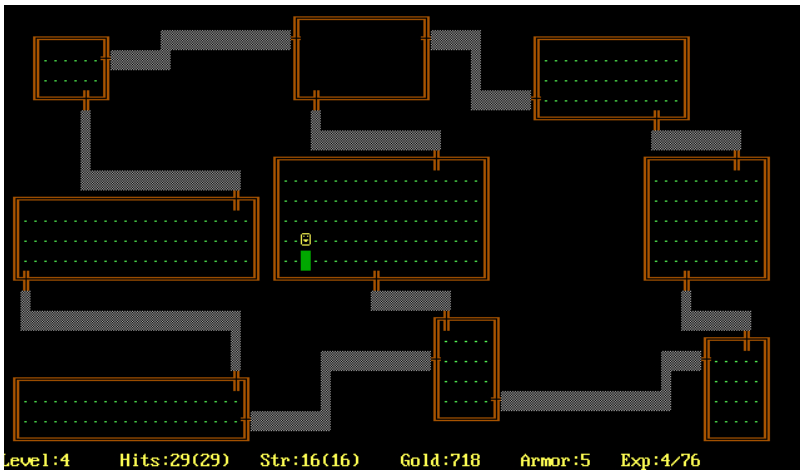
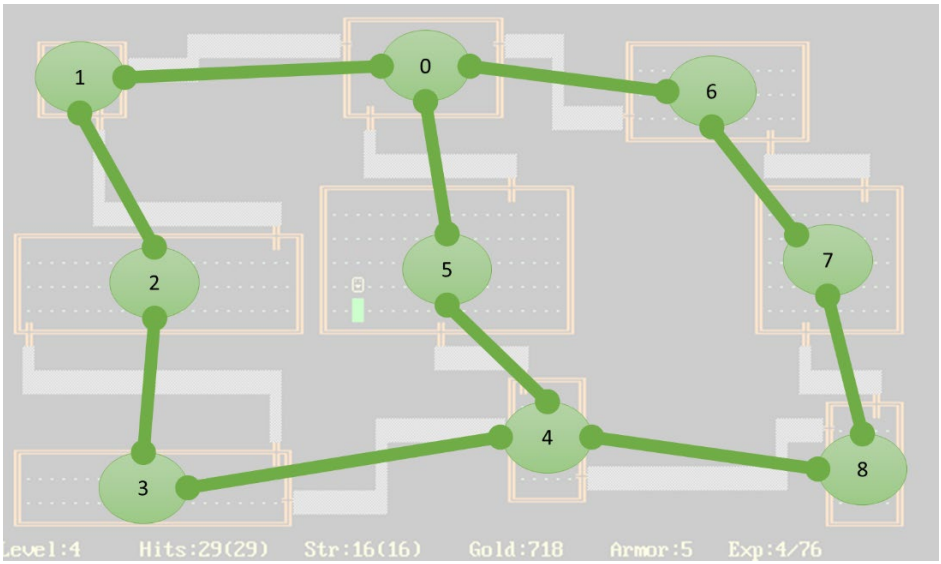
IFT1025 PROGRAMMATION 2 – AUTOMNE 2024 – TRAVAIL PRATIQUE 1

Dans ce TP vous prenez le point de vue des testeurs des jeux vidéo. Vous allez créer l'infrastructure de base pour un jeu du type « Rogue-like ». Pour un tel jeu nous devons représenter un labyrinthe (donjon), dans lequel on peut rencontrer diverses choses, comme des monstres, des trésors, ou même des hyper-méchants Boss.



Vous allez représenter un labyrinthe comme un graphe, où chaque pièce est un nœud et chaque corridor entre pièces est une arête. Pour ce faire, vous allez utiliser les listes d'adjacence, implémentées avec des tableaux. La liste d'adjacence d'un nœud est la liste de ses voisins.



Par exemple, pour le labyrinthe ci-dessus vous avez les listes d'adjacence[1] :

- 0 : 1, 5, 6
- 1 : 0, 2
- 2 : 1, 3
- 3 : 2, 4
- 4 : 3, 5, 8
- 5 : 0, 4
- 6 : 0,7
- 7 : 6, 8
- 8 : 4, 7

[1] L'ordre d'apparition des nombres dans chaque liste n'est pas important.

Chaque pièce va contenir une « rencontre » (monstre, trésor, etc). Pour représenter les différents types de rencontres, vous allez utiliser un «type énuméré» (Voir : Enum et Java Enums) : une classe qui regroupe une liste de constantes.

Nous vous donnons un code-squelette qui comporte 3 packages et 8 fichiers java :

Le package labyrinthe.code_squelette contient du code **que vous ne devez pas changer**. Assurez-vous de bien lire et comprendre le code et la javadoc fournis dans ces fichiers! Le package contient :

1. Interface Labyrinthe : définit les méthodes nécessaires pour la création d'une carte du jeu, qui est en fait un graphe non-orienté.
2. Classe Piece : un nœud dans le graphe du Labyrinthe, qui contient une rencontre et a un nombre qui sert d'identifiant (ID) unique.
3. Énumération RencontreType : définit 4 types de rencontres (rien, monstre, trésor, boss).
4. Classe abstraite Aventure : définit des opérations intéressantes sur la carte du jeu.
5. Classe Extérieur : une pièce spéciale qui représente l'extérieur du labyrinthe, avec ID=0.

Vous devez écrire votre code dans le package labyrinthe.soumission. Vous pouvez tout changer dans ce package, à condition de respecter ces trois exigences :

1. Classe DIROgue : contient une méthode main pour exécuter le programme.
2. Classe MonLabyrinthe : une classe non-abstraite qui doit implémenter l'interface Labyrinthe.
3. Classe MonAventure : une classe non-abstraite qui doit hériter de la classe Aventure.

Le squelette contient également un troisième package vide (labyrinthe.rencontres), pour la partie E du TP.

Cet énoncé comporte plusieurs pages. **Assurez-vous de toutes les lire, avant de commencer :**

A) VOTRE LABYRINTHE (30%)

Dans le package labyrinthe.soumission, créez une classe MonLabyrinthe qui implémente l'interface Labyrinthe tout en implémentant toutes les méthodes nécessaires, **selon les javadoc fournies** dans l'interface Labyrinthe :

1. getPieces()
2. nombreDePieces()
3. ajouteEntree(Exterieur, Piece)
4. ajouteCorridor(Piece, Piece)
5. existeCorridorEntre(Piece, Piece)
6. getPiecesConnectees(Piece)

Comme expliqué dans l'introduction, un Labyrinthe est un graphe non-orienté, représenté par des listes d'adjacence. Chaque liste d'adjacence devrait être un tableau de Pieces (Piece[]).

Vous pouvez faire les hypothèses suivantes :

- Le labyrinthe contient au maximum 50 pièces.
- Une Piece peut être connectée via corridors à un maximum de 8 autres Pieces.
- L'Exterieur fait partie du Labyrinthe.

B) VOTRE AVENTURE (10 %)

Les classes qui implémentent l'interface Labyrinthe ne représentent que les *aspects spatiaux* du jeu. Les classes qui héritent de la classe abstraite Aventure *contiennent* un Labyrinthe et nous permettent d'interroger le labyrinthe du point de vue *du joueur*. Cela se fait grâce à un ensemble de méthodes qui nous permettent d'analyser le contenu d'un donjon : par exemple, s'il contient des dangers, des trésors, s'il est gagnable, etc.

Créez une classe MonAventure, sous-classe non-abstraite de la classe Aventure, tout en implémentant les méthodes nécessaires, **selon les javadoc fournies** dans la classe Aventure.

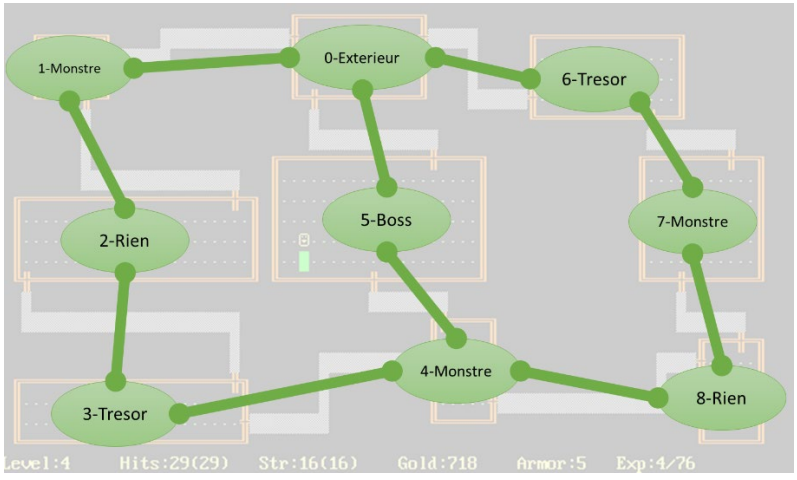
1. estPacifique()
2. contientDuTresor()
3. getTresorTotal()
4. contientBoss()

C1) PARCOURIR LE LABYRINTHE (10%)

Pour que le jeu soit gagnable il faut avoir un chemin dans le labyrinthe, en partant de l'extérieur jusqu'au Boss. Implémentez la méthode cheminJusquAuBoss() de votre classe MonAventure, **selon la javadoc fournie** dans la classe Aventure.

Nous faisons l'hypothèse que les labyrinthes doivent être construits de manière qui nous permet de trouver **facilement** un chemin vers le Boss. En commençant par l'Extérieur (une pièce unique avec ID=0), il faut suivre les ID croissants jusqu'à une des finalités suivantes : (a) on trouve le Boss, ou (b) le prochain ID n'est pas valide, ou (c) il n'y a pas de pièce avec le prochain ID. Dans le cas (a), on retourne le chemin (un tableau de pièces avec des ID en ordre croissant, terminant avec la pièce qui contient le Boss). Dans les cas (b) et (c), on retourne un tableau vide.

Il suffit d'arrêter au premier Boss trouvé.



Dans l'exemple ci-dessus, le chemin vers le Boss sera : {0, 1, 2, 3, 4, 5}.

L'annexe contient des contre-exemples (cas où notre méthode devrait retourner un tableau vide).

C2) BONUS – CHEMIN DU HÉRO (5%)

Cette question est optionnelle et vous n'êtes pas obligé de le faire. Le but est d'implémenter un algorithme de recherche plus efficace pour trouver le boss. La méthode cheminJusquAuBoss() utilise un algorithme assez naïf. Par conséquent, on veut implémenter un algorithme de recherche en profondeur pour cette méthode (Depth-First-Search), de manière récursive, qui devrait toujours trouver un chemin vers le boss. Ce chemin ne sera pas toujours le plus court, mais il devrait toujours le trouver s'il existe.

Le but de cet algorithme est de choisir un chemin et d'aller le plus loin possible sur ce chemin, jusqu'à ce que nous arrivions à une impasse (ou une pièce que nous avons déjà visitée). Si nous sommes arrivés à un des deux cas, nous retournons en arrière, jusqu'à la dernière pièce que nous avons visitée, qui a un voisin que nous n'avons pas encore visité et on continue dans ce chemin. À chaque étape, on sauvegarde la pièce qu'on vient de visiter dans un tableau. On arrête l'algorithme lorsqu'on trouve le boss et on retourne le bon chemin (pas tous les nœuds visités, seulement le bon chemin vers le boss).

En résumé, vous devez implémenter une nouvelle fonction (cheminDuHero()) qui s'appelle elle-même de façon récursive (il y a un exemple avec DFS sur Wikipédia) (voici une autre ressource pour vous aider avec DFS) et qui implémente l'algorithme DFS. Évidemment, vous devrez appeler cette nouvelle méthode à partir de la méthode cheminJusquAuBoss() et retourner le résultat qu'elle donne comme la réponse. Faites bien attention à vos cas de base et vos conditions d'arrêts, car ils sont **cruciaux** pour un bon algorithme récursif.

Cette question est considérée difficile pour votre niveau et elle est ajoutée au devoir pour vous permettre d'explorer de nouveaux sujets de programmation avancés pour ceux que ça intéresse. Elle permet aussi d'ouvrir la discussion sur les algorithmes de recherches pour les graphes non-orientés, ainsi que le paradigme des méthodes récursives. Même si vous ne réussissez pas à terminer le bonus, vous aurez acquis du nouveau savoir en programmation et de nouvelles manières de penser pour résoudre des problèmes de recherche.

Pour ceux qui vont se tenter à ce bonus, la version du corrigé se tient en 20 lignes de codes (commentaires inclus), donc si vous commencez à écrire 100 lignes, vous n'êtes peut-être pas sur le bon chemin. N'oubliez pas votre javadoc!

N'utilisez que les concepts vus en classe à date (donc pas d'ArrayList, LinkedList, LinkedHashSet... ou autre chose que vous pourriez trouver sur internet).

D) CRÉER UNE INTERFACE (20%)

Dans la classe DIROgue, complétez la méthode main(), qui, en utilisant un Scanner et vos classes MonAventure et MonLabyrinthe, nous permet de créer un labyrinthe et générer un rapport de l'aventure. L'idée est que l'utilisateur ajoute des pièces, puis les corridors, et il signale qu'il a terminé en marquant le mot clé FIN (tout en majuscule). L'utilisateur utilise également le mot clé CORRIDORS (tout en majuscule) pour signaler que toutes les pièces ont été ajoutées.

Après FIN, votre programme doit générer le rapport en appelant la méthode **genererRapport()** et l'afficher à la sortie du programme.

En tant qu'utilisateur vous avez 3 types de « commande » possibles en entrée :

1. piece <id> <rencontre>
   exemple : piece 1 monstre
2. corridor <id> <id>[2]
   exemple : corridor 0 1
3. CORRIDORS
4. FIN

A titre d'exemple de la partie C, l'interaction pourrait être comme suit[3] :

// entrée de l'utilisateur
piece 1 monstre
piece 2 rien
piece 3 tresor
piece 4 monstre
piece 5 boss
piece 6 tresor
piece 7 monstre
piece 8 rien
CORRIDORS
corridor 1 0
corridor 1 2
corridor 3 2
corridor 3 4
corridor 4 5
corridor 0 5
corridor 0 6
corridor 6 7
corridor 7 8
corridor 4 8
FIN

// sortie du programme
Rapport:
Donjon avec 9 pieces :
<0-RIEN> : [<1-MONSTRE>, <5-BOSS>, <6-TRESOR>]
<1-MONSTRE> : [<0-RIEN>, <2-RIEN>]
<2-RIEN> : [<1-MONSTRE>, <3-TRESOR>]
<3-TRESOR> : [<2-RIEN>, <4-MONSTRE>]
<4-MONSTRE> : [<3-TRESOR>, <5-BOSS>, <8-RIEN>]
// le reste du donjon…..
Non pacifique.
Contient un boss.
Contient 2 tresors.
Chemin jusqu'au boss :
<0-RIEN>
<1-MONSTRE>
<2-RIEN>
<3-TRESOR>
<4-MONSTRE>
<5-BOSS>

[2] Si lors de la création d'un corridor on nous donne l'ID d'une pièce qui n'existe pas, on doit s'arrêter avec un message d'erreur.
[3] Les lignes qui commencent par «//» ne font pas partie de l'interaction.

E) REMPLACER LES RENCONTRES PAR UNE SOUS-HIERARCHIE DE CLASSES (20%)

Dans le package nommé labyrinthe.rencontres, implémentez une hiérarchie de classes de rencontres.

Définissez une classe racine Rencontre, et des sous-classes pour les autres rencontres. Définissez trois types de monstres : Gobelin, Orque, et Gargouille, et trois types de trésor : SacDeButin, Potion, ArtefactMagique. Un Boss doit être un cas spécial de Gargouille.

Chaque classe doit définir une méthode **rencontrer()** qui retourne un string comme suit :

- Pour les monstres : “Un [type de monstre] affreux!”
- Pour les trésors : “[type de trésor]! Quelle chance!”
- Pour le type de rencontre Rien : “Un moment pacifique.”
- Pour le boss : “La bataille finale!”

F) IMPRIMER LE SCENARIO (10%)

Ajoutez une fonctionnalité dans votre classe DIROgue : Après avoir imprimé le rapport d’un donjon selon les instructions de la partie D, votre programme doit générer le « scenario » en appelant la méthode **genererSenario().** Cette méthode imprime le scénario de l'aventure comme suit :

Pour toute pièce dans le *cheminJusquAuBoss()*, on vérifie quel *RencontreType* est stocké et on crée un objet approprié de la hiérarchie définie dans la partie E. Ensuite on appelle la méthode *rencontrer()* de cet objet.

Pour chaque objet monstre ou trésor rencontré, utilisez la classe java.util.Random pour instancier un monstre ou un trésor plus spécifique, comme suit :

```
Monstre x;
int r = new Random().nextInt(3);
switch (r) {
case 0:
        x = new //… choix 1
        break;
case 1:
        x = new //… choix 2
        break;
case 2:
        x = new //… choix 3
}
```

Pour notre exemple de la partie D, alors, on pourrait avoir à la fin le scenario randomisé:

Scenario :
Un moment pacifique.
Un gobelin affreux!
Un moment pacifique.
Potion! Quelle chance!
Un gobelin affreux!
La bataille finale!

PRÉCISIONS GLOBALES

Travaillez en équipes de 2. Le travail tout seul ne sera pas permis sauf aux rares cas et avec approbation antérieure du professeur. Le TP est dû le **vendredi 1 novembre à 23h59** via StudiUM. Aucun retard ne sera accepté.

Un membre de l'équipe doit soumettre un ZIP avec :

a) Un fichier **README.TXT** avec les noms des 2 membres de l'équipe et toute information qui nous aidera à compiler et exécuter votre code. (Soyez concis – Ce fichier n'a pas besoin d'être très long !)
b) Un dossier labyrinthe avec les sous dossiers suivants :
    a. labyrinthe/soumission : avec les fichiers MonLabyrinthe.java, MonAventure.java, DIROgue.java
    b. labyrinthe/rencontres : avec les fichiers Rencontre.java, Rien.java, Monstre.java, Tresor.java, Gobelin.java, Orque.java, Gargouille.java, SacDeButin.java, Potion.java, ArtefactMagique.java, Boss.java
c) Tout autre fichier nécessaire (par exemple si vous décidez d'ajouter des classes ou des méthodes auxiliaires). Tout code additionnel **doit avoir la javadoc complète**.

L'autre membre doit soumettre juste une copie du fichier README.TXT (les deux fichiers doivent être identiques).

Votre code doit être compilable et exécutable (même s'il manque quelques fonctionnalités). **Code qui ne compile ou n'exécute pas sera accordé un 0**.

Tout élément **public** de votre code, incluant les **classes**, les **méthodes** – **sauf celles avec la balise @Override** (car elles héritent la javadoc des méthodes qu'elles remplacent) et les **attributs**, doit avoir de la javadoc. La mauvaise documentation pourrait entraîner une déduction considérable **(jusqu'à 30%).**

N'utilisez que les concepts vus en classe à date (donc pas d'ArrayList, LinkedList, LinkedHashSet... ou autre chose que vous pourriez trouver sur internet).

ANNEXE

Exemples de labyrinthes pour lesquels la méthode cheminJusquAuBoss() devrait retourner un tableau vide :