# Exceptions en OCaml



Les exceptions sont un mécanisme présent dans de nombreux langages de programmation afin de permettre à la fois d'indiquer qu'une erreur s'est produite en interrompant l'exécution d'une fonction et aussi de pouvoir, pour l'appelant **rattraper** cette erreur afin de poursuivre l'exécution comme il se doit.

En `OCaml`, ce mécanisme va permettre, de plus, de pouvoir interrompre le flot de contrôle, notamment de retrouver des mécanismes comme `break`, `continue` et surtout `return`.

## I Syntaxe des exceptions

Les exceptions forment un type `exn` qui est extensible : il est possible de rajouter un nouveau constructeur pour ce type en écrivant :

OCaml
```
(* Pour une exception sans paramètre *)
exception NomDeLException

(* Pour une exception avec paramètre de type t*)
exception NomDeLException of t
```

On peut alors **lancer** cette exception à l'aide de `raise` :

OCaml
```
(* Pour une exception sans paramètre *)
raise NomDeLException

(* Pour une exception avec paramètre de valeur v *)
(* Attention aux parenthèses *)
raise (NomDeLException v)
```

Il est possible d'évaluer une expression en permettant d'évaluer une autre expression en cas d'exception :

OCaml
```
try
    expression
with ExceptionsARattraper -> expression'
```

En fait, ce `with` est un filtrage, un `match`, sur les valeurs du type `exn` mais sans vérification d'exhaustivité. On peut donc effectuer plusieurs cas :

OCaml
```
try
    expression
with  motif1 -> expr1
    | motif2 -> expr2
    | ...
```

En cas de `try... with` imbriqués, c'est le `try` le plus proche de l'exception qui la rattrape. SI une exception n'est pas rattrapée, elle va produire une erreur qui stoppera l'exécution d'un programme. On peut très bien lancer une exception depuis un `with .. ->`.

# II Exceptions pour la gestion d'erreurs

Il s'agit ici de l'utilisation la plus *logique* des exceptions.
On a déjà pu rencontrer des exceptions en OCaml :

```OCaml
# let t = [|1;2|];;
val t : int array = [|1; 2|]
# t.(2);;
Exception: Invalid_argument "index out of bounds".
# List.hd [];;
Exception: Failure "hd".
```

Dans les deux cas, ces exceptions sont plutôt le signe d'une erreur de programmation qu'il faut corriger qu'un comportement limite qu'il faudrait prendre en compte. Dans le cas d'une fonction comme `List.hd`, on a également vu qu'il était préférable d'utiliser un type option pour pouvoir gérer l'erreur derrière avec :

```OCaml
let hd_opt l =
    match l with
    | [] -> None
    | t :: _ -> Some t
```

Pour autant, il y a des erreurs importantes à gérer en OCaml, ce sont celles qui sont inévitables quand il n'est pas possible de prévoir si un opération va réussir. Citons deux cas :

- la lecture d'un fichier va produire une exception `End_of_file` une fois atteinte la fin de celui-ci ;
- des structures de données qui ne fournissent comme seul moyen efficace de savoir si elles sont vides que d'essayer d'en extraire un élément et d'échouer.

Ainsi, dans le premier cas, on pourra écrire une fonction comme :

```OCaml
let rec input_all_lines ic =
    try
        let l = input_line ic in
        l :: input_all_lines ic
    with End_of_file -> []
```

Cette fonction prend un `input_channel` et renvoie toutes les lignes qu'il contient sous la forme d'une liste. Le rattrapage d'exception est crucial, c'est lui qui permet d'avoir un cas de base pour la récurrence.

**Remarque** **Attention**, on peut être tenté d'écire :

```OCaml
let rec input_all_lines ic =
    try
        input_line ic :: input_all_lines ic
    with End_of_file -> []
```

Or, il se trouve que OCaml évalue tous les arguments du constructeur `::` et il le fait de la droite vers la gauche. Le code précédent va donc effectuer des appels récursifs sans jamais lire une seule ligne.

# III Programmer avec des exceptions

## III.1 Retour prématuré

Si on considère le programme C suivant :

```C
int recherche_element(int t[],
    unsigned int nb_elts, int x)
{
    for(int i = 0; i < nb_elts; i++)
        if(t[i] == x)
            return i;
    return -1;
}
```

On peut l'écrire sous la forme suivante en OCaml :

```OCaml
let recherche_element t x =
    let indice = ref None in
    for i = 0 to Array.length t - 1 do
        if !indice = None && t.(i) = x
        then indice := Some i
    done;
    !indice
```

Cela correspond à ce qu'on aurait fait en C pour se passer du `return` dans la boucle mais cela présente plusieurs problèmes :

- on inhibe les itérations d'une boucle explicitement ;
- avec plusieurs boucles imbriquées, c'est peu lisible.

Notons que cette interruption du flot de controle est très naturelle dans un langage utilisant beaucoup la récursivité. Le code suivant récursif suivant a la même structure que le code C :

```OCaml
let recherche_element t x =
    let rec rech_aux i =
        if i = Array.length t
        then None
        else if t.(i) = x then Some i
        else rech_aux (i+1)
    in rech_aux 0
```

En effet, on remarque deux feuilles possibles pour l'arbre d'appels récursifs : soit `None` en fin de tableau, soit `Some i` quand on a trouvé l'élément.

Traduire des programmes impératifs sous cette forme est également peu satisfaisant.

À l'aide des exceptions, on peut retrouver exactement la structure du code C initiale ainsi :

```OCaml
exception Trouve of int

let recherche_element t x =
    try
        for i = 0 to Array.length t - 1 do
            if t.(i) = x
            then raise (Trouve i)
        done;
        None
    with Trouve i -> Some i
```

Ou même, en utilisant l'exception `Not_found` prédéfinie :

```OCaml
exception Trouve of int

let recherche_element t x =
    try
        for i = 0 to Array.length t - 1 do
            if t.(i) = x
            then raise (Trouve i)
        done;
        raise Not_found
    with Trouve i -> i
```

L'expression `raise (Trouve i)` est ainsi exactement le pendant du `return i;`.

Au lieu d'écrire

```OCaml
...
return v
...
```

on écrit

```OCaml
try
    ....
    raise (Return v)
    ...
    failwith "Unreachable"
with Return v -> v
```

### III.1.i break et continue

On a déjà vu que `break` permettait de sortir d'une boucle et `continue` de passer à l'itération suivante.

Retrouver ces mécanismes avec `OCaml` est plus anecdotiques mais on peut le faire. On définit tout d'abord deux exceptions :

```OCaml
exception Break
exception Continue
```

Pour un `break` dans une boucle

Au lieu d'écrire

```OCaml
for ... do
    ...
    break
    ...
done
```

on écrit

```OCaml
try
    for ... do
        ....
        raise Break (* sortie *)
        ...
    done
with Break -> ()
```

Et pour un `continue` :

Au lieu d'écrire

```OCaml
for ... do
    ...
    continue
    ...
done
```

on écrit

```OCaml
for ... do
    try
        ...
        raise Continue
        ...
    with Continue -> ()
done
```