étrangère est détenue par une des entités (notez que cette colonne de clef étrangère dans la base de données devrait être avoir une contrainte d'unicité pour simuler la cardinalité one-to-one), soit une table d'association est utilisée pour stocker le lien entre les 2 entités (une contrainte d'unicité doit être définie sur chaque clef étrangère pour assurer la cardinalité un à un).

Tout d'abord, nous mappons une véritable association one-to-one en utilisant des clefs primaires partagées :

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @PrimaryKeyJoinColumn
    public Heart getHeart() {
        return heart;
    }
    ...
}
```

```
@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

L'association un à un est activée en utilisant l'annotation `@PrimaryKeyJoinColumn`.

Dans l'exemple suivant, les entités associées sont liées à travers une clef étrangère :

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="passport_fk")
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
    ...
}
```

Un `Customer` est lié à un `Passport`, avec une colonne de clef étrangère nommée `passport_fk` dans la table `Customer`. La colonne de jointure est déclarée avec l'annotation `@JoinColumn` qui ressemble à l'annotation `@Column`. Elle a un paramètre de plus nommé `referencedColumnName`. Ce paramètre déclare la colonne dans l'entité cible qui sera utilisée pour la jointure. Notez que lors de l'utilisation de `referencedColumnName` vers une colonne qui ne fait pas partie de la clef primaire, la classe associée doit être `Serializable`. Notez aussi que `referencedColumnName` doit être mappé sur une propriété ayant une seule colonne lorsqu'elle pointe vers une colonne qui ne fait pas partie de la clef primaire (d'autres cas pourraient ne pas fonctionnner).

L'association peut être bidirectionnelle. Dans une relation bidirectionnelle, une des extrémités (et seulement une) doit être la propriétaire : la propriétaire est responsable de la mise à jour des colonnes de l'association. Pour déclarer une extrémité comme *non* responsable de la relation, l'attribut `mappedBy` est utilisé. `mappedBy` référence le nom de la propriété de l'association du côté du propriétaire. Dans notre cas, c'est `passport`. Comme

vous pouvez le voir, vous ne devez (absolument) pas déclarer la colonne de jointure puisqu'elle a déjà été déclarée du côté du propriétaire.

Si aucune `@JoinColumn` n'est déclarée du côté du propriétaire, les valeurs par défaut s'appliquent. Une(des) colonne(s) de jointure sera(ont) créée(s) dans la table propriétaire, et son(leur) nom sera la concaténation du nom de la relation du côté propriétaire, _ (underscore), et le nom de la (des) colonne(s) de la clef primaire du propriétaire. Dans cet exemple `passport_id` parce que le nom de la propriété est `passport` et la colonne identifiante de `Passport` est `id`.

La troisième possibilité (utilisant une table d'association) est très exotique.

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name = "CustomerPassports",
        joinColumns = @JoinColumn(name="customer_fk"),
        inverseJoinColumns = @JoinColumn(name="passport_fk")
    )
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
    ...
}
```

Un `Customer` est lié à un `Passport` à travers une table d'association nommée `CustomerPassports` ; cette table d'association a une colonne de clef étrangère nommée `passport_fk` pointant vers la table `Passport` (matérialisée par l'attribut `inverseJoinColumn`), et une colonne de clef étrangère nommée `customer_fk` pointant vers la table `Customer` (matérialisée par l'attribut `joinColumns`).

Vous devez déclarer le nom de la table de jointure et les colonnes de jointure explicitement dans un tel mapping.

2.2.5.2. Many-to-one

Les associations Many-to-one sont déclarées au niveau de la propriété avec l'annotation `@ManyToOne` :

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

L'attribut `@JoinColumn` est optionnel, la valeur par défaut est comme l'association un à un, la concaténation du nom de la relation du côté propriétaire, _ (underscore), et le nom de la colonne de la clef primaire du côté propriétaire. Dans cet exemple, `company_id` parce que le nom de la propriété est `company` et la colonne identifiante de Company est `id`.

`@ManyToOne` a un paramètre nommé `targetEntity` qui décrit le nom de l'entité cible. Généralement, vous ne

devriez pas avoir besoin de ce paramètre puisque la valeur par défaut (le type de la propriété qui stocke l'association) est correcte dans la plupart des cas. Il est cependant utile lorsque vous souhaitez retourner une interface plutôt qu'une entité normale.

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}

public interface Company {
    ...
```

Vous pouvez sinon mapper une association plusieurs à un avec une table d'association. Cette association décrite par l'annotation `@JoinTable` contiendra une clef étrangère référençant la table de l'entité (avec `@JoinTable.joinColumns`) et une clef étrangère référençant la table de l'entité cible (avec `@JoinTable.inverseJoinColumns`).

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinTable(name="Flight_Company",
        joinColumns = @JoinColumn(name="FLIGHT_ID"),
        inverseJoinColumns = @JoinColumn(name="COMP_ID")
    )
    public Company getCompany() {
        return company;
    }
    ...
}
```

2.2.5.3. Collections

2.2.5.3.1. Vue d'ensemble

Vous pouvez mapper des `Collection`s, des `List`s (ie des listes ordonnées, pas des listes indexées), des `Map`s et des `Set`s. La spécification EJB3 décrit comment mapper une liste ordonnée (ie une liste ordonnée au chargement) en utilisant l'annotation `@javax.persistence.OrderBy` : pour ordonner la collection, cette annotation prend en paramètre une liste de propriétés (de l'entité cible) séparées par des virgules (p. ex. `firstname asc, age desc`) ; si la chaîne de caractères est vide, la collection sera ordonnée par les identifiants. Pour le moment `@OrderBy` fonctionne seulement sur des collections n'ayant pas de table d'association. Pour les véritables collections indexées, veuillez vous référer à Extensions d'Hibernate Annotation. EJB3 vous permet de mapper des `Map`s en utilisant comme clef une des propriétés de l'entité cible avec `@MapKey(name="myProperty")` (myProperty est un nom de propriété de l'entité cible). Lorsque vous utilisez `@MapKey` sans nom de propriété, la clef primaire de l'entité cible est utilisée. La clef de la map utilise la même colonne que celle pointée par la propriété : il n'y a pas de colonne supplémentaire définie pour la clef de la map, et c'est normal puisque la clef de la map représente en fait un propriété de la cible. Faites attention qu'une fois chargée, la clef n'est plus synchronisée avec la propriété, en d'autres mots, si vous modifiez la valeur de la propriété, la clef ne sera pas changée automatiquement dans votre modèle Java (pour une véritable prise en charge des maps veuillez vous référer à Extensions d'Hibernate Annotation). Beaucoup de gens confondent les capacités de `<map>` et celles de `@MapKey`. Ce sont deux fonctionnalités différentes. `@MapKey` a encore quelques limitations, veuillez vous référer au forum ou au système de suivi de bogues JIRA pour plus d'informations.

Hibernate a plusieurs notions de collections.

Tableau 2.1. Sémantique des collections

Sémantique	Représentation Java	Annotations
Sémantique de Bag	java.util.List, java.util.Collection	@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany
Sémantique de Bag avec une clef primaire (sans les limitations de la sémantique de Bag)	java.util.List, java.util.Collection	(@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany) et @CollectionId
Sémantique de List	java.util.List	(@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany) et @org.hibernate.annotations.IndexColumn
Sémantique de Set	java.util.Set	@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany
Sémantique de Map	java.util.Map	(@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany) et (rien ou @org.hibernate.annotations.MapKey/MapKeyManyToMany pour une véritable prise en charge des maps, ou @javax.persistence.MapKey

Donc spécifiquement, les collections java.util.List sans @org.hibernate.annotations.IndexColumn vont être considérées commes des bags.

Les collections de types primitifs, de types core ou d'objets embarqués ne sont pas prises en charge par la spécification EJB3. Cependant Hibernate Annotations les autorise (voir Extensions d'Hibernate Annotation).

```
@Entity public class City {
    @OneToMany(mappedBy="city")
    @OrderBy("streetName")
    public List<Street> getStreets() {
        return streets;
    }
...
}

@Entity public class Street {
    public String getStreetName() {
        return streetName;
    }

    @ManyToOne
    public City getCity() {
```

```
        return city;
    }
    ...
}


@Entity
public class Software {
    @OneToMany(mappedBy="software")
    @MapKey(name="codeName")
    public Map<String, Version> getVersions() {
        return versions;
    }
...
}

@Entity
@Table(name="tbl_version")
public class Version {
    public String getCodeName() {...}

    @ManyToOne
    public Software getSoftware() { ... }
...
}
```

Donc `City` a une collection de `Street`s qui sont ordonnées par `streetName` (de `Street`) lorsque la collection est chargée. `Software` a une map de `Version`s dont la clef est `codeName` de `Version`.

A moins que la collection soit une "generic", vous devrez définir `targetEntity`. C'est un attribut de l'annotation qui prend comme valeur la classe de l'entité cible.

2.2.5.3.2. One-to-many

Les associations one-to-many sont déclarées au niveau propriété avec l'annotation `@OneToMany`. Les associations un à plusieurs peuvent être bidirectionnelles.

2.2.5.3.2.1. Relation bidirectionnelle

Puisque les associations plusieurs à un sont (presque) toujours l'extrémité propriétaire de la relation bidirectionnelle dans la spécification EJB3, l'association un à plusieurs est annotée par `@OneToMany(mappedBy=...)`.

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

`Troop` a une relation bidirectionnelle un à plusieurs avec `Soldier` à travers la propriété `troop`. Vous ne devez pas définir de mapping physique à l'extrémité de `mappedBy`.

Pour mapper une relation bidirectionnelle un à plusieurs, avec l'extrémité one-to-many comme extrémité propriétaire, vous devez enlever l'élément `mappedBy` et marquer l'annotation `@JoinColumn` de l'extrémité