Chapitre 3
Interface et Implémentation

Ce chapitre s'intéresse à deux notions fondamentales de l'informatique : l'**interface** et l'**implémentation**. Depuis la Première, nous manipulons des structures de données — tableaux, listes, dictionnaires — sans jamais voir ce qui se passe « à l'intérieur » quand on appelle une méthode comme `append` ou `len`. Comprendre la différence entre ce que *propose* une structure de données (son interface) et la façon dont c'est *réalisé* (son implémentation) permet d'écrire des algorithmes réutilisables dans n'importe quel langage de programmation.

I. Des structures de données et des langages

Une **structure de données** est une manière d'organiser et de stocker des données pour pouvoir les traiter efficacement : le tableau, la liste, le dictionnaire, la file, la pile, etc. Chaque langage de programmation propose ses propres réalisations de ces structures : ce sont des **implémentations**.

Exemple 3.1.
Les méthodes qui ont été définies dans la classe `list` de Python — `len`, `pop`, `insert`, `append`, `del`, ...— sont ce que l'on appelle une **implémentation** de la structure de données tableau. ■

Il existe de nombreux langages de programmation qui implémentent le type abstrait tableau. Nous avions vu l'année dernière les différences d'implémentation entre les `list` de Python et les `Array` de JavaScript : les deux langages proposent bien un tableau, mais avec des noms de méthodes et des syntaxes différentes.

Exemple 3.2.
Le même type abstrait tableau, implémenté dans deux langages différents : les opérations existent des deux côtés, mais ne s'écrivent pas de la même façon. ■

Opération	Python (`list`)	JavaScript (`Array`)
Longueur	`len(tab)`	`tab.length`
Ajouter en fin	`tab.append(x)`	`tab.push(x)`
Insérer à l'index `i`	`tab.insert(i, x)`	`tab.splice(i, 0, x)`
Retirer le dernier	`tab.pop()`	`tab.pop()`
Retirer à l'index `i`	`del tab[i]`	`tab.splice(i, 1)`

II. L'implémentation

Définition 3.1.
L'**implémentation** d'une structure de données ou d'un algorithme est une mise en œuvre pratique dans un langage de programmation.

Lorsque l'on écrit `len(tab)` en Python, on ne sait pas — et on n'a pas besoin de savoir — comment Python compte les éléments : une implémentation est cachée derrière chaque opération. De même, un algorithme décrit une méthode de résolution indépendamment du langage ; son implémentation est le programme concret qui le réalise.

■ **Exemple 3.3.**
L'algorithme « compter les occurrences d'une valeur dans une liste » et son implémentation en Python : on parcourt la liste et on incrémente un compteur à chaque fois que l'élément vaut la valeur cherchée. ■

```
def compter_occurrences(liste, valeur):
    compteur = 0
    for element in liste:
        if element == valeur:
            compteur = compteur + 1
    return compteur
```

■ **Exemple 3.4.**
Le même algorithme, implémenté autrement : on parcourt la liste par les indices avec une boucle `while` au lieu d'une boucle `for`. Le résultat est identique. ■

```
def compter_occurrences(liste, valeur):
    compteur = 0
    i = 0
    while i < len(liste):
        if liste[i] == valeur:
```

```
            compteur = compteur + 1
        i = i + 1
    return compteur
```

R Ces deux fonctions ont la même **interface** (même nom, mêmes paramètres, même résultat) mais des **implémentations** différentes. Un programme qui utilise la fonction `compter_occurrences` n'a pas besoin de savoir laquelle des deux versions est utilisée : il lui suffit de connaître son nom, ses paramètres et ce qu'elle renvoie.

III. L'interface

Cependant, on retrouve des méthodes similaires qui sont ce que l'on appelle l'**interface** de la structure de données tableau :

1. « Insérer » : ajoute un élément dans le tableau à l'index souhaité. `ajout(index, élément)`
2. « Retirer » : retire un élément dans le tableau à l'index souhaité. `suppr(index)`
3. « Le tableau est-il vide » : renvoie Vrai si le tableau est vide et faux sinon. `est_vide()`
4. « Nombre d'éléments dans le tableau » : renvoie le nombre d'éléments dans le tableau. `longueur()`

■ Exemple 3.5.
L'interface de la structure de données tableau se résume à quatre méthodes, dont on connaît le nom, les paramètres et le comportement attendu. ■

Méthode	Rôle
`ajout(index, element)`	Ajoute `element` dans le tableau à l'index `index`
`suppr(index)`	Retire du tableau l'élément situé à l'index `index`
`est_vide()`	Renvoie `True` si le tableau est vide, `False` sinon
`longueur()`	Renvoie le nombre d'éléments contenus dans le tableau

Définition 3.2.
L'**interface** d'une structure de données est la spécification des méthodes pouvant être appliquées sur cette structure de données.

■ Exemple 3.6.
L'interface du tableau se retrouve dans tous les langages qui implémentent le type abstrait tableau : en Python comme en JavaScript, on peut insérer un élément,

en retirer un, tester si le tableau est vide et compter ses éléments. L'interface est commune ; l'implémentation diffère. ■

> **R** L'interface fixe le *quoi* : le nom de la méthode, ses paramètres et le comportement attendu. Elle ne dit rien du *comment* : la réalisation interne reste libre. C'est précisément ce qui permet de changer d'implémentation sans changer le reste du programme.

IV. Distinguer interface et implémentation

■ **Exemple 3.7.**
Une analogie courante : la télécommande d'un téléviseur. Son **interface**, ce sont les boutons : allumer, éteindre, changer de chaîne, régler le volume. Son **implémentation**, c'est l'électronique à l'intérieur du téléviseur. On peut remplacer entièrement l'électronique — voire passer d'un téléviseur à tube cathodique à un écran plat — sans changer les boutons : l'utilisateur ne s'aperçoit de rien. ■

■ **Exemple 3.8.**
Deux implémentations de la même méthode `est_vide` : la première compare la longueur à zéro, la seconde compare la liste à la liste vide. Même interface, deux réalisations. ■

```
def est_vide(self):
    return len(self.data) == 0
```

```
def est_vide(self):
    return self.data == []
```

Pourquoi distinguer l'interface de l'implémentation ? Parce que cette distinction permet :

- d'écrire des algorithmes sans dépendre d'un langage particulier, en utilisant le **pseudo-code** (section suivante) ;
- de **changer l'implémentation** d'une structure de données sans modifier le reste du programme : c'est le principe de **modularité** ;
- d'utiliser une structure de données sans connaître son fonctionnement interne, exactement comme on utilise une **bibliothèque** ou une **API** (*Application Programming Interface*) en ne connaissant que les fonctions qu'elle expose.

> **R** Le programme de Terminale établit explicitement le lien avec la rubrique « langages et programmation » : une bibliothèque expose une interface (les fonctions que l'on peut appeler) et cache son implémentation (le code qui les réalise). C'est ce qui permet de réutiliser du code sans avoir à le comprendre entièrement.

V. Le pseudo-langage

Cela permet d'expliquer l'intérêt du pseudo-langage (ou pseudo-code). L'intérêt de définir des structures de données avec une interface commune est de pouvoir écrire des algorithmes sur le papier en utilisant l'interface définie. On utilise alors un pseudo-langage plus ou moins proche de la langue naturelle qui pourra être implémenté dans tous les langages de programmation ayant défini la structure de données.

■ Exemple 3.9.
Le calcul de la moyenne d'une liste, écrit en pseudo-code, puis implémenté en Python. Le pseudo-code utilise le langage naturel ; la traduction Python utilise la structure de données liste. ■

```
somme = 0
pour chaque valeur v de la liste :
    somme = somme + v
renvoyer somme / longueur(liste)
```

```
def moyenne(liste):
    somme = 0
    for v in liste:
        somme = somme + v
    return somme / len(liste)
```

■ Exemple 3.10.
La recherche du maximum d'un tableau, écrite en pseudo-code avec l'interface du tableau (accéder à un élément, connaître la longueur), puis implémentée en Python avec une liste. ■

```
maxi = element(tableau, 0)
pour i de 1 a longueur(tableau) - 1 :
    si element(tableau, i) > maxi :
        maxi = element(tableau, i)
renvoyer maxi
```

```
def maximum(liste):
    maxi = liste[0]
    for i in range(1, len(liste)):
        if liste[i] > maxi:
            maxi = liste[i]
    return maxi
```

R Le pseudo-code écrit avec l'interface du tableau — `element(tableau, i)`, `longueur(tableau)` — se traduit en Python par `liste[i]` et `len(liste)` : l'implémentation change, l'algorithme reste le même. C'est tout l'intérêt d'une interface commune.

VI. Un exemple complet : la classe Tableau

On spécifie une structure de données par son interface, puis on l'implémente. Implémentons en Python la structure de données `Tableau` dont on a fixé l'interface : les données sont stockées dans un attribut `data` de type liste.

Exemple 3.11.
La classe `Tableau` implémente l'interface du tableau : `ajout`, `suppr`, `est_vide` et `longueur`. Chaque méthode de l'interface est réalisée à l'aide d'une opération de la classe `list` de Python. ■

```
class Tableau:
    def __init__(self):
        self.data = []

    def ajout(self, index, element):
        self.data.insert(index, element)

    def suppr(self, index):
        del self.data[index]

    def est_vide(self):
        return len(self.data) == 0

    def longueur(self):
        return len(self.data)
```

■ **Exemple 3.12.**
Utilisation de la classe `Tableau` : on crée un tableau, on insère des éléments, puis on interroge la structure avec les méthodes de l'interface. ■

```
tab = Tableau()
tab.ajout(0, "a")
tab.ajout(1, "b")
tab.ajout(0, "c")
print(tab.est_vide())
print(tab.longueur())
```

Ce programme affiche `False` puis `3` : après les trois insertions, `tab.data` vaut `["c", "a", "b"]`, le tableau n'est pas vide et contient trois éléments.

R La classe `list` de Python sait déjà faire tout cela avec `insert`, `del` et `len`. La classe `Tableau` est une **enveloppe** : elle expose l'interface choisie (`ajout`, `suppr`, `est_vide`, `longueur`) et cache la façon dont les données sont stockées. Tout programme qui utilise la classe `Tableau` ne dépend pas de l'implémentation : on pourrait la remplacer sans rien modifier d'autre.

R On remarque que l'interface du tableau définie plus haut ne permet pas de **consulter** un élément : elle a été conçue pour insérer, retirer, tester la vacuité et compter. Une interface se construit selon les besoins du programme : si l'on veut lire un élément, on l'enrichit avec une méthode telle que `element(index)`. C'est ce que proposent les exercices d'application.

VII. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à écrire des programmes propres et à tester vos classes en créant des objets.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : vocabulaire de l'interface et de l'implémentation, lecture de code court, prédiction d'affichage, classement. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 3.1.

Dire si les affirmations suivantes sont vraies ou fausses.

1. L'implémentation d'une structure de données est une mise en œuvre pratique dans un langage de programmation.
2. L'interface d'une structure de données décrit comment les méthodes sont codées en interne.
3. `len` est une implémentation de la structure de données tableau proposée par Python.
4. On peut écrire un algorithme en pseudo-code puis l'implémenter dans plusieurs langages de programmation.
5. L'interface et l'implémentation désignent exactement la même chose.

■

■ Exercice 3.2.

Associer chaque méthode de l'interface du tableau à son rôle.

1. `ajout(index, element)`
2. `suppr(index)`
3. `est_vide()`
4. `longueur()`

— Renvoie `True` si le tableau est vide, `False` sinon.
— Ajoute un élément dans le tableau à l'index souhaité.
— Renvoie le nombre d'éléments contenus dans le tableau.
— Retire du tableau l'élément situé à l'index souhaité.

■

■ Exercice 3.3.

On exécute le programme suivant, la classe `Tableau` étant celle du cours :

```
tab = Tableau()
tab.ajout(0, 10)
```

```
tab.ajout(0, 20)
print(tab.longueur())
print(tab.est_vide())
```

Que va afficher ce programme ? Justifier en décrivant le contenu de `tab.data` après chaque instruction. ■

■ **Exercice 3.4.**
Pour chacune des affirmations suivantes, dire si elle relève de l'**interface** ou de l'**implémentation** de la structure `Tableau`.

1. « `ajout(index, element)` ajoute un élément dans le tableau à l'index souhaité. »
2. « Les éléments sont stockés dans une liste nommée `data`. »
3. « `est_vide()` renvoie `True` si le tableau est vide, `False` sinon. »
4. « La méthode `longueur` utilise la fonction `len` de Python. »

■

■ **Exercice 3.5.**
On exécute les instructions suivantes (rappel de Première sur les méthodes de la classe `list`) :

```
liste = [1, 2, 3]
liste.insert(1, 9)
liste.append(7)
dernier = liste.pop()
```

Que contient la liste `liste` à la fin ? Que vaut la variable `dernier` ? ■

Exercices d'application

Les exercices d'application demandent d'implémenter une structure de données à partir de son interface, d'enrichir une classe et de traduire un pseudo-code en Python.

■ **Exercice 3.6.**
Créer une classe `Tableau` qui implémente les quatre méthodes de l'interface du cours en stockant les données du tableau dans un attribut appelé `data` de type liste.

```
class Tableau:
    def __init__(self):
        self.data = []
```

Tester ensuite la classe en créant un tableau, en insérant quelques éléments et en affichant sa longueur. ■

■ **Exercice 3.7.**
On reprend la classe `Tableau` de l'exercice précédent. L'interface d'une structure se construit selon les besoins : on veut pouvoir **consulter** un élément.

1. Ajouter une méthode `element(index)` qui renvoie l'élément situé à l'index donné.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères décrivant le contenu du tableau, par exemple `[a, b, c]`.
3. Tester ces deux méthodes sur un objet de votre choix.

■

■ **Exercice 3.8.**
On suppose que la classe `Tableau` possède la méthode `element(index)` ajoutée à l'exercice précédent. Écrire une fonction `contient(tableau, valeur)` qui renvoie `True` si `valeur` est présente dans le tableau, `False` sinon, en n'utilisant *que* les méthodes de l'interface de la classe `Tableau`. ■

■ **Exercice 3.9.**
Traduire en Python le pseudo-code suivant, qui compte les occurrences d'une valeur dans un tableau, en utilisant la classe `Tableau` du cours (avec sa méthode `element`).

```
compteur = 0
pour i de 0 a longueur(tableau) - 1 :
    si element(tableau, i) == valeur :
        compteur = compteur + 1
renvoyer compteur
```

■

■ **Exercice 3.10.**
Au chapitre 2, on a étudié la structure de données **pile** : une structure linéaire où l'on retire toujours le dernier élément ajouté (mode LIFO).

1. Spécifier l'interface d'une pile : proposer les méthodes `empiler(element)`, `depiler()` et `est_vide()`, et décrire le comportement attendu de chacune.
2. Implémenter cette interface dans une classe `Pile` dont l'attribut `data` est une liste.
3. Tester la classe en empilant quelques éléments, en les dépilant un à un et en affichant le résultat.

■

Exercices de réflexion

Les exercices de réflexion demandent de concevoir des interfaces, d'écrire plusieurs implémentations d'une même structure de données et d'analyser l'intérêt de la séparation interface/implémentation.

■ **Exercice 3.11.**
On veut écrire une **autre implémentation** de la structure de données `Tableau` : cette fois, l'attribut `data` est une **chaîne de caractères** au lieu d'une liste. On rappelle que l'on peut insérer le caractère `c` à l'index `i` d'une chaîne `s` par l'affectation `s = s[:i] + c + s[i:]`.

1. Compléter la classe suivante, qui a exactement la même interface que la classe `Tableau` du cours.

```
class TableauStr:
    def __init__(self):
        self.data = ""

    def ajout(self, index, element):
        ...

    def suppr(self, index):
        ...

    def est_vide(self):
        ...

    def longueur(self):
        ...
```

2. Tester cette nouvelle classe avec le même programme que l'exemple du cours : les résultats doivent être identiques.

■

■ **Exercice 3.12.**
On a écrit un programme qui utilise uniquement les méthodes de l'interface de la classe `Tableau` (créer un tableau, insérer, retirer, tester la vacuité, compter).

1. Pourquoi ce programme n'a-t-il pas besoin d'être modifié si l'on remplace la classe `Tableau` par la classe `TableauStr` de l'exercice précédent ?
2. Quel est l'intérêt, pour un développeur, de pouvoir changer l'implémentation d'une structure de données sans toucher au reste du programme ? Donner un exemple concret où ce changement est utile.

■

■ **Exercice 3.13.**
On veut modéliser un carnet d'adresses avec une structure de données `Carnet`.

1. Écrire l'**interface** de cette structure : proposer les méthodes (nom, paramètres, comportement attendu) qui permettent d'ajouter un contact, de

supprimer un contact, de rechercher un contact par son nom et de savoir si le carnet est vide. On n'écrira *pas* le code des méthodes.
2. Implémenter cette interface avec une classe `Carnet` dont l'attribut `data` est un dictionnaire associant chaque nom de contact à son numéro de téléphone (rappel de Première).
3. Expliquer pourquoi un programme qui utilise l'interface de `Carnet` fonctionnerait encore si l'on décidait de stocker les contacts dans une liste de couples (nom, numéro) à la place d'un dictionnaire.

■

■ **Exercice 3.14.**

On s'intéresse au **coût** des opérations de la classe `Tableau` du cours.

1. L'opération `ajout(index, element)` est réalisée par `self.data.insert(index, element)`. Pourquoi `insert(0, x)` — insérer en début de liste — est-il plus coûteux que `append(x)` — ajouter en fin ? On pourra penser à ce qui se passe pour les éléments déjà présents dans la liste.
2. Une liste Python est un tableau contigu en mémoire : pour insérer un élément au début, il faut décaler tous les éléments d'un cran. En déduire, pour une liste de n éléments, une estimation du nombre de décalages nécessaires pour `insert(0, x)`.
3. On veut construire un programme qui insère très souvent un élément au début d'un tableau. D'après vos réponses, quelle implémentation de `ajout` serait préférable ? Justifier.

■