { this.password = password; }

  @Enumerated(value=EnumType.STRING)
  @Column(length=5)
  public SexType getSex() { return sex; }

  public void setSex(SexType sex) { this.sex = sex; }

  @Temporal(TemporalType.DATE)
  public Date getBirthDate() { return birthDate;}

  public void setBirthDate(Date birthDate) { this.birthDate = birthDate; }
  //...
}
```

Annotations liées aux propriétés simples

- Le conteneur considère par défaut que la propriété est annotée avec **@Basic** avec les valeurs par défaut des attributs suivants :
  - **fetch** (**FetchType.EAGER** par défaut) : définit si le contenu de la propriété doit être chargé à la demande. (**FetchType.LAZY**, « paresseusement », en anglais) ou au moment du chargement de l'entité (FetchType.EAGER, « désireux », en anglais).
  - **optional** (true par défaut) : définit si la propriété accepte la valeur « **null** » ou non. Cet attribut ne fonctionne pas pour les types primitifs (qui ne peuvent être nuls).

Annotations liées aux propriétés simples

- Pour indiquer qu'une propriété ne doit pas être enregistrée, il faut annoter son *getter* avec **@Transient** (seulement si cette méthode est de la forme **getXxx()**).
- Les types **java.util.Date** ou **java.util.Calendar** utilisés pour définir des propriétés dites « **temporelles** » peuvent être paramétrés pour spécifier le format le plus adéquat à sa mise en persistance :
  - DATE : utilisé pour la date (java.sql.Date),
  - TIME : utilisé pour l'heure (java.sql.Time),
  - TIMESTAMP : utilisé pour les temps précis (java.sql.TimeStamp)

```
@Temporal(TemporalType.DATE)
public Calendar getDateOfBirth() {
    return dateOfBirth ;
}
```

Annotations liées aux propriétés simple

- L'énumération permet de spécifier un ensemble de valeurs possibles pour une propriété.

```
public enum SexType { MALE, FEMALE };
//...
@Enumerated(value=EnumType.STRING)
public SexType getSex() { return sex; }
```

Annotations liées aux colonnes simples

- On utilise l'annotation **@Column** qui peut être utilisée conjointement avec les précédentes.

```
@Column(updatable = true, name = "price", nullable = false,
  precision=5, scale=2)
public float getPrice() { return price; }
```

Annotations liées aux colonnes simples

- Voici une description des attributs, tous optionnels, de l'annotation **@Column** :
  - **name** : précise le nom de la colonne liée. Le nom de la propriété est utilisé par défaut.
  - **unique** : précise si la propriété est une clé unique ou non (la valeur est unique dans la table).
  - **nullable** : précise si la colonne accepte des valeurs nulles ou non.
  - **insertable** : précise si la valeur doit être incluse lors de l'exécution de la requête SQL INSERT. La valeur par défaut est **true**.
  - **updatable** : précise si la valeur doit être mise à jour lors de l'exécution de la requête SQL UPDATE. La valeur par défaut est **true**.

Annotations liées aux colonnes simples

- **table** : précise la table utilisée pour contenir la colonne. La valeur par défaut est la table principale de l'entité. Cet attribut est utilisé lorsqu'un Entity Bean est *mappé* à plusieurs tables.
- **length** : précise la longueur que la base de données doit associer à un champ texte. La longueur par défaut est 255.
- **precision** : précise le nombre maximum de chiffre que la colonne peut contenir. La précision par défaut est définie par la base de données.
- **scale** : précise le nombre fixe de chiffres après le séparateur décimal (en général le point). Cet attribut n'est utilisable que pour les propriétés décimales (float, double …). Le nombre de décimal par défaut est défini par la base de données.

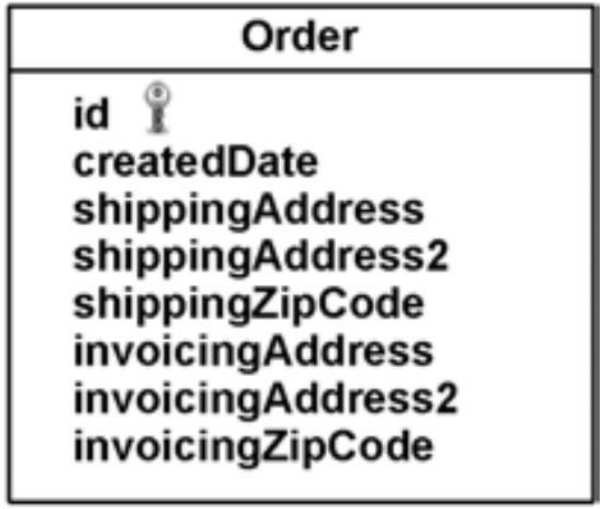
Objets incorporés

```
@Embeddable
public class Address implements Serializable {
  @Column(length=30)
  private String address1;

  @Column(length=30)
  private String address2;

  @Column(length=12)
  private String zipCode;

  public Address() {}

  public String getAddress1() { return address1; }
  public void setAddress1(String address1) { this.address1 = address1; }

  public String getAddress2() { return address2; }
  public void setAddress2(String address2) { this.address2 = address2; }

  public String getZipCode() { return zipCode; }
  public void setZipCode(String zipCode) { this.zipCode = zipCode;}
}
```

Utilisation des objets incorporés

```
@Entity
public class Person {
  @Id
       private String firstName;

  @Embedded
       private Address address;

  public Person() { }
  public Address getAddress() { return address; }
  public void setAddress(Address address) { this.address = address; }
}
```

Utilisation des objets incorporés




```
@Entity
@Table(name="ORDERS")
public class Order {
  private @Id int id;
  private @Temporal(TemporalType.DATE) Date createdDate;

  @Embedded

  @AttributeOverrides({
    @AttributeOverride(name="address1",
      column=@Column(name="shipppingAddress1")),
    @AttributeOverride(name="address2",
      column=@Column(name="shippingAddress2")),
    @AttributeOverride(name="zipCode", column=@Column(name="shippingZipCode"))
})
  private Address shippingAddress;
```

Identification unique

- Un Entity Bean doit posséder un champ dont la valeur est unique, dit **identificateur** unique ou clé primaire.
- Il existe deux types d'identifiants :
  - **simple**
  - **composite**

Identification simple

- On parle d'identifiant simple lorsque celui-ci est composé par **un unique champ dont le type est « simple ».** Les types simples sont :
  - les types **primitifs** (int, float, char…)
  - les **enveloppeurs** (*wrapper*, en anglais. Par exemple Integer, Float, Double…)
  - les types **String** ou **Date** (java.util.Date ou java.sql.Date).
- Pour spécifier au conteneur qu'un champ est une clé primaire, il faut annoter celui-ci avec **@Id**.
- Dans notre Entity Bean User, la clé primaire est assignée au champ id.

Identification simple

```
@Entity
public class User implements Serializable {
  private int id;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO) // optionnel
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }
...
}
```

Stratégies de génération de valeurs

- Il existe quatre stratégies de génération disponibles : AUTO, IDENTITY, SEQUENCE et TABLE.
- Le type IDENTITY indique au fournisseur de persistance d'assigner la valeur de la clé primaire en utilisant la colonne identité de la base de données. Sous MySQL, par exemple, la clé primaire auto-générée est marquée avec « **AUTO_INCREMENT** ».

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public int getId() { ... }
```

Identification simple

- Le type **SEQUENCE**, comme son nom l'indique, oblige le fournisseur de persistance à utiliser une séquence de la base de données. Celle-ci peut être déclarée au niveau de la classe ou au niveau du package grâce à l'annotation **@SequenceGenerator** et ses attributs :
  - **name** (requis) : définit un nom unique pour la séquence qui peut être référencée par une ou plusieurs classes (suivant le niveau utilisé pour la déclaration de l'annotation).
  - **sequenceName** (optionnel) : définit le nom de l'objet séquence de la base de données qui sera utilisé pour récupérer les valeurs des clés primaires liées.
  - **initialValue** (optionnel) : définit la valeur à laquelle doit démarrer la séquence.
  - **allocationSize** (optionnel) : définit le nombre utilisé pour l'incrémentation de la séquence lorsque le fournisseur de persistance y accède.

Identification simple

```
@Entity
@SequenceGenerator (
  name="SEQ_USER",
  sequenceName="SEQ_USER"
)
public class User {
  private int id;

  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_USER")
  public int getId() { return id; }
}
```

Identification simple

- Le type AUTO indique au fournisseur de persistance d'utiliser la meilleure stratégie (entre IDENTITY, TABLE, SEQUENCE) suivant la base de données utilisée. Le générateur AUTO est le type préféré pour avoir une application portable.

Identifiant composite

- Une clé primaire non « simple » (objet personnalisé…), pour être unique, doit rassembler une combinaison de propriétés. On parle alors de clé primaire composite ou identifiant composite.
- La première étape pour utiliser une clé primaire composite est de créer une classe de clé primaire. Cette classe doit :
  - Définir les propriétés devant être liées à l'identifiant unique.
  - Avoir une visibilité public.
  - Avoir un constructeur public sans argument.
  - Implémenter l'interface java.io.Serializable.
  - Surdéfinir les méthodes equals() et hashCode().
  - Déclarer au conteneur une classe de clé primaire composite, *via* l'annotation **@Embeddable**.

Identifiant composite

```
@Embeddable
public class ContactPK implements Serializable {
  private String firstName;
  private String lastName;

  // Constructeur public sans argument
  public ContactPK() {
  }

  public String getFirstName() {
    return firstName;
  }
```

```
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
```

Identifiant composite

```
  @Override
  public boolean equals(Object obj) {
    // vérifie que l'objet passé en paramètre est de type ContactPK
    if(obj instanceof ContactPK) {
      ContactPK pk = (ContactPK) obj;
      // vérifie si les prénoms sont égaux
      if(this.getFirstName().equals(pk.getFirstName())) {
        // vérifie si les noms sont égaux
        if(this.getLastName().equals(pk.getLastName())) {
          return true;
        }
      }
    }
    return false;
  }

  @Override
  public int hashCode() {
    return (getFirstName() + getLastName()).hashCode();
  }
}
```

Identifiant composite

- Il existe, ensuite deux façons d'utiliser cette classe au sein d'un Entity Bean. La première, est de définir une propriété avec le type de votre clé primaire. Cette propriété doit être annotée avec **@EmbeddedId** et non **@Id**.

```
@Entity
public class Contact {

  private ContactPK contactPK;

  private String address;

  protected Contact() {
  }

  public Contact(String firstName, String lastName) {
    ContactPK contactPK = new ContactPK();
    contactPK.setFirstName(firstName);
    contactPK.setLastName(lastName);
    setContactPK(contactPK);
  }
```

```
  @EmbeddedId
  public ContactPK getContactPK() {
    return contactPK;
  }

  public void setContactPK(ContactPK contactPK) {
    this.contactPK = contactPK;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String address) {
    this.address = address;
  }
}
```

Identifiant composite

- La seconde façon n'utilise pas explicitement la classe de la clé primaire mais y fait référence *via* l'annotation **@IdClass**. La classe de l'Entity Bean déclare explicitement les composantes de la clé primaire et chacune de celle-ci est annotée avec **@Id**.

```
@Entity
@IdClass(value=ContactPK.class)
public class Contact {

  private String firstName;

  private String lastName;

  private String address;

  public String getAddress() {
    return address;
  }
  @Id
  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  @Id
  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public void setAddress(String address) {
    this.address = address;
  }
}
```

Les champs relationnels : *Un à Un (One to One)*

- Une relation « One to One » est utilisée pour lier deux entités uniques indissociables.
  - Par exemple, un corps n'a qu'un seul cœur, ou une personne n'a qu'une seule carte d'identité.
- Pour associer deux entités avec ce type de relation, il faut utiliser l'annotation **@OneToOne**.
- Celle-ci reprend les attributs de **@Basic**, vu précédemment, et propose d'autres attributs optionnels dont:
  - **cascade** : spécifie les opérations à effectuer en cascade.
  - **mappedBy** : spécifie le champ propriétaire de la relation dans le cas d'une relation bidirectionnelle.
- Ce type de relation peut être *mappé* de trois manières dans la base de données:

One to One: Mapping par jointure par clé primaire

- Nous supposerons, dans notre exemple, qu'un utilisateur **User** n'a qu'un seul compte **AccountInfo**.

```
@Entity
public class AccountInfo {
//...
  private int id;
  private String cardNumber;
  private double amount;
  private String accountId;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  public int getId() { return id; }
  public void setId(int id) { this.id = id; }

  public String getCardNumber() { return cardNumber; }
  public void setCardNumber(String cardNumber) {
    this.cardNumber = cardNumber;
  }
```

One to One: Mapping par jointure par clé primaire

```
@Entity
public class User {
//...
  @OneToOne
  @PrimaryKeyJoinColumn
  public AccountInfo getAccountInfo() {
    return accountInfo;
  }
//...
}
```

One to One: Mapping clé primaire-clé étrangère

- La deuxième solution consiste à utiliser une clé étrangère d'un côté de la relation. Toutefois, il faut noter que la colonne de cette clé doit être marquée comme unique afin de simuler correctement la relation « One to One ».

```
@Entity
public class User {
//...
  @OneToOne
  @JoinColumn(name="account_id", referencedColumnName="id")
  public AccountInfo getAccountInfo() {
    return accountInfo;
  }
//...
}
```

One to One: Mapping clé primaire-clé étrangère

- L'attribut **mappedBy** déclare que le côté propriétaire est celui détenant la propriété **accountInfo**. C'est donc, ici, l'entité User qui détient la relation et a donc le pouvoir de lier un utilisateur à un compte (l'inverse étant impossible).
-

```
@Entity
public class AccountInfo {
//...
  private int id;
  private User user;

  @OneToOne(mappedBy = "accountInfo")
  public User getUser() {
    return user;
  }
//...
}
```

One to One: Mapping avec une table d'associations

- La multiplicité « One to One » est respectée si et seulement si une contrainte unique est définie sur chaque clé étrangère.

```
@Entity
public class User {
//...
  @OneToOne
  @JoinTable(name = "UserAccountInfo"
    joinColumns = @JoinColumn(name="user_fk", unique=true),
    inverseJoinColumns = @JoinColumns(name="accountinfo_fk", unique=true)
  )
  public AccountInfo getAccountInfo() { ... }
}
```

One to One: Mapping avec une table d'associations

```
@Entity
public class AccountInfo {
//...
  @OneToOne(mappedBy = "accountInfo")
  public User getUser() { ... }
}
```

Un à Plusieurs (One To Many) et Plusieurs à Un (Many To One)

- Une relation « **One To Many** », et respectivement « **Many To One** », est utilisée pour lier à une unique instance d'une entité A, un groupe d'instances d'une entité B.
  - Par exemple, une personne possède plusieurs comptes bancaires, mais un compte bancaire n'appartient qu'à une seule personne.
- Une association « **Many To One** » est définie sur une propriété avec l'annotation **@ManyToOne**. Dans le cas d'une relation bidirectionnelle, l'autre côté doit utiliser l'annotation **@OneToMany**. Les attributs de ces deux annotations correspondent à ceux de l'annotation **@OneToOne**.

Un à Plusieurs (One To Many) et Plusieurs à Un (Many To One)

- Dans nos exemples suivants, un utilisateur peut avoir plusieurs portefeuilles d'actions, mais un portefeuille n'est lié qu'à un seul utilisateur.

```
@Entity
public class Portfolio {
//...
  private User user;


  @ManyToOne
  @JoinColumn(name = "user_fk")
  public User getUser() {
    return user;
  }

  public void setUser(User user) {
    this.user = user;
  }
}
```

Un à Plusieurs (One To Many) et Plusieurs à Un (Many To One)

```
@Entity
public class User implements Serializable {
//...
  private Collection<Portfolio> portfolios;

  @OneToMany(mappedBy = "user")
  public Collection<Portfolio> getPortfolios() {
    return portfolios;
  }

  public void setPortfolios(Collection<Portfolio> portfolios) {
    this.portfolios = portfolios;
  }
}
```

Plusieurs à Plusieurs (Many To Many)

- Cette relation peut être utilisé pour lier des instances de deux entités entre elles.
  - Un article peut être associé à plusieurs catégories (cardinalité *n*) et une catégorie peut regrouper plusieurs articles (cardinalité *m*).
- Pour cela, il suffit d'utiliser des propriétés multi-valuées de chaque côté de la relation (si celle-ci est bidirectionnelle) et de les annoter avec **@ManyToMany**.

Plusieurs à Plusieurs (Many To Many)

```
@Entity
public class User {
//...

  private Collection<Hobby> hobbies;

  @ManyToMany
  @JoinTable(name = "USER_HOBBIES",
    joinColumns = @JoinColumn(name = "user_id", referencedColumnName = "id"),
    inverseJoinColumns = @JoinColumn(name = "hobby_id",
      referencedColumnName = "id"))
  public Collection<Hobby> getHobbies() {
    return hobbies;
  }

  public void setHobbies(Collection<Hobby> hobbies) {
    this.hobbies = hobbies;
  }
}
```

Plusieurs à Plusieurs (Many To Many)

```
@Entity
public class Hobby {
//...
  private Collection<User> users;

  @ManyToMany(mappedBy="hobbies")
  public Collection<User> getUsers() {
    return users;
  }

  public void setUsers(Collection<User> users) {
    this.users = users;
  }
}
```

Opérations en cascade

- La cascade signifie qu'une opération appliquée à une entité est propagée aux relations de celle-ci.
- Par exemple, lorsqu'un utilisateur est supprimé, son compte l'est également.
- Il existe quatre opérations possibles sur les entités : ajout, modification, suppression, rechargement. Ces opérations sont regroupées dans l'énumération **CascadeType** :
  - CascadeType.PERSIST
  - CascadeType.MERGE
  - CascadeType.REMOVE
  - CascadeType.REFRESH
  - CascadeType.ALL

Exemple 1

- Par exemple, sur la relation entre User et Portfolio, il est logique d'enregistrer ou supprimer les portefeuilles lorsque l'utilisateur est respectivement enregistré ou supprimé. Voici le code correspondant :

```
@OneToMany(cascade = { CascadeType.REMOVE, CascadeType.PERSIST },
  mappedBy = "user")
public Collection<Portfolio> getPortfolios() {
  return portfolios;
}
```

Exemple 2

- De la même façon, la relation « One to One » entre User et AccountInfo oblige à sauvegarder, mettre à jour et détruire AccountInfo lorsque ces opérations sont effectuées sur l'instance du User correspondant. Voici le code correspondant :

```
@OneToOne(cascade = CascadeType.ALL)
@PrimaryKeyJoinColumn
public AccountInfo getAccountInfo() {
 return accountInfo;
}
```

Association Many to Many avec propriétés

- Soit l'exemple suivant:
  - Une commande regroupe plusieurs produits et un produit peut figurés dans plusieurs commandes. Nous voulons enregistrer lors de cette association la quantité de produit désiré.
  - Il est nécessaire dans ce cas d'utiliser une double liaison « One to Many » et un Entity Bean intermédiaire.
  - Dans notre exemple, nous avons les Entity Bean suivants : **Order**, **Product** et **OrderLine**, représentant respectivement une commande, un produit et les lignes des commandes.

Exemple : L'entité Order

```
@Entity
@Table(name="ORDERS")
public class Order {

  private int id;

  private Date orderDate;

  private Collection<OrderLine> orderLines;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Date getOrderDate() {
    return orderDate;
  }

  public void setOrderDate(Date orderDate) {
    this.orderDate = orderDate;
  }

  @OneToMany(mappedBy = "order", cascade = {CascadeType.REMOVE})
  public Collection<OrderLine> getOrderLines() { return orderLines; }

  public void setOrderLines(Collection<OrderLine> ol) {this.orderLines = ol;}
}
```

Exemple : l'entité Product

```
@Entity
public class Product {

  private int id;

  private String name;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Exemple : la clé primaire composite

```
@Embeddable
public class OrderLinePk implements Serializable {

  private static final long serialVersionUID = 1L;

  private int orderId;

  private int productId;

  protected OrderLinePk() {
  }

  public OrderLinePk(int orderId, int productId) {
    this.orderId = orderId;
    this.productId = productId;
  }
```

Exemple

```
  public int getOrderId() {
    return orderId;
  }

  public void setOrderId(int orderId) {
    this.orderId = orderId;
  }

  public int getProductId() {
    return productId;
  }

  public void setProductId(int productId) {
    this.productId = productId;
  }

  public int hashCode() { return orderId ^ productId; }

  public boolean equals(Object that) {
    return (that instanceOf OrderLinkPk && orderId == productId);
  }
}
```

Exemple : l'entité OrderLine

```
@Entity
@IdClass(OrderLinePk.class)
public class OrderLine {

  private Product product;

  private Order order;

  private int quantity;

  public OrderLine() {
  }

  public int getQuantity() {
    return quantity;
  }
```

```
public void setQuantity(int quantity) {
  this.quantity = quantity;
}

@ManyToOne
@JoinColumn(name="orderId",
            optional=false,
            insertable=false,
            updatable=false)
public Order getOrder() {
  return order;
}
```

Exemple : l'entité OrderLine

```
public void setOrder(Order order) {
  this.order = order;
}

@ManyToOne
@JoinColumn(name="productId",
            optional=false,
            insertable=false,
            updatable=false)
public Product getProduct() {
  return product;
}

public void setProduct(Product product) {
  this.product = product;
}
```

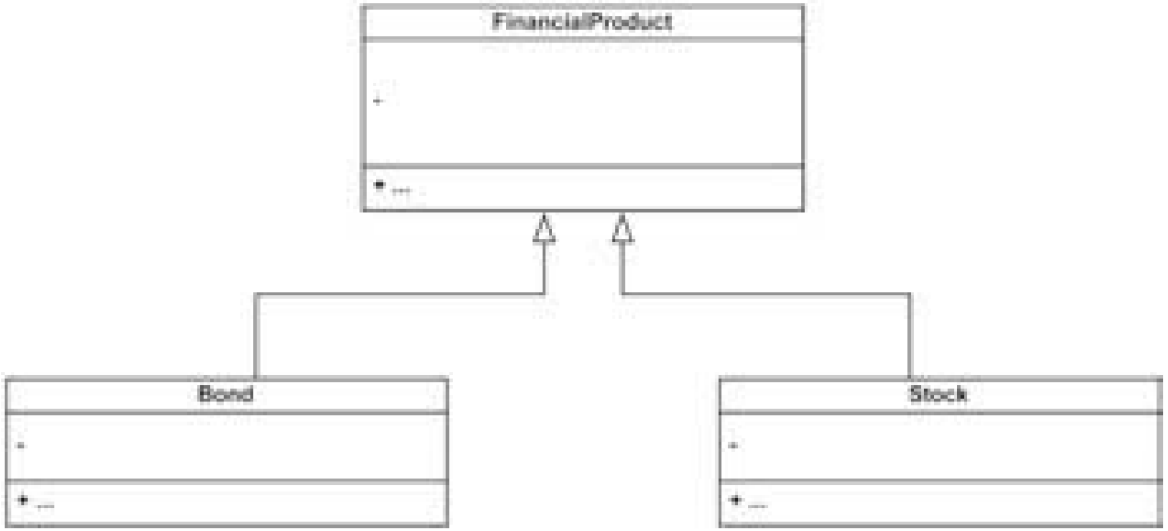
```
  @Id
  public int getProductId() {
    return getProduct().getId();
  }

  @Id
  public int getOrderId() {
    return getOrder().getId();
  }

  public void setOrderId(int orderId) {
    getOrder().setId(orderId);
  }

  public void setProductId(int productId) {
    getProduct().setId(productId);
  }
}
```

Mapping de l'héritage

- Voici les trois types de *mapping* relationnels possibles :
  - Une table unique par hiérarchie de classe.
  - Une table par classe concrète.
  - Une séparation des champs spécifiques d'une classe fille dans une table séparée de la table parente. Une jonction est alors faite pour instancier la classe fille.

L'héritage

- Les exemples de cette partie utiliseront deux Entity Beans : **Bond** et **Stock** qui héritent tous deux de l'Entity Bean **FinancialProduct** (***classe abstraite***).
- **Attention :** une seule clé primaire doit être définie dans une hiérarchie. Ici, la clé primaire se trouve dans la classe racine FinancialProduct.



Une table unique

- Dans cette stratégie, toutes les classes de la hiérarchie sont *mappées* dans une même et unique table. Le type d'héritage utilisé est spécifié au niveau de l'Entity Bean racine par l'annotation **@Inheritance**.

Une table unique

```
@Entity
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="financialproduct_type",
                     discriminatorType=DiscriminatorType.STRING,
                     length=10)
public abstract class FinancialProduct {
  private int id;
  //...
  @Id
  @GeneratedValue(strategy=GeneratorType.AUTO)
  public int getId(){
    return id;
  }
  //...
}
```

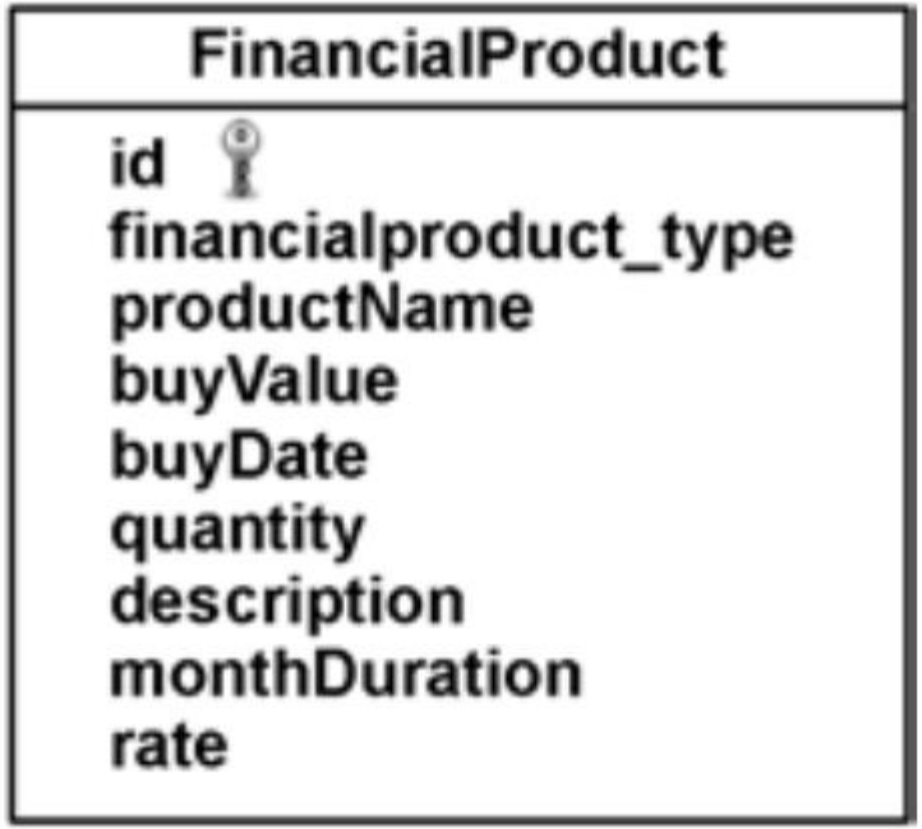
Une table unique

```
@Entity
@DiscriminatorValue("BOND") // valeur par défaut
public class Bond extends FinancialProduct {
  private double rate;        // taux de rendement de l'obligation
  private int monthDuration; // nombre de mois que dure l'obligation
   //...
   @Basic
```

```
@Entity
@DiscriminatorValue("STOCKOPTION")
public class Stock extends FinancialProduct {
...
}
```

Une table unique

- le résultat au niveau de la base de données (une seule table pour deux entités)



Une table par classe concrète

- Dans ce cas, chaque classe Entity Bean concrète est liée à sa propre table.
- Cela signifie que toutes les propriétés de la classe (incluant les propriétés héritées) sont incluses dans la table liée à cette entité.
- Concrètement, sur notre exemple, cela signifie que la table « **FinancialProduct** » n'est pas créée ni utilisée.
- Les Entity Beans **Stock** et **Bond** qui héritent de la classe FinancialProduct sont *mappés* respectivement sur les tables « Stock » et « Bond » ;
- Pour spécifier cette stratégie, il faut spécifier la stratégie définie par InheritanceType.TABLE_PER_CLASS à l'annotation **@Inheritance**.

Une table par classe concrète

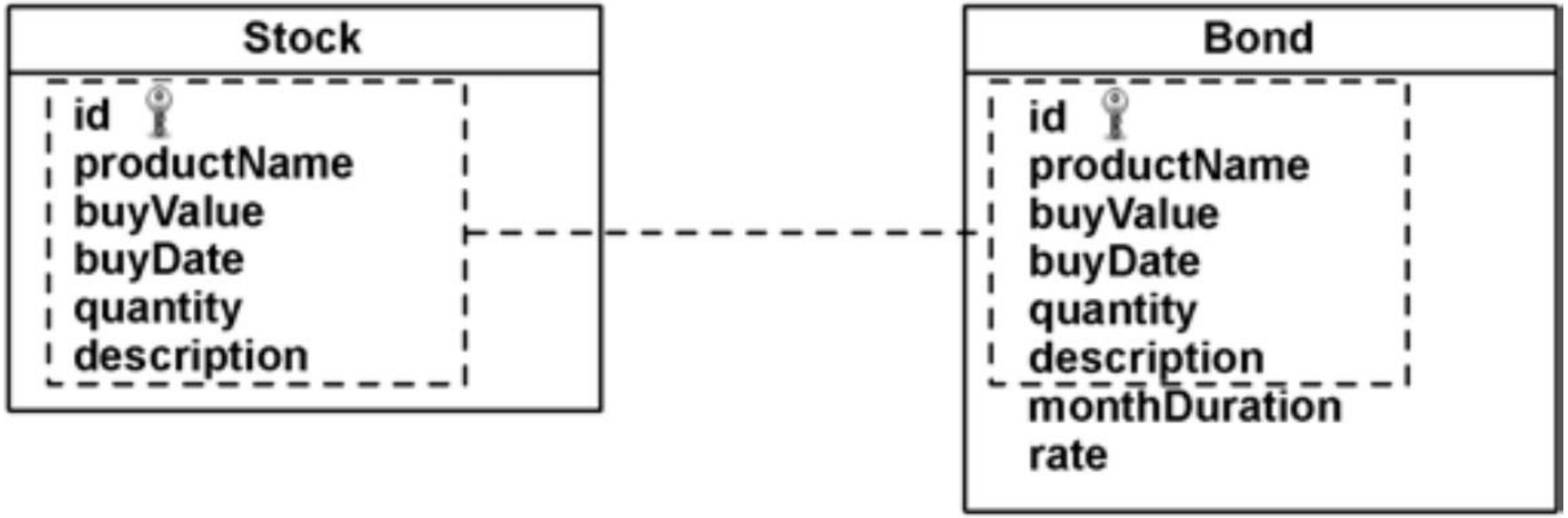
```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class FinancialProduct {
  //...
}
```

```
@Entity
public class Bond extends FinancialProduct {
  //...
}
```

```
@Entity
public class Stock extends FinancialProduct {
  //...
}
```

Une table par classe concrète

- Résultat en base de données relationnelles



Tables jointes

- Dans cette dernière stratégie, la classe racine des entités est représentée par une table.
- Chaque classe fille est liée à sa propre table séparée contenant les propriétés spécifiques de celle-là.
- La liaison entre les tables « filles » et la table racine se fait *via* les clés primaires.
- L'héritage est ici déclaré avec la stratégie InheritanceType.JOINED.

Tables jointes

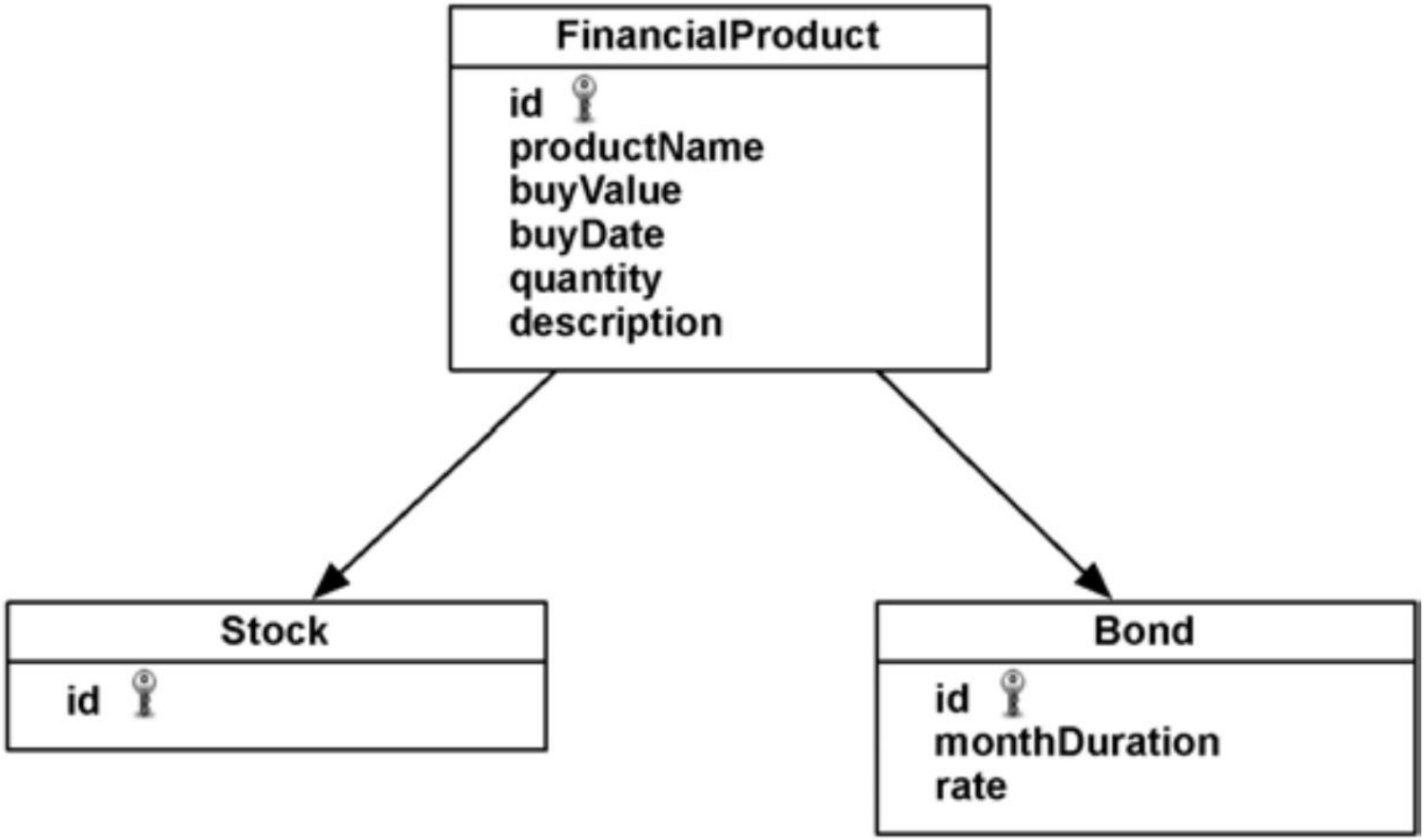
```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public class FinancialProduct {
 //...
}
```

```
@Entity
public class Bond extends FinancialProduct {
 //...
}
```

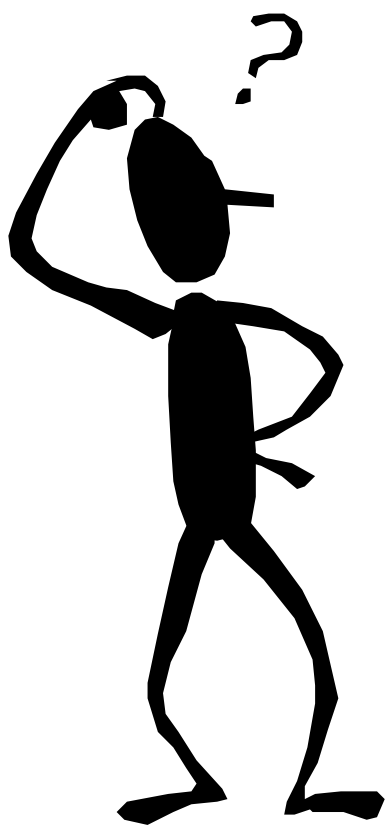
```
@Entity
public class Stock extends FinancialProduct {
  //...
}
```

Tables jointes

- Résultat en base de données relationnelles



Des Questions ??



Démo5

Entity Bean