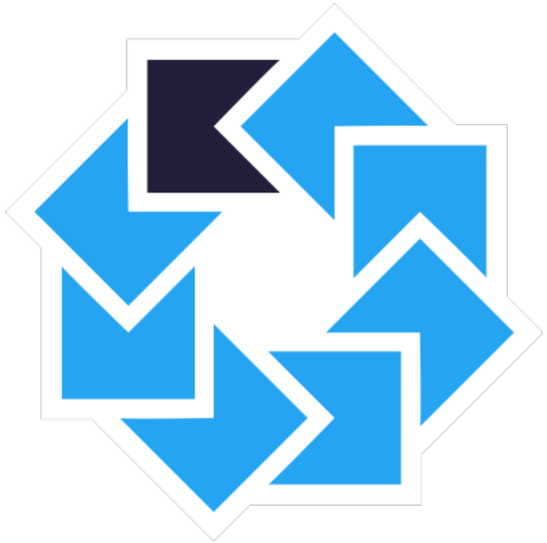

#3:
Functional programming
elements
juno
fitbit
SPACE
JET
BRAINS

Anton Rutkevich

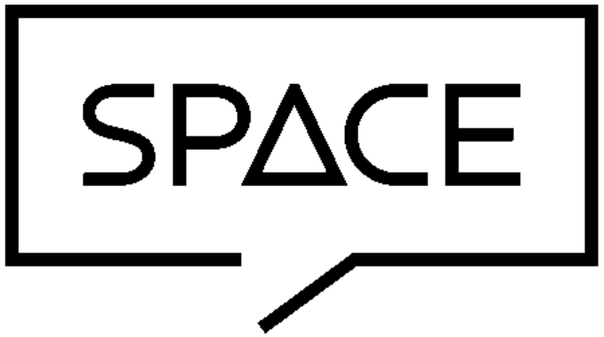
Software Engineer @ Juno

Anton Rutkevich

roottony@gmail.com




fitbit

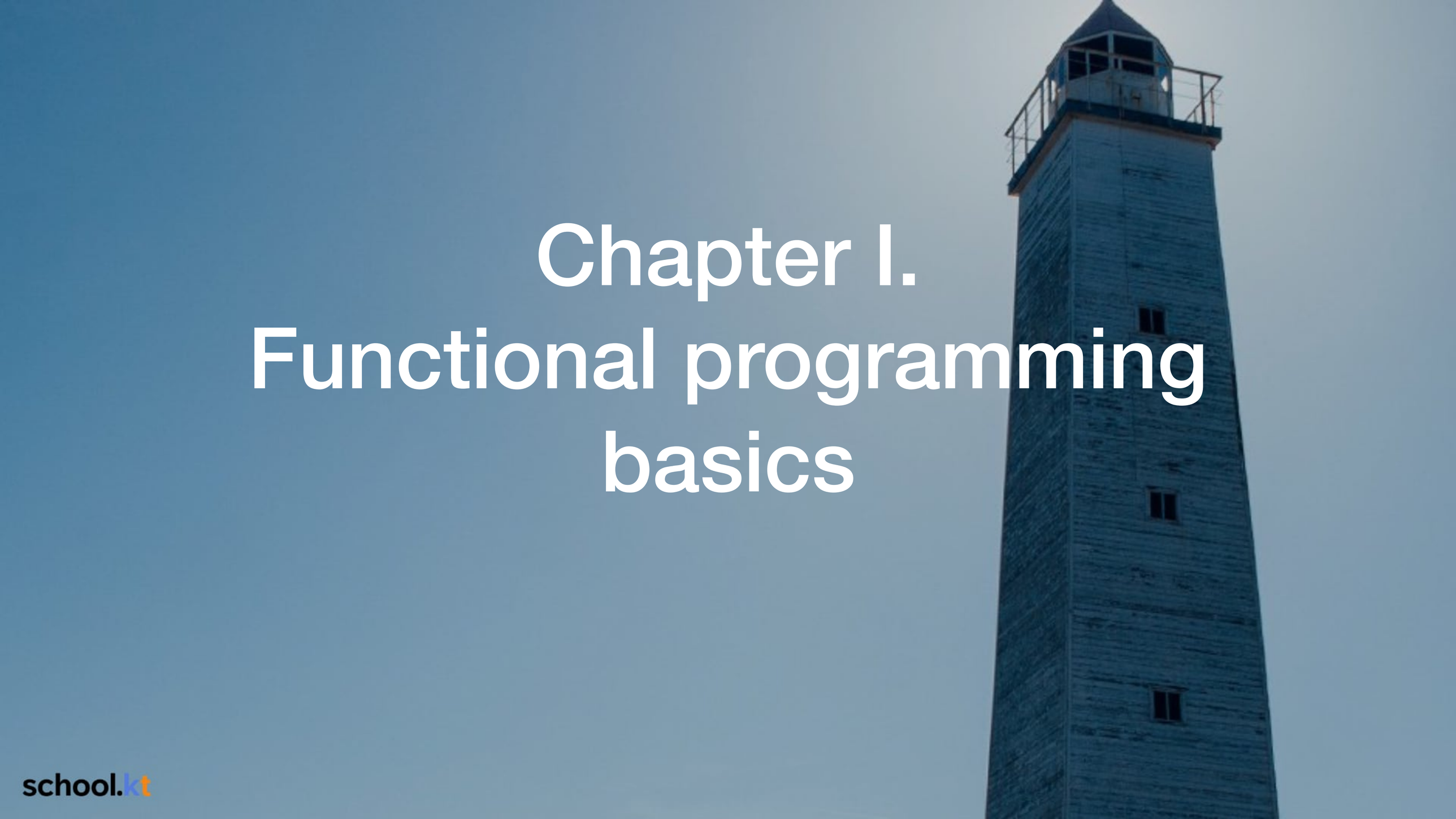

SPACE


juno

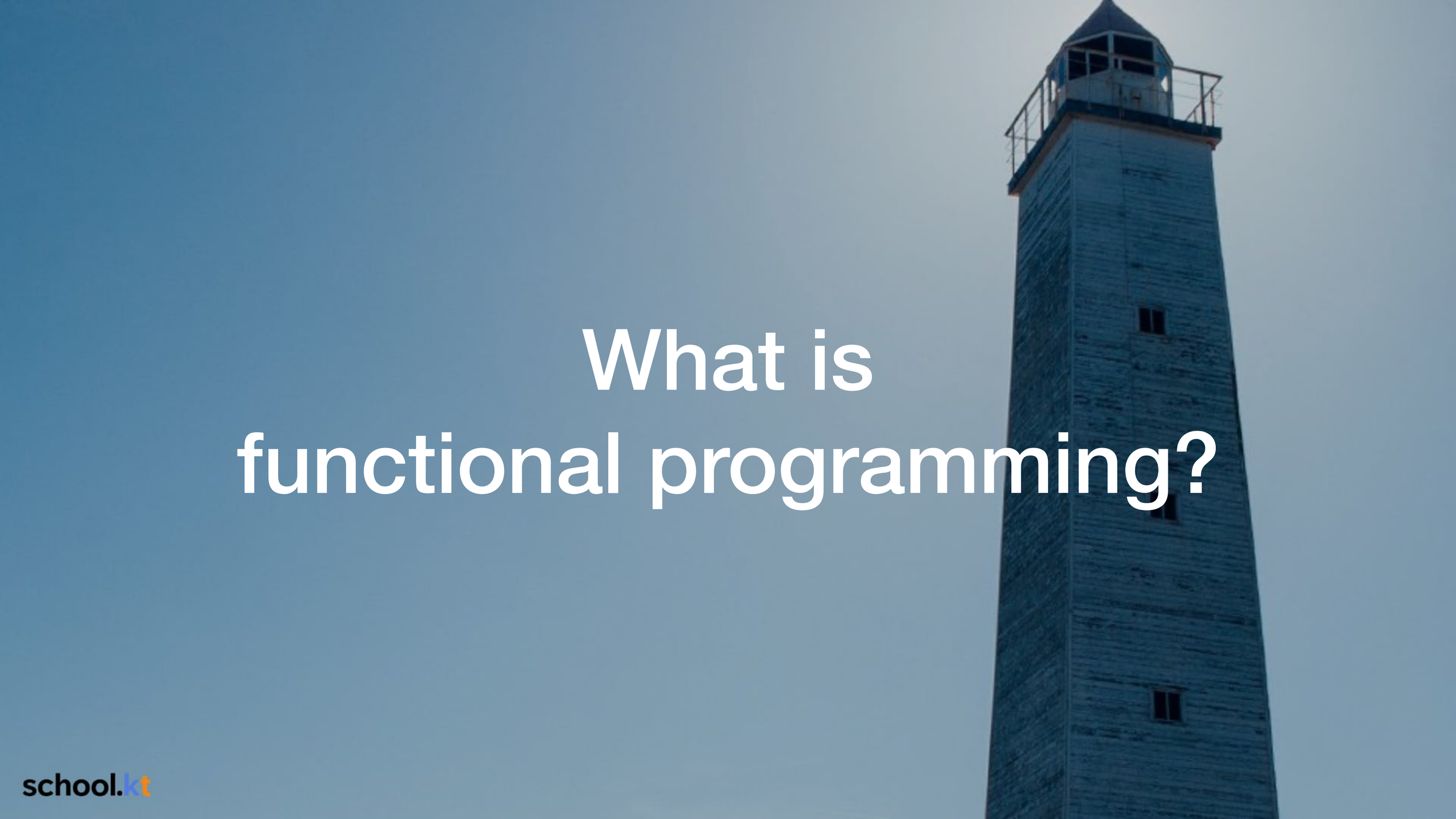

JET
BRAINS

Agenda

- Functional programming basics
- Functional programming concepts in Kotlin



Chapter I.
Functional programming basics


What is
functional programming?

“Functional programming is a programming paradigm that treats computation as the evaluation of **mathematical functions** and **avoids changing-state** and **mutable data**.”

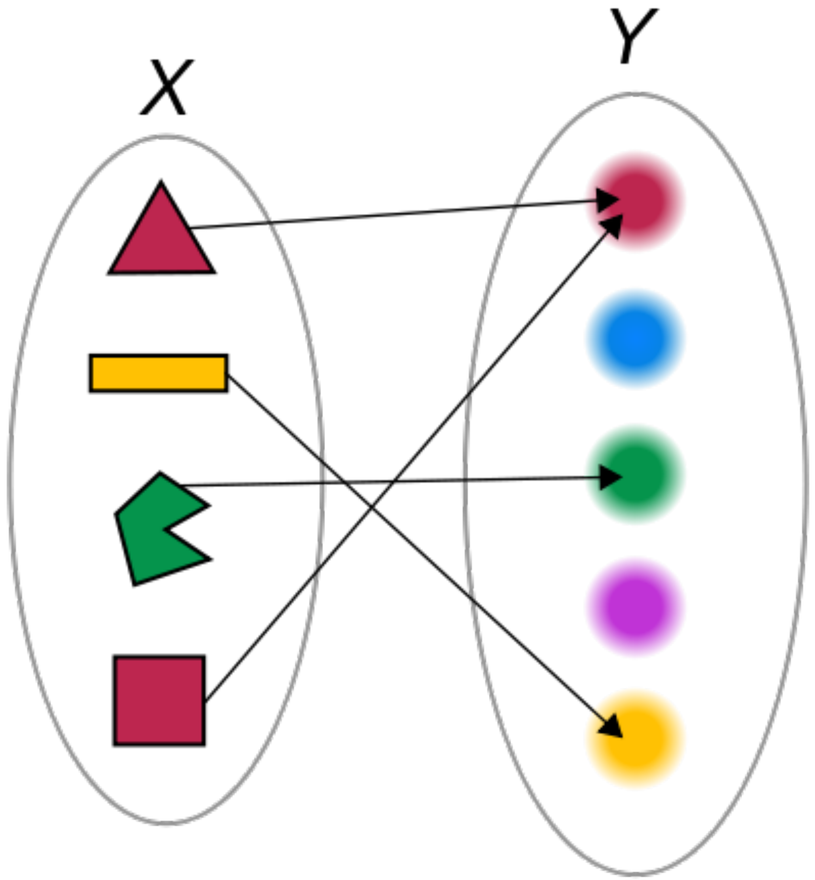
– Wikipedia

… mathematical functions …

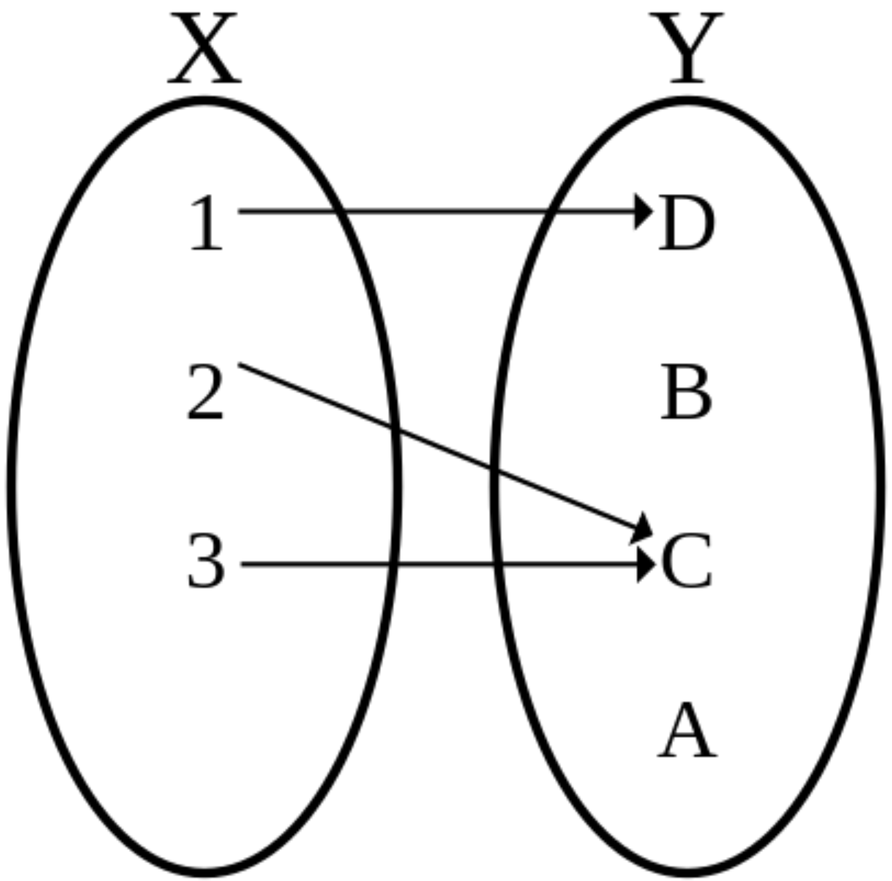
"A (mathematical) **function** is a process or a **relation** that associates each element ***x* of a set *X***, the *domain* of the function, **to a single** element ***y* of another set *Y*** (possibly the same set), the *codomain* of the function."

– *Wikipedia*

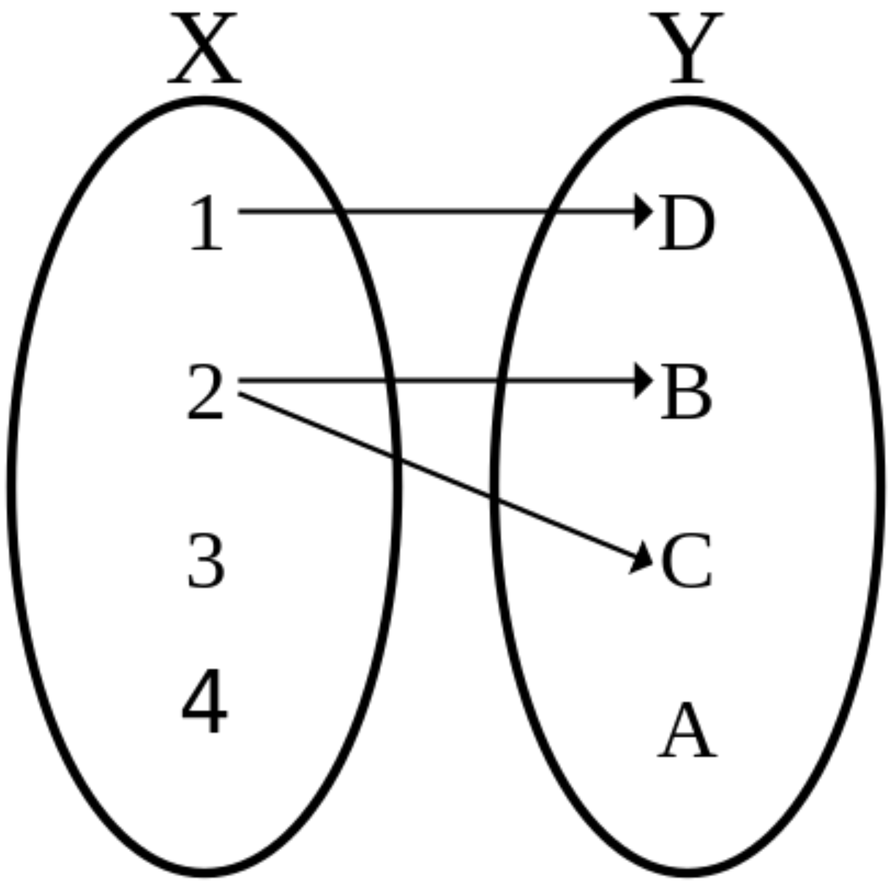
Mathematical function



Mathematical function ?



Mathematical function ?



… **avoids changing-state** …

Changing state

```
var numOfCalls = 0

fun sum(a: Int, b: Int): Int {
    numOfCalls++
    return a + b
}
```

Changing state

```
var numOfCalls = 0

fun sum(a: Int, b: Int): Int {
    numOfCalls++
    return a + b
}

fun sum2(a: Int, b: Int): Int {
    MySingleton.incNumOfCalls()
    return a + b
}
```

… **avoids** … **mutable data** …

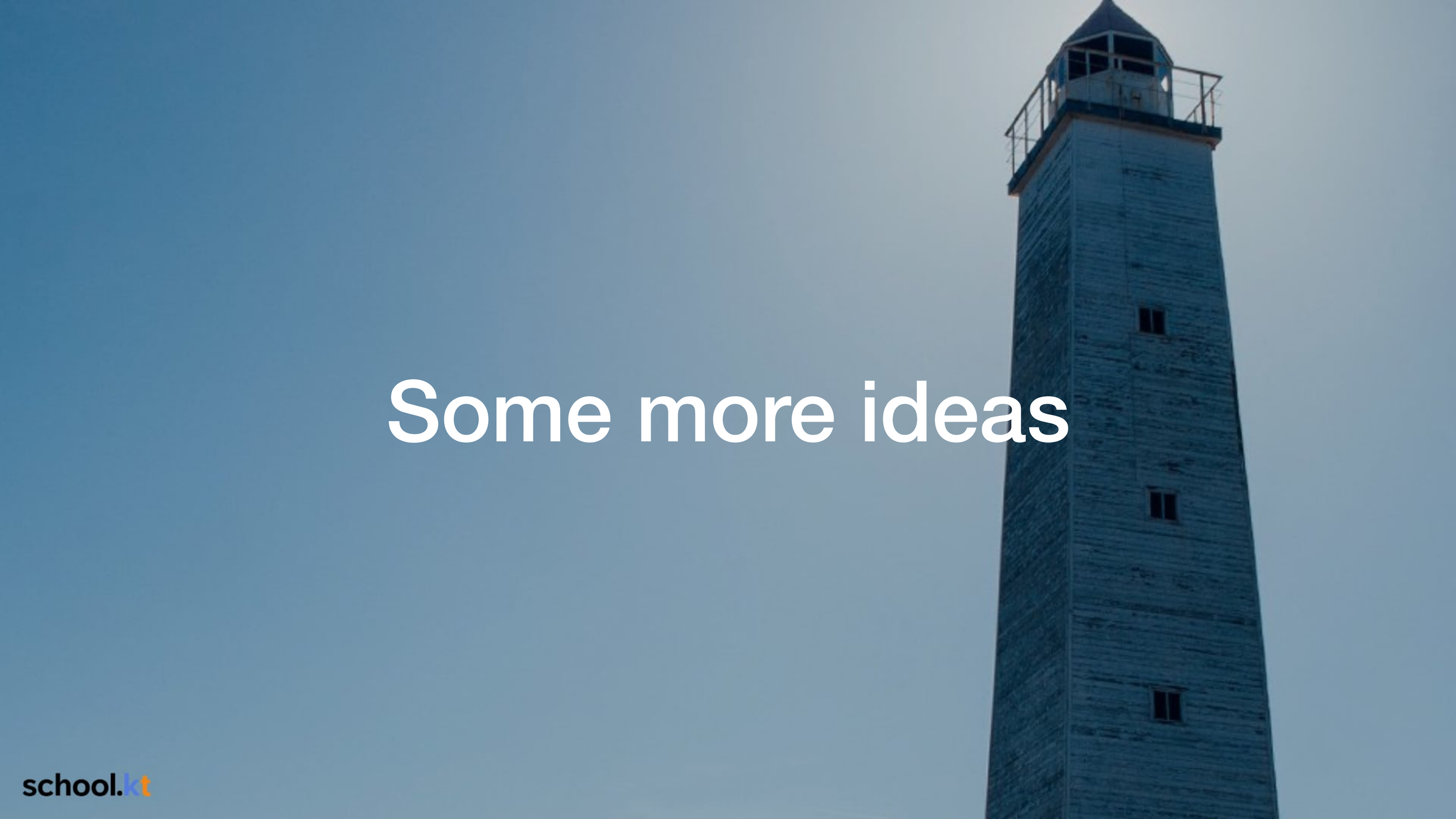
Mutable data

```
data class A(var num: Int, var str: String)

data class B(val num: Int, val str: String)
```

“Functional programming is a programming paradigm that treats computation as the evaluation of **mathematical functions** and **avoids changing-state** and **mutable data**.”

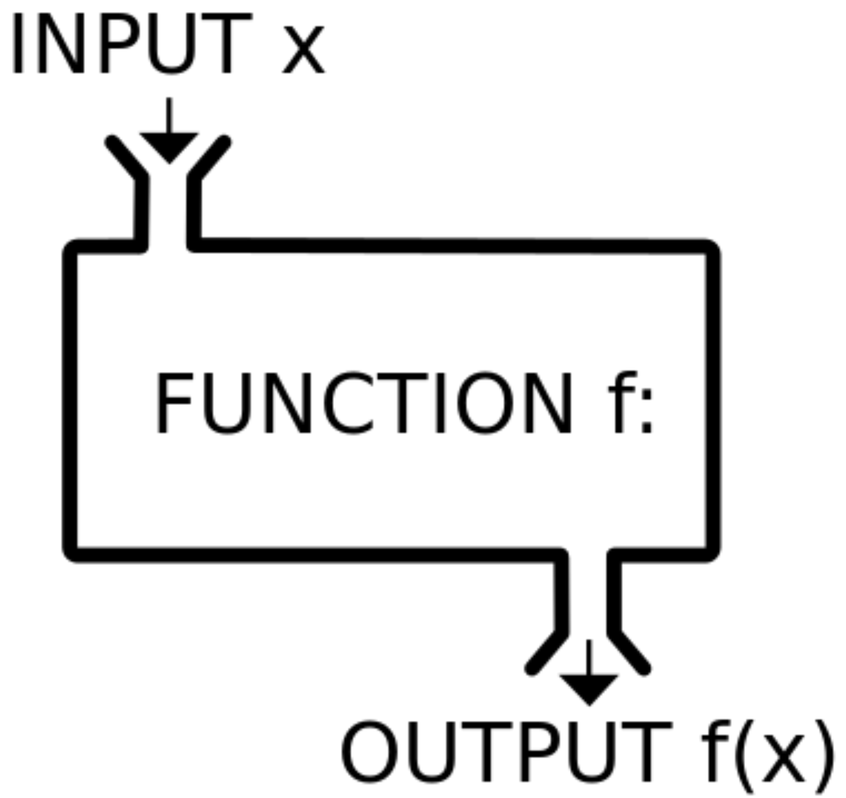
– Wikipedia


Some more ideas

Pure function

- Its return value is the same for the same arguments
  - No variation with local static variables, non-local variables, mutable reference arguments or I/O streams
- Its evaluation has no side effects
  - No mutation of local static variables, non-local variables, mutable reference arguments or I/O streams

Pure function



Pure function benefits

- Simple
- Memoization possibilities
- Parallelization possibilities
- Easy to test

Total vs partial function

```
fun toString(a: Int): String = a.toString()
```

Total vs partial function

```
fun toString(a: Int): String = a.toString()

fun div(a: Int, b: Int): Int = a / b
```

Total vs partial function

```
fun toString(a: Int): String = a.toString()

fun div(a: Int, b: Int): Int = a / b

fun loop(a: Int, b: Int): Int {
    while (true) {

    }
}
```

Total vs partial function

```
fun toString(a: Int): String = a.toString()

fun div(a: Int, b: Int): Int = a / b

fun loop(a: Int, b: Int): Int {
    while (true) {

    }
}

fun exit() {
    System.exit(1)
}
```

Type systems

```
fun div(a: Int, b: Int): Int = a / b
```

Type systems

```
data class Num(val value: Int)
```

Type systems

```
data class Num(val value: Int)

class Denom private constructor(val value: Int) {
  companion object {
    fun create(value: Int): Denom? = if (value != 0) {
      Denom(value)
    } else {
      null
    }
  }
}
```

Type systems

```
fun div(a: Int, b: Int): Int = a / b

fun div2(num: Num, denom: Denom): Int
        = num.value / denom.value
```


Imperative vs declarative

What's going on here?

```
val sourceList = listOf("a", "bb", "ccc", "ddd", "ee", "fff")
val filteredList = mutableListOf<String>()
for (item in sourceList) {
    if (item.length > 1) {
        filteredList.add(item)
    }
}

val itemsByLength = mutableMapOf<Int, List<String>>()
for (item in filteredList) {
    val currentItemsOfLength =
        itemsByLength[item.length] ?: emptyList()
    itemsByLength[item.length] = currentItemsOfLength + item
}

for (entry in itemsByLength) {
    println("${entry.key}: ${entry.value}")
}
```

What's going on here?

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
    .filter { it.length > 1 }
    .groupBy(String::length)
    .forEach { length, values ->
        println("$length: $values")
    }
```

“**Imperative** programming is a programming paradigm that uses statements that change a program's state.”

– *Wikipedia*

“**Declarative** programming is
a programming paradigm that
expresses the logic of
a computation without describing
its control flow.”

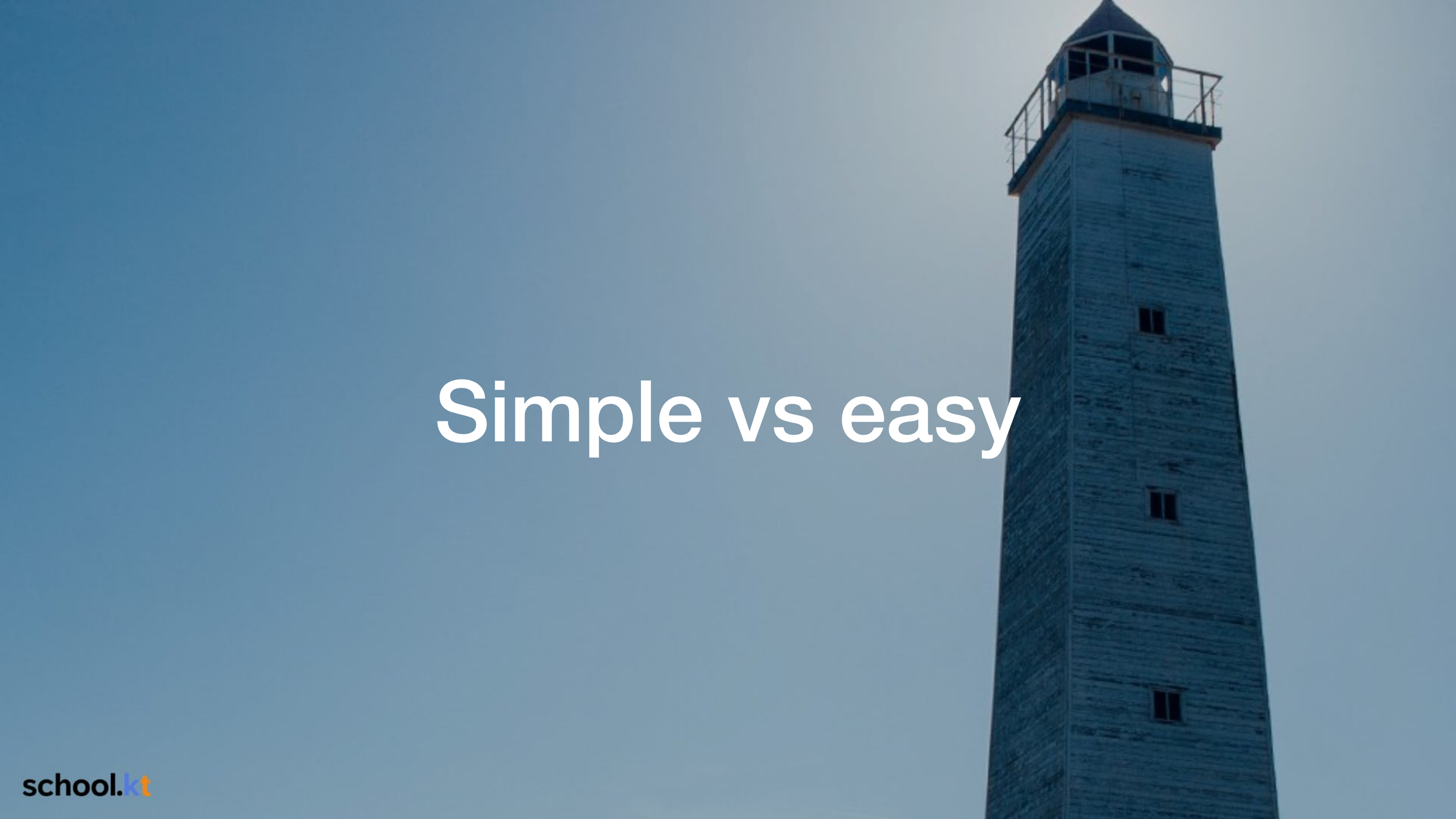
– *Wikipedia*

Imperative

- Machine languages
- Procedural programming
- OOP
- …

Declarative

- Functional programming
- DSL
- Markup languages
- SQL
- Regexp
- Zero-programming builders
- …


Simple vs easy

“Simple made easy” by Rich Hickey

- Simple
  - Easy to read
  - Low cognitive load
  - Easy to maintain
  - Hard to break
  - , but often hard to write

“Simple made easy” by Rich Hickey

- Easy
  - Fast and easy to write
  - Hard to maintain
  - High cognitive load
  - Complex


Why FP?

Why FP? Simpler

- More declarative
- No side effects
- No state
- Flow is clear due to pure & total functions

Why FP? Safer

- Higher compile-time guarantees
  - total functions
  - type systems

Why FP? Optimizations

- Simple parallelism
- Memoizations
- Reordering optimizations
- Laziness
- …

Why not FP? Hard

- Hard (not easy)

Why not FP? Slower

- Heavier CPU & memory usage
  - Immutability
  - Data structures
  - Recursion


FP languages

“**Lambda calculus** is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.”

– *Wikipedia*

What is ‘lambda’ in programming?

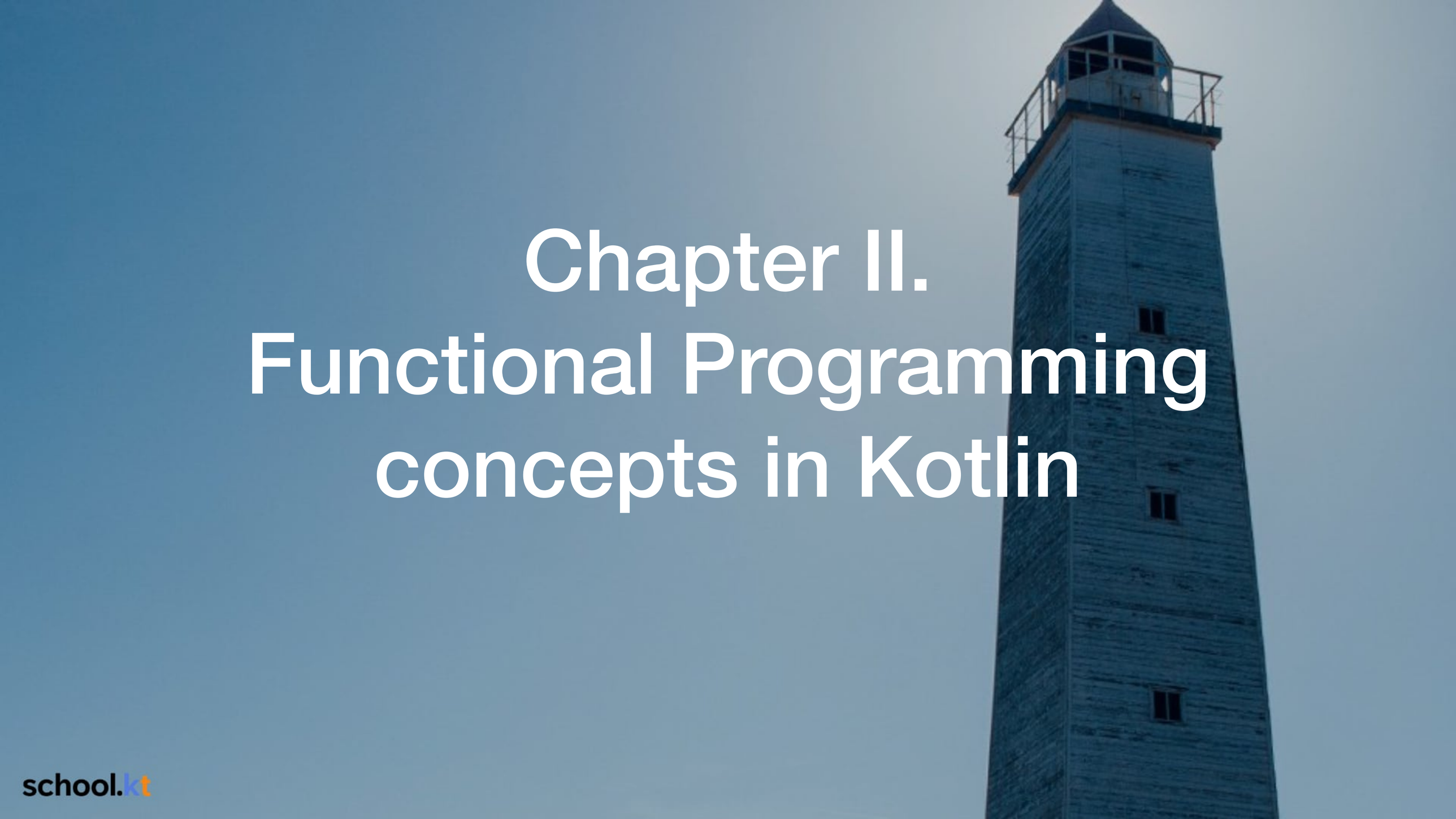
- First-class function
- Has no name
- Can have parameters
- Can/must return a value
- Can access variables from outer scope

FP languages

- Lisp family (Common Lisp, Closure, Scheme, …)
- Wolfram Language
- Erlang
- OCaml
- Haskell
- F#
- XSLT
- R
- …

“Kotlin has both object-oriented and functional constructs. You can use it in both OO and FP styles, or mix elements of the two. With first-class support for features such as higher-order functions, function types and lambdas, …”

– Kotlin FAQ



Chapter II.
Functional Programming concepts in Kotlin

FP concepts in Kotlin

- Functional types (first class functions)
- Higher order functions
- Immutability
- Lazy evaluation
- Algebraic data types
- Recursion
- …


FP concepts in Kotlin.
Function types and literals

Types and literals

```
val someString: String = "Hello, Kotlin"
```

Types and literals

```
val someString: String = "Hello, Kotlin"
```

Types and literals

```
val someString: String = "Hello, Kotlin"
```

Types and literals

```
val someString: String = "Hello, Kotlin"

val someInt: Int = 12345
```

Types and literals

```
val someString: String = "Hello, Kotlin"

val someInt: Int = 12345

val someChar: Char = 'c'
```

Function types

```
val fun1: () -> Unit
```

Function types

```
val fun1: () -> Unit
```

Function types

```
val fun2: (String) -> Unit
```

Function types

```
val fun3: (String) -> Int
val fun4: (String) -> Int?
```

Function types

```
val fun3: (String) -> Int
val fun4: (String) -> Int?

val fun5: ((String) -> Int)?
```

Function types

```
val fun6: ((String) -> Int) -> Long
```

Function types

```
val fun7: Int.() -> Int
```

Function types as supertypes

```
class Square : (Int) -> Int {
    override fun invoke(a: Int): Int = a * a
}

Square()(2)
```

Function literals. Lambdas

```
val square = { a: Int -> a * a }
square(2)

val square2: (Int) -> Int = { it * it }
square2(2)

val square3: Int.() -> Int = { this * this }
2.square3()
```

Function literals. Lambdas

```
val square = { a: Int -> a * a }
square(2)

val square2: (Int) -> Int = { it * it }
square2(2)

val square3: Int.() -> Int = { this * this }
2.square3()
```

Function literals. Lambdas

```
val square = { a: Int -> a * a }
square(2)

val square2: (Int) -> Int = { it * it }
square2(2)

val square3: Int.() -> Int = { this * this }
2.square3()
```

Function literals. Lambdas

```
val square = { a: Int -> a * a }
square(2)

val square2: (Int) -> Int = { it * it }
square2(2)

val square3: Int.() -> Int = { this * this }
2.square3()
```

Function literals. Anonymous functions

```
val square = fun(a: Int): Int = a * a

square(2)
```

Function literals. Anonymous functions

```
val square = fun(a: Int): Int = a * a

square(2)
```

FP concepts in Kotlin. Higher order functions

Higher order functions

```
fun repeat(times: Int, action: (Int) -> Unit) {
  for (index in 0 until times) {
    action(index)
  }
}
```

Higher order functions

```
fun repeat(times: Int, action: (Int) -> Unit) {
  for (index in 0 until times) {
    action(index)
  }
}

repeat(5, { println("Iteration $it") })
```

Higher order functions

```
fun repeat(times: Int, action: (Int) -> Unit) {
  for (index in 0 until times) {
    action(index)
  }
}

repeat(5, { println("Iteration $it") })

repeat(5) { println("Iteration $it") }
```

Higher order extension functions

```
fun <T, R> T.let(block: (T) -> R): R {
  return block(this)
}

fun nullableString(): String? = "test"
```

Higher order extension functions

```
fun <T, R> T.let(block: (T) -> R): R {
  return block(this)
}

fun nullableString(): String? = "test"

nullableString()?.let { "$it length is ${it.length}" }
```

Some built in ones

- apply
- run
- let
- also
- with
- …

Some built in ones

```
fun <T : Closeable?, R> T.use(block: (T) -> R): R {
    try {
        return block(this)
    } finally {
        try {
            this?.close()
        } catch (closeException: Throwable) {
            // ignored here
        }
    }
}
```

Some built in ones

```
fun process(line: String) {
  …
}

fun useExample() {
  File("/path/to/file").reader().use {
    var line = readLine()
    while (line != null) {
      process(line)
      line = readLine()
    }
  }
}
```

Collection pipelines

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
  .filter { it.length > 1 }
  .groupBy { it.length }
  .forEach { length, values ->
    println("$length: $values")
  }
```

Collection pipelines

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
  .filter { it.length > 1 }
  .groupBy { it.length }
  .forEach { length, values ->
    println("$length: $values")
  }
```

Collection pipelines

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
  .filter { it.length > 1 }
  .groupBy(String::length)
  .forEach { length, values ->
    println("$length: $values")
  }
```

Collection pipelines

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
  .filter { it.length > 1 }
  .groupBy(::myLength)
  .forEach { length, values ->
    println("$length: $values")
  }


fun myLength(string: String): Int = string.length
```

Rx pipelines

```
Observable.just("a", "bb", "ccc", "ddd", "ee", "fff")
  .filter { it.length > 2 }
  .map { "Hello, $it" }
  .subscribe { message ->
    println(message)
  }
```

Rx pipelines

```
Observable.just("a", "bb", "ccc", "ddd", "ee", "fff")
    .filter { it.length > 2 }
    .map { "Hello, $it" }
    .subscribe { message ->
        println(message)
    }

// Output
// Hello, ccc
// Hello, ddd
// Hello, fff
```

Rx pipelines

```
Observable.just("a", "bb", "ccc", "ddd", "ee", "fff")
   .filter { it.length > 2 }
   .map { "Hello, $it" }
   .subscribe(::println)
```

Function inlining

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
  .forEach {
    println("${it.length} - $it")
  }
```

Function inlining

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
  .forEach {
    println("${it.length} - $it")
  }
```

```
inline fun <T> Iterable<T>.forEach(action: (T) -> Unit)
{
  for (element in this) action(element)
}
```

Function inlining

```
listOf("a", "bb", "ccc", "ddd", "ee", "fff")
  .forEach {
    println("${it.length} - $it")
  }


// Converted to

val list = listOf("a", "bb", "ccc", "ddd", "ee", "fff")
for (element in list) {
    println("${element.length} - $element")
}
```

Function inlining 2

```
const val DEBUG = true

inline fun logInDebug(message: () -> String) {
  if (DEBUG) {
    println(message())
  }
}

fun main() {
  logInDebug { "Will only be printed in debug" }
}
```

FP concepts in Kotlin.
Lazy evaluation

Lazy evaluation. If statement

```
fun computeSomething(): Boolean {
    println("Computing…")
    return true
}

val a: Boolean = true
```

Lazy evaluation. If statement

```
fun computeSomething(): Boolean {
    println("Computing…")
    return true
}

val a: Boolean = true

if (a || computeSomething()) {
    println("Inside if")
}

// Output ?
```

Lazy evaluation. If statement

```
fun computeSomething(): Boolean {
    println("Computing…")
    return true
}

val a: Boolean = true

if (a || computeSomething()) {
    println("Inside if")
}

// Output
// Inside if
```

Lazy evaluation

```
fun printFirstOf(
    text1: String?,
    text2: String?
) {
    (text1 ?: text2)?.let(::println)
}
```

Lazy evaluation

```
fun printFirstOf(
    text1: String?,
    text2: String?
) {
    (text1 ?: text2)?.let(::println)
}

fun arg1(): String? = …
fun arg2(): String? = …
```

Lazy evaluation

```
fun printFirstOf(
    text1: String?,
    text2: String?
) {
    (text1 ?: text2)?.let(::println)
}

fun arg1(): String? = …
fun arg2(): String? = …


printFirstOf(arg1(), arg2())
```

Lazy evaluation

```
fun printFirstOf(
    text1: () -> String?,
    text2: () -> String?
) {
    (text1() ?: text2())?.let(::println)
}
```

Lazy evaluation

```
fun printFirstOf(
    text1: () -> String?,
    text2: () -> String?
) {
    (text1() ?: text2())?.let(::println)
}
```

```
printFirstOf({ arg1() }, { arg2() })
```

FP concepts in Kotlin. Immutability

Immutability. Data classes

```
data class A(var num: Int, var str: String)
```

Immutability. Data classes

```
data class A(var num: Int, var str: String)



val a = A(1, "a")

a.num = 5
a.str = "other str"
```

Immutability. Data classes

```
data class B(val num: Int, val str: String)
```

Immutability. Data classes

```
data class B(val num: Int, val str: String)



val b = B(2, "b")

// Compile time errors!
b.num = 3
b.str = "blabla"
```

Immutability. Collections

```
val mutable = mutableListOf("a", "b", "c")
mutable.add("d")

println("mutable: $mutable")

// Output ?
```

Immutability. Collections

```
val mutable = mutableListOf("a", "b", "c")
mutable.add("d")

println("mutable: $mutable")

// Output
// mutable: [a, b, c, d]
```

Immutability. Collections

```
val immutable = listOf("a", "b", "c")
val immutable2 = immutable.plus("d")

println("immutable: $immutable")
println("immutable2: $immutable2”)

// Output?
```

Immutability. Collections

```
val immutable = listOf("a", "b", "c")
val immutable2 = immutable.plus("d")

println("immutable: $immutable")
println("immutable2: $immutable2”)

// Output
// immutable: [a, b, c]
// immutable2: [a, b, c, d]
```

Immutability. Composite types

```
data class A(var num: Int, var str: String)
data class C(val num: Int, val a: A)


val c = C(num = 1, a = A(2, "some string"))

println("c.a.str: $c")
c.a.str = "other string"
println("c.a.str: $c")

// Output ?
```

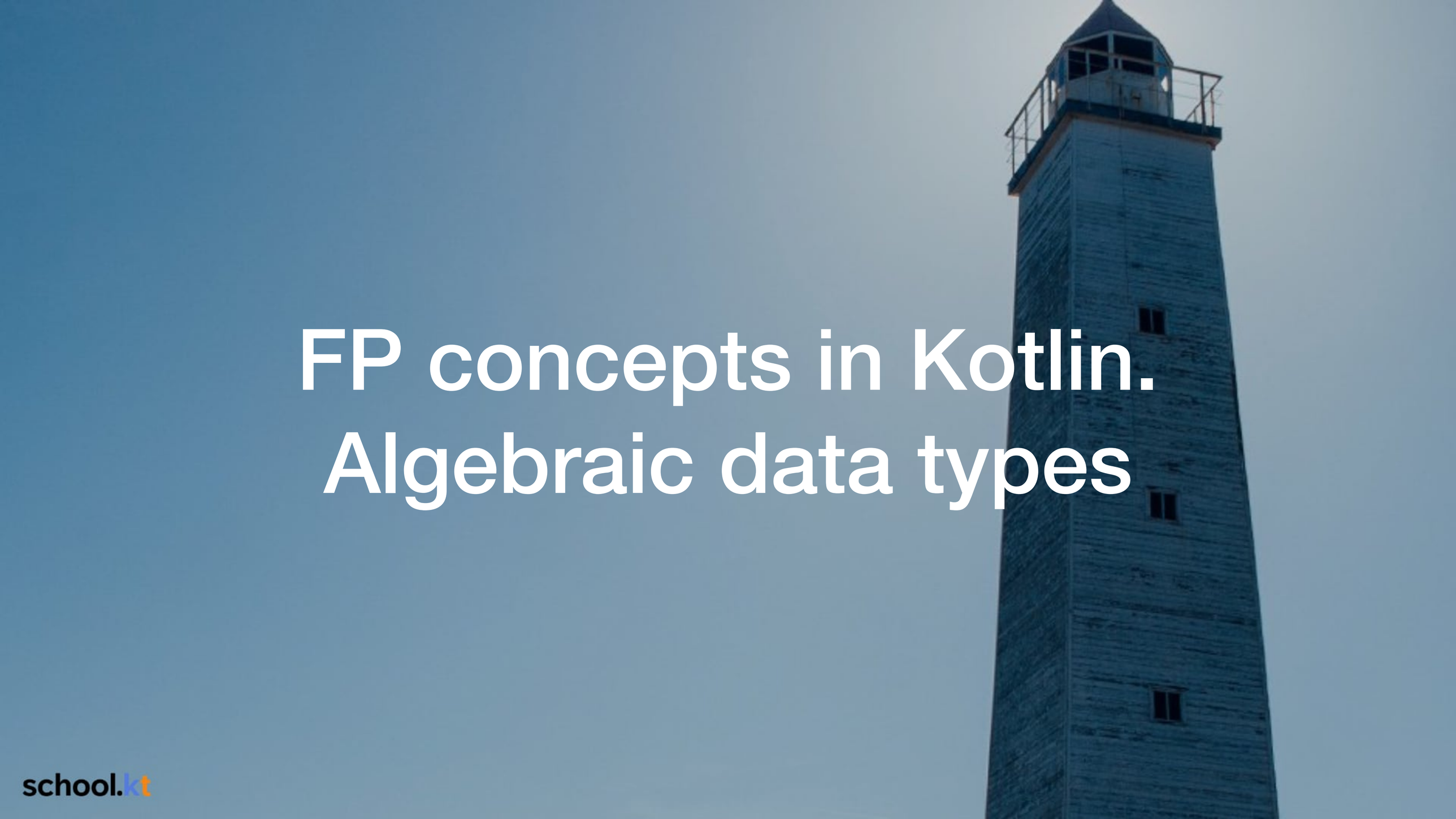
Immutability. Composite types

```
data class A(var num: Int, var str: String)
data class C(val num: Int, val a: A)


val c = C(num = 1, a = A(2, "some string"))

println("c.a.str: $c")
c.a.str = "other string"
println("c.a.str: $c")

// Output
// c.a.str: some string
// c.a.str: other string
```



FP concepts in Kotlin. Algebraic data types

Algebraic data types. Why?

```
data class Result(
  val data: String?,
  val error: Throwable?
)

fun makeNetworkCall(): Result =
    Result("some valid data", null)
```

Algebraic data types. Why?

```
data class Result(
  val data: String?,
  val error: Throwable?
)

fun makeNetworkCall(): Result =
    Result("some valid data", null)

val result = makeNetworkCall()
result.data
result.error
```

Algebraic data types. How?

```
sealed class Result {
    data class Success(val data: String): Result()
    data class Failure(val error: Throwable): Result()
}

fun makeNetworkCall(): Result =
    Result.Success("some valid data")

val result = makeNetworkCall()
result.data
result.error
```

Algebraic data types. How?

```
sealed class Result {
    data class Success(val data: String): Result()
    data class Failure(val error: Throwable): Result()
}

fun makeNetworkCall(): Result =
    Result.Success("some valid data")

val result = makeNetworkCall()
when (result) {
    is Result.Success -> result.data
    is Result.Failure -> result.error
}
```

FP concepts in Kotlin. Recursion

In lambda calculus, there is no

- Arrays
- Numbers
- Arithmetics
- Loops

They are all defined recursively

Recursion

```
fun factorial(n: Int): Int {
  return if (n < 0) {
    1
  } else {
    n * factorial(n - 1)
  }
}

factorial(10000)
```

Recursion. Tailrec

```
tailrec fun factorialWithTailrec(
    n: Long,
    accumulator: Long
): Long {
  val current = n * accumulator
  return if (n <= 1) {
    current
  } else {
    factorialWithTailrec(n - 1, current)
  }
}

factorialWithTailrec(10000, 0L)
```

Recursion. Why tailrec?

```
println("No tailrec: " + measureNanoTime {
  factorial(10000)
})

println("Tailrec: " + measureNanoTime {
  factorialWithTailrec(10000, 0L)
})

// Output ?
```

Recursion. Why tailrec?

```
println("No tailrec: " + measureNanoTime {
  factorial(10000)
})

println("Tailrec: " + measureNanoTime {
  factorialWithTailrec(10000, 0L)
})

// Output
// No tailrec: 804447
// Tailrec: 305630
```

Recursion. Why tailrec?

```java
// Java
public static final int factorial(int n) {
    return n < 0 ? 1 : n * factorial(n - 1);
}
```

Recursion. Why tailrec?

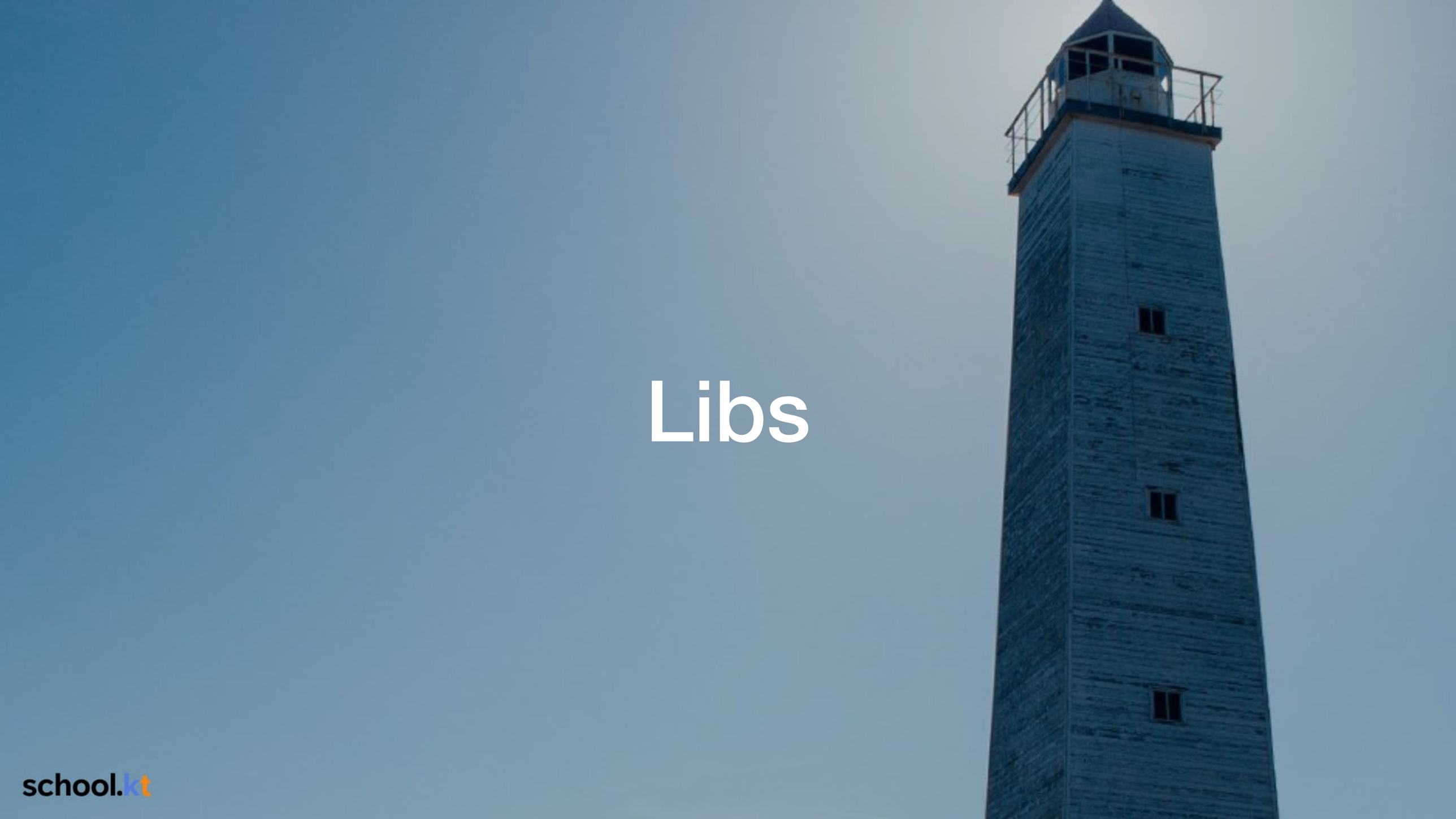
```
// Java
public static final long factorialWithTailrec(
        long n, long accumulator
) {
    while(true) {
        long current = n * accumulator;
        if (n <= 1L) {
            return current;
        }
        long var10000 = n - 1L;
        accumulator = current;
        n = var10000;
    }
}
```


What's under the hood

Lambdas

- Anonymous functions
- Optimized if outer context is not used
  - to static functions
  - to non-capturing functions
- Inlined if possible

Libs



FP related libraries

- Arrow
- Koptional

Thanks!!!

https://www.infoq.com/presentations/Simple-Made-Easy

https://bit.ly/2HcGtdC