Functional programming concepts

Everything is a function

Johan Eikelboom

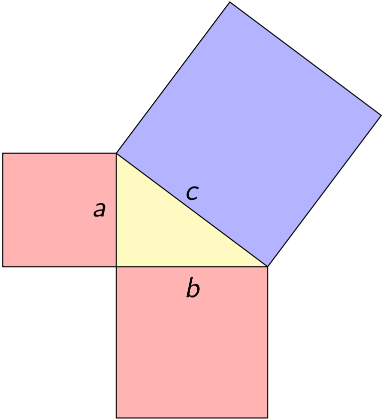
VXCompany

29 november 2022

Overview

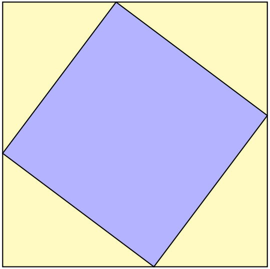
- Mathematical reasoning and programming
- Paradigms, Functions, High order functions.
- Advanced concepts: Functors and Monads
- Not: programming language syntax

Pythagoras

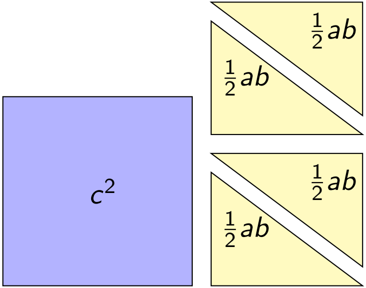


$a^2 + b^2 = c^2$

A proof for Pythagoras

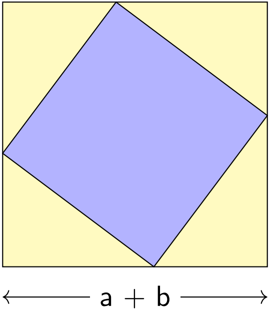


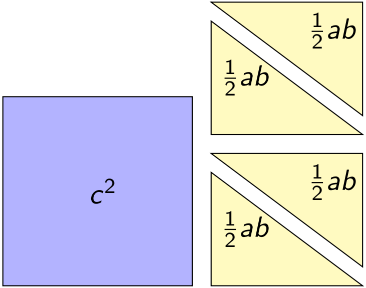
← a + b →



$$(a+b)^2 = c^2 + 4 \times \frac{1}{2}ab$$

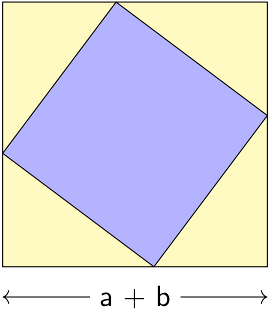
A proof for Pythagoras

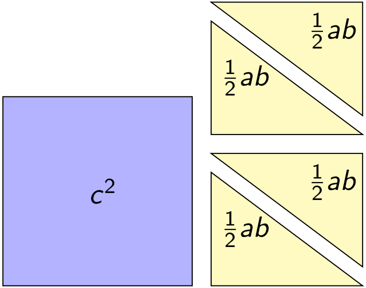




$$(a+b)^2 = c^2 + 4 \times \frac{1}{2}ab$$

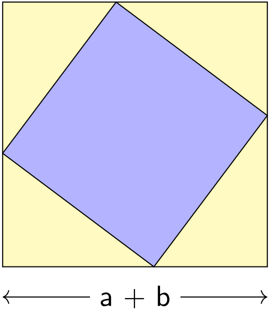
A proof for Pythagoras

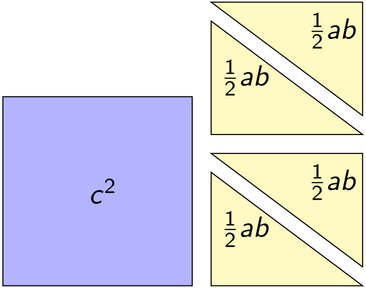




$$(a+b)^2 = c^2 + 2ab$$

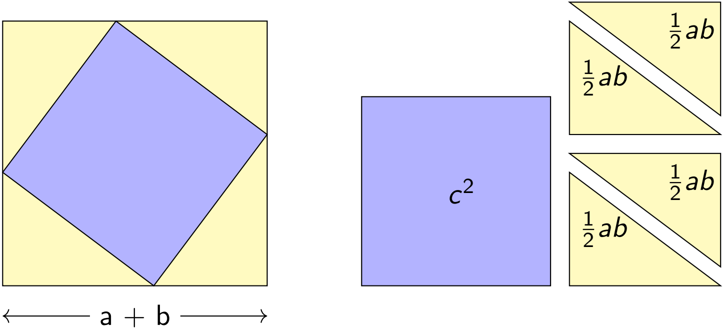
A proof for Pythagoras





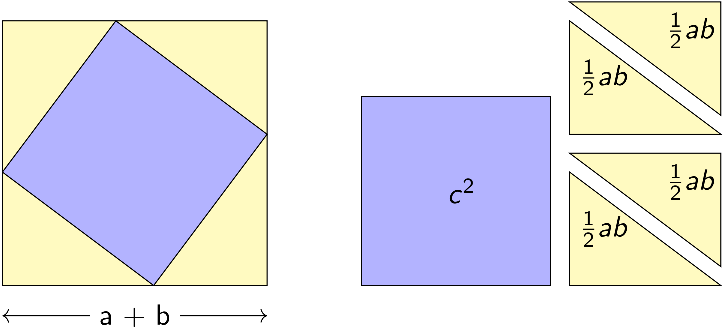
$$(a+b)^2 = c^2 + 2ab$$

A proof for Pythagoras



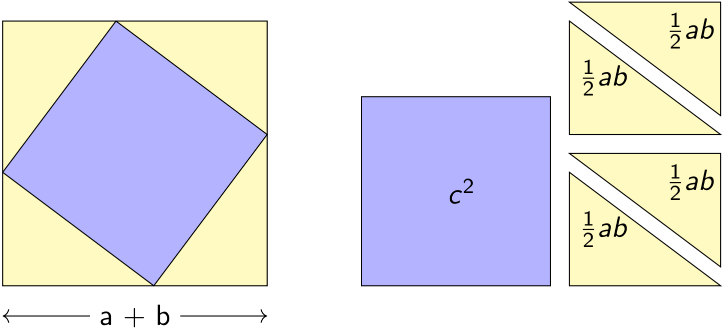
$$a^2 + 2ab + b^2 = c^2 + 2ab$$

A proof for Pythagoras



$$a^2 + 2ab + b^2 = c^2 + 2ab$$

A proof for Pythagoras



$$a^2 + b^2 = c^2$$

What happened?

1. We had a problem in geometry (proof of Pythagoras).
2. We used algebra to solve it and to reason about it.

OK

- But we are programmers.
- Can we also do that with programs?

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (42 == 0) 18 else gcd(42, 18 % 42)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (42 == 0) 18 else gcd(42, 18 % 42)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (false) 18 else gcd(42, 18 % 42)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(42, 18 % 42)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(42, 18 % 42)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(42, 18)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(42, 18)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (18 == 0) 42 else gcd(18, 42 % 18)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (18 == 0) 42 else gcd(18, 42 % 18)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (false) 42 else gcd(18, 42 % 18)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(18, 42 % 18)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(18, 42 % 18)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(18, 6)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(18, 6)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (6 == 0) 18 else gcd(6, 18 % 6)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (6 == 0) 18 else gcd(6, 18 % 6)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (false) 18 else gcd(6, 0)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(6, 0)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = gcd(6, 0)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (0 == 0) 6 else gcd(0, 6 % 0)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (0 == 0) 6 else gcd(0, 6 % 0)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = if (true) 6 else gcd(0, 6 % 0)

Greatest common divisor

A function to calculate the greatest common divisor.

```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = 6

Greatest common divisor

A function to calculate the greatest common divisor.

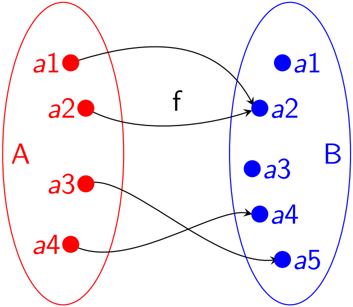
```
fun gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a % b)
```

See if it works:
The greatest common divisor of 18 and 42 is 6.

gcd(18, 42) = 6

This is called: equational reasoning

What is a function



A function $f : A \to B$ maps all elements from set A to B.

The homset B^A is the set of all functions from A to B.

Functional programming vs imperative programming

Functional

- Declarative
  Expression evaluation
- Immutable
- No side effects
- Equational reasoning
- Functions are "first class"

Imperative

- Instructions
  Sequence, selection, iteration, ~~goto~~
- Mutable data and variables
- Side effects

Side effects

Any other effect besides returning a value:

- Heating the CPU
- Execution delays
- Accessing global data
- Modification of input arguments
- Synchronisation
- IO operations
- Getting the time
- Random generation
- Exceptions

Side effects

Any other effect besides returning a value:

- Heating the CPU
- Execution delays
- Accessing global data
- Modification of input arguments
- Synchronisation
- IO operations
- Getting the time
- Random generation
- Exceptions

Side-effects make analysing and reasoning more difficult.

Side effects

Any other effect besides returning a value:

- Heating the CPU
- Execution delays
- Accessing global data
- Modification of input arguments
- Synchronisation
- IO operations
- Getting the time
- Random generation
- Exceptions

Side-effects make analysing and reasoning more difficult.
Pure functional programs are completely useless.

Functional vs object orientation? The Expression problem

Phil Wadler formulated the expression probleem
Given:

- A tree like structure representing and expression.
- One operator is supported: plus
- One operation is supported: evaluate

Then add one operator (minus) and one operation (prettyprint), without changing the existing code

First class functions

- Functions are first class citizens : functions can be used as argument or return value in function calls.
- Higher order functions: functions that accept other functions as argument or return value.
- Referential transparency. Client is not aware if he refers to a value or a function.

High order functions examples

- Java streams and Kotlin collections use predicates
- Dependency inversion, or callback mechanisms
- Abstraction of control structures. (file handling)

```
val mylist = listOf("The", "quick", "brown", "fox")
val lengths = mylist.map { s -> s.length }
val size = mylist.fold(0, {total, value -> total + 1})
```

fold is the mother of all aggregations

Functions as a result: Currying

- Functions with multiple arguments can be curried.
- Uncurried: $f : (a : A, b : B) \to C$
- Curried becomes $f : A \to B \to C$
- Read this as $f : A \to (B \to C)$

```
val add: (Int, Int) -> Int = {x, y -> x + y}
val addCurried: (Int) -> (Int) -> Int = add.curried()
val incr = addCurried(1)
val six = incr(5)
```

Category theory

$$\begin{tikzcd}
A \arrow[loop above, "I_A"] \arrow[r, "f"] \arrow[rr, bend right, "g \cdot f"'] \arrow[rrr, bend right=40, "h \cdot g \cdot f"'] & B \arrow[loop above, "I_B"] \arrow[r, "g"] \arrow[rr, bend right, "h \cdot g"'] & C \arrow[loop above, "I_C"] \arrow[r, "h"] & D.
\end{tikzcd}$$

A category has:

- objects, such as A, B, C and D
- arrows between objects, such as $f : A \rightarrow B$ and $g : B \rightarrow C$
- arrow composition: $g \cdot f : A \rightarrow C$
- associativity: $h \cdot (g \cdot f) = (h \cdot g) \cdot f$
- identity arrows for each object: $f \cdot I_A = I_B \cdot f = f$
- N.B. Identity and composed arrows are typically not drawn.

Category theory usage

Categories can model many concepts, e.g. subsets, logical implications, morphisms, and more.

Here we will only use it as a modelling tool for functions and types:

$$A \xrightarrow{f} B$$

Concrete example with functions and sets

$$
\begin{tikzcd}
\mathit{Unit} \arrow[r, "\text{“}\mathit{BMG}\text{“}"] \arrow[rr, bend right, "3"'] \arrow[rrr, bend right=40, "\mathit{false}"'] & \mathit{String} \arrow[r, "\mathit{length}"] \arrow[rr, bend left, "\mathit{hasEvenLength}"] & \mathit{Integer} \arrow[r, "\mathit{isEven}"] & \mathit{Boolean}
\end{tikzcd}
$$

- Unit is a set with 1 element
- Values can be considered function from unit to another set: "BMG", 3 and false are (constant) functions
- This diagram commutes: every path between objects is equal.
- Thus $\mathit{length} \cdot \mathit{BMG} = 3$ and $\mathit{isEven} \cdot 3 = \mathit{false}$

Product type

$$\begin{tikzcd}
 & Person \arrow[dl, "name"'] \arrow[dr, "address"] & \\
Name & & Address
\end{tikzcd}$$

Kotlin data classes can be used as product types.

We call the functions projections.

Sum type

$$\begin{tikzcd}
Customer \arrow[rr] & & Person \arrow[dl] \arrow[dr] & & Employee \arrow[ll] \\
& Name & & Address &
\end{tikzcd}$$

Reversing the arrows of product type gives a sum type.

A person can be a customer or an employee.

Kotlin implementations is possible with inheritance or delegation.

Function application

$$\begin{tikzcd}
& (f, a) \arrow[rr, "apply"] \arrow[dl, "f"'] \arrow[dr, "a"] & & B \\
B^A & & A &
\end{tikzcd}$$

If we have a function from the homset B^A and an argument we can apply the function to obtain an element of B.

$apply(f, a) = f(a)$

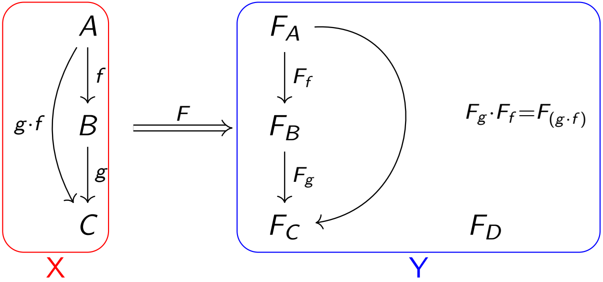
Function composition

$$\begin{tikzcd}
& (f,g) \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[rr, "compose"] & & A \to C \\
A \to B & & B \to C &
\end{tikzcd}$$

We have a functions $A \to B$ and $B \to C$. These can be composed to form a function $A \to C$.

$$compose(f,g) = \{x \to g(f(x))\}$$

Functor



- A functor is an arrow that maps one category to another category
- A functor preserves the structure.
- Functors can be composed, and Identity functors exist.
- Thus this a category of categories.

Functor example: List

$$
\begin{tikzcd}
\mathit{String} \arrow[d, "\mathit{length}"] & & \text{List}<\text{String}> \arrow[d, "\mathit{map}(\mathit{length})"] \\
\mathit{Int} \arrow[d, "\mathit{isEven}"] \arrow[rr, Rightarrow, "\mathit{List}"] & & \text{List}<\text{Int}> \arrow[d, "\mathit{map}(\mathit{isEven})"] \\
\mathit{Boolean} & & \text{List}<\text{Boolean}> \\
X & & Y
\end{tikzcd}
$$

- In Kotlin/Java, Functors are implemented as generics.
- Think of it as a function in the type system.
- List (Stream) and map create the Functor.

EndoFunctor

$$
\begin{tikzcd}
A \arrow[dd, "f"] & & Option{<}A{>} \arrow[dd, "map(f)"] \\
{} \arrow[rr, Rightarrow, "Option"] & & {} \\
Option{<}B{>} & & Option{<}Option{<}B{>}{>}
\end{tikzcd}
$$

Option

- An endofunctor is a functor inside one category.
- What if we map a function $f : A \rightarrow Option{<}B{>}$?
- Can we get $Option{<}Option{<}B{>}{>}$ flattened?

Monad and flatmap

$$
\begin{tikzcd}
A \arrow[d, "f"] \arrow[r, phantom, "\overset{Option}{\Longrightarrow}"] & Option{<}A{>} \arrow[d, "flatmap(f)"] & A \arrow[d, "pure"] \\
Option{<}B{>} & Option{<}B{>} & Option{<}A{>}
\end{tikzcd}
$$

- A monad is a Functor, with a flatmap and pure added.
- Flatmap is the map, that “flattens” the output type.
- Funtion pure lifts a value into the monad.
- Functions typed $f : (A) \rightarrow Option{<}B{>}$ are monadic functions.

Monad category picture

$$\begin{tikzcd} M_A \arrow[loop above, "pure_A"] \arrow[r, "M_f"] \arrow[rr, bend right, "M_{gf}"'] & M_B \arrow[loop above, "pure_B"] \arrow[r, "M_g"] & M_C \arrow[loop above, "pure_C"] \end{tikzcd}$$

- Monadic types and functions turn out to form a similar category as regular functions.
- $pure_T$ is required to be the identity
- so: $M_f = M_f \cdot pure_A = pure_B \cdot M_f$.
- Monadic functions are regular functions with an extra "effect".

Monad application

$$\begin{tikzcd} & (f, a) \arrow[rr, "apply"] \arrow[dl, "f"'] \arrow[dr, "a"] & & M_B \\ A \to M_B & & M_A & \end{tikzcd}$$

$$apply(f : A \to M_B, a : M_A) = a.flatmap(f)$$

Monad composition

$$
\begin{tikzcd}
 & (f,g) \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[rr, "compose"] & & A \to M_C \\
A \to M_B & & B \to M_C &
\end{tikzcd}
$$

$$
\begin{aligned}
&compose(f : A \to M_B, f : B \to M_C) \\
&\qquad = \{a \to f(a).flatmap(g)\} \\
&\qquad = \{a \to f(a).flatmap\{b \to g(b)\}\}
\end{aligned}
$$

Monad composition

$$\begin{tikzcd}
 & (f,g) \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[rr, "compose"] & & A \to M_C \\
A \to M_B & & B \to M_C &
\end{tikzcd}$$

$$\begin{aligned}
&compose(f : A \to M_B, f : B \to M_C) \\
&\quad = \{a \to f(a).flatmap(g)\} \\
&\quad = \{a \to f(a).flatmap\{b \to g(b)\}\}
\end{aligned}$$

So... flatmap is “just“ a wrapper around a monadic function.

Monad composition

$$\begin{tikzcd}
 & (f,g) \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[rr, "compose"] & & A \to M_C \\
A \to M_B & & B \to M_C &
\end{tikzcd}$$

$$\begin{aligned}
&compose(f : A \to M_B, f : B \to M_C) \\
&\quad = \{a \to f(a).flatmap(g)\} \\
&\quad = \{a \to f(a).flatmap\{b \to g(b)\}\}
\end{aligned}$$

So... flatmap is “just“ a wrapper around a monadic function.
“Just“ like proxy objects, servlet filters, error aspects etc

Monad examples

Monads encapsulate side-effects into effects:

- Data structures, containers (Tree, Set, List, Optional).
- Applicatives are a structure between monad and functor.
  e.g. Given a function $f : A \to B \to C$ you can get
  $M_f : M_A \to M_B \to M_C$
  for example to multiply timelines.
- Synchronization (Future, Promise, Actors).
- Exception and error handling (Try monad).
- State management. (reader, writer, state monad).

Monad comprehensions

- Setbuilder
  $range = \{1, 2, 3, ...19, 20\}$
  $\{a, b, c \in range | a < b \cap a^2 + b^2 = c^2\}$
- SQL
  select a.nr as a, b.nr as b, c.nr as c
  from range a, range b, range c
  where $a.nr < b.nr$
  and $a.nr * a.nr + b.nr * b.nr = c.nr * c.nr$
- Kotlin

```
     range.flatMap {
a -> range.filter { b -> a < b }.flatMap {
b -> range.filter { c -> a*a + b*b == c*c }.map {
c -> Triple(a,b,c) }}}
```

Monad remarks

- Ten-thousands of blogs exist on monads.
- There is no good generic way to compose different monads, e.g. Futures of Optionals.
- Frameworks and libraries often avoid the term.
- Many other syntaxes and names are used.
  Bind, $>>=$, $>=>$, pure, return, comprehensions with for or do expressions and even Kotlins ?-operator
- Frameworks: Arrows (Kotlin) ScalaZ and Cats.

Overview

- Equational reasoning.
- Side-effects
- High Order functions
- Functor.
- Monad.