TP R2.03 – Les exceptions en Java

Important : pour certains exercices, la consultation de l'API Java 1.8 est indispensable

https://docs.oracle.com/javase/8/docs/api/

Objectifs du TP (4 X 1h30)

- Comprendre une documentation de classes susceptibles de lancer des exceptions. Savoir capturer des exceptions (application *JeuRoulette*).
- Coder dans sa propre classe des méthodes susceptibles de lancer des exceptions. Savoir tester correctement ces méthodes (applications *Parking* et *MyScanner*).

Codage et rendu

On suppose une arborescence de développement habituelle et obligatoire :

/TPJeuRoulette			*/TPParking*			*/TPMyScanner*		
/	|	\	/	|	\	/	|	\
/ws	*/src*	*/class*	*/ws*	*/src*	*/class*	*/ws*	*/src*	*/class*

Les compilations et les exécutions doivent être lancées avec les commandes *javac* et *java* dans */ws* (version de java : **java11 maximum**). Votre *CLASSPATH* doit être déclaré et doit contenir le chemin *../class*.

Tous vos sources *java* (fichiers *.java* !) doivent être rassemblés dans une archive à votre nom (*Nom_Prénom.zip*). Ce **rendu individuel** doit être déposé sur Moodle R2.03 (https://moodle.univ-ubs.fr/course/view.php?id=7556) pour le dimanche 21/05 à 23h55 au + tard (**attention malus -2 points si retard**).

Exercice 1 : JeuRoulette

L'application *JeuRoulette* génère et capture des exceptions. Elle est constituée de 3 classes : le lanceur (classe *JeuRoulette*), la classe *Roulette* et la classe *Aleatoire*. Pour information, le jeu « la roulette » est cette roue dans un casino qui place aléatoirement une bille sur un nombre en tournant.

Il faut, bien entendu, d'abord bien comprendre le contenu et le rôle de chacune de ces 3 classes. Pour cela, consulter la *javaDoc* des classes *Roulette* et *Aleatoire* sur Moodle, de même que le source (incomplet !) de la classe *JeuRoulette*. Les *.class* de *Roulette* et *Aleatoire* sont fournis et doivent être placés dans */class*.

Question : compléter le code de la classe *JeuRoulette.java* fournie pour capturer les exceptions et permettre au jeu de fonctionner correctement. En l'état, la classe compile et s'exécute.

Attention - il y a 2 paramètres à passer en ligne de commande lors de l'exécution du jeu : <nomJou> et <nbCases> (voir code source).

La ligne de commande s'écrit alors : *java JeuRoulette <nomJou> <nbCases>*. Si vous ne spécifiez ni <nomJou>, ni <nbCases>, l'application s'arrête immédiatement avec l'exception *ArrayIndexOutOfBoundsException*. Pourquoi ?

Exercice 2 : Parking

On désire développer une application qui simule le fonctionnement d'un parking de voitures. On décidera du nombre de places disponibles sur ce parking au départ (il s'agira d'une constante Java *private static final int NB_PLACES = …*).

On modélisera deux actions possibles :

- Garer une voiture sur le parking : une voiture peut se garer à une place à condition que cette place soit libre (*null* en pratique). Si la place n'est pas libre, une exception *RuntimeException* est lancée avec le message d'erreur "*Place déjà occupée*".
- Sortir une voiture du parking : consiste à donner un numéro de place de parking et à renvoyer la voiture stationnée à cette place. S'il n'y a pas de voiture garée à la place de parking, une exception *RuntimeException* est lancée avec le message d'erreur "*Pas de voiture à cette place*".

Dans tous les cas, il faudra vérifier que le numéro de place est compatible avec la plage des valeurs possibles (c'est-à-dire *1 <= numéro de place <= NB_PLACES*). Si le numéro de place n'est pas valable, une exception *ArrayIndexOutOfBoundsException* est lancée avec le message d'erreur "*Numéro de place non valide*".

Mise en œuvre

1. Coder une classe *Voiture* qui contiendra :
   - 3 attributs qui caractérisent une voiture : *String marque* (ex. Renault), *String modele* (ex. Scénic), *int puissance* (ex. 110),
   - un constructeur qui prend 3 paramètres,
   - une méthode qui renvoie une chaîne de caractères qui contient les attributs d'une voiture : *public String toString()*.

   Tester cette classe : classe *TestVoiture* contenant un lanceur et une seule méthode *void testConstructeuEtToString()* qui teste simultanément le constructeur et la méthode *toString* de la classe *Voiture*.

2. Coder la classe *Parking* (respecter l'ordre indiqué pour le développement).
   - Déclarer la constante *NB_PLACES*.
   - Déclarer 1 attribut *lesPlaces* qui est une référence sur un tableau de *Voiture*.
   - Ecrire le constructeur qui construit simplement le tableau de *Voiture* à la taille *NB_PLACES*. A l'issue de cette construction, le parking est créé et toutes les places sont libres. En effet, on considère qu'une place est libre lorsque la case du tableau correspondant au numéro de place contient la valeur *null*. Les places de parking sont numérotées de *1* à *NB_PLACES*.
   - Ecrire la méthode *public String toString()* qui renvoie une chaîne de caractères qui contient l'état du parking. C'est-à-dire pour chaque place de parking, le numéro ainsi que les caractéristiques de la voiture qui l'occupe (ou simplement « place libre » si aucune voiture n'est garée à cette place).
   - Ecrire une méthode privée *void numeroValide (int numPlace) throws ArrayIndexOutOfBoundsException* qui vérifie que le numéro de place est valide (c'est-à-dire *1 <= numéro de place <= NB_PLACES*). Dans le cas contraire, lance l'exception avec le message "*Numéro de place non valide*".
   - Ecrire la méthode publique *void garer (Voiture voit, int numPlace) throws RuntimeException* qui gare la voiture passée en paramètre à la place *numPlace*. Vérifier d'abord que le numéro de place est valide (méthode *numeroValide(…)*) et vérifier ensuite que la place est disponible sinon lancer l'exception avec le message "*Place déjà occupée*".
   - Ecrire la méthode publique *Voiture sortir (int numPlace) throws RuntimeException* qui renvoie la voiture garée au numéro de place passé en paramètre. Vérifier d'abord que le numéro de place est valide et vérifier ensuite qu'une voiture est bien garée à cette place sinon lancer l'exception avec le message "*Pas de voiture à cette place*". Une fois la voiture sortie, remettre la case à la valeur *null* puisqu'elle est à nouveau disponible.

Séquencement du développement - Tests

Il s'agit d'écrire une classe de test (*TestParking*) qui met en évidence le bon fonctionnement de la classe *Parking* et des exceptions (lancement des bonnes exceptions avec le bon message).

Prendre un parking de taille réduite (*NB_PLACES = 4* par ex.) pour faciliter les tests.

Séquencement du développement/test :

- DES la fin de l'écriture du constructeur et de la méthode *toString()* de la classe *Parking*, IL FAUT DEJA tester l'affichage de votre parking qui DOIT indiquer que toutes les places sont libres. Pour cela écrire la méthode de test *void testConstructeurEtToString()* qui teste simultanément le constructeur et la méthode *toString* de la classe *Parking*.
- Coder la méthode privée *numeroValide(…)*. Celle-ci étant privée, son test se fera à travers les méthodes *garer(…)* et *sortir(…)*.
- Coder la méthode *garer(…)*. Considérer 3 cas de test dans une méthode *void testGarer()* : test du fonctionnement normal sans aucune exception et test des cas d'erreurs.

  Pour les cas d'erreurs, il y en a deux :

  - premier cas d'erreur : test avec capture (*try/catch*) obligatoire de l'exception *ArrayIndexOutOfBoundsException* avec affichage du message "*Numéro de place non valide*" (lancement de l'exception par la méthode *numeroValide(…)*),
  - deuxième cas d'erreur : test avec capture obligatoire de l'exception *RuntimeException* et affichage du message "*Place déjà occupée*".
- Coder la méthode *sortir(…)*. Considérer 3 cas de test dans une méthode *void testSortir()* : test du fonctionnement normal sans aucune exception et test des cas d'erreurs.
  - premier cas d'erreur : test avec capture obligatoire de l'exception *ArrayIndexOutOfBoundsException* et affichage du message "*Numéro de place non valide*",
  - deuxième cas d'erreur : test avec capture obligatoire de l'exception *RuntimeException* et affichage du message "*Pas de voiture à cette place*".

Exercice 3 : MyScanner

Construire l'équivalent de la classe *java.util.Scanner* de l'API Java. La classe *Scanner* de l'API permet, en résumé, de faire de la lecture de texte (à partir du clavier, à partir d'un fichier, à partir d'une chaîne de caractères). On se limitera au développement de quelques méthodes qui peuvent servir à la lecture de fichiers textes (notamment).

Cette classe se nommera *MyScanner* et nous voulons respecter pour cette classe *MyScanner* exactement la signature des méthodes de *java.util.Scanner* (y compris le lancement des exceptions, d'où l'importance de consulter la javaDoc de *java.util.Scanner*), à savoir :

- *public MyScanner (File source) throws FileNotFoundException* (voir constructeur de *java.io.FileReader*)
- *public void close ()* (appelle simplement la méthode *close()* de l'objet *BufferedReader* déclaré en attribut)
- *public String nextLine () throws IllegalStateException, NoSuchElementException*
  - exception *NoSuchElementException* : si pas de ligne suivante à lire (i.e. fin de fichier atteinte)
  - exception *IllegalStateException* : si le scanner est fermé (i.e. *isClosed* est à vrai)
- *public boolean hasNextLine () throws IllegalStateException*
  - exception *IllegalStateException* : si le scanner est fermé (i.e. *isClosed* est à vrai)

Cette classe, *MyScanner*, déclare en plus 2 attributs privés : *BufferedReader br* et *boolean isClosed*.

Le booléen *isClosed* est à faux dès que la construction d'un objet *MyScanner* a réussi. Il est à vrai dans le cas contraire (échec de la construction) ou lorsque la méthode *close()* a été appelée.

L'objet de type *BufferedReader* est celui par l'intermédiaire duquel toutes les opérations de lecture, fermeture et détection de fin de fichier doivent se faire.

Pour information :

Un flux en entrée est d'abord connecté au fichier texte de cette manière :

- *File f = new File (nomFichier);* // *nomFichier* de type *String*
- *FileReader fr = new FileReader (f);* // ce constructeur lance *FileNotFoundException*

Ensuite, lier le lecteur *BufferedReader* à ce flux en entrée :

BufferedReader br = new BufferedReader (fr);

Travail à réaliser

1. En utilisant les méthodes de la classe *java.io.BufferedReader* (voir *javaDoc* en *https://docs.oracle.com/javase/8/docs/api/*), coder les 4 méthodes (dont le constructeur) ci-dessus et tester intégralement cette classe (*TestMyScanner*) au point 2. Faire appel à la classe *java.util.Scanner* est évidemment interdit.

   Attention : la méthode *hasNextLine()* de la classe *MyScanner* est à réaliser en dernier lieu (car + compliquée). Elle fera usage des méthodes *void mark (int readAheadLimit)*, *int read()* et *void reset()* de la classe *java.io.BufferedReader*. On choisira la valeur 100 pour le paramètre *readAheadLimit* de la méthode *mark(…)* (lire impérativement la documentation de la classe *java.io.BufferedReader* pour comprendre le rôle de la méthode).

2. Classe *TestMyScanner*
   - test du constructeur (*void testConstructeur()*) : cas normaux, cas d'erreurs
   - test méthode close (*void testClose()*) : cas normaux
   - test méthode hasNextLine (*void testHasNextLine()*) : cas normaux
   - test méthode nextLine (*void testNextLine()*) : cas normaux, cas d'erreurs