FSC104

Partial Application and Pattern Matching

- Lecture will begin shortly
- Download class materials from university.xamarin.com

Xamarin University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft or Xamarin may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft or Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, Xamarin Studio, and Visual Studio are either registered trademarks or trademarks of Microsoft in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Objectives

1. Examine functional aspects of F#
2. Employ pattern matching
3. Apply active patterns

Examine functional aspects of F#

Tasks

1. Discuss some of the functional features of F#
2. Experiment with option types
3. Apply partial application

Thinking functionally

- Although F# is not a purely functional language it is considered a **functional-first language**

Pipelining

Option Types

Higher-order Functions

Forward pipe operator

- The forward pipe operator (|>) allows you to pass a result into the next function

```
let getData =
   List.sum (List.filter (fun x -> x%2=0) ([1..10]))
```

```
val getData : int = 30
```

Forward pipe operator

- The forward pipe operator (|>) allows you to pass a result into the next function

```
let getData =
    List.sum (List.filter (fun x -> x%2=0) ([1..10]))
```

```
let getData =
    [1..10]
        |> List.filter (fun x -> x%2=0)
        |> List.sum
```

```
val getData : int = 30
```

Backward pipe operator

- Backward pipe operator (<|) is similar to the forward pipe operator, but it works from right-to-left; it is often used to replace parenthesis in function calls being passed to other functions

```
let distance x y =
    let square x = x*x
    sqrt (square x + square y)
```

here we are forced to surround our inner call with parenthesis to provide a single value for `sqrt`

Backward pipe operator

- Backward pipe operator (<|) is similar to the forward pipe operator, but it works from right-to-left; it is often used to replace parenthesis in function calls being passed to other functions

```
let distance x y =
    let square x = x*x
    sqrt <| square x + square y
```

parenthesis are unnecessary because expression is evaluated right-to-left and single value is passed into `sqrt`

Option types

- The **Some** or **None** keywords are used to declare an **option type**, useful when an actual value **is missing or invalid** for a given named value

```
let x = Some(7)
```

```
val x : int option = Some 7
```

The Some keyword represents the presence of a value

Option types

- The **Some** or **None** keywords are used to declare an **option type**, useful when an actual value **is missing or invalid** for a given named value

```
let x = Some(7)
```

```
let doubleVal x =
    if x > 0 then Some(x * 2)
    else None
```

The **None** keyword represents the absence of a value

```
> doubleVal 0;;
val it : int option = None
```

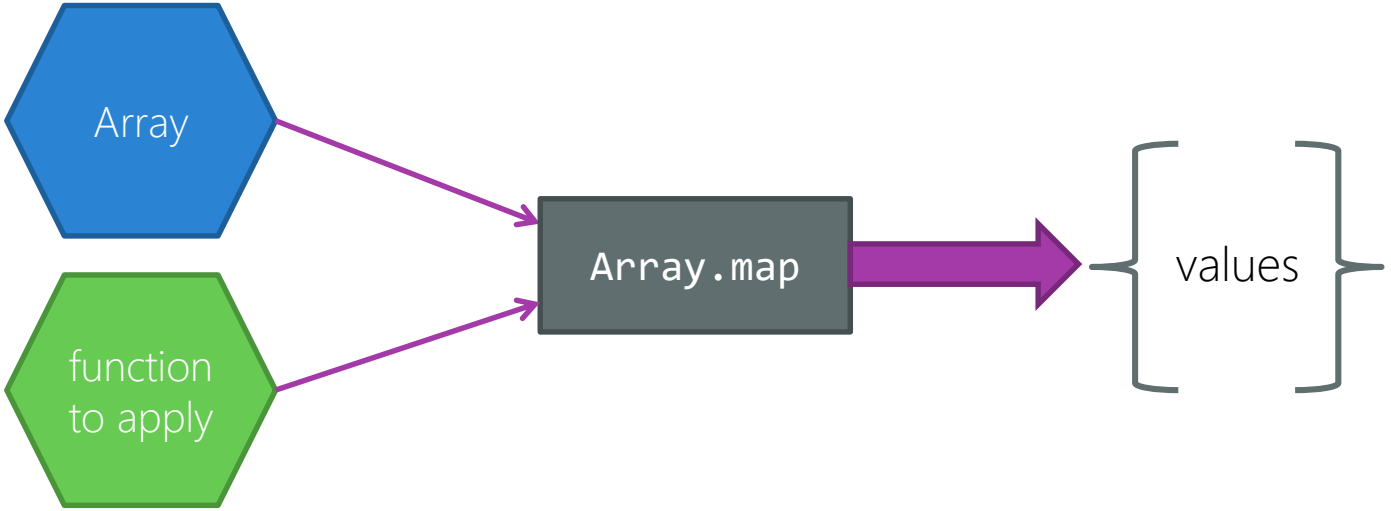
Option types are used in the F# libraries to return **invalid values**, for example **List.tryFind** will return **None** if the value cannot be located

Group Exercise

Utilizing option types to sum numbers

Higher-order functions

- Higher-order functions are those that take functions in as parameters and/or return a function as a value (**`List.filter`**, **`Array.map`**, etc.)



Returning results from functions

- F# *does not* use a **`return`** keyword, instead the expression which is evaluated on the **final line of the function** will be returned as the result of the function

```
let getData numbers =
  let filteredList = numbers |> List.filter (fun x -> x%2=0)
  List.sum <| filteredList
```

```
val getData : numbers:int list -> int
```

This implicitly becomes the result for the **`getData`** function

This results in code which is more concise and easier to read

Currying

- The process of taking a function that has multiple parameters and re-working it to be a chain of several functions that take **just one parameter** is called **currying**

```
let multiply a b = a * b
```

This is a function that takes two integer values and returns a single integer result

```
val multiply : a:int -> b:int -> int
```

F# rewrites it as a function that takes a *single* integer which passes into another function with another integer to create our final integer result

Partial Function Application

- F# allows you to create a new function which **fixes a set number of parameters** to another function, this is called a *partial function application*

```
let multiply a b = a * b

let double a = multiply a 2
```

This is a function that takes a single integer value and returns a doubled integer value by **currying** the multiply method

This technique is a widely used programming style in F# and is used to simplify functions and improve reusability

Partial Function Application

- F# allows you to create a new function which **fixes a set number of parameters** to another function, this is called a *partial function application*

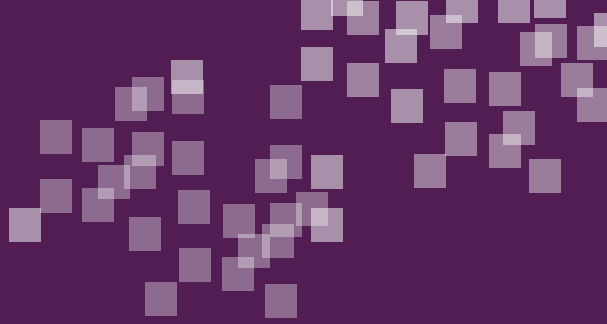
```
let multiply a b = a * b
```

```
let double a = multiply a 2
```

```
List.map (multiply 2) [0..10]
```

This utilizes partial application to map a list of integers to their doubled values

```
val it : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18; 20]
```



Individual Exercise

Applying partial application

Xamarin University

Summary

1. Discuss some of the functional features of F#
2. Experiment with option types
3. Apply partial application

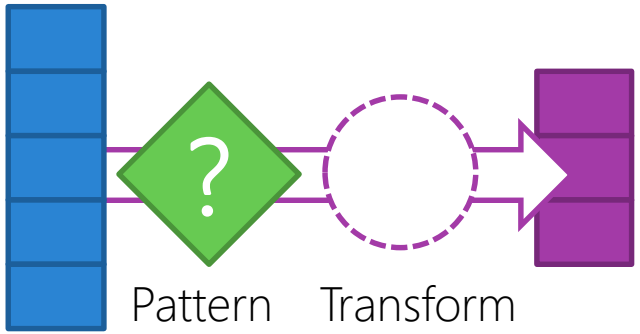
Employ pattern matching

Tasks

1. Define pattern matching
2. Illustrate the syntax of pattern matching
3. Examine some of the different types of pattern matching

What is pattern matching?

- Pattern matching is used to match values against other values of the same type
- It is most often used to apply transformation functions to matching values



Pattern matching in F#

- Pattern matching allows code to decompose, extract and bind values in expressions and functions

```
match expr with
    | pattern1 -> result1
    | pattern2 -> result2
    | pattern3 when expr2 -> result3
    | _ -> defaultResult
```

Patterns are match *in-order*, with the first matching pattern being used, should include the most constrained pattern first, moving down to the most open pattern

Pattern matching in F#

- Pattern matching allows code to decompose, extract and bind values in expressions and functions

The `match` expression starts a pattern match and is used to examine the data to see if it is compatible with the pattern

```
let divisibleByThree x =
    match x%3 with
    | 0 -> true
    | 1 -> false
    | _ -> false
```

Pattern matching in F#

- Pattern matching allows code to decompose, extract and bind values in expressions and functions

```
let divisibleByThree x =
    match x%3 with
    | 0 -> true
    | 1 -> false
    | _ -> false
```

Comparison values provide a function which is executed when the pattern expression generates that value

Wildcard character matches *any* result and is **always placed last** in the comparison value list

Pattern guarding

- To add an additional condition on a pattern match we use the **when** clause, we call these **pattern guards**; pattern guards are similar to using "**if**" statements to filter your data

```
match sign with
    | 0 -> 0
    | x when x < 0 -> -1
    | x when x > 0 -> 1
```

Expression defines the test condition

Pattern matching functions

- You can use the **`function`** keyword as shorthand when writing pattern matching functions which do not require access to the parameter(s)

```
let printZeroValue point =
  match point with
     | (0, 0) -> printfn "Both values zero."
     | (0, _) -> printfn "First value is 0."
     | (_, 0) -> printfn "Second value
     | _ -> printfn "Both nonzero."
```

equivalent

```
let printZeroValue = function
     | (0, 0) -> printfn " ... "
    ...
```

What kind of patterns are there?

- F# supports a variety of pattern styles used as comparison values

Constant pattern	AND pattern
Discriminated union pattern	OR pattern
Tuple pattern	List and array patterns
Wildcard pattern	Cons pattern
Record pattern	Active patterns

Constant pattern

- The constant pattern allows the match expression or function to be compared to any numeric, character, or string literal; similar to a C# **`case`** comparison

```
let isSecretAgent url, agentId =
    match (url,agentId) with
    | "http://www.control.org", "99" -> true
    | "http://www.CHAOS.org", _ -> true
    | _, "007" -> true
    | _ -> false
```

Discriminated Union pattern

- Patterns allow you to provide an action for each of the options of a DU

```
type Shape =
    | Circle of double
    | Triangle of double * double * double
    | Rectangle of double * double
```

```
let getArea shape =
    match shape with
    | Circle(r) -> r * r * System.Math.PI
    | Triangle(a, b, c) ->
        let s = (a + b + c) / 2.
        sqrt s * (s - a) * (s - b) * (s - c)
    | Rectangle(a, b) -> a * b
```

Tuple pattern

- The tuple pattern matches input in tuple form and enables the tuple to be decomposed into its constituent elements by using pattern matching variables for each position in the tuple.

```
let displayPoint point =
    match point with
      | (0, 0) -> printfn "Both values zero."
      | (0, s2) -> printfn "1st value is 0 in (0, %d)" s2
      | (s1, 0) -> printfn "2nd value is 0 in (%d, 0)" s1
      | (s1, s2) -> printfn "(%d, %d)" s1 s2
```

Pattern matching on records

- Matching records allows you to match some fields in the record and provide variables for other fields when the match occurs.

```
type Person = {Name: string; Id: int;}

let MatchByName person name =
    match person with
    | { Name = nameFound; Id = _; } when nameFound = name -> true
    | _ -> false
```

```
let me = {Id=1234; Name="Helen"}
MatchByName me "Mark"                    // false
MatchByName me "Helen"                   // true
```

OR pattern

- The OR pattern allows you to give the program multiple patterns of the same type to match

Multiple conditions return the same result – this is a perfect candidate for the OR pattern

```
let validPoint point =
    match point with
    | (0, 0) -> false
    | (0, _) -> false
    | (_, 0) -> false
    | _ -> true
```

OR pattern

- The OR pattern allows you to give the program multiple patterns of the same type to match

```
let validPoint point =
    match point with
    | (0, 0) | (0,_) | (_,0) -> false
    | _ -> true
```

Matching pattern expressions are separated with vertical bar (|) and all return the same result

AND pattern

- The AND pattern requires multiple patterns match the value, these are primarily used to extract the values out when using wildcard matches

```
let getPoint point =
    match point with
    | (0,0) -> "Origin"
    | (x,y) & (0,_) -> sprintf "Y:%d" y
    | (x,y) & (_,0) -> sprintf "X:%d" x
    | (x,y) -> sprintf "(%d,%d)" x y
```

Can match when one coordinate is on the axis and use the captured value, notice we use a single **&** for the AND pattern

List and array patterns

- A list/array pattern will decompose a one-dimensional list or array and search for specific elements

```
let checkStooges list =
    match list with
    | [] -> "empty"
    | [ "Moe" ] -> "Just Moe"
    | [ "Moe"; "Larry"; n ] -> sprintf "Larry, Moe and %s" n
    | _ -> "Not valid."
```

Notice that we must know the number of elements, and the order we are comparing against – you cannot write pattern matches to handle lists or arrays of unknown lengths

Cons Pattern

- The **cons** pattern transforms a list into two elements: the *head*, which contains the first item, and the *tail*, which contains the rest of the list

Cons pattern is often used recursively as shown here

```
let rec printList lst =
    match lst with
    | [] -> printfn "empty"
    | head :: tail ->printf "%d " head; printList tail
```

Cons symbol (`::`) is used to separate head from tail

Can use semicolon to include multiple statements from result expression

Beware F# warnings

- Patterns are evaluated top-down, the first matching pattern is returned, if there is no match at runtime **an exception is thrown**

```
let CityFromZip zip =
    match zip with
    | 75080 -> "Richardson, Texas"
    | 99701 -> "Fairbanks, Alaska"
    | 90210 -> "Beverly Hills, California"
```

warning FS0025: Incomplete pattern matches on this expression. For example, the value '0' may indicate a case not covered by the pattern(s).

Group Exercise

Practice pattern matching

Xamarin University

Flash Quiz

Xamarin University

Flash Quiz

① How would you rewrite the function below to use the backpipe operator?

a) `printfn "the tripled square of 135 is %f" <| tripleSquare 135.0`
b) `printfn "the tripled square of 135 is %f" tripleSquare 135.0 <|`
c) `printfn <| "the tripled square of 135 is %f" tripleSquare 135.0`

```
printfn "the tripled square of 135 is %f" (tripleSquare 135.0)
```

Flash Quiz

① How would you rewrite the function below to use the backpipe operator?

a) `printfn "the tripled square of 135 is %f" <| tripleSquare 135.0`
b) `printfn "the tripled square of 135 is %f" tripleSquare 135.0 <|`
c) `printfn <| "the tripled square of 135 is %f" tripleSquare 135.0`

```
printfn "the tripled square of 135 is %f" (tripleSquare 135.0)
```

Flash Quiz

② The **when** clause

a) Is used to create a pattern guard

b) Is used to start a pattern match

c) Is handy because it reduces the need for parenthesis

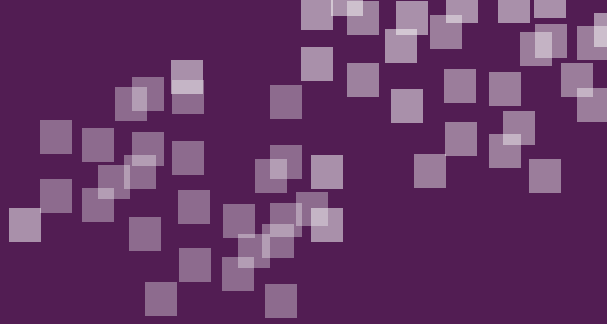
Flash Quiz

② The **when** clause

a) Is used to create a pattern guard

b) Is used to start a pattern match

c) Is handy because it reduces the need for parenthesis



Individual Exercise

Apply pattern matching to create a daily routine

Xamarin University

Summary

1. Define pattern matching
2. Illustrate the syntax of pattern matching
3. Examine some of the different types of pattern matching

Examine active patterns

Xamarin University

Tasks

1. Define active patterns
2. Illustrate the different types of active patterns
3. Employ active patterns

Active patterns

- Active patterns are functions that allow you to partition data into named sections

```
let (|identifer1|identifier2|...|) [ arguments ] = expression
```

Active patterns

- Active patterns are functions that allow you to partition data into named sections

The name of the function includes the names of the partitions and is surrounded by banana clips (| and |)

```
let (|Adult|Child|) age =
    if age >= 21 then Adult else Child
```

Active patterns

- Active patterns are functions that allow you to partition data into named sections

```
let (|Adult|Child|) age =
    if age >= 21 then Adult else Child
```

```
let CanVote person =
    match person.Age with
    | Adult -> true
    | Child -> false
```

Complete active patterns

- A complete active pattern partitions data in **up to seven distinct groups** to handle all of the data

```
let (|Q1|Q2|Q3|Q4|) (date : System.DateTime) =
    let month = date.Month
    match month with
    | 1 | 2 | 3 -> Q1 month
    | 4 | 5 | 6 -> Q2 month
    | 7 | 8 | 9  -> Q3 month
    | _ -> Q4 month
```

Here we break an input date into one of four groups based on the month

Complete active patterns

- A complete active pattern partitions data in **up to seven distinct groups** to handle all of the data

This is then used in a secondary match to make it easier to break up the data

```
|Q3|Q4|) (date : System.DateTime) =
 = date.Month
```

```
| 4 | 5
| 7 | 8
| _ -> Q4
```

```
let myGoals date =
    match date with
    | Q1 _-> printfn "Make New Year resolution!"
    | Q2 _-> printfn "Break out the grill!"
    | Q3 _ -> printfn "Aren't the leaves pretty?"
    | Q4 _-> printfn "Time to eat!"
```

Partial active patterns

- Incomplete, or partial active patterns, only define one partition for the data and exclude data that does not match

The |_| indicates that some values will not produce a result

```
let (|Integer|_|) (s: string) =
    match System.Int32.TryParse(s) with
    | (false, _) -> None
    | (true, n) -> Some(n)
```

Partial active patterns

- Incomplete, or partial active patterns, only match part of the data and ignore everything that does not match

```
let (|Integer|_|) (s: string) = ...
let (|Float|_|) (s: string) = ...
```

Common to define multiple partial active types to partition the same group of data into different groups

Partial active patterns

- Incomplete, or partial active patterns, only match part of the data and ignore everything that does not match

```
let (|Integer|_|) (s: string) = ...
let (|Float|_|) (s: string) = ...
```

```
let printValue str =
    match str with
    | Integer i -> printfn "%d is an integer" i
    | Float f -> printfn "%f is a float" f
    | _ -> printfn "%s is not numeric" str
```

Flash Quiz

Xamarin University

Flash Quiz

① The `function` keyword

a) Is only used with active patterns

b) Is used as shorthand when writing pattern matching functions

c) Is the only way to write a pattern matching function

Flash Quiz

① The `function` keyword

a) Is only used with active patterns

b) Is used as shorthand when writing pattern matching functions

c) Is the only way to write a pattern matching function

Flash Quiz

② Which of these functions is an “active pattern”?

a) `let (IsATron) (s : string) = |s.EndsWith|("TRON”)`

b) `let (IsATron) (s : string) = s.EndsWith("TRON”)`

c) `let (|IsATron|) (s : string) = s.EndsWith("TRON”)`

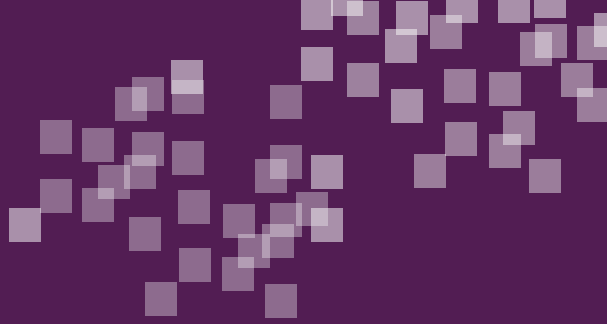
Flash Quiz

② Which of these functions is an “active pattern”?

a) `let (IsATron) (s : string) = |s.EndsWith|("TRON”)`

b) `let (IsATron) (s : string) = s.EndsWith("TRON”)`

c) `let (|IsATron|) (s : string) = s.EndsWith("TRON”)`



Individual Exercise

Utilizing active patterns to create a flight itinerary

Xamarin University

Summary

1. Define active patterns
2. Illustrate the different types of active patterns
3. Employ active patterns

Where are we going from here?

- You now know how to use various pattern matching expressions in F#
- In the next course, we will look at how to use type providers to analyze and process data

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile