

P.O.O. niveau 1

TP n°4 : files et piles... en objet

Sommaire

1	Les piles	3
1.1	Des piles... et des tableaux	4
1.1.1	La classe Pile	5
1.1.2	La classe MainPile.....	8
1.2	Des piles... mais sans tableau !	8
1.2.1	La classe ElémentDePile.....	9
1.2.2	La classe MainElémentDePile	13
1.3	Des piles évoluées	14
1.3.1	La classe ElémentDePileEvoluée	16
1.3.2	La classe MainElémentDePileEvoluée.....	18
2	Les files	19
2.1	Des files... et des tableaux.....	20
2.1.1	La classe File	20
2.1.2	La classe MainFile.....	23
2.2	Des files... mais sans tableau !.....	23
2.2.1	La classe ElémentDeFile.....	24
2.2.2	La classe MainElémentDeFile	28
2.3	Des files encore plus évoluées	29
2.3.1	La classe ElémentDeFileEvoluée	31
2.3.2	La classe MainElémentDeFileEvoluée.....	34
3	Mise en pratique	35
3.1	Une partie de cartes (mise en œuvre des piles).....	35
3.1.1	Conception : le diagramme de classes UML.....	36
3.1.2	La classe Carte	37
3.1.3	La classe ElémentDeTas	38
3.1.4	La classe Belote.....	40
3.1.5	La classe MainPartie	45
3.2	Un système multitâche.....	45
3.2.1	Conception : le diagramme de classes UML.....	46
3.2.2	La classe Processus	46
3.2.3	La classe ElémentDeTourniquet.....	48
3.2.4	La classe Processeur	51
3.2.5	La classe MainSystèmeDExploitation.....	52

En pratique

Pour ce TP, vos classes seront, finalement, organisées en paquetages et sous-paquetages comme suit (cf. Figure 1, sous-paquetage **tp4** et son contenu) :

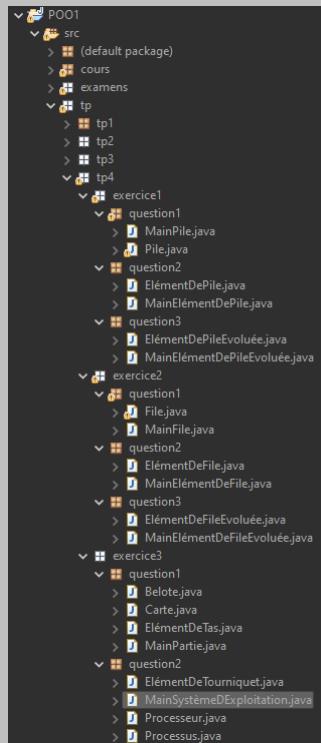


Figure 1. Organisation en paquetages des classes pour ce TP

1 Les piles

Une pile de données est une structure dont les éléments contiennent des données uniformes (*i.e.* de même type), sur laquelle on peut poser de nouveaux éléments (on ne peut donc pas compléter une pile « par-dessous ») et depuis le sommet de laquelle on peut prendre un élément (on ne peut donc pas retirer les éléments en bas de la pile sans avoir préalablement ôté les éléments situés au-dessus).

C'est donc ce que l'on appelle une structure LIFO (« Last In First Out ») :

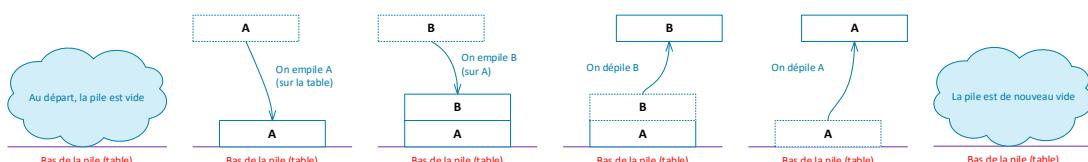


Figure 2. Structure de pile (LIFO)

Les principales opérations que l'on peut effectuer sur une pile sont donc l'ajout d'un élément (au sommet), que l'on appelle « empilement » et l'enlèvement d'un élément (depuis le sommet), que l'on appelle « dépilement » (cf. Figure 3).

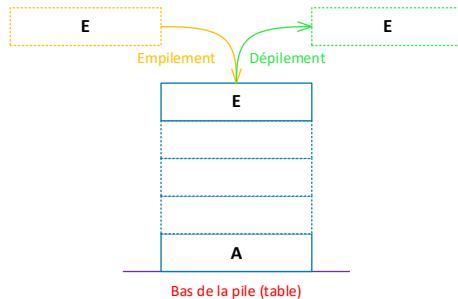


Figure 3. Empilement/dépilement avec une pile

On peut aussi fréquemment être amené à afficher l'ensemble des éléments d'une pile (usuellement pris de haut en bas), à tester si la pile est vide (*i.e.* si elle ne contient aucun élément) ou encore à tester si la pile est pleine (dans le cas où on lui aurait fixé un nombre maximal d'éléments).

On se propose d'écrire en Java de petits programmes de manipulation de piles : d'abord en utilisant une structure statique de tableau encapsulée dans un objet (cf. §1.1), puis en utilisant les possibilités dynamiques offertes par le paradigme objet (cf. §1.2 puis §1.3).

1.1 Des piles... et des tableaux

On va créer deux classes dans un paquetage `tp.tp4.exercice1.question1` (cf. Figure 4) :

- *Une classe Pile* : c'est elle qui va permettre de gérer les piles (éléments contenus dans une pile, nombre d'éléments contenus dans une pile, affichage d'une pile, tests sur une pile, ...),
- *Une classe MainPile* : cette classe a pour rôle de tester la mise en œuvre de piles instances de la classe `Pile`.

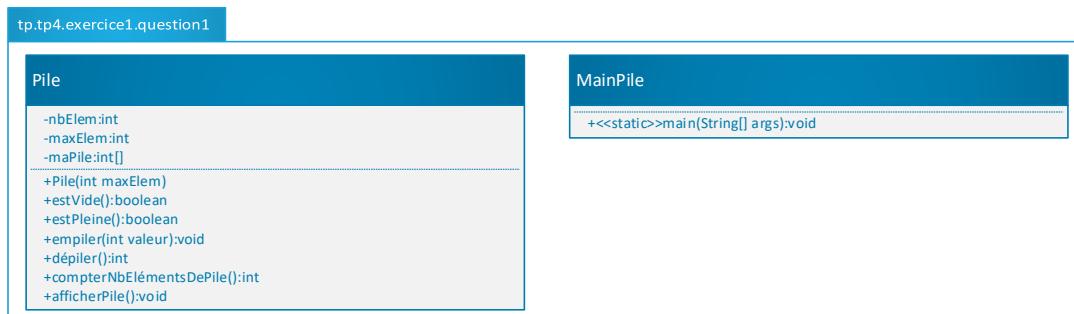


Figure 4. Paquetage et classes à créer pour la gestion de piles au moyen de tableaux

1.1.1 La classe Pile

La classe **Pile** est un « modèle » de piles gérées sous forme de tableaux (ici d'entiers)...

Résumé de la classe	Visibilité	Nom			Héritage explicite
	Publique	Pile			Aucun
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance			
Propriété(s)		Aucune classe n'est utilisée			
	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration
	Privée	Instance	Entier	nbElem	Pas d'initialisation à la déclaration
	Privée	Instance	Entier	maxElem	Pas d'initialisation à la déclaration
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom
	Constructeur	Publique	Instance	À vous de le dire	À vous de le dire
	Divers	Publique	Instance	Booléen	estVide()
	Divers	Publique	Instance	Booléen	estPleine()
	Divers	Publique	Instance	Aucune donnée retournée	empiler()
	Divers	Publique	Instance	Entier	dépiler()
	Divers	Publique	Instance	Entier	compterNbElémentsDePile()
	Divers	Publique	Instance	Aucune donnée retournée	afficherPile()

Tableau 1. Résumé de la classe **Pile**

1.1.1.1 Propriétés de la classe Pile

Les trois propriétés de cette classe sont les suivantes :

- Une propriété privée d'instance (de type entier) **nbElem** : son rôle est d'indiquer le nombre d'éléments actuellement présents dans la pile courante.
- Une propriété privée d'instance (de type entier) **maxElem** : son rôle est d'indiquer le nombre d'éléments que l'on pourra ranger au maximum dans la pile courante.
- Une propriété privée d'instance (de type tableau d'entiers) **maPile** : son rôle est de représenter le contenu de la pile courante (dont les éléments seront donc ici des entiers).

1.1.1.2 Constructeur de la classe Pile

En outre, cette classe contient un unique constructeur, public, ayant comme argument formel un entier. Ce constructeur fonctionne comme suit :

- Il initialise la propriété d'instance **maxElem** à la valeur de l'argument formel,
- Il initialise la propriété d'instance **nbElem** à 0 (au début, la pile est vide),
- Il instancie le tableau **maPile** comme un tableau d'entiers contenant **maxElem** cases (voir remarque ci-dessous),
- Il initialise enfin chaque case du tableau **maPile** à 0.



Remarque

En Java, les tableaux sont des objets (même si vous ne voyez pas forcément de nom de classe lors de la déclaration). Donc, comme tout objet, ils doivent être instanciés. Cela se fait au moyen du mot-clé **new** mais, dans ce cas, il n'est pas suivi d'une invocation de constructeur mais du type des éléments du tableau à créer et de leur nombre. Par exemple, pour instancier un tableau de 10 entiers : **new int[10]**.

Enfin, instancier un tableau crée bien le tableau mais n'initialise pas le contenu de chacune de ses cases. Il est donc souvent utile, après avoir instancié un tableau, de le parcourir pour initialiser chacune de ses cases.

1.1.1.3 Autres méthodes de la classe Pile

Cette classe déclare enfin les six méthodes suivantes :

- Une méthode publique d'instance **estVide()** : elle n'a pas d'argument formel et retourne un booléen afin d'indiquer si la pile courante est vide (dans ce cas elle retourne le booléen **true**) ou pas (elle retourne alors le booléen **false**).



Remarque

On sait que la pile courante est vide quand elle ne contient aucun élément. Or, vous avez défini une propriété qui indique en permanence le nombre actuel d'éléments contenus dans la pile courante. Il est donc facile de tester la valeur de cette propriété.

- Une méthode publique d'instance **estPleine()** : elle n'a pas d'argument formel et retourne un booléen afin d'indiquer si la pile courante est pleine (dans ce cas elle retourne le booléen **true**) ou pas (elle retourne alors le booléen **false**).



Remarque

De même, on sait que la pile courante est pleine quand elle contient un nombre d'éléments égal au nombre maximal d'éléments que l'on peut y stocker. Or, vous avez défini une propriété qui indique en permanence le nombre maximal d'éléments que peut contenir la pile courante et une autre propriété qui indique le nombre actuel d'éléments contenus dans la pile courante. Il est donc facile de tester la valeur de ces propriétés.

- *Une méthode publique d'instance **empiler()*** : elle a argument formel de type entier et ne retourne rien ; cette méthode teste si la pile courante est pleine ; si ça n'est pas le cas, elle rajoute au sommet de la pile courante l'élément dont la valeur a été fournie en argument et met à jour le nombre d'éléments actuellement contenus dans la pile courante (si la pile courante était initialement pleine, vous pouvez éventuellement afficher sur la console un message d'erreur indiquant cet état de fait).

**Remarque**

Pour tester si la pile courante est pleine, vous avez écrit une méthode dont c'est le rôle : utilisez-donc la !

- *Une méthode publique d'instance **dépiler()*** : elle n'a pas d'argument formel et retourne un entier ; cette méthode teste si la pile courante est vide ; si ça n'est pas le cas, elle retire de la pile courante l'élément situé à son sommet, met à jour le nombre d'éléments actuellement contenus dans la pile courante (si la pile courante était initialement vide, vous pouvez éventuellement afficher sur la console un message d'erreur indiquant cet état de fait) et retourne la valeur contenue dans l'élément déplié.

**Remarque**

Pour tester si la pile courante est vide, vous avez écrit une méthode dont c'est le rôle : utilisez-donc la !

- *Une méthode publique d'instance **compterNbElémentsDePile()*** : elle n'a pas d'argument formel et retourne un entier indiquant le nombre d'éléments actuellement placés dans la pile courante.

**Remarque**

Vous avez défini une propriété qui indique en permanence le nombre actuel d'éléments contenus dans la pile courante, utilisez donc la.

- *Une méthode publique d'instance **afficherPile()*** : elle n'a pas d'argument formel et ne retourne rien ; elle affiche à l'écran la valeur contenue dans chacun des éléments de la pile courante (depuis son sommet jusqu'à sa base).

1.1.2 La classe MainPile

La classe **MainPile** est une classe « principale » vouée à tester la classe **Pile** précédemment écrite...

Résumé de la classe	Visibilité	Nom	Héritage explicite				
	Publique	MainPile	Aucun héritage explicite				
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance					
	Pile	À vous de le dire					
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
		Aucune propriété dans cette classe					
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Programme principal	Publique	Classe	Aucune donnée renournée	main()	Un tableau de chaînes de caractères	Aucune

Tableau 2. Résumé de la classe **MainPile**

La classe **MainPile** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes :

- Elle instancie 2 fois la classe **Pile** afin de créer une pile de 20 éléments et une pile de 5 éléments, respectivement référencées depuis les variables **pile20** et **pile5**,
- Elle empile des éléments sur ces 2 piles (l'idée est d'en remplir au moins une des deux),
- Elle affiche le contenu des 2 piles, ainsi que leur nombre d'éléments courants,
- Elle dépile depuis ces 2 piles (jusqu'à en vider au moins une des deux),
- Elle affiche de nouveau le contenu des 2 piles, ainsi que leur nombre d'éléments courants.

1.2 Des piles... mais sans tableau !

On souhaite maintenant implémenter une structure de pile mais sans avoir à gérer de tableau (une bonne raison est qu'un tableau est forcément limité en nombre de cases alors que, en pratique, on souhaite souvent pouvoir gérer une pile « infinie », i.e. sans limite en nombre d'éléments). Réfléchissez donc à une structure de donnée permettant de gérer une pile sans tableau...



Astuce

Imaginez votre pile non pas comme un tout (i.e. comme une propriété d'une instance d'une classe **Pile**, comme dans la question précédente). Au contraire :

- Imaginez une pile comme un ensemble d'éléments de pile se référant les uns les autres (l'élément situé en haut de la pile référence l'élément situé immédiatement sous lui, qui référence l'élément situé immédiatement sous lui, ..., jusqu'à l'élément situé en bas de la pile qui ne référence personne),
- La pile est ainsi « identifiée depuis l'extérieur » par une référence vers l'élément situé à son sommet (puisque, à partir de lui, on peut accéder à tous les éléments situés dans la pile).

On va créer deux classes dans un paquetage **tp.tp4.exercice1.question2** (cf. Figure 6) :

- **Une classe ElémentDePile** : c'est elle qui va permettre de gérer les éléments qui constituent une pile et, par extension, les piles (puisque par le sommet d'une pile on accède en fait à toute la pile),
- **Une classe MainElémentDePile** : cette classe a pour rôle de tester la mise en œuvre de piles gérées « en tout objet » via des instances de la classe **ElémentDePile**.



Figure 5. Paquetage et classes à créer pour la gestion de piles « tout objet »

1.2.1 La classe ElémentDePile

La classe **ElémentDePile** est un « modèle » d'éléments constitutifs des piles, donc par extension de piles (ici d'entiers) gérées « en tout objet »...

Résumé de la classe	Visibilité	Nom			Héritage explicite		
	Publique	ElémentDePile			Aucun		
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance					
Propriété(s)		Aucune classe n'est utilisée					
	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
	Privée	Instance	Entier	donnée	Pas d'initialisation à la déclaration		
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Constructeur	Publique	Instance	À vous de le dire	À vous de le dire	Un entier	Aucune
	Assesseur (getter)	Publique	Instance	Entier	getDonnée()	Aucun	Aucune
	Divers	Publique	Instance	À vous de le dire	empiler()	Un entier	Aucune
	Divers	Publique	Instance	À vous de le dire	dépiler()	Aucun	Aucune
	Divers	Publique	Instance	Entier	compterNbElémentsDePile()	Aucun	Aucune
	Divers	Publique	Instance	Aucune donnée renournée	afficherElémentDePile()	Aucun	Aucune

Tableau 3. Résumé de la classe **ElémentDePile**

Puisqu'on va gérer des éléments de pile, l'idée est alors de voir une pile comme des éléments de pile dont chacun d'entre eux référence celui qui est juste sous lui dans la pile (le dernier élément de la pile, i.e. le plus bas, ne référençant alors personne puisqu'il n'y a rien sous lui, par définition).

La structure de la pile est alors la suivante (cf. Figure 6) :

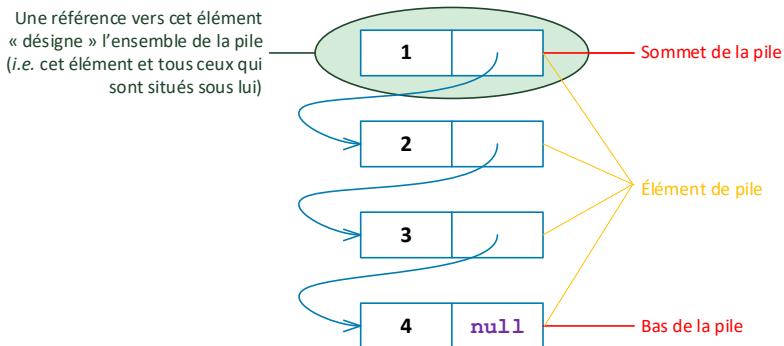


Figure 6. Représentation d'une pile en tout objet, avec des éléments de pile

Ainsi, une référence vers un élément de pile désigne par définition un élément appartenant à une pile et permet aussi d'accéder à tous les éléments qui sont sous lui.



Remarque

Notez que, avec une telle structuration, les méthodes `estPleine()` et `estVide()` deviennent obsolètes puisqu'elles n'ont plus de sens :

- Avec cette structure, une pile ne peut plus être pleine,
- Tester si une pile est vide revient à tester si la référence vers son sommet vaut `null`, ce qui ne peut se faire qu'à un endroit où les piles sont identifiées par des références vers leurs sommets (par exemple, mais pas exclusivement, dans un programme principal).

On ne trouvera donc pas ces deux méthodes ici...

1.2.1.1 Propriétés de la classe `ElémentDePile`

Les deux propriétés de cette classe sont les suivantes :

- Une propriété privée d'instance (de type entier) `donnée` : sa valeur est celle de la donnée stockée dans l'élément de pile courant.
- Une propriété privée d'instance (à vous de déterminer son type) `dessous` : le rôle de cette propriété est de référencer l'élément de pile situé immédiatement sous l'élément de pile courant (elle vaut donc `null` si l'élément de pile courant est le plus bas de la pile puisque, par définition, il n'y a alors aucun élément de pile sous lui).

1.2.1.2 Constructeur de la classe `ElémentDePile`

En outre, cette classe contient un unique constructeur, public, ayant comme argument formel un entier. Ce constructeur fonctionne comme suit :

- Il initialise la propriété `donnée` de l'élément de pile courant avec la valeur de l'argument formel,
- Il initialise la propriété `dessous` de l'élément de pile courant avec la référence `null` (quand on crée un élément de pile, il n'est encore posé sur aucune pile donc on considère qu'il forme une pile à lui tout seul, pile dont il est le seul élément, i.e. dont il est à la fois le sommet et l'élément le plus bas, donc avec rien sous lui).

1.2.1.3 Assesseur de la classe `ElémentDePile`

Cette classe déclare également un unique assesseur : il s'agit d'un getter sur la propriété **donnée**. Cet assesseur est une méthode publique d'instance nommée `getDonnée()`, sans argument formel et retournant un entier qui est la valeur de la donnée située dans l'élément de pile courant.

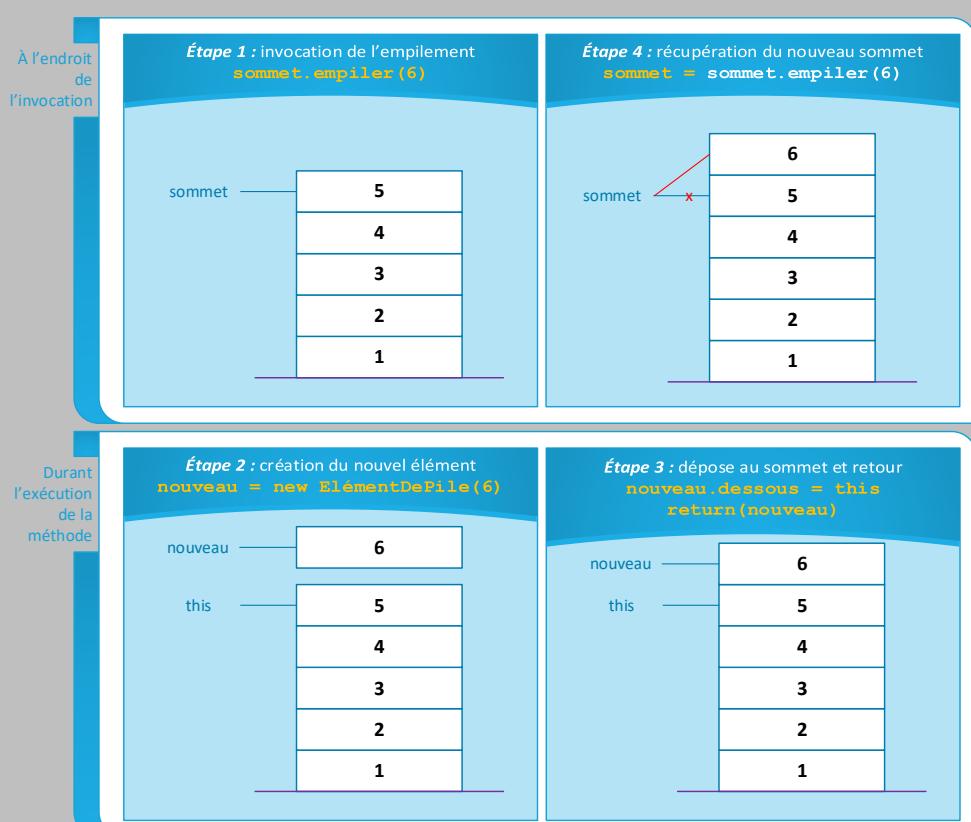
1.2.1.4 Autres méthodes de la classe `ElémentDePile`

Cette classe déclare enfin les quatre méthodes suivantes :

- Une méthode publique d'instance `empiler()` : elle un argument formel de type entier et retourne un élément de pile ; elle crée un nouvel élément de pile encapsulant l'argument formel puis pose ce nouvel élément de pile sur l'élément de pile courant et retourne enfin la référence vers le nouveau sommet de la pile (*i.e.* vers le nouvel élément ainsi déposé au sommet).

Remarque

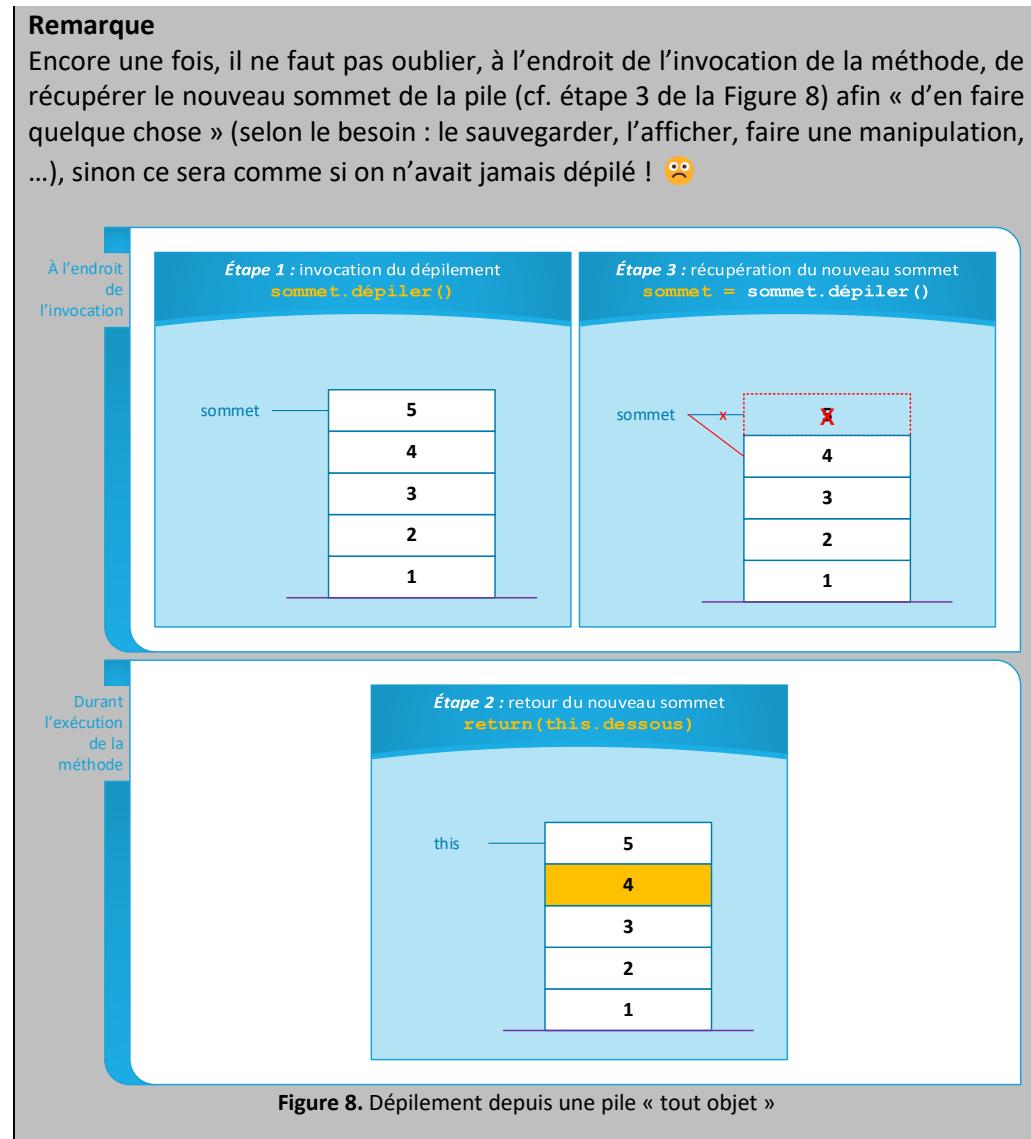
Il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer le nouveau sommet de la pile (cf. étape 4 de la Figure 7) afin « d'en faire quelque chose » (selon le besoin : le sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais empilé ! 😕



- Une méthode publique d'instance `dépiler()` : elle n'a pas d'argument formel et retourne un élément de pile ; elle retourne la référence vers le nouveau sommet de la pile, qui est donc l'élément de pile situé juste sous l'élément de pile courant (l'ancien sommet ne sera donc plus référencé).

Remarque

Encore une fois, il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer le nouveau sommet de la pile (cf. étape 3 de la Figure 8) afin « d'en faire quelque chose » (selon le besoin : le sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais déplié ! 😊



- Une méthode publique d'instance `compterNbElémentsDePile()` : elle n'a pas d'argument formel et retourne un entier : s'il existe un élément de pile sous l'élément de pile courant, la méthode retourne la somme de 1 (ce qui compte l'élément de pile courant) et du résultat de l'invocation de la méthode `compterNbElémentsDePile()` sur l'élément de pile situé sous l'élément de pile courant, sinon (*i.e.* si l'élément de pile courant est le plus bas de sa pile) elle retourne 1 (ce qui compte l'élément de pile courant).



Remarque

Compter le nombre d'éléments d'une pile revient à compter l'élément courant (qui compte donc pour 1) et à ajouter cela au nombre d'éléments de la pile qui se trouve sous l'élément courant...

- Une méthode publique d'instance **afficherElémentDePile()** : elle n'a pas d'argument formel et ne retourne rien ; elle affiche à l'écran la donnée contenue dans l'élément de pile courant ; s'il existe un élément de pile sous l'élément de pile courant, la méthode **afficherElémentDePile()** est alors invoquée sur l'élément de pile qui est situé juste sous l'élément courant.



Remarque

Afficher le contenu d'une pile revient à afficher l'élément courant puis à afficher la pile qui est sous l'élément courant...

1.2.2 La classe MainElémentDePile

La classe **MainElémentDePile** est une classe « principale » vouée à tester la classe **ElémentDePile** précédemment écrite...

Résumé de la classe	Visibilité	Nom	Héritage explicite		
	Publique	MainElémentDePile	Aucun héritage explicite		
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance			
	ElémentDePile	À vous de le dire			
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration
		Aucune propriété dans cette classe			
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Argument(s) formel(s)
	Programme principal	Publique	Classe	Aucune donnée renournée	main()
					Un tableau de chaînes de caractères
					Exception(s) pouvant être levée(s)
					Aucune

Tableau 4. Résumé de la classe **MainElémentDePile**

La classe **MainElémentDePile** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes :

- Elle crée 2 piles, chacune référencée par l'élément de pile situé à son sommet, respectivement depuis une variable **pileA** et depuis une variable **pileB**,
- Elle empile des éléments sur ces 2 piles,
- Elle affiche le contenu des 2 piles et indique, pour chaque pile, le nombre d'éléments de pile qu'elle contient à ce moment-là.
- Elle dépile depuis ces 2 piles (jusqu'à en vider au moins une des deux),
- Elle affiche de nouveau le contenu des 2 piles et indique, pour chaque pile, le nombre d'éléments de pile qu'elle contient à ce moment-là.



Remarque

Vous risquez, à tout instant, de vous retrouver avec le message d'erreur « par excellence » en Java (cf. Figure 9), notamment si vous avez déplié vos piles entièrement puis avez tout de même cherché à accéder à un de leurs membres :

```
Dépilement de la valeur 1 du sommet de la pile.
Valeur dépiler depuis pile B : 2
Dépilement de la valeur 2 du sommet de la pile.
Exception in thread "main" java.lang.NullPointerException
at TP.TP4.Exercice1.Question2.MainElementDePile.main(MainElementDePile.java:36)
```

Figure 9. Message d'erreur le plus classique en Java : `java.lang.NullPointerException`

Ce message d'erreur s'affiche quand vous cherchez à accéder à un membre (propriété ou méthode) d'un objet alors que cet objet est en fait la référence `null` ! C'est donc à vous donc de vous assurer aux moments opportuns que vous n'essayez pas d'accéder à un membre de la référence `null`. Pour cela, vous pouvez par exemple écrire un test comme suit :

```
if(pileA != null)
    pileA.afficherElémentDePile();
```

1.3 Des piles évoluées

Pour pallier le problème de la manipulation d'une pile vide (*i.e.* d'une référence `null`), ce qui pose régulièrement des problèmes (comme on l'a vu à la question précédente et comme indiqué dans la dernière remarque du §1.2), on se propose de faire évoluer notre structure comme suit : l'idée est de « déconnecter » les notions de « pile vide » et de « référence `null` ». Pour cela, on va introduire un élément de pile particulier qui sera toujours situé tout en bas d'une pile (cf. Figure 10). Cet élément sera particulier car il sera le seul, parmi tous les éléments de la pile évoluée à laquelle il appartient, à contenir une donnée ayant pour valeur la référence `null`.

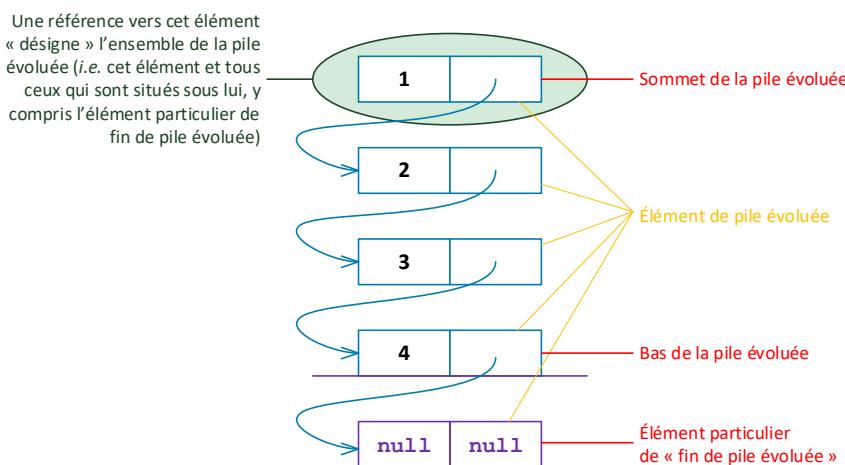


Figure 10. Structure de pile évoluée, toujours en tout objet



Attention

Puisque l'élément particulier est « repéré » par le fait que la donnée qu'il encapsule est la référence **null**, cela implique que la donnée encapsulée dans un élément de pile évoluée ne peut pas être une donnée de type primitif : c'est forcément un objet. Ainsi, si vous souhaitez tout de même encapsuler des données de type primitif, par exemple des entiers, il faudra passer par les classes wrappers.

À partir de là, on va devoir considérer au moins les points suivants lors de l'écriture de la classe :

- À sa création, une pile évoluée contient uniquement cet élément particulier (cf. Figure 11),
- Une pile évoluée est considérée comme « vide » quand elle ne contient que cet élément particulier ; il faut donc pouvoir indiquer si une pile évoluée est vide en prenant cela en compte (cf. Figure 11),
- Un dépilement depuis une pile évoluée « vide » ne fait rien : cela renvoie la pile évoluée telle quelle (*i.e.* vide, donc en fait uniquement l'élément particulier).

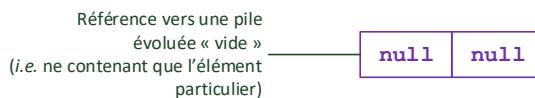


Figure 11. Structure d'une pile évoluée vide

On va créer deux classes dans un paquetage **tp.tp4.exercice1.question3** (cf. Figure 12) :

- Une classe **ElémentDePileEvoluée** : c'est elle qui va permettre de gérer les éléments qui constituent une pile évoluée et, par extension, les piles évoluées (puisque par le sommet d'une pile évoluée on accède en fait à toute la pile évoluée),
- Une classe **MainElémentDePileEvoluée** : cette classe a pour rôle de tester la mise en œuvre de piles évoluées *via* des instances de la classe **ElémentDePileEvoluée**.



Figure 12. Paquetage et classes à créer pour la gestion de piles évoluées

1.3.1 La classe ElémentDePileEvoluée

La classe **ElémentDePileEvoluée** est un « modèle » d'éléments constituants les piles évoluées...

Résumé de la classe	Visibilité	Nom	Héritage explicite				
	Publique	ElémentDePileEvoluée	Aucun				
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance					
Propriété(s)	Intégrer	À vous de le dire					
	Privée	Instance	Integer	donnée	Pas d'initialisation à la déclaration		
	Privée	Instance	À vous de le dire	dessous	Pas d'initialisation à la déclaration		
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire	Un entier (sous forme d'objet)	Aucune
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire	Aucun	Aucune
	Assesseur (méthode)	Publique	Classe	À vous de le dire	créerPileVide()	Aucun	Aucune
	Assesseur (getter)	Publique	Instance	Integer	getDonnée()	Aucun	Aucune
	Divers	Publique	Instance	Booléen	estVide()	Aucun	Aucune
	Divers	Publique	Instance	À vous de le dire	empiler()	Un entier (sous forme d'objet)	Aucune
	Divers	Publique	Instance	À vous de le dire	dépiler()	Aucun	Aucune
	Divers	Publique	Instance	Entier	compterNbEléments DePileEvoluée()	Aucun	Aucune
	Divers	Publique	Instance	Aucune donnée renournée	afficherElément DePileEvoluée()	Aucun	Aucune

Tableau 5. Résumé de la classe **ElémentDePileEvoluée**

1.3.1.1 Propriétés de la classe ElémentDePileEvoluée

Les deux propriétés de cette classe sont les suivantes :

- Une propriété privée d'instance (de type entier, mais sous forme d'objet) **donnée** : la valeur qu'elle encapsule est celle de la donnée stockée dans l'élément courant de la pile évoluée.
- Une propriété privée d'instance (à vous de déterminer son type) **dessous** : le rôle de cette propriété est de référencer l'élément de la pile évoluée situé immédiatement sous l'élément courant de la pile évoluée (elle vaut donc **null** si l'élément courant de la pile évoluée est l'élément particulier de fin de pile évoluée).

1.3.1.2 Constructeurs de la classe ElémentDePileEvoluée

En outre, cette classe contient deux constructeurs :

- Un constructeur privé ayant comme argument formel un entier (sous forme d'objet) :
 - Il initialise la propriété **donnée** de l'élément courant avec l'argument formel,
 - Il initialise la propriété **dessous** de l'élément courant avec la référence **null** (l'élément courant de la pile évoluée n'est donc à ce stade posé sur rien, même pas sur l'élément particulier de fin de pile évoluée).
- Un constructeur privé n'ayant pas d'argument formel : il invoque le constructeur précédent avec comme argument effectif la référence **null** (donc, l'invocation du présent constructeur n'a qu'un rôle : celui de créer un élément particulier de fin de pile évoluée).

1.3.1.3 Assesseurs de la classe *ElémentDePileEvoluée*

Cette classe déclare également deux assesseurs :

- *Un assesseur permettant d'accéder au constructeur privé sans argument formel afin de créer l'élément particulier de fin de pile évoluée* : c'est une méthode publique de classe nommée **créerPileVide()**, sans argument formel et dont le rôle est de créer et retourner un nouvel élément de fin de pile évoluée (donc une pile évoluée vide).
- *Un assesseur permettant d'accéder à la valeur de la propriété donnée* : c'est une méthode publique d'instance nommée **getDonnée()**, sans argument formel et retournant un entier qui est la valeur de la donnée située dans l'élément courant de la pile évoluée.

1.3.1.4 Autres méthodes de la classe *ElémentDePileEvoluée*

Cette classe déclare enfin les cinq méthodes suivantes :

- *Une méthode publique d'instance **estVide()*** : elle n'a pas d'argument formel et retourne un booléen afin d'indiquer si la pile évoluée courante est vide (dans ce cas elle retourne le booléen **true**) ou pas (elle retourne alors le booléen **false**).



Remarque

On sait que la pile évoluée courante est vide quand elle ne contient que l'élément particulier de fin de pile évoluée (*i.e.* le seul élément de la pile évoluée dont la donnée est la référence **null**).

- *Une méthode publique d'instance **empiler()*** : elle a un argument formel de type entier (sous forme d'objet) et retourne un élément de pile évoluée ; elle crée un nouvel élément de pile évoluée (qui sera le nouveau sommet de la pile évoluée à la fin de l'empilement) encapsulant la valeur passée en argument puis pose ce nouvel élément de pile évoluée sur l'élément courant de la pile évoluée et retourne enfin la référence vers le nouveau sommet de la pile évoluée.



Remarque

Comme précédemment (cf. §1.2.1.4), il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer le nouveau sommet de la pile évoluée afin « d'en faire quelque chose » (selon le besoin : le sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais empilé ! 😞

- *Une méthode publique d'instance **dépiler()*** : elle n'a pas d'argument formel et retourne un élément de pile évoluée ; elle retourne la référence vers le nouveau sommet de la pile évoluée, qui est donc l'élément de la pile évoluée situé juste sous l'élément courant de la pile évoluée (l'ancien sommet ne sera donc plus référencé).



Remarque

Encore une fois, et comme précédemment (cf. §1.2.1.4), il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer le nouveau sommet de la pile afin « d'en faire quelque chose » (selon le besoin : le sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais déplié ! 😞

- Une méthode publique d'instance `compterNbElémentsDePileEvoluée()` : elle n'a pas d'argument formel et retourne un entier : s'il existe un élément de pile évoluée sous l'élément courant de la pile évoluée, la méthode retourne la somme de 1 (ce qui compte l'élément courant de la pile évoluée) et du résultat de l'invocation de la méthode sur l'élément de pile évoluée situé sous l'élément courant de la pile évoluée, sinon (*i.e.* si l'élément courant de la pile évoluée est l'élément particulier de fin de pile évoluée) elle retourne 0.



Remarque

Comme précédemment (cf. §1.2.1.4), compter le nombre d'éléments d'une pile évoluée revient à compter l'élément courant (qui compte donc pour 1) et à ajouter cela au nombre d'éléments de la pile évoluée qui se trouve sous l'élément courant...

Sur cette structure de pile évoluée, le test d'arrêt (qui permet de savoir si on est en bas de la pile évoluée ou pas) change cependant !

- Une méthode publique d'instance `afficherElémentDePileEvoluée()` : elle n'a pas d'argument formel et ne retourne rien ; elle affiche à l'écran la donnée contenue dans l'élément courant de la pile évoluée ; s'il existe un élément de pile évoluée sous l'élément courant de la pile évoluée, la méthode `afficherElémentDePileEvoluée()` est alors invoquée sur l'élément de pile évoluée qui est situé juste sous l'élément courant de la pile évoluée.



Remarque

Comme précédemment (cf. §1.2.1.4), afficher le contenu d'une pile évoluée revient à afficher l'élément courant puis à afficher la pile évoluée qui est sous l'élément courant... **Sur cette structure de pile évoluée, le test d'arrêt (qui permet de savoir si on est en bas de la pile évoluée ou pas) change cependant !**

1.3.2 La classe MainElémentDePileEvoluée

La classe `MainElémentDePileEvoluée` est une classe « principale » vouée à tester la classe `ElémentDePileEvoluée` précédemment écrite...

Résumé de la classe	Visibilité	Nom	Héritage explicite				
	Publique	<code>MainElémentDePileEvoluée</code>	Aucun héritage explicite				
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance					
	<code>ElémentDePileEvoluée</code>	À vous de le dire					
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
	Aucune propriété dans cette classe						
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Programme principal	Publique	Classe	Aucune donnée renournée	<code>main()</code>	Un tableau de chaînes de caractères	Aucune

Tableau 6. Résumé de la classe `MainElémentDePileEvoluée`

La classe **MainElémentDePileEvoluée** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes :

- Elle crée 2 piles évoluées, chacune référencée par l'élément de pile évoluée situé à son sommet, respectivement depuis une variable **pileA** et depuis une variable **pileB**,
- Elle empile des éléments sur ces 2 piles évoluées,
- Elle affiche le contenu des 2 piles évoluées et indique, pour chaque pile évoluée, le nombre d'éléments de pile évoluée qu'elle contient à ce moment-là.
- Elle dépile depuis ces 2 piles évoluées (jusqu'à en vider au moins une des deux),
- Elle affiche de nouveau le contenu des 2 piles évoluées et indique, pour chaque pile évoluée, le nombre d'éléments de pile évoluée qu'elle contient à ce moment-là.

2 Les files

Une file est une structure dont les éléments contiennent des données uniformes (*i.e.* de même type), qui est orientée (elle a une « tête » et une « queue »), à laquelle on peut ajouter des données, mais uniquement en queue et de laquelle on peut retirer des données, mais uniquement en tête.

C'est donc ce que l'on appelle une structure FIFO (« First In First Out ») :

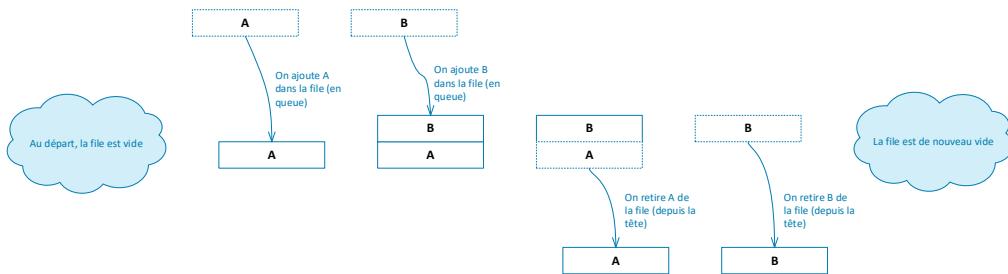


Figure 13. Structure de file (FIFO)

Les principales opérations que l'on peut effectuer sur une file sont donc l'ajout d'un élément à sa queue et le retrait d'un élément à sa tête (cf. Figure 14).

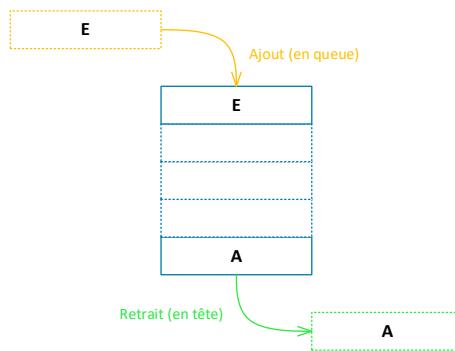


Figure 14. Ajout (en queue)/retrait (en tête) avec une file

On peut aussi fréquemment être amené à afficher l'ensemble des éléments d'une file (usuellement pris de la tête vers la queue), à tester si la file est vide (*i.e.* si elle ne contient aucun élément) ou encore à tester si la file est pleine (dans le cas où on lui aurait fixé un nombre maximal d'éléments).

On se propose d'écrire en Java de petits programmes de manipulation de files : d'abord en utilisant une structure statique de tableau encapsulée dans un objet (cf. §2.1), puis en utilisant les possibilités dynamiques offertes par le paradigme objet (cf. §2.2 puis §2.3).

2.1 Des files... et des tableaux

On va créer deux classes dans un paquetage `tp.tp4.exercice2.question1` (cf.) :

- *Une classe File* : c'est elle qui va permettre de gérer les files (éléments contenus dans une file, nombre d'éléments contenus dans une file, affichage d'une file, tests sur une file, ...),
- *Une classe MainFile* : cette classe a pour rôle de tester la mise en œuvre de files instances de la classe `File`.

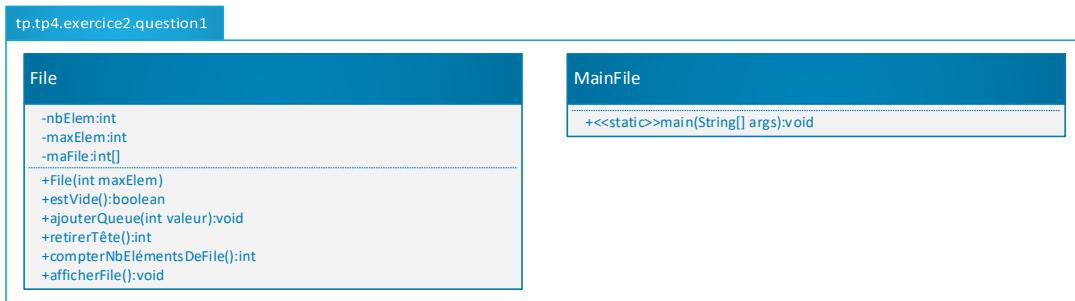


Figure 15. Paquetage et classes à créer pour la gestion de files au moyen de tableaux

2.1.1 La classe File

La classe `File` est un « modèle » de files gérées sous forme de tableaux (ici d'entiers)...

Résumé de la classe	Visibilité	Nom			Héritage explicite		
	Publique	<code>File</code>			Aucun		
Classe(s) utilisée(s)	Nom		Paquetage d'appartenance				
Propriété(s)	Aucune classe n'est utilisée						
	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
	Privée	Instance	Entier	<code>nbElem</code>	Pas d'initialisation à la déclaration		
	Privée	Instance	Entier	<code>maxElem</code>	Pas d'initialisation à la déclaration		
	Privée	Instance	Tableau d'entiers	<code>maFile</code>	Pas d'initialisation à la déclaration		
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
Constructeur	Constructeur	Publique	Instance	À vous de le dire	À vous de le dire	Un entier	Aucune
	Divers	Publique	Instance	Booléen	<code>estVide()</code>	Aucun	Aucune
Divers	Divers	Publique	Instance	Booléen	<code>estPleine()</code>	Aucun	Aucune
	Divers	Publique	Instance	Aucune donnée renournée	<code>ajouterQueue()</code>	Un entier	Aucune
	Divers	Publique	Instance	Entier	<code>retirerTête()</code>	Aucun	Aucune
	Divers	Publique	Instance	Entier	<code>compterNbElémentsDeFile()</code>	Aucun	Aucune
	Divers	Publique	Instance	Aucune donnée renournée	<code>afficherFile()</code>	Aucun	Aucune

Tableau 7. Résumé de la classe `File`

2.1.1.1 Propriétés de la classe File

Les trois propriétés de cette classe sont les suivantes :

- Une propriété privée d'instance (de type entier) **nbElem** : son rôle est d'indiquer le nombre d'éléments actuellement présents dans la file courante.
- Une propriété privée d'instance (de type entier) **maxElem** : son rôle est d'indiquer le nombre d'éléments que l'on pourra ranger au maximum dans la file courante.
- Une propriété privée d'instance (de type tableau d'entiers) **maFile** : son rôle est de représenter le contenu de la file courante (dont les éléments seront donc ici des entiers).

2.1.1.2 Constructeur de la classe File

En outre, cette classe contient un unique constructeur, public, ayant comme argument formel un entier. Ce constructeur fonctionne comme suit :

- Il initialise la propriété d'instance **maxElem** à la valeur de l'argument formel,
- Il initialise la propriété d'instance **nbElem** à 0 (au début, la file est vide),
- Il initialise le tableau **maFile** comme un tableau d'entiers contenant **maxElem** cases (voir remarque ci-dessous),
- Il initialise enfin chaque case du tableau **maFile** à 0.

Remarque

En Java, les tableaux sont des objets (même si vous ne voyez pas forcément de nom de classe lors de la déclaration). Donc, comme tout objet, ils doivent être instanciés. Cela se fait au moyen du mot-clé **new** mais, dans ce cas, il n'est pas suivi d'une invocation de constructeur mais du type des éléments du tableau à créer et de leur nombre. Par exemple, pour instancier un tableau de 10 entiers : **new int[10]**.

Enfin, instancier un tableau crée bien le tableau mais n'initialise pas le contenu de chacune de ses cases. Il est donc souvent utile, après avoir instancié un tableau, de le parcourir pour initialiser chacune de ses cases.

2.1.1.3 Autres méthodes de la classe File

Cette classe déclare enfin les six méthodes suivantes :

- Une méthode publique d'instance **estVide()** : elle n'a pas d'argument formel et retourne un booléen afin d'indiquer si la file courante est vide (dans ce cas elle retourne le booléen **true**) ou pas (elle retourne alors le booléen **false**).

Remarque

On sait que la file courante est vide quand elle ne contient aucun élément. Or, vous avez défini une propriété qui indique en permanence le nombre actuel d'éléments contenus dans la file courante. Il est donc facile de tester la valeur de cette propriété.



- Une méthode publique d'instance **estPleine()** : elle n'a pas d'argument formel et retourne un booléen afin d'indiquer si la file courante est pleine (dans ce cas elle retourne le booléen **true**) ou pas (elle retourne alors le booléen **false**).

**Remarque**

De même, on sait que la file courante est pleine quand elle contient un nombre d'éléments égal au nombre maximal d'éléments que l'on peut y stocker. Or, vous avez défini une propriété qui indique en permanence le nombre maximal d'éléments que peut contenir la file courante et une autre propriété qui indique le nombre actuel d'éléments contenus dans la file courante. Il est donc facile de tester la valeur de ces propriétés.

- Une méthode publique d'instance **ajouterQueue()** : elle a un argument formel de type entier et ne retourne rien ; cette méthode teste si la file courante est pleine ; si ça n'est pas le cas, elle rajoute en queue de file courante l'élément dont la valeur a été fournie en argument et met à jour le nombre d'éléments actuellement contenus dans la file courante (si la file courante était pleine, vous pouvez éventuellement afficher sur la console un message d'erreur indiquant cet état de fait).

**Remarque**

Pour tester si la file courante est pleine, vous avez écrit une méthode dont c'est le rôle : utilisez-donc la !

- Une méthode publique d'instance **retirerTête()** : elle n'a pas d'argument formel et retourne un entier ; cette méthode teste si la file courante est vide ; si ça n'est pas le cas, elle retire de la file courante l'élément situé à sa tête, met à jour le nombre d'éléments actuellement contenus dans la file courante (si la file courante était vide, vous pouvez éventuellement afficher sur la console un message d'erreur indiquant cet état de fait) et retourne la valeur contenue dans l'élément que l'on a retiré de la tête.
- Une méthode publique d'instance **compterNbElementsDeFile()** : elle n'a pas d'argument formel et retourne un entier indiquant le nombre d'éléments actuellement placés actuellement dans la file courante.

**Remarque**

Vous avez défini une propriété qui indique en permanence le nombre actuel d'éléments contenus dans la file courante, utilisez donc la.

- Une méthode publique d'instance **afficherFile()** : elle n'a pas d'argument formel et ne retourne rien ; elle affiche à l'écran la valeur contenue dans chacun des éléments de la file courante (depuis sa tête jusqu'à sa queue).

2.1.2 La classe MainFile

La classe **MainFile** est une classe « principale » vouée à tester la classe **File** précédemment écrite...

Résumé de la classe	Visibilité	Nom	Héritage explicite				
	Publique	MainFile	Aucun héritage explicite				
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance					
	File	À vous de le dire					
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
		Aucune propriété dans cette classe					
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Programme principal	Publique	Classe	Aucune donnée renournée	main()	Un tableau de chaînes de caractères	Aucune

Tableau 8. Résumé de la classe **MainFile**

La classe **MainFile** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes :

- Elle instancie 2 fois la classe **File** afin de créer une file de 20 éléments et une file de 5 éléments, respectivement référencées depuis les variables **file20** et **file5**,
- Elle ajoute des éléments en queue de ces 2 files (l'idée est d'en remplir au moins une des deux),
- Elle affiche le contenu des 2 files, ainsi que leur nombre d'éléments courants,
- Elle retire des éléments de la tête de ces 2 files (jusqu'à en vider au moins une des deux),
- Elle affiche de nouveau le contenu des 2 files, ainsi que leur nombre d'éléments courants.

2.2 Des files... mais sans tableau !

On souhaite maintenant implémenter une structure de file mais sans avoir à gérer de tableau (une bonne raison est qu'un tableau est forcément limité en nombre de cases alors que, en pratique, on souhaite souvent pouvoir gérer une file « infinie », *i.e.* sans limite en nombre d'éléments). Réfléchissez donc à une structure de donnée permettant de gérer une file sans tableau...



Astuce

Imaginez votre file non pas comme un tout (*i.e.* comme une propriété d'une instance d'une classe **File**, comme dans la question précédente). Au contraire :

- Imaginez une file comme un ensemble d'éléments de file se référant les uns les autres (l'élément situé en tête de la file référence l'élément situé immédiatement derrière lui, qui référence l'élément situé immédiatement derrière lui, ..., jusqu'à l'élément situé en queue de file qui ne référence personne),
- La file est ainsi « identifiée depuis l'extérieur » par une référence vers l'élément situé à sa tête (puisque, à partir de lui, on peut accéder à tous les éléments situés dans la file).

On va créer deux classes dans un paquetage **tp.tp4.exercice2.question2** (cf. Figure 16) :

- **Une classe ElémentDeFile** : c'est elle qui va permettre de gérer les éléments qui constituent une file et, par extension, les files (puisque par la tête d'une file on accède en fait à toute la file),
- **Une classe MainElémentDeFile** : cette classe a pour rôle de tester la mise en œuvre de files gérées « en tout objet » via des instances de la classe **ElémentDeFile**.

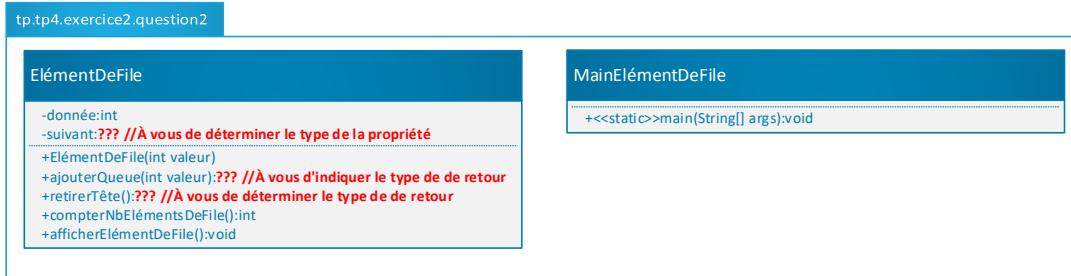


Figure 16. Paquetage et classes à créer pour la gestion de files « tout objet »

2.2.1 La classe ElémentDeFile

La classe **ElémentDeFile** est un « modèle » d'éléments constitutifs des files, donc par extension de files (ici d'entiers) gérées « en tout objet »...

Résumé de la classe	Visibilité	Nom			Héritage explicite		
	Publique	ElémentDeFile			Aucun		
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance					
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
	Privée	Instance	Entier	donnée	Pas d'initialisation à la déclaration		
	Privée	Instance	À vous de le dire	suivant	Pas d'initialisation à la déclaration		
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Constructeur	Publique	Instance	À vous de le dire	À vous de le dire	Un entier	Aucune
	Assesseur (getter)	Publique	Instance	Entier	getDonnée()	Aucun	Aucune
	Divers	Publique	Instance	À vous de le dire	ajouterQueue()	Un entier	Aucune
	Divers	Publique	Instance	À vous de le dire	retirerTête()	Aucun	Aucune
	Divers	Publique	Instance	Entier	compterNbElémentsDeFile()	Aucun	Aucune
	Divers	Publique	Instance	Aucune donnée renournée	afficherElémentDeFile()	Aucun	Aucune

Tableau 9. Résumé de la classe **ElémentDeFile**

Puisqu'on va gérer des éléments de file, l'idée est alors de voir une file comme des éléments de file dont chacun d'entre eux référence celui qui est juste après lui dans la file (le dernier élément de la file, i.e. le plus en queue, ne référençant alors personne puisqu'il n'y a rien après lui, par définition).

La structure de la file est alors la suivante (cf. Figure 17) :

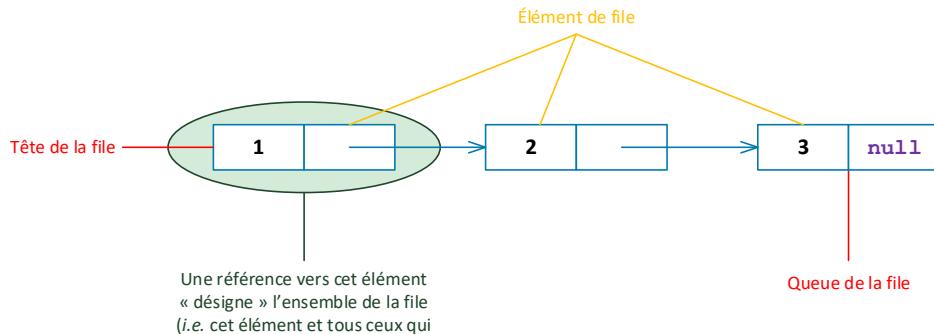


Figure 17. Représentation d'une file en tout objet, avec des éléments de file

Ainsi, une référence vers un élément de file désigne par définition un élément appartenant à une file et permet aussi d'accéder à tous les éléments qui sont après lui.



Remarque

Notez que, avec une telle structuration, les méthodes `estPleine()` et `estVide()` deviennent obsolètes puisqu'elles n'ont plus de sens :

- Avec cette structure, une file ne peut plus être pleine,
- Tester si une file est vide revient à tester si la référence vers sa tête vaut `null`, ce qui ne peut se faire qu'à un endroit où les files sont identifiées par des références vers leurs têtes (par exemple, mais pas exclusivement, dans un programme principal).

On ne trouvera donc pas ces deux méthodes ici...

2.2.1.1 Propriétés de la classe `ElémentDeFile`

Les deux propriétés de cette classe sont les suivantes :

- Une propriété privée d'instance (de type entier) **donnée** : sa valeur est celle de la donnée stockée dans l'élément de file courant.
- Une propriété privée d'instance (à vous de déterminer son type) **suivant** : le rôle de cette propriété est de référencer l'élément de file situé immédiatement après l'élément de file courant (elle vaut donc `null` si l'élément de file courant est le dernier de la file puisque, par définition, il n'y a alors aucun élément de file après lui).

2.2.1.2 Constructeur de la classe `ElémentDeFile`

En outre, cette classe contient un unique constructeur, public, ayant comme argument formel un entier. Ce constructeur fonctionne comme suit :

- Il initialise la propriété **donnée** de l'élément de file courant avec la valeur de l'argument formel,
- Il initialise la propriété **suivant** de l'élément de file courant avec la référence `null` (quand on crée un élément de file, il n'est encore raccroché en queue d'aucune file donc on considère qu'il forme une file à lui tout seul, file dont il est le seul élément, i.e. dont il est à la fois la tête et la queue, donc avec rien après lui).

2.2.1.3 Assesseur de la classe `ElémentDeFile`

Cette classe déclare également un unique assesseur : il s'agit d'un getter sur la propriété **donnée**. Cet assesseur est une méthode publique d'instance nommée `getDonnée()`, sans argument formel et retournant un entier qui est la valeur de la donnée située dans l'élément de file courant.

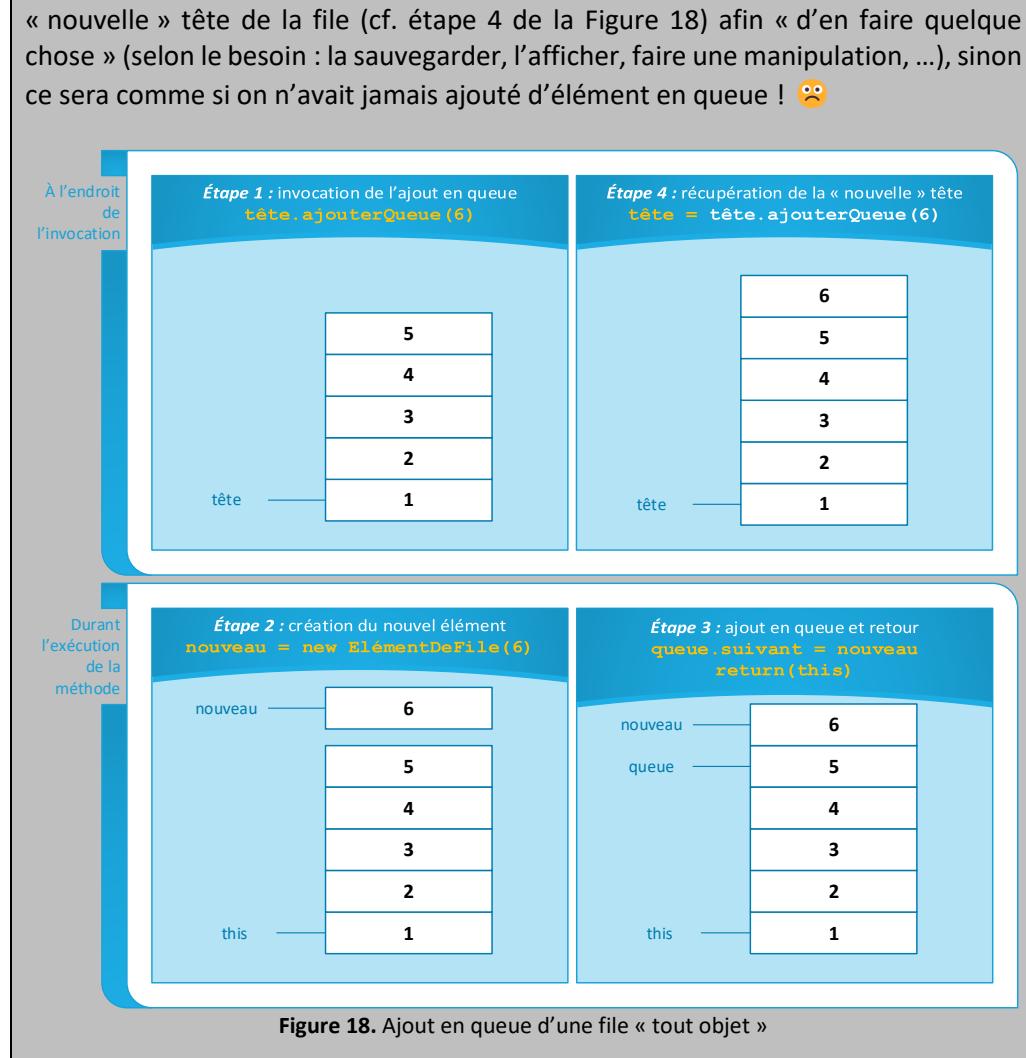
2.2.1.4 Autres méthodes de la classe `ElémentDeFile`

Cette classe déclare enfin les quatre méthodes suivantes :

- Une méthode publique d'instance `ajouterQueue()` : elle a un argument formel de type entier et retourne un élément de file ; elle crée un nouvel élément de file encapsulant l'argument formel puis parcours la file pour arriver jusqu'à son élément de queue afin de raccrocher après cette queue le nouvel élément créé puis, enfin, de retourner la référence vers la tête de file (*a priori* inchangée).

Remarque

Il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer la « nouvelle » tête de la file (cf. étape 4 de la Figure 18) afin « d'en faire quelque chose » (selon le besoin : la sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais ajouté d'élément en queue ! 😊



- Une méthode publique d'instance `retirerTête()` : elle n'a pas d'argument formel et retourne un élément de file ; elle retourne la référence vers la nouvelle tête de la file, qui est donc l'élément de file situé juste après l'élément de file courant (l'ancienne tête ne sera donc plus référencée).

Remarque

Encore une fois, il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer la nouvelle tête de la file (cf. étape 3 de la **Erreur ! Source du renvoi introuvable.**) afin « d'en faire quelque chose » (selon le besoin : la sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais retiré la tête ! 😞

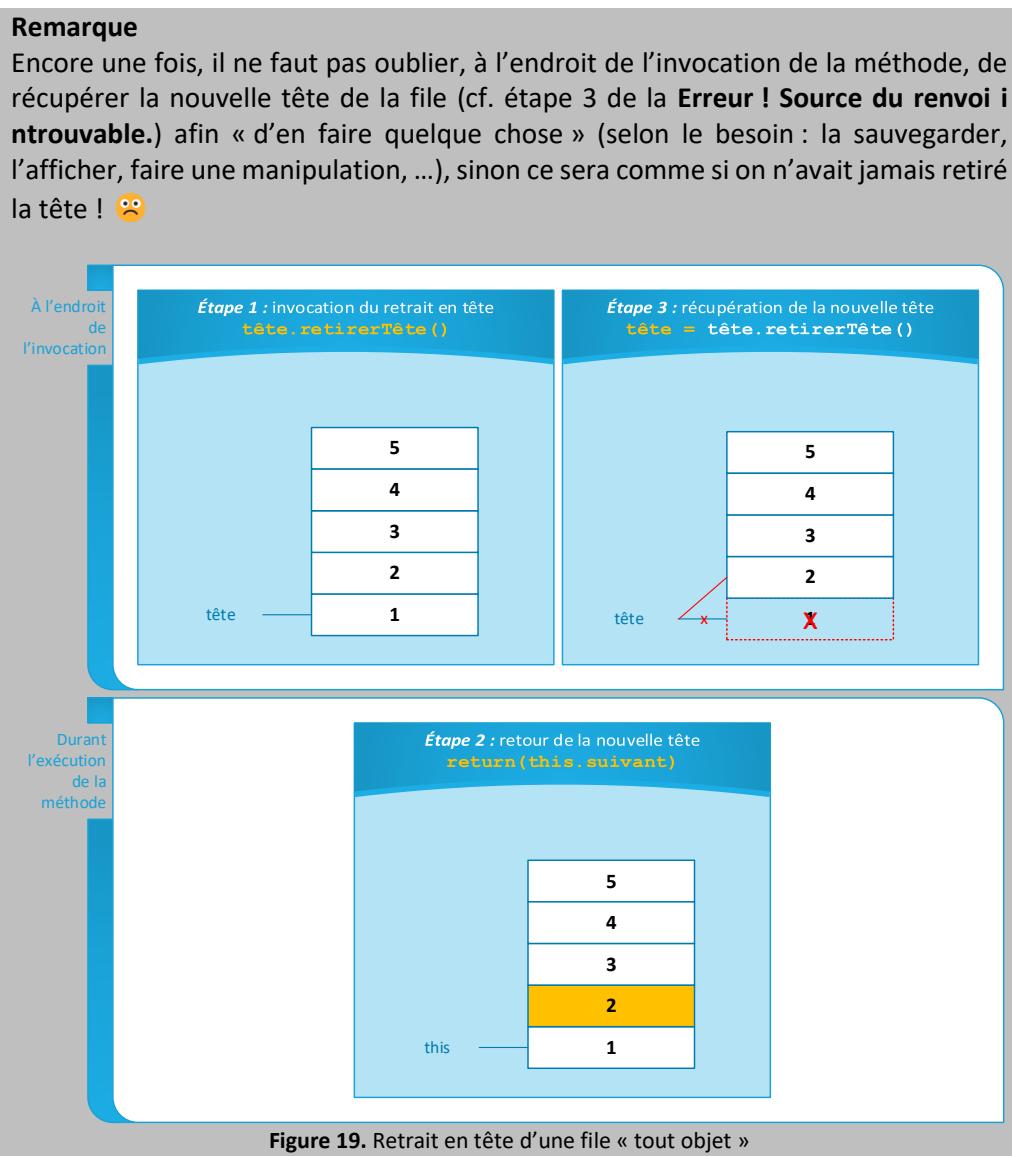


Figure 19. Retrait en tête d'une file « tout objet »

- Une méthode publique d'instance `compterNbElémentsDeFile()` : elle n'a pas d'argument formel et retourne un entier : s'il existe un élément de file après l'élément de file courant, la méthode retourne la somme de 1 (ce qui compte l'élément de file courant) et du résultat de l'invocation de la méthode `compterNbElémentsDeFile()` sur l'élément de file situé après l'élément de file courant, sinon (i.e. si l'élément de file courant est en queue de sa file) elle retourne 1 (ce qui compte l'élément de file courant).



Remarque

Compter le nombre d'éléments d'une file revient à compter l'élément courant (qui compte donc pour 1) et à ajouter cela au nombre d'éléments de la file qui se trouve après l'élément courant...

- Une méthode publique d'instance **afficherElémentDeFile()** : elle n'a pas d'argument formel et ne retourne rien ; elle affiche à l'écran la donnée contenue dans l'élément de file courant ; s'il existe un élément de file après l'élément de file courant, la méthode **afficherElémentDeFile()** est alors invoquée sur l'élément de file qui est situé juste après l'élément courant.



Remarque

Afficher le contenu d'une file revient à afficher l'élément courant puis à afficher la file qui est après l'élément courant...

2.2.2 La classe MainElémentDeFile

La classe **MainElémentDeFile** est une classe « principale » vouée à tester la classe **ElémentDeFile** précédemment écrite...

Résumé de la classe	Visibilité	Nom	Héritage explicite		
	Publique	MainElémentDeFile	Aucun héritage explicite		
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance			
	ElémentDeFile	À vous de le dire			
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration
		Aucune propriété dans cette classe			
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Argument(s) formel(s)
	Programme principal	Publique	Classe	Aucune donnée renournée	main()
					Un tableau de chaînes de caractères
					Exception(s) pouvant être levée(s)
					Aucune

Tableau 10. Résumé de la classe **MainElémentDeFile**

La classe **MainElémentDeFile** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes :

- Elle crée 2 files, chacune référencée par l'élément de file situé à sa tête, respectivement depuis une variable **fileA** et depuis une variable **fileB**,
- Elle ajoute des éléments en queue de ces 2 files,
- Elle affiche le contenu des 2 files et indique, pour chaque file, le nombre d'éléments de file qu'elle contient à ce moment-là.
- Elle retire des éléments depuis la tête de ces 2 files (jusqu'à en vider au moins une des deux),
- Elle affiche de nouveau le contenu des 2 files et indique, pour chaque file, le nombre d'éléments de file qu'elle contient à ce moment-là.



Remarque

Vous risquez, à tout instant, de vous retrouver avec le message d'erreur « par excellence » en Java (cf. Figure 9), notamment si vous avez retiré tous les éléments de vos files et tout de même cherché à accéder à un de leurs membres ensuite :

```
Valeur ôtée en tête de fileA : 9
Valeur ôtée en tête de fileB : 18
Exception in thread "main" java.lang.NullPointerException
        at TP.TP4.Exercice2.Question2.MainElémentDeFile.main(MainElémentDeFile.java:36)
```

Figure 20. Message d'erreur le plus classique en Java : `java.lang.NullPointerException`

Ce message d'erreur s'affiche quand vous cherchez à accéder à un membre (propriété ou méthode) d'un objet alors que cet objet est en fait la référence `null` ! C'est donc à vous donc de vous assurer aux moments opportuns que vous n'essayez pas d'accéder à un membre de la référence `null`. Pour cela, vous pouvez par exemple écrire un test comme suit :

```
if(fileA != null)
    fileA.afficherElémentDeFile();
```

2.3 Des files encore plus évoluées

Pour pallier le problème de la manipulation d'une file vide (*i.e.* d'une référence `null`), ce qui pose régulièrement des problèmes (comme on l'a vu à la question précédente et comme indiqué dans la dernière remarque du §2.2), on se propose de faire évoluer notre structure comme suit : l'idée est de « déconnecter » les notions de « file vide » et de « référence `null` ». Pour cela, on va introduire un élément de file particulier qui sera toujours situé en queue d'une pile (cf. **Erreur ! Source du renvoi introuvable.**). Cet élément sera particulier car il sera le seul, parmi tous les éléments de la file évoluée à laquelle il appartient, à contenir une donnée ayant pour valeur la référence `null`.

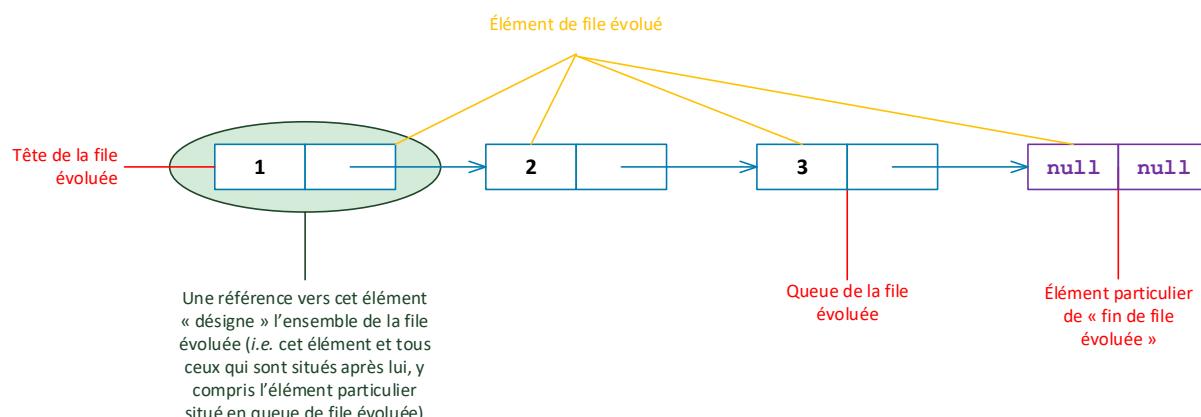


Figure 21. Structure de file évoluée, toujours en tout objet



Attention

Puisque l'élément particulier est « repéré » par le fait que la donnée qu'il encapsule est la référence **null**, cela veut dire que la donnée encapsulée dans un élément de file évoluée ne peut plus être une donnée de type primitif : c'est forcément un objet. Ainsi, si vous souhaitez tout de même encapsuler des données de type primitif, par exemple des entiers, il faudra passer par les classes wrappers.

À partir de là, on va devoir considérer au moins les points suivants lors de l'écriture de la classe :

- À sa création, une file évoluée contient uniquement cet élément particulier (cf. Figure 22),
- Une file évoluée est considérée comme « vide » quand elle ne contient que cet élément particulier ; il faut donc pouvoir indiquer si une file évoluée est vide en prenant cela en compte (cf. Figure 22),
- Un retrait d'un élément depuis la tête d'une file évoluée « vide » ne fait rien : cela renvoie la file évoluée telle quelle (*i.e.* vide, donc en fait uniquement l'élément particulier).

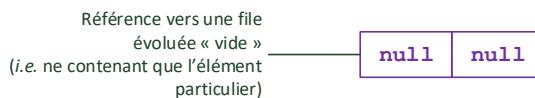


Figure 22. Structure d'une file évoluée vide

On va créer deux classes dans un paquetage **tp.tp4.exercice2.question3** (cf. Figure 12) :

- Une classe **ElémentDeFileEvoluée** : c'est elle qui va permettre de gérer les éléments qui constituent une file évoluée et, par extension, les files évoluées (puisque par la tête d'une file évoluée on accède en fait à toute la file évoluée),
- Une classe **MainElémentDeFileEvoluée** : cette classe a pour rôle de tester la mise en œuvre de files évoluées *via* des instances de la classe **ElémentDeFileEvoluée**.



Figure 23. Paquetage et classes à créer pour la gestion de files évoluées

2.3.1 La classe ElémentDeFileEvoluée

La classe **ElémentDeFileEvoluée** est un « modèle » d'éléments constituants les files évoluées...

Résumé de la classe	Visibilité		Nom		Héritage explicite		
	Publique		ElémentDeFileEvoluée		Aucun		
Classe(s) utilisée(s)	Nom		Paquetage d'appartenance				
	Integer		À vous de le dire				
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
	Privée	Instance	Integer	donnée	Pas d'initialisation à la déclaration		
	Privée	Instance	À vous de le dire	suivant	Pas d'initialisation à la déclaration		
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire	Un entier (sous forme d'objet)	Aucune
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire	Aucun	Aucune
	Assesseur (méthode)	Publique	Classe	À vous de le dire	créerFileVide()	Aucun	Aucune
	Assesseur (getter)	Publique	Instance	Integer	getDonnée()	Aucun	Aucune
	Divers	Publique	Instance	Booléen	estVide()	Aucun	Aucune
	Divers	Publique	Instance	À vous de le dire	ajouterQueue()	Un entier (sous forme d'objet)	Aucune
	Divers	Publique	Instance	À vous de le dire	retirerTête()	Aucun	Aucune
	Divers	Publique	Instance	Entier	compterNbEléments DeFileEvoluée()	Aucun	Aucune
	Divers	Publique	Instance	Aucune donnée renournée	afficherElément DeFileEvoluée()	Aucun	Aucune

Tableau 11. Résumé de la classe **ElémentDeFileEvoluée**

2.3.1.1 Propriétés de la classe ElémentDeFileEvoluée

Les deux propriétés de cette classe sont les suivantes :

- Une propriété privée d'instance (de type entier, mais sous forme d'objet) **donnée** : la valeur qu'elle encapsule est celle de la donnée stockée dans l'élément courant de la file évoluée.
- Une propriété privée d'instance (à vous de déterminer son type) **suivant** : le rôle de cette propriété est de référencer l'élément de la file évoluée situé immédiatement après l'élément courant de la file évoluée (elle vaut donc **null** si l'élément courant de la file évoluée est l'élément particulier de queue de file évoluée).

2.3.1.2 Constructeurs de la classe *ElémentDeFileEvoluée*

En outre, cette classe contient deux constructeurs :

- *Un constructeur privé ayant comme argument formel un entier (sous forme d'objet) :*
 - Il initialise la propriété **donnée** de l'élément courant de la file évoluée avec la valeur de l'argument formel,
 - Il initialise la propriété **suivant** de l'élément courant de la file évoluée avec la référence **null** (l'élément courant de la file évoluée n'est donc à ce stade devant rien, même pas devant l'élément particulier de fin de file évoluée).
- *Un constructeur privé n'ayant pas d'argument formel :* il invoque le constructeur précédent avec comme argument effectif la référence **null** (donc, l'invocation du présent constructeur n'a qu'un rôle : celui de créer un élément particulier de fin de file évoluée).

2.3.1.3 Assesseurs de la classe *ElémentDeFileEvoluée*

Cette classe déclare également deux assesseurs :

- *Un assesseur permettant d'accéder au constructeur privé sans argument formel afin de créer l'élément particulier de fin de file évoluée :* c'est une méthode publique de classe nommée **créerFileVide()**, sans argument formel et dont le rôle est de créer et retourner un nouvel élément de fin de file évoluée (donc une file évoluée vide).
- *Un assesseur permettant d'accéder à la valeur de la propriété donnée :* c'est une méthode publique d'instance nommée **getDonnée()**, sans argument formel et retournant un entier qui est la valeur de la donnée située dans l'élément courant de la file évoluée.

2.3.1.4 Autres méthodes de la classe *ElémentDeFileEvoluée*

Cette classe déclare enfin les cinq méthodes suivantes :

- *Une méthode publique d'instance **estVide()** :* elle n'a pas d'argument formel et retourne un booléen afin d'indiquer si la file évoluée courante est vide (dans ce cas elle retourne le booléen **true**) ou pas (elle retourne alors le booléen **false**).



Remarque

On sait que la file évoluée courante est vide quand elle ne contient que l'élément particulier de fin de file évoluée (*i.e.* le seul élément de la file évoluée dont la donnée est la référence **null**).

- *Une méthode publique d'instance **ajouterQueue()** :* elle un argument formel de type entier (sous forme d'objet) et retourne un élément de file évoluée ; elle crée un nouvel élément de file évoluée encapsulant l'argument formel puis parcours la file évoluée pour arriver jusqu'à son élément de queue (juste avant l'élément spécifique de fin de file évoluée) afin de raccrocher après cette queue (mais avant l'élément spécifique de fin de file évoluée) le nouvel élément créé puis, enfin, de retourner la référence vers la tête de file évoluée (*a priori* inchangée).



Remarque

Comme précédemment (cf. §2.2.1.4), il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer la nouvelle tête de la file évoluée afin « d'en faire quelque chose » (selon le besoin : la sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais ajouté d'élément en queue de la file évoluée ! 😕



Astuce

L'ajout d'un élément en queue d'une file évoluée se fait donc maintenant comme suit (cf. Figure 24) :

- On parcourt la file évoluée jusqu'à arriver à l'élément précédent l'élément particulier (on le désigne par exemple par **avantDernier** et on désigne par exemple l'élément particulier par **particulier**),
- On créer le nouvel élément (on le désigne par exemple par **nouveau**),
- On raccroche à la suite de **nouveau** l'élément **particulier**,
- On raccroche à la suite d'**avantDernier** l'élément **nouveau**,
- On retourne alors la tête de file évoluée (*a priori* inchangée).

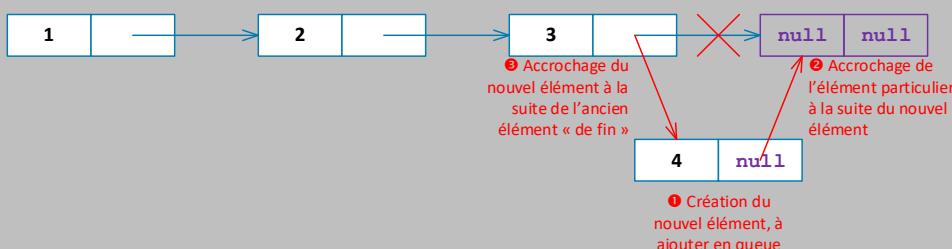


Figure 24. Ajout d'un élément en queue d'une file évoluée



Attention

Il faut **impérativement** prévoir le cas particulier de l'ajout d'un élément en queue d'une file évoluée vide. En effet, on ne peut pas la parcourir jusqu'à arriver sur l'élément qui précède l'élément particulier (pour la simple raison qu'il n'y en a pas !). Dans ce cas, il faut donc :

- Créer le nouvel élément à ajouter en queue de la file évoluée,
- Faire suivre ce nouvel élément de l'élément particulier (qui est en fait, juste avant cela, le seul élément membre de la file évoluée),
- Retourner une référence vers le nouvel élément, qui constitue donc la nouvelle tête de la file évoluée (qui est donc modifiée dans ce cas-là).

- Une méthode publique d'instance **retirerTête()** : elle n'a pas d'argument formel et retourne un élément de file évoluée ; elle retourne la référence vers la nouvelle tête de la file évoluée, qui est donc l'élément de file évoluée situé juste après l'élément courant de la file évoluée (l'ancienne tête ne sera donc plus référencée).



Remarque

Encore une fois, et comme précédemment (cf. §2.2.1.4), il ne faut pas oublier, à l'endroit de l'invocation de la méthode, de récupérer la nouvelle tête de la file évoluée afin « d'en faire quelque chose » (selon le besoin : la sauvegarder, l'afficher, faire une manipulation, ...), sinon ce sera comme si on n'avait jamais retiré la tête de la file évoluée ! 😞

- Une méthode publique d'instance `compterNbElémentsDeFileEvoluée()` : elle n'a pas d'argument formel et retourne un entier : s'il existe un élément de file évoluée après l'élément courant de la file évoluée, la méthode retourne la somme de 1 (ce qui compte l'élément courant de la file évoluée) et du résultat de l'invocation de la méthode `compterNbElémentsDeFileEvoluée()` sur l'élément de file évoluée situé après l'élément courant de la file évoluée, sinon (*i.e.* si l'élément courant de la file évoluée est l'élément particulier de fin de file évoluée) elle retourne 0.



Remarque

Comme précédemment (cf. §2.2.1.4), compter le nombre d'éléments d'une file évoluée revient à compter l'élément courant (qui compte donc pour 1) et à ajouter cela au nombre d'éléments de la file évoluée qui se trouve après l'élément courant...

Sur cette structure de file évoluée, le test d'arrêt (qui permet de savoir si on est en queue de la file évoluée ou pas) change cependant !

- Une méthode publique d'instance `afficherElémentDeFileEvoluée()` : elle n'a pas d'argument formel et ne retourne rien ; elle affiche à l'écran la donnée contenue dans l'élément courant de la file évoluée ; s'il existe un élément de file évoluée après l'élément courant de la file évoluée, la méthode `afficherElémentDeFileEvoluée()` est alors invoquée sur l'élément de file évoluée qui est situé juste après l'élément courant de la file évoluée.



Remarque

Comme précédemment (cf. §2.2.1.4), afficher le contenu d'une file évoluée revient à afficher l'élément courant puis à afficher la file évoluée qui est après l'élément courant...

Sur cette structure de file évoluée, le test d'arrêt (qui permet de savoir si on est en queue de la file évoluée ou pas) change cependant !

2.3.2 La classe MainElémentDeFileEvoluée

La classe `MainElémentDeFileEvoluée` est une classe « principale » vouée à tester la classe `ElémentDeFileEvoluée` précédemment écrite...

Résumé de la classe	Visibilité	Nom		Héritage explicite
	Publique	<code>MainElémentDeFileEvoluée</code>		Aucun héritage explicite
Classe(s) utilisée(s)	Nom		Paquetage d'appartenance	
	<code>ElémentDeFileEvoluée</code>		À vous de le dire	
Propriété(s)	Visibilité	Instance vs classe	Type	Nom
	Aucune propriété dans cette classe			
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour
	Programme principal	Publique	Classe	Aucune donnée renournée
				<code>main()</code>
				Un tableau de chaînes de caractères
				Aucune

Tableau 12. Résumé de la classe `MainElémentDeFileEvoluée`

La classe **MainElémentDeFileEvoluée** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes :

- Elle crée 2 files évoluées, chacune référencée par l'élément de file évoluée situé à sa tête, respectivement depuis une variable **fileA** et depuis une variable **fileB**,
- Elle ajoute des éléments en queue de ces 2 files évoluées,
- Elle affiche le contenu des 2 files évoluées et indique, pour chaque file évoluée, le nombre d'éléments de file évoluée qu'elle contient à ce moment-là.
- Elle retire des éléments depuis la tête de ces 2 files évoluées (jusqu'à en vider au moins une des deux),
- Elle affiche de nouveau le contenu des 2 files évoluées et indique, pour chaque file évoluée, le nombre d'éléments de file évoluée qu'elle contient à ce moment-là.

3 Mise en pratique

Il vous est maintenant proposé de mettre en pratique ce que vous venez de produire comme code Java pour l'implémentation de piles et de files (notamment les versions « tout objet » évoluées, bien sûr) en reprenant quelques exercices que vous avez vus et corrigés en cours d'algorithmique et que vous avez implémentés en langage C (module « introduction à la programmation ») :

- Pour l'utilisation des piles : on va se baser sur la simulation d'une partie de belote (peu importe que vous sachiez y jouer ou pas : on ne va pas vraiment y jouer) ; on va donc manipuler non plus des piles d'entier mais des piles de cartes (les cartes seront des instances d'une classe que l'on créera),
- Pour l'utilisation des files : on va se baser sur la simulation d'un tourniquet de processus dans un processeur ; on va donc manipuler non plus des files d'entiers mais des files de processus (les processus seront des instances d'une classe que l'on créera).

3.1 Une partie de cartes (mise en œuvre des piles)



Remarque

Cet exercice est une adaptation de l'exercice 9 de la planche de TD n°5 du module « Introduction à la programmation » de L3 Gestion parcours MIAGE, module dispensé par Mme Monique ROLBERT, que je remercie pour son autorisation de réutilisation au sein de la présente planche de TP.

On veut simuler le comportement des tas de cartes lors d'une partie de contrée. On rappelle que la contrée se joue avec 32 cartes (7, 8, 9, 10, valet, dame, roi, as dans les 4 couleurs : pique, cœur, carreau, trèfle) et à 2 équipes de 2 joueurs. Écrivez les classes nécessaires pour représenter :

- Le tas initial, qui doit assurer les comportements suivants :
 - Se couper (aléatoirement),
 - Se mélanger,
 - Se distribuer en 4 mains (une par joueur) : les cartes seront distribuées de la manière suivante : 3 cartes pour chaque joueur, puis 2 cartes pour chaque joueur, puis encore 3 cartes pour chaque joueur,
 - Se reconstituer à partir des tas finaux (cf. ci-dessous),

- Au cours de la partie, chaque joueur est susceptible de remporter un des plis gagnés par son équipe. Il constitue ainsi un tas de plis remportés qui lui est propre (forcément vide au début, éventuellement à la fin de la partie également s'il n'a remporté aucun pli).



Remarque

L'idée ici n'est surtout pas de gérer une « vraie » partie de belote ! On souhaite en fait plus simplement « suivre les cartes ». Peu importe quel joueur joue quelle carte (pourvu qu'elle fasse partie de sa main), peu importe qui aurait réellement gagné ou perdu, ...

3.1.1 Conception : le diagramme de classes UML

On vous propose ci-dessous un diagramme de classes pour répondre au problème donné.

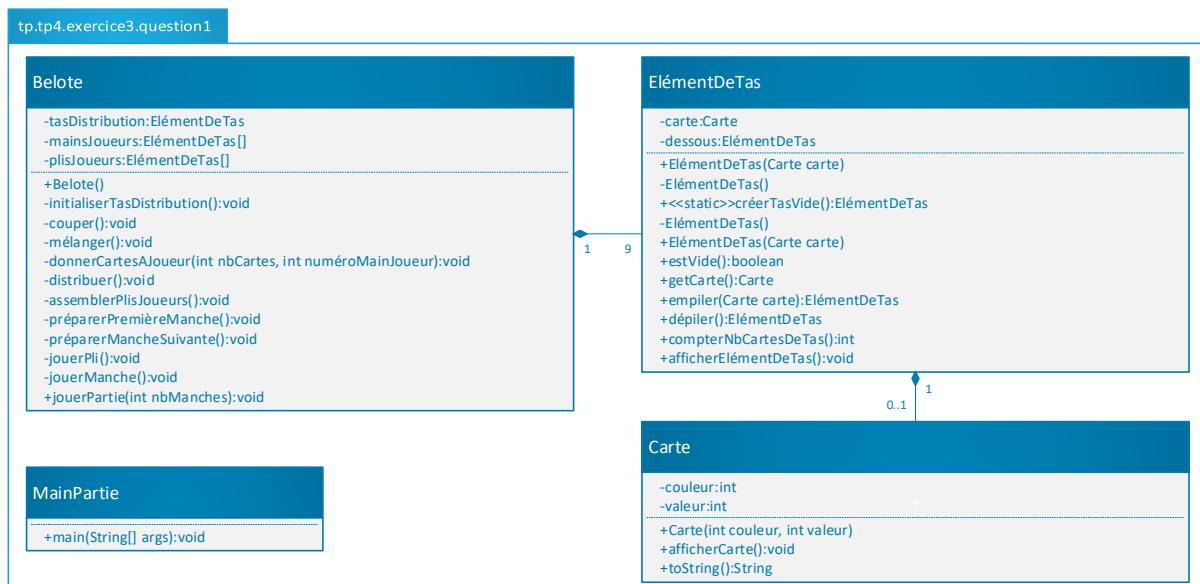


Figure 25. Diagramme de classes UML pour le suivi de cartes lors d'une partie de belote

Les 2 idées centrales sont les suivantes :

- Le tas à distribuer, les mains des joueurs et les plis ramassés par les joueurs lorsqu'ils les remportent sont finalement tous de même nature : ce sont tous des tas de cartes (surtout que, ici, dans cette simulation, on ne s'attache pas vraiment à savoir qui joue quoi, qui gagne, qui perd),
- Les tas sont assimilables en fait à des piles. On va donc se baser sur les piles évoluées précédemment traitées (cf. §1.3) pour gérer nos 9 tas de cartes (1 tas à distribuer, 4 mains de joueurs et 4 tas de plis gagnés).

3.1.2 La classe Carte

Une instance de cette classe représente une carte à jouer (donc avec une « couleur » et une valeur).

Résumé de la classe	Visibilité	Nom			Héritage explicite
	Publique	Carte			Aucun
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance			
Propriété(s)		Aucune classe n'est utilisée			
	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration
Méthode(s)	Privée	Instance	Entier	couleur	Pas d'initialisation à la déclaration
	Privée	Instance	Entier	valeur	Pas d'initialisation à la déclaration
« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)
Constructeur	Publique	Instance	À vous de le dire	À vous de le dire	Deux entiers
Divers	Publique	Instance	Chaîne de caractères	toString() (redéfinition)	Aucun
Divers	Publique	Instance	Aucune donnée renournée	afficherCarte()	Aucun
					Exception(s) pouvant être levée(s)

Tableau 13. Résumé de la classe Carte

3.1.2.1 Propriétés de la classe Carte

Cette classe déclare les deux propriétés suivantes :

- Une propriété publique d'instance (de type entier) **couleur** : elle indique si la carte est un pique, un cœur, un carreau ou un trèfle.
- Une propriété publique d'instance (de type entier) **valeur** : elle indique si la carte est un as, un 7, un 8, un 9, un 10, un valet, une dame ou un roi.



Remarque

On part du postulat suivant : chaque couleur et chaque valeur est associée à un entier, comme le montre le tableau ci-dessous :

Entier	1	2	3	4	5	6	7	8	9	10	11	12	13
Couleur	Pique	Cœur	Carreau	Trèfle									
Valeur	1						7	8	9	10	Valet	Dame	Roi

Tableau 14. Désignation de la couleur et de la valeur d'une carte par des entiers

3.1.2.2 Constructeur de la classe Carte

En outre, cette classe déclare un unique constructeur, public, ayant comme arguments formels deux entiers. Ce constructeur fonctionne comme suit : il crée une carte avec comme couleur le premier argument formel et comme valeur le second argument formel.

3.1.2.3 Autres méthodes de la classe Carte

Enfin, cette classe déclare les deux méthodes suivantes :

- Une redéfinition de la méthode d'instance publique `toString()` : héritée de la classe `Object`, elle n'a pas d'argument formel et retourne une chaîne de caractères ; la chaîne de caractères construite et retournée est la représentation, sous une forme « compréhensible », de la carte courante (par exemple, « Valet de Cœur ») ; à cette fin, elle doit donc tester les valeurs des 2 propriétés afin d'afficher la chaîne de caractères « correcte » qui leur correspond.
- Une méthode publique d'instance `afficherCarte()` : elle n'a pas d'argument formel et ne retourne rien ; elle permet d'afficher sous une forme « compréhensible » la carte courante.



Remarque

Vous avez écrit une méthode dont le rôle est de générer une chaîne de caractères « compréhensible » à partir d'une carte. Utilisez donc cette méthode pour faire un affichage « compréhensible » d'une carte !

3.1.3 La classe ElémentDeTas

Cette classe permet de gérer un tas comme une pile évoluée (cf. §1.3) de cartes. Une instance de cette classe est donc un élément faisant partie d'un tas de cartes, chaque tas de cartes étant identifié par une référence vers l'élément de tas placé à son sommet. Ainsi, la classe `ElémentDeTas` reprend les membres de la classe `ElémentdePileEvoluée`, en les adaptant lorsque c'est nécessaire au fait que ce sont des cartes qui sont encapsulées dans les éléments d'un tas (et non plus des entiers).

Résumé de la classe	Visibilité		Nom		Héritage explicite
	Publique		<code>ElémentDeTas</code>		Aucun
Classe(s) utilisée(s)					
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration
	Privée	Instance	<code>Carte</code>	<code>carte</code>	Pas d'initialisation à la déclaration
	Privée	Instance	À vous de le dire	dessous	Pas d'initialisation à la déclaration
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire
	Assesseur (méthode)	Publique	Classe	À vous de le dire	<code>créerTasVide()</code>
	Assesseur (getter)	Publique	Instance	<code>Carte</code>	<code>getCarte()</code>
	Divers	Publique	Instance	Booléen	<code>estVide()</code>
	Divers	Publique	Instance	À vous de le dire	<code>empiler()</code>
	Divers	Publique	Instance	À vous de le dire	<code>dépiler()</code>
	Divers	Publique	Instance	Entier	<code>compterNbCartesDeTas()</code>
	Divers	Publique	Instance	Aucune donnée retournée	<code>afficherElémentDeTas()</code>

Tableau 15. Résumé de la classe `ElémentDeTas`

3.1.3.1 Propriétés de la classe *ElémentDeTas*

Cette classe définit les deux propriétés suivantes :

- Une propriété privée d'instance (de type **Carte**) **carte** : elle représente la carte encapsulée dans l'élément de tas courant.
- Une propriété privée d'instance (à vous de déterminer son type) **dessous** : c'est une référence vers l'élément de tas situé sous l'élément courant au sein du tas.



Remarque

On gère bien ici un tourniquet comme une pile évoluée (cf. §1.3). Ainsi :

- Le dernier élément d'un tas est toujours l'élément particulier de fin,
- Cet élément particulier de fin se reconnaît au fait qu'il est le seul, au sein d'un tas donné, à référencer **null** au lieu d'une carte au niveau de sa propriété d'instance **carte**.

3.1.3.2 Constructeurs de la classe *ElémentDeTas*

En outre, cette classe déclare les deux constructeurs suivants :

- Un constructeur privé avec un argument formel de type **Carte** : son rôle est de créer une instance dont la propriété **carte** vaudra la carte reçue en argument formel et dont la propriété **dessous** sera la référence **null** (l'élément courant du tas n'est donc à ce stade posé sur rien, même pas sur l'élément particulier de fin de tas).
- Un constructeur privé sans argument formel : il invoque le constructeur précédent avec comme argument effectif la référence **null** (donc, l'invocation du présent constructeur n'a qu'un rôle : celui de créer un élément particulier de fin de tas).

3.1.3.3 Assesseurs de la classe *ElémentDeTas*

Cette classe déclare également deux assesseurs :

- Un assesseur permettant d'accéder au constructeur privé sans argument formel afin de créer l'élément particulier de fin de tas : c'est une méthode publique de classe nommée **créerTasVide()**, sans argument formel et dont le rôle est de créer et retourner un nouvel élément de fin de tas (donc un tas vide).
- Un assesseur permettant d'accéder à la valeur de la propriété **carte** : c'est une méthode publique d'instance nommée **getCarte()**, sans argument formel et retournant une carte qui est la valeur de la donnée située dans l'élément courant du tas.

3.1.3.4 Autres méthodes de la classe *ElémentDeTas*

Enfin, cette classe déclare les cinq méthodes suivantes :

- Une méthode publique d'instance **estVide()** : elle n'a pas d'argument formel et retourne un booléen ; elle indique si, oui ou non, le tas courant est « vide » (i.e. si son sommet est l'élément de tas particulier dont la donnée, i.e. la carte, est la référence **null**).
- Une méthode publique d'instance **empiler()** : elle a un argument formel de type **Carte** et retourne un élément de tas ; elle crée un nouvel élément de tas encapsulant la carte reçue en argument, puis elle pose ce nouvel élément de tas sur le tas courant et, enfin, elle retourne une référence vers ce nouvel élément, puisqu'il est maintenant au sommet du tas.

- Une méthode publique d'instance **dépiler()** : elle n'a pas d'argument formel et retourne un élément de tas ; si le tas n'est pas vide, elle retourne une référence vers l'élément de tas situé en seconde position (donc juste sous le sommet courant), sinon, i.e. si le tas est vide, elle le retourne tel quel (puisque il n'y a rien à dépiler).
- Une méthode publique d'instance **compteNbCartesDeTas()** : elle n'a pas d'argument formel et retourne un entier ; si l'élément de tas courant est le tas vide, elle retourne 0, sinon elle retourne la somme de l'entier 1 avec le résultat d'un appel récursif sur l'élément inférieur dans le tas.
- Une méthode publique d'instance **afficherElémentDeTas()** : elle n'a pas d'argument formel et ne retourne rien ; si l'élément de tas courant n'est pas le tas vide, elle affiche la carte contenue dans l'élément de tas courant et se rappelle récursivement sur l'élément inférieur dans le tas, sinon elle n'affiche rien.

3.1.4 La classe Belote

Une instance de cette classe permet de mettre en place et de réaliser une partie de belote entre 4 joueurs.

Résumé de la classe	Visibilité		Nom		Héritage explicite		
	Publique		Belote		Aucun		
Classe(s) utilisée(s)	<i>Nom</i>	<i>Paquetage d'appartenance</i>					
	<i>Carte</i>	À vous de le dire					
	<i>ElémentDeTas</i>	À vous de le dire					
	<i>Random</i>	java.util					
Propriété(s)	Visibilité	<i>Instance vs classe</i>	Type	Nom	<i>Initialisation à la déclaration</i>		
	Privée	Instance	ElémentDeTas	tasDistribution	Pas d'initialisation à la déclaration		
	Privée	Instance	Tableau de 4 ElémentDeTas	mainsJoueurs	Pas d'initialisation à la déclaration		
	Privée	Instance	Tableau de 4 ElémentDeTas	plisJoueurs	Pas d'initialisation à la déclaration		
Méthode(s)	« Fonction »	Visibilité	<i>Instance vs classe</i>	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Constructeur	Publique	Instance	À vous de le dire	À vous de le dire	Aucun	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	initialiserTasDistribution()	Aucun	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	couper()	Aucun	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	mélanger()	Un entier	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	donnerCartesAJoueur()	Deux entiers	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	distribuer()	Aucun	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	assemblerPlisJoueurs()	Aucun	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	préparerPremièreManche()	Aucun	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	jouerPli()	Aucun	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	jouerManche()	Un entier	Aucune
	Divers	Privée	Instance	Aucune donnée retournée	jouerPartie()	Un entier	Aucune

Tableau 16. Résumé de la classe **Belote**

3.1.4.1 Propriétés de la classe Belote

Cette classe déclare les trois propriétés suivantes :

- Une propriété privée d'instance (de type `ElémentDeTas`) `tasDistribution` : elle représente le tas de cartes à distribuer.
- Une propriété privée d'instance (de type tableau d'éléments de tas) `mainsJoueurs` : chaque case de ce tableau représente une main d'un des joueurs de la partie (qui sont au nombre de quatre), donc un tas de cartes.
- Une propriété privée d'instance (de type tableau d'éléments de tas) `plisJoueurs` : chaque case de ce tableau représente les plis ramassés pendant la partie par un des joueurs de la partie (qui sont au nombre de quatre), donc un tas de cartes.



Remarque

La belote se joue à 4 joueurs. L'idée est donc de numérotter chacun de ces joueurs (de 0 à 3 ou de 1 à 4, comme vous voulez). Ainsi :

- Le tas de cartes à distribuer est unique et commun,
- Les cartes en position n^i dans le tableau des mains des joueurs représentent les cartes qui sont encore dans la main du joueur n^i ,
- Les cartes en position n^i dans le tableau des plis des joueurs représentent les cartes gagnées par le joueur n^i .

3.1.4.2 Constructeur de la classe Belote

En outre, cette classe déclare un unique constructeur, public et sans argument formel : il initialise le tas de distribution avec un tas vide, puis il initialise le tableau des mains des joueurs comme un tableau de 4 cases pouvant contenir chacune un élément de tas, et enfin il initialise le tableau des plis des joueurs comme un tableau de 4 cases pouvant contenir chacune un élément de tas.

3.1.4.3 Autres méthodes de la classe Belote

Enfin, cette classe déclare les dix méthodes suivantes :

- Une méthode privée d'instance `initialiserTasDistribution()` : elle n'a pas d'argument formel et ne retourne rien ; son rôle est de remplir le tas à distribuer avec les 32 cartes d'un jeu de belote (elle peut également ensuite, si nécessaire, afficher le contenu des 9 tas, i.e. du tas de distribution, des mains des 4 joueurs et des plis des 4 joueurs) ; pour cela :
 - Elle fait une boucle sur les 4 couleurs possibles,
 - Elle fait une boucle imbriquée sur les 13 valeurs de cartes possibles,
 - Si la valeur est égale à 1 (as) ou comprise entre 7 et 13 (7 à roi), elle crée une nouvelle carte avec la couleur courante et la valeur courante,
 - Elle empile cette nouvelle carte sur le tas à distribuer.

- Une méthode privée d'instance **couper ()** : elle n'a pas d'argument formel et ne retourne rien ; son rôle est de couper en 2 le tas à distribuer, le rang de coupe étant déterminé aléatoirement (elle peut également ensuite, si nécessaire, afficher le contenu des 9 tas) ; pour cela :
 - Elle déclare 2 variables locales représentant des tas « intermédiaires »,
 - Elle tire au hasard un nombre entre 1 et 32 afin de choisir un rang de coupe¹,
 - Elle fait une boucle de 1 au rang de coupe pour dépiler une carte depuis le tas à distribuer et l'empiler sur le premier tas intermédiaire,
 - Elle fait une boucle tant que le tas à distribuer n'est pas vide pour dépiler une carte depuis le tas à distribuer et l'empiler sur le second tas intermédiaire,
 - Elle fait une boucle tant que le premier tas intermédiaire n'est pas vide pour dépiler une carte depuis le premier tas intermédiaire et l'empiler sur le tas à distribuer,
 - Elle fait une boucle tant que le second tas intermédiaire n'est pas vide pour dépiler une carte depuis le second tas intermédiaire et l'empiler sur le tas à distribuer.

En pratique

De façon « visuelle », voici ce que donnent les 3 étapes de la coupe :

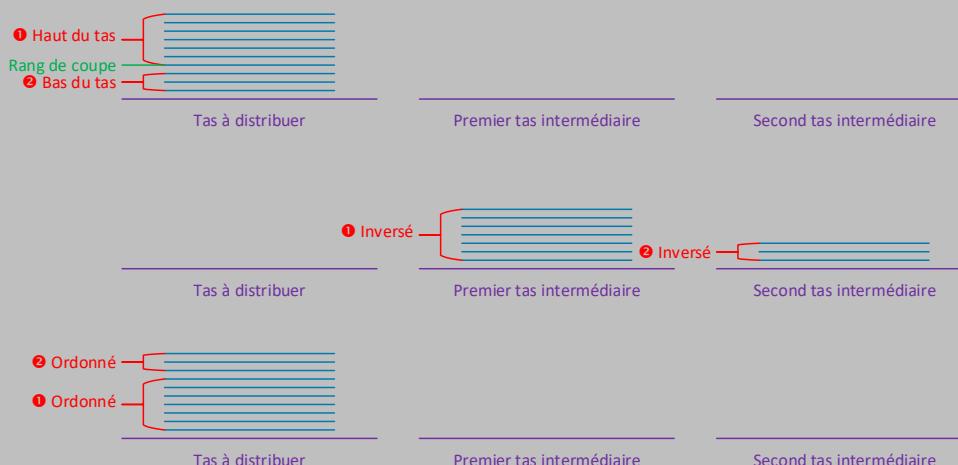


Figure 26. Représentation visuelle des 3 étapes de la coupe d'un tas de cartes

¹ Pour ce tirage aléatoire, utilisez la classe **java.util.Random** (reportez-vous à la documentation de l'API Java).

- Une méthode privée d'instance **mélanger()** : elle a un argument formel de type entier et ne retourne rien ; son rôle est de mélanger le tas à distribuer autant de fois que la valeur de l'argument formel.



Attention

La problématique d'un mélange aléatoire de donnée est assez complexe. Cependant, l'ensemble des solutions que l'on peut trouver fonctionne globalement uniquement sur des ensembles de données aux éléments desquels on accède par un indice de position (comme pour un tableau, aux éléments duquel on accède via une position débutant à 0). L'idée est alors globalement toujours la suivante :

- Le mélange va se faire n fois,
- Chaque mélange implique une opération d'échange qui va se faire autant de fois que d'éléments à mélanger (ici, donc, 32 fois puisqu'il y a 32 cartes),
- Une opération d'échange consiste à tirer aléatoirement² 2 positions dans l'ensemble à mélanger et à échanger les données situées dans ces 2 positions.



En pratique

Le souci est que les tas que l'on manipule ne sont pas prévus pour que l'on accède aux données qu'ils contiennent via la position de ces données dans le tas. Qu'à cela ne tienne ! On peut pallier cela simplement :

- Juste avant le mélange, on construit un tableau temporaire contenant exactement autant de cases que de cartes dans le tas à mélanger et on le remplit, case par case, de la façon suivante : on récupère la carte au sommet du tas à mélanger, on la dépile et on la place dans la case courante du tableau,
- Pendant le mélange : c'est ce tableau qu'on mélange aléatoirement,
- Après le mélange, on parcourt le tableau et on récupère dans la case courante la carte qu'elle contient ; cette carte est alors empilée dans le tas à mélanger, que l'on reconstitue ainsi, carte après carte, après mélange du tableau.

- Une méthode privée d'instance **donnerCartesAJoueur()** : elle a 2 arguments formels entiers (le premier représente le nombre de cartes à donner et le second représente le numéro du joueur auquel on veut les donner) et ne retourne rien ; il s'agit de donner depuis le tas à distribuer X cartes au joueur de numéro Y ; cette méthode dépile donc X cartes depuis le tas à distribuer, une par une, et, toujours une par une, les empile sur la main du joueur Y.
- Une méthode privée d'instance **distribuer()** : elle n'a pas d'argument formel et ne retourne rien ; cette méthode donne, depuis le tas à distribuer, d'abord 3 cartes à chaque joueur, puis 2 cartes à chaque joueur et enfin de nouveau 3 cartes à chaque joueur.



Remarque

Vous avez écrit une méthode permettant de donner X cartes au joueur Y, utilisez donc la !

- Une méthode privée d'instance **assemblerPlisJoueurs()** : elle n'a pas d'argument formel et ne retourne rien ; son rôle est d'assembler les tas des plis ramassés pour les joueurs afin de reconstituer un tas à distribuer (il suffit pour cela, pour chaque tas de plis ramassés, de dépiler depuis lui, tant qu'il n'est pas vide, et d'empiler sur le tas à distribuer).

² Pour ce tirage aléatoire, utilisez la classe `java.util.Random` (reportez-vous à la documentation de l'API Java).

- Une méthode privée d'instance **préparerPremièreManche()** : elle n'a pas d'argument formel et ne retourne rien ; son rôle est d'initialiser le tas à distribuer, de le mélanger, de le couper et de le distribuer.

**Remarque**

Vous avez écrit des méthodes prenant en charge ces 4 tâches, utilisez donc les !

- Une méthode privée d'instance **préparerMancheSuivante()** : elle n'a pas d'argument formel et ne retourne rien ; son rôle est d'assembler les plis ramassés par les joueurs pour reformer le tas à distribuer, de couper le tas à distribuer ainsi reformé et de le distribuer aux joueurs.

**Remarque**

Vous avez écrit des méthodes prenant en charge ces 3 tâches, utilisez donc les !

- Une méthode privée d'instance **jouerPli()** : elle n'a pas d'argument formel et ne retourne rien ; le rôle de cette méthode est de « simuler » un pli ; pour cela, elle déclare un tas de cartes intermédiaire temporaire, initialement vide, et le remplit en empilant sur lui une carte dépliée depuis le sommet de chacune des mains des 4 joueurs ; le joueur qui gagne le pli est tiré au sort³ et le pli courant est alors placé sur le tas des plis ramassés par le joueur gagnant (en dépliant les cartes depuis le pli courant, jusqu'à ce qu'il soit vide, et en les empilant sur le tas des plis ramassés par le gagnant).

**Remarque**

Bien sûr, dans la réalité, cela ne se passe pas comme ça :

- C'est le joueur qui choisit, parmi les cartes de sa main, laquelle il va jouer dans le pli (alors qu'ici on dépile toujours la carte au sommet de sa main),
- Le joueur qui gagne le pli dépend des cartes jouées sur ce pli par chacun des joueurs (alors qu'ici on choisit au hasard le joueur qui gagne le pli).

- Une méthode privée d'instance **jouerManche()** : elle a un argument formel de type entier et ne retourne rien ; elle fait simplement jouer la manche dont le numéro est celui reçu en argument, i.e. réaliser 8 plis et préparer les tas pour la manche suivante.

**Remarque**

Vous avez écrit des méthodes prenant en charge ces 2 tâches, utilisez donc les !

³ Pour ce tirage aléatoire, utilisez la classe **java.util.Random** (reportez-vous à la documentation de l'API Java).

- Une méthode publique d'instance **jouerPartie()** : elle a un argument formel de type entier et ne retourne rien ; elle « simule » une partie complète, i.e. elle prépare les 9 tas pour une première manche, et tant que le nombre de manches à jouer (reçu en argument formel) n'est pas atteint, elle fait jouer une manche.



Remarque

Vous avez écrit des méthodes prenant en charge ces 2 tâches, utilisez donc les !

3.1.5 La classe MainPartie

Cette classe est la classe principale : son rôle est de lancer la partie.

Résumé de la classe	Visibilité	Nom	Héritage explicite				
	Publique	MainPartie	Aucun héritage explicite				
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance					
	Belote	À vous de le dire					
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration		
	Aucune propriété dans cette classe						
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)	Exception(s) pouvant être levée(s)
	Programme principal	Publique	Classe	Aucune donnée renournée	main()	Un tableau de chaînes de caractères	Aucune

Tableau 17. Résumé de la classe **MainPartie**

La classe **MainPartie** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes : elle instancie la classe **Belote** puis demande à cette instance de jouer une partie en 4 manches.

3.2 Un système multitâche



Remarque

Cet exercice est une adaptation de l'exercice 3 de la planche de TD n°5 du module « Introduction à la programmation » de L3G parcours MIAGE, module dispensé par Mme Monique ROLBERT, que je remercie pour son autorisation de réutilisation au sein de la présente planche de TP.

Les systèmes multitâches donnent l'illusion de faire tourner plusieurs processus à la fois, même s'ils ne disposent que d'un seul processeur. Pour simuler ce parallélisme, le système peut utiliser un algorithme dit « du tourniquet »⁴ : il consiste à faire exécuter le premier processus demandeur pendant un certain temps T , puis de l'arrêter, de le mettre en dernier dans la file et de lancer le deuxième, etc. Quand un processus est fini (ce qui ne peut arriver que lorsqu'il est en train de s'exécuter), il sort du tourniquet. Quand un processus veut démarrer, il entre à la fin du tourniquet.

⁴ C'est en réalité un « ordonnanceur préemptif circulaire ».

On vous demande donc de créer les classes nécessaires pour représenter cette gestion « parallèle » des processus sur un processeur au moyen du système du tourniquet.

3.2.1 Conception : le diagramme de classes UML

On vous propose ci-dessous un diagramme de classes pour répondre au problème donné.

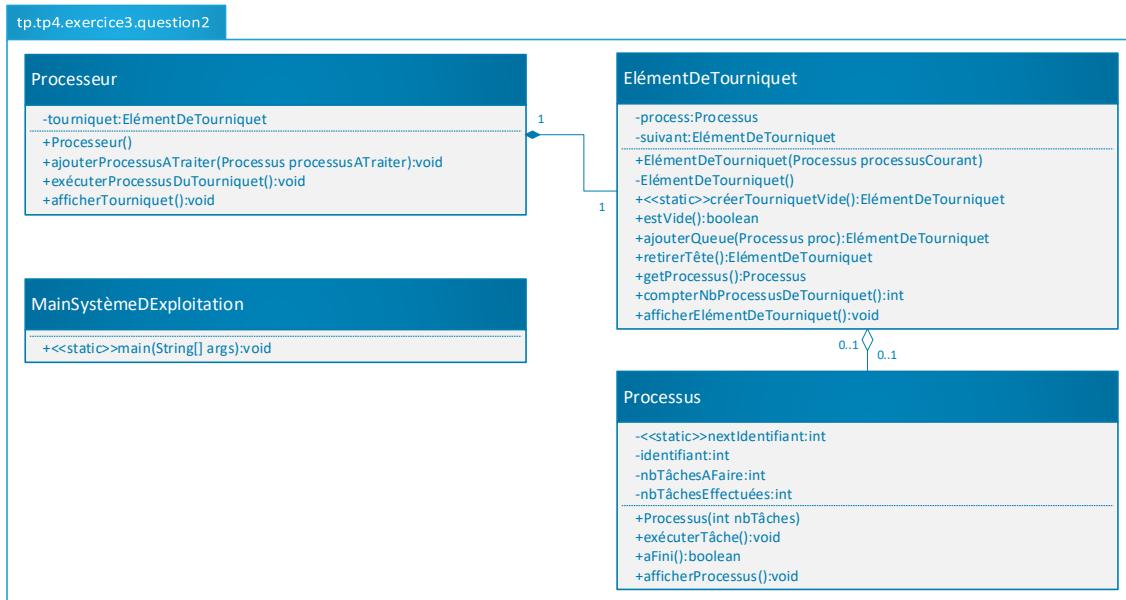


Figure 27. Paquetage et classes pour la simulation de systèmes multitâches

Le point central de ce diagramme est le suivant : le tourniquet d'un processeur est vu comme une file évoluée dont les éléments contiennent des processus.

3.2.2 La classe Processus

Une instance de cette classe représente un processus à faire exécuter par un processeur, *via* son système de tourniquet. Pour simplifier ici, on supposera :

- Qu'un processus doit faire un nombre de tâches prédéterminé (connu à sa création) pour finir son exécution (et, donc, « sortir » du tourniquet du processeur),
- Que chaque tâche qu'un processus doit réaliser consistera simplement à incrémenter un compteur dont le rôle est d'indiquer le nombre de tâches que le processus a déjà exécutées.

Résumé de la classe	Visibilité Publique	Nom Processus	Héritage explicite Aucun				
Classe(s) utilisée(s)	<i>Nom</i> Aucune classe n'est utilisée	<i>Paquetage d'appartenance</i>					
Propriété(s)	Visibilité Privée	Instance vs classe Classe	Type Entier	Nom nextIdentifiant	Initialisation à la déclaration Valeur littérale 0		
	Privée	Instance	Entier	identifiant	Pas d'initialisation à la déclaration		
	Privée	Instance	Entier	nbTâchesAFaire	Pas d'initialisation à la déclaration		
	Privée	Instance	Entier	nbTâchesEffectuées	Pas d'initialisation à la déclaration		
Méthode(s)	« Fonction » Constructeur Divers Divers	Visibilité Publique Publique Publique	Instance vs classe Instance Instance Instance	Type de retour À vous de le dire exécuterTâche() aFini() afficherProcessus()	Nom À vous de le dire Aucune donnée retournée Aucune donnée retournée	Argument(s) formel(s) Un entier Aucun Aucun	Exception(s) pouvant être levée(s) Aucune Aucune Aucune

Tableau 18. Résumé de la classe **Processus**

3.2.2.1 Propriétés de la classe *Processus*

Cette classe déclare les quatre propriétés suivantes :

- Une propriété privée de classe (de type entier) **nextIdentifiant** : valant initialement 0, elle servira à attribuer à chaque processus créé un identifiant unique.
- Une propriété privée d'instance (de type entier) **identifiant** : c'est l'identifiant du processus courant.
- Une propriété privée d'instance (de type entier) **nbTâchesAFaire** : c'est le nombre de tâches que le processus devra traiter avant de se terminer.
- Une propriété privée d'instance (de type entier) **nbTâchesEffectuées** : c'est le nombre de tâches que le processus courant a déjà effectuées.

3.2.2.2 Constructeur de la classe *Processus*

En outre, cette classe déclare un unique constructeur, public, ayant comme argument formel un entier. Ce constructeur fonctionne comme suit : il initialise l'identifiant du processus courant en lui affectant la valeur courante de la propriété de classe **nextIdentifiant**, puis il incrémentera la valeur de cette propriété de classe, il initialise ensuite le nombre de tâches que le processus courant devra faire avant de se terminer à la valeur de l'argument formel, et enfin il initialise le nombre de tâches déjà effectuées par le processus courant à 0.

3.2.2.3 Autres méthodes de la classe Processus

Enfin, cette classe déclare les quatre méthodes suivantes :

- *Une méthode publique d'instance **exécuterTâche()** : elle n'a pas d'argument formel et ne retourne rien ; cette méthode doit « simuler » l'exécution de la prochaine tâche que le processus courant doit traiter. Pour cette « simulation », cette méthode réalisera les traitements suivants : elle incrémente le nombre de tâches effectuées par le processus courant et affiche un message (du genre : « Le processus ID effectue la tâche X sur Y », où « ID » est l'identifiant du processus courant, « X » est le nombre de tâches que ce processus a déjà exécutées, y compris la tâche en cours, et « Y » est le nombre total de tâches que le processus doit exécuter avant de se terminer).*



Remarque

Vous allez écrire (ci-dessous) une méthode permettant de faire « l'affichage d'un processus ». Utilisez donc la.

- *Une méthode publique d'instance **aFini()** : elle n'a pas d'argument formel et retourne un booléen ; elle indique si, oui ou non, le processus courant a traité toutes les tâches qu'il était censé traiter.*
- *Une méthode publique d'instance **afficherProcessus()** : elle n'a pas d'argument formel et ne retourne rien ; elle affiche sur la console un message du genre : « Processus ID : X tâches effectuées sur Y » (où « ID » est l'identifiant du processus courant, « X » est le nombre de tâches que ce processus a déjà exécutées et « Y » est le nombre total de tâches que le processus doit exécuter avant de se terminer).*

3.2.3 La classe ElémentDeTourniquet

Une instance de cette classe est un composant du tourniquet d'un processus (un tel tourniquet étant une file évoluée de processus). En ce sens, elle reprend le principe de fonctionnement de la classe **ElémentDeFileEvoluée** traitée à l'exercice précédent (cf. 2.3), en les adaptant lorsque c'est nécessaire au fait que ce sont des processus qui sont encapsulés dans les éléments d'un tourniquet (et non plus des entiers).

Résumé de la classe		Visibilité	Nom	Héritage explicite					
		Publique	ElémentDeTourniquet	Aucun					
Classe(s) utilisée(s)		Nom	Paquetage d'appartenance						
		Processus	À vous de le dire						
Propriété(s)		Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration			
		Privée	Instance	Processus	process	Pas d'initialisation à la déclaration			
		Privée	Instance	À vous de le dire	suivant	Pas d'initialisation à la déclaration			
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)			
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire	Un processus			
	Constructeur	Privée	Instance	À vous de le dire	À vous de le dire	Aucun			
	Assesseur (méthode)	Publique	Classe	À vous de le dire	créerTourniquetVide()	Aucun			
	Assesseur (getter)	Publique	Instance	Processus	getProcessus()	Aucun			
	Divers	Publique	Instance	Booléen	estVide()	Aucun			
	Divers	Publique	Instance	À vous de le dire	ajouterQueue()	Un processus			
	Divers	Publique	Instance	À vous de le dire	retirerTête()	Aucun			
	Divers	Publique	Instance	Entier	compterNbProcessusDeTourniquet()	Aucun			
	Divers	Publique	Instance	Aucune donnée retournée	afficherElémentDeTourniquet()	Aucun			

Tableau 19. Résumé de la classe **ElémentDeTourniquet**

3.2.3.1 Propriétés de la classe **ElémentDeTourniquet**

Cette classe déclare les deux propriétés suivantes :

- Une propriété privée d'instance (de type **Processus**) **process** : c'est le processus encapsulé dans l'élément de tourniquet courant.
- Une propriété privée d'instance (à vous de déterminer son type) **suivant** : c'est une référence vers l'élément suivant au sein du tourniquet.



Remarque

On gère bien ici un tourniquet comme une file évoluée (cf. §2.3). Ainsi :

- Le dernier élément d'un tourniquet est toujours l'élément particulier de fin,
- Cet élément particulier de fin se reconnaît au fait qu'il est le seul, au sein d'un tourniquet donné, à référencer **null** au lieu d'un processus au niveau de sa propriété d'instance **process**.

3.2.3.2 Constructeurs de la classe *ElémentDeTourniquet*

En outre, cette classe déclare deux constructeurs :

- *Un constructeur privé ayant un argument formel de type Processus* : il initialise la propriété **process** de l'élément courant du tourniquet avec le processus reçu en argument formel puis il initialise la propriété **suivant** de l'élément courant du tourniquet avec la référence **null**.
- *Un constructeur privé sans argument formel* : il invoque le constructeur précédent avec comme argument effectif la référence **null** (donc, l'invocation du présent constructeur n'a qu'un rôle : celui de créer un élément particulier de fin de tourniquet).

3.2.3.3 Assesseurs de la classe *ElémentDeTourniquet*

Cette classe déclare également deux assesseurs :

- *Un assesseur permettant d'accéder au constructeur privé sans argument formel afin de créer l'élément particulier de fin de tourniquet* : c'est une méthode publique de classe nommée **créerTourniquetVide()**, sans argument formel et dont le rôle est de créer et retourner un nouvel élément de fin de tourniquet (donc un tourniquet vide).
- *Un assesseur permettant d'accéder à la valeur de la propriété process* : c'est une méthode publique d'instance nommée **getProcessus()**, sans argument formel et retournant un processus qui est la valeur de la donnée située dans l'élément courant du tourniquet.

3.2.3.4 Autres méthodes de la classe *ElémentDeTourniquet*

Enfin, cette classe déclare les cinq méthodes suivantes :

- *Une méthode publique d'instance **estVide()*** : elle n'a pas d'argument formel et retourne un booléen : elle retourne vraie si le tourniquet est « vide » (i.e. si l'élément situé à sa tête est l'élément particulier reconnaissable au fait qu'il encapsule un processus dont la valeur est la référence **null**) et faux sinon.
- *Une méthode publique d'instance **ajouterQueue()*** : elle a un argument formel de type **Processus** et retourne un élément de tourniquet ; elle crée un nouvel élément de tourniquet encapsulant le processus reçu en argument formel, puis elle parcourt le tourniquet jusqu'à arriver à son dernier élément « réel » (i.e. à celui qui précède l'élément particulier de fin) afin de placer le nouvel élément entre le dernier élément « réel » et l'élément particulier de fin pour ensuite retourner une référence vers la tête du tourniquet (*a priori* inchangée).



Remarque

Pour l'ajout en queue de file, avec une telle structure de file évoluée, reportez-vous plus encore à ce que l'on a fait dans l'exercice des files évoluées (cf. Figure 24).

- *Une méthode publique d'instance **retirerTête()*** : elle n'a pas d'argument formel et retourne un élément de tourniquet ; si le tourniquet est « vide », elle retourne l'élément courant du tourniquet, sinon elle retourne l'élément qui suit l'élément courant du tourniquet.
- *Une méthode publique d'instance **compterNbProcessusDeTourniquet()*** : elle n'a pas d'argument formel et retourne un entier : le nombre d'éléments situés dans le tourniquet dont l'élément courant est la tête.

- Une méthode publique d'instance **afficherElémentDeTourniquet()** : elle n'a pas d'argument formel et ne retourne rien ; elle affiche à l'écran le processus contenu dans l'élément de tourniquet courant (en se basant, bien sûr, sur les méthodes définies dans la classe **Processus**). S'il existe un élément de tourniquet après l'élément courant, la méthode **afficherElémentDeTourniquet()** est alors ensuite appelée sur celui qui est situé après l'élément courant.

3.2.4 La classe Processeur

Une instance de cette classe représente un processeur. Son rôle ici est de « gérer » le tourniquet en son sein (notamment pour y placer les processus à exécuter et les lancer).

Résumé de la classe	Visibilité	Nom			Héritage explicite
	Publique	Processeur			Aucun
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance			
	Processus	À vous de le dire			
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration
	Privée	Instance	À vous de le dire	tourniquet	Pas d'initialisation à la déclaration
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom
	Constructeur	Publique	Instance	À vous de le dire	À vous de le dire
	Divers	Publique	Instance	Booléen	estVide()
	Divers	Publique	Instance	Aucune donnée retournée	ajouterProcessus ATraiter()
	Divers	Publique	Instance	Aucune donnée retournée	exécterProcessus DuTourniquet()
	Divers	Publique	Instance	Aucune donnée retournée	afficherTourniquet()

Tableau 20. Résumé de la classe **Processeur**

3.2.4.1 Propriété de la classe Processeur

Cette classe ne définit qu'une seule propriété : une propriété publique d'instance (à vous de déterminer son type) **tourniquet**, étant une référence vers la tête du tourniquet du processeur courant.

3.2.4.2 Constructeur de la classe Processeur

En outre, cette classe déclare un unique constructeur, public et sans argument formel, dont le rôle est d'initialiser le tourniquet du processeur courant avec un tourniquet vide.

3.2.4.3 Autres méthodes de la classe Processeur

Enfin, cette classe déclare les trois méthodes suivantes :

- Une méthode publique d'instance **ajouterProcessusATraiter()** : elle a un argument formel représentant un processus et ne retourne rien ; son rôle est d'ajouter dans le tourniquet du processeur courant le processus passé en argument.
- Une méthode publique d'instance **exécuterProcessusDuTourniquet()** : elle n'a pas d'argument formel et ne retourne rien ; elle vise à exécuter l'ensemble des processus situés dans le tourniquet, jusqu'à ce qu'ils aient tous terminé leur exécution ; le principe est le suivant, tant que le tourniquet du processeur courant n'est pas vide, on récupère le processus situé dans la tête du tourniquet puis on retire alors la tête du tourniquet avant de demander au processus récupéré d'exécuter une tâche et enfin, s'il lui reste encore des tâches à accomplir (*i.e.* s'il n'a pas fini entièrement son exécution), on l'ajoute dans un nouvel élément à la fin du tourniquet du processeur courant.
- Une méthode publique d'instance **afficherTourniquet()** : elle n'a pas d'argument formel et ne retourne rien ; elle vise à afficher l'ensemble des éléments du tourniquet du processeur courant.

3.2.5 La classe MainSystèmeDExploitation

C'est la classe principale : elle représente le système d'exploitation de la machine contenant le ou les processeurs sur lequel (lesquels) on souhaite faire exécuter des processus *via* leur tourniquet interne.

Résumé de la classe	Visibilité	Nom		Héritage explicite					
	Publique		MainSystèmeDExploitation	Aucun héritage explicite					
Classe(s) utilisée(s)	Nom	Paquetage d'appartenance							
	Processeur	À vous de le dire							
Propriété(s)	Visibilité	Instance vs classe	Type	Nom	Initialisation à la déclaration				
	Aucune propriété dans cette classe								
Méthode(s)	« Fonction »	Visibilité	Instance vs classe	Type de retour	Nom	Argument(s) formel(s)			
	Programme principal	Publique	Classe	Aucune donnée renvoyée	main()	Un tableau de chaînes de caractères			
						Aucune			

Tableau 21. Résumé de la classe **MainSystèmeDExploitation**

La classe **MainSystèmeDExploitation** ne définit qu'une méthode publique de classe, **main()**, prenant comme unique argument formel un tableau de chaînes de caractères et ne retournant rien. Cette méthode réalise les tâches suivantes : elle instancie un processeur, puis lui rajoute des processus à traiter (chacun devant exécuter un nombre de tâches différent avant de se terminer), demande au processeur d'afficher son tourniquet et finit par lui demander d'exécuter les processus contenus dans son tourniquet.

Table des illustrations

Figure 1. Organisation en paquetages des classes pour ce TP	3
Figure 2. Structure de pile (LIFO).....	3
Figure 3. Empilement/dépilement avec une pile	4
Figure 4. Paquetage et classes à créer pour la gestion de piles au moyen de tableaux	4
Figure 5. Paquetage et classes à créer pour la gestion de piles « tout objet ».....	9
Figure 6. Représentation d'une pile en tout objet, avec des éléments de pile	10
Figure 7. Empilement sur une pile « tout objet »	11
Figure 8. Dépilement depuis une pile « tout objet »	12
Figure 9. Message d'erreur le plus classique en Java : java.lang.NullPointerException.	14
Figure 10. Structure de pile évoluée, toujours en tout objet.....	14
Figure 11. Structure d'une pile évoluée vide	15
Figure 12. Paquetage et classes à créer pour la gestion de piles évoluées.....	15
Figure 13. Structure de file (FIFO)	19
Figure 14. Ajout (en queue)/retrait (en tête) avec une file.....	19
Figure 15. Paquetage et classes à créer pour la gestion de files au moyen de tableaux	20
Figure 16. Paquetage et classes à créer pour la gestion de files « tout objet ».....	24
Figure 17. Représentation d'une file en tout objet, avec des éléments de file	25
Figure 18. Ajout en queue d'une file « tout objet »	26
Figure 19. Retrait en tête d'une file « tout objet »	27
Figure 20. Message d'erreur le plus classique en Java : java.lang.NullPointerException	29
Figure 21. Structure de file évoluée, toujours en tout objet.....	29
Figure 22. Structure d'une file évoluée vide	30
Figure 23. Paquetage et classes à créer pour la gestion de files évoluées	30
Figure 26. Ajout d'un élément en queue d'une file évoluée.....	33
Figure 25. Diagramme de classes UML pour le suivi de cartes lors d'une partie de belote	36
Figure 26. Représentation visuelle des 3 étapes de la coupe d'un tas de cartes.....	42
Figure 27. Paquetage et classes pour la simulation de systèmes multitâches	46

Table des tableaux

Tableau 1. Résumé de la classe Pile.....	5
Tableau 2. Résumé de la classe MainPile	8
Tableau 3. Résumé de la classe ElémentDePile	9
Tableau 4. Résumé de la classe MainElémentDePile	13
Tableau 5. Résumé de la classe ElémentDePileEvoluée	16
Tableau 6. Résumé de la classe MainElémentDePileEvoluée.....	18
Tableau 7. Résumé de la classe File.....	20
Tableau 8. Résumé de la classe MainFile	23
Tableau 9. Résumé de la classe ElémentDeFile	24
Tableau 10. Résumé de la classe MainElémentDeFile	28
Tableau 11. Résumé de la classe ElémentDeFileEvoluée	31
Tableau 12. Résumé de la classe MainElémentDeFileEvoluée.....	34
Tableau 13. Résumé de la classe Carte	37
Tableau 13. Désignation de la couleur et de la valeur d'une carte par des entiers.....	37
Tableau 15. Résumé de la classe ElémentDeTas	38
Tableau 16. Résumé de la classe Belote.....	40
Tableau 17. Résumé de la classe MainPartie	45
Tableau 18. Résumé de la classe Processus.....	47
Tableau 19. Résumé de la classe ElémentDeTourniquet.....	49
Tableau 20. Résumé de la classe Processeur	51
Tableau 21. Résumé de la classe MainSystèmeDExploitation.....	52