```
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?

Find out -> Domain Driven Design with the F# type system

```
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?

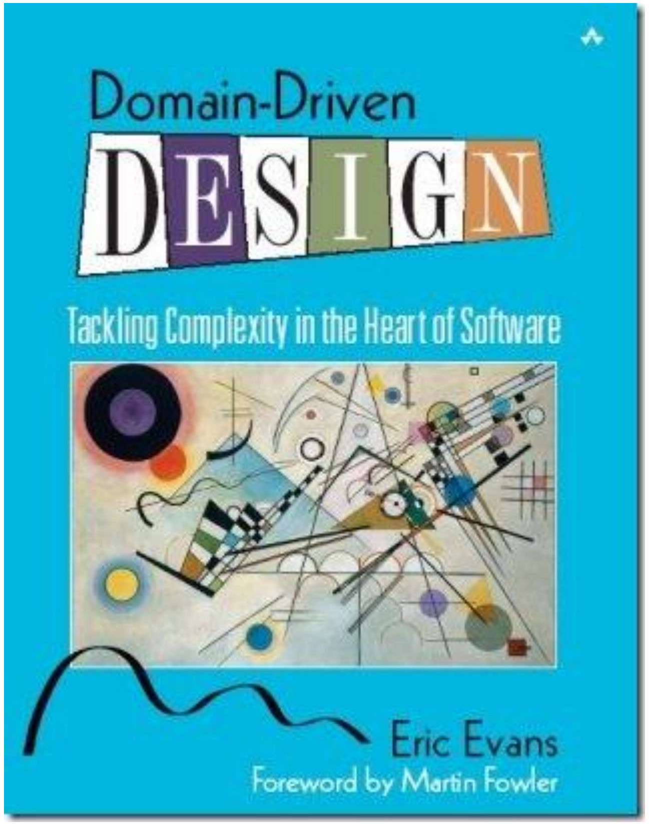
F# can help with all these questions!

Domain Driven Design with the F# type system

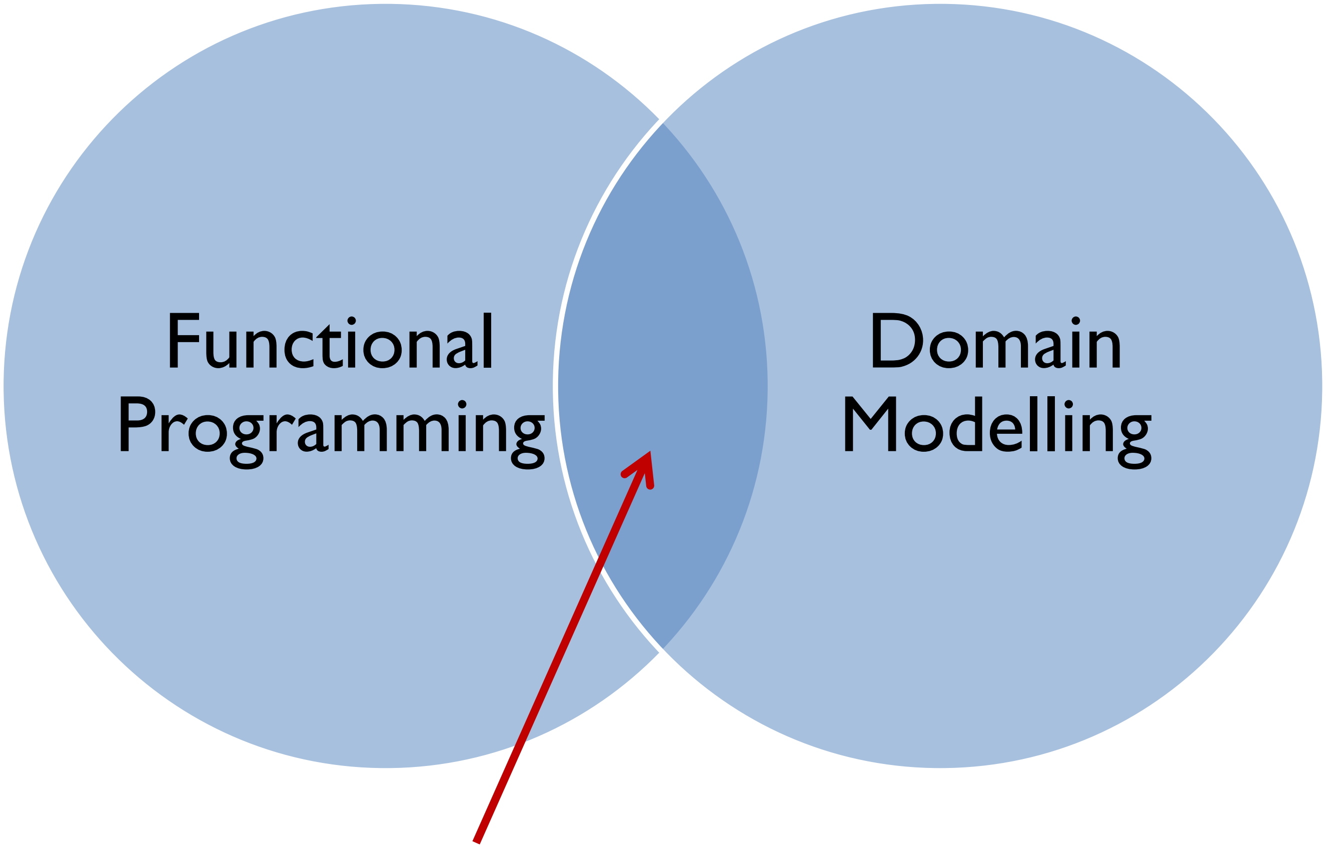
Scott Wlaschin
@ScottWlaschin

fsharpforfunandprofit.com /ddd
FPbridge.co.uk

What is DDD?


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

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



Functional Programming

Domain Modelling

This talk

What I'm going talk about:

- Demystifying functional programming
- Functional programming for real world applications
- F# vs. C# for domain driven design
- Understanding the F# type system
- Designing with types

Demystifying functional programming

Why is it so hard?

Functional programming is scary

Functor

Currying

Catamorphism

Applicative

Monad

Monoid

Functional programming is ~~scary~~ unfamiliar

“Mappable”

Currying

“Collapsable”

Applicative

“Chainable”

“Aggregatable”

This!

Object oriented programming is scary

Generics

Interface

Polymorphism

Inheritance

SRP, OCP, LSP, ISP, DIP, Oh noes..

SOLID

Covariance

...don't forget IoC, DI, ABC, MVC, etc., etc...

Functional programming is scary

Functor
Currying
Catamorphism
Applicative
Monad
Monoid

YAGNI!

Functional programming for real world applications

I've heard that...

Functional programming is...

... good for mathematical and scientific tasks

... good for complicated algorithms

... *really* good for parallel processing ← All true...

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

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

F# has a REPL and
many conveniences to
avoid boilerplate

- High quality deliverables

F# type system ensures
correctness

- Fun

"fun" is a keyword in #fsharp

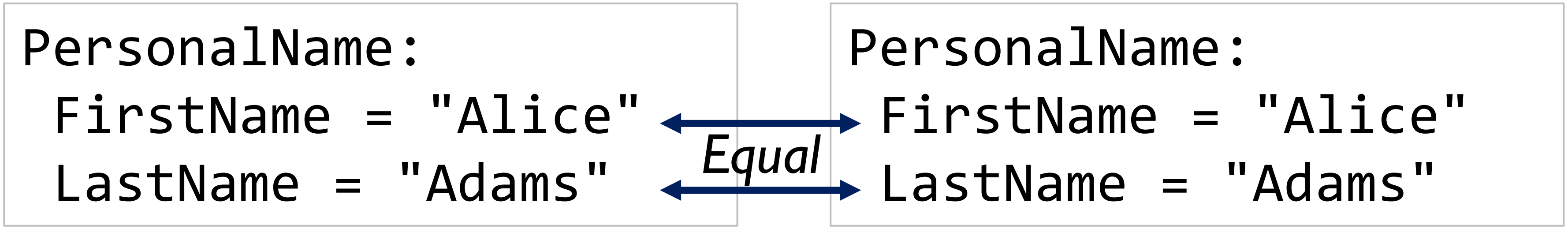
#fsharp <3 #bloba development!

F# vs. C#
for Domain Driven Design

A simple immutable object

How do you implement a Value object?

Equality based on comparing all properties



```
PersonalName:
 FirstName = "Alice"
 LastName = "Adams"
```

Equal

```
PersonalName:
 FirstName = "Alice"
 LastName = "Adams"
```

→ Therefore must be immutable

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; }
}
```

use "private set" for immutability

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}
```

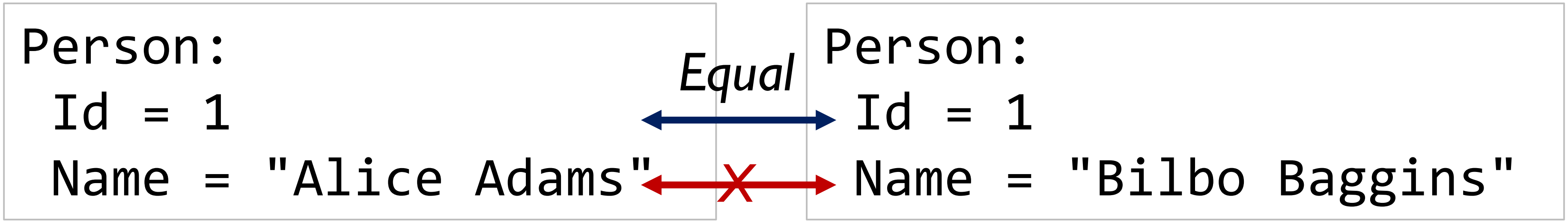
Value object definition in F# (extra code for equality)

This page intentionally left blank

the best code is no code
at all

How do you implement an Entity object?

Equality based on some sort of id



```
Person:
 Id = 1
 Name = "Alice Adams"
```

Equal

```
Person:
 Id = 1
 Name = "Bilbo Baggins"
```

→ Generally has mutable content

Entity object definition in C# (part 1)

```
class Person
{
    public Person(int id, PersonalName name)
    {
        this.Id = id;
        this.Name = name;
    }

    public int Id { get; private set; }
    public PersonalName Name { get; set; }
}
```

removed private set

Entity object definition in C# (part 2)

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

    public override int GetHashCode()
    {
        return this.Id.GetHashCode();
    }

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

    public bool Equals(Person other)
    {
        if ((object) other == null)
        {
            return false;
        }
        return Id == other.Id;
    }
}
```

Compare on Id now...

Entity object definition in F# with equality override

```
[<CustomEquality; NoComparison>]
type Person = {Id:int; Name:PersonalName} with

    override this.GetHashCode() = hash this.Id

    override this.Equals(other) =
        match other with
        | :? Person as p -> (this.Id = p.Id)
        | _ -> false
```

If its a person...

...compare by Id

No null checking!

Entity object definition in F# with no equality allowed

even better

```
[<NoEquality; NoComparison>]
type Person = {Id:int; Name:PersonalName}
```

Advantages of immutability

```
type Person = { ... ... ... }
```

immutable

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

The only way to create an object

All changes must go through this checkpoint

Great for enforcing invariants in one place

Reviewing the C# code so far…

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

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

    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;
    }
}
```

```
class Person : IEntity
{
    public Person(int id, PersonalName name)
    {
        this.Id = id;
        this.Name = name;
    }

    public int Id { get; private set; }
    public PersonalName Name { get; set; }

    public override int GetHashCode()
    {
        return this.Id.GetHashCode();
    }

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

    public bool Equals(Person other)
    {
        if ((object) other == null)
        {
            return false;
        }
        return Id == other.Id;
    }
}
```

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

Do you think a non programmer could understand this?

Reviewing the F# code so far…

```
[<StructuralEquality;NoComparison>]
type PersonalName = {
    FirstName : string;
    LastName : string }
```

```
[<NoEquality; NoComparison>]
type Person = {
    Id : int;
    Name : PersonalName }
```

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

Do you think a non programmer could understand this?

Comparing C# vs. F#

C# vs. F# for DDD

	C#	F#
Value objects?	Non-trivial	Easy
Entity objects?	Non-trivial	Easy
Value objects by default?	No	Yes
Immutable objects by default?	No	Yes
Can you tell Value objects from Entities at a glance?	No	Yes
Understandable by non-programmer?	No	Yes

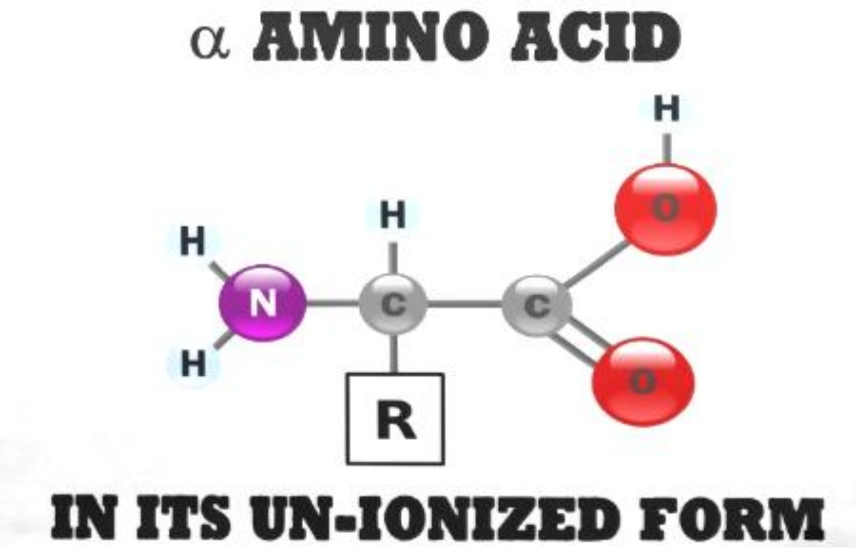
Very important thing for DDD!

F# for Domain Driven Design

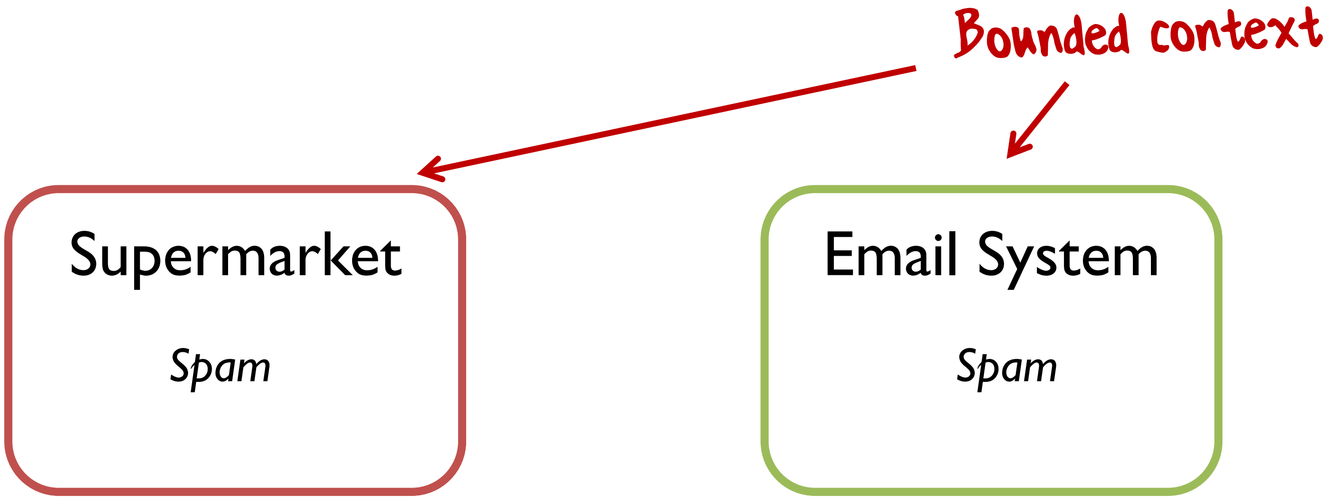
Communicating a domain model

Communication is hard...

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



Communication in DDD: “Bounded Context”



Bounded context

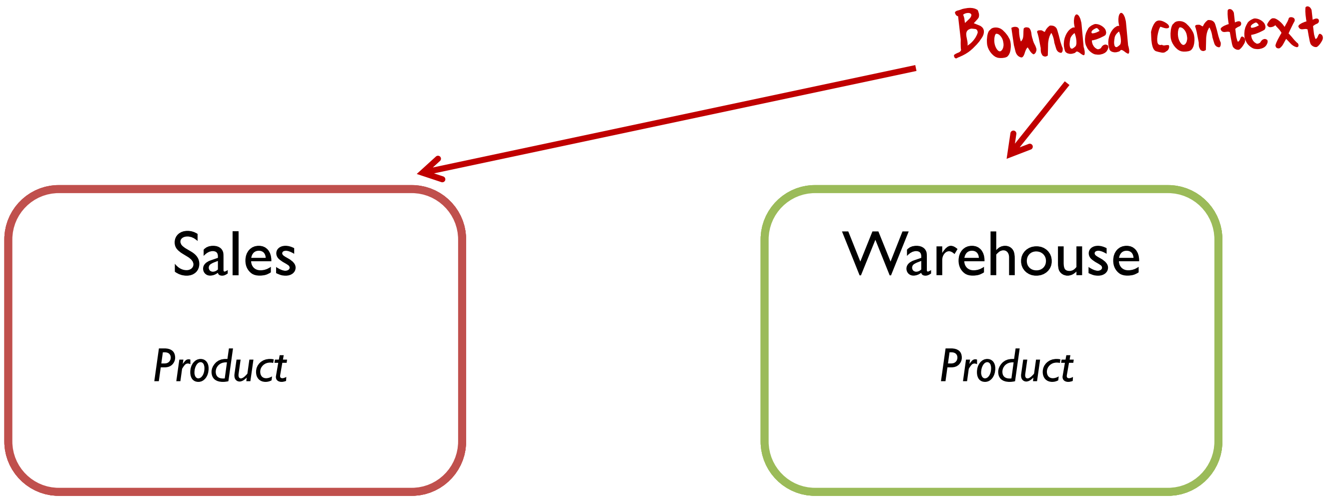
Supermarket

Spam

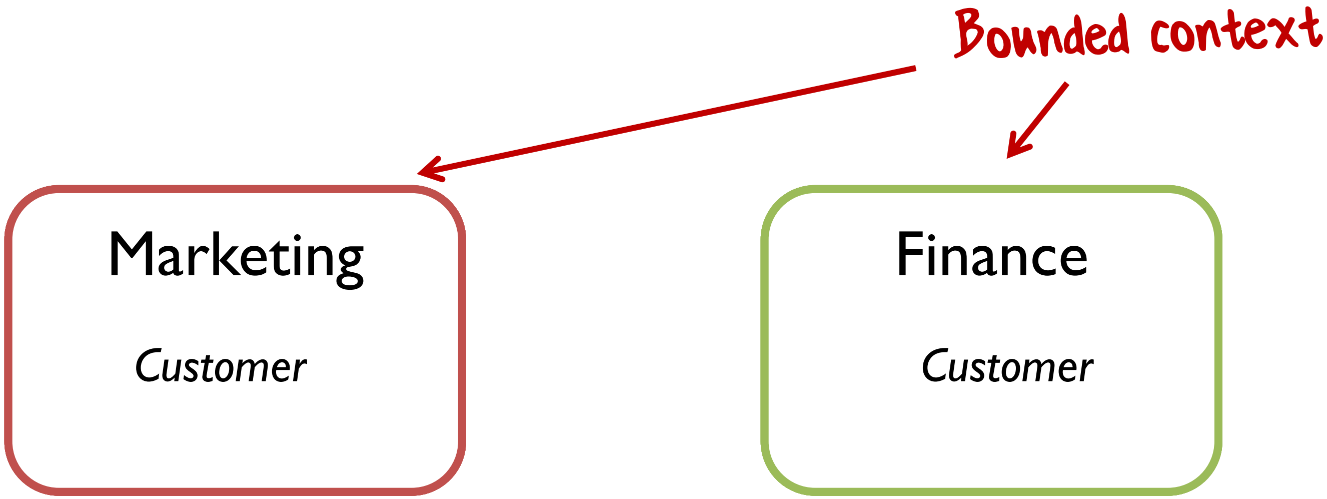
Email System

Spam

Communication in DDD: “Bounded Context”



Communication in DDD: "Bounded Context"



Bounded context

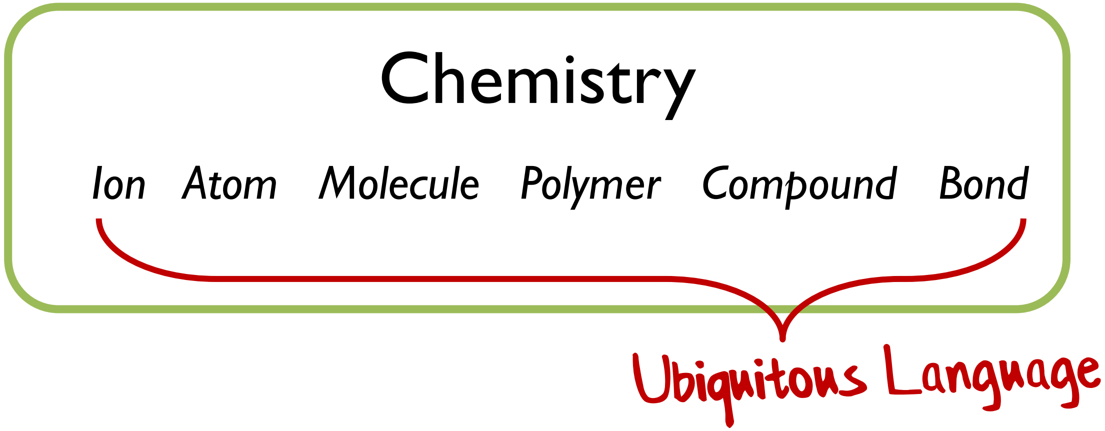
Marketing

Customer

Finance

Customer

Communication in DDD: “Ubiquitous Language”

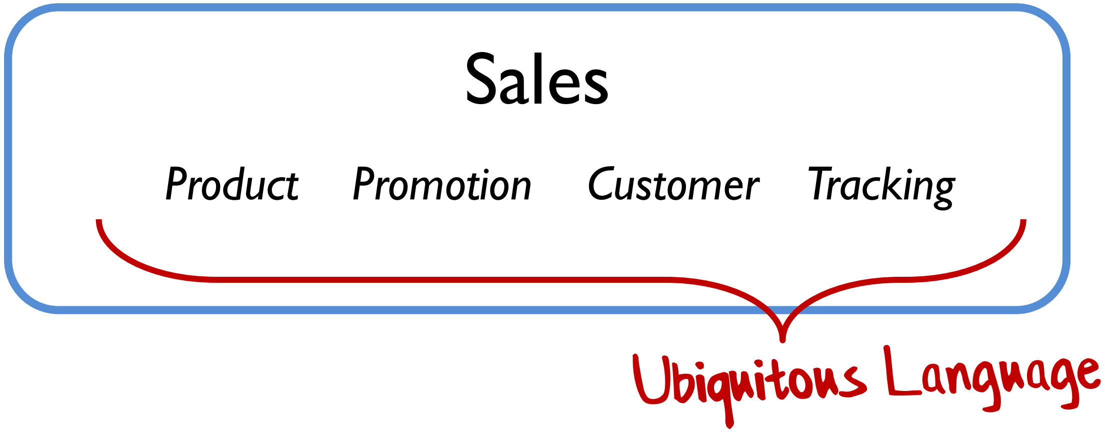


Chemistry

Ion *Atom* *Molecule* *Polymer* *Compound* *Bond*

Ubiquitous Language

Communication in DDD: “Ubiquitous Language”

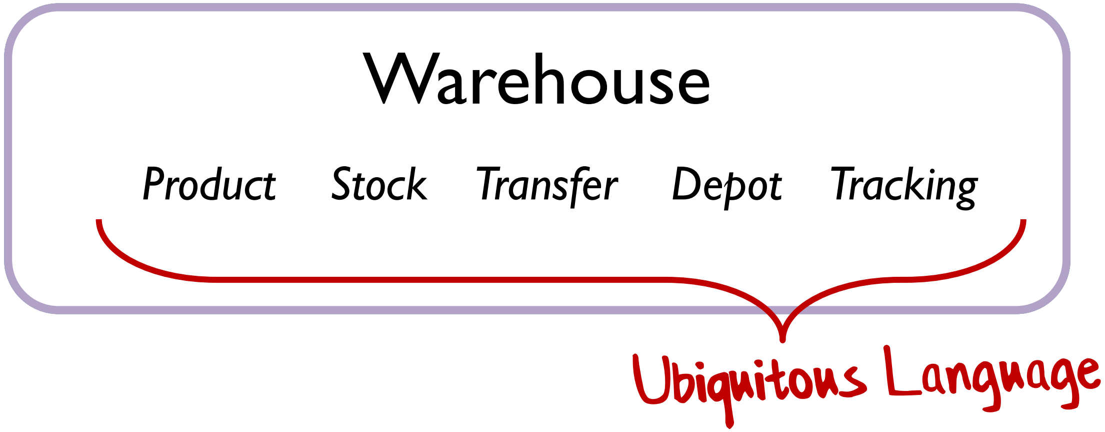


Sales

Product *Promotion* *Customer* *Tracking*

Ubiquitous Language

Communication in DDD: "Ubiquitous Language"



Warehouse

Product *Stock* *Transfer* *Depot* *Tracking*

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

    type Deal = Deck –› (Deck * Card)

    type PickupCard = (Hand * Card) –› Hand
```

Bounded context

'|' 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

Ubiquitous language

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

```
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 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!

WE DON'T NEED NO STINKING
UML DIAGRAMS

Understanding the F# type system

An introduction to “~~algebraic” types~~

"composable types"

Composable types

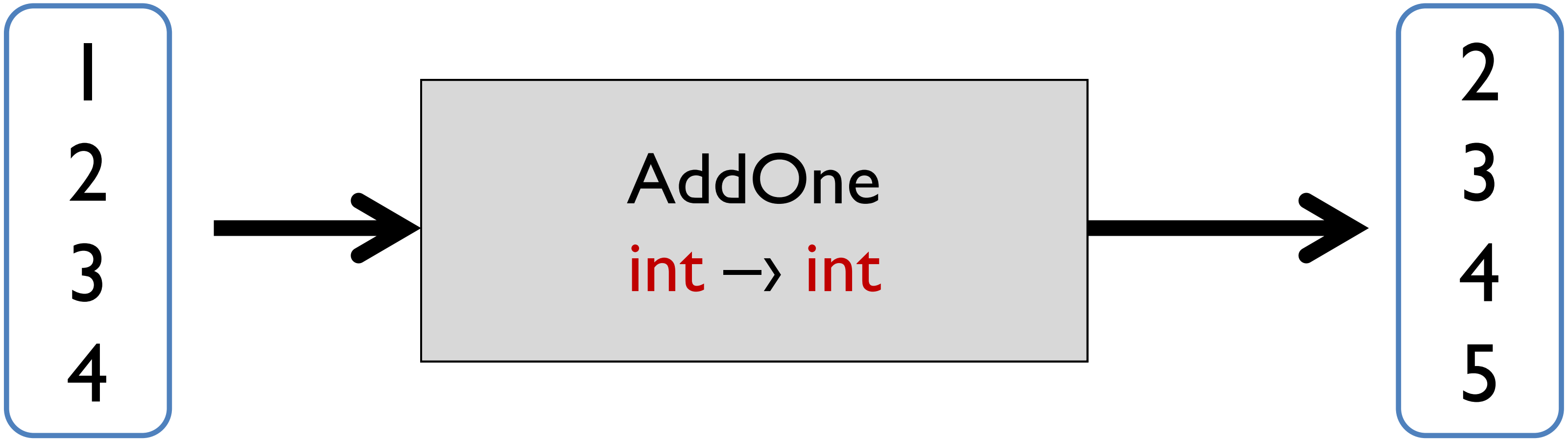
composable means => like Lego

Creating new types

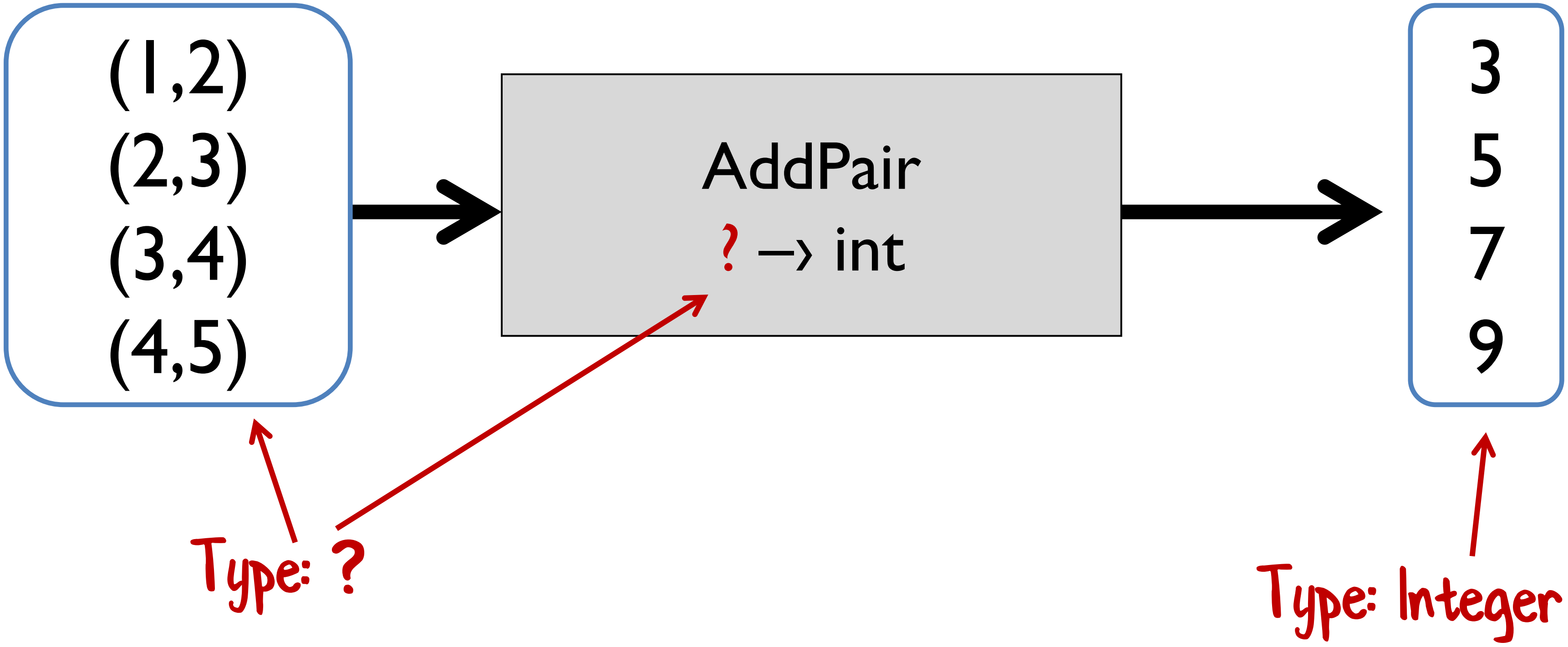
New types are constructed by combining other types using two basic operations:

```
type typeW = typeX "times" typeY
type typeZ = typeX "plus" typeY
```

Creating new types



Representing pairs



(1,2)
(2,3)
(3,4)
(4,5)

AddPair

? → int

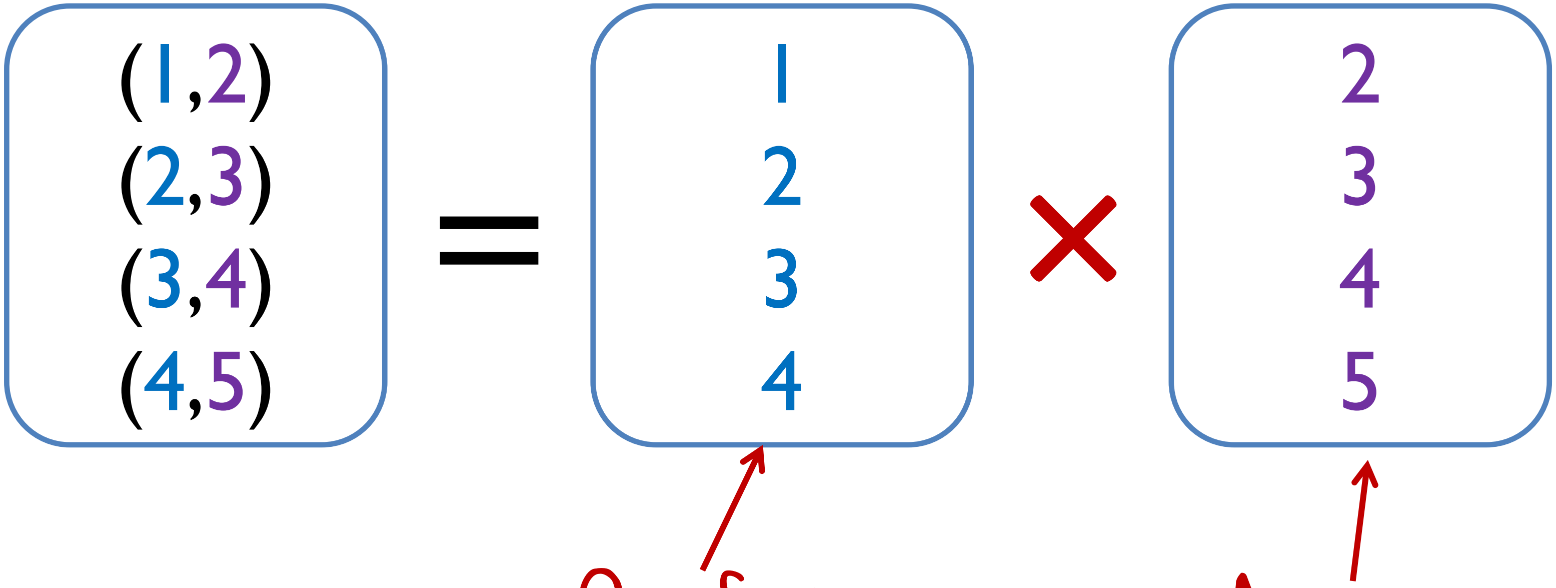
3
5
7
9

Type: ?

Type: Integer

How can we
represent this type?

Representing pairs



(1,2)
(2,3)
(3,4)
(4,5)

=

1
2
3
4

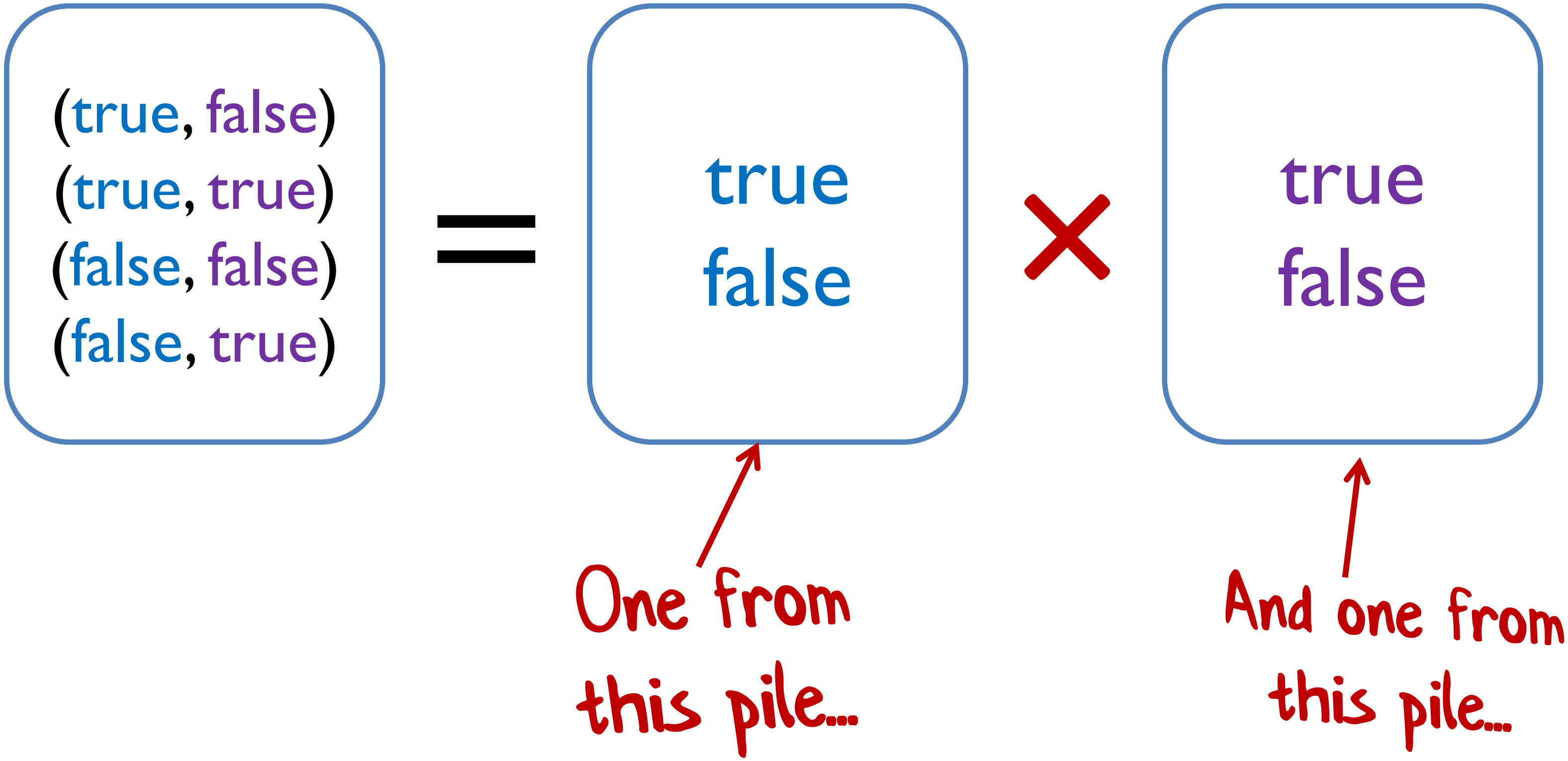
×

2
3
4
5

One from this pile...

And one from this pile...

Representing pairs



(true, false)
(true, true)
(false, false)
(false, true)

=

true
false

×

true
false

One from this pile...

And one from this pile...

Representing pairs

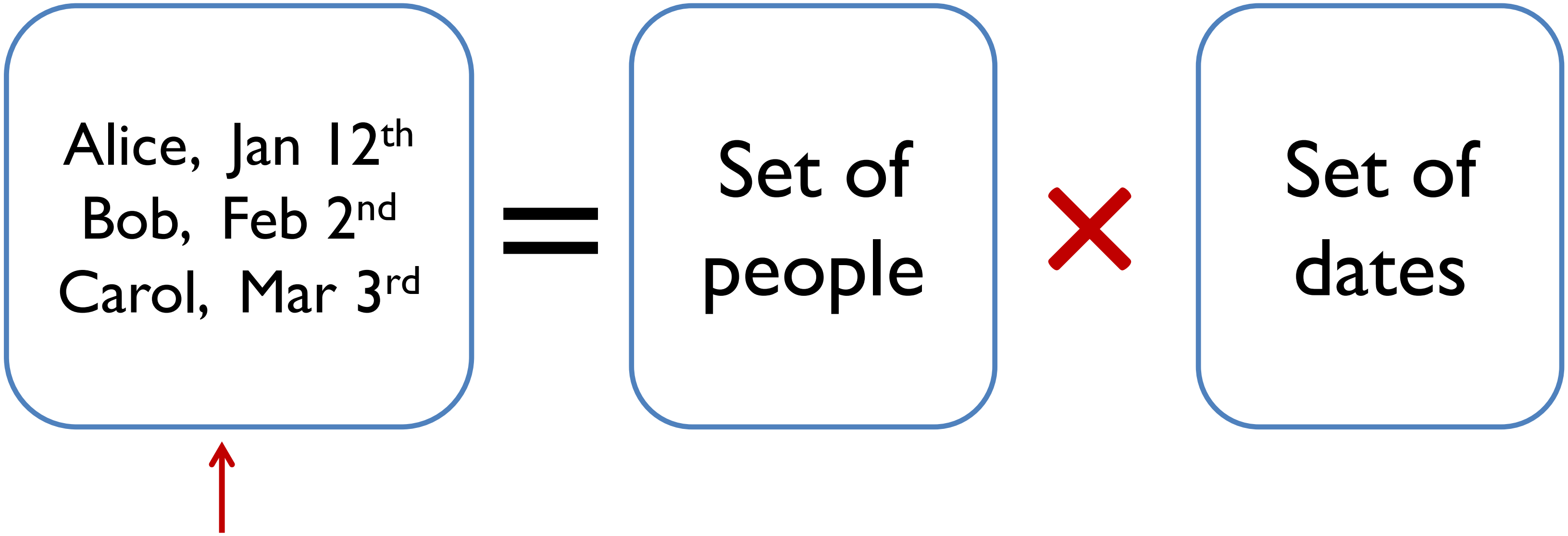
pair of ints

written int * int

pair of bools

written bool * bool

Using tuples for data



Alice, Jan 12th
Bob, Feb 2nd
Carol, Mar 3rd

=

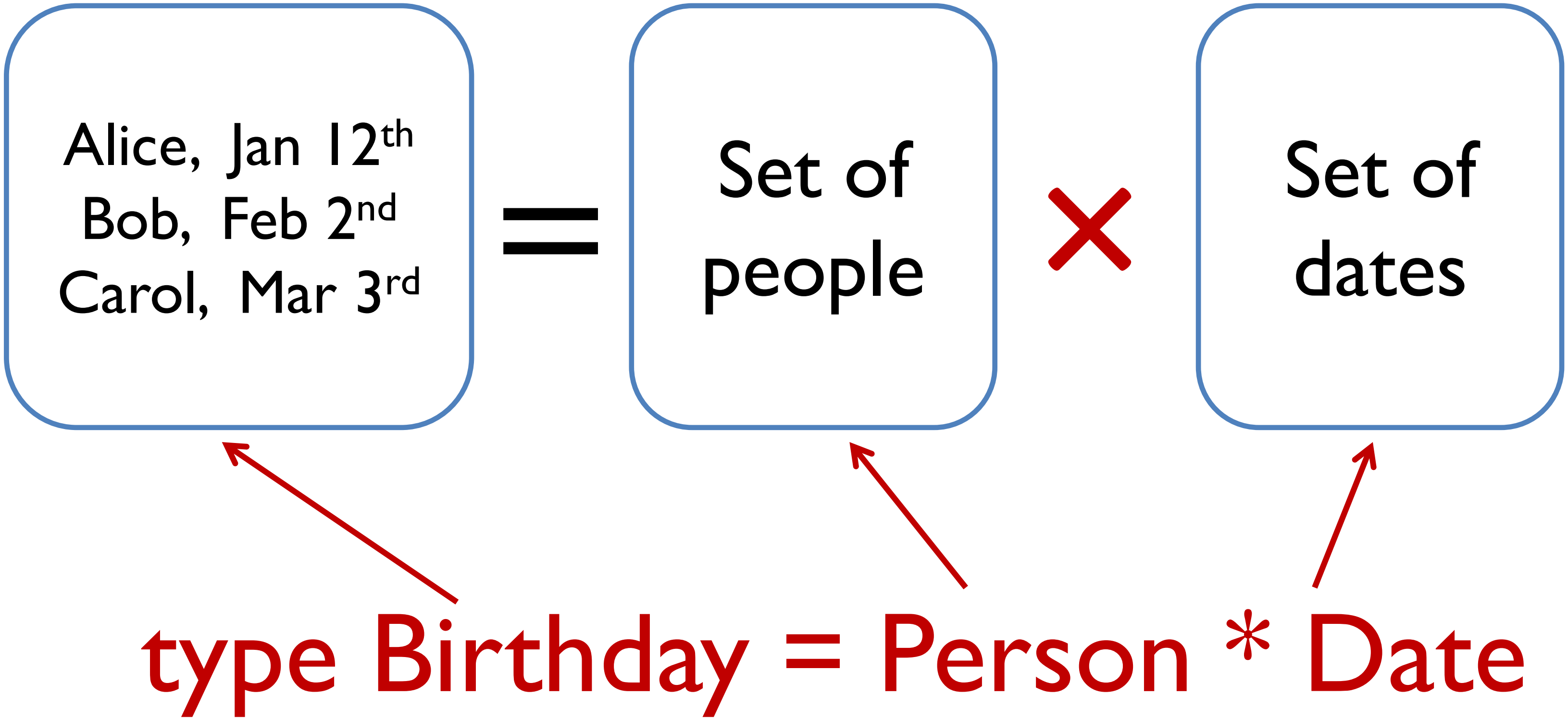
Set of people

×

Set of dates

How to represent this?

Using tuples for data



Alice, Jan 12th
Bob, Feb 2nd
Carol, Mar 3rd

=

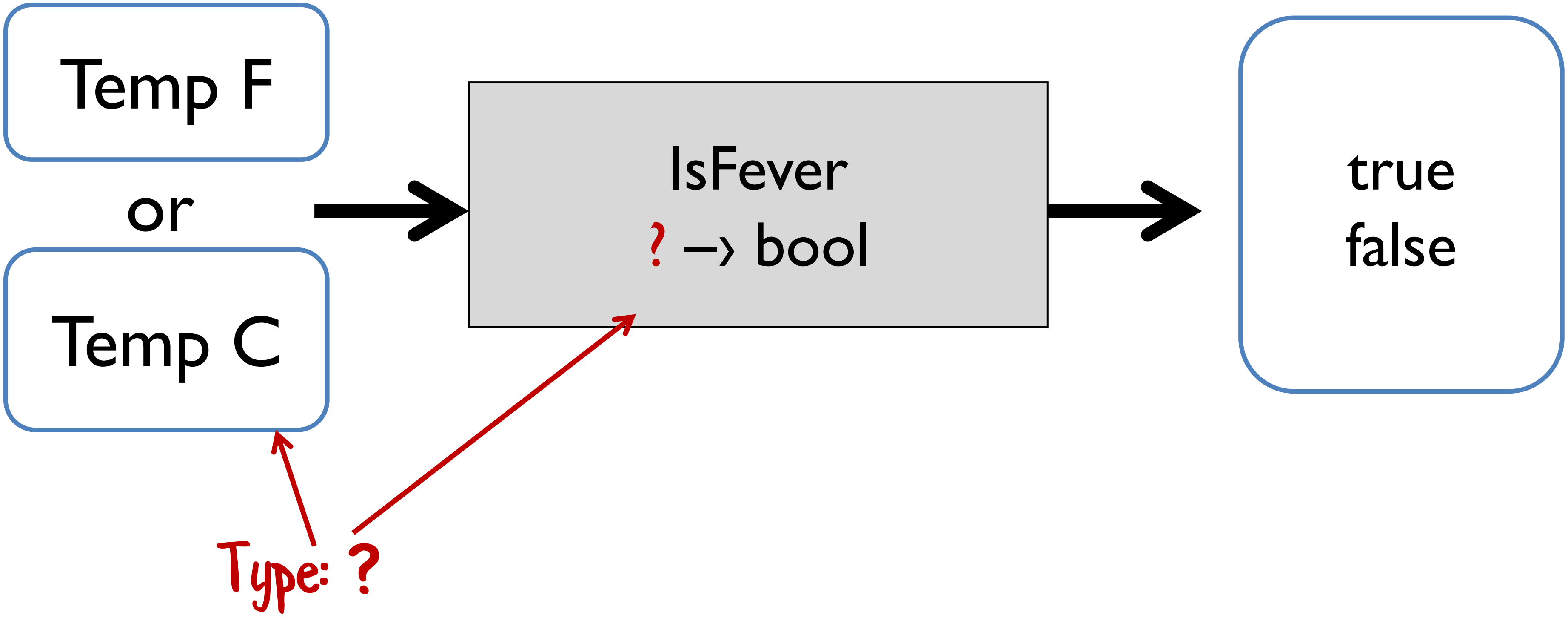
Set of people

×

Set of dates

type Birthday = Person * Date

Representing a choice



Temp F

or

Temp C

IsFever

? –› bool

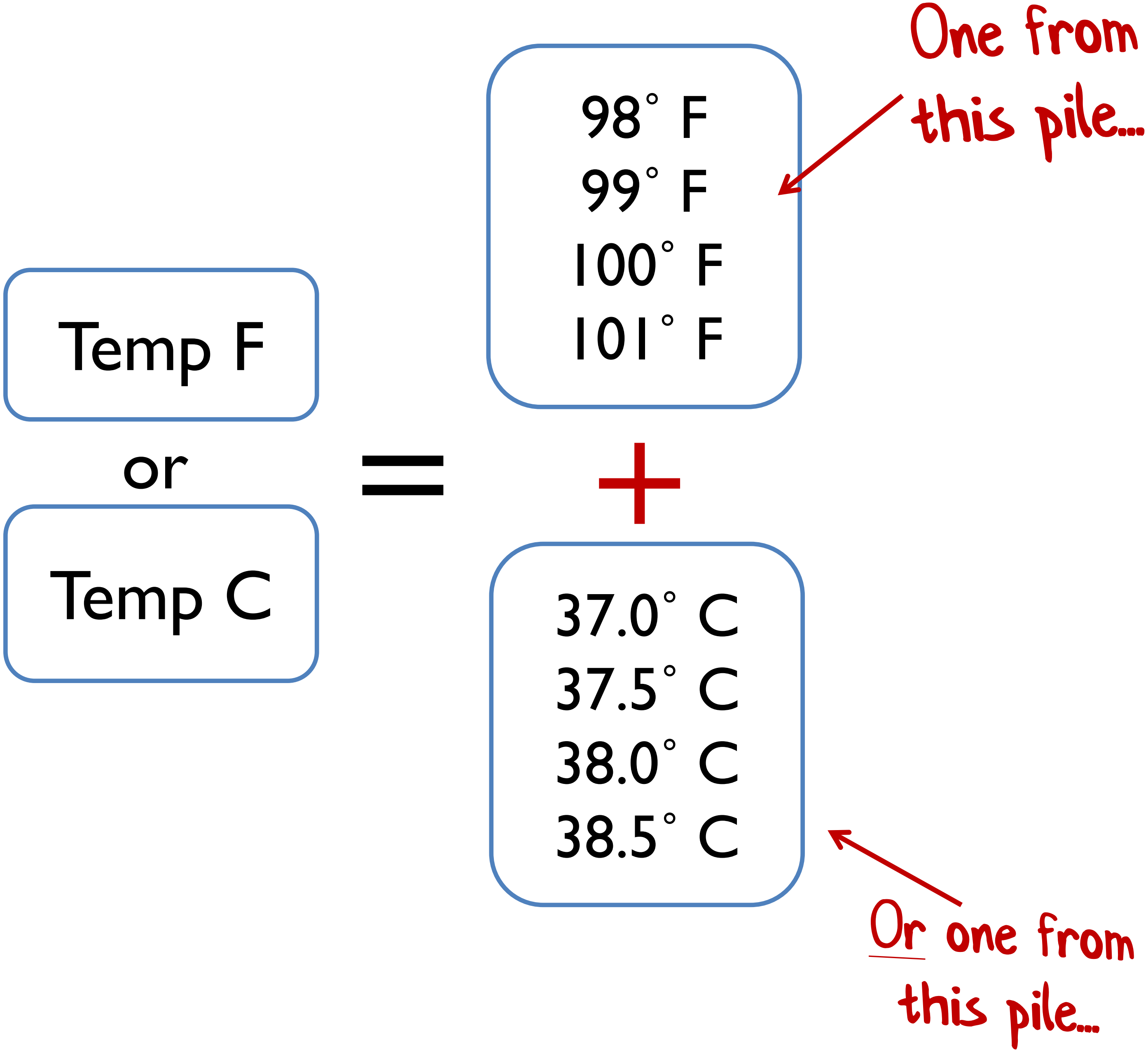
true

false

Type: ?

How can we represent this type?

Representing a choice



Temp F

or

Temp C

=

98° F
99° F
100° F
101° F

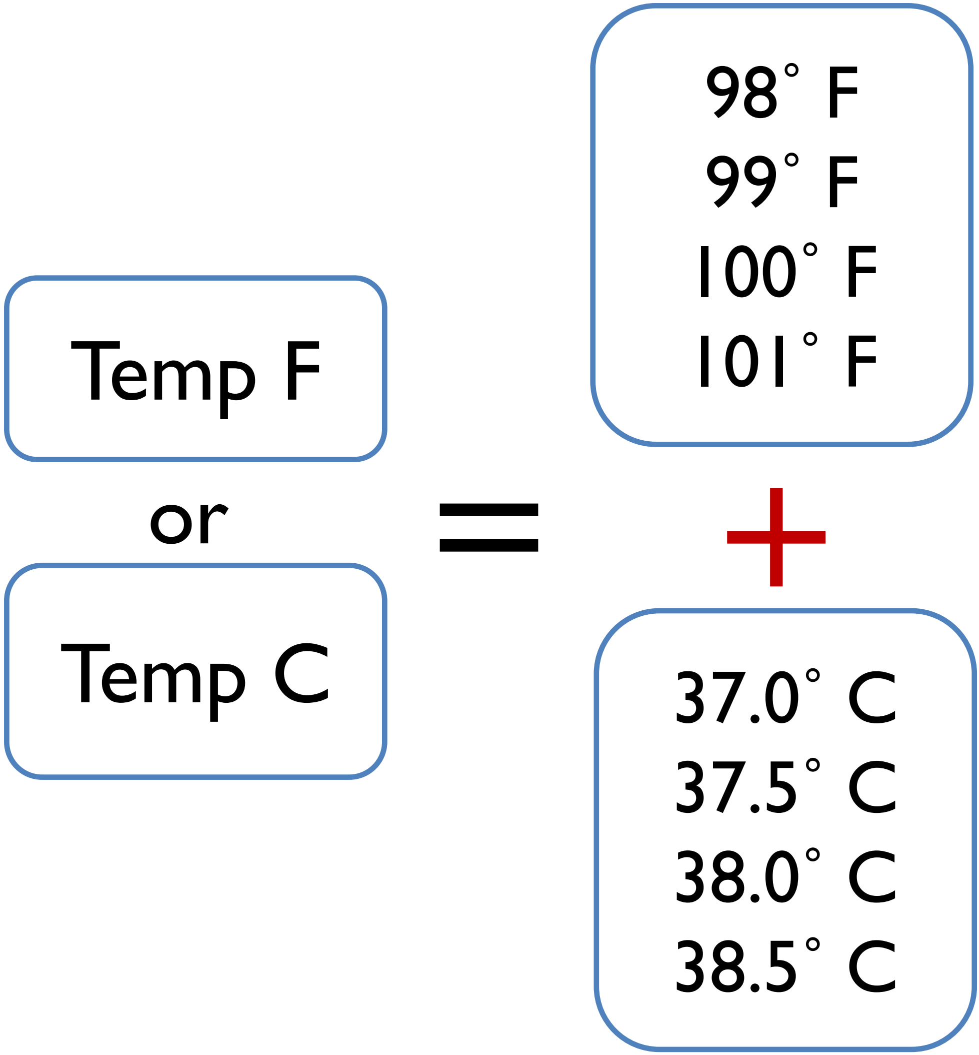
+

37.0° C
37.5° C
38.0° C
38.5° C

One from this pile...

Or one from this pile...

Representing a choice


Temp F

or

Temp C

=

98° F
99° F
100° F
101° F

+

37.0° C
37.5° C
38.0° C
38.5° C

Tag these with "F"

Tag these with "C"

```
type Temp =
  | F of int
  | C of float
```

Using choices for data

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

No extra data needed

Cheque no.

2 pieces of extra data

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
```

Match and assign in one step!

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)

both at once!

"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?

Required vs. Optional

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

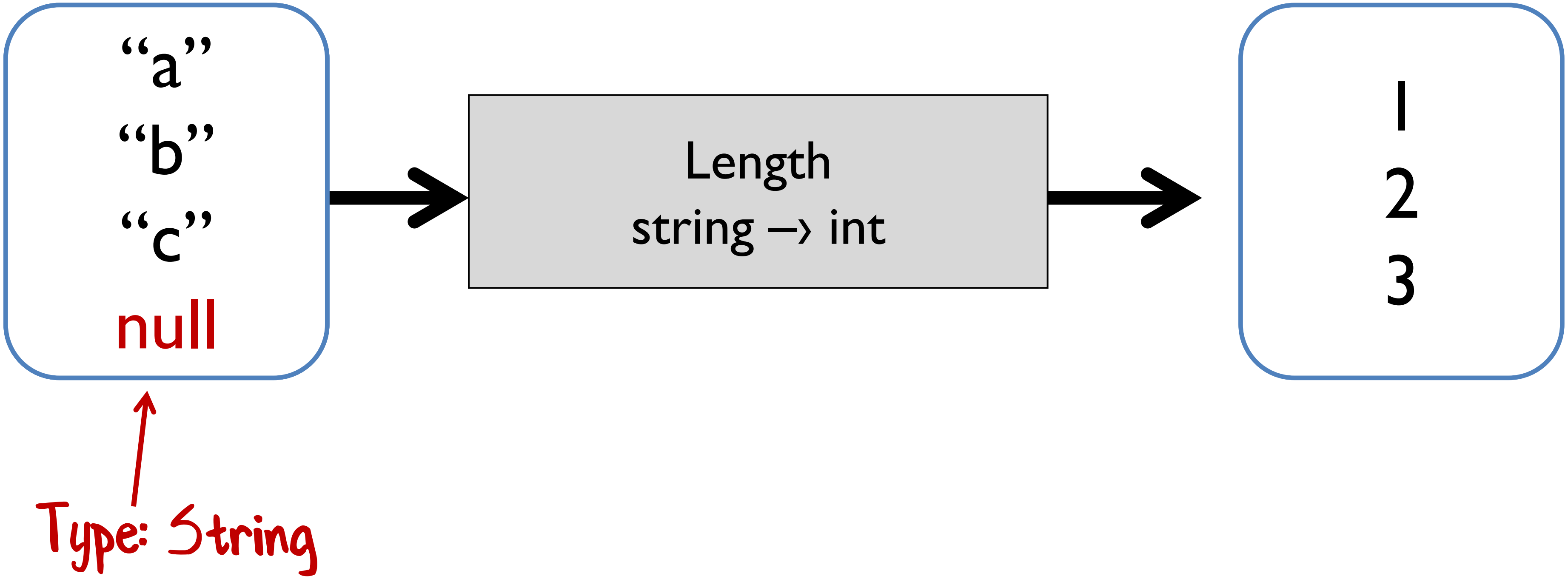
required

optional

required

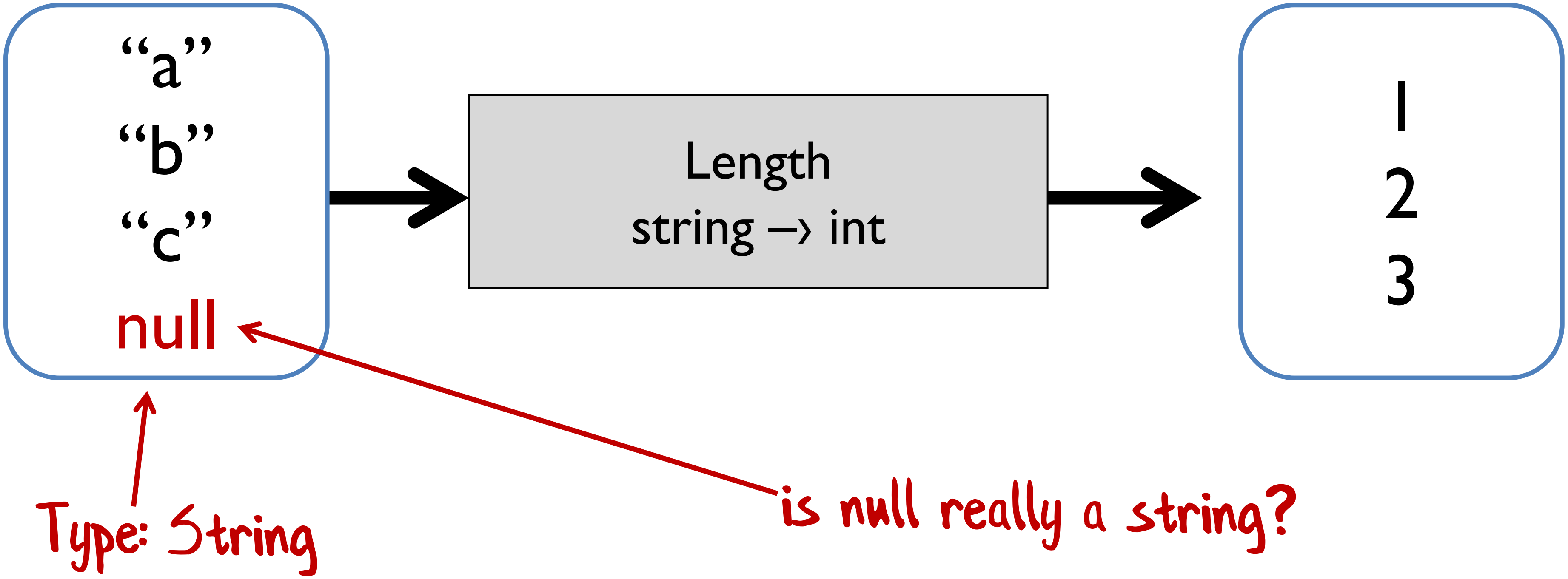
How can we represent optional values?

Null is not the same as “optional”



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

Null is not the same as "optional"



"a"
"b"
"c"
null

Length
string –› int

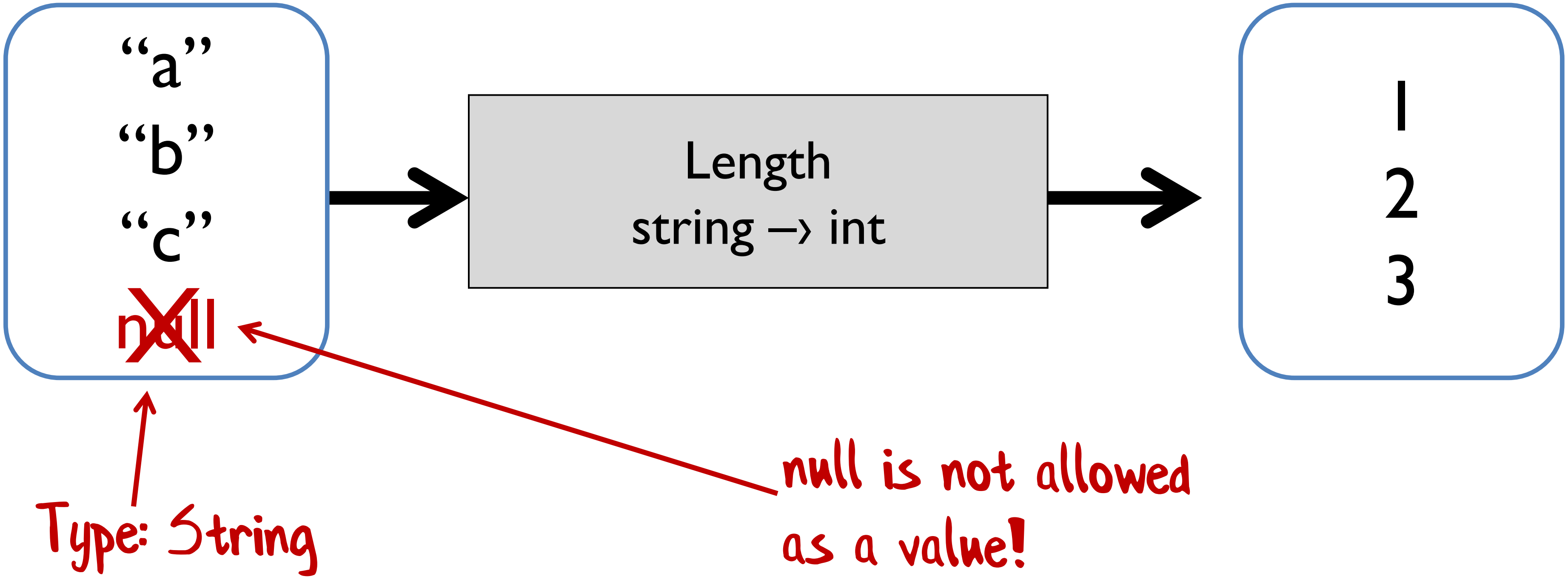
1
2
3

Type: String

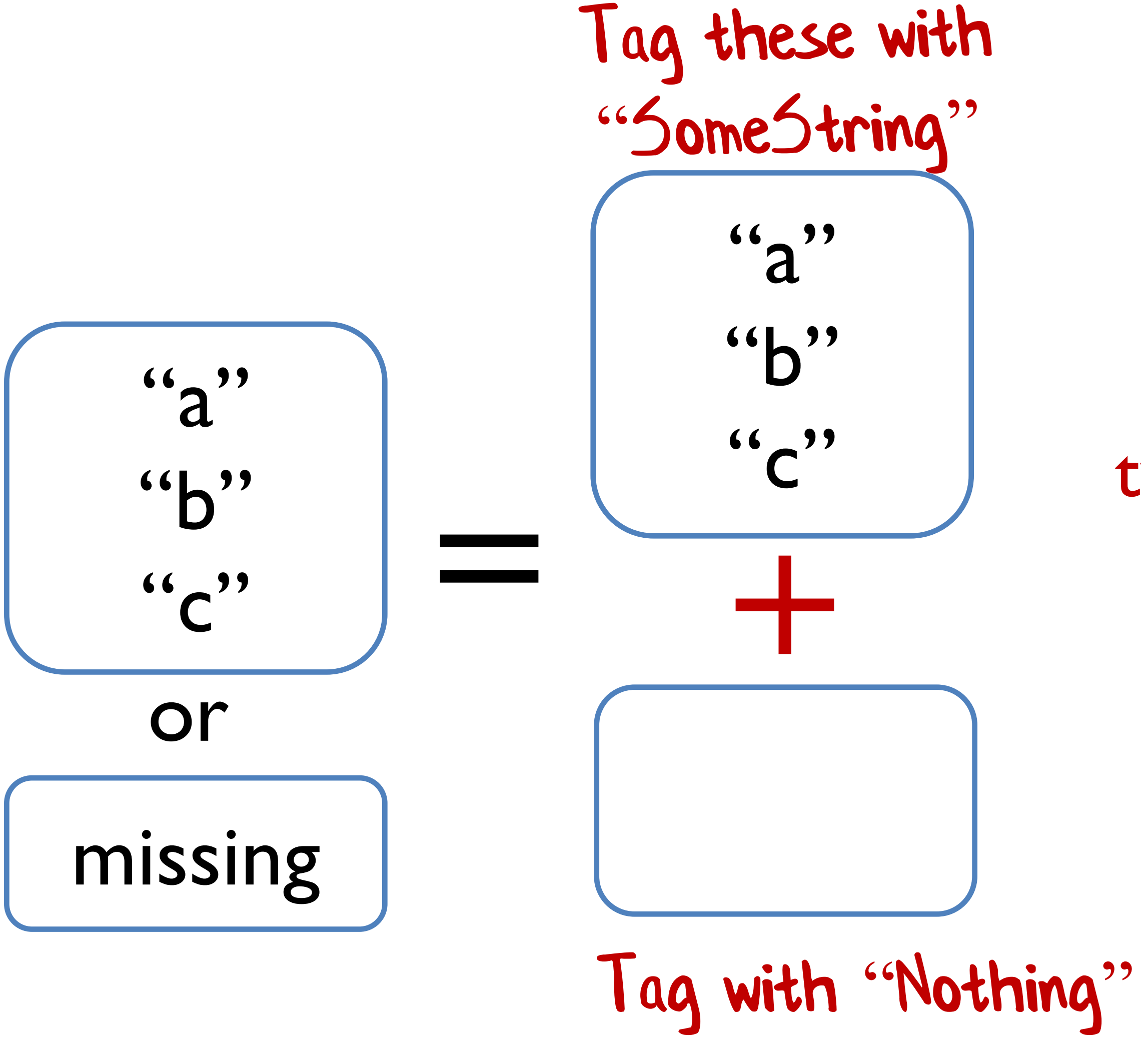
is null really a string?

“null is the Saruman of static typing”

Null is not allowed



A better way for optional values



Tag these with "SomeString"

"a"
"b"
"c"

or

missing

=

"a"
"b"
"c"

+

Tag with "Nothing"

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

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
    LastName: string
    }
```

The built-in “Option” type

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

generic type

```
type PersonalName =
   {
   FirstName: string
   MiddleInitial: Option<string>
   LastName: string
   }
```

Change to optional

The built-in “Option” type

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

generic type

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

nice and readable!

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
```

Distinct types

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

Also distinct types

Creating the EmailAddress type

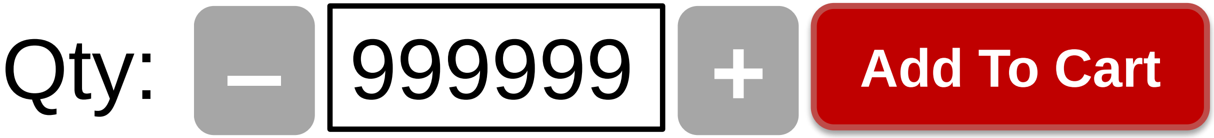
```
let createEmailAddress (s:string) =
    if Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then Some (EmailAddress s)
        else None

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

```
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
  }
```

I SEE WHAT YOU DID THERE
U.S. ARM

The challenge, revisited

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

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

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

Encoding domain logic

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

anyone can set this to true

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

```
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

```
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
```

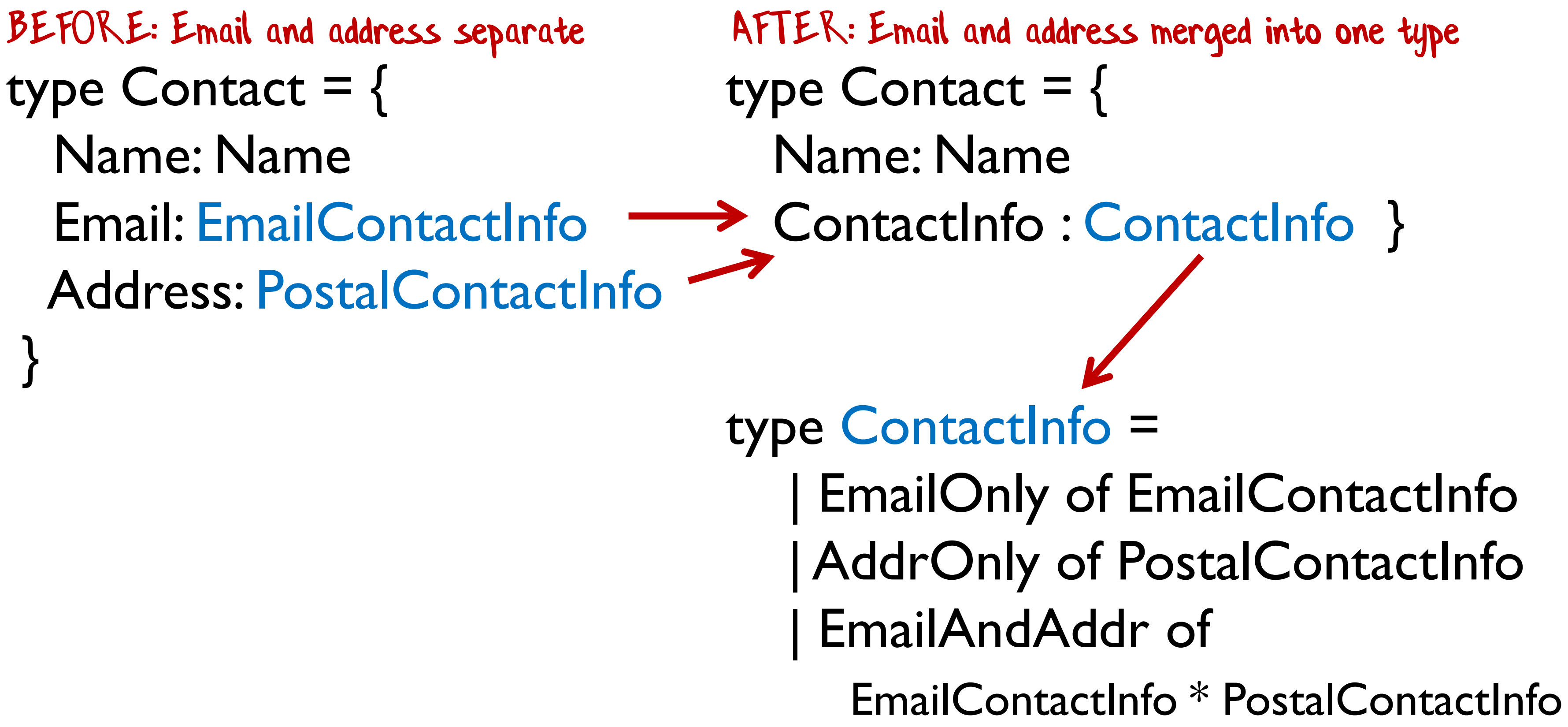
only three possibilities

requirements are now encoded in the type!

```
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

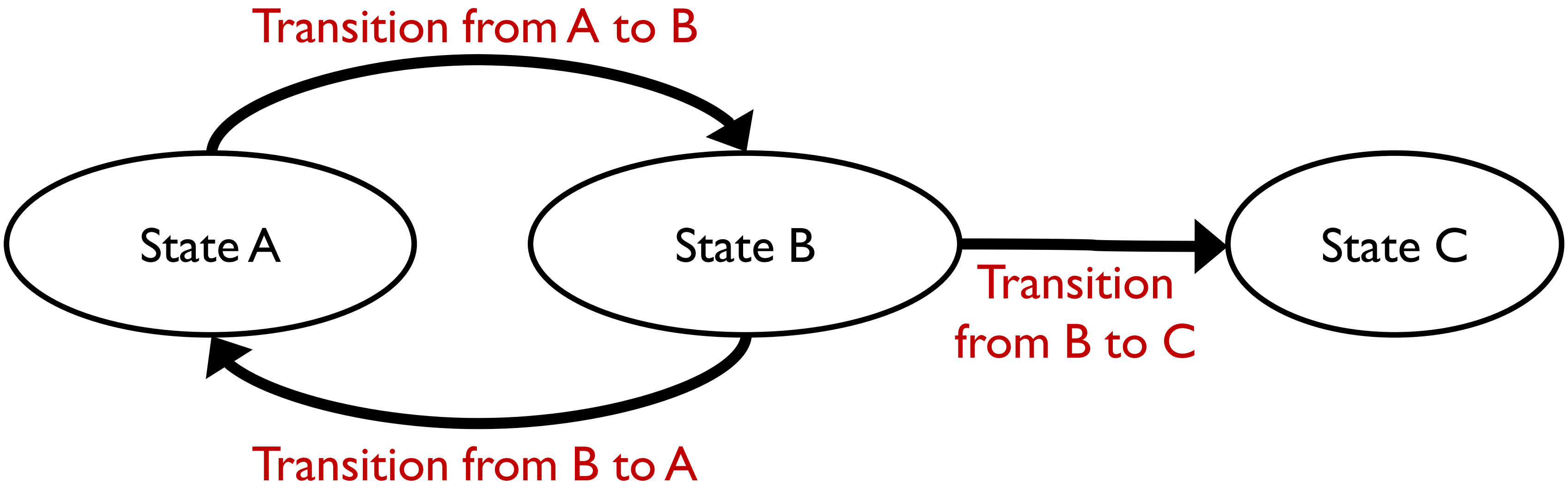
```
type Contact = {
   Name: Name
   PrimaryContactInfo: ContactInfo
   SecondaryContactInfo: ContactInfo option }
```

One way of being contacted is required

States and Transitions

Modelling a common scenario

States and transitions



Transition from A to B

State A

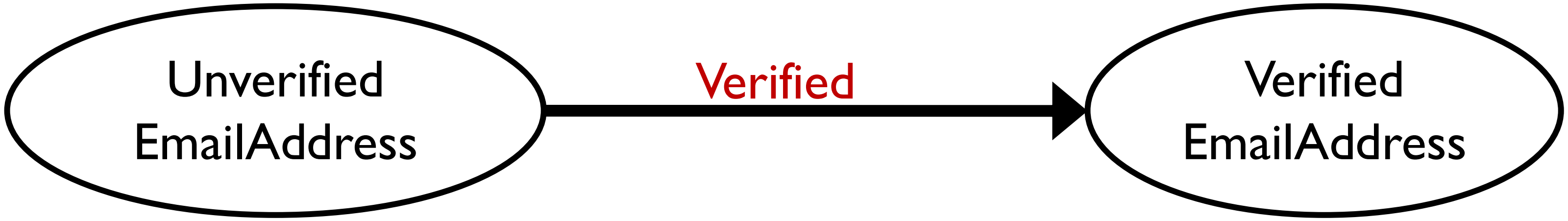
State B

State C

Transition from B to C

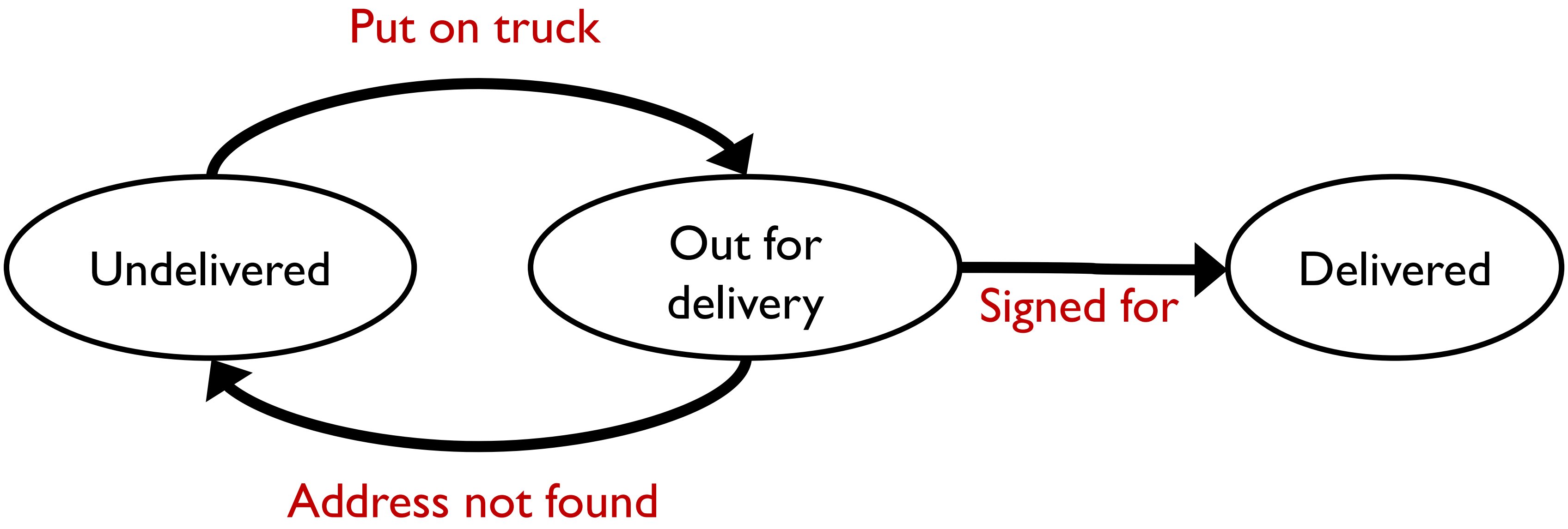
Transition from B to A

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 "

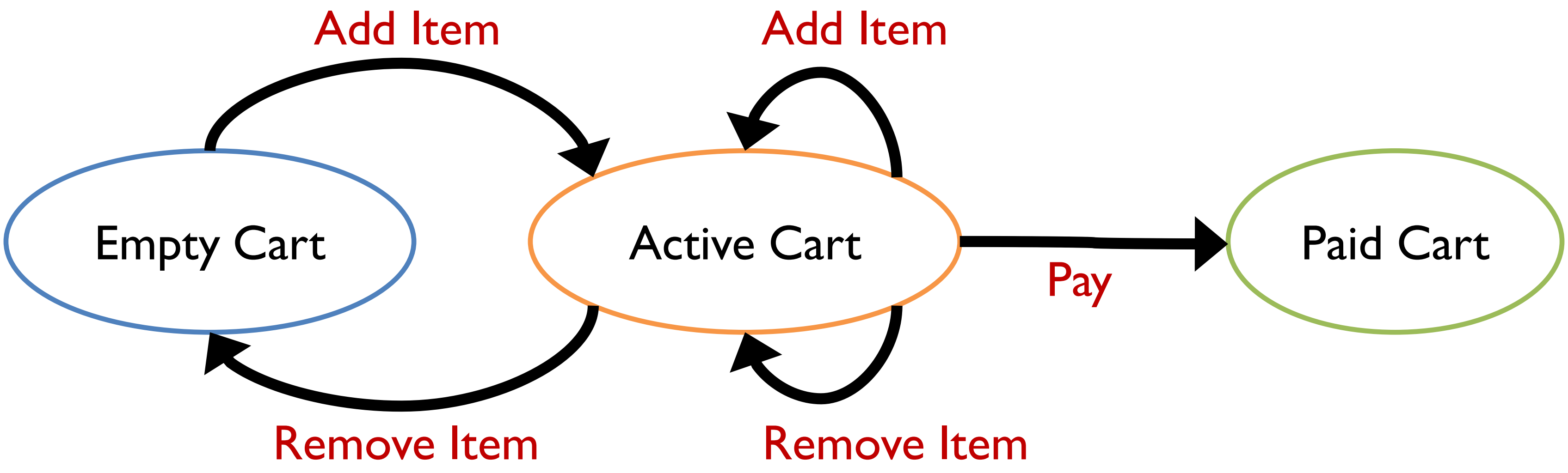
States and transitions for shipments



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"

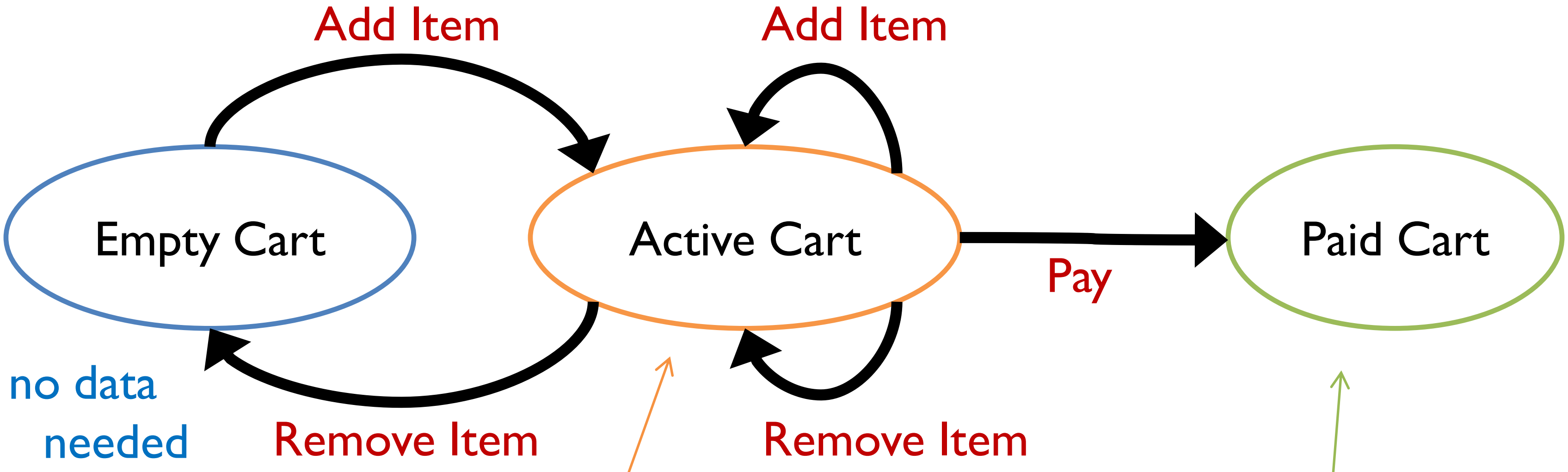
States and transitions

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



Add Item

Add Item

Empty Cart

Active Cart

Paid Cart

Pay

no data needed

Remove Item

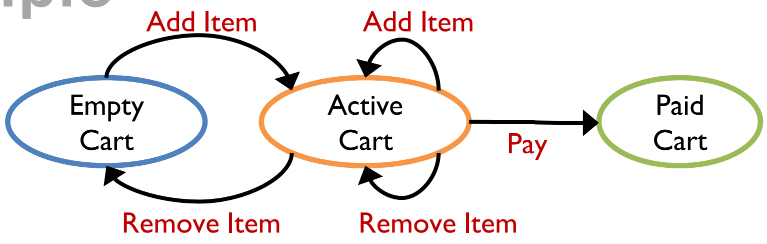
Remove Item

```
type ActiveCartData =
    { UnpaidItems: Item list }
```

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

What data do we need to store?

Modelling the shopping cart example



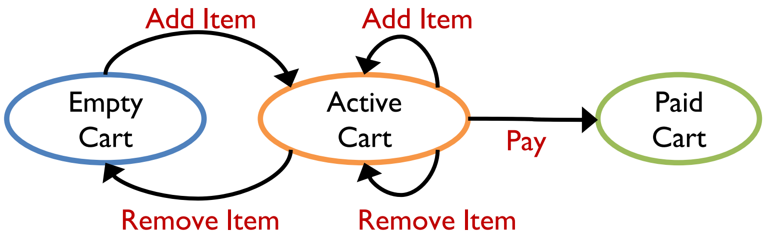
```
type ActiveCartData =
    { UnpaidItems: Item list }

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

One of three states

```
type ShoppingCart =
    | EmptyCart  // no data
    | ActiveCart of ActiveCartData
    | PaidCart of PaidCartData
```

No data needed for empty cart state

Shopping cart example



Shopping Cart API

initCart :
Item –› ShoppingCart

addToActive:
(ActiveCartData * Item) –› ShoppingCart

might be empty or active – can't tell

removeFromActive:
(ActiveCartData * Item) –› ShoppingCart

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

Server code to remove an item

```
let removeFromActive (cart:ActiveCart) item =
    let remainingItems =
        removeFromList cart.existingItems item
    match remainingItems with
    | [ ] ->
        EmptyCart
    | _ ->
        {cart with UnpaidItems = remainingItems}
```

create a new ActiveCart with the
item removed

Shopping cart example

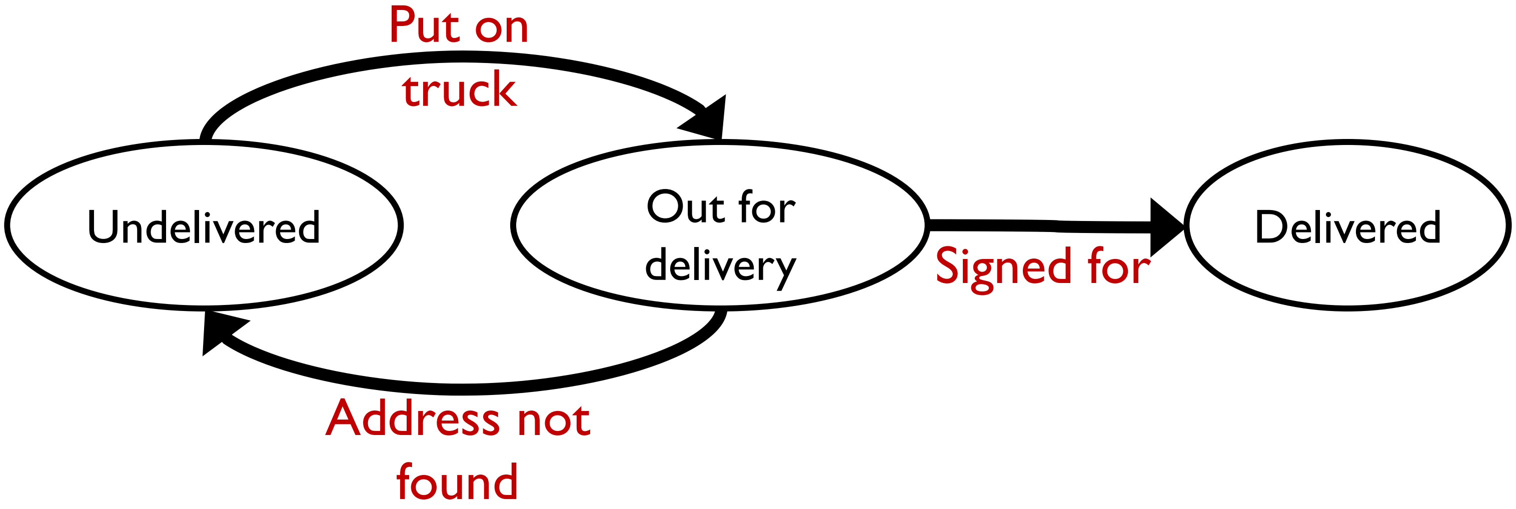
Client code to remove an item using the API

```
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.



Review

What I covered in this talk:

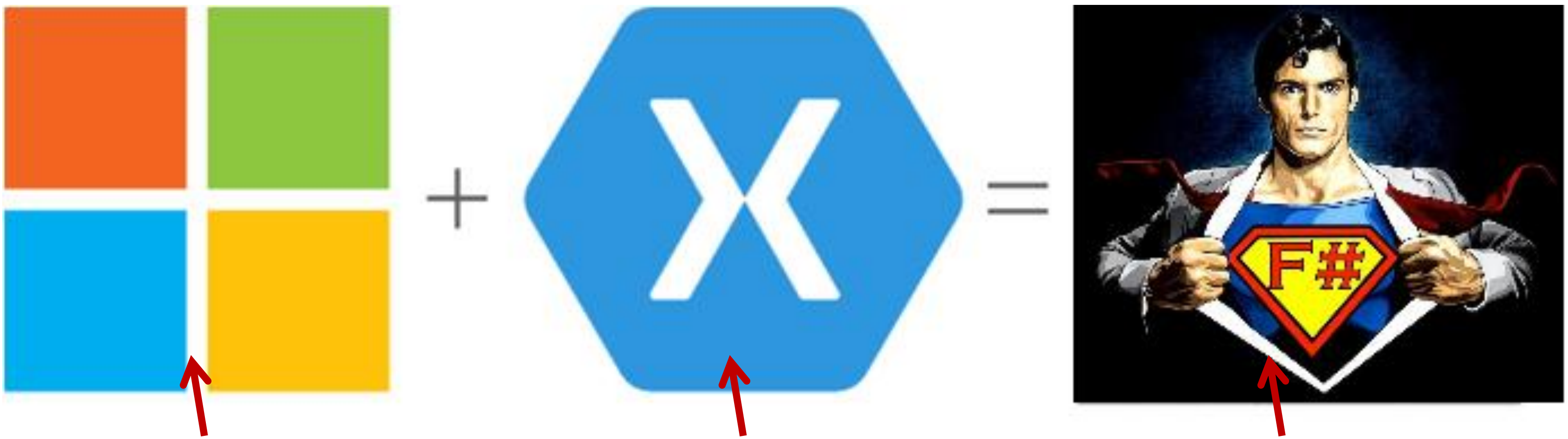
- Ubiquitous language
  - Self-documenting designs
- Algebraic types
  - products and sums
- Designing with types
  - Options instead of null
  - Single case unions
  - Choices rather than inheritance
  - Making illegal states unrepresentable
- States and transitions

Stuff I haven't had time to cover:

just scratching the surface today...

- Services
- CQRS
- The functional approach to use cases
- Domain events
- Error handling
- And much more...

F# is low risk



Enterprise development

Mobile development

F# on over 2 billion devices

F# is the safe choice for functional-first development

Need to persuade your manager? -> FPbridge.co.uk/why-fsharp.html

credit: @7sharp9 @MattDrivenDev

DDD in F# resources

fsharpforfunandprofit.com/ddd

gorodinski.com

tomasp.net/blog/type-first-development.aspx/

#fsharp on Twitter

F# |> I ♥

Contact me

@ScottWlaschin

FPbridge.co.uk

Let me know if you need help with F#