Les structures de données
Abtraction : pile et file

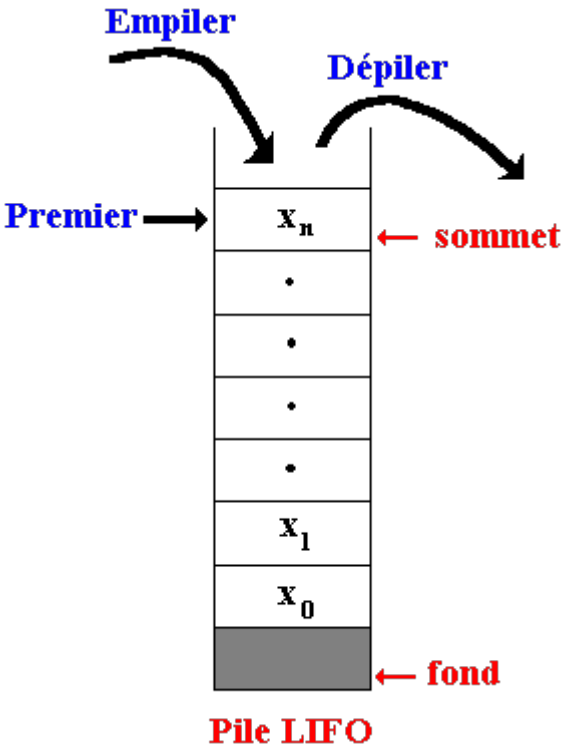
Abstraction et Modularité

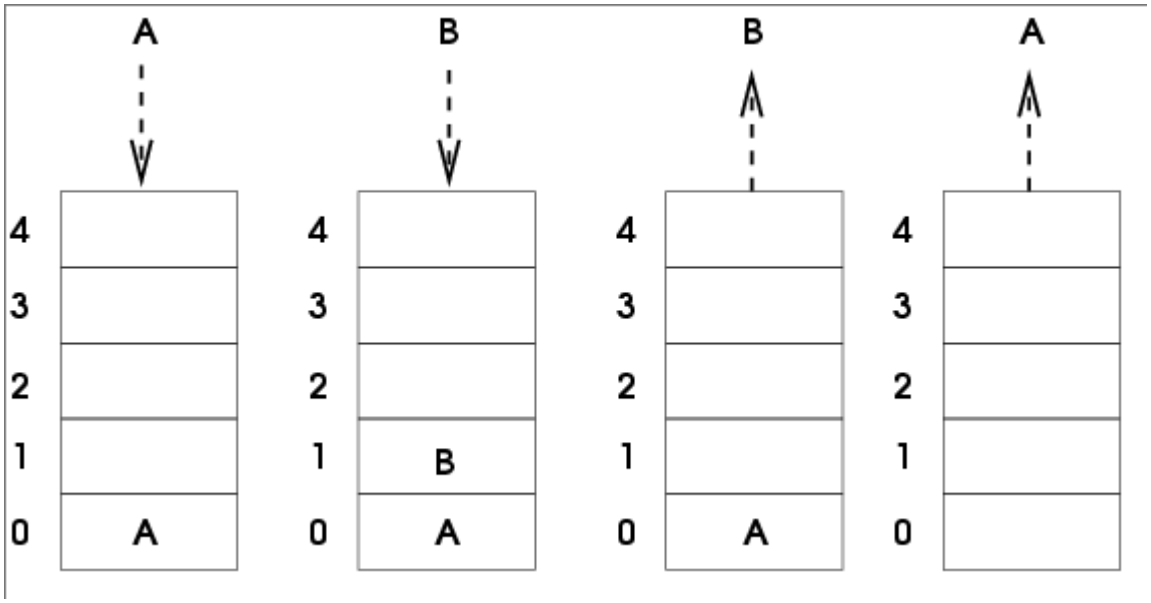
- Dès qu'il devient trop long, un programme doit être divisé en parties cohérentes et indépendantes. On parle alors de **modularité**
  - En Java, javadoc explicitent le fonctionnement des bibliothèques (API sont vus comme des interfaces)
  - D'autres langages ont des fichiers spécifiques à écrire (OCaml, Ada, etc.) ;
- L'**abstraction** a pour but de « *cacher* » fonctionnement interne des API (des classes) ; Ceci pour limiter leurs usages (sûreté, sécurité, *etc.*) avec des **primitives ;** Cacher la réalisation c'est abstraire la structure de donnée qui n'est plus visible dans sa **réalisation** mais uniquement à travers les **fonctionnalités (primitives)** définies par son interface. L'implantation concrète peut prendre plusieurs formes qu'on ne devrait pas pouvoir distinguer par la seule utilisation de l'interface (sauf exceptions claires explicitées dans l'API) ; Plus de détails en L2 !

Les piles et les files

- Pour illustrer tout cela, nous allons voir le fonctionnement de 2 structures de données (classiques) : pile et files
- Structures données = moyen de stocker des données avec des méthodes particulières ; la structure varie aussi l'ordonnancement des données Exemple = tableau (tout aligné, accès direct, taille fixe)
- Pile=stack=LIFO (Last In First Out)
- File=queue=FIFO (First In First Out)

Piles





On dit **push** (empiler), **pop** (dépiler), **top** (sommet)

Pourquoi ne pas lire en milieu de pile ? Imaginez une pile d'assiettes…

Primitives des piles

- Nous allons avoir (sémantique) :
  - **Push** ajoute une valeur au sommet de la pile ;
  - **Pop** retire une valeur du sommet de la pile ;
  - **Top** renvoie la valeur présente au sommet de la pile.
- Précisions (il faut bien lire/écrire une API) :
  - Combien d'éléments dans la pile ? N ou infini
  - Quelles erreurs possibles (exceptions) ?
  - Taille optimisée (en mémoire) de la pile ?

Premier Code

```
class Stack {
    int top;
    int[] data;
        // Le constructeur ≡ création d'une pile vide
    public Stack(int capacite){
      data = new int[capacite];
            //créer une nouvelle pile vide de taille maximal capacite
      top = -1;
            //lorsque tu auras empiler un élément tu l'obtiendras à piles[0]
      }
```

```
public int top() { return piles[sommet]; }
public void push(int element){
    data[top+1] = element;
    top++;
}
public void pop(){
    top--;
```

- //inutile de la remplacer par une valeur 0 ou null car c'est avec l'indice sommet qu'on accède aux valeurs dans le tableau (mais le tableau va contenir pleins de valeurs inutiles...)

```
}
```

Pour créer une pile, on écrit :

Stack myStack = new Stack(110);

Exercice

- (1) Recopiez puis créez un programme avec 2 piles de 6 éléments max. Vous mettrez 4 valeurs (lu au clavier) dans la première pile puis vous mettrez toutes ces valeurs dans la seconde (boucle for avec 4 itérations) ; Comment faire pour N valeurs ? Comment connaître ce N ?
- (2) Que se passe-t-il si on tente les opérations suivantes :
  - Appel de Top ou de Pop sur une pile vide ; (top==-1)
  - Appel de Push sur une pile contenant plus de 6 éléments.

  Ce comportement est-il souhaitable ? Expliquez en quoi l'implantation de Push donnée est meilleure que si on faisait :

  ```
  Top=Top+1;
  Data[Top]=element ;
  ```

Rajoutons des primitives

```
public boolean isEmpty(){
      if(top== -1){
        return true;                    ===> boolean isEmpty(){ return (top==-1) }
      }                                 ===> boolean isFull(){ return (top==data.length) }
      return false;
    }
// Affichage sous le forme [1;2;3;4;]
public String toString() {
   String res= "[" ;
   for(int i=0;i<top;i++) { s=s+data[i]+"; "}
   return res+"]"
}
```

Gestion des erreurs

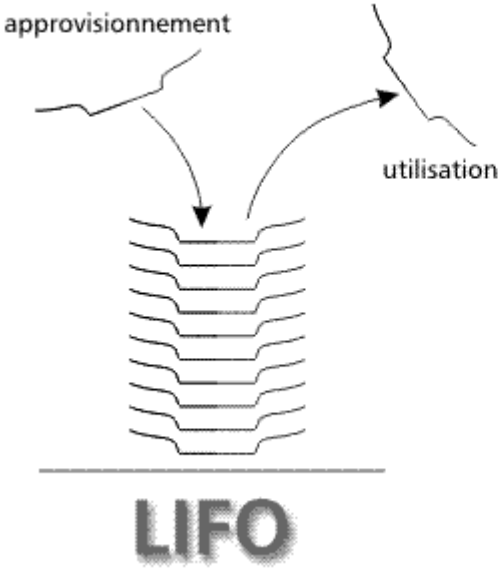
- On va utiliser les exceptions pour gérer les cas problématiques ;

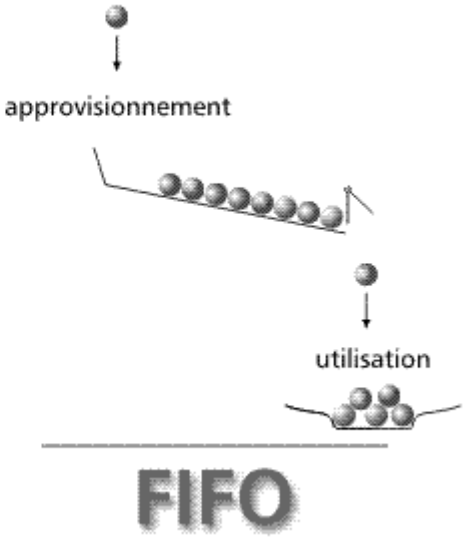
```
public void pop(){
    if (isEmpty()){ throw new Error("Stack is empty"); } ;
    top--;
}
```

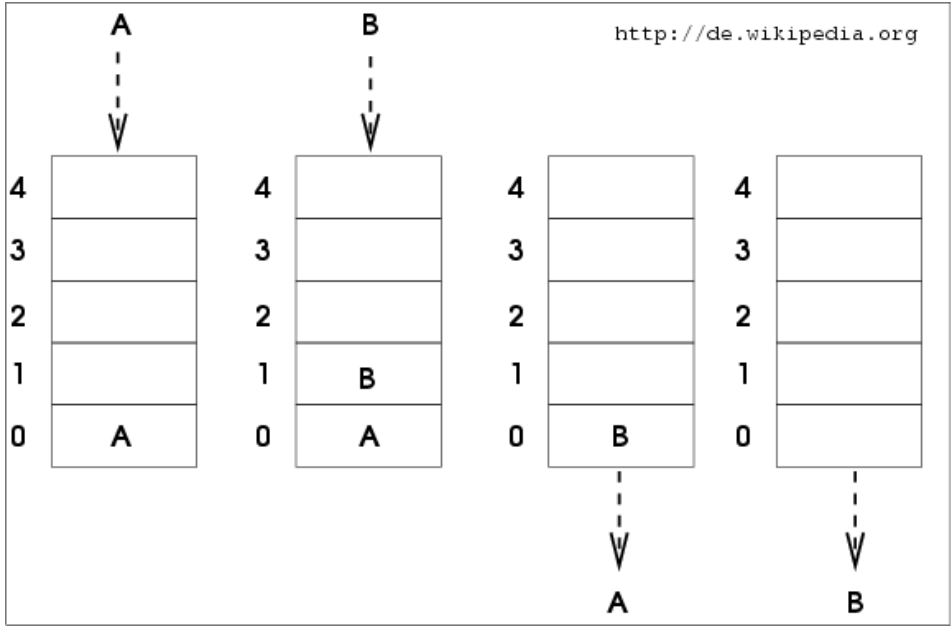
Exercice : faire de même pour « push » ; Reprendre tout les codes et bien documenter (javadoc)

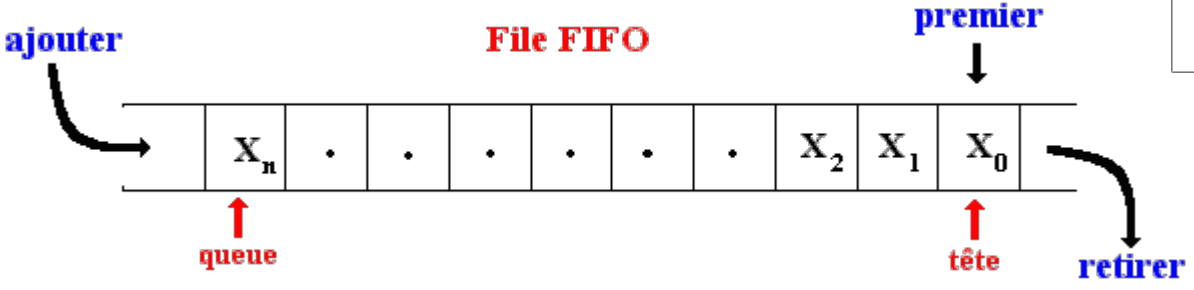
Exercice : reprendre votre précèdent programme et vider toute la pile pour mettre les valeurs dans l'autre pile ; Est-ce qu'une méthode «int size() » ne serait pas utile ?

Les files


approvisionnement
utilisation
LIFO


approvisionnement
utilisation
FIFO


A
B
http://de.wikipedia.org
4
3
2
1
0
A
B
A
B


ajouter
File FIFO
premier
X_n
X_2
X_1
X_0
queue
tête
retirer

Primitives des files

- Nous allons avoir (sémantique) :
  - **Put** ajoute une valeur à la fin de la file ;
  - **Get** retire l'élément du début de la file ;
  - **First** renvoie la valeur se trouvant en tête de la file.
- Exercice :
  - Sur le modèle des piles, écrivez une classe « queue » déclarant un type représentant des files d'entiers et les primitives décrites ci-dessus.
  - Documentez (javadoc) votre travail
  - Quelles erreurs possibles (exceptions) ?