



September 2020
INNOQ Technology Lunch

Functions, Functors and Categories

INNOQ

Images taken from unsplash.com



Ludvig Sundström

Consultant
INNOQ Deutschland GmbH

ludvig.sundstroem@innoq.com





BEGIN.

"Programming s*cks"

- Peter Hunt Welch

"Every programmer occasionally, when nobody's home, turns off the lights, pours a glass of scotch, puts on some light German electronica, and opens up a file on their computer. It's a different file for every programmer. Sometimes they wrote it, sometimes they found it and knew they had to save it. They read over the lines, and weep at their beauty, then the tears turn bitter as they remember the rest of the files and the inevitable collapse of all that is good and true in the world."

"This file is Good Code. It has sensible and consistent names for functions and variables. It's concise. It doesn't do anything obviously stupid. It has never had to live in the wild, or answer to a sales team. It does exactly one, mundane, specific thing, and it does it well. It was written by a single person, and never touched by another. [...]"

"Every programmer starts out writing some perfect little snowflake like this. Then they're told on Friday they need to have six hundred snowflakes written by Tuesday, so they cheat a bit here and there and maybe copy a few snowflakes and try to stick them together or they have to ask a coworker to work on one who melts it and then all the programmers' snowflakes get dumped together in some inscrutable shape and..."



What can we learn from the snowflake?

- What is good about them?

- What is good about them?
- Why doesn't this goodness seem to scale in practice?

- What is good about them?
- Why doesn't this goodness seem to scale in practice?
- Could it scale in theory?

In this talk

In this talk

- Develop a mental framework of what we mean by **problem solving**

In this talk

- Develop a mental framework of what we mean by **problem solving**
- Define a **category** to reinforce this model in different contexts

In this talk

- Develop a mental framework of what we mean by **problem solving**
- Define a **category** to reinforce this model in different contexts
- Transcend the world of maths and programming with **functors**

Problem solving 101



Nick

@Zorchenhimer



Found this in production today. I need a drink.

```
public static bool CompareBooleans(bool orig, bool val)
{
    return AreBooleansEqual(orig, val);
}

internal static bool AreBooleansEqual(bool orig, bool val)
{
    if(orig == val)
        return false;
    return true;
}
```

12:54 AM · 31 May 19 · [Twitter Web Client](#)

2,519 Retweets **7,054** Likes



- We reason about code ...

- We reason about code ...
- ...in order to write good programs, and don't think twice



A snowflake ...

- has sensible, consistent names

A snowflake ...

- has sensible, consistent names
- is concise

A snowflake ...

- has sensible, consistent names
- is concise
- does what it is supposed to do

A snowflake ...

- has sensible, consistent names
- is concise
- does what it is supposed to do
- additionally doesn't do anything stupid

A snowflake ...

- has sensible, consistent names
- is concise
- does what it is supposed to do
- additionally doesn't do anything stupid
- is understandable

- Elegant code \Leftrightarrow Understandable code

- Elegant code \iff Understandable code
- Understandable code \iff code, broken up into just big enough chunks

- Elegant code \iff Understandable code
- Understandable code \iff code, broken up into just big enough chunks
- Divided and conquered, by the books

// Print whether the read string has an even length

```
int main() {  
    string str;  
    getline(cin, str);  
    int len = str.length();  
    int is_even = len % 2 == 0;  
    printf("%d");  
    return 0;  
}
```

```
hasEvenLength :: String -> String
```

```
hasEvenLength = show . even . length
```

```
-- Print wether the read string has an even length
```

```
main = interact hasEvenLength
```



```
length :: String -> Int
```

```
even :: Int -> Bool
```

```
show :: Bool -> String
```

```
showEvenLength :: String -> String
```

```
showEvenLength = show . even . length
```

```
-- Print whether the read string has an even length
```

```
main = interact showEvenLength
```

- Call it procedures or functions, they are just a way of describing "smaller problem"

- Call it procedures or functions, they are just a way of describing "smaller problem"
- Programming is about composing them, yielding structure

- Call it procedures or functions, they are just a way of describing "smaller problem"
- Programming is about composing them, yielding structure
- To study structure, it helps to make composition it explicit

- Call it procedures or functions, they are just a way of describing "smaller problem"
- Programming is about composing them, yielding structure
- To study structure, it helps to make composition it explicit
- (Program) structure \iff (function) composition

Next: Let's see how we can study composition

A taste of Category Theory

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch



ADDISON-WESLEY PUBLISHING COMPANY, INC.

Category Theory

- Is the science of patterns through composition

Category Theory

- Is the science of patterns through composition
- Abstracts structure across different fields

Category Theory

- Is the science of patterns through composition
- Abstracts structure across different fields
- Applies well to programming ...

```

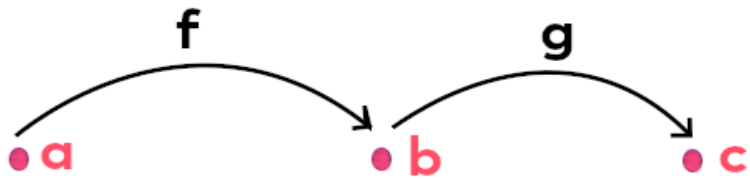
    }
    public static int isEven(int a) {
        if (a == 0) return 1;
        if (a == 2) return 1;
        if (a == 4) return 1;
        if (a == 6) return 1;
        if (a == 8) return 1;
        if (a == 10) return 1;
        if (a == 12) return 1;
        if (a == 14) return 1;
        // TODO: Add more checks.
        return 0;
    }
}

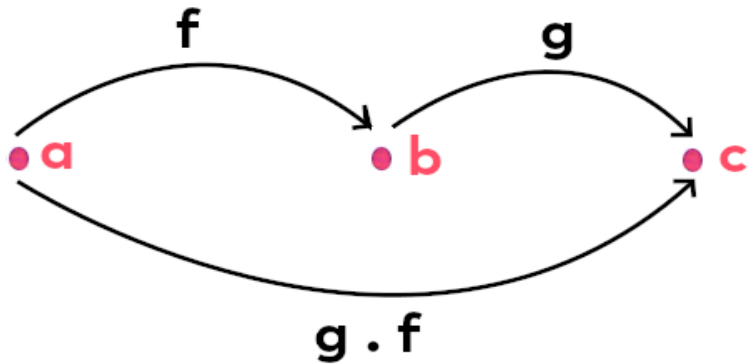
```

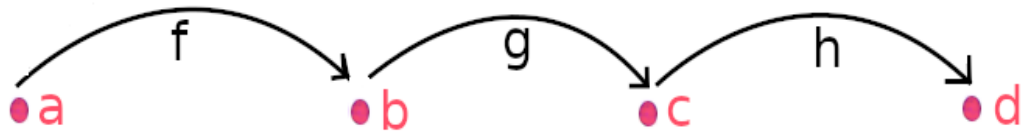
Category Theory

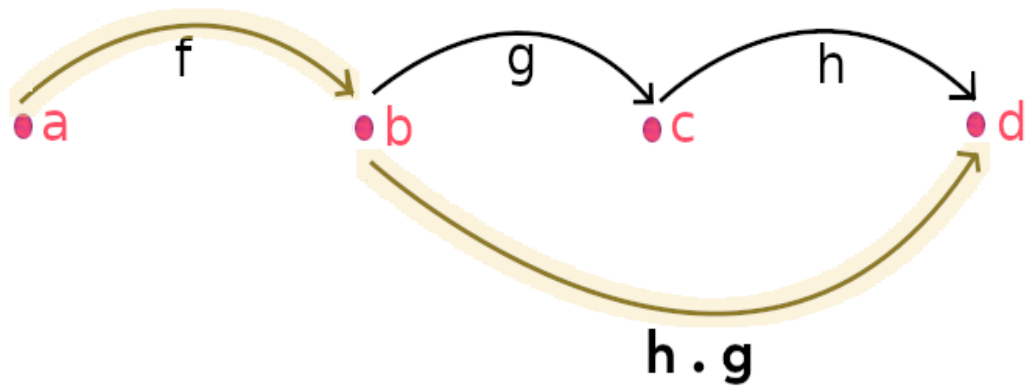
- Is the study of composition
- Is the science of patterns
- Is a language that abstracts structure across different fields
- Applies well to programming ...
- ...because programming is all about structure (of our problems)

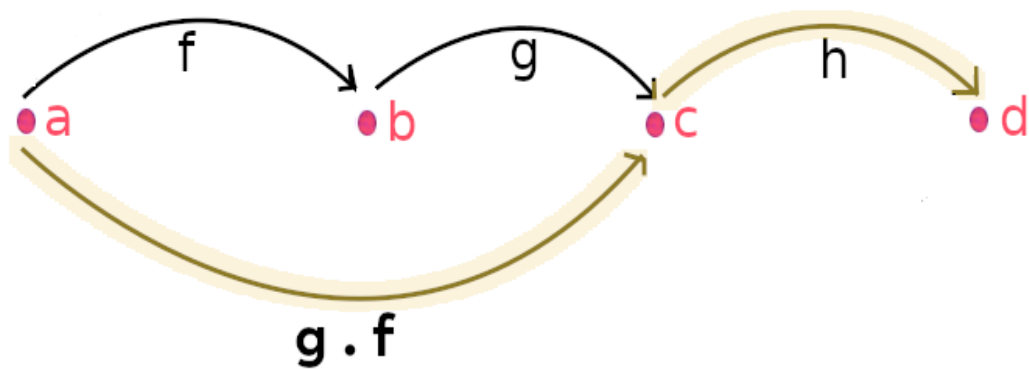
Let's define a category



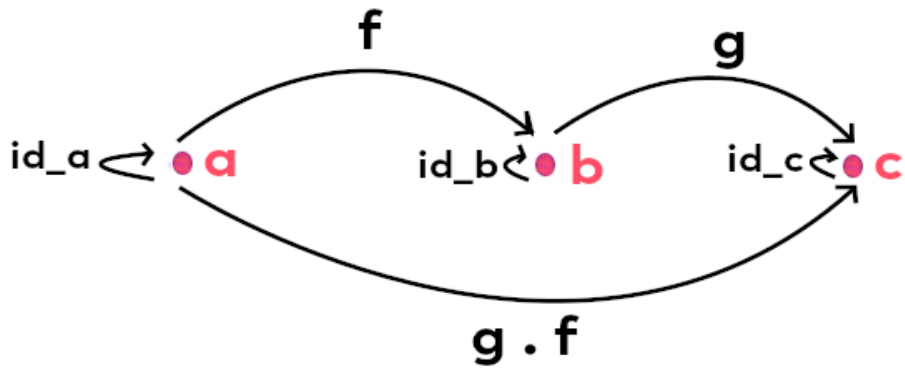


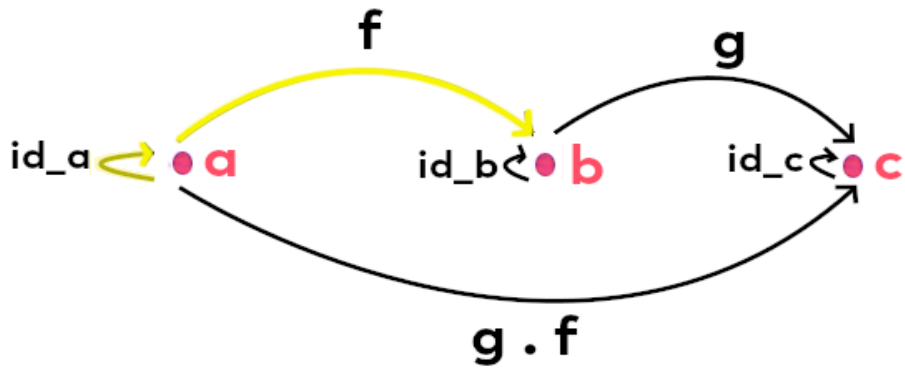


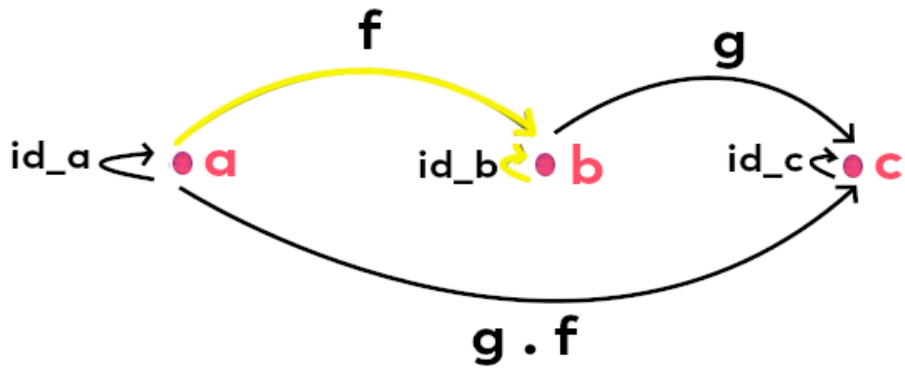












- That's it

- That's it
- Simple definition, but can be used to derive a surprising amount of properties

- That's it
- Simple definition, but can be used to derive a surprising amount of properties
- We give categories meaning, then compare and reuse structure across different contexts

How to define meaning

1. Say what the objects are

How to define meaning

1. Say what the objects are
2. Say what the arrows are

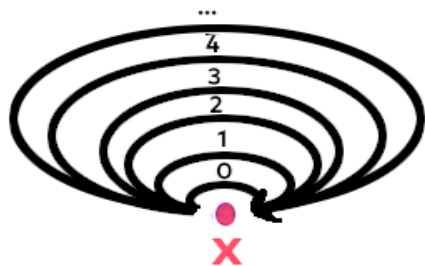
How to define meaning

1. Say what the objects are
2. Say what the arrows are
3. Say what the identities are

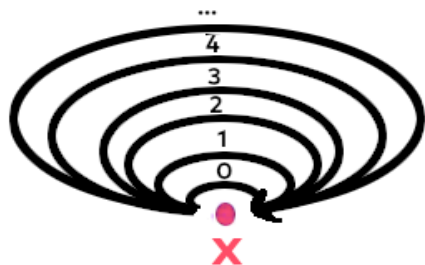
How to define meaning

1. Say what the objects are
2. Say what the arrows are
3. Say what the identities are
4. Say how the arrows compose

Category M

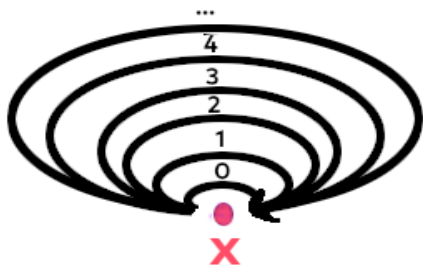


Category M



$$\text{Obj}(M) = \{x\}$$

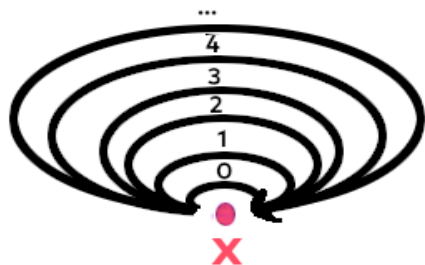
Category M



$$\text{Obj}(\mathbf{M}) = \{x\}$$

$$\text{Hom}(\mathbf{M}) = \mathbb{N}$$

Category M

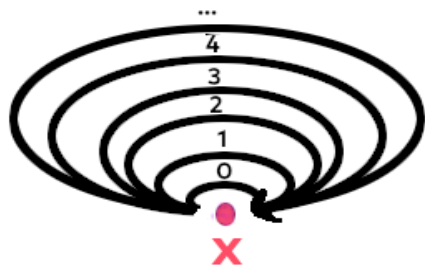


$$\text{Obj}(M) = \{x\}$$

$$\text{id}_x = 0$$

$$\text{Hom}(M) = \mathbb{N}$$

Category M

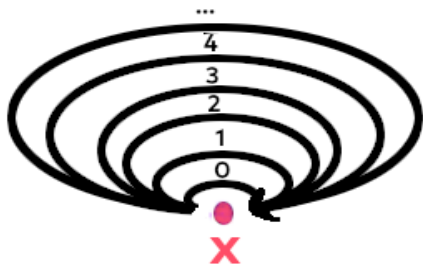


$$\text{Obj}(M) = \{x\}$$

$$\text{id}_x = 0$$

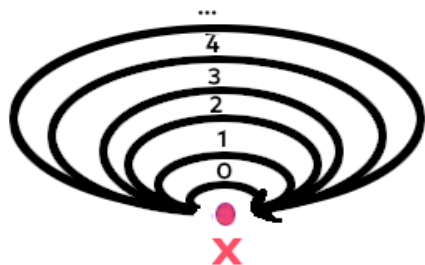
$$\text{Hom}(M) = \mathbb{N} \quad \text{composition} = (+)$$

Category M



Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

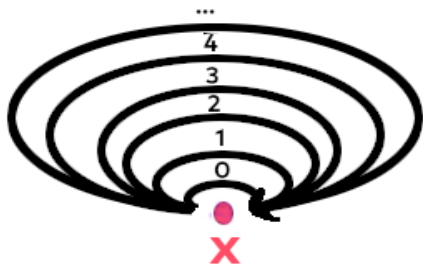
Category M



Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

Identity: Any arrow can be composed with identity $(n + 0)$

Category M

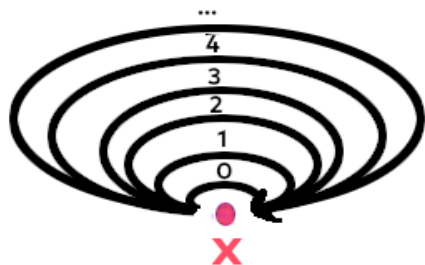


Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

Identity: Any arrow can be composed with identity $(n + 0)$

Associativity: Composing arrows $(i + j) + k$ is the same as composing $i + (j + k)$

Category M



Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

Identity: Any arrow can be composed with identity $(n + 0)$

Associativity: Composing arrows $(i + j) + k$ is the same as composing $i + (j + k)$

All logic is encoded in the composition

Programmers Category

The Category of Types and Functions

The Category of Types and Functions

1. Objects \rightarrow Types

The Category of Types and Functions

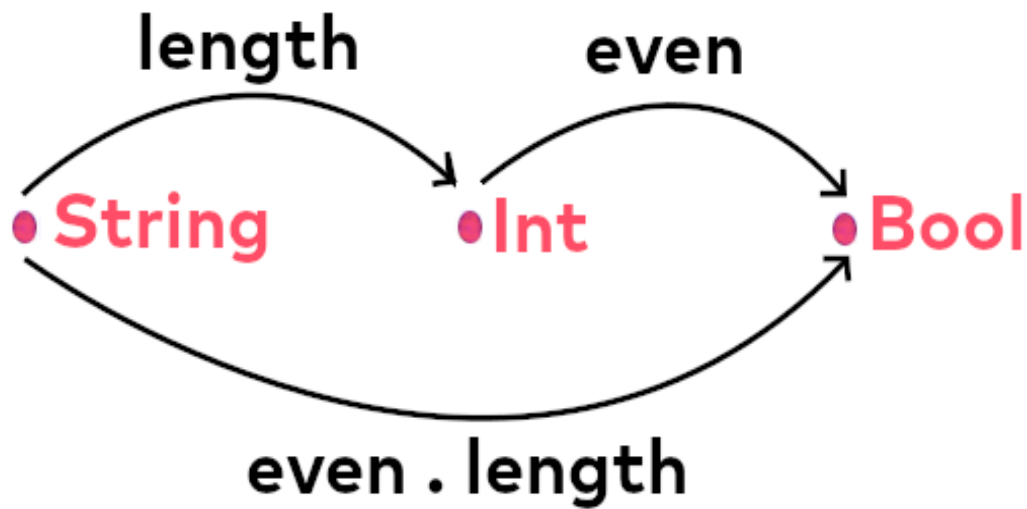
1. Objects \rightarrow Types
2. Arrows \rightarrow Functions

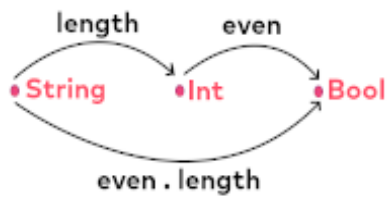
The Category of Types and Functions

1. Objects \rightarrow Types
2. Arrows \rightarrow Functions
3. Identity $\rightarrow \text{fn}(x) \Rightarrow x$

The Category of Types and Functions

1. Objects \rightarrow Types
2. Arrows \rightarrow Functions
3. Identity $\rightarrow \text{fn}(x) \Rightarrow x$
4. Composition \rightarrow Function composition



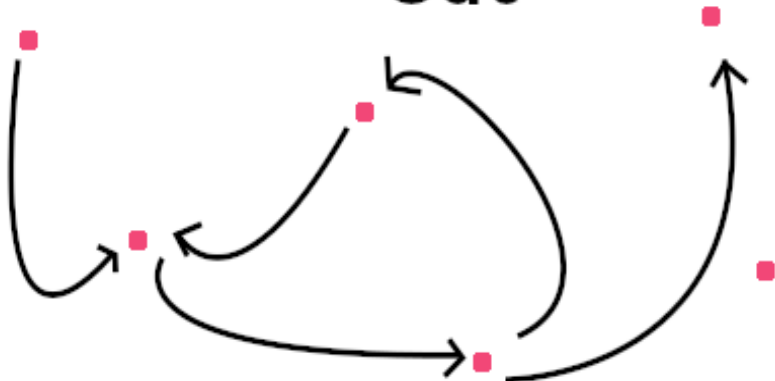








Cat



$\text{Ob}(\mathbf{Cat}) = \text{categories}$

$\text{Hom}(\mathbf{Cat}) = \text{functors}$

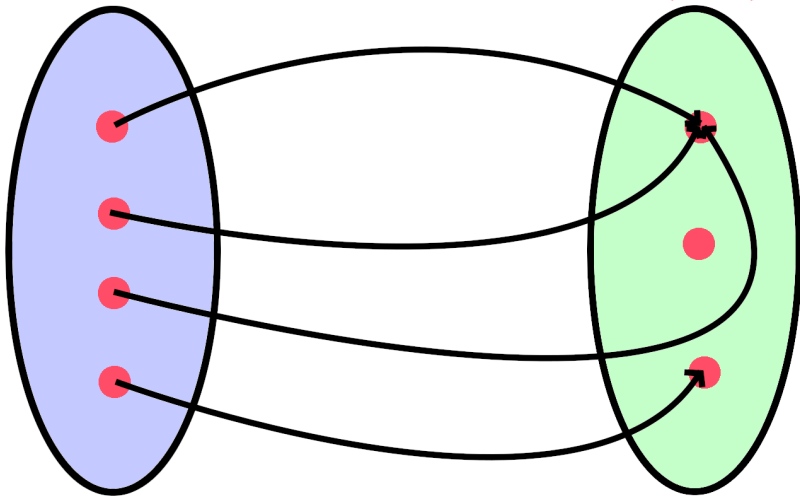


The Functor

- Is a mapping between categories

X

$f(X)$

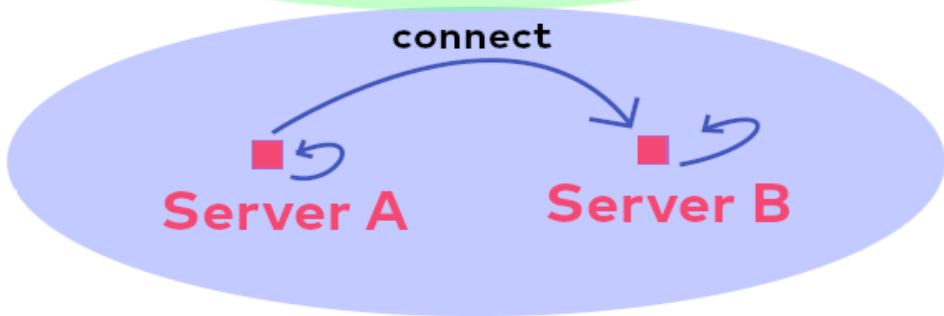
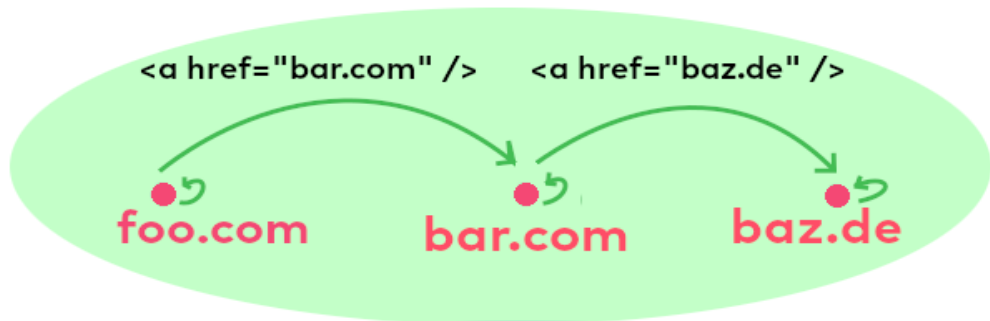


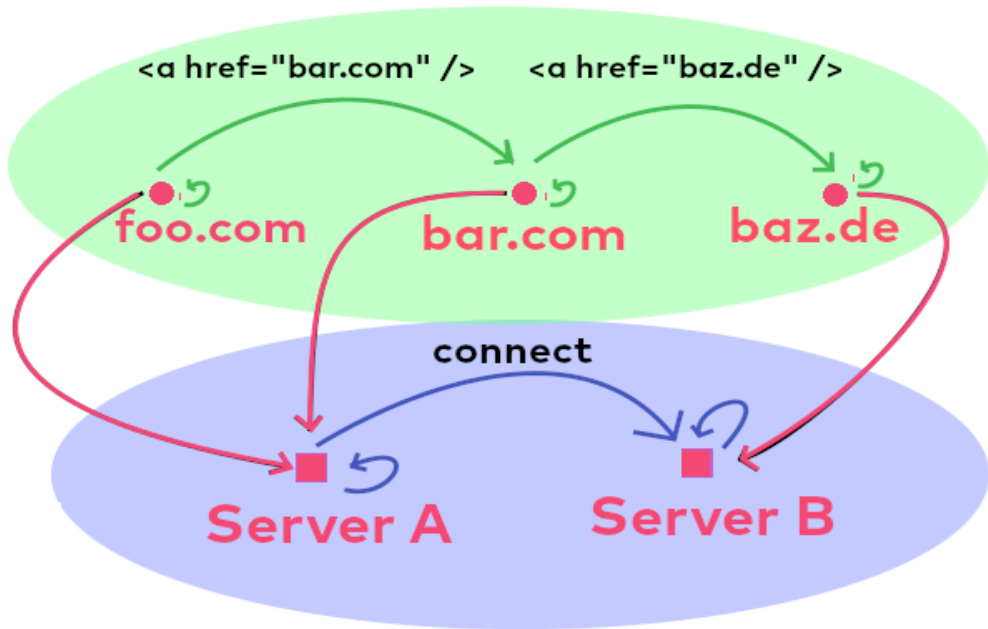
The Functor

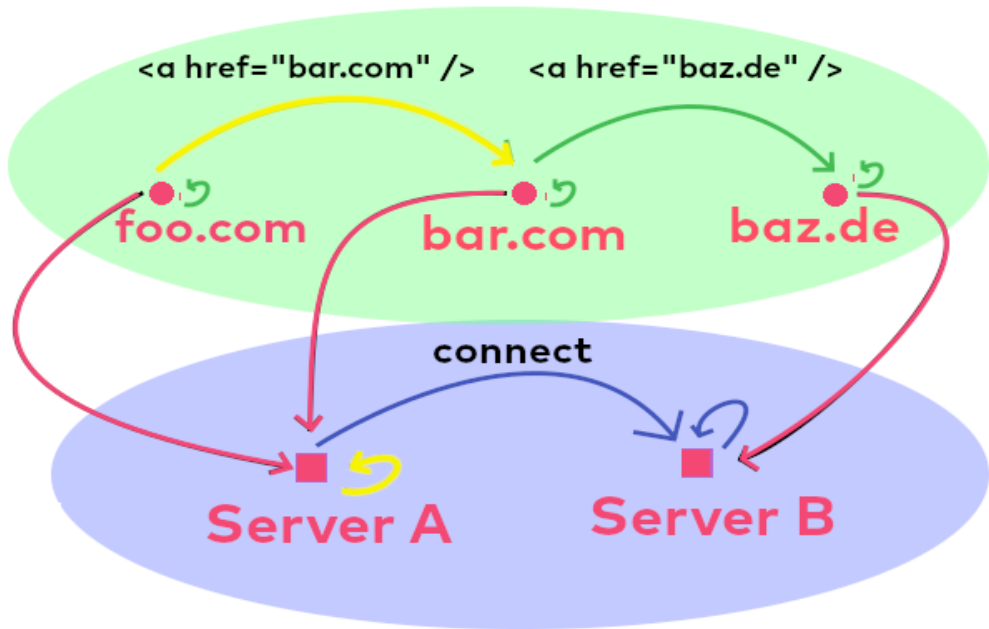
- Is a mapping between categories
- Maps objects into objects and arrows into arrows ...

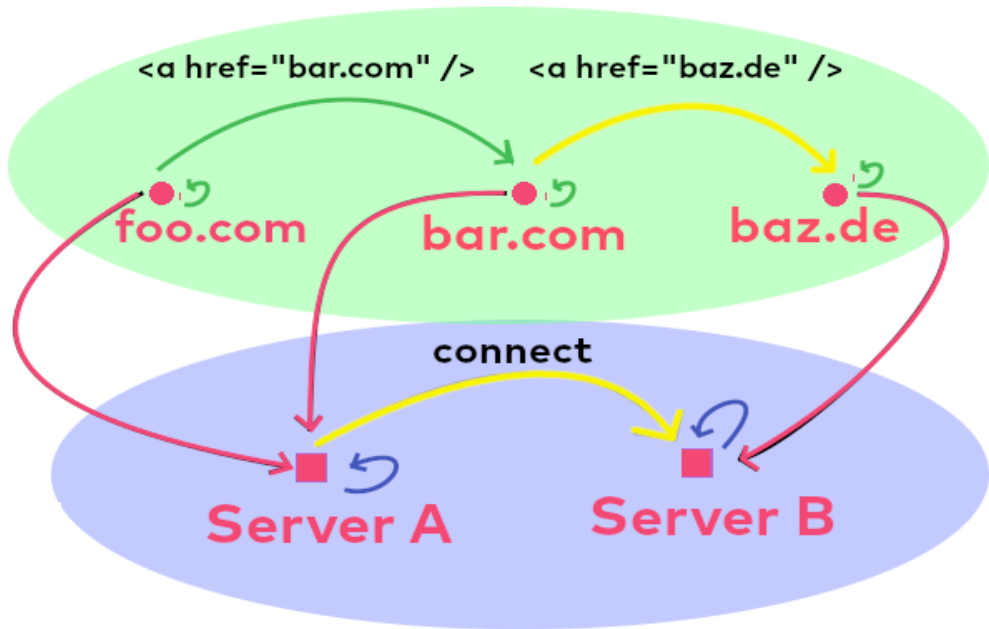
The Functor

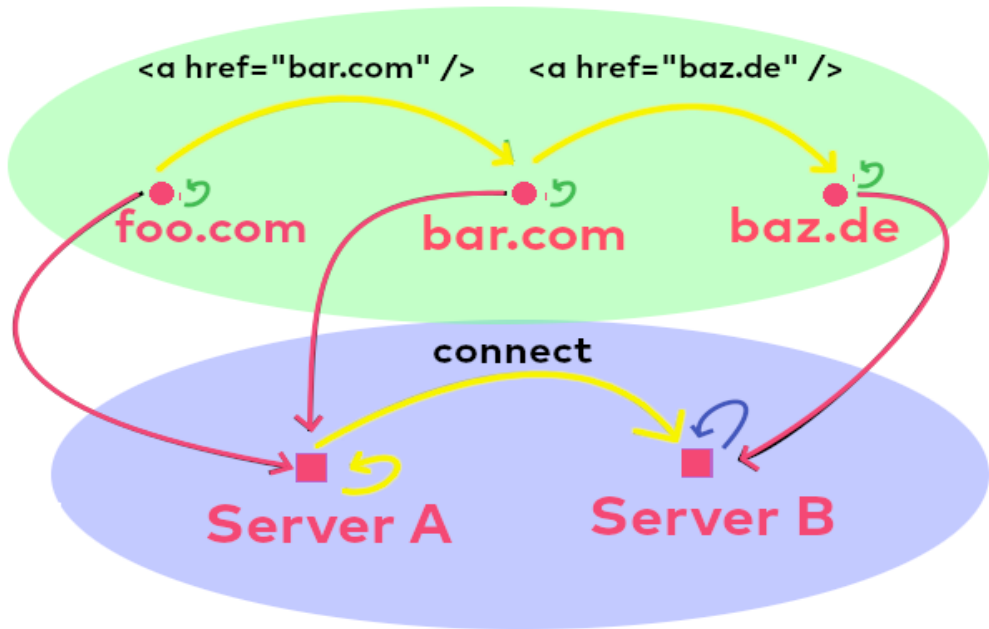
- Is a mapping between categories
- Maps objects into objects and arrows into arrows ...
- ...Preserving structure!

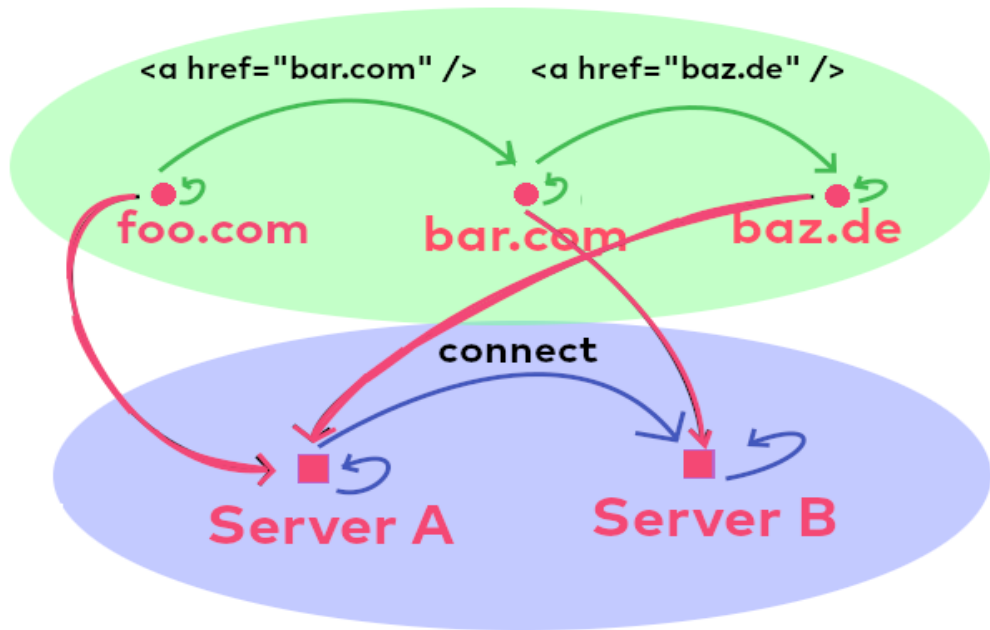








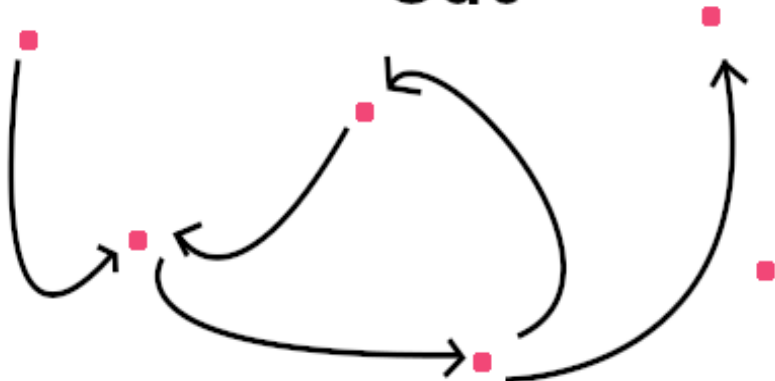




Endofunctors

- Remember Cat?

Cat

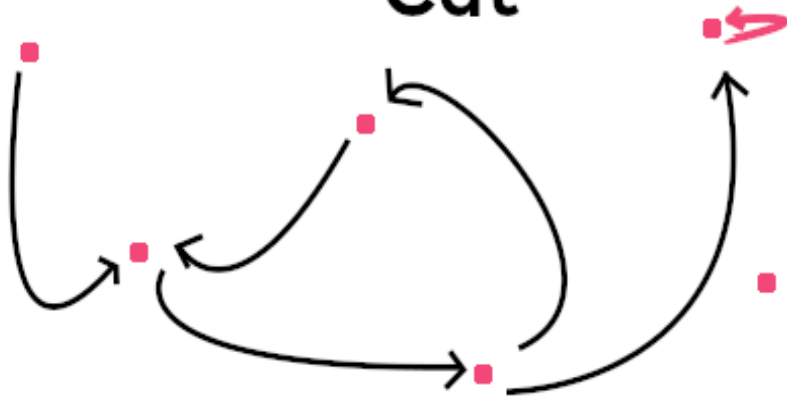


$\text{Ob}(\mathbf{Cat}) = \text{categories}$

$\text{Hom}(\mathbf{Cat}) = \text{functors}$

Cat

The category of
types and functions

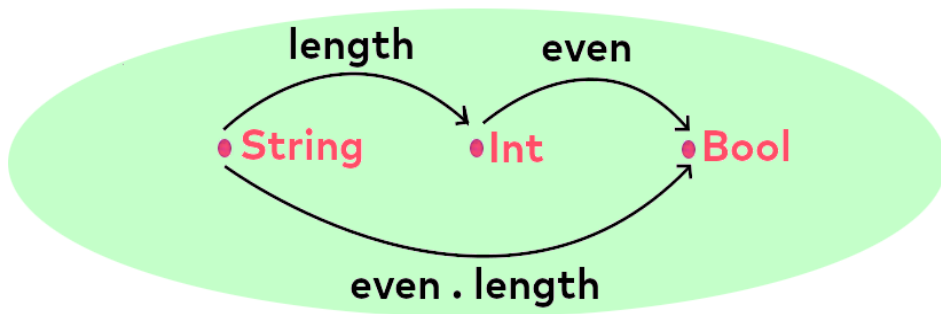
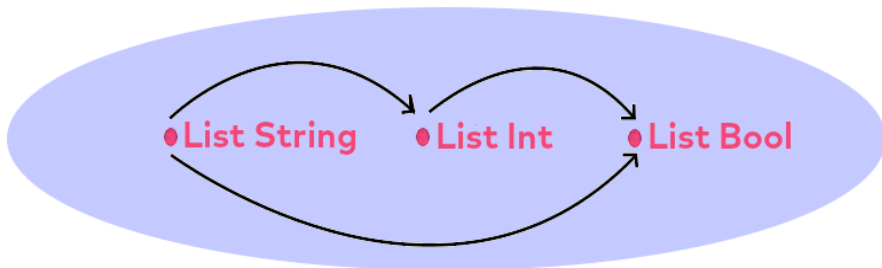


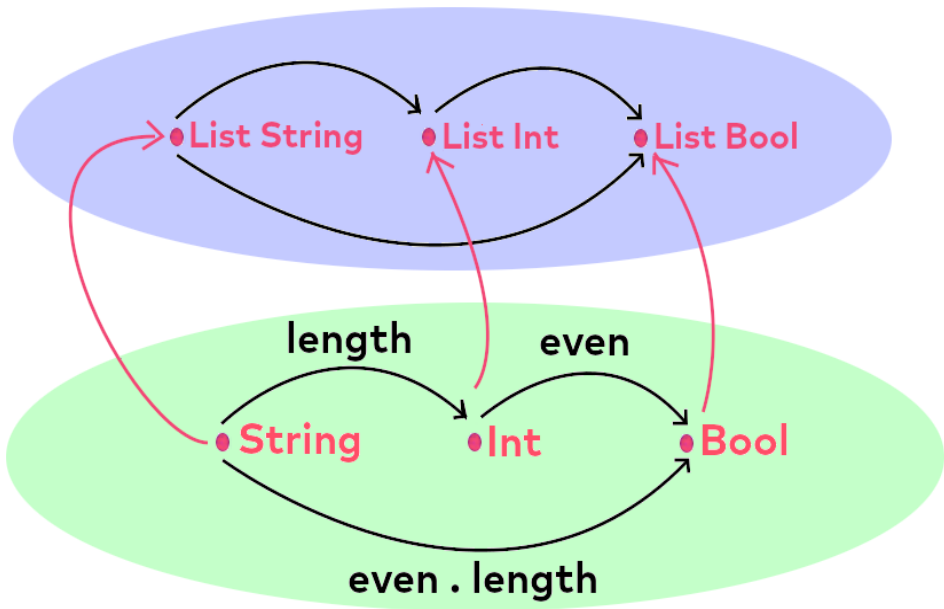
$\text{Ob}(\text{Cat}) = \text{categories}$

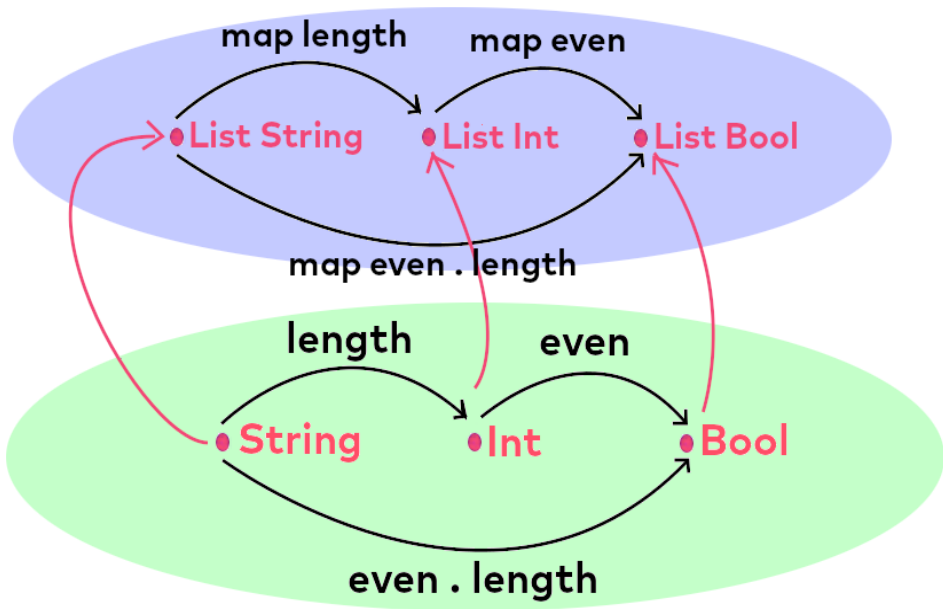
$\text{Hom}(\text{Cat}) = \text{functors}$

- Remember Cat?
- Arrows between the same object in Cat \rightarrow endofunctor

- Remember Cat?
- Arrows between the same object in Cat \rightarrow endofunctor
- The kind of functor used in programming!





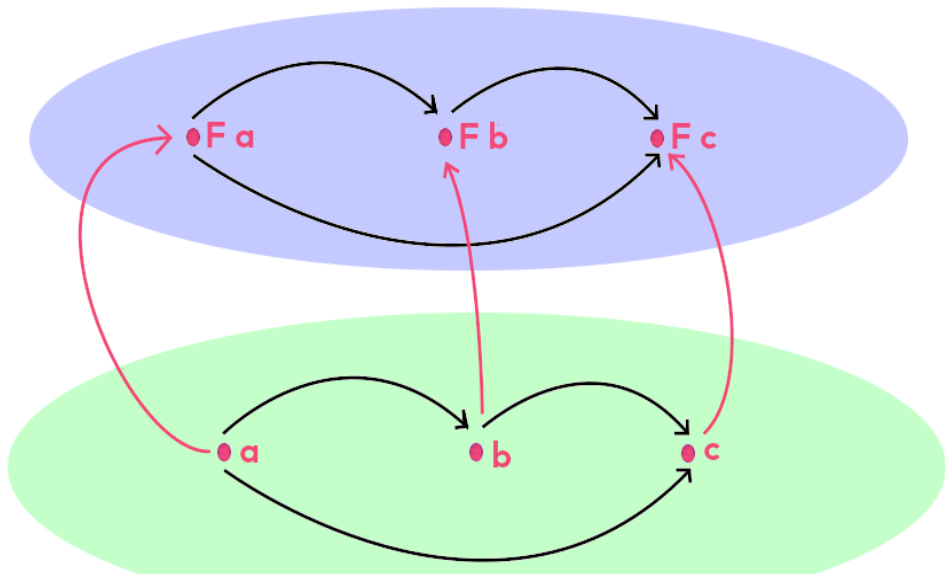


- A functor is a mapping between categories

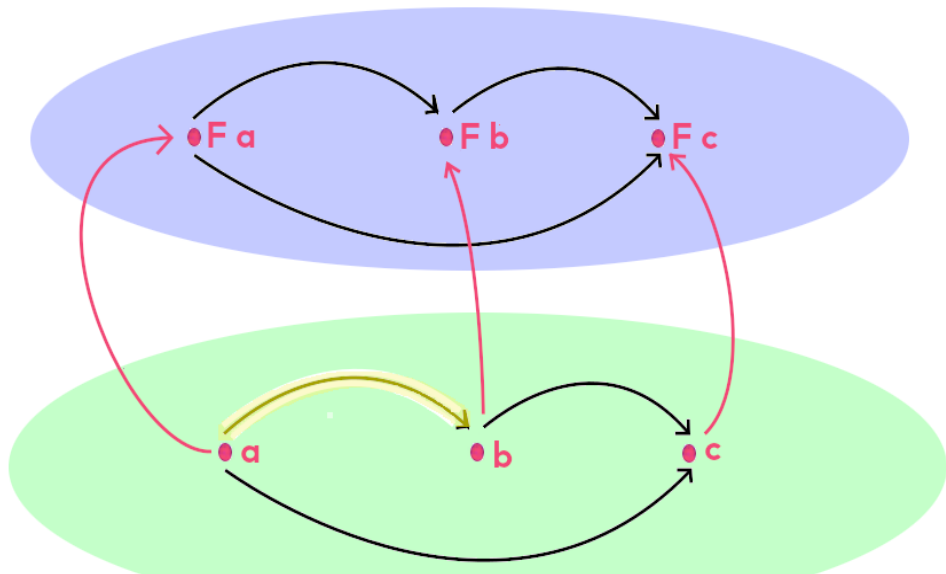
- A functor is a mapping between categories
- Used in programming when we find similar structure (e.g. for mapping `Int -> List Int`)

- A functor is a mapping between categories
- Used in programming when we find similar structure (e.g. for mapping `Int -> List Int`)
- Preserving structure \iff Preserving composition

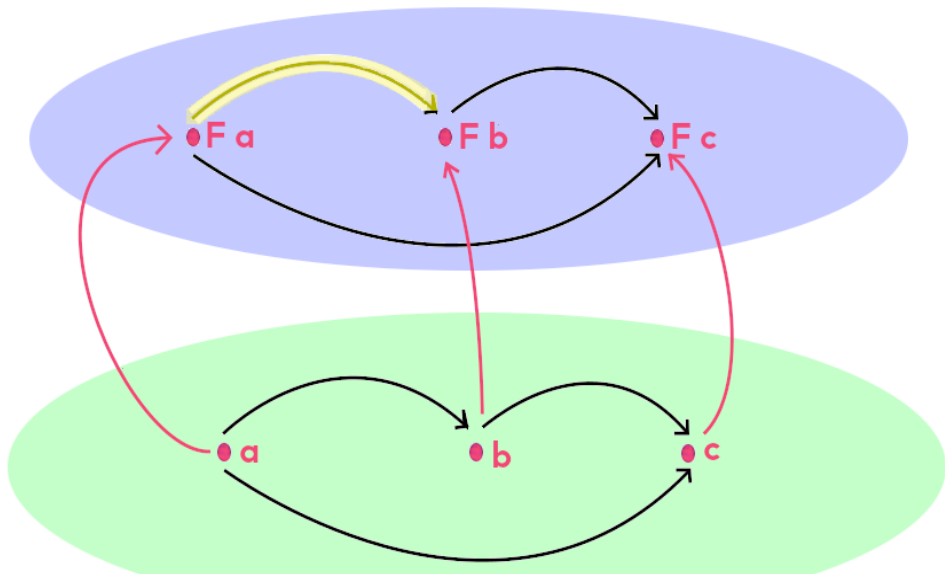
- A functor is a mapping between categories
- Used in programming when we find similar structure (e.g. for mapping `Int -> List Int`)
- Preserving structure \iff Preserving composition
- How do we describe a functor in programming?



`fmap :: Functor F => (a -> b) -> (F a -> F b)`



`fmap :: Functor F => (a -> b) -> (F a -> F b)`



`fmap :: Functor F => (a -> b) -> (F a -> F b)`

```
-- Functor interface
```

```
--
```

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
--                               Input 1: Function  
--                               ^^^^^^^^
```

```
fmap :: Functor f => (a -> b) -> (f a -> f b)
```



```
--                                Output: Enriched function  
--                                ^^^^^^^^^^^^^^^  
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

- A functor represents **new parts** of categories

- A functor represents **new parts** of categories
- In programming, it represents **new computational contexts**

- A functor represents new parts of categories
 - ▶ Retaining structure!
- In programming, it represents new computational contexts
 - ▶ Retaining structure!

Example Contexts

- List: Where computations may have multiple return values

Example Contexts

- List: Where computations may have multiple return values
- Maybe (Optional): Where failures might occur

Example Contexts

- List: Where computations may have **multiple return values**
- Maybe (Optional): Where **failures** might occur
- IO: Where **side effects** can happen

Example Contexts

- List: Where computations may have **multiple return values**
- Maybe (Optional): Where **failures** might occur
- IO: Where **side effects** can happen

→ Use the functor to abstract over the context!

List Implements fmap!

```
prompt> fmap length ["Y0", "Y00", "Y000"]  
[2,3,4]
```

```
prompt> fmap even [1..10]  
[False,True,False,True,False,True,False,True,False,True]
```

```
prompt> fmap (even . length) ["ah", "aha", "ehhhhh"]  
[True,False,True]
```

Maybe Implements fmap!

```
ghci> fmap even Nothing
```

```
Nothing
```

```
ghci> fmap length (Just "Y000")
```

```
Just 4
```

```
ghci> fmap (even . Length) (Just "Y000")
```

```
(Just True)
```

IO implements fmap!

IO implements fmap!

Get a string from the command line...

```
prompt> fmap length getLine
```

```
HELLOWORLD
```

```
10
```

... and an integer

```
prompt> fmap even getInt
```

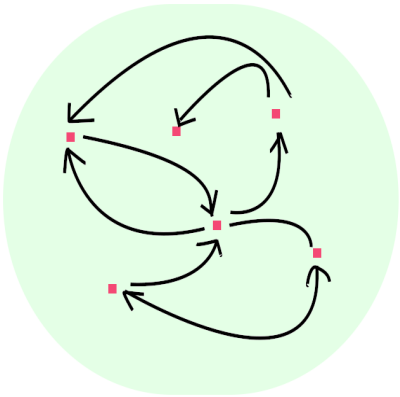
```
33
```

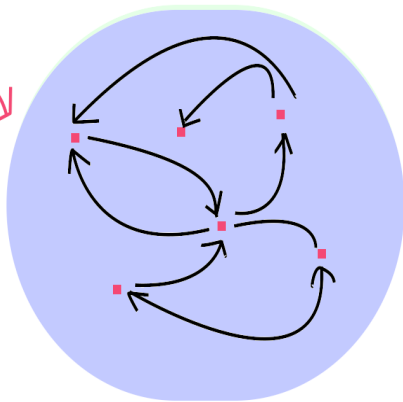
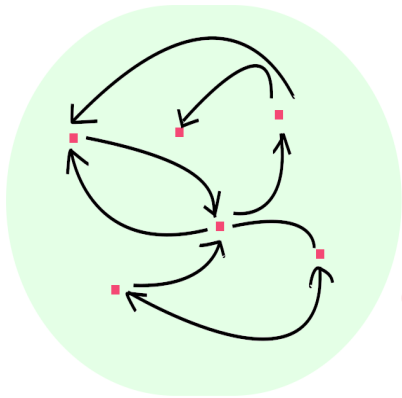
```
False
```

```
prompt> fmap (even . length) getLine
```

```
HELLO
```
























```
False
```





Instances

 Functor []	# Source	Since: 2.1
 Functor Maybe	# Source	Since: 2.1
 Functor IO	# Source	Since: 2.1
 Functor Par1	# Source	Since: 4.9.0.0
 Functor NonEmpty	# Source	Since: 4.9.0.0
 Functor ReadP	# Source	Since: 2.1
 Functor ReadPrec	# Source	Since: 2.1
 Functor Down	# Source	Since: 4.11.0.0
 Functor Product	# Source	Since: 4.8.0.0
 Functor Sum	# Source	Since: 4.8.0.0
 Functor Dual	# Source	Since: 4.8.0.0
 Functor Last	# Source	Since: 4.8.0.0
 Functor First	# Source	Since: 4.8.0.0
 Functor STM	# Source	Since: 4.3.0.0
 Functor Handler	# Source	Since: 4.6.0.0
 Functor Identity	# Source	Since: 4.8.0.0
 Functor ZipList	# Source	Since: 2.1
 Functor ArgDescr	# Source	Since: 4.6.0.0
 Functor OptDescr	# Source	Since: 4.6.0.0
 Functor ArgOrder	# Source	Since: 4.6.0.0
 Functor Option	# Source	Since: 4.9.0.0
 Functor Last	# Source	Since: 4.9.0.0
Functor First	# Source	Since: 4.9.0.0

 Functor Max	# Source	Since: 4.9.0.0
 Functor Min	# Source	Since: 4.9.0.0
 Functor Complex	# Source	Since: 4.9.0.0
 Functor (Either a)	# Source	Since: 3.0
 Functor (V1 :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor (U1 :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor ((,) a)	# Source	Since: 2.1
 Functor (ST s)	# Source	Since: 2.1
 Functor (Proxy :: Type -> Type)	# Source	Since: 4.7.0.0
 Arrow a => Functor (ArrowMonad a)	# Source	Since: 4.6.0.0
 Monad m => Functor (WrappedMonad m)	# Source	Since: 2.1
 Functor (ST s)	# Source	Since: 2.1
 Functor (Arg a)	# Source	Since: 4.9.0.0
 Functor f => Functor (Rec1 f)	# Source	Since: 4.9.0.0
 Functor (URec Char :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor (URec Double :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor (URec Float :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor (URec Int :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor (URec Word :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor (URec (Ptr ()) :: Type -> Type)	# Source	Since: 4.9.0.0
 Functor f => Functor (Alt f)	# Source	Since: 4.8.0.0
 Functor f => Functor (Ap f)	# Source	Since: 4.12.0.0
 Functor (Const m :: Type -> Type)	# Source	Since: 2.1
Arrow a => Functor (WrappedArrow a b)	# Source	Since: 2.1

The functor provides

- Consistent, predictable structural sharing

The functor provides

- Consistent, predictable structural sharing
- Instant context switching

The functor provides

- Consistent, predictable structural sharing
- Instant context switching
- With that follows ...

The functor provides

- Consistent, predictable structural sharing
- Instant context switching
- With that follows ...
 - ▶ Flexibility

The functor provides

- Consistent, predictable structural sharing
- Instant context switching
- With that follows ...
 - ▶ Flexibility
 - ▶ Code reuse

The functor provides

- Consistent, predictable structural sharing
- Instant context switching
- With that follows ...
 - ▶ Flexibility
 - ▶ Code reuse
 - ▶ Separation of concerns

The functor provides

- Consistent, predictable structural sharing
- Instant context switching
- With that follows ...
 - ▶ Flexibility
 - ▶ Code reuse
 - ▶ Separation of concerns
 - ▶ Modularity





- Thinking categorically has inspired me to ...

- Thinking categorically has inspired me to ...
 - ▶ Explore math and programming side by side

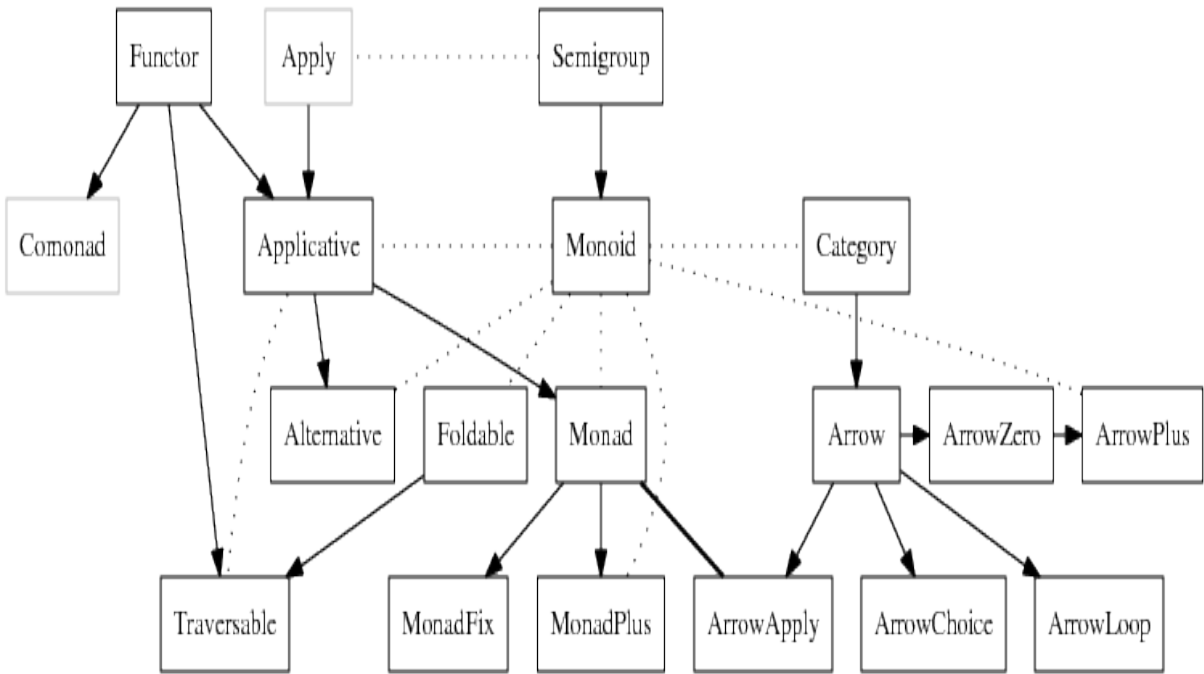
- Thinking categorically has inspired me to ...
 - ▶ Explore math and programming side by side
 - ▶ Be more curious about writing programs

- Thinking categorically has inspired me to ...
 - ▶ Explore math and programming side by side
 - ▶ Be more curious about writing programs
 - ▶ Think differently about program structure

- Thinking categorically has inspired me to ...
 - ▶ Explore math and programming side by side
 - ▶ Be more curious about writing programs
 - ▶ Think differently about program structure
 - ▶ Be more confident in knowing when structure can be reused

- Thinking categorically has inspired me to ...
 - ▶ Explore math and programming side by side
 - ▶ Be more curious about writing programs
 - ▶ Think differently about program structure
 - ▶ Be more confident in knowing when structure can be reused

The functor is just the beginning ...



Summary

- Fundamentally, we solve all problems the same way: splitting, solving, and composing

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, and composing
- Structure emerges through composition

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, and composing
- Structure emerges through composition
- Functional programming makes composition explicit

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, and composing
- Structure emerges through composition
- Functional programming makes composition explicit
- Category theory helps us formally reason about structure

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, and composing
- Structure emerges through composition
- Functional programming makes composition explicit
- Category theory helps us formally reason about structure
- And provides concepts like the functor that lets us put perfect little snowflakes into complicated contexts, without thinking twice 😊

Thank you! Questions?



Ludvig Sundström
ludvig.sundstroem@innoq.com



+49 1516 1181270



@l5und

innoQ Deutschland GmbH

Krischerstr. 100
40789 Monheim a. Rh.
Germany
+49 2173 3366-0

Ohlauer Str. 43
10999 Berlin
Germany

Ludwigstr. 180E
63067 Offenbach
Germany

Kreuzstr. 16
80331 München
Germany

c/o WeWork
Hermannstrasse 13
20095 Hamburg
Germany

innoQ Schweiz GmbH

Gewerbestr. 11
CH-6330 Cham
Switzerland
+41 41 743 01 11

Albulastr. 55
8048 Zürich
Switzerland

- 0** : <http://www.stilldrinking.org/programming-sucks>
- 1** : <https://insights.stackoverflow.com/survey/2019#technology>
- 2** : https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence
- 3** : https://en.wikipedia.org/wiki/Design_Patterns
- 4** : https://golem.ph.utexas.edu/category/2012/01/vorsicht_funktor.html

Laws

- Associativity in a category: $h \cdot g \cdot f = (h \cdot g) \cdot f = h \cdot (g \cdot f)$
- Identity in a category (for $f :: a \rightarrow b$): $f \cdot \text{id}_a = f, \text{id}_b \cdot f = f$
- Functor retains structure under composition:
if $h = g \cdot f$, then $F h = F g \cdot F f$
- Functor retains structure under identity: $F \text{id}_a = \text{id}_{\{F a\}}$

Curry-Howard Isomorphism

- $\text{Void} \iff \text{False}$
- $() \iff \text{True}$
- $\text{Product Types} \iff \text{OR}$
- $\text{Sum Types} \iff \text{AND}$
- $A \rightarrow B \iff \text{If } A \text{ then } B$

Notes on functor as a typeclass

Interfaces methods are always associated with an object instance. In other words, there is always an implied 'this' parameter that is the object on which the method is called. All inputs to a type class function are explicit.