Maia Grotepass
Luno

Four of a Kotlin Kind

Imperative, declarative, object oriented, functional

Disclaimer

Engineering perspective - Opinions are my own

Basic -> Pascal -> Machine Language

C -> C++ -> Bash -> Python

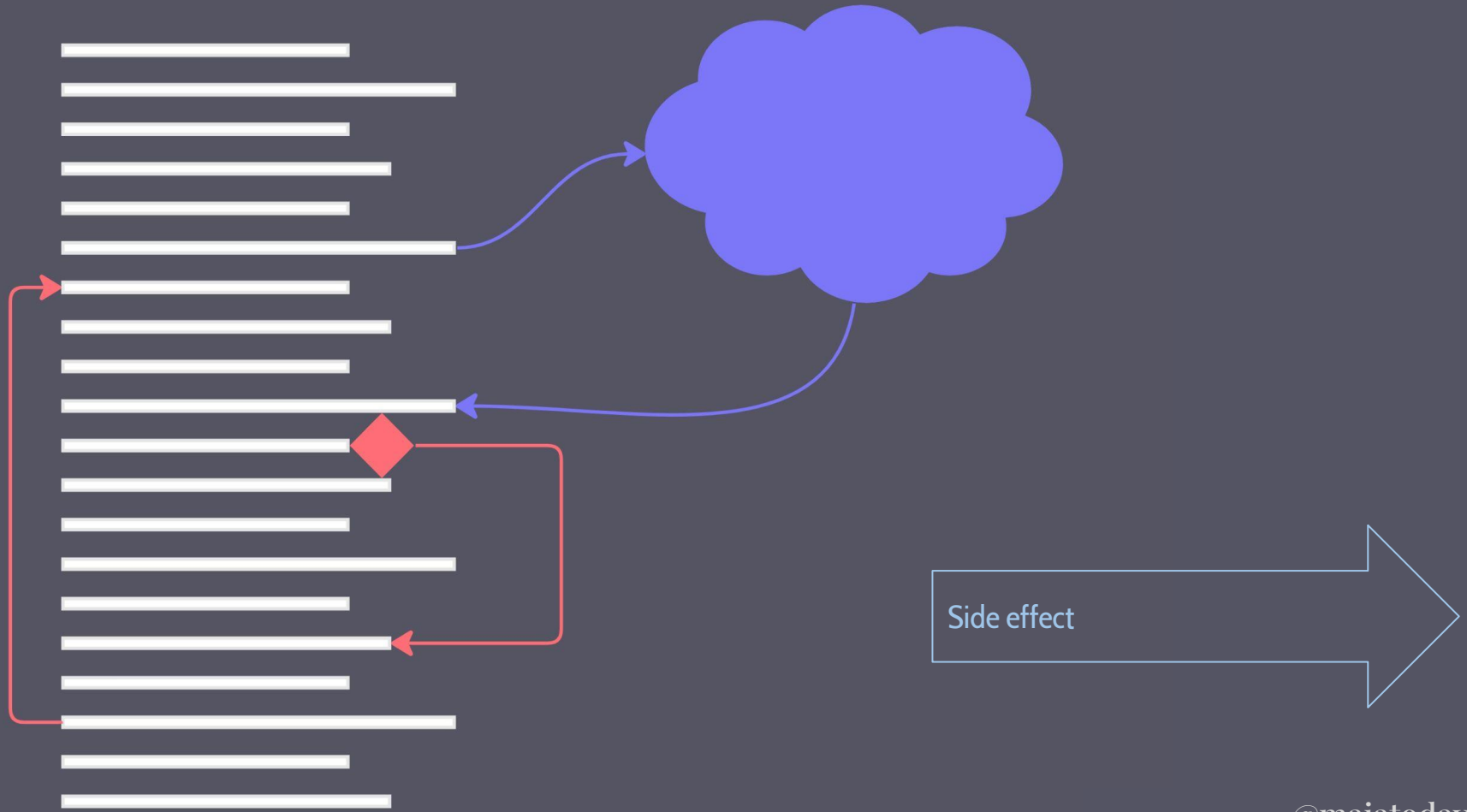
Java -> Scala -> **Kotlin**

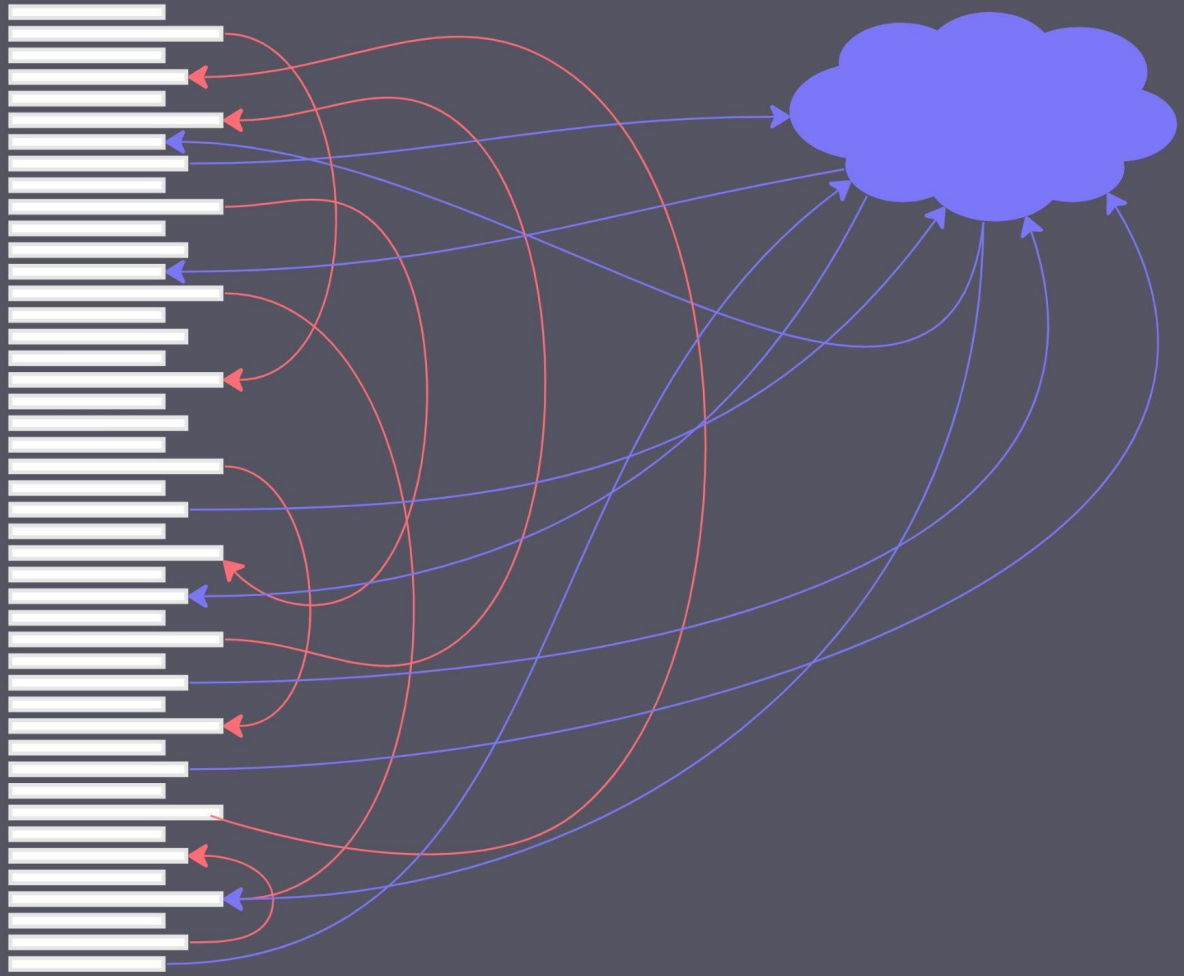
Mobile

Imperative

that consists (mostly) of a series of commands

```
fun sortDeck(deck: IntArray): IntArray {
    for (i in deck.indices)
        for (j in 0 until deck.size - i - 1)
            if (deck[j] > deck[j + 1]) {
                val temp = deck[j]
                deck[j] = deck[j + 1]
                deck[j + 1] = temp
            }
    return deck
}
```

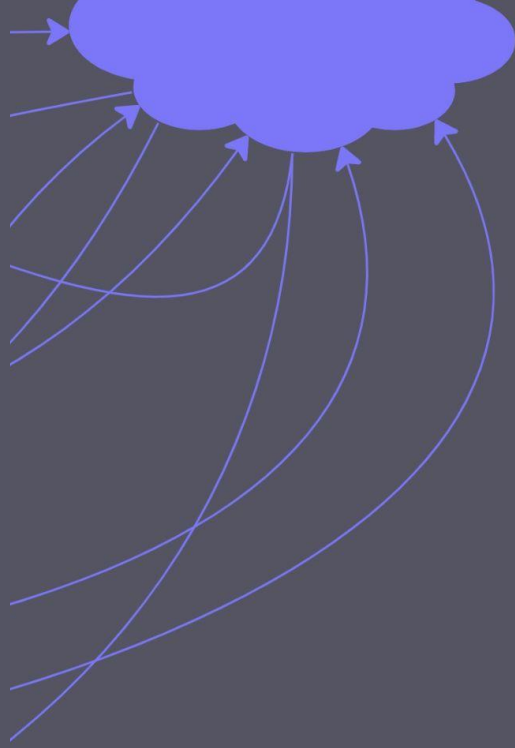

Side effect



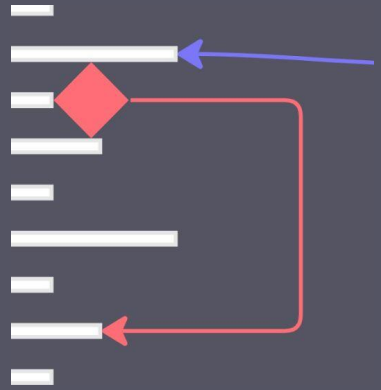
Cut through complexity

Variables

```
val pi = 3.14 // This can't change

var myAnswer = 3 // This can change

myAnswer = 42

var name:String? = null // nullable is a separate type

name = "Maia"
```



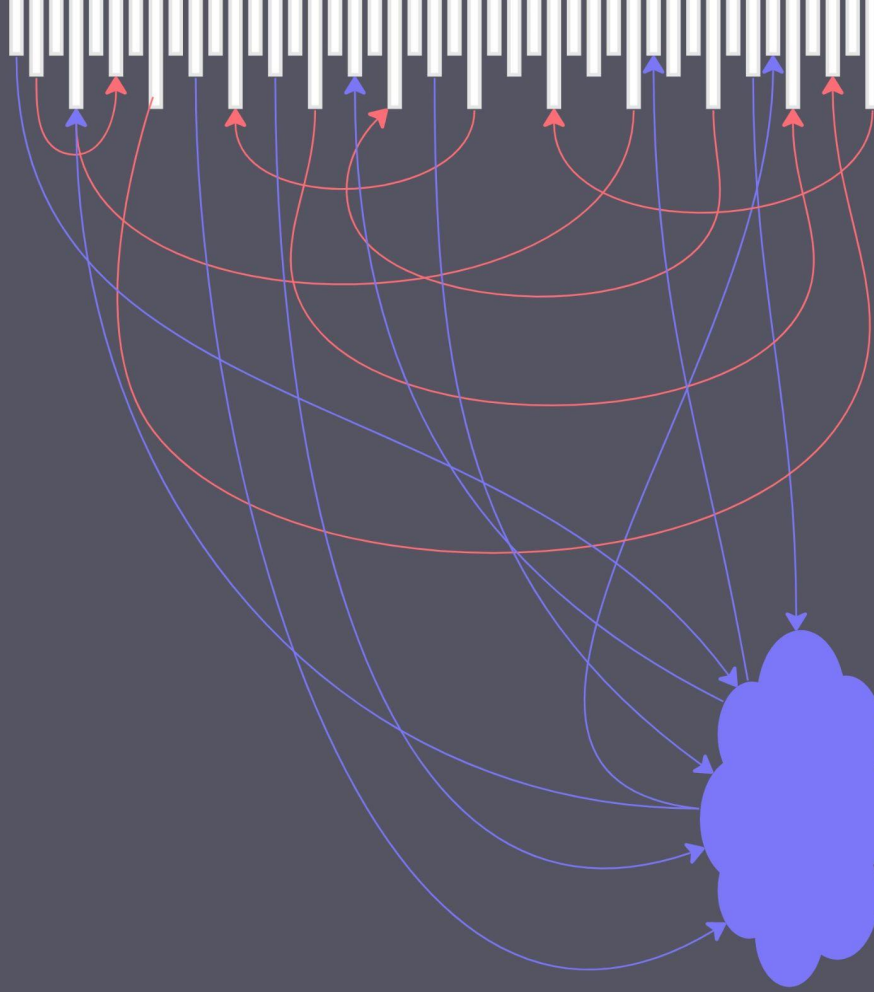
Conditional



```
val max = if (a > b) a else b

when (getSuit()) {
    Suit.HEART -> println("heart")
    Suit.DIAMOND -> println("diamond")
    Suit.SPADE -> println("spade")
    Suit.CLUB -> println("club")
    // 'else' is not required because all cases are covered
}
```

Loops

```
for (i in 6 downTo 0 step 2) {
    println(i)
}

while (x > 0) {
    x--
}

do {
    val y = pickCard()
} while (y != null) // y is visible here!
```



Top level function

```
fun double(x: Int): Int {
    return 2 * x
}
```

Jumps

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) continue // skips rest of code and continues with next inner loop
        if (...) break // breaks out of inner loop
        if (...) break@loop // breaks out of outer loop
    }
}
throw Exception(“Ooops!”)
```

Pros

- Easy to **grasp**

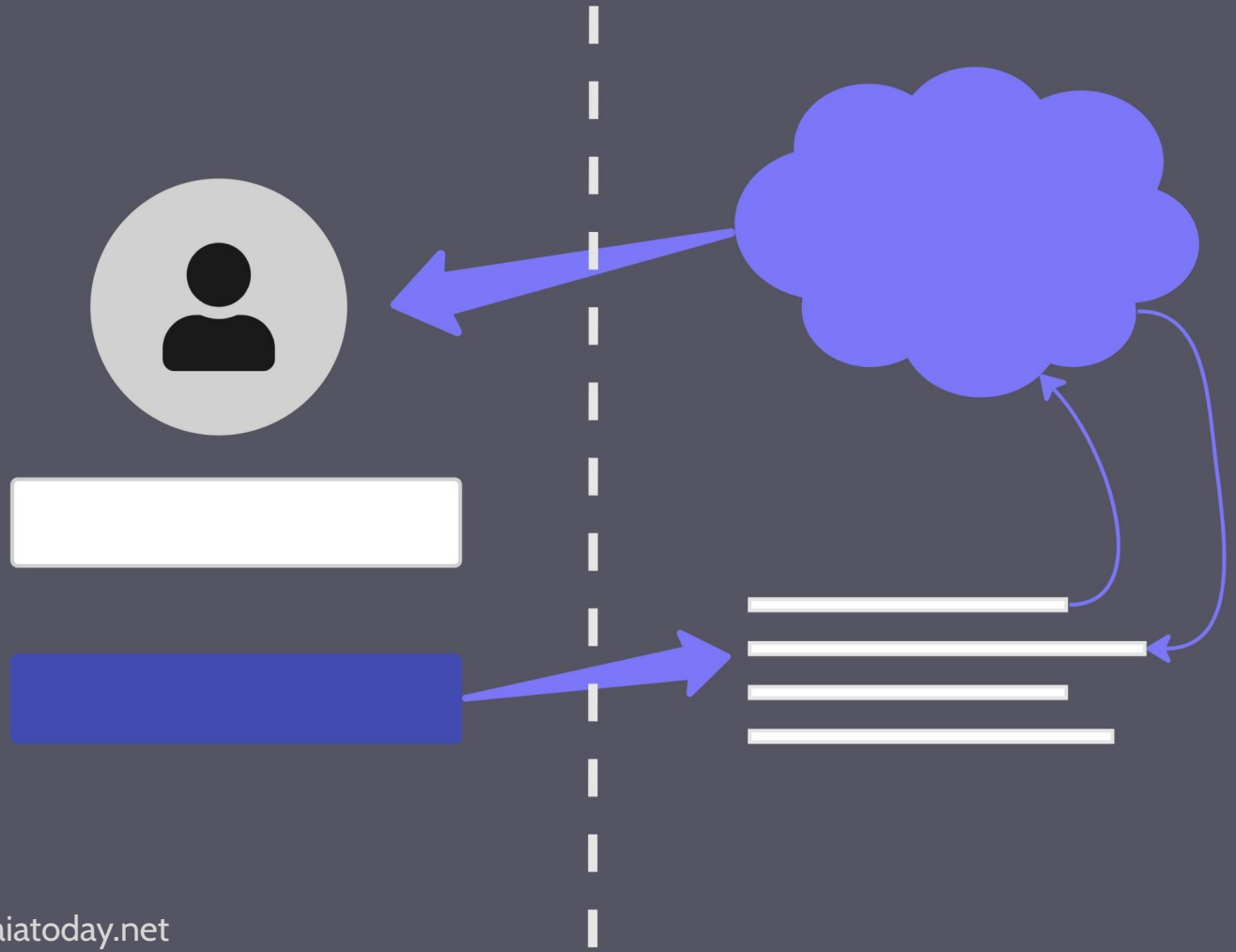
Cons

- Messy **spaghetti**
- **State** and **behaviour** intertwined

Declarative

is a non-imperative style of programming in which programs **describe** their desired results **without** explicitly listing commands or **steps** that must be performed

```
fun PlayingCard(
    suit: Suit,
    isFaceUp: Boolean,
    flipCard: () -> Unit
): Unit {
    Card(onClick = flipCard) {
        if (isFaceUp) {
            when (suit) {
                // ... redacted for brevity
                Suit.HEART -> Image(bitmap = i
            }
        } else {
            Image(bitmap = imageResource("back
        }
    }
}
```



Mlle RIEGO'S

KNITTING BOOK.

I.

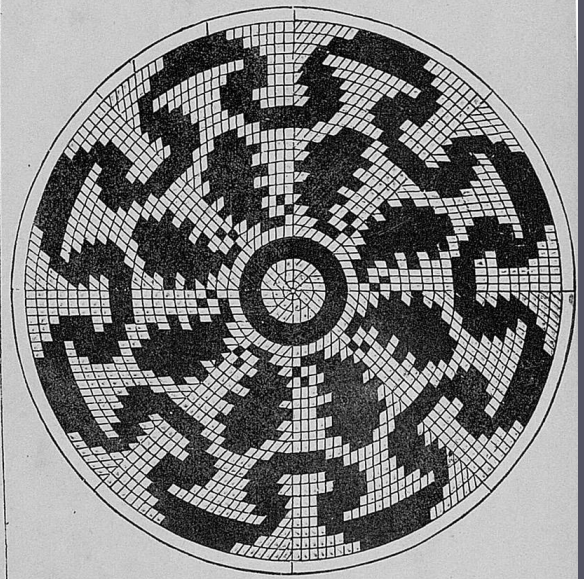
German Pattern of Open Double Knitting, both Sides alike.

Large wooden pins are required for this pattern, which is done in double or eight-thread wool, in 5 colours that contrast well—claret, gold colour, blue, white, scarlet; and 4 rows of each, worked in the order they are here placed. Cast on 71 stitches.

First row:—Seam 1, make 1, slip 1: this row is only to begin with, and is not repeated, the whole of the knitting being done as the

Second row:—Seam 2 together, make 1, by passing the wool round the pin, slip 1, and repeat. At the end of the row, if correctly knitted, there will be 1 stitch, which seam. When the colours have been repeated 6 times, the cover will be the proper size.

No. 10. Bottom of Crochet Bag, page 245.
The Vine Leaf Pattern, next page, is for Bag part.



Function as parameter - higher order function

Higher order function

```
fun IntArray.myMap(transform: (Int) -> Int): List<Int> {
    // loop through array and apply transform on each element
    // return an array of transformed elements
}

val numbers = listOf(1, 2, 3)
numbers.myMap({ it * it }) // [1, 4, 9]
```

varargs

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

Extension functions and infix operator

```
infix fun Int.myPlus(x: Int): Int = this + x

1.myPlus(2)

1 myPlus 2
```

DSL setup

```
enum class Suit { CLUB, DIAMOND, SPADE, HEART, JOKER }

data class Card(
    var suit: Suit = Suit.JOKER,
    var points: Int = 50
)
```

Lambda with receiver - DSL

```
fun card(init: Card.() -> Unit): Card {
    val p = Card()
    p.init()
    return p
}
```

DSL

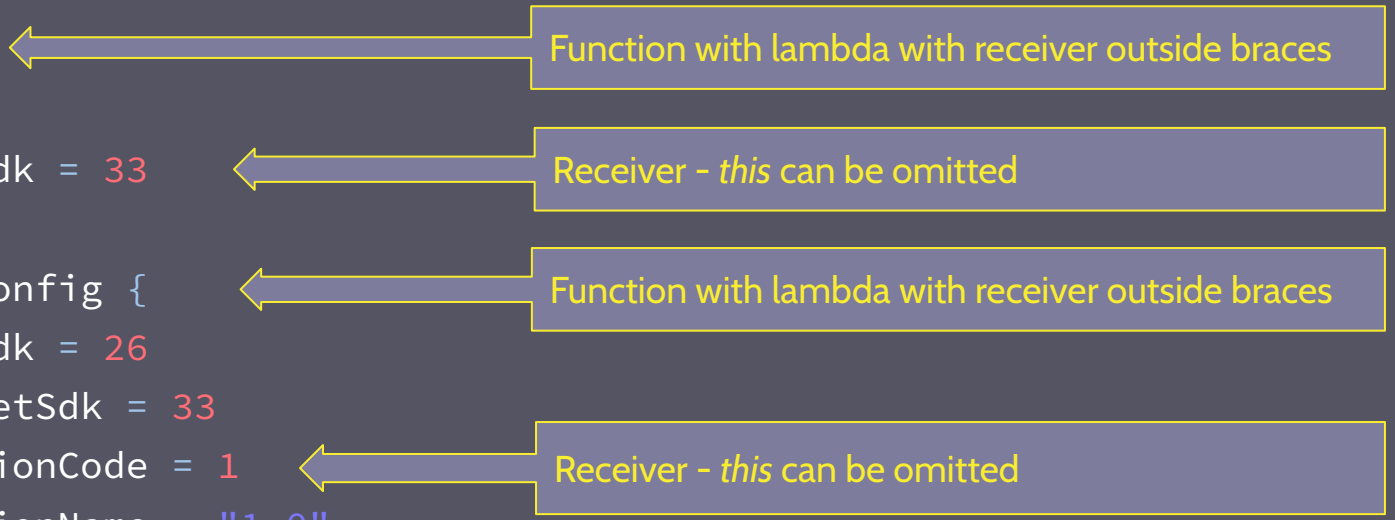
```
card {
    suit = Suit.DIAMOND
    points = 2
}
```

Function with lambda with receiver outside braces

Receiver - *this* can be omitted

Declarative example

```
android {
    compileSdk = 33

    defaultConfig {
        minSdk = 26
        targetSdk = 33
        versionCode = 1
        versionName = "1.0"
    }
}
```



Pros

- Easier to **reason separately** about state/behaviour
- State and behaviour handled **consistently** e.g. unidirectional data flow
- Less code

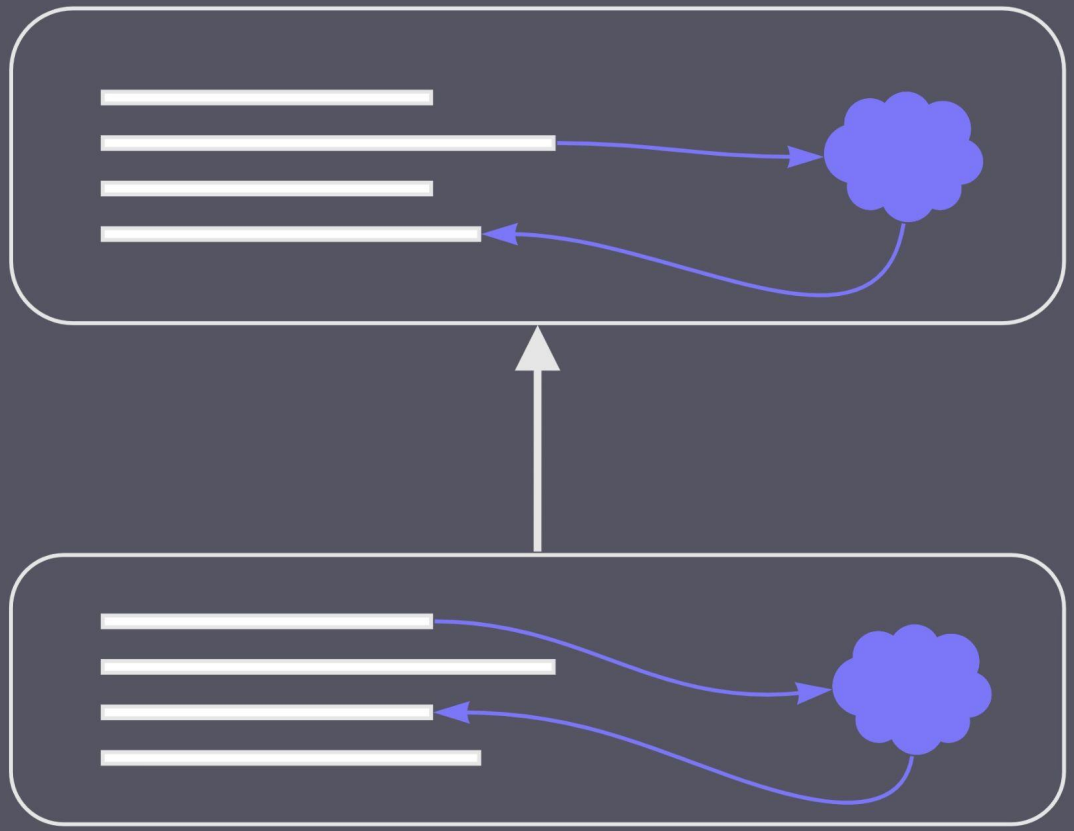
Cons

- Domain specific **frameworks**
- state/actions happen elsewhere **look in two places**
- **Learning curve** learning the framework

Object Oriented

encapsulates the state and operations **inside objects** that support inheritance

```
class SuitCard(
    // state
    pointValue: Int,
    val suit: Suit,
    val faceValue: Int
) : PlayingCard(pointValue) {
    // behaviour
    fun isKing(): Boolean = faceValu
    fun isQueen(): Boolean = faceVal
    fun isJoker(): Boolean = suit ==
}
```

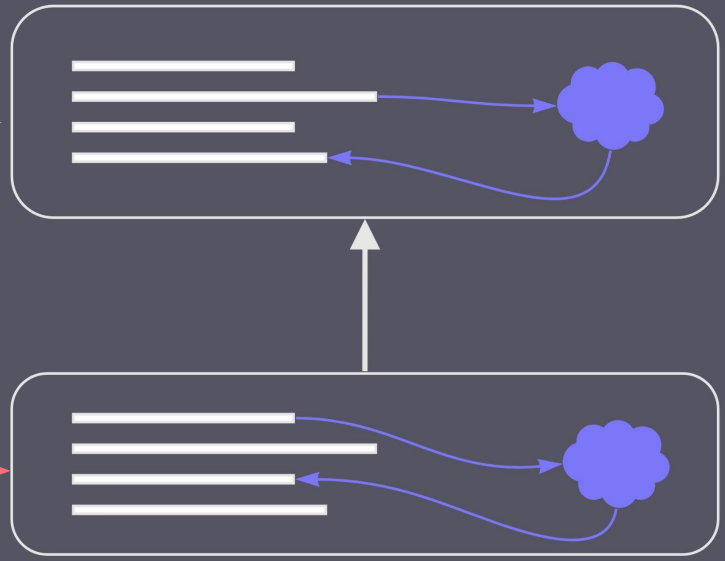


Class - data class - enum - interface

```
class Game { /*...*/ }

data class Player(val name: String, val age: Int)

enum class Suit { CLUB, DIAMOND, SPADE, HEART, JOKER }

interface IdentifyCard {
    fun isKing()
    fun isQueen()
}
```

Inheritance

```
abstract class PlayingCard(val pointValue: Int)

open class SuitCard(
    pointValue: Int,
    val suit: Suit,
    val faceValue: Int
) : PlayingCard(pointValue), IdentifyCard {
    override fun isKing(): Boolean = (faceValue == 13)
    override fun isQueen(): Boolean = (faceValue == 12)
}

class Joker : SuitCard(pointValue = 50, suit = Suit.JOKER, faceValue = 0)
```



Object creation

```
Joker()

val first = SuitCard(suit = Suit.HEART, faceValue = 10)
val second = SuitCard(suit = Suit.DIAMOND, faceValue = 11)
val third = SuitCard(suit = Suit.CLUB, faceValue = 12)
val fourth = SuitCard(suit = Suit.SPADE, faceValue = 13)

if (third.isQueen()) println(“Found a Queen!”)
```

Generics

```
class Box<T>(t: T) {
    var value = t
}
fun <T> someFunction(item: T): List<T>
fun <T> T.someExtension(): String
```

Pattern support - e.g. Delegate setup

```
interface CanPrint {
    fun print(s: String)
}
class Printer() : CanPrint {
    override fun print(s: String) {
        print("Printing $s")
    }
}
```

Pattern support – e.g. Delegate

```
class Derived(p: CanPrint) : CanPrint by p

fun main() {
    val p = Printer()
    Derived(p).print("Hello World!")
}
```

Delegated properties

```
val lazyHi: String by lazy {
    // some heavy computation
    // runs once only when needed
    "Oh hello there"
}
```

Pros

- **Easy** to understand
- State + behaviour **encapsulated**

Cons

- Everything is NOT an object
- Pattern of **utils classes** TextUtil StringUtil
- **Memory usage** and **performance** for object instantiation

Functional

in lieu of assignment, uses **procedure calls** to bind variables to values, so that referential transparency is preserved

```
fun scoreJokers(
    deck: List<SuitCard>
): Int =
    deck.filter { card ->
            card.isJoker()
        }
        .map { card ->
            card.pointValue
        }
        .sum()
```

Referential transparency

```
var x = giveMeFive()
fun giveMeFive() = { 5 }
// replace function invoke with result
x = 5
```

Memoization


```
for (i in list) {

    // compare and
       discard or keep

}
```

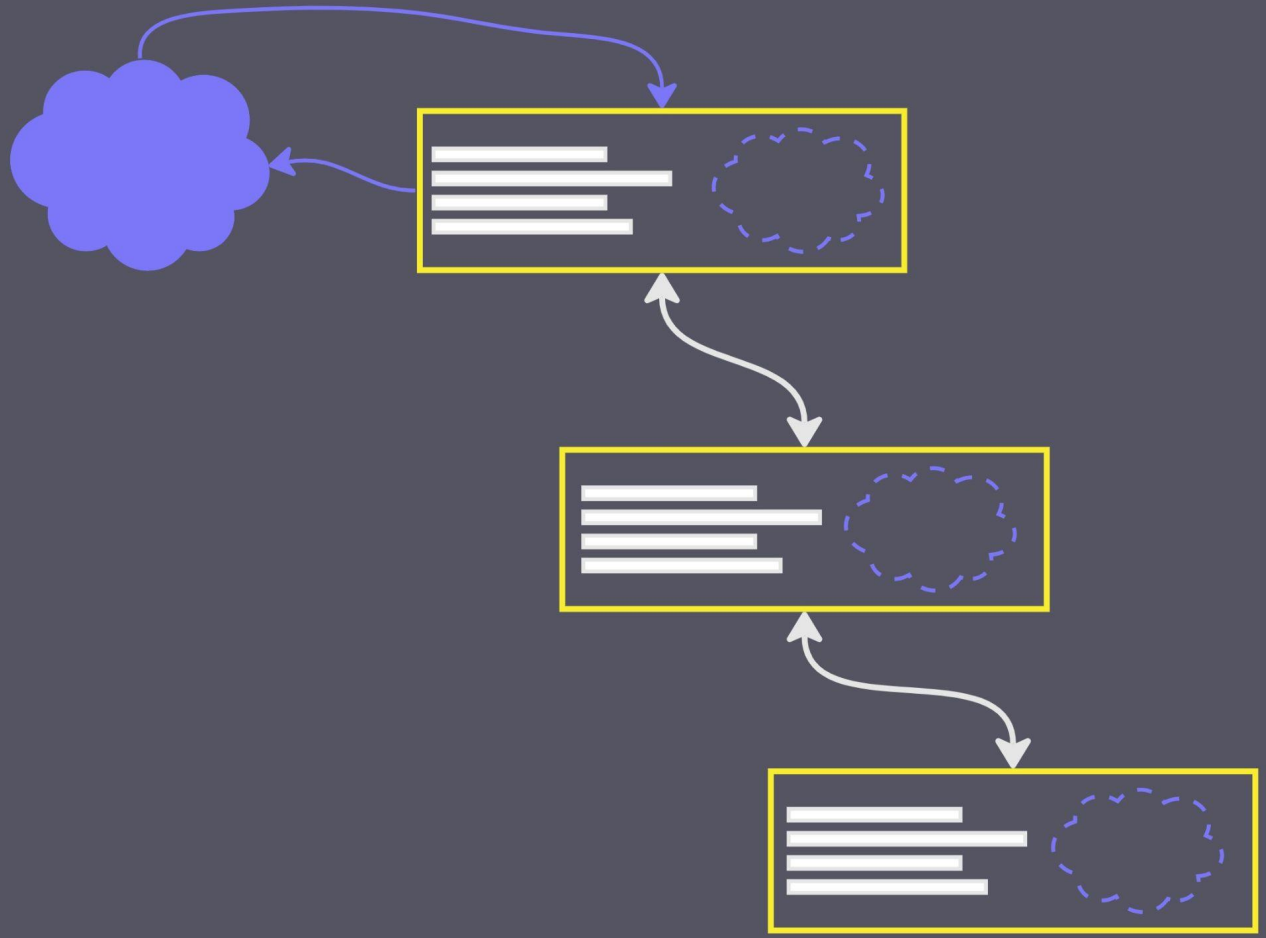
```
for (i in list) {

    // convert i to
       Something else

}
```

```
for (i in list) {

    // extract and
         convert

}
```

```
list.map { i ->

    // compare and
       discard or keep

}
```

```
list.map { i ->

    // convert i to
       Something else

}
```

```
list.map { i ->

    // extract and
         convert

}
```

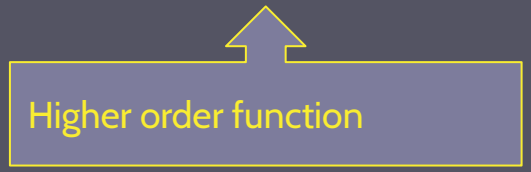


Top level function

```
fun double(x: Int): Int {
    return 2 * x
}
```

Functional types and lambdas

```
val operation: (Int, Int) -> Int = { a, b -> a + b }

fun IntArray.myMap(transform: (Int) -> Int): List<Int>
```




```
val numbers = listOf(1, 2, 3)
numbers.myMap { it * it } // [1, 4, 9]
```

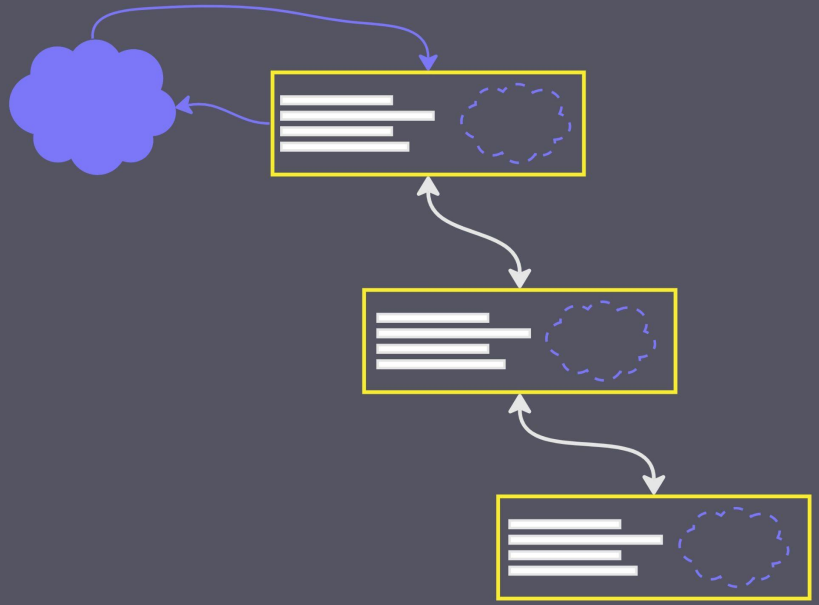
Monad 😰

Pattern □

Type to wrap - interface and generics

Function to create - *return*

Function to transform - *bind - map*



~~Curry~~

```
fun subtract(x: Int, y: Int): Int
                    = { x - y }

subtract(50, 8)
```

```
fun subtract(x: Int): (Int) -> Int
          = { y: Int -> x - y }

subtract(50)(8)
```

se. ex quibus in ipso pregnantur paria 2 et sunt in quarto mense paria 5 ex quibus paria 3 geminant alia paria 3 quibus additis cum pariis 5 fiunt in quinto mense paria 8 ex quibus paria 5 que geminata fuerunt in ipso mense non concipiunt in ipso mense, sed alia 8 paria pregnantur; et sic sunt in sexto mense paria 13 cum quibus additis pariis 8 que geminant in septimo erunt in ipso paria 21 cum quibus additis pariis 13 que geminant in octavo mense erunt in ipso paria 34 cum quibus additis pariis 21 que geminant in nono mense erunt in ipso paria 55 cum quibus additis rursum pariis 34 que geminant in decimo erunt in ipso paria 89 cum quibus additis rursum pariis 55 que geminant in undecimo mense erunt in ipso paria 144 cum quibus additis rursum pariis 89 que geminant in ultimo mense erunt paria 233 et tot paria peperit summa par in prefato loco in capite unius anni. Potest enim videri in hac margine qualiter hoc operati fuimus, scilicet quod iunximus primum numerum cum secundo, videlicet 1 cum 2, et secundum cum tertio, et tertium cum quarto, et quartum cum quinto, et sic deinceps donec iunximus decimum cum undecimo, videlicet 144 cum 233, et habuimus suprascriptorum cuniculorum summam, videlicet 377, et sic posses facere per ordinem de infinitis numeris mensibus.

Quatuor homines sunt, quorum primus, secundus et tertius habent denarios. Secundus, tertius et quartus ...

Primus 1
2
Secundus
3
tercius
5
Quartus
8
Quintus
13
Sextus
21
Septimus
34
Octavus
55
Nonus
89

Tail recursion

```
tailrec fun fibonacci(n: Int, a: Int = 0, b: Int = 1): Int =
    when (n) {
        0 -> a
        1 -> b
        else -> fibonacci(n - 1, b, a + b)
    }
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...

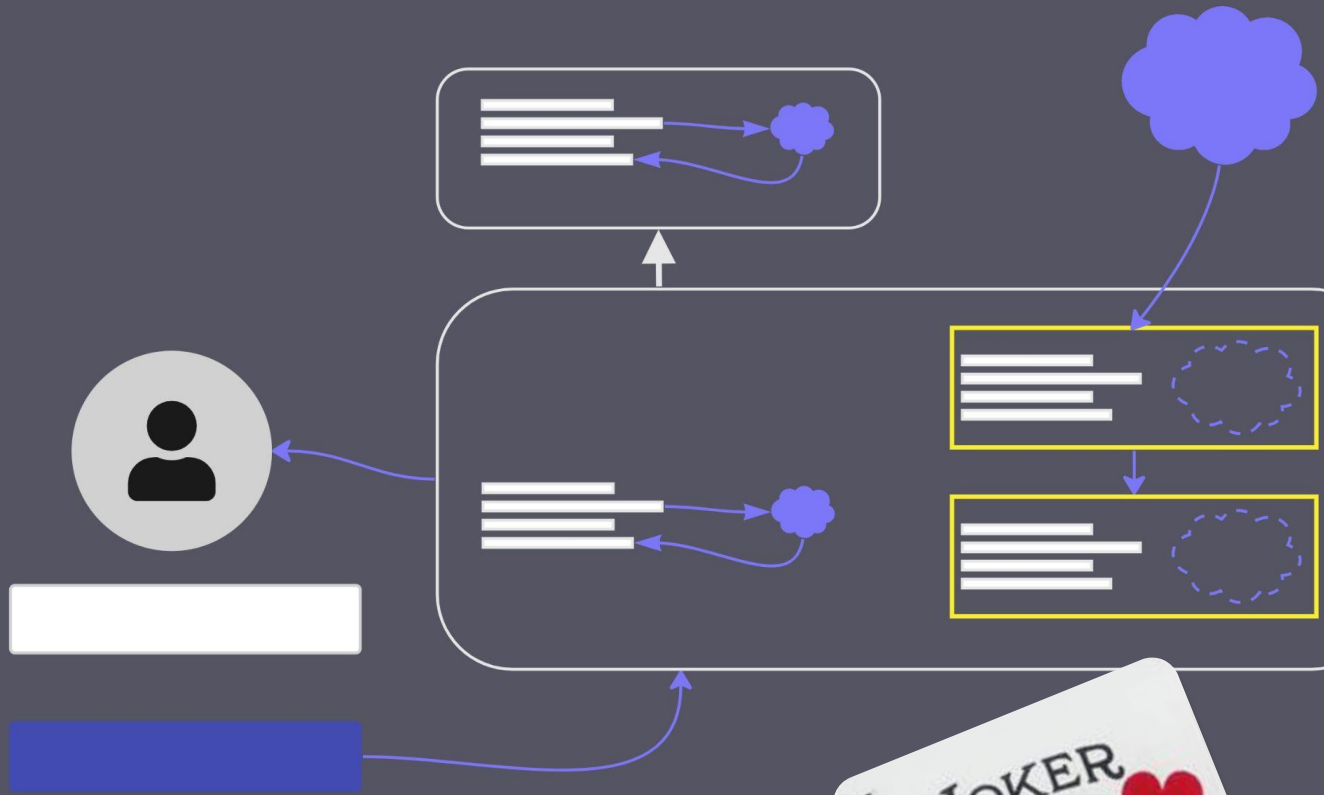
Pros

- Infinite data **streams** and transforming data
- Abstract **reusable algorithms**
- **Testing** small pure functions
- **Concurrency** simplified
- **Compact**
- Referential transparency -> **Memoization**

Cons

- Immutable objects - memory allocation
- Learn abstractions
- Complex/Dense - Debug

The real world



Side effect

“Kotlin is **pragmatic**. We **solve**. We’re not into any kind of pure theoretical computer science. We’re not here to do any religious wars. We’re not here to promote our vision.

We are here to solve problems developers face day to day.”

Roman Elizarov

Resources

maiatoday.net

Imperative programming

Declarative programming

Object-oriented programming

Functional programming

Kotlin docs

History of knitting

Kotlin monad library

Roman Elizarov on Medium

Liber Abaci by Leonardo Pisano

Thinking in Compose

Human working memory

```java
public static final int fibonacci(int n, int a, int b) {
    while(true) {
        int var10000;
        switch(n) {
        case 0:
            var10000 = a;
            break;
        case 1:
            var10000 = b;
            break;
        default:
            var10000 = n - 1;
            int var10001 = b;
            b += a;
            a = var10001;
            n = var10000;
            continue;
        }
        return var10000;
    }
}
```

maiatoday.net

@maiatoday