TME 1 : Programmation, compilation et exécution en Java

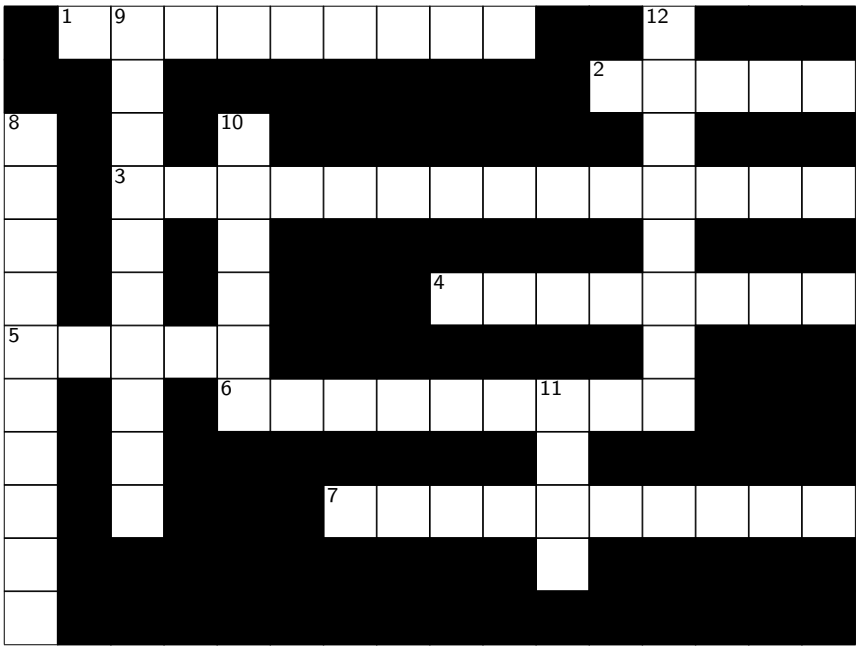
Objectifs pédagogiques :

- **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 1 à 12. 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 du mot croisé donné 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 accompagner sur trois séances.

Au TME 1, nous allons commencer par charger une grille (*a priori* vide) depuis un fichier. Nous allons ensuite rechercher les emplacements des mots dans la grille, et calculer leur longueur.

Au TME 2, nous allons charger un dictionnaire et lier la grille à ce dictionnaire : pour chaque emplacement de la grille, nous allons déterminer l'ensemble des *mots candidats*. Au départ ce sont tous les mots du dictionnaire de la bonne longueur. Puis nous réduirons le nombre de candidats en examinant les intersections entre deux emplacements de mots : les lettres des mots choisis doivent coïncider. Enfin, 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.

Au TME 3, nous allons réaliser 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 des TME 1 et 2 aux besoins de l'algorithme. L'outil final permettra de résoudre de grandes grilles en peu de temps : nous ajouterons des heuristiques pour améliorer les performances.

Cette UE de programmation Java avancée suppose une familiarité avec Java (syntaxe de base). Les premières séances sont l'occasion de se remettre à niveau. Cet énoncé contient donc plusieurs encadrés, en gris, qui rappellent les notions clés à appliquer et constituent des « rappels » vis-à-vis de l'UE fortement recommandée en pré-requis « Éléments de programmation par objets avec Java – 2I002 ».

Comme le niveau est assez hétérogène, prenez le temps de bien absorber les concepts si nécessaire, nous pouvons prendre un peu de retard. Le TME 3 propose des extensions dont les réalisations sont optionnelles (bonus) mais qui permettent de traiter efficacement des grilles plus grandes. Si vous êtes à l'aise, vous pouvez aussi prendre de l'avance pour avoir le temps de traiter ces extensions.

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

Dans tous nos TME, nous utiliserons simultanément deux logiciels :

- **Eclipse** : un environnement de développement (IDE) pour Java.
  Cet environnement graphique gérera votre projet localement sur votre compte PPTI. Il vous permettra d'éditer les sources, de les compiler et de les exécuter.
- **GitLab** : un gestionnaire de projets logiciels.
  Basé sur le gestionnaire de versions distribué `git`, GitLab centralise votre projet sur un serveur externe. Il vous permet de garder un historique des modifications (visible avec une interface web), de vous synchroniser avec votre binôme et de faire le rendu du projet au chargé de TME. Nous y avons ajouté une fonction d'intégration continue : toute propagation des modifications locales (Eclipse) vers le serveur GitLab exécutera automatiquement une batterie de tests pour valider votre nouvelle version. Note : nous utilisons ici un serveur GitLab privé, dédié au cours : `https://stl.algo-prog.info`, et pas le GitLab du site public `gitlab.com`.

Nous allons travailler dans un projet nommé `MotCroise`, commun aux TME 1 à 3. Pour mettre en place le projet Eclipse et GitLab du TME, vous devrez suivre les étapes suivantes :

1. **Vérification de la place disponible (quota).**
   Si une des étapes ci-dessous échoue (lancement d'Eclipse, import de projet, etc.), cela peut être dû à un manque d'espace sur votre compte. Avant toutes choses, utilisez la commande `quota` dans un terminal pour déterminer si votre compte dispose d'espace disponible. En particulier, si la première colonne (*blocks*) est égale à la colonne (*limit*) et une étoile ($*$) apparaît, il ne reste plus de place sur le disque ; vous devez absolument libérer de l'espace avant de continuer.
2. **Lancement d'Eclipse.**
   Attention, parfois, plusieurs versions d'Eclipse cohabitent. Il faut utiliser une version pour développement Java. La manière la plus sûre est d'ouvrir un terminal et d'y entrer la commande : `eclipse &`.
   Si c'est la première fois que vous utilisez Eclipse, celui-ci vous demande le nom du répertoire *workspace* où il placera par défaut vos projets. Vous pouvez utiliser le nom par défaut `~/workspace`, sachant que les projets gérés par `git` ne seront pas en réalité hébergés par défaut dans le *workspace*, mais dans un répertoire `~/git` séparé.
   Il est également possible que celui-ci vous demande de choisir un « Subversive SVN Connector ». Vous pouvez ignorer cette étape de configuration, liée au *plugin* « Subversive », en cliquant sur « Cancel » ; en effet, nous n'utiliserons pas ce *plugin*.
3. **Connexion au GitLab.**
   Lancez un navigateur, allez sur la page du serveur GitLab : `https://stl.algo-prog.info` et connectez-vous. Normalement, vous avez du recevoir un *email* du serveur vous précisant les informations de connexion. Si vous n'avez pas reçu cet *email*, prévenez les enseignants. Ne vous inscrivez pas directement sur le serveur, vous n'aurez pas alors automatiquement accès aux projets des TME du cours.
   Votre *username* pour la connexion est votre numéro d'étudiant. Lors de votre première connexion, vous devrez choisir un mot de passe.
4. **Configuration du proxy de navigation** (valable uniquement à la PPTI).
   Si la page web du serveur GitLab n'est pas accessible, vous devez configurer le *proxy* (serveur mandataire) de votre navigateur. Choisissez, pour les protocoles HTTP et HTTPS, le serveur `proxy.ufr-info-p6.jussieu.fr` et le numéro de port 3128.
5. **Configuration du proxy pour Eclipse** (valable uniquement à la PPTI).
   Dans Eclipse, cliquez sur « Window > Preferences » puis « General > Network Connections ». Changez « Active Provider » pour qu'il indique « Manual ». Vous pouvez alors modifier les entrées HTTP et HTTPS pour ce *provider* en cliquant dessus puis sur « Edit... ». Pour les deux entrées HTTP et HTTPS, choisissez comme « Host » : `proxy.ufr-info-p6.jussieu.fr` et comme port 3128.
6. **Configuration de proxy pour git** (valable uniquement à la PPTI).

Afin d'éviter les erreurs d'authentification, vous devez ouvrir un terminal et y taper les commandes suivantes :

```
git config --global http.sslVerify false
git config --global http.proxy https://proxy.ufr-info-p6.jussieu.fr:3128
git config --global https.proxy https://proxy.ufr-info-p6.jussieu.fr:3128
```

(note : les deux dernières commandes ne sont utiles que si vous utilisez `git` en ligne de commande, tandis que la première est aussi nécessaire pour travailler sous Eclipse).

7. **Configuration des notifications de GitLab.**
   Il peut être utile de réduire les notifications envoyées par *email*. Pour cela, dans le menu « Settings », choisissez l'option « Notifications », ouvrez le menu « Global notification level » et cliquez sur « Custom ». Une liste apparaît, et vous pouvez désélectionner certaines notifications. Nous conseillons en particulier de désélectionner « Failed pipeline » pour éviter d'être submergé de notifications lors de l'intégration continue (voir sous-section 1.10).

8. ***Fork* du squelette de projet GitLab.**
   Vous êtes automatiquement membre du groupe `POBJ-XXYY`, où `XXYY` indique l'année en cours (par exemple `POBJ-1819`). Ce groupe contient le projet `MotCroise`, auquel vous pouvez accéder en lecture seule. Le projet est un squelette que vous allez compléter durant les TME 1 à 3. Il fournit quelques sources : exemples, tests, interfaces, etc. Pour réaliser le TME et compléter ce projet, vous devez d'abord en faire **une copie** (*fork*) qui sera **privée** à votre binôme et vous.

   Note : assurez-vous que votre fenêtre de navigateur occupe tout l'écran. Si la fenêtre est trop petite, certains menus de GitLab sont remplacés par des icônes, et vous ne trouverez plus les noms des menus tels qu'indiqués dans la suite de l'énoncé.

   Pour chaque binôme, **un seul d'entre vous** devra faire un *fork* de ce squelette de projet, puis y ajouter son camarade de binôme, et ainsi partager le projet sous GitLab. Pour cela :

   - Dans l'onglet « Projects > Your projects », sélectionnez le projet `POBJ-XXYY/MotCroise`, cliquez sur le bouton « Fork » puis cliquez sur votre nom. Ceci crée le projet personnel privé nommé *`username`*`/MotCroise` (où *username* est votre numéro d'étudiant). Vous travaillerez désormais dans ce projet, où vous avez les droits d'écriture.
   - Dans l'onglet « Projects > Your projects », sélectionnez ce projet privé. Il doit avoir pour nom *`username`*`/MotCroise` et porter la mention « Forked from `POBJ-XXYY/MotCroise` ». Cliquez sur « Settings » dans le menu de gauche, puis « Members ». **Invitez votre binôme** en lui donnant pour rôle « Maintainer ».

   Si c'est votre binôme qui a créé le projet et vous y a ajouté avec les bons droits, vous devriez le trouver dans l'onglet « Projects > Your projects » sous le nom *`partnername`*`/MotCroise` où *partnername* est le numéro d'étudiant de votre binôme.

9. **Ajout du chargé de TME à votre projet GitLab.**
   Dans votre projet personnel sur GitLab, *`username`*`/MotCroise`, invitez également votre chargé TME avec le rôle « Maintainer » (même procédure que ci-dessus).

10. **Import du projet dans Eclipse.**
    L'import d'un projet peut être fait simultanément par tous les membres du projet, par l'élève qui a fait le *fork* comme par son binôme, et sur plusieurs ordinateurs simultanément : cela vous permet de partager et d'échanger les sources développées lors des TME.
    Dans Eclipse, choisissez dans les menus déroulants : « File > Import... > Git > Projects from Git » puis « Next ». Dans l'écran suivant, choisissez l'option « Clone URI » puis « Next ». Dans le champ « URI » de l'écran suivant, entrez l'URI du dépôt du projet. Celle-ci est disponible sur la page web du projet à côté du bouton « Fork » et a la forme : `https://stl.algo-prog.info/`*`username`*`/MotCroise.git` (où *username* est le numéro de l'étudiant qui a fait le *fork* ; attention, l'URI doit se terminer en `.git`).
    Prenez garde à bien importer votre projet de binôme, *`username`*`/MotCroise`, et pas `POBJ-`*`XXYY`*`/MotCroise`. Vous ne pourrez pas travailler dans ce dernier, qui est en lecture seule.
    Vous devez entrer un nom d'utilisateur et le mot de passe associé. Le nom d'utilisateur est

votre *username* sur le serveur GitLab, c'est-à-dire votre numéro d'étudiant (i.e., la personne qui fait l'import dans Eclipse, pas forcément la personne qui a fait le *fork*) ; le mot de passe est celui que vous avez choisi en vous connectant au serveur GitLab la première fois.
Attention : à la PPTI, vous devez utiliser le protocole HTTPS, donc une adresse commençant par `https://`. Une adresse commençant par `git` utilisera le protocole SSH qui ne fonctionnera pas à la PPTI. Le reste de la fenêtre se remplit automatiquement ; cliquez sur « Next ». Les écrans suivants, « Branch Selection » et « Local Destination », sont déjà pré-remplis à des valeurs acceptables ; il suffit de cliquer « Next ». Choisissez « Import existing Eclipse projects » et « Next », puis « Finish ».

Un nouveau projet `MotCroise` apparaît dans le « Package Explorer » à gauche. La mention « `[MotCroise master]` » associée au projet indique que le projet est bien géré par `git`.

Attention : la copie locale des fichiers n'est pas hébergée dans le répertoire `~/MotCroise`, mais dans `~/git/MotCroise` (si vous avez laissé les champs de « Local Destination » à leurs valeurs par défaut).

Nous pouvons à présent créer des classes Java, en sélectionnant un nom de package sous `src` dans le « Package Explorer » et avec clic droit « New > Java 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. Celles-ci ne compilent pas encore, 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. Il a été produit à partir du fichier `data/liste.de.mots.francais.frgut.txt`, fourni mais que nous n'utiliserons pas.
- Plusieurs grilles (`data/*.grl`) vides ou partiellement remplies dans un format textuel (pratique pour les tests).

En cas de problème de connexion au serveur GitLab, afin de vous permettre de commencer néanmoins à travailler, nous fournissons sur la page web de l'UE une archive avec une copie de ces fichiers fournis : `MotCroise.zip`.

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.

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 au minimum :

- 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 s'en servir sous Eclipse, d'abord s'assurer 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 gros 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 fonctions `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 ($expected == actual$) n'est pas vrai, 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]`. Les cases sont accédées 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]`. Les cases sont accédées par `m[i][j]` ou 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 fonctions 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 au minimum :

- 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 rend la case à la position (lig, col) dans la matrice. Nous nous assurerons que les coordonnées de la case sont cohérentes, c'est-à-dire que pour tout couple (l, c) : $getCase(l, c).getLig() == l$ et $getCase(l, c).getCol() == c$,
- une méthode standard `public String toString()` qui permet d'afficher une grille sur la console de façon « lisible » (nous allons utiliser le `GrilleLoader`, fourni, pour construire cet affichage),
- deux accesseurs `public int nbLig()`, `public int nbCol()` rendant respectivement le nombre de lignes et de colonnes de la grille,

- une méthode `public Grille copy()` qui rend 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 le 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'en 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 rendre 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`.

```
@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 au minimum :

- un attribut qui représente les cases (contiguës) de la grille qui composent l'emplacement du mot, nous pouvons utiliser un attribut `private List<Case> lettres` par exemple,
- `public String toString()` qui permet d'afficher le mot à cet emplacement (juste les caractères qui le constituent),
- `public int size()` qui rend la taille de l'emplacement de mot (en nombre de caractères ou cases).

Dans les questions suivantes, nous ajouterons petit à petit un constructeur et des accesseurs (e.g., ajout d'une case, accès aux cases de l'emplacement, etc.) adaptés aux besoins de la classe `GrillePlaces` décrite juste après.

`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)` rend l'objet d'indice i.
- `T set(int i, T val)` affecte *val* à l'objet d'indice i, et rend 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. (c.f. 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.

```
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) {  // parcourir tous les éléments
  System.out.println("Element : " + elt);  // affiche l'élement courant
}
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 Classe `GrillePlaces`

La classe `GrillePlaces` doit explorer une `Grille` pour trouver tous les emplacements des mots qu'elle contient. Nous imposons l'ordre des mots au sein d'une `GrillePlaces`, afin de permettre l'exécution des tests : nous trouvons d'abord les mots *horizontaux*, triés par numéro de ligne croissant. Notons qu'il peut très bien y avoir plusieurs mots sur une ligne, contrairement à l'exemple fourni en

introduction. À numéro de ligne égal, nous rencontrons les mots par colonne de la première lettre croissante, i.e., nous lisons de gauche à droite. Ensuite, nous trouvons les mots verticaux, triés par colonne croissante, et à colonne égale par numéro de ligne de la première lettre du mot croissant.

Par exemple la grille donnée en introduction de cet énoncé comporte au total 12 mots, dont 7 sont horizontaux. La numérotation des cases indiquant la première lettre de chaque mot sur la figure reflète l'ordre dans lequel nous devons rencontrer les mots dans `GrillePlaces`.

⇒ **Donnez l'implantation de la classe `GrillePlaces`**, elle comportera au minimum :

- un constructeur `public GrillePlaces (Grille grille)` qui explore la grille fournie et calcule les emplacements de mots qu'elle contient. Nous pourrons stocker les emplacements trouvés dans un attribut de la classe `private List<Emplacements> places`,
- un accesseur `public List<Emplacement> getPlaces()` pour accéder aux mots détectés,
- un accesseur `public int getNbHorizontal()` pour obtenir le nombre de mots horizontaux (notons que `getPlaces().size()` permet déjà d'obtenir le nombre total de mots détectés),
- une méthode standard `public String toString()` qui permet d'afficher les emplacements de mots détectés de façon lisible.

Sans que ce soit imposé, pour factoriser le code de détection de mots, nous suggérons de :

- définir deux méthodes `private List<Case> getLig (int lig), private List<Case> getCol (int col)` qui rendent les cases qui constituent une ligne ou une colonne donnée (sans les copier),
- écrire une méthode `private void cherchePlaces(List<Case> cases)` qui cherche les mots dans la liste de cases fournie (une ligne ou une colonne) et qui ajoute les emplacements de mot trouvés aux places de la grille. Un emplacement de mot est défini dès que nous avons deux cases contiguës non pleines (donc vides ou avec une lettre déjà placée),
- écrire le constructeur `public GrillePlaces (Grille grille)` qui consiste alors à itérer sur les lignes, chercher les emplacements de mots, noter le nombre d'emplacements horizontaux détectés, puis itérer sur les colonnes en cherchant les emplacements verticaux.

Pour écrire `cherchePlaces`, nous pouvons utiliser un `Emplacement` initialement vide. À chaque case rencontrée :

- si elle est non pleine, nous l'ajoutons à l'emplacement,
- sinon, nous examinons la taille de l'emplacement construit,
  - s'il fait au moins deux lettres, nous l'ajoutons aux emplacements de mots détectés,
  - sinon, nous réinitialisons l'emplacement.

N'oubliez pas (à la fin de l'itération) d'ajouter le dernier emplacement trouvé s'il est assez long.

⇒ **Exécutez le jeu de test `pobj.motx.tme1.test.GrillePlacesTest` fourni.**

1.8 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

```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 et votre programme principal.**

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).

1.9 Synchronisation avec le serveur GitLab

Après avoir exécuté `git clone` en début de TME, nous avons travaillé sur une copie locale du projet. Il est nécessaire de synchroniser périodiquement votre projet local avec le projet GitLab pour :

- communiquer vos fichiers à l'enseignant pour le rendu (celui-ci a accès aux fichiers sous GitLab, mais pas à ceux sur votre compte local),
- vous synchroniser avec votre binôme,
- éventuellement synchroniser des copies locales sur plusieurs ordinateurs,
- garder une trace des modifications et pouvoir éventuellement revenir à une version précédente en cas d'erreur.

Les opérations utiles sont donc la propagation d'une copie locale vers le serveur (*push*) et depuis le serveur vers une copie locale (*pull*). Il est possible d'utiliser `git`, soit depuis l'interface graphique d'Eclipse, soit en ligne de commande.

Vous pouvez consulter l'état des fichiers sur le serveur GitLab en utilisant le site web `https://stl.algo-prog.info`. Vous y trouverez la dernière version des fichiers et l'historique des modifications. Vous pourrez en particulier vérifier que le projet a bien été synchronisé pour le rendu de TME.

1.9.1 Git sous Eclipse

Eclipse incorpore un *plug-in* EGit d'intégration avec `git`. Celui-ci a dû reconnaître automatiquement que le projet `MotCroise` était un dépôt `git` lié au serveur GitLab.

Push. Pour propager des modifications depuis le projet local vers GitLab, faites un clic droit sur le projet, puis « Team > Commit… ». Un onglet « Git staging » apparaît en bas, ou dans une fenêtre (l'interface varie d'une version d'Eclipse à l'autre ; votre interface peut donc varier sensiblement de celle décrite ici). La zone « Unstaged changes » contient la liste des fichiers modifiés (ou ajoutés) localement mais non encore sur le serveur. Déplacez les fichiers que vous voulez synchroniser (fichiers modifiés ou nouveaux fichiers) vers la zone « Staged changes ». Entrez un bref message décrivant les modifications dans « Commit message ». Vérifiez la validité des zones « Author » et « Comitter ». En principe, elles devraient être identiques et contenir *`username <email>`* où *username* est votre numéro d'étudiant. Si c'est bien le cas, alors cliquez sur « Commit and push… », sinon, avant de cliquer, vous pouvez renseigner ces zones manuellement.

Pull. Pour récupérer localement des modifications disponibles sur le serveur GitLab, faites un clic droit sur le projet, puis « Team > Pull ».

Conflits. Si des modifications ont été faites sur le serveur (par exemple par une propagation, *push*, de votre camarade) depuis votre dernier *pull*, vous ne pourrez pas propager vos modifications locales directement ; `git` refusera avec une erreur. En effet, cela provoquerait des conflits entre deux nouvelles versions d'un fichier. `git` vous force à résoudre les conflits localement, avant de propager vos fichiers corrigés vers le serveur :

- Faites d'abord un *pull*.
- Les fichiers marqués d'un diamant rouge dans le « Package Explorer » à gauche sont les fichiers avec un conflit. `git` s'est efforcé de fusionner les modifications locales avec celles présentes sur le serveur, mais il a pu faire des erreurs ; vous devez examiner chaque fichier et corriger à la main les problèmes causés par la fusion. Les zones non fusionnées sont identifiées par des balises `<<<<<<`, `-----` et `>>>>>>` dans votre source Java. `git` vous indique de cette manière les deux versions disponibles (version locale et dernière version disponible sur le serveur). Il s'agit souvent de choisir une de deux versions, en supprimant les lignes redondantes et les balises.
- Après avoir examiné et éventuellement corrigé un fichier en conflit, vous devez faire un clic droit sur le nom du fichier dans le « Package Explorer », puis « Team > Add to index » pour indiquer que le fichier est maintenant correct. Le diamant rouge disparaît. Ceci doit être fait pour chaque fichier ayant un conflit.
- Après suppression de tous les conflits, vous devez faire un *commit*, avec « Team > Commit ». Le message de *commit* est renseigné automatiquement : il indique les fichiers qui étaient en conflit (vous pouvez bien sûr modifier ce message).
- Vous pouvez enfin faire un *push*.

Bonnes pratiques. C'est une bonne idée d'anticiper les conflits en **commençant toute session**

de travail par un *pull*, pour repartir avec les dernières versions des fichiers disponibles sur le serveur, et en **terminant toute session de travail par un *push***, pour que vos modifications locales soient envoyées sur le serveur et puissent être importées par votre binôme ou vous-même sur un autre ordinateur.

Notez qu'il est possible d'éditer les fichiers du projet directement sur le serveur GitLab dans l'interface web. Une telle modification compte comme un *push*, et vous devez donc l'importer dans Eclipse avec un *pull* avant de continuer à travailler sous Eclipse.

La documentation complète du *plug-in* EGit se trouve à `http://www.eclipse.org/egit/documentation`.

1.9.2 Git en ligne de commande (alternative à Eclipse)

Dans un terminal, placez-vous dans le sous-répertoire de `~/git` contenant votre projet. Les commandes les plus utiles sont :

- `git add fichiers` pour indiquer les fichiers ajoutés ou modifiés localement ;
- `git commit` pour enregistrer localement les ajouts ou modifications des fichiers spécifiés par `git add` ;
- `git push` pour effectivement propager l'enregistrement local vers le serveur ;
- `git pull` pour rapatrier localement les modifications depuis le serveur.

Le système `git` est décrit dans le livre en ligne : `https://git-scm.com/book/en/v2`.

1.10 Intégration continue sous GitLab

L'intégration continue est une pratique de développement logiciel consistant à s'assurer que, à chaque instant, le dépôt est correct et passe tous les tests. Le serveur GitLab est configuré pour l'intégration continue : après chaque propagation de votre copie locale vers le serveur, le code sur le serveur est compilé et les tests JUnit fournis pour les TME sont exécutés automatiquement.

Vous pouvez consulter le résultat des tests sur le serveur GitLab `https://stl.algo-prog.info` en cliquant sur votre projet, puis dans le menu à gauche sur « CI / CD > pipelines ». Les tests de la dernière version apparaissent en haut. Un icône « V » vert ou une croix rouge indique l'état du test (un croissant ou un symbole pause indique que le test est en cours ou en attente, il faut donc patienter). Cliquer sur l'icône dans la colonne « *Status* » permet de voir l'ensemble des classes de test. Cliquer sur un nom de test vous donne un rapport complet de test, indiquant en particulier quelles méthodes de test ont échoué, et avec quelles erreurs.

Ce mécanisme vient compléter l'exécution des tests unitaires que vous devriez lancer périodiquement (au moins avant chaque propagation vers le serveur) sur votre copie locale depuis Eclipse. Par ailleurs, le chargé de TME a accès aux rapports de tests sur le serveur GitLab, ce qui lui permet d'évaluer votre rendu de TME.

Le serveur est configuré pour exécuter tous les tests des TME 1 à 3. Tant que vous n'avez pas programmé toutes les classes demandées, de nombreux tests vont échouer. Vous ignorerez donc pour l'instant les tests liés aux TME 2 et 3, et essaierez de faire fonctionner les tests liés au TME 1.

L'intégration continue est gérée par le fichier `.gitlab-ci.yml` à la racine de votre projet, qui spécifie les classes de test. Pour chaque classe de test, le script `scripts-ci/run.sh` est exécuté.

1.11 Rendu du TME (OBLIGATOIRE)

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

Chaque semaine, il est obligatoire de rendre le TME à votre chargé de TME en fin de la séance. Si vous le souhaitez, vous pouvez aussi rendre **une seconde version améliorée avant le début du TME suivant**.

Pour chaque TME, le rendu devra comporter toutes les classes demandées dans l'énoncé du TME. La section *Rendu du TME* comporte également des questions auxquelles vous devrez répondre dans votre rendu, ou bien des fichiers ou images à fournir.

Le rendu se fait en propageant vos modifications vers le serveur GitLab, comme indiqué à la sous-section 1.9, et en y associant un *tag*. Pour cela :

- Connectez-vous sur la page de votre projet sous `https://stl.algo-prog.info`.
- Assurez-vous que votre chargé de TME est membre de votre projet, avec le rôle « Maintainer ».
- Vérifiez que toutes les classes demandées sont bien présentes sous GitLab et bien synchronisées avec le projet local visible sous Eclipse.
- Vérifiez également que les tests unitaires du TME 1 lancés par l'intégration continue sur le serveur GitLab se sont exécutés correctement (voir 1.10).
- Dans le menu de gauche, sélectionnez « Repository > Tags » et cliquez sur « New Tag ».
- Donnez un nom à votre *tag* : « rendu-initial-tme1 » ou « rendu-final-tme1 », selon qu'il s'agit d'un rendu partiel en fin de séance ou bien d'un rendu de TME finalisé.
- Dans le champ « Release notes », entrez la réponse aux questions posées ci-dessous. Vous pourrez également, quand c'est demandé dans l'énoncé, attacher des fichiers (texte, image). N'hésitez pas à compléter ce champ avec toutes les informations que vous jugerez utiles pour évaluer votre TME : difficultés rencontrées, choix qui s'éloignent de l'énoncé, justification pour l'échec de certains tests JUnit, etc.
- Cliquez sur « Create tag ».
- En cas d'erreur, il est toujours possible de créer un nouveau *tag*.

Il est impératif de créer un *tag* pour chaque rendu, et de réaliser au moins un rendu par TME. Ces rendus réguliers (code source, validation des tests, réponses aux questions) sont évalués dans le cadre du contrôle continu.

⇒ **Répondez à ces questions dans 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 aussi (ou attachez) dans le champ la trace d'exécution de la suite de tests unitaires : `pobj.motx.tme1.test.TME1Tests`.