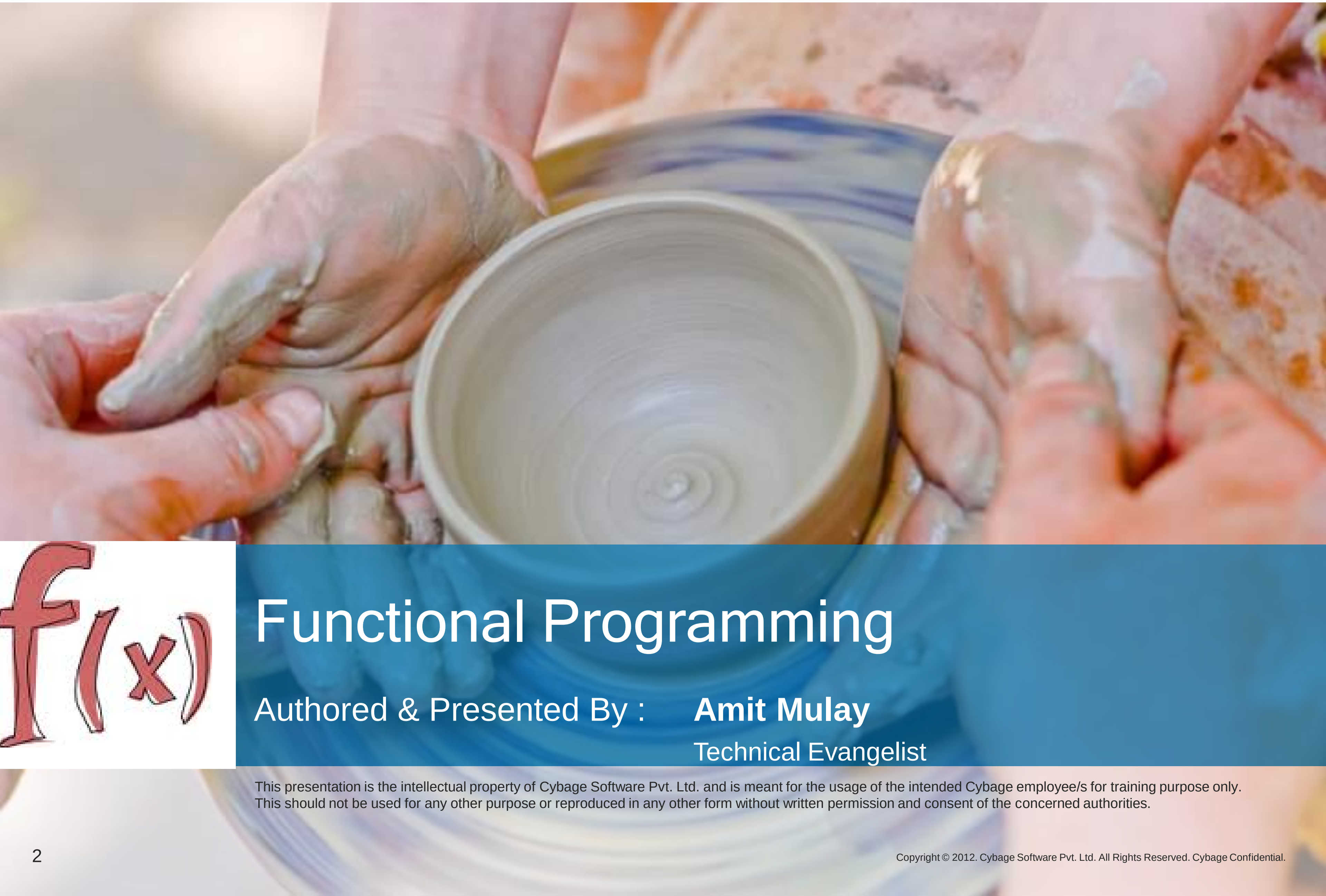
Technology Group

Complexity

Simplified



Functional Programming

Authored & Presented By : **Amit Mulay**

Technical Evangelist

Recap

- Balanced approach to Noun and Verb concepts
- References are todays global variables
- Value types are preferred over references
- Immutability is preferred
- Declarative coding style is preferred
- Generic coding is preferred
- Compile time checks are preferred.

Functions

- Lamda

  `f(x) = x + 10`

  `f(32) = 32 + 10 = 42`

  corresponding lambda

  $(\lambda x.x + 10)$

  $(\lambda x.x + 10)\ 32 = 32 + 10 = 42$

Lamda is just fancy name to functions with single parameter and single return value.

Functions

One more Example

(λop.λx.(op x x))

Try

(λop.λx.(op x x)) (+) 21

Hello world

```
let message = "Hello world!"
printfn "%s" message
```

O/P is

```
> (...);;
Hello world!
val message : string = "Hello world!"
```

Points to Note

1. Message is value not variable. Let is for value binding.
2. Function calling – No parentheses, comma
3. Types are not mentioned. And yet compiler infered its type to string.

Functions

Requirement : Sqr all odd numbers from 1 to 10

```
Array a = new Array(10);
for(int i = 1; i <= 10; i++)
{
        if (i % 2   == 1) then
                a.Add(i*i);
}
```

Lets Analyze our thought process.

1. We thought about memory locations to store results.
2. We thought of moving through numbers by incrementing 1 each time
3. We combined logic using sequence of steps and loops.
4. As a result logic of odd and square is hidden in imperative code.

[Imperative vocabulary is added by sequence of commands. OOPs does great job in arranging it.]

Functions

Requirment : Sqr all odd numbers from 1 to 10

Instead of

1. We thought about memory locations to store results.
2. We thought of moving through numbers by incrementing 1 each time

Can’t we just think of list of numbers. No memory allocations or how to create list.

How about?

```
let numbers = [1..10]

/*
val numbers : int list
*/
```

Most common datastructure

List : basic data-structure in almost every functional language

[1..10]

[1;2;3;4] (→ Beware comma vs semicolon!)

[1..2..10] -> list of odds -> [1;3;5;7;9]

Cons operator : 1 ::[2;3]

Concat : [1;2] @ [3;4]

Pattern matching

match list with

| [] -> …

| h::t -> …

Most common datastructure

List : basic data-structure in almost every functional language

It is recursive type!!

```
type AList<'a> =
    | Empty
    | Cons of 'a * AList<'a>
```

Recursive functions!

```
let rec sumlist l =
            match l with
            | [] -> 0
            | h::t -> h + processlist t
```

Functions

Requirement : Sqr all odd numbers from 1 to 10

Instead of

1. We combined logic using sequence of steps and loops.
2. As a result logic of odd and square is hidden in imperative code.

Can’t we think first of business logic and then about how to sequence logic?

```
let numbers = [1..10]
let isOdd n = n % 2 = 1
let sqr n = n * n
/*
val numbers : int list
val isOdd : int -> bool
val square : int -> int
*/
```

Functions

```
Requirment : Sqr all odd numbers from 1 to 10
```

Now we need to combine following,

```
let numbers = [1..10]
let isOdd n = n % 2 = 1
let sqr n = n * n
```

We can use High order functions for combining things instead of loops/sequencing

```
List.filter isOdd numbers      (* try using sqr instead of isodd and you will get error*)
/*
val it : int list = [1;3;5;7;9]
*/
List.map square (List.filter isodd numbers)
/*
val it : int list = [1;9;25;49;81]
*/
```

Functions :Higher order functions

List.map

List.filter

List.fold

List.collect (bind)

1. Takes functions as parameters
2. Gives us ability to avoid loops
3. Gives better abstraction [I mean logic of iterating can be changed.
   like parallel]

Map, filter, fold, collect are like design patterns. You will find it is good idea and practice to define them on any data-structure (say tree, graph) that you define.

Functions : Readability

```
List.map square (List.filter isodd numbers)
```

WHAT ABOUT READABILITY??

Functions : Readability

List.map square (List.filter isodd numbers)

```
Soln : Pipelining operator
```

```
let (|>) x f = f x
```

Following

    List.map square (List.filter isodd numbers)

Becomes

```
numbers
|> List.filter isodd
|> List.map square
```

Functions and lambda

```
let multiply num1 num2 =     num1 * num2;;
val multiply : int -> int -> int
```

Remember lambda.

$(\lambda num1.\lambda num2.(num1 * num2))$

Lets check lambda equivalent

```
    let multiply = fun num1 num2 -> num1 * num2
    /* val multiply : int -> int -> int */
    let multiply = fun num1 -> (fun num2 -> num1 * num2))
    /* val multiply : int -> int -> int */
```

Another example

```
let square x = x * x              /* val square : int -> int*/
let square = fun x -> x * x  /* val square : int -> int*/
```

Immutability

If something can’t be modified and lets see what implication it has on some of object oriented concept like Class.

Public/private/protected : You don’t need.
Setter you don’t need.
Member functions you don’t need…since everything is static…you need static functions…which are normal functions in functional programming

Constructors : You need
Getter you will probably need.
Inheritance you need.

Can we have light weight replacement of class since setter, public/private/protected, member function is not needed?

Discriminated union

Defining shapes (ellipse, rectangle, composed)

```
type Shape =
| Rectangle of Point * Point
| Ellipse of Point * Point
| Composed of Shape * Shape
```

(How easily we defined recursive data structure?   IT IS COMPACT Class hierarchy!!)

For OOPS,

1. Shape is base class
2. Ellipse, Rectangle and Composed are child classes
3. Ellipse, Rectangle constructor takes tuple (point, point) as parameter

4. Composed takes base classes as parameters.

Discriminated union

How to create? ….Like any constructor

```
let ellipse = Ellipse(Point(1,2), Point(3,5))
let rec = Rectangle(Point(1,2), Point(3,5))
Let comb = Composed(ellipse, rec)
```

How to fetch ???

Soln is pattern match

```
match shape with
| Ellipse(point1, point2) -> …
| Rectangle(point1, point2) -> …
| Shape(shape1, shape2) -> …
```

Serves as Getter and also ensures opn is define for all subclasses

FP and OOPs

As we have seen
Because of Immutability
lot of things of OOPS concept of class are not needed.

FP's modules and functions are equivalent to static classes and static methods.

When it comes to types/Nouns, its not that FP wants to go away with it. FP uses concepts of types etc which are equivalent of what we find in OOPS. Just that lot of features/concepts of OOPS are not needed in FP because of immutability.

Tuples

let k = (1, “aaaa”)
/*
Val k : int * string
*/
Same as class just that it doesn’t have field names. Getters are done through pattern matching.

let a,b = k
…
Match k with
|(1,_) -> ..
| (a, b) -> ..
Very light weight especially for lot of Mathematical algos/concepts.
And used in DUs.

You can return tuple from a function i.e. you can return multiple values. So no need of references or out parameters.

Units of measure!

NASA's metric confusion caused Mars orbiter loss

September 30, 1999
Web posted at: 1:46 p.m. EDT (1746 GMT)

(CNN) -- NASA lost a $125 million Mars orbiter because one engineering team used metric units while another used English units for a key spacecraft operation, according to a review finding released Thursday.

Units of measure!

```
[<Measure>]
type m
[<Measure>]
type sec
[<Measure>]
type kg
let distance = 1.0<m>
let time = 2.0<sec>
let speed = 2.0<m/sec>
let acceleration = 2.0<m/sec^2>
let force = 5.0<kg m/sec^2>
```

Units of measure!

```
[<Measure>]
type foot
[<Measure>]
type inch
let distance = 3.0<foot> // type inference for result
let distance2 = distance * 2.0 // type inference for input and output
let addThreeFeet ft = ft + 3.0<foot>

addThreeFeet 1.0 //error
addThreeFeet 1.0<inch> //error
addThreeFeet 1.0<foot> //OK
```

Function signatures!!

- int -> int -> int
- int -> unit
- unit -> string
- int -> (unit -> string)
- (**'**a -> bool) -> **'**a list -> **'**a list
- (**'**a -> **'**b) -> **'**a list -> **'**b list

type Adder = int -> int
type AdderGenerator = int -> **Adder**

Any Questions?

Thank you!