

Der Programmierer kann selbst Exceptions auslösen.

Das geht mit dem Schlüsselwort `raise ...`

```
# 1 + (2/0);;
```

```
Exception: Division_by_zero.
```

```
# 1 + raise Division_by_zero;;
```

```
Exception: Division_by_zero.
```

Eine Exception ist ein Fehlerwert, der jeden Ausdruck ersetzen kann.

Bei Behandlung wird sie durch einen anderen Ausdruck (vom richtigen Typ) ersetzt — oder durch eine andere Exception `;-)`

Exception Handling kann nach jedem beliebigen Teilausdruck, auch geschachtelt, stattfinden:

```
# let f (x,y) = x / (y-1);;
# let g (x,y) = try let n = try f (x,y)
                    with Division_by_zero ->
                        raise (Failure "Division by zero")
                    in string_of_int (n*n)
  with Failure str -> "Error: "^str;;

# g (6,1);;
- : string = "Error: Division by zero"
# g (6,3);;
- : string = "9"
```

## 7.2 Imperative Features im Ocaml

Gelegentlich möchte man Werte **destruktiv** verändern ;-)

Dazu benötigen wir neue Konzepte ...

### Modifizierbare Record-Komponenten:

- Records fassen benamte Werte zusammen ;-)
- Einzelne Komponenten können als **modifizierbar** deklariert werden ...

```
# type cell = {owner: string; mutable value: int};;  
type cell = { owner : string; mutable value : int; }
```

```
...  
# let x = {owner="me"; value=1};;  
val x : cell = {owner = "me"; value = 1}  
# x.value;;  
- : int = 1  
# x.value <- 2;;  
- : unit = ()  
# x.value;;  
- : int = 2
```

```
...
# let x = {owner="me"; value=1};;
val x : cell = {owner = "me"; value = 1}
# x.value;;
- : int = 1
# x.value <- 2;;
- : unit = ()
# x.value;;
- : int = 2
```

- Modifizierbare Komponenten werden mit `mutable` gekennzeichnet.
- Die Initialisierung erfolgt wie bei einer normalen Komponente.
- Der Ausdruck `x.value <- 2` hat den Wert `()`, aber modifiziert die Komponente `value` als **Seiteneffekt !!!**

## Spezialfall: Referenzen

Eine Referenz `tau ref` auf einen Typ `tau` ist ein Record mit der einzigen Komponente `mutable contents: tau`:

```
# let ref_hallo = {contents = "Hallo!"};;
val ref_hallo : string ref = {contents = "Hallo!"}
# let ref_1 = ref 1;;
val ref_1 : int ref = {contents = 1}
```

## Spezialfall: Referenzen

Eine Referenz `tau ref` auf einen Typ `tau` ist ein Record mit der einzigen Komponente `mutable contents: tau`:

```
# let ref_hallo = {contents = "Hallo!"};;
val ref_hallo : string ref = {contents = "Hallo!"}
# let ref_1 = ref 1;;
val ref_1 : int ref = {contents = 1}
```

Deshalb kann man auf den Wert einer Referenz mit Selektion zugreifen:

```
# ref_hallo.contents;;
- : string = "Hallo!"
# ref_1.contents;;
- : int = 1
```

Eine andere Möglichkeit ist der **Dereferenzierungs-Operator** `!:`

```
# !ref_hallo;;  
- : string = "Hallo!"
```



Eine andere Möglichkeit ist der **Dereferenzierungs-Operator** `!:`

```
# !ref_hallo;;  
- : string = "Hallo!"
```

Der Wert, auf den eine Referenz zeigt, kann mit `<-` oder mit `:=` verändert werden:

```
# ref_1.contents <- 2;;  
- : unit = ()  
# ref_1 := 3;;  
- : unit = ()
```

## Gleichheit von Referenzen

Das Setzen von `ref_1` mittels `:=` erfolgt als **Seiteneffekt** und hat keinen Wert, d.h. ergibt `()`.

```
# (:=);;  
- : 'a ref -> 'a -> unit = <fun>
```

Zwei Referenzen sind **gleich**, wenn sie auf **den gleichen** Wert zeigen:

```
# let x = ref 1  
  let y = ref 1;;  
val x : int ref = {contents = 1}  
val y : int ref = {contents = 1}  
# x = y;;  
- : bool = true
```

# Sequenzen

Bei Updates kommt es nur auf den Seiteneffekt an :-)

Bei Seiteneffekten kommt es auf die Reihenfolge an :-)

Mehrere solche Aktionen kann man mit dem **Sequenz-Operator** ; hintereinander ausführen:

```
# ref_1 := 1; ref_1 := !ref_1 + 1; ref_1;;  
- : int ref = {contents = 2}
```

In **Ocaml** kann man sogar **Schleifen** programmieren ...

```
# let x = ref 0;;  
val x : int ref = {contents = 1}  
# while !x < 10 do x := !x+1 done;;  
- : unit = ()  
# x;;  
- : int ref = contents = 10
```

```
# let x = ref 0;;
val x : int ref = {contents = 1}
# while !x < 10 do x := !x+1 done;;
- : unit = ()
# x;;
- : int ref = contents = 10
```

Ein wichtiges Listenfunktional ist `List.iter`:

```
# let rec iter f = function
  []      -> ()
  | x::xs -> f x; iter f xs;;

val iter : ('a -> unit) -> 'a list -> unit = <fun>
```

# Arrays und Strings

Ein Array ist ein Record fester Länge, auf dessen modifizierbare Elemente mithilfe ihres Index in **konstanter Zeit** zugegriffen wird:

```
# let arr = [|1;3;5;7|];;  
val arr : int array = [|1; 3; 5; 7|]  
# arr.(2);;  
- : int = 5
```

Zugriff außerhalb der Array-Grenzen löst eine Exception aus:

```
# arr.(4);;  
Invalid_argument "index out of bounds"
```

Ein Array kann aus einer Liste oder als **Wertetabelle** einer Funktion erzeugt werden ...

```
# Array.of_list [1;2;3];;  
- : int array = [|1; 2; 3|]  
# Array.init 6 (fun x -> x*x);;  
- : int array = [|0; 1; 4; 9; 16; 25|]
```

... und wieder zurück in eine Liste transformiert werden:

```
Array.fold_right (fun x xs -> x::xs)  
                [|0; 1; 4; 9; 16; 25|] [];;  
- : int list = [0; 1; 4; 9; 16; 25]
```

Modifizierung der Array-Einträge funktioniert analog der Modifizierung von Record-Komponenten:

```
# arr.(1) <- 4;;  
- : unit = ()  
# arr;;  
- : int array = [|1; 4; 5; 7|]  
# arr.(5) <- 0;;  
Exception: Invalid_argument "index out of bounds".
```



Ähnlich kann man auch Strings manipulieren :-)

```
# let str = "Hallo";  
val str : string = "Hallo"  
# str.[2];;  
- : char = 'l'  
# str.[2] <- 'r';;  
- : unit = ()  
# str;;  
- : string = "Harlo"
```

Für Arrays und Strings gibt es übrigens auch die Funktionen `length` und `concat` (und weitere :-).

## 7.3 Textuelle Ein- und Ausgabe

- Selbstverständlich kann man in **Ocaml** auf den Standard-Output schreiben:

```
# print_string "Hello World!\n";;  
Hello World!  
- : unit = ()
```

- Analog gibt es eine Funktion: `read_line : unit -> string`

...

```
# read_line ();;  
Hello World!  
- : "Hello World!"
```

Um aus einer **Datei zu lesen**, muss man diese zum Lesen **öffnen ...**

```
# let infile = open_in "test";;  
val infile : in_channel = <abstr>  
# input_line infile;;  
- : "Die einzige Zeile der Datei ...";;  
# input_line infile;;  
Exception: End_of_file
```

Gibt es keine weitere Zeile, wird die Exception **End\_of\_file** geworfen **:-)**

Benötigt man einen Kanal nicht mehr, sollte man ihn geregelt **schließen ...**

```
# close_in infile;;  
- : unit = ()
```

## Weitere nützliche Funktionen:

```
stdin           : in_channel
input_char      : in_channel -> char
in_channel_length : in_channel -> int
input : in_channel -> string -> int -> int -> int
```

- `in_channel_length` liefert die Gesamtlänge der Datei.
- `input chan buf p n` liest aus einem Kanal `chan n` Zeichen und schreibt sie ab Position `p` in den String `buf`  
:-)

Die [Ausgabe in Dateien](#) erfolgt ganz analog ...

```
# let outfile = open_out "test";;  
val outfile : out_channel = <abstr>  
# output_string outfile "Hello ";;  
- : unit = ()  
# output_string outfile "World!\n";;  
- : unit = ()  
...
```

Die einzeln geschriebenen Wörter sind mit Sicherheit in der Datei erst zu finden, wenn der Kanal geregelt [geschlossen wurde](#) ...

```
# close_out outfile;;  
- : unit = ()
```