NOM :	PRÉNOM :	GROUPE :
LIÉNARDY	**SIMON**	**1I7**

Méthode : `decalageCircDroite`

1. DESCRIPTION DU « QUOI » = COMPRENDRE LE PROBLÈME

Nom de la méthode?

`decalageCircDroite`

Sa tâche ? *Expliquer brièvement le traitement que doit réaliser la fonction/méthode*

Dans un tableau d'entier, décaler les éléments de manière circulaire d'une case vers la droite.

Préconditions (= ses entrées et les valeurs admises) ? *Lister et nommer les données nécessaires à la réalisation du traitement – ces données peuvent varier d'une exécution à l'autre de la méthode. Dans un contexte orienté objets, listez aussi les attributs de l'objet dont la méthode a besoin mais qu'elle ne modifie pas.*

A côté de chaque donnée ou attribut de la liste, indiquez les éventuelles conditions qu'il/elle doit remplir pour que la méthode puisse réaliser le traitement demandé.

Une référence non-nulle vers un tableau d'entiers.

Ici, on peut déjà décider du nom et du type des paramètres de la fonction :

```
int[] valeurs
```

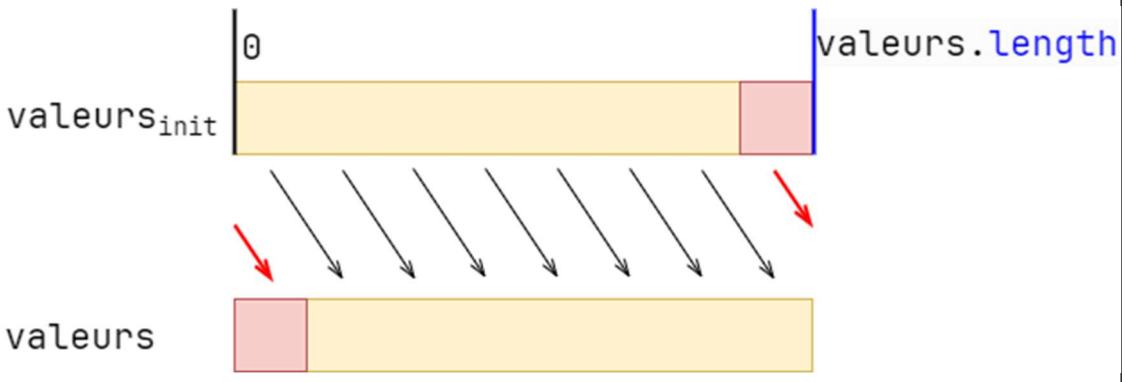
Postconditions (= sa sortie et les garanties concernant celle-ci) ? *Indiquer la donnée qui doit être retournée au terme du traitement réalisé par la fonction. Dans un contexte orienté objets, listez et nommez aussi les attributs de l'objet que la méthode initialise ou modifie.*

A côté du résultat et des attributs initialisés ou modifiés, indiquez les conditions qu'il doit remplir pour être considéré comme correct.

Dans ce cadre, je vais devoir expliquer qu'une portion de tableau est identique à une portion du tableau initial. Je vais utiliser l'indice « `init` » pour le tableau du départ. Sans l'indice, c'est l'état actuel du tableau tel qu'il est après l'exécution de la fonction `decalageCircDroite`

- `valeurs[1..length-1] = valeurs`$_{init}$`[0 .. length-2]`
- `valeurs[0] = valeurs`$_{init}$`[length-1]`

C'est encore plus simple de faire un dessin :



Son plan de test = résolution d'au moins six cas concrets : 2 cas « normaux », 1 cas par « branchement » réalisé par la méthode, 1 cas d'erreur géré par la méthode, 1 cas limite (ensemble vide, première ou dernière valeur, zéro ou nul, etc.), éventuellement un cas de dépassement de capacité, 1 à 2 combinaisons de cas

Pour chacun, indiquer 1. le type de cas 2. les entrées spécifiées et les attributs utilisés en lecture 3. les effets attendus = le résultat et les valeurs des attributs d'objets initialisés ou modifiés

(Quand vous avez l'habitude, vous pouvez faire ceci conjointement avec le point 2).

Cas normaux

- `{1, 2, 3, 4, 5}` → `{5, 1, 2, 3, 4}`
- `{18, 25, 317, -4, 59, 67, 3, 12}` → `{12, 18, 25, 317, -4, 59, 67, 3}`

Cas de branchement

- *Il n'y en a pas vraiment ici. Si la méthode permettait de décaler le tableau de plus d'une case ou de changer la direction du décalage selon une valeur passée en argument, c'est cela qu'il faudrait tester*

Cas d'erreur

- Référence `null` : assurez-vous que le programme génère une `NullPointerException`.
  Conseil, utilisez cette assertion :

```
assertThrows(NullPointerException.class,
    () -> {TableauEntiers.decalageCircDroite(null);});
```

Cas limites

- Un tableau vide ne doit pas être modifié (et ne doit pas générer d'erreurs) :
  `{}` → `{}`
- Un tableau d'un élément ne doit pas être modifié :
  `{42}` → `{42}`

Combinaisons de cas

- `{1,2}` → `{2,1}` *(cas normal proche de la limite)*
- Quelques grands tableaux au choix, de longueur paire et impaire.
- *Ici, remplir le tableau avec des valeurs comme Integer.INT_MAX ou INT_MIN n'a pas beaucoup d'intérêt.*

2. FORMALISATION DES TESTS = PRÉPARER LEUR ÉCRITURE EN JAVA

La traduction de chaque test du plan de test

Ecrivez chaque test en respectant le schéma Given-When-Then. Exemple : étant donné un objet dictionnaire contenant les 5 mots arbre, fleur, chien, chat, ordinateur et leurs définitions, quand j'appelle la méthode de recherche avec le mot « arbre » en entrée, alors j'obtiens bien la définition d'un arbre

…

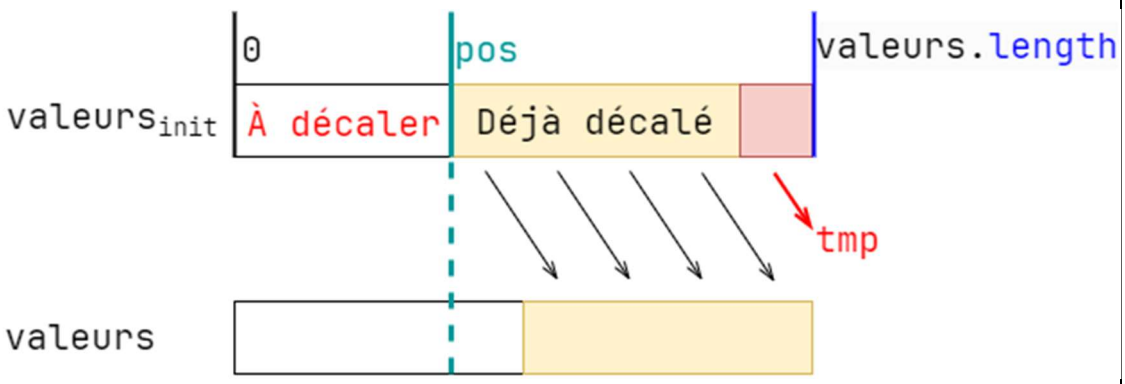
3. DESCRIPTION DU « COMMENT » = IMAGINER UN ALGORITHME

Nom de l'algorithme ? *Si vous appliquez un algorithme célèbre, indiquez son nom.*

Pour l'exercice, je vous le détaille. En guise d'entrainement. En pratique, on utiliserait `System.arraycopy`

Dessin. Réalisez un dessin de la situation finale de votre problème et déduisez-en une situation plus générale

La situation finale est déjà représentée plus haut. L'algorithme consiste à décaler les cases une par une vers la droite. Puisqu'il y a plusieurs cases dans le tableau, une ***boucle*** *me semble indiquée pour résoudre mon problème. Quand je veux décaler la case* `x` *vers la droite, j'ai envie de copier son contenu dans la case à sa droite,* `x+1`*. Pour ce faire, il faudrait que j'aie déjà décalé cette case aussi. Si je continue mon raisonnement, je peux conclure que la première case à devoir être décalée est l'avant-dernière du tableau. Quant à la valeur de la dernière case, il faut la sauvegarder pour plus tard. On va donc parcourir le tableau dans le sens décroissant de ses indices, comme représenté dans le schéma ci-dessous :*



J'ai représenté une sorte de « photo intermédiaire » de mon programme. Une section (en jaune) a déjà été décalée alors qu'une autre doit encore l'être. Je trace une délimitation verticale entre les deux zones et j'introduis une variable qui va me permettre de repérer où j'en suis dans mes décalages. La position `pos` *indique la prochaine case dans laquelle on doit décaler une valeur (celle qui vient de la case* `pos - 1`*).*

Ses étapes ? *Répertorier textuellement ou en « pseudocode » ou encore sous forme d'organigramme, dans l'ordre, les étapes principales que la méthode doit appliquer pour résoudre le problème pour toutes les entrées possibles. Vérifiez que votre algorithme « fonctionne » au moins pour tous les cas de tests prévus ! Assurez-vous de la* ***terminaison*** *de votre algorithme*

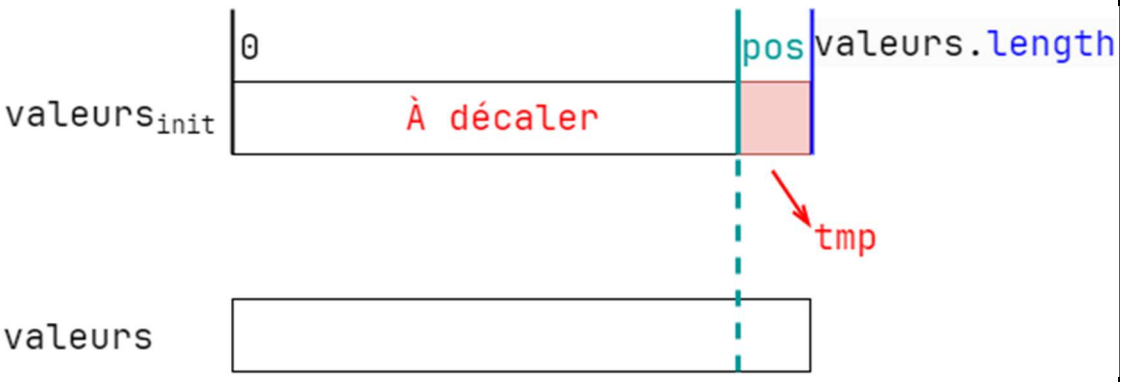
Une boucle suit le modèle suivant :

```
INITIALISATION
while(GARDIEN) {
   CORPS DE LA BOUCLE
}
FIN
```

Je dois donc expliquer ce qui doit remplacer `INITIALISATION`, `GARDIEN`, `CORPS DE LA BOUCLE`, et `FIN`. *Je vais m'aider de mon dessin.*

`INITIALISATION`

Les initialisations sont les opérations qui **précèdent** la boucle. Dans mon dessin, j'ai identifié deux variables : `tmp` et `pos` dont je dois donner les valeurs initiales. Je n'ai pas envie de calculer cela moi-même donc j'ai « juste » modifié mon dessin général pour représenter la situation dans laquelle je n'ai encore rien fait (il n'y a plus de section jaune) :



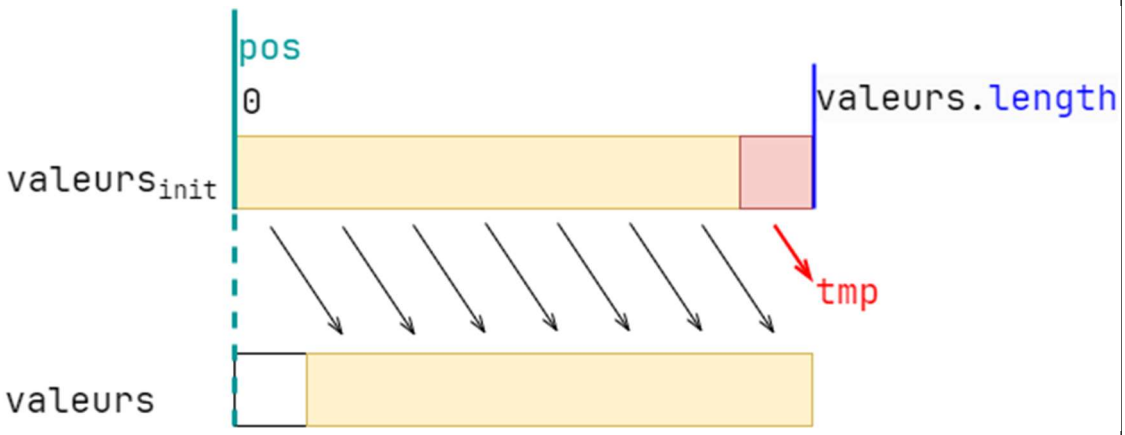
On lit directement dans le dessin qu'il faut initialiser

- `tmp` avec `valeurs[valeurs.length-1]`
- `pos` à `valeurs.length-1`

`GARDIEN`

Le gardien est une condition qui est vraie tant qu'il faut continuer la boucle. Donc, si on sait quand il faut s'arrêter, il suffit d'exprimer cette condition d'arrêt et d'en prendre la négation logique (autrement dit : « on continue tant qu'on ne s'arrête pas »).

Si on reprend notre dessin et qu'on le modifie pour représenter `pos` à sa position finale, on a ceci :



Il faut s'arrêter quand `pos <= 0` (le cas < 0 ne devrait absolument pas se produire, au risque de déborder). Le gardien de la boucle est donc `pos > 0`

`FIN`

Le schéma ci-dessus représente la situation à la fin de l'exécution de la boucle. Si vous faites un jeu des sept différences avec le résultat final attendu plus haut, vous remarquez qu'il n'y a qu'une seule différence : le contenu de `tmp` doit être recopié dans `valeurs[0].`

`CORPS DE LA BOUCLE`

Du dessin de la situation générale, j'essaie de faire progresser la zone en jaune. Cela consiste à :

- Décaler la bonne valeur dans `valeurs[pos]` (donc `valeurs[pos - 1]`)
- Faire progresser `pos` vers `0` en le décrémentant

`TERMINAISON`

`pos` est une valeur entière, positive quand le gardien est vrai qui décroit d'une itération à l'autre. On en conclut que la fonction se termine.

4. CODAGE = PROGRAMMER ET VALIDER LA SOLUTION EN JAVA