

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Introduction

- Pour le moment, nous nous sommes limités aux aspects fonctionnels d'OCaml et donc aux variables et aux structures de données **non mutables**.

### Champ mutable d'un enregistrement

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Introduction

- Pour le moment, nous nous sommes limités aux aspects fonctionnels d'OCaml et donc aux variables et aux structures de données **non mutables**.
- Cependant, OCaml est un langage de programmation multi-paradigme et la programmation impérative (et donc les variables mutables) peuvent être manipulées en OCaml.

### Champ mutable d'un enregistrement

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Introduction

- Pour le moment, nous nous sommes limités aux aspects fonctionnels d'OCaml et donc aux variables et aux structures de données **non mutables**.
- Cependant, OCaml est un langage de programmation multi-paradigme et la programmation impérative (et donc les variables mutables) peuvent être manipulées en OCaml.

### Champ mutable d'un enregistrement

On peut déclarer en OCaml un enregistrement ayant des champs mutables grâce au mot-clé **mutable**.

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Introduction

- Pour le moment, nous nous sommes limités aux aspects fonctionnels d'OCaml et donc aux variables et aux structures de données **non mutables**.
- Cependant, OCaml est un langage de programmation multi-paradigme et la programmation impérative (et donc les variables mutables) peuvent être manipulées en OCaml.

### Champ mutable d'un enregistrement

On peut déclarer en OCaml un enregistrement ayant des champs mutables grâce au mot-clé **mutable**. Par exemple,

```
type variable = {mutable valeur : int};;
```

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Introduction

- Pour le moment, nous nous sommes limités aux aspects fonctionnels d'OCaml et donc aux variables et aux structures de données **non mutables**.
- Cependant, OCaml est un langage de programmation multi-paradigme et la programmation impérative (et donc les variables mutables) peuvent être manipulées en OCaml.

### Champ mutable d'un enregistrement

On peut déclarer en OCaml un enregistrement ayant des champs mutables grâce au mot-clé **mutable**. Par exemple,

```
type variable = {mutable valeur : int};;
```

⚠ Pour modifier la valeur du champ on utilise **<-**. Le symbole **=** est réservé à la comparaison.

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Exemple

```
1  (* Type "variable" ayant un champ valeur mutable*)
2  type variable = {mutable valeur : int};;
3
4  (* on crée une variable ayant son champ valeur à 42 *)
5  let u = {valeur = 42};;
6
7  (* comme ce champ est mutable, on change la valeur avec <-* *)
8  u.valeur <- 20;
9  (* cette expression renvoie unit et modifie la valeur de u *)
10
11 (* Affichera 20 *)
12 print_int u.valeur
```

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)  
Par exemple `let a = {contents = 5}`; crée une variable ayant son champ mutable `contents` qui vaut 5.



# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)  
Par exemple `let a = {contents = 5}`; crée une variable ayant son champ mutable `contents` qui vaut 5.
- Afin d'alléger la syntaxe, on peut écrire directement `let a = ref 5;;`.

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)  
Par exemple `let a = {contents = 5}`; crée une variable ayant son champ mutable `contents` qui vaut 5.
- Afin d'alléger la syntaxe, on peut écrire directement `let a = ref 5;;`.  
On retiendra que l'effet reste le même : on a créé une variable `a` ayant un champ mutable entier qui contient 5.

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)  
Par exemple `let a = {contents = 5};` crée une variable ayant son champ mutable `contents` qui vaut 5.
- Afin d'alléger la syntaxe, on peut écrire directement `let a = ref 5;;`.  
On retiendra que l'effet reste le même : on a créé une variable `a` ayant un champ mutable entier qui contient 5.
- Si on veut accéder à la valeur du champ mutable, une syntaxe "adoucie" est aussi disponible avec `!`. On écrira par exemple `print_int !a` pour afficher le contenu du champ mutable de `a`.

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)  
Par exemple `let a = {contents = 5}`; crée une variable ayant son champ mutable `contents` qui vaut 5.
- Afin d'alléger la syntaxe, on peut écrire directement `let a = ref 5;;`.  
On retiendra que l'effet reste le même : on a créé une variable `a` ayant un champ mutable entier qui contient 5.
- Si on veut accéder à la valeur du champ mutable, une syntaxe "adoucie" est aussi disponible avec `!`. On écrira par exemple `print_int !a` pour afficher le contenu du champ mutable de `a`.  
On retiendra que c'est identique à `print_int a.contents`

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)  
Par exemple `let a = {contents = 5}`; crée une variable ayant son champ mutable `contents` qui vaut 5.
- Afin d'alléger la syntaxe, on peut écrire directement `let a = ref 5;;`.  
On retiendra que l'effet reste le même : on a créé une variable `a` ayant un champ mutable entier qui contient 5.
- Si on veut accéder à la valeur du champ mutable, une syntaxe "adoucie" est aussi disponible avec `!`. On écrira par exemple `print_int !a` pour afficher le contenu du champ mutable de `a`.  
On retiendra que c'est identique à `print_int a.contents`
- Pour modifier la valeur de `a`, une syntaxe plus simple est aussi disponible avec `:=`. On écrira par exemple `a := 5`

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Les références

- Le type `ref` est prédéfini dans OCaml et correspond exactement à ce que nous venons de faire (sauf que le champ mutable s'appelle `contents`)  
Par exemple `let a = {contents = 5}`; crée une variable ayant son champ mutable `contents` qui vaut 5.
- Afin d'alléger la syntaxe, on peut écrire directement `let a = ref 5;;`.  
On retiendra que l'effet reste le même : on a créé une variable `a` ayant un champ mutable entier qui contient 5.
- Si on veut accéder à la valeur du champ mutable, une syntaxe "adoucie" est aussi disponible avec `!`. On écrira par exemple `print_int !a` pour afficher le contenu du champ mutable de `a`.  
On retiendra que c'est identique à `print_int a.contents`
- Pour modifier la valeur de `a`, une syntaxe plus simple est aussi disponible avec `:=`. On écrira par exemple `a := 5`  
Par exemple, `a := !a + 1` permet d'incrémenter de 1 la valeur (du champ mutable) de `a`.

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Exemple

Créer une référence a vers 42 et une référence b vers 2023. Echanger le contenu de ces deux variables en utilisant une troisième référence temp

# C9 OCaml : aspects impératifs

## 1. Variables mutables

### Exemple

Créer une référence a vers 42 et une référence b vers 2023. Echanger le contenu de ces deux variables en utilisant une troisième référence temp

```
1 let a = ref 42;;  
2 let b = ref 2023;;  
3 let temp = ref !a;;  
4 a := !b;  
5 b := !temp;
```



### Boucle for

Il y a deux versions :

- Indice croissant :

```
1  for indice = debut to fin do
2      expression
3  done
```

### Boucle for

Il y a deux versions :

- Indice croissant :

```
1  for indice = debut to fin do
2      expression
3  done
```

- Indice décroissant :

```
1  for indice = debut downto fin do
2      expression
3  done
```

### Boucle for

Il y a deux versions :

- Indice croissant :

```
1  for indice = debut to fin do
2      expression
3  done
```

- Indice décroissant :

```
1  for indice = debut downto fin do
2      expression
3  done
```



Attention :

### Boucle for

Il y a deux versions :

- Indice croissant :

```
1  for indice = debut to fin do
2      expression
3  done
```

- Indice décroissant :

```
1  for indice = debut downto fin do
2      expression
3  done
```



Attention :

- expression est de type `unit` (ex : affectation, affichage, ...)

### Boucle for


Il y a deux versions :

- Indice croissant :

```
1  for indice = debut to fin do
2      expression
3  done
```

- Indice décroissant :

```
1  for indice = debut downto fin do
2      expression
3  done
```

 Attention :

- expression est de type `unit` (ex : affectation, affichage, ...)
- Les `deux` bornes de la boucle sont incluses

### Boucle for

Il y a deux versions :

- Indice croissant :

```
1  for indice = debut to fin do
2      expression
3  done
```

- Indice décroissant :

```
1  for indice = debut downto fin do
2      expression
3  done
```



Attention :

- expression est de type `unit` (ex : affectation, affichage, ...)
- Les `deux` bornes de la boucle sont incluses
- Pas de `break` ou de `continue` et indice ne peut pas être modifié dans la boucle

### Exemple de boucle `for`

- Ecrire une fonction `harmonique_asc` qui prend en argument un entier `n` et calcule la somme des inverses des entiers non nuls jusqu'à `n` de façon ascendante.

### Exemple de boucle `for`

- Ecrire une fonction `harmonique_asc` qui prend en argument un entier `n` et calcule la somme des inverses des entiers non nuls jusqu'à `n` de façon ascendante.
- Même question de façon descendante avec la fonction `harmonique_desc`



### Exemple de boucle `for`

- Ecrire une fonction `harmonique_asc` qui prend en argument un entier `n` et calcule la somme des inverses des entiers non nuls jusqu'à `n` de façon ascendante.
- Même question de façon descendante avec la fonction `harmonique_desc`
- Que pensez de l'écart observé entre les deux valeurs pour `n` assez grand ?

### Boucle while

```
1  while condition do  
2      expression  
3  done
```

### Boucle while

```
1  while condition do  
2      expression  
3  done
```

### Boucle while

```
1 while condition do  
2   expression  
3 done
```



Attention :

### Boucle while

```
1 while condition do  
2     expression  
3 done
```



Attention :

- expression est de type `unit` (ex : affectation, affichage, ...)

### Boucle while

```
1 while condition do
2     expression
3 done
```



Attention :

- expression est de type `unit` (ex : affectation, affichage, ...)
- condition est de type `bool`

### Boucle while

```
1 while condition do
2     expression
3 done
```



Attention :

- expression est de type `unit` (ex : affectation, affichage, ...)
- condition est de type `bool`
- Pas de `break`

### Exemple

- Ecrire une fonction permettant de calculer la somme des  $n$  premiers entiers à l'aide d'une boucle `while`.



### Exemple

- Ecrire une fonction permettant de calculer la somme des  $n$  premiers entiers à l'aide d'une boucle `while`.
- Ecrire une fonction `log2` qui prend en argument un entier  $n$  et renvoie l'entier  $k$  tel que  $2^{k-1} < n \leq 2^k$

### Syntaxe

Les tableaux en OCaml, sont identiques à ceux vus en C. c'est-à-dire qu'il s'agit d'une structure mutable dont les éléments sont rangés de façon contigue en mémoire (et donc avec un accès en  $O(1)$ ).

- Les tableaux sont notés entre `[]` et `[]`

### Syntaxe

Les tableaux en OCaml, sont identiques à ceux vus en C. c'est-à-dire qu'il s'agit d'une structure mutable dont les éléments sont rangés de façon contigue en mémoire (et donc avec un accès en  $O(1)$ ).

- Les tableaux sont notés entre `[|` et `|]`

```
let ex = [|2; 7; 9; 14 |]
```

### Syntaxe

Les tableaux en OCaml, sont identiques à ceux vus en C. c'est-à-dire qu'il s'agit d'une structure mutable dont les éléments sont rangés de façon contigue en mémoire (et donc avec un accès en  $O(1)$ ).

- Les tableaux sont notés entre `[|` et `|]`  
`let` `ex` = `[|2; 7; 9; 14 |]`
- L'accès à un élément se fait avec la notation `tab.(i)`

### Syntaxe

Les tableaux en OCaml, sont identiques à ceux vus en C. c'est-à-dire qu'il s'agit d'une structure mutable dont les éléments sont rangés de façon contigue en mémoire (et donc avec un accès en  $O(1)$ ).

- Les tableaux sont notés entre `[|` et `|]`

```
let ex = [|2; 7; 9; 14 |]
```

- L'accès à un élément se fait avec la notation `tab.(i)`

```
print_int ex.(1) affiche 7
```

### Syntaxe

Les tableaux en OCaml, sont identiques à ceux vus en C. c'est-à-dire qu'il s'agit d'une structure mutable dont les éléments sont rangés de façon contigue en mémoire (et donc avec un accès en  $O(1)$ ).

- Les tableaux sont notés entre `[|` et `|]`

```
let ex = [|2; 7; 9; 14 |]
```

- L'accès à un élément se fait avec la notation `tab.(i)`

```
print_int ex.(1) affiche 7
```

- La structure est mutable, attention l'affectation se fait avec `<-` (sans `let`)

### Syntaxe

Les tableaux en OCaml, sont identiques à ceux vus en C. c'est-à-dire qu'il s'agit d'une structure mutable dont les éléments sont rangés de façon contigue en mémoire (et donc avec un accès en  $O(1)$ ).

- Les tableaux sont notés entre `[|` et `|]`

```
let ex = [|2; 7; 9; 14 |]
```

- L'accès à un élément se fait avec la notation `tab.(i)`

```
print_int ex.(1) affiche 7
```

- La structure est mutable, attention l'affectation se fait avec `<-` (sans `let`)  
`ex.(1)<-13` le contenu du tableau est maintenant `[|2; 13; 9; 14 |]`

### Fonctions de Array

- `Array.make` qui prend en argument un entier et une valeur et crée le tableau contenant n fois cette valeur



### Fonctions de Array

- `Array.make` qui prend en argument un entier et une valeur et crée le tableau contenant  $n$  fois cette valeur
- `Array.length` donne le nombre d'élément du tableau (en  $O(1)$ )

### Fonctions de Array

- `Array.make` qui prend en argument un entier et une valeur et crée le tableau contenant  $n$  fois cette valeur
- `Array.length` donne le nombre d'élément du tableau (en  $O(1)$ )
- `Array.of_list` `Array.to_list` permet de convertir depuis ou vers une liste.

#### Fonctions de Array

- `Array.make` qui prend en argument un entier et une valeur et crée le tableau contenant  $n$  fois cette valeur
- `Array.length` donne le nombre d'élément du tableau (en  $O(1)$ )
- `Array.of_list` `Array.to_list` permet de convertir depuis ou vers une liste.
- `Array.map`, `Array.iter`, `Array.fold_left` (ou `Array.fold_right`) sont les équivalents sur les tableaux des fonctions de même nom sur les listes.

### Fonctions de Array

- `Array.make` qui prend en argument un entier et une valeur et crée le tableau contenant  $n$  fois cette valeur
- `Array.length` donne le nombre d'élément du tableau (en  $O(1)$ )
- `Array.of_list` `Array.to_list` permet de convertir depuis ou vers une liste.
- `Array.map`, `Array.iter`, `Array.fold_left` (ou `Array.fold_right`) sont les équivalents sur les tableaux des fonctions de même nom sur les listes.

### Aliasing

⚠ Quand `tab` est un tableau, alors `let new_tab = tab` fait pointer `new_tab` vers la *même* zone mémoire que `tab` et donc toute modification de l'un se répercute sur l'autre ! Faire particulièrement attention lors de la création de matrices (donc de tableaux de tableaux).

### Exemples

- Crée le tableau contenant les doubles des 100 premiers entiers.

### Exemples

- Crée le tableau contenant les doubles des 100 premiers entiers.
- Ecrire la fonction affiche permettant d'afficher les éléments d'un tableau d'entier. On utilisera une boucle `for` et `Array.length`.

### Exemples

- Crée le tableau contenant les doubles des 100 premiers entiers.
- Ecrire la fonction affiche permettant d'afficher les éléments d'un tableau d'entier. On utilisera une boucle `for` et `Array.length`.
- Ecrire une fonction permettant de trouver le minimum des éléments d'un tableau non vide.

### Fonction d'affichage

- Les fonctions permettant d'afficher les valeurs de type de base `print_int`, `print_float`, ...



### Fonction d'affichage

- Les fonctions permettant d'afficher les valeurs de type de base `print_int`, `print_float`, ...
- La fonction `printf` du module `Printf` permettant d'afficher, à la façon du C, des données. Les spécificateurs de format sont :
  - `%d` : un entier
  - `%s` : une chaîne de caractères
  - `%f` : un flottant
  - `%b` : un booléen
  - `%c` : un caractère

Par exemple :

```
Printf.printf "Date de naissance : le %d %s %d" jour mois annee
```

### Fonction de saisie

Plusieurs fonctions permettent de récupérer des informations entrées au clavier :

- La fonction `read_line` de type `unit->string` attend la saisie d'une chaîne de caractère au clavier (suivie d'un retour chariot) puis renvoie cette chaîne (sans le retour chariot final)

### Fonction de saisie

Plusieurs fonctions permettent de récupérer des informations entrées au clavier :

- La fonction `read_line` de type `unit->string` attend la saisie d'une chaîne de caractère au clavier (suivie d'un retour chariot) puis renvoie cette chaîne (sans le retour chariot final)
- Des fonctions similaires existent qui tentent directement la conversion dans le type attendu comme `read_int` pour lire un entier. Une exception est levée si cette conversion échoue.

### Fonction de saisie

Plusieurs fonctions permettent de récupérer des informations entrées au clavier :

- La fonction `read_line` de type `unit->string` attend la saisie d'une chaîne de caractère au clavier (suivie d'un retour chariot) puis renvoie cette chaîne (sans le retour chariot final)
- Des fonctions similaires existent qui tentent directement la conversion dans le type attendu comme `read_int` pour lire un entier. Une exception est levée si cette conversion échoue.

### Exemple

```
1  (* Une variation du traditionnel Hello World : avec le nom*)
2  let hello () =
3      print_string "Entrez votre nom : ";
4      let nom = read_line () in
5      Printf.printf "Bonjour %s \n" nom;;
```

### Lecture et écriture dans un fichier

Les accès vers les fichiers sont traités comme des canaux (`in_channel` pour un canal de lecture et `out_channel` pour un canal de sortie). Pour lire/écrire dans un fichier on retrouve les trois étapes habituels

- L'ouverture du fichier de type `string` -> `in_channel` pour une ouverture en lecture et `string` -> `out_channel` pour une ouverture en écriture.

### Lecture et écriture dans un fichier

Les accès vers les fichiers sont traités comme des canaux (`in_channel` pour un canal de lecture et `out_channel` pour un canal de sortie). Pour lire/écrire dans un fichier on retrouve les trois étapes habituels

- L'ouverture du fichier de type `string` -> `in_channel` pour une ouverture en lecture et `string` -> `out_channel` pour une ouverture en écriture.
- La fonction `input_line` de type `in_channel` -> `string` permet de lire une ligne du fichier (sans le retour chariot final) et `output_string` de type `out_channel` -> `string` -> `unit` permet d'écrire dans un canal de sortie.  
⚠ Dans le cas d'une lecture, une exception est levée lorsque la fin de fichier est atteinte.

### Lecture et écriture dans un fichier

Les accès vers les fichiers sont traités comme des canaux (`in_channel` pour un canal de lecture et `out_channel` pour un canal de sortie). Pour lire/écrire dans un fichier on retrouve les trois étapes habituels

- L'ouverture du fichier de type `string` -> `in_channel` pour une ouverture en lecture et `string` -> `out_channel` pour une ouverture en écriture.
- La fonction `input_line` de type `in_channel` -> `string` permet de lire une ligne du fichier (sans le retour chariot final) et `output_string` de type `out_channel` -> `string` -> `unit` permet d'écrire dans un canal de sortie.  
⚠ Dans le cas d'une lecture, une exception est levée lorsque la fin de fichier est atteinte.
- Les fonctions `close_in` et `close_out` permettent de fermer un fichier (respectivement d'entrée ou de sortie).

# C9 OCaml : aspects impératifs

## 4. Entrées-Sorties

### Compte le nombre de lignes dans un fichier

Le programme suivant demande le nom d'un fichier puis affiche son nombre de lignes dans le terminal.

```
1  print_string "Nom du fichier : ";
2  let filename = read_line () in
3  let reader = open_in filename in
4  let nl = ref 0 in
5  let end_of_file = ref false in
6  while not !end_of_file do
7      try
8          (let _ = input_line reader in
9           nl := !nl + 1;)
10     with
11         End_of_file -> end_of_file := true;
12 done;
13 Printf.printf "Le fichier %s contient %d lignes \n" filename !nl;;
```



## C9 OCaml : aspects impératifs

### 5. Argument en ligne de commande

#### Module Sys

A la façon de ce qui a été vu en C, on peut passer des arguments en ligne de commande à un programme. Les arguments présents sur la ligne de commande sont rangés dans le tableau `Sys.argv`.

⚠ Comme en C, le premier élément du tableau (donc celui d'indice 0) contient le nom de l'exécutable.

# C9 OCaml : aspects impératifs

## 5. Argument en ligne de commande

### Module Sys

A la façon de ce qui a été vu en C, on peut passer des arguments en ligne de commande à un programme. Les arguments présents sur la ligne de commande sont rangés dans le tableau `Sys.argv`.

⚠ Comme en C, le premier élément du tableau (donc celui d'indice 0) contient le nom de l'exécutable. Par exemple, si on compile le programme suivante sous le nom `arg.exe`

```
1 Printf.printf "Nombre d'arguments : %d\n" (Array.length (Sys.argv));  
2 Printf.printf "Premier argument : %s\n" Sys.argv.(0);
```

# C9 OCaml : aspects impératifs

## 5. Argument en ligne de commande

### Module Sys

A la façon de ce qui a été vu en C, on peut passer des arguments en ligne de commande à un programme. Les arguments présents sur la ligne de commande sont rangés dans le tableau `Sys.argv`.

⚠ Comme en C, le premier élément du tableau (donc celui d'indice 0) contient le nom de l'exécutable. Par exemple, si on compile le programme suivante sous le nom `arg.exe`

```
1 Printf.printf "Nombre d'arguments : %d\n" (Array.length (Sys.argv));  
2 Printf.printf "Premier argument : %s \n" Sys.argv.(0);
```

Alors, `./arg.exe blabla` affichera :

Nombre d'arguments fournis : 2

Le premier argument est : `./arg.exe`

### Bloc try-with

Les **exceptions** sont un mécanisme de récupération et de gestion des erreurs pouvant se produire lors de l'exécution d'un programme. Un exemple a déjà été vu dans le programme permettant de compter les lignes d'un fichier. Le mécanisme de gestion est toujours de la forme :

### Bloc try-with

Les **exceptions** sont un mécanisme de récupération et de gestion des erreurs pouvant se produire lors de l'exécution d'un programme. Un exemple a déjà été vu dans le programme permettant de compter les lignes d'un fichier. Le mécanisme de gestion est toujours de la forme :

```
try <expr>  
with  
| <motif1> -> <expr1>  
..  
| <motifn> -> <exprn>
```

### Bloc try-with

Les **exceptions** sont un mécanisme de récupération et de gestion des erreurs pouvant se produire lors de l'exécution d'un programme. Un exemple a déjà été vu dans le programme permettant de compter les lignes d'un fichier. Le mécanisme de gestion est toujours de la forme :

```
try <expr>  
with  
| <motif1> -> <expr1>  
..  
| <motifn> -> <exprn>
```

Ce mécanisme permet notamment de quitter les boucles `for` avant leur terme (par exemple si on cherche un élément dans un tableau)

### Exemple

On peut éventuellement choisir de poursuivre l'exécution d'un programme lors d'une division par zéro :

### Exemple

On peut éventuellement choisir de poursuivre l'exécution d'un programme lors d'une division par zéro :

```
1  let division a b =  
2    try  
3      a/b  
4    with  
5    | Division_by_zero -> Int.max_int
```



### Exemple

On peut éventuellement choisir de poursuivre l'exécution d'un programme lors d'une division par zéro :

```
1  let division a b =  
2    try  
3      a/b  
4    with  
5    | Division_by_zero -> Int.max_int
```

Ici on choisit de renvoyer le plus grand entier représentable (ce qui n'est pas forcément un choix judicieux!)

### Exemple

Fonction qui renvoie True si elt est présent dans le tableau tab. On utilise les exceptions pour terminer la boucle dès que l'élément est trouvé.

```
1 let est_dans elt tab =  
2   try  
3     for i=0 to (Array.length tab -1) do  
4       if tab.(i)=elt then raise Exit;  
5     done;  
6     false  
7   with  
8   | Exit -> true;;
```