Sujet de l'EI 2022-2023

Conditions de l'épreuve	
Accès au réseau :	● **Au début** de l'épreuve : accès au réseau pour récupérer les fichiers d'installation de l'EI. ● **Pendant** l'épreuve : réseau coupé. Aucune communication ne doit être possible avec l'extérieur. ● **À la fin** de l'épreuve : rétablissement de l'accès au réseau pour remettre le travail.
Durée :	1h30
Documents autorisés :	Documents personnels autorisés. Avant le début de l'EI, les élèves peuvent télécharger des fichiers personnels via le réseau et les copier sur leur poste. L'utilisation de clés USB est interdite.
Installation de l'EI :	Voir les instructions ci-après, au début du document.
Remise du travail :	Voir les instructions à la fin de ce document.

0. Préliminaires

Mise en place

- Créer un dossier qui servira de workspace Eclipse pour cette EI
- Télécharger le fichier **ZIP Outils-Java-EI-Revision.zip**
- Extraire son contenu dans le dossier workspace.
- Démarrer Eclipse et ouvrir le dossier workspace.

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

Chaque exercice est basé sur l'utilisation d'une petite application permettant de jouer au Nombre Mystère.

- Dans le projet **exo-3-maven**, dépliez la branche **src/main/java**.
- Dans le package **jeux**, exécutez la classe **AppliExo3**.
- La 1ère vue permet de jouer au Nombre Mystère. Il faut deviner un nombre compris entre 0 et 32, en faisant des essais successifs.
- Il est possible de tricher en double-cliquant n'importe où dans la vue.
  Cela fait afficher le nombre à deviner pendant un bref instant.
- Le bouton "Config" donne accès la vue n°2, qui permet de paramétrer la difficulté du jeu.

Structure de l'EI

Cette EI est découpée en 4 parties :

1. API Reflection	(4,5 points)
2. Spring	(8,0 points)
3. Maven	(2,0 points)
4. JUnit	(5,5 points)

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

1. API Reflection

Présentation

- Dans le projet **exo-1-reflection**, dépliez le package **jeux**.
- Ouvrez le code de la classe **Context**.

Il s'agit d'une classe **Context** artisanale similaire à celle que vous avec créée en TP. Comme d'habitude, cette classe est dotée d'une méthode **getBean()** et d'une méthode **close()**.

Contrairement à ce que vous avez fait en TP, les méthodes **getBean()** et **close()** ne contiennent pas l'intégralité du code dont elles ont besoin. Elles font appel à des méthodes auxiliaires qui se trouvent début de la classe.

Votre travail va consister à écrire le code de ces méthodes auxiliaires.

Exécution de la classe de test

- Dans la branche **`src/test/java`**, dépliez le package **jeux**.
- Exécutez la classe **TestContext** en tant que test JUnit.
  Vous devez constater que tous les tests échouent.

La classe est composée de 7 tests :

- 5 tests qui correspondent aux 5 méthodes auxiliaires : **find()**, **create()**, **inject()**, **start()** et **stop()**.
- 2 tests qui correspondent aux 2 méthodes principales : **getBean()** et **close()**.

Méthode find()

- Dans la classe **Context**, écrivez le code de la méthode **find()** :
  - Cette méthode reçoit en paramètre une variable qui représente une classe.
  - La méthode doit rechercher dans la liste représentée par la variable **beans**, s'il s'y troue un objet donc la classe est égale à celle représentée par la variable **clazz**.
  - Si un objet est trouvé dans la liste, il est retourné par la méthode.
    Sinon, la méthode retourne **null**.
- Si le code de cette méthode est correct, le premier test doit réussir.

Méthode create()

- Écrivez le code de la méthode **create()** :
  - 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 le code de cette méthode est correct, le test n°2 doit réussir.

Méthode inject()

Attention ! La classe **Context** sur laquelle vous travaillez ne fonctionne pas exactement comme celle que vous avez écrite en TP.

En TP, la classe **Context** utilisait les annotations standards de Java EE.

Ici, on utilise nos propres annotations qui se nomment **@Resource**, **@Init** et **@Finish**.
Ces annotations se trouvent dans le package **jeux.annotations**.

- Écrivez le code de la méthode **inject()** :
  - Cette méthode reçoit en paramètre un objet qui a été créé par le **Context**.
  - Elle doit parcourir la liste des variables (champs) définis par la classe de cet objet.
    N.B. : Il faut prendre en compte toutes les variables, y compris celles qui ne sont pas publiques.
  - Pour chaque variable qui porte l'annotation **@Resource** :
    - Il faut affecter une valeur à cette variable. Cette valeur est récupérée grâce à la méthode **getBean()** à qui l'on passe en paramètre le type de cette variable.
    - Pensez à traiter le cas où la variable est privée. Il faut la rendre accessible avant de pouvoir l'initialiser.
- Si le code de cette méthode est correct, le test n°3 doit réussir.

Méthode start()

- Écrivez le code de la méthode **start()** :
  - Cette méthode reçoit en paramètre un objet qui a été créé par le **Context**.
  - Elle doit parcourir la liste des méthodes définies par la classe de cet objet.
    N.B. : Il faut prendre en compte toutes les méthodes, y compris celles qui ne sont pas publiques.
  - Pour chaque méthode qui porte l'annotation **@Init** :
    - Il faut l'exécuter.
    - N.B. :il s'agit d'une méthode sans paramètre.
    - Pensez à traiter le cas où la variable 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 stop()

- Écrivez le code de la méthode **stop()** :
  - Le code de cette méthode est identique à celui de la méthode **start()**.
    La seule différence est que, au lieu de rechercher l'annotation **@Init**, c'est l'annotation **@Finish** qui est prise en compte.
- Si le code de cette méthode est correct, le test n°5 doit réussir.

Méthode close()

- Observez le code de la méthode **close()** (à la fin de la classe **Context**) :
  - Cette méthode parcourt la liste représentée par la variable **beans**.
  - Pour chaque élément trouvé dans la liste, elle exécute la méthode **stop()**.

Ce fonctionnement n'est pas correct et fait échouer le test n°7, car la liste est parcourue en sens normal, c'est-à-dire du début à la fin.

- Modifiez la méthode **close()** de telle façon que la liste **beans** soit parcourue en sens inverse, c'est-à-dire en partant du dernier élément jusqu'au premier.
- Si le code de cette méthode est correct, le test n°7 doit réussir.

2. Spring

Le projet **exo-2-spring**, contient une version de l'application Nombre Mystère qui n'utilise pas Spring, mais une classe **ContextScan** fournie par le framework JFox.

Dans un premier temps, vous allez commencer par tâcher de faire fonctionner cette application en utilisant cette classe **ContextScan**.

Classe AppliExo2

- Dans le package **jeux**, ouvrez le code de la classe **AppliExo2**.
  Ce code est inachevé. Il faut le compléter.
- Ligne 35 : la variable **managerGui** est initialisée à null. C'est une erreur.
  Il faut utiliser l'objet **context** pour l'initialiser.
- Ligne 37 : il faut indiquer à JavaFX que pour instancier chaque contrôleur associé à une vue, il doit faire appel à l'objet **context** en lui demandant d'utiliser sa méthode **getBean()**.
  Il faut donc remplacer null, par une expression du type `context::xxxxxxx`
  Notez la présence de l'opérateur **::** juste après **context**.
  Dans cette expression, il manque juste le nom de la méthode.

Ajout de la bibliothèque Annotations JEE

Pour effectuer les injections de dépendances, la classe **ContextScan** de JFox utilise les annotations standards de JEE. Elle n'utilise pas les annotations spécifiques à Spring.

Dans cette version de l'application, aucune annotation n'a été mise. C'est à vous de les ajouter.

Pour commencer, il faut ajouter au projet la bibliothèque contenant les annotations.

- Faites le nécessaire pour ajouter au projet **exo-2-spring**, la User Library nommée **Annotations JEE**.

Injections de dépendances

- Démarrez l'application et essayez de jouer au Nombre Mystère.
  Cela génère des erreurs, car il manque toutes les annotations permettant de faire les injections de dépendances.
- Recherchez la cause de l'erreur. Il s'agit d'une **NullPointerException**: qui s'est produite dans la classe **ControllerNombreJeu**. C'est la variable **modelNombre** qui vaut null, car elle n'a pas été initialisée.
  Faites le nécessaire pour qu'elle soit initialisée par injection de dépendance.
- Exécutez de nouveau l'application. Vous allez encore avoir une **NullPointerException**.
  Résolvez le problème et recommencez autant de fois que nécessaire, jusqu'à ce qu'il n'y ait plus d'erreurs.
- Lorsque le jeu s'affiche, actionnez le bouton "Config". Si des erreurs apparaissent, apportez les corrections nécessaires.

À ce stade, l'application doit fonctionner correctement :

- Vous devez pouvoir jouer au Nombre Mystère et changer sa configuration.
- Si vous arrêtez une partie avant sa fin, lorsque vous redémarrez l'application vous devez retrouver votre partie dans l'état où vous l'aviez laissée, car son état a été sauvegardé dans un fichier.

Utilisation de Spring

À présent vous allez abandonner la classe **Context** fournie par JFox et utiliser une classe **Context** fournie par Spring.

- Faites le nécessaire pour ajouter au projet **exo-2-spring**, la User Library nommée **Spring**.

Création de la classe de configuration de Spring

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

Cette classe permet de définir la liste des composants qui devront être 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 **ConfigSimple**, écrivez les 5 méthodes qui permettent d'indiquer à Spring comment instancier chacun des composants de l'application.

  Voici la liste des 5 composants :
  - **DaoNombre**
  - **ModelNombre**
  - **ControllerNombreJeu**
  - **ControllerNombreConfig**
  - **ManagerGui**

  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 instanciations soient faites en mode Lazy.

Adaptation de la classe AppliExo2

- Ouvrez le code de la classe **AppliExo2**.
- Faites le nécessaire pour que la variable **context** ne soit plus de l'ancien type **ContextScan**, mais du type fourni par Spring qui peut être configuré grâce aux annotations. Le nom de cette classe commence par **AnnotationConfig**.
- Dans la méthode **start()**, écrivez les 3 instructions qui sont nécessaires pour initialiser le contexte Spring.

Si vous démarrez l'application, la vue s'affiche, mais il manque les messages dans la partie supérieure. Si on essaie de jouer, cela génère des erreurs.

Passez à l'étape suivante pour résoudre ces anomalies.

Méthodes d'initialisation

Le problème vient du fait que certains composants comportent une méthode d'initialisation qui doit être exécutée après l'instanciation du composant, pour que l'application fonctionne correctement.

Le développeur a choisi de donner systématiquement à ces méthodes le nom **init()**. C'est pratique, mais ça ne suffit pas pour que Spring sache qu'il doit les exécuter.

Vous allez donc commencer par chercher dans quels composants il y a une méthode **init()**. Ensuite vous ferez le nécessaire pour qu'elles soient exécutées automatiquement par Spring.

- Dans Eclipse, menu "Search > Java…".
- Dans le champ "Search string", indiquez : `init`
- Dans la section "Search For", sélectionnez : ⦿ **Method**
- Dans la section "Limit To", sélectionnez : ⦿ **Declaration**

- Dans la section "Search In", cochez uniquement la case : ☑ **Source**
  Toutes les autres cases doivent être décochées : ☐
- Dans la section "Scope", sélectionnez : ⦿ **Workspace**.
- Actionnez le bouton "Search".
- Dans le volet du bas d'Eclipse, sélectionnez le projet **exo-2-spring** et appuyez sur la touche **[✱]** du pavé numérique. Cela devrait déplier complètement la branche.

  N.B. : Pour la replier complètement, on peut utiliser la touche **[-]** (moins) du pavé numérique.
- Ceci doit vous permettre de voir qu'il y a 2 méthodes **init()** au sein du projet **exo-2-spring**.

  Faites le nécessaire pour que chacune de ces méthodes soie considérée par Spring comme une méthode d'initialisation.
- Exécutez l'application. Dans la vue "Console" d'Eclipse, vous devez voir s'afficher 2 lignes qui se terminent par **#init()**. Elles indiquent que les 2 méthodes **init()** ont bien été exécutées.

Méthode de clôture

Certains composants contiennent une méthode de clôture. Le développeur a choisi de lui donner systématiquement le nom **finish()**.

- Utilisez la même technique que précédemment pour trouver toutes les méthodes nommées **finish()** qui se trouvent dans le projet **exo-2-spring**.
- En fait, il y a 1 seule méthode **finish()**

  Faites le nécessaire pour que cette méthode soit considérée par Spring comme une méthode de clôture, c'est-à-dire une méthode à exécuter lorsque l'application se termine.
- Exécutez l'application, puis actionnez le bouton "Quitter". Dans la vue "Console" d'Eclipse, vous devez voir s'afficher 1 ligne qui se termine par **#finish()**.

Configuration par exploration de packages

- Dans le package **jeux**, créez une nouvelle classe nommée **ConfigExplo**.
- Ouvrez le code de la classe **ConfigExplo**.
- 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.
- Ouvrez le code de la classe **AppliExo2**.
- Faites le nécessaire pour indiquer que Spring doit utiliser la classe **ConfigExplo** 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. Ils ont été définis dans la classe **ConfigSimple**.
- Exécutez l'application. Elle doit fonctionner correctement.

Instanciation des contrôleurs

Dans JavaFX, les contrôleurs sont des composants sensibles. Il est impératif qu'une nouvelle instance du contrôleur soit créée, si on crée une nouvelle instance de la vue.

Pour le tester, on va avoir recours à un stratagème.

- Exécutez l'application comme si vous vouliez jouer au Nombre Mystère.
- Appuyez sur la touche **[Ctrl]** du clavier et, sans la relâcher, faites un clic sur le bouton "Quitter".
- Cela a ouvert une 2ème fenêtre qui s'est superposée à la première. Décalez-la sur le côté afin de voir les 2 fenêtres côte à côte.
- Si une seule instance du contrôleur a été créée alors qu'il y a 2 vues, cela doit poser un problème.
  Que l'on joue dans la 1ère ou la 2ème fenêtre, c'est uniquement l'affichage de la 2ème fenêtre qui est mis à jour. L'affichage de la 1ère fenêtre reste figé et ne reflète plus l'état de la partie.
- Si vous constatez le problème, faites le nécessaire pour que Spring sache qu'il doit créer plusieurs instances du composant **ControllerNombreJeu**.
- Faites la même chose pour le composant **ControllerNombreConfig**.
- Exécutez l'application. Si vous ouvrez 2 fenêtres, elles doivent toutes les deux afficher le même contenu lorsqu'on joue au Nombre Mystère.

Utilisation d'un Model Light

La classe **ModelNombre** qui se trouve dans le package **jeux.model.standard** a beoin de la présence d'un DAO pour fonctionner.

Il existe également une version Lignt du **ModelNombre**. Cette version peut fonctionner sans la présence d'une DAO. En conséquence, il n'y a pas d'enregistrement des données dans un fichier.

Ici, il vous est demandé de faire le nécessaire pour que l'application utilise la version Light au lieu d'utiliser la version Standard.

- Dans le projet **exo-2-spring**, recherchez le dossier **light** qui se trouve vers la fin.
- Sélectionnez le dossier **light** et faites "Copier".

- Dans la branche **`src/main/java`**, sélectionnez le package **`jeux.model`** et faites "Coller".

Ceci a pour effet d'ajouter un nouveau package nommé **jeux.model.light** qui contient une classe **ModelNombre**.

À présent, l'application contient 2 classes **ModelNombre**. Une en version **standard** et l'autre en version **light**.

Actuellement, l'application utilise la version Standard.
Lorsque vous l'avez exécutée, vous avez constaté dans la vue "Console" d'Eclipse que les lignes qui concernent leDAO y étaient bien affichées.

```
jeux.dao.serial.DaoNombre#init()
...   ...
jeux.dao.serial.DaoNombre#finish()
```

Ces lignes reflètent la lecture et l'écriture du fichier.

- Faites les opérations de configuration nécessaire pour que ce soit la version Light du Model qui soit utilisée et non pas la version Standard.
  - Plusieurs solutions sont possibles.
  - Attention ! Si vous êtes obligés de défaire une configuration que vous avez faite précédemment, ne la supprimez pas. Mettez la en commentaire afin d'en conserver la trace.
  - Le nouveau Model doit être instancié en mode Lazy.
- Démarrez l'application et arrêtez-la. Dans la vue "Console", il ne doit plus y avoir de traces correspondant au DAO.

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 :* **jeu-nombre**
  - *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, f**aites le nécessaire pour supprimer du Build Path, toutes les User Libraries. Il y en a 3 s :
  - **JavaFX**
  - **JFox**
  - **Spring**

  Attention ! 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 en 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.

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

  N.B. : **JFox** n'est pas référencé sur le site MVNRepository. Les instructions vous sont données ci-après.

Dépendance pour JFox

- Faites le nécessaire pour ajouter, dans le fichier POM, la dépendance à JFox.

  Voici les informations qui permettent de l'identifier :
  - *Group Id :* `eu.jfox-dev`
  - *Artifact Id :* `jfox`
  - *Version :* `2023-03-EI`

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 fin.
- 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 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 **TestNombre**.

4. JUnit

- Dans le projet **exo-4-junit**, dépliez la branche **`src/test/java`**.
- Ouvrez le code de la classe **TestModelNombre**.
  Une partie du code a déjà été écrit, mais il est inachevé.
  Il s'agit d'une classe de tests qui portent sur la classe **ModelNombre**.

La classe **ModelNombre** nécessite la présence d'une classe **DaoNombre**.

Pour faire les tests vous allez utiliser un DAO de type Mock, c'est-à-dire qui n'utilise pas de fichier. Il s'agit d'un DAO factice destiné à être utilisé pendant la phase de tests.

Test n°1

- Observez le code de la méthode **test1_jouer_trop_petit()**.
  Elle contient déjà 6 instructions dont le rôle est le suivant :
  1. Déclarer 2 variables de types DAO et Model.
  2. Initialiser le DAO de telle façon que le nombre-mystère soit **15**
  3. Initialiser le Model de façon à ce qu'il fasse appel au DAO pour y récupérer l'état de la partie.

Le but des tests va être de vérifier le bon comportement de la méthode **jouer()** du Model.

En fonction de la valeur qui est jouée, cela va produire une réponse différente.

Pour exécuter la méthode **jouer()** en lui passant la valeur **23**, il faut écrire :

```
modelNombre.jouer( "23" );
```

N.B. : Il s'agit de la chaîne de caractères **"23"** et non du nombre **23**, car le joueur saisit la valeur dans une zone de texte. S'il se trompe, il peut taper des lettres. C'est donc bien une chaîne de caractères qui est passée à la méthode **jouer()**.

Pour connaître la réponse du jeu, il faut utiliser l'expression :

```
modelNombre.getReponse()
```

Il y a 3 valeurs possible pour la réponse :
- **`-1`** signifie "Trop petit"
- **`1`** signifie "Trop grand"
- **`0`** signifie que le joueur a indiqué la bonne valeur.

Voici le travail qui vous est demandé :

- Faites le nécessaire pour que la méthode **test1_jouer_trop_petit()** soit considérée par Junit comme une méthode de test.
- Complétez son code pour qu'il vérifie que :
  - si on exécute la méthode **jouer()** en lui passant une valeur inférieure au nombre mystère,
  - alors la réponse vaut **-1**.

Rappel : le nombre mystère vaut **15**.
La valeur jouée doit être passée sous la forme d'une chaîne de caractères.

- 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_jouer_trop_grand()** pour que celle-ci fasse un test similaire au test n°1, mais avec une valeur plus grande que celle du nombre mystère.
  La réponse attendue est donc : **1**
- 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_jouer_gagnant()** pour que celle-ci fasse un test similaire aux tests précédents, mais en passant la valeur exacte du nombre mystère.
  La réponse attendue est donc : **0**
- Vérifiez que le test fonctionne et qu'il réussit.

Mutualisation de code

Élimination des redondances de code

Comme vous l'avez remarqué, les méthodes de test commencent toutes par la même série de 6 instructions qui initialisent les objets DAO et Model.

On souhaite éviter cette redondance de code. Voici quelques conseils

- Créez 2 variables privées au niveau de la classe : une pour le DAO et l'autre pour le Model.
  Rappel : au niveau de la classe, il peut y avoir des variables statiques et des variables d'instance (non statiques). Ici, le choix n'est pas évident. Il faudra peut-être procéder par tâtonnements.
- JUnit permet de définir des méthodes auxiliaires qui sont exécutées :
  - soit une seule fois, au tout début, c'est-à-dire avant le tout premier test.
  - soit de façon répétitive, avant chaque test.

  Vous pouvez utiliser l'une de ces techniques (ou les deux à la fois) pour éviter d'avoir à écrire systématiquement les 6 premières instructions de chaque méthode de test.
- À la fin, supprimez les 6 premières instructions de chaque méthode de test.
- Exécutez la classe de test et vérifiez que les 3 tests ont bien été effectués et qu'ils ont tous réussi.

Optimisation du code mutualisé

Selon la stratégie que vous avez adoptée, plusieurs cas peuvent se présenter. L'objectif ici est de trouver la stratégie optimale.

- Observez la vue "Console" d'Eclipse.
  Celle-ci contient un certain nombre de fois les messages.
  `new DaoNombre`
  `new ModelNombre`
- Comptez le nombre de chacun de ces messages.
  S'il y en a plus de 3 pour l'un ou l'autre de ces messages, c'est vraiment trop.
  Faites le nécessaire pour qu'il y ait au maximum 3 exemplaires de chaque message.

En fait, le contenu de la variable **daoNombre** n'est jamais modifié au cours des tests. Cela paraît donc inutile de l'initialiser 3 fois. Il devrait suffire de l'initialiser une seule fois, au tout début, avant le tout premier test.

- Si, dans la vue "Console", le message **`new DaoNombre`** est affiché plusieurs fois, faites le nécessaire pour qu'il ne soit plus affiché qu'une seule fois.

L'affichage qui traduit la stratégie optimale est donc le suivant :

```
new DaoNombre
new ModelNombre
new ModelNombre
new ModelNombre
```

Test n°4

La classe **ModelNombre** a une méthode qui permet de savoir si la partie est terminée.

Pour obtenir cette information, il faut utiliser l'expression :

`modelNombre.isPartieFinie()`

Au départ, cette méthode retourne **faux**, car la partie est en cours.

Mais, si le joueur joue 5 fois de suite en proposant des mauvaises valeurs, alors il a perdu et la partie est finie. La méthode retourne donc la valeur **vrai**.

Voici le travail à effectuer :

- Dans la méthode **test4_jouer_perdant()**, écrivez le code qui permet de tester que, au départ, la méthode **isPartieFinie()** retourne **faux**.
- Ensuite, toujours dans la même méthode de test, exécutez 5 fois la méthode **jouer()** en lui passant une mauvaise valeur. Ça peut être 5 fois la même valeur.
- Puis, vérifiez que, juste après avoir exécuté 5 fois la méthode **jouer()**, la méthode **isPartieFinie()** retourne **vrai**.

Pour effectuer ces 2 tests (**faux** et **vrai**), il vous est demandé d'utiliser 2 méthodes différentes parmi celles qui sont fournies par JUnit. Choisissez celles qui sont les plus appropriées pour ces types de tests.

Test n°5

Si on passe à la méthode **jouer()**, une valeur qui n'est pas un nombre, il est prévu qu'une exception de type **NumberFormatException** soit émise.

- Dans la méthode **test5_jouer_exception()**, exécutez la méthode **jouer()** en lui passant une chaîne de caractères qui ne contienne pas un nombre, mais un texte.
- Faites le nécessaire pour que JUnit vérifie que, dans ce cas, une exception de type **NumberFormatException** est bien émise.

Test n°6

Ici, il vous est demandé d'écrire une méthode qui effectue une série de tests paramétrés.

Comme vous l'avez vu, les tests n°1, 2 et 3 sont très similaires.

L'idée est de faire en sorte qu'une seule méthode puisse faire une série de tests similaires portant sur des valeurs différentes.

Voici les valeurs proposées :

valeur	réponse
2	-1
50	1
15	0

- Configurez la méthode **test6_jouer_reponse()** afin qu'elle soit considérée par JUnit comme un test paramétré appliqué aux 3 couples de données présentés ci-dessus.
- Écrivez le code qui vérifie que lorsqu'on exécute la méthode **jouer()** du Model, pour chaque valeur en entrée, on obtient la réponse prévue.
- Le test n°6 doit produire 3 itérations qui réussissent toutes les 3.

Remise du travail à la fin de l'épreuve

Procédure normale

- L'accès à Internet doit avoir été rétabli.
- Dans le volet de gauche d'Eclipse, dépliez le projet **script-envoi**.
- Double-cliquez sur le script **envoyer-fichiers.bat** pour l'exécuter.
- Lorsque le script vous le demanda, indiquez votre nom suivi de votre prénom.

  Ce script crée une archive **ZIP** et la dépose directement dans un espace Cloud sur Internet.

  À la fin, le script ouvre le navigateur web et vous montre le contenu de l'espace Cloud.
  - Si vous y voyez un fichier **ZIP** qui porte votre nom, c'est que l'opération a réussi. Votre travail a bien été transmis.
  - Si vous ne voyez pas votre fichier **ZIP**, l'opération a échoué. Utilisez l'une des procédures de secours ci-dessous.

Procédures de secours

Si la procédure précédente ne fonctionne pas, voici 2 autres solutions pour envoyer votre travail.

1. Si l'archive **ZIP** qui porte votre nom a bien été créée dans le dossier de l'EI :
   Envoyez-la par e-mail à mon adresse : **`amblard@3il.fr`**

2. Sinon, si l'archive **ZIP** n'a pas été créée :
   - Créez une archive **ZIP** contenant tout le dossier **workspace**.
   - Utilisez un service de transfert de gros fichiers,
     par exemple : **`swisstransfer.com`**
     qui est particulièrement simple d'emploi.
   - Envoyez l'archive **ZIP** à mon adresse : **`amblard@3il.fr`**
     en utilisant le service de transfert de gros fichiers.