

Chapter 12: Structures de données

Presentation slides for

Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley

Presentation slides are copyright 2000 by John Lewis and William Loftus. All rights reserved.

Instructors using the textbook may use and modify these slides for pedagogical purposes.

Structures de données

- ⌚ Nous pouvons maintenant explorer certaines techniques avancées pour organiser et gérer de l'information
- ⌚ Chapitre 12 se concentre sur:
 - Structures dynamiques
 - Les types abstraits de données (« Abstract Data Types (ADTs) »)
 - Listes chaînées
 - Queues
 - Piles

Structures statiques vs. dynamiques

- ❧ Une structure de données *statique* a une taille fixe
- ❧ Ici, la signification de *statique* est différente de la signification de `static` du Java
- ❧ Les tableaux sont statiques. En effet, une fois que le nombre d'éléments que le tableau peut contenir soit défini, il ne change pas dans la suite
- ❧ La taille d'une structure de données dynamique change dépendamment du changement du nombre d'informations qu'elle peut contenir

Références sur des objets

- ❧ Rappelons qu'une référence sur un objet est une variable qui stocke l'adresse de cet objet
- ❧ Une référence peut aussi être appelée un *pointeur*
- ❧ Une référence est souvent représentée graphiquement par:

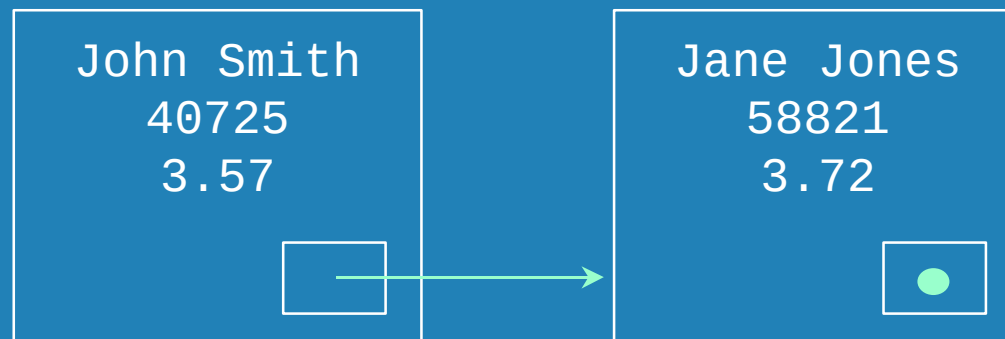
student



John Smith
40725
3.57

Références comme liens

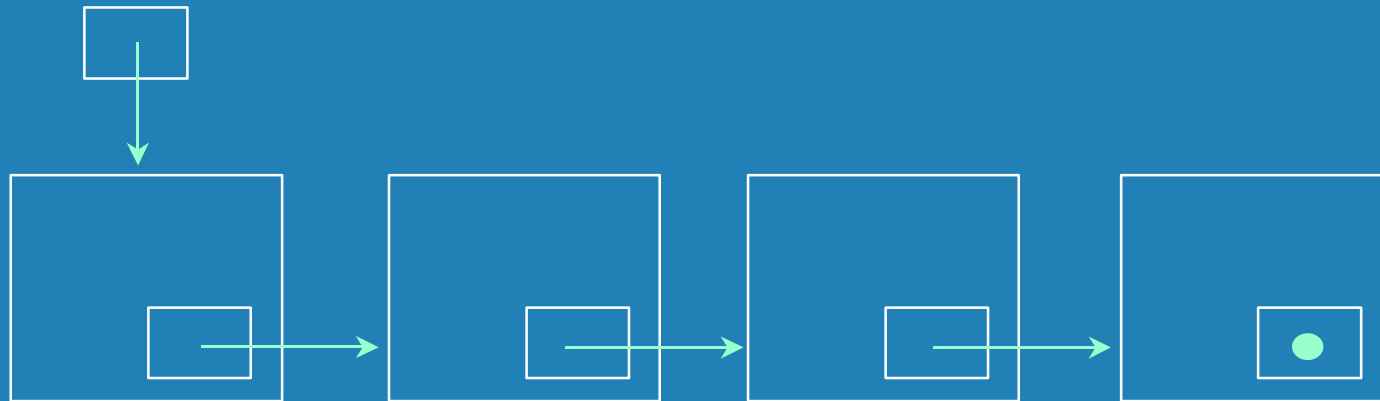
- ❧ Les références sur les objets peuvent être utilisés pour créer des liens entre des objets
- ❧ Supposons qu'une classe appelée **Student** contient une référence sur un autre objet **Student**



Références comme liens

- Les références peuvent être utilisées pour créer une variété de structures. Par exemple, pour créer une liste chaînée

studentList



Types abstraits de données (ADTs)

- ❧ Un type abstrait de données est une collection organisée d'information et un ensemble d'opérations à utiliser pour gérer cette information
- ❧ L'interface est définie par l'ensemble des opérations de l'ADT
- ❧ Tant que l'ADT répond correctement aux besoins de son monde extérieur (réponse via l'interface du ADT), alors il n'est pas important de connaître les détails de son implémentation
- ❧ Les objets sont un mécanisme de programmation parfait pour créer des ADTs, puisque leurs détails internes sont *encapsulés*

Abstraction

- ⌚ Nos structures de données doivent être des abstractions
- ⌚ C'est-à-dire, elles doivent cacher des détails
- ⌚ Nous voulons séparer l'interface de la structure de l'implémentation de celle-ci
- ⌚ Ceci permet de gérer la complexité et de faire des structures plus utiles

Nœuds intermédiaires

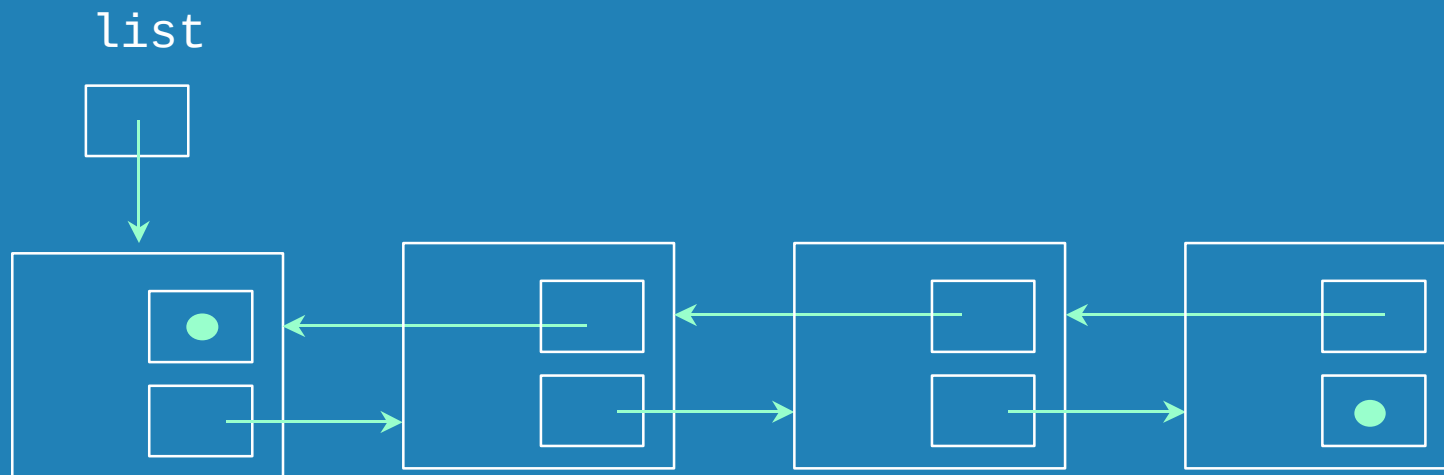
- ❧ Les objets en cours de stockage dans une structure de données ne doivent pas avoir de relations avec les détails de cette structure de données
- ❧ Par exemple, la classe `Student` stocke un lien sur le prochain objet `Student` dans la liste
- ❧ Au lieu de cela, nous pouvons utiliser une classe nœud séparée qui contient une référence sur l'objet stocké et un lien sur le prochain nœud de la liste
- ❧ Donc, la représentation interne devient en fait une liste chaînée de nœuds

Collection de livres

- ❧ Laissons-nous explorer un exemple d'une collection d'objets **Book**
- ❧ Cette collection est gérée par la classe **BookList** qui a une classe privée appelée **BookNode**
- ❧ Car **BookNode** est privée à **BookList**, les méthodes de **BookList** peuvent accéder directement les données de **BookNode** sans violer l'encapsulation
- ❧ Voir **Library.java** (page 500)
- ❧ Voir **BookList.java** (page 501)
- ❧ Voir **Book.java** (page 503)

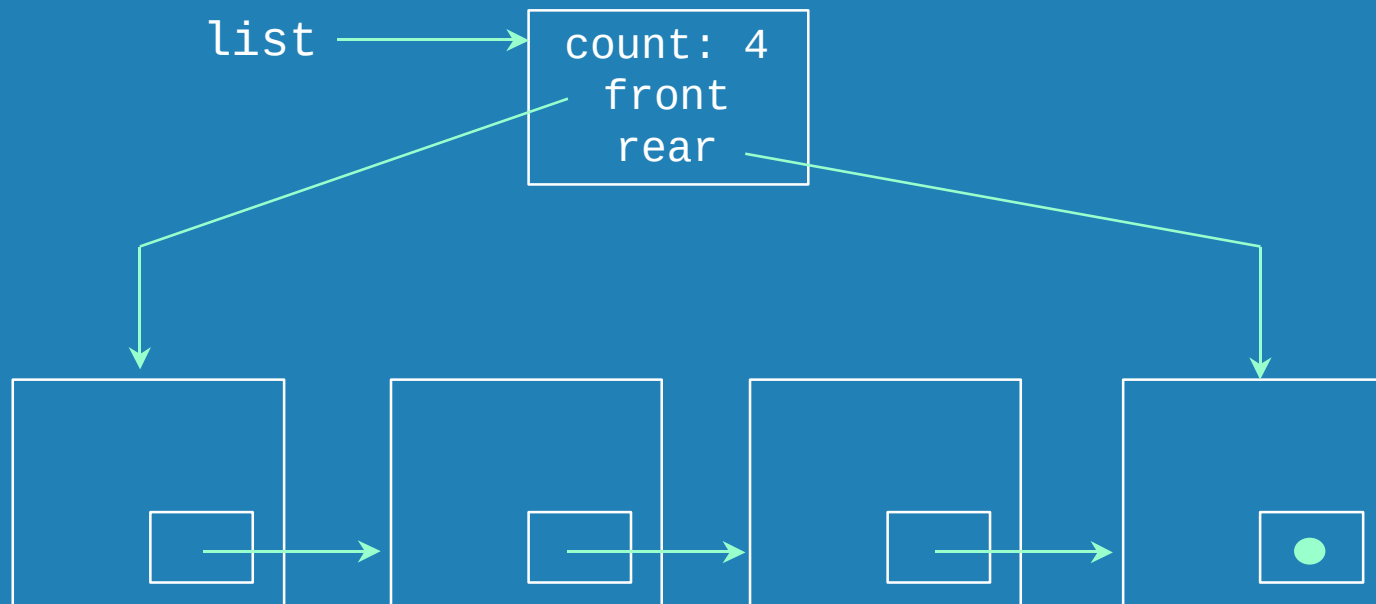
Autres implémentations de listes dynamiques

- ❧ Dans certains cas, il est pratique d'implémenter une liste comme une liste doublement chaînée, avec une référence successeur et une référence prédécesseur



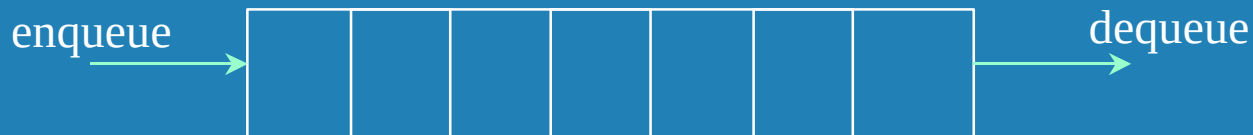
Autres implémentations de listes dynamiques

- ⌚ Dans certains autres cas, il est pratique d'utiliser un nœud séparé *tête* qui réfère au début et à la fin de la liste



Queues

- ❧ Une queue est semblable à une liste, mais on ne peut ajouter un élément qu'à la fin de la liste et on ne peut supprimer un élément qu'à partir du début de la liste
- ❧ Elle est appelée une structure de données FIFO (« First-In, First-Out»)
- ❧ Analogie: une ligne de gens devant un guichet d'une banque



Queues

⌚ Les opérations sur une queue peuvent être:

- enqueue – ajouter un élément à la fin de la queue
- dequeue – supprime un élément à partir du début de la queue
- empty – retourne vrai si la queue est vide

⌚ Comme avec l'exemple de liste chaînée, en stockant des références sur des objets génériques, n'importe quel objet peut être stocké dans la queue

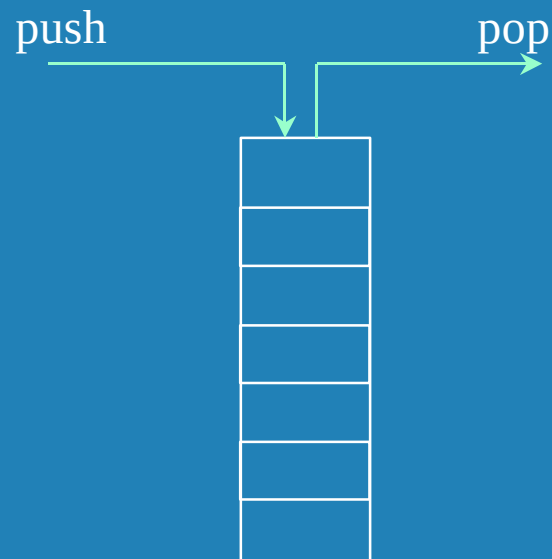
⌚ Les Queues sont souvent utilisées en simulations

Piles « Stacks »

- ⌚ Comme une liste ou une pile, un ADT pile est linéaire
- ⌚ Les éléments sont ajoutés et supprimés à partir seulement d'une des extrémités de la pile
- ⌚ Elle est donc LIFO (« Last-In, First-Out »)
- ⌚ Analogie: une pile d'assiettes

Pile (« Stacks »)

Les piles sont souvent schématisées verticalement:



Pile (« Stacks »)

❧ Des opérations sur les piles peuvent être:

- push – ajouter un élément au sommet de la pile
- pop - supprimer un élément du sommet de la pile
- peek - extraire l'élément du sommet de la pile
- empty – retourne vide si la pile est vide

❧ Le `java.util` package contient une classe `Stack` qui est implémentée en utilisant `Vector`

❧ Voir `Decode.java` (page 508)