Système d'Information à Objets : Java Persistence API

http://ocaron.polytech-lille.net

Informatique et Statistique 4ème année par apprentissage

Olivier Caron[1]

[1]École d'ingénieurs Polytech Lille
Université de Lille

6 février 2025

© Bjarne Stroustrup

J'ai toujours rêvé que mon ordinateur soit aussi simple que mon téléphone. Ce rêve est devenu réalité : je ne comprends plus comment utiliser mon téléphone.

Critique de JDBC

- 👍 API unique quelque soit le S.G.B.D

Critique de JDBC

- 👍 API unique quelque soit le S.G.B.D
- 👍 Performances, puissance d'expression (compatibilité SQL)

Critique de JDBC

- 👍 API unique quelque soit le S.G.B.D
- 👍 Performances, puissance d'expression (compatibilité SQL)
- 👎 API plus orientée vers SQL que le paradigme Objet

Critique de JDBC

- 👍 API unique quelque soit le S.G.B.D
- 👍 Performances, puissance d'expression (compatibilité SQL)
- 👎 API plus orientée vers SQL que le paradigme Objet

Objectif

Disposer d'un système efficace de persistance d'objets.

La solution : les ORM

La solution : les ORM

La solution : les ORM

- Acronyme pour **O**bject **R**elational **M**apping : couche logicielle d'un programme qui se place en interface entre un programme applicatif et une base de données relationnelle.

La solution : les ORM

- Acronyme pour **O**bject **R**elational **M**apping : couche logicielle d'un programme qui se place en interface entre un programme applicatif et une base de données relationnelle.
- 👍 Paradigme 100% Objet

La solution : les ORM

- Acronyme pour **O**bject **R**elational **M**apping : couche logicielle d'un programme qui se place en interface entre un programme applicatif et une base de données relationnelle.

Paradigme 100% Objet

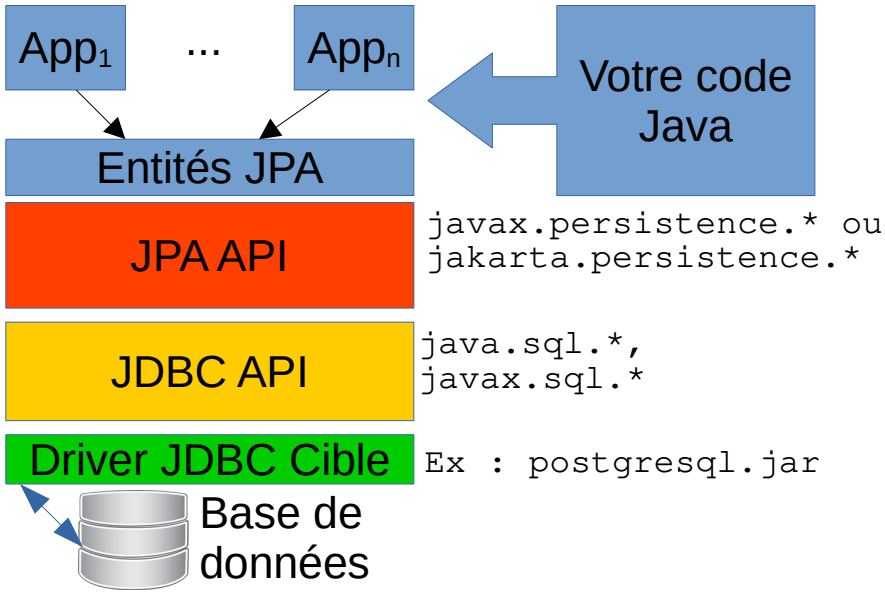
- Découpage des objets entre objets "services" (sessions) et objets "métiers" (entités)

La solution : les ORM

- Acronyme pour **O**bject **R**elational **M**apping : couche logicielle d'un programme qui se place en interface entre un programme applicatif et une base de données relationnelle.
- 👍 Paradigme 100% Objet
- Découpage des objets entre objets "services" (sessions) et objets "métiers" (entités)
- Plusieurs implémentations ORM, ORM Java standard : JPA

La solution : les ORM

- Acronyme pour **O**bject **R**elational **M**apping : couche logicielle d'un programme qui se place en interface entre un programme applicatif et une base de données relationnelle.
- 👍 Paradigme 100% Objet
- Découpage des objets entre objets "services" (sessions) et objets "métiers" (entités)
- Plusieurs implémentations ORM, ORM Java standard : JPA



Les pré-requis Java pour la couche logicielle JPA

- JPA exploite diverses technologies JAVA afin d'analyser puis exécuter les entités JPA :

Les pré-requis Java pour la couche logicielle JPA

- JPA exploite diverses technologies JAVA afin d'analyser puis exécuter les entités JPA :
  - La **généricité** pour gérer les associations entre entités.

Les pré-requis Java pour la couche logicielle JPA

- JPA exploite diverses technologies JAVA afin d'analyser puis exécuter les entités JPA :
  - La **généricité** pour gérer les associations entre entités.
  - L'**introspection** (`Java Reflection`) des entités afin d'analyser dynamiquement leur structure

Les pré-requis Java pour la couche logicielle JPA

- JPA exploite diverses technologies JAVA afin d'analyser puis exécuter les entités JPA :
  - La **généricité** pour gérer les associations entre entités.
  - L'**introspection** (`Java Reflection`) des entités afin d'analyser dynamiquement leur structure
  - L'**invocation dynamique** des objets Java

Les pré-requis Java pour la couche logicielle JPA

- JPA exploite diverses technologies JAVA afin d'analyser puis exécuter les entités JPA :
  - La **généricité** pour gérer les associations entre entités.
  - L'**introspection** (`Java Reflection`) des entités afin d'analyser dynamiquement leur structure
  - L'**invocation dynamique** des objets Java
  - Les **annotations** pour ajouter de la sémantique aux entités JPA

Les pré-requis Java pour la couche logicielle JPA

- JPA exploite diverses technologies JAVA afin d'analyser puis exécuter les entités JPA :
  - La **généricité** pour gérer les associations entre entités.
  - L'**introspection** (`Java Reflection`) des entités afin d'analyser dynamiquement leur structure
  - L'**invocation dynamique** des objets Java
  - Les **annotations** pour ajouter de la sémantique aux entités JPA
  - Mais aussi l'héritage, la surcharge, ...

Introspection et invocation dynamique (1/10)

- Soit le programme `Reflection.java` [1]

1. git clone `https://gitlab.univ-lille.fr/olivier.caron/democoursjpa.git`

Introspection et invocation dynamique (1/10)

- Soit le programme `Reflection.java` [1]
- Ce programme accepte en argument n'importe quel nom de classe (accessible dans le `CLASSPATH`)

1. **git clone** `https://gitlab.univ-lille.fr/olivier.caron/democoursjpa.git`

Introspection et invocation dynamique (1/10)

- Soit le programme `Reflection.java` [1]
- Ce programme accepte en argument n'importe quel nom de classe (accessible dans le `CLASSPATH`)
- Objectif du programme :

1. **git clone** `https://gitlab.univ-lille.fr/olivier.caron/democoursjpa.git`

Introspection et invocation dynamique (1/10)

- Soit le programme `Reflection.java` [1]
- Ce programme accepte en argument n'importe quel nom de classe (accessible dans le `CLASSPATH`)
- Objectif du programme :
  1. Découvrir si la classe en question dispose de propriétés booléennes, entières, réelles ou chaînes de caractères.

1. **git clone** `https://gitlab.univ-lille.fr/olivier.caron/democoursjpa.git`

Introspection et invocation dynamique (1/10)

- Soit le programme `Reflection.java`[1]
- Ce programme accepte en argument n'importe quel nom de classe (accessible dans le `CLASSPATH`)
- Objectif du programme :
  1. Découvrir si la classe en question dispose de propriétés booléennes, entières, réelles ou chaînes de caractères.
  2. Modifier les valeurs des propriétés découvertes.

1. **git clone** `https://gitlab.univ-lille.fr/olivier.caron/democoursjpa.git`

Introspection et invocation dynamique (2/10)

- Pour créer une instance d'une classe :

Introspection et invocation dynamique (2/10)

- Pour créer une instance d'une classe :
  - Cas 1 : le programme connaît la structure de la classe (et notamment son nom) :

```
Object o = new Truc() ;
```

Introspection et invocation dynamique (2/10)

- Pour créer une instance d'une classe :
  - Cas 1 : le programme connaît la structure de la classe (et notamment son nom) :

```
Object o = new Truc() ;
```

  - Cas 2 : le programmeur ne connaît pas la structure mais reçoit un nom de classe

```
Object o = java.beans.Beans.instantiate(null, "Truc");
```

Introspection et invocation dynamique (2/10)

- Pour créer une instance d'une classe :
  - Cas 1 : le programme connaît la structure de la classe (et notamment son nom) :

```
Object o = new Truc() ;
```

  - Cas 2 : le programmeur ne connaît pas la structure mais reçoit un nom de classe

```
Object o = java.beans.Beans.instantiate(null, "Truc");
```

  - **Important :** seules les classes disposant d'une méthode constructeur sans paramètre pourront être analysées.

La classe `java.beans.Beans` (3/10)

- Fournit différents services

La classe java.beans.Beans (3/10)

- Fournit différents services
- La création d'un Beans **peut** ne pas se faire par new

```
package java.beans ;
public class Beans extends Object {
  ...
  public static Object instantiate(ClassLoader cls,
    String beanName)
  throws IOException, ClassNotFoundException ;
  ...
```

La classe `java.beans.Beans` (3/10)

- Fournit différents services
- La création d'un Beans **peut** ne pas se faire par `new`

```
package java.beans ;
public class Beans extends Object {
  ...
  public static Object instantiate(ClassLoader cls,
    String beanName)
  throws IOException, ClassNotFoundException ;
  ...
```

  - Chargement du class loader (`null` équivaut au system class loader)

La classe `java.beans.Beans` (3/10)

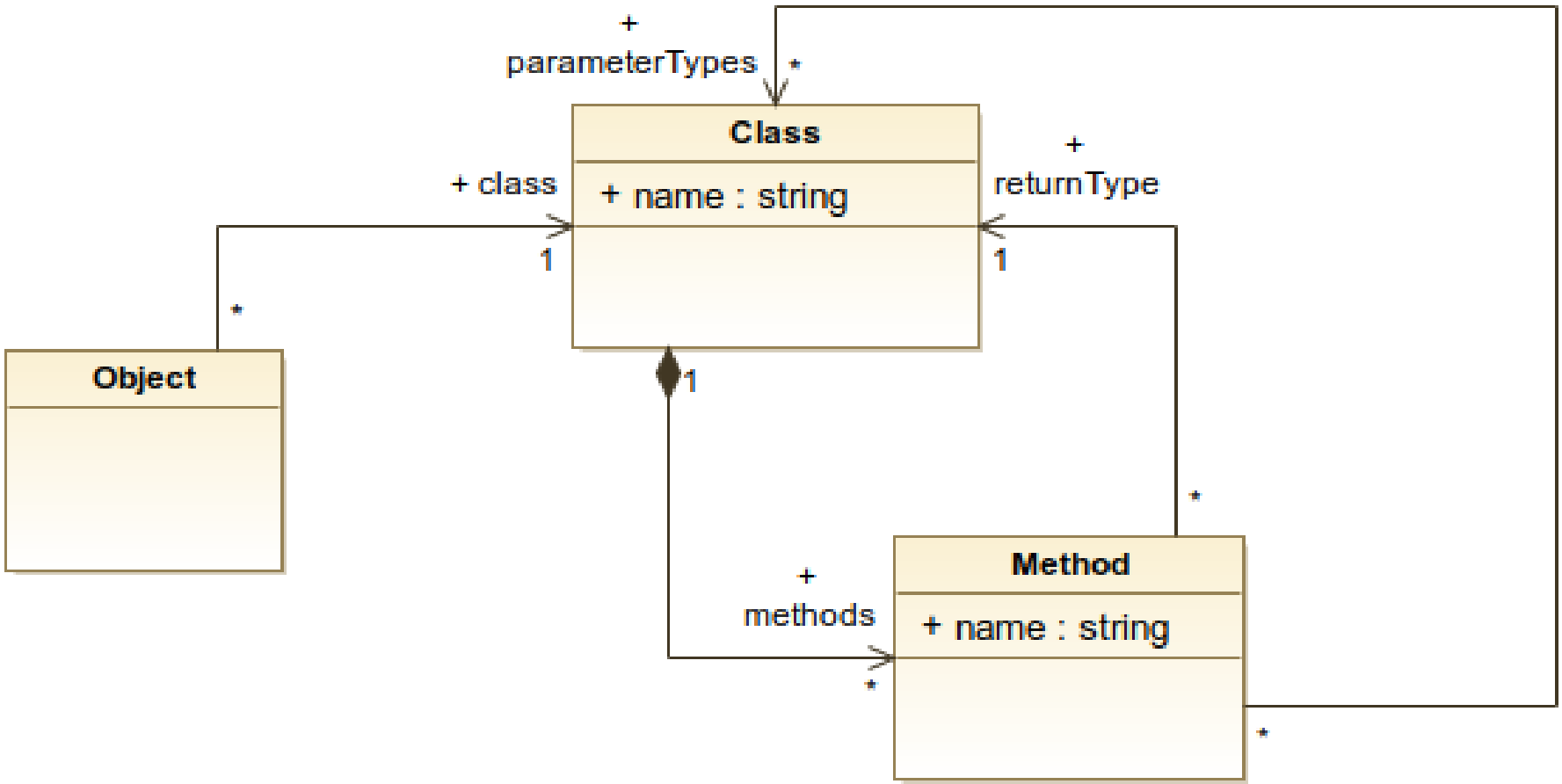
- Fournit différents services
- La création d'un Beans **peut** ne pas se faire par `new`

```
package java.beans ;
public class Beans extends Object {
  ...
  public static Object instantiate(ClassLoader cls,
    String beanName)
  throws IOException, ClassNotFoundException ;
  ...
```

  - Chargement du class loader (`null` équivaut au system class loader)
  - Tentative de charger `beanName.ser` puis création du bean

Extrait du métamodèle Java

- Partie du méta-modèle Utilisé :



Introspection et invocation dynamique (4/10)

java Reflection java.awt.Label

```
-- Partie Introspection de java.awt.Label --
int property found : alignment
java.lang.String property found : text
boolean property found : focusable
boolean property found : visible
boolean property found : enabled
java.lang.String property found : name
-- Partie Invocation dynamique --
invoke method setFocusable with boolean parameter
invoke method setAlignment with int parameter
invoke method setText with java.lang.String parameter
invoke method setEnabled with boolean parameter
invoke method setVisible with boolean parameter
invoke method setName with java.lang.String parameter
```

Introspection et invocation dynamique (5/10)

```
import java.lang.reflect.* ;
import java.beans.Beans ;
import java.util.HashMap;

public class Reflection {
  private HashMap<String, Class> recognizedTypes ;
  private HashMap<Method, String> detectedSetMethods ;

  public Reflection() {
    this.recognizedTypes=new HashMap<String, Class>() ;
    this.recognizedTypes.put("boolean",Boolean.TYPE) ;
    this.recognizedTypes.put("int",Integer.TYPE) ;
    this.recognizedTypes.put("double",Double.TYPE) ;
    this.recognizedTypes.put("java.lang.String",String.class) ;
  }
```

Introspection et invocation dynamique (6/10)

```
public boolean isRecognizedType (String typeName) {
  return this.recognizedTypes.get(typeName) !=null ;
}

public Class [] getParamForSetter(String typeName) {
  return new Class[] { this.recognizedTypes.get(typeName)};
}

public static void main(String args[]) {
  Reflection analyseComponent = new Reflection() ;
  analyseComponent.run(args[0]) ;
}
```

Introspection et invocation dynamique (7/10)

```
public void run(String className) {
 String setName, propertyName ; int ind ;
 this.detectedSetMethods = new HashMap<Method,String>();
 System.out.println("--- Partie Introspection de "+className+" ---")
 try {
  Object obj = Beans.instantiate(null,className) ;
  Class laClasse = obj.getClass() ;
  for (Method m :  laClasse.getMethods()) {
   String typeName=m.getReturnType().getName() ;
   if ((m.getName().startsWith("get") ||
       (m.getName().startsWith("is") &&  typeName.equals("boolean"))
       && m.getParameterTypes().length==0
       && isRecognizedType(typeName)) { // getter method found
     if (typeName.equals("boolean")) ind=2 ; else ind=3 ;
      setName="set"+m.getName().substring(ind) ;
      propertyName = m.getName().substring(ind,ind+1).toLowerCase()
                     + m.getName().substring(ind+1) ;
```

Introspection et invocation dynamique (8/10)

```
        try {
         Method methodeSet=
         laClasse.getMethod(setName,getParamForSetter(typeName));
         detectedSetMethods.put(methodeSet,typeName) ;
         System.out.println(typeName+" property found : "+propertyName) ;
        } catch(NoSuchMethodException e1) {
          // No corresponding setter method found (setName)
        }
    }
}
```

Introspection et invocation dynamique (9/10)

```
System.out.println("--- Partie Invocation dynamique ---") ;
for (Method m : detectedSetMethods.keySet()) {
  String typeName=detectedSetMethods.get(m) ;
  System.out.println("invoke method "+m.getName()
                     +" with "+typeName+" parameter") ;
  if (typeName.equals("boolean")) m.invoke(obj,true) ;
  else if (typeName.equals("int") ) m.invoke(obj,0) ;
  else if (typeName.equals("double") ) m.invoke(obj,0.0) ;
  else if (typeName.equals("String") ) m.invoke(obj,"Hello World !")
}
```

Introspection et invocation dynamique (10/10)

```
} catch (ArrayIndexOutOfBoundsException e) {
  System.err.println("Erreur : java Reflection className") ;
} catch (SecurityException e) {
  System.err.println("exception raised...") ;
} catch(IllegalAccessException e) {
  System.err.println("Access...") ;
} catch(InvocationTargetException e) {
  System.err.println("Argument...") ;
} catch(IllegalArgumentException e) {
  System.err.println("Argument...") ;
} catch(java.io.IOException e) {
  System.err.println("IO...") ;
} catch(ClassNotFoundException e) {
  System.err.println("class ?...") ; } } }
```

Pré-requis Java : les annotations

- Les informations à fournir à la couche logicielle JPA pour que le code métier soit correctement déployé sont définies par **des annotations Java**.

Pré-requis Java : les annotations

- Les informations à fournir à la couche logicielle JPA pour que le code métier soit correctement déployé sont définies par **des annotations Java**.

  👍 Codage très rapide (annotations par défaut)

Pré-requis Java : les annotations

- Les informations à fournir à la couche logicielle JPA pour que le code métier soit correctement déployé sont définies par **des annotations Java**.
  - 👍 Codage très rapide (annotations par défaut)
  - 👎 Nécessite une recompilation des classes si changement.

Pré-requis Java : les annotations

- Les informations à fournir à la couche logicielle JPA pour que le code métier soit correctement déployé sont définies par **des annotations Java**.
  - 👍 Codage très rapide (annotations par défaut)
  - 👎 Nécessite une recompilation des classes si changement.
  - 👎 Toutes les incohérences au niveau des annotations ne peuvent être détectées lors de la compilation.

Les annotations Java

- Introduite à partir de Java 5
- Permet de spécifier des **métadonnées**
- Les annotations peuvent être analysées par introspection
- Certaines annotations standards sont exploitées lors de la compilation :

```
public class Foo1 extends Foo {
   ...
   @Override
   public void bar() ;
}
```

La compilation provoque une erreur si la méthode `bar` n'existe pas dans l'une des sur-classes de `Foo1`.

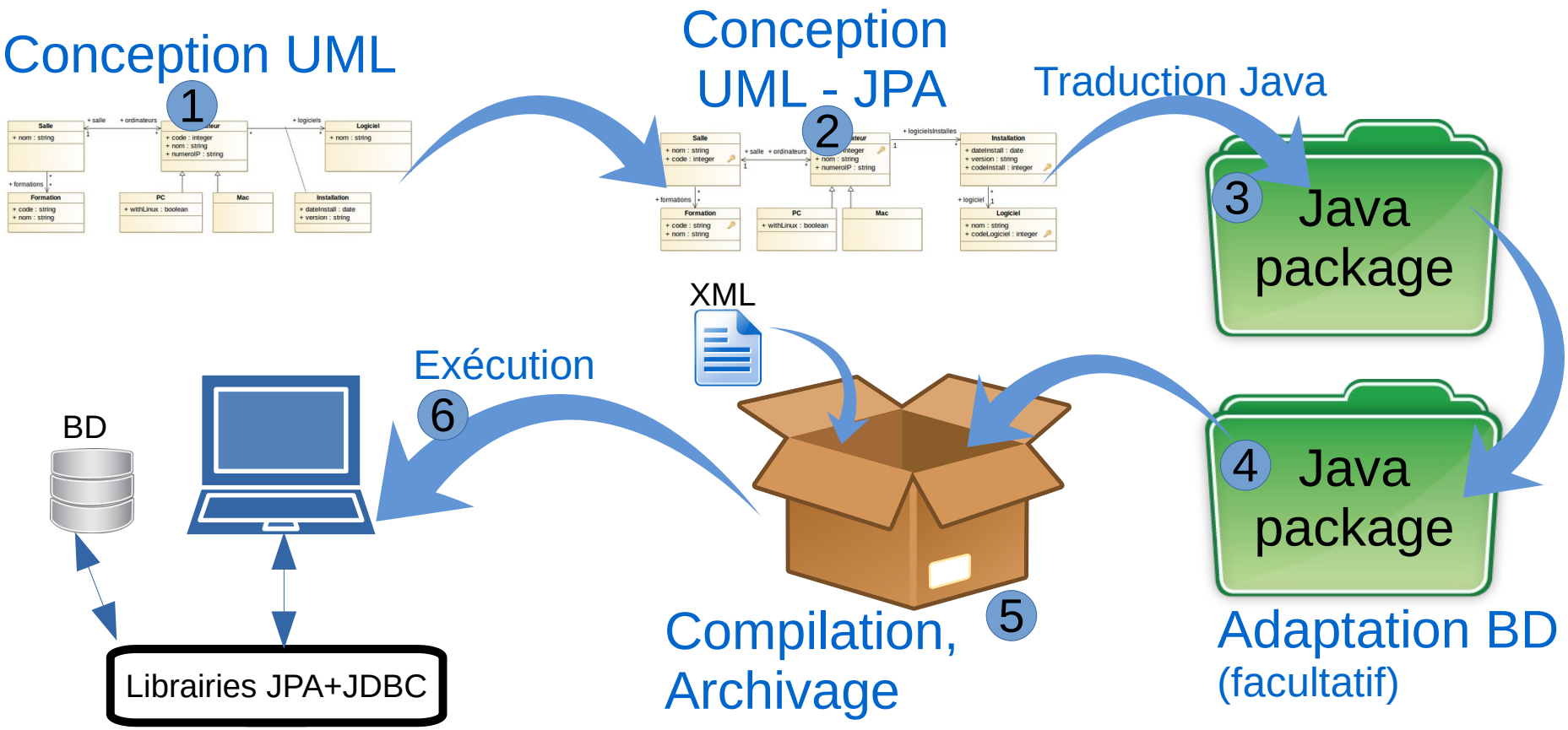
Définition des annotations (par l'exemple)

```java
package fr.polytech.lille.is;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Retention;

// cette annotation sera conservée dans le code source et byte code:
@Retention(RetentionPolicy.RUNTIME)

// cette annotation est uniquement applicable aux méthodes
@Target({ElementType.METHOD})
public @interface Developpeur {
  String createur();
  String[] autresContributeurs() default {}; // optionnel
  String dateDerniereModif();
}
```

Utilisation des annotations (par l'exemple)

```
import fr.polytech.lille.is.Developpeur ;

public class Test {
  ...
   @Developpeur(createur="john", dateDerniereModif="27-03-2017")
   public void process() {...}

   @Developpeur(createur="laura",
                autresContributeurs={"james","john"},
                dateDerniereModif="25-03-2017")
   public void anotherProcess() { ...}
}
```

Processus de développement des entités JPA



Étape 1 : conception UML des objets métiers (Entity)

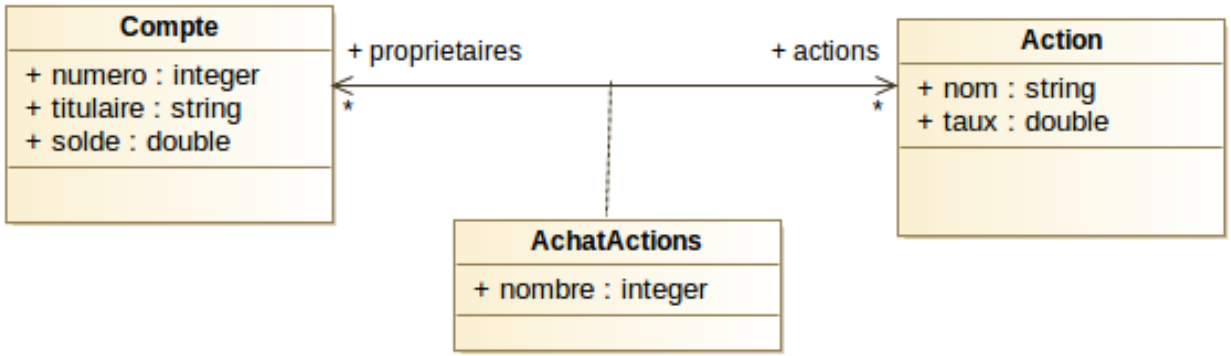
- Sources Java disponibles sur le serveur GitLab de l'université [2]

2. **git clone** `https://gitlab.univ-lille.fr/olivier.caron/democoursjpa.git`

Étape 1 : conception UML des objets métiers (Entity)

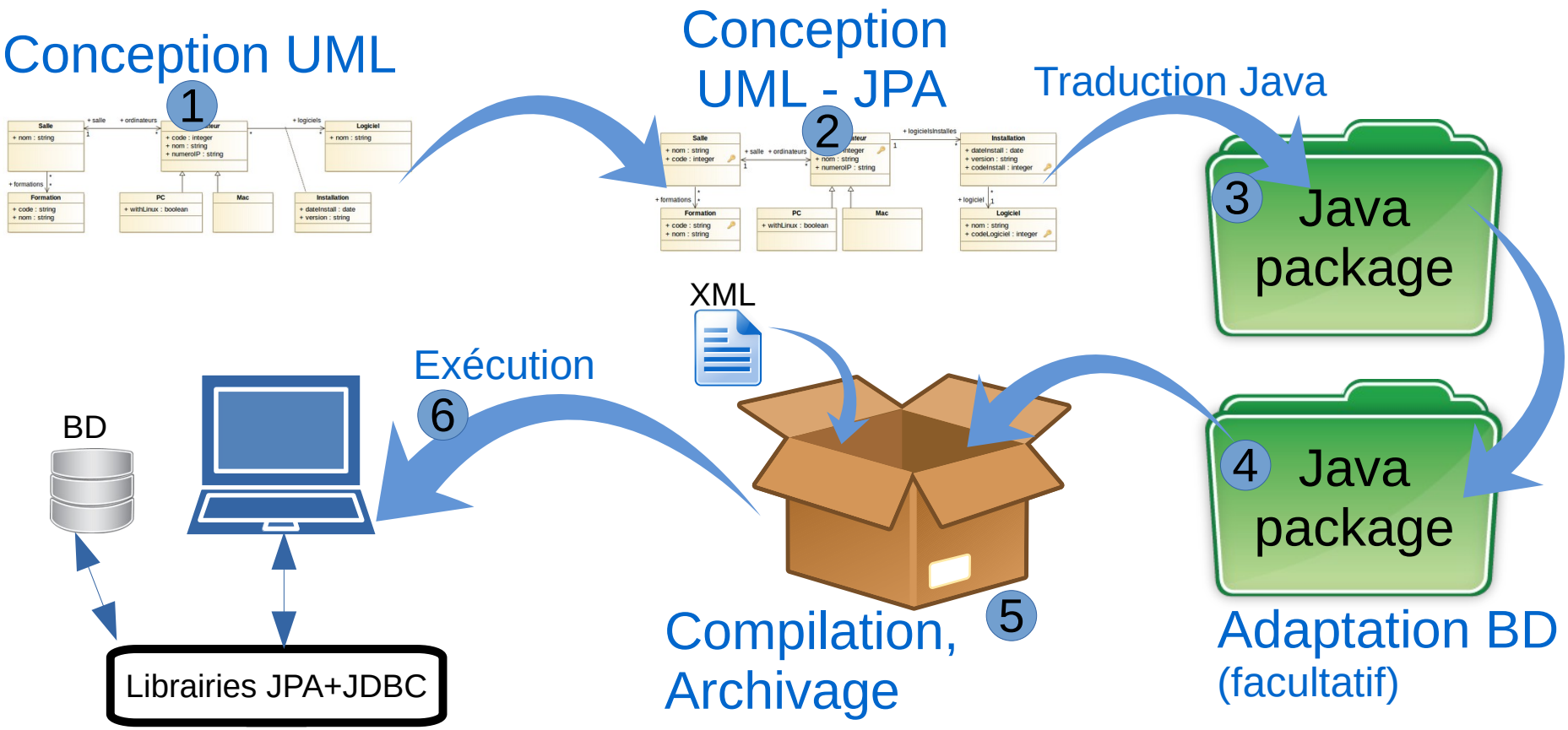
- Sources Java disponibles sur le serveur GitLab de l'université [2]
- Durant cette phase, on se focalise sur les **données** métiers (classes, attributs, associations) et pas sur les **traitements** (méthodes).

Figure – La modélisation UML des entités



2. git clone `https://gitlab.univ-lille.fr/olivier.caron/democoursjpa.git`

Étape 2 : conception UML-**JPA**



Étape 2 : conception UML-**JPA** des objets métiers (Entity) (2/2)

- Modélisation UML non directement exploitable pour JPA :

Étape 2 : conception UML-**JPA** des objets métiers (Entity) (2/2)

- Modélisation UML non directement exploitable pour JPA :
  - Le modèle EJB ne gère que l'héritage simple

Étape 2 : conception UML-**JPA** des objets métiers (Entity) (2/2)

- Modélisation UML non directement exploitable pour JPA :
  - Le modèle EJB ne gère que l'héritage simple
  - Le modèle JPA ne gère **que** des associations binaires

Étape 2 : conception UML-**JPA** des objets métiers (Entity) (2/2)

- Modélisation UML non directement exploitable pour JPA :
  - Le modèle EJB ne gère que l'héritage simple
  - Le modèle JPA ne gère **que** des associations binaires
  - Le modèle JPA ne gère pas les classes-associations (propriétés des associations)

Étape 2 : conception UML-**JPA** des objets métiers (Entity) (2/2)

- Modélisation UML non directement exploitable pour JPA :
  - Le modèle EJB ne gère que l'héritage simple
  - Le modèle JPA ne gère **que** des associations binaires
  - Le modèle JPA ne gère pas les classes-associations (propriétés des associations)
  - Les entités doivent posséder un identifiant

Étape 2 : conception UML-**JPA** des objets métiers (Entity) (1/2)

- Transformation du schéma UML pour devenir compatible JPA :

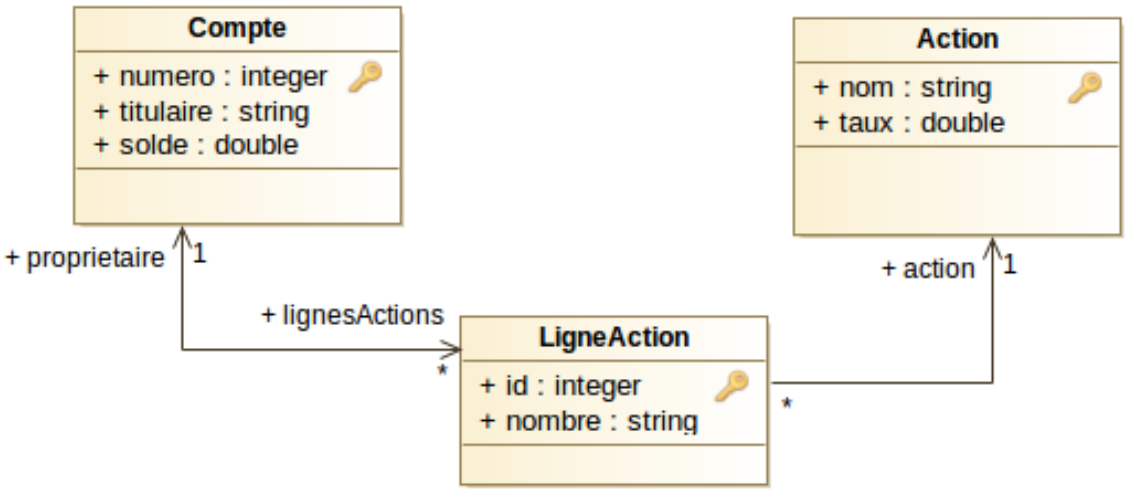
Étape 2 : conception UML-**JPA** des objets métiers (Entity) (1/2)

- Transformation du schéma UML pour devenir compatible JPA :
  - Ajout d'entité(s) pour se substituer aux classes n-aires ($n > 2$) ou aux classes-associations

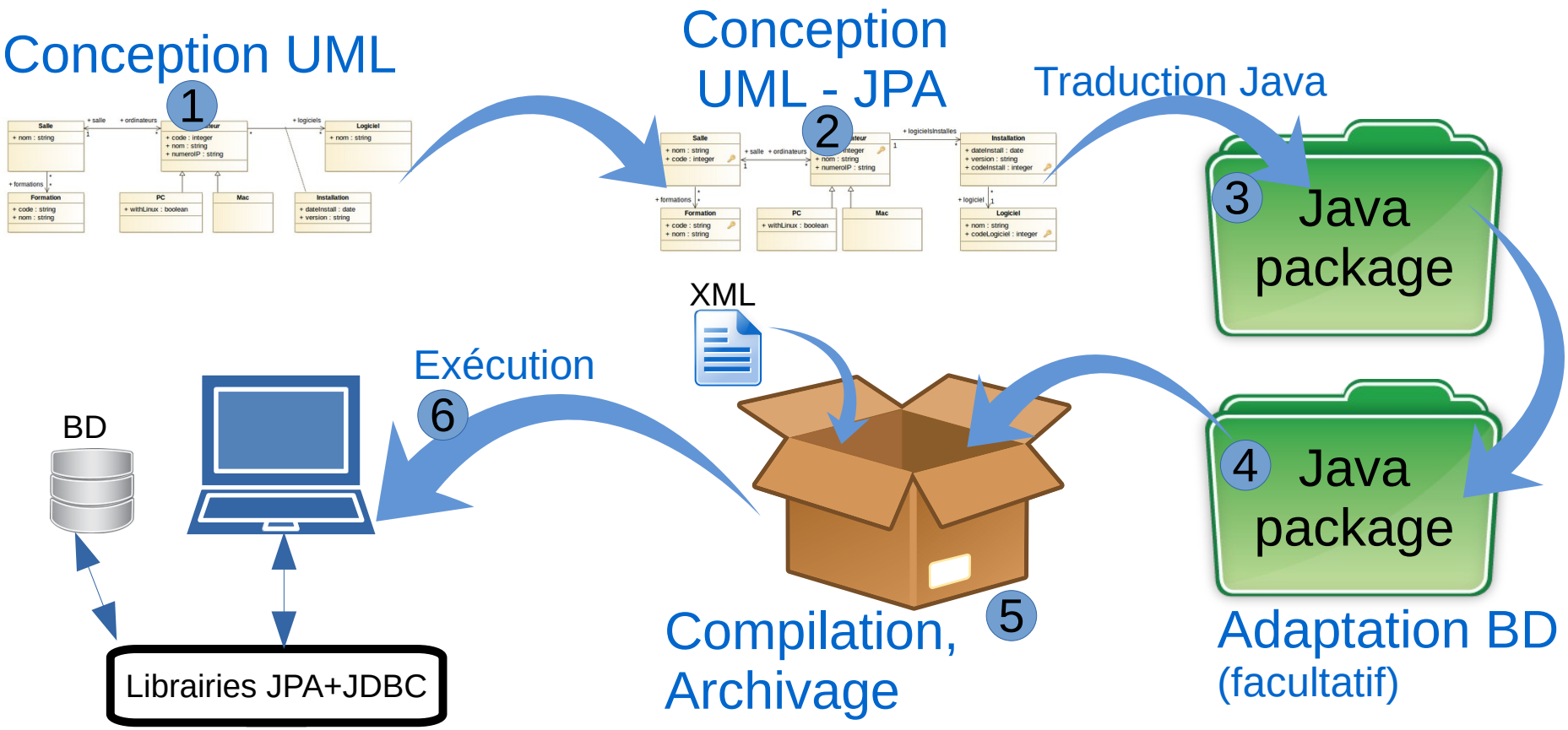
Étape 2 : conception UML-**JPA** des objets métiers (Entity) (1/2)

- Transformation du schéma UML pour devenir compatible JPA :
  - Ajout d'entité(s) pour se substituer aux classes n-aires ($n > 2$) ou aux classes-associations
  - Utilisation d'un stéréotype UML `id` pour définir des identifiants pour chaque entité.

Figure – La modélisation UML-**JPA** des entités



Étape 3 : traduction Java

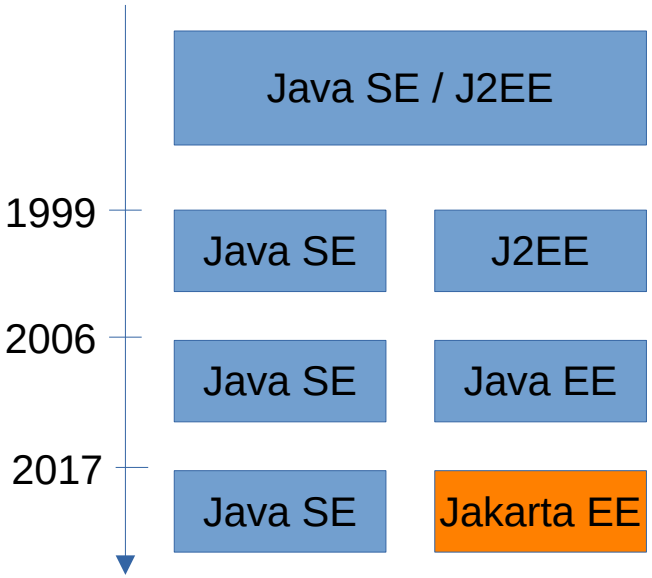


Découpage des distributions Java au fil des ans

Découpage des distributions Java au fil des ans

: droits détenus par Oracle

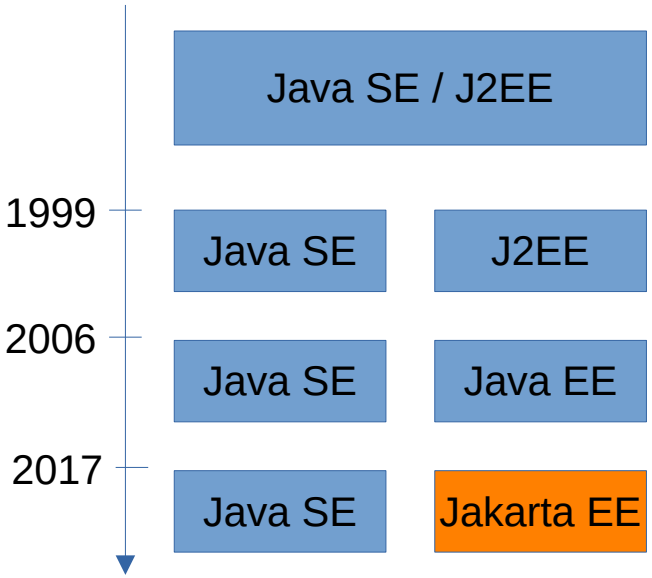
: droits détenus par Eclipse Foundation



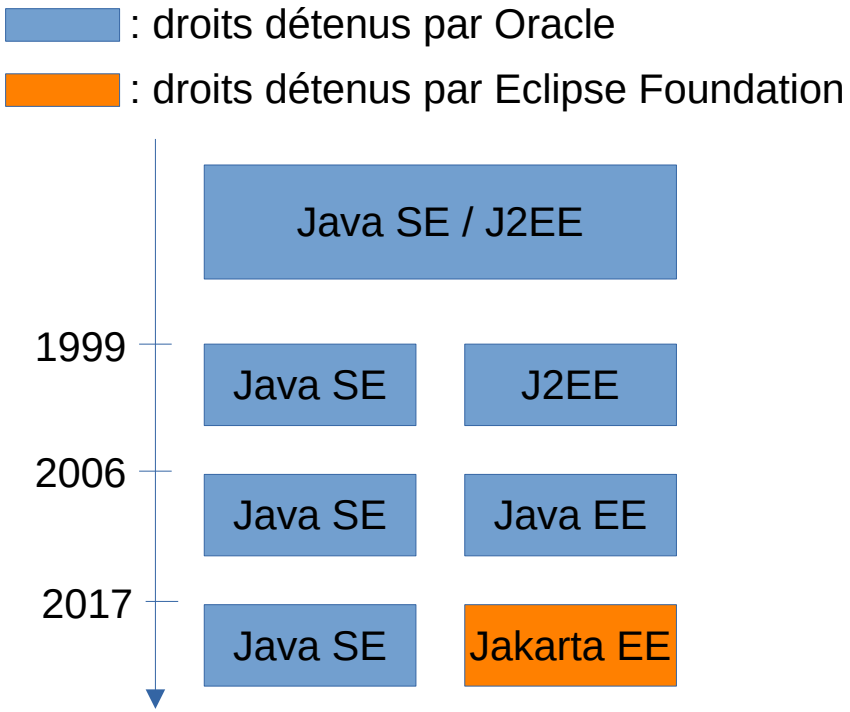
Découpage des distributions Java au fil des ans

: droits détenus par Oracle

: droits détenus par Eclipse Foundation



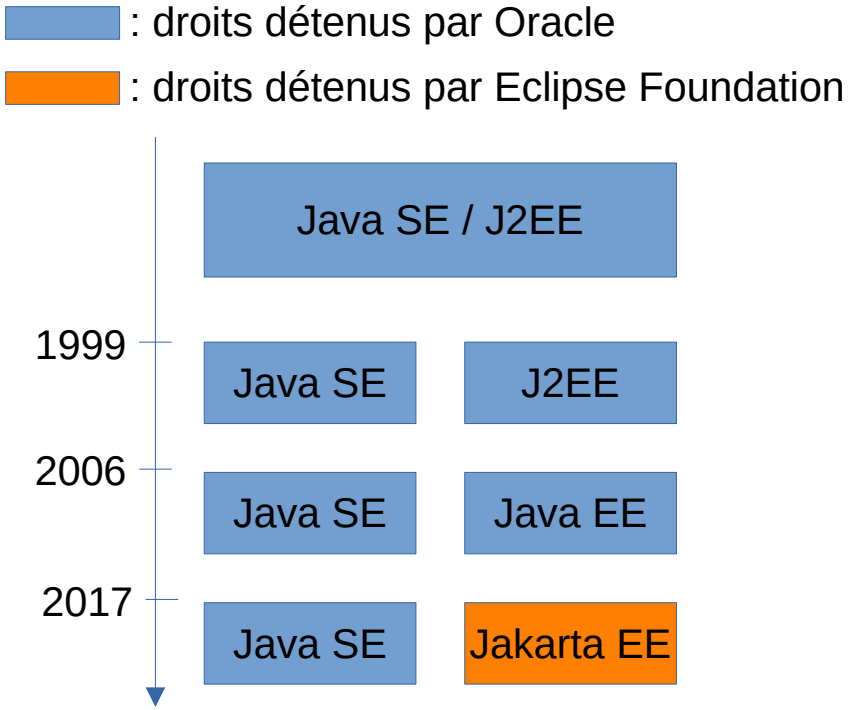
Découpage des distributions Java au fil des ans



- Pour utiliser JPA, avant 2017 :

```
import javax.persistence.* ;
```

Découpage des distributions Java au fil des ans



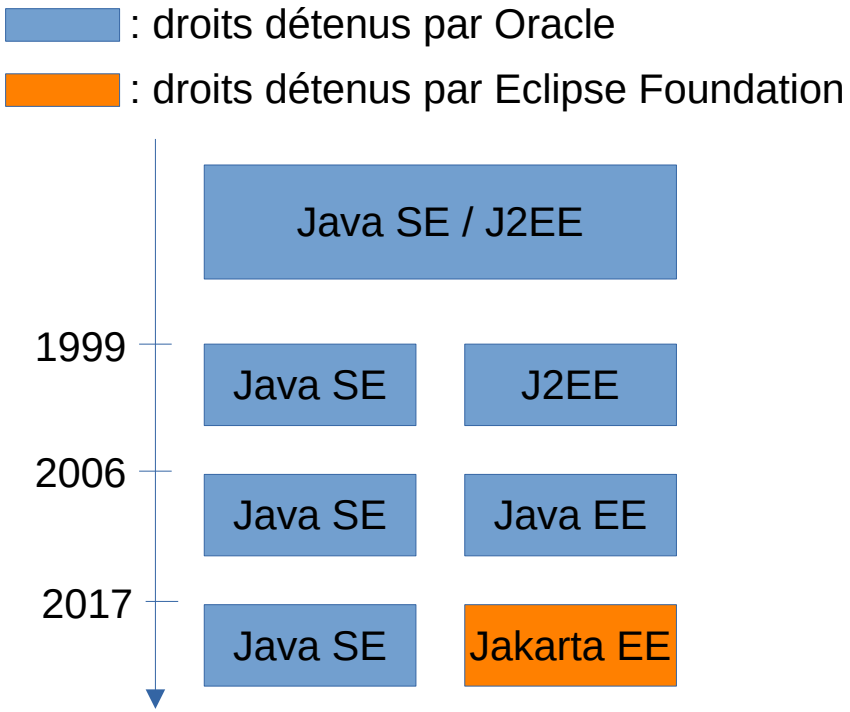
- Pour utiliser JPA, avant 2017 :

```
import javax.persistence.* ;
```

- A partir de 2017 :

```
import jakarta.persistence.* ;
```

Découpage des distributions Java au fil des ans



- Pour utiliser JPA, avant 2017 :

```
import javax.persistence.* ;
```

- A partir de 2017 :

```
import jakarta.persistence.* ;
```

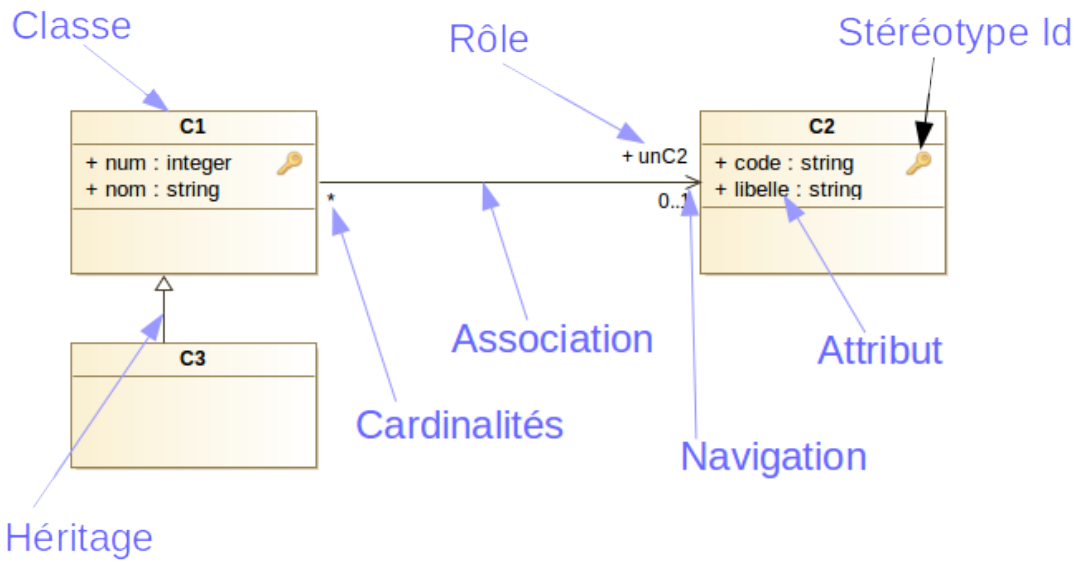
- La programmation ne change pas !

Étape 3 : traduction Java

- Règles de traduction entre concepts UML et concepts Java :

Étape 3 : traduction Java

- Règles de traduction entre concepts UML et concepts Java :
- Les concepts UML :



Étape 3 : traduction Java

- Traduction des classes :
  - Une classe UML devient une classe Java annotée par `@Entity`
  - Fonction constructeur sans paramètre obligatoire (pour introspection)
  - Classe sérializable (propagation réseau)
- Traduction de l'héritage UML via `extends`

```
package demo ;

import jakarta.persistence.Entity ;
import java.io.Serializable ;

@Entity public class C1
  implements Serializable {
  ...
  public C1() {...}
  ...
}
```

```
package demo ;

import jakarta.persistence.Entity ;


@Entity public class C3 extends C1 {
  ...
  public C3() {...}

  ...
}
```

Étape 3 : traduction Java des attributs et identifiants

- Annotation `@Id` pour désigner la clé primaire

```java
package demo ;  /* méthode 1 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {

  @Id private int num ;
  private String nom ;  ...
}
```

```java
package demo ;  /* méthode 2 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {
  @Id public int getNum() { ...}
  public void setNum(int v) { ... }

  public String getNom() {...}
  public void setNom(String v} {...}
```

Étape 3 : traduction Java des attributs et identifiants

- Annotation `@Id` pour désigner la clé primaire
- Annotation `@GeneratedValue` pour générer automatiquement des valeurs entières de clés.

```java
package demo ;  /* méthode 1 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {

  @Id private int num ;
  private String nom ;  ...
}
```

```java
package demo ;  /* méthode 2 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {
  @Id public int getNum() { ...}
  public void setNum(int v) { ... }

  public String getNom() {...}
  public void setNom(String v} {...}
```

Étape 3 : traduction Java des attributs et identifiants

- Annotation `@Id` pour désigner la clé primaire
- Annotation `@GeneratedValue` pour générer automatiquement des valeurs entières de clés.
- La norme JPA 2.0 autorise deux modes de traduction :

```
package demo ;  /* méthode 1 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {

  @Id private int num ;
  private String nom ;  ...
}
```

```
package demo ;  /* méthode 2 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {
  @Id public int getNum() { ...}
  public void setNum(int v) { ... }

  public String getNom() {...}
  public void setNom(String v} {...}
```

Étape 3 : traduction Java des attributs et identifiants

- Annotation `@Id` pour désigner la clé primaire
- Annotation `@GeneratedValue` pour générer automatiquement des valeurs entières de clés.
- La norme JPA 2.0 autorise deux modes de traduction :
  1. Via les variables d'instances (**celle qu'on utilisera**)

```
package demo ;  /* méthode 1 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {

  @Id private int num ;
  private String nom ;  ...
}
```

```
package demo ;  /* méthode 2 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {
  @Id public int getNum() { ...}
  public void setNum(int v) { ... }

  public String getNom() {...}
  public void setNom(String v} {...}
```

Étape 3 : traduction Java des attributs et identifiants

- Annotation `@Id` pour désigner la clé primaire
- Annotation `@GeneratedValue` pour générer automatiquement des valeurs entières de clés.
- La norme JPA 2.0 autorise deux modes de traduction :
  1. Via les variables d'instances (**celle qu'on utilisera**)
  2. Via le "getter" des fonctions accesseurs (getter/setter)

```
package demo ;  /* méthode 1 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {

  @Id private int num ;
  private String nom ;  ...
}
```

```
package demo ;  /* méthode 2 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {
  @Id public int getNum() { ...}
  public void setNum(int v) { ... }

  public String getNom() {...}
  public void setNom(String v} {...}
```

Étape 3 : traduction Java des attributs et identifiants

- Annotation `@Id` pour désigner la clé primaire
- Annotation `@GeneratedValue` pour générer automatiquement des valeurs entières de clés.
- La norme JPA 2.0 autorise deux modes de traduction :
  1. Via les variables d'instances (**celle qu'on utilisera**)
  2. Via le "getter" des fonctions accesseurs (getter/setter)
     Pas de mixage possible des modes dans une même classe

```
package demo ;  /* méthode 1 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {

  @Id private int num ;
  private String nom ;  ...
}
```

```
package demo ;  /* méthode 2 */
import jakarta.persistence.* ;

@Entity public class C1
  implements java.io.Serializable {
  @Id public int getNum() { ...}
  public void setNum(int v) { ... }

  public String getNom() {...}
  public void setNom(String v} {...}
```

Étape 3 : traduction Java des attributs

- **Quelques** types d'attributs possibles autorisés sont :

Étape 3 : traduction Java des attributs

- **Quelques** types d'attributs possibles autorisés sont :
  - Les types primitifs : int, double, boolean, char, etc

Étape 3 : traduction Java des attributs

- **Quelques** types d'attributs possibles autorisés sont :
  - Les types primitifs : int, double, boolean, char, etc
  - Les "wrappers" des types primitifs (Integer, Double, etc)

Étape 3 : traduction Java des attributs

- **Quelques** types d'attributs possibles autorisés sont :
  - Les types primitifs : int, double, boolean, char, etc
  - Les "wrappers" des types primitifs (Integer, Double, etc)
  - Les types objets : `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time` et `java.sql.Timestamp`.

Étape 3 : traduction Java des attributs

- **Quelques** types d'attributs possibles autorisés sont :
  - Les types primitifs : int, double, boolean, char, etc
  - Les "wrappers" des types primitifs (Integer, Double, etc)
  - Les types objets : `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time` et `java.sql.Timestamp`.
  - Des classes "sérializables" définies par l'utilisateur

Étape 3 : traduction Java des attributs

- **Quelques** types d'attributs possibles autorisés sont :
  - Les types primitifs : int, double, boolean, char, etc
  - Les "wrappers" des types primitifs (Integer, Double, etc)
  - Les types objets : `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time` et `java.sql.Timestamp`.
  - Des classes "sérializables" définies par l'utilisateur
  - Les types `byte[]`, `Byte[]`, `char []`, `Character[]`

Étape 3 : traduction Java des attributs

- **Quelques** types d'attributs possibles autorisés sont :
  - Les types primitifs : int, double, boolean, char, etc
  - Les "wrappers" des types primitifs (Integer, Double, etc)
  - Les types objets : `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time` et `java.sql.Timestamp`.
  - Des classes "sérializables" définies par l'utilisateur
  - Les types `byte[]`, `Byte[]`, `char []`, `Character[]`
- Utilisation de l'annotation `@Transient` pour indiquer que la variable d'instance ne doit pas être une donnée persistante.

Étape 3 : traduction Java des associations

- Lien navigable $\rightarrow$ une variable d'instance (nom du rôle)

Étape 3 : traduction Java des associations

- Lien navigable $\rightarrow$ une variable d'instance (nom du rôle)
- Lien non navigable $\rightarrow$ rien

Étape 3 : traduction Java des associations

- Lien navigable $\rightarrow$ une variable d'instance (nom du rôle)
- Lien non navigable $\rightarrow$ rien
- Cardinalité max $> 1 \rightarrow$ variable de type `Collection`, `Set` (souvent), `List` ou `Map`

Étape 3 : traduction Java des associations

- Lien navigable $\rightarrow$ une variable d'instance (nom du rôle)
- Lien non navigable $\rightarrow$ rien
- Cardinalité max $> 1 \rightarrow$ variable de type `Collection`, `Set` (souvent), `List` ou `Map`
- Ajout annotation Java pour les cardinalités de l'association : `ManyToOne`, `ManyToMany`, `OneToMany`, `OneToOne`.

Étape 3 : traduction Java des associations

- Lien navigable $\rightarrow$ une variable d'instance (nom du rôle)
- Lien non navigable $\rightarrow$ rien
- Cardinalité max $> 1 \rightarrow$ variable de type `Collection`, `Set` (souvent), `List` ou `Map`
- Ajout annotation Java pour les cardinalités de l'association : `ManyToOne`, `ManyToMany`, `OneToMany`, `OneToOne`.
- Attribut `mappedBy` de l'annotation pour les associations bi-directionnelles (les deux liens navigables).

Étape 3 : traduction Java des associations

- Lien navigable $\rightarrow$ une variable d'instance (nom du rôle)
- Lien non navigable $\rightarrow$ rien
- Cardinalité max $> 1 \rightarrow$ variable de type `Collection`, `Set` (souvent), `List` ou `Map`
- Ajout annotation Java pour les cardinalités de l'association : `ManyToOne`, `ManyToMany`, `OneToMany`, `OneToOne`.
- Attribut `mappedBy` de l'annotation pour les associations bi-directionnelles (les deux liens navigables).

Mieux comprendre avec

`http://ocaron.polytech-lille.net/enseignement/is2a4/sio/polyJPA.pdf`

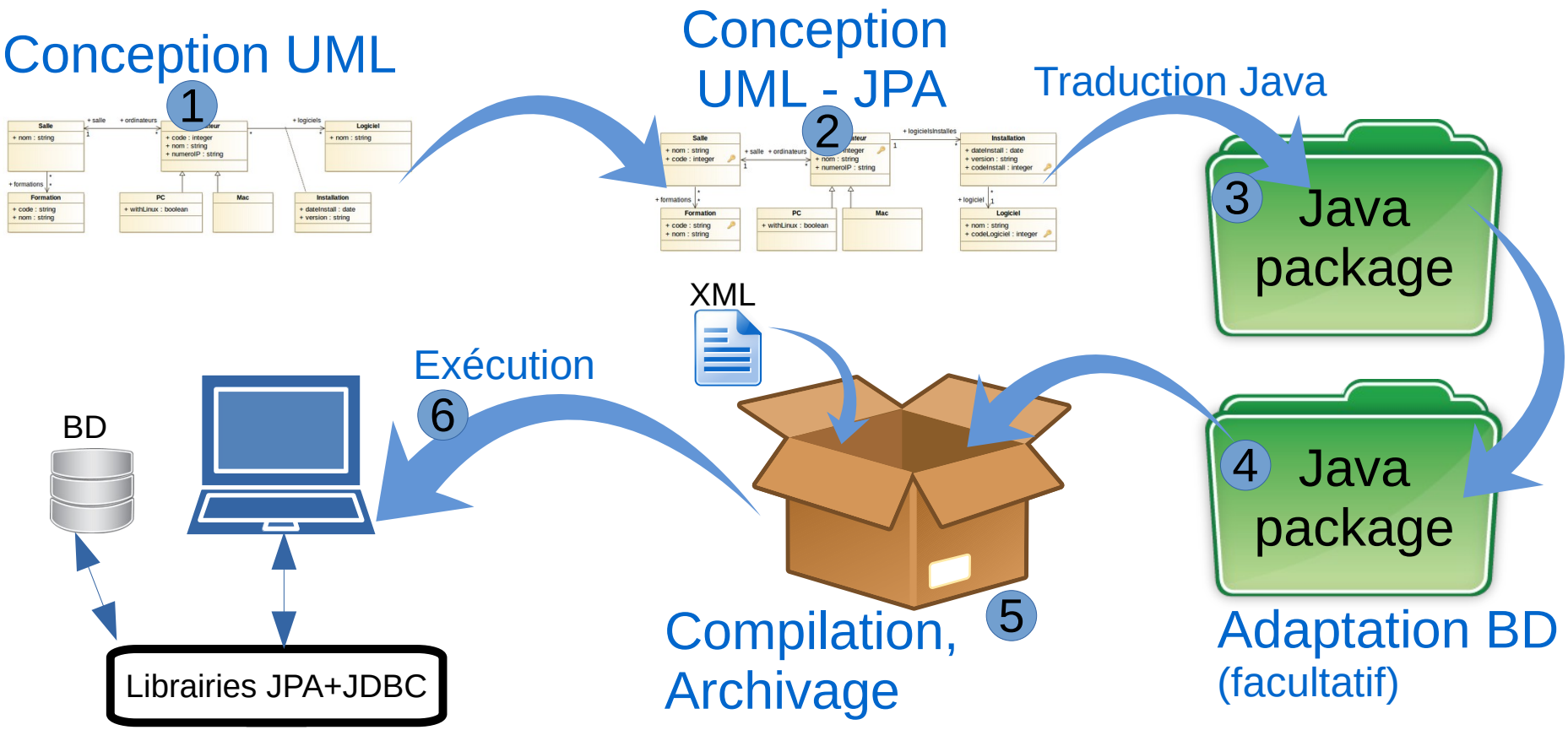
Étape 3 : traduction Java des associations, illustration [3]

```java
package jpa.entites;
import jakarta.persistence.*

@Entity public class LigneAction
 implements java.io.Serializable {

 @Id @GeneratedValue
 private int id ;
 private int nombre ;
 @ManyToOne
 private Compte proprietaire ;
 @ManyToOne(fetch=FetchType.EAGER)
 private Action action ;
 ...
```

```java
package jpa.entites;
import java.io.Serializable ;
import java.util.Set;
import jakarta.persistence.*

@Entity public class Compte
 implements Serializable {
 @Id private int numero ;
 private double solde ;
 private String titulaire ;
 @OneToMany(mappedBy="proprietaire",
            fetch=FetchType.EAGER)
 private Set<LigneAction>
   lignesactions ;
 ...
```

3. `FetchType.EAGER` permet de charger automatiquement en mémoire les objets reliés par association

Étape 4 : Adaptation BD (Base de Données)



Étape 4 : Adaptation BD (Base de Données), problématique

- Les entités Java représentent des objets issus d'une base de données.

Étape 4 : Adaptation BD (Base de Données), problématique

- Les entités Java représentent des objets issus d'une base de données.
- Il existe un mapping (traduction) **par défaut** pour chaque concept.
  Ex : une entité Java de nom `Action` représente une table de nom `Action`.

Étape 4 : Adaptation BD (Base de Données), problématique

- Les entités Java représentent des objets issus d'une base de données.
- Il existe un mapping (traduction) **par défaut** pour chaque concept.
  Ex : une entité Java de nom `Action` représente une table de nom `Action`.
- Aucun problème dans le cas de création d'une base de données (création automatique de la base) ➔étape 4 optionnelle.

Étape 4 : Adaptation BD (Base de Données), problématique

- Les entités Java représentent des objets issus d'une base de données.
- Il existe un mapping (traduction) **par défaut** pour chaque concept.
  Ex : une entité Java de nom `Action` représente une table de nom `Action`.
- Aucun problème dans le cas de création d'une base de données (création automatique de la base) ➔étape 4 optionnelle.
- Que faire lorsqu'on dispose déjà d'une base et qu'on désire **réutiliser** du code JPA ?

Étape 4 : Adaptation BD (Base de Données), problématique

- Les entités Java représentent des objets issus d'une base de données.
- Il existe un mapping (traduction) **par défaut** pour chaque concept.
  Ex : une entité Java de nom `Action` représente une table de nom `Action`.
- Aucun problème dans le cas de création d'une base de données (création automatique de la base) ➔étape 4 optionnelle.
- Que faire lorsqu'on dispose déjà d'une base et qu'on désire **réutiliser** du code JPA ?
  - 👎 Modifier le code Java (plus vraiment de la réutilisation de code)

Étape 4 : Adaptation BD (Base de Données), problématique

- Les entités Java représentent des objets issus d'une base de données.
- Il existe un mapping (traduction) **par défaut** pour chaque concept.
  Ex : une entité Java de nom `Action` représente une table de nom `Action`.
- Aucun problème dans le cas de création d'une base de données (création automatique de la base) ➔étape 4 optionnelle.
- Que faire lorsqu'on dispose déjà d'une base et qu'on désire **réutiliser** du code JPA ?
  - 👎 Modifier le code Java (plus vraiment de la réutilisation de code)
  - 👍 Disposer d'autres règles de mapping

Étape 4 : Adaptation BD (Base de Données), problématique

- Les entités Java représentent des objets issus d'une base de données.
- Il existe un mapping (traduction) **par défaut** pour chaque concept.
  Ex : une entité Java de nom `Action` représente une table de nom `Action`.
- Aucun problème dans le cas de création d'une base de données (création automatique de la base) ➔étape 4 optionnelle.
- Que faire lorsqu'on dispose déjà d'une base et qu'on désire **réutiliser** du code JPA ?
  - 👎 Modifier le code Java (plus vraiment de la réutilisation de code)
  - 👍 Disposer d'autres règles de mapping
- Solution JPA : jeu d'annotations Java pour modifier les règles par défaut

Étape 4 : Adaptation BD des classes

- **Règle par défaut** : le nom de la classe entité correspond au nom de la table.

Étape 4 : Adaptation BD des classes

- **Règle par défaut** : le nom de la classe entité correspond au nom de la table.
- L'annotation `Table` permet de modifier le nom de table par défaut.

```
package jpa.entites;
import jakarta.persistence.*

@Table(name="t_ligne_action")
@Entity public class LigneAction
 implements java.io.Serializable {

  ...
```

Étape 4 : Adaptation BD des attributs

- **Règle 1 par défaut** : le nom de l'attribut correspond au nom de la colonne.

Étape 4 : Adaptation BD des attributs

- **Règle 1 par défaut** : le nom de l'attribut correspond au nom de la colonne.
- **Règle 2 par défaut** : un attribut de type `String` correspond à une colonne de type `char(255)`.

Étape 4 : Adaptation BD des attributs

- **Règle 1 par défaut** : le nom de l'attribut correspond au nom de la colonne.
- **Règle 2 par défaut** : un attribut de type `String` correspond à une colonne de type `char(255)`.
- L'attribut `@Column.name` permet de modifier le nom de la colonne par défaut.

Étape 4 : Adaptation BD des attributs

- **Règle 1 par défaut** : le nom de l'attribut correspond au nom de la colonne.
- **Règle 2 par défaut** : un attribut de type `String` correspond à une colonne de type `char(255)`.
- L'attribut `@Column.name` permet de modifier le nom de la colonne par défaut.
- L'attribut `@Column.length` permet de modifier la taille du type `char` de la colonne par défaut.

```
...
@Entity public class Compte
implements Serializable {
@Id @Column(name="num_compte") private int numero ;
private double solde ;
@Column(name="client", length=50)
private String titulaire ;
```

Étape 4 : Adaptation BD des associations

- **Règle par défaut** : dans une classe, un attribut Java de nom `r1` de type d'une classe `C` correspond à une colonne clé étrangère `r1_id`, `id` étant le nom de la colonne clé primaire de la table associée à `C`.

Étape 4 : Adaptation BD des associations

- **Règle par défaut** : dans une classe, un attribut Java de nom `r1` de type d'une classe `C` correspond à une colonne clé étrangère `r1_id`, `id` étant le nom de la colonne clé primaire de la table associée à `C`.
- L'attribut `@JoinColumn.name` permet de modifier le nom de la colonne par défaut.

```
...
@Entity public class LigneAction
                implements Serializable {
  ...
  @ManyToOne
  @JoinColumn(name="ref_compte") // "proprietaire_numero" -> "ref_compte"
  private Compte proprietaire ;
  @JoinColumn(name="ref_action") // "action_nom" -> "ref_action"
  @ManyToOne(fetch=FetchType.EAGER)
  private Action action ;
```

Étape 4 : Adaptation BD de l'héritage de classes

- L'héritage Java entre JPA entités est possible.

Étape 4 : Adaptation BD de l'héritage de classes

- L'héritage Java entre JPA entités est possible.
- La spécification JPA propose trois mappings :

Étape 4 : Adaptation BD de l'héritage de classes

- L'héritage Java entre JPA entités est possible.
- La spécification JPA propose trois mappings :
  - Une table pour une hiérarchie de classe (**Règle par défaut**)

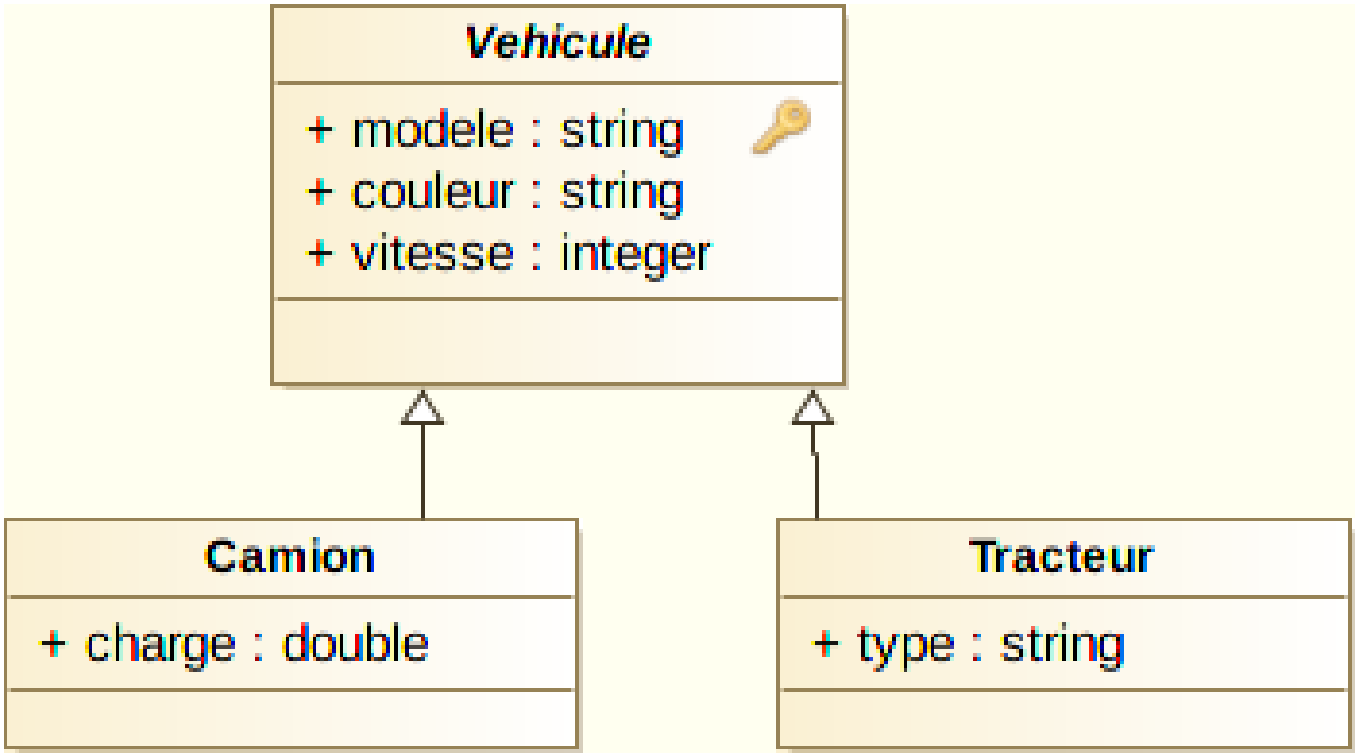
Étape 4 : Adaptation BD de l'héritage de classes

- L'héritage Java entre JPA entités est possible.
- La spécification JPA propose trois mappings :
  - Une table pour une hiérarchie de classe (**Règle par défaut**)
  - Une table par classe concrète

Étape 4 : Adaptation BD de l'héritage de classes

- L'héritage Java entre JPA entités est possible.
- La spécification JPA propose trois mappings :
  - Une table pour une hiérarchie de classe (**Règle par défaut**)
  - Une table par classe concrète
  - Jonctions de tables

Étape 4 : Adaptation BD de l'héritage de classes, exemple



Étape 4 : traduction héritage, option une table par hiérarchie (1/4)

- C'est l'option par défaut sans annotations supplémentaires

Étape 4 : traduction héritage, option une table par hiérarchie (1/4)

- C'est l'option par défaut sans annotations supplémentaires
- Une seule table pour toute la hiérarchie de classes

Étape 4 : traduction héritage, option une table par hiérarchie (1/4)

- C'est l'option par défaut sans annotations supplémentaires
- Une seule table pour toute la hiérarchie de classes
- Une colonne permet de distinguer le type de l'instance (**Règle par défaut** : colonne de nom "dtype" de type chaîne de caractères)

Étape 4 : traduction héritage, option une table par hiérarchie (1/4)

- C'est l'option par défaut sans annotations supplémentaires
- Une seule table pour toute la hiérarchie de classes
- Une colonne permet de distinguer le type de l'instance (**Règle par défaut** : colonne de nom "dtype" de type chaîne de caractères)
- Chaque classe fixe une valeur pour cette colonne (**Règle par défaut** : nom de l'entité)

Étape 4 : traduction héritage, option une table par hiérarchie (2/4)

```
import jakarta.persistence.DiscriminatorColumn;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Inheritance;
import jakarta.persistence.InheritanceType ;

@Entity
@DiscriminatorColumn(name="record_type")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE,
discriminatorValue="vehicule")
public class Vehicule implements java.io.Serializable {

   @Id private String modele ;
   ...
}
```

Étape 4 : traduction héritage, option une table par hiérarchie (3/4)

```java
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Inheritance;

@Entity
@Inheritance(discriminatorValue="camion")
public class Camion extends Vehicule  {
  private float charge ;

  public Camion() {}
  ...
}
```

Étape 4 : traduction héritage, option une table par hiérarchie (4/4)

- Le mapping BD correspondant :

```
                Table "public.vehicule"
   Colonne    |           Type           | Modificateurs
--------------+--------------------------+---------------
 record_type | character varying(10)   | not null
 modele      | character varying(255) | not null
 couleur     | character varying(255) |
 vitesse     | integer                | not null
 charge      | double precision       |
 type        | character varying(255) |

 Index : "vehicule_pkey" PRIMARY KEY, btree (modele)
```

Étape 4 : traduction héritage, option une table par classe (1/4)

- Duplication des propriétés héritées

Étape 4 : traduction héritage, option une table par classe (1/4)

- Duplication des propriétés héritées
- Nécessite d'utiliser l'opérateur algébrique `union` pour collecter les instances de la hiérarchie.

Étape 4 : traduction héritage, option une table par classe (2/4)

```java
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Inheritance;
import jakarta.persistence.InheritanceType ;

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Vehicule implements java.io.Serializable {
   ...
}
```

Étape 4 : traduction héritage, option une table par classe (3/4)

```java
import jakarta.persistence.Entity;

@Entity
public class Camion extends Vehicule  {
   private float charge ;
   ...
}
```

Étape 4 : traduction héritage, option une table par classe (4/4)

- Le mapping BD correspondant :

```
                Table "public.camion"
 Colonne |           Type           | Modificateurs
---------+--------------------------+---------------
 modele  | character varying(255) | not null
 couleur | character varying(255) |
 vitesse | integer                | not null
 charge  | double precision       | not null

Index : "camion_pkey" PRIMARY KEY, btree (modele)
```

Étape 4 : traduction héritage, option jonction de tables (1/4)

- Une table par classe mais pas de duplication de propriétés

Étape 4 : traduction héritage, option jonction de tables (1/4)

- Une table par classe mais pas de duplication de propriétés
- Ajout d'une colonne pour relier les tables

Étape 4 : traduction héritage, option jonction de tables (1/4)

- Une table par classe mais pas de duplication de propriétés
- Ajout d'une colonne pour relier les tables
- Nécessite d'utiliser l'opérateur algébrique de jointure pour collecter les instances de la hiérarchie.

Étape 4 : traduction héritage, option jonction de tables (2/4)

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Inheritance;
import jakarta.persistence.InheritanceType ;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Vehicule implements java.io.Serializable {
  @Id private String modele ;
  private  String couleur ;
  private int vitesse ;
  ...
}
```

Étape 4 : traduction héritage, option jonction de tables (3/4)

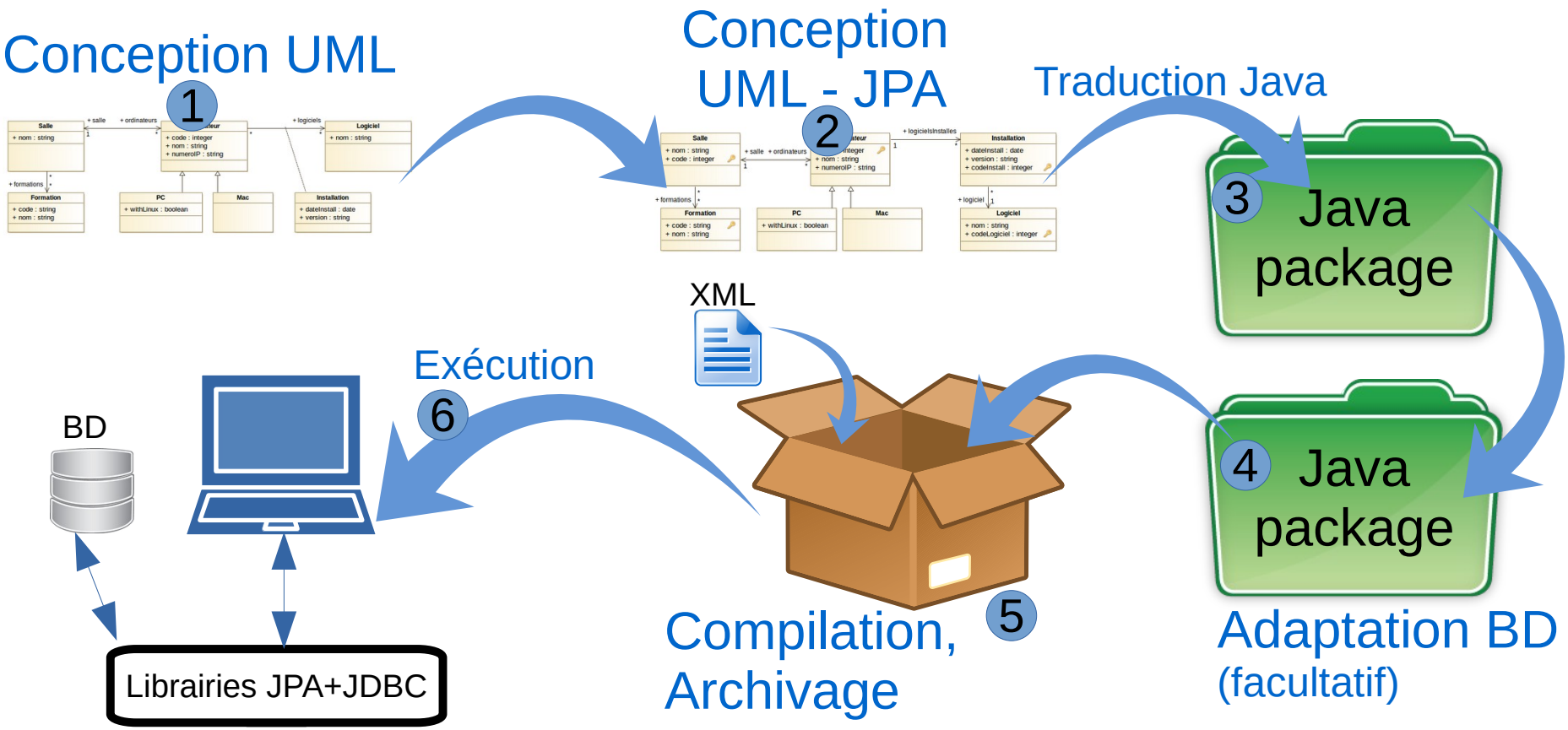
```
import jakarta.persistence.Entity;
import jakarta.persistence.PrimaryKeyJoinColumn;

@Entity
@PrimaryKeyJoinColumn(name="ref_vehicule")
public class Camion extends Vehicule  {
  private float charge ;

  public Camion() {}
  ...
}
```

Étape 4 : traduction héritage, option jonction de tables (4/4)

- Le mapping BD correspondant :

```
  Table "public.camion"
   Colonne    |          Type          | Modificateurs
--------------+------------------------+---------------
 ref_vehicule | character varying(255) | not null
 charge       | double precision       | not null

 Index :"camion_pkey" PRIMARY KEY, btree (ref_vehicule)
 Contraintes de clés secondaires :
 "fkce0ca1498abbb75c" FOREIGN KEY (ref_vehicule)
 REFERENCES vehicule(modele)
```

Étape 5 : Archivage



Étape 5 : Archivage

- Une archive des classes Java compilées ne suffit pas.

Étape 5 : Archivage

- Une archive des classes Java compilées ne suffit pas.
- Dans quelle base de données, le serveur JEE va associer les entités de l'archive ?

Étape 5 : Archivage

- Une archive des classes Java compilées ne suffit pas.
- Dans quelle base de données, le serveur JEE va associer les entités de l'archive ?
- Quel va être le comportement du serveur lors du déploiement/retrait de l'archive ?

Étape 5 : Archivage

- Une archive des classes Java compilées ne suffit pas.
- Dans quelle base de données, le serveur JEE va associer les entités de l'archive ?
- Quel va être le comportement du serveur lors du déploiement/retrait de l'archive ?
- Solution JPA :

Étape 5 : Archivage

- Une archive des classes Java compilées ne suffit pas.
- Dans quelle base de données, le serveur JEE va associer les entités de l'archive ?
- Quel va être le comportement du serveur lors du déploiement/retrait de l'archive ?
- Solution JPA :
  - Un descripteur XML `persistence.xml` va spécifier ces informations.

Étape 5 : Archivage

- Une archive des classes Java compilées ne suffit pas.
- Dans quelle base de données, le serveur JEE va associer les entités de l'archive ?
- Quel va être le comportement du serveur lors du déploiement/retrait de l'archive ?
- Solution JPA :
  - Un descripteur XML `persistence.xml` va spécifier ces informations.
  - Ce descripteur se trouve dans le répertoire `META_INF` situé à la racine de l'archive.

Étape 5 : Archivage, l'outil `jar`

- Implémentation Java structure ZIP

Étape 5 : Archivage, l'outil `jar`

- Implémentation Java structure ZIP
- Java donc multi-plate-forme

Étape 5 : Archivage, l'outil `jar`

- Implémentation Java structure ZIP
- Java donc multi-plate-forme
- exemples utilisation commande :

```
jar cvf mesComposants.jar mesComposants/
jar tvf mesComposants.jar
jar xvf mesComposants.jar
```

Archivage des composants entités

```
mesEntites/
  META-INF/
    persistence.xml
  jpa/
    entites/
      Compte.class
      Action.class
      LigneAction.class

jar cvf mesEntites.jar
   mesEntites/*
```

Fichier persistence.xml (un exemple) (1/2)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
6
7  <persistence-unit name="appliBanque" transaction-type="RESOURCE_LOCAL">
8    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
9
10   <class>jpa.entites.Action</class>
11   <class>jpa.entites.Compte</class>
12   <class>jpa.entites.LigneAction</class>
13
14   <properties>
15     <property name="jakarta.persistence.jdbc.user" value="demo" />
16     <property name="jakarta.persistence.jdbc.password" value="postgres" />
17     <property name="jakarta.persistence.jdbc.driver"
18               value="org.postgresql.Driver" />
19     <property name="jakarta.persistence.jdbc.url"
20               value="jdbc:postgresql://localhost/banque" />
21     <!-- Configuration [4] -->
22     <!-- <property name="eclipselink.logging.level" value="INFO" /> -->
23     <!-- EclipseLink should create the database schema automatically : -->
24     <property name="jakarta.persistence.schema-generation.database.action"
25               value="create"/>
26   </properties>
27 </persistence-unit>
28 </persistence>
```

Fichier persistence.xml (un exemple) (1/2)

- Ligne 7 : définition d'une unité de persistance, fournit un identifiant `appliBanque` qui sera exploité par le gestionnaire d'entités

Fichier persistence.xml (un exemple) (1/2)

- Ligne 7 : définition d'une unité de persistance, fournit un identifiant `appliBanque` qui sera exploité par le gestionnaire d'entités
- Lignes 10-12 : énumération des entités à gérer (attention aux fautes de frappe)

Fichier persistence.xml (un exemple) (1/2)

- Ligne 7 : définition d'une unité de persistance, fournit un identifiant `appliBanque` qui sera exploité par le gestionnaire d'entités
- Lignes 10-12 : énumération des entités à gérer (attention aux fautes de frappe)
- Lignes 15-20 : éléments de connexion à la base de données associée (JDBC)

Fichier persistence.xml (un exemple) (1/2)

- Ligne 7 : définition d'une unité de persistance, fournit un identifiant `appliBanque` qui sera exploité par le gestionnaire d'entités
- Lignes 10-12 : énumération des entités à gérer (attention aux fautes de frappe)
- Lignes 15-20 : éléments de connexion à la base de données associée (JDBC)
- Lignes 24-25 : options d'initialisation des tables, valeurs possibles : `none, create, drop-and-create, drop`

Conclusion

- Ce cours est une **introduction** aux EJB/JPA (EJB = JPA dans un serveur d'applications)

Conclusion

- Ce cours est une **introduction** aux EJB/JPA (EJB = JPA dans un serveur d'applications)
- Très peu de contraintes de programmation

Conclusion

- Ce cours est une **introduction** aux EJB/JPA (EJB = JPA dans un serveur d'applications)
- Très peu de contraintes de programmation
- Configuration avec très peu d'annotations grâce aux règles par défaut.

Conclusion

- Ce cours est une **introduction** aux EJB/JPA (EJB = JPA dans un serveur d'applications)
- Très peu de contraintes de programmation
- Configuration avec très peu d'annotations grâce aux règles par défaut.
- Description (un peu) approfondie dans le poly JPA

Pour finir

Si vous ne réussissez pas du premier coup, appelez ça "Version 1.0"