Using F# with an Existing Codebase

Wesley Wiser

The Codebase

- Actively developed for 10+ years
- 100,000s lines of C#
- 10,000s lines of Js
- ~10,000 lines of F#
- A large amount of Web-Forms style ASPX

Why Add F#?

- Strengths of the language
  - Pattern Matching
  - Data transformation & processing
- Developer productivity
- Easier to write bug-free code

How we use F#

- Data transformation
- Define data types

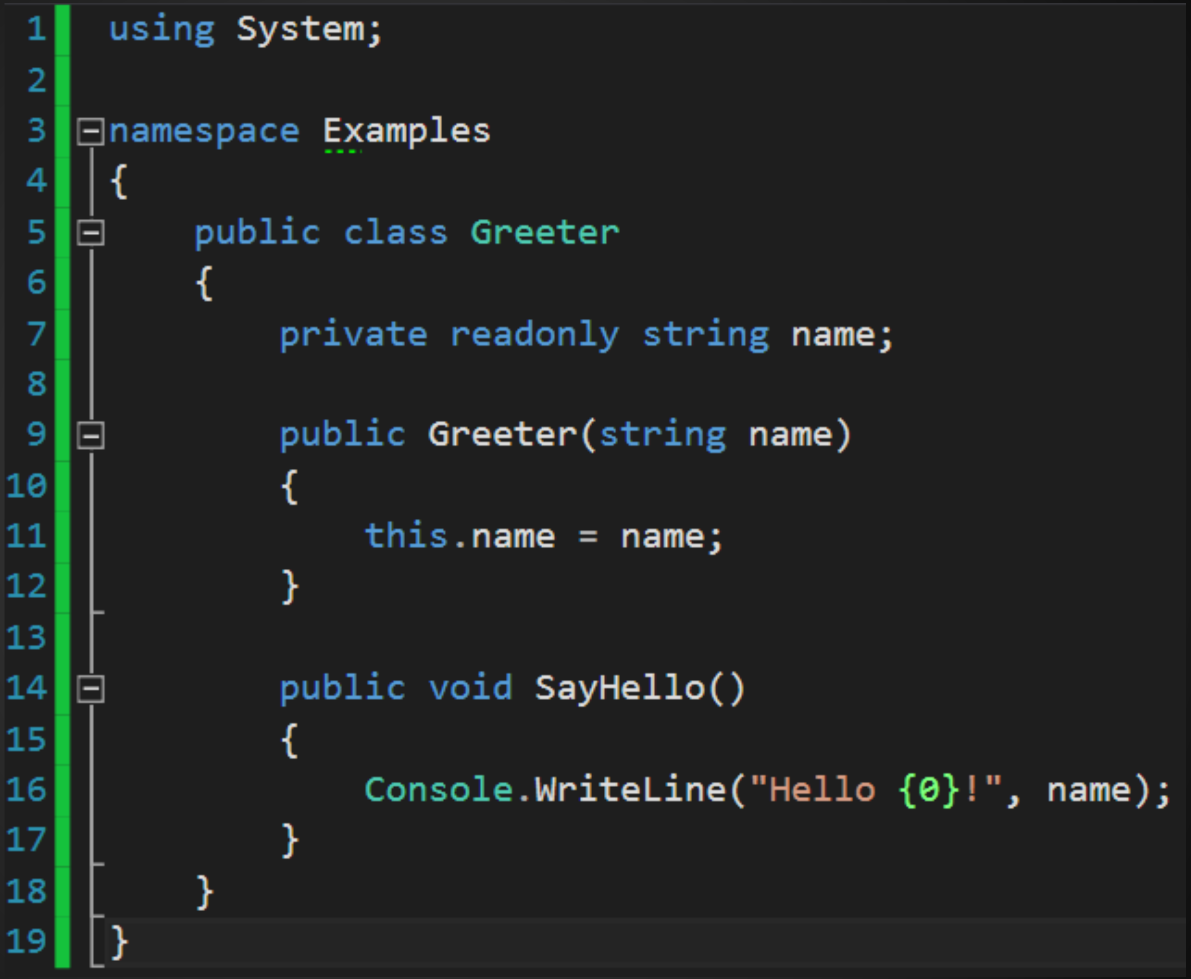
Data transformation

- Map-Reduce
  - Not just for big data
- Translation
  - Text markup -> Html or plain text
  - Xml -> Business objects
  - Events -> Configuration

Classes

- “F# is a better object oriented language than C# in some ways” - Erik Meijer

Classes



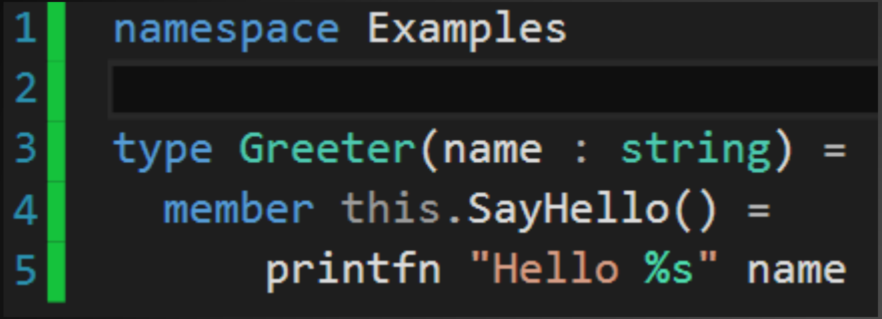
```
using System;

namespace Examples
{
    public class Greeter
    {
        private readonly string name;

        public Greeter(string name)
        {
            this.name = name;
        }

        public void SayHello()
        {
            Console.WriteLine("Hello {0}!", name);
        }
    }
}
```

Classes



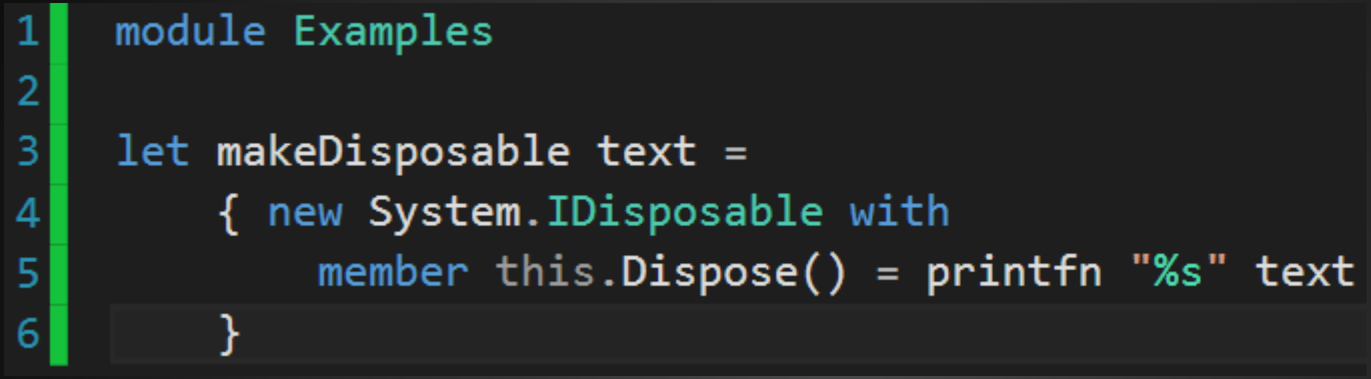
```fsharp
namespace Examples

type Greeter(name : string) =
  member this.SayHello() =
      printfn "Hello %s" name
```

Object Expressions

- Similar to Java’s Anonymous Class feature
- Useful when you need to return an object’s interface but wish to hide the implementation

Object Expressions



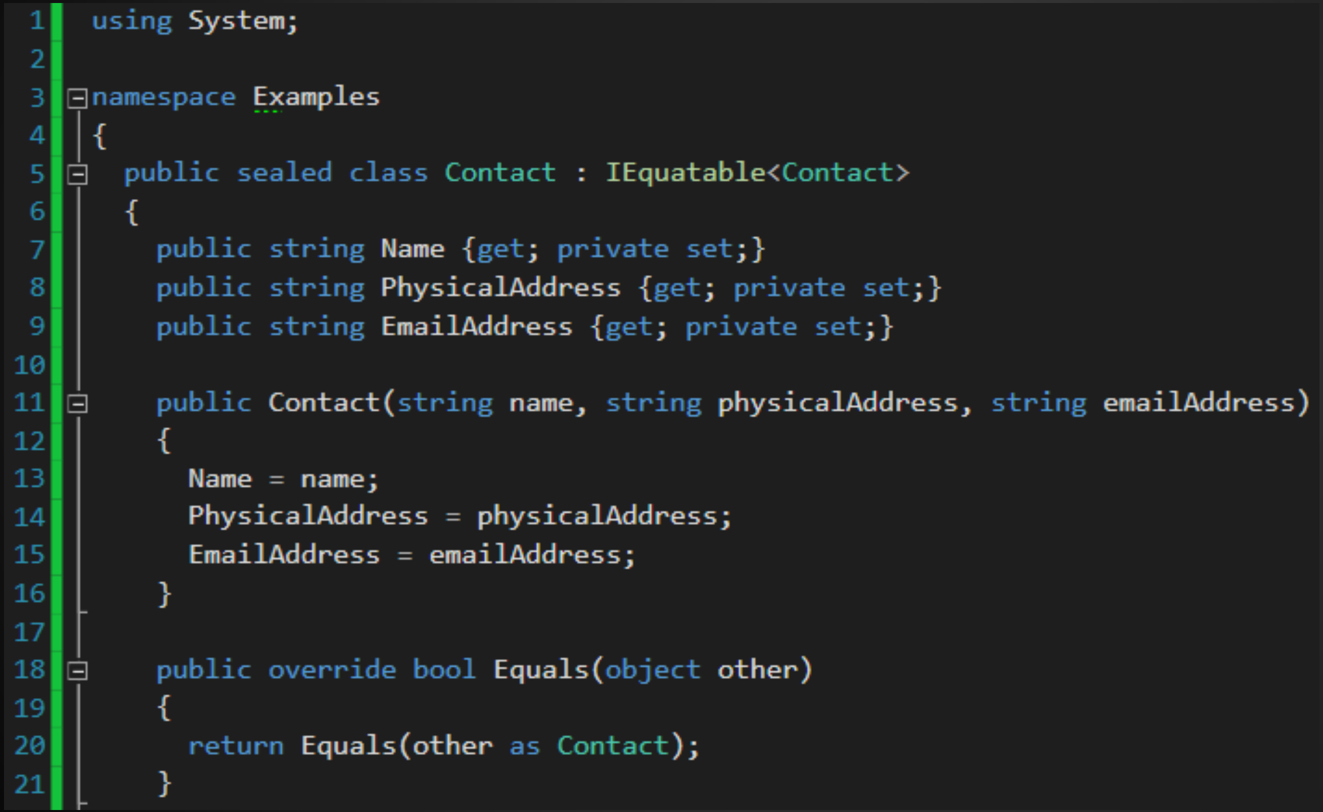
```fsharp
module Examples

let makeDisposable text =
    { new System.IDisposable with
        member this.Dispose() = printfn "%s" text
    }
```

Data Objects

- F# syntax is lighter weight
- F# compiler provides high-quality Equals(), GetHashCode(), and ToString() implementations for you

Data Objects



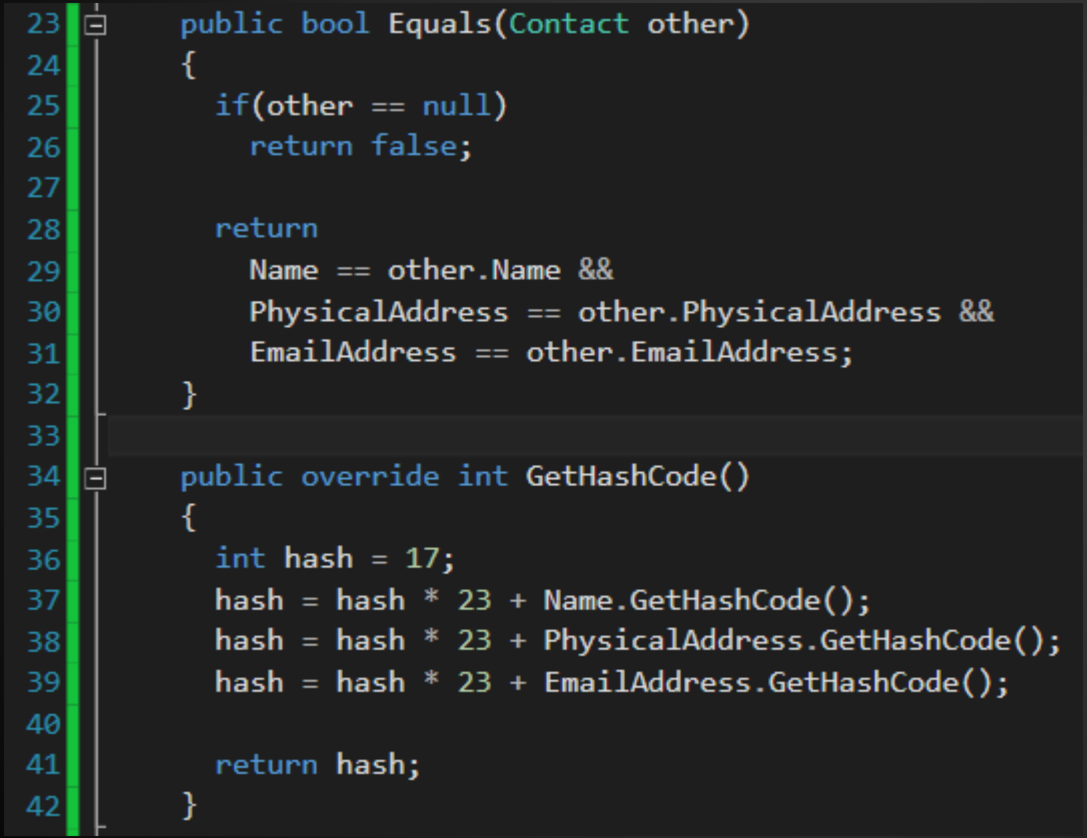
```
using System;

namespace Examples
{
  public sealed class Contact : IEquatable<Contact>
  {
    public string Name {get; private set;}
    public string PhysicalAddress {get; private set;}
    public string EmailAddress {get; private set;}

    public Contact(string name, string physicalAddress, string emailAddress)
    {
      Name = name;
      PhysicalAddress = physicalAddress;
      EmailAddress = emailAddress;
    }

    public override bool Equals(object other)
    {
      return Equals(other as Contact);
    }
```

Data Objects



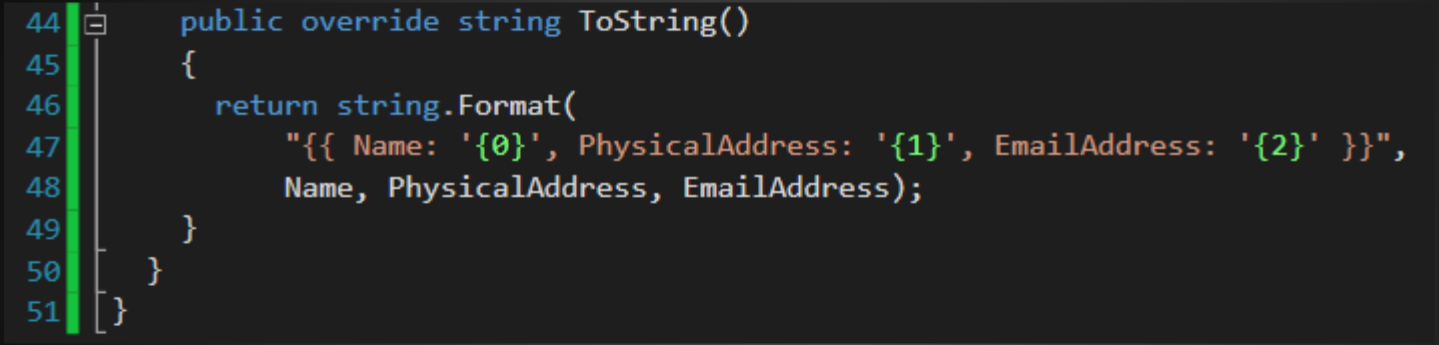
```
public bool Equals(Contact other)
{
  if(other == null)
    return false;

  return
    Name == other.Name &&
    PhysicalAddress == other.PhysicalAddress &&
    EmailAddress == other.EmailAddress;
}

public override int GetHashCode()
{
  int hash = 17;
  hash = hash * 23 + Name.GetHashCode();
  hash = hash * 23 + PhysicalAddress.GetHashCode();
  hash = hash * 23 + EmailAddress.GetHashCode();

  return hash;
}
```

Data Objects



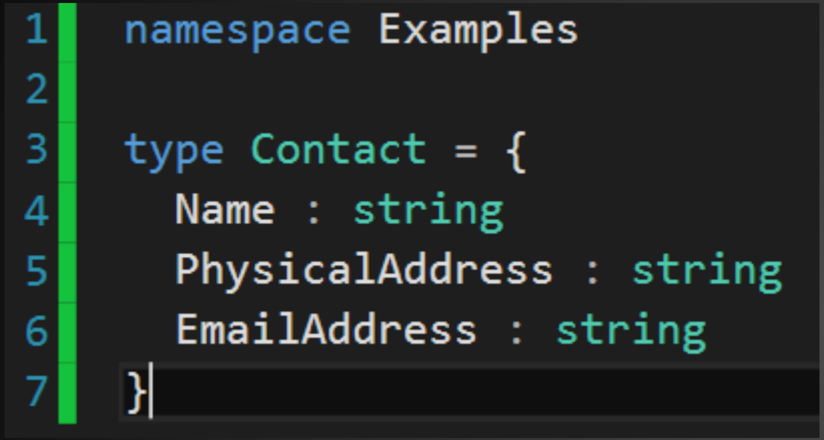
```
    public override string ToString()
    {
      return string.Format(
          "{{ Name: '{0}', PhysicalAddress: '{1}', EmailAddress: '{2}' }}",
          Name, PhysicalAddress, EmailAddress);
    }
  }
}
```

Data Objects

We could have also implemented

- IStructuralEquatable
- IComparable<Contact>
- IComparable
- IStructuralComparable

Data Objects



```
namespace Examples

type Contact = {
  Name : string
  PhysicalAddress : string
  EmailAddress : string
}
```

Data Objects

The F# compiler automatically implements

- IEquatable<Contact>
- IStructuralEquatable
- IComparable<Contact>
- IComparable
- IStructuralComparable

and provides overrides for

- Equals
- GetHashCode
- ToString

Primitive Obsession

- Primitive Obsession is using primitive data types to represent domain ideas. For example, we use a String to represent a message, an Integer to represent an amount of money, or a Struct/Dictionary/Hash to represent a specific object.
    - http://c2.com/cgi/wiki?PrimitiveObsession

Primitive Obsession

What does this function do?

```
fun1 : string -> string -> int option
```

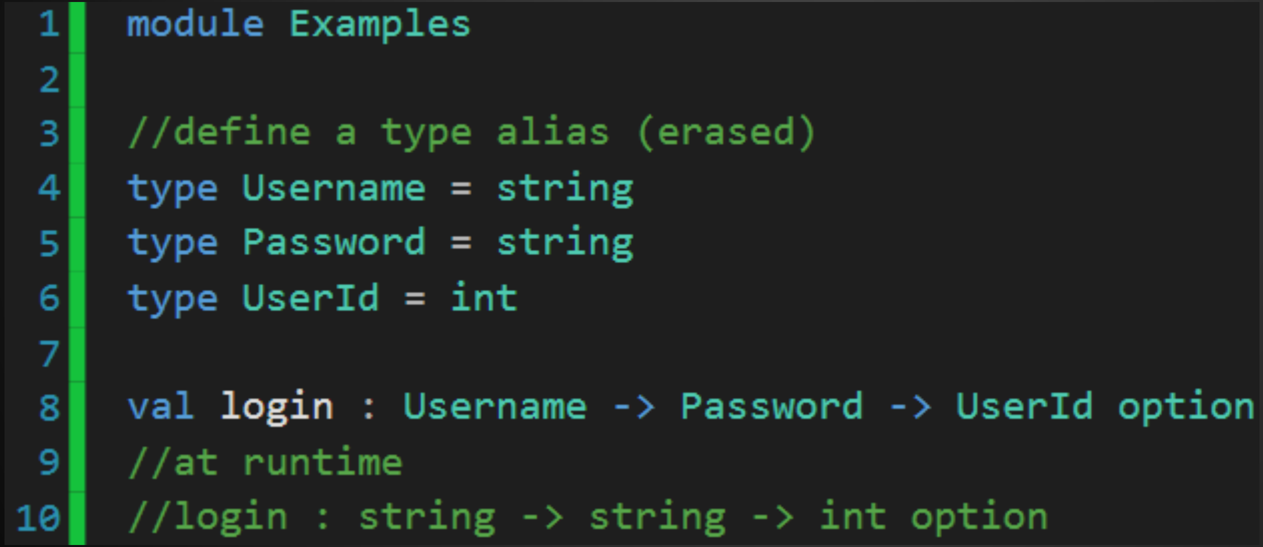
Primitive Obsession

What does this function do?

```
fun2 : Username -> Password -> UserId option
```

Primitive Obsession

If you were only writing F#:



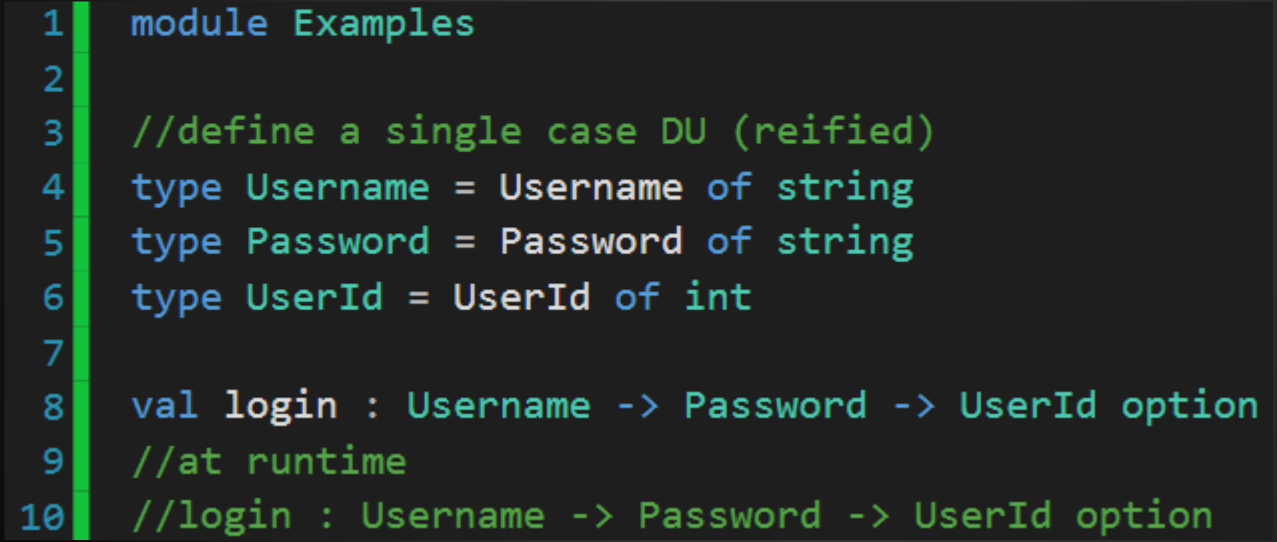
```fsharp
module Examples

//define a type alias (erased)
type Username = string
type Password = string
type UserId = int

val login : Username -> Password -> UserId option
//at runtime
//login : string -> string -> int option
```

Primitive Obsession

If you need reified types:



```
module Examples

//define a single case DU (reified)
type Username = Username of string
type Password = Password of string
type UserId = UserId of int

val login : Username -> Password -> UserId option
//at runtime
//login : Username -> Password -> UserId option
```

Other places

- Build scripts
    - FAKE
    - .fsx scripts
- Infrastructure
- Tests

That’s it

Questions or comments?

@wesleywiser