Série G : Structures de données

Table des matières

Exercice G.1: Labyrinth

Résumé: Définir les classes **Block, Brick, Grass, Water, Start** et **End** qui servent à représenter des blocs carrés par lesquels on peut composer un labyrinthe. La classe **BlockMap** sert à mémoriser et à gérer la composition du labyrinthe. Le polymorphisme joue un rôle fondamental dans la gestion des blocs.

Le programme **Labyrinth** utilise **BlockMap** pour réaliser un jeu de labyrinthe.

1. Définissez la classe abstraite **Block** est caractérisée par une seule méthode **draw** avec les paramètres suivants :
   - **g** indique le canevas sur lequel le bloc sera dessiné
   - **x,y** les coordonnées du bloc dans la grille ([0,0] étant la première case en haut à gauche)
2. Définissez les classes **Brick, Grass, Water, Start** et **End** qui sont dérivées de **Block**. Chacune de ces classes est caractérisée par une seule méthode **draw** qui surcharge la méthode **draw** héritée de **Block**.

Indications pour l'implémentation des méthodes **draw** des différentes classes :

- Chaque bloc est un carré de côté **size** points (p.ex. 40 points). **size** est un attribut unique et commun à tous les blocs. Dans une première version, **SIZE** peut être une constante, plus tard **size** peut être adapté en fonction de la surface disponible.
- Les blocs du type **Brick** sont gris avec des bords et des diagonales noirs.
- Les blocs du type **Grass** sont verts.
- Les blocs du type **Water** sont bleus.
- Les blocs du type **Start** sont verts et leurs bords sont noirs. Approximativement au milieu de ces blocs se trouve le texte 'START'.
- Les blocs du type **End** sont verts et leurs bords sont noirs. Approximativement au milieu de ces blocs se trouve le texte 'END'.

Remarque : La taille de la police peut être adaptée à l'aide de

```
g.setFont(g.getFont().deriveFont(<float>));
```

La classe **BlockMap** sert à gérer un labyrinthe composé de **dim x dim** blocs des types **Brick, Grass, Water, Start** et **End.** **BlockMap** possède les méthodes et attributs suivants :

- **dim** est la dimension maximale de la carte, c.-à-d. le nombre maximal de blocs qui peuvent se trouver dans une ligne ou une colonne. **dim** est défini lors de la construction d'un objet du type **BlockMap**.
- **map** est la carte du labyrinthe. Chaque élément de ce tableau peut contenir une référence à un bloc d'un type dérivé de **Block**. Au départ, tous les éléments doivent être initialisés à *null*. ***Conseil:*** Lors de la réalisation des points 7 et 8, il est utile d'employer l'opérateur **instanceof** pour déterminer les types des différents blocs référencés dans la carte **map**.
- Les coordonnées du pion joueur sur la carte sont mémorisées dans un attribut nommé **player**. Si le pion n'est pas encore placé, il est **null**.

Définissez l'implémentation des méthodes de **BlockMap** :

1. Définissez le constructeur qui prend comme paramètre la dimension de **map**.
2. Définissez la méthode **addBlock** qui sert à ajouter le bloc donné par **pBlock** à la position **[pI**,**pJ]** de la carte **map**. Un bloc doit uniquement être ajouté à une position de la carte qui n'a pas encore été affectée. Si la case donnée par **[pI**,**pJ]** est déjà occupée alors **addBlock** réagit par un bip sonore (p.ex. **beep**() de *java.awt.Toolkit*). La carte doit contenir un seul bloc **Start** au maximum.
3. Définissez la méthode **draw** qui dessine tous les blocs qui sont actuellement définis dans la carte **map** sur le canevas **g** passé comme paramètre. Les positions des blocs sont calculées en respectant les dimensions des blocs (**size x size** points). Enfin, si le joueur se trouve déjà dans le labyrinthe, il est dessiné au milieu du bloc référencé par **player**. Il est constitué par un cercle et un rectangle rouge superposés:

   Diamètre du cercle : **size**/2 points
   Longueur du rectangle : **size**/2 points
   Hauteur du rectangle : **size**/4 points

   
4. Définissez la méthode **resetPlayer** qui place le pion **player** sur un bloc du type **Start** qui se trouve dans le labyrinthe. Si le labyrinthe ne contient pas de bloc du type **Start**, le pion reste invisible.
5. Définissez la méthode **movePlayer** qui sert à déplacer le pion dans la direction indiquée par **pMove**. **pMove** est un nombre du type entier.Pour améliorer la lisibilité, définissez et employez 4 constantes pour les différents mouvements :

   ```
   MOVE_UP, MOVE_DOWN, MOVE_LEFT, MOVE_RIGHT
   ```

   Le pion peut uniquement se déplacer sur les blocs verts (**Grass**, **Start**, **End**). Evidemment, le pion ne doit pas sortir de la carte **map**. Si le mouvement défini par **pMove** ne peut pas être effectué, alors **movePlayer** réagit par un bip sonore (*Sound.beep*). Si le pion est déplacé sur un champ du type **End**, alors **movePlayer** retourne **true**, sinon **false**. [Ceci permettra d'afficher le message 'YOU WON!' dans le programme principal.]
6. Complétez le programme **Labyrinth** qui utilise un objet du type **BlockMap** pour dessiner un labyrinthe sur le canevas de la fiche **MainFrame**. Les boutons **resetPlayerButton**, **upButton**, **leftButton**, **downButton**, **rightButton** servent à placer et à déplacer le pion du joueur. Le labyrinthe est construit en cliquant sur l'espace libre de la fiche **MainFrame**. Le type du bloc à ajouter est sélectionné dans **fillBlockRadioGroup**. La position du bloc à ajouter est calculée en respectant les dimensions constantes des blocs (**size x size** points). Réaffichez le labyrinthe dès que la fiche a été cachée en partie ou en entier.
7. Ajoutez une possibilité pour sauvegarder et lire **BlockMap** dans un fichier (texte).
8. Ajouter une possibilité pour effacer des blocs (mais pas celui où le joueur se trouve actuellement).

La méthode **draw** est à appeler automatiquement lors de chaque modification de la carte **map** ou de la position du pion.

Pour avancés :

- Définissez une méthode récursive qui cherche et retourne un chemin de la position du pion joueur jusqu'à la sortie (bloc *END*).
- Développez ensuite une méthode qui marque ce chemin jusqu'à la sortie.
- Essayez de déterminer et d'afficher le chemin le plus court jusqu'à une sortie.

Exercice G.2: Reversi

Le jeu *Reversi* (aussi connu sous le nom de *Othello*) se joue avec des jetons blancs (du joueur 1) et des jetons noirs (du joueur 2) placés à tour de rôle sur une grille de 8x8 cases.

Le composant **Reversi** contient les propriétés privées suivantes:

grid	la grille principale de dimensions 8x8 cases (DIM=8) contenant les informations sur les jetons placés sur la table de jeu: 0: EMPTY - case vide 1: WHITE - jeton blanc (joueur 1) 2: BLACK - jeton noir (joueur 2)
cellWidth	la largeur et la hauteur d'une case en pixels. La largeur et la hauteur sont égales (une case est donc nécessairement carrée) et les cellules s'adaptent automatiquement à la largeur et la hauteur du composant pour en occuper le maximum d'espace possible.
player, **adversary**	*player* indique le numéro (1 ou 2) du joueur actuel (c.-à-d. le numéro du joueur qui doit actuellement placer un jeton) *adversary* indique le numéro (1 ou 2) de l'adversaire actuel. A chaque tour, les contenus de *player* et *adversary* sont échangés.

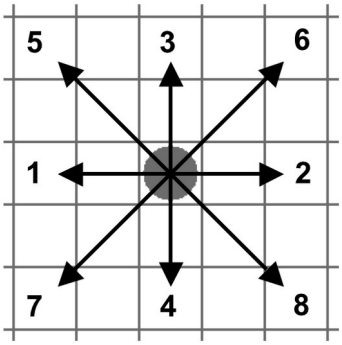
1) Réalisez la procédure **init** qui initialise la grille de la façon suivante: Toutes les cases sont à vider, 2 jetons noirs et deux jetons blancs sont à placer au milieu de la grille. Le joueur 1 joue le premier tour, le joueur 2 est l'adversaire. La grille et tous les jetons sont redessinés automatiquement (→ *repaint*).
   (positions des jetons: voir modèle ou la représentation sur la dernière page du questionnaire)
2) Réalisez le constructeur qui initialise le composant comme décrit ci-dessus.
3) Redéfinissez la méthode **draw** comme suit: *cellWidth* est à recalculer comme décrit plus haut. Laissez en bas 30 pixels pour afficher les inscriptions suivantes : Score des deux joueurs, joueur actif. Pour reconnaître plus facilement à qui est le tour de jouer, toutes les bordures des cases et des jetons sont dessinées dans la couleur du joueur actuel (joueur 1: blanc; joueur 2: noir).
   Dessinez toutes les cases et les jetons comme décrit dans ***grid***. Les jetons sont des cercles dont le diamètre est inférieure de 6 points (pixels) à *cellWidth*.
4) Ajoutez un composant **DrawPanel** avec un attribut du type ***Reversi*.** L'objet doit être redimensionnable en hauteur et en largeur. Le jeu sera dessiné sur ce panneau.
5) Ajoutez une fiche principale qui possède un panneau qui contient le bouton de réinitialisation ainsi q'un panneau **DrawPanel** sur lequel sera dessiné le jeu.

Règles pour le placement des jetons:

On peut ajouter un jeton seulement si par ce placement, on entoure un ou plusieurs jetons de l'adversaire. Les jetons entourés sont alors remplacés par des jetons de la couleur du joueur actuel.
Si par le placement d'un jeton, l'adversaire est entouré dans plusieurs directions à la fois, les jetons adversaires de tous ces nouveaux entourages sont remplacés.

Par **entourer** on entend que le nouveau jeton se trouve à une extrémité et un autre jeton de la même couleur se trouve à l'autre extrémité d'une série continue de jetons de l'adversaire. L'entourage peut se faire en horizontale, en verticale ou en diagonale. On a donc 8 possibilités pour entourer les jetons adversaires.

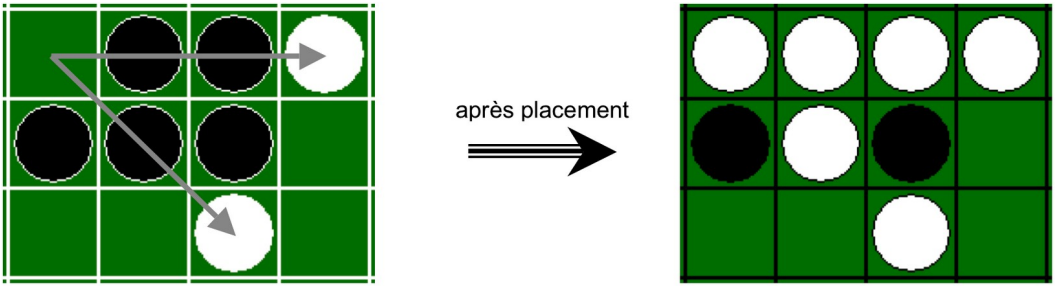
1) entourage horizontal vers la gauche
2) entourage horizontal vers la droite
3) entourage vertical vers le haut
4) entourage vertical vers le bas
5) entourage diagonal 1 (haut, gauche)
6) entourage diagonal 2 (haut, droite)
7) entourage diagonal 3 (bas, gauche)
8) entourage diagonal 4 (bas, droite)



Exemples:

- *Entourage horizontal vers la gauche*: En plaçant un jeton sur la cellule vide, le joueur 2 (noir) encercle les 2 jetons adversaires, parce que: (1) le nouveau jeton se trouvera à une extrémité d'une série continue de jetons adversaires et (2) à l'autre extrémité de la série se trouve déjà un jeton du joueur actuel.

- *Entourage horizontal vers la droite et entourage diagonal 4:*



5) Réalisez la méthode ***checkLeft*** qui retourne *true*, si le joueur actuel peut effectuer un entourage horizontal vers la gauche s'il place un jeton à la position de la grille définie par les paramètres ***column***, ***row***.
La fonction retourne *false*, si tel n'est pas le cas.

Testez bien la méthode *checkLeft*, puis réalisez de façon analogue les méthodes:
checkRight pour détecter la possibilité d'un entourage horizontal vers la droite
checkAbove pour détecter la possibilité d'un entourage vertical vers le haut
checkBelow pour détecter la possibilité d'un entourage vertical vers le bas

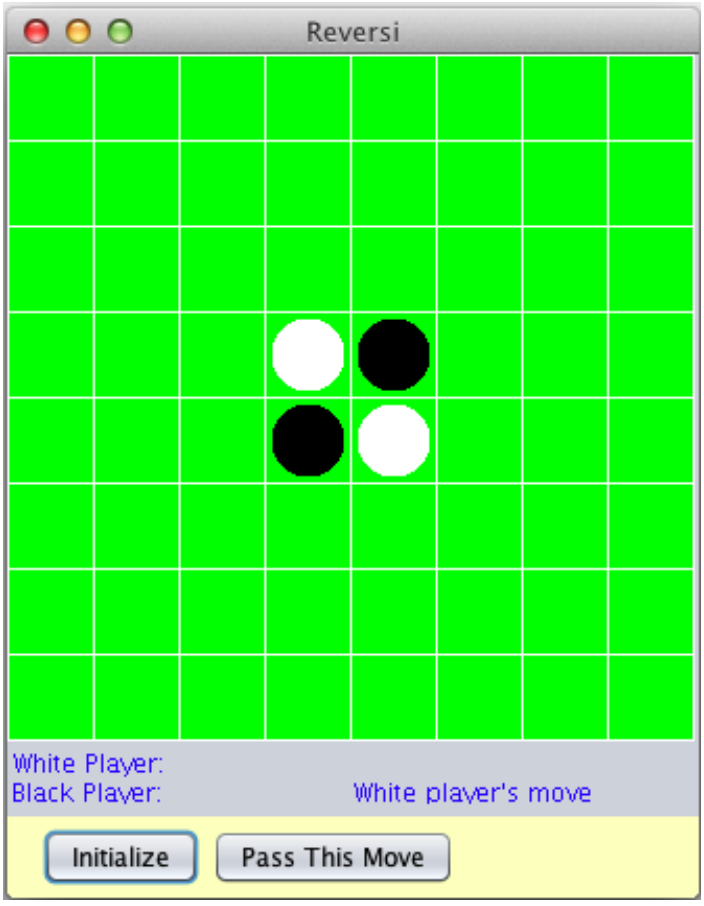
En vous basant sur la fonction *checkLeft*, réalisez de façon analogue la fonction:
checkDiag1 pour détecter la possibilité d'un entourage diagonal 1 (haut, gauche)
Testez bien la méthode *checkDiag1*, puis réalisez de façon analogue les méthodes:
checkDiag2 pour détecter la possibilité d'un entourage diagonal 2 (haut, droite)
checkDiag3 pour détecter la possibilité d'un entourage diagonal 3 (bas, gauche)
checkDiag4 pour détecter la possibilité d'un entourage diagonal 4 (bas, droite)

6) Réalisez la méthode ***validMove*** qui retourne *true*, si le joueur actuel peut placer un jeton dans la case définie par les paramètres ***column***, ***row***. C.-à-d.: la case doit être vide et en y plaçant le jeton, le joueur peut entourer un ou plusieurs jetons dans une ou plusieurs directions. La méthode retourne *false*, si tel n'est pas le cas.

7) Réalisez la méthode ***swapPieces*** qui place un nouveau jeton de la couleur actuelle dans la case définie par les paramètres ***column***, ***row***, puis remplace tous les adversaires encerclés par des jetons du joueur actuel.
Méthode: réalisez d'abord le traitement d'un seul entourage (p.ex. entourage horizontal vers la gauche) et testez cette partie. Ajoutez ensuite le traitement des autres entourages en vous basant sur la partie déjà réalisée avec succès.

8) Réalisez la méthode ***changePlayer*** qui change le rôle du joueur et de l'adversaire. Ensuite le composant est redessiné.

9) Réalisez la méthode ***placePiece*** qui détermine d'abord les coordonnées de la case qui correspond aux coordonnées X,Y de la souris au-dessus du composant. Les coordonnées X,Y de la souris sont reçues comme paramètre. Si le placement d'un jeton à la position de la souris est permise, le jeton y est placé, tous les jetons adversaires encerclés sont remplacés et le tour de jouer passe à l'adversaire. Enfin le composant est redessiné. Si le placement à la position de la souris est permis, la méthode *placePiece* retourne *true*, sinon elle retourne *false*.
Evidemment, la méthode *placePiece* fait appel aux sous-programmes correspondants décrits ci-dessus.



Exercice G.3: PlayGrid

Réalisez une classe **PlayGrid** qui sait représenter une surface de jeu carrée avec **n** x **n** cases. Chaque case peut contenir un pion (DE : *Spielstein*). Il existe différentes sortes de pions avec chaque fois une représentation différente, p.ex : différentes couleurs ou formes (croix bleues, carrés rouges, disques verts, etc.).

Intégrés dans un programme, on peut placer des pions sur les cases libres et déplacer les pions avec la souris.

La surface de jeu doit être redimensionnable, mais elle est sera toujours carrée.

Exemples d'application : Jeu de dames, TicTacToe, …

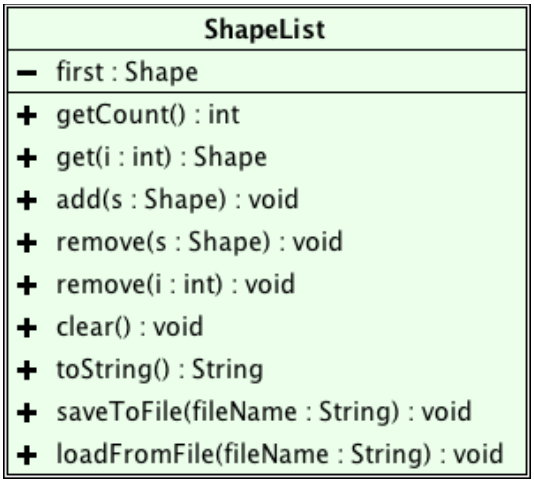
Exercice G.4: Linked List

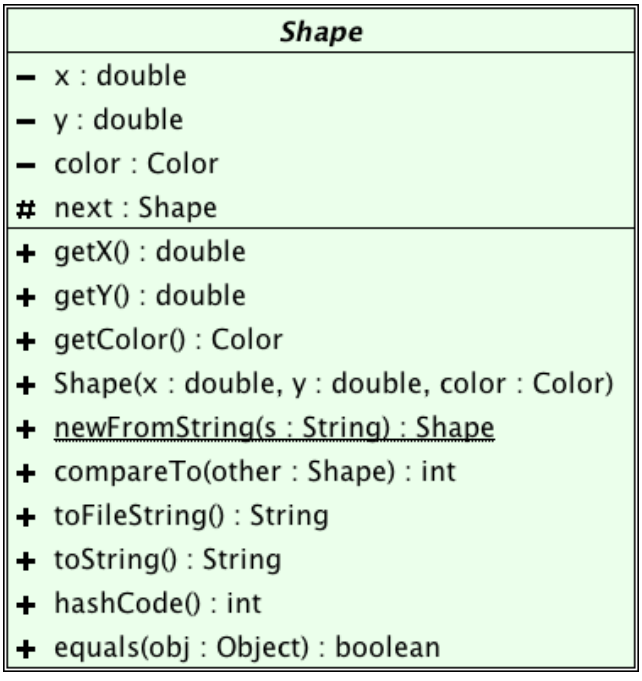
Reprendre l'exercice **'ShapeList'** et réalisez-le à l'aide d'une liste chaînée.

Le lien **next** se trouve dans **Shape** et la gestion se fait dans **ShapeList** à partir d'un attribut **first.** Réalisez les méthodes: **add**, **getCount, get**, **clear, toString, remove, saveToFile** (fichier texte), **loadFromFile** (fichier texte)**.**

Pour réaliser **remove(Shape s)**, il faut comparer les valeurs des attributs. Pour ce faire, il est utile de redéfinir **equals** & **hashcode** (en NetBeans, ceci peut se faire de façon automatique par *'Insert Code...'*)

La méthode statique **newFromString** a ici été définie comme 'réciproque' de **toFileString** : Elle permet de créer un nouvel objet à partir du texte fourni par **toFileString**. Elle est utilisée dans **loadFromFile**.



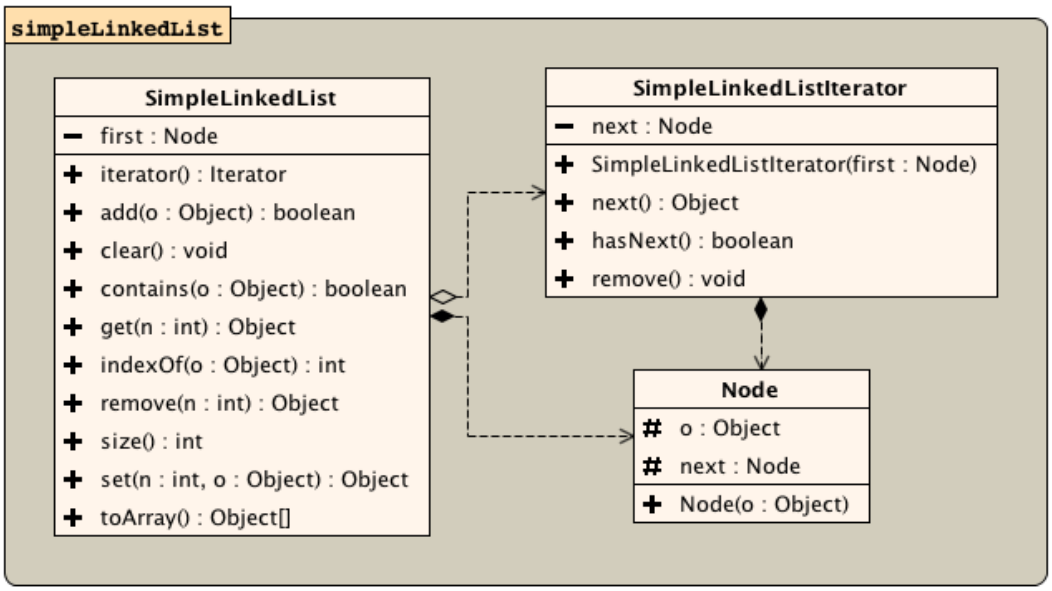


Exercice G.5: Linked List - MiniCollection

Dans la suite, vous allez programmer votre propre collection qui fonctionne selon le principe d'une **liste chaînée** (EN : **linked list**). La classe s'appellera **`SimpleLinkedList`** (pour la distinguer de la classe prédéfinie **`LinkedList`**).

Dans un nouveau paquet **`simpleLinkedList`**, réalisez les classes suivantes :

- Réalisez d'abord la classe **`Node`** avec un constructeur. Sous condition de placer les classes de gestion de la liste dans un paquet séparé, nous pouvons utiliser la visibilité **`protected`** dans **`Node`**. Ainsi, nous n'avons pas besoin de définir des accesseurs et des manipulateurs, ce qui allégera un peu le code.
- Réalisez ensuite la classe **`SimpleLinkedList`** avec les méthodes du schéma UML. Les méthodes réalisées fonctionnent de la même façon que celles des autres collections (p.ex. **`ArrayList`**). Cependant, nous n'allons pas implémenter l'interface intégral **`Collection`**, mais seulement un petit sous-ensemble de ses méthodes. (→ voir **JavaDoc** pour les détails).
- Réalisez enfin un itérateur **`SimpleLinkedListIterator`** pour la classe **`SimpleLinkedList`**. Il n'est pas nécessaire de réaliser la méthode optionnelle **`remove`** (un appel de la méthode **`remove`** lance une exception **`UnsupportedOperationException`**).

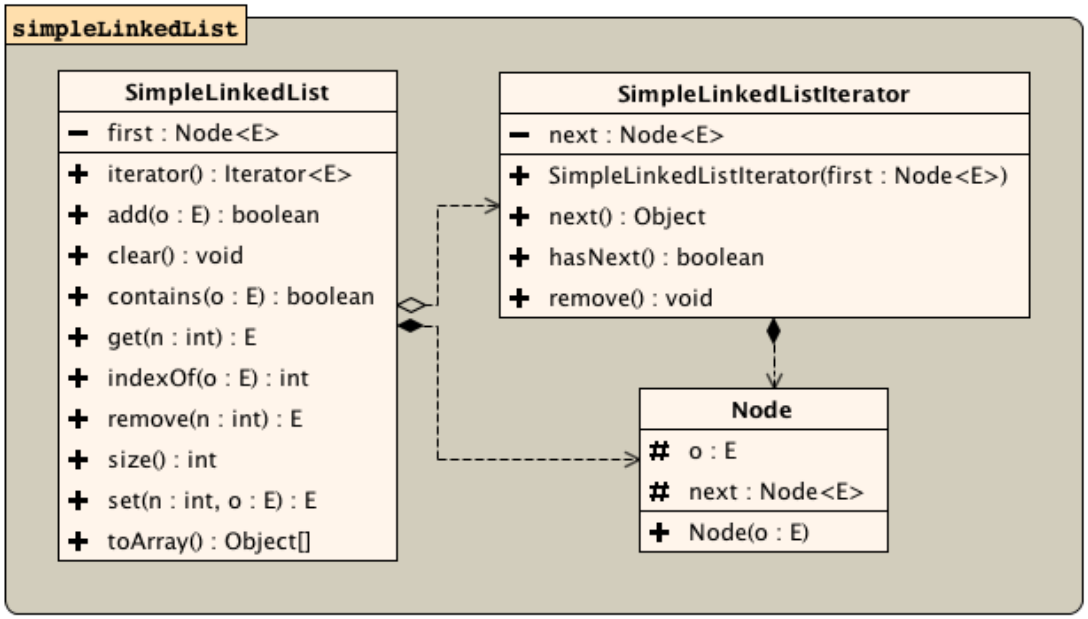


Pour avancés :

Copiez le projet et réalisez la classe **SimpleLinkedList** cette fois en utilisant la récursion au lieu de boucles. Relisez d'abord la dernière remarque dans le chapitre du cours traitant les listes chaînées.

Exercice G.6: Linked List - MiniCollection & Generics

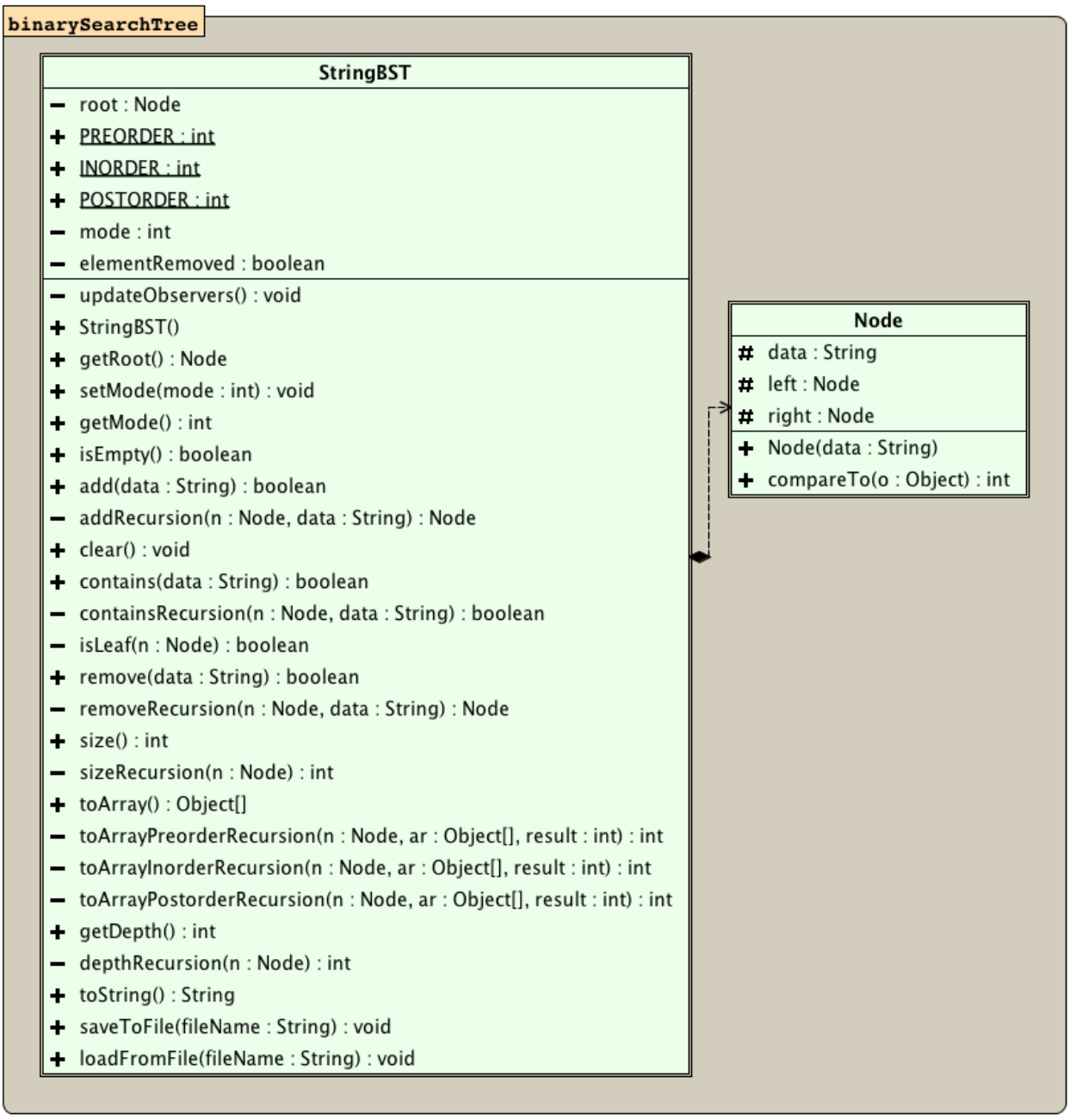
- Comme pour les autres collections, il est possible de spécifier un type pour les éléments de notre collection en l'indiquant comme paramètre lors de la définition (**types paramétrés** EN:**generics** →http://en.wikipedia.org/wiki/Generics_in_Java).
  P.ex.: `private SimpleLinkedList<Person> shapes = new SimpleLinkedList<Person>();`
- Reprenez l'exercice **ShapeList2** (avec la gestion dans une **ArrayList** et la sauvegarde sous forme de texte). Après avoir intégré le paquet **simpleLinkedList** dans le projet, remplacez **ArrayList** par votre **SimpleLinkedList.** En principe le projet devrait fonctionner sans d'autres modifications avec votre nouvelle liste chaînée.



Exercice G.7: StringBST - Arbre binaire de recherche pour textes

Réalisez un ABR nommé **`StringBST`** pour gérer des textes. L'arbre possède les opérations de base (ajout, recherche, suppression, etc.), ainsi qu'une méthode **toArray** qui retourne tous les éléments dans un tableau. Pour pouvoir influencer l'ordre de parcours de l'arbre, la classe possède un attribut **mode** qui définit le mode de parcours (**mode** peut avoir l'une des valeurs constantes PREORDER, INORDER, POSTORDER). Les méthodes **toString**() et **saveToFile**(...) dépendent de **toArray** et par conséquence du mode de parcours .

Les méthodes avec le suffixe '...**Recursion**' sont nécessaires pour réaliser les appels récursifs. Les méthodes **loadFromFile** et **saveToFile** sauvegardent les données dans un fichier (texte) de façon à ce que la structure précédente de l'arbre soit rétablie. (Si vous sauvegardiez les données 'en ordre' l'arbre serait rétabli dans une longue liste d'éléments (noeuds de gauche vides) et tous les avantages de l'arbre seraient perdus.)



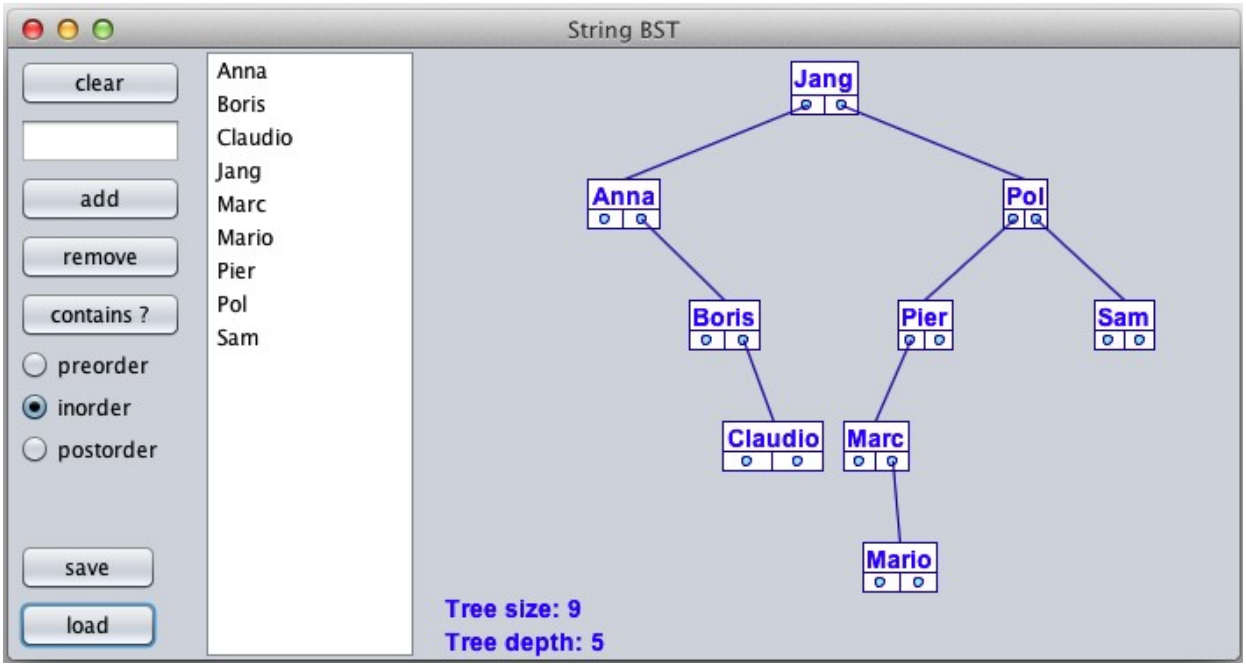
Exercice G.8: StringBST, MVC et PropertyChangeListener/Support

Recherchez d'abord des informations sur le modèle de conception (EN: *design pattern*) **Observer** (aussi connu sous le nom *"Event-Subscriber"*) et l'interface **PropertyChangeListener** (voir aussi Annexe I. du cours).

A partir d'un **JPanel** développez la classe **StringBSTView** qui sert à représenter l'arbre de façon graphique. La vue doit être informée des changements dans l'arbre. Elle est alors un 'observer' (observateur, abonné) du publicateur **StringBST.**

Intégrez l'affichage dans le contrôleur (**MainFrame**) en essayant de donner accès à autant de fonctionnalités de **StringBST** que possible. Affichez les éléments de **StringBST** dans une liste. **MainFrame** est alors aussi abonné aux notifications de **StringBST**.

Profitez du modèle **Observer** et essayez d'éviter au maximum les appels explicites de **repaint**().

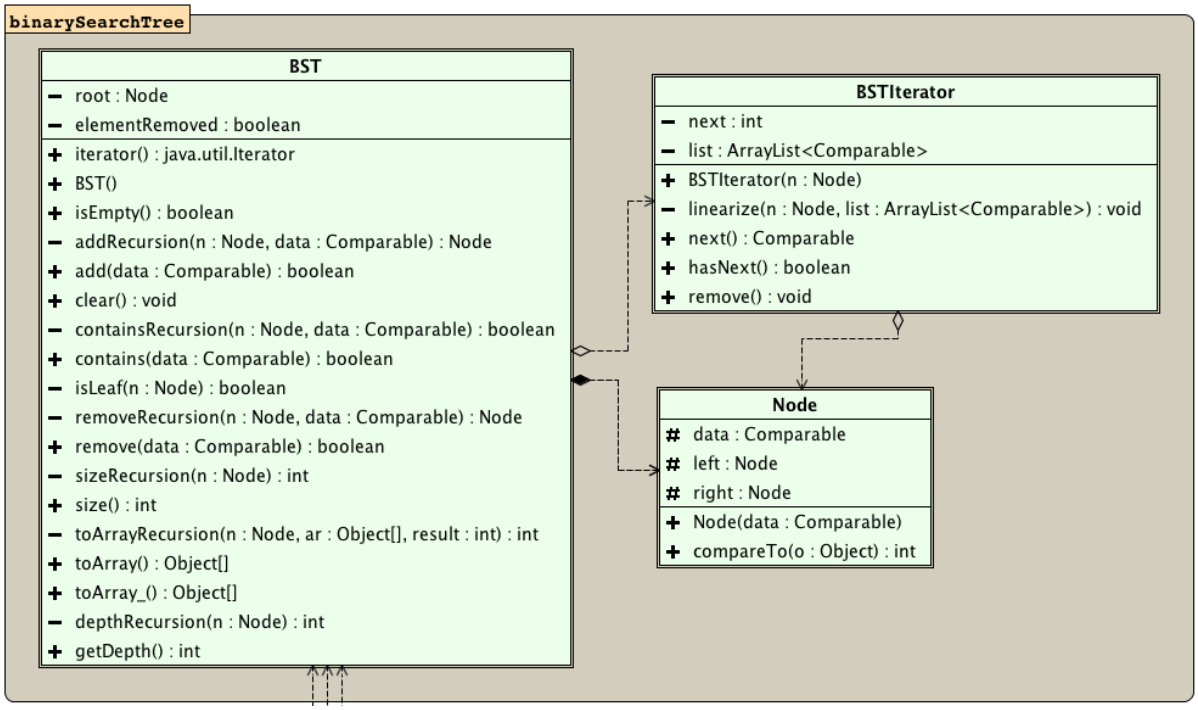


Exemple d'application :

Ajoutez à StringBST une méthode '*runWild*' qui ajoute à des intervalles irréguliers des textes aléatoires à l'ABR. (p.ex. avec un Timer)

Exercice G.9: Arbres binaires de recherche - Binary Search Trees

- Développez un arbre binaire de recherche (ABR) pour une classe de votre choix. La seule condition est que la classe implémente l'interface **Comparable**. Pour cette raison, aussi les objets (**data**) transportés par les noeuds doivent implémenter **`Comparable`**. Réalisez l'ABR d'abord sans classes génériques, d'après le schéma UML ci-dessous.



Remarques :

Lors de la réalisation de ces méthodes, il faut programmer quelques méthodes (privées) supplémentaires pour les opérations qui nécessitent une solution récursive (comme avant, nous utilisons le suffixe *'...Recursion'* pour marquer ces méthodes).

Détails de quelques méthodes :

- **`getDepth()`** retourne la profondeur de l'arbre.
- **`size()`** retourne le nombre de noeuds de l'arbre (0 si vide)
- **`toArray()`** les éléments de l'arbre dans un tableau (trié selon l'ordre interne de l'arbre, c.-à-d. par ordre croissant selon ***`compareTo`***)

- Comme il est impossible de passer des paramètres par référence, la racine de l'arbre ne peut pas être changée directement par une méthode. Ainsi, les méthodes qui changent l'arbre obtiennent un noeud comme paramètre et retournent le noeud changé comme résultat. Lors de l'appel d'une telle méthode, il faut affecter le résultat de la méthode au noeud utilisé dans le paramètre.
- Un arbre **vide** est un arbre avec une racine vide (**`null`**).

Exercice G.10: BST générique

Développez les classes **Node<E>** et **BST<E>** qui réalisent un arbre binaire de recherche. Implémentez pour **BST** les mêmes méthodes suivantes :

- un constructeur
- **add(E e)** ajoute l'objet/élément **e** à l'arbre (en gardant l'ordre interne)
- **isEmpty()** retourne **true** si l'arbre est vide (voir remarques ci-dessous)
- **clear()** efface l'arbre (voir remarques ci-dessous)
- **contains(E e)** retourne **true** si l'élément fourni se trouve dans l'arbre
- **getDepth()** retourne la profondeur de l'arbre.
- **size()** retourne le nombre de noeuds de l'arbre (0 si vide)
- **toArray()** les éléments de l'arbre dans un tableau (triés)
- **remove(E e)** supprime l'élément de l'arbre et retourne **true** si réussi
- Pour assurer que les éléments **E** implémentent l'interface **Comparable** il faut définir **BST** comme :

  BST<E *extends* Comparable<E>>

  (Il est impossible de définir **<E implements Comparable<E>>**, mais **extends** peut être appliqué sur des interfaces lors de la définition de génériques.)

  La même remarque s'applique à **Node** si on veut que deux **Node**s soient comparables directement entre eux.

Exercice G.11: BST générique & fichiers

Ajoutez des méthodes pour sauvegarder et lire des données du BST dans un fichier.

Quelle alternative se présente pour la sauvegarde ? Réalisez cette alternative.

Exercice G.12: BST générique & String

Refaites l'exercice BST-String en utilisant le BST générique comme base.

Ajoutez d'abord à BST les options et opérations nécessaires (mode, PropertyChange..., ...).