```
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!

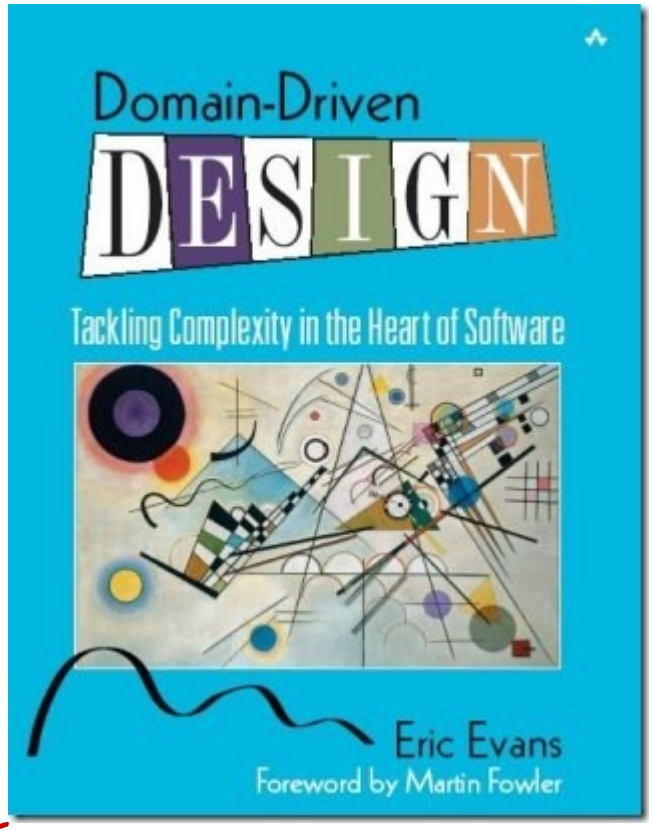
Domain Driven Design with the F# type system

Scott Wlaschin

@ScottWlaschin

fsharpforfunandprofit.com/ddd

What is DDD?

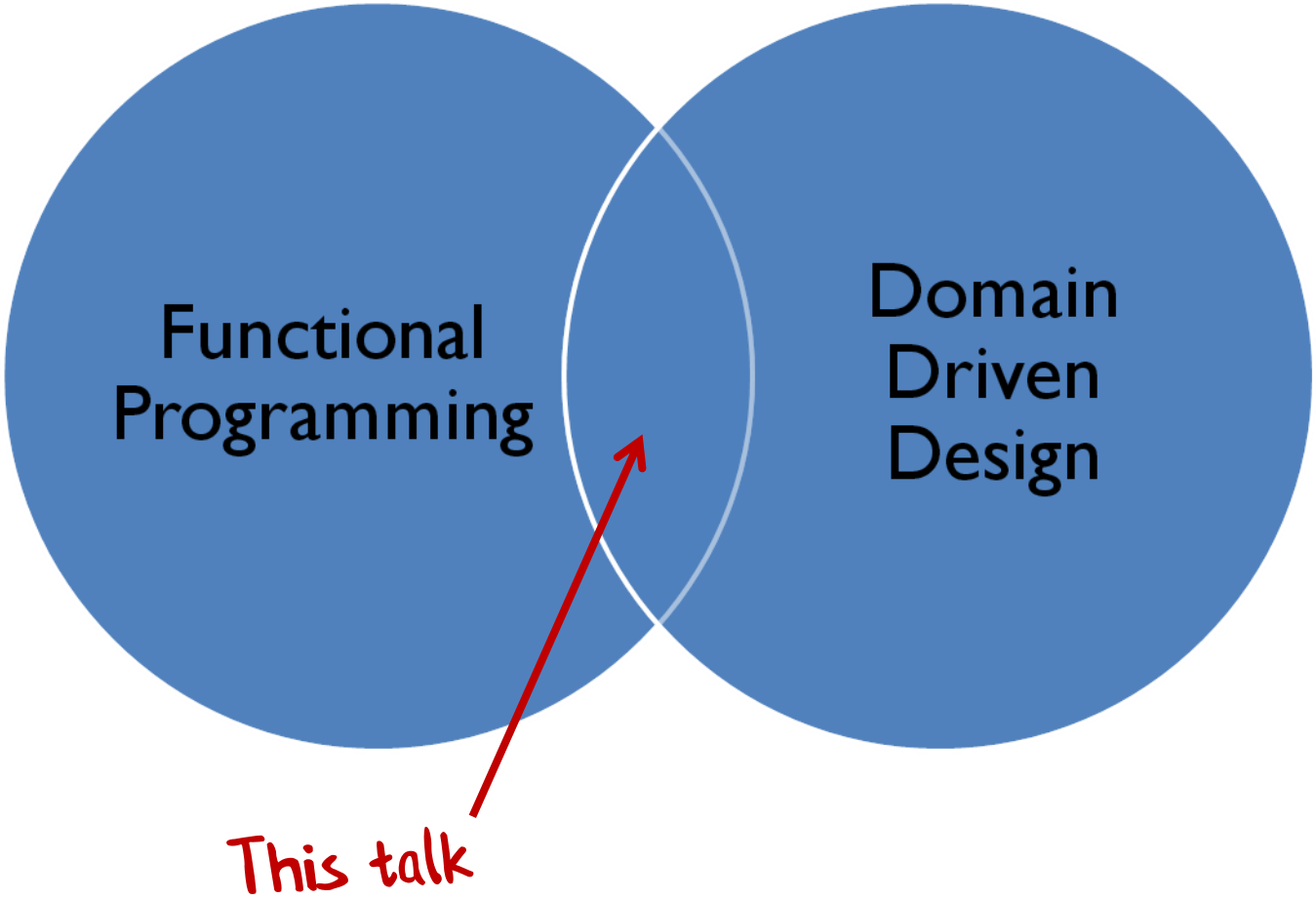


"Focus on the domain and domain logic rather than technology"
-- Eric Evans

What is F#?

"F# is a mature, open source, cross-platform, functional-first programming language which empowers users and organizations to tackle complex computing problems with simple, maintainable and robust code." — *fsharp.org*

"#fsharp is C# and python with extra awesome" --me


Functional Programming
Domain Driven Design
This talk

What I'm going talk about:

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

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

... 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

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

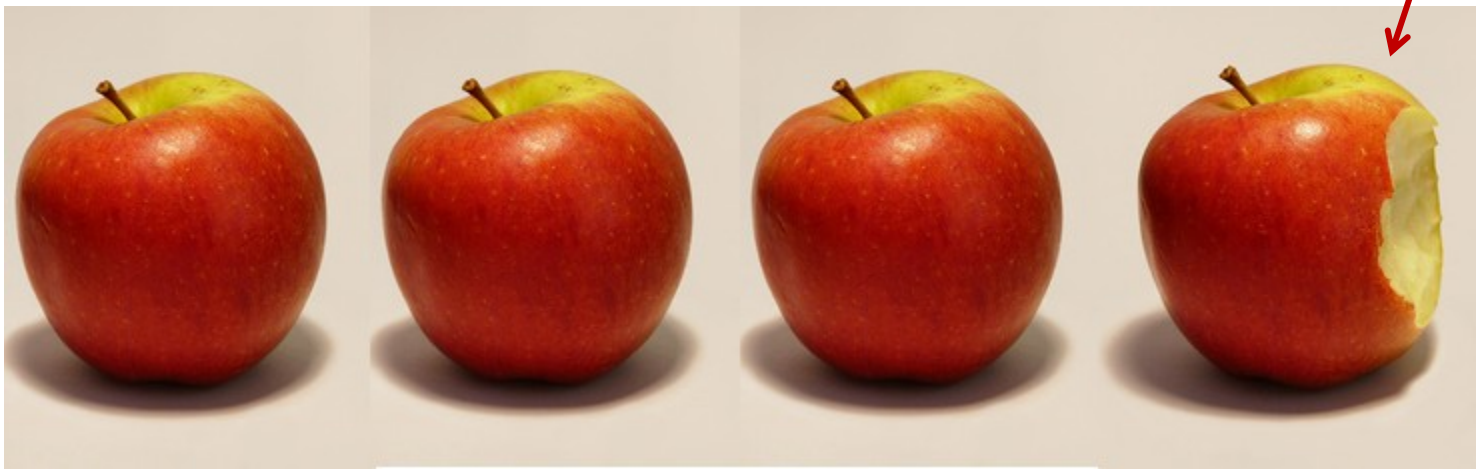
#fsharp <3 #bloba development!

F# vs. C#
for Domain Driven Design

Values vs. Entities

Values vs. Entities

My apple!



Value objects
– can't tell them apart

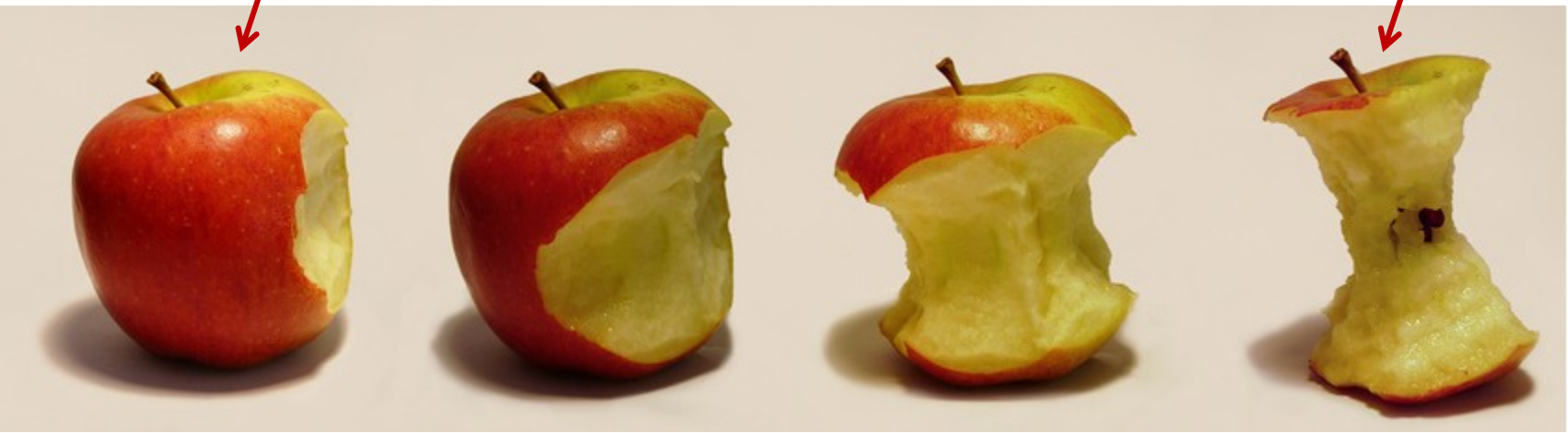
Entity object
– has identity

Photo credit: http://www.flickr.com/photos/anacooke/

Values vs. Entities

My apple!

Still my apple!



An Entity's properties may change over its lifetime
but it preserves its identity

Photo credit: http://www.flickr.com/photos/anacooke/

Values vs. Entities

Examples of Values:

- Personal name
- Email address
- Postal address
- Product code

uneaten apple

Examples of Entities:

- Customer
- Order
- Product

half eaten apple

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

stop talking and show me some code!

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# (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

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

Entity immutability

```
[<NoEquality; NoComparison>]
type Person = { ... ... ... }
```

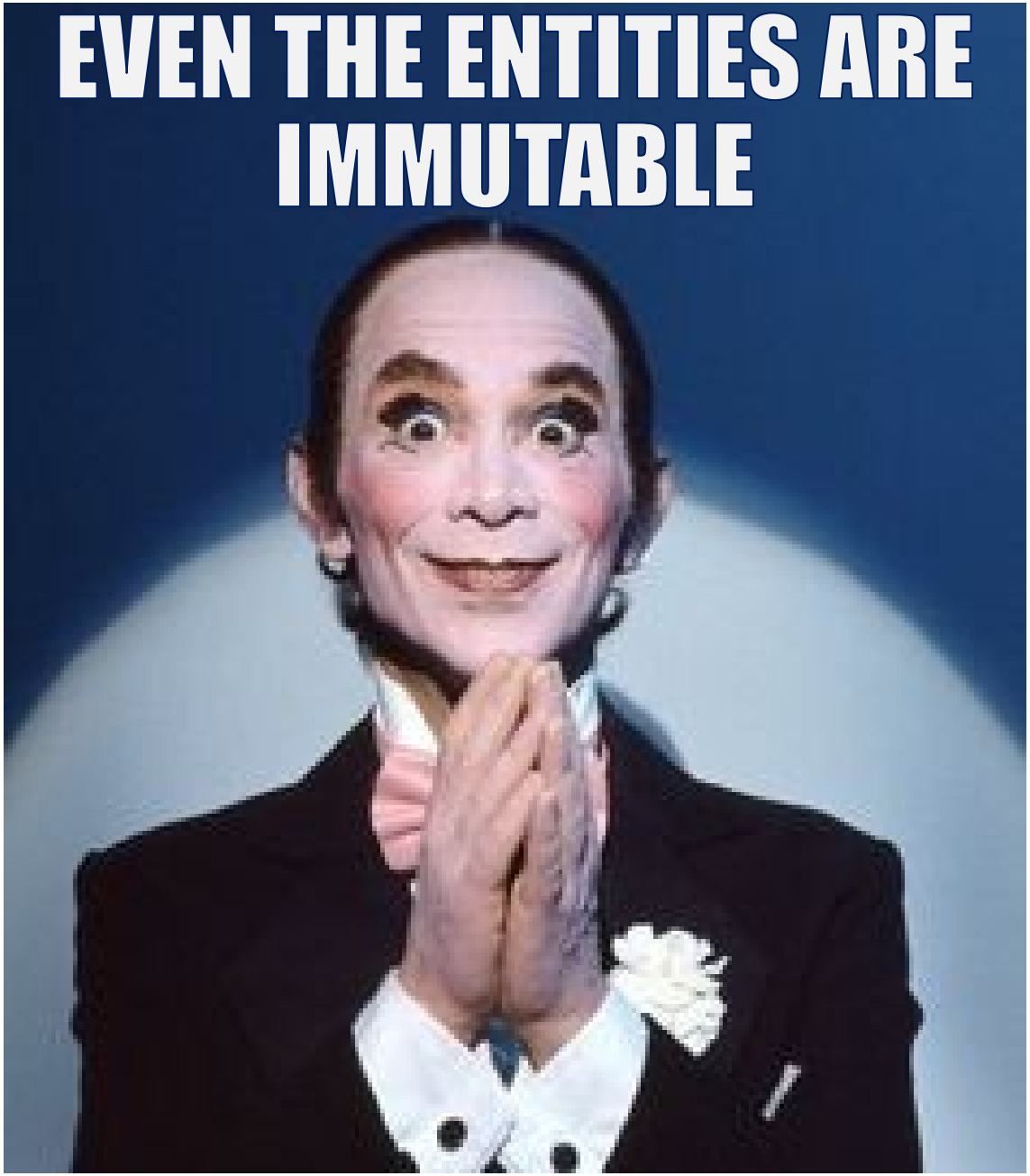
immutable

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


EVEN THE ENTITIES ARE
IMMUTABLE

Entity object definition in F# with mutability

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

In #fsharp, "mutable" is a code smell

Reviewing the C# code so far...

Can you tell values from entities at a glance?

```
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;
    }
}
```

Adding a marker interface doesn't stop you from getting the logic wrong!

```
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;
    }
}
```

Reviewing the C# code so far...

1) Is this a reasonable amount of code to write?
2) Could a non-technical person understand it?

```
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;
    }
}
```

Reviewing the F# code so far...

1) Is this a reasonable amount of code to write?
2) Could a non-technical person understand it?

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

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

Comparing C# vs. F#

	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

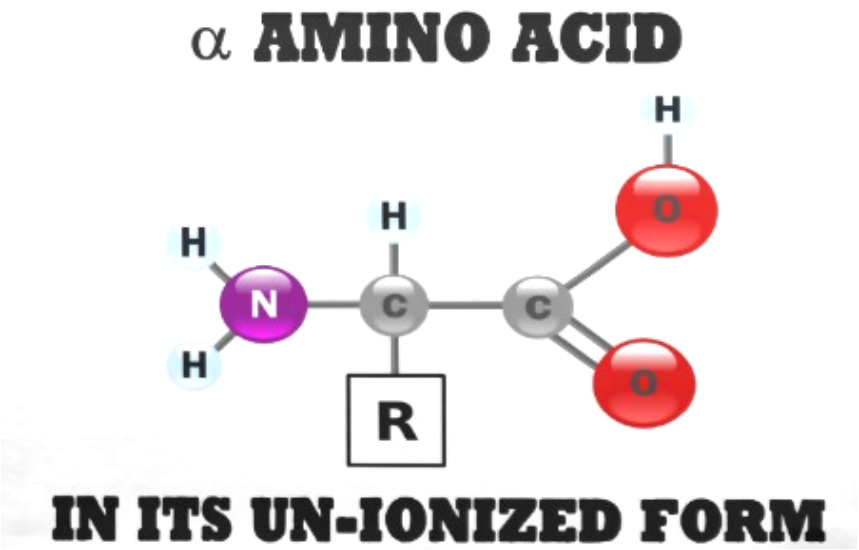
Very important thing for DDD!

F# for Domain Driven Design

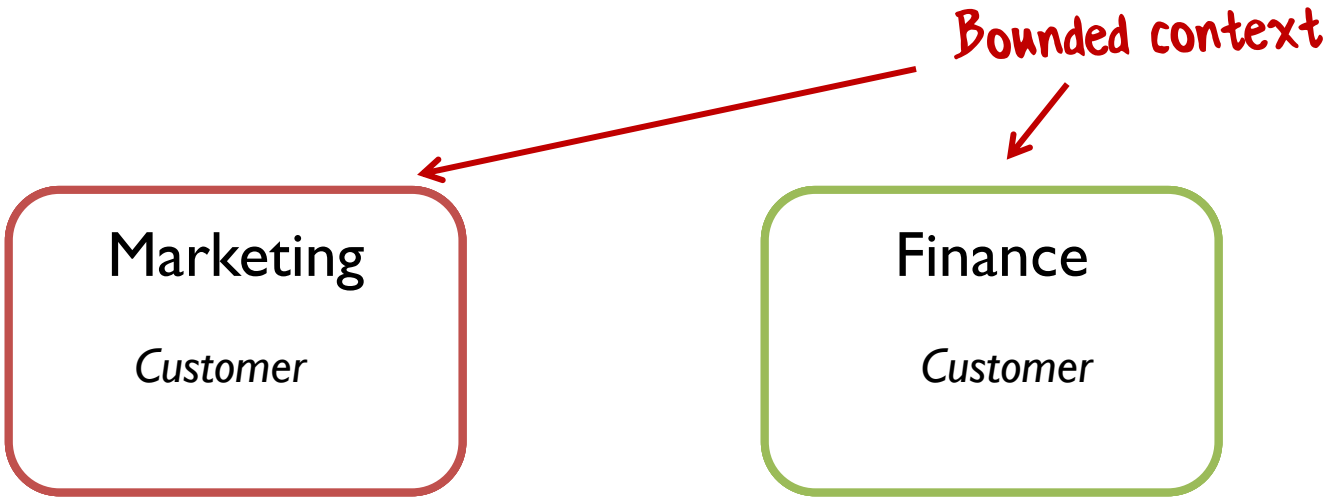
Communicating a domain model

Communication is hard...

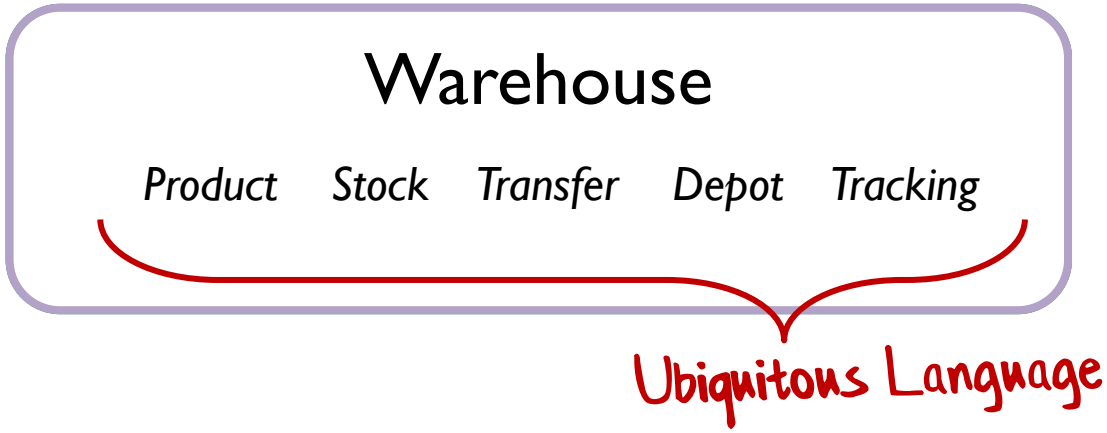
U-N-I-O-N-I-Z-E



Communication in DDD: “Bounded Context”



Communication in DDD: “Ubiquitous Language”



module **CardGame** =

← Bounded context

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** =

← Bounded context

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

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

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

type **Hand** = Card list

list type is built in

type **Deck** = Card list

type **Player** = {Name:string; Hand:Hand}

type **Game** = {Deck:Deck; Players: Player list}

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

type **Deal** = Deck –› (Deck * Card)

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

1) Is this a reasonable amount of code to write?
2) Could a non-technical person understand it?

```
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
```

```
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"

Domain driven, not database driven!

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

This page looks like pseudo-code,
but is actually real compilable code.


WE DON’T NEED NO STINKING
UML DIAGRAMS

Understanding the F# type system

An introduction to “algebraic” types

Understanding the F# type system

An introduction to “composable” types

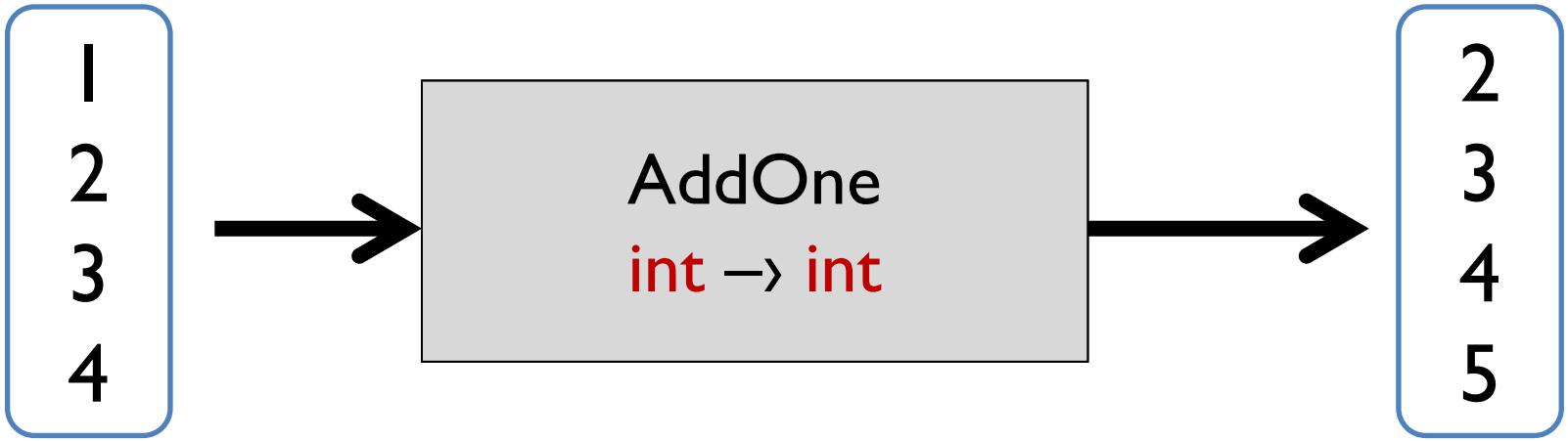
Composable types

Creating new types in F#

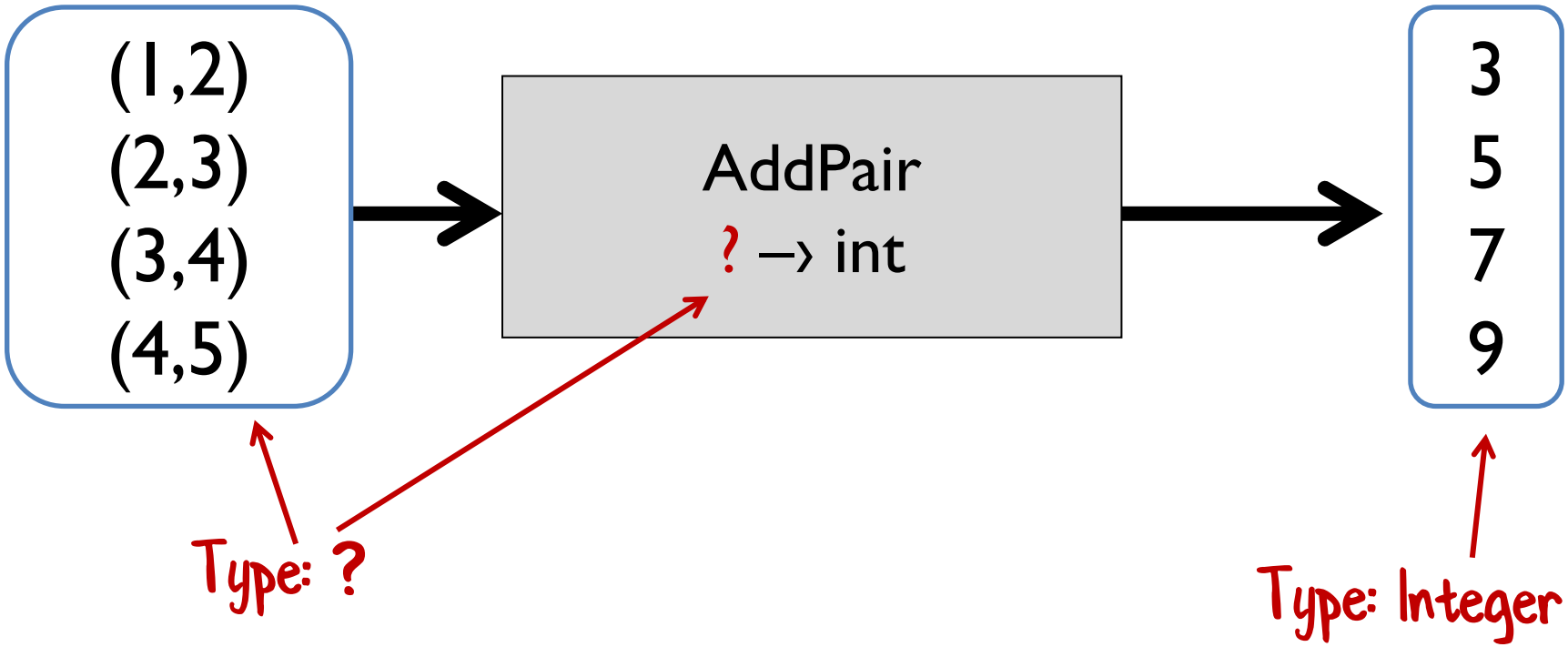
New types in F# are constructed by combining other types using two basic operations:

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

Creating new types in F#



Representing pairs



How can we
represent this type?

Representing pairs

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

=

1
2
3
4

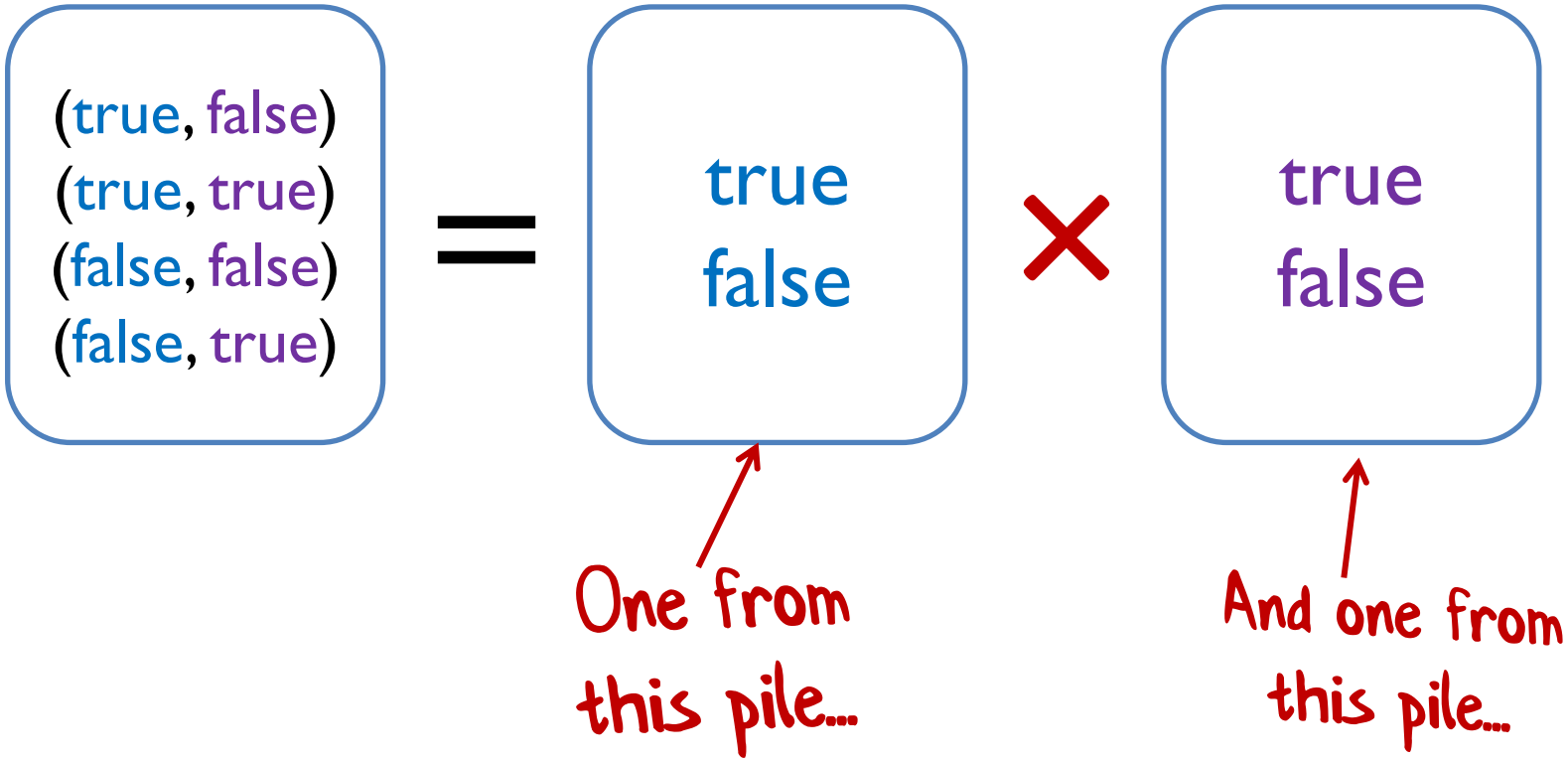
×

2
3
4
5

One from this pile...

And one from this pile...

Representing pairs



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

=

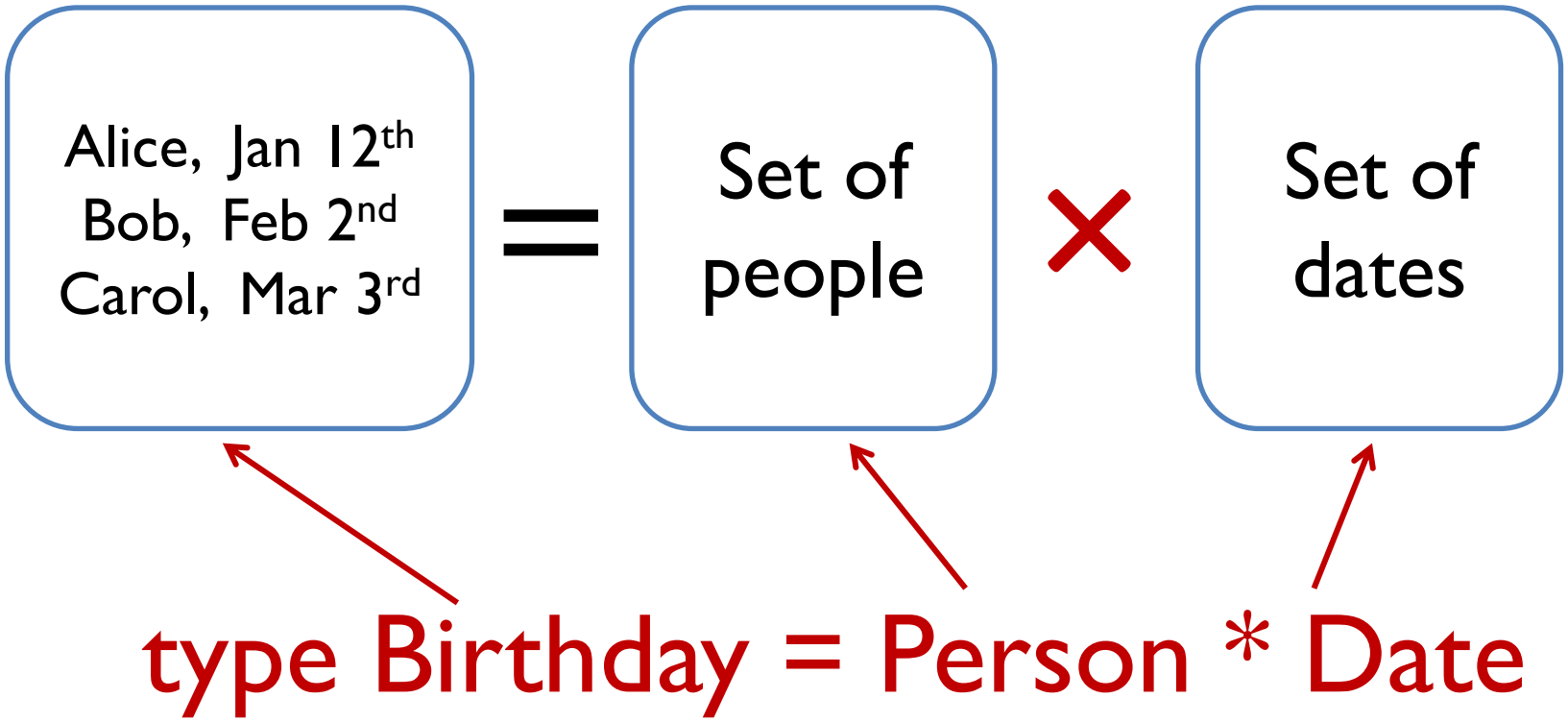
Set of people

×

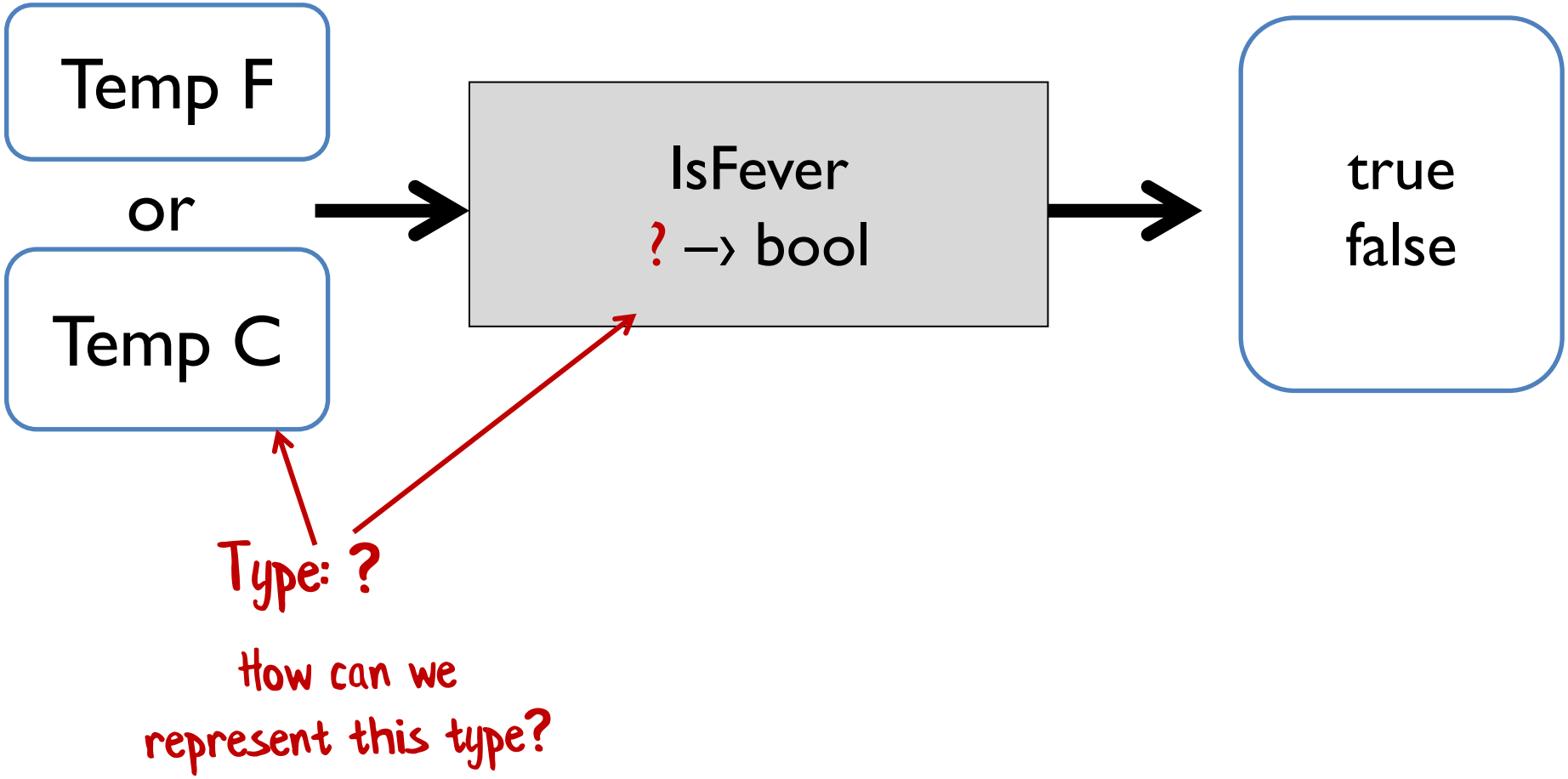
Set of dates

How to represent this?

Using tuples for data



Representing a choice



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

Check 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
```

What are types for in F#?

An annotation to a value for type checking

type AddOne: int –› int

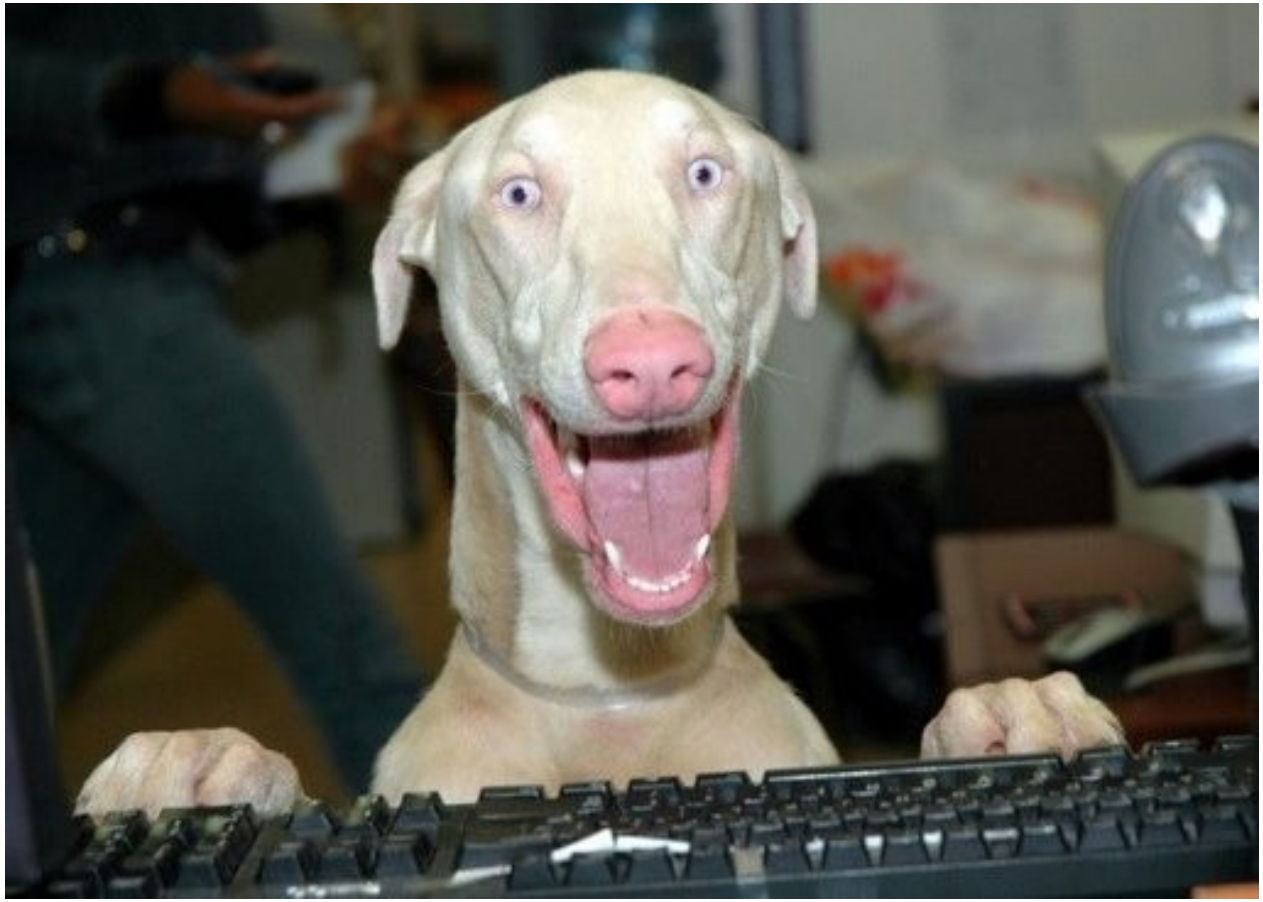
Domain modelling tool

type Deal = Deck –› (Deck * Card)

both at once!

#fsharp has "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;
    }
```

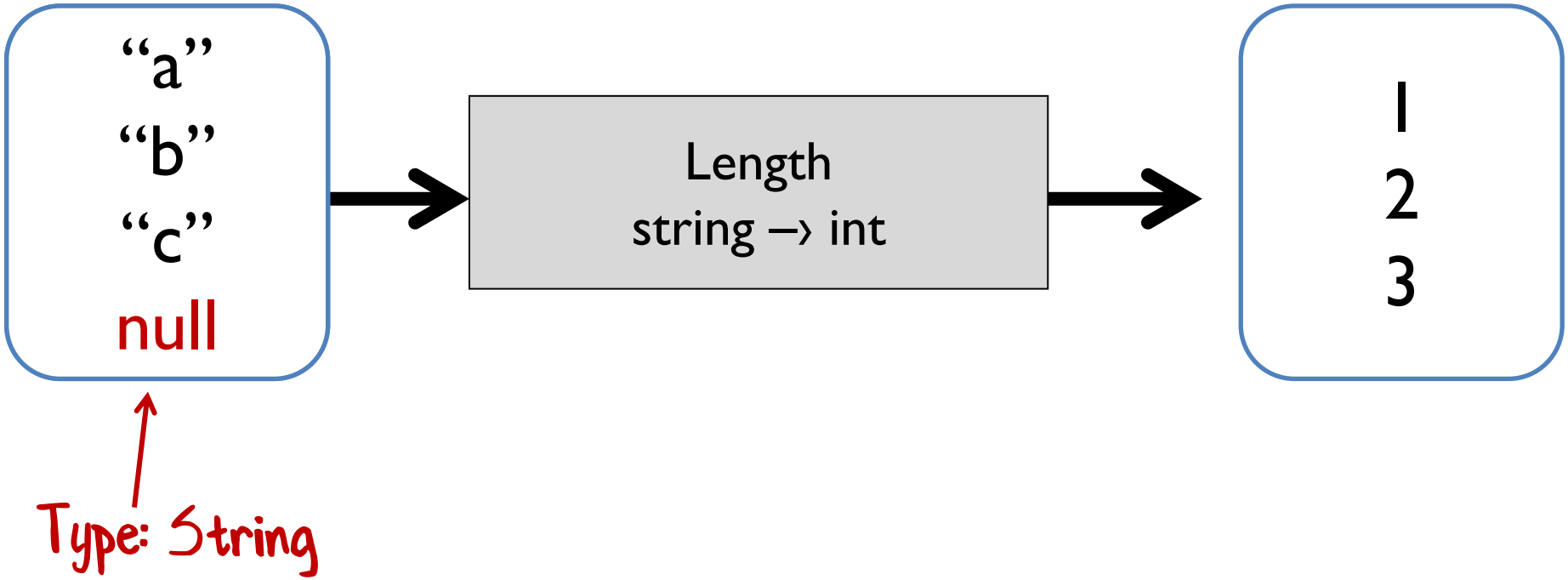
required

optional

required

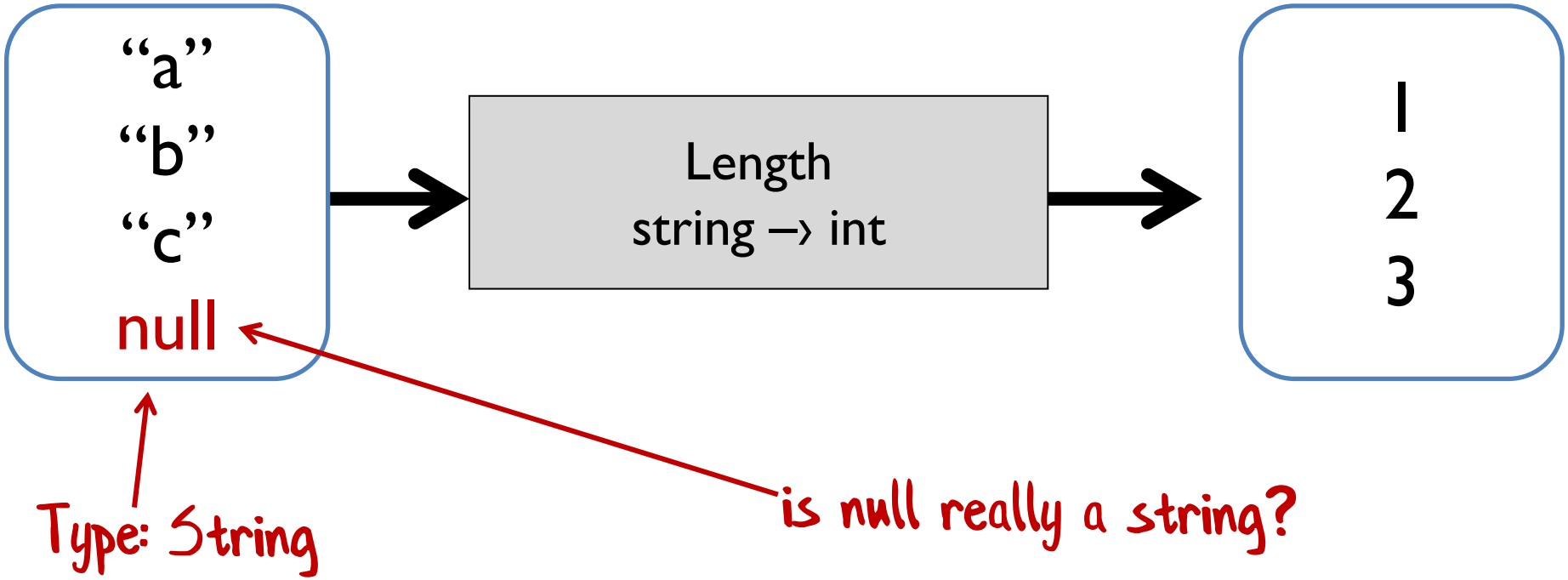
How can we represent optional values?

Null is not the same as “optional”



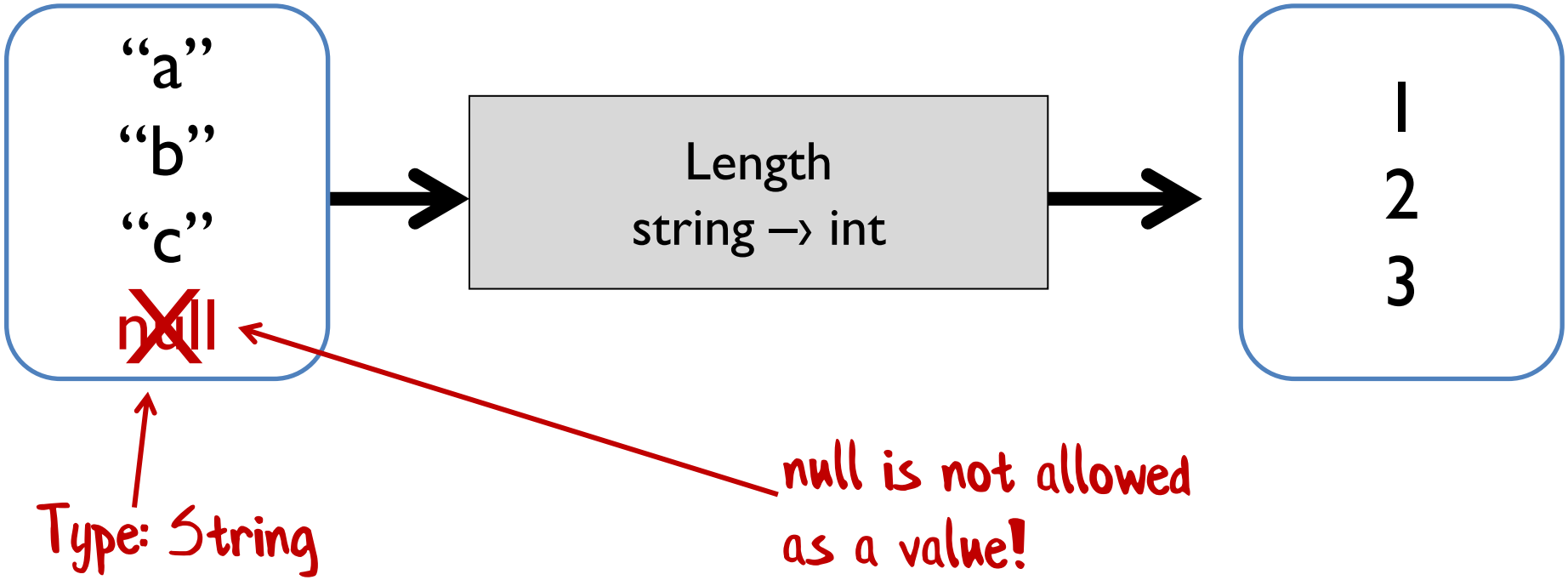

Spock, set phasers to null!
That is illogical, Captain

Null is not the same as “optional”

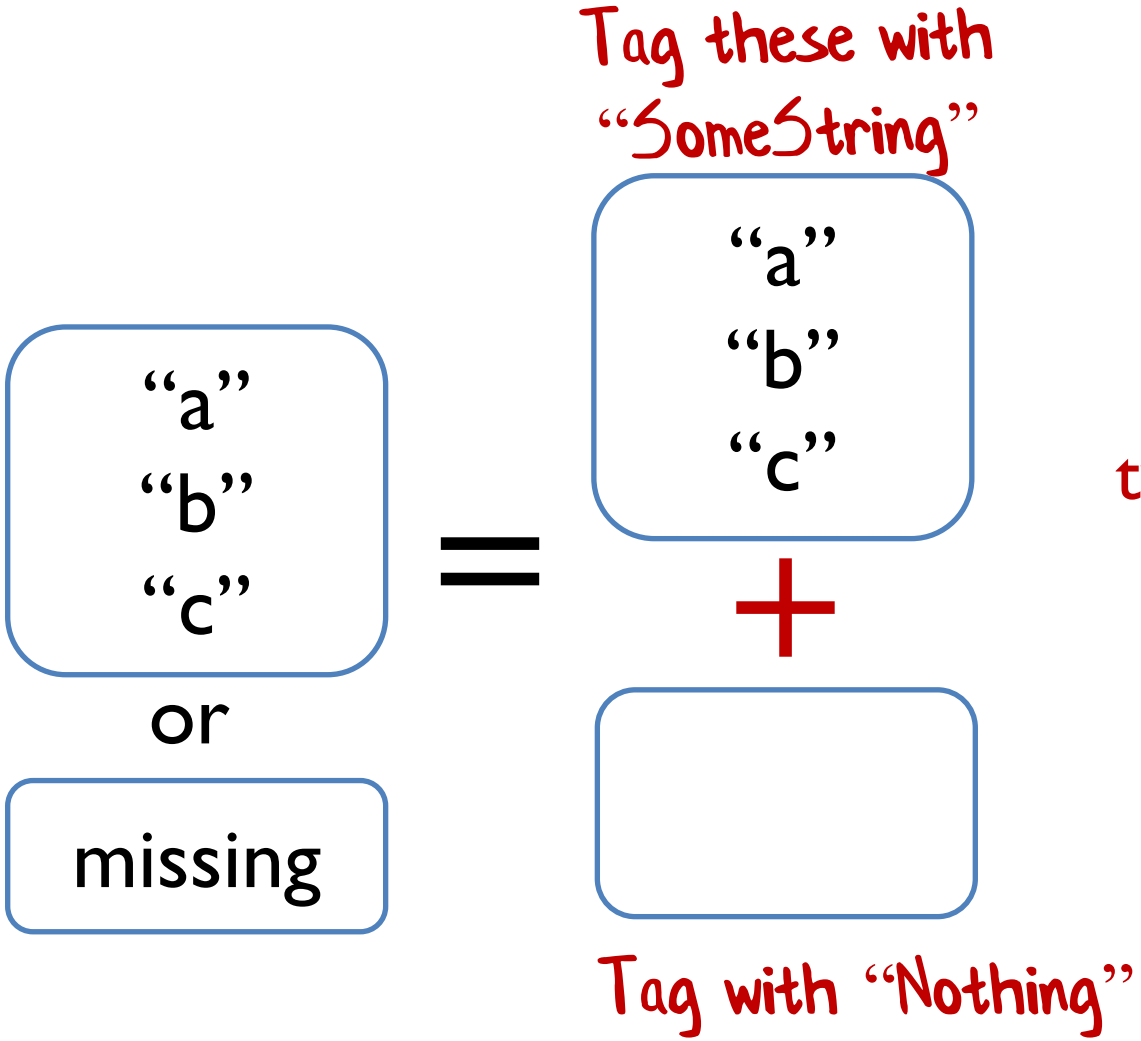


"null is the Saruman of static typing"

Null is not allowed in F#



A better way for optional values




```
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
    }
```

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

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

Distinct types

Also distinct types

Creating the EmailAddress type

```
let createEmailAddress (s:string) =
    if Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then (EmailAddress s)
        else ?
```

What to return if not valid?

```
createEmailAddress:
        string –> EmailAddress
```

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?

Qty: – 999999 + Add To Cart

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

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 a 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

New rule:

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

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

New!

This design does not meet the 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
}
```

Could both be missing?

This design does not meet the requirements either!

"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
```

F# is almost as awesome as this

Making illegal states unrepresentable

Is this really what the business wants?

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

Making illegal states unrepresentable

Better rule perhaps?

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

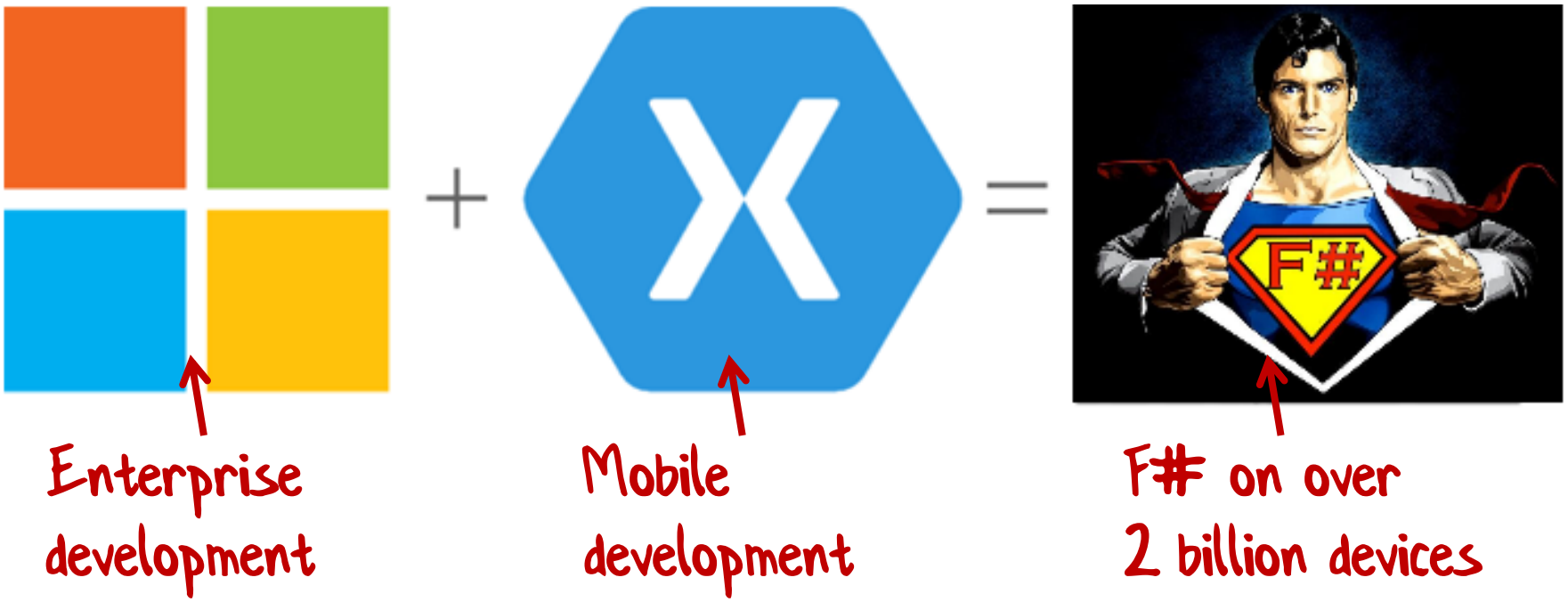
```
type ContactInfo =
    | Email of EmailContactInfo
    | Addr of PostalContactInfo

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

Stuff I haven't had time to cover:

just scratching the surface today...

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

F# is low risk



F# is the <u>safe</u> choice for functional-first development

credit: @7sharp9 @MattDrivenDev

Thank you!

fsharpforfunandprofit.com/ddd

fsharp.org/testimonials

tryfsharp.org

try F# in your web browser!

@ScottWlaschin

#fsharp on Twitter