akquinet
WIR. WISSEN. WIE.

Functional Development with Kotlin

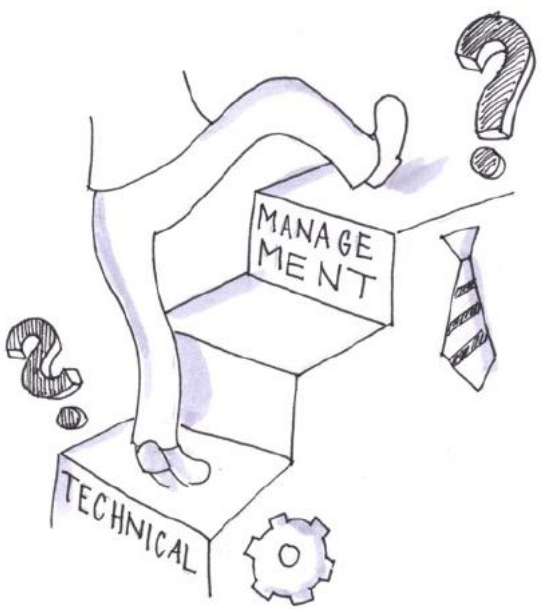
Torsten Fink

torsten.fink@akquinet.de

Some words about me

From developer to manager – with functional touchpoints

- 1989-1996: Study at the university in Erlangen/Nuremberg
  - Student thesis about a functional programming language for parallel applications
- 1996-1998: Research project at the university in Jena
  - usage of Haskell to specify parallel applications
- 1998-2011: Developer, architect, consultant trainer at akquinet
  - Lots of enterprise application stuff
  - Experience: functional programming style increases maintainability and correctness
- 2011- : Executing manager at akquinet tech@spree
  - How to build up development skills in projects without participating in projects?



https://plan.io/blog/from-developer-to-manager-and-back-again/

akquinet
WIR. WISSEN. WIE.

Some words about Kotlin

A worthy successor to Java

akquinet
WIR. WISSEN. WIE.

- Java started 1996, had IMHO lots of design flaws, but was fun and productive.
- In the following years its flaws were patched. The result: lots of special rules, you have to keep in mind (e.g. autoboxing, integration of lambdas, „odd“ behavior of old classes).
- Several competitors for Java came up, e.g. Clojure, Ceylon, Scala, Groovy and Kotlin (V 1.0 in 2016).
- Kotlin
  - is IMHO new, modern, clean, pragmatic, and continously improved,
  - compiles to the JVM, to JavaScript, and to binary code,
  - Is developed and used by JetBrains for their own IDE-products.

Some words about this tutorial

What are the key characteristics of FP?

What about your experience level in FP?

- I do not know anything but I am curious.
- I know some basics but do not practice it.
- I use FP regularly for development.
- I am a senior FP citizen.
- Nothing fits to me.

akquinet
WIR. WISSEN. WIE.

Functional Programming from the academical perspective

… a.k.a. Wikipedia

A programming paradigm

- .. functional programming is a **programming paradigm**
  - where programs are constructed by applying and composing functions.
- It is a **declarative** programming paradigm in which
  - function definitions are trees of expressions that **map values to other values**,
  - rather than a sequence of imperative statements which update the running state of the program.

```
val sumWithDiscountFP =
   { article1: Article, article2: Article,
     discount: Double ->
       val sumArticles =
          article1.price + article2.price
       val discountMultiplier =
          1.0 - discount
       discountMultiplier *
            sumArticles
   }
```

```
fun sumWithDiscountImp(
   article1: Article, article2: Article,
   discount: Double
): Double {
   var result = 0.0
   result += article1.price
   result += article2.price
   result *= (1.0 - discount)
   return result
}
```

With functions as 1st class citizens

- … **functions** are treated as **first-class citizens**, meaning that
  - they can be **bound to names** (including local identifiers),
  - passed as **arguments**, and
  - **returned** from other functions, just as any other data type can.
- This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

```
typealias ProbandEvaluator = (Proband) -> Double

val probands = loadProbands()
val evaluator = createEvaluator()
val sortedProbands = sortProbands(probands, evaluator)
```

```
val sortProbands: (Set<Proband>, ProbandEvaluator) ->
        List<Proband> =
 { probands: Set<Proband>, evaluator: ProbandEvaluator ->

   val evaluatedProbands = probands.map()
     { proband: Proband -> Pair(evaluator(proband), proband) }
   evaluatedProbands
          .sortedBy { pair -> pair.first }
          .map { pair -> pair.second }
}
```

And they are pure!

akquinet
WIR. WISSEN. WIE.

- Functional programming is sometimes treated as synonymous with purely functional programming, a subset of functional programming which treats all functions as **deterministic mathematical functions**, or pure functions.
- When a **pure function is called with some given arguments, it will always return the same result**, and cannot be affected by any mutable state or other side effects. …
- Proponents of purely functional programming claim that by restricting side effects, programs can have fewer bugs, be easier to debug and test, and be more suited to formal verification.

```
fun doSeveralThings(input:Int) {
    val a = doSomething(input)
    doSomethingElse()
    val b = doSomething(input)
    // a == b
}
```

Boils down to: No side effects, only immutable state

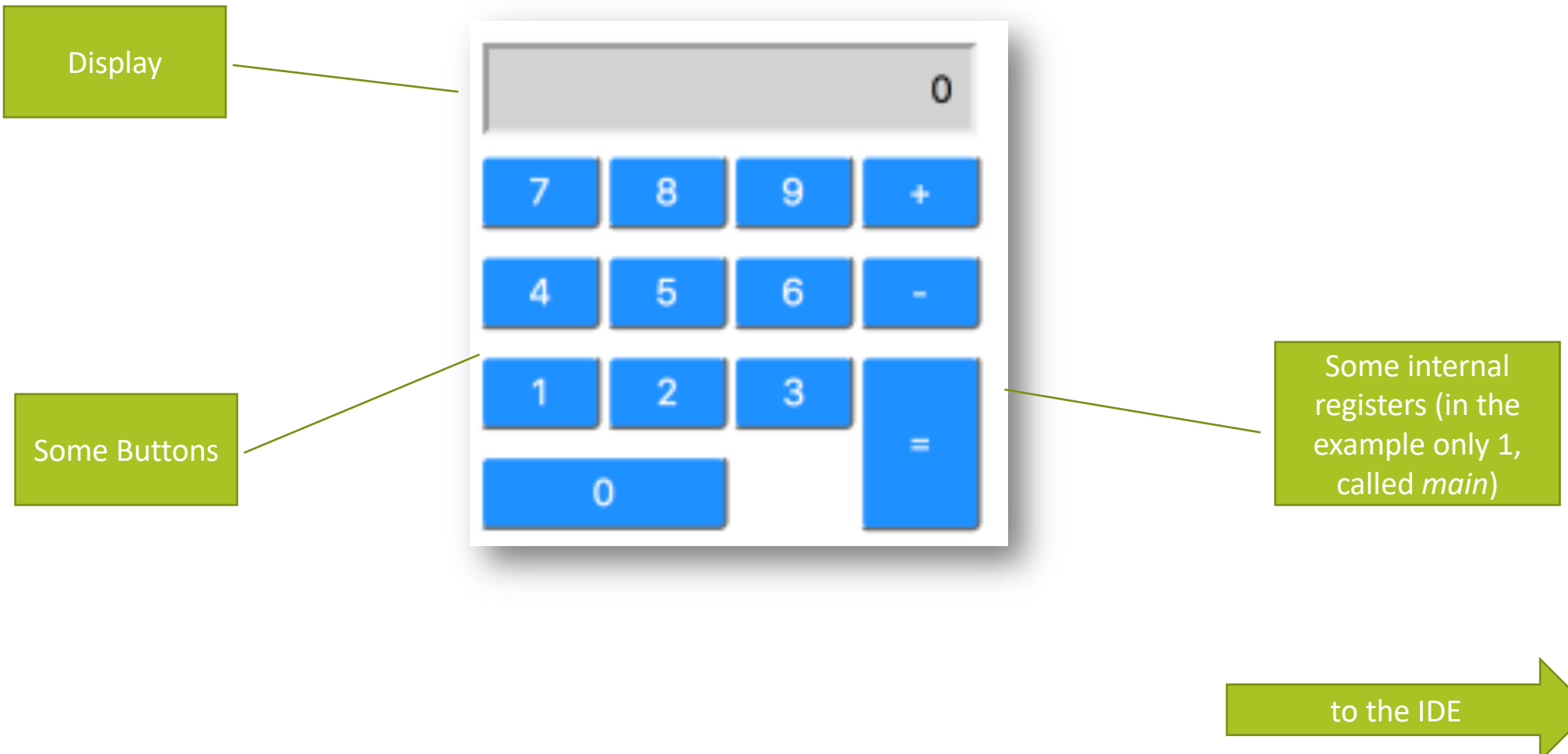
akquinet
WIR. WISSEN. WIE.

Too much input, this is a tutorial so let us dive into code

How to participate…

- You need:
  - a computer with some up-to-date browser
  - an internet connection.
- First question: Who would like to play with code?
- For anyone who woul like to join:
  - Navigate to:
    https://github.com/tnfink/kotlinfptutorial/tree/forParticipants/src/main/kotlin
    https://tinyurl.com/4pn2np2s
  - Show: CalculatorDemoDemo.kt
  - Copy‘n‘paste into: https://play.kotlinlang.org/
  - Run the code.

The domain: a calculator



Back to the conceptual view

An Example for: Why mutable state is bad!

akquinet
WIR. WISSEN. WIE.

- Task:
  set up a pair of conferences for developers and give discount for attendees of both conferences

```
// domain model

data class PriceM(
   var amount: Double
) {}

data class ConferenceM(
   var name: String,
   var price: PriceM
) {}
```

```
// computation

val developersEpisode1 = ConferenceM(
   "Developers - Episode 1", PriceM(200.0)
)
val developersEpisode2 = ConferenceM(
   "Developers - Episode 2", developersEpisode1.price
)
// later in the code ... give a discount
developersEpisode2.price.amount *= 0.5

val ticketFee =
   developersEpisode1.price.amount +
         developersEpisode2.price.amount
```

Voting:
Value of ticketFee
300, 200, 400 ?

Result is 200
Intended was 300

The cause:
Impurity of the attribute lookup

Doing the same in the pure way ☺

akquinet
WIR. WISSEN. WIE.

```
data class PriceIM(
    val amount: Double
) {}

data class ConferenceIM(
    val name: String,
    val price: PriceIM
) {}
```

```
val developersEpisode1 = ConferenceIM(
    "Developers - Episode 1",  PriceIM(200.0) )
val developersEpisode2 =  developersEpisode1.copy(
        name = "Developers - Episode 2")

// later in the code ... give a discount
val discountedPrice = developersEpisode2
    .price.copy(developersEpisode2.price.amount / 2)

val developersEpisode2Discounted =  developersEpisode2.copy(
        price = discountedPrice  )

val ticketFee =
    developersEpisode1.price.amount +
          developersEpisode2Discounted.price.amount
```

Create new values with copy() instead of reuse and modify

ticketFee = 300

Make changes safe by creating new entities

akquinet
WIR. WISSEN. WIE.

Functional Programming now from the Developer Perspective

Working on Data

The Triumvirate: Map, Fold, Pipe

- **Map**
  apply a function on all elements of a collection
  `[a,b,c].map(f) = [f(a), f(b), f(c)]`
- **Fold**
  combine all elements using an initial value and a function
  `[a,b,c].fold(0,f) = f(f(f(0,a),b),c)`
  (Btw, this a a left fold.)
- **Compose / Pipe**
  (f `compose` g)(x) = f(g(x))
- **Let us dive into some examples: `WorkingOnData.kt` …**

to the IDE

Is this functional code?

- Task:
  Compute the sum of all numbers in a list

```
var sum = 0
list.forEach
  { n -> sum += n }
```

Side effect on sum

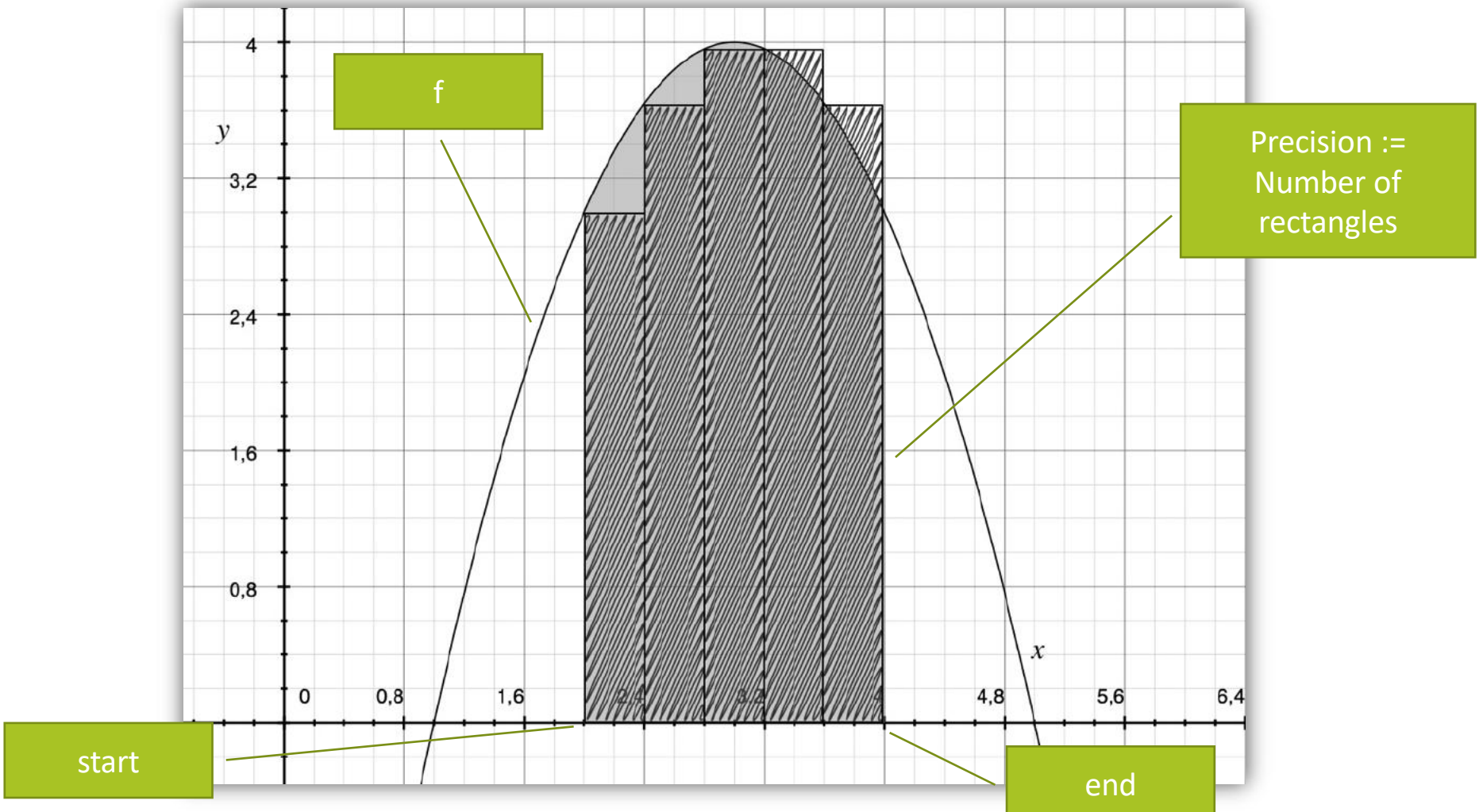
Separation of reusable algorithmic structure from specific domain code

```
val sum =
  list.fold(0)
    { acc, n -> acc + n }
```

akquinet
WIR. WISSEN. WIE.

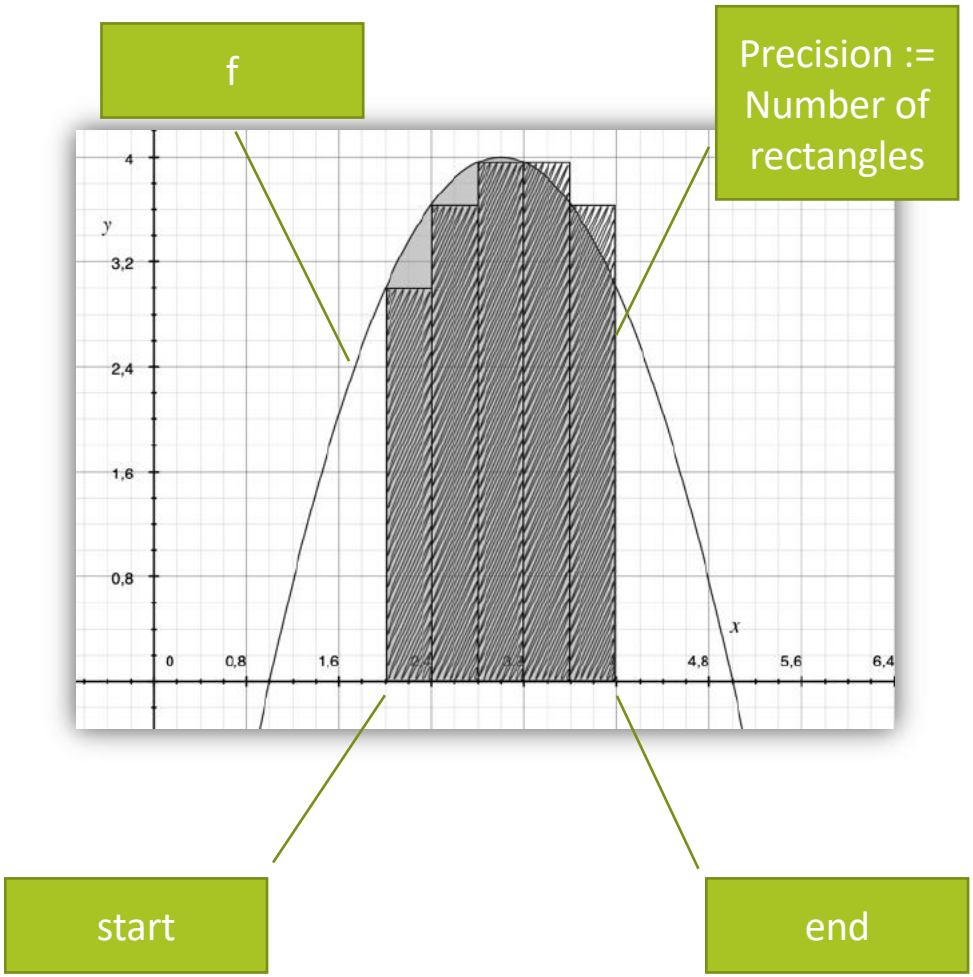
Implementing Algorithms

Numerical integration



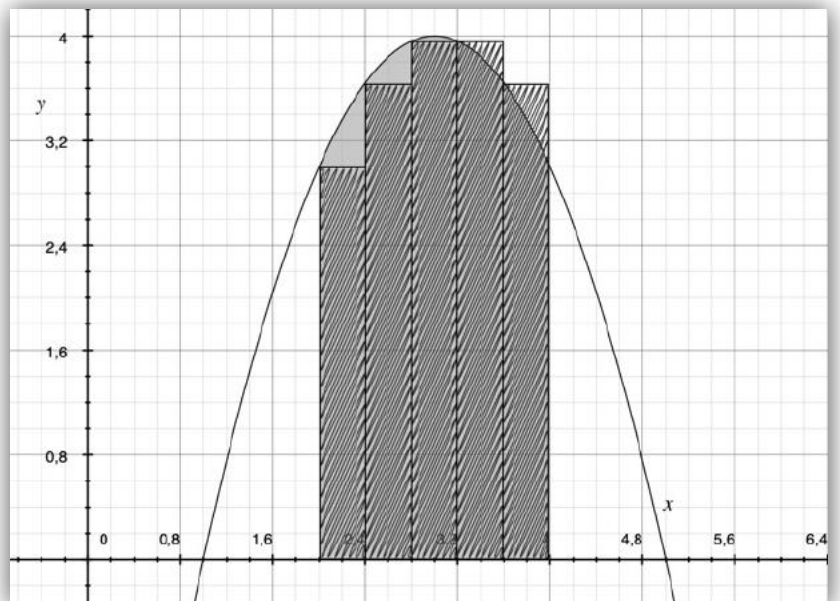
Typical imperative code

```
fun integrateImperative(
    start: Double, end: Double, precision: Long,
    f: (Double) -> Double
): Double {
    val step = (end - start) / precision
    var result = 0.0
    var x = start
    for (i in 0 until precision) {
        result += f(x) * step
        x += step
    }
    return result
}
```



And now the same algorithm with functional programming

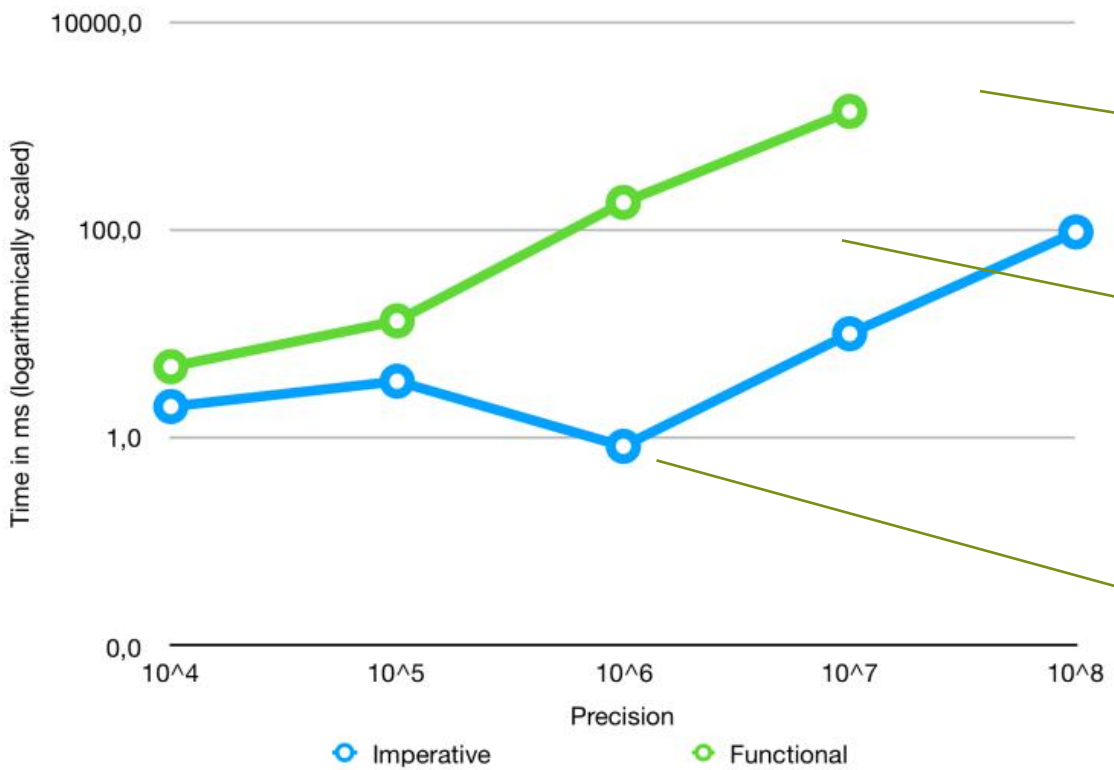
`IntegrationDemo.kt`



to the IDE

What about performance?

- Some measurements for larger numbers on my (old) MacBook pro 2015



Out of Memory ☹

The imperative version is a factor of 140 faster

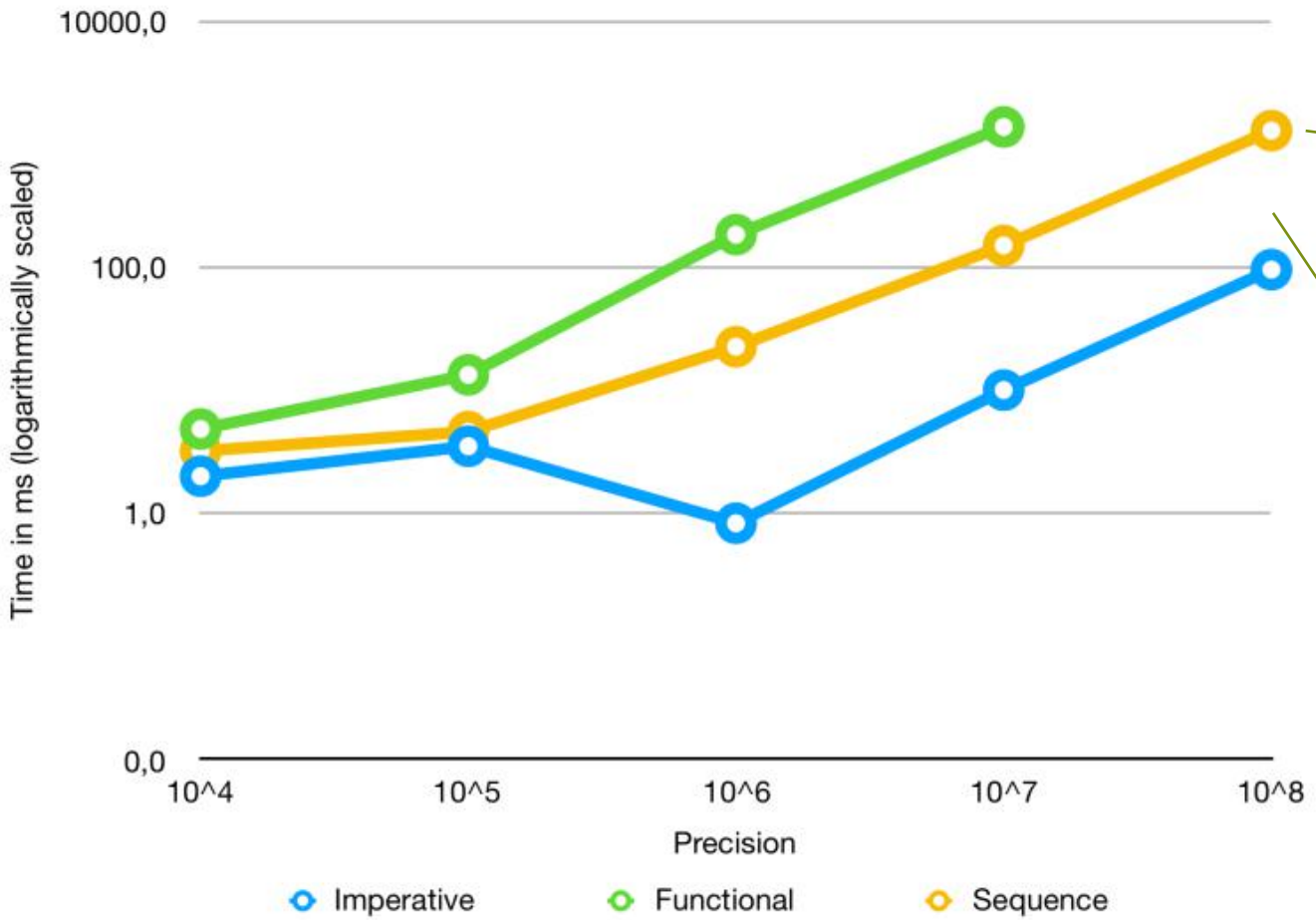
Some loop optimization or JIT-compilation kicks in

Sequences to your rescue

akquinet
WIR. WISSEN. WIE.

- A sequence is a list
  - generated on demand,
  - and potentially infinite.
- ```
  val oddNumbers =
      generateSequence(1) { it + 2 }
  ```
- Simple switch to sequences in our example:
  ```
  val xCoordinates = (0 until precision).asSequence()
      .map { index -> start + index * step }
  ```
- … let‘s see, if it helps
  ```

And, the results are:



No memory problem ☺

Still a factor of 15 slower ☹

Just for the sake of completenes, a look at Haskell

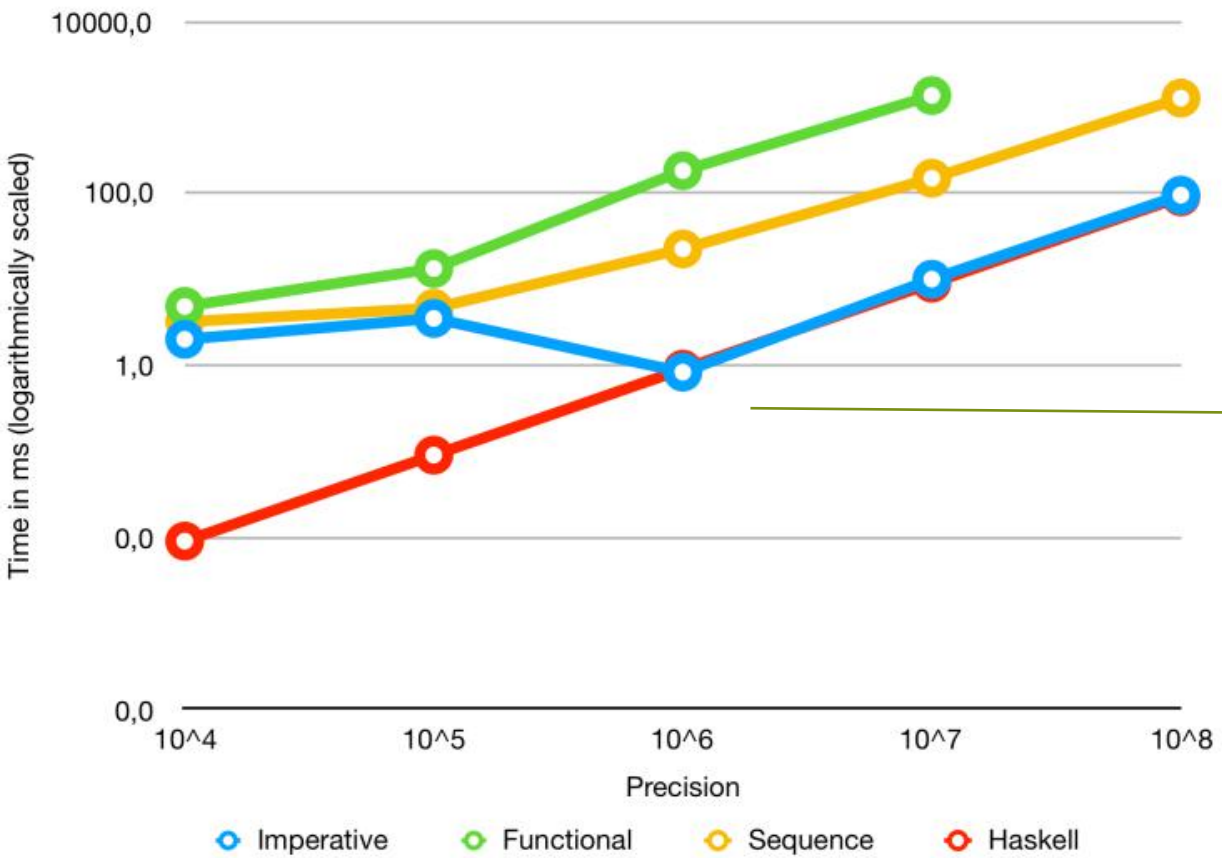
akquinet
WIR. WISSEN. WIE.

- Haskell is a pure functional programming language.
- Our example:

```
integrate :: Double -> Double -> Int -> (Double -> Double) -> Double

integrate start end precision function = sum allRectangles
  where
    step = (end - start) / (fromIntegral precision)
    xCoordinates = map (\i -> start + (fromIntegral i) * step)
                       [ 0 .. (precision-1)]
    allRectangles = map (\x -> (function x) * step) xCoordinates
```

And, the final results are:



Old Haskell is pretty fast.

- Conclusion:
  if you need performance, use imperative code hidden behind an FP interface

What did I leave out?

Topics for some other tutorials

akquinet
WIR. WISSEN. WIE.

- How to develop unbounded loops.
  The idea: Sequences und takeUntil, for more information see:
  https://blog.akquinet.de/2019/09/17/unbounded-functional-loops-in-kotlin/
- How to handle side effects and non deterministics behavior, such as external input/output and random numbers, in an ideomatic way for Kotlin.
- More sophisticated examples to use FP in the real world. For example validation:
  https://funktionale-programmierung.de/2023/01/19/kotlin-validation.html
- All the sophisticated FP stuff, such as Applicative Functors, Monads etc.
  - If you are interested in these and willing to learn a lot, checkout Arrow https://arrow-kt.io

Sum it all up

The key points from my perspective

- Functional programming (FP) provides IMHO
  - Less errors (immutable data)
  - Better maintainability (functions as 1st class citizens).
- Kotlin
  - is a modern (aka cool) programming language,
  - that provides nearly all features for FP
  - but is still an imperative language in its core, leading to performance problems with FP.