FSC101

Introduction to F#

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

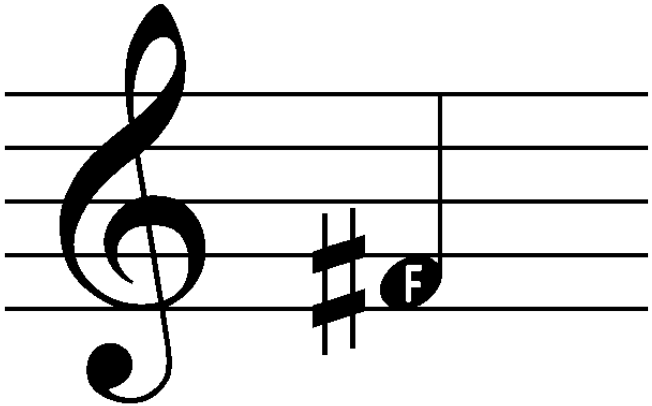
Objectives

1. Explain why is F# important
2. Execute F# code in the REPL
3. Working with expressions and loops

Explain why is F# important

Tasks

1. Outline the history of F#
2. Describe functional programming
3. Define and examine immutability
4. Identify advanced features of F#
5. Evaluate the benefits of using F#



What is F#?

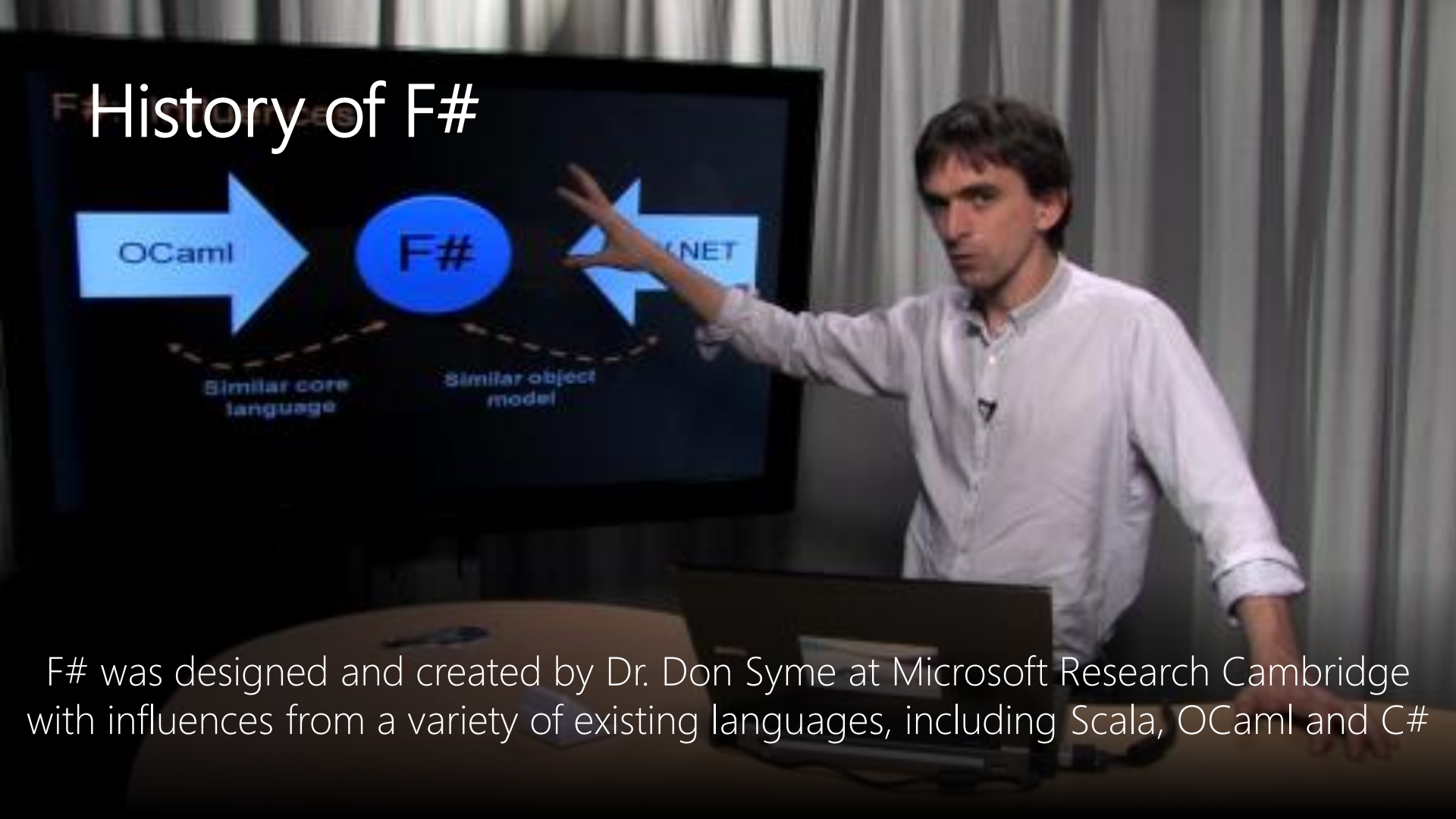
- F# is a succinct, expressive and efficient hybrid-functional programming language for the .NET platform

```
open System
let a = 2
Console.WriteLine a
```

F# vs. C#

```
using System;

namespace CSharpExample
{
    class Program
    {
        public static void Main() {
            int a = 2;
            Console.WriteLine(a);
        }
    }
}
```

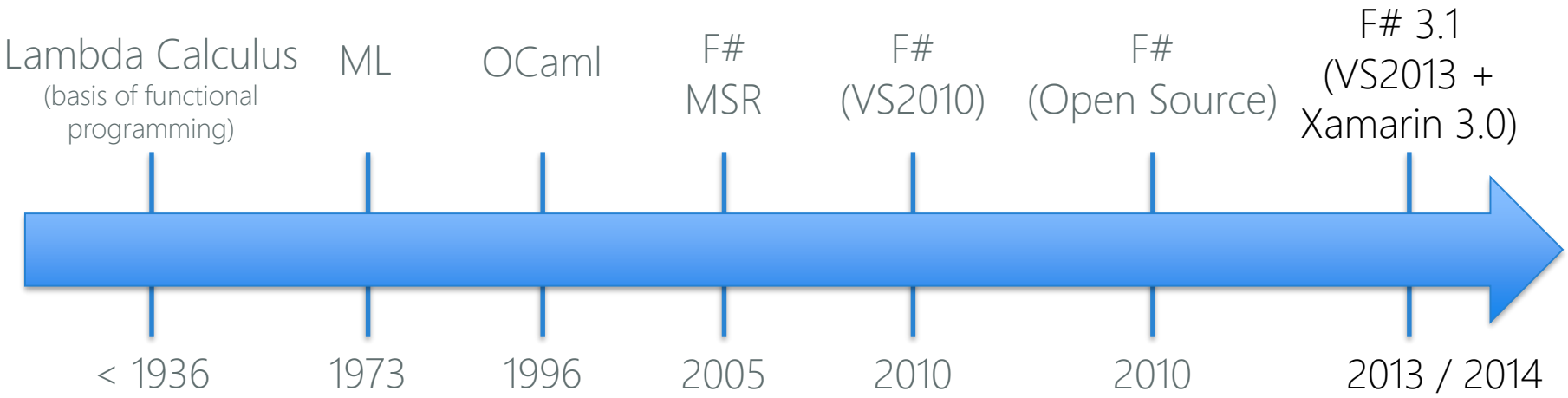
History of F#



F# was designed and created by Dr. Don Syme at Microsoft Research Cambridge with influences from a variety of existing languages, including Scala, OCaml and C#

Evolution of F#

- F# started as an unsupported download but is now fully supported and included with VS2010 and beyond



What is Functional Programming?

- Functional programming is a style of programming that models computations as the evaluation of expressions while avoiding state

```
Func<int,bool> isEvenNumber = n => n % 2 == 0;

var numbers = Enumerable.Range(1, 10);

numbers.Where(isEvenNumber)
       .ToList()
       .ForEach(Console.WriteLine);
```

Results are calculated only on the basis of input values – no information is stored

LINQ is an example of functional programming using the C# language

Thinking about expressions

- What does this expression mean to a programmer?
- How about a mathematician?

$$X = X + 1$$

Problems with changing data

- Changing (mutable) data means we cannot accurately predict values without complete knowledge of what has happened before the current statement – this directly impacts code optimization

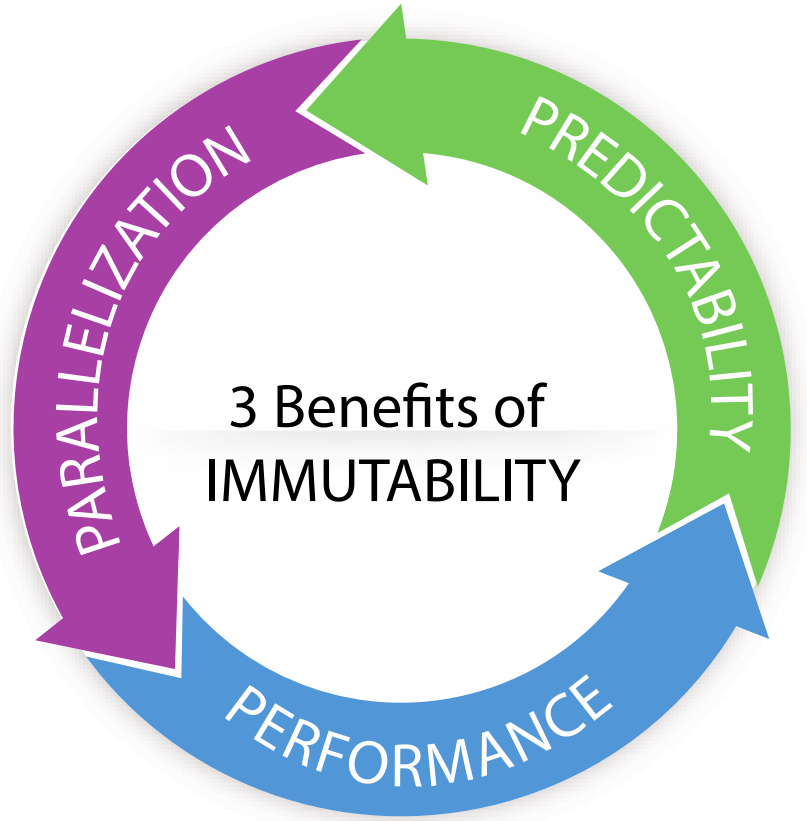
```
int a = 10;
int b = 20;
int c = a * b;

a = 50;
```

should **c** change?

This style of code can create side-effects and bugs which are hard to identify and reproduce, which makes them hard to fix

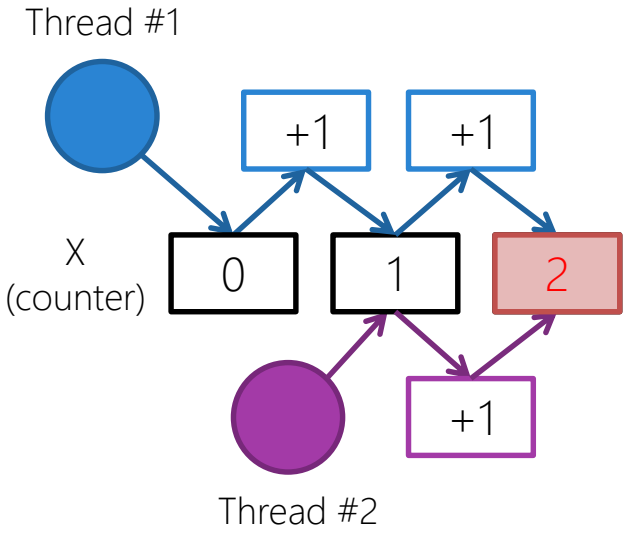
What is immutability?

- Immutability means that values cannot be modified once they are assigned which simplifies our code and provides three key benefits



Parallelization in C#

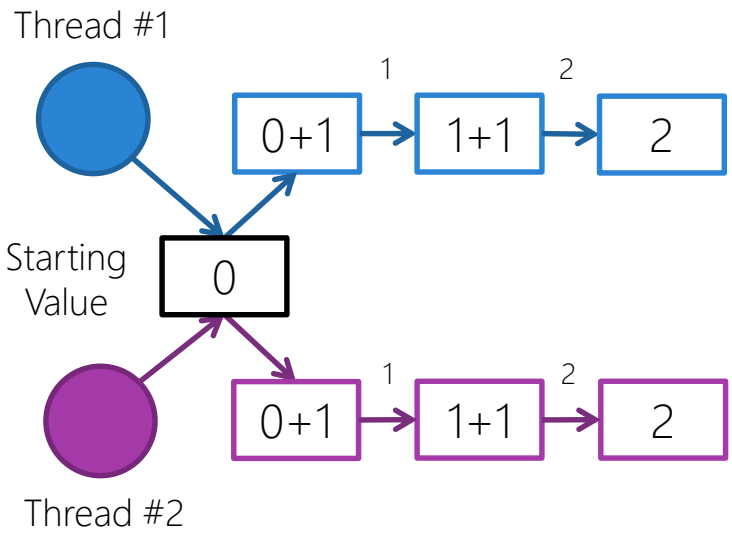
- Mobile applications must remain responsive – which requires multiple threads to execute our code
- We often create difficult bugs when we manipulate shared data simultaneously



Three increments occurred,
what should the counter's value be?

Parallelization in F#

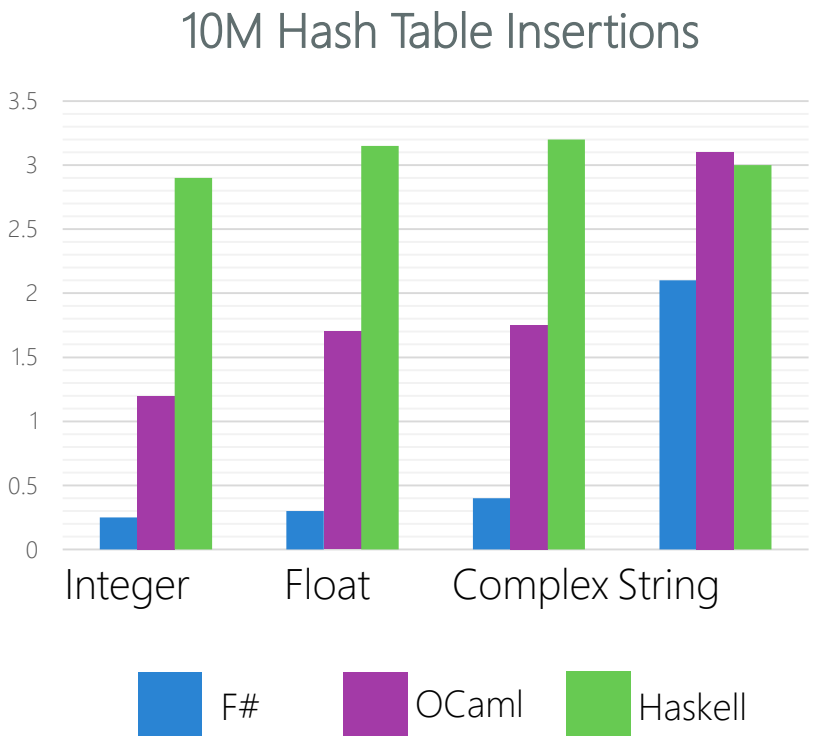
- F# values are **immutable by default** – they cannot be changed once they are assigned
- Unrelated functions executed in parallel that work on immutable data do not have to worry about ordering, or whether one function will change the result of the other function
- This **solves the most common bugs** we encounter in asynchronous programming



Both threads start with the same initial value, but the result is never stored – it is passed through from one increment into the next to arrive at the final value

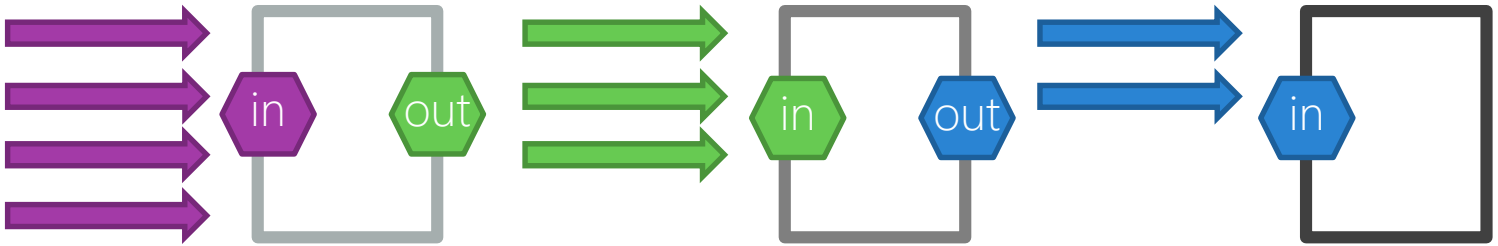
Performance

- Immutable data means the JIT compiler can produce better code that utilizes more caching
- Concurrency is simpler, you don't have to worry about using locks to protect shared data because the data is always read-only
- Parallelization is easier to take advantage of, that means it is more likely to be used in more situations



Pipelining

- Pipelining allows operations to be connected together where the output from one statement becomes the input into the next



Pipelining is a key performance feature because it can process the data *concurrently*

Predictability

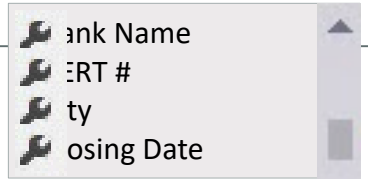
- Data tends to be localized, making it easier to verify the correctness of the code
- Side effects tend to be minimized because data is mostly immutable
- Language has several key features to ensure data is used correctly

Type Providers

- **Type Providers** provide strongly-typed data from external data sources which can reduce the amount of code as well as provide type safety

```
type BankData = Samples.Csv.CsvFile<"bankdata.csv",
   InferRows = 10, InferTypes = true, IgnoreErrors = true>

let results = new BankData()

for x in results.Data do
    x.
```

Bank Name	CERT #	City	Closing Date
Bank of America	12030584	Dallas	2014-09-08



properties listed come directly from the comma-delimited file's header line

Units of measure

- F# can define formal units of measure for signed numeric types

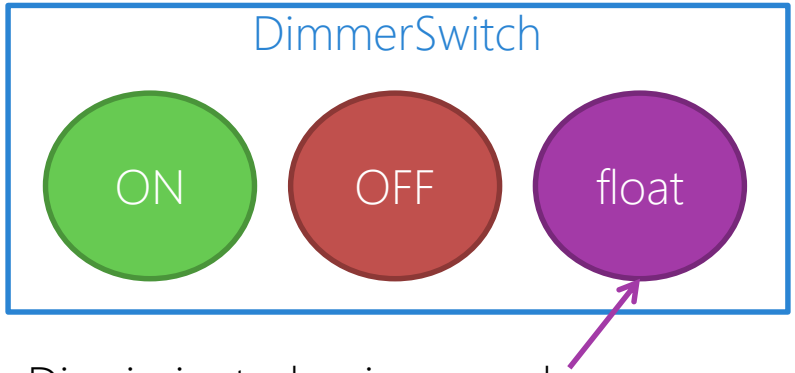
```
let speed = 55.0f<mile/hour>
let length = 12.0<cm>
let CmToInches (x : float<cm>) = ...
```

- Helps avoid bugs by ensuring the numeric type is exactly what is expected and providing conversions when possible

Discriminated unions

- Discriminated unions provide support for values that can be one of a set of named cases – similar to an `enum` in C#, but more powerful

Switch has both an on/off state, as well as a variable state.. how would you represent this in C#?



Discriminated unions can have zero or more pieces of named data associated with them *of any type*

"Big Picture" benefits to using F#

Solves problems without requiring programmer to specify exact procedures.

Declarative

F# is expressive, which allows you to implement algorithms directly, making the code easier to read.

Simple

Programming what to do as opposed to how to do it results in less code which is easier to read and debug.

Concise

Less code to read, write and debug leads to higher levels of productivity.

Productive

Flash Quiz

Xamarin University

Flash Quiz

① Immutability means that values cannot be changed once they are assigned.

a) True

b) False

Flash Quiz

① Immutability means that values cannot be changed once they are assigned.

a) True

b) False

Flash Quiz

② Functional programming is based on

a) Differential Calculus

b) Lambda Calculus

c) Fractional Calculus

Flash Quiz

② Functional programming is based on

a) Differential Calculus

b) Lambda Calculus

c) Fractional Calculus

Flash Quiz

③ Functional programming is a style of programming that models computations as the evaluation of expressions while avoiding ____

a) Bugs

b) Class

c) State

Flash Quiz

③ Functional programming is a style of programming that models computations as the evaluation of expressions while avoiding ____

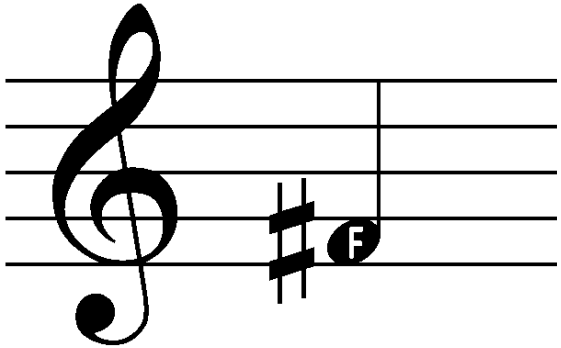
a) Bugs

b) Class

c) State

Summary

1. Outline the history of F#
2. Describe functional programming
3. Define and examine immutability
4. Identify advanced features of F#
5. Evaluate the benefits of using F#



Execute F# code in the REPL

Tasks

1. Identify the REPL
2. Use the REPL in our IDE
3. Create and display values in F#

```
F# Interactive for F# 3.1 (Open Source
Edition)Freely distributed under the
Apache 2.0 Open Source License
For help type #help;;

> let x = 42;;

val x : int = 42
```

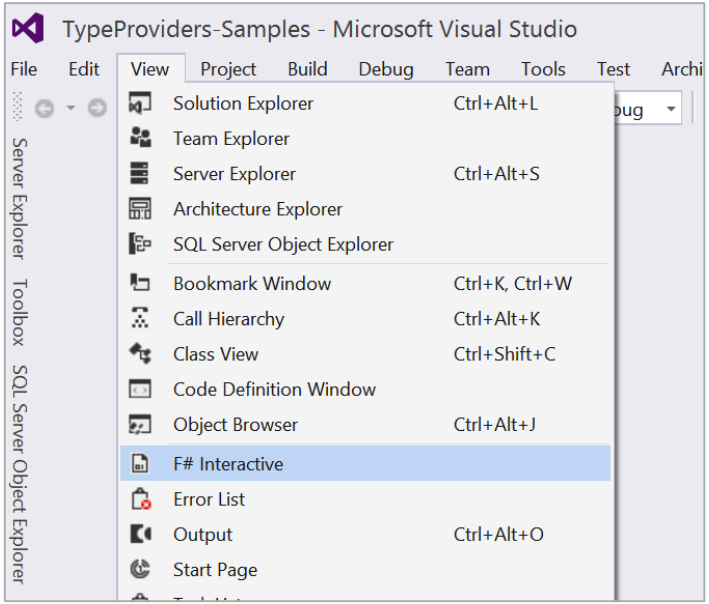
What is a REPL?

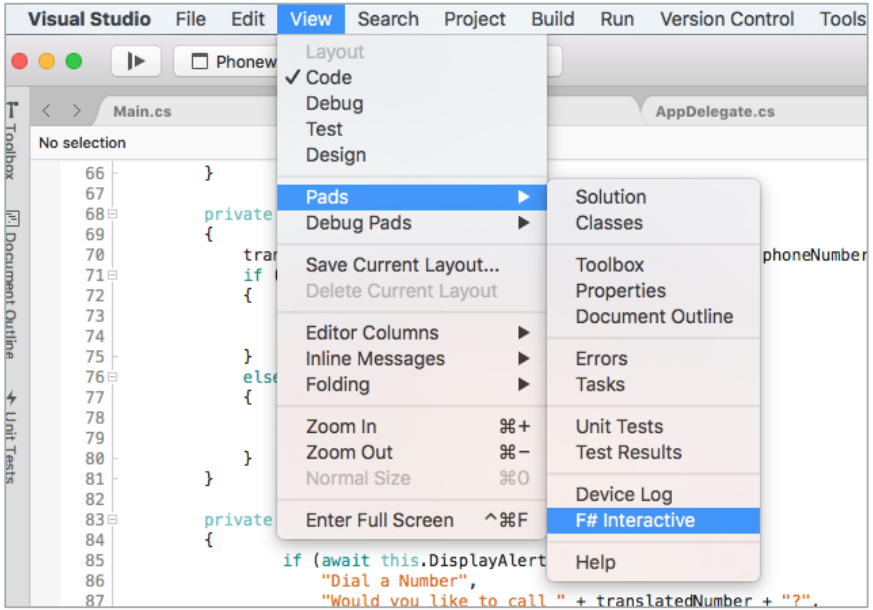
- Read-Evaluate-Print-Loop (REPL) is a language shell that provides a simple, interactive programming environment
- Allows developers to explore the language independent of a program

```
F# Interactive for F# 3.1 (Open Source
Edition)Freely distributed under the
Apache 2.0 Open Source License
For help type #help;;

> let x = 42;;

val x : int = 42
```

Using the REPL in our IDE

- Visual Studio includes an F# Interactive Console





Creating values

- Assign *values*, not variables, variables implies mutability (i.e. it's variable)
- Values cannot be changed once assigned (by default)
- Use the `let` keyword to assign named values

```
let x = 10

let y = 20.

let name = “Forest”
```

`let` is used to define values, functions and modules in F#

Return values

- F# always returns *something* for every evaluated expression
- Statements which have no return value return `unit = ()` which is a placeholder for "no value" – similar to `void` in C#

```
> let x = 42
val x : int = 42

> printfn "%i" x
42
val it : unit = ()
```

Display values to the console

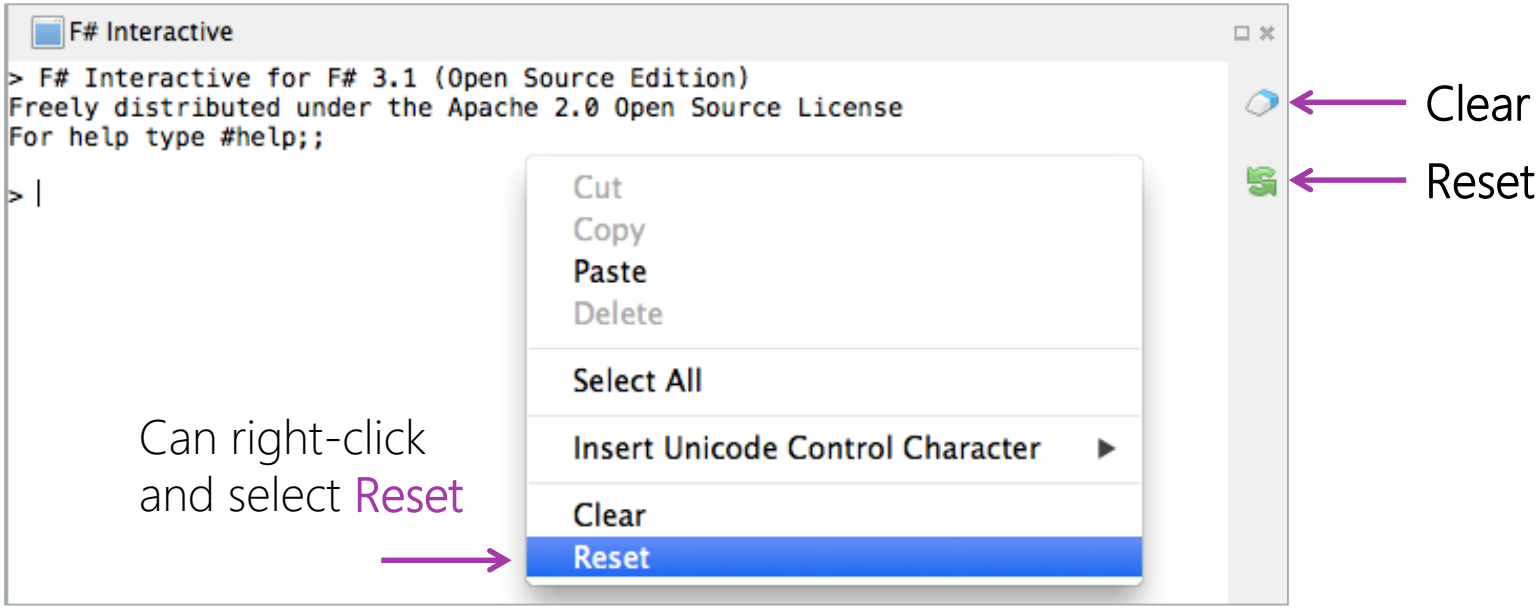
- Use the built-in **`printfn`** function to display a set of statically checked values and literals to the console or REPL

```
printfn "A string: %s, int: %i, float: %f, and bool: %b" "Helen"
42 3.14 true
```

`printfn` is very similar to the C-format style strings, but the values are checked at compile-time for type-safety, you should prefer this over other approaches like **`Console.WriteLine`**

Resetting the REPL environment

- Can *reset* the REPL to remove all existing values from memory, or *clear* the REPL to clear the screen (all values remain)



Individual Exercise

Discover the REPL

Xamarin University

Comments

- F# supports comments – descriptive statements which are ignored by the F# compiler, two forms are available: single-line and multi-line

```
// single line comments use double-slash like C#
let number = 5   // comments out remainder of line

(* multi-line comments use (* ... *) pair and can have
embedded comments *)
```

Language Rules – case sensitivity

- F# is a case-sensitive language

```
let name = "Helen"    // OK

Let name = "Helen"    // Nope, keywords are lowercase
```

```
// Three different values defined
let name = "Helen"
let Name = "Mark"
let NAME = "Rachel"
```

Language Rules - Terminators

- F# does not utilize a statement termination character

```
let number = 42      // no termination character used
```

No semicolons

Note: F# does not have statement terminators, *but* the REPL uses a double-semicolon (;;) to terminate input

Language Rules - parameters

- F# uses *spaces* not commas to separate parameters

use parenthesis to surround expressions to evaluate – for example to pass the result of the expression as a parameter, or to call a subsequent method on the result

```
let number = (new System.Random()).Next()
let name = "Helen"

printfn "Hi %s your lucky number is %i" name number
```

parameter #1

parameter #2

Language Rules – multiline statements

- Multiline statements use **spaces**, not braces or tabs to denote a block

```
let evens nums =
  let isEven x = x%2 = 0
  List.filter isEven nums
```

Define a new function named **evens** which takes a list and returns the even values – notice we have no braces, but the statements for the function are indented under the **let** definition with one or more spaces

```
List<int> evens(List<int> nums)          C#
{
  Func<int,bool> isEven = x => x%2 == 0;
  return nums.Where(isEven).ToList();
}
```

Aligning indentations

- F# will give errors when columns are not aligned properly

Column is misaligned

```
let x =
  let a = 1
  let b = 1
   a+b
```

```
let x =
  let a = 1
  let b = 1
  a+b
```

Available types

- F# supports the .NET Common Type System (CTS), which means you have all the same basic types you use today in C#, plus a few extra

C#

```
string name = "Molly";

int favoriteNumber = 3;
```

- F# does not explicitly declare the type – it is always *inferred* from the initialization, which means it must always be initialized

F#

```
let name = "Molly"

let favoriteNumber = 3
```

Type Inference

- F# infers the type of values automatically from the surrounding information

```
let y = 7
val y : int
```

F# infers that the value is an integer based on the assignment

```
let y = 7
let square x = x*x;;
val y : int
val square : x:int -> int
```

F# assumes you will pass an integer into the function automatically

Type Inference

- F# infers the type of values automatically from the surrounding information

```
let y = 2.
let square x = x*x
square y

val y : float = 2.0
val square : x:float -> float
val it : float = 4.0
```

Now, F# sees you will pass a float into the function and changes the type automatically

Type annotations

- Sometimes type inference can't fully determine the type

```
type Person (name) =
    member this.SayHi = printfn "Hi %s" name

module Test =
    let sayHellos list =
        List.iter (fun (x)  -> x.SayHi) list
    sayHellos [new Person("you"); new Person("me");]
```

Error FS0072: Lookup on object of indeterminate type based on information prior to this program point. A type annotation may be needed prior to this program point to constrain the type of the object. This may allow the lookup to be resolved. (FS0072)

Type annotations

- This can be fixed by annotating or constraining the type for the compiler

```
type Person (name) =
    member this.SayHi = printfn "Hi %s" name

module Test =
    let sayHellos list =
        List.iter (fun (x : Person)  -> x.SayHi) list
    sayHellos [new Person("you"); new Person("me");]
```

Type Comparison (C# > F#)

- Numeric types depend on the suffix of the number to infer the type

C# keyword	F# Assignment
int	32
double	32.
float	32.f
BigInt	32.i
long	4l
ulong	4ul

C# keyword	F# Assignment
short	4s
ushort	4us
sbyte	4y
byte	4uy
string	"text value"

Math Operators

- Math operators (and precedence) is similar to C#

Op	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Integer modulo
**	Exponent

Op	Purpose
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
=	Equal (comparison)
<>	Not equal

Op	Purpose
||	Boolean OR
&&	Boolean AND
&&&	Bitwise AND
|||	Bitwise OR
^^^	Bitwise XOR
~~	Bitwise NOT

Working with mutable data

- F# can support mutable data when the `mutable` keyword is applied
- Mutability is an *explicit decision* and should be *avoided if possible*, but might be required in special cases, such as interacting with C# code

```
let mutable x = 5

x <- 10
```

mutable values can be changed using the assignment operator

Individual Exercise

Working with immutable and mutable values in the F# REPL

Xamarin University

Flash Quiz

Xamarin University

Flash Quiz

① Use the keyword _____ to specify a value that can be changed.

a) alter

b) let

c) mutable

Flash Quiz

① Use the keyword _____ to specify a value that can be changed.

a) alter

b) let

c) mutable

Flash Quiz

② F# types are not_____, they are _____.

a) declared, inferred

b) inferred, declared

Flash Quiz

② F# types are not_____, they are _____.

a) declared, inferred

b) inferred, declared

Flash Quiz

③ `printfn` can be used to display _____.

a) a single value and a string

b) any number of formatted values and strings

c) a Console.WriteLine type of formatted string

Flash Quiz

③ `printfn` can be used to display _____.

a) a single value and a string

b) <u>any number of formatted values and strings</u>

c) a Console.WriteLine type of formatted string

Summary

1. What is a REPL?
2. Using the REPL in our IDE
3. Create and display values in F#

```
F# Interactive for F# 3.1 (Open Source
Edition)Freely distributed under the
Apache 2.0 Open Source License
For help type #help;;

> let x = 42;;

val x : int = 42
```

Working with expressions and loops

Tasks

1. Explore basic F# syntax
2. Illustrate expressions
3. Identify loops in F#

```
F# Interactive for F# 3.1 (Open Source
Edition)Freely distributed under the
Apache 2.0 Open Source License
For help type #help;;

> let x = 42;;

val x : int = 42
```

Expressions in F#

- Nearly everything in F# provides some result value, which makes almost everything an expression

```
type Product (name, price, onSale) =
    let isFree = price = 0.0
    member this.Name = name
    member this.IsFree = isFree
    member this.SalePrice = if onSale && price <> 0.
                            then price/2.
                            else price
```

The **if** statement is actually an expression that returns a value – which is being assigned to a property

Conditional expressions

- **`if-then-else`** is an expression, *not* a statement and all expressions return a value

```
if expr then statement else statement
```

```
let greeting = if gender = "m" then "Mr." else "Ms."
```

```
let evaluate myArray =
    if Array.isEmpty myArray then
        printfn "Oh no, empty!"
    elif Array.length myArray > 10 then
        printfn "Array too long!"
```

Chaining functions together

- When you create F# programs, you are often combining functions and expressions together to generate a final result, the pipe operator makes this very easy to do without creating temporary intermediate values

```
let randNums count =
    let rng = new System.Random()
    List.init count (fun _ -> rng.NextDouble() * Math.PI * 2)

randNums 200
|> List.average                    // Average all the numbers
|> System.Math.Sin                 // Get the Sin(avg)
|> printfn "Average sin: %f"       // Output the value
```

Three kinds of loops

- F# has three explicit loop styles which parallel C# loops

for-in-do (foreach)	for-to-do (for)	while-do (while)

Note: F# has a set of **list** and **sequence** functions which can be used in place of explicit loops, you will see these functions and how to use them in a future module

for-in-do

- **`for-in-do`** is the same as **`foreach`** in C# and is the most commonly used loop in F#

```
for val in enumerable do something
```

```
for i in [1..10] do
  printfn "%d" i
```

```
1
2
3
...
9
10
```

loop executes over set of numbers and outputs each to the console and returns **`unit = ()`**

Unit constraint on loops

- All loop expressions must return **unit**, there is no way to return a value from inside a loop, this often causes a warning to be produced when the final expression in the loop block returns a value

```
let n =
  for f in [10..100] do
    f + f
```

returns an integer value

warning FS0020: This expression should have type 'unit', but has type 'int'. Use 'ignore' to discard the result of the expression, or 'let' to bind the result to a name.

Unit constraint on all loops

- To fix this you need to add `|> ignore` at the end of the expression

```
let n =
  for f in [10..100] do
    f + f |> ignore
```

`|> ignore` tells the compiler to ignore the return value and gets rid of the unit warning

for-to-do

- **`for-to-do`** has the same functionality as a **`for`** statement in C#

```
for val = start to finish do something
```

creates function to print out 1 to 10 with a blank line at the end

```
let counter() =
    for f = 1 to 10 do
        printf "%i " f
    printfn ""
...
counter()
```

```
1 2 3 4 5 6 7 8 9 10
```

while-do

- **while-do** is similar to the **while** loop in C#

```
while condition do something
```

```
let nums = [|1.0..10.0|]   // double[]

let mutable i = 0
while i < nums.Length do
    nums.[i] <- nums.[i] ** nums.[i]  // pow
    i <- i + 1

for v in nums do
    printfn "%f" v
```

Missing features

- No **`do-while`** style loop available
- No ability to **`break`** or **`continue`** – prefer sequences or lists if you need this behavior
- **`for-to-do`** loops only support integers and can only increment by one, can use **`for-in-do`** instead

Flash Quiz

Xamarin University

Flash Quiz

① The for-in-do loop is similar to the _____ C# loop

a) for
b) foreach
c) do-while
d) None of the above

Flash Quiz

① The for-in-do loop is similar to the _____ C# loop

a) for

b) foreach

c) do-while

d) None of the above

Flash Quiz

② Conditional statements and loops return values in F#, True or False?

a) True

b) False

Flash Quiz

② Conditional statements and loops return values in F#, True or False?

a) True

b) False

Flash Quiz

③ To ignore a return value from an expression you would use:

a) <| Ignore

b) < ignore

c) > ignore

d) |> ignore

Flash Quiz

③ To ignore a return value from an expression you would use:

a) <| Ignore

b) < ignore

c) > ignore

d) |> ignore

Summary

1. Explore basic F# syntax
2. Illustrate expressions
3. Identify loops in F#

```
F# Interactive for F# 3.1 (Open Source
Edition)Freely distributed under the
Apache 2.0 Open Source License
For help type #help;;

> let x = 42;;

val x : int = 42
```

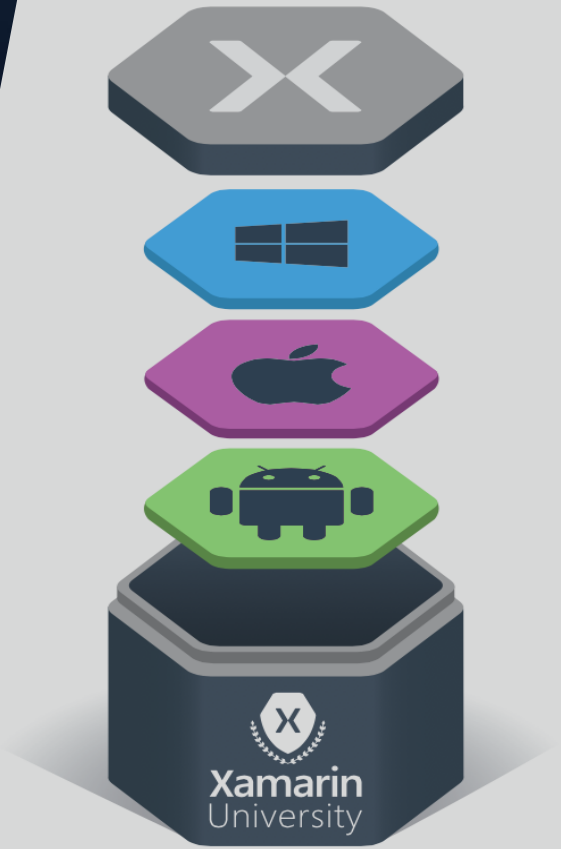
Where are we going from here?

- You now have some basic knowledge of the history and usage of the F# programming language

- In the next course, we will look at how to manage solutions and projects in F# which has some surprising differences from C#!

Thank You!

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



Microsoft