Programming in F#

1. Why F#?

Reason 1: You're programming on the .NET platform
Reason 2: You're working for a company doing a lot of mathematical modelling, lots of financial analysis, etc.

- F# is very expressive in representing your models.

2. What is F#?

F# is a language on the .NET platform developed by Microsoft.

- Statically typed and Strongly typed
  - Very nice type inference.
    - Because of type inference you won't specify types as much.
- You can intermix with other .NET languages on the CLR.
- F# has a different convention from other .NET languages.
  - Lowercase function names

3. Type Inference

You don't need to specify the type in most cases. F# will go deep down into a method, look at your usage pattern in the code and try to infer the type based on those usage patterns.

Let's take a look at an example...

```
let add a b = a + b
```

Notice that we have not declared any data types of `a` and `b`. F# is going to lean to inferring that the types of `a` and `b` are integers because of the `+` operator.

```
let add a b = a + b

printfn "%d" (add 2 4)
```

Lets go a bit further

```
let add a b = a + b

print fn "%d" (add 2 4)

let printSomeThing something = printfn "%s" something

printSomeThing "Hello F#"
```

This code should first return `6` and then returned a string. This is because it inferred the string data type from line 6.

```
let add a b = a + b

print fn "%d" (add 2 4)

let printSomeThing something =
        printfn "%s" something

printSomeThing "Hello F#"

let printSomeThingElse something =
        printfn "%g" something

printSoomethingElse "Hello F#"
```

In this case, we get an error because `printSomeThingElse` is expecting a `double`.

What if I want to send an int, a double, a float. Would I need to write several different methods with different parameters?

- Nope! You can use a **generic type**

Generic Types

```
let add a b = a + b

print fn "%d" (add 2 4)

let printSomeThing something =
        printfn "%s" something

printSomeThing "Hello F#"

let printSomeThingElse something =
```

```
        printfn "%g" something

printSoomethingElse 1.0

// using generic types
let printSomeOtherThing something =
        printfn "%s" (something.ToString())

printSomeOtherThing 2
printSomeOtherThing 1.0
printSomeOtherThing "Hello F#"
```

When we send in `2`, `1.0`, `Hello F#`, it all works. This is because `something` is treated as a **generic type**.

You can let type inference do its job, but you may need to be a bit careful of it. So that the data type that F# thinks it is is the same as what you're thinking.

4. Mutability and Immutability

```
let max = 100

printfn "The max is %s" ((max = 1).toString())
```

When you use the `=` in this case you are making a comparison.

- An `=` in F# is not an assignment operator its a **comparison operator**.
  The **assignment operator** in F# is `<-`

```
let max = 100

printfn "The max is %s" ((max = 1).toString())

let mutable total = 0

printfn "The total is %d" total

total <- 2
printfn "The total is %d" total
```

> Even though F# gives you the capability to use both mutability and immutability, it is best to lean towards using its powerful immutability in a functional style.

5. Functional Programming in F#

F# is not a purely functional language. It's a hybrid functional language.

- Functional language but features **OOP** and mutability.
  * You can create classes in F# but also use functional typing.
  In fact, F# is a multi-paradigm programming language and you can see both imperative and functional styles of coding on display.

So let's run through these styles with some example code that doubles, totals, and finds the max of a list.

6. Imperative Style

```
let printList list =
        for e in list do
                printf "%d" e
        printfn ""

let list = [1; 2; 3; 4; 5; 6]

printf "The original list is "
printList list

// used the mutable keyword to define a mutable list
let mutable doubledList = []

for e in list do
        doubledList <- doubleList @ [e * 2]

printf "The doubled list is "
printList doubledList

//total the elements
let mutable total = 0
for e in list do
        total <- total + e

printfn "The total is %d" total

let mutable max = System.Int32.MinValue
for e in list do
        if max < e do
                if max < e then max <- e
```

```
printfn "The max of values in the lsit is %d" max
```

7. Functional Style

```
// internal iterator to print
let printList list =
        List.iter(fun e -> printf "%d" e) list
        pritnln ""

let list = [1; 2; 3; 4; 5; 6]
printf "The original list is "
printList list

//using the map function to double values.
printf "The doubled list is "
printList (List.map(fun e -> e * 2) list)

printfn "The total is "
printfn "%d" (List.reduce(fun carryOver e -> carryOver + e) list)

// or you can use the sum function
printfn "The total is %d" (List.sum list)

// using a fold method to find the maximum
printfn "The max of values in the lsit is %d" (
        List.fold(fun max e ->
                if max < e then e else max
        ) System.Int32.MinValue list
)

// Or you can simply use the max function
printfn "The max of values in the lsit is %d" (List.max list)
```

> the `fold` function needs to take in the anonymous function, the initial value, and the list it needs to operate on.

> In functional programming, fold functions are a family of high order functions that process a data structure in an order, then they accumulate and return a value.

7. Forward Pipe Operator `|>`

```
let lsit [1; 2; 3; 4; 5; 6]
```

```
printfn "%s" (list.ToString())

// extract only the even numbers out of the list
let evenValues = List.filter(fun e -> e % 2 == 0) list

printfn "%s" (evenValues.ToString())

let doubledEvenValues = List.map (fun e -> e * 2) evenValues

printfn "%s" (doubledEvenValues.ToString())
```

Side-notes from the code above:

> **The Difference between `iter` and `map`:** When you use the `iter` iterator, it iterates over your list without returning anything. While when you use `map`, you are able to iterate and return values.

> In our code we used `filter` which creates a subset of the collection that meet a certain criteria. So it's very similar to `map` but your collection can be smaller in size.

The **forward pipe operator** `|>` allows us to chain functions together to create a **functional composition**.

- Instead of awkwardly creating two variables and referring to the former variable in the declaration of the latter, we can make use of the pipe operator.

Let's take a look at an example.

```
let lsit [1; 2; 3; 4; 5; 6]

printfn "%s" (list.ToString())

// extract only the even numbers out of the list
let evenValues = List.filter(fun e -> e % 2 == 0) list

printfn "%s" (evenValues.ToString())

let doubledEvenValues = List.map (fun e -> e * 2) evenValues

printfn "%s" (doubledEvenValues.ToString())

let giveString obj = obj.ToString()

// using function chaining instead.
list
```

```
        |> List.filter (fun e-> e % 2 = 0)
        |> List.map (fun e -> e * 2)
        |> giveString
        |> prinf "%s"
```

We pipe the list to the `filter`. The `filter` which takes two parameters, gets the first parameter, but the second parameter it expects will be coming in from the chaining.

We can then chain to our print operator.

- We make `giveString` since we would need to call the `toString` method when we print.

Using the pipe operator allows us to eloquently flow between functional operations. We can take values returned from an expression and send it to the next function that's expecting data.

8. List Comprehension

List comprehension in F# isn't as eloquent as it is in Erlang.

```
let isPrime number =
        if [2..number-1] |> List.exists (fun e -> number % e = 0) then false else
true

printfn "3 is prime?: %s" ((isPrime 3).ToString())
printfn "4 is prime?: %s " ((isPrime 4).ToString())
```

> `exists` will tell us whether there is at least one element in our list that meets our criteria. Unlike `filter` which will iterate across all of the elements, `exists` will return as soon as it finds one element.

```
let isPrime number =
        if [2..number-1] |> List.exists (fun e -> number % e = 0) then false else
true

printfn "Primes between 1 and 25 are: "
// list comprehension
[for e in 2..25 do
        if isPrime e then yield e]
        |> List.iter (printf "%d, ")
```

9. Function Values

Let's take a look at defining functions that are expecting functions as parameters.

```
let totalSelectedValues selector list =
        list |> List.filter(fun e -> selector(e)) |> List.sum
let totalValues list = list |> List.sum
let totalEvenValues list =
        list |> List.filter (fun e -> e % 2 = 0)  |> List.sum
let totalOddValues list =
        list |> List.filter (fun e -> e % 2 <> 0) |> List.sum

let list = [1; 2; 3; 4; 5; 6]

// send a function that accepts all the elements given to the function.
list |> totalSelectedValues (fun e-> true) |> printfn "The total is %d"
// send a function that checks that the element sent is even
list |> totalSelectedValues (fun e-> e % 2 = 0) |> printfn "The total is %d"
// send a function that checks that the element sent is odd
list |> totalSelectedValues (fun e-> e % 2 <> 0) |> printfn "The total is %d"
```

We've created a **selector**, (note: selector is not a keyword!), which is a selector of values among a list.

- There are cases where we don't want to select all values from a list. Maybe we only want the even or odd values for instance.

So let's look at an example of a closure

> A **closure** is a persistent scope which holds on to local variables even after the code execution has moved out of that block.

Another definition for context....

> A **closure** is a technique for implementing lexically scoped name binding in a language with first-class functions

```
let list = [1; 2; 3; 4; 5; 6]
let mutable factor = 2
let multiplier e = e * factor

printfn "%s" ((List.map (fun e-> e * factor) list).ToString())
printfn "%s" ((List.map multiplier list).ToString())

factor <- 3

printfn "%s" ((List.map (fun e-> e * factor) list).ToString())
printfn "%s" ((List.map multiplier list).ToString())
```

Notice that the `multiplier` is bound to `e`. We started with `e` equal to 2, and then we changed the value. So as a result we used a new value of e. So this is a closure even though we defined this as a separate function.

Let's use a different way to find a total in a more imperative way as opposed to earlier.

```
let list = [1; 2; 3; 4; 5; 6]
let total = 0
list |> List.iter(fun e -> total <- total + e)

printfn "The total is %d" total
```

So this works just fine, we modify the mutable variable within the closure itself. So this shouldn't be a problem right? But this will need to depend on the situation - it depends on what this value total is.

Let's look at such a tricky case...

```
let totalValues list =
        let mutable total = 0
        list |> List.iter(fun e -> total <- total + e)
        total

let list = [1; 2; 3; 4; 5; 6]
let total = list |> totalValues

printfn "The total is %d" total
```

F# is complaining now about line 3, it will say that the mutable variable 'total' is used in an invalid way.

- We aren't allowed to use `total` in line 3 in our anonymous code block.
  *When you have a code block or a closure, **you can store it in a pointer or a variable**, you can pass it around. Rather than storing it in a code block, you can store that into a local variable and call it later on.*
  If that code block is using a certain variable, the life of that variable is under question.
  Because, If I store your code block as a pointer and that code block is a reference to some variable and that variable is citing some other function. The minute you leave that other function, the stack collapses, that variable is gone... what happens to my function?
  If you call `totalValues`, `total` is on the stack and you create this code block: `fun e -> total <- total + e` and assume that you send this code block to some other function, and you leave `totalValues`. That folds the `total` so **now this code block is holding onto a**

variable that doesn't exist!!
If we did not declare `total` as `mutable` and we were simply referring to it then its not an issue, F# could make a copy of it locally.

So in order to fix this problem with a `mutable` variable `total`, we will need to push it onto the heap so our function can access it without difficulty.

```
let totalValues list =
        let mutable total = ref 0
        list |> List.iter(fun e -> total := <- total + e)
        !total

let list = [1; 2; 3; 4; 5; 6]
let total = list |> totalValues

printfn "The total is %d" total
```

> We put `total` on the heap by using the `ref` keyword. We then use a different syntax to assign `total` on the heap: `:=`.