A primer to the world of Functional Programming with F#

- Kai Ito
- https://www.xing.com/profile/Kai_Ito/

- https://github.com/kaeedo/IntroductionToFunctionalProgramming

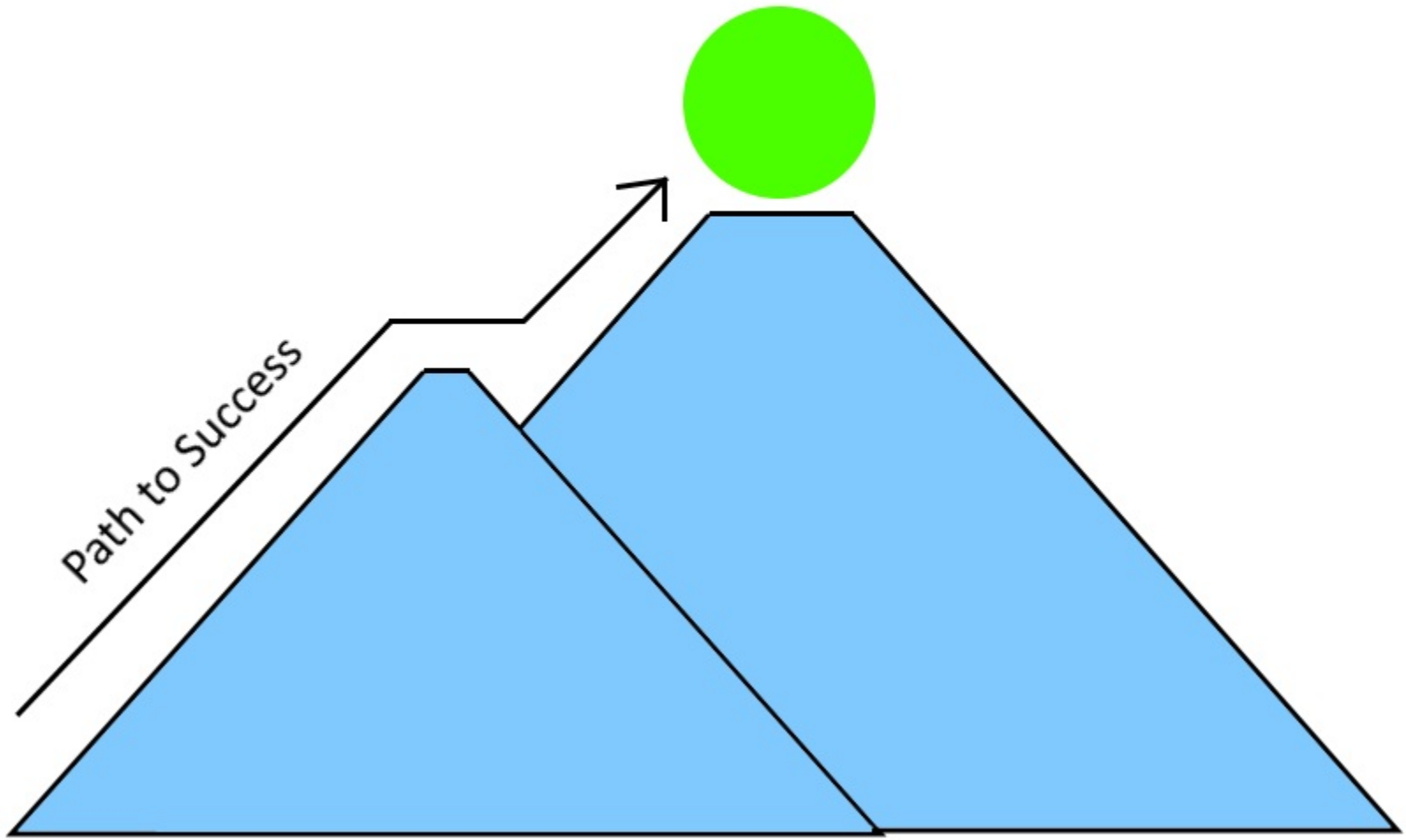
What is Functional Programming

- Function Composition over Inheritance
- Expressions over Statements
- Immutability
- Use of higher-order functions
- Use of pure functions

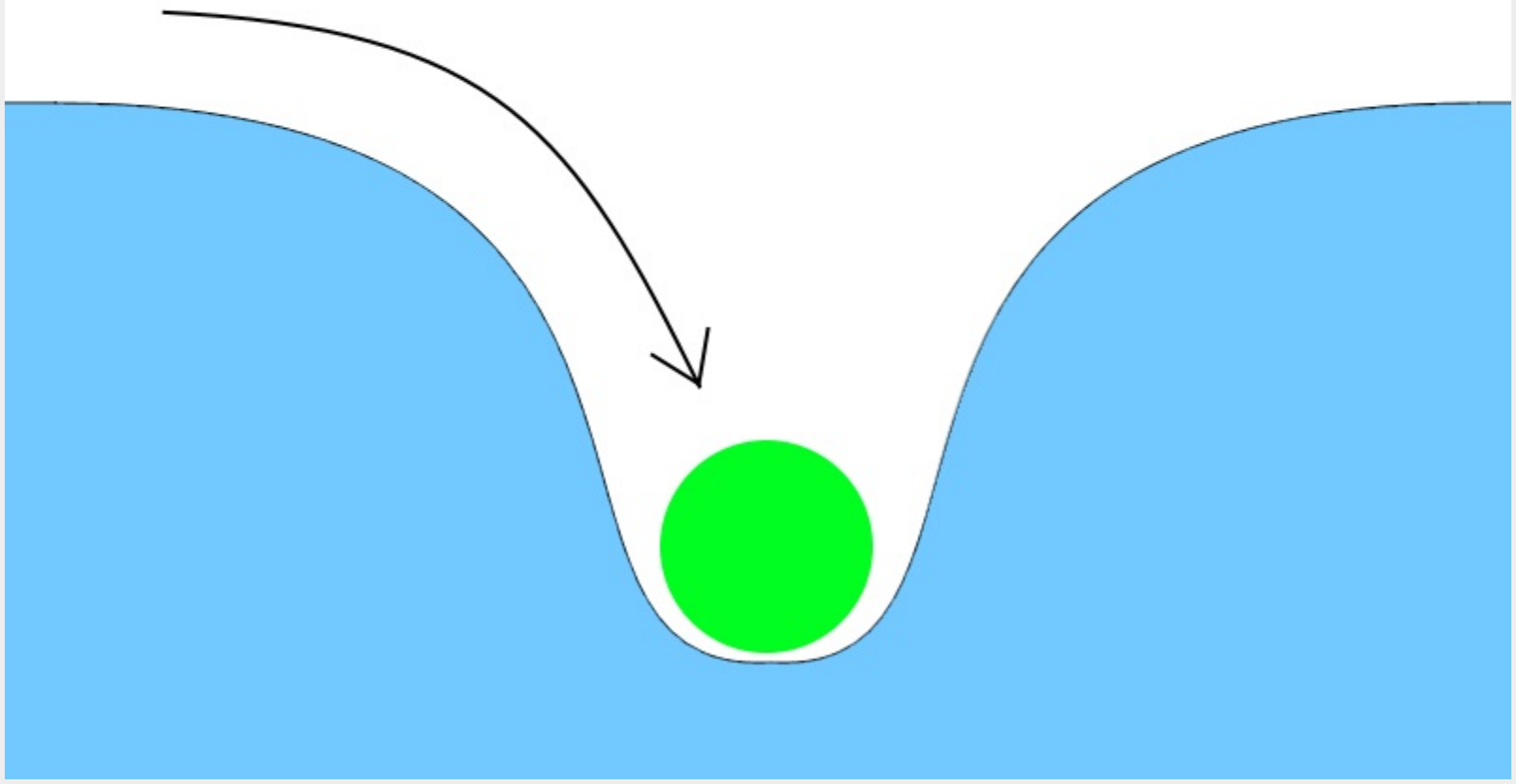
Benefits of the Functional paradigm

- Higher order functions
  - High level abstractions
  - Code reusability
- Pure Functions
  - Easier to reason about
  - Easier to test
  - Easier to debug
- Immutability
  - Less bugs
  - No invalid state
  - Thread safety
- "If it compiles, it works"
- "Pit of Success"

Mountain of Success



Pit of Success



What is F#

- Functional first, multi-paradigm language
- Runs on .Net (and Mono, Xamarin, .Net Core)
- First released in 2005 by Microsoft Research
- Now belongs to The F# Software Foundation
- Open Source

Syntax in a nutshell

- ML syntax
- Type inference
- Whitespace significant
- Expression-based
- Immutable by default

```
1: let thisIsAnInt = 1
2: let thisIsAString = "This is a string"
3: let optionalTypeAnnotation: bool = true
4:
5: let listOfInts = [ 1; 2; 3 ]
6: let listOfStrings =
7:     [ "www.zuehlke.com"
8:       "www.google.com"
9:       "www.microsoft.com" ]
```

Functions

- First class functions
- Last expression is the return "statement"

```
 1: let add a b =
 2:     a + b
 3:
 4: let sign num =
 5:     if num > 0 then "positive"
 6:     elif num < 0 then "negative"
 7:     else "zero"
 8:
 9: let rec fib n =
10:     match n with
11:     | 1 -> 1
12:     | 2 -> 1
13:     | n -> fib(n-1) + fib(n-2)
```

Piping

```
let printer f = printfn "Number is: %f" f

[0.0..100.0]
|> List.filter (fun i -> i % 2.0 = 0.0)
|> List.map (fun i -> i ** 2.0)
|> List.iter printer

let (|>) value fn =
    fn value

```

Records

- Simple aggregates of data
- Can be struct or reference types
- Has structural equality

```
 1: type User =
 2:     { FirstName: string
 3:       LastName: string
 4:       Age: int }
 5:
 6: let kai = { FirstName = "Kai"; LastName = "Ito"; Age = 27 }
 7: let cloneOfKai = { FirstName = "Kai"; LastName = "Ito"; Age = 27 }
 8:
 9: printfn "%b" (kai = cloneOfKai) // true
10:
11: let olderKai = { kai with Age = kai.Age + 1 }
12:
13: printfn "%i" (olderKai.Age) // 28
14: printfn "%i" (kai.Age) // 27
```

Discriminated Unions

- More powerful enum
- Data point that can have multiple different types

```
 1: type Shape =
 2: | Rectangle of width: float * length: float
 3: | Circle of radius: float
 4: | Triangle of float * float * float
 5:
 6: let rectangle: Shape = Rectangle (2.0, 5.0)
 7: let circle: Shape = Circle 2.5
 8: let triangle: Shape = Triangle (6.1, 2.0, 3.7)
 9:
10: let whichShape shape =
11:     match shape with
12:     | Rectangle (width, length) ->
13:         printfn "Rectangle with sides %f %f" width length
14:     | Circle radius ->
15:         printfn "Circle with radius %f" radius
16:     | Triangle (side1, side2, side3) ->
17:         printfn "Triangle with sides %f %f %f" side1 side2 side3
```

Benefits of the F# type system

- No null
- Make illegal states unrepresentable
- Use types to represent the domain
- Types can also be used to encode business logic
- Files and code must be in dependency order

The Option type

```fsharp
type Option<'a> =
| Some of 'a
| None

let validInt: int option = Some 1
let invalidInt: int option = None

let numbers = [ 1; 2; 3; 4; ]
let foundNumber = numbers |> List.tryFind (fun x -> x = 4)
let missingNumber = numbers |> List.tryFind (fun x -> x = 50)

printfn "The number is: %i" foundNumber
// Compile Error: Type mismatch: Expecting "int" but got "int option"

let printInt input =
    match input with
    | Some i -> printfn "The number is: %i" i
    | None -> printfn "Didn't find number"

printInt foundNumber // The number is: 4
printInt missingNumber // Didn't find number
```

Making illegal state unrepresentable

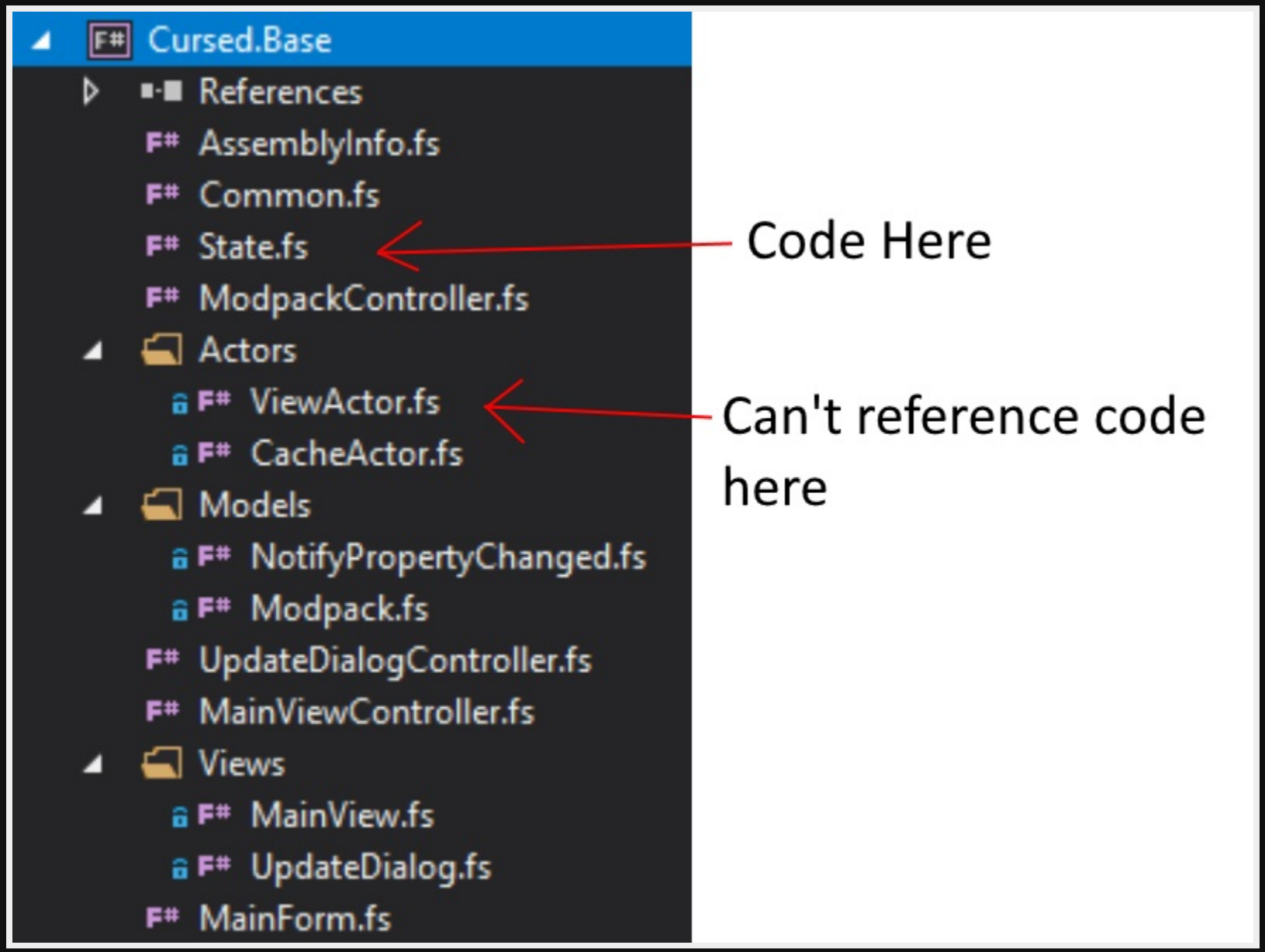
- Imagine business logic where a User either needs an email address or phone number or both
- Required to have at least one of them

```
1: type ContactUser = { Username: string; Email: string option; PhoneNumber:
2:
3: let createUser username emailAddress phoneNumber =
4:     // Logic to assign either email address or phone or both
5:     // Error prone
6:     { Username = "kaiito"; Email = "kai.ito@zuehlke.com"; PhoneNumber = "
7:
```

F# Type System to the rescue

```fsharp
type ContactInformation =
| Email of string
| PhoneNumber of string
| EmailAndPhone of string * string

type SafeContactUser = { Username: string; Contact: ContactInformation }

let email = Email "kai.ito@zuehlke.com"
let phoneNumber = PhoneNumber "089 555 1234"
let emailAndPhone = EmailAndPhone ("kai.ito@zuehlke.com", "089 555 1234")

let user1 = { Username = "kaiito1"; Contact = email }
let user2 = { Username = "kaiito2"; Contact = phoneNumber }
let user3 = { Username = "kaiito3"; Contact = emailAndPhone }

let user4 = { Username = "kaiito4"; Contact = null } // Compiler Error
let user5 = { Username = "kaiito5"; Contact = "someString" } // Compiler
```

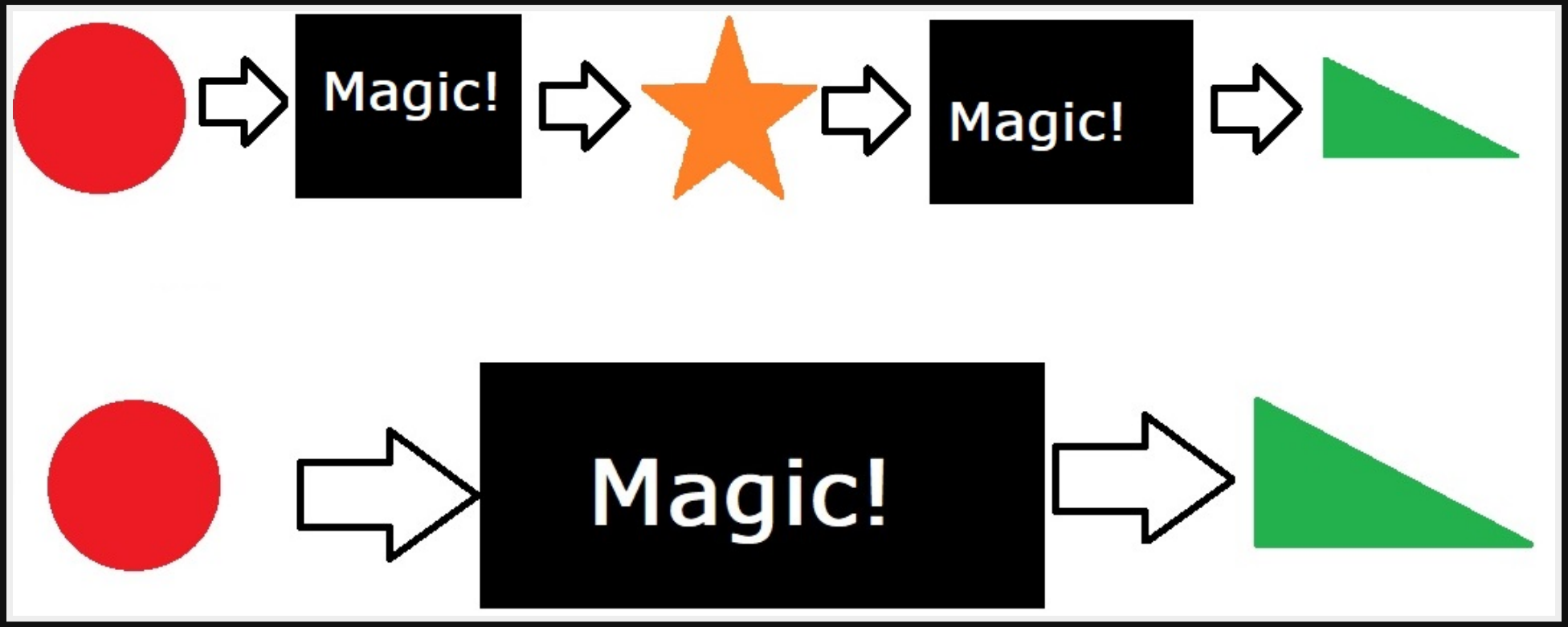
Dependency Order



Function Composition

- Compose multiple functions into one function
- Code reusability without verbosity


Magic!
Magic!


Magic!
Magic!
Magic!

```fsharp
let parseDateTime (dateTime: string) = System.DateTime.Parse(dateTime)
let getYear (date: System.DateTime) = date.Year

let date = parseDateTime "13-03-2018 12:00am"
let currentYear: int = getYear date

printfn "The current year is: %i" currentYear
// The current year is: 2018

let composedGetYear: string -> int = parseDateTime >> getYear
let yearFromComposed: int = composedGetYear "13-03-2018 12:00am"

printfn "The current year from composed function is: %i" yearFromCompose
// The current year from composed function is: 2018


```

Function Currying and Partial Function Application

- Creating new functions by not supplying all parameters
- F# curries all functions by default
- What does `x: int -> y: int -> int` mean

Curried function

```
1: let add x y =
2:     x + y
```

```
 1: let add x =
 2:     fun y ->
 3:         x + y
 4:
 5: let threeParams firstName middleName lastName =
 6:     printfn "Full name is: %O, %O, %O" firstName middleName lastName
 7:
 8: let threeParams firstName =
 9:     fun middleName ->
10:         fun lastName ->
11:             printfn "Full name is: %O, %O, %O" firstName middleName lastN
```

Partial Function Application

- Using curried functions
- Only supply some of the parameters

```
 1: let add x y =
 2:     x + y
 3:
 4: let curriedAdd = add 3
 5: printfn "Result is: %i" (curriedAdd 5)
 6: // Result is: 8
 7:
 8: let double = (*) 2
 9:
10: [1..10]
11: |> List.map double
12: // [2; 4; 6; 8; 10; 12; 14; 16; 18; 20]
```

Point-free programming

- What happens when you abuse currying and partial function application
- Avoid explicitly specifying parameters
- Use Higher-order functions everywhere

```
1: let sum list = List.reduce (+) list
2: let freeSum = List.reduce (+)
3:
4: let doubleAndIncrement x = x * 2 + 1
5: let freeDoubleAndIncrement = (*) 2 >> (+) 1
```

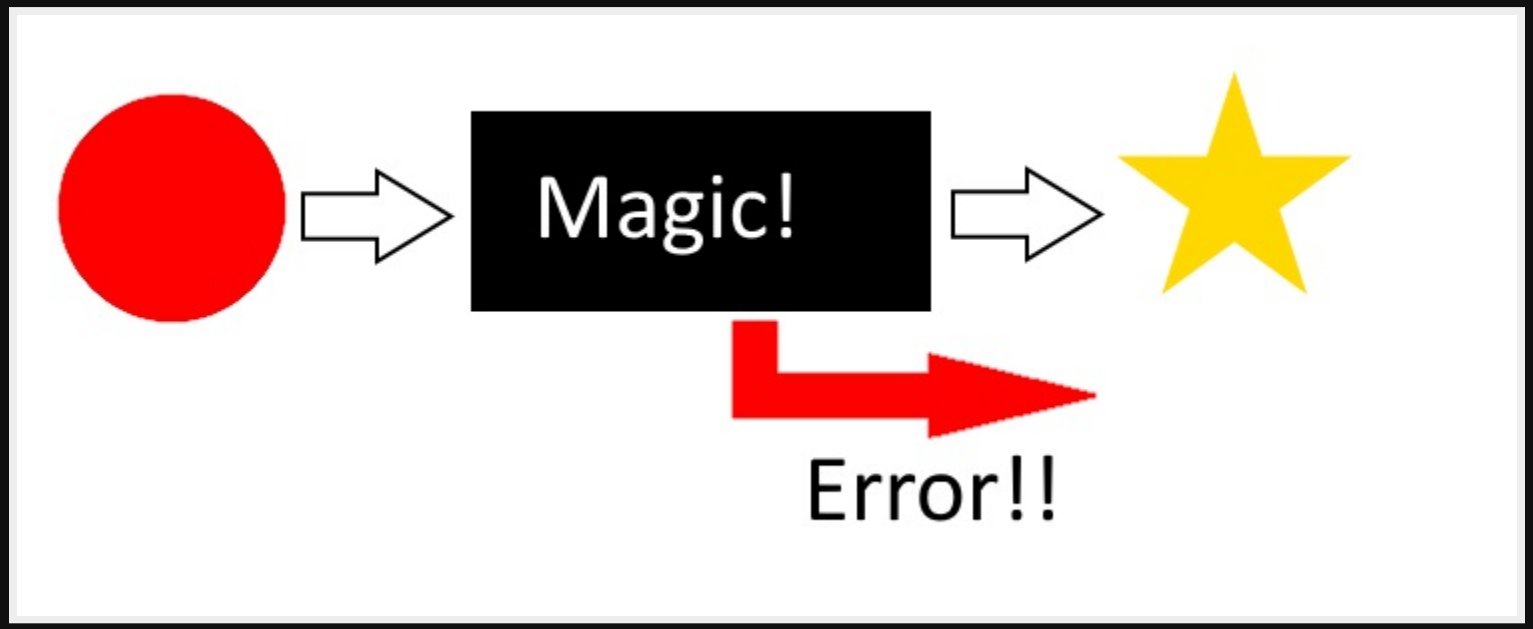
Demo

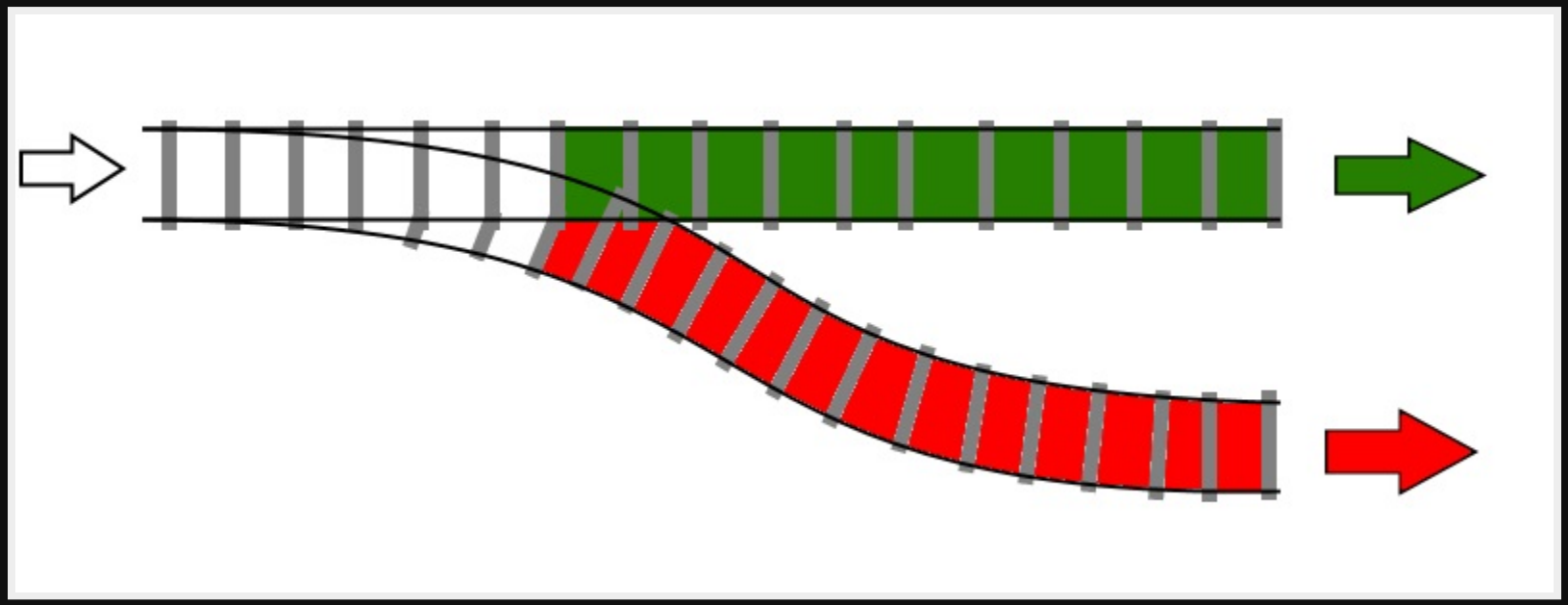
Railway Oriented Programming

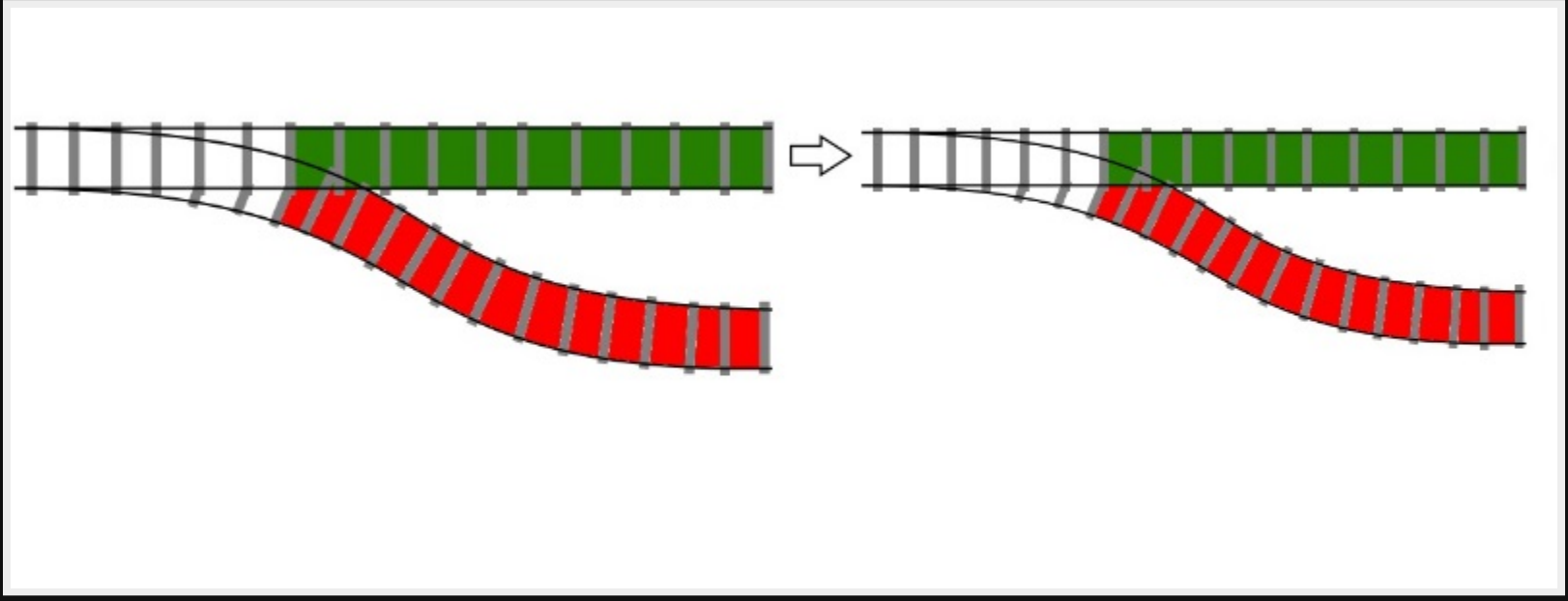
- Error handling through function composition
- Clean control flow
- Treat errors as first class citizens

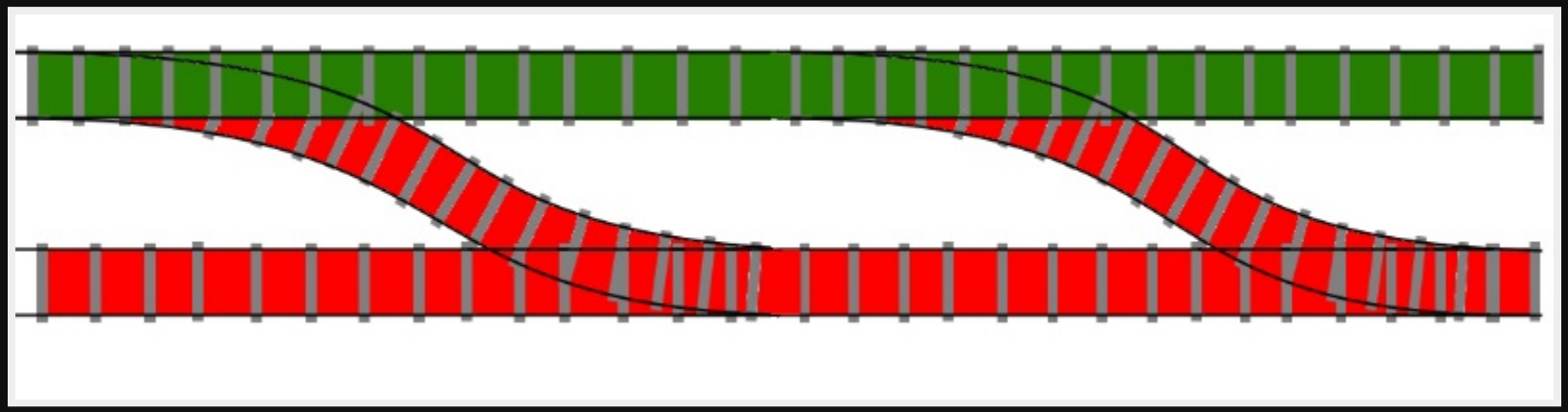
Problems with the Imperative approach

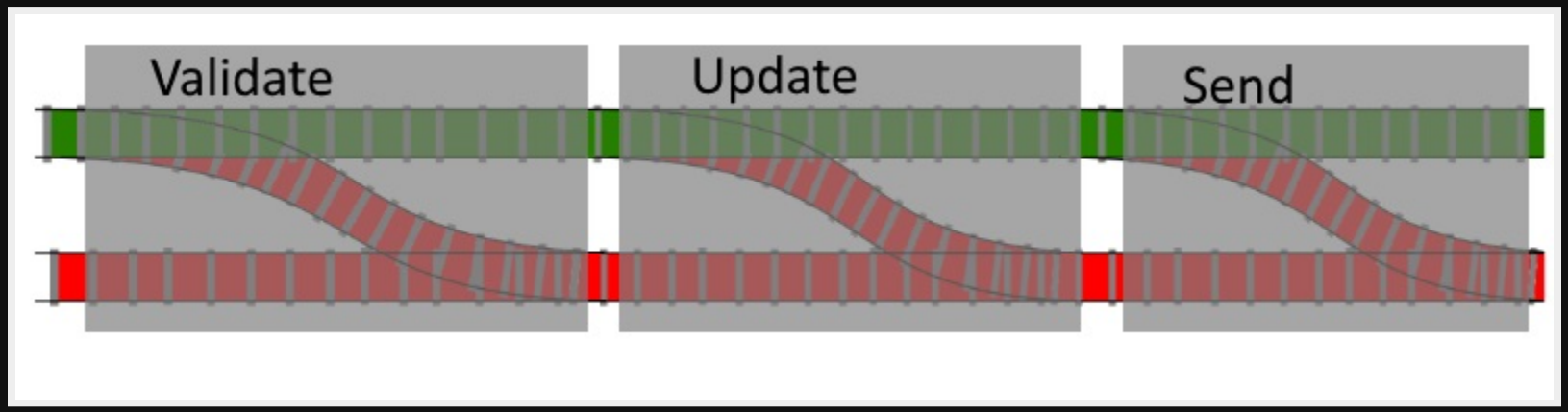
```
1: public string UpdateAndSend(string input)
2: {
3:     if (string.IsNullOrWhitespace(input))
4:         return "empty input"; // Or should we throw an Exception?
5:
6:     try {
7:         var updatedUser = UpdateUserInDatabase(input);
8:         return ConvertToJson(updatedUser);
9:     }
10:    catch (DatabaseException e) {
11:        logger.log(e.Message);
12:        return "Problem updating user in DB"; // Or should we rethrow?
13:    }
14: }
```


Magic!
Error!!








Validate
Update
Send

Demo

Monads

- A monad is just a monoid in the category of endofunctors
- Chainable wrapper around a data structure that performs an extra operation after each expression
- Semicolon at end of statement performs extra action

Monad in Detail

- A constructor that wraps a value: *the monadic value M*
- A bind function that accepts a function as its parameter
  - Applies this function to the internally wrapped value *M*
  - Returns the function's output wrapped as a monad
- A return function that simply unwraps the monadic value

F# Computation Expression

- NOT Monads
- Can be used to express monads

```
1: let result =
2:     async {
3:         let! (username: string) = getByIdAsync 1
4:         let! (replies: string list) = getRepliesByUsernameAsync username
5:
6:         let count = replies |> List.length
7:
8:         return (replies, count)
9:     } |> Async.RunSynchronously
```

Demo

Additional resources

- http://fsharp.org/learn.html
- https://fsharpforfunandprofit.com/
- http://www.tryfsharp.org/Learn/getting-started (Requires Silverlight...)
- https://kaeedo.github.com/IntroductionToFunctionalProgramming

Thank you!

- Questions?
- Feedback?