TME 1 : Programmation, compilation et exécution en Java

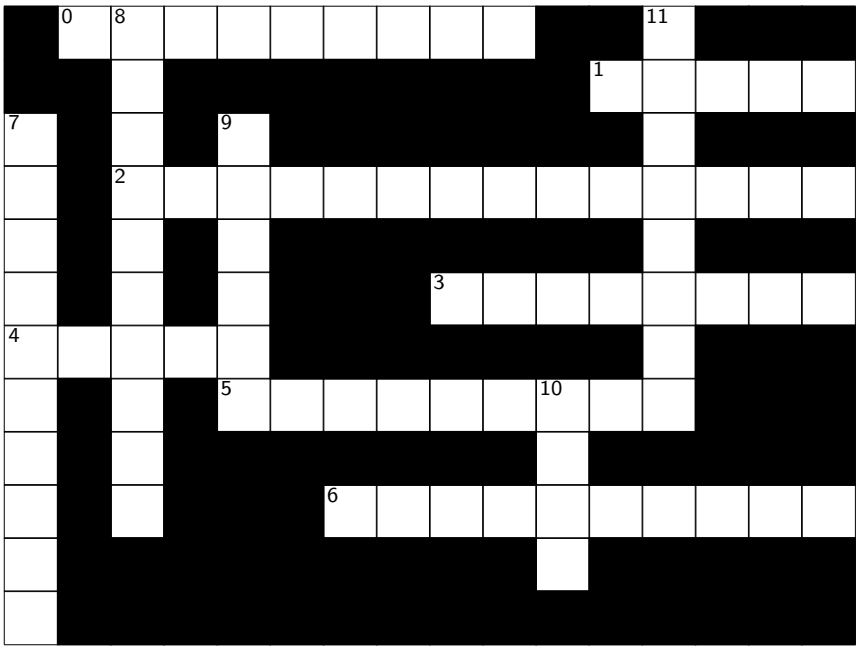
Objectifs pédagogiques :

- **classes**
- **instances**
- **tableaux**
- **itérations**

1.1 Préambule : mots croisés

Au cours des TME 1 à 3 nous allons bâtir une application permettant de construire des mots croisés. L'objectif global est un programme qui prend en entrées un dictionnaire et une grille vierge, et qui rend en sortie une grille résolue.

Par exemple, une grille vide possible est représentée ci-dessous. Elle comporte 12 emplacements de mot, les premières cases de chaque emplacement de mot y sont numérotées de 0 à 11. Nous considérons que les mots croisés sont donnés comme une grille dont chaque case est soit pleine (noire) soit vide (blanche).



Le but est d'inscrire une lettre dans chaque case vide, en s'assurant que tous les mots horizontaux et verticaux sont des mots du dictionnaire. Par exemple, la grille ci-dessous est solution des mots croisés donnés ci-dessus. Normalement, un indice est donné pour chaque emplacement de mot, ce qui aide le cruciverbiste (un humain) à deviner les mots à placer. Ici, c'est l'inverse : nous souhaitons générer des problèmes de mots croisés à l'aide de notre outil.

1.2 Plan des séances

La construction d'une solution à ce problème va nous occuper sur trois séances.

Au TME 1, nous allons construire la structure de données permettant de représenter une grille de mots-croisés en mémoire puis charger une grille (*a priori* vide) depuis un fichier.

Au TME 2, nous allons rechercher les emplacements des mots dans la grille et calculer leur longueur. Nous définirons ensuite, pour chaque emplacement de mot, un *dictionnaire* qui maintient l'ensemble des mots possibles à cet emplacement. Au départ ce sont tous les mots de la bonne longueur.

Au TME3, nous raffinons ces dictionnaires en examinant les intersections entre deux emplacements de mots : les lettres des mots choisis doivent coïncider. Nous nous donnerons le moyen de tester si une grille partiellement remplie est *vraisemblable*, c'est-à-dire si les mots complets sont dans le dictionnaire et s'il existe encore des candidats pour tous les mots incomplets. Enfin, nous réaliserons un algorithme de *satisfaction de contraintes*. Cet algorithme sera formulé de façon abstraite (grâce à l'utilisation d'interfaces) et il faudra adapter le code précédent aux besoins de l'algorithme.

L'outil final permettra de résoudre de grandes grilles en peu de temps. En extension (bonus optionnels), nous ajouterons des heuristiques pour améliorer les performances.

Cette UE de programmation Java avancée suppose une familiarité avec Java. Les premières séances sont l'occasion de se remettre à niveau. L'énoncé contient donc plusieurs encadrés, en gris, qui rappellent les notions clés à appliquer. Prenez le temps de bien absorber les concepts si nécessaire. Si vous êtes à l'aise, vous pouvez aborder les TME suivants et prendre de l'avance pour pouvoir réaliser certaines des extensions proposées à la fin du TME 3.

Vous soumettrez l'état de votre TME à la fin de chaque séance. Les instructions sont données à la fin de cet énoncé. Ce rendu hebdomadaire de TME contribue à votre note d'évaluation continue.

Le caractère incrémental des TME impose une contrainte : en général, il sera nécessaire que vous ayez fini le TME précédent pour pouvoir attaquer le TME suivant. Pour vous aider à atteindre cet objectif et construire sur une base saine, exécutez les tests fournis.

1.3 Mise en place du TME 1

Vous devez suivre les instructions de mise en place communes à tous les TME, décrites dans le document *Mise en place et rendu des TME* (TME 0) sur la page Moodle du cours.

Dans le TME 1, nous utilisons le projet `MotCroise`. Vous devez donc commencer par faire un *fork* (une copie) de ce projet sur le serveur GitLab du cours `https://stl.algo-prog.info` pour obtenir un projet personnel (dans votre espace de nom) autorisé en écriture et partagé avec votre binôme, puis importer ce projet personnel dans Eclipse. Un nouveau projet `MotCroise` apparaît dans le « Package Explorer » à gauche dans Eclipse. La mention « `[motcroise main]` » associée au projet indique que le projet est bien géré par `git`.

Nous pouvons à présent créer des classes Java dans Eclipse, en sélectionnant un nom de package sous `src` dans le « Package Explorer » à gauche, puis faire « clic droit > New > Class ». Toutes les classes développées dans cette séance seront placées dans le package `pobj.motx.tme1`.

Notez que le projet contient déjà quelques fichiers fournis. Les fichiers utiles pour ce TME sont :

- Dans le package `pobj.motx.tme1.test`, des classes de test. Elles ne compilent pas, puisque les classes qu'elles testent n'ont pas encore été écrites (Eclipse les marque d'une croix rouge).
- Une classe `pobj.motx.tme1.GrilleLoader` servant à charger des grilles de mots croisés depuis un fichier. Celle-ci ne compile pas encore.
- Un dictionnaire (`data/frgut.txt`) de mots français, sans accents.
- Plusieurs grilles (`data/*.grl`) vides ou partiellement remplies dans un format textuel (pratique pour les tests).

NB : Nous vous conseillons de garder pendant votre TME plusieurs fenêtres ouvertes : l'IDE Eclipse, le sujet de TME, la page du serveur GitLab et la page de documentation de l'API Java.

Pensez à consulter, avant la fin de la séance, la section 1.8 sur le rendu de TME en fin de document : un rendu initial est demandé en fin de séance même si vous n'avez pas fini toutes les questions (ce rendu pourra être complété après la séance).

1.4 Classe `Case`

Nous souhaitons définir les classes `Case` et `Grille`, dans le package `pobj.motx.tme1`, représentant une grille de mots croisés.

Chaque `Case` a en attribut des coordonnées (un couple d'entiers désignant la ligne et la colonne de la case), et un caractère qui représente la valeur dans la case.

Nous identifions les cases pleines par le caractère '*', et les cases vides par le caractère espace ' '. Les autres cases peuvent contenir un caractère parmi les 26 lettres de l'alphabet, en minuscule.

⇒ **Donnez l'implantation de la classe `Case`**, elle comportera :

- un constructeur `public Case(int lig, int col, char c)` qui initialise les attributs aux valeurs données,
- des accesseurs `public` en lecture pour chacun des attributs `int getLig()`, `int getCol()` et `char getChar()`,
- un accesseur en écriture `public void setChar(char c)` pour modifier le contenu de la case,
- une opération booléenne `public boolean isVide()` qui répond vrai si la case est vide (blanche),
- une opération booléenne `public boolean isPleine()` qui répond vrai si la case est pleine (noire). Notez qu'une case contenant une lettre n'est ni vide (blanche) ni pleine (noire).

⇒ **Exécutez le test `pobj.motx.tme1.test.CaseTest`** présent dans le projet (lire l'encadré ci-dessous sur JUnit 4). Assurez-vous d'abord que le code du test compile (corrigez éventuellement le nom des méthodes, attributs, packages de votre classe `Case`), puis lancez-le.

JUnit 4.
JUnit est un outil permettant d'écrire et d'exécuter des tests unitaires en Java.
Pour vous en servir sous Eclipse, assurez-vous d'abord que JUnit est activé pour le projet. Pour cela, sélectionnez le projet, puis « clic droit > Build Path > Add Library > JUnit 4 ».
Pour exécuter un test, sélectionnez le fichier contenant le test et faire « clic droit > Run As... > JUnit Test Case ». Vous pouvez aussi ouvrir le fichier puis cliquer sur le bouton vert « Run » dans la barre d'outils en haut. Une fenêtre affichera alors les résultats des tests : en vert les tests ayant réussi, et en rouge les tests ayant échoué.
Pour créer de nouveaux tests pour une classe, sélectionnez la classe puis « clic droit > New > JUnit Test Case > Next > cochez les opérations voulues > Finish ».
Les tests JUnit sont du code Java ordinaire, cependant il est aussi possible de spécifier des résultats attendus à l'aide d'une famille de méthodes `assert`. Par exemple, `assertTrue(boolean b)` fait échouer un test si la condition b n'est pas vraie, `assertEquals(int expected, int actual)` fait échouer un test si `expected` et `actual` ont des valeurs différentes, etc.

1.5 Classe `Grille`

La grille est constituée d'une matrice de cases de *hauteur* lignes sur *largeur* colonnes.

Tableaux et matrices d'objets en Java.
Soit `Contenu` une classe arbitraire. Un tableau t de `Contenu` se déclare comme : `Contenu[] t`. Une matrice m de `Contenu` (tableau à deux dimensions) se déclare comme : `Contenu[][] m`.
La taille d'initialisation n'est donc pas fixée à la déclaration, mais à l'allocation.
Soit max une taille (un `int`), pour allouer un tableau de max cases on utilise `t = new Contenu[max]`. On accède aux cases par `t[i]` où i est un entier compris entre 0 et $max - 1$.
Soient h et w deux tailles, pour allouer une matrice de taille (h, w) cases on utilise `m = new Contenu[h][w]`. On accède aux cases par `m[i][j]` où i et j sont des indices. `m[i]` est alors un tableau de taille w.
Les tableaux Java stockent leur taille d'allocation. Il est possible d'y accéder comme si c'était un attribut du tableau, avec la syntaxe **`t.length`**. Inutile donc de stocker séparément la taille d'allocation dans un entier, et de passer aux méthodes opérant sur un tableau sa taille en plus du tableau lui-même, (comme en C par exemple).
Dans tous les cas, l'allocation initialise toutes les cases `t[i]` ou `m[i][j]` à `null`. Un constructeur devra donc typiquement itérer sur les cases pour les initialiser avec de nouveaux contenus : `t[i] = new Contenu(...)`.

⇒ **Donnez l'implantation de la classe `Grille`**, elle comportera :

- un attribut représentant une matrice `Case[][]`,
- un constructeur `public Grille(int hauteur, int largeur)` qui alloue puis initialise chaque cellule de la matrice avec une *nouvelle* `Case` vide aux coordonnées cohérentes avec sa position,
- un accesseur `public Case getCase(int lig, int col)` qui retourne la case à la position (lig, col) dans la matrice. Prenez garde à ce que les coordonnées de la case soient cohérentes, c'est-à-dire que pour tout couple (l, c) : `getCase(l,c).getLig() == l` et `getCase(l,c). getCol() == c`, sans inverser les lignes et les colonnes,
- une méthode standard `public String toString()` qui permet d'afficher une grille sur la console de façon « lisible » (nous allons utiliser la méthode `serialize` de la classe `GrilleLoader` fournie pour construire cet affichage, voir ci-dessous),
- deux accesseurs `public int nbLig()`, `public int nbCol()` retournant respectivement le nombre de lignes et de colonnes de la grille,
- une méthode `public Grille copy()` qui retourne une copie à l'identique de la grille courante. Attention, les deux grilles ne doivent pas partager de référence sur une même instance de `Case` (il faut les copier).

Pour tester le comportement, nous vous fournissons une classe `pobj.motx.tme1.GrilleLoader` permettant de charger et de sauver des grilles au format « `.grl` ». Assurez-vous qu'elle compile correctement, en corrigeant la classe `Grille` si nécessaire.

Utilisez la méthode `static String serialize(Grille g, boolean isGRL)` de `GrilleLoader` pour définir `toString()` dans `Grille`. Vous passerez `false` pour le paramètre *isGRL* pour utiliser un format de sortie plus lisible que le format GRL, destiné au stockage dans un fichier.

⇒ **Exécutez le test JUnit `pobj.motx.tme1.test.GrilleTest` fourni.** Affichez des grilles sur la console et assurez-vous que l'aspect est cohérent et lisible.

`toString()`.
La méthode `public String toString()` est définie sur la classe `Object`, donc toute classe Java en est munie, et retourne une représentation de l'objet sous forme de chaîne de caractères. Cependant, l'implantation par défaut retourne un texte contenant la classe de l'objet et un *hash* unique à cet objet, que nous pouvons apparenter à son adresse mémoire. Il est conseillé de redéfinir `toString()` pour vos propres classes, afin d'améliorer leurs affichages.
Dans certains cas, si le compilateur détecte que le contexte nécessite un `String` mais qu'il a à disposition un objet non-chaîne, il invoquera `toString()` de façon implicite. Par exemple, le code `Grille g = ...; String s = "La grille est :  " + g;` invoquera implicitement `g.toString()`, car le contexte du `+` nécessite deux `String`.
Si nous avons besoin de construire la `String` à retourner petit à petit, il est fortement conseillé d'utiliser un `StringBuilder`, dans lequel du contenu est ajouté à l'aide de `append`. Ci-dessous, un bloc de code donne la structure générale d'une méthode `toString()`, voir aussi la documentation en ligne de `StringBuilder`.

```java
@Override
public String toString() {
   StringBuilder sb = new StringBuilder();
   // pour chaque attribut "attXX" de "this"
   sb.append(" attribut1 = " + this.att1 );
   sb.append(" attribut2 = " + this.att2 );
   // etc.
   return sb.toString();
}
```

1.6 Classe `Emplacement`

Nous souhaitons à présent identifier les emplacements des mots dans la grille, c'est-à-dire les séquences contigües d'au moins deux cases qui ne sont pas pleines. Les emplacements peuvent être horizontaux (lus de gauche à droite) ou verticaux (lus de haut en bas). Un emplacement est simplement représenté par une liste de cases.

⇒ **Donnez l'implantation de la classe `Emplacement`**, elle comportera :

- un attribut qui représente les cases (contiguës) de la grille qui composent l'emplacement du mot, nous pouvons utiliser un attribut `private List<Case> cases` par exemple,
- un constructeur `public Emplacement()` sans argument qui crée un emplacement initialement vide,
- une méthode `public void add(Case e)` qui ajoute une case à l'emplacement (c'est-à-dire, à la fin de la liste des cases),
- une méthode `public int size()` qui retourne la taille (nombre de cases) de l'emplacement,
- une méthode `public Case getCase(int i)` qui permet de retrouver la case à l'indice `i`,

- une méthode `public String toString()` qui permet d'afficher le mot à cet emplacement (juste les caractères qui le constituent).

⇒ **Exécutez le test JUnit `pobj.motx.tme1.test.EmplacementTest` fourni.**

`List<T>` EN JAVA. Consultez la documentation en ligne (documentation de l'API) régulièrement. L'ensemble des bibliothèques standards de Java (Java API) sont commentées en `Javadoc`. Sous Eclipse, survolez un nom de classe ou une opération pour voir sa documentation, ou ouvrez une fenêtre `javadoc` avec F1.
Commencez par regarder les méthodes de `java.util.List`[a] par exemple.
Une `java.util.List<T> l` représente un ensemble de `l.size()` objets de type `T`. Les objets sont triés dans un certain ordre, c'est-à-dire que nous pouvons accéder aux éléments par leur index.

- `T get(int i)` retourne l'objet d'indice i.
- `T set(int i, T val)` assigne *val* à l'objet d'indice i, et retourne la valeur qu'avait précédemment cette cellule.
- `boolean add(T elt)` ajoute *elt* en dernière position dans la liste.
- `boolean add(int i, T elt)` ajoute *elt* en position i et décale les cellules d'indice supérieur ou égal à i vers la droite d'une case.

Une `List<T>` est `Iterable<T>`, ce qui signifie qu'il est possible d'utiliser une boucle « foreach » pour explorer ses éléments. Soit `List<T> list` une liste, il faut écrire `for (T elt : list) { /* utiliser elt */ }`.
`java.util.List` est une *interface*, donc abstraite. Nous typerons de préférence les attributs, variables et paramètres des opérations par `List<T>`, plutôt que de mentionner une classe concrète comme `ArrayList`. En effet, en dehors du `new`, il est le plus souvent inutile de savoir quel conteneur concret est utilisé.
`java.util.List` est implémentée (entre autres) par `ArrayList` (stockage contigu dans un tableau), `LinkedList` (liste doublement chaînée), etc. (cf. documentation en ligne).
À l'initialisation, nous utiliserons le plus souvent l'implantation `ArrayList`, qui stocke un tableau nu en sous-jacent, mais gère pour nous les problèmes de réallocation en cas de débordement, etc. Il est donc rare de manipuler des tableaux nus en Java (sauf si la taille est fixée à la construction), par défaut ayez le réflexe de définir des `List<T>`, plutôt que des `T[]`.
Nous écrivons donc `List<Contenu> l = new ArrayList<>();`.
Voici un extrait de programme illustrant les principales méthodes disponibles sur `List`.

[a] Il y aura une séance de cours dédiée aux Collections plus tard dans l'UE. Nous expliquons ici les manipulations de base.

```java
List<String> tab = new ArrayList<>();  // contient des chaînes
System.out.println(tab.size());  // affiche 0 : la taille est vide
tab.add("hello");  // ajoute la chaîne "hello" (position 0)
System.out.println(tab.size());  // affiche 1 : un élément
String s = tab.get(0);  // récupère le premier élément
System.out.println(tab);  // affiche [hello]
tab.add("world");  // ajouté en position 1
System.out.println(tab.get(0));  // affiche le premier élément
for (String elt : tab) {  // parcourt tous les éléments
  System.out.println("Element : " + elt);  // affiche l'élement courant elt
}
tab.remove(0);  // retire le premier élément
System.out.println(tab.size());  // taille 1 ("world" en position 0)
System.out.println(tab);  // affiche [world]
tab.clear();  // retire tous les éléments
System.out.println(tab.size());  // taille 0
```

1.7 Documentation du code

Dans tout projet de taille conséquente, il ne faut pas écrire du code pour soi mais pour les autres programmeurs participant au projet, ou pour les futurs utilisateurs de l'application. La documentation du code occupe donc une place fondamentale. L'environnement de développement Java met à disposition l'outil `javadoc` permettant de générer des documentations hypertextes à partir de commentaires spéciaux placés dans le code source des classes.

Les commentaires spéciaux commencent par `/**` et se terminent par `*/`. Sous Eclipse, placez-vous au dessus d'une déclaration de classe, tapez `/**` puis « entrée » pour obtenir un cadre à remplir. Vous pouvez aussi sélectionner un élément à commenter et faire « Menu Source > Generate Element Comment » ou « Shift-Alt-J ».

Ces commentaires se placent juste au-dessus de ce que nous souhaitons documenter, en priorité les classes elles-mêmes, les constructeurs et méthodes publiques (cependant, une bonne pratique consiste à tout documenter). Des balises spéciales comme `@param` ou `@return` peuvent être utilisées pour standardiser la mise en page. Voici par exemple le code source d'une classe `Point` totalement documentée :

pobj.axiom5.noyau.Point.java

```
package pobj.axiom5.noyau;
/**
* Classe de représentation de Point dans un repère cartésien
*/
public class Point {
  /** abscisse du point */
  private double x;
  /** ordonnée du point */
  private double y;

  /**
  * Construit un point de coordonnées initiales spécifiées
  * @param x l'abscisse initiale du point
  * @param y l'ordonnée initiale du point
  */
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  /**
  * Accède a l'abscisse de ce point
  * @return l'abscisse x du point
  */
  public double getX() {
    return x;
  }
  /**
  * Accède à l'ordonnée de ce point
  * @return l'ordonnée y du point
  */
  public double getY() {
    return y;
  }
  /**
  * Effectue une translation du vecteur de translation spécifié
  * @param dx translation sur l'axe des abscisses
  * @param dy translation sur l'axe des ordonnées
  */
  public void translater(int dx, int dy) {
    x+=dx;
```

```
    y+=dy;
  }
}
```

⇒ **En suivant ce modèle, documentez vos classes.**

Pour générer la documentation depuis Eclipse, faites « sélection du dossier `src` > clic droit > Export > Java > Javadoc > Finish ». Elle sera produite sous la forme de pages HTML dans le répertoire `doc/` de votre projet (localisation par défaut).

1.8 Rendu de TME (OBLIGATOIRE À LA FIN DE LA SÉANCE)

Le TME que vous venez de réaliser va resservir par la suite. Il est donc impératif de le terminer avant la séance suivante.

N'oubliez pas de faire un premier rendu de TME **obligatoirement en fin de séance** (rendu initial), et éventuellement un **deuxième rendu avant la prochaine séance** si vous n'avez pas fini ce TME lors de cette séance (rendu final).

Les instructions générales de rendu sont détaillées dans le document *Mise en place et rendu des TME* sur la page Moodle du cours. Nous les résumons succinctement :

- propagez vos dernières modifications locales vers le serveur GitLab sur le projet `MotCroise` privé à votre binôme et à vous (*push*) ;
- assurez-vous que les fichiers visibles dans l'interface web GitLab correspondent bien à la dernière version de vos fichiers ;
- assurez-vous que tous les tests unitaires du TME 1 passent correctement sur GitLab dans la page d'intégration continue « Build > Pipelines » ;
- assurez-vous que votre chargé de TME est membre de votre projet, avec le rôle « Maintainer » ;
- créez une *release* avec pour nom de *tag* « rendu-initial-tme1 » ou « rendu-final-tme1 » et répondez aux questions ci-dessous dans le champ « Release notes ». Créer la *release* est important car cela permet de distinguer votre rendu d'un *commit* intermédiaire et indique au chargé de TME quelle version de votre code noter.

⇒ **Répondez à ces questions dans les « Release notes » votre rendu.**

- Pourquoi dans la méthode `copy` de la grille est-il nécessaire de créer des copies des `Case` qui constituent la grille ? Donnez votre réponse dans le champ « Release notes ».
- Copiez (ou attachez) dans le champ la trace d'exécution de la suite de tests unitaires : `pobj.motx.tme1.test.TME1Tests`.