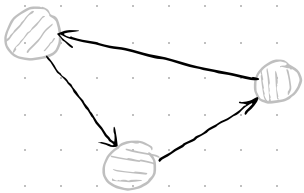
What is a Monad?

„A Monad is a Monoid in the category of endofunctors."

That's an easy explanation, isn't it? Ok, let's start this explanation from the beginning. This definition has its source in category theory.

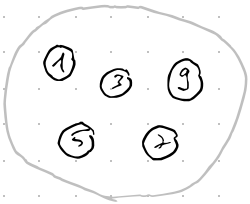
>>= introduction to category theory

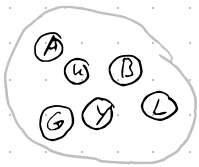
Category theory is a field of mathematics which studies the relations between categories.

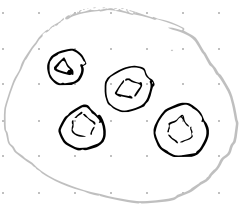


But what is a category?

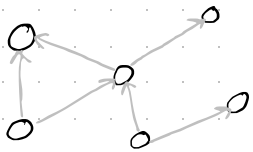
Categories are collections of objects. An object can represent anything:





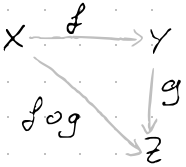


Objects can be mapped to other objects. This is formally known as a morphism: $f : X \to Y$.



A morphism maps an object of a category to another object of the same category. They can be seen as functions which take one object from a category as an argument.

Compositions of morphisms are represented in a commutative diagram:



There is no difference in applying f to X and g to this result or applying the composition $f \circ g$ directly to X.

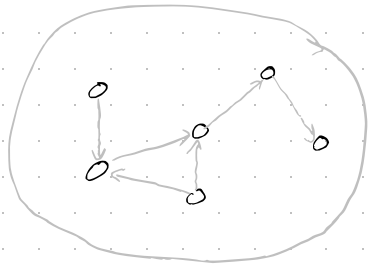
There is one special kind of morphism:

the **Endomorphism**

which points back to its original object:

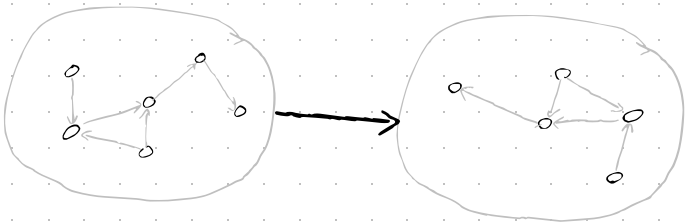
$f : X \to X$

A collection of objects and the morphisms between each other is defined as a category.



In category theory you can have as many categories as you want.

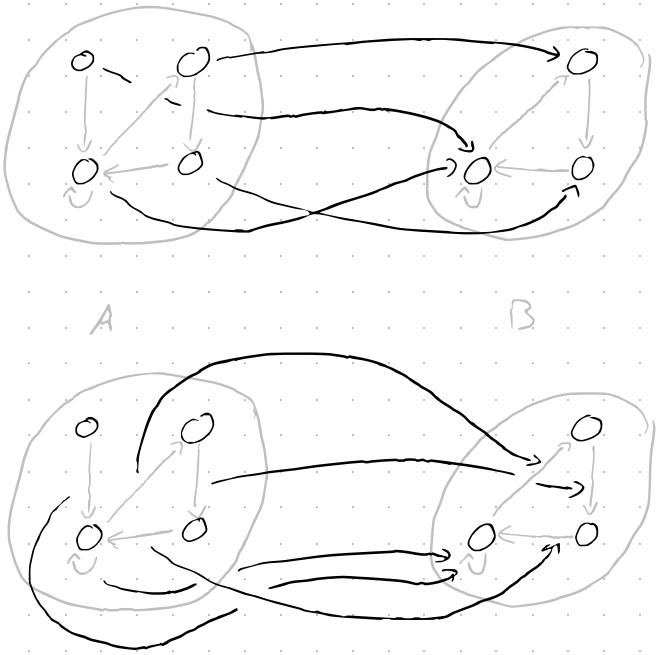
A category can be mapped to another category with **Functors**:



Functors can be seen as morphisms between categories but they do a lot more.

A functor from category A to category B will map every object in A to an object in B

and every morphism in A do a morphism in B. Multiple objects or morphisms in A can map to the same one in B.

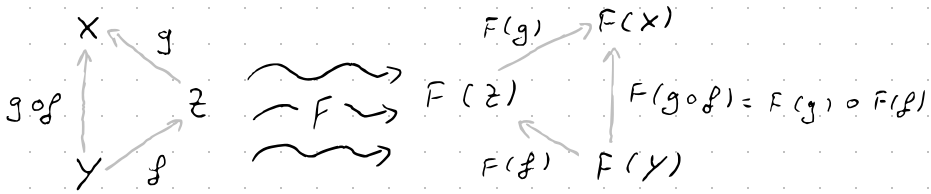


The important rule is that structure must be preserved. This means that for a given path (a composition of morphisms $f \circ g \circ h$ follows a path in the commutative diagram) the end result should not depend on when you went through it.

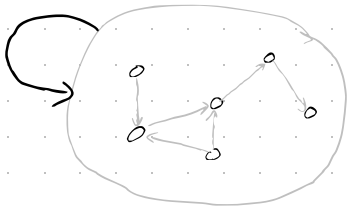
In mathematical terms it can be described the following way:

Let C, D be categories. A functor $F: C \to D$ is a mapping that

- associates each object X in C to an object $F(X)$ in D
- associates each morphism $f: X \to Y$ in C to a morphism $F(f): F(X) \to F(Y)$ in D such that the following conditions hold:
  1. $F(id_X) = id_{F(X)} \quad \forall X \in C$
  2. $F(g \circ f) = F(g) \circ F(f) \quad \forall f: X \to Y$ and $g: Y \to Z$ with $f, g \in C$



Just like objects can be mapped to themselves with an endomorphism a category can be mapped to itself with an endofunctor.



Unlike endomorphisms this can have an observable effect because a endofunctor can map an object to a different one in the same category.

Categories can come alongside **Operators** which are essential functions taking two elements of the category and returning something else.

These operators can have some common properties. A simple property is **totality** which means that two elements of a category create another element of the same category.

In programming this applies do many operators:

string + string → string

int * int → int

However there are some exceptions:

int ÷ int → float

Associativity is another common property which means that the hierarchy of the execution of the operation does not have any effect on the result.

$$A \times B \times C = (A \times B) \times C = A \times (B \times C)$$

Some categories have an **identity element** where a combination with another element returns the element itself:

$$id \times x = x$$

For example the number zero for numeric additions:

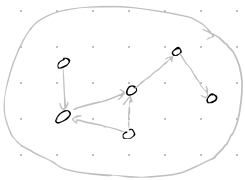
$$x + 0 = x \quad \text{or} \quad 3 + 0 = 3$$

A category with a total and associative operation is a **semigroup**. When the category also has an identity element it becomes a **Monoid**.

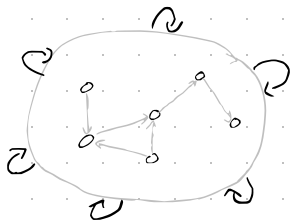
Now we can put the definition from the beginning together. In reality the sentence is wrong and the definition should be:

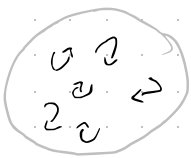
> A monad in X is a monoid in the category of endofunctors of X

In other words: Take a random category X. What is inside X doesn't matter. Then take all the endofunctors of X:

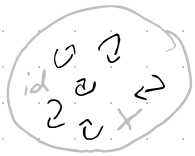


and put all of them in a separate category:





We now have the category of endofunctors of X. To make it a monoid we simply give it a total and associative operator as well as an identity element.



This category is a **Monad.**

==< What does this have to do with programming?

Let's take a look at the following Python example:

```
class Wrapper:
    def __init__(self, value):
        self.value = value
    def map(self, function):
        result = function(self.value)
        return Wrapper(result)
```

This is an implementation of the endofunctor pattern because calling the map method with a function as an argument results in an object of the same category (Wrapper) with another object.

Now what if we require the function being passed to map return an instance of the wrapper? In this implementation the map method calls the function and now operates on Wrapper objects. If we rewrite it to the following code:

```
def map(self, function):
    result = function(self.value)
    return self + result
```

In this case the map method is a total operation acting on the Wrapper type. In practise this operator is associative:

```
a.map(b).map(c)  is equivalent to
a.map(b.map(c))
```

and the conversion `X -> Wrapper(X)` where `X` is already a Wrapper object acts as an identity element.

In total the map method forms a monoid within the category of Wrappers which is a category of endofunctors

<*> How can we use this in programming?

Sometimes we have to convert one value into another by applying multiple operations one after the other.

We can do this by chaining function calls with the return value of a function being the input for the next

```
func1( func2( func3( 7 ) ) )  -> 16
```

But this is not always possible.
Imagine we want to query the age of the best friend from a person whose information is stored in a database.

```
username = "Justin"
userObject = database.fetch(username)
userFriend = userObject.friends
bestFriend = userFriends.first()
birthday = bestFriend.DateBirthday
```

This might be an easy task but in the example any one of the single instructions could fail because the data is not existing. Therefore a lot of checks have to be made in between the operation in order to prevent a function call on a None value:

```
userObject = database.fetch(username)
if userObject != None:
    userFriend = userObject.friends
    if userFriend != None:
        bestFriend = userFriends.first()
        if bestFriend != None:
            birthday = bestFriend.DateBirthday
```

This is not a fun way of programming.
Let's rewrite this by defining the wrapper type Maybe which stores a value:

```
class Maybe:
    def __init__(self, value):
        self.value = value

    def bind(self, func):
        value = func(self.value)
        return Maybe(value)
```

We also give this class a bind function which takes a function as an argument, applies the function to the value and returns Maybe object with the new value.
With this class we can rewrite our previous code to a chain of operations via the bind method of a Maybe object:

```
birthday = Maybe("Justin")
    .bind(database.fetch)
    .bind(lambda x: x.friends)
    .bind(lambda x: x.first())
    .bind(lambda x: x.DateBirthday)
```

Now there is only one place where functions are being called which is within the bind method. This means we can add any extra logic there and it will be applied at every step. P.ex. let's add a check that returns the object without any execution when it contains a None:

```
def bind(self, func):
    if self.value == None:
        return self
    value = func(self.value)
    return Maybe(value)
```

This whole process has two essential advantages:

1. The database query gets more readable
2. Important changes to check for invalid values can be made at a single place.

Here we used a **Monad** as a design pattern to abstract our data query pipeline by wrapping a value into a type.

In comparison the first version of our database query is imperative whereas the monad based version is declarative because it describes what we want and not how to do it.

In some languages data is or can be set as an immutable. This design pattern does not overwrite any data and therefore would also work in languages like haskell.