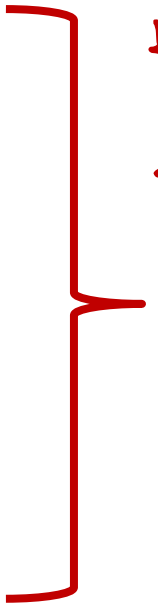
Domain Modelling with the F# type system

fsharpforfunandprofit.com/ddd

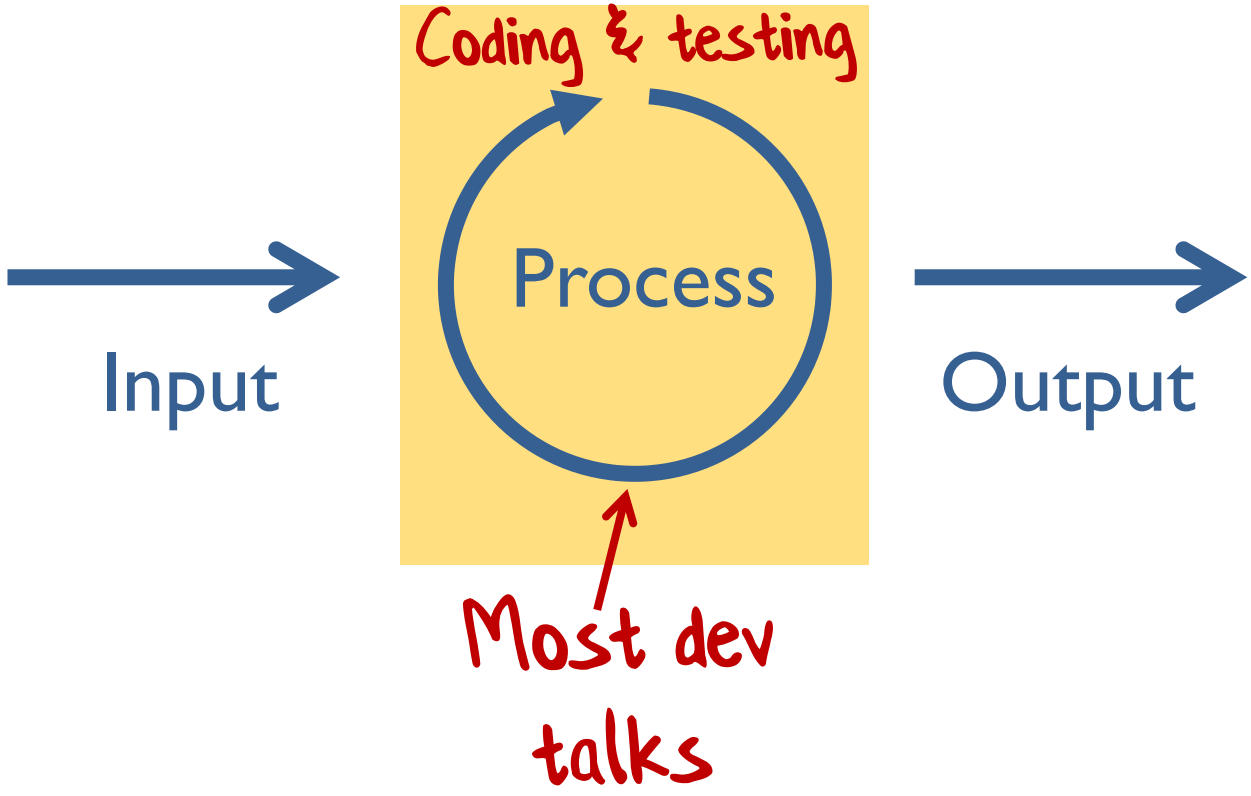
```
type Contact = {

   FirstName: string
   MiddleInitial: string
   LastName: string

   EmailAddress: string
   IsEmailVerified: bool
   }      // true if ownership of
          // email address is confirmed
```



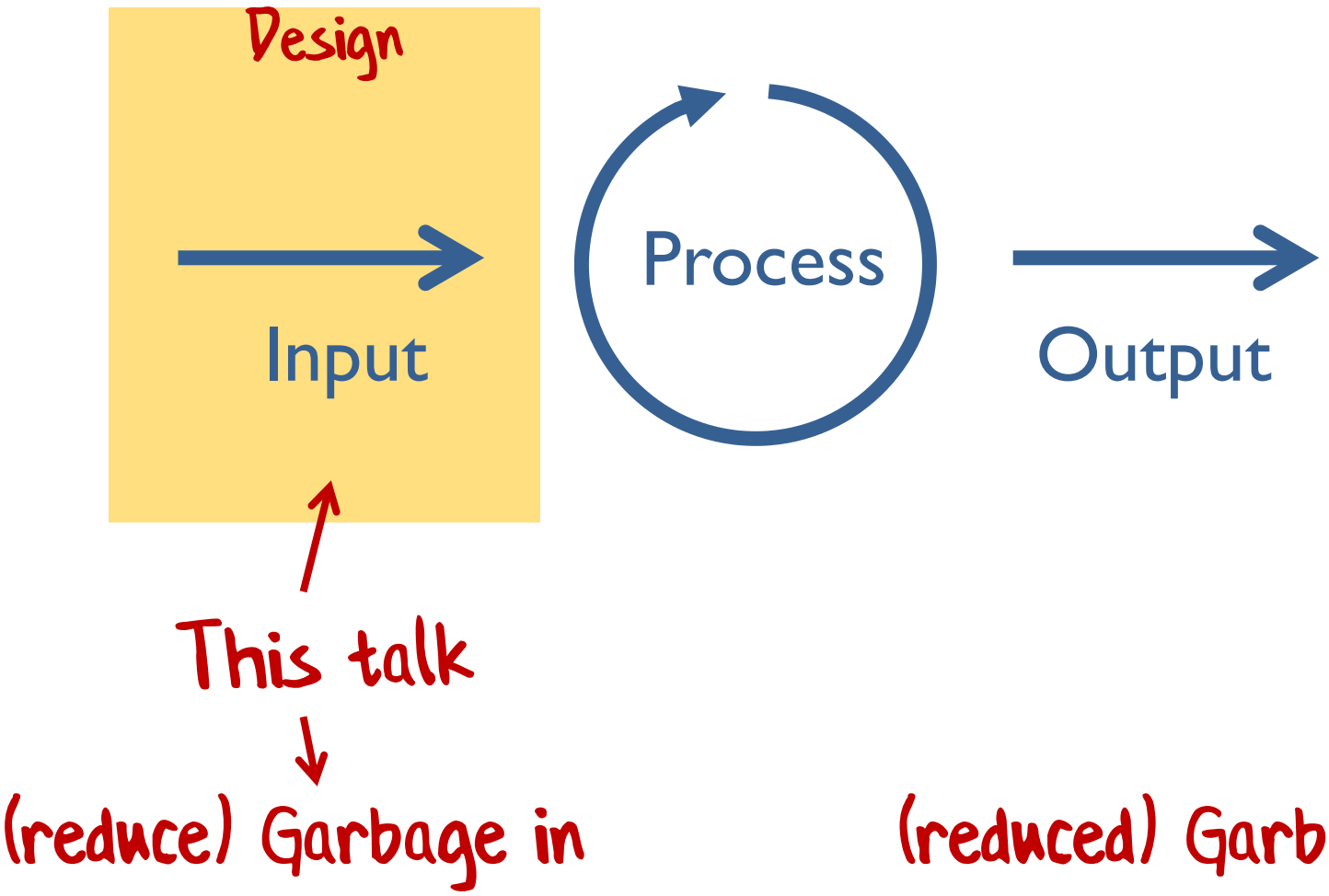
How many things are wrong with this design?

The software development process

The software development process



The software development process



```
type Contact = {

   FirstName: string
   MiddleInitial: string
   LastName: string

   EmailAddress: string
   IsEmailVerified: bool
   }
```

Which values are optional?

```
type Contact = {
    FirstName: string
    MiddleInitial: string
    LastName: string

    EmailAddress: string
    IsEmailVerified: bool
    }
```

Must not be more than 50 chars

What are the constraints?

```
type Contact = {
    FirstName: string
    MiddleInitial: string
    LastName: string

    EmailAddress: string
    IsEmailVerified: bool
    }
```

Must be updated as a group

Which fields are linked?

```
type Contact = {

   FirstName: string
   MiddleInitial: string
   LastName: string

   EmailAddress: string
   IsEmailVerified: bool
   }
```

Must be reset if email is changed

What is the domain logic?

```
type Contact = {

   FirstName: string
   MiddleInitial: string
   LastName: string

   EmailAddress: string
   IsEmailVerified: bool
   }
```

Which values are optional?

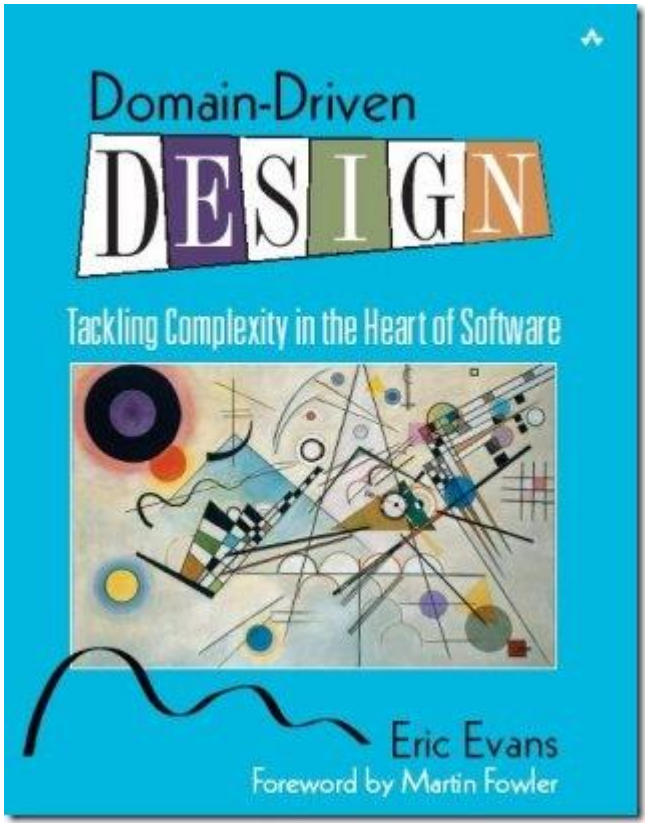
What are the constraints?

Which fields are linked?

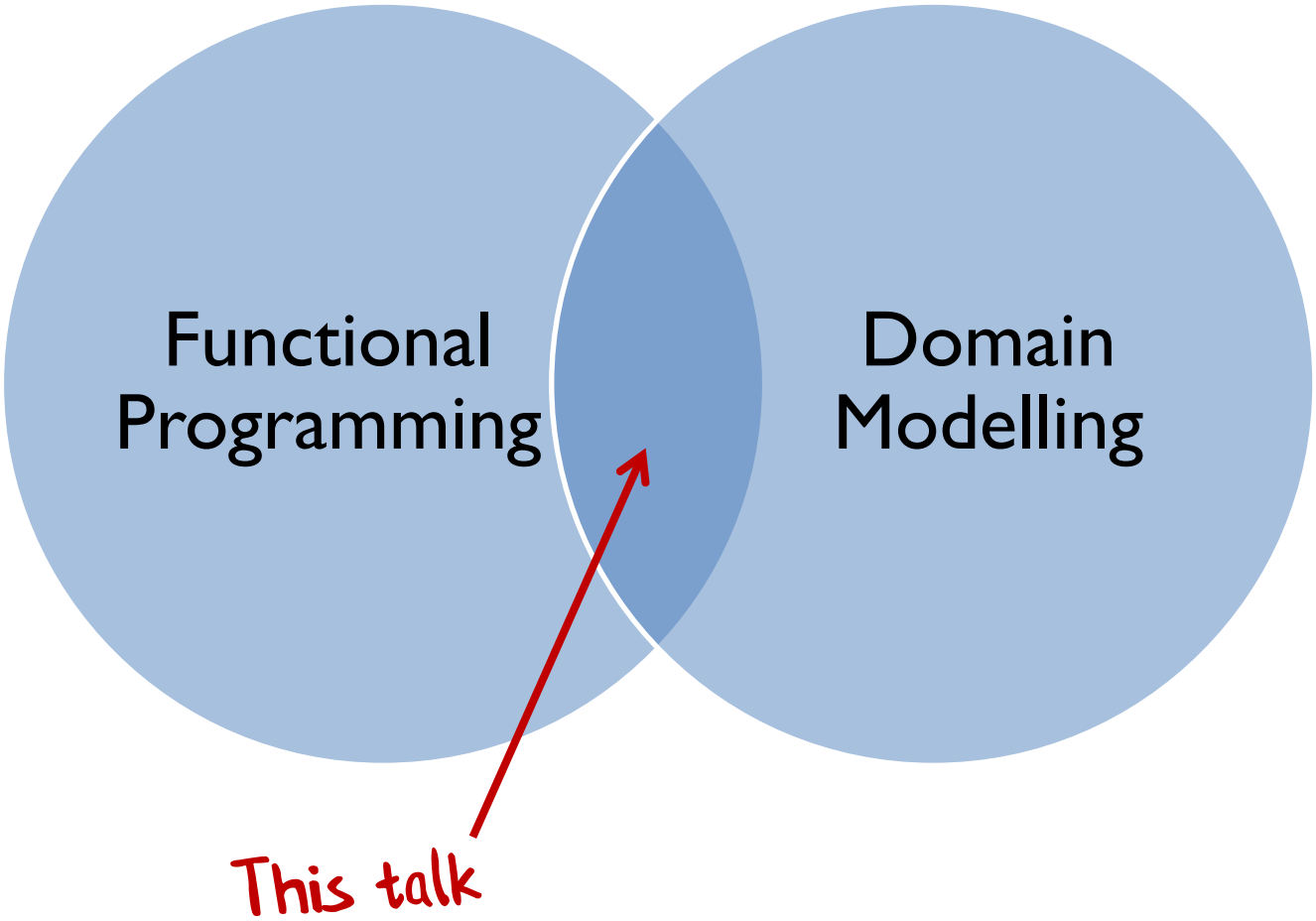
Any domain logic?

F# can help with all these questions!

What is DDD?



"Focus on the domain and domain logic rather than technology"
-- Eric Evans

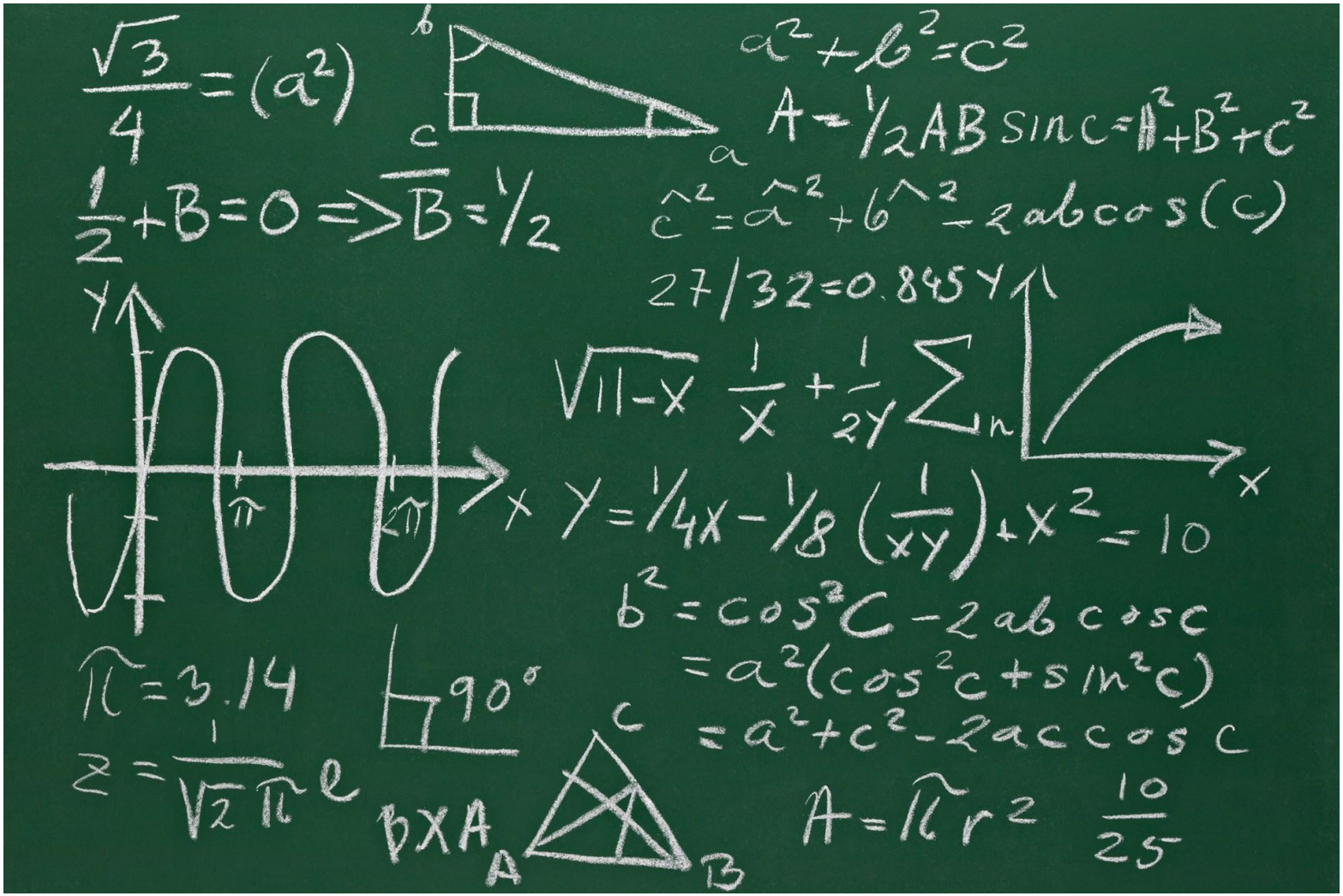

Functional
Programming
Domain
Modelling
This talk

UNDERSTANDING FUNCTIONAL PROGRAMMING

A non-technical overview

I've heard that...

Functional programming is...



I've heard that...

Functional programming is...

... good for mathematical and scientific tasks

... good for complicated algorithms

... *really* good for parallel processing

... but you need a PhD in computer science ☹

All true...

So not true...

Functional programming is really good for...

Boring

Line Of Business

Applications

(BLOBAs)

Must haves for BLOBA development...

- Express requirements clearly

F# is concise!
Easy to communicate.

- Rapid development cycle

An interactive environment and many conveniences to make teams highly productive

- High quality deliverables

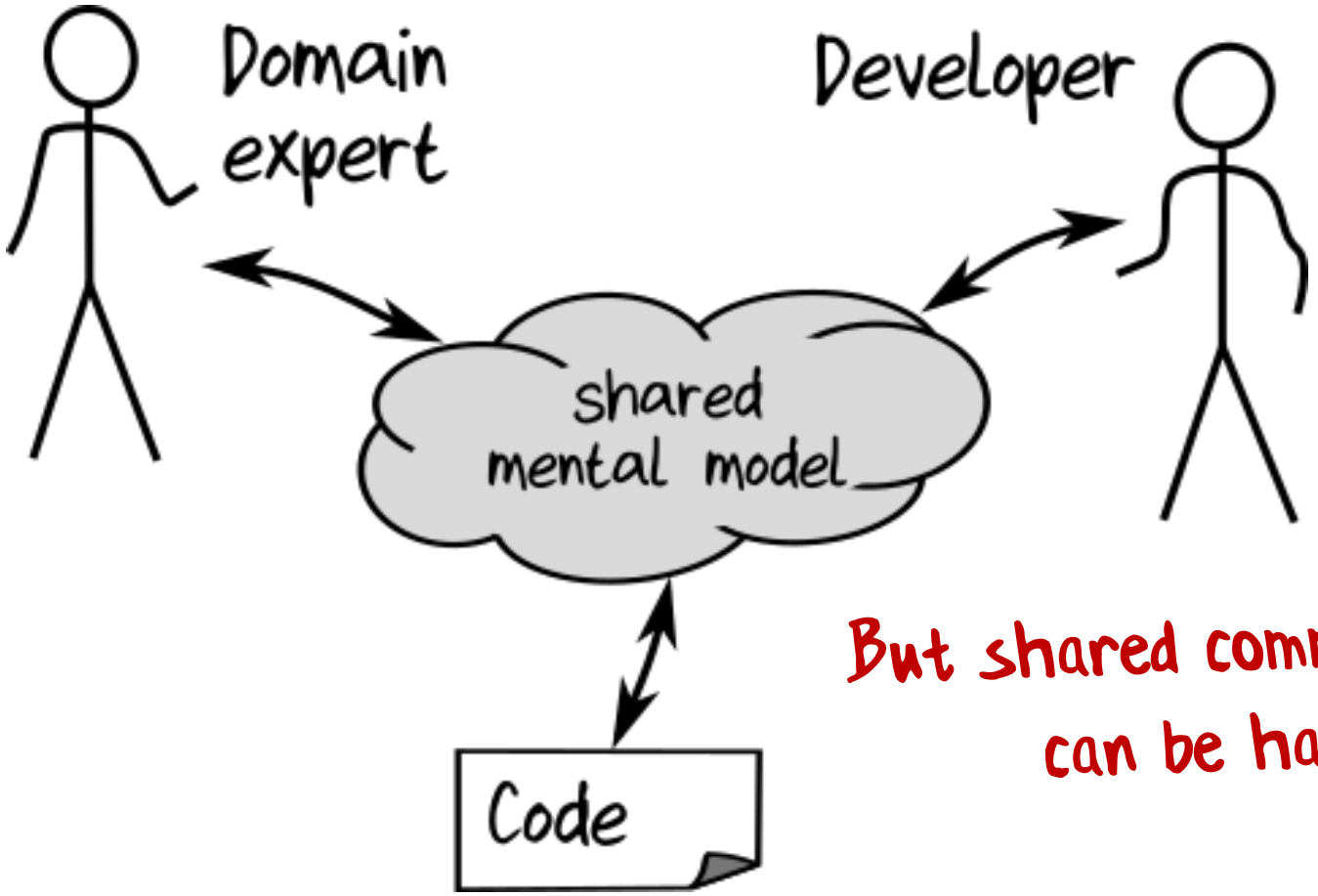
type system ensures correctness

- Fun

"fun" is a keyword in F#

F# for Domain Driven Design

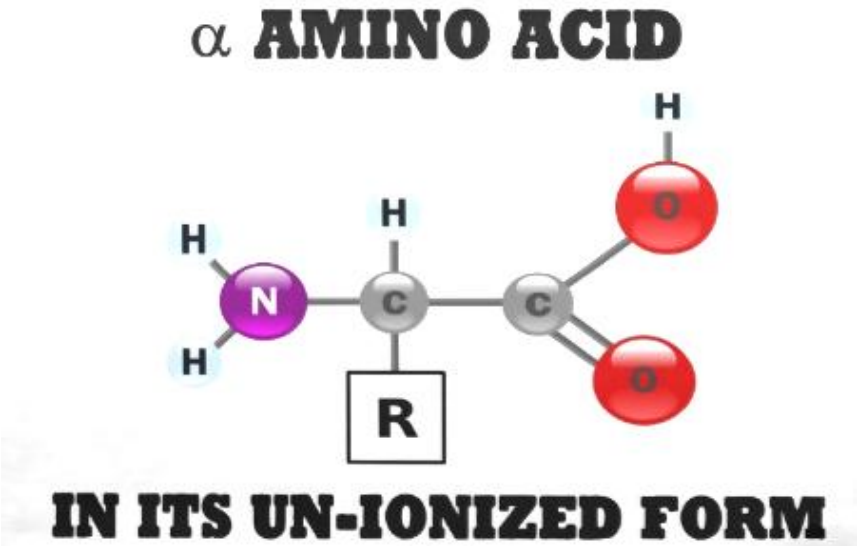
Communicating a domain model

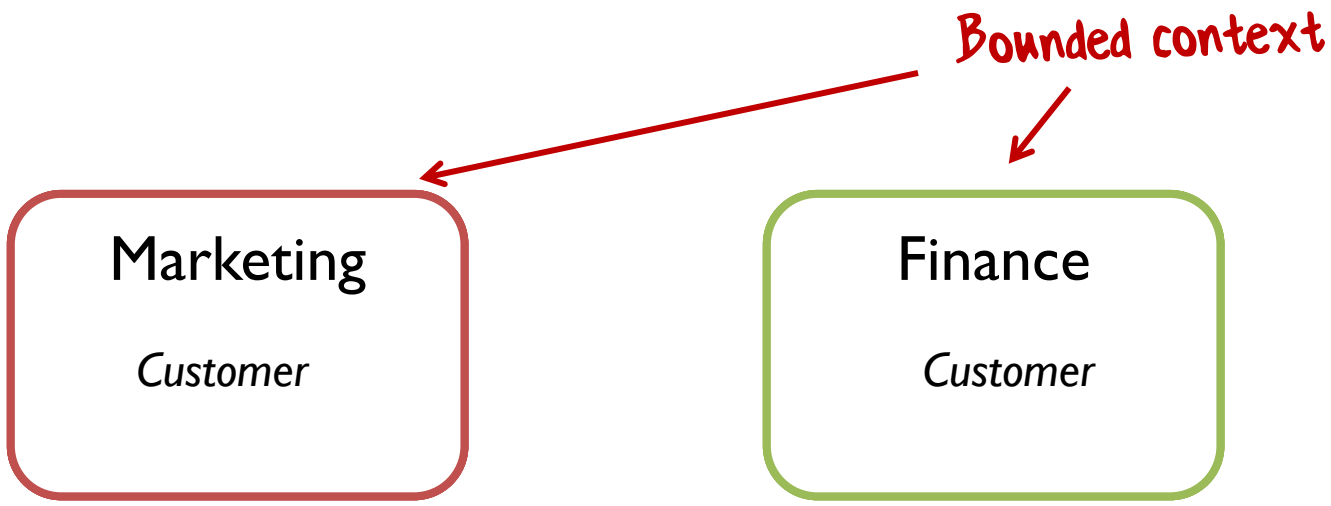


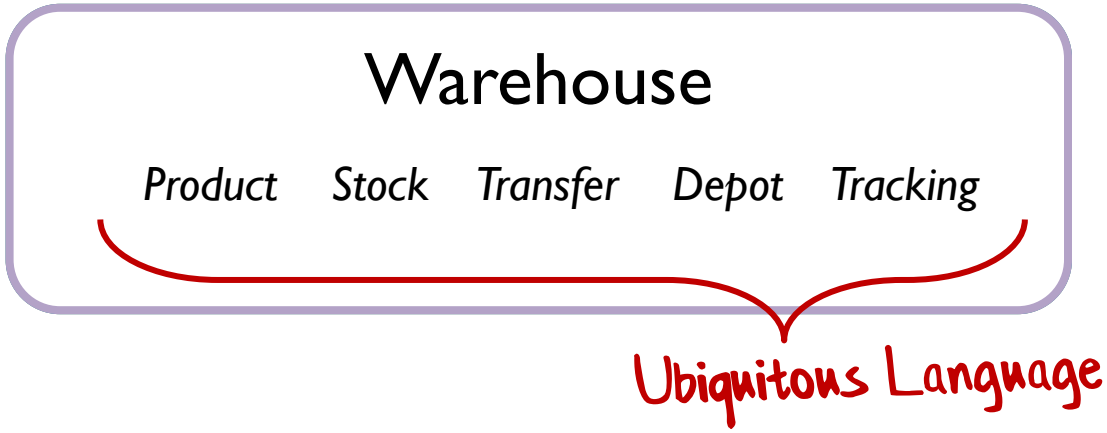
But shared communication can be hard!

U-N-I-O-N-I-Z-E


BETTER
MEDICAL
SCHOOLS
GIFT PACKAGES


α AMINO ACID
H
H
H
N
C
C
O
O
R
IN ITS UN-IONIZED FORM


Bounded context
Marketing
Customer
Finance
Customer


Warehouse
Product
Stock
Transfer
Depot
Tracking
Ubiquitous Language

Bounded context

module **CardGame** =

Ubiquitous language

```
type Suit = Club | Diamond | Spade | Heart

type Rank = Two | Three | Four | Five | Six | Seven | Eight
          | Nine | Ten | Jack | Queen | King | Ace

type Card = Suit * Rank

type Hand = Card list
type Deck = Card list

type Player = {Name:string; Hand:Hand}
type Game = {Deck:Deck; Players: Player list}

type Deal = Deck -> (Deck * Card)

type PickupCard = (Hand * Card) -> Hand
```

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
              | Nine | Ten | Jack | Queen | King | Ace

    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = {Name:string; Hand:Hand}
    type Game = {Deck:Deck; Players: Player list}

    type Deal = Deck -> (Deck * Card)

    type PickupCard = (Hand * Card) -> Hand
```

'|' means a choice -- pick one from the list

'*' means a pair. Choose one from each type

list type is built in

X -> Y means a function
- input of type X
- output of type Y

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
              | Nine | Ten | Jack | Queen | King | Ace

    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = {Name:string; Hand:Hand}
    type Game = {Deck:Deck; Players: Player list}

    type Deal = Deck -> (Deck * Card)

    type PickupCard = (Hand * Card) -> Hand
```

Do you think this is a reasonable amount of code to write for this domain?

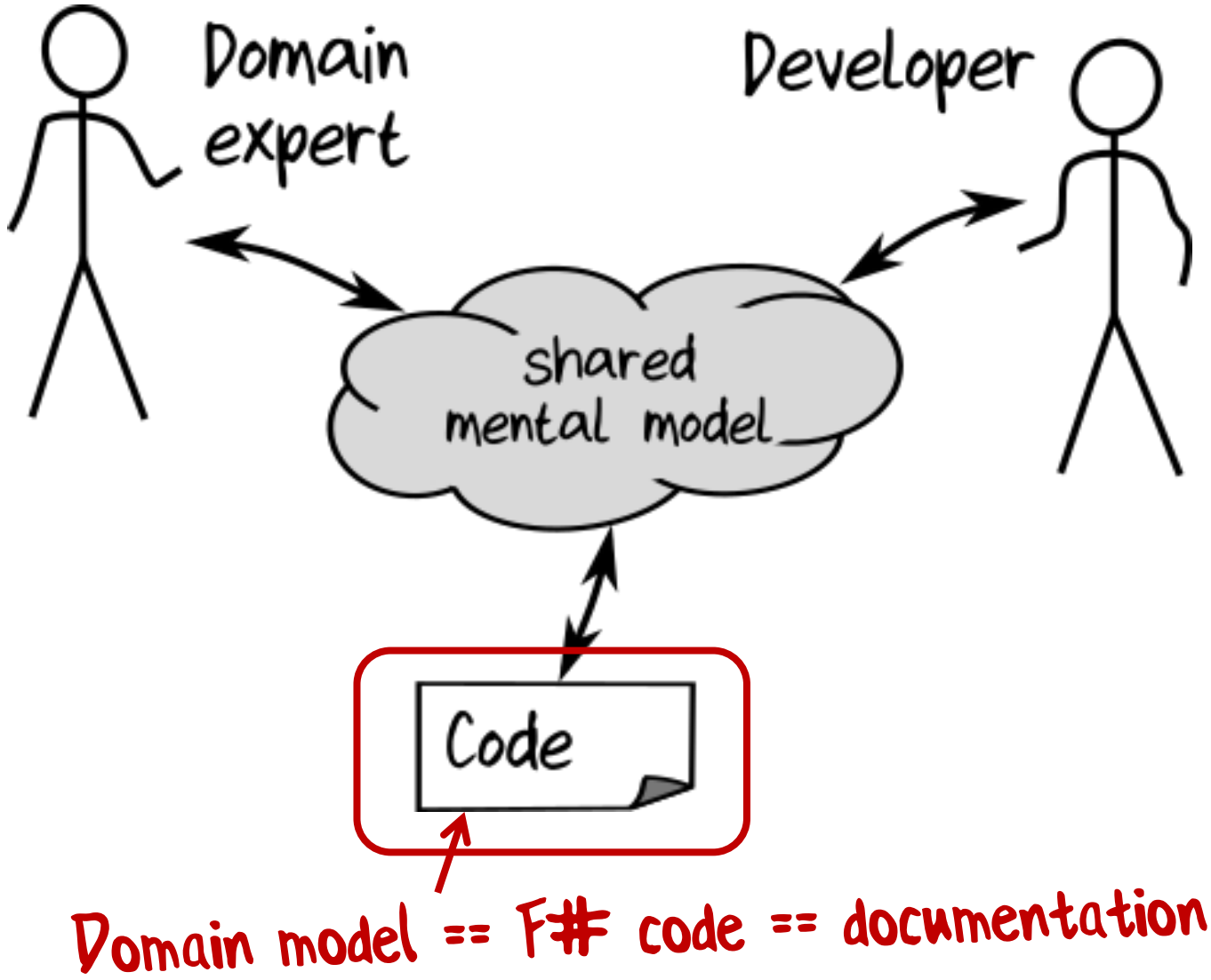
Do you think a non programmer could understand this?

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
              | Nine | Ten | Jack | Queen | King | Ace

    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = {Name:string; Hand:Hand}
    type Game = {Deck:Deck; Players: Player list}

    type Deal = Deck -> (Deck * Card)

    type PickupCard = (Hand * Card) -> Hand
```

"persistence ignorance"

"The design is the code, and the code is the design."

This is not pseudocode – this is executable code!


Domain expert
Developer
shared mental model
Code
Domain model == F# code == documentation

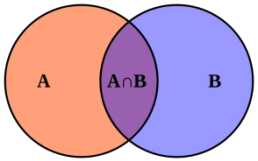

WE DON'T NEED NO STINKING
UML DIAGRAMS

Functional programming in 3 easy steps

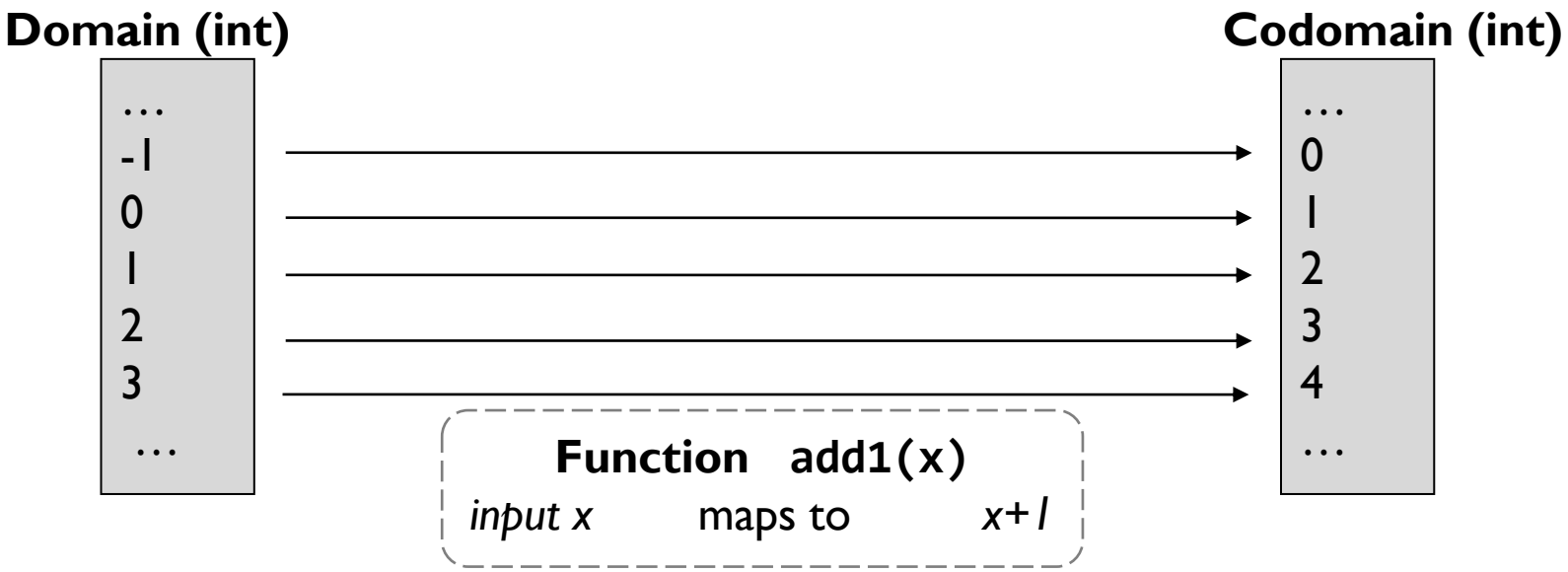
Functions

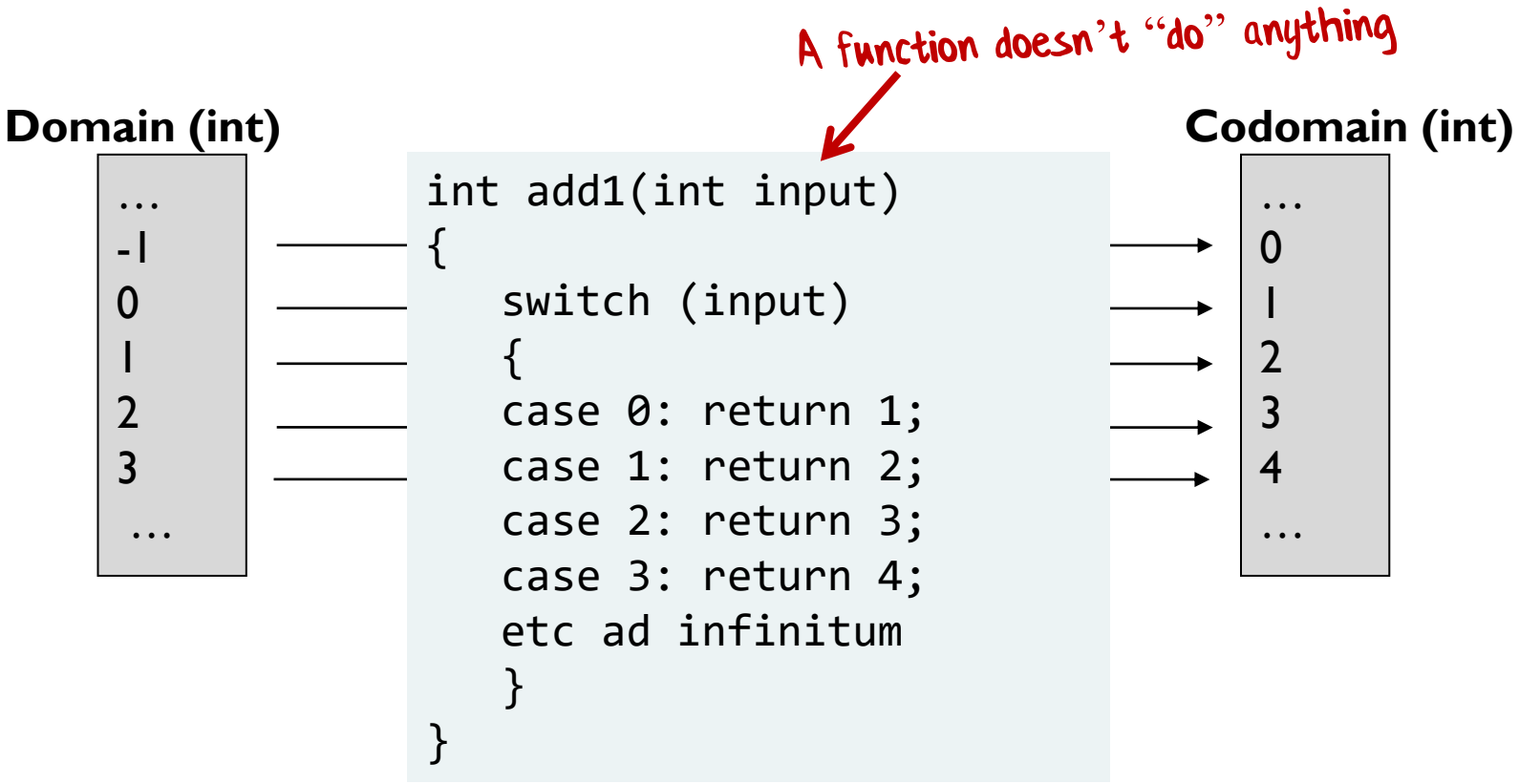
Composition

Types

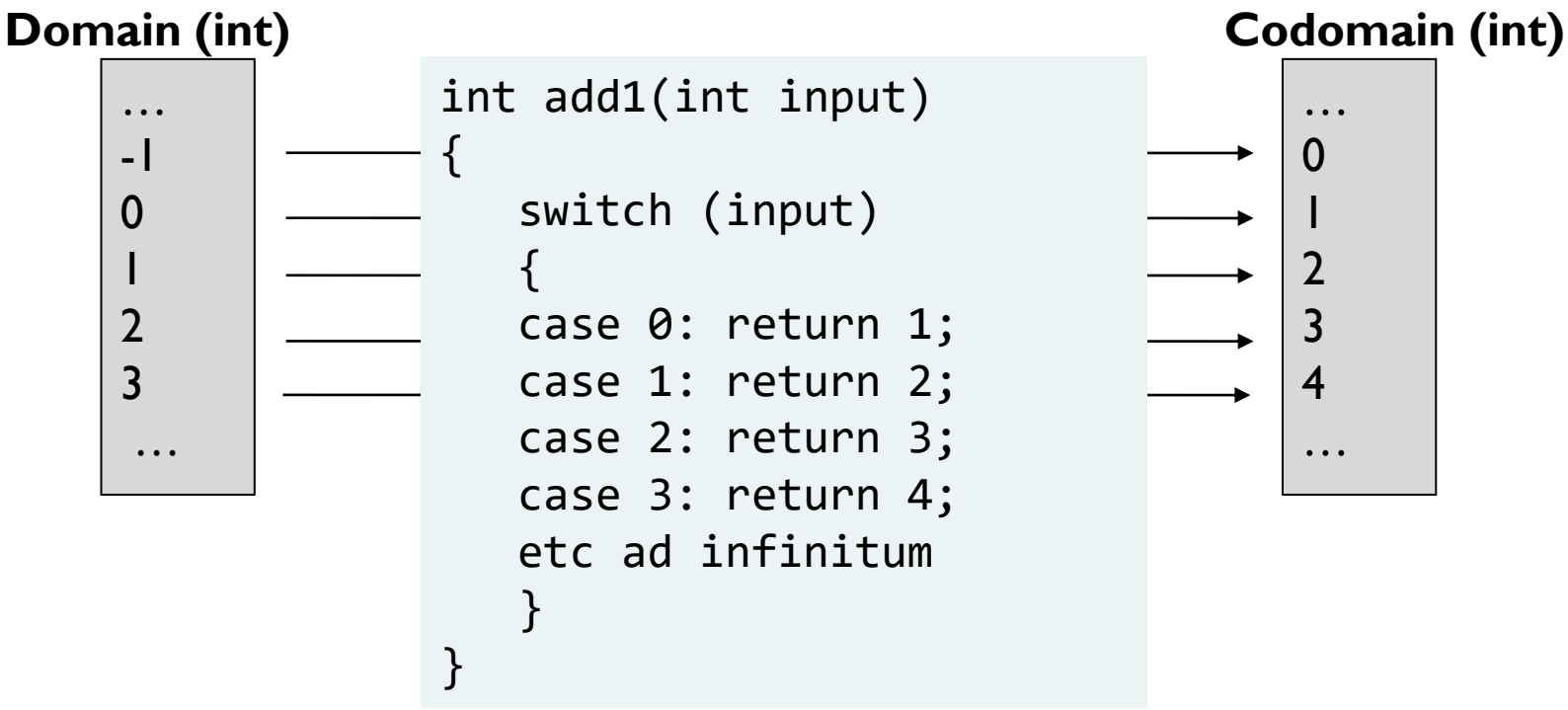


What are mathematical functions?


Domain (int)
…
-1
0
1
2
3
…
Codomain (int)
…
0
1
2
3
4
…
Function add1(x)
input x maps to x+1

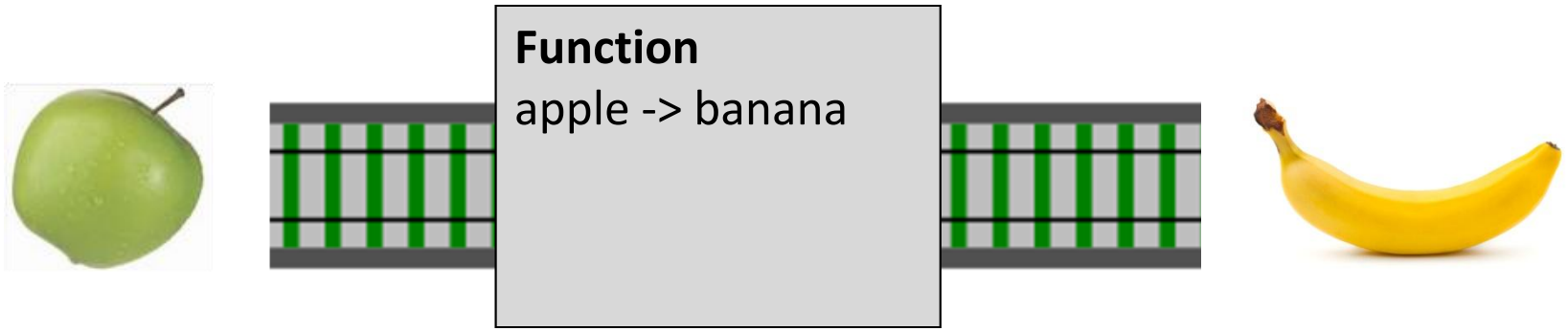


- Input and output values already exist
- A function is not a calculation, just a mapping
- Input and output values are unchanged (immutable)



- A (mathematical) function always gives the same output value for a given input value
- A (mathematical) function has no side effects

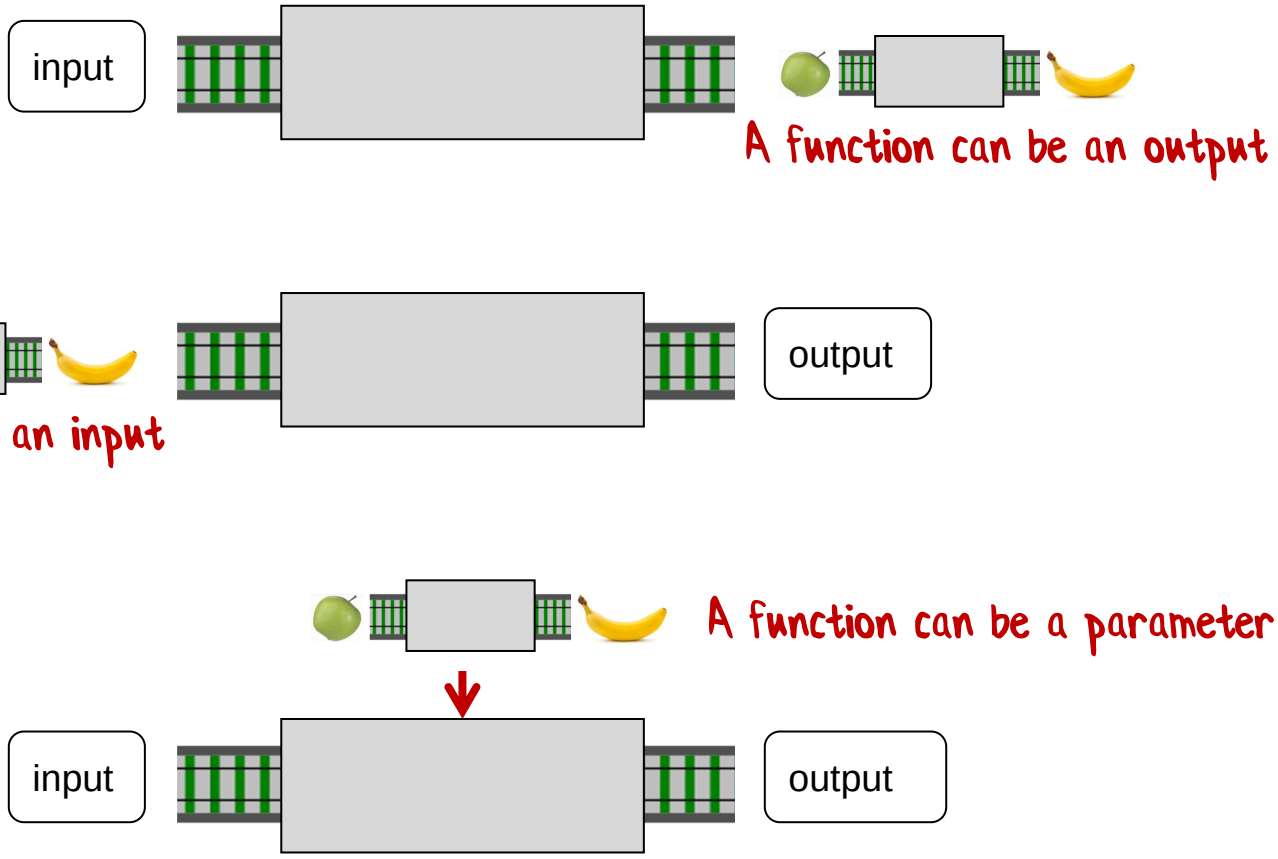
What are functions in F#?



A function is a thing which transforms inputs to outputs

A function is a standalone thing, not attached to a class

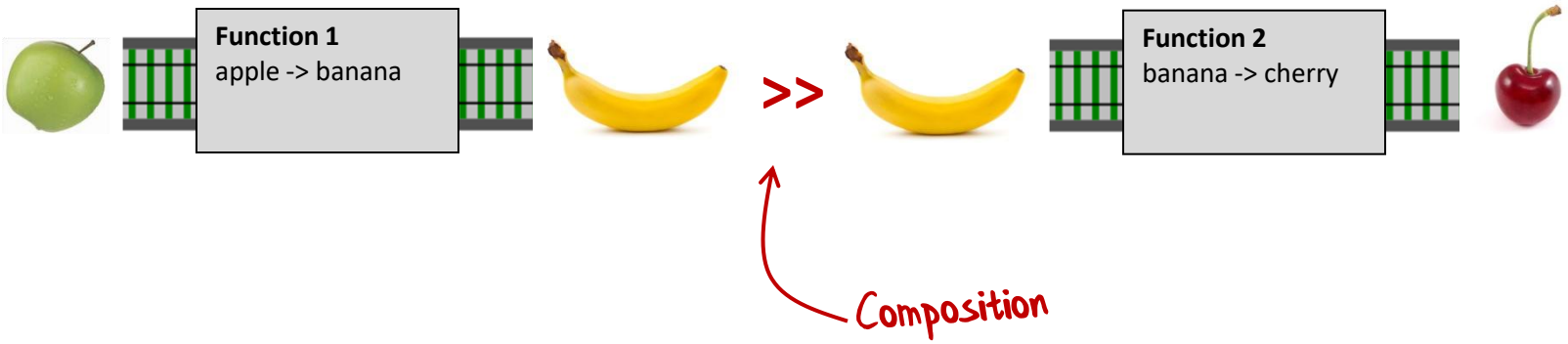
It can be used for inputs and outputs of other functions

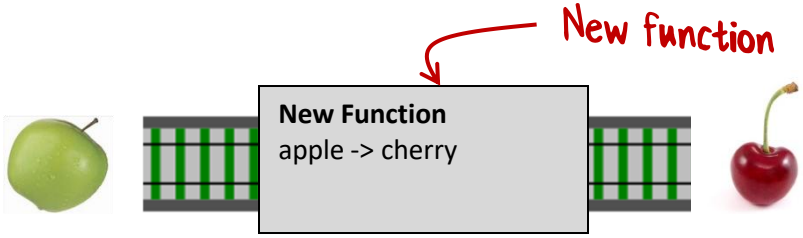


from a simple foundation => build complex systems

What is composition?


Function 1
apple -> banana
Function 2
banana -> cherry


Function 1
apple -> banana
>>
Function 2
banana -> cherry
Composition

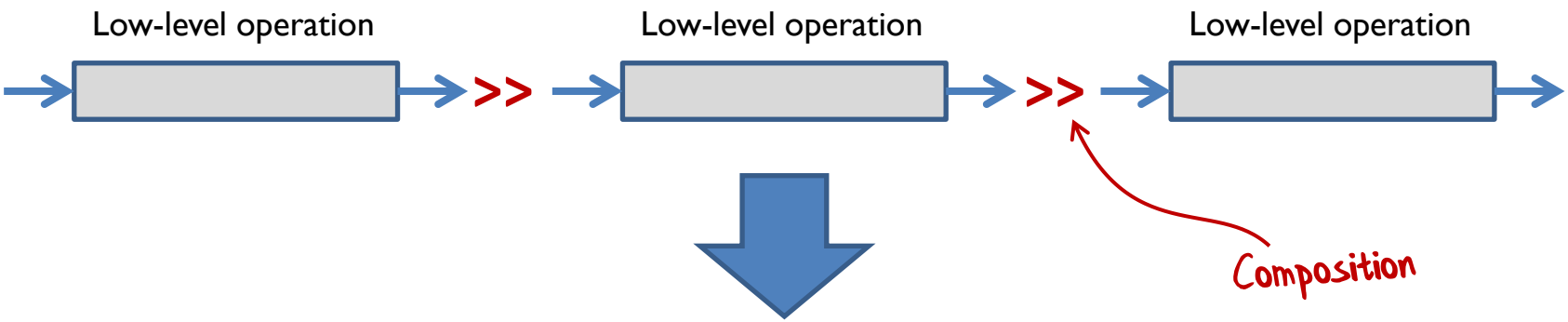


Can't tell it was built from smaller functions!

Where did the banana go?

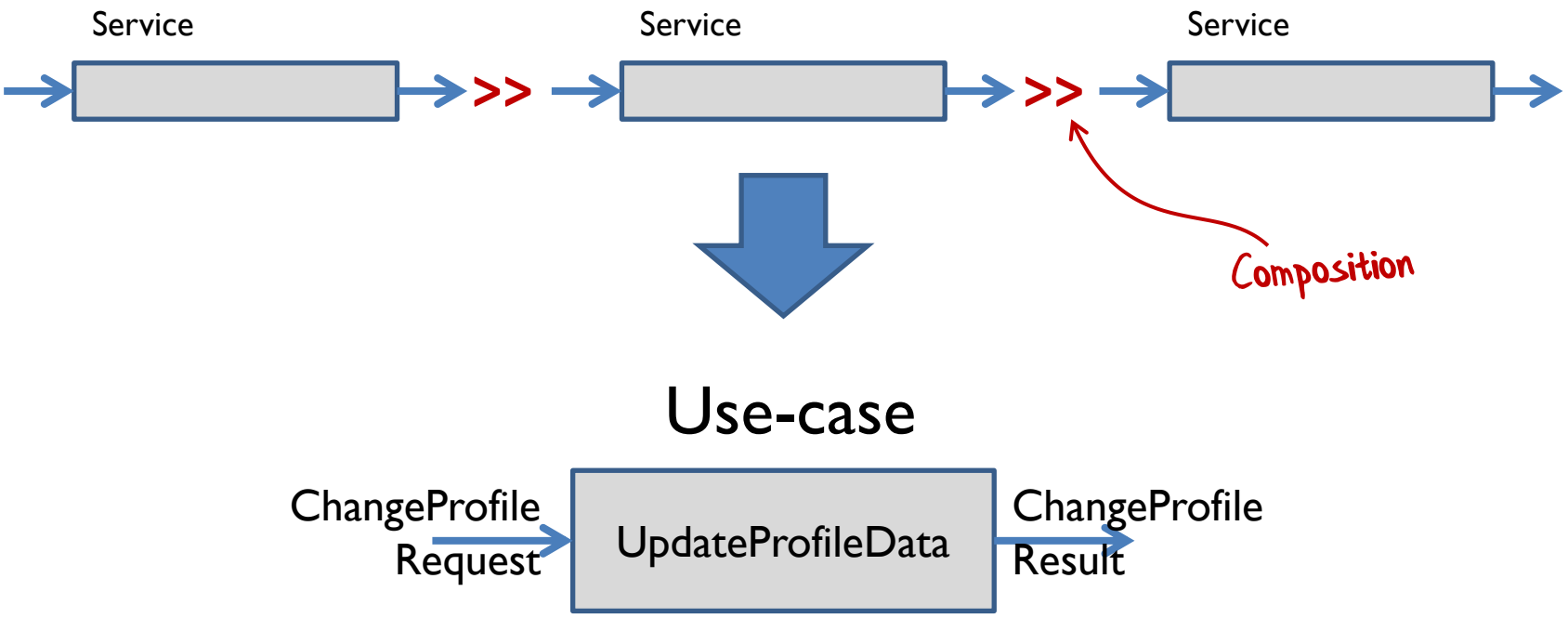
An application is glued together from smaller functions

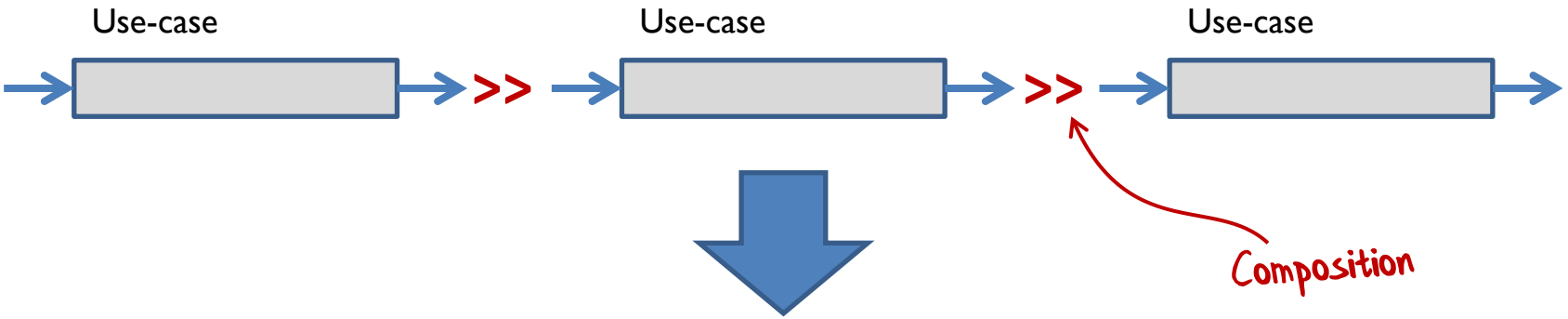
Low-level operation



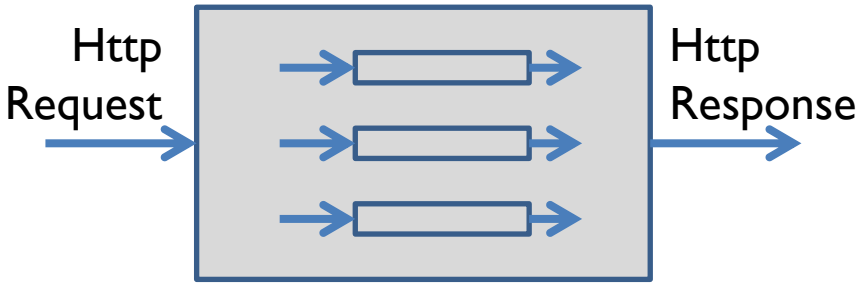
Service

Btw, a "Service" is just like a microservice but without the "micro" in front


Service
Service
Service
>>
>>
Composition
Use-case
ChangeProfile
Request
UpdateProfileData
ChangeProfile
Result

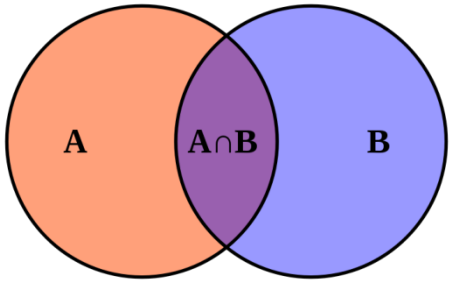


Web application

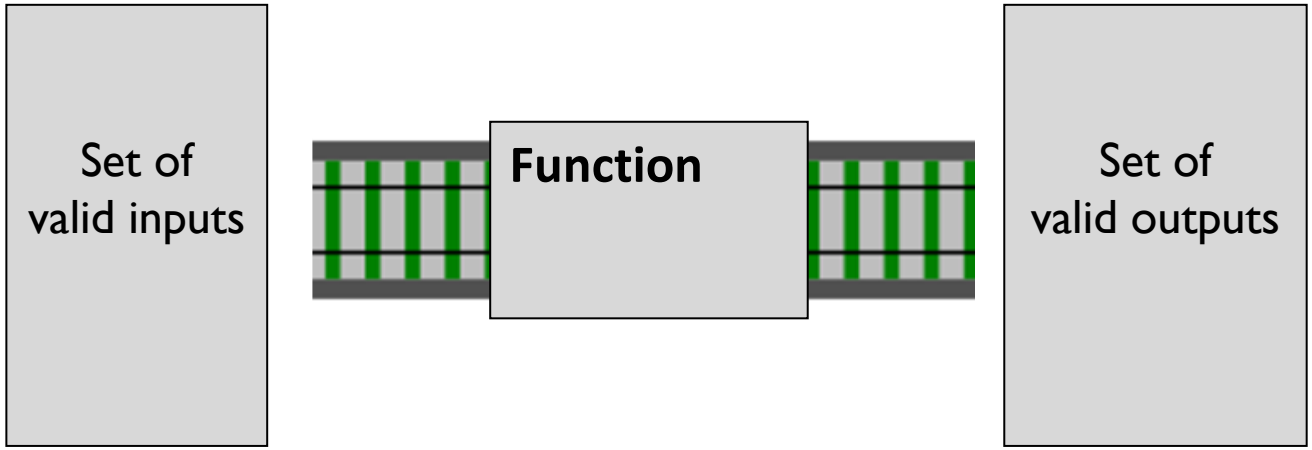


"Composition is fractal"

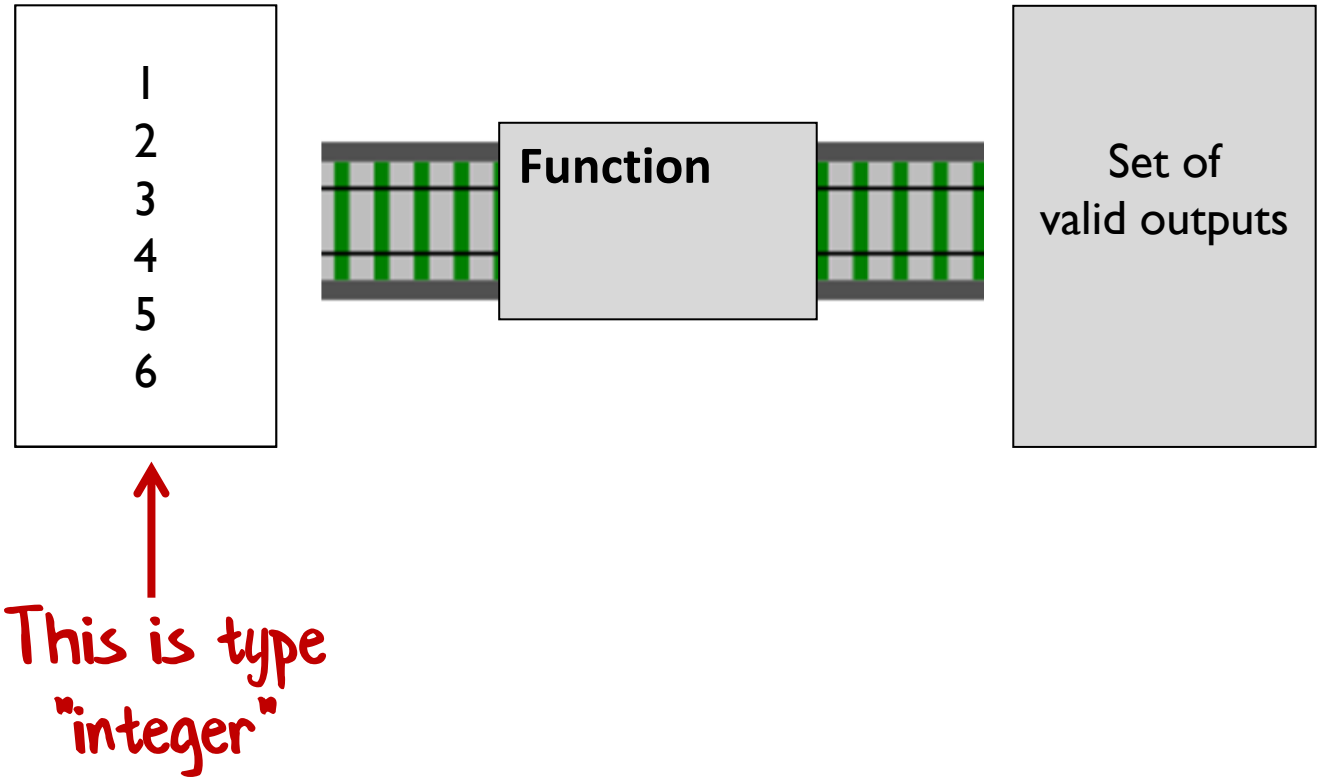
What are types?



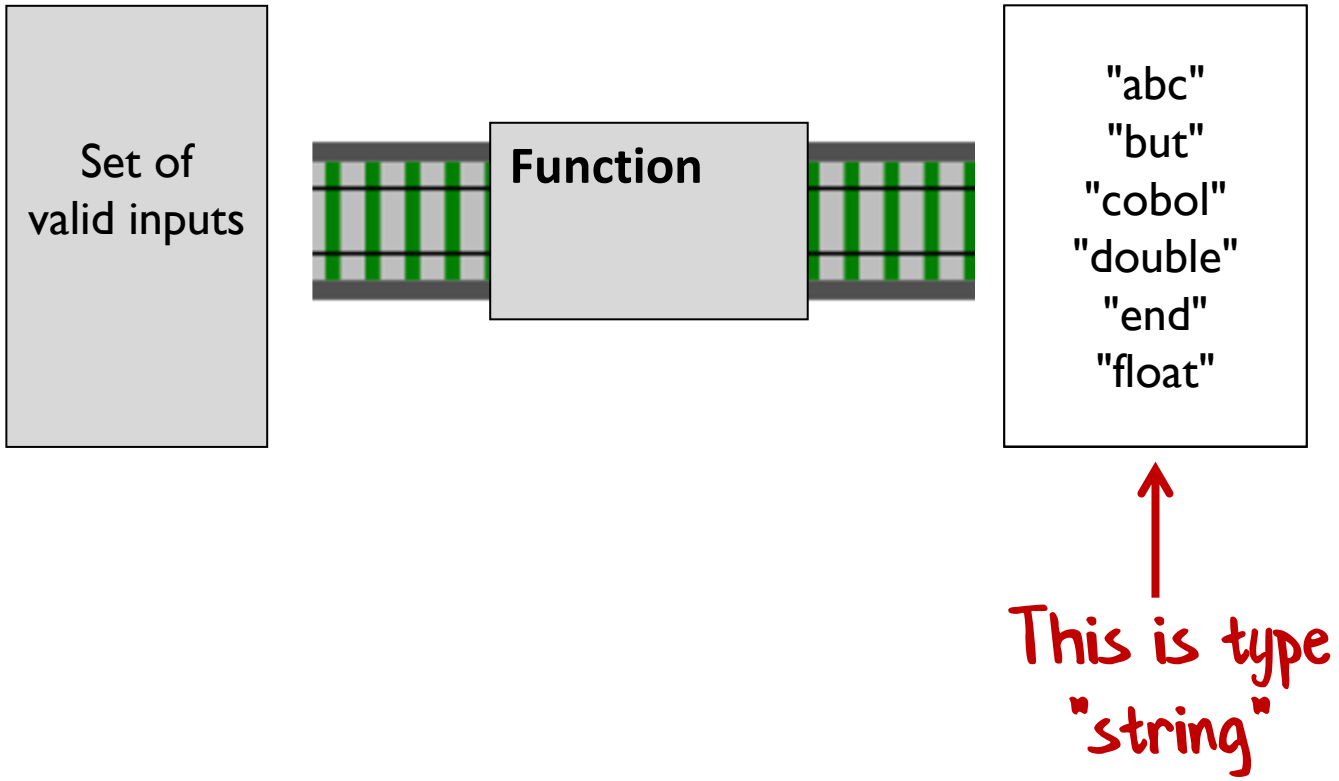
What is a type?



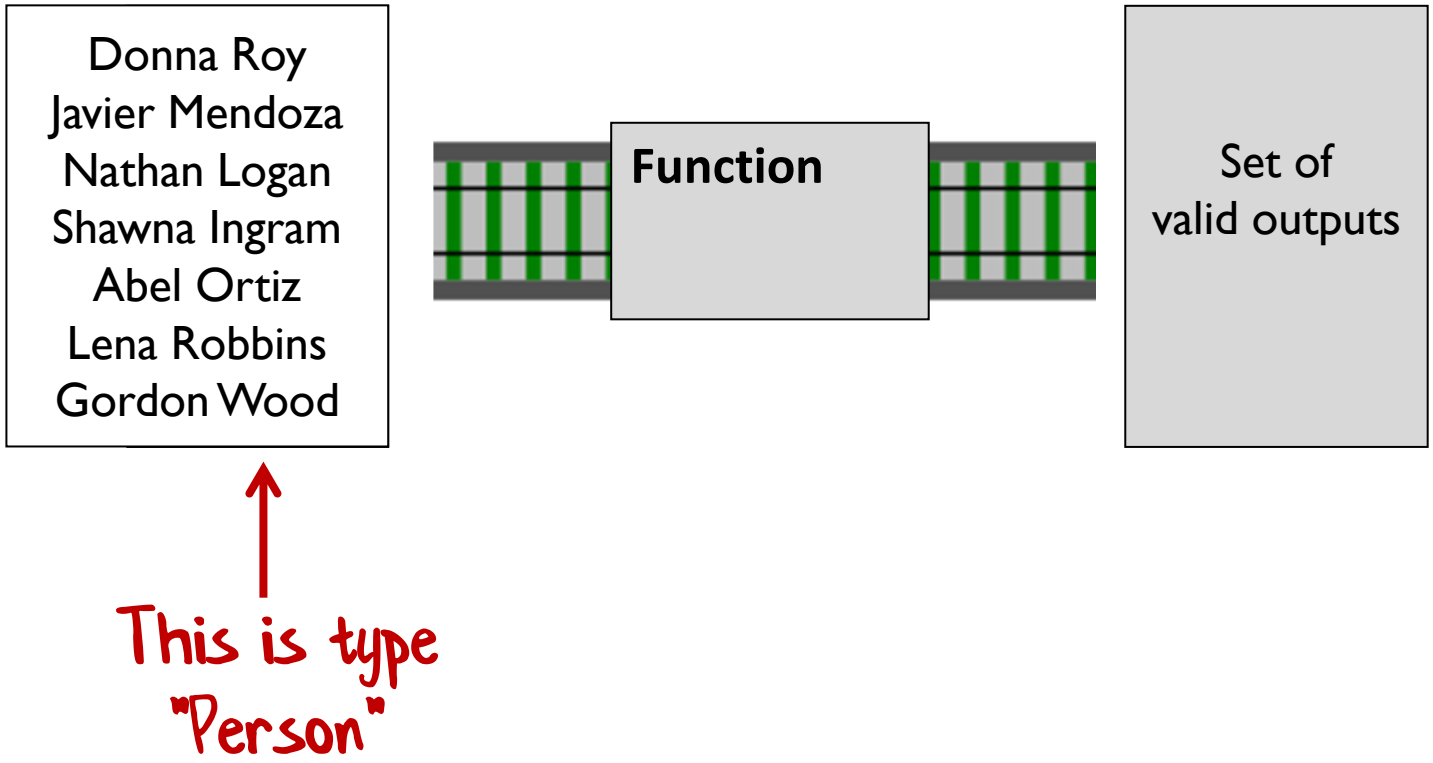
A type is a just a name
for a set of things



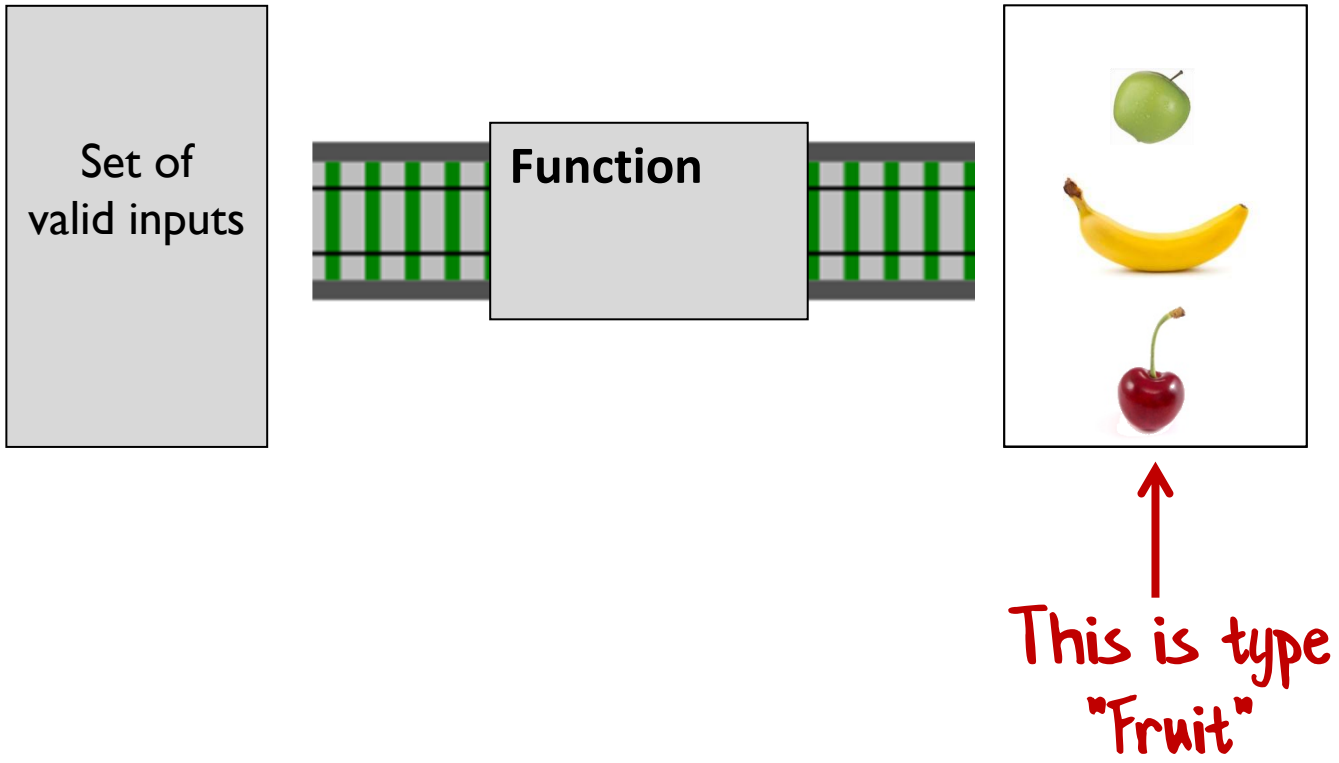
A type is a just a name
for a set of things



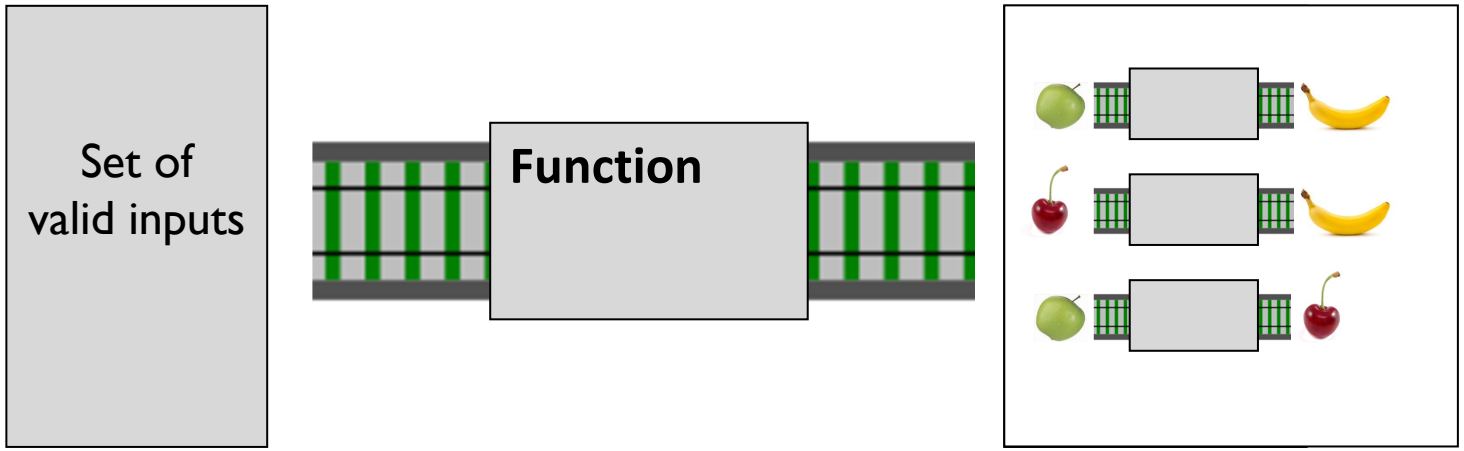
A type is a just a name
for a set of things



A type is a just a name
for a set of things



A type is a just a name
for a set of things



This is a type
containing functions

A type is a just a name
for a set of things

Types can be composed

composable means => like Lego

New types are built from smaller types in two ways

New types are built from smaller types in two ways

"AND"

type FruitSalad =

2 each of  and  and 

Example: pairs, tuples, records

New types are built from smaller types in two ways

"OR"

type Snack =  or  or 

Not available in C# or Java
Example coming up!

Real world example of type composition

type **ChequeNumber** = int

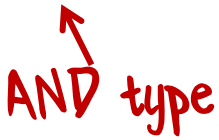
type **CardNumber** = string

type **ChequeNumber** = int

type **CardNumber** = string

type **CardType** = Visa | Mastercard

type **CreditCardInfo** = CardType * CardNumber



```
type ChequeNumber = int
type CardNumber = string
type CardType = Visa | Mastercard
type CreditCardInfo = CardType * CardNumber

type PaymentMethod =
  | Cash
  | Cheque of ChequeNumber
  | Card of CreditCardInfo
```

```
type ChequeNumber = int
type CardNumber = string
type CardType = Visa | Mastercard
type CreditCardInfo = CardType * CardNumber

type PaymentMethod =
  | Cash
  | Cheque of ChequeNumber
  | Card of CreditCardInfo

type PaymentAmount = decimal
type Currency = EUR | USD
```

Another primitive type

Another OR type

```
type ChequeNumber = int
type CardNumber = string
type CardType = Visa | Mastercard
type CreditCardInfo = CardType * CardNumber

type PaymentMethod =
  | Cash
  | Cheque of ChequeNumber
  | Card of CreditCardInfo

type PaymentAmount = decimal
type Currency = EUR | USD

type Payment = {
    Amount : PaymentAmount
    Currency: Currency
    Method: PaymentMethod }
```

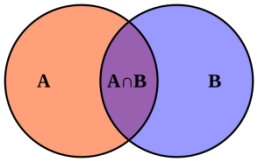
AND type

Final type built from many smaller types

Functions

Composition

Types



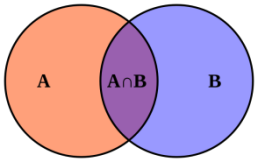
Congratulations!

You now understand functional programming.

Functions

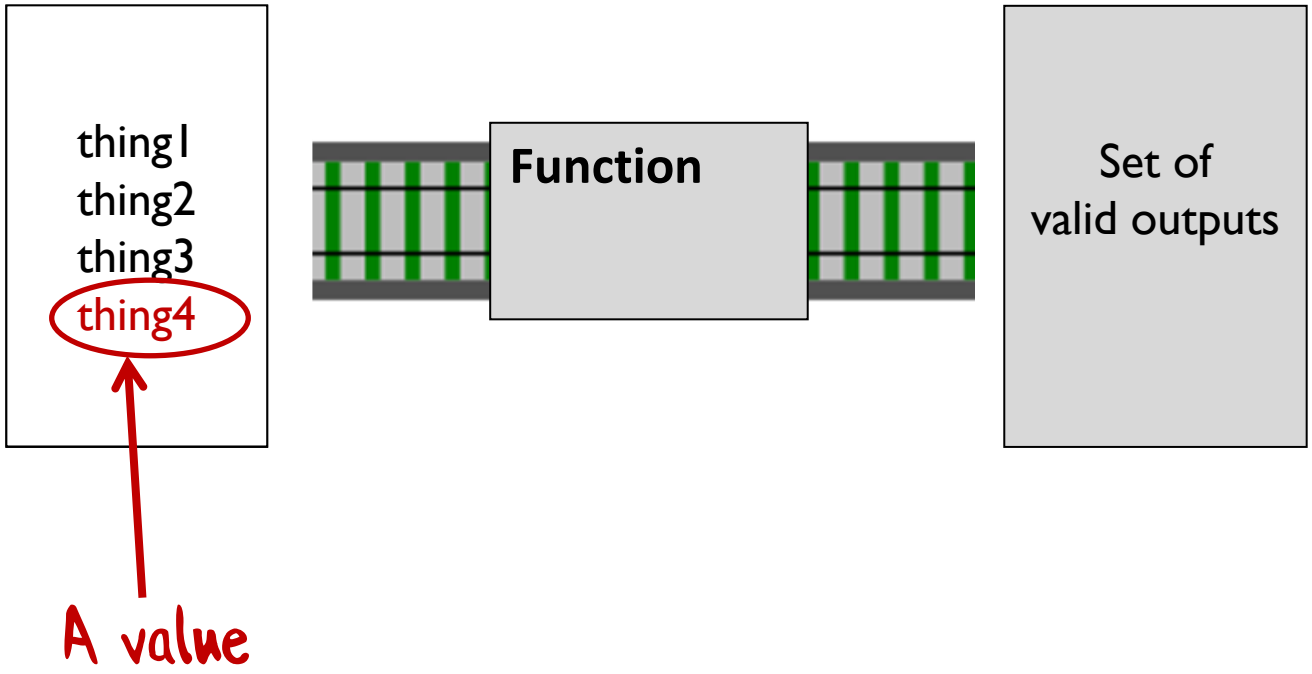
Composition

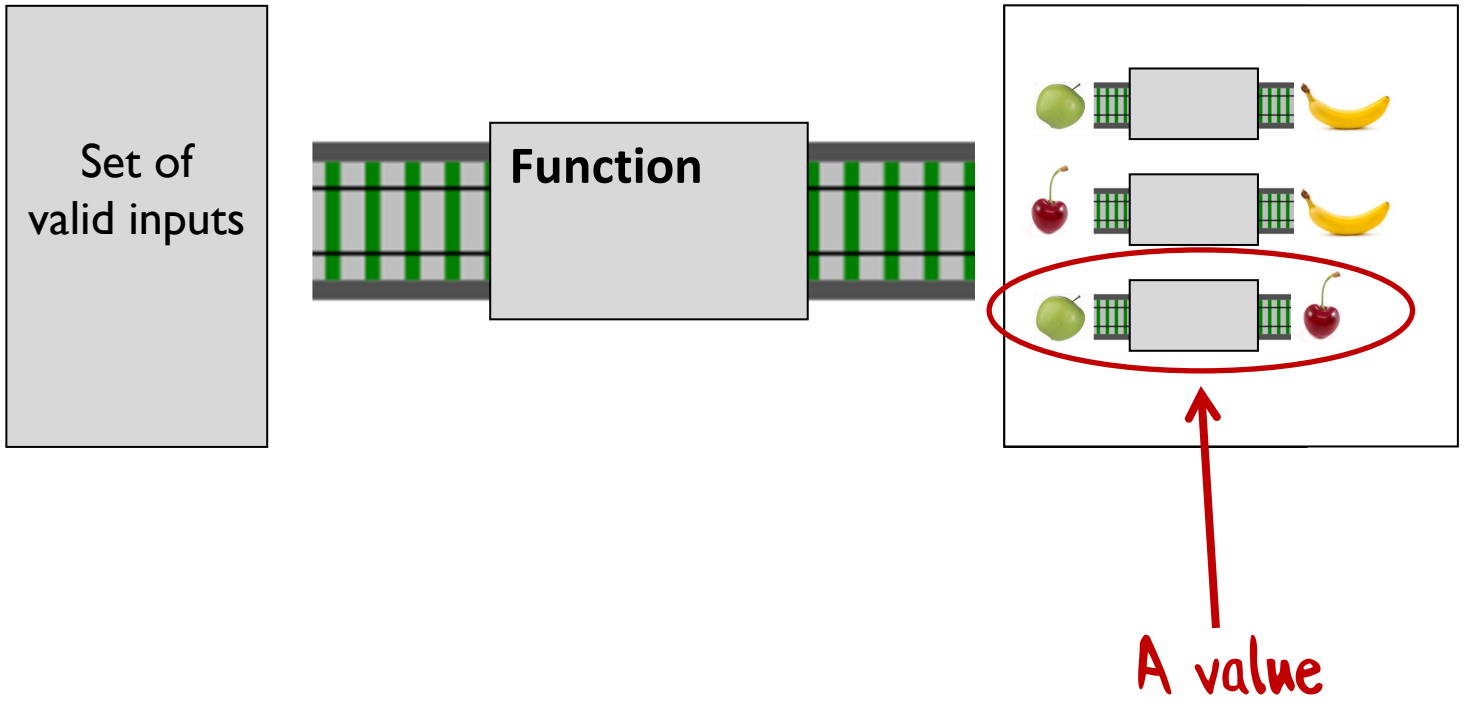
Types



We'll be focusing on this bit today

"Values" not "variables"


thing1
thing2
thing3
thing4
A value
Function
Set of
valid outputs


Set of
valid inputs
Function
A value

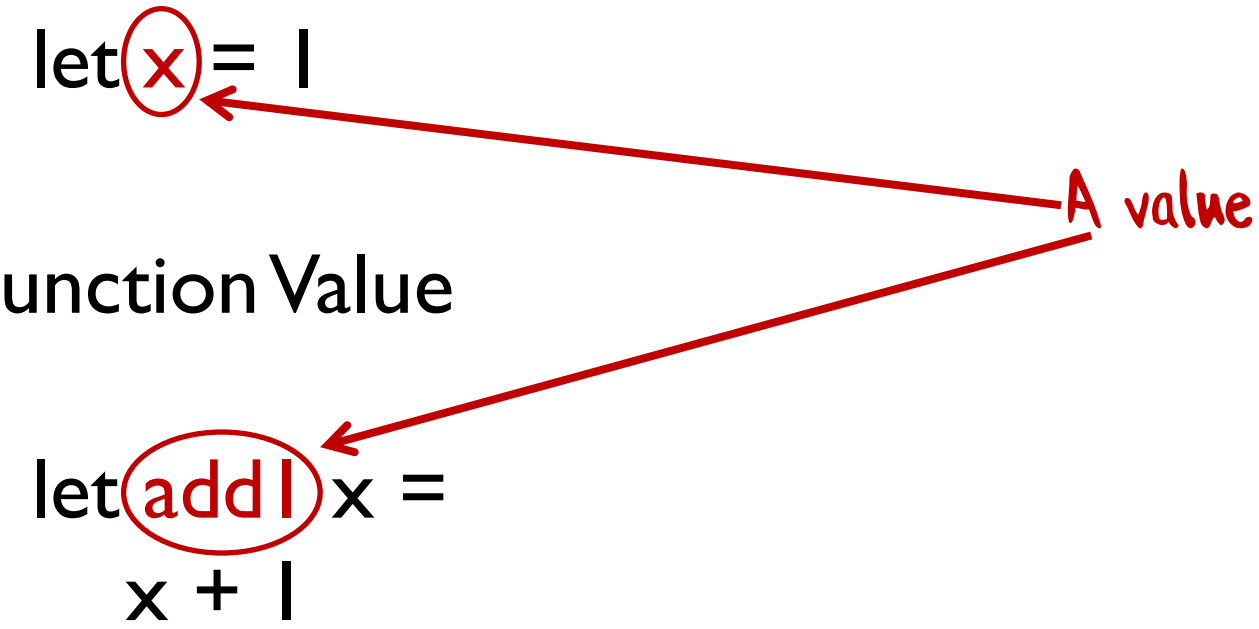
Values (F#)

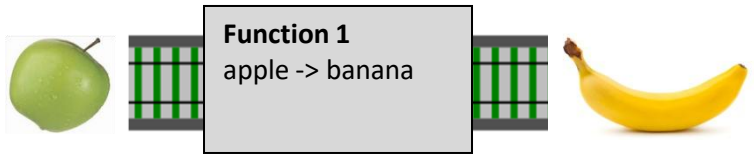
Simple Value

```
let x = 1
```

Function Value

```
let add1 x =
    x + 1
```





What is the type?

Example of a value?

Values vs. Objects

- A “value” is just a member of a type
- Values have no behaviour
- Values are immutable
- Changes done by external functions

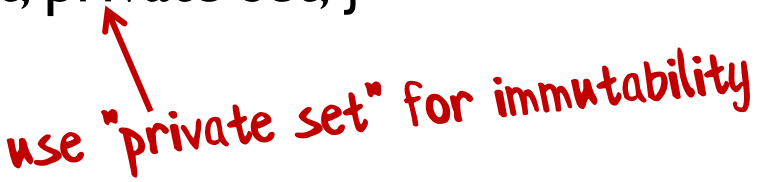
- A “object” is an encapsulation of a data structure …
- … with its associated behavior
- Objects are expected to have state (that is, be mutable)
- All operations that change the internal state must be provided by the object itself (via "dot" notation).

Benefits of immutable values for domain modelling

Immutability also helps parallelism, etc. But that is not relevant here.

Value object definition in C#

```
class PersonalName
{
    public PersonalName(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }
}
```



Value object definition in C#

```
class PersonalName
{
    public PersonalName(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }
}
```

Classes are reference types

Value object definition in C# (extra code for equality)

```
class PersonalName
{
    // all the code from above, plus...

    public override int GetHashCode()
    {
        return this.FirstName.GetHashCode() + this.LastName.GetHashCode();
    }

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

    public bool Equals(PersonalName other)
    {
        if ((object) other == null)
        {
            return false;
        }
        return FirstName == other.FirstName && LastName == other.LastName;
    }
```

Value object definition in F#

```
type PersonalName = {FirstName:string; LastName:string}
```

Value object definition in F# (extra code for equality)

This page intentionally left blank

the best code is no code at all

F# values and DDD

- F# values are excellent for Value Objects
  - F# values are immutable by default
  - F# values define equality and comparison
- F# values are excellent for Entities too
  - Immutable entities are good!

Advantages of immutability

type **Person** = { ... ... }

immutable

The only way to create an object

let **tryCreatePerson** name =
// validate on construction
// if input is valid return something ✓
// if input is not valid return error ✗

All changes must go through this checkpoint

Great for enforcing invariants in one place

WHY USE TYPES FOR DOMAIN MODELLING?

Types are executable documentation

```
type CardType = Visa | Mastercard
type CardNumber = CardNumber of string
type ChequeNumber = ChequeNumber of int

type PaymentMethod =
  | Cash
  | Cheque of ChequeNumber
  | Card of CardType * CardNumber
```

```
module CardGame =
  type Suit = Club | Diamond | Spade | Heart
  type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list
  type ShuffledDeck = ShuffledDeck of Deck

  type Player = {Name:string; Hand:Hand}
  type Game = {Deck:Deck; Players: Player list}

  type ToShuffle = Deck -> ShuffledDeck
  type ToDeal = ShuffledDeck -> (ShuffledDeck * Card)
```

X -> Y means a function
- input of type X
- output of type Y

```
module CardGame =
  type Suit = Club | Diamond | Spade | Heart
  type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list
  type ShuffledDeck = ShuffledDeck of Deck

  type Player = {Name:string; Hand:Hand}
  type Game = {Deck:Deck; Players: Player list}

  type ToShuffle = Deck -> ShuffledDeck
  type ToDeal = ShuffledDeck -> (ShuffledDeck * Card)
```

Question : can you deal with an unshuffled deck?

```
module CardGame =
  type Suit = Club | Diamond | Spade | Heart
  type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list
  type ShuffledDeck = ShuffledDeck of Deck

  type Player = {Name:string; Hand:Hand}
  type Game = {Deck:Deck; Players: Player list}

  type ToShuffle = Deck -> ShuffledDeck
  type ToDeal = ShuffledDeck -> (ShuffledDeck * Card)
```

Types can be nouns

Types can be verbs

Bounded context

Ubiquitous language

```
module CardGame =
  type Suit = Club | Diamond | Spade | Heart
  type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list
  type ShuffledDeck = ShuffledDeck of Deck

  type Player = {Name:string; Hand:Hand}
  type Game = {Deck:Deck; Players: Player list}

  type ToShuffle = Deck –› ShuffledDeck
  type ToDeal = ShuffledDeck –› (ShuffledDeck * Card)
```

```
module CardGame =
  type Suit = Club | Diamond | Spade | Heart
  type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list
  type ShuffledDeck = ShuffledDeck of Deck

  type Player = {Name:string; Hand:Hand}
  type Game = {Deck:Deck; Players: Player list}

  type ToShuffle = Deck -> ShuffledDeck
  type ToDeal = ShuffledDeck -> (ShuffledDeck * Card)
```

Do you think this is a reasonable amount of code to write for this domain?

Do you think a non programmer could understand this?

```
module CardGame =
  type Suit = Club | Diamond | Spade | Heart
  type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list
  type ShuffledDeck = ShuffledDeck of Deck

  type Player = {Name:string; Hand:Hand}
  type Game = {Deck:Deck; Players: Player list}

  type ToShuffle = Deck -> ShuffledDeck
  type ToDeal = ShuffledDeck -> (ShuffledDeck * Card)
```

"The design is the code,
and the code is the design."

This is not psendocode –
this is executable code!

Types encourage
accurate domain modelling

Business rule:

"First and last name must not be more than 50 chars"

```
type Contact = {
                Must not be more than 50 chars
  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Business rule:

"First and last name must not be more than 50 chars"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Business rule:

"Email field must be a valid email address"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Must contain an "@" sign

Business rule:

"Email field must be a valid email address"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

Define a type that has the required constraint

Business rule:

"Middle initial is optional"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

Required?

Business rule:

"Middle initial is optional"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

Optional can be applied to any type

Business rule:

"Verified emails are different from unverified emails"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1  option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

What is the business logic?

Business rule:

"Verified emails are different from unverified emails"

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

Business rule:

"Verified emails are different from unverified emails"

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

Better modelling

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50

  EmailAddress: EmailContactInfo
  }
```

And boolean has gone!

Types can encode business rules

"compile time unit tests"

Business rule:

"A contact must have an email or a postal address"

```
type Contact = {
   Name: Name
   Email: EmailContactInfo
   Address: PostalContactInfo
   }
```

Not right. This implies both are required.

Business rule:

"A contact must have an email or a postal address"

```
type Contact = {
    Name: Name
    Email: EmailContactInfo option
    Address: PostalContactInfo option
    }
```

Not right either. Both could be missing.

"Make illegal states unrepresentable!"

– Yaron Minsky

"*A contact must have an email or a postal address*"

implies:

- email address only, *or*
- postal address only, *or*
- both email address and postal address

only three possibilities

"A contact must have an email or a postal address"

```
type ContactInfo =
    | EmailOnly of EmailContactInfo
    | AddrOnly of PostalContactInfo
    | EmailAndAddr of EmailContactInfo * PostalContactInfo
```

only three possibilities

requirements are now encoded in the type!

```
type Contact = {
    Name: Name
    ContactInfo : ContactInfo  }
```

What we've seen so far...

- Executable documentation
  - Ubiquitous language
  - Design and code are synchronized
  - Code is understandable by domain expert
- Accurate domain modelling
  - Constraints are explicit
- Encode business rules
  - Illegal states can be made unrepresentable

More at fsharpforfunandprofit.com/ddd

Paulmichael Blasucci
@pblasucci

Following

"The domain model [code] is so succinct the business analysts have started using it as documentation."

Simon Cousins
@simontcousins

Following

pm: "that code is clearer than the spec", me: "can i paste it into the documentation then?", pm: "yes"

Reid Evans
@ReidNEvans

Follow

We had been having difficulty understanding the domain. 40 lines of #fsharp later we were all on the same page

End of part 1

What do you want to do next?

1. I live code a domain suggested by audience
2. I keep going and explain types more deeply
3. You work on exercises

Part II

Understanding the F# type system

An introduction to "~~algebraic~~" types

"composable types"

Composable types

composable means => like Lego

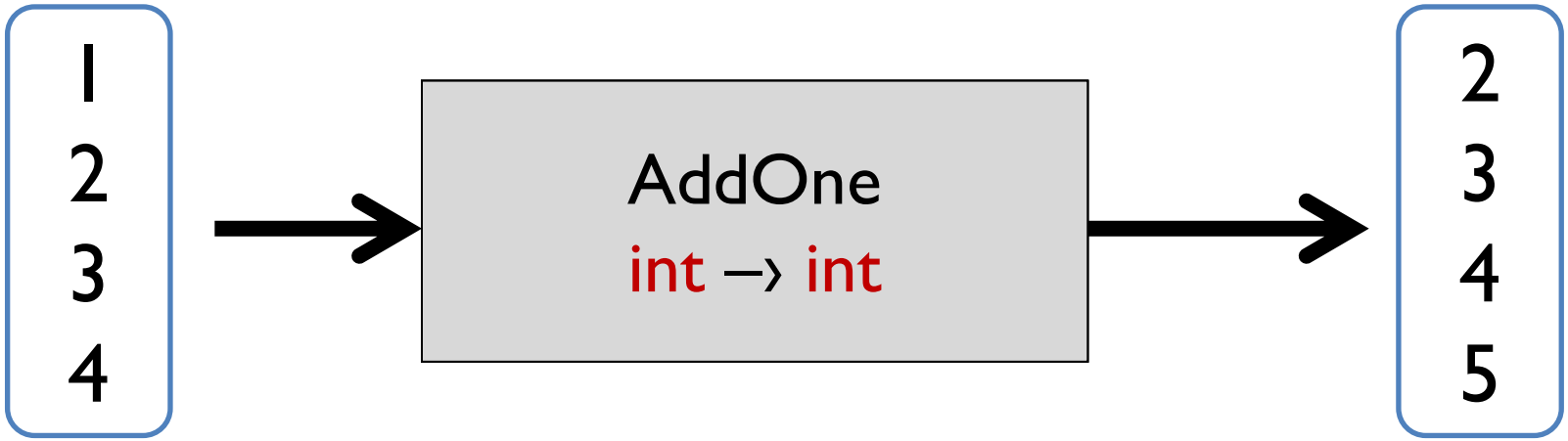
Creating new types

New types are constructed by combining other types using two basic operations:

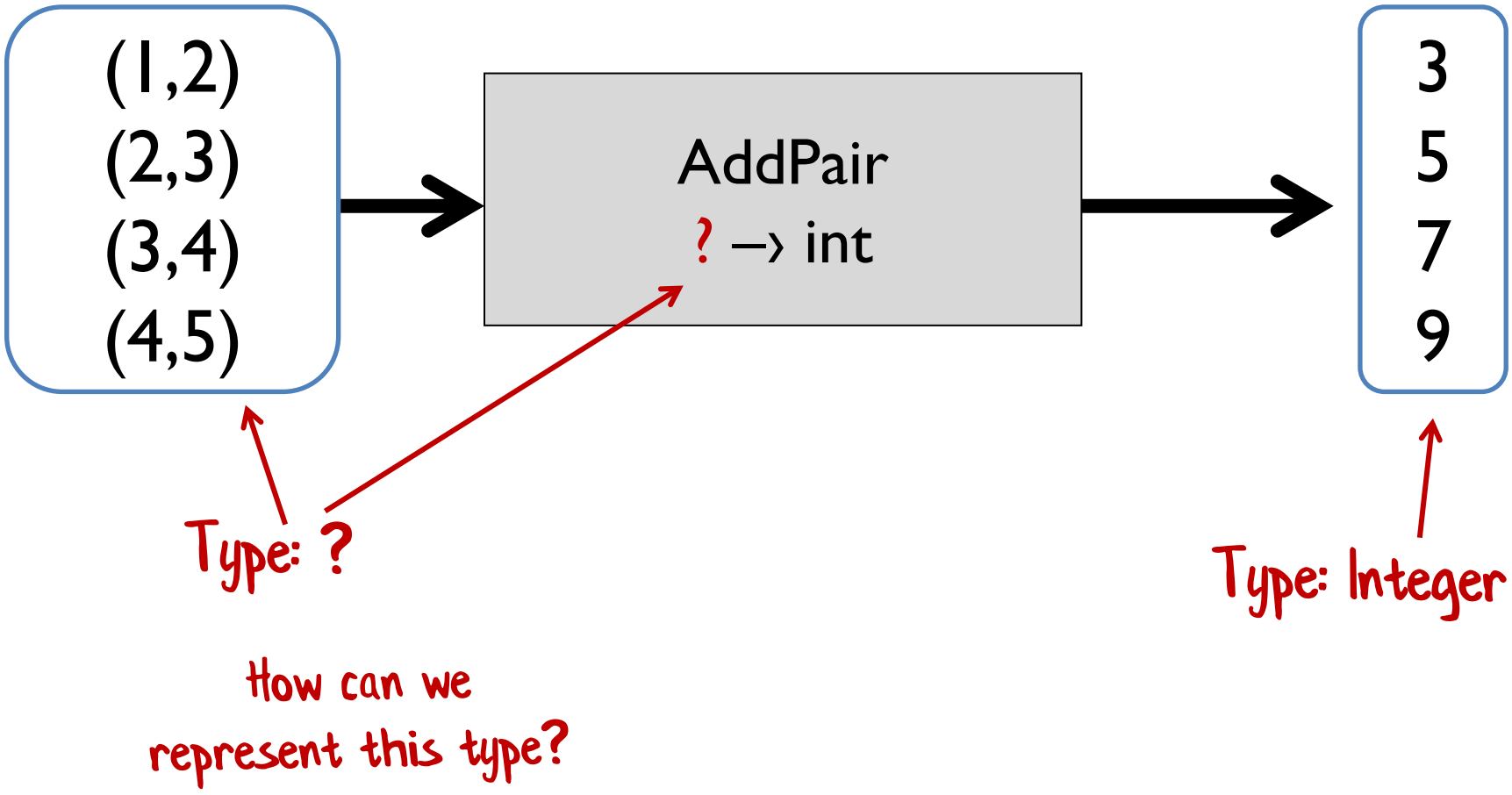
type typeW = typeX "times" typeY

type typeZ = typeX "plus" typeY

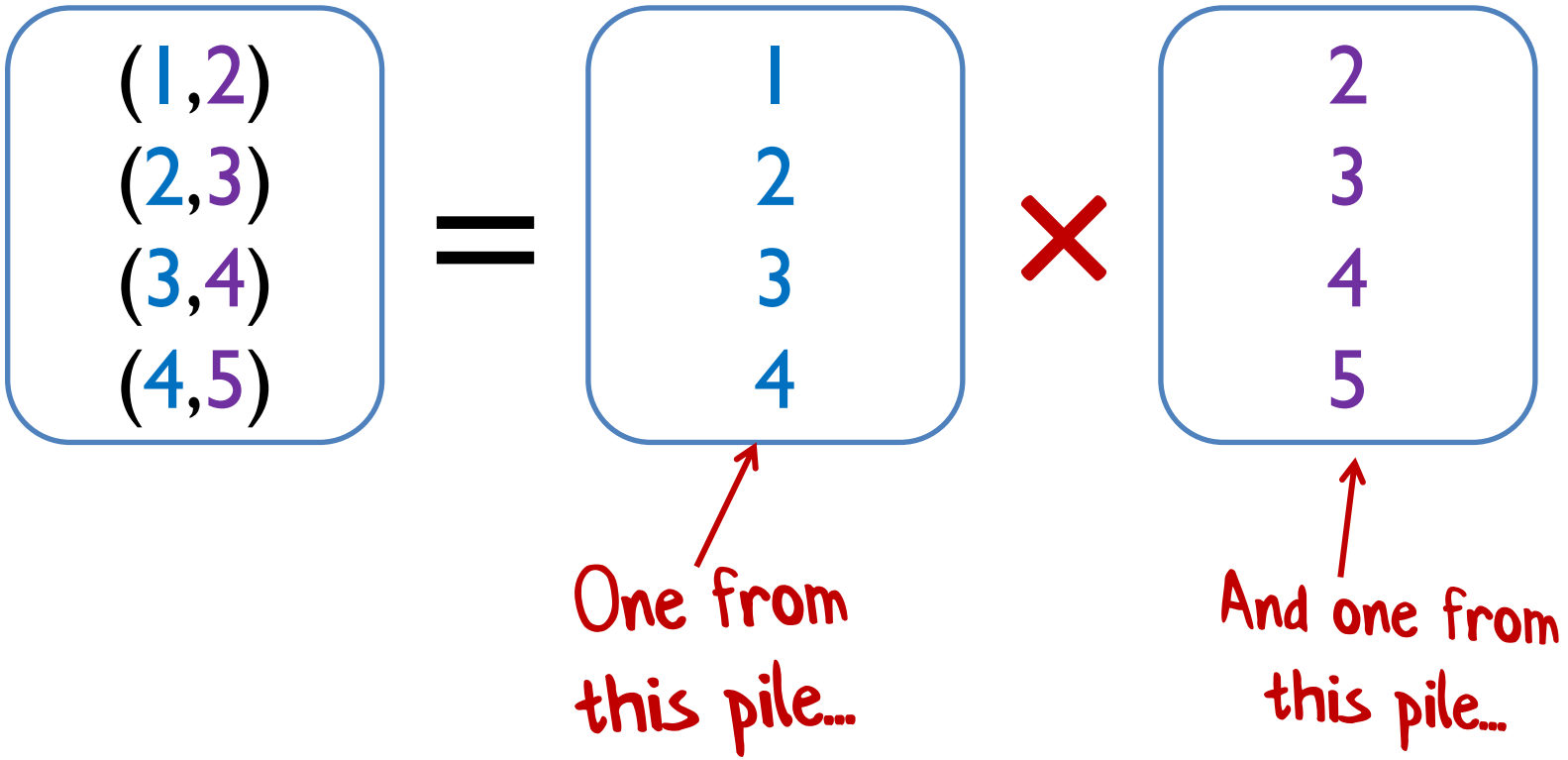
Creating new types



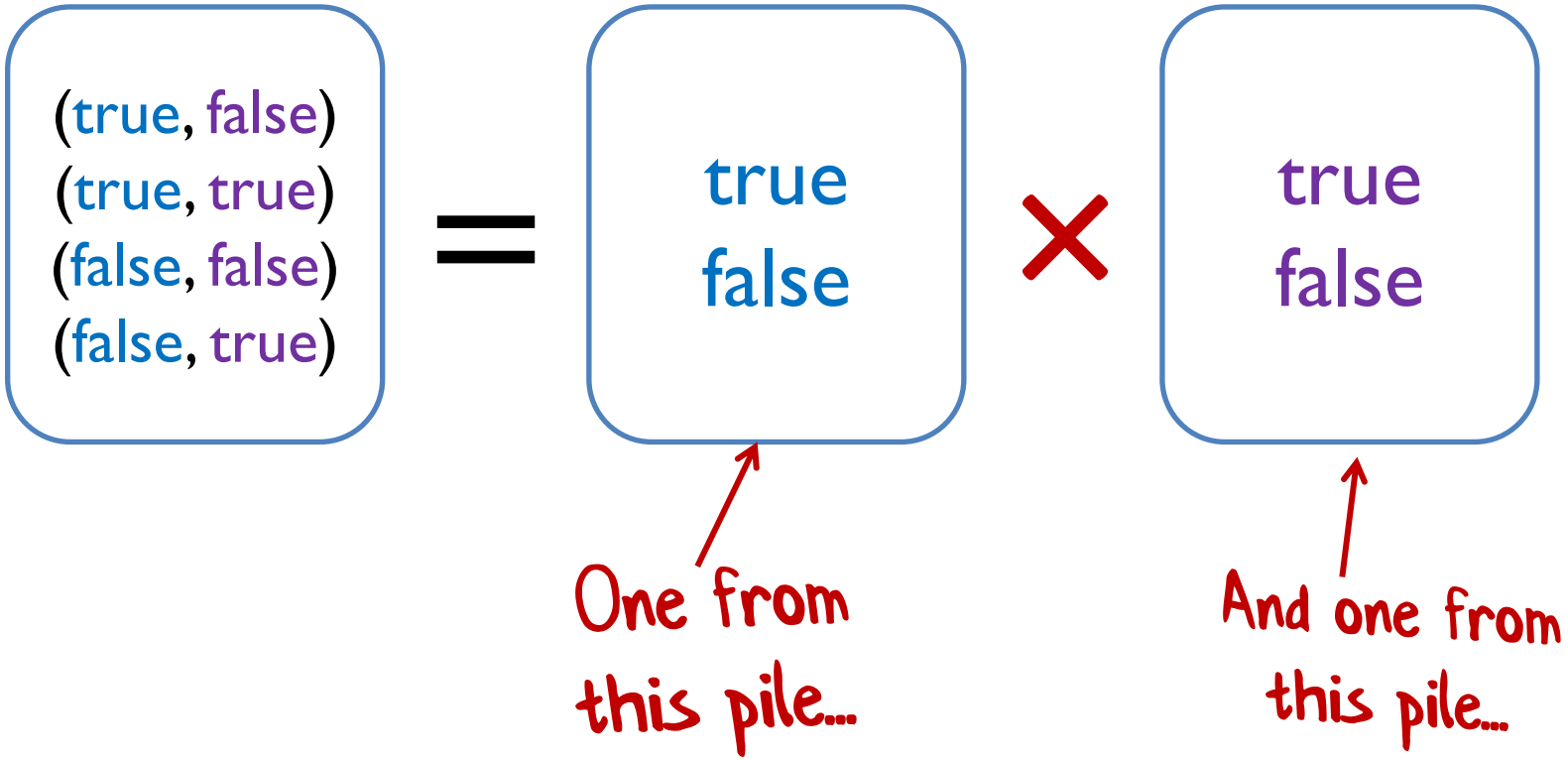
Representing pairs



Representing pairs



Representing pairs



Representing pairs

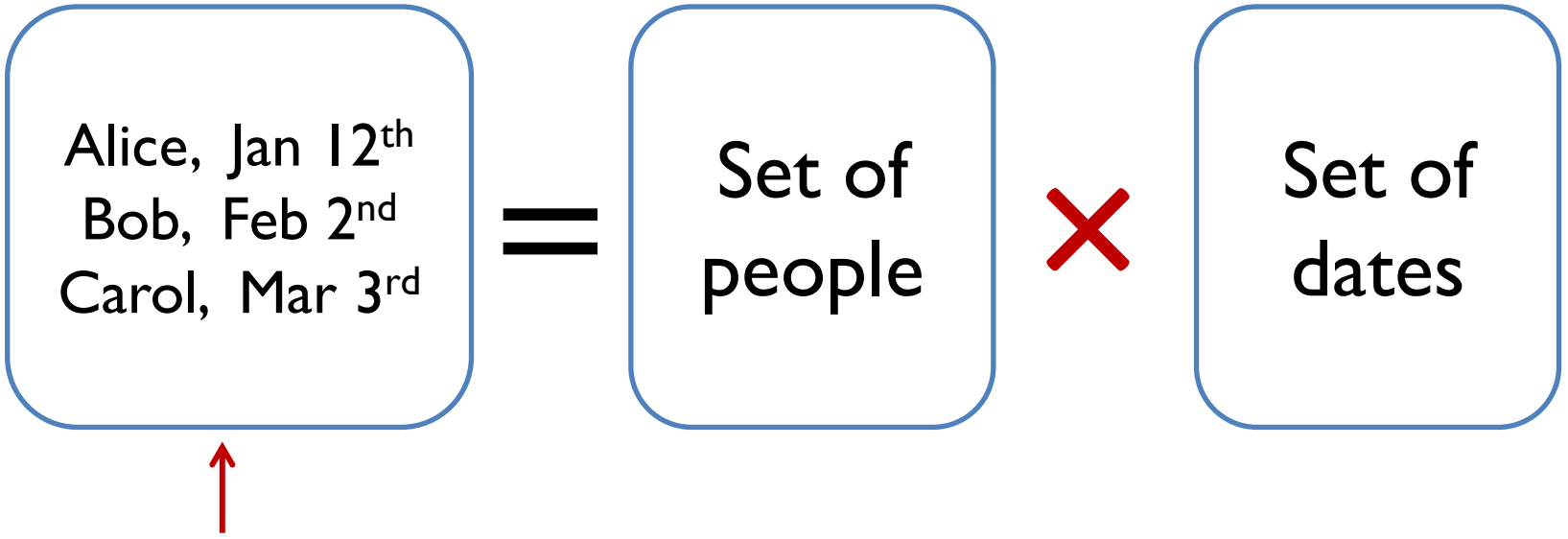
pair of ints

written int * int

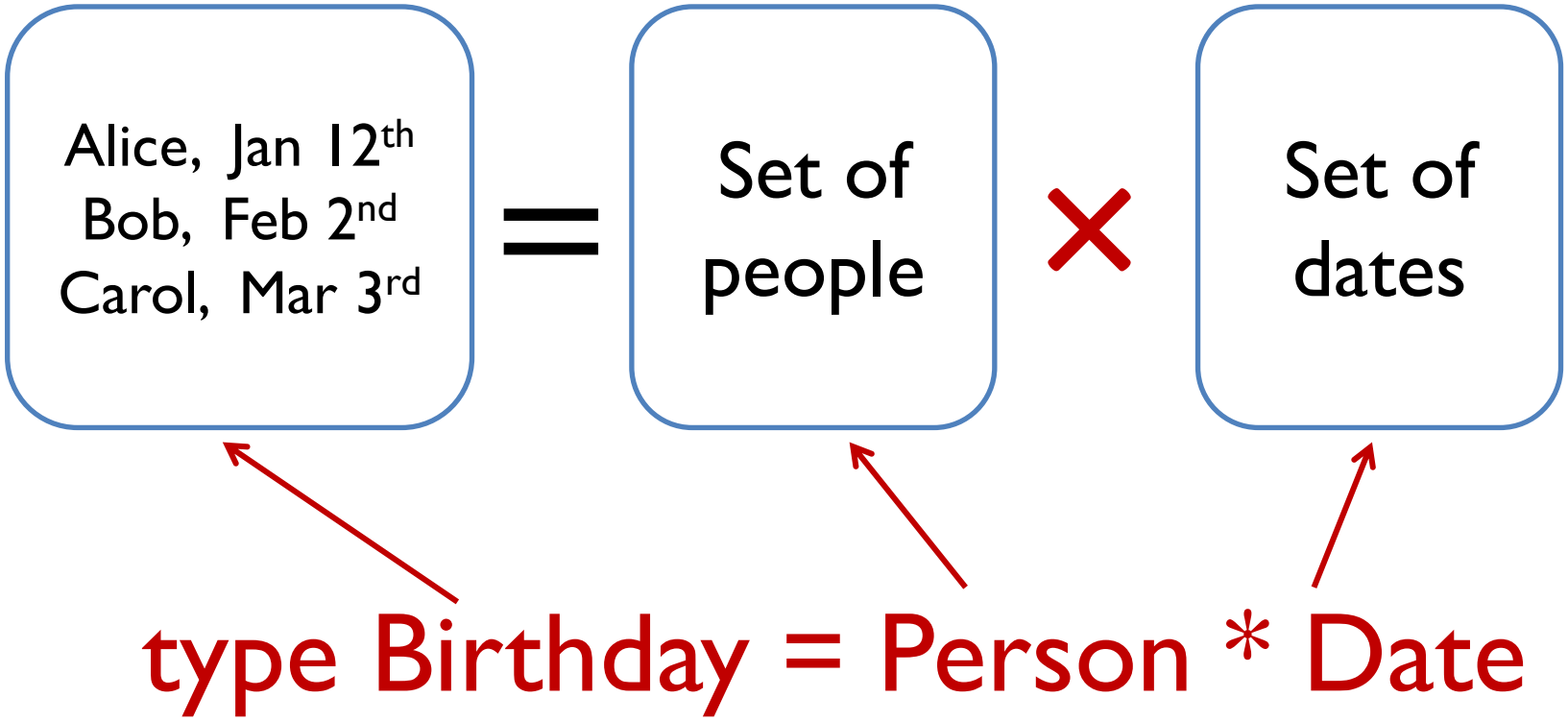
pair of bools

written bool * bool

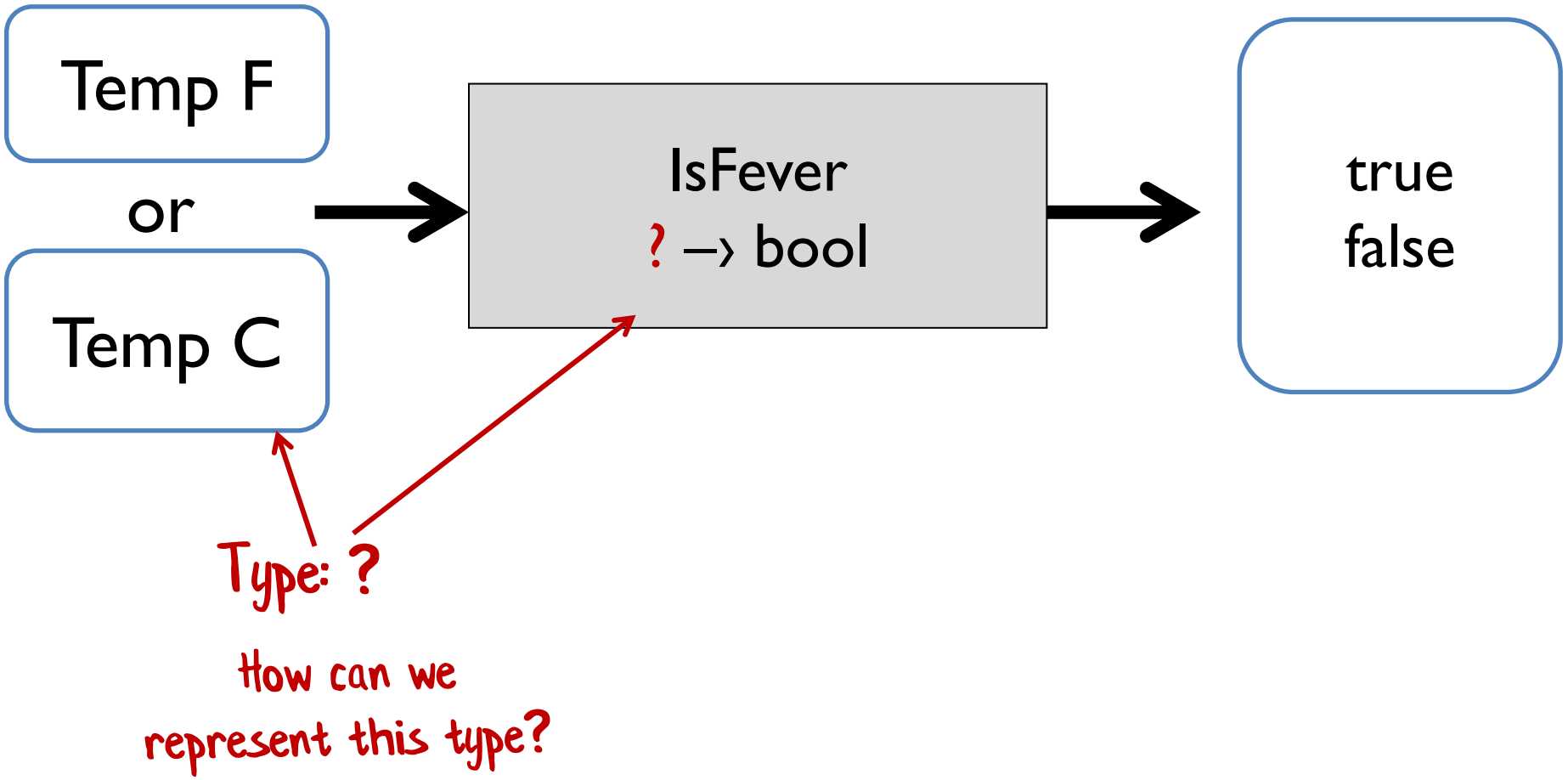
Using tuples for data



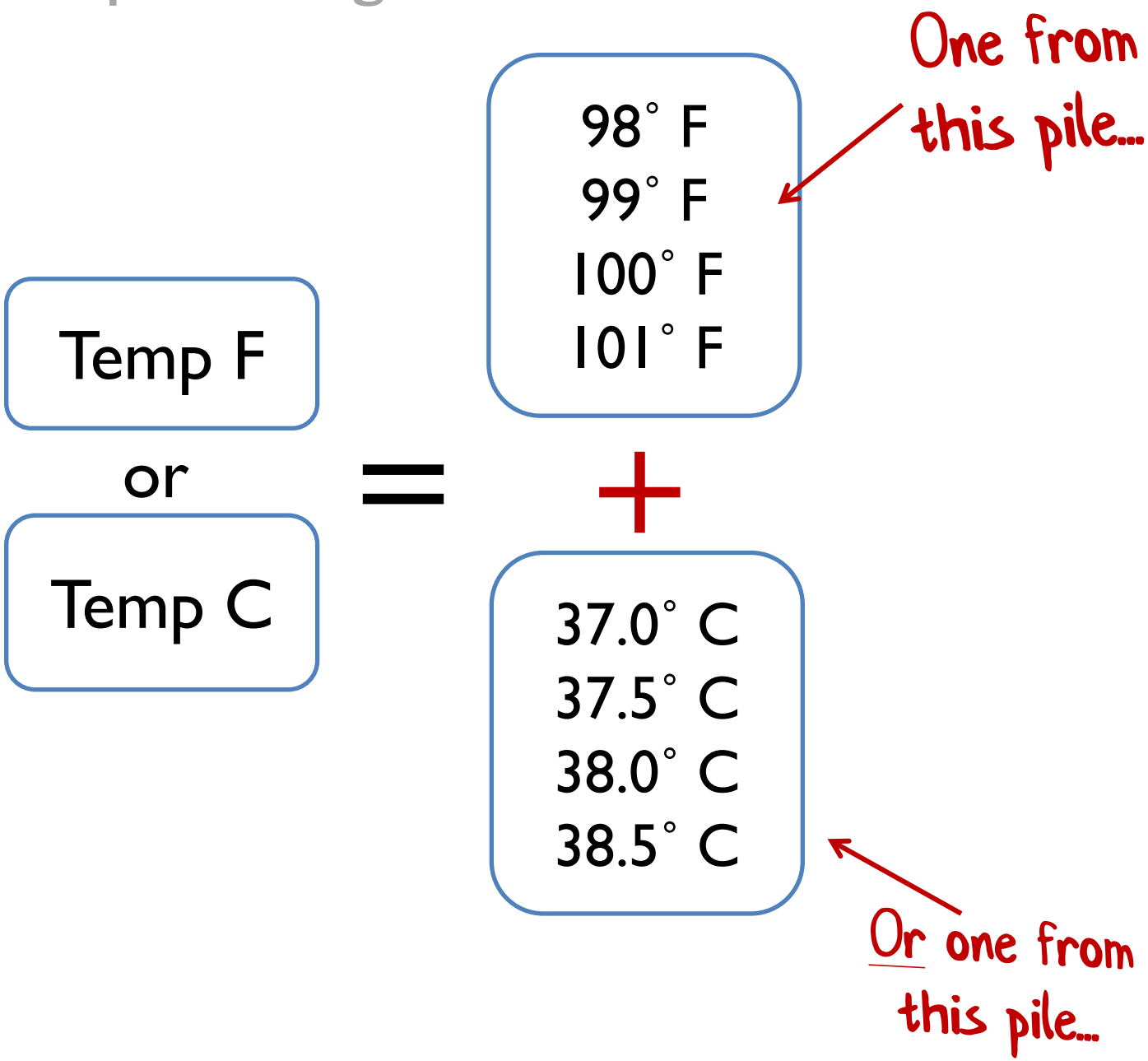
Using tuples for data



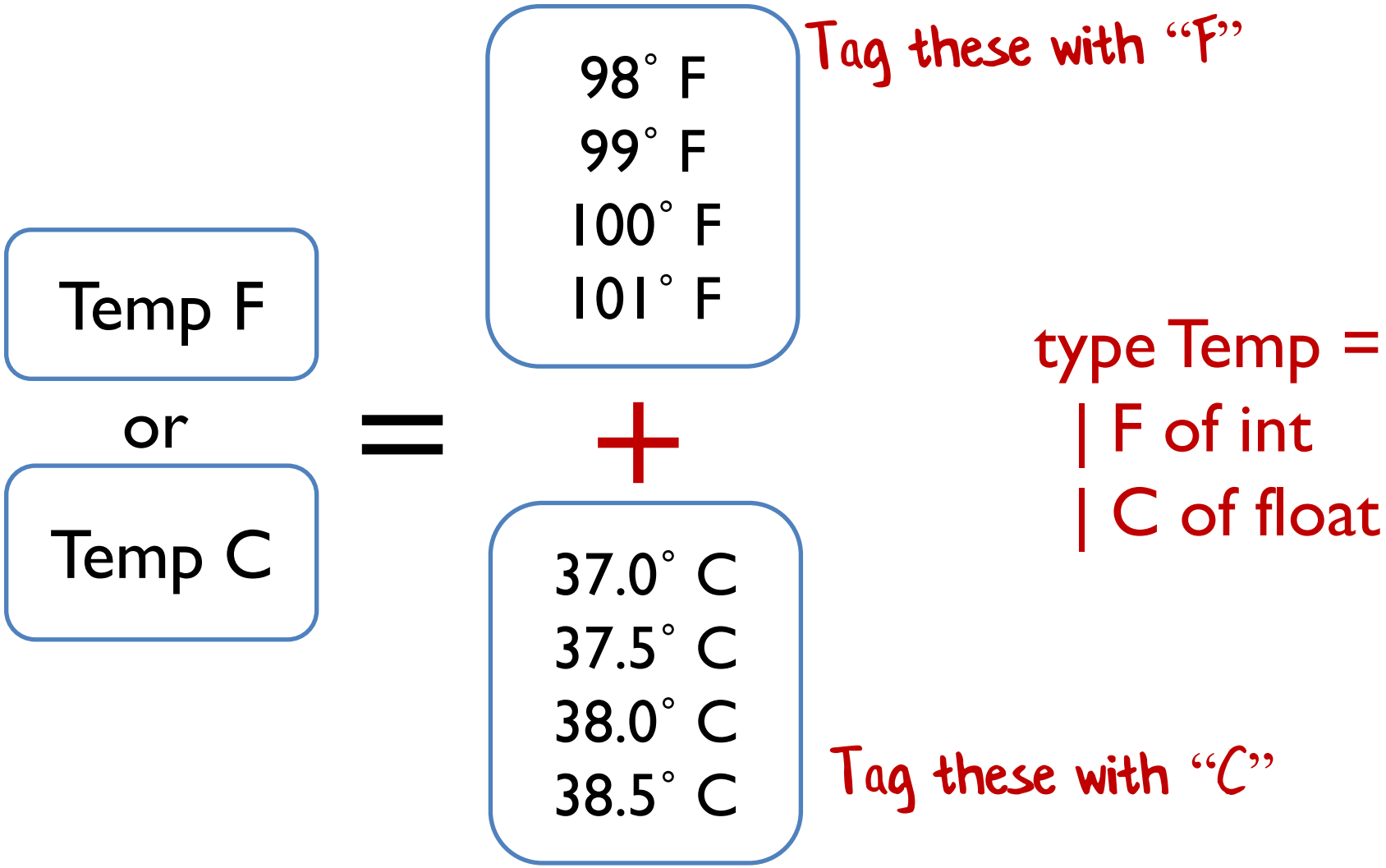
Representing a choice



Representing a choice

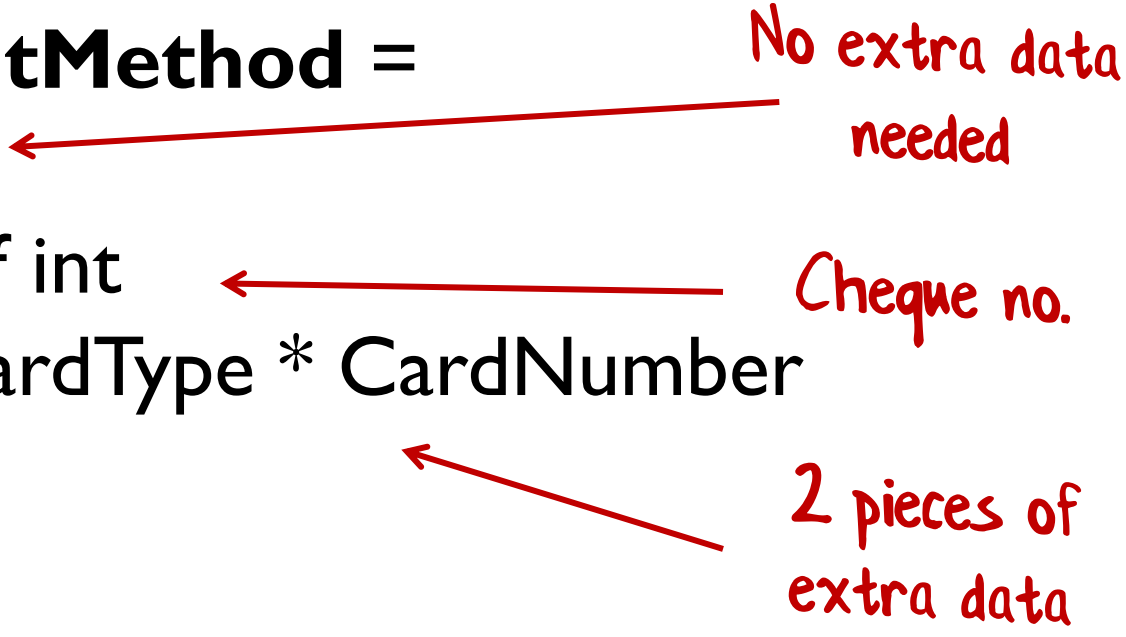


Representing a choice



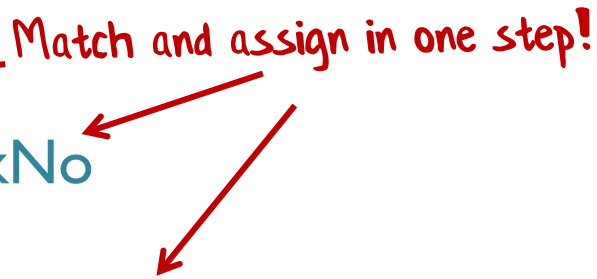
Using choices for data

```
type PaymentMethod =
  | Cash
  | Cheque of int
  | Card of CardType * CardNumber
```



Working with a choice type

```
type PaymentMethod =
  | Cash
  | Cheque of int
  | Card of CardType * CardNumber

let printPayment method =
    match method with
    | Cash –›
        printfn “Paid in cash"
    | Cheque checkNo –›
        printfn “Paid by cheque: %i" checkNo
    | Card (cardType,cardNo) –›
        printfn “Paid with %A %A" cardType cardNo
```



Using choices vs. inheritance

"closed" set of options

```
type PaymentMethod =
  | Cash
  | Cheque of int
  | Card of CardType * CardNumber
```

extra data is obvious

OO version:

```
interface IPaymentMethod {..}
class Cash : IPaymentMethod {..}
class Cheque : IPaymentMethod {..}
class Card : IPaymentMethod {..}
class Evil : IPaymentMethod {..}
```

What goes in here? What is the common behaviour?

Data and code is scattered around many locations

"open" set of options – unpleasant surprises?

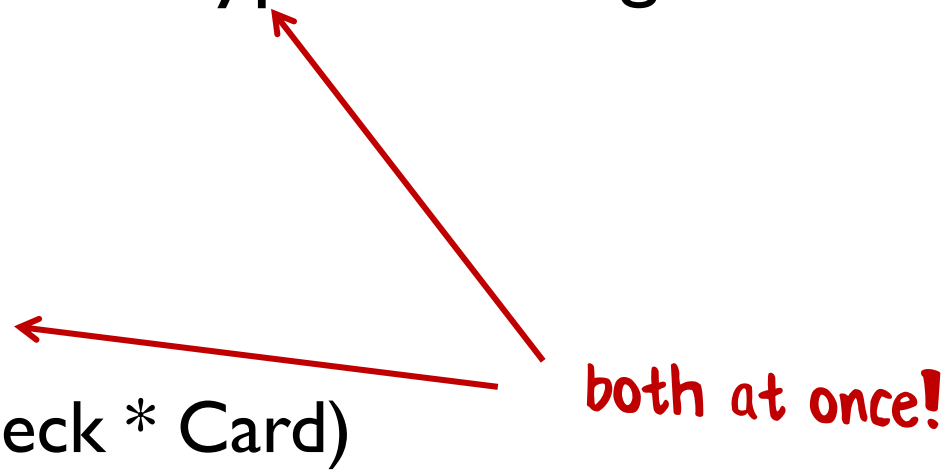
Summary: What are types for in FP?

An annotation to a value for type checking

type AddOne: int –› int

Domain modelling tool

type Deal = Deck –› (Deck * Card)



"a good static type system is like having compile-time unit tests"

TYPE ALL THE THINGS

Designing with types

What can we do with this type system?

Optional values

Required vs. Optional

```
type PersonalName =
    {
    FirstName: string;
    MiddleInitial: string;
    LastName: string;
    }
```

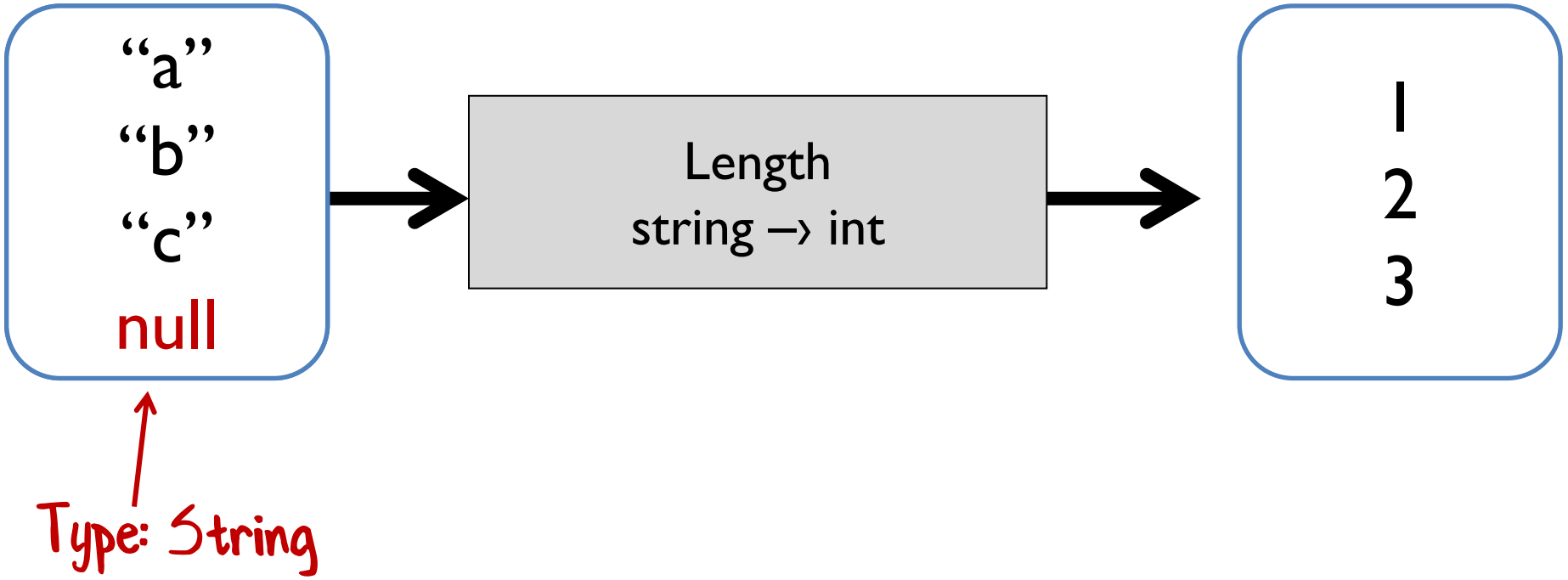
required

optional

required

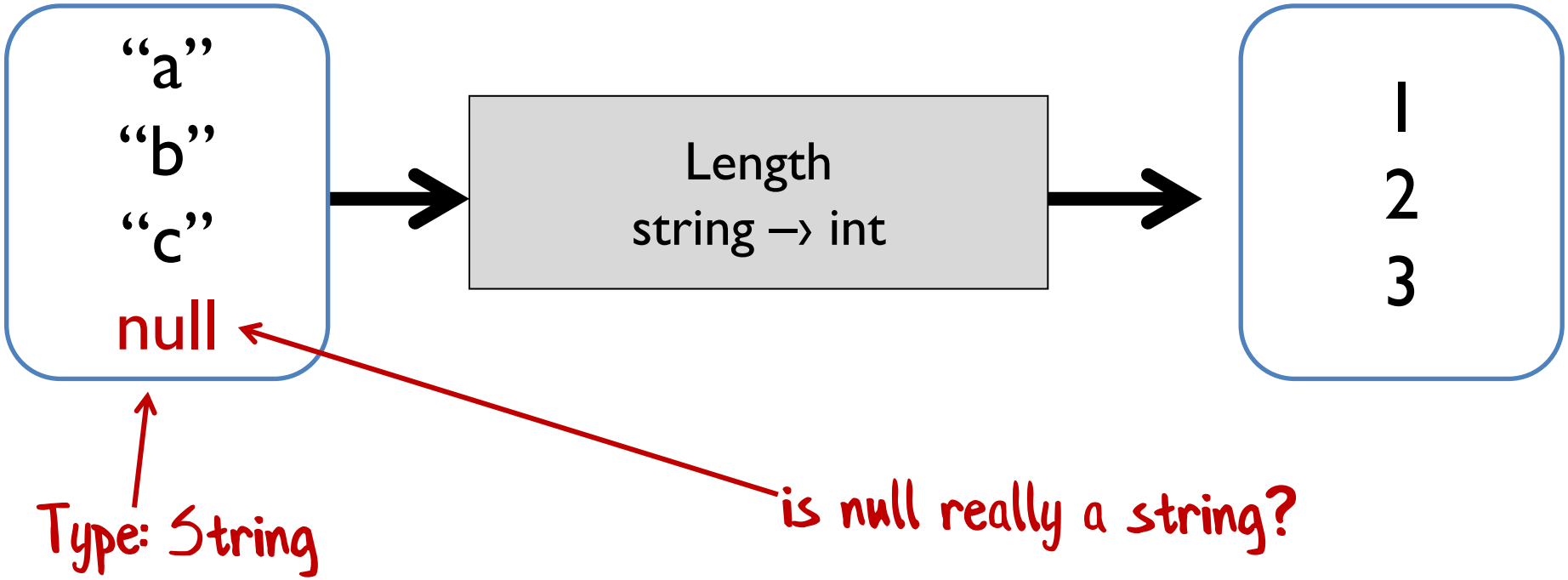
How can we represent optional values?

Null is not the same as “optional”



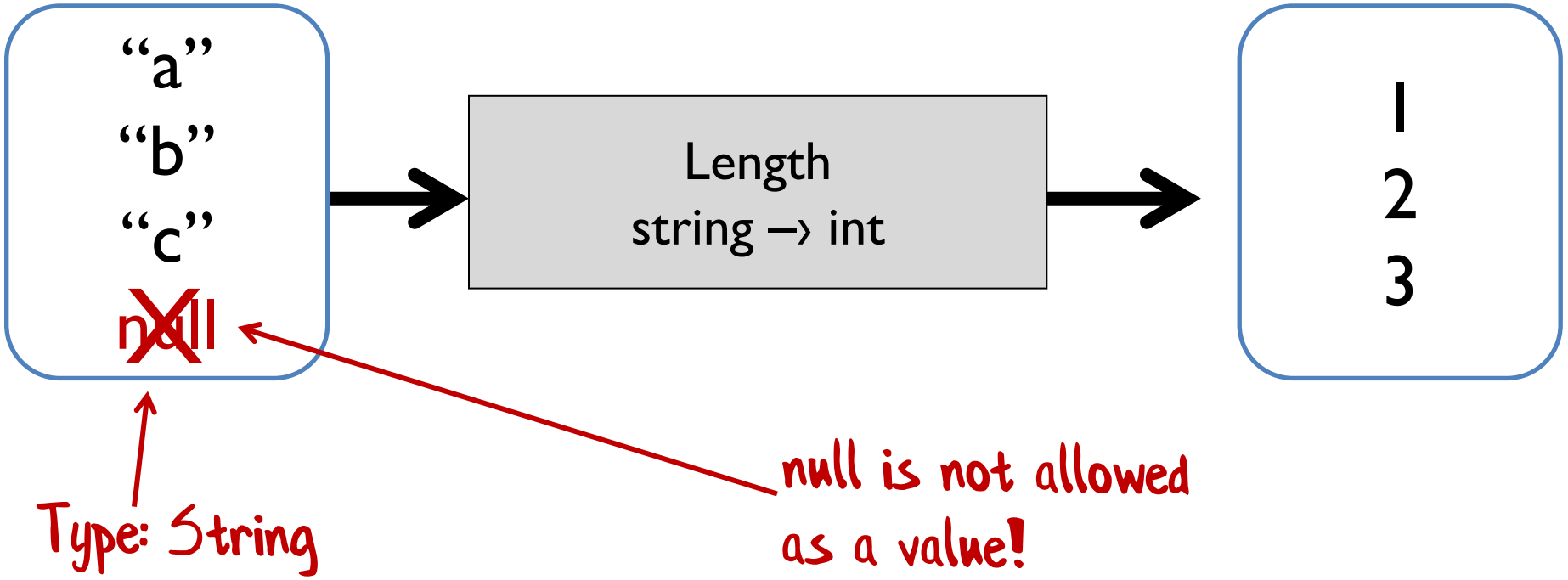

Spock, set phasers to null!
That is illogical, Captain

Null is not the same as “optional”

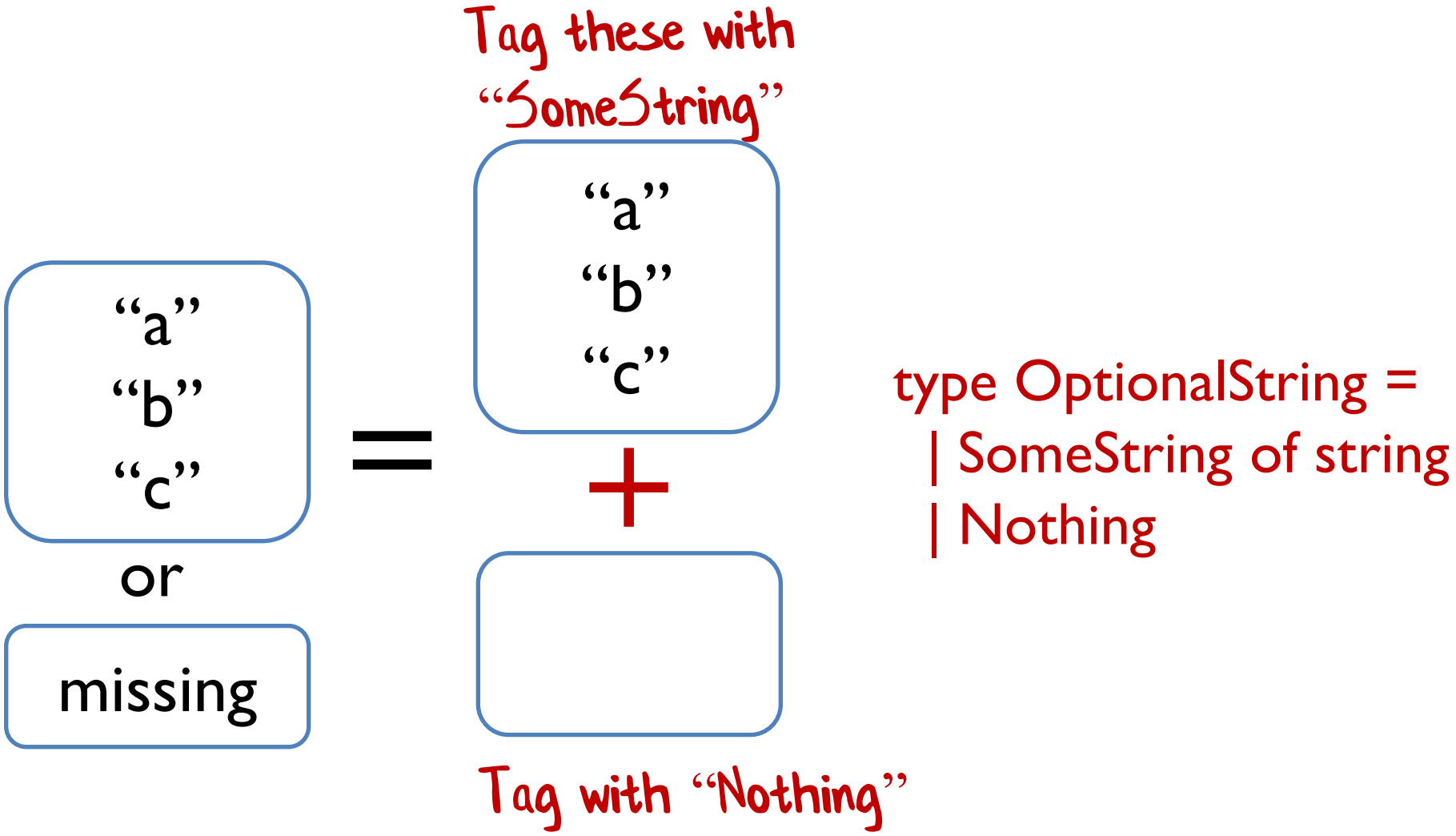


"null is the Saruman of static typing"

Null is not allowed



A better way for optional values



Defining optional types

```
type OptionalString =
    | SomeString of string
    | Nothing

type OptionalInt =
    | SomeInt of int
    | Nothing

type OptionalBool =
    | SomeBool of bool
    | Nothing
```

Duplicate code?

The built-in “Option” type

```
type Option<'T> =
    | Some of 'T
    | None
```

generic type

```
type PersonalName =
    {
    FirstName: string
    MiddleInitial: string option
    LastName: string
    }
```

Option<string>

nice and readable!

Single choice types

Single choice types

```
type Something =
    | ChoiceA of A

type Email =
    | Email of string

type CustomerId =
    | CustomerId of int
```

One choice only?
Why?

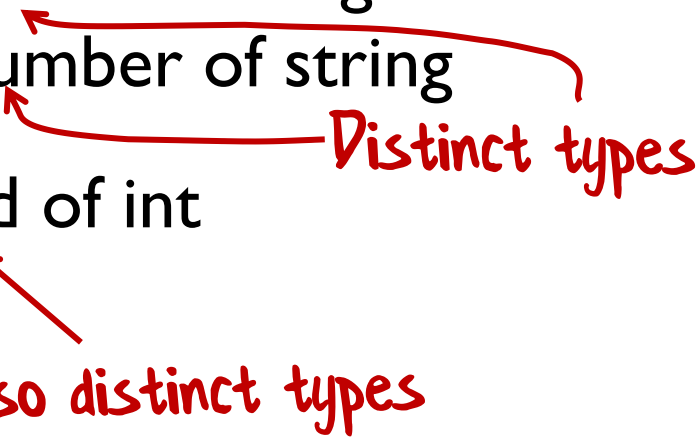
Wrapping primitive types

Is an EmailAddress just a string?

Is a CustomerId just a int?

Use single choice types to keep them distinct

```
type EmailAddress = EmailAddress of string
type PhoneNumber = PhoneNumber of string

type CustomerId = CustomerId of int
type OrderId = OrderId of int
```



Creating the EmailAddress type

```
let createEmailAddress (s:string) =
    if Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then Some (EmailAddress s)
        else None

createEmailAddress:
        string –› EmailAddress option
```

Constrained strings

```
type String50 = String50 of string

let createString50 (s:string) =
    if s.Length <= 50
        then Some (String50 s)
        else None

createString50 :
        string –› String50 option
```

Constrained numbers

What's wrong with this picture?

How could this happen?

Constrained numbers

How many people ever do this?

New type just for this domain

```
type OrderLineQty = OrderLineQty of int

let createOrderLineQty qty =
    if qty >0 && qty <= 99
        then Some (OrderLineQty qty)
        else None

createOrderLineQty:
        int –› OrderLineQty option
```

The challenge, revisited

The challenge, revisited

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

The challenge, revisited

```
type Contact = {

  FirstName: string
  MiddleInitial: string option
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

The challenge, revisited

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1  option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

The challenge, revisited

```
type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }
```

```
type EmailContactInfo = {
  EmailAddress: EmailAddress
  IsEmailVerified: bool }
```

```
type EmailContactInfo = {
  EmailAddress: EmailAddress
  IsEmailVerified: bool }
```

Rule 1: If the email is changed, the verified flag must be reset to false.

Rule 2: The verified flag can only be set by a special verification service

Encoding domain logic

"there is no problem that can't be solved by wrapping it in another type"

```
type VerifiedEmail = VerifiedEmail of EmailAddress

type VerificationService =
  (EmailAddress * VerificationHash) –› VerifiedEmail option

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

The challenge, completed

The challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

The challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Which values are optional?

What are the constraints?

Which fields are linked?

Domain logic clear?

The challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Which values are optional?

The challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

What are the constraints?

The challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Which fields are linked?

The challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1  option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Domain logic clear?

The challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

The ubiquitous language is evolving along with the design

(all this is compilable code, BTW)

Making illegal states unrepresentable

Making illegal states unrepresentable

```
type Contact = {
    Name: Name
    Email: EmailContactInfo
    Address: PostalContactInfo
    }
```

Added some time later

Making illegal states unrepresentable

New rule:

"*A contact must have an email or a postal address*"

```
type Contact = {
    Name: Name
    Email: EmailContactInfo
    Address: PostalContactInfo
    }
```

Doesn't meet new requirements

Making illegal states unrepresentable

New rule:

"*A contact must have an email or a postal address*"

```
type Contact = {
    Name: Name
    Email: EmailContactInfo option
    Address: PostalContactInfo option
    }
```

Doesn't meet new requirements either

Could both be missing?

"Make illegal states unrepresentable!"

– Yaron Minsky

Making illegal states unrepresentable

"*A contact must have an email or a postal address*"

implies:

- email address only, *or*
- postal address only, *or*
- both email address and postal address

only three possibilities

Making illegal states unrepresentable

"A contact must have an email or a postal address"

```
type ContactInfo =
   | EmailOnly of EmailContactInfo
   | AddrOnly of PostalContactInfo
   | EmailAndAddr of EmailContactInfo * PostalContactInfo
```

requirements are now encoded in the type!

only three possibilities

```
type Contact = {
   Name: Name
   ContactInfo : ContactInfo  }
```

Making illegal states unrepresentable

"A contact must have an email or a postal address"

BEFORE: Email and address separate

```
type Contact = {
   Name: Name
   Email: EmailContactInfo
   Address: PostalContactInfo
}
```

AFTER: Email and address merged into one type

```
type Contact = {
   Name: Name
   ContactInfo : ContactInfo  }

type ContactInfo =
    | EmailOnly of EmailContactInfo
    | AddrOnly of PostalContactInfo
    | EmailAndAddr of
        EmailContactInfo * PostalContactInfo
```

Static types are almost as awesome as this

Making illegal states unrepresentable

Is this really what the business wants?

"A contact must have at least one way of being contacted"

```
type ContactInfo =
    | Email of EmailContactInfo
    | Addr of PostalContactInfo
```

Way of being contacted

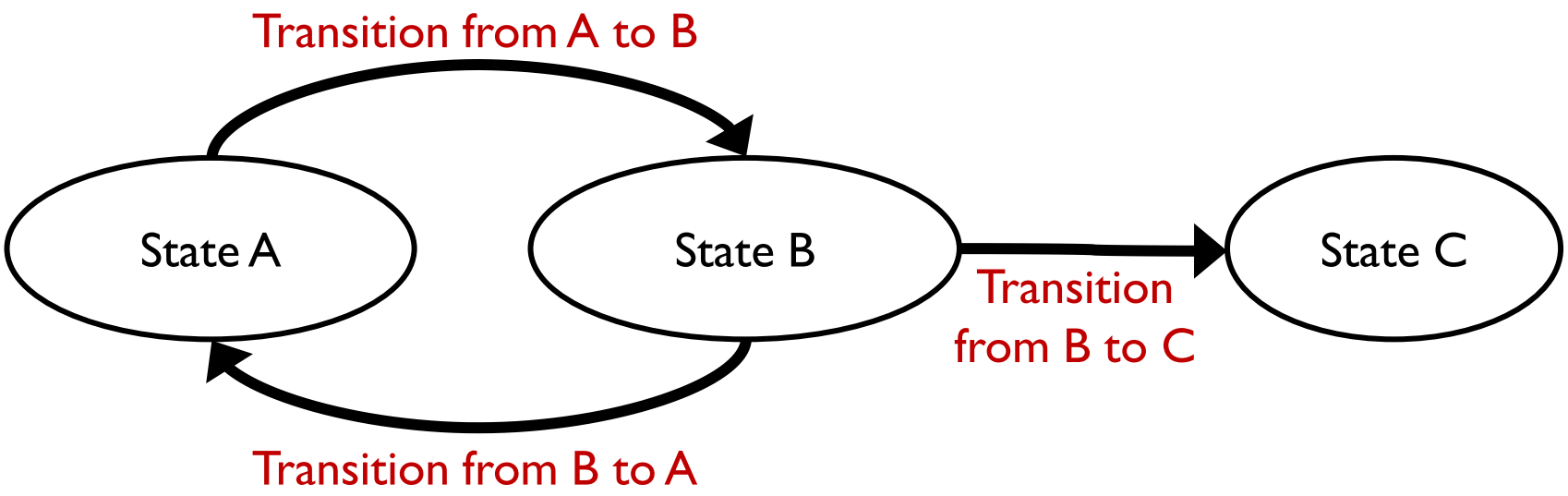
```
type Contact = {
   Name: Name
   PrimaryContactInfo: ContactInfo
   SecondaryContactInfo: ContactInfo option }
```

At least one way of being contacted is required

States and Transitions

Modelling a common scenario

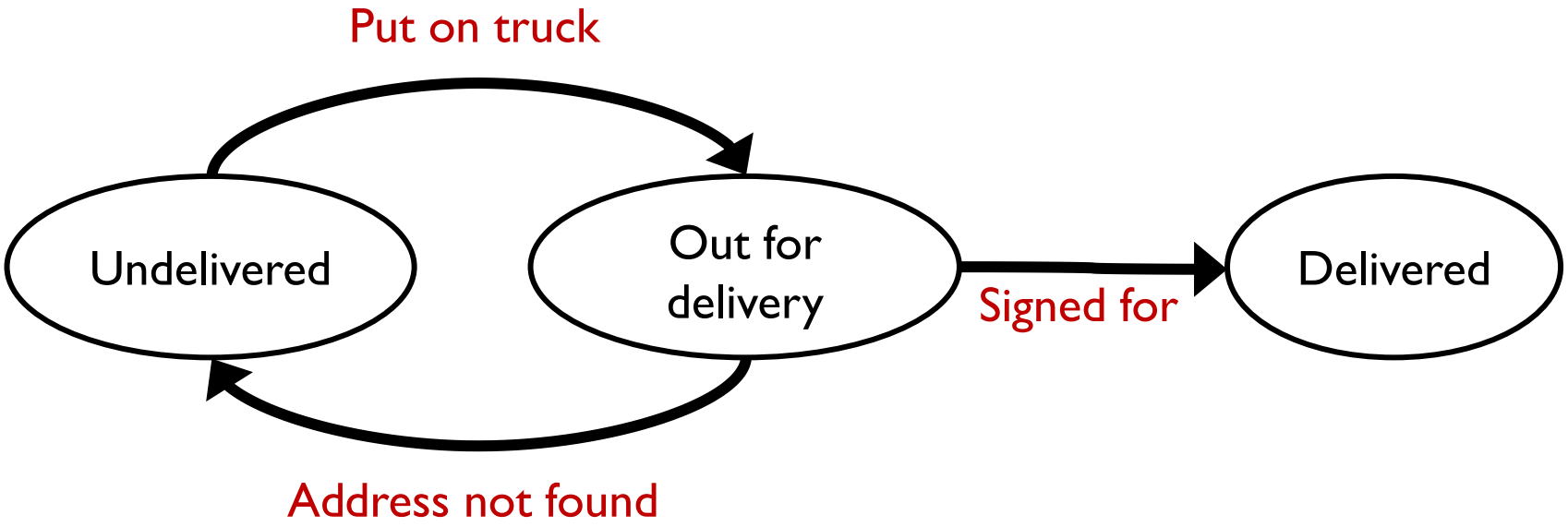
States and transitions



States and transitions for email address

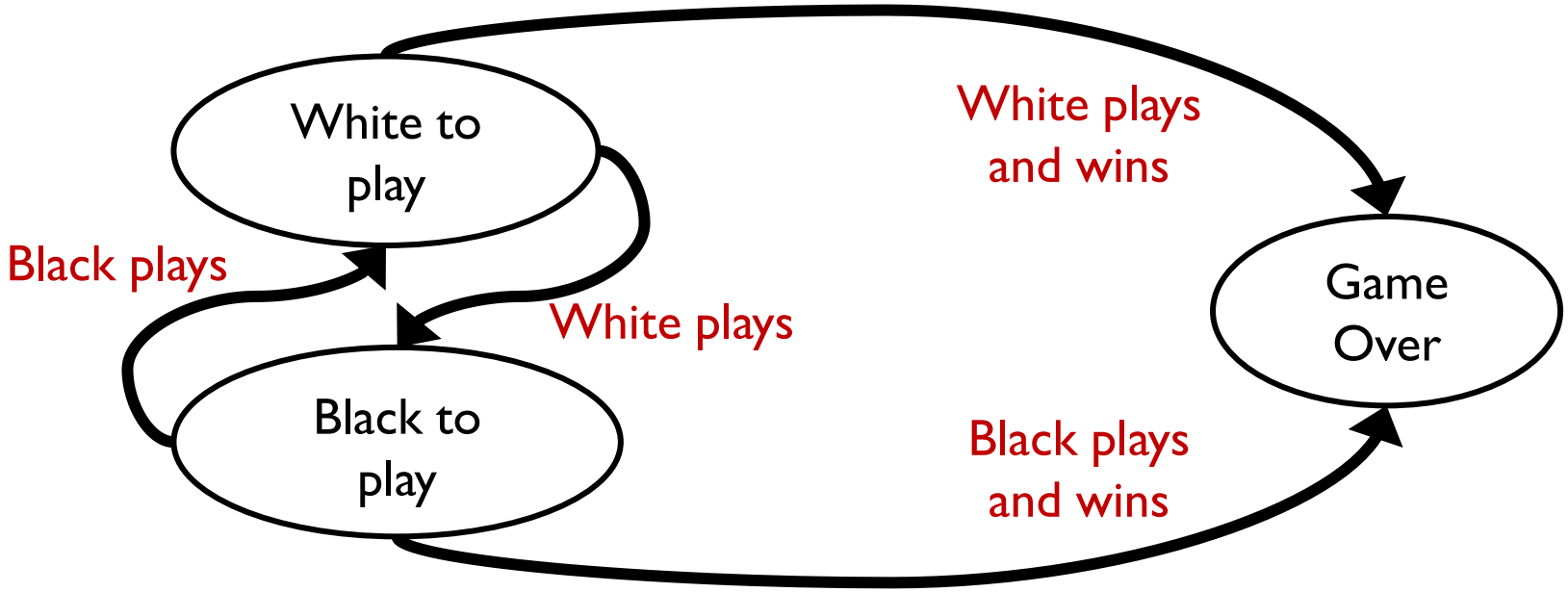
Rule: "You can't send a verification message to a verified email"
Rule: "You can't send a password reset message to a unverified email "

States and transitions for shipments



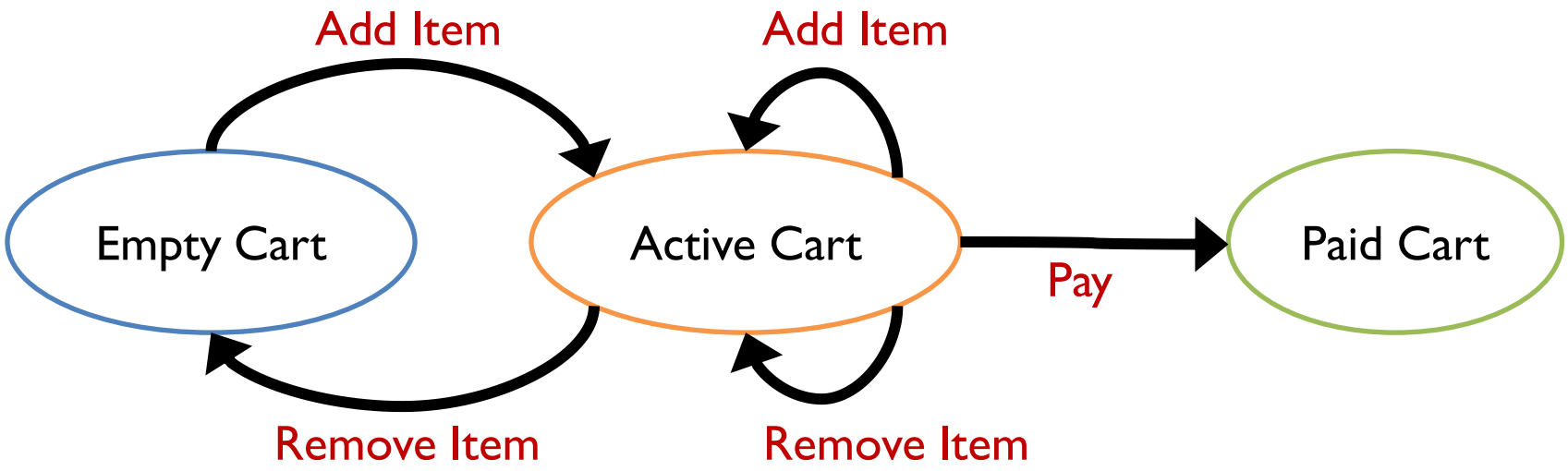
Rule: "You can't put a package on a truck if it is already out for delivery"
Rule: "You can't sign for a package that is already delivered"

States and transitions for chess game



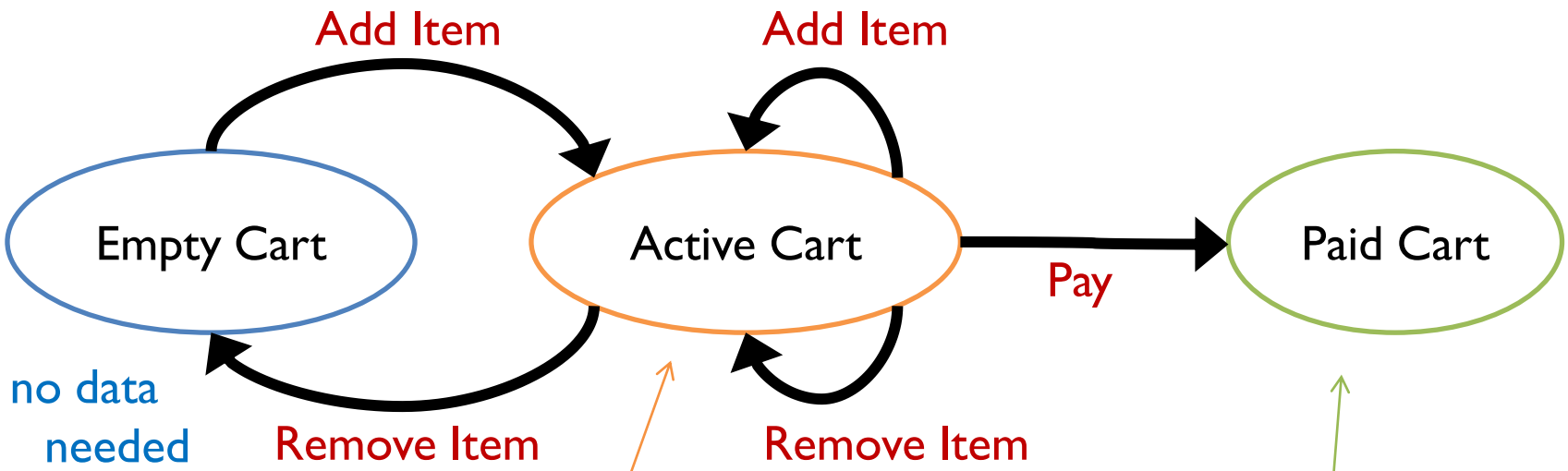
Rule: "White and Black take turns playing.
White can't play if it is Black's turn and vice versa"
Rule: "No one can play when the game is over"

States and transitions for shopping cart



Rule: "You can't remove an item from an empty cart"
Rule: "You can't change a paid cart"
Rule: "You can't pay for a cart twice"

States and transitions for shopping cart

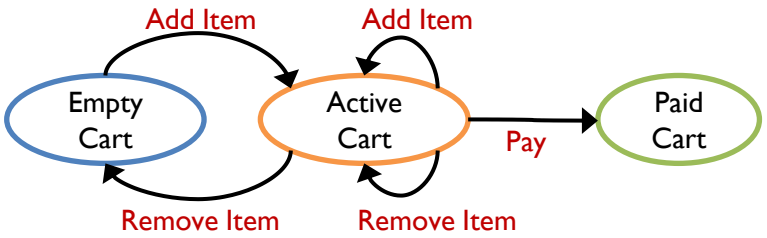


type ActiveCartData =
{ UnpaidItems: Item list }

type PaidCartData =
{ PaidItems: Item list;
Payment: Payment }

What data do we need to store?

Shopping cart example



Shopping Cart API

initCart :
Item –› ShoppingCart

addToActive:
(ActiveCartData * Item) –› ShoppingCart

removeFromActive:
(ActiveCartData * Item) –› ShoppingCart

might be empty or active – can't tell

pay:
(ActiveCartData * Payment) –› ShoppingCart

Shopping cart example

Server code to add an item

```
let initCart item =
    { UnpaidItems=[item] }
```

create a new ActiveCart with list of one item

```
let addToActive (cart:ActiveCart) item =
    { cart with UnpaidItems = item :: cart.existingItems }
```

Prepends item to list

Shopping cart example

Client code to add an item using the API

```
let addItem cart item =
    match cart with
    | EmptyCart –›
        initCart item
    | ActiveCart activeData –›
        addToActive(activeData,item)
    | PaidCart paidData –›
        ???
```

Cannot accidentally alter a paid cart!

Shopping cart example

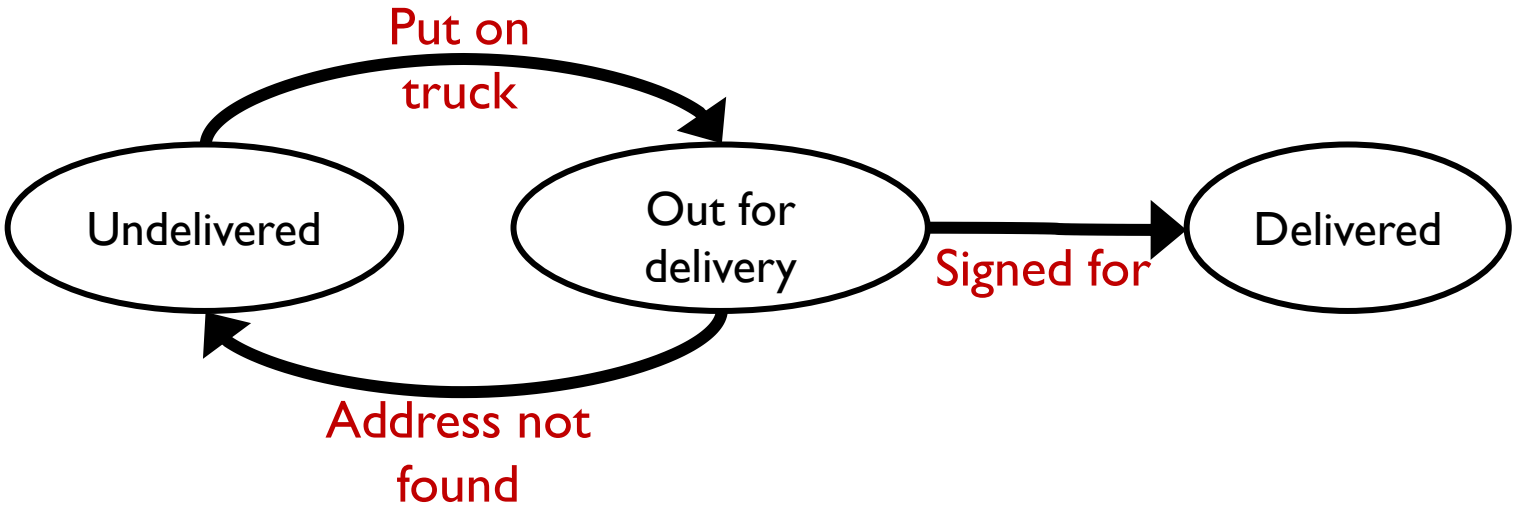
Client code to remove an item using the API

```
let removeItem cart item =
    match cart with
    | EmptyCart –›
        ???
    | ActiveCart activeData –›
        removeFromActive(activeData,item)
    | PaidCart paidData –›
        ???
```

Compiler will not let you remove from an empty cart!

Why design with state transitions?

- Each state can have different allowable data.
- All states are explicitly documented.
- All transitions are explicitly documented.
- It is a design tool that forces you to think about every possibility that could occur.



Summary: What types are good for

- Types as executable documentation
  - Ubiquitous language
  - Design and code are synchronized
  - Code is understandable by domain expert
- Types for accurate domain modelling
  - Constraints are explicit
- Types can encode business rules
  - Illegal states can be made unrepresentable

Review

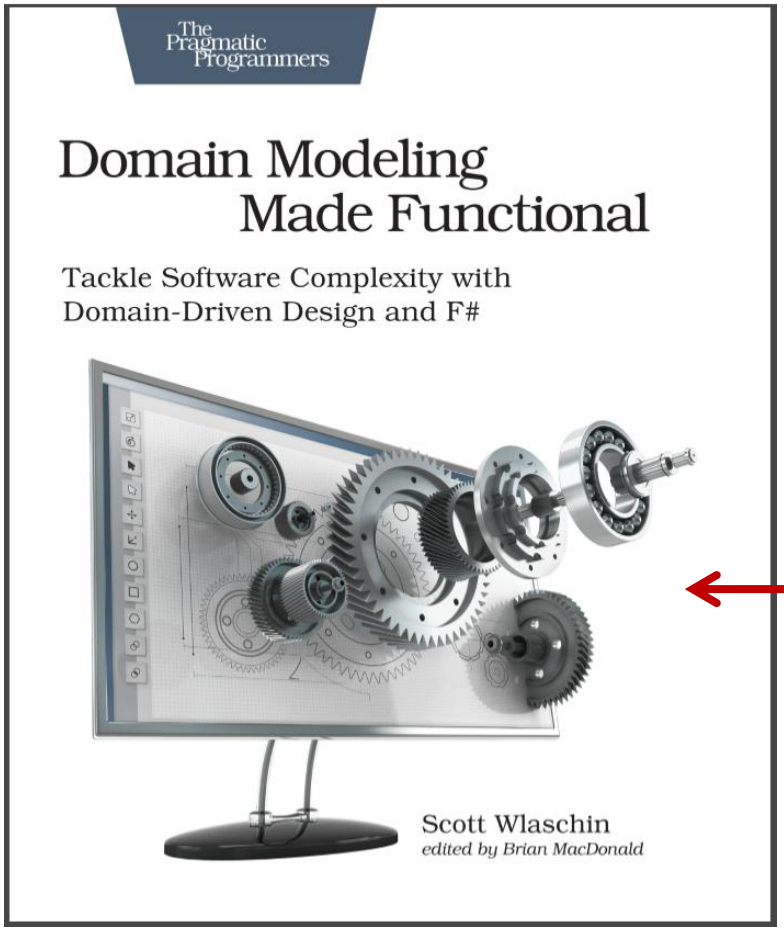
What we covered in this talk:

- Ubiquitous language
  - Self-documenting designs
- Algebraic types
  - products and sums
- Designing with types
  - Options instead of null
  - Single case unions
  - Choices rather than inheritance
  - Making illegal states unrepresentable
- States and transitions

Domain Modeling Made Functional

fsharpforfunandprofit.com/ddd

Slides and video here



I have a book coming soon!

More F# at fsharp.org

fsharp.org