Projet de POO : SIROP – SImulation of a RObotic Planet

Jean-Marie Normand, Équipe pédagogique POO
École Centrale de Nantes,
Bureau E211,
jean-marie.normand@ec-nantes.fr

Septembre 2014

0 Introduction

0.1 Présentation générale

Ce projet de TP qui va s'étendre sur toute la durée du module de Programmation Orientée Objet (POO) a pour but de vous initier au langage de programmation Java de manière progressive, en partant de l'écriture de quelques classes simples jusqu'à la réalisation d'un mini projet complet abordant les principales notions vues en cours, à savoir :

- les classes, instances et méthodes,
- le concept d'héritage, de redéfinition/surchage de méthodes,
- le polymorphisme et la résolution dynamique des liens,
- l'utilisation des principales structures de données (listes, etc.),
- la définition et l'utilisation de classes/méthodes abstraites et d'interfaces,
- la gestion des entrées/sorties,
- les exceptions,
- une introduction aux interfaces utilisateurs (ou *Graphical User Interface*, GUI),
- la notion de thread,
- le travail sera réalisé en utilisant un environnement de développement, nous recommandons d'utiliser l'édtieur **NetBeans** présent dans les salles de cours.

Ce projet est prévu pendant des séances de "TD/TP" c'est à dire qu'elles se déroulent en salle machine mais que le début des séances est consacré à

l'approfondissement de notions vues en cours et dont vous aurez besoin pour la réalisation du projet.

Chaque TD/TP a un objectif précis lié au projet réalisable dans la durée de la séance (2h). Néanmoins, ce projet est construit comme un succession d'objectifs et repose sur votre capacité à réaliser le travail dans le temps imparti. Votre code de la séance n servira donc de base à celui de la séance $n+1$, aussi **il vous revient de vous assurer que vous êtes en mesure de commencer la séance en ayant terminé le travail qui vous était demandé à la séance précédente**. Au besoin, vous devrez travailler en dehors des heures prévues au module de POO afin de récupérer votre retard éventuel.

0.2 Simulation d'un jeu de robots

L'objectif de ce projet est de réaliser un mini jeu de simulation d'un ensemble de robots sur un plateau de cases. Le plateau de jeu contient en plus des robots et des cases vides, un ensemble de bonus et d'obstacles divers. La taille du plateau est connue avant de lancer le jeu.

Pour fonctionner, les robots possèdent un niveau d'énergie et chaque action réalisée par un robot va lui coûter un point d'énergie. De même, les robots possèdent un nombre de points de vie caractérisant son état de santé (de parfaitement fonctionnel à bon pour la casse).

Il existe plusieurs catégories de robots, chacune représentant un type de comportement bien particulier : belliqueux, affamé (d'énergie), reproducteur, etc.

Les bonus peuvent être de plusieurs types, mais intéressons nous tout d'abord :

- aux générateurs d'énergie : qui permettent de recharger un robots en énergie,
- aux stations de réparation : qui permettent de réparer un robot (et donc d'augmenter son état de santé).

Il existe également plusieurs types d'obstacles, empêchant les robots de se déplacer librement sur le plateau de jeu :

- les obstacles fixes : qui occupent une position fixe sur le plateau de jeu,
- les obstacles mobiles : qui peuvent se mouvoir sur le plateau de jeu.

Ces éléments de base de notre projet peuvent évidemment être étendus en fonction de vos envies et idées.

0.3 Travail attendu

Le travail que vous allez devoir réaliser tout au long de ce projet peut être regroupé dans les grandes étapes suivantes :

1. définition du plateau de jeu,
2. représentations des éléments du jeu : différentes sortes de robots, bonus et obstacles,
3. définition des différents comportements des robots (en fonction de leurs catégories),
4. sauvegarde et chargement d'une partie
5. affichage du déroulement du jeu (tout d'abord sous forme textuelle, puis sous forme d'interface graphique).

Voici une capture d'écran de ce à quoi pourrait ressembler votre projet une fois terminé, lorsque l'interface utilisateur sera ajoutée :

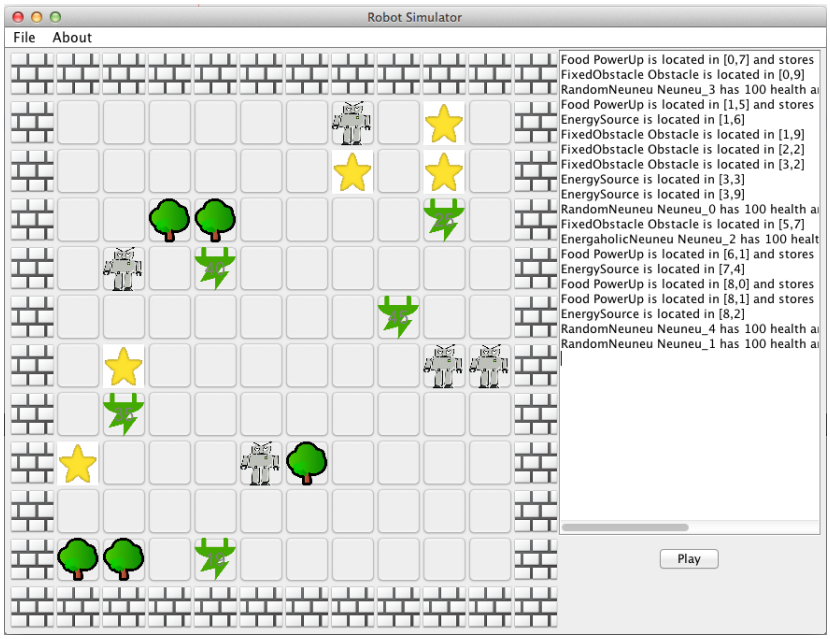

FIGURE 1 – Un exemple du résultat du projet de simulation de robots. Les obstacles sont représentés par des arbres, les stations de réparation par des étoiles et les générateurs d'énergie par des prises de courant.

0.4 Déroulement du projet

Ce projet est prévu pour se dérouler sur 20 heures de TD/TP. Chaque séance est prévue pour approfondir les notions vues en cours et voir leur

application à une des étapes de réalisation du projet.

Le découpage approximatif des heures allouées aux TD/TP est le suivant :

1. Introduction au Java, écriture de vos premières classes Java, représentation d'un robot générique ⇒ 2 heures
2. Représentation d'une catégorie de robot du jeu, d'un obstacle et d'un bonus. Notion d'héritage et définition d'une hiérarchie de classe pour les robots ⇒ 2 heures
3. Représentation d'un ensemble de robots différents, structures de données, notion de polymorphisme ⇒ 2 heures
4. Définition des différents comportements des robots en fonction de leurs catégories, notion d'interface ⇒ 2 heures
5. Bilan à presque mi-parcours : vérification de : vos diagrammes de classes, la qualité de votre documentation. Réponses à vos questions éventuelles. ⇒ 2 heures
6. Gestion des entrées/sorties en Java : sauvegarde (écriture) et chargement (lecture) d'une partie du jeu de simulation des robots ⇒ 2 heures
7. Intégration d'une interface utilisateur en utilisant la bibliothèque SWING de Java ⇒ 4 heures
8. Bilan, finalisation du travail : retour sur ce que vous avez réalisé jusqu'ici, enrichissement des éléments du jeu (nouveaux types de robots, nouveaux obstacles, bonus, etc.). Préparation du rendu final ⇒ 4 heures

Pour toutes les séances, nous rappelons que la programmation sera effectuée en Java, donc vous trouverez la documentation en ligne à l'adresse suivante : http://docs.oracle.com/javase/7/docs/api/index.html

1 Première Séance : introduction à Java et premiers pas

1.1 Avant-propos

Durant cette séance, vous allez faire connaissance avec le langage Java, vous serez amenés à écrire vos premières classes et à exécuter vos premiers programmes en Java.

Nous rappelons que vous avez vu en cours que :

- un fichier Java (d'extension `.java`) contient une et une seule classe

- vos classes doivent être testées dans un programme de test (classe spécifique contenant une méthode `main`) dont l'entête est le suivant : `public static void main(String[] args)...`
- une fois les fichiers `.java` écrits, il faut les compiler en utilisant la commande `javac` qui appelle le compilateur Java. Par exemple la commande suivante exécutée dans un terminal va compiler la classe contenue dans le fichier `monPremierProg.java` :
  `javac monPremierProg.java`
- lorsqu'un programme de test (ayant une méthode `main`) est compilé (voir point précédent), il est possible de l'exécuter en exécutant dans le terminal en utilisant la commande `java` qui invoque la machine virtuelle Java et dont la syntaxe est la suivante :
  `java monPremierProg`

1.2 Première classe du projet : représentation d'un point en 2D

La simulation des robots a lieu sur un plateau de jeu 2D dont les dimensions sont connues a priori.

Ainsi chaque case du plateau peut contenir un élément du jet : robot, obstacle, bonus, etc. Dit autrement, on peut voir qu'un élément du jeu a une position 2D sur le plateau de jeu.

Nous cherchons ici à construire une classe `Point2D` représentant un point de l'espace cartésien à deux dimensions à coordonnées entières.

Travail à faire : Définir une classe représentant un point en deux dimensions, à coordonnées entières et permettant de réaliser au moins les opérations suivantes :

- créer un point à une position définie,
- créer un point à partir d'un point déjà existant (i.e. copier un point existant),
- accéder et modifier l'abscisse d'un point,
- accéder et modifier l'ordonnée d'un point,
- modifier la position d'un point,
- afficher les coordonnées d'un point dans le terminal,
- comparer deux points entre eux (les deux points sont identiques si ils ont les mêmes coordonnées).

Une fois cette classe écrite en Java, proposez une classe `TestPoint2D` qui permet de créer quelques points en deux dimensions, d'afficher leurs coor-

données et de tester les différentes fonctionnalités que vous avez implémentées.

Veillez à bien documenter votre classe en utilisant l'outil et les conventions Javadoc.

1.3 Vers une première classe robot

Maintenant que vous savez écrire une classe en Java et que vous avez pu la tester grâce à un premier programme en Java, penchons nous sur une première version de notre classe `Robot`.

Les caractéristiques principales d'une instance de notre classe `Robot` sont les suivantes :

- un nom,
- une position sur le plateau de jeu,
- un niveau d'énergie,
- une jauge de santé.

Vous pourrez rajouter ultérieurement d'autres caractéristiques au fur et à mesure de ce projet.

Implémentez la classe `Robot` avec les caractéristiques décrites ci dessus ainsi que l'ensemble des fonctionnalités permettant de :

- créer un robot,
- modifier et d'accéder à son nom,
- modifier et d'accéder à son niveau d'énergie,
- modifier et d'accéder à son niveau de santé,
- modifier et d'accéder à sa position sur le plateau de jeu,
- déplacer un robot,
- pouvoir afficher un robot et ses caractéristiques.

Testez votre classe `Robot` en écrivant un programme de test permettant de créer quelques instances de cette classe et de vérifier les fonctionnalités que vous venez de créer.

2 Deuxième séance : enrichissement de la classe `Robot` et définition d'une hiérarchie de classes

Si vous n'avez pas eu le temps de terminer la première version de votre classe `Robot`, prenez le temps de le faire.

2.1 Enrichissement de notre classe de robot

Avant d'aller plus loin, nous allons enrichir notre classe `Robot` :

- ajouter une méthode `recharger` permettant de redonner de l'énergie à un `Robot`,
- ajouter une méthode `reparer` permettant de redonner de la santé à un `Robot`,
- ajouter une méthode `depenserEnergie` permettant d'utiliser de l'énergie d'un `Robot`,
- ajouter une constante `ROBOT_MAX_ENERGIE` représentant le niveau d'énergie maximal d'un `Robot` (la valeur de cette constante est de 150),
- ajouter une constante `ROBOT_MAX_SANTE` représentant le niveau de santé maximal d'un `Robot` (la valeur de cette constante est de 200),
- ajouter une constante `ROBOT_DEFAUT_ENERGIE` représentant le niveau d'énergie par défaut d'un `Robot` (la valeur de cette constante est de 100),
- ajouter une constante `ROBOT_DEFAUT_SANTE` représentant le niveau de santé par défaut d'un `Robot` (la valeur de cette constante est de 100),
- ajouter un mécanisme permettant de connaitre le nombre d'objets de la classe `Robot` qui ont été créés (à vous de voir comment faire cela),
- modifier vos méthodes afin de représenter le fait que lorsqu'un `Robot` se déplace sur le plateau de jeu, cela lui coûte un point d'énergie.

2.2 Hiérarchie de robots

Réfléchissez à une hiérarchie de classes vous permettant de représenter différents types de `Robot` :

1. un robot "énergivore" (`RobotEnergivore`) : qui se préoccupera principalement de son niveau d'énergie,
2. un robot "combattant" (`RobotCombattant`) : qui abimera les robots qui auront le malheur de se trouver près de lui,
3. un robot "neuneu" (`RobotNeuneu`) : dont le comportement sera difficile à prévoir.

Pour le moment, **nous ne sommes pas capables de modéliser les comportements** de ces différents types de robots, cela viendra au fur et à mesure du projet. Néanmoins, nous souhaitons pouvoir différencier les différents types de robots, ainsi vous devrez déclarer pour chaque classe de robot une méthode capable d'afficher les objets sous la forme suivante :
"`Je m'appelle nomRobot, je suis un robot de type typeRobot, je possede`

`EE points d'energie, VV points de vie et je me trouve en position [X,Y].`"
avec :

- `nomRobot` représente le nom du robot,
- `typeRobot` est le type de robot (énergivore, combattant, neuneu),
- `EE` représente le nombre de points d'énergie du robot (vous devez afficher un nombre entier),
- `VV` représente le nombre de points de santé du robot (vous devez afficher un nombre entier),
- `X` et `Y` représentent les coordonnées (un point 2D) du robot sur le plateau de jeu.

2.3 Obstacles et hiérarchie d'obstacles

Écrivez la classe `Obstacle`, un obstacle empêche les robots de se positionner sur la case sur laquelle ils se trouvent. Ils n'ont aucune autre caractéristique particulière, mais nécessitent l'écriture des méthodes relatives à leurs attributs, ainsi qu'au moins un constructeur et une méthode permettant d'afficher un obstacle.

Comme pour les robots, nous vous demandons de réfléchir sur une hiérarchie d'objets de type `Obstacle`, et d'implémenter les classes correspondantes.

Rappelons que nous tablons au départ sur deux types d'obstacles :

- les obstacles fixes (`ObstacleFixe`) : qui ont une position sur le plateau de jeu et qui n'en bougent pas,
- les obstacles mobiles (`ObstacleMobile`) : qui peuvent se déplacer de case en case sur le plateau de jeu.

2.4 Bonus et hiérarchie de bonus

Il nous reste à représenter les bonus du jeu. Écrivez la classe `Bonus`, un obstacle empêche les robots de se positionner sur la case sur laquelle ils se trouvent, et peuvent leur rendre de l'énergie ou de la santé.

Enfin, vous pouvez écrire les classes représentant une hiérarchie de `Bonus` du jeu. Nous considérons pour le moment deux types de bonus :

- des bornes d'énergie (`BorneEnergie`) : dont la position est fixe sur le plateau de jeu, et qui possèdent une quantité limitée d'énergie et peuvent recharger les robots se trouvant dans les cases adjacentes,
- des bornes de santé (`BorneSante`) : dont la position est fixe sur le plateau de jeu, et qui possèdent une quantité limitée de points de

réparation. Elles peuvent réparer les robots se trouvant dans les cases adjacentes et ainsi remonter leurs niveaux de santé.

2.5 Plateau de jeu

La dernière étape de cette séance consiste à écrire une classe représentant le plateau de jeu (`PlateauJeu`) qui comporte les caractéristiques suivantes :

- une largeur (entier représentant la largeur du plateau de jeu, i.e. le nombre de cases en largeur),
- une hauteur (le nombre de cases en hauteur),
- un `Robot` de la classe que vous souhaitez,
- un obstacle (à déterminer parmi la hiérarchie définie plus haut),
- un bonus (de votre choix).

Vous devrez bien évidemment déclarer également toutes les méthodes que vous jugerez nécessaires pour cette classe.

Afin de tester toutes ces classes, écrivez un petit programme de test qui vous permet de créer un `PlateauJeu`.

Si vous avez le temps, faites déplacer le robot qui s'y trouve. Pour vérifier que tout va bien, vous devrez essayer de modifier la position du robot (et de l'obstacle si celui-ci est un `ObstacleMobile`) une fois et ré-afficher le plateau pour vérifier que tout va bien.

3 Troisième séance : polymorphisme et types abstraits, listes, etc.

Dans cette séance, nous allons nous atteler à la création de listes de différents robots, bonus et obstacles (pouvant être tous de **classes différentes**) et à comprendre pourquoi le mécanisme de **polymorphisme** permet de *gérer plus simplement des ensembles de données de types compatibles mais pas forcément identiques.*

Avant d'aller plus loin, assurez-vous d'avoir bien terminé et testé toutes les classes et fonctionnalités demandées dans la séance précédente.

3.1 Gestion d'un ensemble d'éléments de jeu

Afin d'étoffer un peu notre simulation, nous souhaitons pouvoir avoir plusieurs robots, obstacles et bonus sur notre plateau de jeu.

En vous basant sur les notions présentées en cours, modifiez la classe `PlateauJeu` afin de permettre la gestion **d'un ensemble de robots, d'obstacles et de bonus**.

Rajoutez une méthode `tourDeJeu` à la classe `PlateauJeu` qui permet d'afficher pour chacun des éléments du plateau de jeu (robots, obstacles, bonus) leurs caractéristiques (nom, attributs et positions). Pour ce faire, rappelez vous que chacune de vos classes doit déjà posséder une méthode `toString()`.

La mise en œuvre de votre solution pourra nécessiter des modifications dans un certain nombre de classes déjà écrites. En particulier, nous vous demandons de réfléchir aux aspects suivants :

- comment stocker les différents éléments du jeu ?
- comment organiser nos classes pour permettre d'effectuer plusieurs tours de jeu ?
- comment intégrer un joueur humain dans ce jeu de simulation ? (pour lui demander à chaque fin de tour si il veut continuer ou non la partie, i.e. **relancer un tour de jeu**)

Compléments sur la lecture au clavier en Java Java fournit différents mécanismes pour la lecture d'informations fournies au clavier par l'utilisateur. Le plus simple et qui nous convient parfaitement ici est l'utilisation de la classe `Scanner` du paquetage `java.util`.

Pour ce faire, nous devons :

- créer un objet de type `Scanner` en spécifiant à son constructeur que nous souhaitons scanner le clavier (représenté par l'objet statique `System.in` en Java)
- utiliser sa méthode `next()` : qui retourne la chaine de caractère (objet de type `String`) rentrée par l'utilisateur
- la classe `Scanner` fournit de nombreuses autres méthodes permettant de lire des entiers, etc., voir la Javadoc ici : http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

Voici un exemple de programme effectuant la lecture au clavier d'informations depuis le terminal (source http://java67.blogspot.fr/2012/12/how-to-read-user-input-from-console-in-java.html)

```
1 // Dans un fichier UserInputExample.java
2 import java.util.Scanner;
3
```

```
4 public class UserInputExample {
5     public static void main(String args[]) {
6
7         //Creating Scanner instance to scan console for User input
8         Scanner console = new Scanner(System.in);
9
10        System.out.println("System is ready to accept input, please enter
      name : ");
11        String name = console.nextLine();
12        System.out.println("Hi " + name + ", Can you enter an int number
      now?");
13        int number = console.nextInt();
14        System.out.println("You have entered : " + number);
15        System.out.println("Thank you");
16    }
17 }
```

Résultat :

System is ready to accept input, please enter name :
John
Hi John, Can you enter an int number now ?
56
You have entered : 56
Thank you

3.2 Avant d'aller plus loin !

À cet instant du projet, vous devez avoir terminé les éléments suivants :

- des classes permettant de représentant différents types :
  - de robots (`Robot`, `RobotCombattant`, `RobotEnergivore`, `RobotNeuneu`)
  - d'obstacles (`Obstacle`, `ObstacleFixe`, `ObstacleMobile`)
  - de bonus (`Bonus`, `BorneEnergie`, `BorneSante`)
- ces classes doivent également posséder chacune une méthode `toString()` permettant d'afficher en mode texte les informations relatives à ces classes
- une classe `PlateauJeu` qui contient :
  - un ensemble de :
    - robots
    - obstacles
    - bonus
  - une méthode `tourDeJeu` qui permet :
    - d'afficher les caractéristiques de chacun des éléments du jeu
    - de demander à un utilisateur si il/elle veut continuer la simulation

Si jamais vous n'avez pas terminé toutes ces classes, nous vous recommandons de terminer en priorité :

- **la classe `Robot` ainsi qu'une de ses sous-classes, p. ex. `RobotCombattant`**
- **la classe `Obstacle` ainsi qu'une de ses sous-classes, p. ex. `ObstacleFixe`**
- **la classe `Bonus` ainsi qu'une de ses sous-classes, p. ex. `BorneEnergie`**
- **la classe `PlateauJeu` complète !**

3.3 Une première simulation : des déplacements aléatoires simples

Le but de cette sous-section est de pouvoir réaliser une première version simple de la simulation des robots : l'idée est de permettre aux **robots de se déplacer de manière aléatoire sur le plateau de jeu**.

Toutefois, bien que paraissant relativement aisée, cette première version demande néanmoins à prendre en compte un certain nombre de remarques :

- les robots **ne peuvent pas se déplacer sur des cases en dehors du plateau de jeu** (rappelez-vous que celui-ci possède une `largeur` et une `hauteur` définissant le nombre de cases du plateau, de plus les positions des cases ne peuvent être négatives)
- les robots **ne peuvent pas se déplacer sur des cases occupées par d'autres éléments du jeu** (autres robots, bonus, obstacles)
- les robots ne peuvent se déplacer que sur des cases contiguës à celle sur laquelle il se trouve
- lorsqu'un robot **s'est déplacé sur le plateau de jeu, les cases libres et occupées doivent être mises à jour** (deux robots ne peuvent se trouver sur une même case, etc.)
- tout déplacement du robot sur le plateau va consommer un point d'énergie

Ces remarques nous amènent à la réflexion suivante : **les éléments du jeu (qui doivent se déplacer) ont besoin de connaitre l'ensemble du plateau de jeu** !

Étudiez les différentes solutions permettant de résoudre ce problème et choisissez en une.

Une fois que nos robots sont capables de connaitre le plateau de jeu en entier, ils vont pouvoir se déplacer. Pour ce faire, vous allez devoir rajouter

les méthodes suivantes à votre classe `PlateauJeu` :

- `horsPlateau` : méthode prenant en paramètre un objet de type `Point2D` et qui retourne un booléen valant 'vrai' si cette position est en dehors du plateau et 'faux' sinon
- `caseLibre` : prenant en paramètre un objet de type `Point2D` et retournant un booléen valant 'vrai' si la case correspondante est libre et 'faux' sinon
- `casesLibresAutourDe` : prenant en paramètre un objet de type `Point2D` représentant la position de la case étudiée et qui retourne une liste d'objets de type `Point2D` contenant toutes les cases libres autour de la case étudiée
- `deplacer` : méthode devant gérer le déplacement d'un robot et faisant appel, entres autres, aux méthodes décrites ci-dessus

Une fois toutes ces méthodes écrites et testées, vous allez pouvoir rajouter une méthode `jouer` à la classe `Robot` dont le but est de faire se déplacer de manière aléatoire les robots sur le plateau de jeu.

Cette méthode `jouer` pourra faire appel à une autre méthode `deplacer`, qui gèrera le déplacement d'un objet de type `Robot`. En effet, le tour d'un jeu d'un robot peu consister en un déplacement, mais pas forcément, il est ainsi plus logique de déclarer une méthode dédiée au déplacement des robots plutôt que d'implémenter ce déplacement à l'intérieur de la méthode `jouer`.

Gardez à l'esprit que pour se déplacer, un `Robot` **a besoin de connaître le plateau de jeu** ! Étudiez les différentes possibilités qui s'offrent à vous pour résoudre ce problème, argumentez votre choix et implémentez le !

Compléments sur la génération de nombre entier pseudo-aléatoire en Java Java fournit différents mécanismes pour la génération de nombres pseudo-aléatoires. Ici nous souhaitons pouvoir générer un nombre entier pseudo-aléatoire, pour ce faire, il faut utiliser la classe `Random` du paquetage `java.util`.

Pour ce faire, nous devons :

- créer un objet de type `Random`
- utiliser sa méthode `nextInt` : qui prend en paramètre un entier N et qui génère un nombre aléatoire dans l'intervalle $[0, N]$

Voici un exemple de programme générant dix nombres entiers pseudo-aléatoires (source http://www.javapractices.com/topic/TopicAction.do?Id=62)

```java
// Dans un fichier RandomInteger.java
import java.util.Random;

// On genere 10 entiers pseudo-aleatoires dans l'intervalle [0,99]
public class RandomInteger {
  public static void main(String[] args) {
    System.out.println("Generation de 10 nombres entiers dans
    l'intervalle [0,99]");

    // NB : il est INUTILE de creer plusieurs objets de type Random
    // UN SEUL suffit pour generer plusieurs
    // nombres pseudo-aleatoires
    Random generateurAleatoire = new Random();

    // Boucle de generation des 10 nombres
    for(int i=0; i<10; i++) {
      // 100 ici definit la borne sup de l'intervalle
      int entierAlea = generateurAlea.nextInt(100);
      System.out.println("On vient de generer : "+entierAlea);
    }
  }
}
```

3.4 Comparaisons de différents types de conteneurs en Java

En Java, il existe plusieurs manières de stocker des objets à l'intérieur de conteneurs, les structures de données les plus utilisées sont :

- les listes tableaux (`ArrayList` en Java)
- les listes chainées (`LinkedList` en Java)
- les tables de hachage (`HashMap` ou `TreeMap` en Java)

Comme nous l'avons vu en cours, ces structures de données ont chacune leurs particularités et doivent être choisies avec parcimonie dans vos projets informatiques en Java.

Afin d'illustrer les différences entre ces différentes structures de données, nous vous demandons :

- d'ajouter de très nombreux éléments de jeu dans vos listes de robots, obstacles et bonus, pour ce faire vous devrez :
  - augmenter la taille de votre `PlateauJeu` afin de disposer d'un nombre de cases suffisantes
  - utiliser la génération de positions aléatoires (en utilisant le générateur

aléatoire présenté ci dessus) afin de pouvoir générer facilement un grand nombre d'éléments de jeu

Nous voulons étudier les différences de comportement (en termes de temps d'exécution par exemple) en fonction des structures de données Java utilisées lorsque celles-ci contiennent un très grand nombre d'objets. Vous devrez donc réaliser des tests en stockant les listes d'objets décrites précédemment en utilisant soit :

- des objets de type `ArrayList`
- des objets de type `LinkedList`

Ainsi, nous souhaitons que vous testiez les cas de figure suivants :

- lors de l'insertion des nouveaux objets dans les listes, tester l'insertion :
  - en **tête de liste**
  - en **fin de liste**
  - en **milieu de liste**
- le parcours des listes d'objets pour par exemple afficher l'ensemble des éléments contenus sur le plateau de jeu
- la recherche d'éléments particuliers sur le plateau de jeu, que ceux-ci se trouvent plutôt en :
  - **tête de liste**
  - **fin de liste**
  - **milieu de liste**

Veillez à bien **mesurer l'impact du choix de la structure de donnée sur les performances de vos algorithmes** (en fonction des opérations d'insertion et d'affichage effectuées).

En fonction des résultats obtenus, et des opérations que vous allez réaliser dans notre projet, choisissez (en argumentant) la meilleure structure de donnée à utiliser dans notre choix.

4 Quatrième séance : Interfaces, classes abstraites

Cette séance de TP va nous amener à aborder les notions vues en cours d'interfaces et de classes et méthodes abstraites. Nous allons donc continuer à améliorer notre projet en y ajoutant de nouvelles fonctionnalités.

4.1 Avant de commencer cette nouvelle séance

Nous vous demandons de bien avoir réaliser les tests prévus en Section 3.4. En effet, il est important que vous mettiez en œuvre différentes structures de données et que vous puissiez constater les différences de performances en termes de temps de calcul (et/ou en termes d'espace mémoire occupé si vous le souhaitez).

Si vous le souhaitez, vous pouvez également utiliser d'autres structures de données que celles conseillées plus haut.

4.2 Intégration d'Interfaces dans notre projet

Faisons un point sur les classes et objets de notre projet SIROP, nous avons jusqu'ici des objets de type :

`Bonus` : des éléments fixes (i.e. qui ne se déplacent pas sur le plateau) du jeu qui permettent de réparer et/ou de recharger les robots.

`Robot` (et de ses sous-classes) : chaque robot doit être capable de se déplacer sur le plateau de jeu. En plus chacun des types de robot peut avoir des caractéristiques ou comportements qui lui sont propres.

`Obstacle` (et de ses sous-classes) : les obstacles empêchent les robots de se déplacer sur les cases du plateau de jeu qu'ils occupent. Toutefois, certains obstacles, les objets de type `ObstacleMobile` et doivent pouvoir se déplacer eux aussi sur le plateau de jeu. Pourtant, pour le moment ces derniers ne peuvent pas se mouvoir sur le plateau !

4.2.1 Mise en place du déplacement d'obstacles mobiles

Comme nous venons de le rappeler, nous avons écrit une classe d'`ObstacleMobile` mais ceux-ci sont pour le moment immobiles. Voyons comment nous pouvons remédier à ce problème.

Tout d'abord, précisons les conditions de déplacement d'un obstacle :

- il doit **respecter les mêmes contraintes qu'un robot** (déplacement seulement sur les cases contigües, interdiction de se déplacer sur une case déjà occupée, etc.)
- nous allons rajouter une petite contrainte supplémentaire pour les obstacles :

— les obstacles mobiles vont se déplacer **uniquement** de manière **horizontale** ou **verticale** (les déplacements en diagonale sont interdits)
— si aucun déplacement horizontal ou vertical n'est possible, alors l'obstacle reste immobile

Ensuite, nous **voulons nous assurer** que les objets de type `ObstacleMobile` (ou que les objets de types dérivés de cette classe) **soient obligés d'implémenter une méthode de déplacement** !

Comment faire ? Avec ce que vous connaissez sur l'héritage, le polymorphisme et les interfaces en Java, étudiez les différentes possibilités qui s'offrent à vous pour obliger les objets de type `ObstacleMobile` d'avoir une méthode de déplacement. Notez que **nous ne souhaitons pas** que les objets de type `ObstacleFixe` aient une méthode de déplacement puisqu'ils ne sont pas censés pouvoir se déplacer.

Gardez à l'esprit que pour se déplacer, les éléments de jeu doivent avoir connaissance du plateau de jeu pour connaître les cases libres, etc. ⇒ cela peut vous amener à **devoir modifier votre hiérarchie de classes** déjà établie.

4.2.2 Créer de nouveaux éléments de jeu : des bonus mobiles

En vous basant sur la solution mise en place pour les objets de type `ObstacleMobile`, **proposez une solution pour la mise en place** d'un nouveau type de `Bonus`, des **bonus mobiles**.

Choisissez le type de bonus que vous voulez mettre en place, et leur comportement. Bien évidemment, même si nous avons choisi de nommer notre super-classe `Bonus`, vous pouvez choisir de créer des éléments de jeu qui se comportement comme des **malus** et qui vont par exemple retirer des points de santé et/ou des points d'énergie aux robots !

5 Cinquième séance : premier bilan, diagrammes de classes, JavaDoc

Cette séance est consacrée à un premier bilan du projet. Nous vous demandons de :

- vous assurer d'avoir terminé les classes, méthodes et interfaces demandées dans les quatre séances précédentes

- pouvoir faire tourner la simulation de nos robots, obstacles et bonus avec affichage de ce qui se passe le tout en mode texte. Vous devrez offrir la possibilité à l'utilisateur d'arrêter ou non la simulation à chaque tour
- réaliser un diagramme de classes UML représentant la hiérarchie des classes et des interfaces de votre projet
- rédiger la JavaDoc de votre projet : pour chaque classe, méthode, attribut. Via votre IDE préféré, vous pourrez générer automatiquement la documentation sous forme de pages HTML (a l'instar de la JavaDoc disponible en ligne pour l'ensemble des classes de l'API Java)
- vous assurer d'avoir compris et de maîtriser toutes les notions vues en cours et en TD jusqu'à présent

Une fois toutes ces étapes complétées, **faites valider rapidement par votre encadrant de TD votre diagramme de classes**, avant de continuer le projet.

6 Sixième séance : gestion des entrées/sorties, écriture et lecture de fichiers

L'objectif de cette séance est de vous faire manipuler les mécanismes d'entrées/sorties en Java. En particulier, nous souhaitons vous faire **lire un fichier texte** qui correspondra à une **sauvegarde du plateau de jeu**, et vous faire **écrire un fichier texte**, ce qui correspondra à faire une **sauvegarde du plateau de jeu**, ce qui permettra de reprendre la simulation plus tard.

Lors de cette séance, vous serez amenés à manipuler des objets du paquetage `java.io`, qui regroupe l'ensemble des classes et méthodes fournies par Java pour la lecture et l'écriture de fichiers de plusieurs sortes. Nous rappelons que la documentation en ligne de Java pour ce paquetage se trouve ici : http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html

Dans notre projet, nous nous concentrerons sur des fichiers texte contenant uniquement des caractères et chaînes de caractères. Les principales classes que nous serons amenés à manipuler sont :

- `File`
- `FileReader`
- `FileWriter`
- `BufferedReader`
- `BufferedWriter`

6.1 Avant propos : les flux en Java

Pour gérer les entrées/sorties, c'est-à-dire l'échange d'informations entre une application Java et une source extérieure (une autre application Java, un fichier stocké sur le disque dur, des informations situées en mémoire vive, une connexion réseau, etc.), Java a recours à des **flux** (*stream* en anglais).

Les flux permettent de gérer cet échange de données, et ce de **manière toujours séquentielle**. En Java il existe énormément de classes permettant de gérer différents types de flux, mais on distingue plusieurs catégories principales :

- les flux d'entrée (*input stream*) et les flux de sortie (*output stream*),
- les flux de traitement de caractères (données textuelles) et les flux de traitement d'octets (données brutes)

Comme nous l'avons dit plus haut, nous nous intéresserons ici aux flux de traitement de caractères, mais utiliserons un type de flux d'entrée (lecture d'un fichier de sauvegarde) et un type de flux de sortie (écriture d'une sauvegarde).

6.1.1 Les flux de traitement de caractères

Ces flux transportent des données sous formes de caractères. Les classes Java qui gèrent ces flux héritent de deux classes abstraites : les classes `Reader` et `Writer`. Comme vous le verrez dans la Figure 2, il existe de nombreuses classes, ayant des caractéristiques différentes.

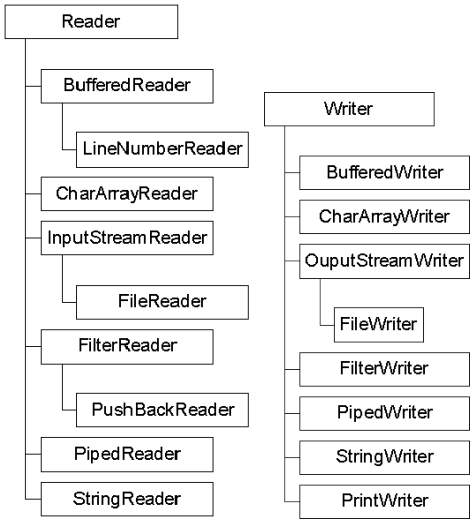

FIGURE 2 – Hiérarchie de classes pour les flux de traitement de caractères en Java.

Nous détaillerons dans les sous-sections suivantes l'utilisation de certaines de ces classes Java, pour la lecture et l'écriture de fichiers texte.

Avant de terminer cet avant-propos, il est important de noter que la plupart des méthodes des classes traitant les flux **sont susceptibles de lever des exceptions Java** ⇒ voir vos cours !

6.2 Format du fichier de sauvegarde

Avant de nous concentrer sur comment écrire et lire un fichier en Java, intéressons-nous au format que nous allons utiliser pour représenter un plateau de jeu dans un fichier texte. Avant de donner un exemple que vous pourrez utiliser pour tester vos méthodes, voici une description succincte de ce format :

- chaque ligne du fichier texte représente une seule information sur le plateau de jeu
- une information peut correspondre à :
  - la largeur du plateau de jeu
  - la hauteur du plateau de jeu
  - un des éléments du plateau de jeu (i.e. un robot, un bonus ou un obstacle)
- une ligne se suffit à elle-même, par exemple un robot sera décrit sur une seule ligne

L'exemple suivant vous donne un exemple de fichier respectant le format décrit de manière informelle ci-dessus :

```
Largeur 10
Hauteur 10
RobotCombattant Jack 0 0 100 100
RobotEnergivore John 5 5 300 50
RobotNeuneu Toby 2 2 80 38
RobotCombattant Bill 8 8 20 10
ObstacleMobile 1 9
ObstacleMobile 9 9
ObstacleFixe 3 6
ObstacleFixe 7 2
BorneEnergie 4 2 50
BorneSante 1 1 46
```

Comme nous l'avons décrit plus haut, les deux premières lignes mises à part (elles donnent la taille du plateau de jeu), **chaque ligne correspond à**

un élément du jeu et contient toutes les informations nécessaires à la création d'un objet dont le type est donné comme premier mot de la ligne, par exemple :

```
BorneEnergie 4 2 50
```

Signifie que vous devez créer un objet de type `BorneEnergie` en position $[4, 2]$ du plateau de jeu avec une quantité d'énergie de 50.

Maintenant que nous connaissons le format de fichier que nous allons utiliser pour le chargement et la sauvegarde de notre plateau de jeu, nous pouvons étudier le fonctionnement des classes Java permettant de lire et d'écrire des fichiers texte.

6.3 Chargement d'un fichier de sauvegarde

Dans cette section, nous allons voir :

1. comment fonctionne un `BufferedReader`, qui permet de lire un fichier texte ligne par ligne en retournant une chaîne de caractères par ligne
2. comment parcourir une chaîne de caractères contenant une ligne entière d'une fichier texte (i.e. comment découper cette chaîne mot par mot)
3. comment mettre en œuvre une sauvegarde de notre plateau de jeu et les conséquences sur nos classes

6.3.1 Utilisation d'un `BufferedReader`

La lecture d'un fichier texte en Java va s'effectuer en utilisant la classe `BufferedReader`. Comme nous l'avons vu plus haut (cf. Figure 2), il existe de nombreuses classes pour la lecture de flux, nous avons choisi `BufferedReader` car :

- elle nous permet de lire un fichier ligne par ligne
- elle offre de très bonnes performances, via l'utilisation d'un tampon (*buffer*, l'explication de ces bonnes performances est hors programme)

Voici un exemple d'utilisation de la classe `BufferedReader` qui va :

- ouvrir un fichier texte nommé "source.txt"
- lire ce fichier ligne par ligne avec la méthode `readLine()`
- afficher chaque ligne du fichier qui vient d'être lue

```
1 import java.io.*;
2
3 public class TestBufferedReader {
4   protected String source;
```

```
5
6   public TestBufferedReader(String source) {
7     this.source = source;
8     lecture();
9   }
10
11  public static void main(String args[]) {
12    new TestBufferedReader("source.txt");
13  }
14
15  private void lecture() {
16    try {
17      String ligne ;
18      BufferedReader fichier = new BufferedReader(new
      FileReader(source));
19
20      while ((ligne = fichier.readLine()) != null) {
21          System.out.println(ligne);
22      }
23
24      fichier.close();
25    } catch (Exception e) {
26      e.printStackTrace();
27    }
28  }
29 }
```

Notons que la création d'un objet de type `BufferedReader` (cf. ligne 18 de l'exemple) passe par la création d'un objet de type `FileReader`. Toutefois, comme celui-ci ne nous est pas indispensable, nous pouvons appeler le constructeur de `FileReader` à l'intérieur de l'appel du constructeur de `BufferedReader`.

De plus `BufferedReader` possède une méthode `readLine()` retournant une chaîne de caractères (`String`) correspondant à la ligne courante du fichier et décale un curseur interne lui permettant de parcourir tout le fichier.

6.3.2 Décomposition d'une chaîne de caractères en sous-chaînes

Maintenant que nous sommes capables d'obtenir une chaîne de caractères (i.e. un objet de type `String`) contenant une ligne complète d'un fichier texte, il nous faut maintenant être capable de la **découper mot par mot** afin de récupérer chaque terme qui nous intéresse.

En effet, supposons que nous ayons un objet de type `String` contenant

la ligne suivante :

```
Largeur 10
```

Nous voudrions être capable de découper cette chaîne de caractères en deux mots : `Largeur` et `10` !

Cela est faisable en utilisant la classe Java `StringTokenizer`. Cette classe permet de découper un objet de type `String` en fonction d'un ensemble de délimiteurs de mots. Et de parcourir ainsi l'ensemble des unités lexicales (mots séparés par les délimiteurs) contenus dans une chaîne de caractères.

Voici un exemple complet d'utilisation de `StringTokenizer` :

```
1  import java.io.*;
2
3  public class TestStringTokenizer {
4
5    public static void main(String args[]) {
6      String test = new String("Largeur 10");
7      String delimiteurs = " ,.;";
8
9      // on declare un 'tokenizer' qui va decouper
10     // 'test' en fonction de l'ensemble des delimiteurs
11     StringTokenizer tokenizer = new StringTokenizer(test,
       delimiteurs);
12
13     // Parcours de l'ensemble des unites lexicales de 'test'
14     // hasMoreTokens() retourne 'vrai' tant qu'il reste des 'mots'
15     // dans 'test' separes par un des delimiteurs (espace,
16     // virgule, etc.) declares plus haut
17     while(tokenizer.hasMoreTokens()) {
18       // nextToken() retourne la prochaine unite lexicale decoupee
       par les delimiteurs
19       String mot = tokenizer.nextToken();
20       // pour l'exemple, on transforme 'mot' en lettres minuscules
21       mot = mot.toLowerCase();
22       // on affiche 'mot' qui est maintenant en minuscules
23       System.out.println(mot);
24     }
25   }
26 }
```

L'extrait de code ci-dessus va afficher le résultat suivant :

```
largeur
10
```

6.3.3 Chargement d'une sauvegarde dans notre projet

Vous avez maintenant toutes les informations en main pour réaliser notre classe `ChargementPartie`. Cette classe aura pour but de charger un fichier de sauvegarde et de **retourner un objet de type** `PlateauJeu` qui permettra de reprendre la simulation là où elle avait été enregistrée.

Écrivez une classe `ChargementPartie` avec :

- un attribut représentant le nom du fichier de sauvegarde à charger
- un attribut de type `BufferedReader`
- un constructeur prenant en paramètre le nom du fichier à charger
- une méthode `chargerPartie` retournant un objet de type `PlateauJeu` contenant l'ensemble des éléments du jeu qui étaient sauvegardés dans le fichier texte

6.4 Sauvegarde de la simulation

Dans la section précédente, vous avez appris à manipuler des objets de types `BufferedReader` et `StringTokenizer` afin de pouvoir charger un fichier de sauvegarde. Nous allons maintenant pouvoir écrire une classe permettant la création d'un fichier respectant le format de sauvegarde et ainsi terminer notre mécanisme de sauvegarde/chargement de partie !

6.4.1 Utilisation d'un `BufferedWriter`

De manière identique à la lecture d'un fichier texte en Java, basée sur l'utilisation de la classe `BufferedReader`, notre mécanisme de sauvegarde va être basé sur l'utilisation de la classe `BufferedWriter`, cf. Figure 2.

Comme son pendant dédié à la lecture, la classe `BufferedWriter` possède de nombreux avantages, en particulier :

- elle nous permet d'écrire un fichier ligne par ligne
- elle offre de très bonnes performances, via l'utilisation d'un tampon (*buffer*, l'explication de ces bonnes performances est hors programme)

Voici un exemple d'utilisation de la classe `BufferedWriter` qui va :

- ouvrir un fichier texte nommé "source.txt"
- lire ce fichier ligne par ligne avec la méthode `readLine()`
- afficher chaque ligne du fichier qui vient d'être lue

```
1 import java.io.*;
2
3 public class TestBufferedWriter {
4
```

```
5   public static void main(String args[]) {
6     BufferedWriter bufferedWriter = null;
7     String filename = "monFichier.txt";
8
9     try {
10
11      // Creation du BufferedWriter
12      bufferedWriter = new BufferedWriter(new
      FileWriter(filename));
13
14      // On ecrit dans le fichier
15      bufferedWriter.write("Ecriture ligne un dans le fichier");
16      bufferedWriter.newLine();
17      bufferedWriter.write("Ecriture ligne deux dans le fichier");
18
19    }
20    // on attrape l'exception si on a pas pu creer le fichier
21    catch (FileNotFoundException ex) {
22      ex.printStackTrace();
23    }
24    // on attrape l'exception si il y a un probleme lors de
      l'ecriture dans le fichier
25    catch (IOException ex) {
26      ex.printStackTrace();
27    }
28    // on ferme le fichier
29    finally {
30
31      try {
32        if (bufferedWriter != null) {
33          bufferedWriter.flush();
34          bufferedWriter.close();
35        }
36      }
37      // on attrape l'exception potentielle
38      catch (IOException ex) {
39        ex.printStackTrace();
40      }
41    }
42  }
43 }
```

Notons que la création d'un objet de type `BufferedWriter` (cf. ligne 12 de l'exemple) passe par la création d'un objet de type `FileWriter`. Tou-

tefois, comme celui-ci ne nous est pas indispensable, nous pouvons appeler le constructeur de `FilWriter` à l'intérieur de l'appel du constructeur de `BufferedWriter`.

De plus `BufferedWriter` possède une méthode `write()` permettant d'écrire une chaîne de caractères dans le fichier. Notez bien qu'il est nécessaire d'ajouter vous même les retours à la ligne si votre chaîne de caractères n'en contient pas ! Cela s'effectue grâce à la méthode `newLine()`.

Rappel : pour rajouter un retour à la ligne à une chaîne de caractères, vous devez y ajouter le caractère spécial `\n`.

Exemple :

```
1 import java.io.*;
2
3 public class TestString {
4
5   public static void main(String args[]) {
6     String sansRetourLigne = new String("Bla");
7     String avecRetourLigne = new String("Bla\n");
8   }
9 }
```

6.4.2 Sauvegarde d'un plateau de jeu dans notre projet

Vous avez maintenant toutes les informations en main pour réaliser notre classe `SauvegardePartie`. Cette classe aura pour but de sauvegarder l'état courant d'un objet de type `PlateauJeu` dans un fichier texte qui respectera le format de fichier présenté dans la Section 6.2.

Écrivez une classe `SauvegardePartie` avec :

- un attribut représentant le nom du fichier de sauvegarde à créer
- un attribut de type `BufferedWriter`
- un constructeur prenant en paramètre le nom du fichier à sauvegarder
- une méthode `sauvegarderPartie` prenant en paramètre un objet de type `PlateauJeu` contenant l'ensemble des éléments du jeu