Projet ENSTA 2021
Application de gestion d'une bibliothèque

Présentation du sujet

L'objectif de ce projet est de vous faire développer une application web de gestion d'une bibliothèque : gestion des membres, des livres référencés, des emprunts. L'application sera réalisée à l'aide de Java EE.

Plusieurs fichiers vous sont fournis, en particulier les fichiers JSP ainsi que les interfaces de DAO et de services. Les JSP seront à compléter pour afficher les informations souhaitées. Vous ne **devez pas** modifier les interfaces.

Vous devrez implémenter toutes les couches nécessaires de l'application afin de la rendre fonctionnelle. Les exercices ci-après sont là pour vous guider dans la réalisation de ce projet.

Les membres ont tous un abonnement (BASIC, PREMIUM ou VIP). Chaque abonnement donne la possibilité d'emprunter plus ou moins de livres en même temps : un abonnement BASIC donne accès à deux emprunts simultanés, un abonnement PREMIUM donne accès à cinq emprunts simultanés, et un abonnement VIP donne accès à 20 emprunts simultanés.

Protocole de rendu

Vous rendrez votre TP par mail, sous forme d'une archive au format zip ou tgz contenant votre dossier de travail (c'est-à-dire le dossier contenant vos fichiers source) ou un lien git. Cette archive devra être nommée en respectant la convention suivante :

- `projet_nom_prenom.zip` (ou `.tgz`, bien sûr)

Sous Linux, afin de générer l'archive, vous pourrez utiliser la commande suivante :

```
tar cvfz nomDeLArchive.tgz dossierDeTravail
```

Sauf mention contraire vous étant communiquée par mail et provenant de l'un des quatre responsables d'IN205, vous enverrez votre mail **avant le 04/04/2021 à 23h59** aux adresses suivantes : aflotte@excilys.com, tbezenger@excilys.com, avan-dalen@excilys.com, amohamed@excilys.com en indiquant dans l'objet de votre mail « *[Rendu projet ENSTA]* ».

Informations utiles

Avant de vous plonger dans le TP, voici quelques informations/rappels utiles :

- Pensez à **documenter votre code** lorsque cela est nécessaire :
    - Faites en sorte d'écrire un code lisible par lui-même (des noms d'attributs et de méthodes clairs, cohérents, qui ont un sens).
    - Pour le code complexe, commentez son fonctionnement.

- Vous trouverez en annexe un certain nombre d'informations utiles pour le projet, en particulier les requêtes SQL pouvant être utilisées dans les DAO pour interagir avec la base de données.

Enfin, remarques importantes :

- Lisez la totalité d'un exercice (ce que vous allez devoir faire et les indications associées) avant de vous lancer dans le code correspondant !

Si vous avez lu tout ce qui précède, vous pouvez attaquer le projet. Bon courage !

Exercice 1 : Posons les bases...

1. Télécharger les fichiers présent sur le git, et décompressé les. Vous pouvez maintenant décompressé Projet2, qui contient l'architecture de base de votre projet.

2. Dans le package `com.ensta.librarymanager.utils`, exécutez le fichier FillDatabase.java en tant qu'application Java standard afin d'initialiser votre base de données. Une fois exécuté, il n'est plus nécessaire de l'exécuter à nouveau plus tard, vous pouvez le faire avec :

   mvn clean install exec:java

   Néanmoins, il peut vous servir à réinitialiser votre base de données avec les valeurs fournies par défaut une fois que vous aurez effectué quelques tests.

Le package `persistence`, situé à l'intérieur du package racine, contient le fichier ConnectionManager.java. Il s'agit de la classe qui vous permettra d'interagir avec la base de données H2.

Dans le dossier src/main/webapp`/WEB-INF`, situé à la racine de votre projet, vous trouverez plusieurs éléments :

- le dossier `View` qui contient le code de toutes les JSP que vous devrez compléter au fur et à mesure du projet.
- le fichier web.xml que vous devrez le compléter au cours du projet.

Indications :

- Bon, soyons honnête : cet exercice comptait pour un total de 0 points. Mais il fallait bien commencer par les fondamentaux ! Et sans avoir cette structure de base, vous ne seriez pas allés bien loin dans le projet. :-)

Exercice 2 : La représentation des données

À présent que notre application est initialisée, nous allons pouvoir y ajouter les premiers éléments de notre application web Java EE : les classes de représentation des données.

1. Créez *dans un endroit approprié* les trois classes permettant de représenter les objets stockés dans les tables Membre, Livre et Emprunt de la base de données.

2. Créez un package `test` dans votre package racine, et créez-y une classe `ModeleTest`. Cette classe vous permettra de vérifier le fonctionnement de vos classes de modèle.

Indications :

- Vous pourrez retrouver en annexe A de ce sujet le schéma des tables de la base de données.

- Le champ **abonnement** de la table **membre** est de type ENUM('BASIC', 'PREMIUM', 'VIP'). Vous pourrez avantageusement utiliser une structure de données similaire en Java. La méthode statique `valueOf()` et la méthode d'instance `name()` pourront vous être utiles dans l'exercice 3.

- La table emprunt contient les identifiants des livres et des membres. Dans notre classe de modèle correspondante, nous préférerions stocker directement des objets de type `Livre` et `Membre`.

- Pour représenter les éléments de type DATETIME de la table Emprunt, vous utiliserez la classe `java.time.LocalDate` en Java. Vous trouverez la JavaDoc à ce sujet ici : https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html.

- Afin de tester vos classes de modèle, il peut être pertinent de redéfinir leur méthode `toString()`. La méthode générée automatiquement par Eclipse est une bonne base et pourra vous faire gagner du temps.

Exercice 3 : L'accès aux données

Ajoutons maintenant la couche d'accès aux données de notre application web Java EE.

1. Copiez-collez *dans un endroit approprié* les trois interfaces de DAO qui vous sont fournies.

2. Créez, *dans un endroit approprié*, la classe `DaoException` qui hérite de la classe `Exception`.

3. Créez trois classes implémentant les interfaces demandées. Vous les nommerez de manière cohérente par rapport aux interfaces qu'elles implémentent (par exemple en ajoutant « Impl » à la fin du nom de l'interface, pour indiquer qu'il s'agit de l'implémentation de cette interface).

   Les méthodes de ces interfaces sont en principe suffisamment bien nommées pour comprendre ce que chacune est censée faire. Nous allons juste mettre en lumière le fait que les méthodes `create()` de `LivreDao` et `MembreDao` ont pour type de retour `int`. En effet, ces deux méthodes retournent l'identifiant du nouvel élément créé en base de données.
   Pour récupérer l'identifiant de l'élément inséré, vous devrez rajouter un paramètre lors de la création de votre Statement. Un exemple vous est fourni en annexe D de ce sujet.

4. Dans votre package `test`, ajoutez une classe `DaoTest` dans laquelle vous pourrez tester le fonctionnement des classes que vous venez d'écrire.

Indications :

- Vous pourrez retrouver en annexe B de ce sujet des requêtes SQL que nous vous proposons d'utiliser.
- Vous respecterez le design pattern **Singleton**. Des indications concernant le fonctionnement et la mise en place standard de ce design pattern sont disponibles en annexe C de ce sujet.
- Dans l'interface `EmpruntDao`, vous remarquerez qu'on n'a pas déclaré de méthode pour la suppression. C'est volontaire : on ne veut pas pouvoir supprimer un emprunt.
- Pour récupérer au format Java `Date` des données stockées dans un champ DATETIME de la base de données, vous pouvez utiliser la méthode `getDate()` du `ResultSet`. Il est ensuite possible de convertir un objet de type `Date` en objet de type `LocalDate` à l'aide de la méthode `toLocalDate()`.

Exercice 4 : Manipulation des données par les services

Maintenant que nous avons mis en place

1. Copiez-collez *dans un endroit approprié* les trois interfaces de services qui vous sont fournies.
2. Créez la classe `ServiceException`, qui hérite de la classe `Exception`.
3. Créez trois classes implémentant les interfaces demandées. Tout comme pour les DAO, vous prendrez garde à nommer vos classes de façon cohérente. Vous mettrez en place dans ces classes de service un système de traitement et de vérification des entrées utilisateur. En particulier :
    a. On empêchera la création ou la mise à jour d'un livre si son titre est vide. Si une telle opération est tentée, on enverra une `ServiceException`.
    b. On empêchera la création ou la mise à jour d'un membre si son nom ou son prénom est vide. Si une telle opération est tentée, on enverra une `ServiceException`.
    c. Lors de la création ou la mise à jour d'un membre, on fera en sorte que son nom de famille soit enregistré en MAJUSCULES dans la base de données.
4. Dans votre package `test`, ajoutez une classe `ServiceTest` dans laquelle vous pourrez tester le fonctionnement des classes que vous venez d'écrire.

Indications :

- De même que dans le cas des DAO, vous respecterez le design pattern **Singleton**.
- Dans l'interface `EmpruntService`, on ne retrouve pas de méthode `update()` comme dans le DAO correspondant. Cependant, on trouve une

méthode `returnBook()` qui va faire appel à la méthode `update()` du DAO en modifiant simplement la date de retour de l'emprunt (la date de retour sera la date courante).

- L'interface `EmpruntService` contient également deux autres méthodes qui ne correspondent pas à ce qui existe dans le DAO : `isLivreDispo(int idLivre)` et `isEmpruntPossible(Membre membre)`.
  La première renvoie un booléen indiquant si le livre indiqué est disponible à l'emprunt ou non (on ne peut évidemment pas emprunter un livre qui n'a pas encore été rendu).
  La seconde renvoie un booléen indiquant si le membre indiqué peut faire un nouvel emprunt ou s'il a atteint le quota imposé par son abonnement.

- L'interface `LivreService` contient une méthode supplémentaire par rapport à son DAO : `getListDispo()`. Cette méthode doit renvoyer la liste des livres actuellement disponibles à l'emprunt. Pour déterminer cette liste, vous pourrez utiliser la méthode `isLivreDispo()` définie dans `EmpruntService`.

- L'interface `MembreService` contient une méthode supplémentaire par rapport à son DAO : `getListMembreEmpruntPossible()`. Cette méthode doit renvoyer la liste des membres qui peuvent réaliser un nouvel emprunt (car n'ayant pas encore atteint leur quota. Pour déterminer cette liste, vous pourrez utiliser la méthode `isEmpruntPossible()` définie dans `EmpruntService`.

Exercice 5 : Interface utilisateur

Nous arrivons quasiment à la dernière étape de notre application web Java EE : la mise en place des servlets.

1. Créez un package `servlet` dans votre package racine, dans lequel vous placerez toutes vos servlets.

2. Les servlets attendues sont nombreuses (12), aussi sont-elles listées ci-dessous, avec une rapide description de leur utilité :

   a. **DashboardServlet** : cette servlet possède une méthode `doGet()` et permet d'afficher les informations du tableau du bord : nombre de livres, nombre de membres, nombre d'emprunts, liste des emprunts en cours.
   Le fichier JSP à afficher est `dashboard.jsp`.

   b. **EmpruntAddServlet** : cette servlet possède deux méthodes : `doGet()` et `doPost()`.
   La méthode `doGet()` permet d'afficher un formulaire d'ajout d'emprunt, basé sur deux champs de type `<select>` qui ne devront contenir respectivement que les livres disponibles à l'emprunt et que les membres pouvant réaliser un nouvel emprunt.
   La méthode `doPost()` a pour unique rôle de traiter le formulaire de création d'un nouvel emprunt à partir des données récupérées via le formulaire précédent. En cas de problème dans le processus de création d'emprunt, vous devrez envoyer une `ServletException`.

Le fichier JSP à afficher pour le formulaire est `emprunt_add.jsp`. Lorsque la création d'un emprunt s'est déroulée correctement, l'utilisateur doit être redirigé vers la liste des emprunts en cours.

c. **EmpruntListServlet** : cette servlet possède une méthode `doGet()` et permet d'afficher la liste des emprunts. Par défaut, elle n'affiche que les emprunts en cours. Si le paramètre `show` est spécifié et est égal à « all », alors la totalité des emprunts doit être affichée.
Le fichier JSP à afficher est `emprunt_list.jsp`.

d. **EmpruntReturnServlet** : cette servlet possède deux méthodes : `doGet()` et `doPost()`.
La méthode `doGet()` permet d'afficher un formulaire de retour d'emprunt, basé sur un champ de type `<select>` contenant les emprunt en cours.
La méthode `doPost()` a pour unique rôle de traiter le formulaire de retour d'un emprunt à partir des données récupérées via le formulaire précédent. En cas de problème dans le processus, vous devrez envoyer une `ServletException`.
Le fichier JSP à afficher pour le formulaire est `emprunt_return.jsp`. Lorsqu'un emprunt est correctement retourné, l'utilisateur doit être redirigé vers la liste des emprunts en cours.

e. **LivreAddServlet** : cette servlet fonctionne sur le même principe que **EmpruntAddServlet**.
Le fichier JSP à afficher pour le formulaire est `livre_add.jsp`. Lorsque la création d'un emprunt s'est déroulée correctement, l'utilisateur doit être redirigé vers la page de détails du livre nouvellement créé. Il faudra donc penser à récupérer son identifiant.

f. **LivreDeleteServlet** : cette servlet possède deux méthodes : `doGet()` et `doPost()`.
La méthode `doGet()` permet d'afficher un formulaire de suppression d'un livre, basé sur un champ de type `<select>` contenant les emprunt en cours. Si un paramètre `id` est transmis dans la requête, alors le livre correspondant devra être présélectionné dans le `<select>`.
La méthode `doPost()` a pour unique rôle de traiter le formulaire de suppresson d'un livre à partir des données récupérées via le formulaire précédent. En cas de problème dans le processus, vous devrez envoyer une `ServletException`.
Le fichier JSP à afficher pour le formulaire est `livre_delete.jsp`. Lorsqu'un emprunt est correctement retourné, l'utilisateur doit être redirigé vers la liste des livres.

g. **LivreDetailsServlet** : cette servlet possède deux méthodes : `doGet()` et `doPost()`.
La méthode `doGet()` permet d'afficher les informations d'un livre, dans un formulaire avec les champs pré-remplis. Ce formulaire permet de mettre à jour les informations du livre. En dessous du formulaire sont affichées les informations relatives à l'emprunt actuel du livre (le cas échéant).

La méthode `doPost()` a pour unique rôle de traiter le formulaire de mise à jour des informations du livre. En cas de problème dans le processus, vous devrez envoyer une `ServletException`.
Le fichier JSP à afficher pour la page de détails et de formulaire est `livre_details.jsp`.
Lorsqu'un livre est correctement mis à jour, l'utilisateur doit être redirigé vers la page de détails du livre (donc la page sur laquelle il se trouvait déjà).

h. **LivreListServlet** : cette servlet possède une méthode `doGet()` et permet d'afficher la liste des livres.
Le fichier JSP à afficher est `livre_list.jsp`.

i. Les servlets correspondant aux membres (c'est-à-dire **MembreAddServlet**, **MembreDeleteServlet**, **MembreDetailsServlet** et **MembreListServlet**) fonctionnent de manière semblable à leurs équivalents pour les livres.

3. Les routes à définir sont les suivantes :

a. `/dashboard` → DashboardServlet

b. `/membre_list` → MembreListServlet
c. `/membre_details` → MembreDetailsServlet
d. `/membre_add` → MembreAddServlet
e. `/membre_delete` → MembreDeleteServlet

f. `/livre_list` → LivreListServlet
g. `/livre_details` → LivreDetailsServlet
h. `/livre_add` → LivreAddServlet
i. `/livre_delete` → LivreDeleteServlet

j. `/emprunt_list` → EmpruntListServlet
k. `/emprunt_add` → EmpruntAddServlet
l. `/emprunt_return` → EmpruntReturnServlet

4. Pensez également à compléter les fichiers JSP lorsque cela est nécessaire.

Indications :

- Afin de vous simplifier la vie, nous vous proposons d'utiliser une « taglib » dans certaines pages JSP. Cette taglib permet de simplifier l'écriture de code Java à l'intérieur des JSP en fournissant des balises de type HTML qui seront remplacées au moment de la compilation. Les informations utiles à ce propos sont disponibles en annexe E de ce sujet.

Si vous avez réussi tous les exercices précédents, vous devriez à présent disposer d'une application fonctionnelle !

Annexes

A. Schéma des tables de la base de données

Membre	Livre	Emprunt
+ INT **id** PRIMARY KEY AUTOINCREMENT + VARCHAR **nom** + VARCHAR **prenom** + TEXT **adresse** + VARCHAR **email** + VARCHAR **telephone** + ENUM **abonnement**	+ INT **id** PRIMARY KEY AUTOINCREMENT + VARCHAR **titre** + VARCHAR **auteur** + VARCHAR **isbn**	+ INT **id** PRIMARY KEY AUTOINCREMENT + INT **idMembre** + INT **idLivre** + DATETIME **dateEmprunt** + DATETIME **dateRetour**

B. Requêtes SQL proposées

Livres

Lister tous les livres

```
SELECT id, titre, auteur, isbn FROM livre;
```

Récupérer un livre par son identifiant

```
SELECT id, titre, auteur, isbn FROM livre WHERE id = ?;
```

Créer un nouveau livre

```
INSERT INTO livre(titre, auteur, isbn) VALUES (?, ?, ?);
```

Mettre à jour un livre

```
UPDATE livre SET titre = ?, auteur = ?, isbn = ? WHERE id = ?;
```

Supprimer un livre

```
DELETE FROM livre WHERE id = ?;
```

Compter le nombre de livres total

```
SELECT COUNT(id) AS count FROM livre;
```

Membres

Lister tous les membres

```
SELECT id, nom, prenom, adresse, email, telephone, abonnement
FROM membre
ORDER BY nom, prenom;
```

Récupérer un membre par son identifiant

```
SELECT id, nom, prenom, adresse, email, telephone, abonnement
FROM membre
WHERE id = ?;
```

Créer un nouveau membre

```
INSERT INTO membre(nom, prenom, adresse, email, telephone,
abonnement)
VALUES (?, ?, ?, ?, ?, ?);
```

Mettre à jour un membre

```
UPDATE membre
SET nom = ?, prenom = ?, adresse = ?, email = ?, telephone = ?,
abonnement = ?
WHERE id = ?;
```

Supprimer un membre

```
DELETE FROM membre WHERE id = ?;
```

Compter le nombre de membres total

```
SELECT COUNT(id) AS count FROM membre;
```

Emprunts

Lister tous les emprunts

```
SELECT e.id AS id, idMembre, nom, prenom, adresse, email,
telephone, abonnement, idLivre, titre, auteur, isbn, dateEmprunt,
dateRetour
FROM emprunt AS e
INNER JOIN membre ON membre.id = e.idMembre
INNER JOIN livre ON livre.id = e.idLivre
ORDER BY dateRetour DESC;
```

Lister les emprunts pas encore rendus

```
SELECT e.id AS id, idMembre, nom, prenom, adresse, email,
telephone, abonnement, idLivre, titre, auteur, isbn, dateEmprunt,
dateRetour
FROM emprunt AS e
INNER JOIN membre ON membre.id = e.idMembre
INNER JOIN livre ON livre.id = e.idLivre
WHERE dateRetour IS NULL;
```

Lister les emprunts pas encore rendus pour un membre donné

```
SELECT e.id AS id, idMembre, nom, prenom, adresse, email,
telephone, abonnement, idLivre, titre, auteur, isbn, dateEmprunt,
dateRetour
FROM emprunt AS e
INNER JOIN membre ON membre.id = e.idMembre
INNER JOIN livre ON livre.id = e.idLivre
WHERE dateRetour IS NULL AND membre.id = ?;
```

Lister les emprunts pas encore rendus pour un livre donné

```
SELECT e.id AS id, idMembre, nom, prenom, adresse, email,
telephone, abonnement, idLivre, titre, auteur, isbn, dateEmprunt,
dateRetour
FROM emprunt AS e
INNER JOIN membre ON membre.id = e.idMembre
INNER JOIN livre ON livre.id = e.idLivre
WHERE dateRetour IS NULL AND livre.id = ?;
```

Récupérer un emprunt par son identifiant

```
SELECT e.id AS idEmprunt, idMembre, nom, prenom, adresse, email,
telephone, abonnement, idLivre, titre, auteur, isbn, dateEmprunt,
dateRetour
FROM emprunt AS e
INNER JOIN membre ON membre.id = e.idMembre
INNER JOIN livre ON livre.id = e.idLivre
WHERE e.id = ?;
```

Créer un nouvel emprunt

```
INSERT INTO emprunt(idMembre, idLivre, dateEmprunt, dateRetour)
VALUES (?, ?, ?, ?);
```

Mettre à jour un emprunt

```
UPDATE emprunt
SET idMembre = ?, idLivre = ?, dateEmprunt = ?, dateRetour = ?
WHERE id = ?;
```

Compter le nombre d'emprunts total

```
SELECT COUNT(id) AS count FROM emprunt;
```

C. Le design pattern Singleton

Le singleton est un design pattern dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires. (https://fr.wikipedia.org/wiki/Singleton_(patron_de_conception))

Il existe plusieurs façons de mettre en place le design pattern Singleton. Vous trouverez ci-dessous la façon que nous vous proposons d'utiliser. Si vous souhaitez vous renseigner sur les autres possibilités, vous pouvez jeter un oeil à l'article suivant :
https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples

Prenons pour exemple une classe que nous nommerons `Singleton`. Nous allons utiliser la « lazy instanciation ». Cela consiste à créer un attribut statique `instance` dont le type est `Singleton`. La classe `Singleton` doit disposer d'exactement un constructeur **privé**. On ajoute à cette classe une méthode statique `getInstance()` ne prenant pas d'argument, et renvoyant l'instance stockée dans l'attribut du même nom. Cette méthode `getInstance()` teste d'abord si l'attribut `instance` est null ou non. Si oui, alors elle appelle le constructeur de la classe pour créer une instance de celle-ci, et la stocke dans l'attribut. Elle renvoie ensuite l'attribut.

Code minimal pour la mise en place du design pattern Singleton :

```
public class Singleton {
    private static Singleton instance;

    private Singleton(){}

    public static Singleton getInstance(){
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

D. Récupérer l'identifiant de l'élément inséré avec JDBC

Pour récupérer l'identifiant de l'élément inséré dans le cadre d'une table possédant un champ `id` autoincrement, vous devez ajouter l'attribut `Statement.RETURN_GENERATED_KEYS` lors de la création de votre objet Statement.

```
PreparedStatement stmt =
connection.prepareStatement("INSERT ...",
```

```
Statement.RETURN_GENERATED_KEYS);
```

Vous pouvez ensuite utiliser l'objet `stmt` de façon normale. Une fois que vous aurez exécuté la requête à l'aide de `stmt.executeUpdate()`, vous serez en mesure de récupérer les éléments générés de la façon suivante :

```
ResultSet resultSet = stmt.getGeneratedKeys();
```

Manipulez ensuite votre ResultSet comme dans le cas d'une requête SELECT : parcourez les enregistrements qu'il contient (si tout s'est bien passé il ne doit y avoir qu'un seul élément dans votre ResultSet), et utilisez les méthodes `getType(element)`.
Exemple :

```
if (resultSet.next()) {
    int id = resultSet.getInt(1);
}
```

E. Utiliser la taglib *core*

Pour manipuler les éléments fournis par la taglib *core* dans une JSP, il est nécessaire de placer le code suivant en début de fichier (avant la balise <html>) :

```
<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"
%>
```

Cela permet d'importer la taglib et de lui associer un raccourci (ici « c »).
Une fois l'import effectué, vous serez en mesure d'utiliser les balises de type `<c:machin>`.

Faire des boucles foreach

On considère que l'attribut items est une liste d'éléments qui a été ajoutée à la requête transmise à la JSP à l'aide du code `request.setAttribute("items", uneListeDElements);`

```
<c:forEach items="${items}" var="item">
...
</c:forEach>
```

Chaque élément de la liste `items` sera considéré comme un élément `item` qui pourra être manipulé à l'intérieur de la boucle, avec les Expression Languages par exemple.

Faire des if

On considère que l'attribut value a été ajouté à la requête transmise à la JSP à l'aide du code `request.setAttribute("value", uneValeur);`

```
<c:if test="${value == false}">
...
</c:if>
```

Si le résultat du test est évalué à vrai, alors le bloc de code contenu entre les balises sera exécuté.