FSC103

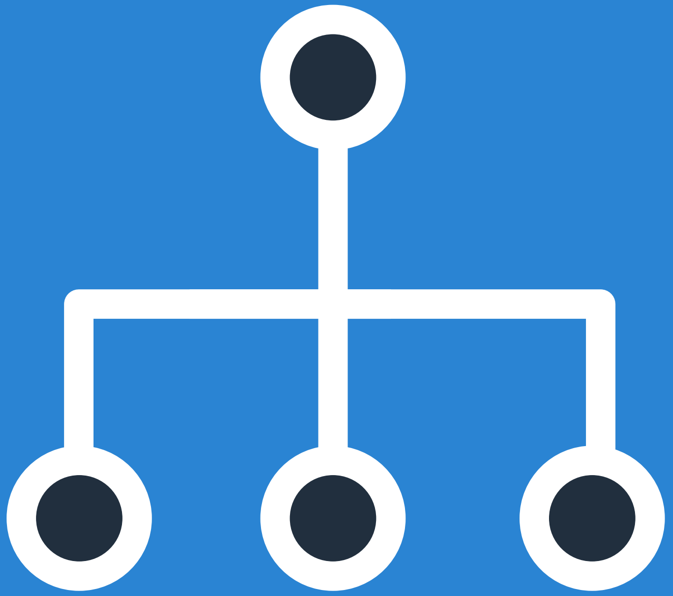
Data Structures

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

Xamarin University

Objectives

1. Utilize arrays and lists
2. Organize and transform data using data structures

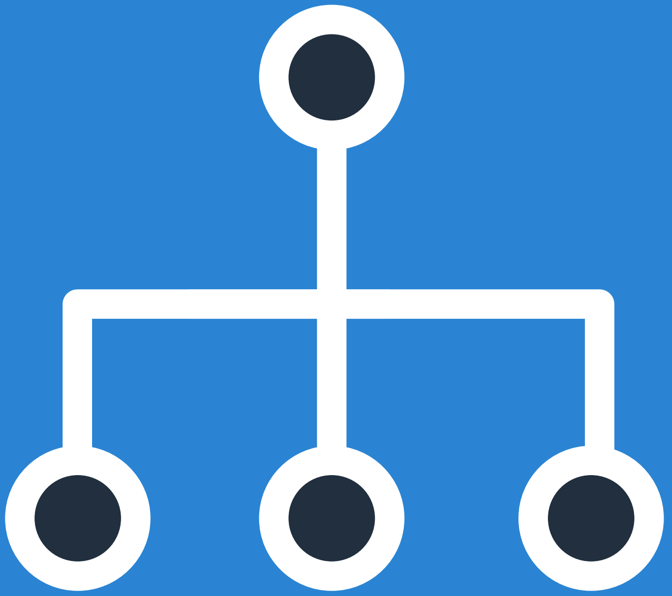


Utilize arrays and lists

Xamarin University

Tasks

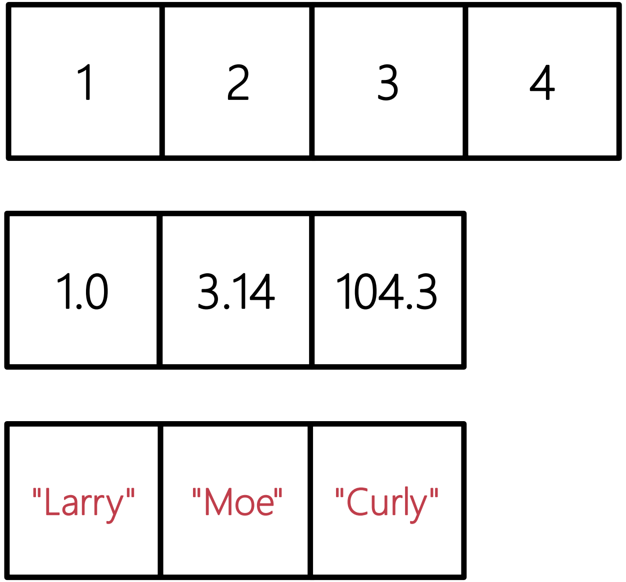
1. Store data in an array
2. Store data in a list
3. Compare list and array



What is an Array?

- Arrays are fixed size sequences of **homogenous** data

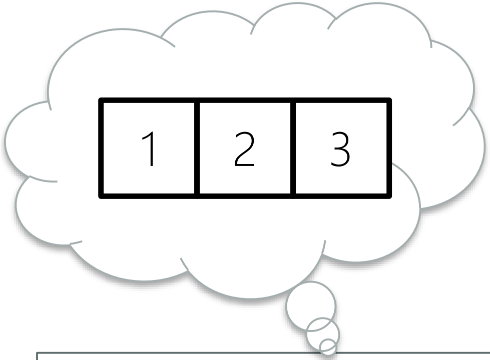
- F# uses `System.Array` under the covers so any created arrays can be passed into other .NET code



Creating Arrays

- Arrays are defined by providing data in between `[|` and `|]`

- Must be initialized with data – this determines the type and size of the array

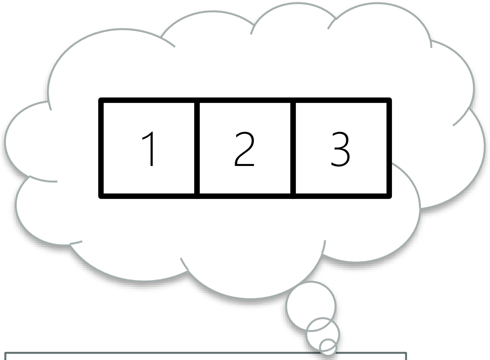



```
let array1 = [|1; 2; 3|]
```

initializing values are delimited by semicolons

Creating Arrays

- Arrays are defined by providing data in between [| and |]
- Must be initialized with data – this determines the type and size of the array



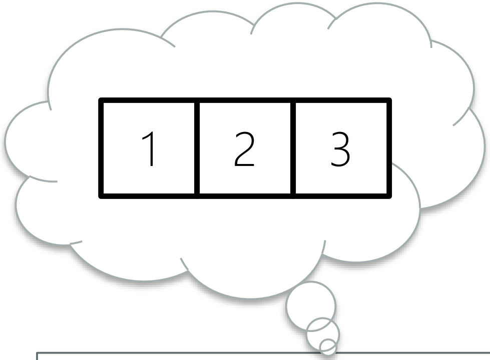

```
let array1 =
  [|
     1
     2
     3
  |]
```

… or can put elements on a separate line, then semicolon is *optional*

Creating Arrays

- Arrays are defined by providing data in between `[|` and `|]`

- Must be initialized with data – this determines the type and size of the array




```
let array1 = [| 1..3 |]
```

can also specify a *sequence* where F# provides the intermediate values, this is *very powerful*

Other ways to create arrays

- F# also include several functions that are included in the **`Array`** module to create and initialize array types

```
let array1 = Array.empty<int>     // int[] with no elements

let array2 = Array.create 3 1.0  // [| 1.0; 1.0; 1.0; |]

let array3 = Array.zeroCreate 3  // [| 0; 0; 0 |]

let array3 = Array.init 3 (fun n -> n.ToString())
// [| "0"; "1"; "2" |] (function is passed index)
```

Accessing Arrays

- Array indices start at zero and elements can be accessed using the dot (.) operator combined with brackets ([and])

```
let evens = [| 0..2..10 |]    // = [| 0; 2; 4; 6; 8; 10 |]

evens.[1]                     // = 2
evens.[0..2]                  // = [| 0; 2; 4 |]
evens.[..4]                   // = [| 0; 2; 4; 6; 8 |]
evens.[3..]                   // = [| 6; 8; 10 |]
```

always returns a **new value** – either a single value, or a new array with the requested subset of data

Modifying arrays

- The array size and shape is fixed, but the data inside can be changed

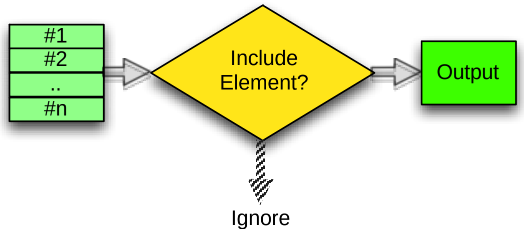
```
let array1 = [| 1; 2; 3; |]

let n = array1.[1]

array1.[1] <- n + 1    // [| 1; 3; 3; |]
```

we use the **assignment operator** to assign a value into the array after it has been created

Specialized Array Operations

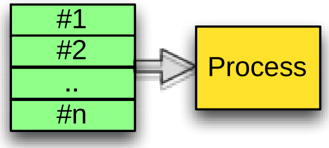
❖ `Collections.Array` module provides array operations to:

Filter



Transform

Process



Filtering elements

❖ Filter operation is used to return a new array that contains elements which are matched with the passed predicate filter function

```
let nums = [| 0..12 |]    // 0-12 inclusive

Array.filter (fun n -> n%2 = 0) nums
```

```
[| 0; 2; 4; 6; 8; 10; 12 |]
```

source array is passed as 2nd parameter

lambda filter is passed as 1st parameter and returns true / false for each item

Transforming elements

- Map operation is used to return a new array with the elements transformed by the passed transformation function

```
let nums = [| 0..5 |] // 0-5 inclusive

Array.map (fun n -> n*2) nums
```

```
[| 0; 2; 4; 6; 8; 10 |]
```

Processing arrays

- `Array.Iter` operation can be used to process each element in the array with a passed processing function

```
let nums = [| 0..5 |] // 0-5 inclusive

Array.iter (fun n -> printfn "%i" n) nums
```

```
0
1
2
3
4
5
```

`Array.iter` is similar to the C# `foreach` statement – it processes each item in the array individually but returns no direct result

Other Array operations

- Array also includes operations to perform common mathematical calculations, prefer these methods over manual iteration + calculations

```
let x = Array.average nums
```

returns the average of the #s in the array

```
let x = Array.max nums
```

returns the max # in the array

```
let x = Array.min nums
```

returns the min # in the array

```
let sorted = Array.sort nums
```

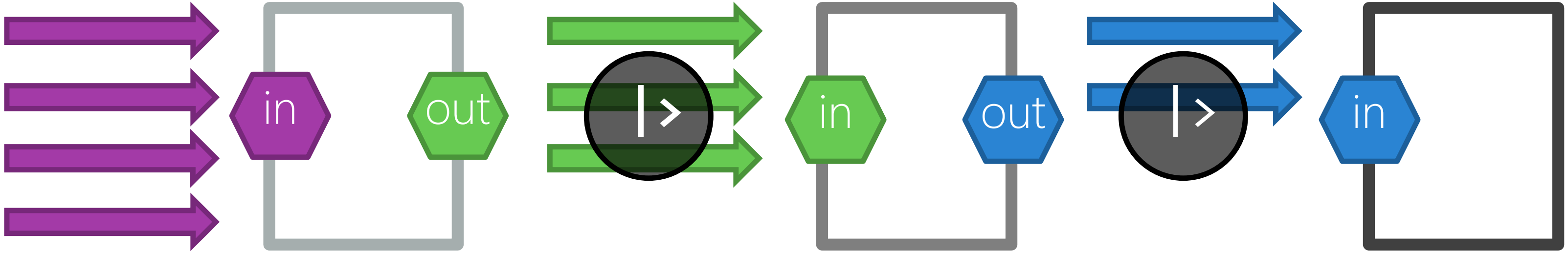
returns the array in sorted order

```
let total = Array.sum nums
```

returns the sum of the #s in the array

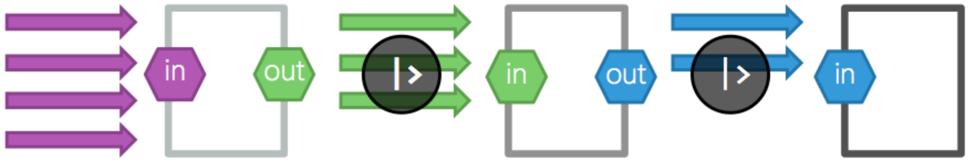
Pipelining

❖ Common to use pipelining (|>) + operations to process arrays



Pipelining

❖ Common to use pipelining (|>) + operations to chain functions




```
let nums = [| 0..4 |]
nums
   |> Array.filter (fun n -> n%2 = 0)
   |> Array.sum
```

first get all the even #s

sum the result from filter

pipeline operator

```
val it : int = 6
```

Parallel Arrays

- Can parallelize some operations by adding `.parallel` before the operation function call

```
open System.IO    // include System.IO namespace

let files = Directory.GetFiles(@"C:\", "*.txt")

Array.parallel.iter (fun fn -> File.Encrypt(fn)) files
```

Take all the text files on the C: drive (Windows) and encrypt each file using Windows file system encryption – this is done in parallel so we are encrypting several files simultaneously on multi-core machines

Beware: you must be performing a fair amount of CPU or I/O bound work to make this effective, otherwise the operation will be slower than if it were done serially

Lists

- An F# *list* is a singly-linked-list data structures storing homogenous data

```
let evens= [0..2..10]
```

no parallel bars – creates a list

```
Array.toList [|0..2..10|]
```

can also turn array into list
(or vice-versa)

Adding elements to a list

- You cannot update elements of a list, but you can create a new list by adding elements to the front, this is done in O(1) time as it is not necessary to copy the old list

```
let list1 = [1;2;3;4]          // 1-4 in the list

let list2 = 0::list1           // [0;1;2;3;4]
```

this is referred to as the cons operator and it prepends elements to the list. There is no operator to *append* to the list, because it wouldn't be efficient for a singly-linked list

Combining lists

- Can also combine lists to create a new list which contains both – this is also an efficient operation because the list data is not duplicated

```
let list1 = [ 1;2;3;4 ]
let list2 = [ 5;6;7;8 ]

let list3 = list1 @ list2     // [ 1;2;3;4;5;6;7;8 ]
```

concatenation operator which combines two lists together to generate a third unique list

List Operations

- List has many of the same operations as array, but also has some unique functions for working with the list
  - head
  - tail
  - recursion
  - pattern matching

```
let list = [1;2;3;4]

// Define recursive function
// named "sum" to sum all the
// elements in a list
// parameter "values"
let rec sum values =
   match values with
   | [] -> 0
   | head::tail -> head +
                   sum(tail)

let total = sum list   // 10
```

Should I use lists or arrays?

- F# programmers tend to prefer list over array for many cases, but the parallelization support in array can be very helpful for CPU intensive calculations

Use Lists for:

- Variable size
- Supports recursion
- Head/Tail pattern matching

Use Arrays for:

- Parallelization
- Mutable elements
- C# interop

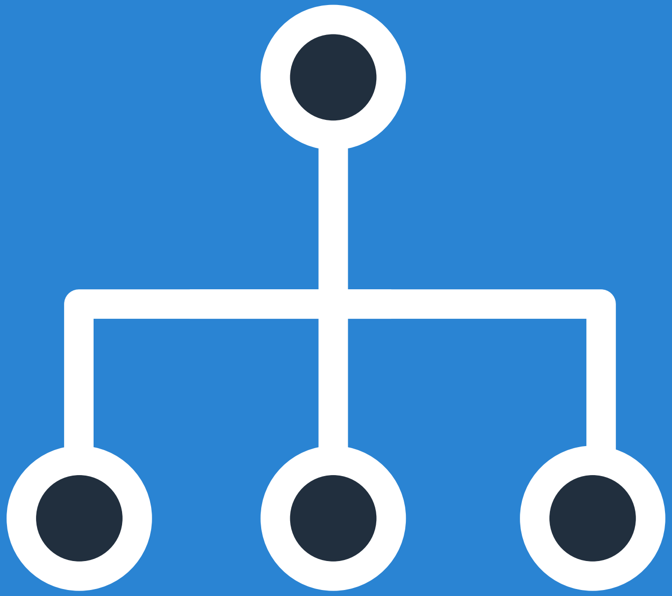
Individual Exercise

Working with arrays and lists in the REPL

Xamarin University

Summary

1. Store data in an array
2. Store data in a list
3. Compare list and array

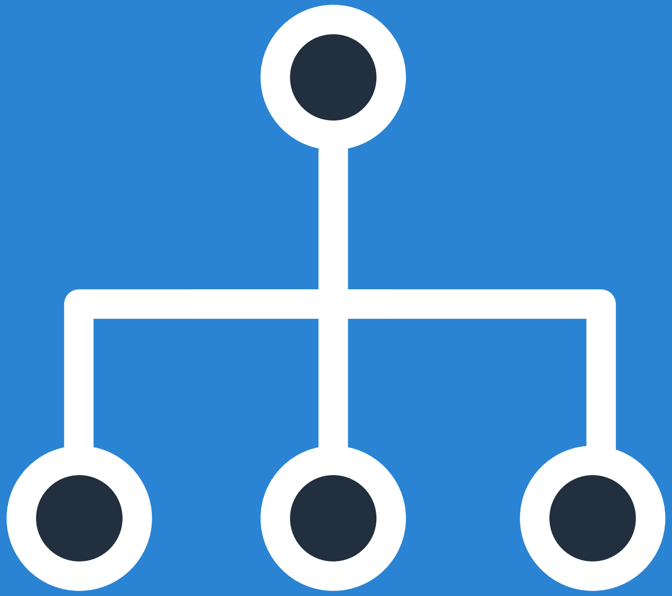


Organize and transform data using data structures

Xamarin University

Tasks

1. Discuss tuples
2. Describe sequences and their uses
3. Create and utilize records
4. Compose discriminated unions



Tuples

- A *tuple* is a grouping of unnamed but ordered values, possibly of different types

```
// Tuple of two integers: ( 1, 2 )

// Tuple of strings: ( "Rachel", "Helen", "Mark" )

// Tuple that has mixed types: ( "BillG", 2014, 20240332. )

// Tuple of integer expressions: ( a + 1, b + 1)
```

Under the covers, this construct creates a .NET **`Tuple<T1,T2,T3,…>`** type

Passing tuples as parameters

- Tuples are commonly used as parameters to functions

```
let average (a, b) =
    (a + b) / 2.0
...
average (10., 20.)
```

```
val it : float = 15.0
```

Working with tuples

- F# will allow you to choose the first or second item in a pair tuple using the `fst` and/or `snd` keywords

```
// Tuple of two integers: ( 1, 2 )
```

```
fst ( 1, 2 )
```

```
( 1, 2 ) |> fst
```

```
val it : int = 1
```

```
snd ( 1, 2 )
```

```
( 1, 2 ) |> snd
```

```
val it : int = 2
```

Getting values from tuples

- Common to use pattern matching to assign names to tuple elements

```
let (a, b) = (1, 2)
printfn "%d : %d" a b
```

```
1 : 2
```

Use underscore as *wildcard* match

```
let (_, _, c) = (1, 2, 3)
```

```
val c : int = 3
```

Pattern matching tuples

- Can also use formal match expression to process values

```
let greeting (name, language) =
    match (name, language)  with
    | ("Yoda", _) -> "Greetings Master"
    | (name, "English") -> "Hello " + name
    | (_, "French") -> "Bonjour!"
    | (name, _) when language.StartsWith("Span") -> "Hola, " + name
    | (name, "Klingon") -> "nuqneH" + name
    | _ -> "Error!"
```

Type signature

- When you display a tuple, the * symbol is used to separate the components in the type signature

string   int[]

```
let data = (0, "pumpkin", 18., [0;1;2;3]);;
```

int   float

```
val data : int * string * float * int list = (0, "pumpkin",
18.0, [0; 1; 2; 3])
```

Xamarin University

What are sequences?

- Sequences are a logical series of elements of one type that may be iterated in a forward-only, read-only fashion
- Can be created explicitly through the `seq` keyword, or implicitly by an `IEnumerable<T>`

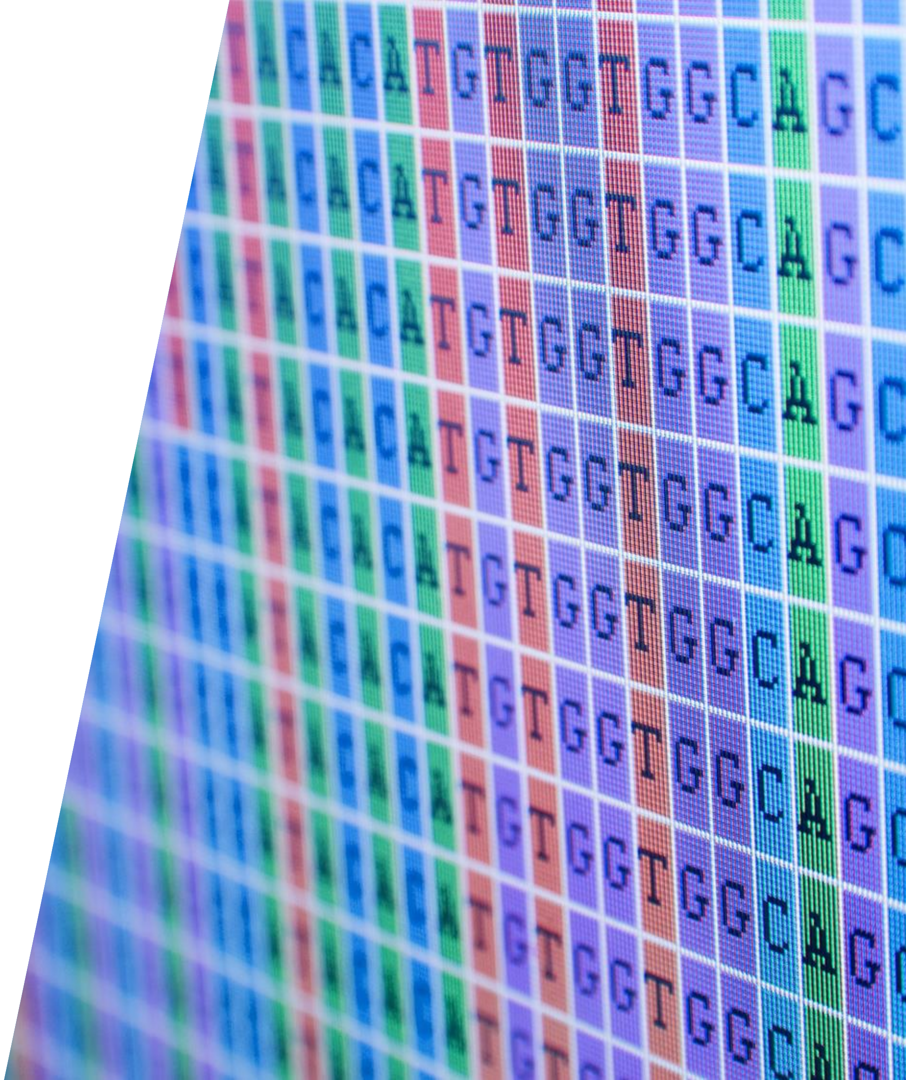
```
seq {0..10..100}
```

Creates a sequence of multiples of tens from 0 to 100

Why use sequences?

❖ Sequences are useful when you have **large amounts of data**, but only want to use certain parts of it at one time, for example in genomics or never-ending series

❖ Sequences are *lazily evaluated* so they only generate the values as the client requests them



Creating Sequences

- Sequences can be generated from functions

```
let phi = (1. + sqrt 5.) / 2.
let fibonacci = Seq.initInfinite (fun index ->
    let num = float index
    (((phi ** num)) - ((-phi) ** -num)) / sqrt 5. |> int64)
printfn "%A" fibonacci
```

```
seq [0L; 1L; 1L; 2L; 3L;...]
```

`Seq.initInfinite` creates a sequence from a supplied function using the integer index of the item to return, the **`fibonacci`** function here will generate as many numbers as requested (up to **`Int32.MaxValue`**)

Creating Sequences

❖ Sequences can be generated from functions

```
let ByTwos current =
   current |> Seq.unfold (fun num -> Some(num, num+2))

printfn "%A" <| ByTwos 1
```

```
seq [1; 3; 5; 7; ...]
```

`Seq.unfold` generates each value based on the prior value and is passed an initial value to start the generation

Creating Sequences

- Sequences can also be generated from **`IEnumerable`**

```
Brie, France
Cambozola, Germany
Cheddar, England
Fontina, Denmark
Gorgonzola, Italy
Havarti, Denmark
Limburger, Germany
Parmesan, Italy
```

```fsharp
let cheeses = File.ReadLines("cheeses")
    |> Seq.cast<string>
```

Read all the lines in with a **`StreamReader`** and create a new sequence in memory

Note: **`Seq.cast`** is not strictly necessary in this case since strings are being returned already and **`IEnumerable`** is automatically turned into sequences in F#

Explicit sequences

- Can also generate sequences using the **yield** keyword, and combine sequences with the **yield!** keyword

```
let names = seq {
    yield "Rachel";
    yield "Helen";
    yield "Mark";
    yield! [ "Adrian", "Glenn", "René" ] // Subsequence
    ...
}
```

```
val it : seq<string> = seq ["Rachel"; "Helen"; "Mark"; "Adrian";
                            "Glenn", "René"]
```

Sequence Expressions

- Expressions can also generate sequences, most common form is to use the **do-yield** keywords

```
let squares = seq { for i in 1 .. 10 do yield (i, i*i) }

Seq.iter (fun (n,n2) -> printfn "%d squared is %d" n n2)
    <| squares
```

`Collections.Seq` contains a variety of useful functions including support for filtering, combining and iterating sequences

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
...
```

Sequence Operations

❖ Sequences have many of the same capabilities as lists and arrays

```
let x = Seq.average nums
```

```
val nums : seq<float>
val x : float = 10.5
```

```
let x = Seq.min nums
```

```
val nums : seq<float>
val x : float = 1.0
```

```
let sort = Seq.sort nums
```

```
val nums : seq<float>
val sort : seq<float>
```

```
let total = Seq.sum nums
```

```
val nums : seq<float>
val total : float = 210.0
```

Individual Exercise

Create a sequence of triangular numbers

What are records?

- Records are simple aggregates of named heterogeneous values

```
type Point = { x : float; y : float; }
```

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

semicolons are used to separate the values defined as part of the record, they are *optional* if the values are listed on separate lines

Records actually generate a .NET class with public properties, the big advantage to records is when you start *using* them

Creating records

- Records are *inferred* when you create a new variable – remember F# always knows all the types you have defined up to that point

```
let helen = { Id = 1; Name = "Helen"; Email = "..."; }
```

There is no mention of the **`Person`** record type – instead, F# figures out what you want based on the fields being assigned here

Creating records

- When there are record conflicts, the instance can declare the record type as part of the field definitions

```
type Point = { x : float; y : float; }
type Point3D = { x : float; y : float; z : float; }
```

```
let pt = { x = 10.; y = 20.; }
```

Ambiguous – which record should be used?

Creating records

- When there are record conflicts, the instance can declare the record type as part of the field definitions

```
type Point = { x : float; y : float; }
type Point3D = { x : float; y : float; z : float; }
```

```
let pt = { Point.x = 10.; y = 20.; }
```

Can define the record as part of one or more of the fields to indicate which one to use

Comparing records

- F# records are **compared by value**, so two records with the same values are considered equal

```
type Point = { x : float; y : float; }

let pt1 = { x = 100.; y = 200.; }
let pt2 = { x = 100.; y = 200.; }

printfn "%s" |< if pt1 = pt2 then "Equal" else "Not Equal"
```

```
Equal
```

This works for other .NET languages as well, F# implements both `IComparable` and `IEquatable` as well as overriding the `Equals` and `GetHashCode` methods

Making records mutable

- Records, like all data structures, are immutable by default, but you can declare fields to be explicitly changeable using the **mutable** keyword

```
type recipe = { mutable A: string; mutable B: string;
                mutable C: string }

let pieRecipe = { A = "flour"; B = "water"; C = "salt"; }

pieRecipe.C <- "sugar"      // Change the value
printfn "%s" pieRecipe.C
```

```
sugar
```

Copying records

- Copying a record duplicates all of the fields in the source record with the requested changes

```
let mark = { Id = 1; Name = "Mark"; ... }
...
let jen = { mark with Name = "Jenny" }
```

mark	
Name	"Mark"
Id	1
LastUpdated	2014-12-01

jen	
Name	"Jenny"
Id	1
LastUpdated	2014-12-01

Individual Exercise

Create a record type using tuples

Xamarin University

What is a discriminated union?

- A discriminated union (DU) is a type which includes a closed set of known values – similar to an `enum` in C# or a `union` in C/C++

- Unlike `enum`s, DUs will **always** be one of the specified values and cannot be used as bit flags

```
type Fruit =
        | Apple
        | Pear
        | Raspberry
        | Kiwi
        | Banana
        | Grape
        | Blueberry
        | Tangerine
```

Using a discriminated union

- Can assign a DU directly to a variable, F# infers the type being created

```
type Fruit =
     | Apple
     | Pear
     | Raspberry
     | Kiwi
     | Banana
     | Grape
     | Blueberry
     | Tangerine
```

Must select one of the known values – cannot use casts or assign unknown values

```
let forbiddenFruit = Apple
...
printfn "%A" forbiddenFruit
```

```
Apple
```

Under the covers, F# creates a static property for each known `Fruit` value on the discriminated union type

Xamarin University

Discriminated Union field types

- Discriminated Unions can have a field type associated with each value

```
type Shape =
     | Circle of radius : float
     | Square of size : uint32
     | Rectangle of length : (double * double)
```

each supported value can have a different associated *type*, here for example we use a **tuple** for the rectangle width/height value

Xamarin University

Discriminated Union field types

- Discriminated Unions can have a field type associated with each value

```
type Shape =
     | Circle of radius : float
     | Square of size : uint32
     | Rectangle of length : (double * double
```

```
let shape1 = Circle radius = 50.
let shape2 = Rectangle length = (50.,25.)
```

Based on the selected field, the value is initialized with the appropriate type

assigned variables select *one* of the defined values to use – this approach is used instead of inheritance (e.g. a **Shape** base class)

Discriminated Union field types

- Discriminated Unions can have a field type associated with each value

```
type Shape =
     | Circle of float
     | Square of uint32
     | Rectangle of (double * double)
```

```
let shape1 = Circle 50.
let shape2 = Rectangle (50.,25.)
```

Can also leave off the field name in the definition and assignment

Under the covers, F# actually generates an abstract **Shape** class with concrete versions for each value you create – but that's all hidden away with syntactic sugar!

Discriminated Union field types

- Can associate *multiple* values for a given identifier using the asterisk (*) as a separator

```
identifier [of [fieldname1 :] type1 [* [fieldname2 :] type2 ...]
```

```
type Shape =
     | Circle of radius : float
     | Square of size : uint32
     | Rectangle of length : double * double
     | Prism of width : double * height : double
```

```
let shape1 = Prism (width = 10., height = 5.)
```

Individual Exercise

Convert C# code into a discriminated union

Flash Quiz

Flash Quiz

① What symbol do you use to separate components in a tuple?

a) |>

b) _

c) *

Flash Quiz

① What symbol do you use to separate components in a tuple?

a) |>

b) _

c) *

Flash Quiz

② Records __________ (select all that apply)

a) Can have multiple constructors
b) Cannot be compared
c) Can contain different types of data
d) Are immutable

Flash Quiz

② Records __________ (select all that apply)

a) Can have multiple constructors

b) Cannot be compared

c) Can contain different types of data

d) Are immutable

Flash Quiz

③ Which of these two examples employ the proper syntax of a record expression?

a) A is correct
b) A and B are both correct
c) B is correct

A

```
let me = { Id = 1234; Name="Mark"; }
```

B

```
let me = {
      Id = 1234
      Name="Mark"
}
```

Flash Quiz

③ Which of these two examples employ the proper syntax of a record expression?

a) A is correct

b) A and B are both correct

c) B is correct

A

```
let me = { Id = 1234; Name="Mark"; }
```

B

```
let me = {
      Id = 1234
      Name="Mark"
}
```

Summary

1. Discuss tuples
2. Describe sequences and their uses
3. Create and utilize records
4. Compose discriminated unions

Where are we going from here?

- You now know about some of the common data structures you use in F#
- In the next course, we will look at how to match patterns in F# which can replace the common **`if-else`** statement

Thank You!

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