

Building Xamarin apps with F#

Sam Williams - Mobile Developer at EROAD

-twitter @Willsam100 -blog: codingWithSam.com

```
99 little bugs in the code,  
99 bugs in the code,  
1 bug fixed...compile again,  
100 little bugs in the code.
```

What is F#

- Supports Object Orientated programming
- Functional programming language
- Runs on .NET
- Xamarin support for cross-platform mobile developement
- Developed by Microsoft
- Fully supported by Microsoft tools

F# Forms Hello World

```
1: type App() =  
2:     inherit Application()  
3:  
4:     let stack = new StackLayout(VerticalOptions = LayoutOptions.Center)  
5:     let label = Label(XAlign = TextAlignment.Center,  
6:                       Text = "Welcome to F# Xamarin.Forms!")  
7:     do  
8:         stack.Children.Add(label)  
9:         base.MainPage <- ContentPage(Content = stack)
```

Working through a simple app

- UI to square a number
- input, button, output
- Handle failure
- Still a little challenging

The UI

```
1: let stack = new StackLayout(  
2:     VerticalOptions = LayoutOptions.FillAndExpand)  
3:  
4: let first = Entry()  
5: let button = Button(Text = "Square")  
6: let output = Label(XAlign = TextAlignment.Center, Text = "")  
7: do  
8:     stack.Children.Add(first)  
9:     stack.Children.Add(button)  
10:    stack.Children.Add(output)  
11:    this.Content <- stack  
12:
```

A note about the Syntax

- F# uses spaces instead commas for parameters
- F# uses indetentation instead of curly braces

I think we need a function

```
1: let square x = x * x
```

- pure function
- will always return the same result with same input
- no mocking required
- type inference; it's an int

Button handlers

```
1: // ...
2:
3: let output = new Label(XAlign = TextAlignment.Center, Text = "")
4: let buttonHandler x = () // Goal is to implement this
5: do
6:     button.Clicked.AddHandler (new EventHandler(fun x -> buttonHandler(x)
7:     stack.Children.Add(first)
8:     // ...
9:
```


Following the types

- use types to model the domain
- follow the types till it compiles
- design the types to ensure correctness
- more types the better!

First attempt

```
1: // hmm this is not going to work
2: let buttonHandler x =
3:     output.Text <- (square first.Text) // no compile
4: // first.text: string
5: // output.Text: string
6: // square: int -> int
7:
```

Pattern matching

Coming to C#

```
1: // this function will help
2: Int32.TryParse : string -> (bool * int)
3:
4: // tryParse: string -> int option
5: let tryParse x =
6:     let result = Int32.TryParse x
7:     match result with
8:     | true, v    -> Some v
9:     | false, _  -> None
```

Handling failure

- why not just check result of `tryParse`?
- error prone
- Option type is a better abstraction of failure
- better compiler errors

Option type

```
1: type Option =  
2:   | Some of a'  
3:   | None
```

- encapsulates the case of failure
- very similar to a list with only 1 item
- built into F#

Using Option

```
1: // Call tryParse, then check result, returning a string
2: let result = match tryParse first.Text with
3:     | Some y -> (square y).ToString()
4:     | None   -> "enter a number"
5: // Won't compile if missing either Some or None
6: // Both branches must return the same type
7:
```

The final handler

```
1: let buttonHandler x =  
2:   let result = match tryParse first.Text with  
3:                 | Some y -> (square y).ToString()  
4:                 | None   -> "enter a number"  
5:   output.Text <- result
```

What have we learned

- A stronger type system helps for the error case
- type inference makes the solution flexible
- Option is great for modelling

Data types

- the option type represented failure
- you can write your own types
- immutability is free
- F# is built to handle data

Awesome things not covered

- type providers
- units of measure
- property based testing
- F# has great support for parallelism:
- async and MailboxProcessor

Thast a wrap

- I hope you learning something
- Any questions?