This topic covers the background and motivations for imperative programming, as well as the imperative constructs in F# - reference cells and mutable records.
Background – effects in machine code

- The first programs were written in *machine code*.
  - Machine code is just a sequence of bytes placed into memory that the circuits in the CPU directly read and execute.
  - E.g. The bytes “53 0 123” might mean: “increment memory location 123”

- Machine code is *imperative*: each instruction does something.
  - It may modify the internal state of the machine.
  - Or it may cause the machine to do some external action.
  - The actions performed by an instruction are called its *effects*.

- *Assembly language* makes machine code easier for humans by using names for instructions instead of numbers. Eg.
  - `inc 123`
  - Also, names can be used for locations instead of addresses.
    - `inc count` (with count earlier defined as 123)
Imperative languages

- Imperative languages grew naturally from assembly language.
  - Fortran (“formula translation”) first appeared in 1957.
  - It made writing mathematical expressions easier than assembly

- Imperative languages have remained dominant since.
  - OO grew out imperative (and functional) programming, but is still generally considered an imperative sub-paradigm.

- In imperative languages the emphasis is on statements and the effects that have.

Imperative effects are important in this unit for two reasons:

- Concurrent programming generally involves sequences of effects.
- Seeing imperative effects in a functional context gives a new perspective.
- In a functional language an effect is a way that an expression affects the rest of the program, or the outside world, beyond just returning a value.
F# imperative programming

- F# includes full support for imperative programming.
  - Mutable variables, arrays, loops, I/O, exceptions.
  - Also .NET objects.

- However, a largely functional style is preferred.
  - Functional code is generally easier to read and reason about.

- The functional and imperative features generally work well together when used carefully.
  - Be careful mixing imperative code with h.o.functions.

- Unlike non-strict functional languages like Haskell, there is no separation between “pure” and “imperative” code.
  - The types do not distinguish which functions have effects.
  - This is convenient because we can easily replace a pure implementation by an imperative one.
  - However, this can make hybrid code less structured.
  - Also, it requires a value restriction on polymorphism
    - Only values can be polymorphic (later).
Mutable reference cells

- Ordinary F# variables & values cannot be modified.
- We would like to be able to modify some vars & values, without making everything mutable.
- Reference cells do this by adding a type constructor for cells that hold a value that can be read and set.

Operations:

```fsharp
let cell1 = ref 1;; // create cell
val cell1 : int ref

cell1;;
val it : int ref = { contents = 1 }

!cell1;;  // get contents
val it : int = 1

cell1 := 3;;  // set contents
val it : unit = ()

cell1;;
val it : int ref = { contents = 3 }

!cell1;;
val it : int = 3
```
References can be passed around, stored in lists, etc.

**Aliasing** is when a cell \( a \) has many references to it.

Aliasing can make the behaviour of code complex.

Note that it is not the variable \( \text{cell1} \) that is modified, it is the cell that it refers to.

This is similar to having many references to an object.

Aliasing complicates many issues in imperative programs.

E.g., Fortran can be better optimized than C (due to pointers)

```ocaml
let cell2 = cell1;;
val cell2 : int ref

!cell2;;
val it : int = 3

cell1 := 7;;
val it : unit = ()

!cell2;;
val it : int 7
```
Records

- **Records** are basically tuples with named components called **fields**.
- Record types must be defined by giving the field names and types.

```haskell
type Person = { Name: string; Age: int }
```

- You can construct record values by providing a value for each field.

```haskell
> { Name="Bill"; Age=23 };;
val it : Person = { Name="Bill"; Age=23 }
```

- You can also give the record type explicitly.

```haskell
> { new Person with Name = "Anna" and Age = 29+4 };;
val it : Person = { Name="Anna"; Age=33 }
```

- The second form is useful when record types share field names.
- The field names are often called **labels**.
Records – accessing fields.

• Fields of records can be accessed via patterns.

  > let isOldBill {Name=n; Age=a} = (n="Bill" && a>21)
  val isOldBill : Person -> bool

• Some fields can be left out provided the type can be determined when labels are shared (use a constraint).

  > let isOld{Age=a} = a>21
  val isOld : Person -> bool

• Fields can also be accessed using the “.” notation (like Java, C)

  let isBill p = (p.Name = "Bill")

• Records can be cloned to create a new record, replacing some fields.

  > let changeName p nm = { p with Name = nm }
  val changeName : Person -> string -> Person

• So far records act in a functional way, very similar to tuples.
Records - mutable fields

- Records can also have mutable fields.

  ```
  type DiscreteEventCounter =
  { mutable Total: int;
    mutable Positive: int;
    Name : string }
  
  let recordEvent (s: DiscreteEventCounter) isPositive =
    s.Total <- s.Total+1
    if isPositive then s.Positive <- s.Positive+1
  ```

- In fact, reference cells are defined as records:

  ```
  type 'a ref = { mutable contents: 'a }
  
  let (!) r = r.contents
  let (:=) r v = r.contents <- v
  let ref v = { contents = v }
  ```

- You can always replace a mutable field by one having a reference type, but this may use more memory.

- Using reference cells rather than mutable fields is more flexible sometimes because you can extract and pass the cell itself around.
Reference cells vs objects & pointers

- A reference cell is essentially an object reference.
  - It has one instance variable and methods for get and set.
  - The reference is immutable – it always refers to the same cell.
  - The contents of the cell/instance variable can be changed.

- Another view is that it is a immutable pointer.
  - The pointer cannot be changed, but what it points to can.

- Consider the following example:
  ```
  let x : int ref = ref 3
  let y : int = !x
  x := (!x) + 1
  y + (!x)
  val it : int = 7
  ```

- We get 7 because y is 3 and x contains 4 (Note terminology).
- y is 3 means that y is always 3. (Just as 3 is always 3.)
- Variables are just names for values in FP, and reference cells are special values which have a contents that can be changed.
Arrays and other imperative collections

- F# arrays work similarly to arrays in Java, etc.
  - They allow modification via the operator (\(-\) : 'a[] -> 'a -> unit
  - They provide fast access, but the length of an array cannot be changed

  ```fsharp
  let arr = [| 1.0; 2.0; 4.0 |];;
  val arr : float[]
  arr.[1];;
  val it : float = 1.0
  arr.[1] <- 3.0;;
  val it : unit = ()
  arr;;
  val it : float[] = [| 1.0; 3.0; 4.0 |]
  ```

- They can also be created via comprehensions and used in generators.

  ```fsharp
  [| for x in arr -> (x,x*x) |];;
  val it : (float * float) array = [|(1.0, 1.0); (3.0, 9.0); (4.0, 16.0)|]
  ```

- The Array library contains versions of nearly all the list/seq funs.
  - map, filter, length, append, ... these work functionally, via copying.
  - There are also functions that update arrays instead of copying.

- There’s also 2D & 3D arrays, and .NET imperative collections (subtypes of seq): resizable arrays, dictionaries (via hash tables), ...
**Input Output (I/O)**

- I/O is another kind of imperative effects.
- F# mostly relies on .NET libraries for I/O
  - System.IO (files, folders, streams), System.Net (network)
    System.Windows.Forms (GUI), many more...
- However, F# has its own formatted printing in FSharp.Text.Printf
  - printf and sprintf work like formatted printing in C, but are statically typed (via special format strings) and printfn adds a newline.
  - %s (strings) %d (decimal) %f (float) %A (lists, tuples,...) %O (objects)

Example - fetching a web page, then closing and printing:

```fsharp
donet

open System.IO; open System.Net

let http(url: string) =
    try
        let reader = new StreamReader(resp.GetResponseStream())
        let html = reader.ReadToEnd()
        html
    with
        exn -> raise exn
    finally
        printfn "Closing %s" url; resp.Close() //Done even when exn.
```

Exceptions

- Exceptions work similarly to those in Java.
  - Although all exceptions are “unchecked”, meaning that they are never included in function types, unlike Java interfaces.
  - Also, throwing .NET exceptions can be slow, even when they are caught, because the OS keeps a record of them anyway.

```plaintext
exception BlockedURL of string
let safeHttp url =
    if url = "http://www.kaos.org" then raise(BlockedURL url)
    else http url    // Assuming fetch was previously defined.

try
    printfn "%d" (safeFetch "http://www.kaos.org").Length
with BlockedURL(url) -> printfn "blocked! url = '%s'\n" url
| _: System.UriFormatException -> printfn "bad url"
| _ -> printfn "Some other exception occurred"
```
Effects and higher-order functions

- Combining imperative effects and higher-order fun’s requires care.
- Effects happen when the function causing the effects has all its arguments.
- Effects inside a fun are delayed, and then repeated each time the fun is applied.
- A semicolon (;) may be used to separate expressions so that the effects of both are done, but the result of the second is returned.

```plaintext
let r = ref 0
let inc () = (r := !r + 1)

let id1 = (inc(); fun x -> x*2.0)
!r;;  // yields 1
let id2 = fun x -> (inc(); x*2.0);
!r;;  // yields 1
id1 2.0; !r;;  // yields 1
id2 2.0; !r;;  // yields 2

let f = List.map (inc(); fun y -> y *2.0);
!r;;  // yields 3
let g = List.map (fun x -> inc(); x*2.0);
!r;;  // yields 3
f [1.0;2.0;3.0]; !r;;  // yields 3
g [1.0;2.0;3.0]; !r;;  // yields 6
```
Objects via records and cells

- Functions, records & cells allow us to “roll our own” objects.

```fsharp
type point = {  
  SetX : float -> unit; // Interface  
  Dist : unit -> float  
}

let point initX initY =  
  let x = ref initX       // Instance variables (state)  
  let y = ref initY  
  let setX newX = (x:=newX)  
  let dist () = sqrt (!x * !x + !y* !y)  
  {SetX = setX; Dist = dist}  // Package into a record “object”

let p = point 2.0 3.0  // Construct an object
p.SetX  4.0          // Call it’s methods
p.Dist()
val it : float = 5.0
```

- The state is hidden, and can only be accessed via the “methods”

- We can also roll similar sorts of structures in F#, as we choose
  - This was roughly how people experimented with early objects