Exercice 3

1) On définit une classe **Marin**. Un marin possède un nom, un prénom et un salaire, qui sont trois propriétés. Le salaire est la seule de ces trois propriétés à pouvoir être modifiée une fois une instance de **Marin** créée. Surcharger la méthode **toString()** et façon à pouvoir afficher un marin à l'écran.

On définit une classe **Capitaine**. Cette classe étend **Marin**, et lui ajoute une propriété : **grade**. Un grade est une énumération, qui peut prendre les valeurs **COMMANDANT**, **CAPITAINE**, **AMIRAL**.

2) Créer la classe **Capitaine** avec sa méthode **toString()**.
3) Surcharger les méthodes **equals()** et **hashCode()** de ces deux classes. Quels champs choisir pour définir l'identité d'une instance de **Marin** ? De **Capitaine** ?

On reprend la classe **Equipage** de l'exercice 2.

4) Est-il possible de ranger un capitaine dans un équipage, sans modifier la classe **Equipage** ?
5) Montrer sur un exemple ce qui se passe si l'on tente d'afficher (en utilisant sa méthode **toString()**) un équipage alors qu'un capitaine s'est glissé dedans.

Exercice 4

On définit une classe **EquipageCommande**. Un équipage commandé est un équipage qui possède de plus une propriété **commandant**, instance de la classe **Capitaine**.

1) Créer cette classe **EquipageCommande**.
2) Montrer sur un exemple ce qui se passe si l'on tente de mettre un marin à la place d'un capitaine en tant que commandant de cet équipage.

Exercice 5

On souhaite créer une classe **Bateau**, qui servira de base à un ensemble de types de bateaux différents. Un bateau possède un **nom** et un **tonnage**, de type **String** et **int** respectivement. Il possède également un **equipage**, de type **EquipageCommande**.

1) Écrire totalement la classe **Bateau**. Les champs **nom** et **tonnage** seront en lecture seule.
2) Deux bateaux sont égaux si leurs noms sont égaux. Modifier la classe **Bateau** pour prendre en compte cette contrainte.
3) Écrire une méthode **toString()** pour la classe **Bateau**. Elle affichera le nom du bateau, son tonnage, le nom du capitaine et les membres de l'équipage.
4) On distingue trois types de bateaux : les corvettes, les frégates et les croiseurs. Tous ces bateaux partagent les caractéristiques définies dans la classe **Bateau**.
   a. Proposer une organisation de classes pour **Corvette**, **Fregate**, et **Croiseur**. Chacune de ces classes aura une méthode **getTypeBateau()**, renvoyant une chaîne de caractères (**String**) ayant pour valeurs respectives : "**Corvette**", "**Frégate**", "**Croiseur**".
   b. Modifier la méthode **toString()** de la classe **Bateau** pour afficher le type du bateau. On ne souhaite pas écrire de méthode **toString()** dans les classes **Corvette**, **Fregate**, et **Croiseur**.

Exercice 3

Cet exercice utilise un bean `Marin`, qui possède deux champs : `nom` et `prenom`. On suppose que la méthode `toString()` a été correctement surchargée, et que deux marins sont égaux s'ils ont même nom et même prénom. On pourra réutiliser et adapter une classe `Marin` d'un autre exercice.

Créer une classe `Equipage`, qui contient un tableau capable de stocker ces 5 marins. Ce tableau doit être correctement initialisé dès qu'une instance d'`Equipage` a été créée.

On impose qu'un même marin ne puisse pas être présent deux fois dans l'équipage.

1) Créer une méthode `addMarin(Marin)`. Cette méthode tente d'ajouter un marin à l'équipage. Dans quels cas l'ajout d'un marin échoue-t-il ? Cette méthode retournera un booléen, `true` si le marin a été ajouté à l'équipage, `false` s'il n'a pas pu l'être.
2) Créer une méthode `isMarinPresent(Marin)`. Cette méthode retournera `true` si le marin passé en paramètre est présent dans l'équipage, `false` dans le cas contraire. On pourra réécrire la méthode `addMarin(Marin)` pour qu'elle utilise `isMarinPresent(Marin)`.
3) Créer une méthode `getNombreMarins()`. Cette méthode retourne le nombre de marins effectivement présents dans l'équipage.
4) Écrire alors une méthode `main()` de test, qui tentera d'ajouter 6 marins dans l'équipage. Vérifier que les marins sont bien ajoutés, sauf le sixième. Vérifier que la méthode `addMarin()` a bien retourné `false` dans ce cas.
5) Écrire une méthode `toString()` pour la classe `Equipage`.

Attention, la suite de l'exercice va imposer que l'on modifie le code de la méthode `addMarin()`.

6) Créer une méthode `removeMarin(Marin)`. Cette méthode retire de l'équipage le marin passé en paramètre. Dans quels cas le retrait d'un marin échoue-t-il ? Cette méthode retournera `true` si le marin a bien été retiré de l'équipage, `false` dans le cas contraire.
7) Doit-on modifier alors la méthode `addMarin(Marin)` ?
8) Créer une méthode `clear()` qui efface le contenu de l'équipage.
9) Créer une méthode `addAllEquipage(Equipage)` qui ajoute les marins de l'équipage passé en paramètre à ceux de l'équipage courant. On souhaite que cet ajout soit fait en bloc. Cela signifie que si l'un des marins de l'équipage passé en paramètre ne peut pas être ajouté à l'équipage courant, alors aucun des marins de cet équipage ne sera ajouté. Proposer un type de retour pour cette méthode.

On souhaite à présent pouvoir ajouter autant de marins que l'on veut à notre équipage. Il va falloir pour cela « étendre » le tableau lorsqu'il est plein. Comme il n'est pas possible d'étendre un tableau, il va falloir créer un nouveau tableau et copier le contenu de l'ancien tableau dans ce nouveau tableau.

10) Écrire une méthode `etendEquipage(int)`, qui prend un entier en paramètre. Cette méthode ajoute à l'équipage le nombre de places disponibles passé en paramètre. Donc, si la capacité initiale du tableau contenu dans `Equipage` est 5, et que cette méthode est invoquée avec 7, alors la nouvelle capacité de l'équipage doit être de 12.
11) Modifier la méthode `addMarin(Marin)` de telle sorte qu'elle étende le tableau des marins en fonction de ses besoins. On pourra étendre la taille du tableau de 50% à chaque demande. Dans quel cas la méthode `addMarin(Marin)` va-t-elle alors retourner `false` ?
12) Écrire le code nécessaire pour tester le bon fonctionnement des méthodes écrites dans la classe `Equipage`.

On veut écrire une méthode `equals()` pour cette classe `Equipage`. Deux équipages seront égaux s'ils possèdent les même marins, mais pas nécessairement enregistrés dans le même ordre.

13) Programmer cette méthode `equals()`. Mettre en évidence son bon fonctionnement.
14) Proposer l'écriture de la méthode `hashCode()` associée. Vérifier qu'elle fonctionne correctement sur les mêmes cas de test que ceux utilisés pour la méthode `equals()`.

Java I – Langage

Série 0

Exercice 0

Le but de ce premier exercice est de s'assurer que l'environnement est correctement configuré, notamment qu'il est possible de compiler et d'exécuter du code Java.

1) Télécharger et installer la dernière version du JDK (et non pas le JRE) sut **http://java.sun/com/**
2) Vérifier l'existence d'une variable `JAVA_HOME` pointant bien vers une installation Java 8.
3) Vérifier que `$JAVA_HOME/bin` est bien dans ajouté à la variable `PATH`.
4) Vérifier la version de Java installée en tapant la commande :
   ```
   $ java -version
   ```
5) À l'aider d'un éditeur de base (type Notepad), écrire une classe ne comportant qu'une fonction `main`, capable d'écrire `Bonjour le monde` dans une fenêtre de commandes (shell). Compiler cette classe et l'exécuter.

Exercice 1

Le but de ce deuxième exercice est de prendre en main les fonctions de base d'Eclipse, de créer un premier projet et de l'exécuter. On installera une version d'Eclipe téléchargeable sur **http://www.eclise.org/**.

1) Télécharger et installer Eclipse. Configurer Eclipse de sorte qu'il utilise bien le JDK qui vient d'être installé. Sous Linux :
   a. Créer une variable d'environnement `ECLIPSE_HOME` pointant vers ce répertoire.
   b. Ajouter le répertoire `$ECLIPSE_HOME/bin` à votre variable `PATH`.
   c. Lancer Eclipse dans une fenêtre shell par la commande `eclipse`. Si votre environnement est correctement configuré, Eclipse doit se lancer.
2) Suivre le tutorial de prise en main et de création d'un premier projet qui se trouve en ligne à l'adresse suivante : **http://blog.paumard.org/tutoriaux/eclipse/**

Exercice 2

Créer une classe `Factorielle`, permettant de calculer la factorielle d'un nombre. On appellera cette classe de la façon suivante.

```
public static void main(String[] args) {
   Factorielle fact = new Factorielle() ;
   System.out.println("Factorielle de 10 = " +
      fact.factorielle(10)) ;
}
```

1) Créer les différentes classes et méthodes de façon à ce que ce code fonctionne correctement.
2) Prendre les types suivants pour le calcul de la factorielle : `int`, `double` et `BigInteger`. Que constate-t-on ?

Exercice 3

On sait tous qu'un nombre premier est un nombre divisible seulement pas deux autres nombres : lui-même et 1. Pour savoir si un nombre est premier, il suffit de vérifier qu'il n'est divisible par aucun nombre plus petit que sa racine carrée.

Un nombre palindrome est un nombre qui peut se lire dans les deux sens. Quelques exemples de nombres palindromes sont par exemple : 121, 1331, 1234321.

Écrire un programme qui permette de trouver des nombres premiers palindromes. Quels nombres premiers palindromes votre programme vous a-t-il permis de trouver ?

Java I – Langage

Série 4

Ce TD a aussi pour but de se commencer à se familiariser avec la Javadoc de l'API standard de l'API Java. Cette Javadoc est en ligne, à l'adresse :

- Java 6 : **http://download.oracle.com/javase/6/docs/api/**
- Java 7 : **http://download.oracle.com/javase/7/docs/api/**
- Java 8 : **http://download.oracle.com/javase/8/docs/api/**

Exercice 6

On souhaite réécrire la classe **`Equipage`** de l'exercice 2, en utilisant une **`List`** plutôt qu'un tableau. Reprendre la classe telle qu'elle a été écrite dans cet exercice 2, et modifier la dans un nouveau projet, de façon à répondre à la consigne.

Exercice 7

On souhaite mettre en évidence le fonctionnement du **`Set`**. On reprend un classe **`Marin`** habituelle, avec un **`nom`** et un **`prenom`**.

1) Rappeler quelle est la sémantique du **`Set`** dans l'API Collection. Quelle est sa classe d'implémentation ?
2) Créer 3 instances de **`Marin`**. Les deux premières possèdent même nom et même prénom, différents de la dernière. Lorsque **`equals()`** / **`hashCode()`** sont surchargées dans **`Marin`**, les deux premiers sont donc égaux et différents du troisième.
3) Créer une méthode **`main()`**. Ajouter ces marins dans un **`Set<Marin>`**.
   a. Faire cette opération une première fois avec une classe **`Marin`** qui ne surcharge ni **`equals(Object)`**, ni **`hashCode()`**. Qu'observe-t-on ? La sémantique du **`Set<Marin>`** est-elle respectée comme on s'y attend ?
   b. Reprendre le même test avec une classe **`Marin`** qui surcharge **`equals(Object)`** et **`hashCode()`**. Qu'observe-t-on cette fois-ci ?
   c. Reprendre enfin le même test, avec **`hashCode()`** correctement surchargé. Qu'observe-t-on enfin ?

Exercice 8

On souhaite pouvoir comparer nos marins en utilisant les mécanismes standard de Java. Deux marins sont comparés par l'ordre alphabétique de leurs noms. Si leurs noms sont égaux, alors c'est l'ordre de leurs prénoms qui est pris en compte.

1) Créer une classe **`MarinComparator`**, qui implémente **`Comparator<Marin>`**. Vérifier dans la Javadoc les méthodes à implémenter pour ce faire. Créer cette classe.
2) Écrire une méthode **`main()`**, qui crée quelques marins (on pourra reprendre du code existant). Ranger ces marins dans un **`SortedSet`** et montrer que le comparateur écrit fonctionne correctement.
3) Créer une classe **`ComparableMarin`**, extension de **`Marin`**, qui implémente de plus **`Comparable<Marin>`**.
4) Ranger à nouveau ces objets dans un **`SortedSet`** afin de mettre en évidence le fonctionnement des objets comparables.

Exercice 9

On souhaite avoir deux fonctionnalités à la classe **`Equipage`** de l'exercice 4 :

- Conserver nos marins triés dans l'ordre de leurs noms ;
- Avoir une méthode **`getMarinByName(String nom)`**, qui nous retourne un marin lorsque l'on donne son nom.

1) Quelle interface de l'API Collection est-elle adaptée à la résolution de ce problème ?
2) Écrire cette nouvelle classe **`Equipage`**, en reprenant les méthodes définies dans l'exercice 2.
3) Faire le bilan de ce que l'on gagne à utiliser l'API Collection, plutôt qu'un tableau.

Java I – Langage

Série 5

On reprendra les classes `Marin` de l'exercice 3. Cette classe possède un nom, un prénom et un salaire.

Exercice 10

Le but de cet exercice est de créer une procédure de sauvegarde de nos marins dans un fichier. On créera une classe `Sauvegarde`, qui ne comportera que des méthodes statiques. Pour chaque question, on écrira quelques lignes de code dans une méthode `main()`, permettant de tester sur quelques instances de `Marin` que la fonctionnalité demandée est correctement implémentée.

1) Chaque instance de `Marin` est sauvée sur une ligne d'un fichier texte, où chaque champ est écrit sous forme d'une chaîne de caractères. On pourra utiliser le modèle suivant :
   `nom_du_marin|prénom_du_marin|salaire`
   a. Créer une méthode `sauveFichierTexte(String nomFichier, Marin marin)` pour ce faire ;
   b. Créer une méthode `lisFichierTexte(String nomFichier)`, qui retournera une liste de marins, comportant dans le bon ordre, les marins écrits dans ce fichier. On pourra utiliser la méthode `Integer.parseInt(String)`.
2) Chaque instance de `Marin` est sauvée champ par champ dans un fichier binaire.
   a. Quel flux doit-on utiliser pour écrire les types primitifs Java et les chaînes de caractères dans des fichiers binaires ?
   b. Utiliser ce flux pour écrire chaque marin, champ par champ, dans une méthode `sauveChampBinaire(String nomFichier, Marin marin)`.
   c. Créer une méthode `lisChampBinaire(String nomFichier)` retournant la liste des marins qui ont été sauvés dans ce fichier, dans l'ordre de la sauvegarde.
3) Chaque instance de `Marin` est sauvée en tant qu'objet dans un fichier binaire.
   a. Quel flux doit-on utiliser pour écrire les objets Java dans des fichiers binaires ?
   b. Utiliser ce flux pour écrire chaque marin, en tant qu'objet, dans une méthode `sauveObjet(String nomFichier, Marin marin)`.
   c. Créer une méthode `lisObjet(String nomFichier)` retournant la liste des marins qui ont été sauvés dans ce fichier, dans l'ordre de la sauvegarde.
4) Donner la taille de chaque fichier, pour la même liste de marins. Quelle est la façon la plus efficace d'enregistrer nos marins ?