

# A primer to the world of Functional Programming with F#

- Kai Ito
- [https://www.xing.com/profile/Kai\\_Ito/](https://www.xing.com/profile/Kai_Ito/)
- <https://github.com/kaeedo/IntroductionToFunctionalProgramming>

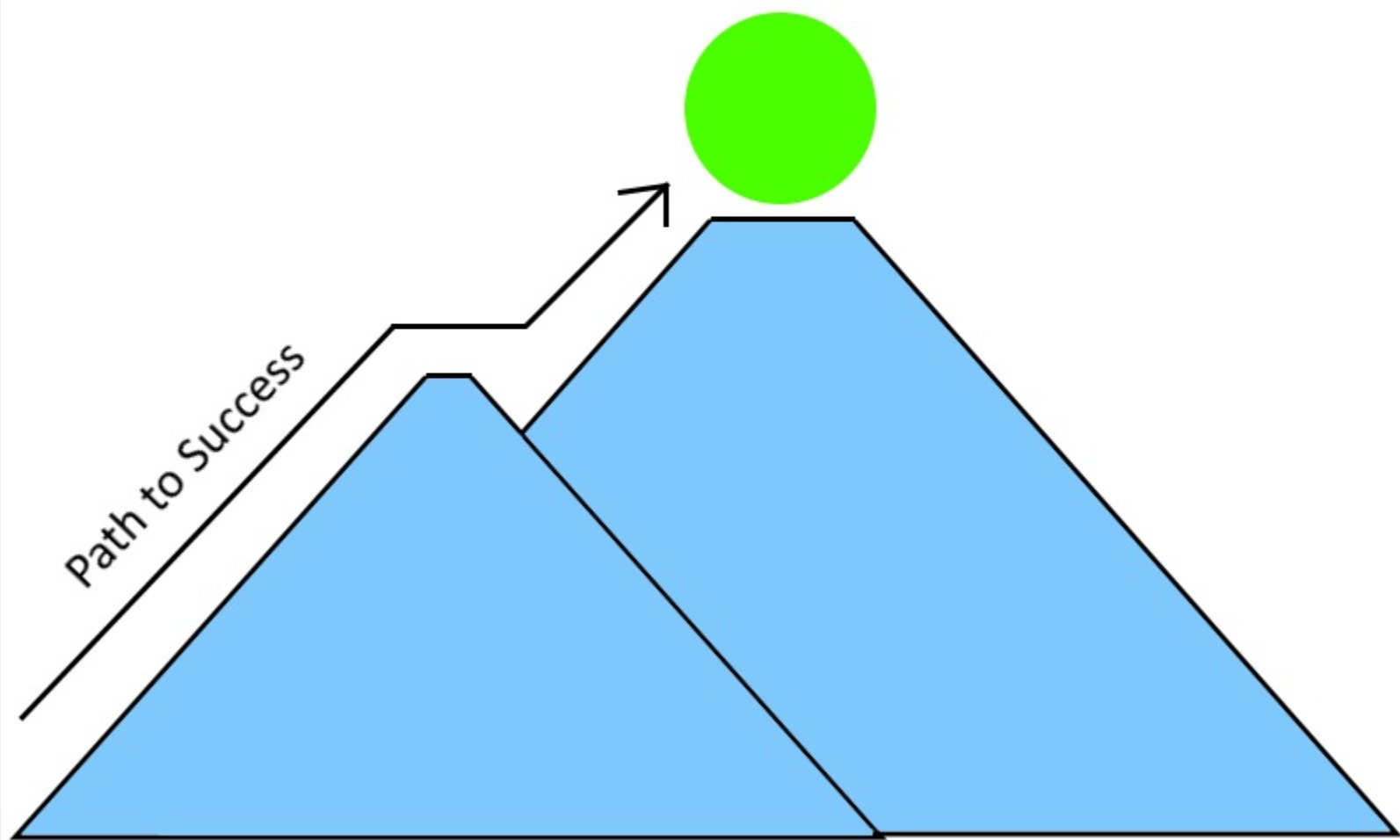
# What is Functional Programming

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

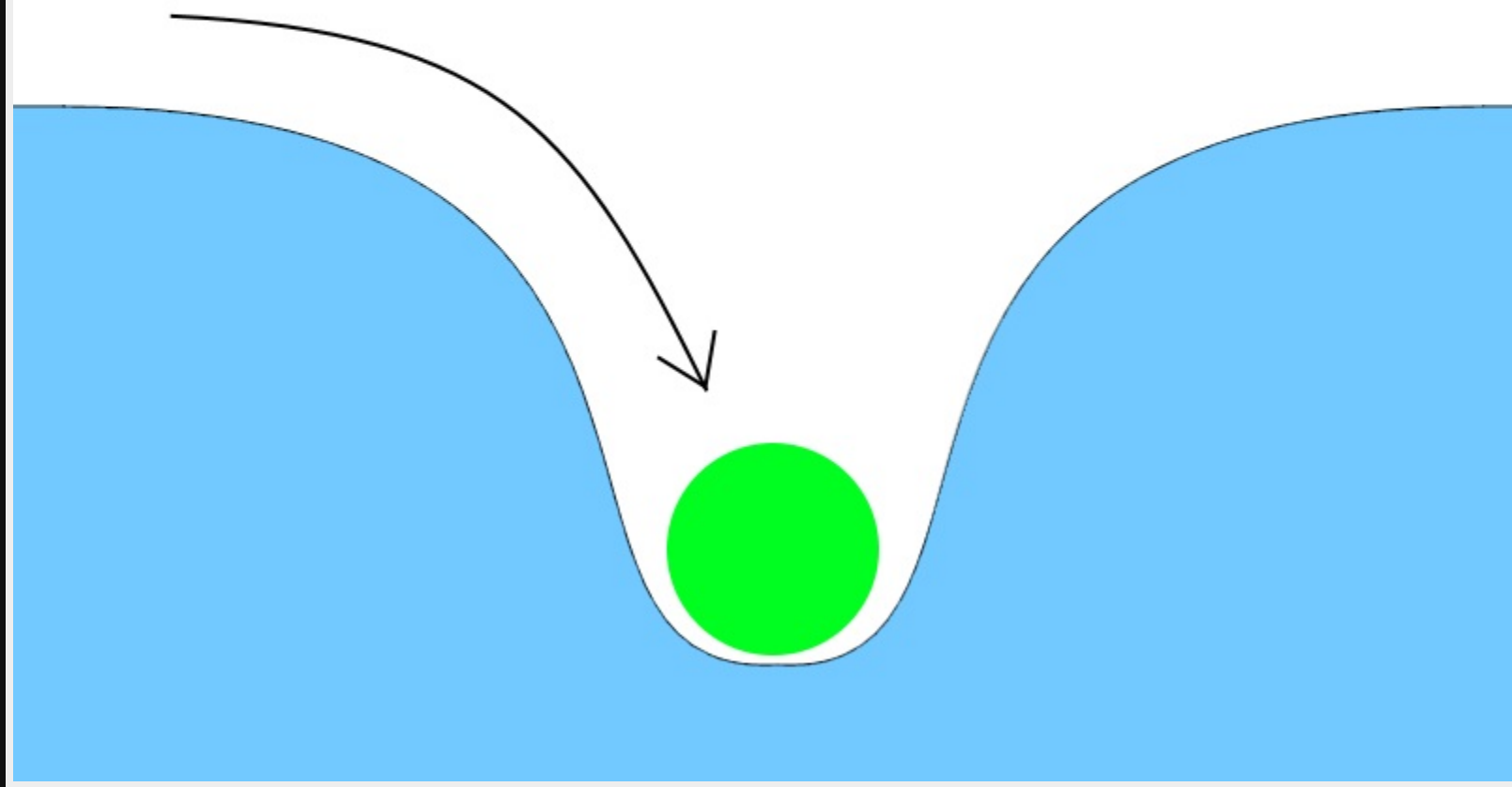
# Benefits of the Functional paradigm

- 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: let listOfInts = [ 1; 2; 3 ]
5: let listOfStrings =
6:   [ "www.zuehlke.com"
7:     "www.google.com"
8:     "www.microsoft.com" ]
9:
```

# 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

```
1: let printer f = printfn "Number is: %f" f
2:
3: [0.0..100.0]
4: |> List.filter (fun i -> i % 2.0 = 0.0)
5: |> List.map (fun i -> i ** 2.0)
6: |> List.iter printer
7:
8: let (|>) value fn =
9:     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: let kai = { FirstName = "Kai"; LastName = "Ito"; Age = 27 }
6: let cloneOfKai = { FirstName = "Kai"; LastName = "Ito"; Age = 27 }
7:
8: printfn "%b" (kai = cloneOfKai) // true
9:
10: let olderKai = { kai with Age = kai.Age + 1 }
11: printfn "%i" (olderKai.Age) // 28
12: printfn "%i" (kai.Age) // 27
13:
14:
```

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

# 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

```
1: type Option<'a> =
2: | Some of 'a
3: | None
4: let validInt: int option = Some 1
5: let invalidInt: int option = None
6:
7: let numbers = [ 1; 2; 3; 4; ]
8: let foundNumber = numbers |> List.tryFind (fun x -> x = 4)
9: let missingNumber = numbers |> List.tryFind (fun x -> x = 50)
10: printfn "The number is: %i" foundNumber
11: // Compile Error: Type mismatch: Expecting "int" but got "int option"
12:
13: let printInt input =
14:     match input with
15:     | Some i -> printfn "The number is: %i" i
16:     | None -> printfn "Didn't find number"
17:
18: printInt foundNumber // The number is: 4
19: printInt missingNumber // Didn't find number
20:
21:
```

# 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 = "0
7:
```

# F# Type System to the rescue

```
1: type ContactInformation =
2: | Email of string
3: | PhoneNumber of string
4: | EmailAndPhone of string * string
5: type SafeContactUser = { Username: string; Contact: ContactInformation }
6:
7: let email = Email "kai.ito@zuehlke.com"
8: let phoneNumber = PhoneNumber "089 555 1234"
9: let emailAndPhone = EmailAndPhone ("kai.ito@zuehlke.com", "089 555 1234")
10: let user1 = { Username = "kaiito1"; Contact = email }
11: let user2 = { Username = "kaiito2"; Contact = phoneNumber }
12: let user3 = { Username = "kaiito3"; Contact = emailAndPhone }
13: let user4 = { Username = "kaiito4"; Contact = null } // Compiler Error
14: let user5 = { Username = "kaiito5"; Contact = "someString" } // Compiler
15:
16:
17:
```

# Dependency Order

The screenshot shows the 'Cursed.Base' project in Visual Studio. The file explorer on the left lists the following structure:

- References
  - AssemblyInfo.fs
  - Common.fs
  - State.fs
  - ModpackController.fs
- Actors
  - ViewActor.fs
  - CacheActor.fs
- Models
  - NotifyPropertyChanged.fs
  - Modpack.fs
  - UpdateDialogController.fs
  - MainViewController.fs
- Views
  - MainView.fs
  - UpdateDialog.fs
  - MainForm.fs

Two red arrows point from text annotations to specific files:

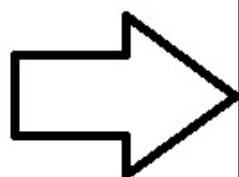
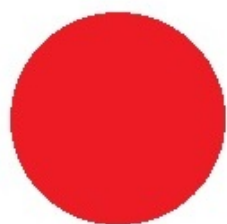
- An arrow points from the text "Code Here" to **State.fs** in the References folder.
- An arrow points from the text "Can't reference code here" to **ViewActor.fs** in the Actors folder.



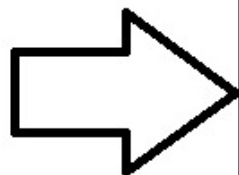
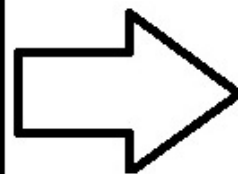
```
1: type Customer =  
2:     | T1 of CustomerObserver  
3:  
4: type CustomerObserver =  
5:     | T1 of Customer  
6:  
7: // Compiler error: The type 'Customer' is not defined  
8:  
9: type FirstType =  
10:     | T1 of SecondType  
11:  
12: and SecondType =  
13:     | T1 of FirstType  
14:
```

# Function Composition

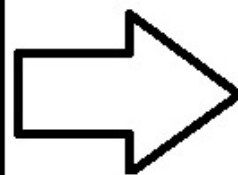
- Compose multiple functions into one function
- Code reusability without verbosity



**Magic!**



**Magic!**





```
1: let parseDateTime (dateTime: string) = System.DateTime.Parse(dateTime)
2: let getYear (date: System.DateTime) = date.Year
3:
4: let date = parseDateTime "13-03-2018 12:00am"
5: let currentYear: int = getYear date
6:
7: printfn "The current year is: %i" currentYear
8: // The current year is: 2018
9:
10: let composedGetYear: string -> int = parseDateTime >> getYear
11: let yearFromComposed: int = composedGetYear "13-03-2018 12:00am"
12:
13: printfn "The current year from composed function is: %i" yearFromComposed
14: // The current year from composed function is: 2018
15:
16:
```

# 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: let threeParams firstName middleName lastName =  
5:   printfn "Full name is: %O, %O, %O" firstName middleName lastName  
6:  
7: let threeParams firstName =  
8:   fun middleName ->  
9:     fun lastName ->  
10:       printfn "Full name is: %O, %O, %O" firstName middleName lastName  
11:
```

# 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
```

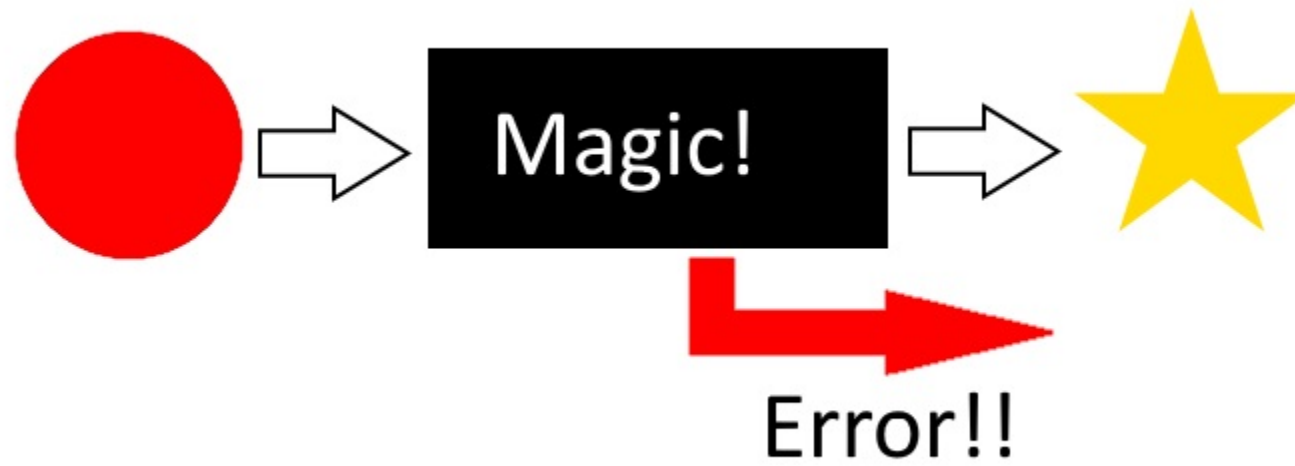
**Demo**

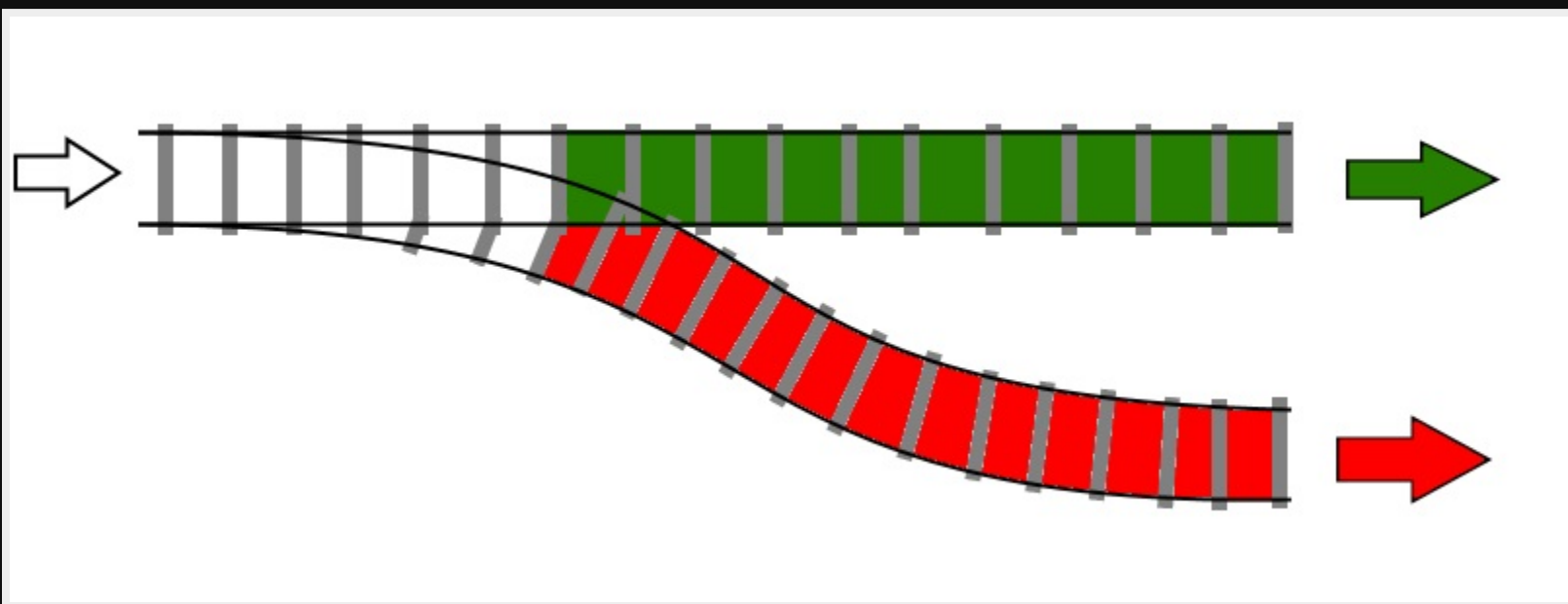
# Railway Oriented Programming

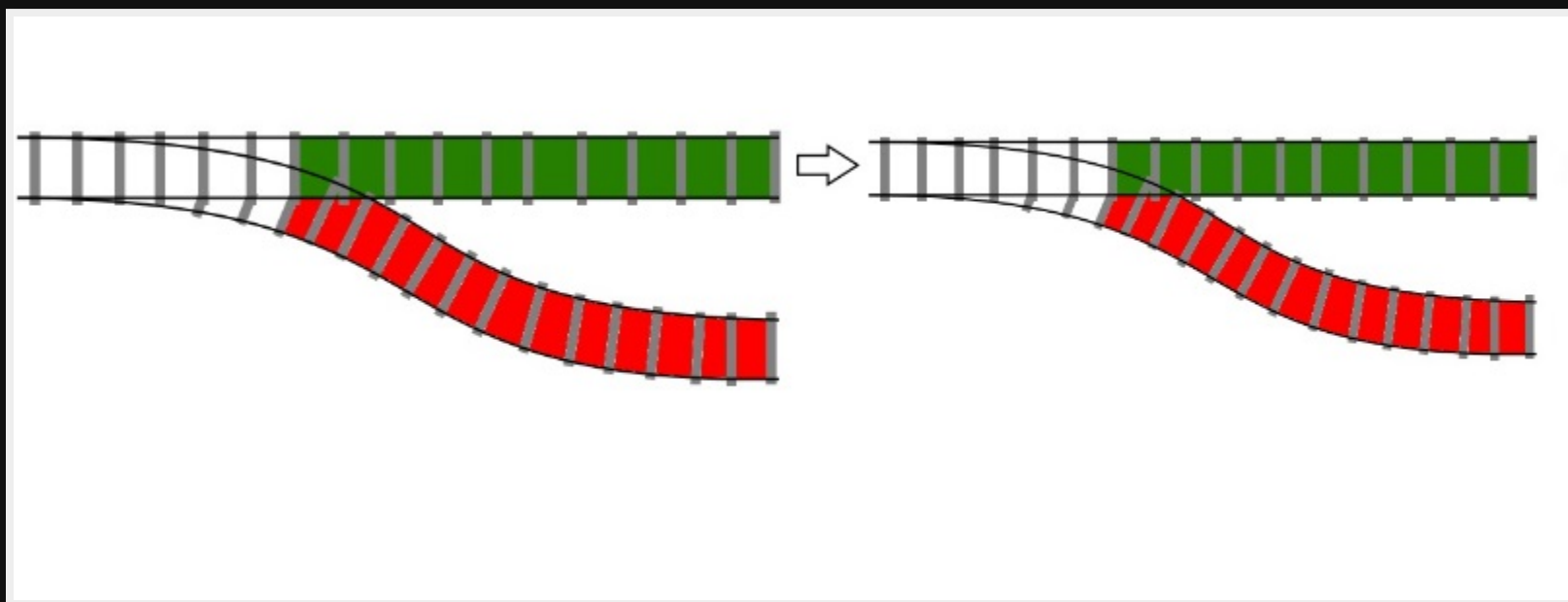
- Error handling through function composition
- Clean control flow
- Treat errors as first class citizens

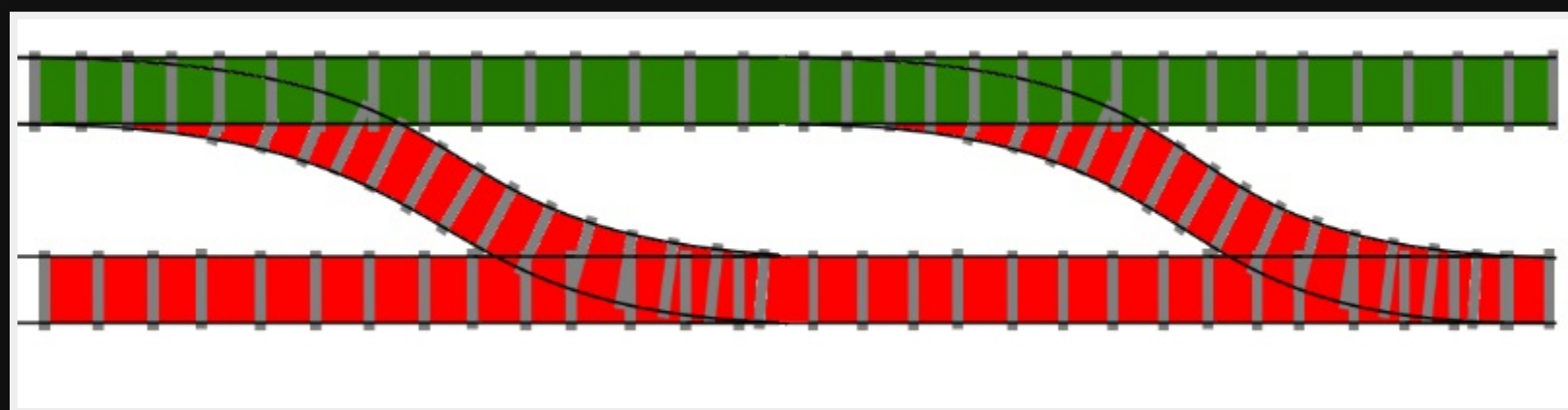
# Problems with the Imperative approach

```
1: public string UpdateAndSend(string input)
2: {
3:     if (string.IsNullOrEmpty(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:        return "Problem updating user in DB"; // Or should we rethrow?
12:    }
13:
14:    return "Failed to update and send"; // Or should we throw an Exception?
15: }
```

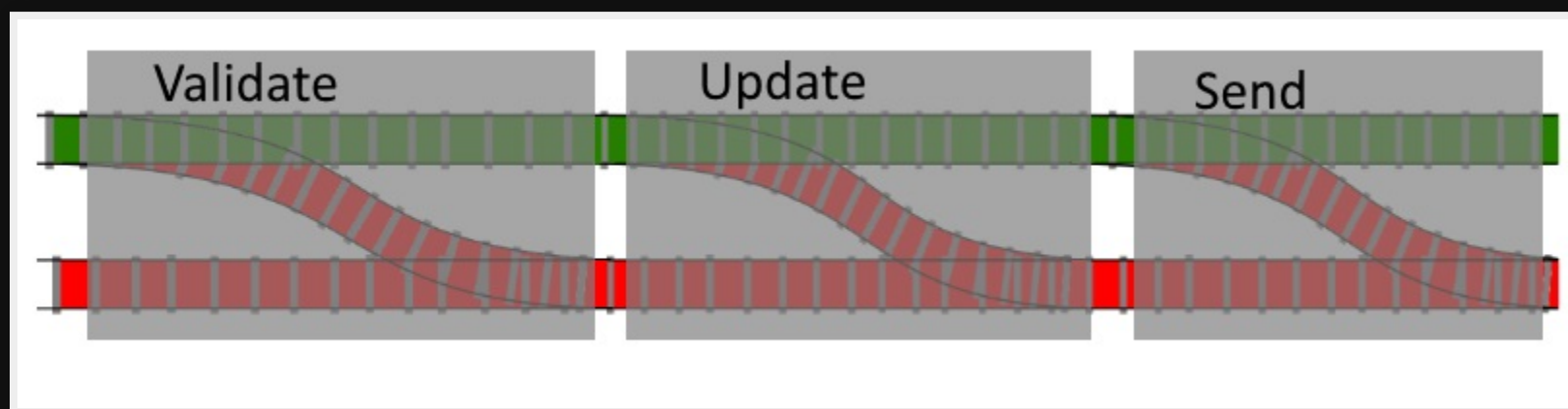












**Demo**

# 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) = getIdAsync 1  
4:         let! (replies: string list) = getRepliesByUsernameAsync username  
5:         let count = replies |> List.length  
6:         return (replies, count)  
7:     } |> Async.RunSynchronously  
8:  
9:
```

**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?