FSC102

Projects in F#

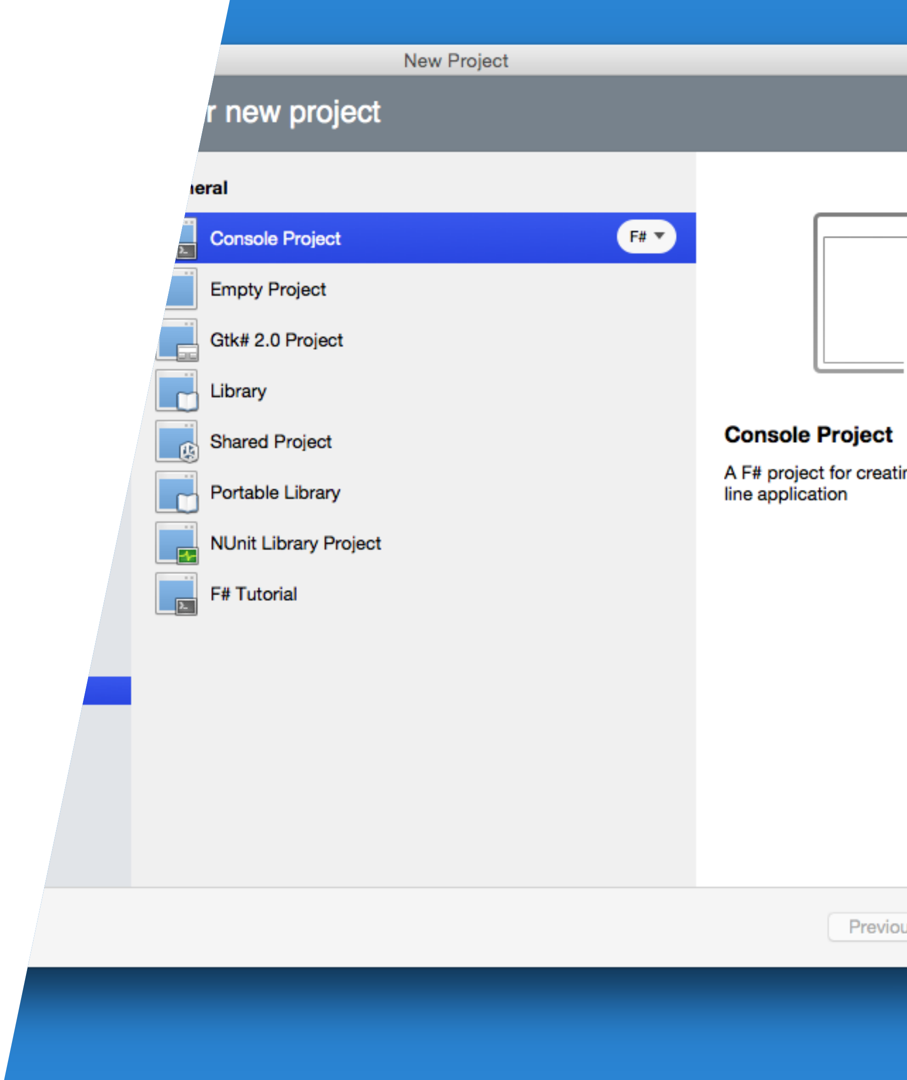
□ Lecture will begin shortly

□ Download class materials from university.xamarin.com

Xamarin University

Objectives

1. Break down the structure of an F# project
2. Identify the components of an F# program



Console Project

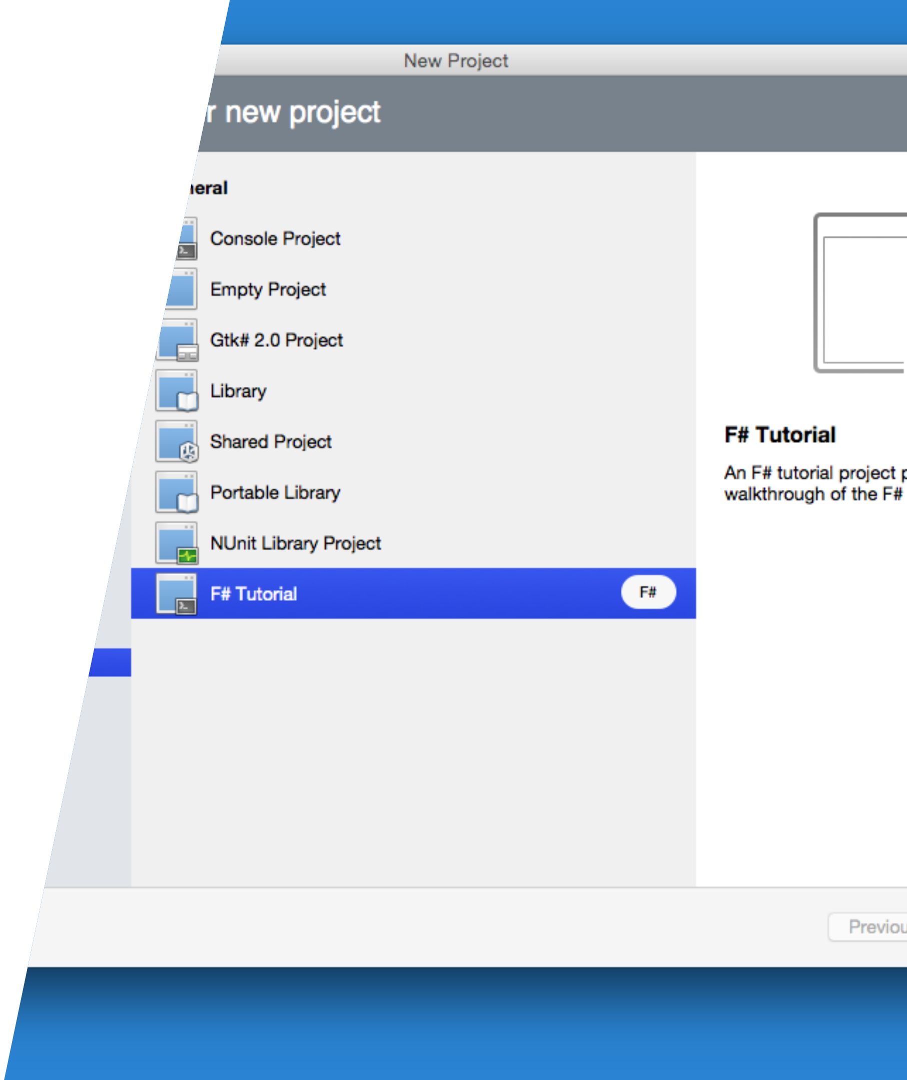
A F# project for creatir line application

Break down the structure of an F# project

Xamarin University

Tasks

1. Identify the types of projects
2. Compare script vs. code files



F# Tutorial

An F# tutorial project p
walkthrough of the F#

Projects in F#

- Available project templates are likely very familiar to you – they provide the starting code for the same types of projects you build in C#

ASP.NET

Android

iOS

Xamarin Forms

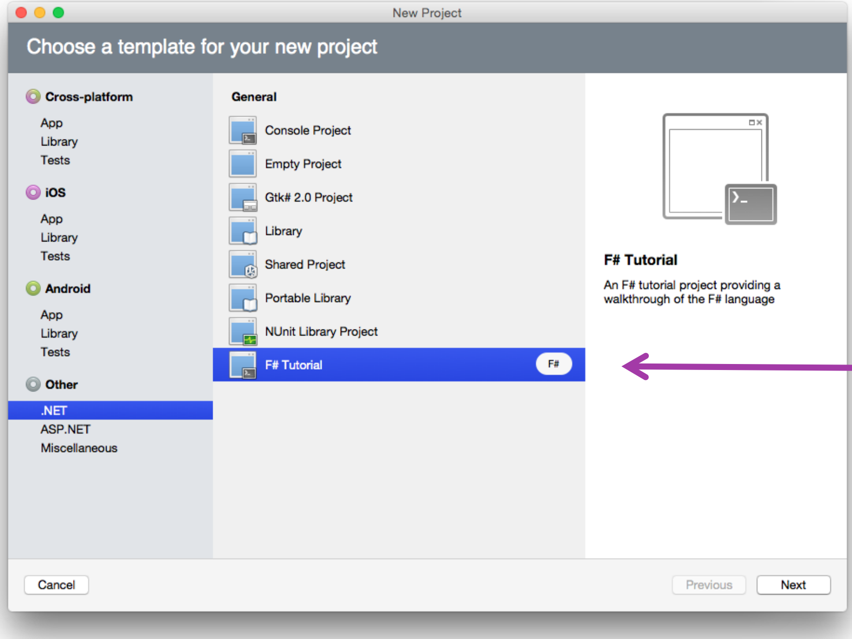
.NET Library

Console App

Unit Test

Tutorial application

- F# Tutorial template generates an app with example code and snippets



New Project

Choose a template for your new project

F# Tutorial

An F# tutorial project providing a walkthrough of the F# language

Generated project has some great examples and code you can use in your own projects

Demonstration

Explore the F# tutorial

C# project structure

- C# projects have a standard structure
  - Folders are above files
  - Files are listed alphabetically

- Any code within the project can reference any other code in the same project, regardless of location

Typical C# program file structure

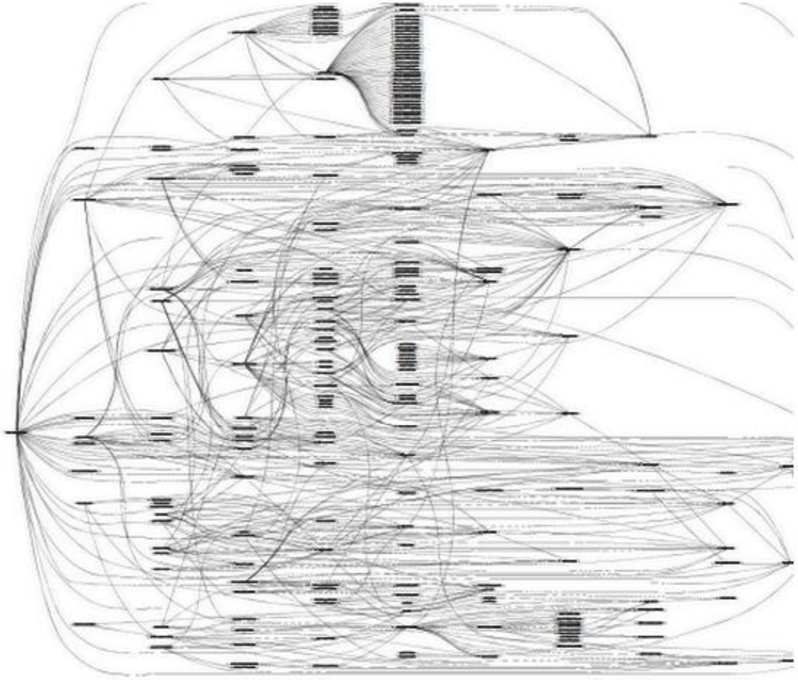


Image courtesy of: http://fsharpforfunandprofit.com/posts/cycles-and-modularity-in-the-wild/

F# project structure

- F# organizes the project differently and has some interesting rules which are designed to make the project structure and it's dependencies easier to read and manage

Typical program file structure in F#

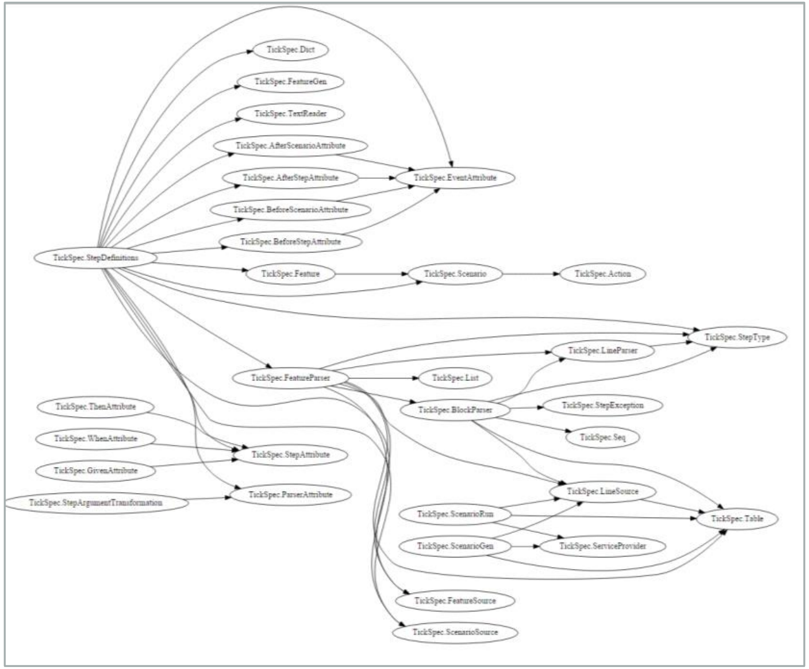
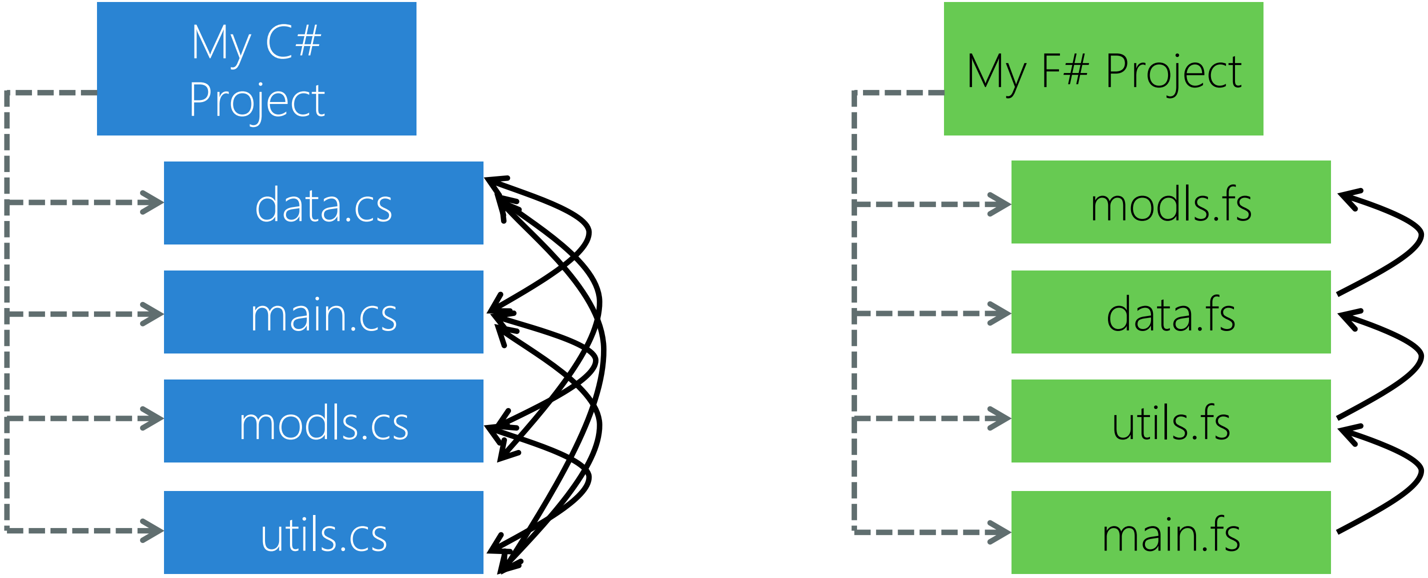


Image courtesy of: http://fsharpforfunandprofit.com/posts/cycles-and-modularity-in-the-wild/

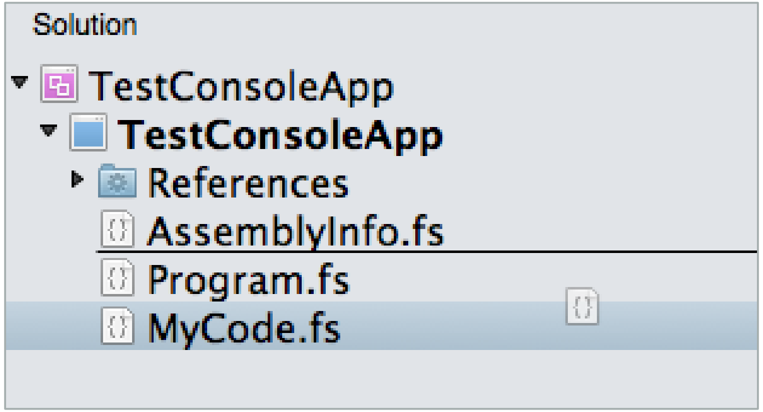
File ordering

- In F#, files aren't listed in your project alphabetically; they're listed in order of use

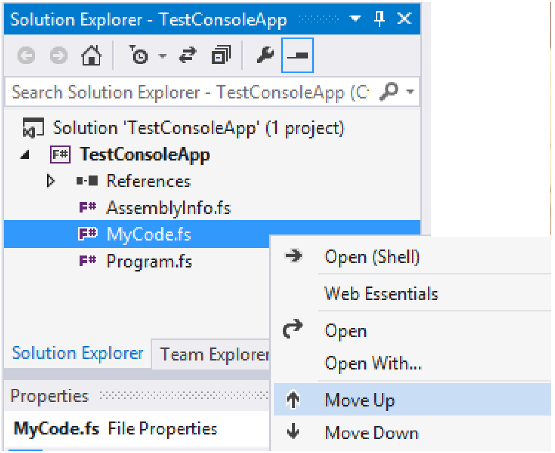


Consequences of file ordering

- When you add a new file to an F# project, you must move it *above* any file which wants to uses the types and values defined in it – this allows the compiler to infer types very easily



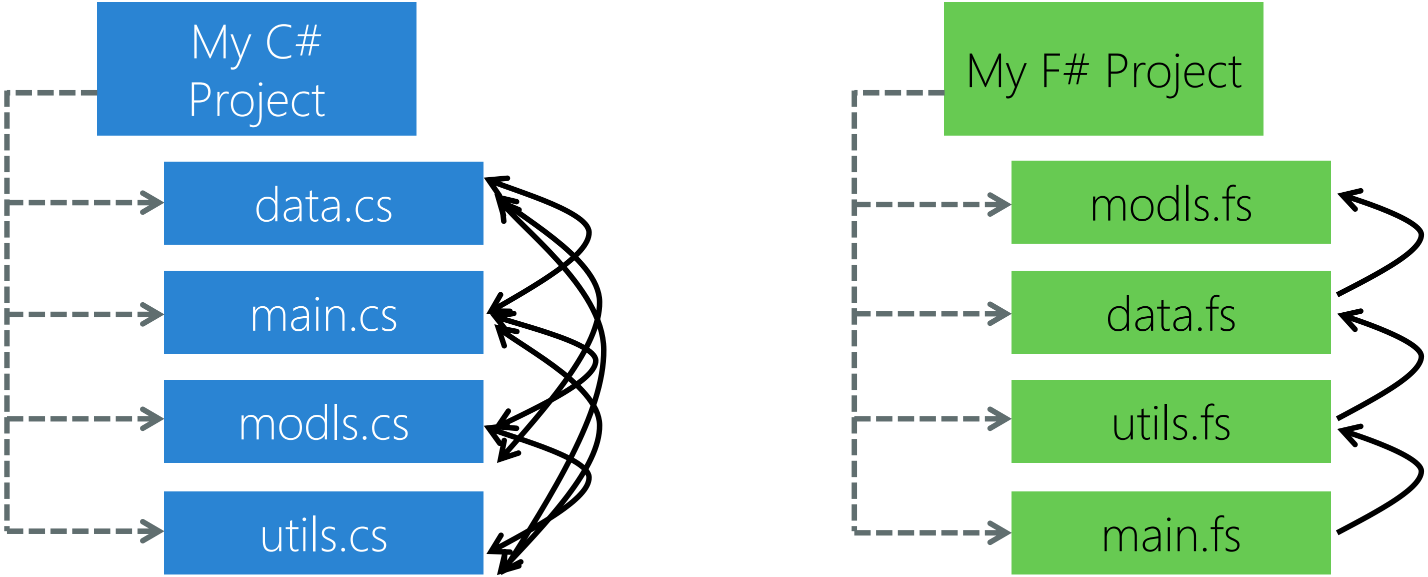
In Xamarin Studio, just drag and drop the source files in the solution



In Visual Studio, use the context menu to move files up and down

Top-down hierarchy

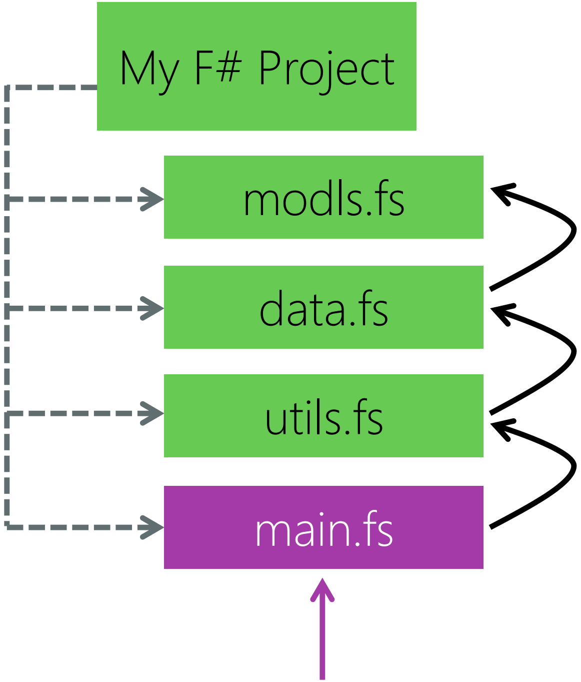
- In F#, you can only reference code that is declared above your current code, both in declaration and in file ordering



F# entry point

- Because of the enforced file ordering the entry point has to be the last method in the last file

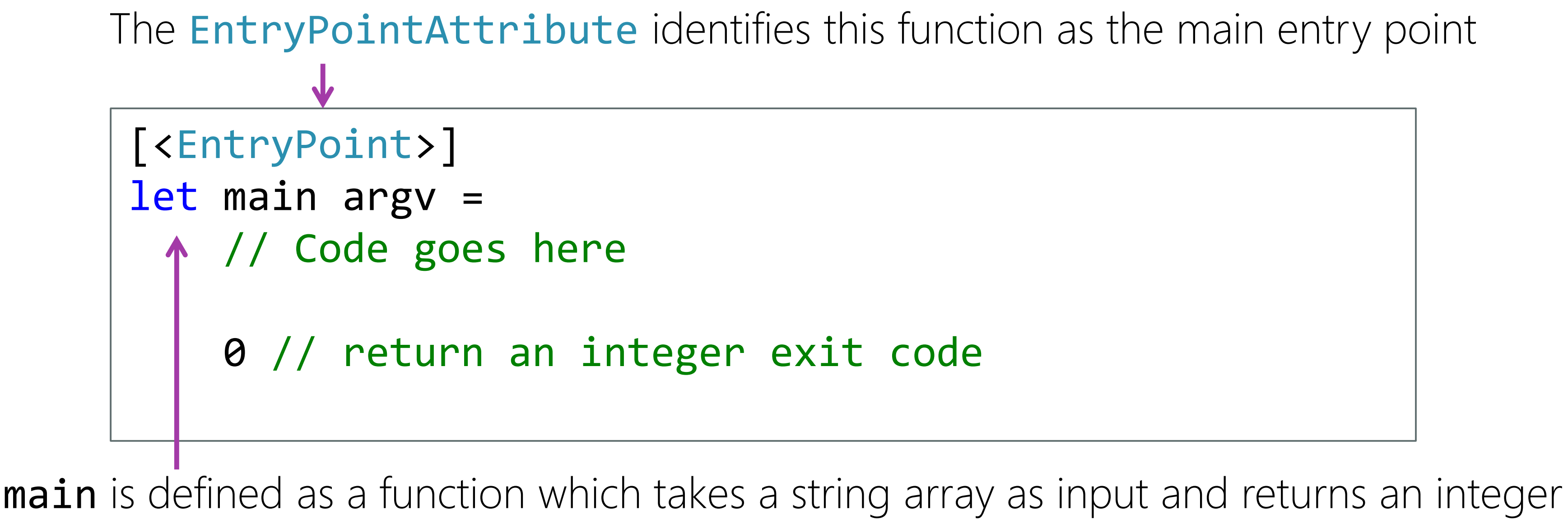
- This structure makes it much easier to analyze an existing F# program – both for people, but also for the IDE and compiler



Entry point will always be found here

What does the EntryPoint look like?

- F# projects have a defined main entry point – just like any other .NET app



The `EntryPointAttribute` identifies this function as the main entry point

```
[<EntryPoint>]
let main argv =
    // Code goes here

    0 // return an integer exit code
```

`main` is defined as a function which takes a string array as input and returns an integer

Individual Exercise

Reorder files in F#

Adding source files to your project

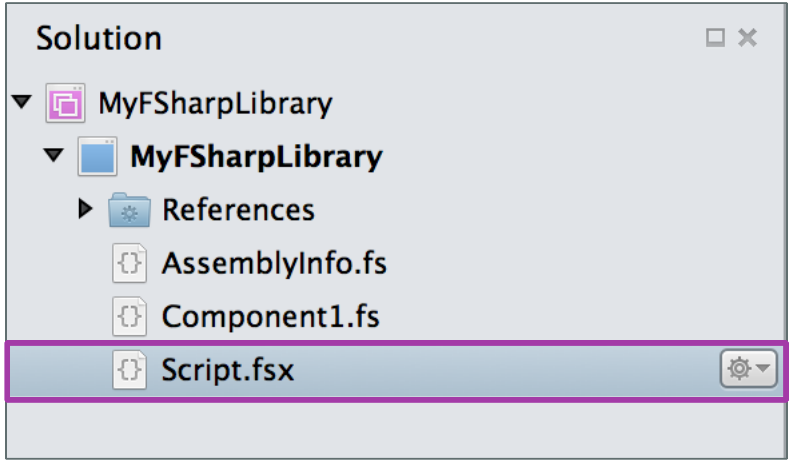
- F# supports two different types of source files for a project

Script Files

Code Files

Script Files

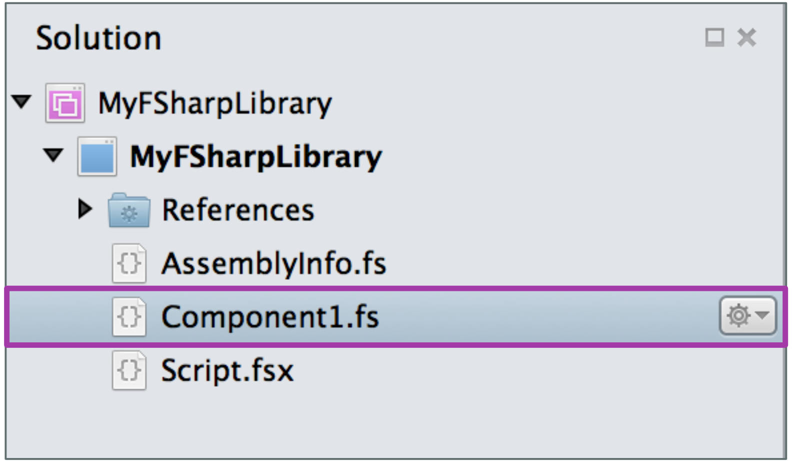
- Script files allow for interactive code exploration
  - Do not need to be part of an executable
  - Can test code without creating an application
  - Similar to PowerShell, Python or Ruby
  - Can leverage .NET types



Xamarin University

Code Files

- Code files are compiled into an executable and must exist within a project



Adding References

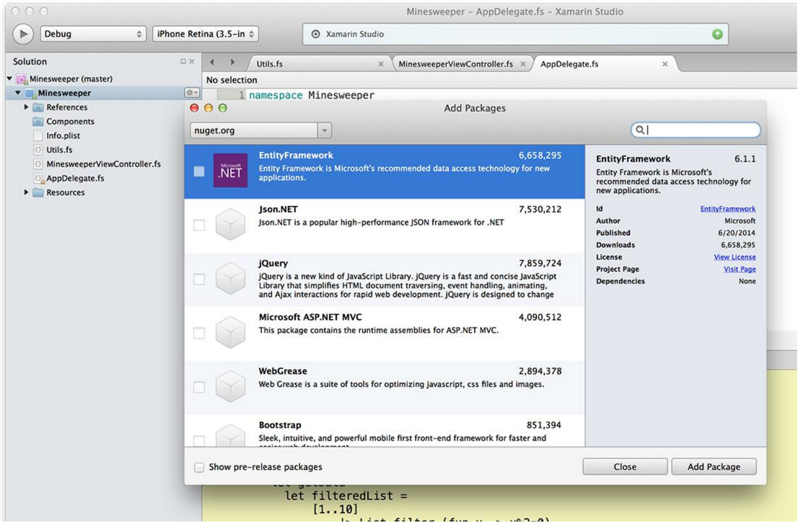
- To access code contained in an external assembly you must add a reference to it
  - for executable projects this works the same as it does in C# (References > Add or use Nuget)
  - for script files and the REPL you must use "#r" to load the assembly

```
> #r "System.Net";;

--> Referenced
'/Library/Frameworks/Mono.framework/Versions/3.10.0/lib/mono/4.5
/System.Net.dll' (file may be locked by F# Interactive process)
```

Referencing components

- Can add packages from NuGet
  - SendGrid
  - jQuery
  - Entity Framework
- Can also use components from the Xamarin Component Store



```
namespace Minesweeper
```

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

☐	**EntityFramework** Entity Framework is Microsoft's recommended data access technology for new applications.	6,658,295
☐	**Json.NET** Json.NET is a popular high-performance JSON framework for .NET	7,530,212
☐	**jQuery** jQuery is a new kind of JavaScript Library. jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change	7,859,724
☐	**Microsoft ASP.NET MVC** This package contains the runtime assemblies for ASP.NET MVC.	4,090,512
☐	**WebGrease** Web Grease is a suite of tools for optimizing javascript, css files and images.	2,894,378
☐	**Bootstrap** Sleek, intuitive, and powerful mobile first front-end framework for faster and	851,394

EntityFramework 6.1.1

Entity Framework is Microsoft's recommended data access technology for new applications.

Id	EntityFramework
Author	Microsoft
Published	6/20/2014
Downloads	6,658,295
License	View License
Project Page	Visit Page
Dependencies	None

Individual Exercise

Use a Nuget component to combine images into a PDF

Flash Quiz

Flash Quiz

① In F#, files are listed ________

a) Randomly

b) to increase circular dependencies

c) in order of use

Flash Quiz

① In F#, files are listed ________

a) Randomly

b) to increase circular dependencies

c) <u>in order of use</u>

Flash Quiz

② Script files can be included in an executable

a) True

b) False

Flash Quiz

② Script files can be included in an executable

a) True

b) False

Flash Quiz

③ In F#, you can *only* reference code which is ________

a) in a folder

b) in alphabetical order

c) above the current code

Flash Quiz

③ In F#, you can *only* reference code which is __________

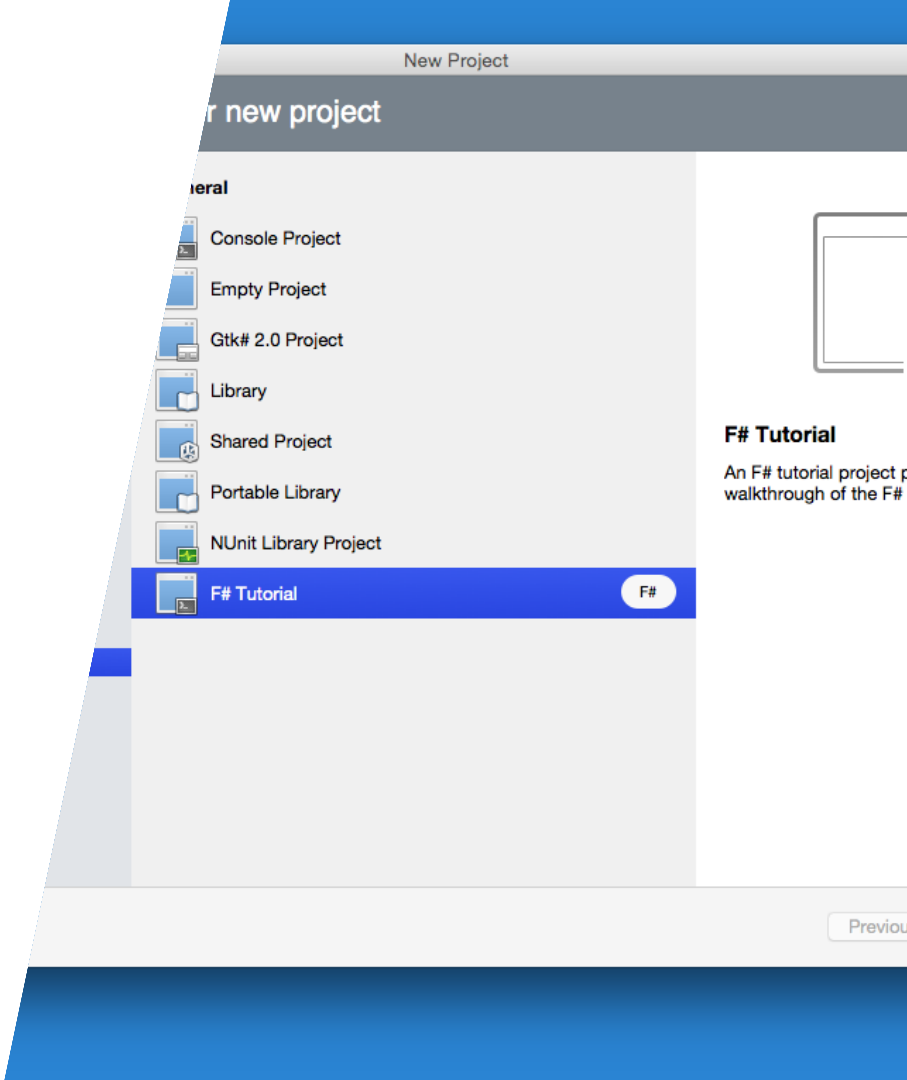
a) in a folder

b) in alphabetical order

c) above the current code

Summary

1. Identify the types of projects
2. Compare script vs. code files
3. Manage references



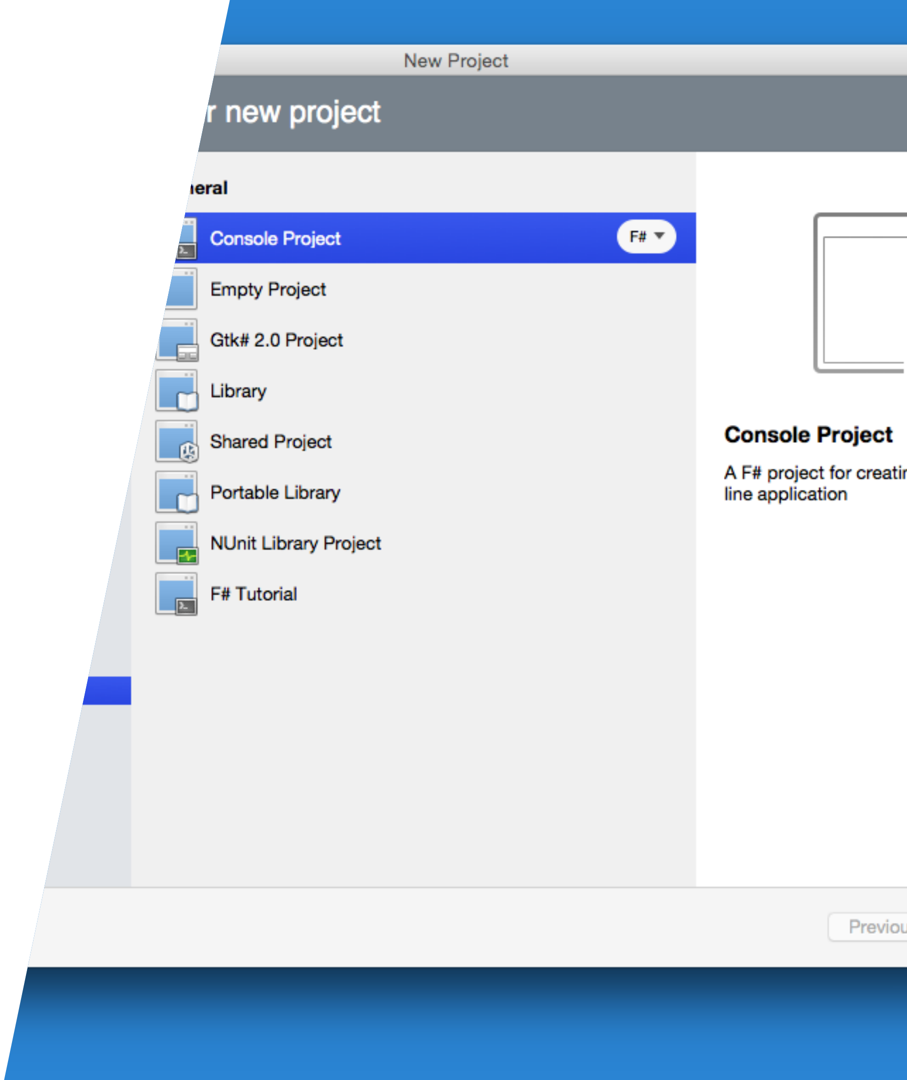
F# Tutorial

An F# tutorial project p
walkthrough of the F#

Identify the components of an F# program

Tasks

1. Namespaces
2. Types
3. Modules
4. Compare modules vs. types



New Project

r new project

ıeral

Console Project F#

Empty Project

Gtk# 2.0 Project

Library

Shared Project

Portable Library

NUnit Library Project

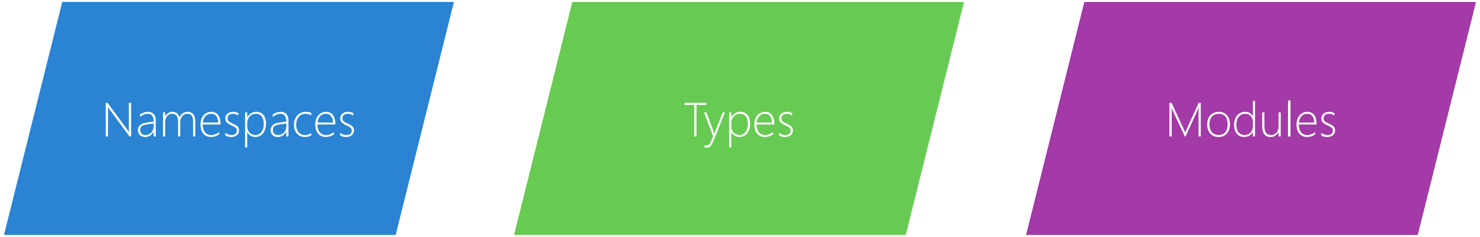
F# Tutorial

Console Project

A F# project for creatir line application

What makes up an F# program?

- F# programs consist of three basic elements



Namespaces

Types

Modules

Namespaces

- Namespaces can be used to organize our code and disambiguate types

The namespace definition must be the first thing in the source file – it applies to all the code that follows

The open keyword makes the contents of a namespace available to the code in this file, similar to a using statement but without quotes

```
namespace Minesweeper

open System
open System.IO
open System.Net
```

Namespace rules in F#

- Namespaces can span multiple source files (very common)
- Almost always have one namespace per file, however this is not a rule
- Namespaces can only contain classes, just like in C#

```
namespace Minesweeper.Utils

// ... Define types here

// Switch namespaces
namespace Minesweeper.Data

// ... more types here
```

can use dotted syntax to create nested namespaces

Using .NET types

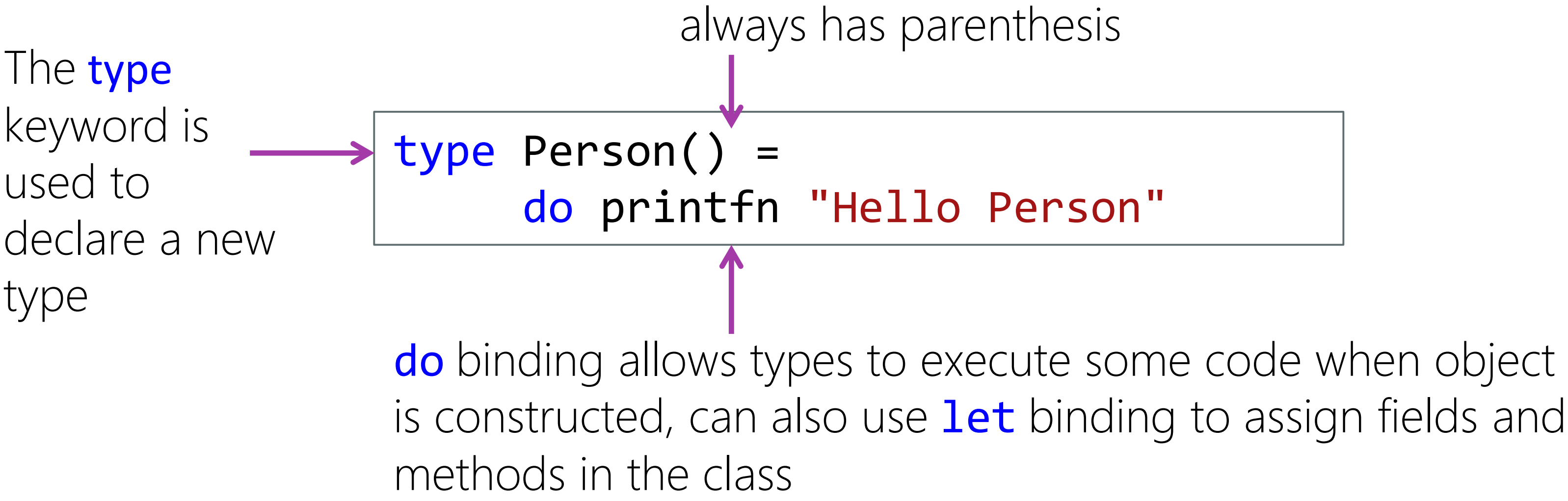
- .NET types are fully supported in F# and can be invoked using all the syntax rules you already know, this makes it very easy to utilize any existing C# or .NET Framework code in your programs or scripts

```
open System
open System.IO

let documents = Directory.GetFiles "/Users/mark/Desktop/"
for file in documents do
   Directory.SetLastAccessTime(file, DateTime.Now))
```

Creating custom types

- We define classes in F# through the **type** keyword, this is the same as the `class` **keyword in C#**



The **type** keyword is used to declare a new type

always has parenthesis

```
type Person() =
     do printfn "Hello Person"
```

do binding allows types to execute some code when object is constructed, can also use **let** binding to assign fields and methods in the class

Unlike C#, it is common to have several type definitions in one file

Initializing a type

❖ Types must define a primary constructor as part of their definition

```
type Person() =
     do printfn "Hello Person"
```

default constructor (no parameters)

.. constructor parameters can be required as part of definition

```
type Instructor (subject: string) =
     let focus = subject
     do printfn "Hello Instructor"
```

Type members

- Types can define public *properties* and *methods* using the **member** keyword

```
type Person (name, dob, gender) =
     member this.Name = name
     member this.Dob = dob
     member this.Gender = gender
  ...
```

Type members

- Types can define public *properties* and *methods* using the **member** keyword

```
type Person (name, dob, gender) =
     member this.Name = name
     member this.Dob = dob
     member this.Gender = gender
     member this.GetName() = name
     member this.Hello msg = printfn "%s %s" msg name
```

methods are assigned as F# **functions** and can have parameters

Self identifiers

- F# allows you to define the keyword used to represent the current instance for a class or method

```
type Person (name, dob, gender) =
     member me.Name = name
     member this.Dob = dob
     member identity.Gender = gender
     member self.GetName() = name
     member current.Hello msg = printfn "%s %s" msg name
```

Can even mix different keywords – F# understands your intension because of how it's being used

Secondary constructors

- Can declare additional constructors by using the **new** keyword

```
type Instructor (subject) =
     do printfn "Hello Instructor"
     // Add a second "default" constructor
     new () = Instructor("F#!")
```

Secondary constructors always chain to the primary constructor

Code in secondary constructors

- Secondary constructors must use the **then** keyword to execute code

```
type Person(name : string) =
    member this.Name = name
    new() as this =
        Person("Unknown")
        then printfn "Initializing Person with = %s" this.Name
```

must define a self identifier to get to **Name** property, that is what the **as** keyword does for the method

Inheritance

- Types can inherit from a single base class – just like C#

```
type Instructor (name, subject) =
     inherit Person (name)
     ...
```

can control which constructor is used

Defining interfaces

- Interfaces can be defined using the **type** keyword as well, but all the members must be **abstract**

Missing parenthesis indicates *no constructor* which is what makes this an interface vs. an abstract class type

```
type ITeacher =
   // method void Teach(string)
   abstract member Teach: string -> unit
   // property string Focus
   abstract member Focus: string
```

Xamarin University

Implementing interfaces

- Types can implement multiple interfaces, also like C#

```
type Instructor (name, subject) =
   inherit Person (name)

   // Implement ITeacher interface
   interface ITeacher with
      member this.Teach(name) = printfn "Teaching %s" name
      member this.Focus = subject
```

Interfaces are always implemented *explicitly* in F# and will require a cast to access the interface implementation

Using interfaces in F#

- Since interfaces implementations are explicit, they will require a cast to get to the functionality

```
let instr = new Instructor("Mark", "Projects in F#")
(instr :> ITeacher).Teach("Helen")
```

Dealing with type ordering

- F# types can only refer to other types *declared above them*, but what if we have two classes that mutually reference each other

```
public class Invoice
{
    private List<Product> productList =
          new List<Product>();
    public void Add(Product product)
    {
        productList.Add(product);
    }
}
```

```
public class Product
{
    ...
    public void Purchase(
           Invoice invoice)
    {
        invoice.Add(this);
    }
}
```

We have defined a *circular* reference between these two classes, valid in C#

Type Ordering in F#

- It is best to avoid circular references, however you *can* indicate there is a relationship so that types are defined together using the **and** keyword

```
type Product (price, onSale) =
    member this.SalePrice = if onSale && price <> 0.
                               then price/2. else price
    member this.Purchase(invoice : Invoice) =
        invoice.Add(this)

and Invoice () =
    let productList = List.empty<Product>
    member this.Add(product : Product) = List.append productList
```

Note: this approach is discouraged by most F# programmers, instead try to change the relationship so that the dependency is one direction

Flash Quiz

Xamarin University

Flash Quiz

① F# types can implement interfaces

a) True

b) False

Flash Quiz

① F# types can implement interfaces

a) True

b) False

Flash Quiz

② F# types must ___________

a) be defined in a namespace

b) define a primary constructor as part of the type definition

c) Both of these

Flash Quiz

② F# types must __________

a) be defined in a namespace

b) define a primary constructor as part of the type definition

c) Both of these

Modules

- A *module* is a grouping of related F# code, such as values, function values, and types; it is similar in concept to namespaces however it can contain more things
- Modules are compiled to a static class and cannot span source files
- Code defined at the top-level in a file is automatically placed in a module

```
namespace MineSweeper

open System

module Utilities =

type ITeacher =
    ...
type Person () =
    ...
```

These two types are contained in the `MineSweeper.Utilities` static class

Organizing our code

- In F#, data structures and the functions that work with them are often placed in modules instead of types – this provides a similar structure but is more common in F#

```
module Geometry =
    module Utilities =
        let Area width height = width * height
        let Perimeter width height = 2*width + 2*height
```

can *nest* modules to provide more specific containment and isolation

```
let area = Geometry.Utilities.Area 10 20
```

Modules vs. types in F#

- Modules and types are similar in many ways – in fact a module *actually is* a type to .NET, however we use them for different purposes

Modules	Types
Not necessary to appear in a namespace	Must appear in a namespace
More common for standalone F# projects	Easier to use when interoperating with C#
Common to nest modules inside one another	Uncommon to nest inside one another

Putting it all together

- F# programs consist of namespaces, types and modules

```
namespace Minesweeper

type Cell (x, y) =
    let mutable isChecked = false
    member this.XPos = x
    member this.YPos = y
    member this.IsChecked
        with get() = isChecked
        and set(value) = isChecked <- value

module Geometry =
    module Utilities =
        let Area width height = width * height
        let Perimeter width height = 2*width + 2*height
```

`type` definitions often appear *before* modules

`module` will contain the code to act on the types listed above

Individual Exercise

Create and run a console app

Flash Quiz

Flash Quiz

① Modules ________

a) are made up of classes, types and references

b) contain values, function values and types

c) must appear in namespaces

Flash Quiz

① Modules ________

a) are made up of classes, types and references

b) contain values, function values and types

c) must appear in namespaces

Flash Quiz

② The `member` keyword identifies ______

a) properties and methods in a class
b) a constructor to create an object
c) a class

Flash Quiz

② The `member` keyword identifies ______

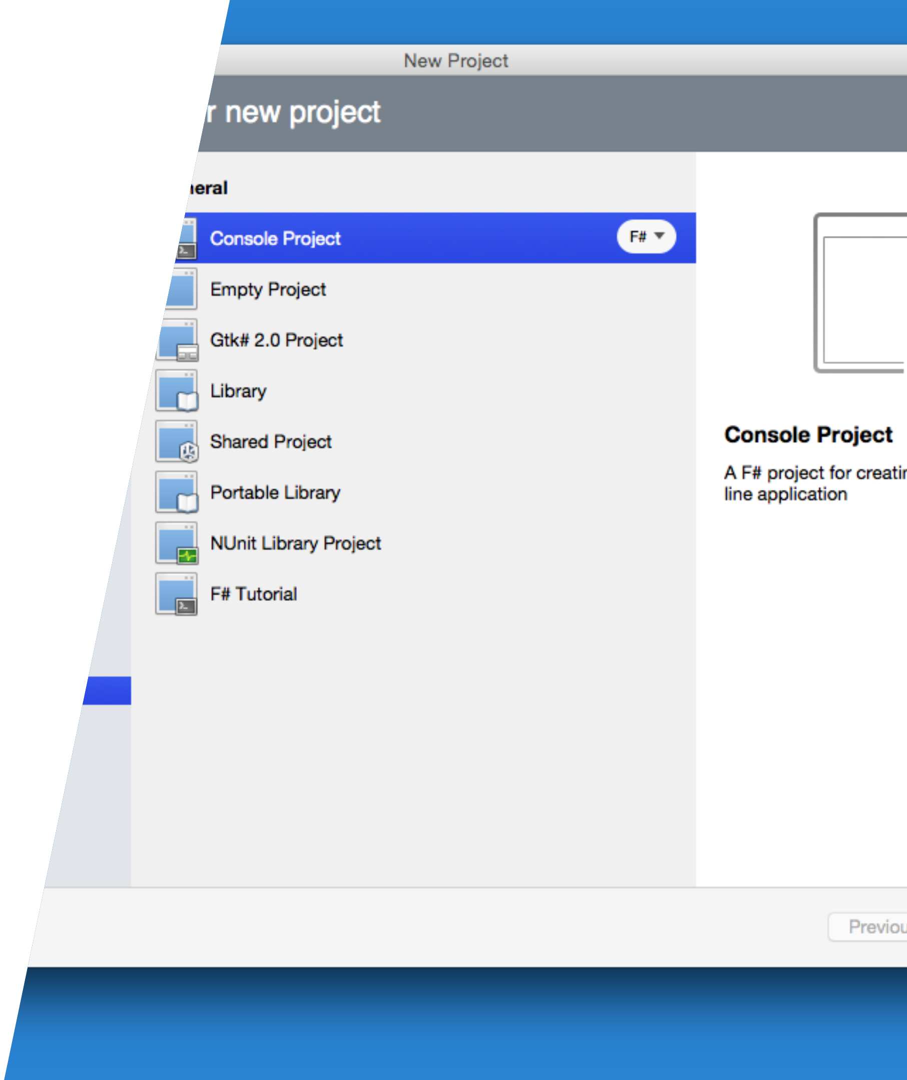
a) properties and methods in a class

b) a constructor to create an object

c) a class

Summary

1. Namespaces
2. Types
3. Modules
4. Compare modules vs. types



Console Project

A F# project for creatir
line application

Where are we going from here?

- You now know how to structure an F# application and the various templates that are available
- In the next course, we will look at some of the common data structures you use in F#

Thank You!

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