



# Activité Java : Jeu de la grenouille



## Tables des matières

### Les bases de Netbeans 2

Niveau 0 : Introduction aux variables *	★	3
Niveau 1 : Utilisation des variables *	★	5
Niveau 2 : Appels de fonctions	★★	7
Niveau 3 : Boucle répéter	★★★	9
Niveau 4 : Mise en pratique	★★★★	11
Niveau 5 : Boucle faire tant que	★★★★★	12
Niveau 6 : Boucle faire tant que	★★★★★★	14
Bonus : Easter Eggs	★★★★★	15
Bonus : Tests des fonctions	★★★★★	15

Cet atelier vous expliquera les **bases de l'algorithmique** grâce au langage de programmation **Java**. Pour écrire le code, nous allons utiliser l'environnement de développement Netbeans. Les ★ représentent la difficulté de chaque niveau. Les niveaux précédés d'un \* sont obligatoires pour faire la suite.



Le jeu de la grenouille consiste à faire **déplacer** un personnage sur un plateau en utilisant les 4 **directions cardinales** (haut, bas, gauche, droite). L'objectif est de faire **arriver** la grenouille sur le nénuphar, en faisant attention à ne pas toucher un **mur** et à ne pas **sortir** du plateau.

## Les bases de Netbeans

### Netbeans, c'est quoi ?

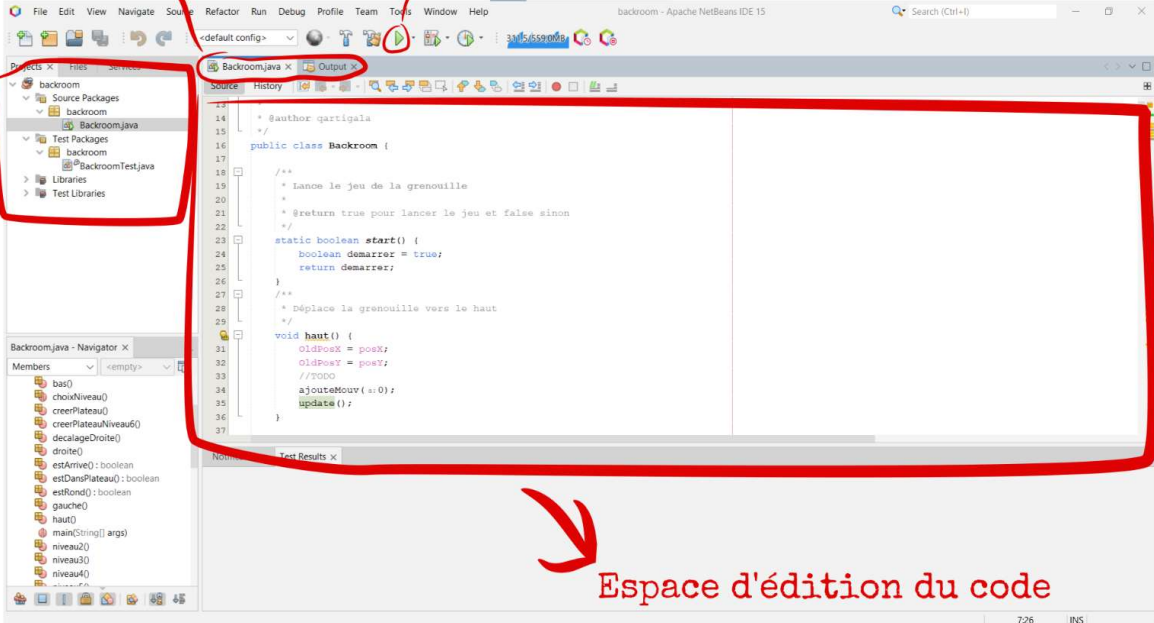
Netbeans est un **environnement de développement** libre, c'est-à-dire un logiciel comportant un ensemble d'outils permettant de développer un logiciel. C'est notamment dans ce dernier que vous allez taper votre **code**, le **compiler** puis le **lancer**.

**Onglet Code/Console**

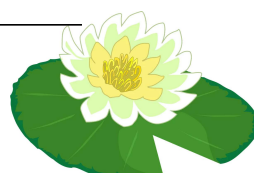
**Fichiers composant le projet**

**Compiler et lancer le programme**

**Espace d'édition du code**



```
14  * @author qartigala
15  */
16  public class Backroom {
17
18      /**
19       * Lance le jeu de la grenouille
20       * @return true pour lancer le jeu et false sinon
21       */
22      static boolean start() {
23          boolean demarrer = true;
24          return demarrer;
25      }
26      /**
27       * Déplace la grenouille vers le haut
28       */
29      void haut() {
30          oldPosX = posX;
31          oldPosY = posY;
32          //posX
33          ajouteMour(a:0);
34          update();
35      }
36  }
37
```



## Niveau 0 : Introduction aux variables \*



### Que dois-tu comprendre ? 🧐

En programmation, une variable est un élément contenu dans la **mémoire** de l'ordinateur. On y associe un **nom** et une **valeur**. Ainsi, on pourra réutiliser la valeur contenue dans une variable à divers endroits du code en l'appelant par son nom. Les variables possèdent chacune un **type**, qui peut être un nombre **entier**, un nombre à **virgule**, un **caractère**, un enchaînement de plusieurs caractères (**phrase**) ou encore un **état** (vrai / faux).

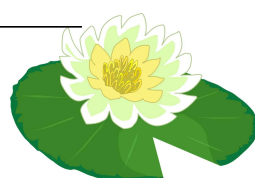
Voici un code Java utilisant le concept de variable :

Type      Nom      Valeur

```
int entier = 1;  
double decimal = 0.5;  
boolean faux = false;  
boolean vrai = true;  
char car = 'a';  
String chaine = "coucou";
```

### À toi de jouer ! ▶

Avant de pouvoir commencer à jouer, tu vas devoir faire **démarrer** le jeu. Pour ce faire, il va falloir modifier une fonction *start*. Une **fonction** est une **portion de code** auquel on attribue une **tâche** particulière. Nous verrons ce concept plus en détail par la suite.



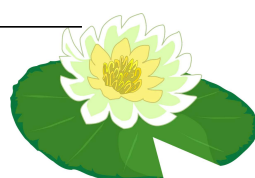
Placez-vous dans l'éditeur de code de Netbeans, trouvez la fonction `start()` et corrigez l'erreur afin qu'elle fonctionne correctement.

```
static boolean start() {  
    boolean demarrer = false;  
    return demarrer;  
}
```

Ce bout de code déclare une fonction nommée `start`. Dans cette dernière, nous créons une **variable d'état** (`demarrer`) initialisée comme étant **false** (fausse). Ensuite, le mot clé **return** permet de **renvoyer** la valeur de la variable associée. En somme, cette fonction renvoie la valeur "faux". Normalement, elle devrait renvoyer "**true**" (vrai). Pour cela, changez la valeur de la variable `demarrer`.

Pour **vérifier** si le jeu se lance **bien**, cliquez sur la **flèche verte** en bas de la barre d'outils (**F6**). Cela va **compiler** et **lancer** le projet. Vous allez alors pouvoir constater le **résultat** du programme dans la **console (Output)** se trouvant dans la barre d'onglets. C'est ici que le jeu s'affichera.

Ensuite un menu se **présente** à vous, il vous suffira d'**écrire** dans la console le choix que vous voulez faire (Ici ce sera avec des chiffres 2, 3 ...) puis tapez sur entrée. Une fois l'exécution du programme **terminée**, vous pouvez cliquer sur l'onglet **Backroom.java** pour repasser à l'édition du **code**.



## Niveau 1 : Utilisation des variables \*



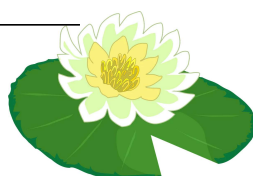
### Un peu de pratique ! ►

Maintenant, vérifiez si vous avez bien assimilé le concept de variables en complétant le contenu de la fonction `haut()`. Quand ce bout de code est appelé, il est censé faire **déplacer** la grenouille sur la case située au-dessus d'elle, sur la grille de jeu. Pour ce faire, **inspirez-vous** des fonctions `bas()`, `gauche()` et `droite()` déjà codées.

Décortiquons la fonction **`bas()`** pour savoir comment elle fonctionne en détail.

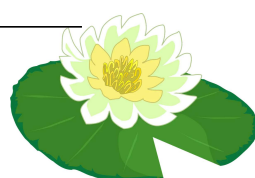
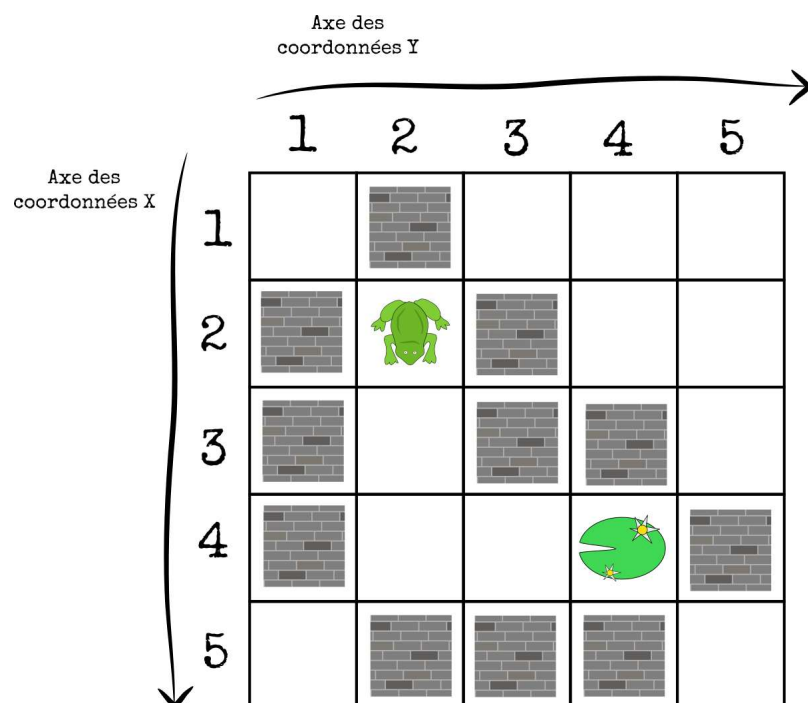
```
/**
 * Déplace la grenouille vers le bas
 */
void bas() {
    OldPosX = posX;
    OldPosY = posY;
    posX = posX + 1;
    ajouteMouv(1);
    update();
}
```

D'abord, on **sauvegarde** l'ancienne position de la grenouille en mettant sa **position actuelle** (contenue dans `posX` et `posY`), dans les variables `OldPosX` et `OldPosY` qui contiendront son **ancienne position** (à des fins d'affichage).



Ensuite, on **augmente** la coordonnée X de la grenouille sur le plateau de **1**. Puis on ajoute le mouvement dans un tableau (à des fins d'affichage).

Enfin, on appelle la fonction `update()` qui permet entre autres de changer l'affichage... Le plateau sur lequel **évolue** le joueur peut se schématiser de la façon suivante :



## Niveau 2 : Appels de fonctions



### Que dois-tu connaître ? 🧐

Comme évoqué précédemment, une **fonction** est un **morceau de code** qui a un but **prédéfini** et qui peut être **utilisé** à n'importe quel endroit du code. Pour **appeler** une fonction, il suffit d'écrire son nom, suivi de 2 parenthèses (avec des arguments à l'intérieur si nécessaire, ici non), comme ceci :

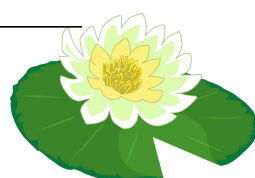
```
haut () ;
```

⚠️ La **syntaxe est très importante** ! N'oubliez pas de mettre le point virgule en fin de ligne ';'.

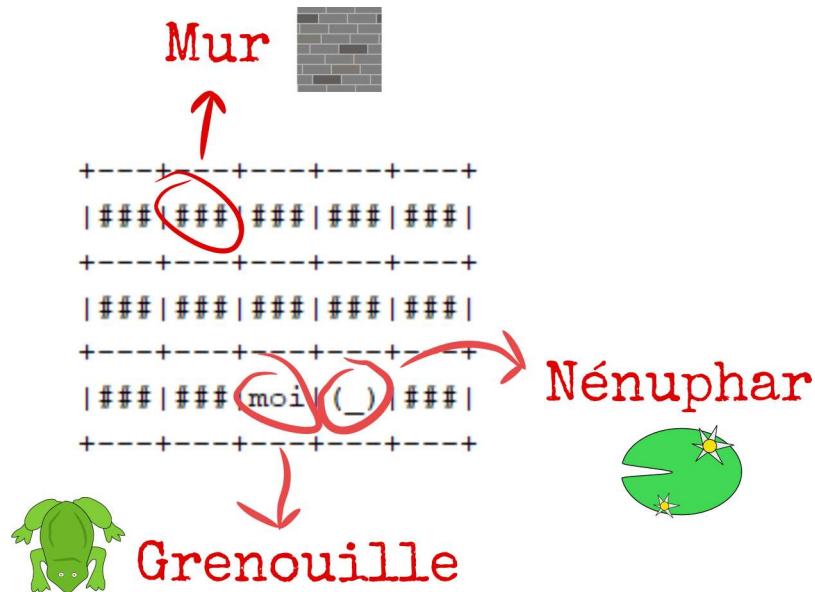
### Que suis-je censé faire ? 🧐

Maintenant que le jeu est lançable, on va voir quelles sont les **règles** du jeu. L'objectif est de faire **déplacer** une grenouille sur un plateau de 5x5 pour la faire arriver sur son **nénuphar**, sans toucher les **murs**.

Pour cela, elle peut se déplacer dans les 4 **directions cardinales** : haut, bas, gauche et droite. Ainsi, en appelant la fonction gauche, la grenouille se déplace d'une case vers la gauche sur le plateau. L'objectif va être, grâce à une **suite d'appels des fonctions cardinales**, de faire arriver la grenouille sur son nénuphar.

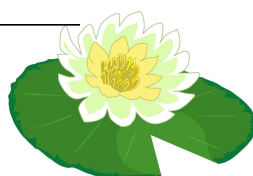


Voici à quoi ressemble le plateau actuel du niveau 2 :



### À toi de jouer ! ►

Placez-vous dans la fonction appelée *niveau2()* et écrivez la suite d'instruction permettant de faire aller la grenouille sur son **nénuphar**. Pour ce faire, appelez les **fonctions cardinales** vues précédemment.





## Niveau 3 : Boucle répéter



### Que dois-tu connaître ? 🧐

Une boucle **répéter** permet de répéter un **nombre défini** de fois une suite **d'instructions**. Ainsi, avec la syntaxe Java suivante, la fonction *droite()* est appelée **3** fois avec le code ci-dessous: le personnage se déplace de 3 cases vers la gauche.

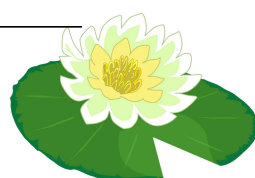
Déclarer une boucle

```
for (int i = 0; i < 3; i++) {  
    droite();  
}
```

Nombre de répétitions

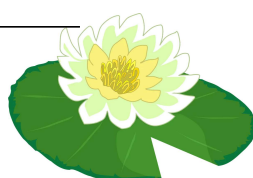
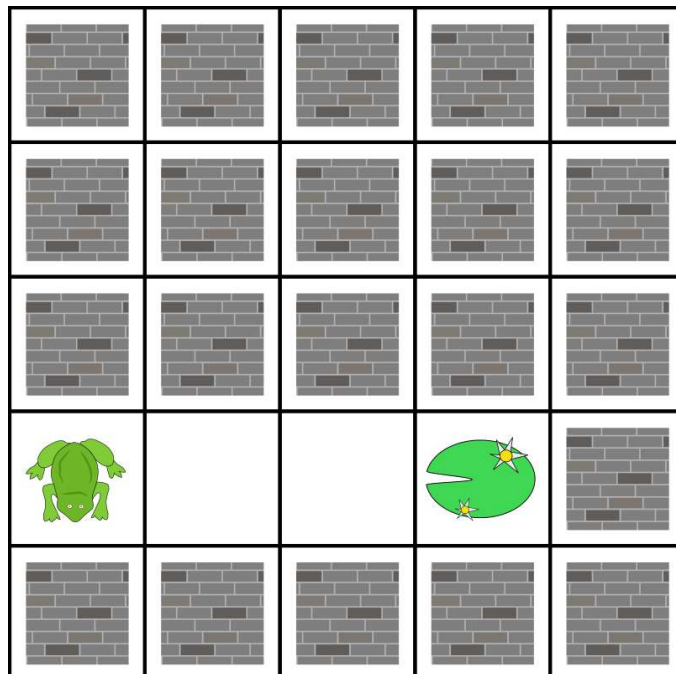
Délimitent le code à répéter

⚠ La **syntaxe est très importante** ! N'oubliez pas de mettre les **parenthèses**, les **accolades** et les **“;”**, notamment en fin de ligne.



À toi de jouer ! ►

Placez-vous dans la fonction appelée *niveau3()* et écrivez la suite d'instruction permettant de faire aller la grenouille sur son *nénuphar*. Pour ce faire, vous pouvez utiliser la boucle **répéter** vue précédemment. Voici à quoi ressemble le plateau du niveau 3 :

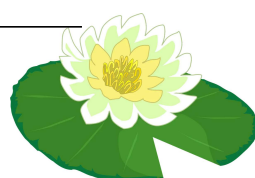
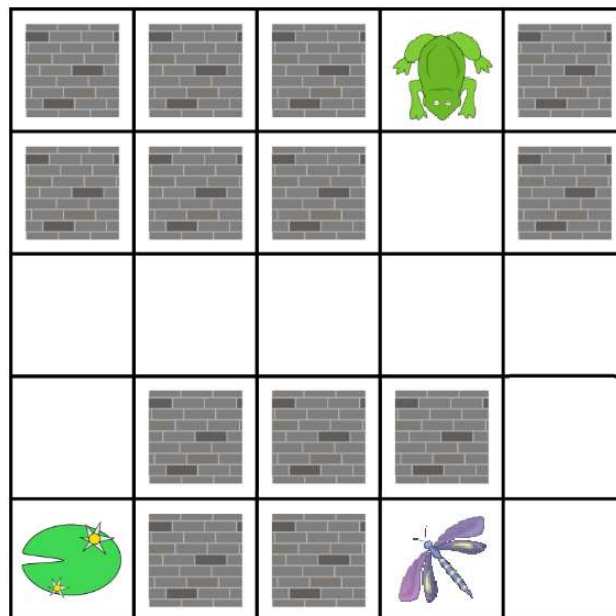


## Niveau 4 : Mise en pratique



### Un peu de pratique ! ►

Pour vous entraîner, essayez de remplir la fonction `niveau4()` afin de faire arriver la grenouille sur son nénuphar. Pour ce faire, vous pouvez réutiliser les notions vues précédemment telles que les boucles **répéter** et les appels de **fonctions**. Voici le plateau correspondant au niveau 4 :



## Niveau 5 : Boucle faire tant que



### Que dois-tu connaître ? 🧐

Il se peut que vous voulez **répéter** une instruction, **sans connaître** dès le départ son nombre exact d'**exécutions**. Vous pouvez alors utiliser une boucle **faire tant que**. Cette dernière permet de répéter tant que l'expression qui suit est **vraie**. Voici un exemple concret :

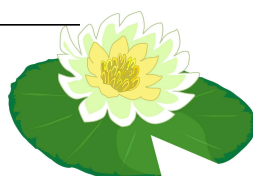
Déclarer un tant que

```
while ( !estArrive() ){  
    droite();  
}
```

Diagram annotations:

- An arrow points from the text "Déclarer un tant que" to the `while` keyword.
- An arrow points from the text "Expression vérifiée" to the `!estArrive()` expression.
- A red bracket under the code block is labeled "Délimitent le code à répéter".

Le signe "!" devant `estArrive()` signifie "**non**". Ainsi, cette boucle permet de se **décaler** d'une case à droite **tant qu'on n'a pas atteint l'arrivée**.

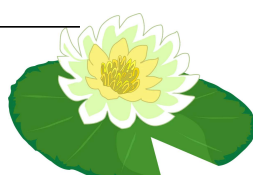
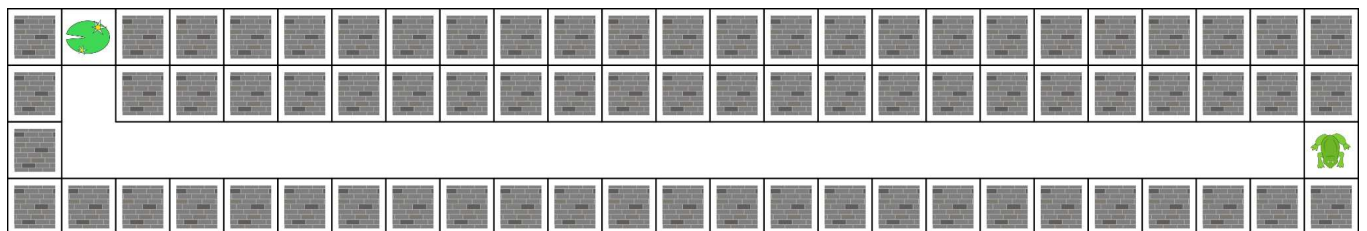


## Un peu de pratique ! ▶

Placez-vous dans la fonction appelée *niveau5()* et écrivez la suite d'instruction permettant de faire aller la grenouille sur son **nénuphar**. Pour ce faire, vous pouvez utiliser la structure **faire tant que** vue précédemment ainsi que des fonctions suivantes prédéfinies dans le code :

- `estArrive()` est vraie si la grenouille **se trouve** sur le nénuphar.
- `pasDeMur(char)` est vraie s'il n'y a pas de **mur** dans la **direction** choisie. Pour ceci, changez **char** par l'**initial** de la direction voulue. (**haut,bas,gauche,droite**)

Voici à quoi ressemble le plateau du niveau 5 :



## Niveau 6 : Plateau à défilement

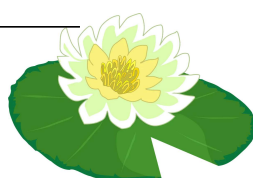


### Que suis-je censé faire ? 🤖

Pour les plus fêrus, voici une touche de **difficulté** supplémentaire ! Dans ce niveau, vous n'avez **aucun aperçu global** de la **disposition** du **plateau**. Ainsi, vous ne pouvez voir que **3 cases autour** de la grenouille. Vous devez vous-mêmes, essais après essais, **découvrir** et **mémoriser** le plateau afin de pouvoir coder le bon enchaînement d'instructions.

### À toi de jouer ! ▶

Placez-vous dans la fonction appelée *niveau6()* et écrivez la suite d'instruction permettant de faire aller la grenouille sur son **nénuphar**. Pour ce faire, faites plusieurs **essais** afin de **découvrir** le parcours à suivre.



Bonus : Easter Eggs



### Des secrets cachés, où ça ? 🧐

Nous avons **caché** dans le jeu de nombreux **événements** et **messages secrets**. Arriverez-vous à les déceler ? Pour cela, exploitez toutes les **fonctionnalités** offertes par les règles, soyez perspicaces et astucieux ! Il faut avoir l'**œil**, bonne chance !

Astuce : Saisissez **8** dans le menu du jeu pour **afficher** des **indices** ! 😊

Bonus : Tests des fonctions



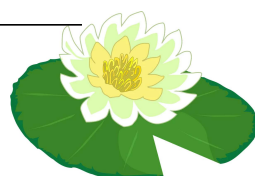
### Tester votre code, pour quoi faire ? 🧐

Comme vu précédemment, un programme est **découpé** en plusieurs bouts de code ayant un rôle spécifique, ce sont des **fonctions**. On peut alors effectuer des **tests**, pour voir si les fonctions qu'on a créées répondent bien à nos **attentes**.

### Mais comment faire ? 🧐

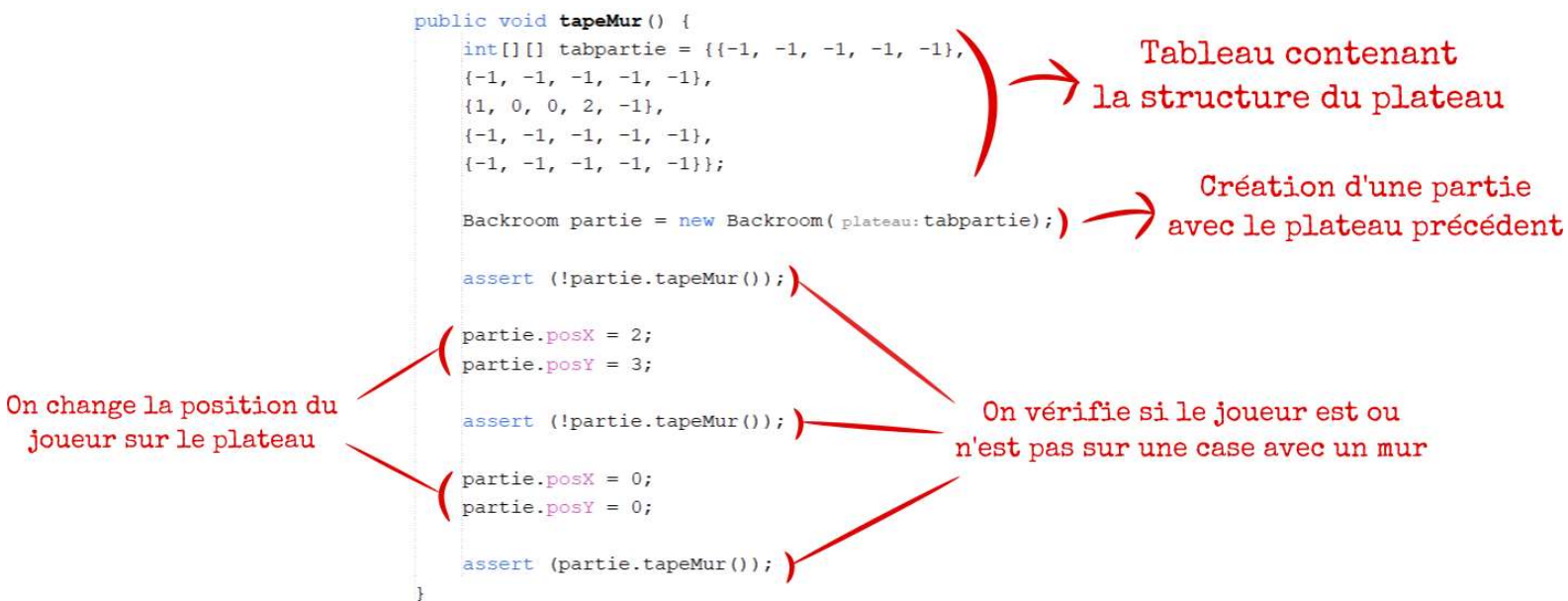
Pour procéder à la **vérification**, on va se placer dans un **fichier particulier** consacré aux tests. Dans votre cas, double-cliquez sur le fichier **backroomTest.java**, dans l'explorateur de fichiers (partie gauche de la fenêtre).

À chaque **fonction importante** du code principal est associée une **fonction de test**. Cette dernière est précédée du mot clé **@Test**, et se nomme pareil que la fonction testée, suivie du mot **Test**.



Pour vérifier qu'une fonction renvoie bien la valeur **attendue**, on utilise le mot clé **assert()**. On passe alors dans les parenthèses le nom de la fonction **testée**. Une **erreur** sera renvoyée si la fonction ne renvoie pas vrai, nous permettant ainsi de savoir si le test s'est déroulé comme **prévu**.

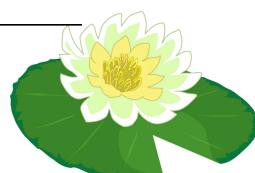
Ci-dessous, voici un schéma **explicitant** le code de la fonction de test **tapeMur()** :



Le tableau **tabpartie** représente le **plateau** testé. Voici à quoi correspondent les **chiffres** contenus dans ce dernier :

- **-1** : case avec **mur**
- **0** : case **vide**
- **1** : case avec la **grenouille**
- **2** : case avec le **nénuphar**

⚠ Si vous souhaitez vérifier que la grenouille **n'est pas** dans un mur, il faut ajouter le symbole "!" dans la fonction **assert()**.





À toi de jouer ! ►

Complétez les fonctions de tests `estArriveTest()` et `estDansPlateauTest()`, données dans le fichier *backroomTest.java*.

Pour ce faire, inspirez-vous de la fonction de test de *tapeMur()* et du schéma explicatif ci-dessus.

Pour vérifier si votre fonction de test **fonctionne** correctement, cliquez sur la **flèche verte** à gauche de cette dernière, puis cliquez sur “**Run [...] method**”. **Aucune** erreur ne devrait s’afficher dans la console.

Copyright : Fait par © Leo Graziani,

© Adrien Duollé,

© Quentin Artigala ,

© Clément Aubier

