Functional Programming in C#

Part 1

Danial Khosravi

Talk Objectives

Using a progressive example to learn more about:

- Immutability
- Purity
- Introduction to Functional Effects
- FP things to avoid in C#

Disclaimer

- I'm not a C# expert
- I'm not a functional programming expert
- FP is full of great techniques that are language agnostic
- These technique have helped me to be a better OOP programmer
- C# and OOP are great and are here to stay
- This talk invites you give functional approach to programming a try

Example 1

Immutability

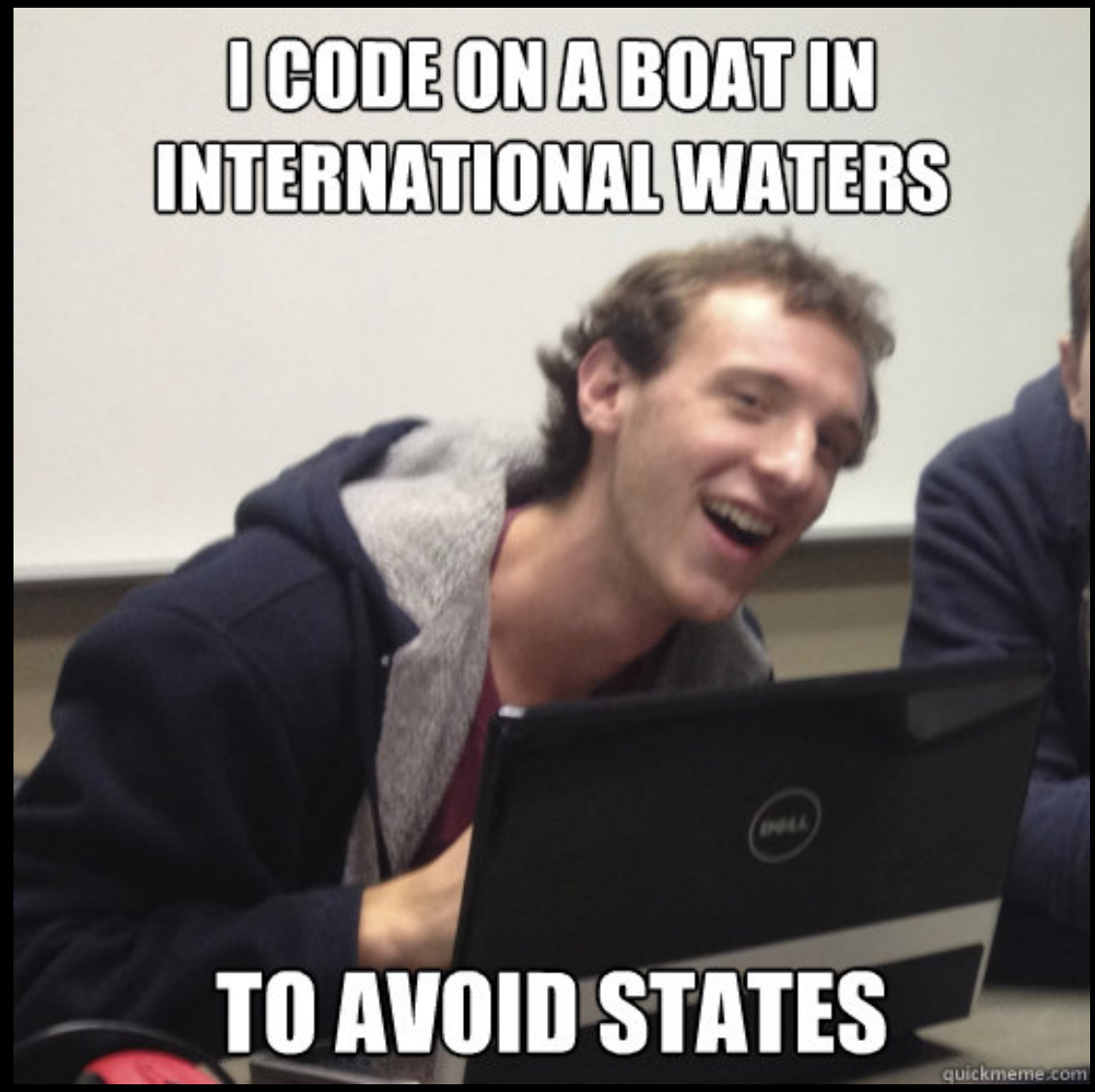
Change vs Mutation

- Values **Change** over time, such as an inventory going up or down
- **Mutation** means data is update in place, the previous value is lost
- In FP we represent change without mutation
- Create new instances that represent the data with the desired changes

  How? wait for example 2

Encapsulate Mutation

- Local mutation is OK
- As long as it modifies local state that's only visible within the scope of a function
- Be practical and use these principles to your benefit



Enforcing immutability in C#

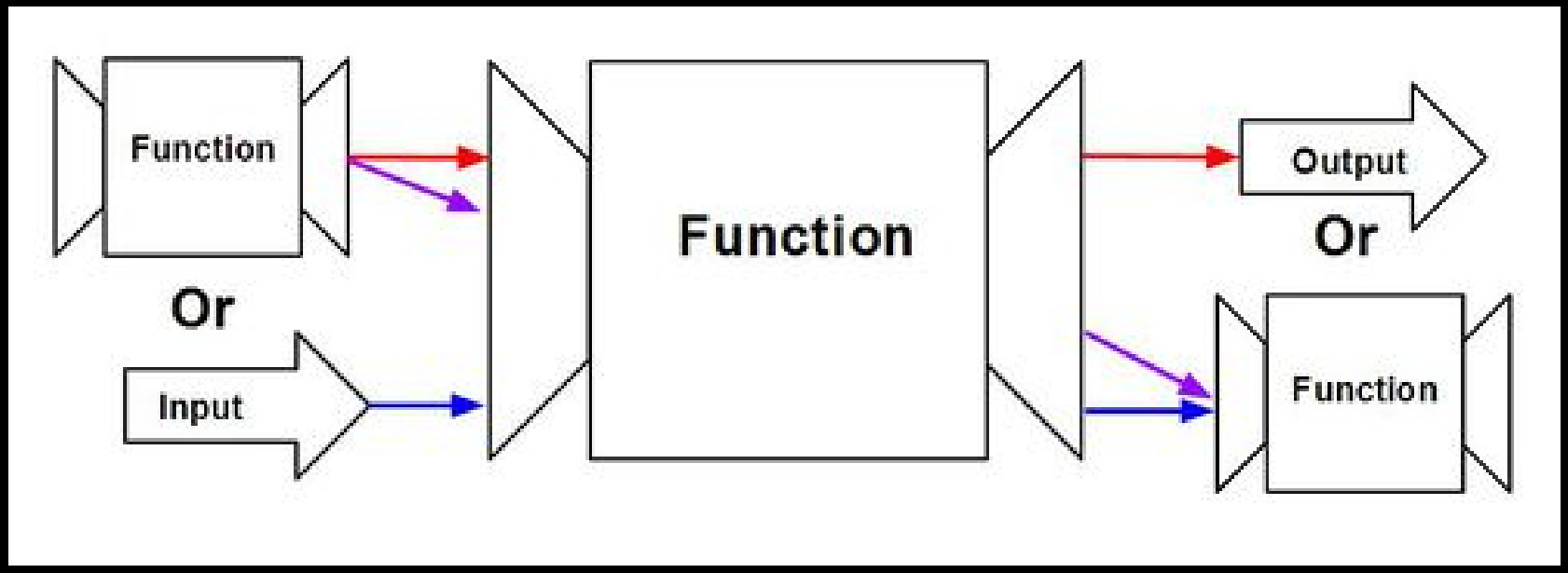
See what works the best for your team
You can have immutability by convention by just *avoiding* it
Or you can use some of the following techniques to enforce it:

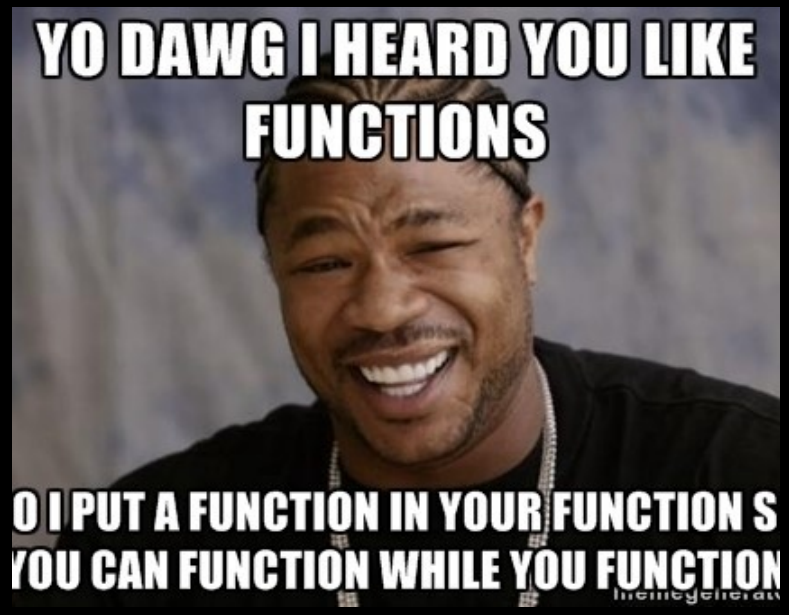
- Remove property setters
- Avoid object initializer syntax
- Pass values through the constructor (more boilerplate)
- Create **Copy**/**With** functions to create an updated instances
- Mark your classes as sealed to prevent mutable subclasses
- Use ImmutableLists from **System.Collections.Immutable**

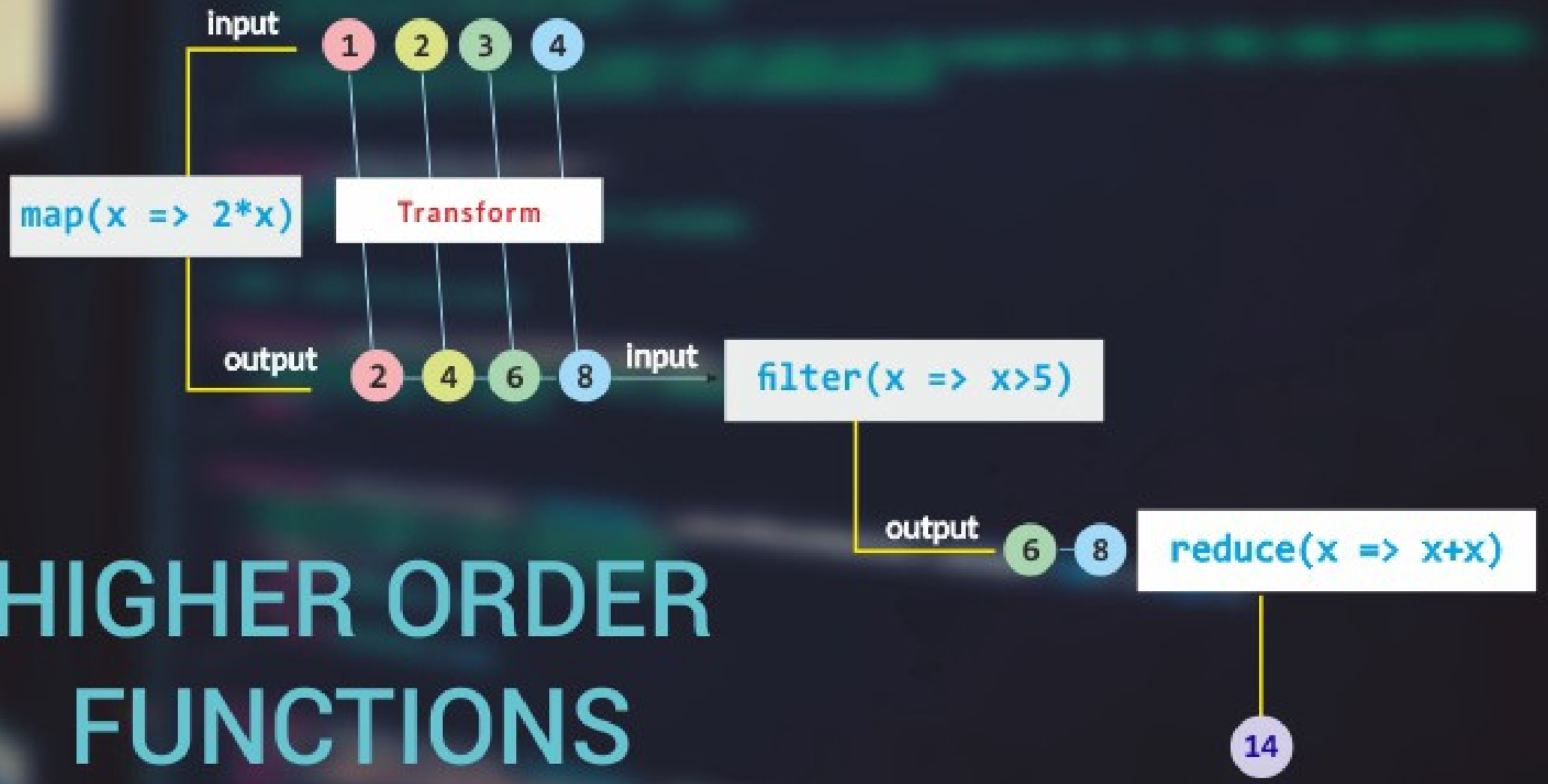
Higher Order Functions

Higher Order Functions

Functions that take other functions as inputs or return functions as output or both.







HIGHER ORDER FUNCTIONS

Higher Order Functions in C# using LINQ

- Select (== map)
- Where (== filter)
- Aggregate (== reduce)
- Zip (== zip)
- SelectMany (== flatMap or bind)
  * bind is the most powerful HOF

Benefits of Higher Order Functions

- Readability
- Avoid code duplication and better testability
- Enables **Imperative** vs **Declarative** programming approach
- Allow programming with **Expressions** as opposed to **Statements**
- Makes your code more composable
- Increase code reusability
- Specially useful when introducing FP in a non functional c
  *more on this when we cover Functional Effects

Drawbacks of Higher Order Functions

- Increased stack use, the performance impact is often negligible
- Debugging will be more complex because of the callbacks

Example 2

- Using HOF and Immutability principles to improve our code

Purity

Pure functions

- Output depends entirely on the input arguments
- Cause no side effects
- Hold no state and do not mutate global state directly
- Avoid mutating arguments

Side Effects

- Mutates global state (state visible outside of the function's scope)
- Mutates its input arguments
- Throws exceptions (there are arguments for and against this)
- Performs any I/O operation

Benefits of Pure Functions

- Easy to test and reason about
- Order of evaluation isn't important
- Can be Parallelized, Lazily evaluated and Cached/Memoized
- Easier to refactor and maintain

Purity

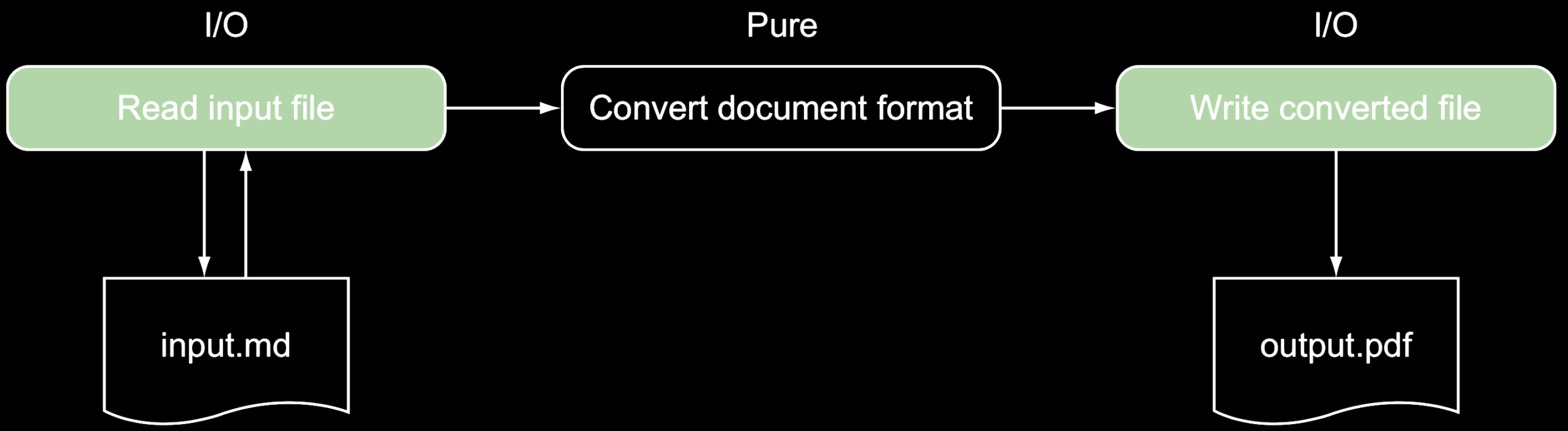
There are many facets to the concept of purity
It's a powerful principle that can be deemed impractical

Most practical benefits of pure code are often achieved by:

- Encapsulating I/O operations away from business logic
- Avoid mutating the arguments
- Avoiding partial functions

Strategies for Managing Side Effects

- Isolate I/O effects by keeping them at the edges of your code
- Write business logic by composing pure code



- Hard to achieve this all the time, so be practical about it

Avoid mutating the arguments

```
public IActionResult GetThings(ThingDto thingDto)
{
  thingDto.UserId = DecodeJwt().UserId;
  thingDto.ReceivedAt = DateTime.Now();
  var result = DoWord(thingDto);
  return Ok(result);
}
```

Alternative solution without mutation

```
public IActionResult GetThings(ThingDto thingDto)
{
  var userId = DecodeJwt().UserId;
  var receivedAt = DateTime.Now();
  var result = DoWord(thingDto, userId, receivedAt);
  return Ok(result);
}
```

Avoid partial functions

Partial Functions

- Mappings defined for **some** of elements of the domain
- Not clear what the function should do, given an input for which it can’t compute a result
- Type signature does not tell you about this shortcoming

Partial Functions

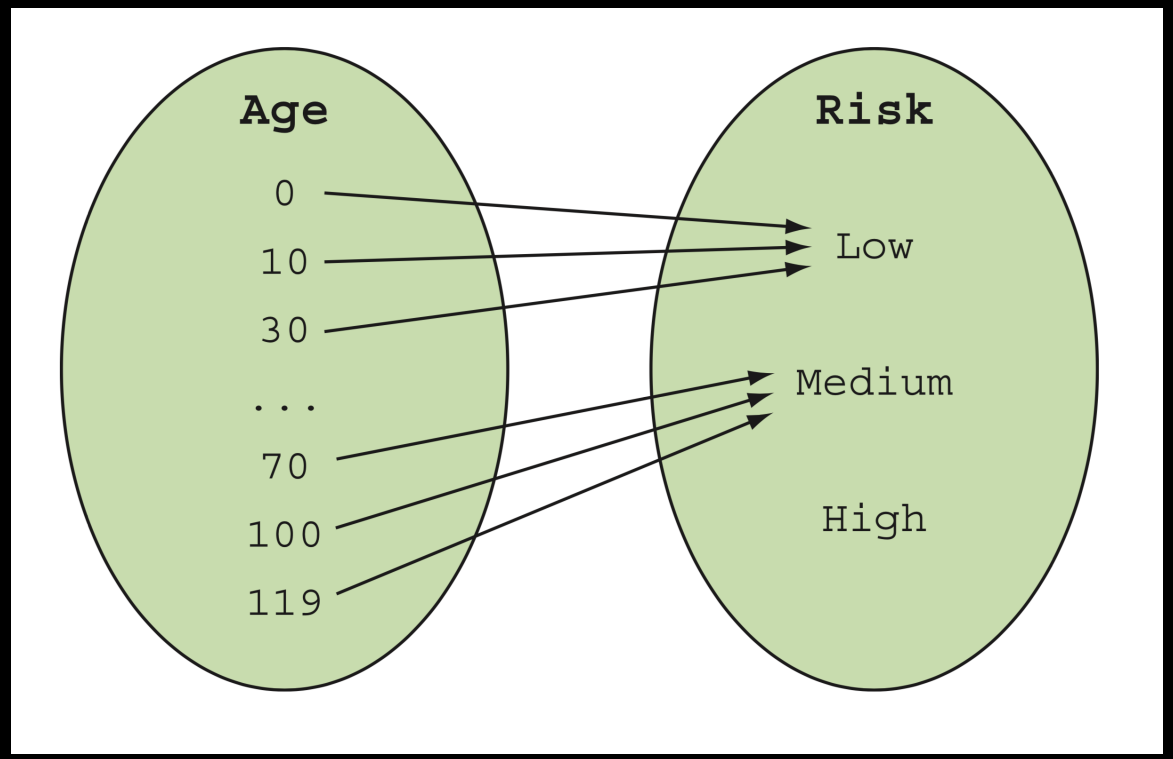
```
public static Risk ClassifyRisk(int age)
{
  if (age < 0 || age > 119)
  {
    throw new ArgumentException();
  }
  ...
}
```

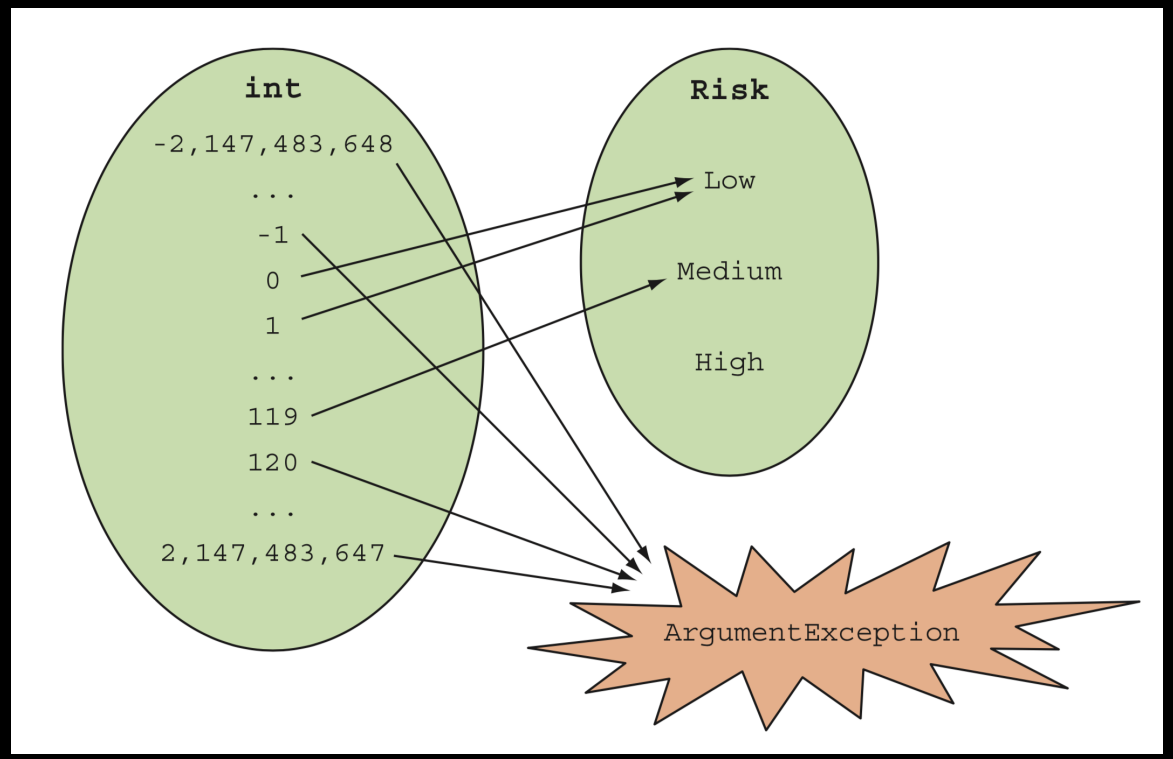
Total Functions

mappings defined for **every** the elements of the domain

```
public static Risk ClassifyRisk(int age)
{
  if (age < 0 || age > 119)
  {
    return new Risk { Invalid = true };
  }
  ...
}
```

Avoid partial functions

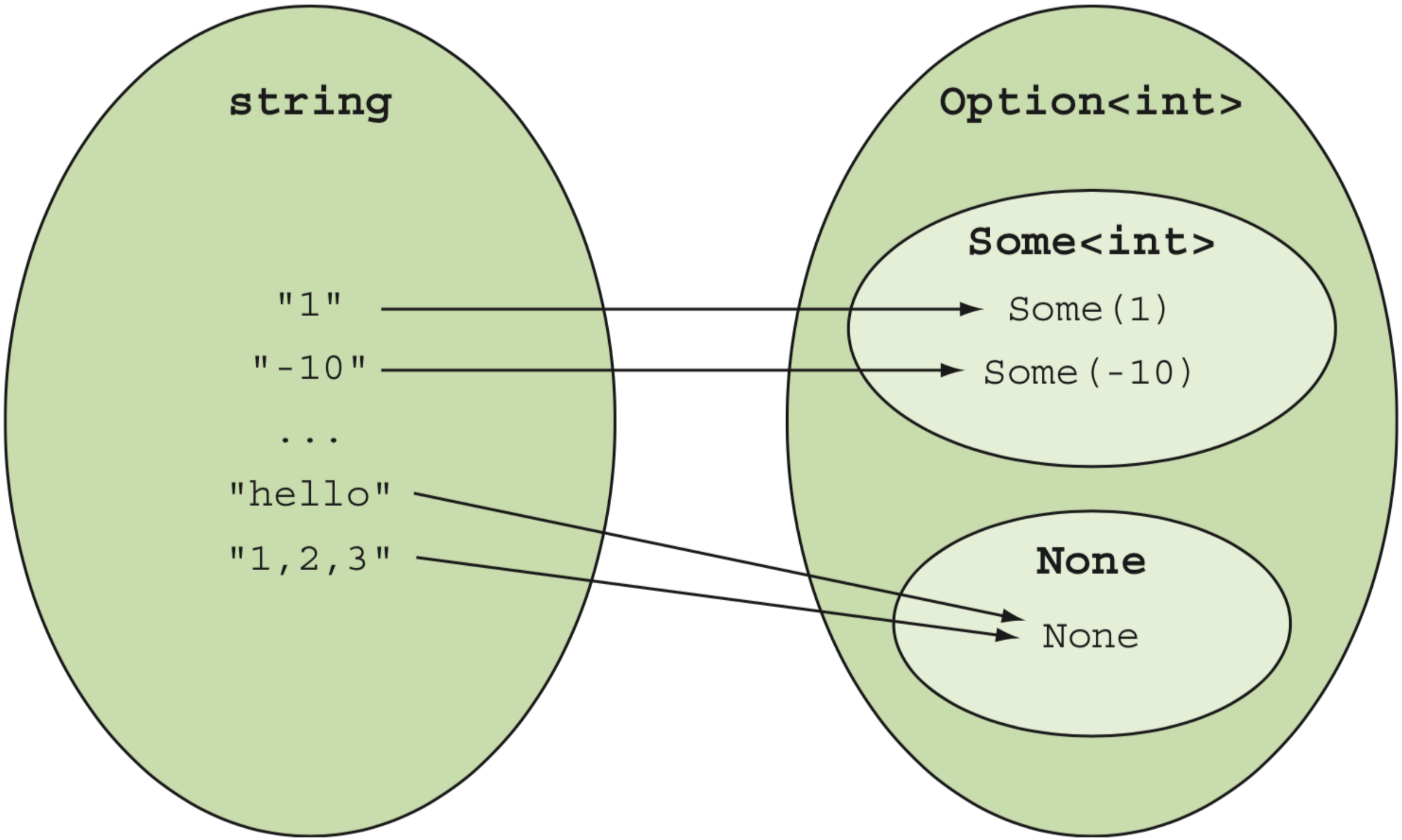




Option, a natural solution to Partial Functions

- Models the possible absence of data
- A container that wraps a value or no value
- **Option<T>** can have values of **None** or **Some(T)**
- Gain robustness by using **Option** instead of **null** (compile time safety)
- Perfect for modeling nullable complex types
- Natural way for encoding partial functions

parseInt as a total function



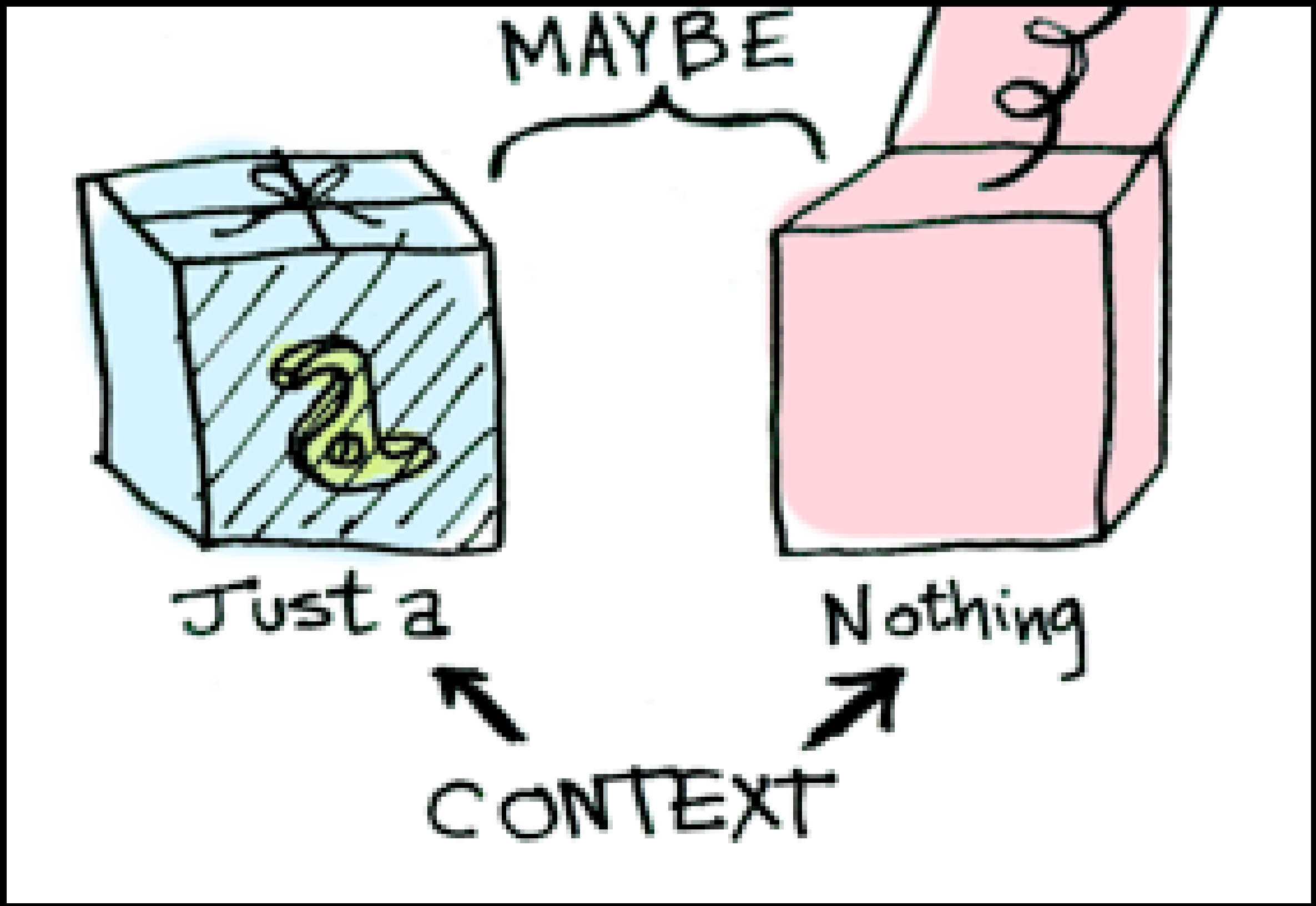
```
using LanguageExt;

public static Option<Risk> ClassifyRisk(int age)
{
  if (age < 0 || age > 119)
  {
    return None;
  }
  var result = ...
  return Some(result);
}
```

Functional Effects

- Coding at different levels of abstraction
- Regular (**T**) vs Elevated values (**M<T>**)
- It is a way to add an **Effect** to the underlying type
- Elevated values are values within a specific **Context**
- Depending on the **Context**, additional behavior is added to our value **T**
- Effects are essentially a container that wraps a value

Option



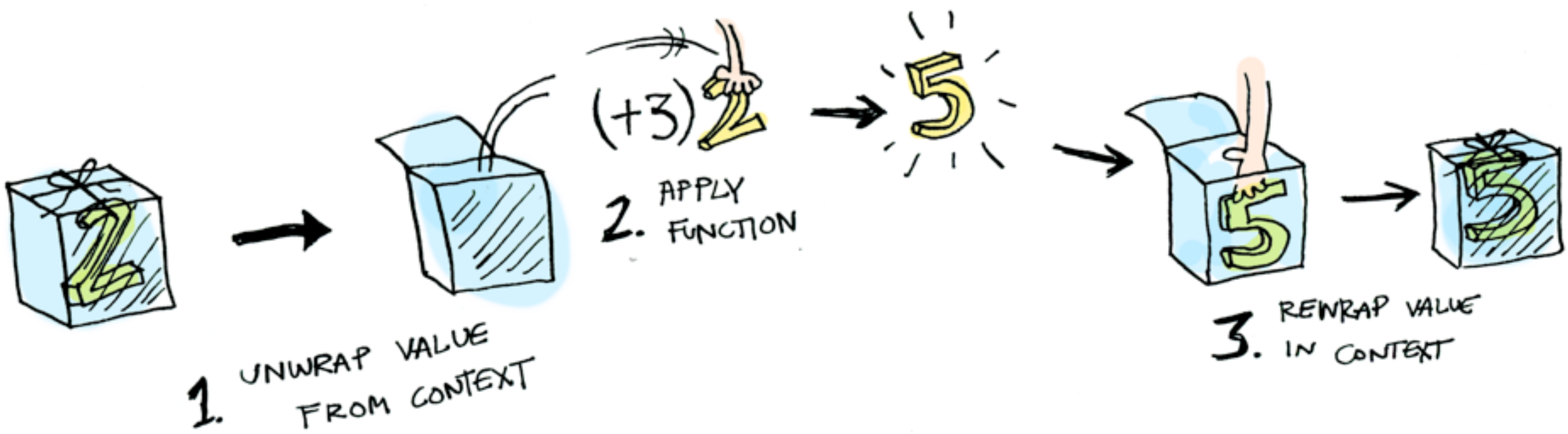
- Maybe == Option
- Just(T) == Some(T)
- Nothing == None

Changing the value in the context

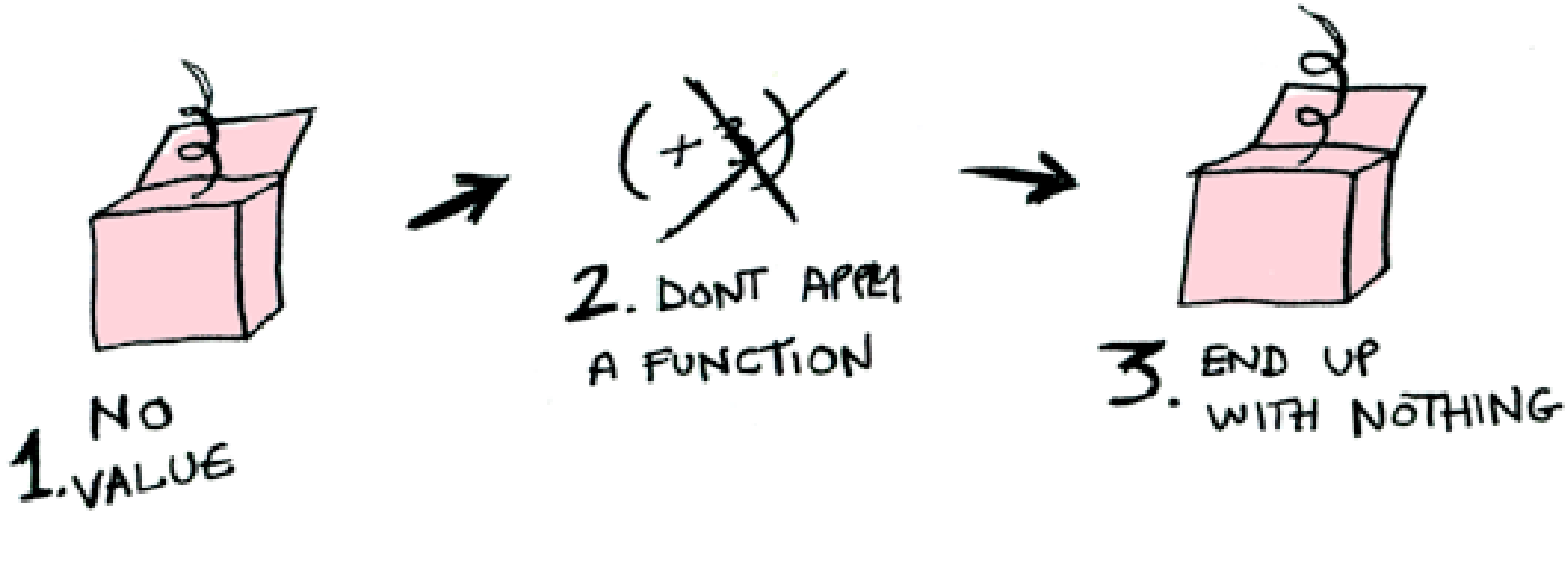
using **map** to change the value within a context

map allows us to apply a function to a wrapped value

Changing the value in the context



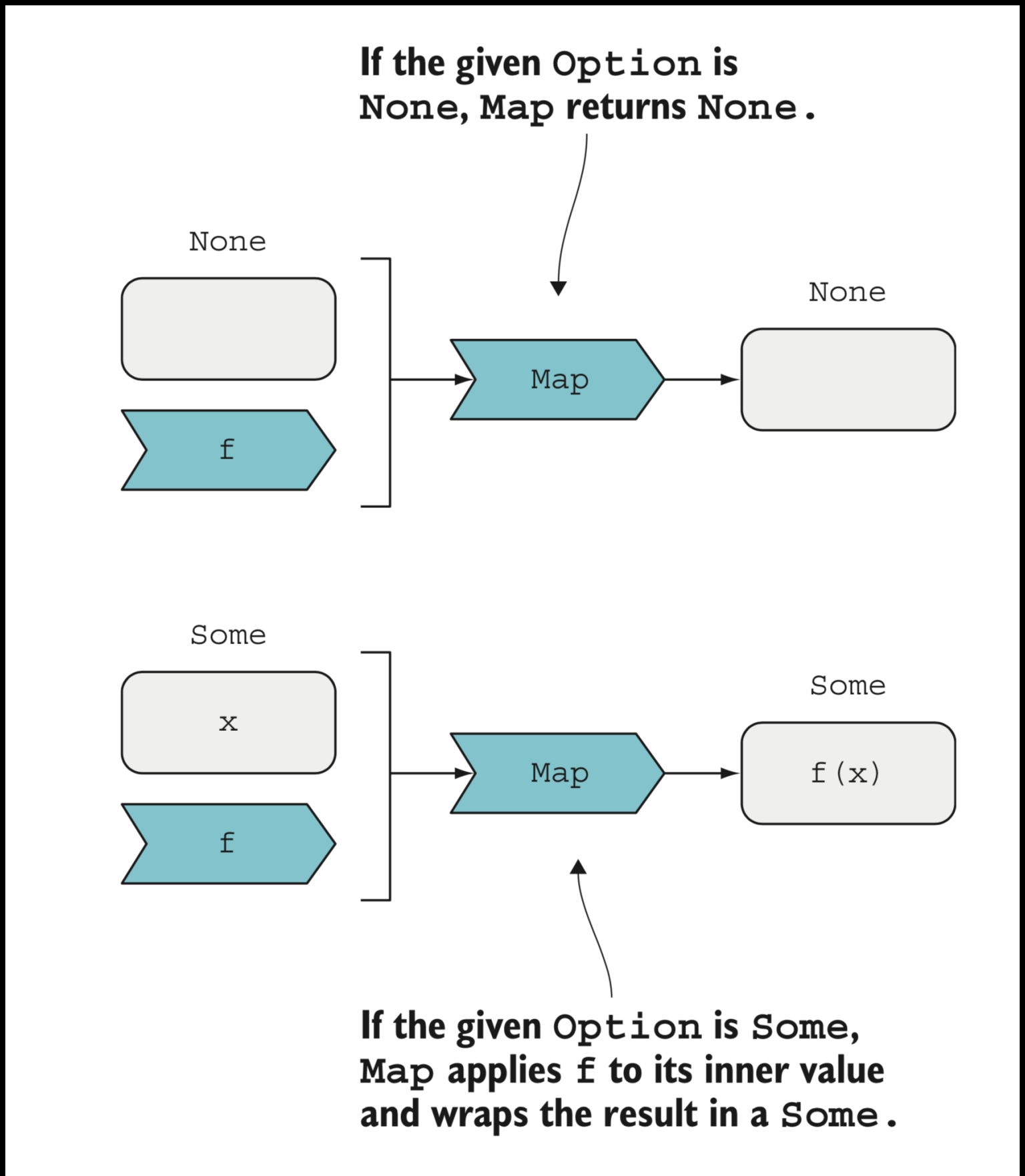
Changing the value in the context



NOTHING GOES IN, NOTHING COMES OUT

CAN'T EXPLAIN THAT!

Option's Properties



Unit as a functional replacement for *void*

- It's not type (it's magical)
- Doesn't allow things like *Option<void>*
- Doesn't compose

Unit as a functional replacement for *void*

Unit

- Type that models absence of data without the problems of void
- Usual return type of functions that cause side effects and return void
  e.g Logging
- Can only take the singleton/const value of *unit*
- Allow us to replace *Action* with *Func<Unit>*
  to benefit from the compositional power of Functional Effects.
  More on this later

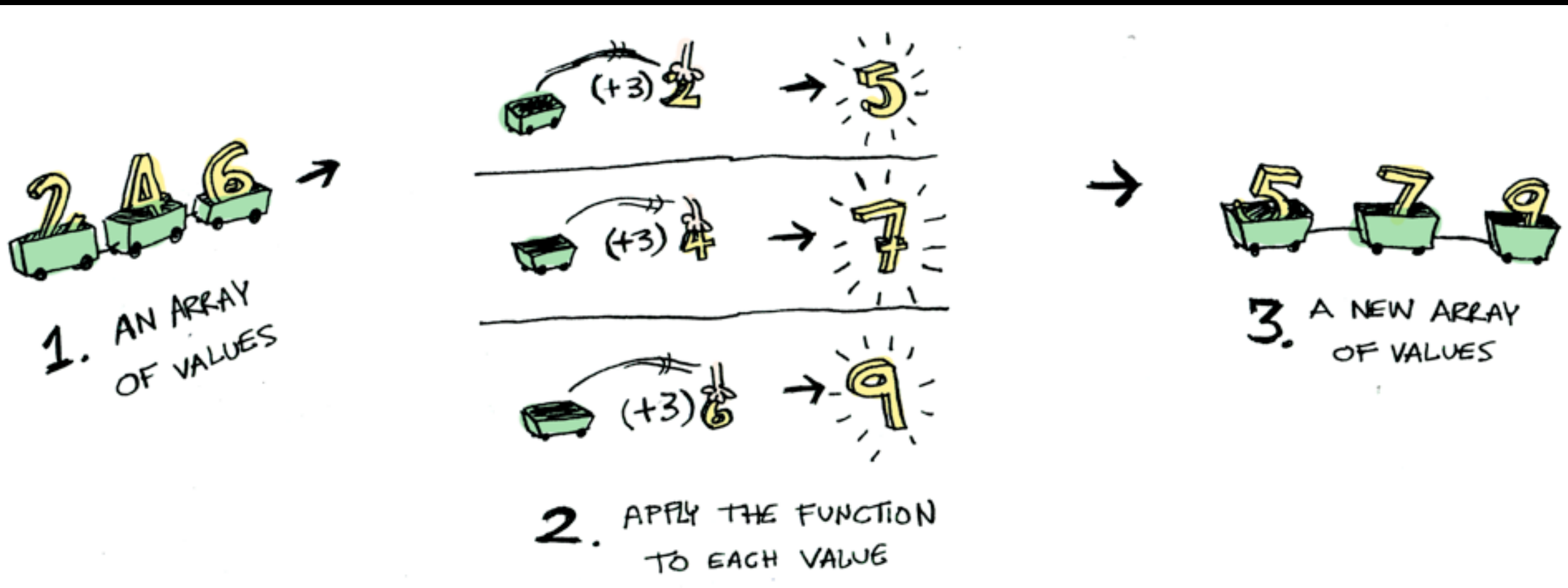
Example 3

- Using Option effect
- Using Unit

Functional Effects Continued

Lists as a useful Effect

- Think of effects as containers for value(s)
- List/Arrays are effects



Functional Effects

- **Option<T>** adds the effect of *optionality effect* - not **T** but possibility of a **T**
- **IEnumreable<T>** adds the effect of *aggregation effect* - a sequence of **T**'s
- **Func<T>** adds effect of *laziness effect* - not a **T**, but a computation that can be evaluated to obtain a **T**
- **Task<T>** adds effect of *asynchrony* - not a **T**, but a promise that at some point you'll get a **T**

Functional Effects

Functional effects are commonly known as **Monad**s

A few of the useful effects:

- Option
- List
- Either, Try and Validation
- Lens
- Writer
- Reader
- State

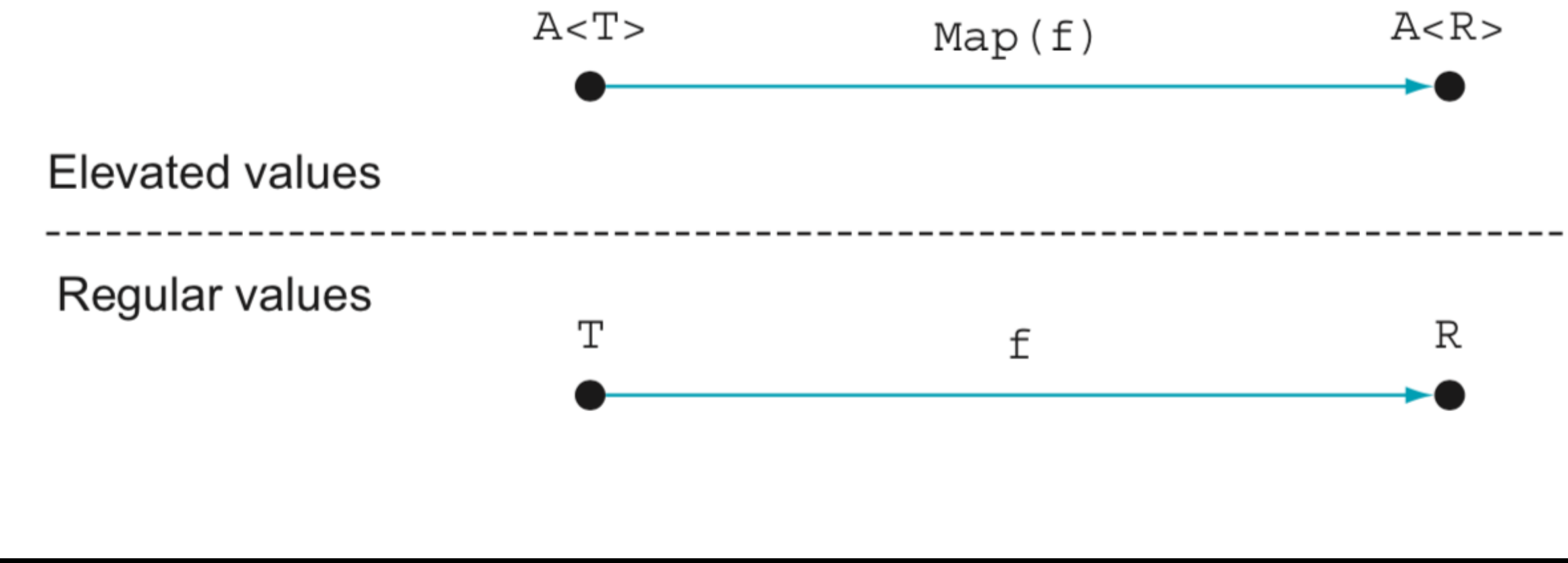

MONADS, MONADS
MONADS EVERYWHERE
memegenerator.net

Common Properties of Functional Effects

- Have implementation for
  map General FP name
  Map LanguageExt
  Select LINQ
- Have implementation for
  flatMap General FP name
  Bind LanguageExt
  SelectMany LINQ
- These HOFs make Functional Effects very composable, reusable and useful!

Map/Select

- **Apply a function to a wrapped value**
- Given a currency value of *AUD*, get *USD*
- Given a possible currency value of *Option<AUD>*, get *Option<USD>*



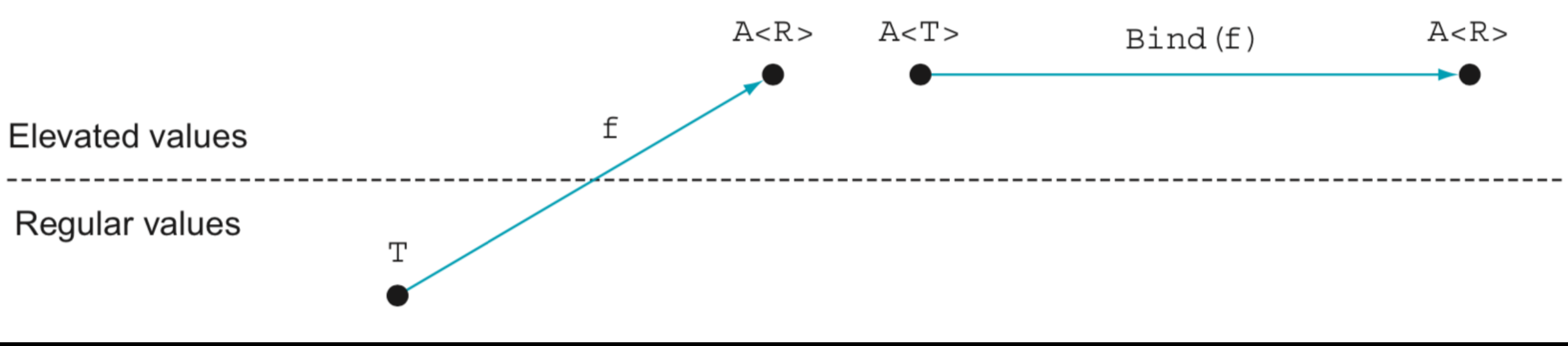
Implementation *Map* for Option

```
public static Option<R> Map<T, R>
    (this Option<T> optT, Func<T, R> f)
     => optT.Match(
          () => None,
          (t) => Some(f(t)));
```

Useful for reading or transforming the value wrapped in an effect

Bind/flatMap/SelectMany

- **Apply a function that returns a wrapped value to a wrapped value**
- given userId of type *int*
- *FetchUser(userId)* returns *Task<User>*
- *GetUserProfile(User)* returns *Task<UserProfile>*



Implementation *Bind* for Option

```
public static Option<R> Bind<T, R>
    (this Option<T> optT, Func<T, Option<R>> f)
     => optT.Match(
          () => None,
          (t) => f(t));
```

Useful for chaining multiple functions that return effects and sequential application flows

Why use Functional Effects?

- Increases code composability
- Increases code reusability
- Increase the robustness of the code
- More informative type signatures (types as docs)
- Extra compile time safety makes certain class of unit tests redundant
- Reduces complexity as most problems often fall within a few categories
  and are commonly solved by mixing a few functional effects (often like LEGO)

Functional Patterns

- OOP gives us lots of useful patterns (e.g adapter, factory, strategy, ...)
- Functional effects and data structures are FP's "patterns"
- Common techniques and functions to combine/compose elevated values
- Topic of another day ...

OO pattern/principle	FP pattern/principle
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory pattern	• Yes, functions
• Strategy pattern	• Oh my, functions again!
• Decorator pattern	• Functions
• Visitor pattern	• Functions □