# Cours 4.0: Listes. Piles. Interface/Implements

Olivier Bournez

bournez@lix.polytechnique.fr
LIX, Ecole Polytechnique

2011-12

Algorithmique

# Aujourd'hui

Piles. Files

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

Interface/Implements en JAVA

# Le type abstrait “sac”

- On souhaite considérer une structure de données, que l'on va appeler *sac*, permettant de stocker des `Objets`.

- On veut pouvoir:

  - tester si un sac est vide?
  - ajouter un élément dans un sac.
  - retirer un élément d'un sac non vide.

```
class Sac  {
  ...
  Sac() { ... } //Construire un sac vide
  boolean EstVide() { ... } //Tester si un sac est vide
  void Ajouter(Objet e) { ... } //Ajouter un élément à un sac
  Objet Enlever() { ... } //Enlever un élément d'un sac
}
```

# Les types abstraits

permettent

- de décrire un objet uniquement par ses fonctions de base.
- de faire abstraction de la réalisation.
  - Un sac peut être implémenté par une liste, par un tableau, ...
  - Modifier son implémentation ne modifie pas le reste du programme.

La notion d'interface, d'héritage, et de module sera vue plus avant dans le cours INF431.

# Files. Piles

- Les piles et les files sont des sacs.
- Une pile est un sac LIFO (Last-In First-Out)
  - lorsqu'on retire un élément, on récupère toujours le dernier ajouté.
- Une file est un sac FIFO (First-in First-Out)
  - on retire les éléments dans l'ordre de leur insertion.

# Files. Piles

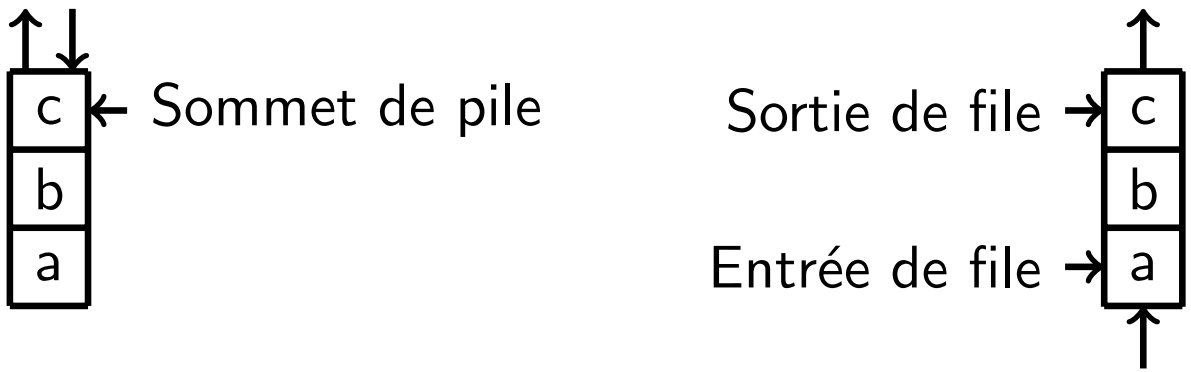
- Les piles et les files sont des sacs.
- Une pile est un sac LIFO (Last-In First-Out)
  - lorsqu'on retire un élément, on récupère toujours le dernier ajouté.
  - ajouter se dit *empiler* (push), retirer se dit *dépiler* (pop).
- Une file est un sac FIFO (First-in First-Out)
  - on retire les éléments dans l'ordre de leur insertion.
  - ajouter se dit *enfiler*, retirer se dit *défiler*.

# Piles, Files d'entiers

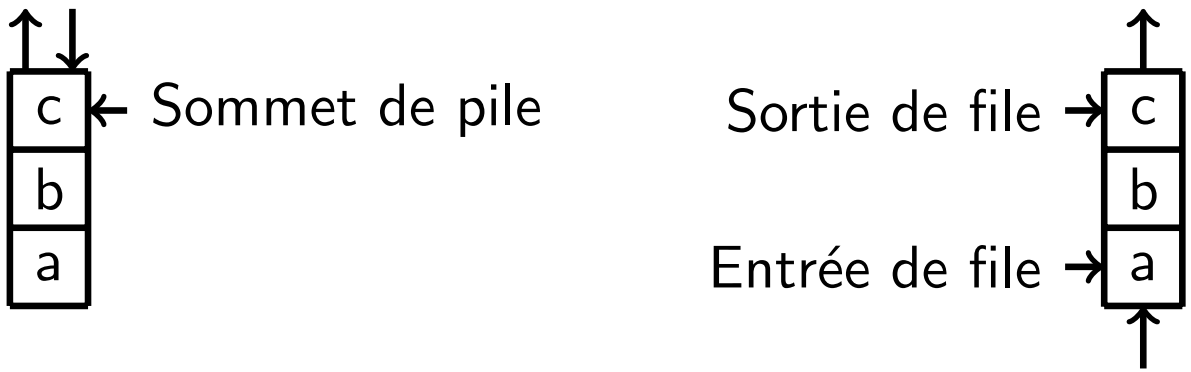
```
class Pile  {
  ...
  Pile() { ... }
  boolean EstVide() { ... }
  int     Tete() { ...}

  void Empiler(int e) { ... }
  int Dépiler() { ... }
}

Pile s= new Pile();
```

```
class File  {
  ...
  File() { ... }
  boolean EstVide() { ... }
  int     Tete() { ...}

  void Enfiler(int e) { ... }
  int Défiler() { ... }
}

File s= new File();
```

# Une pile et une file de caractères

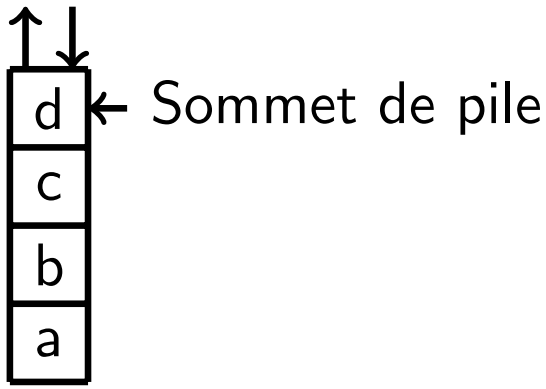


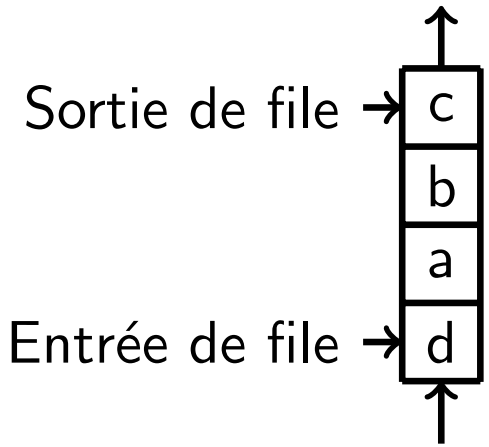
# Une pile et une file de caractères



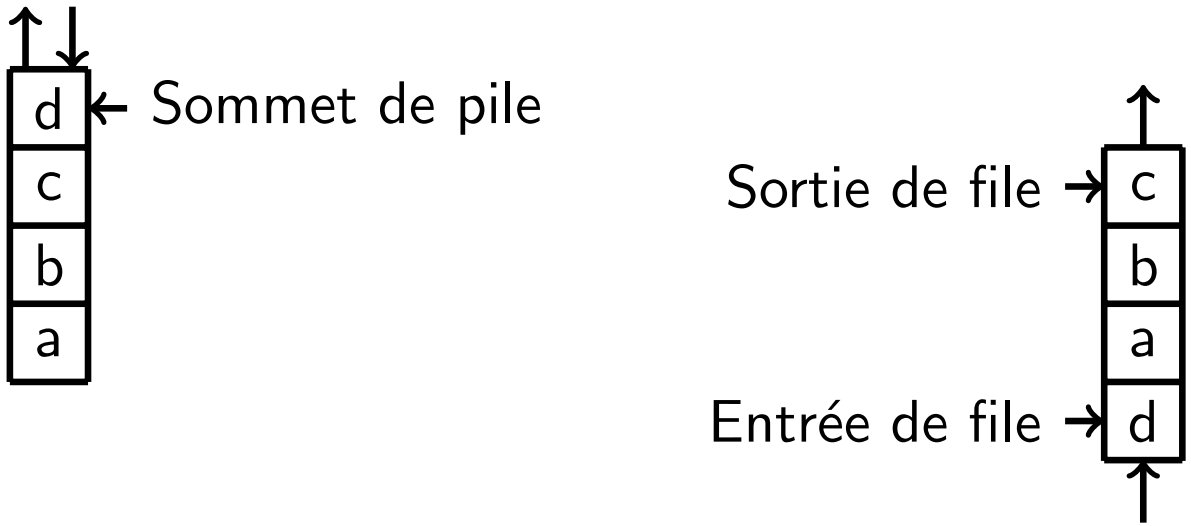

s.Ajouter('d')

# Une pile et une file de caractères



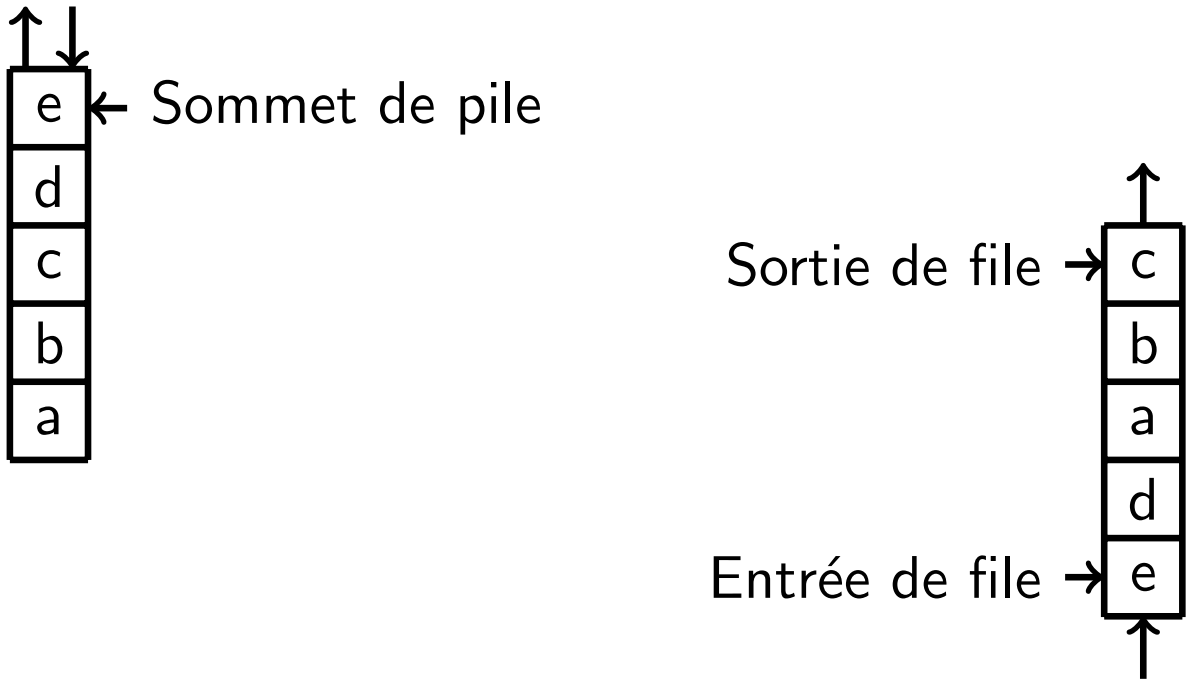



# Une pile et une file de caractères

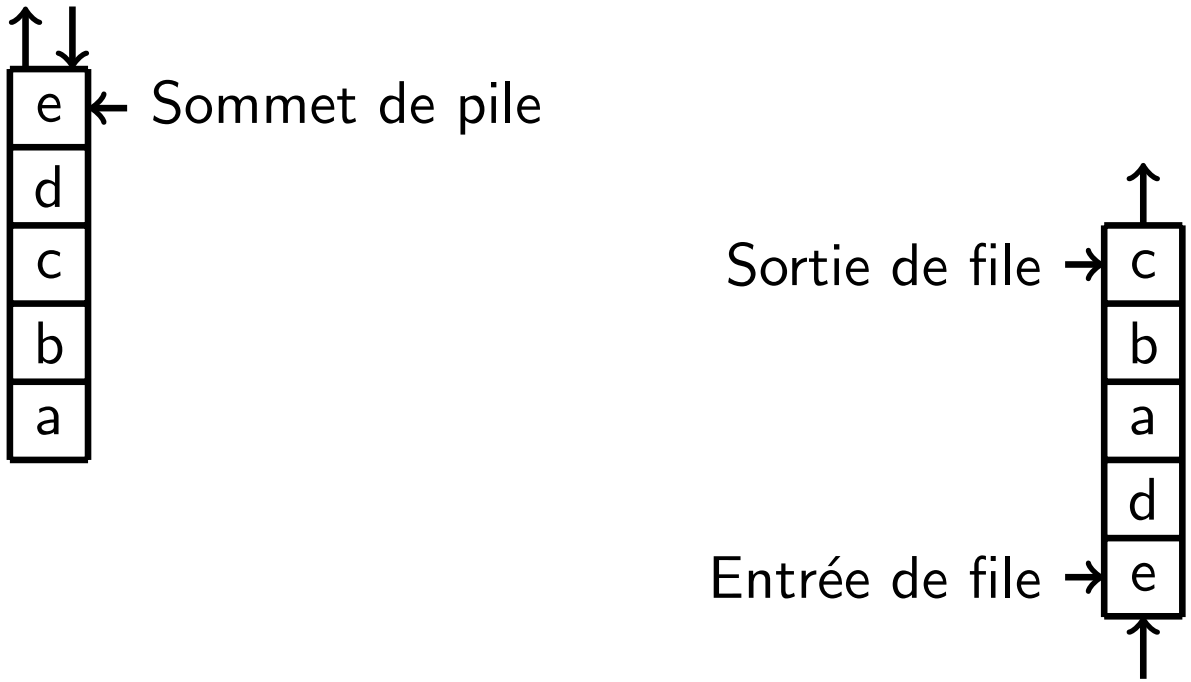



s.Ajouter('e')

# Une pile et une file de caractères

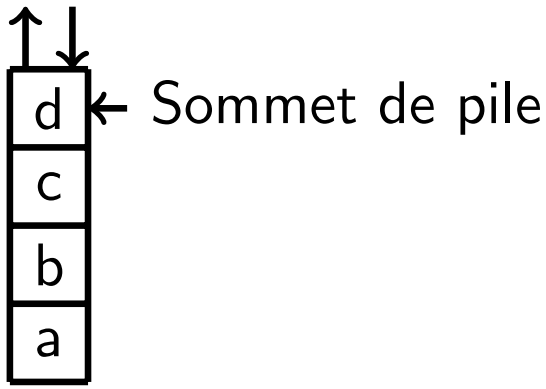


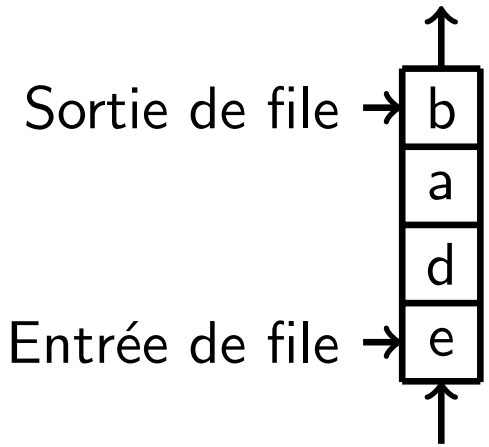
# Une pile et une file de caractères



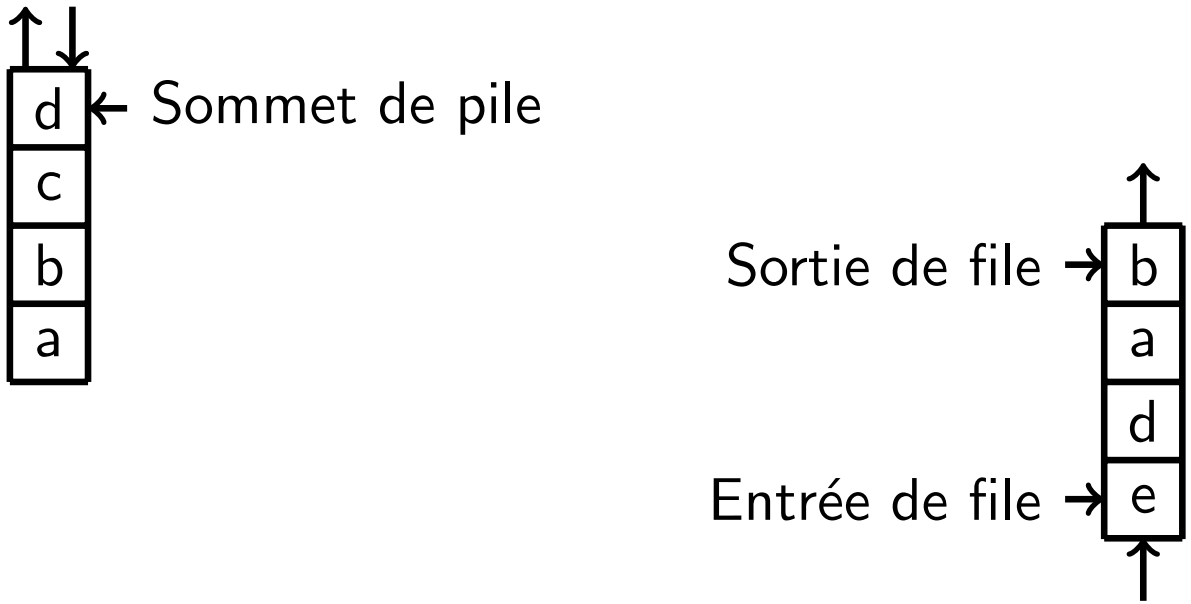

s.Enlever()

# Une pile et une file de caractères



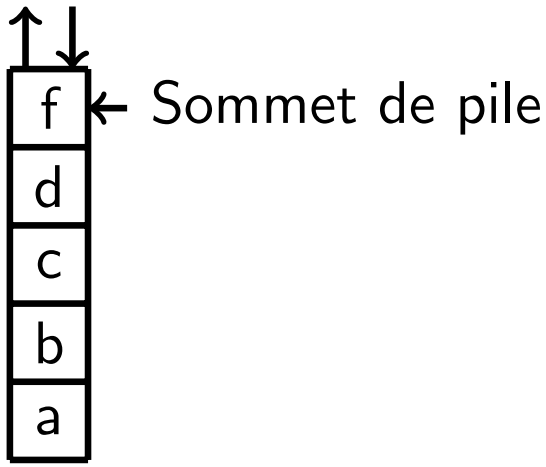



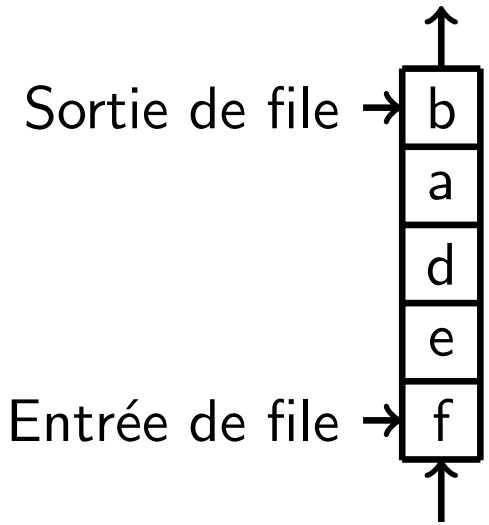
# Une pile et une file de caractères



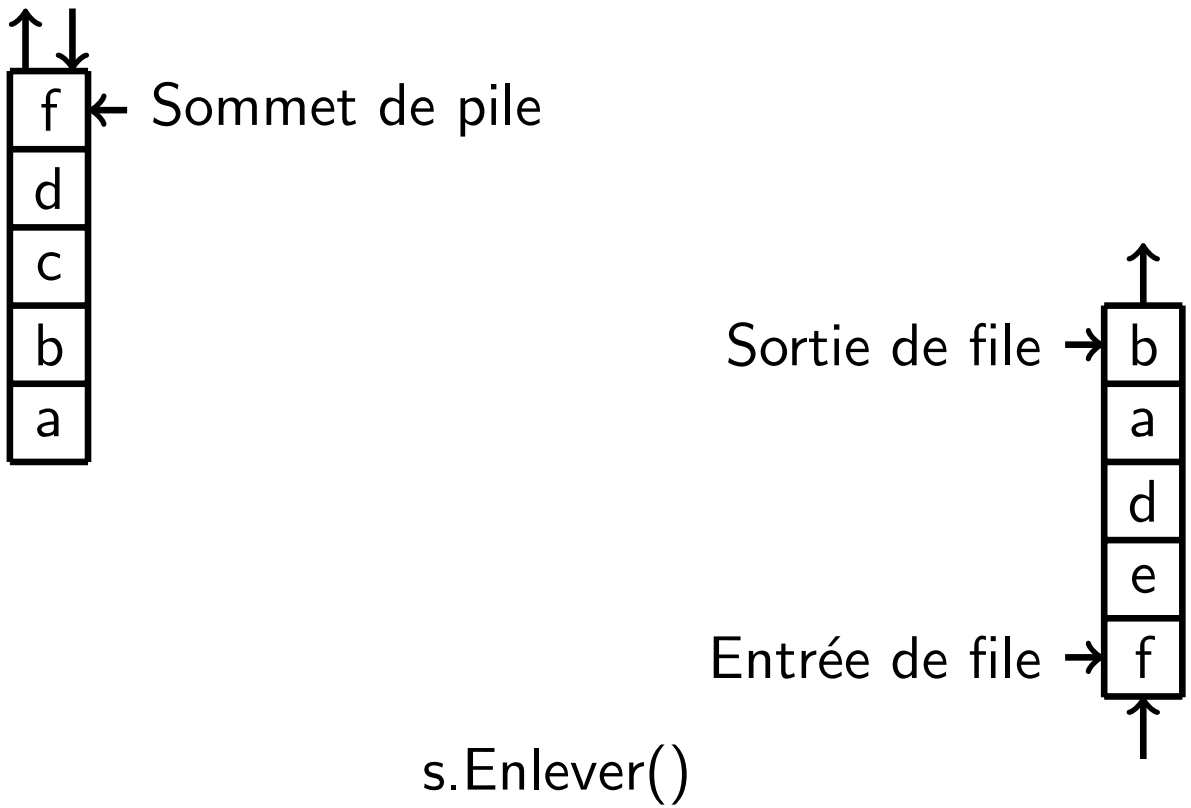

s.Ajouter('f')

# Une pile et une file de caractères

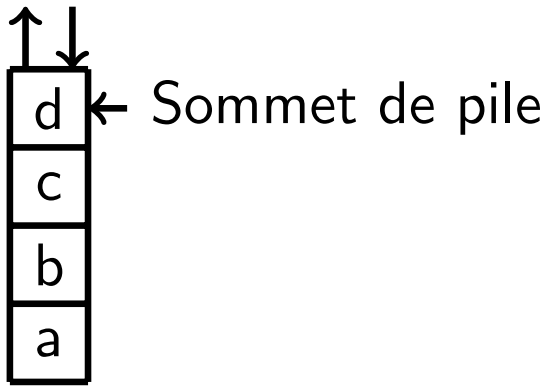


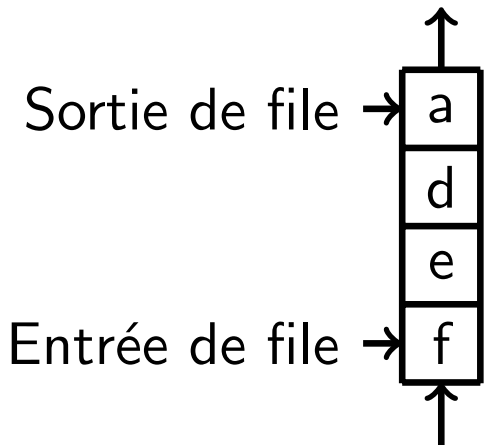



# Une pile et une file de caractères



# Une pile et une file de caractères



# Aujourd'hui

Piles. Files

Utilisation des piles et des files

Programmer des piles et des files

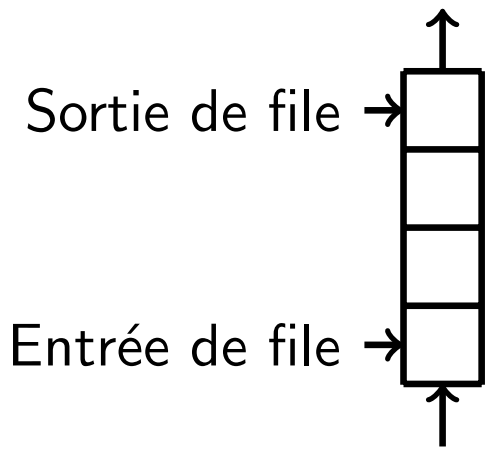
Piles et files dans la bibliothèque JAVA

Interface/Implements en JAVA

# Les files

- Elles sont présentes dès qu'on a besoin d'utiliser ou de modéliser une file d'attente.

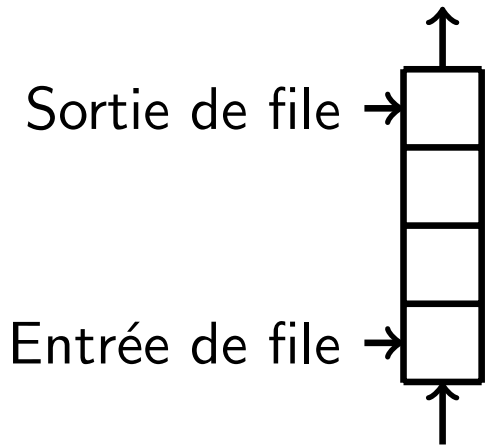
  par exemple:
  - les impressions dans un système d'exploitation sont gérées par une file;
  - dans une simulation d'un guichet, on modélisera généralement l'arrivée des clients par une file.



# Les files

- Elles sont présentes dès qu'on a besoin d'utiliser ou de modéliser une file d'attente.

  par exemple:
  - les impressions dans un système d'exploitation sont gérées par une file;
  - dans une simulation d'un guichet, on modélisera généralement l'arrivée des clients par une file.



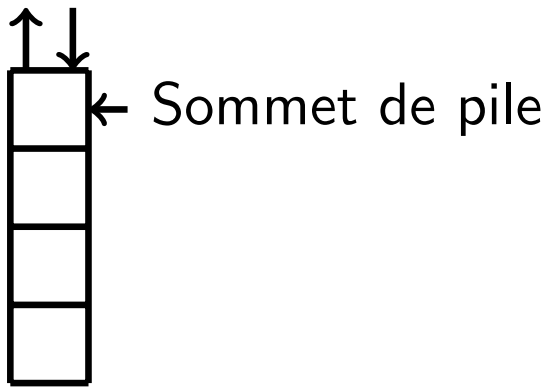

- Il existe une théorie des files d'attente en évaluation de performances/recherche opérationnelle.

# Les piles

- Les piles sont des objets informatiques omniprésents:

  par exemple:

  - les annulations d'édition dans un traitement de texte sont gérées par une pile;
  - reconnaître une expression bien parenthésée nécessite une pile;
  - les appels récursifs sont gérés par une pile de récursion;
  - évaluer une expression arithmétique nécessite une pile.



# Analyse syntaxique

- Reconnaître une expression bien parenthésée avec deux types de parenthèses `[` et `(`.
  - `[([])[]]` est bien parenthésé.
  - `[(])` est mal parenthésé.

```
static boolean BienParenthésé(String s) {
    Pile p = new Pile () ;
    for (int i=0; i<s.length(); i++) {
        char c= s.charAt(i);
        if (c=='[' || c=='(')  p.Empiler(c);
        else if (c==']') {
                if (!p.EstVide() && p.Dépiler() ≠'[')
                return false;}
        else if (c==')') {
            if (!p.EstVide() && p.Dépiler() ≠ '(')
            return false;}}
    return p.EstVide();}
```

- Les méthodes `s.length()` et `s.charAt(i)` de la classe `String` retournent la longueur et le caractère numéro $i$ de $s$.

# Aujourd'hui

Piles. Files

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

Interface/Implements en JAVA

# Programmer des piles et des files

- Solution 1: avec des listes.
- Solution 2: avec des tableaux.

# Une pile avec une liste

```
class Pile  {
  private Liste l; // sommet de la pile
  Pile() { l=null; }
  boolean EstVide() { return (l==null); }
  int     Tete() { return l.contenu;}

  void Empiler(int e) { l=new Liste(e,l);  }
  int Dépiler() { int c=l.contenu; l=l.suivant; return(c); }
}
```


- Le mot clé **private** permet de garantir qu'aucune autre classe ne puisse modifier ou accéder au champ `l`.
- Cela permet de garantir qu'il n'est pas modifié autrement que par les méthodes de la classe `Pile`.

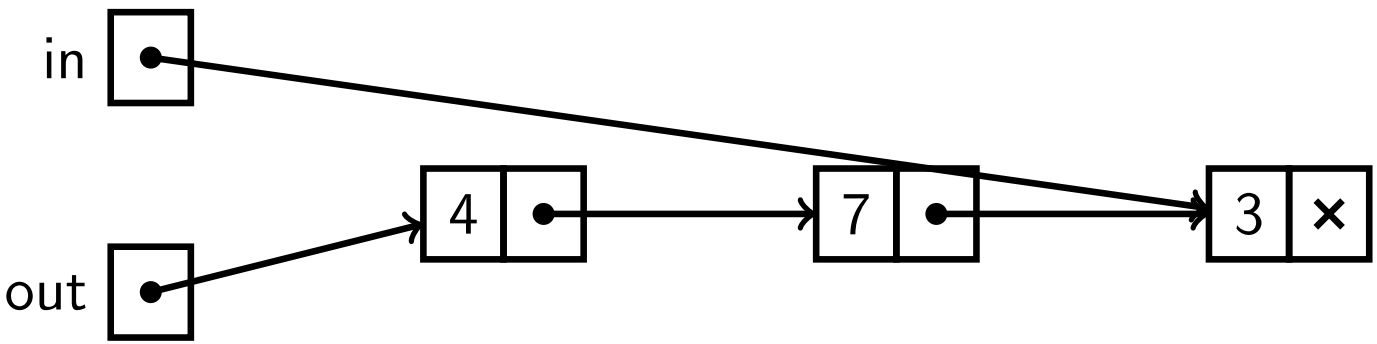
# Gérer les erreurs

```
class Pile  {
  private Liste l; // sommet de la pile
  Pile() { l=null; }
  boolean EstVide() { return (l==null); }
  int     Tete() { if (l==null) throw new Error("Pile Vide");
                   return l.contenu;}

  void Empiler(int e) { l=new Liste(e,l);  }
  int Dépiler() { if (l==null) throw new Error("Pile Vide");
                  int c=l.contenu; l=l.suivant; return(c); }
}
```

- **throw new** `Error("Pile Vide")` arrête l'exécution du programme en lançant (instruction **throw**) une exception (ici un objet de la classe `Error`).
- Affiche: *Exception in thread "main" java.lang.Error: Pile Vide* en cas d'erreur.

# Une file avec une liste

```
class File  {
  private Liste in; // entrée de la liste
  private Liste ou; // sortie de la liste
  File() { in=out=null; }
  boolean EstVide() { return (in==null && out==null); }
  int     Tete() { return out.contenu;}
  ...
}
```

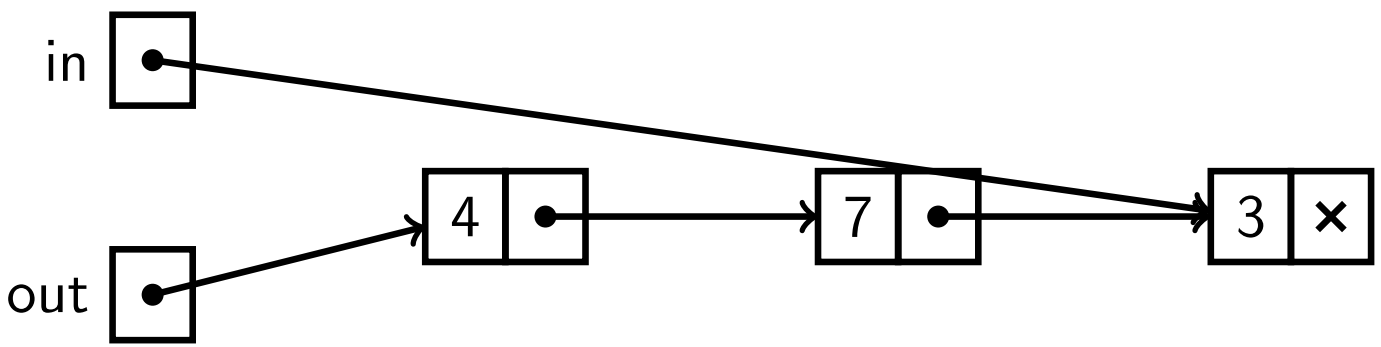



Le champ

- `out` pointe sur la première cellule de liste (pour enlever)
- `in` pointe sur la dernière cellule de liste (pour ajouter)

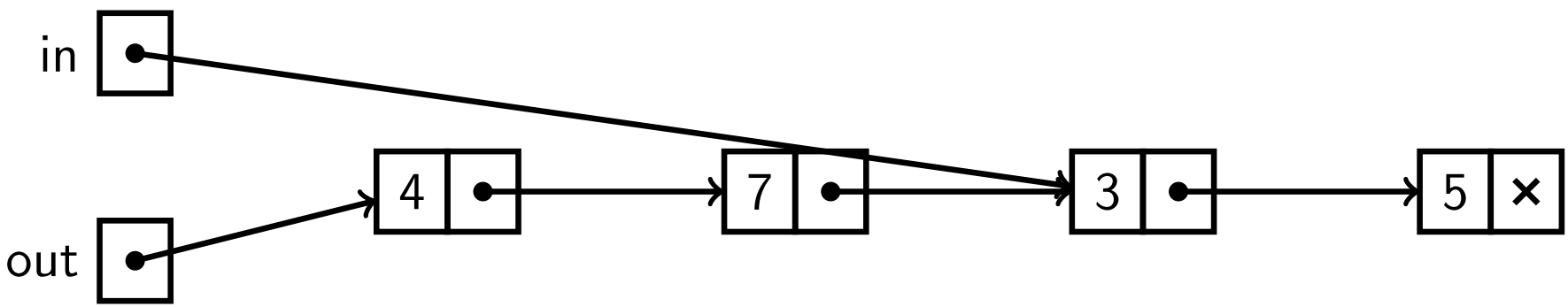
# Enfiler

```
void Enfiler(int i)  {
  if (EstVide())
    in = out = new List(i,null);
    else {
    in.suivant = new List(i,null);
    in = in.suivant;
    }}
```



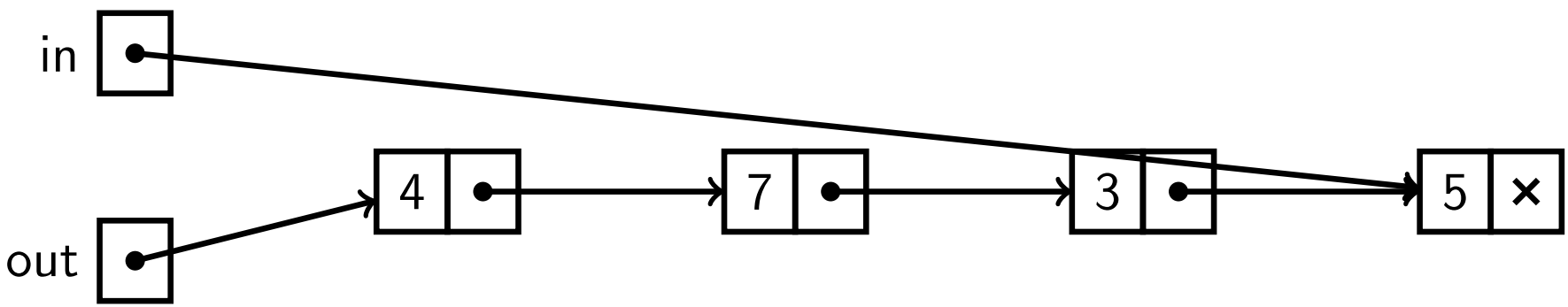
# Enfiler

```
void Enfiler(int i)  {
  if (EstVide())
    in = out = new List(i,null);
    else {
    in.suivant = new List(i,null);
    in = in.suivant;
    }}
```



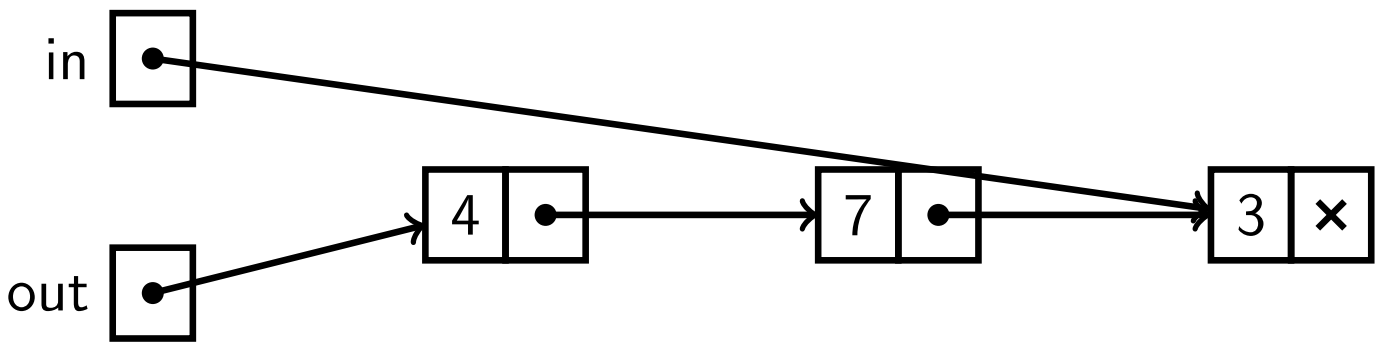
# Enfiler

```
void Enfiler(int i)  {
  if (EstVide())
    in = out = new List(i,null);
    else {
    in.suivant = new List(i,null);
    in = in.suivant;
    }}
```



# Défiler

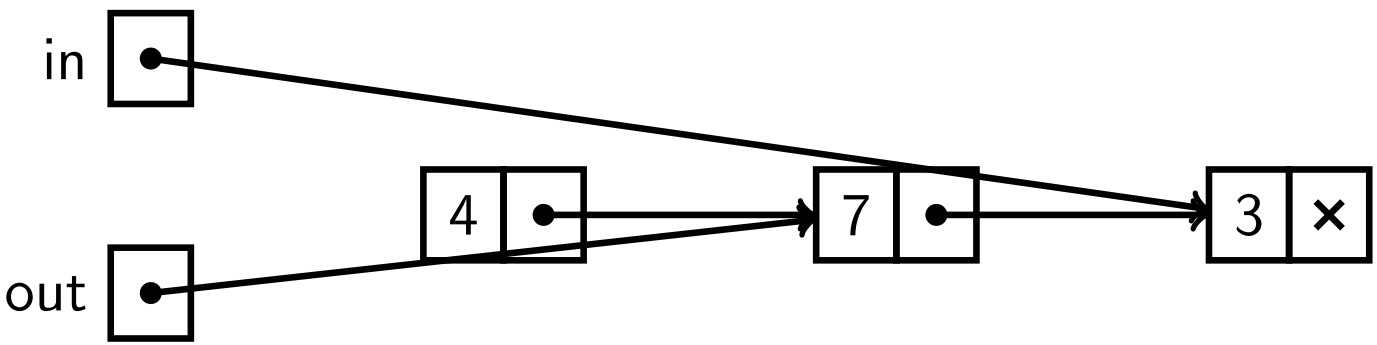
```
int Défiler()  {
    if (EstVide())
      throw new Error("Pile Vide");
    int r= out.contenu;
    out = out.suivant;
    if (out==null)
       in = null;
    return r;
    }
```




- Ne pas oublier de remettre `in` à **null** lorsque la pile est vide.

# Défiler

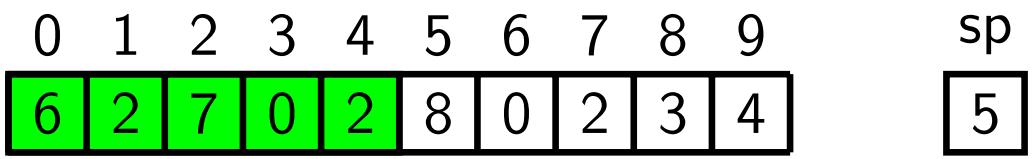
```
int Défiler()  {
    if (EstVide())
      throw new Error("Pile Vide");
    int r= out.contenu;
    out = out.suivant;
    if (out==null)
       in = null;
    return r;
    }
```




- Ne pas oublier de remettre `in` à **null** lorsque la pile est vide.

# Une pile avec un tableau

- On crée un tableau suffisamment grand.
- On utilise un entier `sp`, le *pointeur de pile*, qui compte le nombre d'éléments dans la pile.
- Les éléments d'indice inférieur à `sp` contiennent les éléments de la pile.
- Les autres, ici en blanc, sont disponibles.




- Principe:
  - Tete(): `t[sp-1]`.
  - Empiler(e): `t[sp++] =e`.
  - Dépiler(): `t[--sp]`.

# Une pile avec un tableau

```
class Pile  {
  final static int TAILLE = 100;
  private int[] t;
  private int sp;

  Pile() { t=new int[TAILLE]; sp=0; }
  boolean EstVide() { return (sp ≤0); }

  int     Tete() {
    if (EstVide()) throw new Error("Pile Vide");
      return t[sp-1];}

  void Empiler(int e) {
    if (sp ≥ t.length) throw new Error("Pile Pleine");
    t[sp++] = e;}

  int Dépiler() {
    if (EstVide()) throw new Error("Pile Vide");
    return t[--sp];} }
```

# Une pile avec un tableau dynamique

```
private void resize() {
  int[] newT = new int [2*t.length];
  for (int i=0; i<t.length; i++)
    newT[i] = t[i];
  t= newT;
 }

void Empiler(int e) {
  if (sp ≥ t.length) resize();
  t[sp++] = e;}
```

# Analyse de complexité

- Les versions précédentes de `Empiler()` et `Dépiler()` fonctionnaient en temps $O(1)$ (temps constant).
- Maintenant, `Empiler()` peut provoquer la recopie d'un tableau arbitrairement grand, et donc prendre un temps arbitrairement grand.
- Cependant, `Empiler()` reste en coût $O(1)$ *amorti* (coût global ramené au nombre d'opérations).

# Analyse de complexité

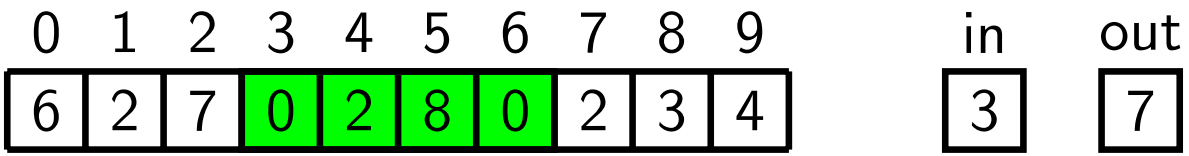
- Les versions précédentes de `Empiler()` et `Dépiler()` fonctionnaient en temps $O(1)$ (temps constant).
- Maintenant, `Empiler()` peut provoquer la recopie d'un tableau arbitrairement grand, et donc prendre un temps arbitrairement grand.
- Cependant, `Empiler()` reste en coût $O(1)$ *amorti* (coût global ramené au nombre d'opérations).
  - Supposons que l'on fasse $N$ fois `Empiler()` ou `Dépiler()`, et que la taille initiale du tableau soit $2^{p_0}$.
  - Dans le pire cas, le tableau est redimensionné $p + 1 - p_0$ fois, avec $2^p < N \leq 2^{p+1}$.
  - Coût total dans ce cas:

$$O(2^{p_0} + 2^{p_0+1} + \cdots + 2^{p+1}) = O(2^{p+2}) < O(4.N) = O(N).$$

  - Ce qui fait un coût *amorti* $O(1)$ par élément. En effet, le coût global/ le nombre d'éléments est $\leq O(N)/N = O(1)$.

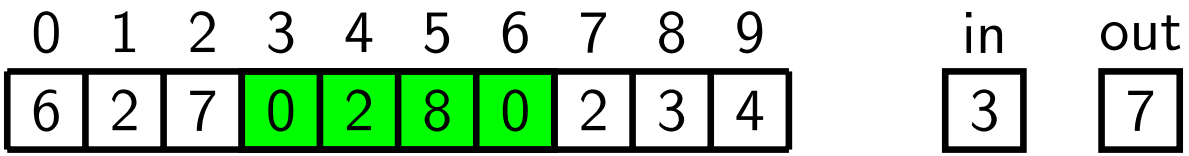
# Une file avec un tableau

- On créer toujours un tableau de taille `TAILLE` suffisamment grande.
- On utilise cette fois deux entiers `in`, et `out`.
- Les éléments entre `in` et `out-1` contiennent les éléments de la file.
- Les autres, ici en blanc, sont disponibles.

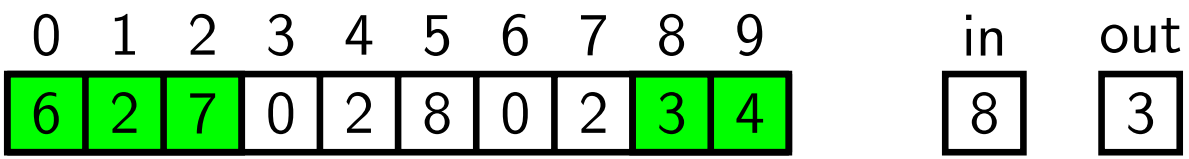


# Une file avec un tableau

- On créer toujours un tableau de taille `TAILLE` suffisamment grande.
- On utilise cette fois deux entiers `in`, et `out`.
- Les éléments entre `in` et `out-1` contiennent les éléments de la file.
- Les autres, ici en blanc, sont disponibles.




- Variante: on peut supposer le tableau circulaire: les éléments entre `in` et `out-1` modulo `TAILLE` contiennent les éléments de la file.



# Une file avec un tableau

```
class File  {
  final static int TAILLE = 100;
  private int[] t;
  private int in,out;

  File() { t=new int[TAILLE]; in=out=0; }
  boolean EstVide() { return (in == out); }

  boolean EstPleine() { return ((out+1) % TAILLE == in);}

  void Enfiler(int e) {
    if (EstPleine()) throw new Error("Pile Pleine");
    t[out]=e;
    out = (out +1) % TAILLE;}

  int Défiler() {
    if (EstVide()) throw new Error("Pile Vide");
    int c=t[in];
    in = (in+1) % TAILLE;
    return (c); }
```

# Aujourd'hui

Piles. Files

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

Interface/Implements en JAVA

# Utiliser les bibliothèques JAVA

- Piles et files sont tellement utilisées qu'elles sont proposées dans les classes JAVA standards.
- Exemple: la classe `Stack` du package `java.util` (qui correspond à l'implémentation par tableaux dynamiques précédente) pour les piles.
- Des piles de quoi?

# Classes génériques

Les codes

- d'une pile/file d'entiers
- d'une pile/file de réels
- d'une pile/file de chaînes de caractères
- d'une pile/file de piles
- d'une pile/file de files
- d'une pile/file de ...

se ressemblent beaucoup.

# Classes génériques

- Les classes de la bibliothèque JAVA sont en fait génériques.
  - `Stack` est une classe générique qui prend une classe en argument.
  - Exemple: `Stack<String>` désigne une pile de chaînes de caractères.
  - Plus généralement, `Stack<E>` est une pile d'objets de la classe `E`.

# Exemple (attention ordre opérations incorrect)

```
import java.util.*;

class Traduit {
 public static void main (String [] args) {
   Stack<String> p = new Stack<String> () ;
   for (int i = 0 ; i < args.length ; i++) {
     String cmd = args[i] ;
     if (cmd.equals("+") || cmd.equals("-")
      || cmd.equals("*") || cmd.equals("/"))
      p.push("(" + p.pop() + cmd + p.pop() + ")");
     else
      p.push(cmd) ;
   }
   System.out.println(p.pop()) ;
 }}
```

- Dans les librairies standards, `Empiler` se dit `push`, `Dépiler` se dit `pop`, `EstVide()` se dit `empty`, `Tete` se dit `peek` (voir documentation JAVA).

# Comment créer une pile d'entiers

- On ne peut pas utiliser le type `Stack<int>`, car **int** n'est pas une classe.
- Cependant, la bibliothèque JAVA standard fournit une classe appropriée par type scalaire.
- Par exemple, la classe `Integer` pour **int**:
  - La classe `Integer` est une classe avec un champ **private** de type **int**.
  - On convertit un scalaire **int** en un objet `Integer` et réciproquement par:
    - **public static** `Integer valueOf(int i)`: du scalaire vers un `Integer`.
    - **public int** `intValue()`: réciproquement.
- Il y a des classes semblables (`Char`, `Short`, etc.) pour tous les types scalaires (**char**, **short**, etc.).

# Exemple

```
import java.util.* ;
class Calc {
  public static void main (String [] arg) {
    Stack<Integer> stack = new Stack<Integer> () ;
    ...
        int i1 = stack.pop().intValue() ;
        int i2 = stack.pop().intValue() ;
        stack.push(Integer.valueOf(i2+i1)) ;
    ...
  }
}
```

- En fait, le compilateur accepte `stack.push(i2+i1)`, car il fait les conversions **int** vers `Integer` tout seul.

# Files de la bibliothèque

- La classe `LinkedList<C>` (package `java.util`) fait l'affaire.
- Il s'agit en fait d'une *double-ended queue*.
- On peut utiliser
  - les méthodes `addFirst` et `addLast` (`put`) pour ajouter un élément au debut ou à la fin.
  - les méthodes `removeFirst` (`get`) et `removeLast` pour récupérer le premier ou dernier élément.
  - Le tout en temps constant, par des listes doublement chaînées (voir le poly, section 2.3.3).
- Il existe d'autres classes de la bibliothèque standard avec des noms plus séduisants (comme `Queue`), mais `LinkedList<C>` est peut-être la plus pratique.

# Aujourd'hui

Piles. Files

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

Interface/Implements en JAVA

# Définir un type abstrait proprement

- On veut définir le type abstrait “sac”

```
class Sac  {
  ...
  Sac() { ... } //Construire un sac vide
  boolean EstVide() { ... } //Tester si un sac est vide
  void Ajouter(int e) { ... } //Ajouter un entier à un sac
  int Enlever() { ... } //Enlever un entier d'un sac
}
```

- Pour le faire proprement en JAVA, on peut définir

```
interface Sac  {
  boolean EstVide(); //Tester si un sac est vide
  void Ajouter(int e); //Ajouter un entier à un sac
  int Enlever(); //Enlever un entier d'un sac
}
```

# Ce qui permet de l'utiliser

- A ce moment là, `sac` n'est pas une classe, mais on peut l'utiliser comme un type...

```
static void count(Sac s) {
  for (int i=0; !s.EstVide(); i++) {
    s.Enlever();
}}
```

- ... sans se préoccuper de l'implémentation.

# Créer une implémentation

- On utilise le mot clé **implements**.

Implémentation 1:

```
class Pile implements Sac {

 private int[] t;
 private int sp;
 Pile() { t=new int[100]; sp=0; }

 public boolean EstVide() {
   return (sp ≤0); }
 int
Tete() {return t[sp-1];}

 public void Ajouter(int e) {
   t[sp++] = e;}

 public int Enlever() {
   return t[--sp];} }}
```

Implémentation 2:

```
class File implements Sac {

 private int[] t;
 private int in,out;
File() { t=new int[100];
   in=out=0;}

 public boolean EstVide() {
   return (in == out); }

 public void Ajouter(int e) {
   t[out]=e; out=(out +1)% 100;}

 public int Enlever() {
   int c=t[in]; in=(in+1) % 100;
   return c;} }
```

# Intérêts

- On peut travailler sans se préoccuper de l'implémentation.

  1. ce qui permet le travail en groupe, ou modulaire.
  2. et de remplacer une implémentation par une autre sans modifier le reste du programme.

- Le compilateur refuse de compiler si une des méthodes de l'**interface** est manquante,
  - et donc aide à vérifier que l'implémentation de chacune des fonctionnalités est bien présente.

- Note: Les méthodes déclarées dans l'interface doivent être qualifiées de **public**.