

Outils de productivité pour Java - EI

0. Préliminaires

Installation de l'EI

- Ouvrez un navigateur web (par exemple, Google Chrome).
- Dans la barre d'adresse, tapez l'une des adresses suivantes :
 - zou.ovh/ei345** (*lien principal*)
 - tiny.cc/ei345** (*lien de secours, si le 1^{er} lien ne fonctionne pas*)
- Sur la page web qui s'affiche, cliquez sur le bouton "Télécharger".
- Lorsque le fichier est téléchargé, ouvrez-le pour voir son contenu.
- Double-cliquez sur le fichier **2023-11-i3-outils-java.bat** qu'il contient, afin de démarrer l'installation de l'EI.
- Si le panneau "Windows a protégé votre ordinateur" s'affiche :
 - Cliquez sur le lien "Informations complémentaires".
 - Puis, actionnez le bouton "Exécuter quand même".
- Attendez jusqu'à ce que l'exécution du traitement soit terminée.
- Une fenêtre de l'explorateur de fichiers a dû s'ouvrir pour afficher le contenu du dossier de travail. Attention ! cette fenêtre est peut-être cachée derrière les autres fenêtres.
Ce dossier comporte 2 sous-dossiers :
 - **Sujets-des-TPs** contient les sujets des TPs que vous avez faits.
 - **workspace** est le Workspace Eclipse dans lequel vous allez travailler.
- Double-cliquez sur le raccourci **Eclipse** pour démarrer Eclipse.
- Si tout est OK, on peut couper l'accès au réseau.

Changement de nom du projet ^eleve-

- Faites un clic-droit sur le nom du projet **^eleve-**, puis "Refactor > Rename..."
- À la suite de **^eleve-**, mettez votre nom. Le résultat doit ressembler à **^eleve-Mon-Nom**.

Important ! : Ajoutez bien votre nom au projet **^eleve-**, car c'est le moyen qui permettra de vous identifier pour l'attribution de la note.

NB. Ce projet est vide et n'a pas d'autre utilité que de vous identifier.

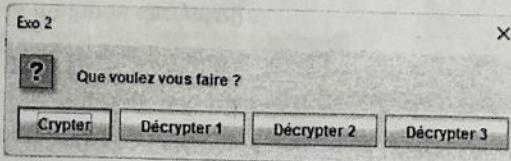
Clean général

- Pour être sûr d'avoir un environnement de travail propre, effectuez l'opération **clean** sur l'ensemble du workspace :
 - Menu "Project > Clean...".
 - Sélectionnez la case "Clean all projects", puis appuyez sur le bouton "Clean".

Thème de l'EI

Des chercheurs expérimentent différentes façons de construire une application capable de crypter et de décrypter un message secret. Une fois crypté le message est stocké dans un fichier afin de pouvoir l'envoyer à un correspondant qui devra le décrypter.

- Dans le projet **exo-3-maven**, dépliez la branche **src/main/java**.
- Dans le package **jrypto**, exécutez la classe **AppliExo3**.



Il s'agit d'un exemple d'application développée par les chercheurs.

Cette application met en œuvre plusieurs objets **Crypteurs** correspondant à des algorithmes cryptographiques différents.

Les messages cryptés sont stockés dans des fichiers nommés **data.serial**.

Lorsqu'on exécute l'application, la vue "Console" permet de visualiser les composants qui ont été instanciés, ainsi que le résultat du cryptage ou du décryptage.

Pour fermer l'application, appuyez sur la touche **Echap** ou cliquez sur la croix en haut et à droite de la fenêtre.

Structure de l'EI

Cette EI est découpée en 4 parties :

1. API Reflection (5 points)
2. Spring (7 points)
3. Maven (2 points)
4. JUnit (6 points)

Les 4 parties sont indépendantes et peuvent être traitées dans n'importe quel ordre.

1. API Reflection

Présentation

Le projet exo-1-reflection contient une version très particulière de l'application qui utilise l'API Reflection.

Cette application comporte plusieurs classes Crypteur qui elles-mêmes comportent plusieurs méthodes.

Lorsqu'elle doit déchiffrer un message contenu dans un fichier, l'application doit se servir de l'API Reflection pour déterminer quelle classe et quelle méthode elle doit utiliser.

Une grande partie du code de l'application est déjà écrite. La partie qui utilise l'API Reflection se trouve dans la classe UtilCrypto et c'est à vous de l'écrire.

Exécution de la classe de test

- Dans le projet exo-1-reflection, dans la branche `src/test/java`, dépliez le package crypto.
- Exécutez la classe `TestUtilCrypto` en tant que test JUnit.
Vous devez constater que tous les tests échouent.

La classe est composée de 5 tests qui correspondent aux 5 méthodes de la classe `UtilCrypto`.

Méthode rechercher()

- Dans la branche `src/main/java`, dépliez le package crypto.
- Ouvrez le code de la classe `UtilCrypto`.
- Écrivez le code de la méthode `rechercher()` :
 - Cette méthode reçoit en paramètres :
 - une variable `type` qui représente une classe,
 - et une variable `data` qui contient un tableau d'objets.
 - La méthode doit rechercher dans le tableau `data`, s'il s'y trouve un objet dont la classe est égale à celle représentée par la variable `type`.
 - Si un objet est trouvé dans la liste, il est retourné par la méthode.
Sinon, la méthode retourne null.
- Si votre code est correct, le premier test de la classe `TestUtilCrypto` doit réussir.

Méthode creerCrypteur()

- Écrivez le code de la méthode `creerCrypteur()` :
 - Cette méthode reçoit en paramètre, une variable qui représente une classe.
 - Elle doit retourner un nouvel objet (une nouvelle instance) de cette classe.
N.B. : Cet objet est créé au moyen d'un constructeur sans paramètre.
- Si votre code est correct, le test n°2 doit réussir.

Méthode setSecret()

- Écrivez le code de la méthode `setSecret()` :
 - Cette méthode reçoit en paramètres, un objet crypteur et une chaîne de caractères qui contient un texte codé, destiné à être décrypté.
 - Elle doit parcourir la liste des variables (champs) déclarés dans la classe de l'objet crypteur.
N.B. : Il faut prendre en compte toutes les variables, y compris celles qui ne sont pas publiques.
 - Lorsqu'on trouve une variable qui porte l'annotation `@Secret` :
 - Il faut affecter à cette variable, la valeur contenue dans le paramètre `code`.
 - Pensez à traiter le cas où la variable est privée. Il faut la rendre accessible avant de pouvoir l'initialiser.
- Si votre code est correct, le test n°3 doit réussir.

Méthode decrypter()

- Écrivez le code de la méthode `decrypter()` :
 - Cette méthode reçoit en paramètre un objet crypteur.
 - Elle doit parcourir la liste des méthodes déclarées dans la classe de l'objet crypteur.
N.B. : Il faut prendre en compte toutes les méthodes, y compris celles qui ne sont pas publiques.
 - Lorsqu'on trouve une méthode qui porte l'annotation `@Decoder` :
 - Il faut l'exécuter.
 - N.B. : il s'agit d'une méthode sans paramètre.
 - Pensez à traiter le cas où la méthode est privée. Il faut la rendre accessible avant de pouvoir l'initialiser.
- Si le code de cette méthode est correct, le test n°4 doit réussir.

Méthode getEnclair()

- Écrivez le code de la méthode `getEnclair()` :
 - Cette méthode reçoit en paramètre un objet **crypteur**.
 - Elle doit parcourir la liste des variables déclarés dans la classe de l'objet crypteur.
N.B. : Il faut prendre en compte toutes les variables, y compris celles qui ne sont pas publiques.
 - Lorsqu'on trouve une variable qui porte l'annotation `@EnClair` :
 - Il faut récupérer le contenu de cette variable et le retourner.
 - Pensez à traiter le cas où la variable est privée. Il faut la rendre accessible avant de pouvoir l'initialiser.
- NB : Vous n'avez pas rencontré ce type de situation en TP. Cependant, si vous avez compris les grands principes de l'API Reflection, vous devriez être en mesure de deviner comment obtenir le résultat souhaité.
- Si aucune variable ne porte l'annotation `@EnClair`, la méthode retourne null.
- Si votre code est correct, le test n°5 doit réussir.

Test final

- Exécutez la classe `AppliExo1`.
Si tout est correct, elle vous affiche dans la vue "Console" le texte du message après l'avoir décrypté.

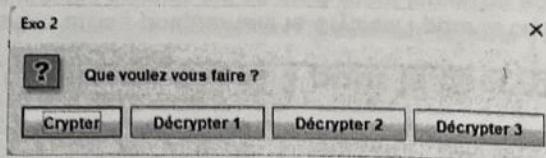
2. Spring

Observation de l'application

Le projet `exo-2-spring`, contient une version de l'application qui est destinée à évaluer l'utilisation de Spring.

Actuellement, l'application n'utilise pas du tout Spring. Cela va être votre travail d'adapter le code pour qu'il fasse appel à Spring.

- Démarrez l'application en exécutant la classe `AppliExo2` et actionnez chacun des 4 boutons.
Seuls les 2 premiers fonctionnent correctement.



Le bouton "Crypter", crée un fichier crypté en utilisant le crypteur n°1.
Dans la vue "Console", on peut voir les composants qui ont été instanciés, ainsi que la version cryptée du message.

Les 3 autres boutons tentent de déchiffrer 3 fichiers qui ont été chiffrés avec les crypteurs n°1, 2 et 3.

Dans la vue "Console", on peut voir la version déchiffrée du message (lorsqu'il n'y a pas d'erreurs).

Actuellement, l'application n'est capable d'utiliser que le crypteur n°1. C'est la raison pour laquelle les 2 derniers boutons échouent.

Ajout des bibliothèques nécessaires

- Faites le nécessaire pour ajouter au projet exo-2-spring, les 2 User Libraries suivantes :
 - Annotations JEE.
 - Spring.

Création de la classe de configuration de Spring

- Dans le package crypto, créez une nouvelle classe nommée **Config1**.
- Ouvrez le code de la classe **Config1**.

Cette classe doit définir la liste des composants instanciés par Spring.

Pour cela, vous allez utiliser la technique qui consiste à écrire une méthode pour chaque composant à instancier. C'est un peu laborieux, mais il n'y a que 5 composants au total.

- Dans la classe **Config1**, écrivez les 5 méthodes qui permettent d'indiquer à Spring comment instancier chacun des composants de l'application.

Voici la liste des 5 composants :

- Manager
- DaoCrypto
- Crypteur1
- Crypteur2
- Crypteur3

Rappel : les noms de méthodes doivent commencer par une minuscule.

- Pensez à ajouter les annotations nécessaires et uniquement celles qui sont nécessaires (Ici, on n'utilise pas l'exploration de packages).
- Faites en sorte que les instantiations soient faites en mode Lazy.

Adaptation de la classe AppliExo2

- Ouvrez le code de la classe `AppliExo2`.
- Au début de la méthode `main()`, faites le nécessaire pour qu'un objet `context`, créé par Spring, soit initialisé en utilisant la classe `Config1` comme classe de configuration.
 - Il faut déclarer une variable `context`.
 - 3 instructions sont nécessaires pour l'initialiser complètement.
- Modifiez la ligne :

```
var manager = new Manager();
```

afin que la variable `manager` ne soit plus initialisée par une opération `new`, mais en faisant appel au `context` fourni par Spring.

Injections de dépendances

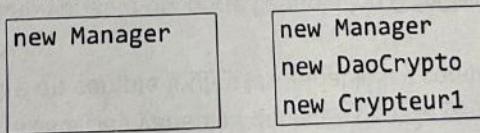
- Dans le package `crypto.core`, ouvrez le code de la classe `Manager`. Observez qu'elle contient un constructeur qui initialise les variables `dao` et `crypteur` grâce à l'opération `new`.
- Supprimez complètement le constructeur (ou mettez-le en commentaire).
- Puis, faites le nécessaire pour que Spring initialise les variables `dao` et `crypteur` par injection de dépendance.

NB. Il n'est pas nécessaire d'initialiser la variable `decrypteur`. Celle-ci est initialisée dans la méthode `decrypter()`, par une instruction :

```
decrypteur = crypteur;
```
- Démarrez l'application. Les 2 premiers boutons doivent fonctionner correctement.

Comportement Lazy

- Démarrez l'application, mais n'actionnez aucun bouton. Appuyez sur la touche **Echap** du clavier pour fermer la boîte de dialogue sans effectuer aucune action.
- Observez le contenu de la vue "Console". Il y a plusieurs possibilités :



- Si l'affichage dans la vue "Console" correspond à la figure de gauche (ci-dessus), c'est parfait. Vos composants ont bien un comportement Lazy. Car, pour afficher la boîte de dialogue, l'application a besoin uniquement du composant `Manager`.
- Si l'affichage dans la vue "Console" correspond à la figure de droite, alors vos composants n'ont pas un comportement Lazy, car ils sont tous instanciés dès le démarrage de l'application.

Dans ce cas, apportez les modifications nécessaires pour que vos composants aient un comportement Lazy et que l'affichage dans la vue "Console" ressemble à la figure de gauche.

Rappel : pour qu'un composant ait un comportement Lazy, 2 conditions sont nécessaires :

1. Dans la classe de configuration de Spring, le composant doit être considéré comme Lazy.
2. À l'endroit où est effectuée l'injection de dépendance, cette injection doit elle-même être considérée comme Lazy.

Lorsque vos composants ont un comportement Lazy, vous devez avoir l'affichage suivant dans la vue "Console" :

Au démarrage de l'application	Après appui sur le bouton "Crypter"
new Manager	new Manager new Crypteur1 CZ6JKxoZUJ2 ... new DaoCrypto

Les composants sont donc bien un comportement Lazy.

Méthodes d'initialisation et de clôture

- Dans la classe Manager, recherchez les 2 méthodes nommées init() et close().
- Faites le nécessaire pour que la méthode init() soit exécutée automatiquement par Spring, juste après l'instanciation du composant Manager.
- Faites le nécessaire pour que la méthode close() soit exécutée automatiquement par Spring, lors de l'arrêt de l'application.
- Dans la classe AppliExo2, à la fin de la méthode main(), faites le nécessaire pour que l'objet context soit informé que l'application est en train de s'arrêter.

Lorsque vous démarrez l'application et que vous n'actionnez aucun bouton, mais que vous appuyez sur la touche [Echap] du clavier, vous devez avoir l'affichage suivant dans la vue "Console".

```
new Manager
==== Démarrage de l'application ====
==== Fermeture de l'application ====
```

Utilisation des crypteurs n°2 et 3 pour le décryptage

Actuellement, l'application utilise uniquement le Crypteur1 pour le cryptage et le décryptage. Cela explique les erreurs qui se produisent lorsqu'on actionne les 2 derniers boutons, puisqu'ils tentent de décrypter des messages qui correspondent aux crypteurs n°2 et 3.

- Dans la classe Manager, ajoutez une nouvelle variable privée nommée context, destinée à contenir l'objet Context fourni par Spring.

Comme type de cette variable, utilisez de préférence l'interface commune à tous les objets Context de Spring.

Faites le nécessaire pour que cette variable soit initialisée par injection de dépendance.

- Puis, recherchez la méthode nommée decrypter().

Cette méthode contient l'instruction :

```
decrypteur = crypteur;
```

À cause d'elle, l'objet qui sert de décrypteur est toujours un Crypteur1. Il va donc falloir modifier cette instruction.

Juste au-dessus, vous voyez une ligne qui définit la variable classe. Cette variable contient la classe qu'il faut utiliser pour pouvoir décrypter correctement le message

- Modifiez l'instruction `decrypteur = ...` en utilisant le context Spring pour récupérer un objet dont la classe est indiquée dans la variable classe.

Lorsque vous exécutez l'application et que vous actionnez les 2 derniers boutons ("Décrypter 2" et "Décrypter 3"), il ne doit plus y avoir d'erreur.

Vous devez constater que ce sont bien les composants Crypteur2 et Crypteur3 qui sont utilisés.

Changement de crypteur pour le cryptage

On souhaite que le cryptage ne soit pas toujours effectué par le Crypteur1.

Le crypteur à utiliser ne sera pas déterminé par un bouton, mais par la configuration de Spring.

- Au début de la classe Manager, observez la déclaration de la variable crypteur. Cette variable est de type Crypteur1. C'est ce qui fait qu'on utilise toujours le crypteur n°1 pour effectuer le cryptage.
- Modifiez le type de la variable crypteur afin qu'elle puisse contenir un objet de n'importe quel type de crypteur (Crypteur1, Crypteur2 ou Crypteur3).
- Démarrez l'application et actionnez le bouton "Crypter". Cela doit générer une erreur car, comme la variable crypteur peut contenir n'importe quel type de crypteur, Spring ne sait pas lequel choisir.

Pour permettre à Spring de savoir quel crypteur utiliser, vous allez utiliser un "Qualifier".

- Appliquez à la variable crypteur, l'annotation suivante :

```
@Qualifier("crypteur")
```

Cette annotation est fournie par le framework Spring.

Elle indique à Spring, qu'il doit utiliser un composant qui porte le même "Qualifier".

- Dans la classe Config1, appliquez cette même annotation à la méthode qui définit le composant de type Crypteur1.
- Exécutez l'application et actionner le bouton "Crypter". Il ne doit plus y avoir d'erreur et vous devez constater que c'est bien le Crypteur1 qui est utilisé.

- Dans la classe Config1, déplacez l'annotation `@Qualifier("crypteur")` afin qu'elle soit appliquée au composant Crypteur2.
- Exécutez l'application et actionner le bouton "Crypter". Vous devez constater que c'est le Crypteur2 qui est utilisé.

Configuration par exploration de packages

Classe de configuration

- Dans le package crypto, créez une nouvelle classe nommée Config2.
- Ouvrez le code de la classe Config2.
- Il vous est demandé que cette classe contienne le moins de code possible. Faites le nécessaire pour qu'il s'agisse d'une classe de configuration qui indique à Spring que les composants doivent être identifiés en utilisant le mécanisme d'exploration de packages. L'instanciation de ces composants doit se faire en mode Lazy. NB. Des pénalités seront appliquées, si cette classe contient du code inutile.
- Ouvrez le code de la classe AppliExo2.
- Faites le nécessaire pour indiquer que Spring doit utiliser la classe Config2 en tant que classe de configuration.

Identification des composants à instancier

Si on exécute l'application à ce stade, cela génère une anomalie. Spring n'est pas capable d'identifier les composants à instancier.

- Faites le nécessaire pour que Spring sache quels sont les composants qu'il doit instancier en utilisant le mécanisme d'exploration de packages. Rappel : il y a 5 composants à instancier.
- Exécutez l'application. Les 3 boutons "Décrypter" doivent fonctionner correctement. Il est probable que le bouton "Crypter" génère une erreur. Vous allez la corriger à l'étape suivante.

Identification du composant crypteur

Si le bouton "Crypter" génère une erreur, c'est parce que Spring cherche un composant annoté avec `@Qualifier("crypteur")`. Or, celle-ci a été indiquée dans la classe Config1 qui n'est plus utilisée.

- Ouvrez le code de la classe Crypteur3.
- Appliquez à cette classe, l'annotation `@Qualifier("crypteur")`.
- Exécutez l'application. Le bouton "Crypter" ne doit plus générer d'erreur et vous devez constater que c'est le Crypteur3 qui est utilisé.

3. Maven

Le projet **exo-3-maven** est un projet Java qui n'utilise pas Maven (pas encore)

Le travail qui vous est demandé ici, est de transformer ce projet en projet Maven et de supprimer l'utilisation des User Libraries.

Transformation du projet en projet Maven

- Faites un clic-droit sur le projet **exo-3-maven**, puis sélectionnez "Configure > Convert to Maven Project".
- Dans la boîte de dialogue, indiquez les paramètres suivant :
 - **Group Id** : **fr.3il.ei**
 - **Artifact Id** : **crypto**
 - **Version** : **2.0.1-SNAPSHOT**Puis, actionnez le bouton "Finish".

Cette opération a créé un fichier **pom.xml** qui, pour le moment, ne contient aucune dépendance. Cela va être votre travail de les ajouter.

Ajout des dépendances dans le fichier POM

Suppression des User Libraries

- Dans le projet **exo-3-maven**, faites le nécessaire pour supprimer du Build Path, toutes les User Libraries. Il y en a 2 :
 - Annotations JEE
 - SpringAttention ! Ne supprimez pas le JRE. Ce n'est pas une User Library.
- Immédiatement, des erreurs apparaissent. Vous allez les faire disparaître en utilisant Maven, lors des prochaines étapes.

Préparation du fichier POM

- Faites en sorte que le fichier **POM** soit ouvert dans la vue centrale d'Eclipse et choisissez l'onglet "pom.xml" au bas de cette vue.
- Placez-vous tout à la fin du fichier et insérez une ligne vide juste avant la balise **</project>**.
- Faites le nécessaire pour que, sur cette ligne, soit écrit le code :
<dependencies></dependencies>
- Puis, insérez une ligne vide entre ces 2 balises **<dependencies>** **</dependencies>**. C'est à cet endroit, entre la balise ouvrante **<dependencies>** et la balise fermante **</dependencies>**, que vous allez insérer les dépendances aux différentes bibliothèques.

Ajout des dépendances dans le fichier POM

Il n'est pas facile d'utiliser Maven dans les conditions où vous êtes, car vous n'avez pas accès à Internet.

On a donc reproduit, en local, des conditions qui simulent les ressources auxquelles vous auriez accès grâce à Internet.

- Dans le volet de gauche d'Eclipse, dépliez le projet ~MVNRepository. Ce projet contient des pages HTML qui ont été capturées sur le site web <https://mvnrepository.com>
- Parmi ces pages HTML, identifiez une qui correspond à une bibliothèque dont votre projet a besoin.
- Double-cliquez sur la page pour l'ouvrir dans un navigateur web et récupérez-y les informations dont vous avez besoin.
- Ensuite, revenez à Eclipse et faites le nécessaire pour ajouter la dépendance dans le fichier POM.
- Ensuite, recommencez. Ajoutez les bibliothèques qui vous paraissent nécessaires pour que l'application puisse fonctionner. C'est-à-dire celles qui remplacent les User Libraries que vous avez supprimées.

Attention ! Une pénalité sera attribuée si vous ajoutez des bibliothèques inutiles.

Test du fonctionnement

- Enregistrez les modifications du fichier POM.
Il ne doit plus y avoir aucune erreur.
- Démarrez l'application en exécutant la classe AppliExo3 pour vérifier qu'elle fonctionne correctement.

Ajout d'une classe de test

- Dans le projet exo-3-maven, recherchez le dossier test qui se trouve à la racine du projet.
- Faites un clic-droit sur le dossier test, puis "Refactor > Move"
- Dans la boîte de dialogue, dépliez le projet exo-3-maven et sélectionnez le dossier src. Actionnez le bouton "OK".
- Une nouvelle branche src/test/java a dû apparaître et celle-ci contient des erreurs. En effet, cette branche contient une classe de test basée sur le framework JUnit 5 et sur Spring.

Faites disparaître ces erreurs en ajoutant au fichier POM, les dépendances nécessaires.

Attention ! Une pénalité sera attribuée si vous ajoutez des dépendances inutiles.

Les erreurs doivent disparaître et on doit pouvoir exécuter la classe de test TestCrypteur1.

4. JUnit

Les chercheurs travaillent sur une nouvelle classe crypteur : **Crypteur4**.
Vous êtes mandaté pour écrire une batterie de tests unitaires.

La classe **Crypteur4** comporte 2 méthodes : **crypter()** et **decrypter()**.

Chacune de ces méthodes reçoit en paramètre une chaîne de caractères et retourne une autre chaîne de caractères qui contient le résultat du traitement.

Vous allez devoir tester la validité de ces 2 méthodes.

- Dans le projet **exo-4-junit**, dépliez la branche **src/test/java**.

• Ouvrez le code de la classe **TestCrypteur4**.

Une partie du code a déjà été écrit, mais il est inachevé.

- Au début de la classe se trouvent 2 tableaux : **enClair** et **cryptes**.

Ils contiennent des valeurs que vous pourrez utiliser pour définir vos tests.

Le tableau **enClair** contient 3 chaînes qui sont destinées à être cryptées.

Le tableau **cryptes** contient le résultat attendu de leur cryptage. Cela vous permettra de tester si les méthodes **crypter()** et **decrypter()** fonctionnent correctement.

Test n°1

- Observez le code de la méthode **test1_crypter_null()**.

Elle contient déjà 2 instructions qui permettent de déclarer une variable **crypteur** et de l'initialiser.

Le but de ce test est de vérifier que, si on passe une valeur null à la méthode **crypter()**, elle retourne la valeur null.

Voici le travail qui vous est demandé :

- Faites le nécessaire pour que la méthode **test1_crypter_null()** soit considérée par JUnit comme une méthode de test.

- Complétez son code pour qu'il vérifie que :

- si on exécute la méthode **crypter()** en lui passant null,
- alors la méthode retourne bien null.

- Exécutez la classe de test (avec JUnit) pour vérifier que le test fonctionne et qu'il réussit.

Test n°2

- Adaptez le code de la méthode **test2_crypter_non_null()** pour que celle-ci teste que, si on exécute la méthode **crypter()** en lui passant une valeur autre que null, elle retourne une valeur qui n'est pas null.

- Vérifiez que le test fonctionne et qu'il réussit.

Conseil : Dans la vue qui affiche les résultats de l'exécution des tests, vérifiez qu'il y a bien deux tests qui ont été exécutés (et non un seul).

Test n°3

- Adaptez le code de la méthode `test3_crypter_ok()` pour que celle-ci vérifie que, si on exécute la méthode `crypter()` en lui passant la chaîne "coucou", elle retourne la chaîne "7W3ok7CHdF4=".
- Vérifiez que le test fonctionne et qu'il réussit.

Tests n°1, 2 et 3

Une contrainte supplémentaire vous est imposée pour les tests n°1, 2 et 3 que vous venez d'écrire.

Pour effectuer chacun d'eux, vous avez dû faire appel à une méthode "assert" fournie par JUnit.

Il est demandé que pour faire ces 3 tests vous utilisiez 3 méthodes "assert" différentes. Si c'est déjà le cas, c'est parfait. Sinon apportez les modifications nécessaires.

Mutualisation de code

Élimination des redondances de code

Comme vous l'avez remarqué, les méthodes de test commencent toutes par la même série de 2 instructions qui initialisent l'objet `crypteur`.

On souhaite éviter cette redondance de code. Voici quelques conseils

- Créez 1 variable privée `crypteur` au niveau de la classe.
- Créez une méthode nommée `avantChaqueTest()` et faites le nécessaire pour que :
 - Cette méthode initialise la variable `crypteur`.
 - Cette méthode soit exécutée automatiquement par JUnit juste avant chacun des tests.
- À la fin, supprimez de chaque méthode de test, les 2 premières instructions qui sont devenues inutiles.
- Exécutez la classe de test et vérifiez que les 3 tests ont bien été effectués et qu'ils ont tous réussi.

Mesure de la durée totale des tests

On souhaite afficher dans la vue "Console", la durée totale mise par le traitement pour exécuter l'ensemble des tests.

Cette durée sera exprimée en millisecondes.

- Déclarez une variable privée de type `long`, nommée `débutSérie`.
Cette variable contiendra le moment du début de la série de tests.

- Créez une méthode nommée `avantLesTests()`.
 - Faites en sorte que cette méthode soit exécutée automatiquement par JUnit, une seule fois, juste avant le tout premier test.
 - Dans cette méthode, initialisez la variable `debutSerie`.

```
debutSerie = System.currentTimeMillis();
```

- Exécutez les tests pour vérifier que cela ne génère pas d'erreurs.

S'il y a une erreur, apportez la correction nécessaire.

Rappel : À gauche, dans la sous-vue "Failure Trace", vous pouvez cliquer sur la 1^{re} icône dans la barre de titre de cette sous-vue (icône "Show Stack Trace in Console View"). Cela permet d'afficher la trace complète de l'exception dans la vue "Console" d'Eclipse.

- Ensuite, écrivez une méthode nommée `apresLesTests()`.

- Faites en sorte que cette méthode soit exécutée automatiquement par JUnit, une seule fois, juste après la fin de l'exécution du dernier test.
- Dans cette méthode, calculez la durée mise pour exécuter l'ensemble des tests et affichez-la dans la vue "Console".

```
var duree = System.currentTimeMillis() - debutSerie;  
System.out.println("Durée totale : " + duree + " ms");
```

- Exécutez les tests pour vérifier que la durée est affichée correctement.

Mesure de la durée de chacun des tests

À présent, on veut que soit affichée la durée d'exécution de chacun des tests.

- Déclarez une variable privée de type `long`, nommée `debutTest`.
Cette variable permettra de stocker le moment de début du test en cours d'exécution.
- Dans la méthode `avantChaqueTest()`, écrivez le code qui initialise la variable `debutTest` en lui affectant la valeur du temps courant (`currentTimeMillis()`).
- Écrivez une méthode nommée `apresChaqueTest()`, en vous inspirant du code suivant :

```
public void apresChaqueTest( TestInfo testInfo ) {  
    var duree = ... ...  
    System.out.println( testInfo.getDisplayName() + " : " + duree + " ms" );  
}
```

La variable `duree` doit contenir la durée d'exécution du test courant.

La variable `testInfo` permet d'obtenir le nom à afficher pour le test courant.

- Faites en sorte que cette méthode soit appelée automatiquement par JUnit, juste après l'exécution de chacun des tests.
- Exécutez les tests pour vérifier que la durée est affichée correctement.
- Vous devez obtenir un affichage similaire à celui qui est présenté ci-dessous.
NB. Les durées dépendent de la rapidité de votre machine.

Test n°4

La méthode `decrypt`

Si on lui passe une exception de type

- Dans la méthode `decrypt`, ajoutez une chaîne de caractères.
- Faites le nécessaire pour gérer cette exception.

Test n°5

Ici, il vous est demandé

On souhaite vérifier que le même test, avec

- L'idée est de faire des tests similaires pour les autres méthodes.
- Configurez la méthode `debutTest` pour qu'un test passe.
- Écrivez la méthode `apresChaqueTest` en utilisant la variable `debutTest`.
NB. Il y a une erreur dans ce code.
- Le test n°5 devrait maintenant réussir.

```
test3_crypter_ok() : 12 ms
test1_crypter_null() : 1 ms
test2_crypter_non_null() : 1 ms
Durée totale : 73 ms
```

Test n°4

- La méthode `decrypter()` permet de décoder la version cryptée d'un texte. Si on lui passe une chaîne de caractères qui contient une valeur erronée, elle émet une exception de type `CryptoException`.
- Dans la méthode `test4_decrypter_erreur()`, exécutez la méthode `decrypter()` en lui passant une chaîne de caractères quelconque, par exemple "xxx".
 - Faites le nécessaire pour que JUnit vérifie que, dans ce cas, une exception de type `CryptoException` est bien émise.

Test n°5

Ici, il vous est demandé d'écrire une méthode qui effectue une série de tests paramétrés. On souhaite vérifier la validité de la méthode `decrypter()`, en lui faisant exécuter 3 fois le même test, avec 3 valeurs différentes :

code à décrypter	résultat attendu
z9dZlgKM1Ug=	bonjour
9t7uBIYM2B0=	bonsoir
7W3ok7CHdF4=	coucou

- L'idée est de faire en sorte qu'une seule méthode puisse faire une série de tests similaires portant sur des valeurs différentes.
- Configurez la méthode `test5_decrypter_ok()` afin qu'elle soit considérée par JUnit comme un test paramétré appliquée aux 3 couples de données présentés ci-dessus.
- Écrivez le code qui vérifie que lorsqu'on exécute la méthode `decrypter()`, pour chaque valeur en entrée, on obtient la réponse prévue.
NB. Il y a plusieurs façons d'obtenir ce résultat; Choisissez celle qui vous convient. La seule contrainte est d'utiliser un test paramétré.
- Le test n°5 doit produire 3 itérations qui réussissent toutes les 3.