Refactoring to Functional

Hadi Hariri

Develop with pleasure!

Functional Programming

In computer science, functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical **functions** and **avoids state and mutable data**. It is a **declarative** programming paradigm, which means programming is done with **expressions**.

Let's refactor to functional

Wait. But why?

Goals

- Write less code.
- Write expressive code.
- Write correct code.
- Write performant code.
  (because Silicon Valley)

Be part of the “in-crowd”

Can I use any language?

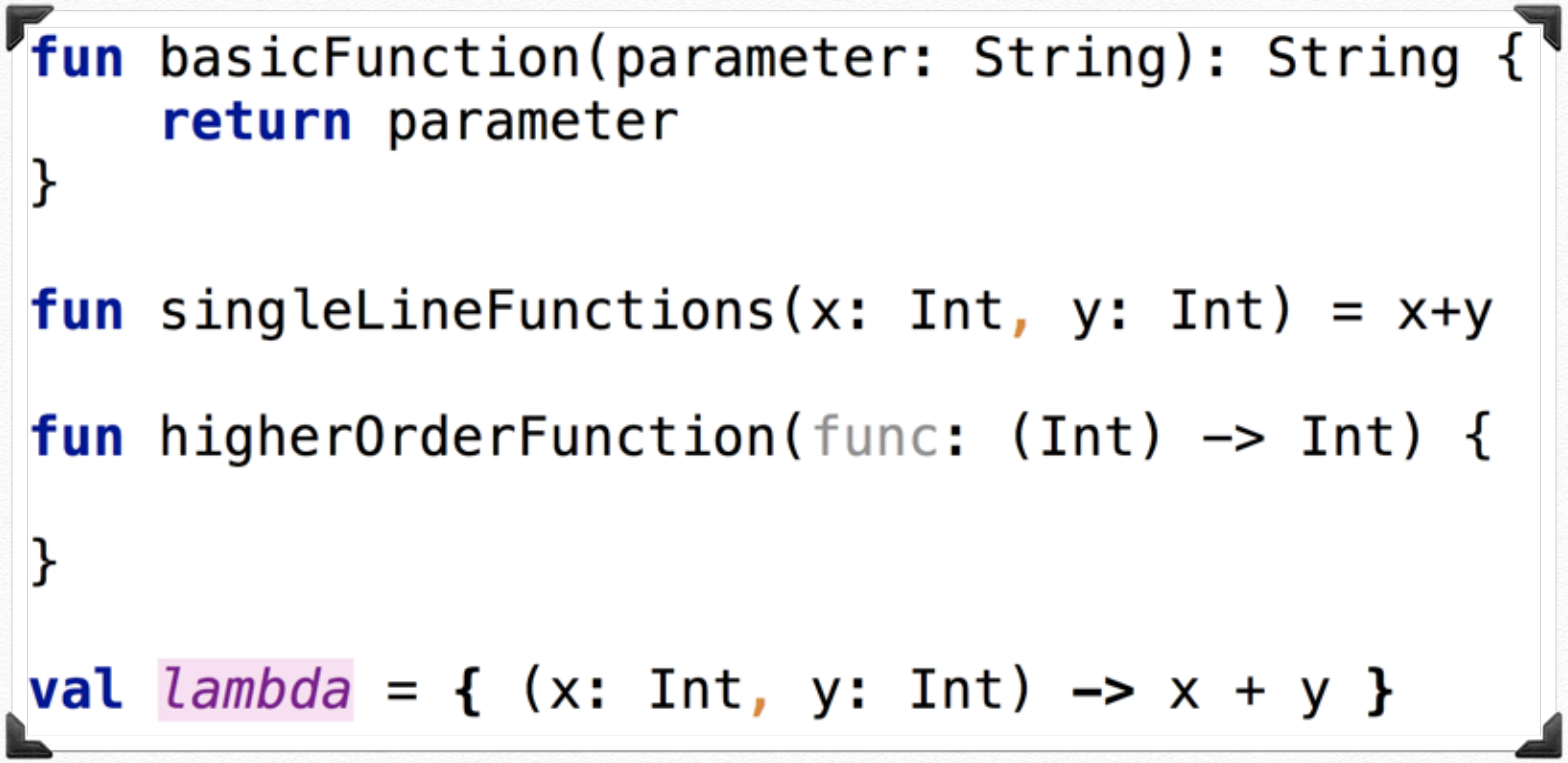
What is a functional language?

A language with…

- Functions as First-Class Citizens
  - Higher-Order Functions
  - Lambdas
  - Top-level Functions*
- Immutable Data

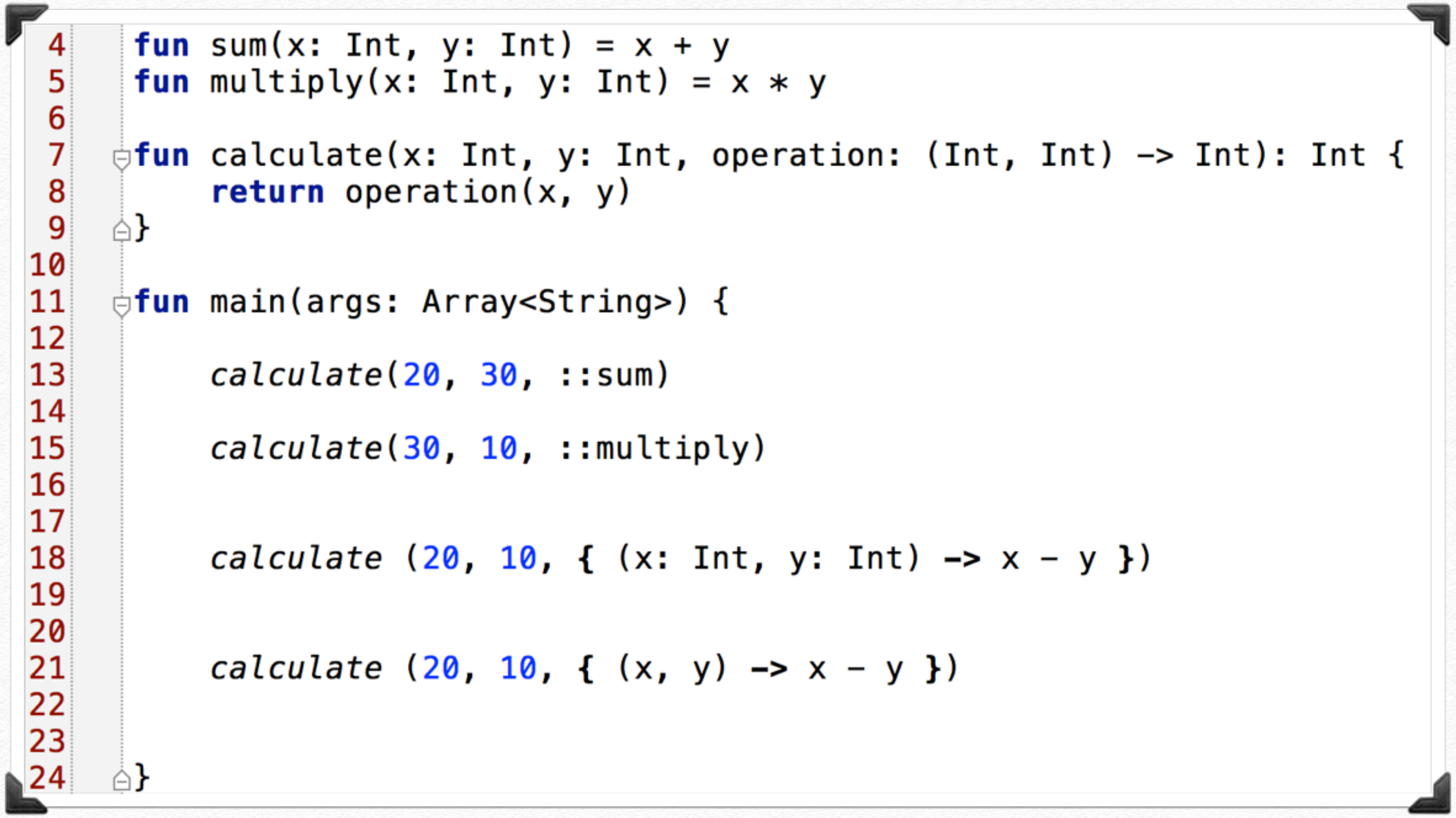
…I'll be using Kotlin. You can use Java 8, Java 7..5 +FJ, Scala, JavaScript, Clojure, Haskell, F#, C#, OCaml, Lisp.

Basic Function Syntax



```kotlin
fun basicFunction(parameter: String): String {
    return parameter
}


fun singleLineFunctions(x: Int, y: Int) = x+y

fun higherOrderFunction(func: (Int) -> Int) {

}

val lambda = { (x: Int, y: Int) -> x + y }
```

Higher-Order Functions and Lambdas



```
fun sum(x: Int, y: Int) = x + y
fun multiply(x: Int, y: Int) = x * y

fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int): Int {
    return operation(x, y)
}

fun main(args: Array<String>) {

    calculate(20, 30, ::sum)

    calculate(30, 10, ::multiply)


    calculate (20, 10, { (x: Int, y: Int) -> x - y })


    calculate (20, 10, { (x, y) -> x - y })


}
```

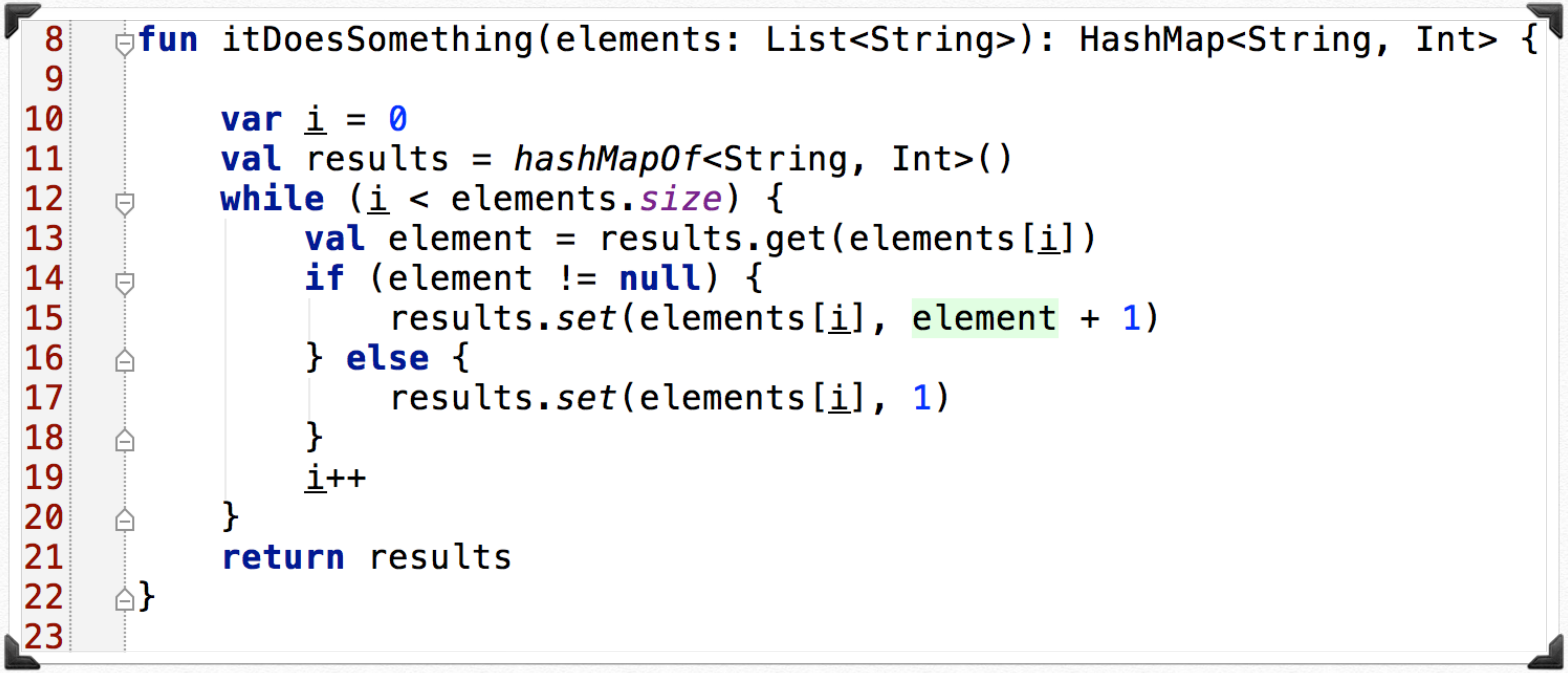
Pure Functions

- Given same input, function always provides same output
- Execution does not cause observable side-effects
- Referential Transparency

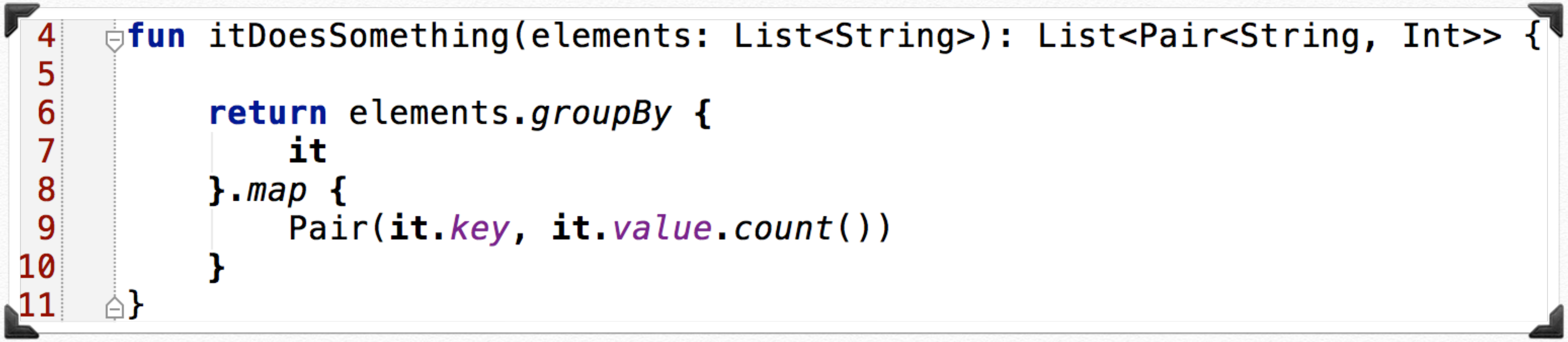
Reduce moving parts

Aim for Pure Functions

Can we achieve our goals?



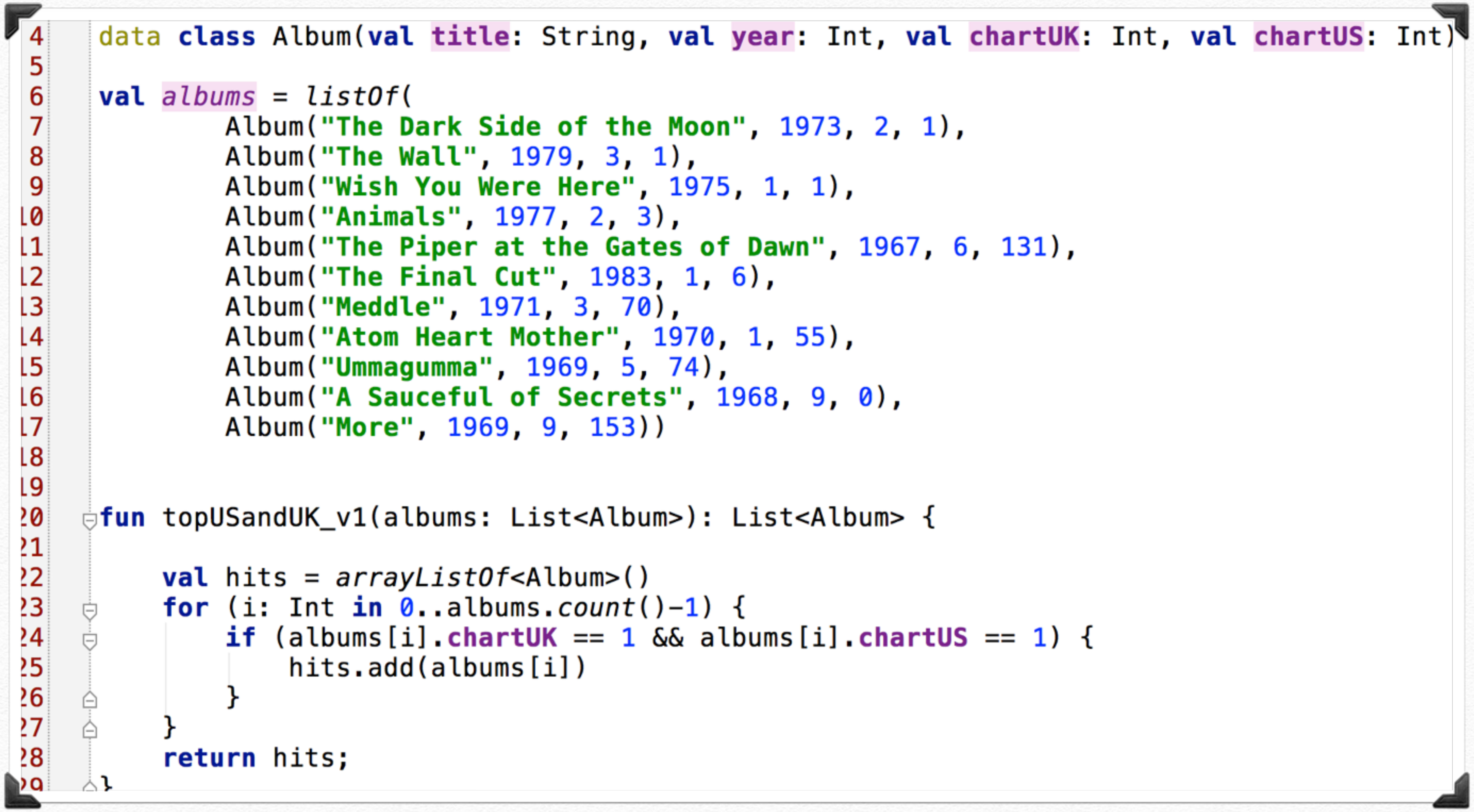
```kotlin
fun itDoesSomething(elements: List<String>): HashMap<String, Int> {

    var i = 0
    val results = hashMapOf<String, Int>()
    while (i < elements.size) {
        val element = results.get(elements[i])
        if (element != null) {
            results.set(elements[i], element + 1)
        } else {
            results.set(elements[i], 1)
        }
        i++
    }
    return results
}

```



```kotlin
fun itDoesSomething(elements: List<String>): List<Pair<String, Int>> {

    return elements.groupBy {
        it
    }.map {
        Pair(it.key, it.value.count())
    }
}
```

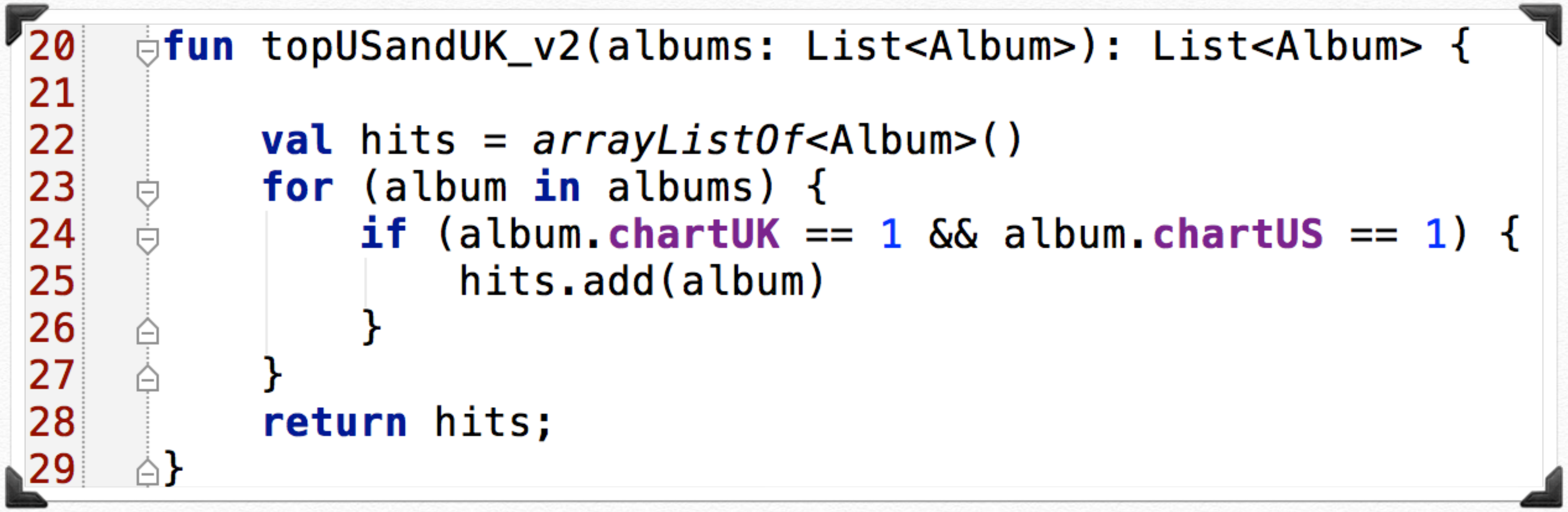
What's functional about that?

The Loops



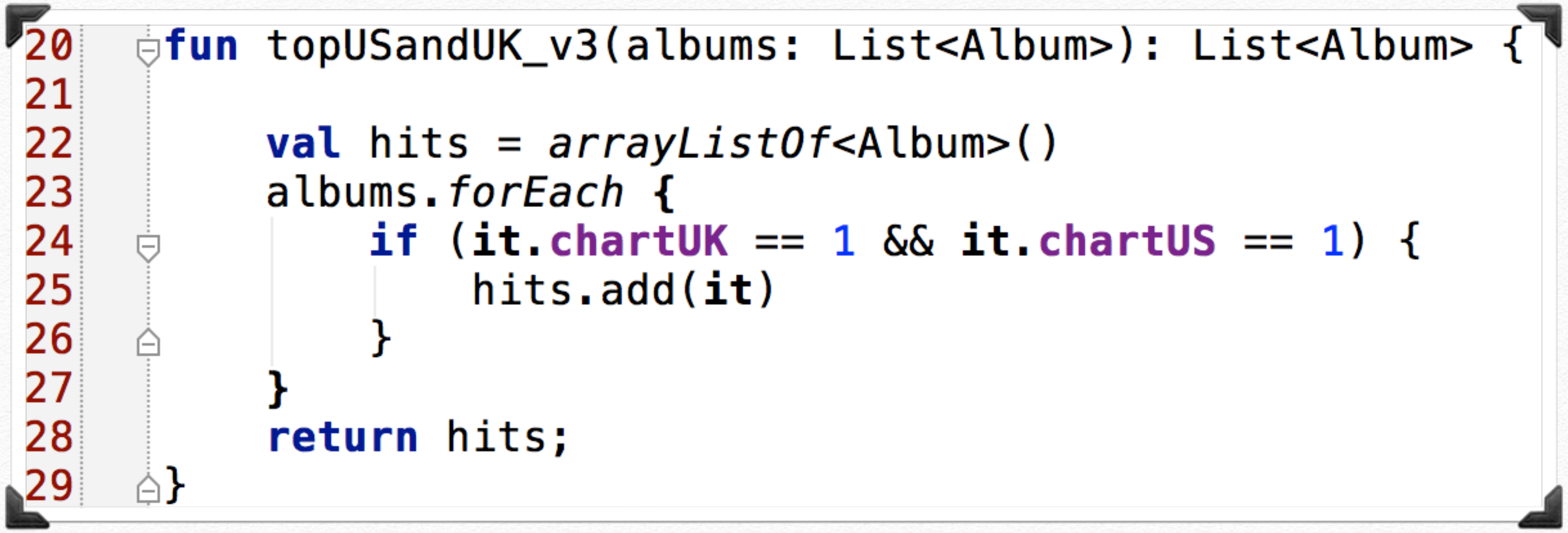
```
data class Album(val title: String, val year: Int, val chartUK: Int, val chartUS: Int)

val albums = listOf(
        Album("The Dark Side of the Moon", 1973, 2, 1),
        Album("The Wall", 1979, 3, 1),
        Album("Wish You Were Here", 1975, 1, 1),
        Album("Animals", 1977, 2, 3),
        Album("The Piper at the Gates of Dawn", 1967, 6, 131),
        Album("The Final Cut", 1983, 1, 6),
        Album("Meddle", 1971, 3, 70),
        Album("Atom Heart Mother", 1970, 1, 55),
        Album("Ummagumma", 1969, 5, 74),
        Album("A Sauceful of Secrets", 1968, 9, 0),
        Album("More", 1969, 9, 153))


fun topUSandUK_v1(albums: List<Album>): List<Album> {

    val hits = arrayListOf<Album>()
    for (i: Int in 0..albums.count()-1) {
        if (albums[i].chartUK == 1 && albums[i].chartUS == 1) {
            hits.add(albums[i])
        }
    }
    return hits;
}
```

Removing State



```kotlin
fun topUSandUK_v2(albums: List<Album>): List<Album> {

    val hits = arrayListOf<Album>()
    for (album in albums) {
        if (album.chartUK == 1 && album.chartUS == 1) {
            hits.add(album)
        }
    }
    return hits;
}
```

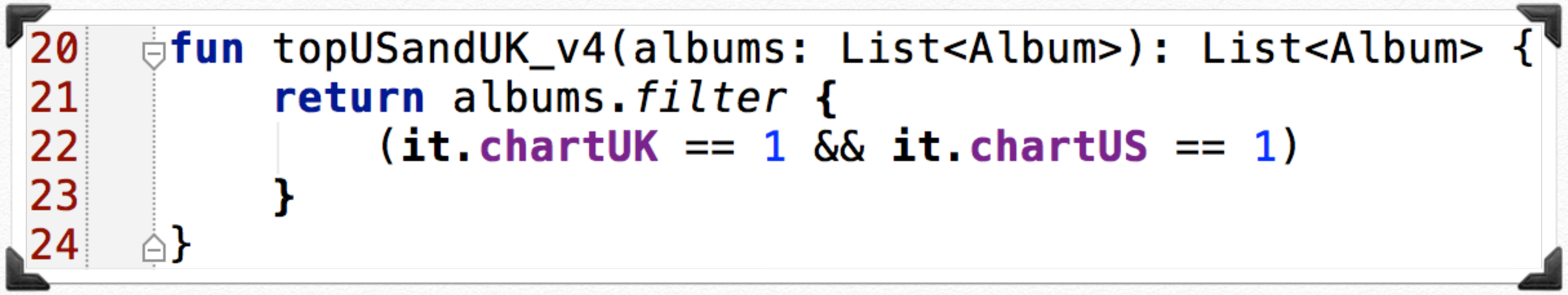
Some syntax sugar



```kotlin
fun topUSandUK_v3(albums: List<Album>): List<Album> {

    val hits = arrayListOf<Album>()
    albums.forEach {
        if (it.chartUK == 1 && it.chartUS == 1) {
            hits.add(it)
        }
    }
    return hits;
}
```

Still emphasis on how, not what

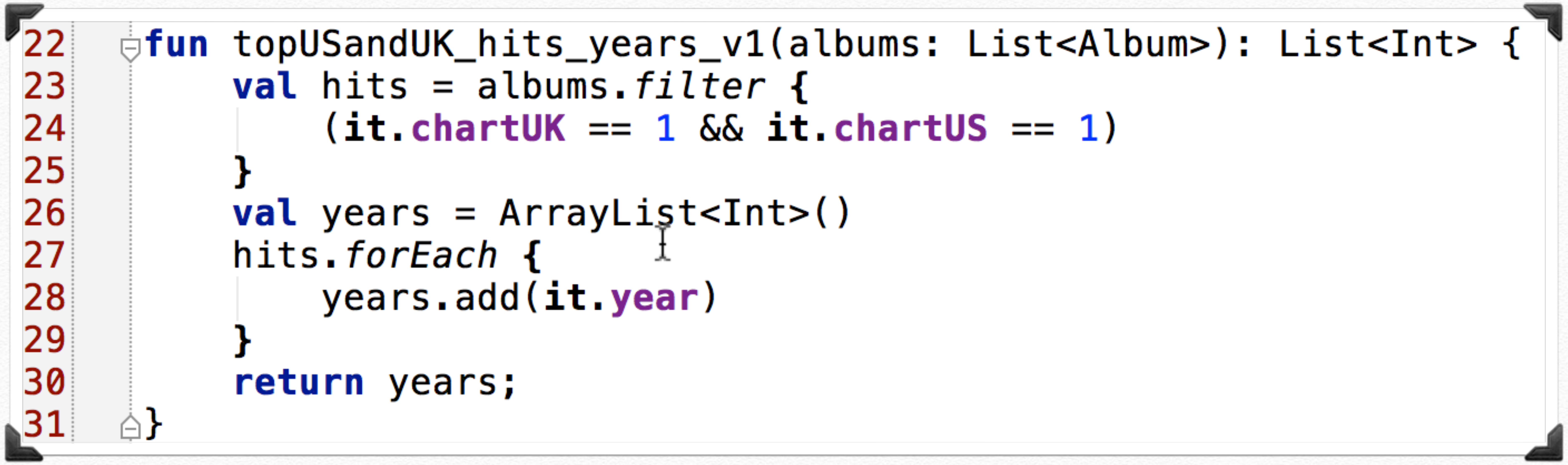
Writing less code



```kotlin
fun topUSandUK_v4(albums: List<Album>): List<Album> {
    return albums.filter {
        (it.chartUK == 1 && it.chartUS == 1)
    }
}
```

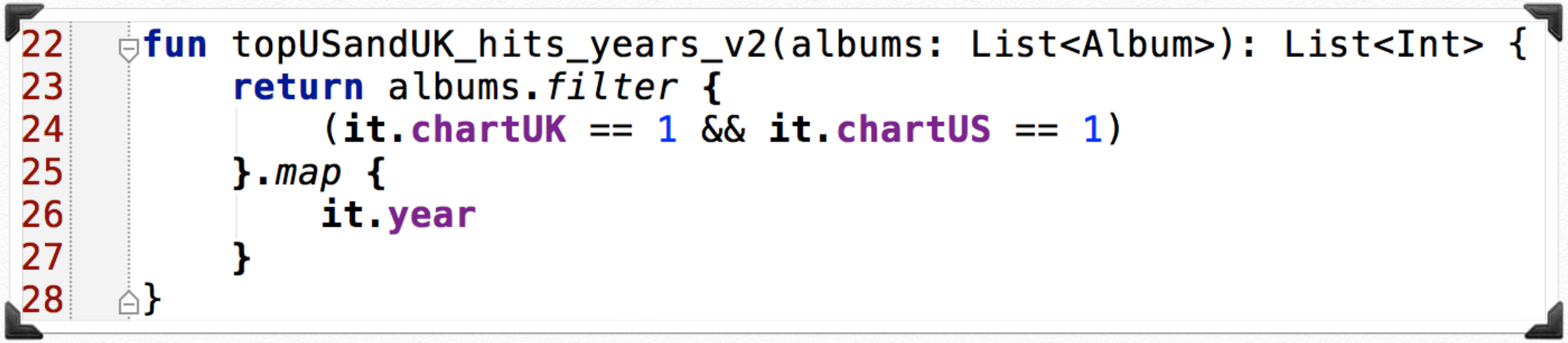
Getting the data the way we need it

Transforming data



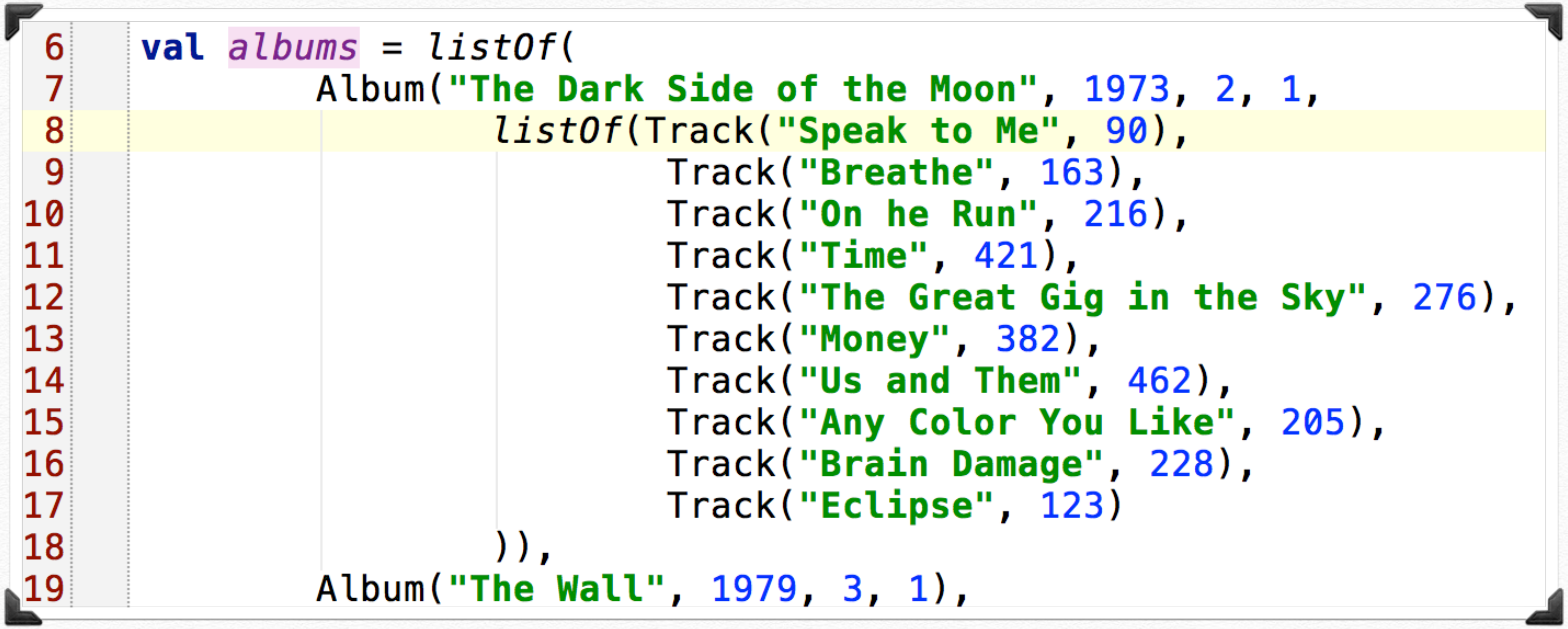
```kotlin
fun topUSandUK_hits_years_v1(albums: List<Album>): List<Int> {
    val hits = albums.filter {
        (it.chartUK == 1 && it.chartUS == 1)
    }
    val years = ArrayList<Int>()
    hits.forEach {
        years.add(it.year)
    }
    return years;
}
```

Again, let's write less



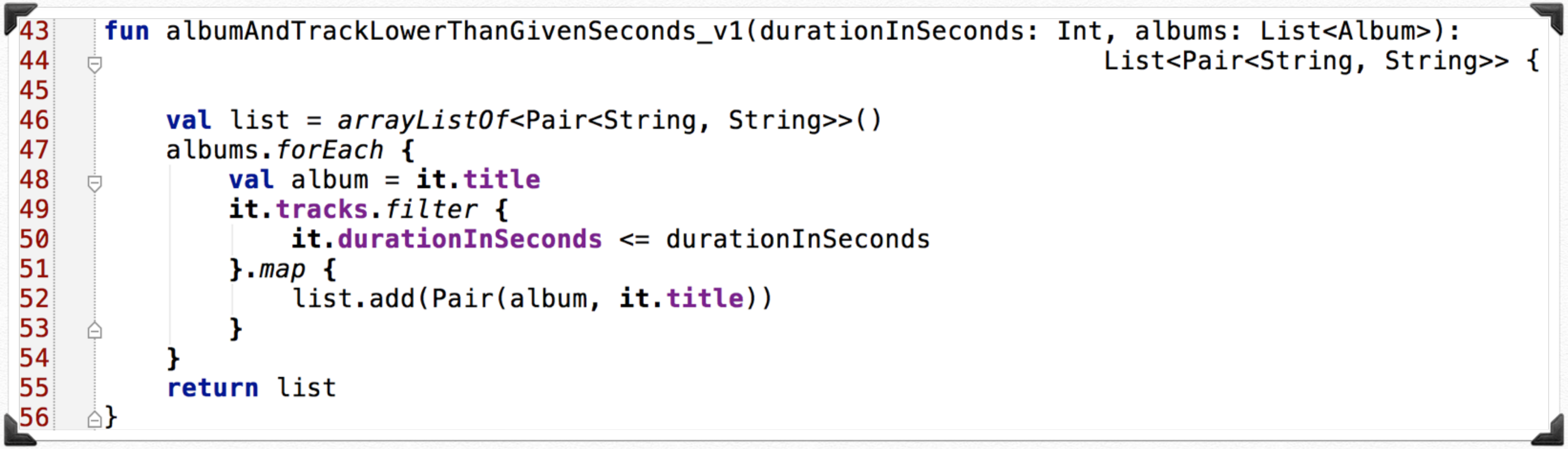
```kotlin
fun topUSandUK_hits_years_v2(albums: List<Album>): List<Int> {
    return albums.filter {
        (it.chartUK == 1 && it.chartUS == 1)
    }.map {
        it.year
    }
}
```

A more involved sample…



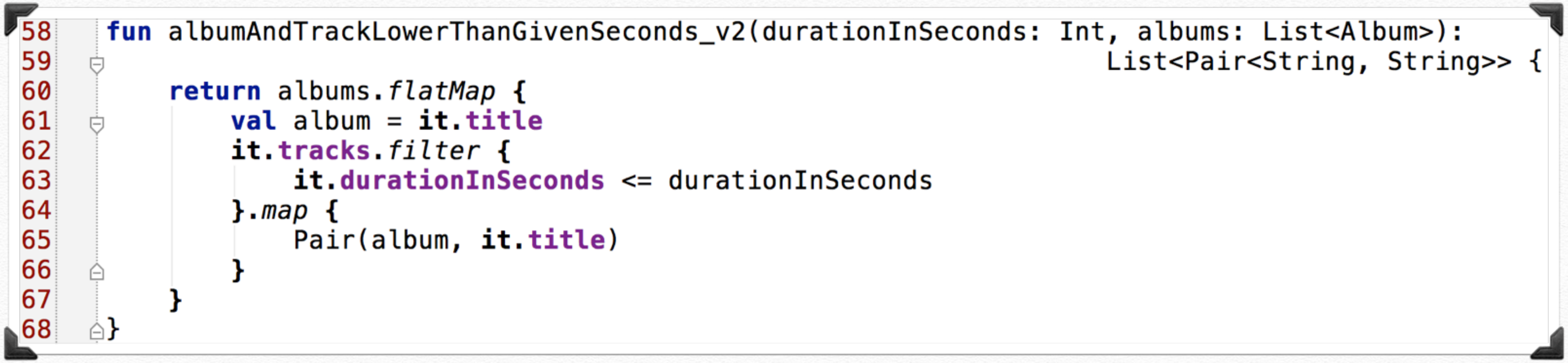
```
val albums = listOf(
        Album("The Dark Side of the Moon", 1973, 2, 1,
                listOf(Track("Speak to Me", 90),
                        Track("Breathe", 163),
                        Track("On he Run", 216),
                        Track("Time", 421),
                        Track("The Great Gig in the Sky", 276),
                        Track("Money", 382),
                        Track("Us and Them", 462),
                        Track("Any Color You Like", 205),
                        Track("Brain Damage", 228),
                        Track("Eclipse", 123)
                )),
        Album("The Wall", 1979, 3, 1),
```

The “manual” way

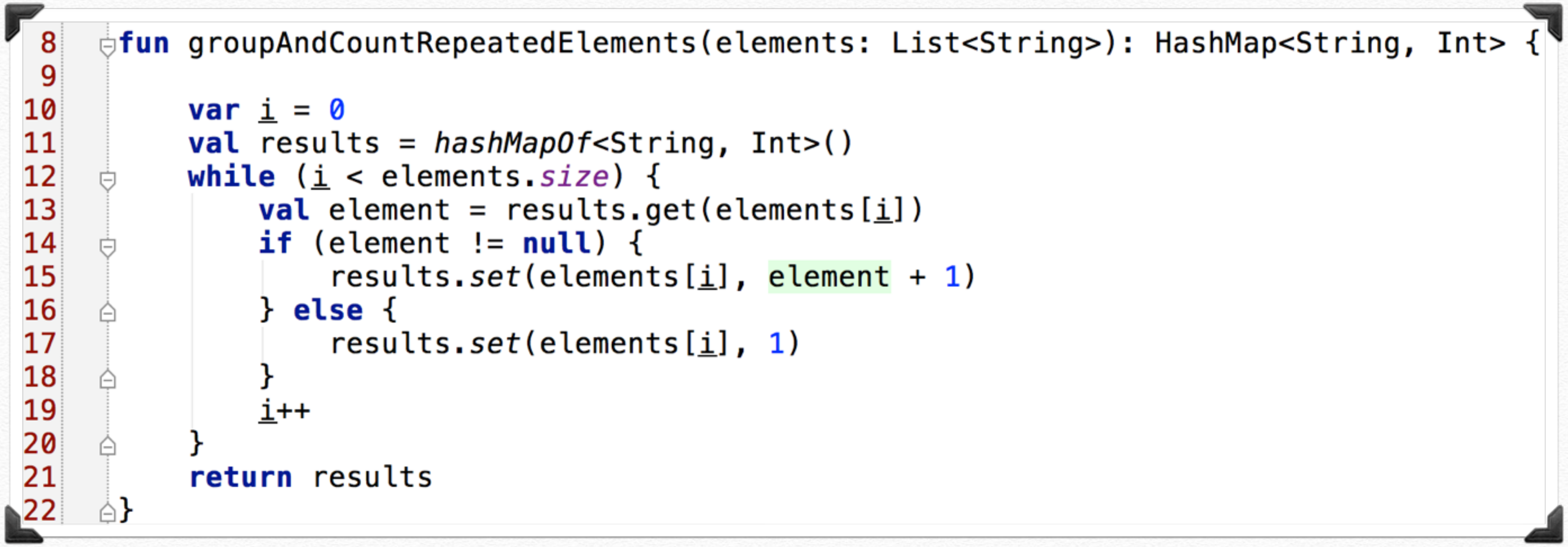


```kotlin
fun albumAndTrackLowerThanGivenSeconds_v1(durationInSeconds: Int, albums: List<Album>):
                                                                          List<Pair<String, String>> {

    val list = arrayListOf<Pair<String, String>>()
    albums.forEach {
        val album = it.title
        it.tracks.filter {
            it.durationInSeconds <= durationInSeconds
        }.map {
            list.add(Pair(album, it.title))
        }
    }
    return list
}
```

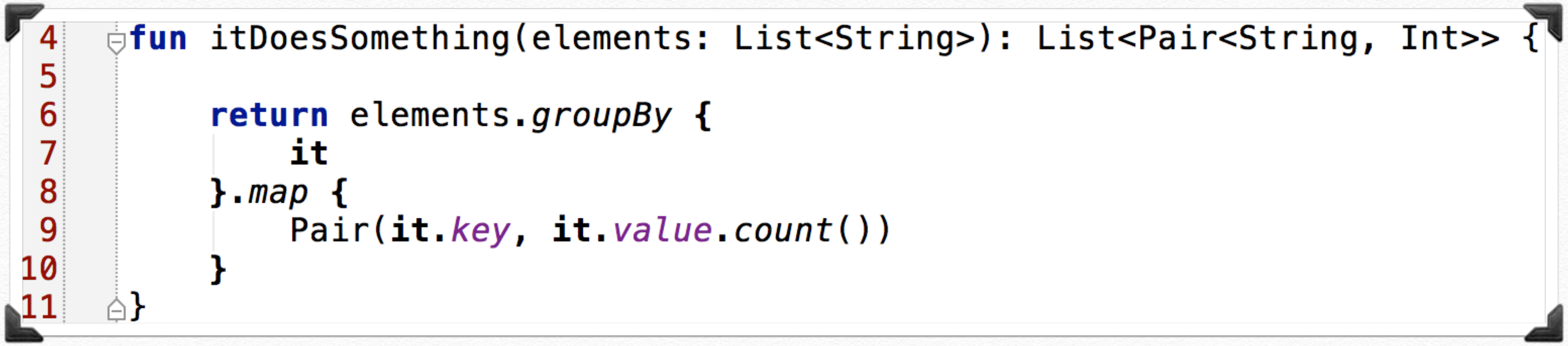
Once again, writing less code



```kotlin
fun albumAndTrackLowerThanGivenSeconds_v2(durationInSeconds: Int, albums: List<Album>):
                                                                        List<Pair<String, String>> {
    return albums.flatMap {
        val album = it.title
        it.tracks.filter {
            it.durationInSeconds <= durationInSeconds
        }.map {
            Pair(album, it.title)
        }
    }
}
```



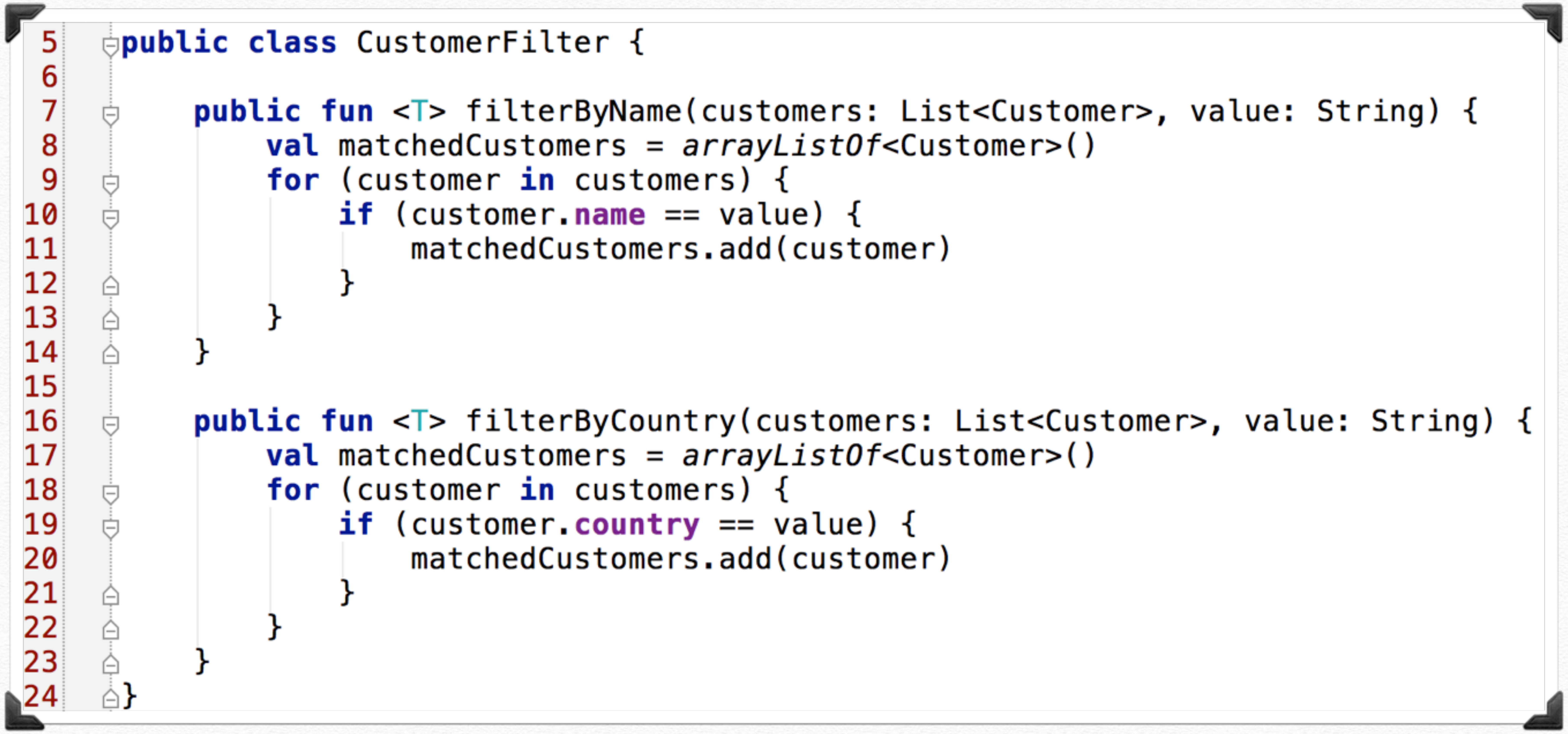
```kotlin
fun groupAndCountRepeatedElements(elements: List<String>): HashMap<String, Int> {

    var i = 0
    val results = hashMapOf<String, Int>()
    while (i < elements.size) {
        val element = results.get(elements[i])
        if (element != null) {
            results.set(elements[i], element + 1)
        } else {
            results.set(elements[i], 1)
        }
        i++
    }
    return results
}
```



```kotlin
fun itDoesSomething(elements: List<String>): List<Pair<String, Int>> {

    return elements.groupBy {
        it
    }.map {
        Pair(it.key, it.value.count())
    }
}
```

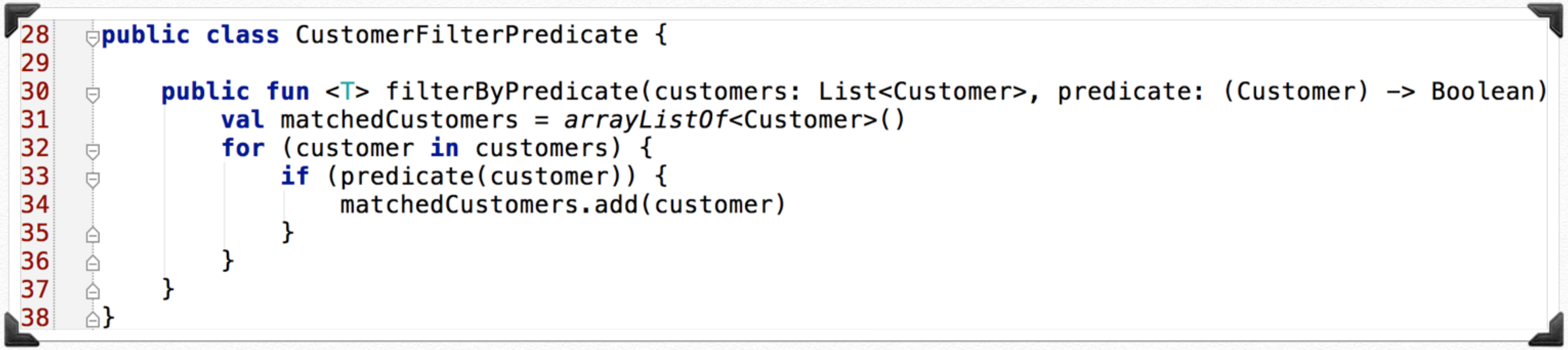
Applying these techniques to common scenarios

Customer Search



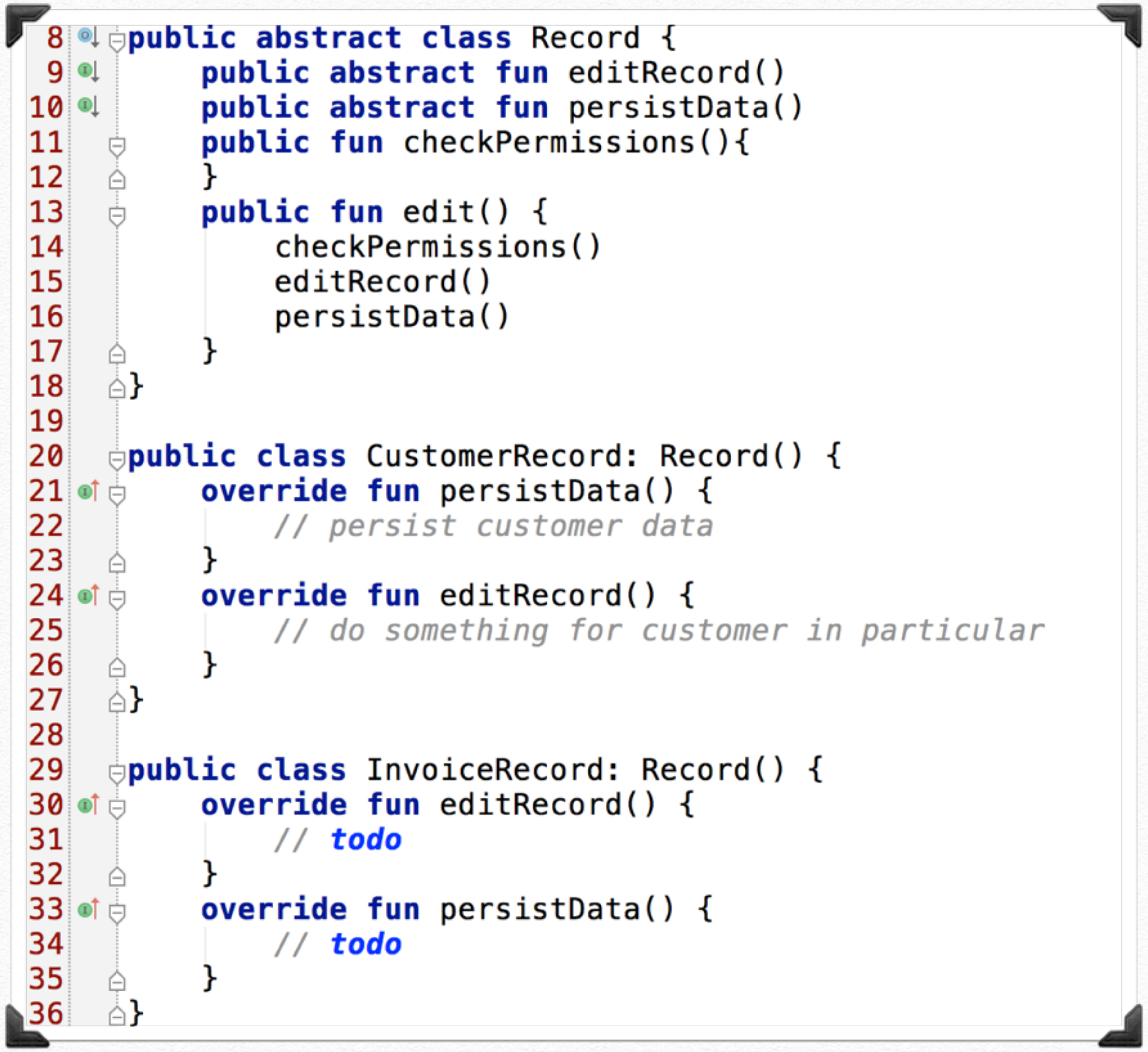
```kotlin
public class CustomerFilter {

    public fun <T> filterByName(customers: List<Customer>, value: String) {
        val matchedCustomers = arrayListOf<Customer>()
        for (customer in customers) {
            if (customer.name == value) {
                matchedCustomers.add(customer)
            }
        }
    }

    public fun <T> filterByCountry(customers: List<Customer>, value: String) {
        val matchedCustomers = arrayListOf<Customer>()
        for (customer in customers) {
            if (customer.country == value) {
                matchedCustomers.add(customer)
            }
        }
    }
}
```

Customer Search

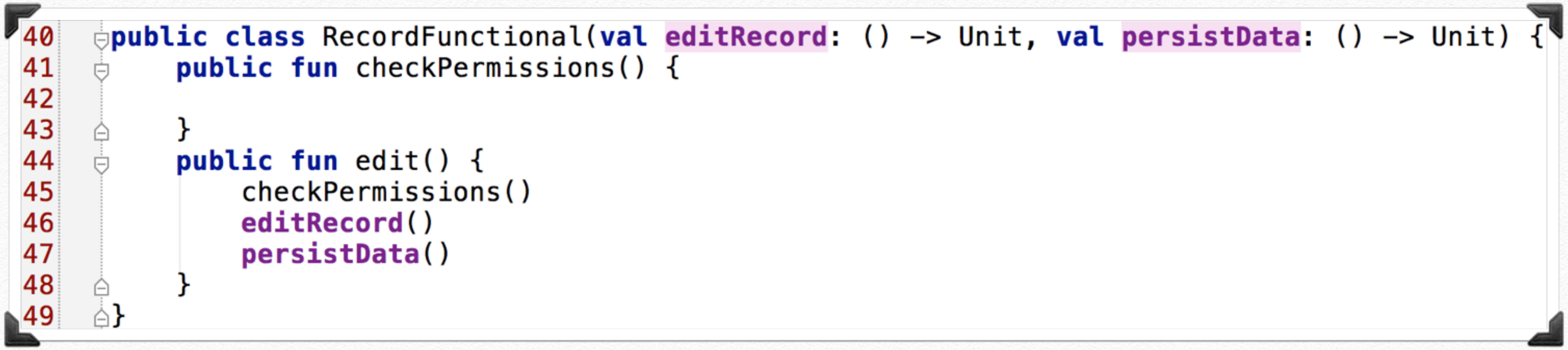


```
public class CustomerFilterPredicate {

    public fun <T> filterByPredicate(customers: List<Customer>, predicate: (Customer) -> Boolean)
        val matchedCustomers = arrayListOf<Customer>()
        for (customer in customers) {
            if (predicate(customer)) {
                matchedCustomers.add(customer)
            }
        }
    }
}
```

Templates Patterns

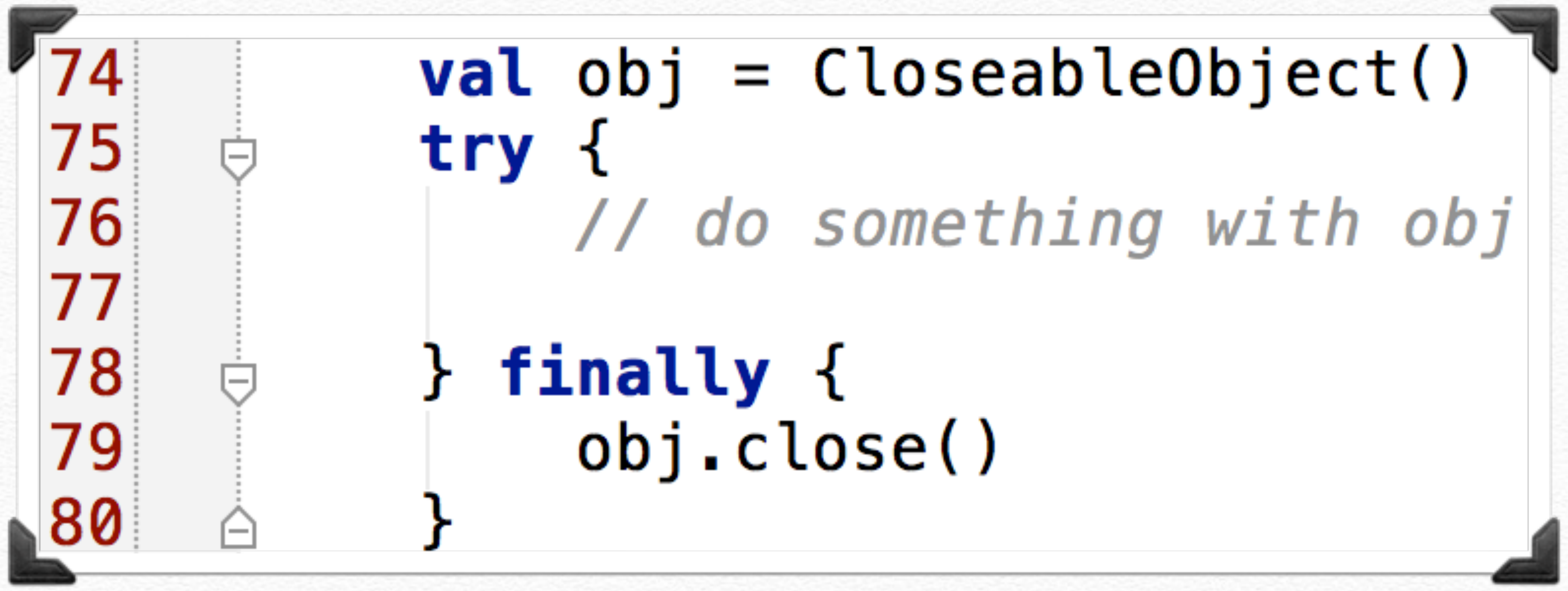

```
public abstract class Record {
    public abstract fun editRecord()
    public abstract fun persistData()
    public fun checkPermissions(){
    }
    public fun edit() {
        checkPermissions()
        editRecord()
        persistData()
    }
}

public class CustomerRecord: Record() {
    override fun persistData() {
        // persist customer data
    }
    override fun editRecord() {
        // do something for customer in particular
    }
}

public class InvoiceRecord: Record() {
    override fun editRecord() {
        // todo
    }
    override fun persistData() {
        // todo
    }
}
```

Less boilerplate



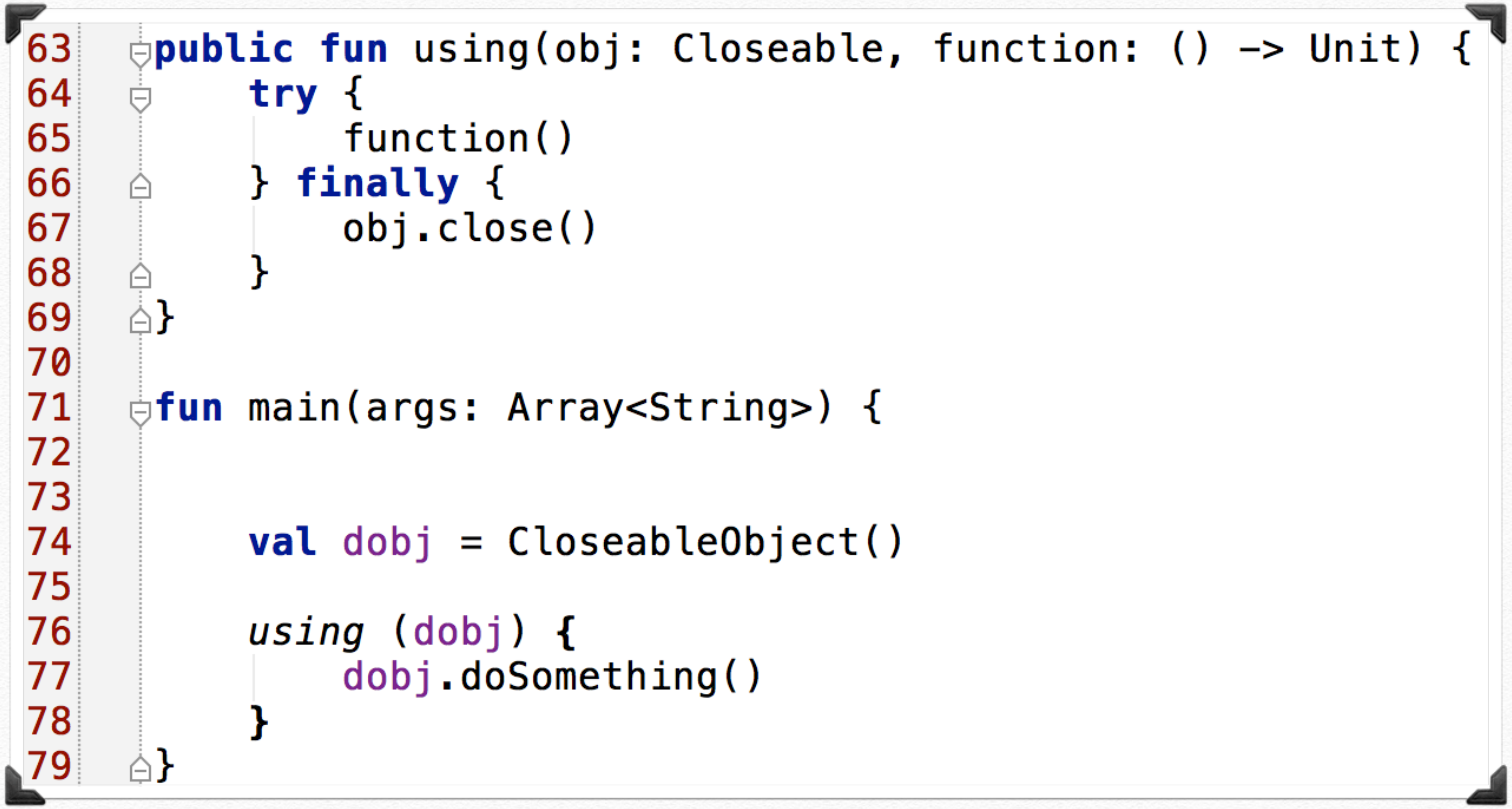
```kotlin
public class RecordFunctional(val editRecord: () -> Unit, val persistData: () -> Unit) {
    public fun checkPermissions() {

    }
    public fun edit() {
        checkPermissions()
        editRecord()
        persistData()
    }
}
```

Guaranteeing things take place



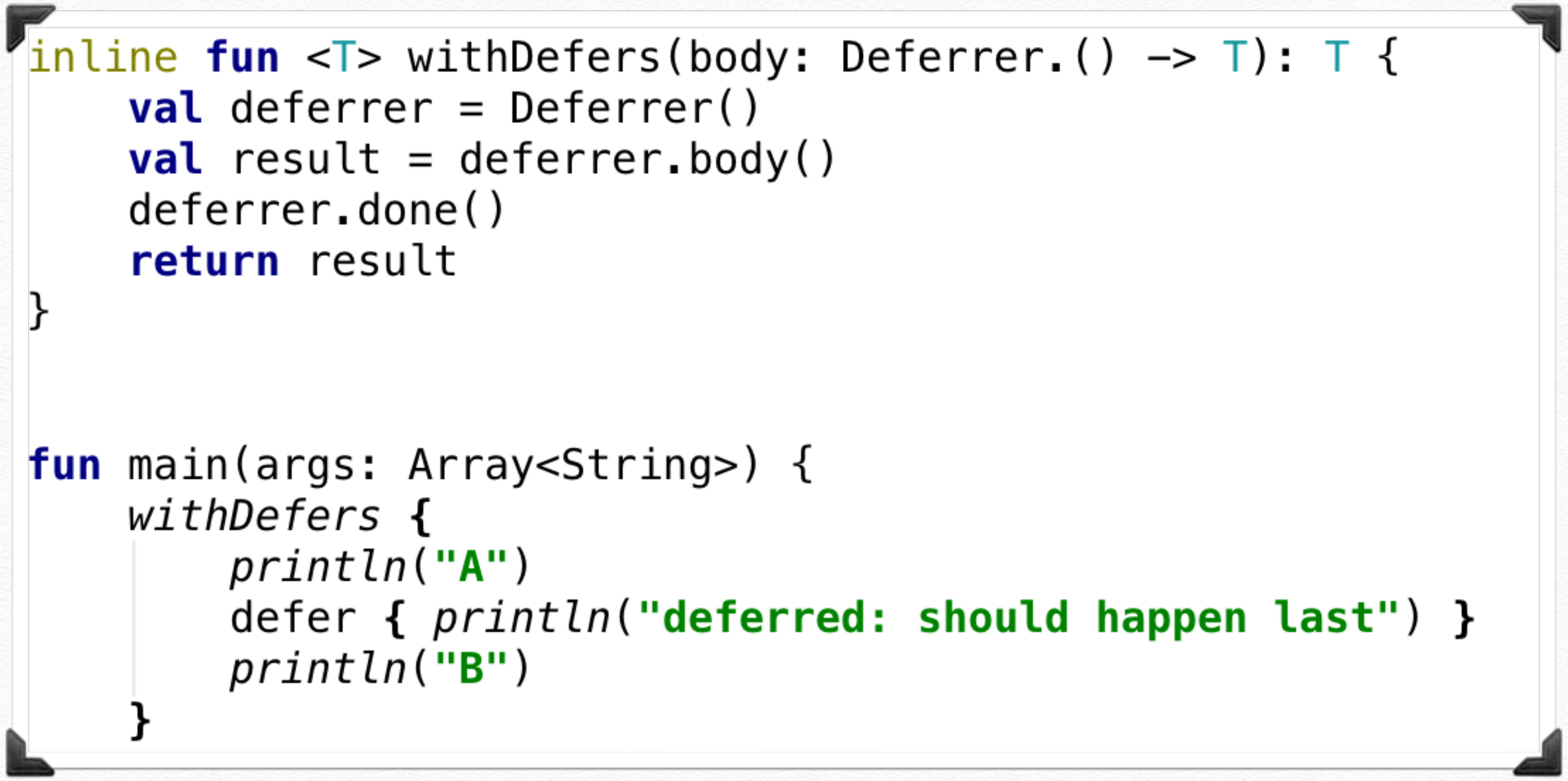
```
val obj = CloseableObject()
try {
    // do something with obj

} finally {
    obj.close()
}
```

Guaranteeing things take place



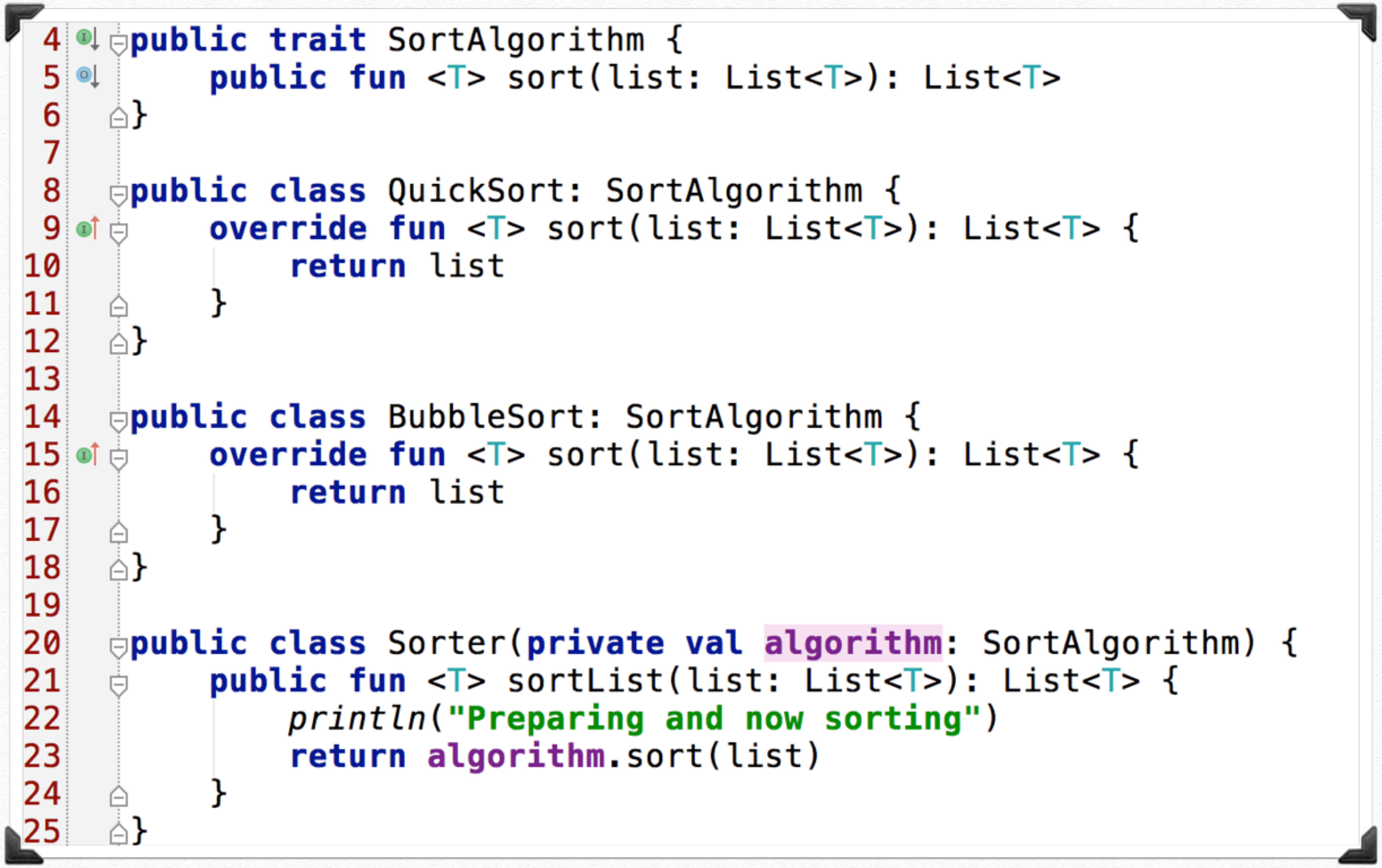
```kotlin
public fun using(obj: Closeable, function: () -> Unit) {
    try {
        function()
    } finally {
        obj.close()
    }
}

fun main(args: Array<String>) {


    val dobj = CloseableObject()

    using (dobj) {
        dobj.doSomething()
    }
}
```

Or when they take place



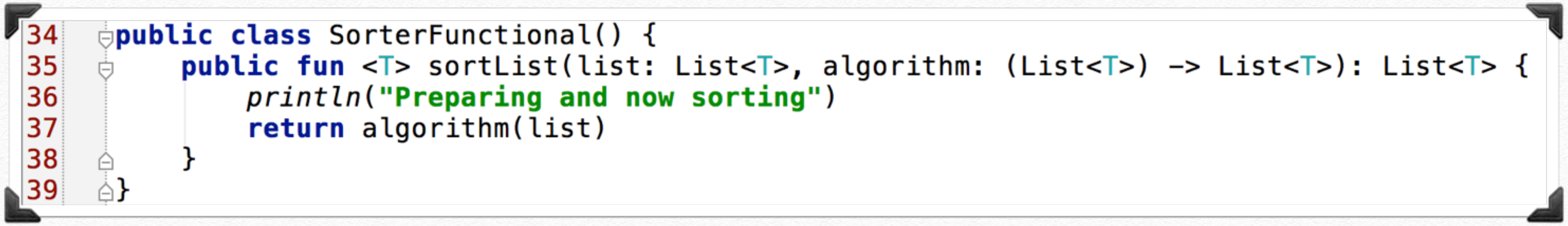
```kotlin
inline fun <T> withDefers(body: Deferrer.() -> T): T {
    val deferrer = Deferrer()
    val result = deferrer.body()
    deferrer.done()
    return result
}


fun main(args: Array<String>) {
    withDefers {
        println("A")
        defer { println("deferred: should happen last") }
        println("B")
    }
```

Strategy Pattern



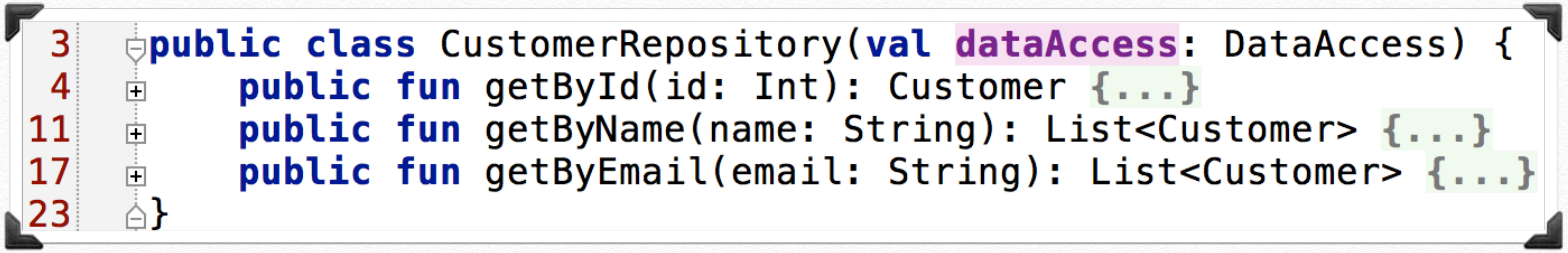
```kotlin
public trait SortAlgorithm {
    public fun <T> sort(list: List<T>): List<T>
}

public class QuickSort: SortAlgorithm {
    override fun <T> sort(list: List<T>): List<T> {
        return list
    }
}

public class BubbleSort: SortAlgorithm {
    override fun <T> sort(list: List<T>): List<T> {
        return list
    }
}

public class Sorter(private val algorithm: SortAlgorithm) {
    public fun <T> sortList(list: List<T>): List<T> {
        println("Preparing and now sorting")
        return algorithm.sort(list)
    }
}
```

Same strategy, less code



```
public class SorterFunctional() {
    public fun <T> sortList(list: List<T>, algorithm: (List<T>) -> List<T>): List<T> {
        println("Preparing and now sorting")
        return algorithm(list)
    }
}
```

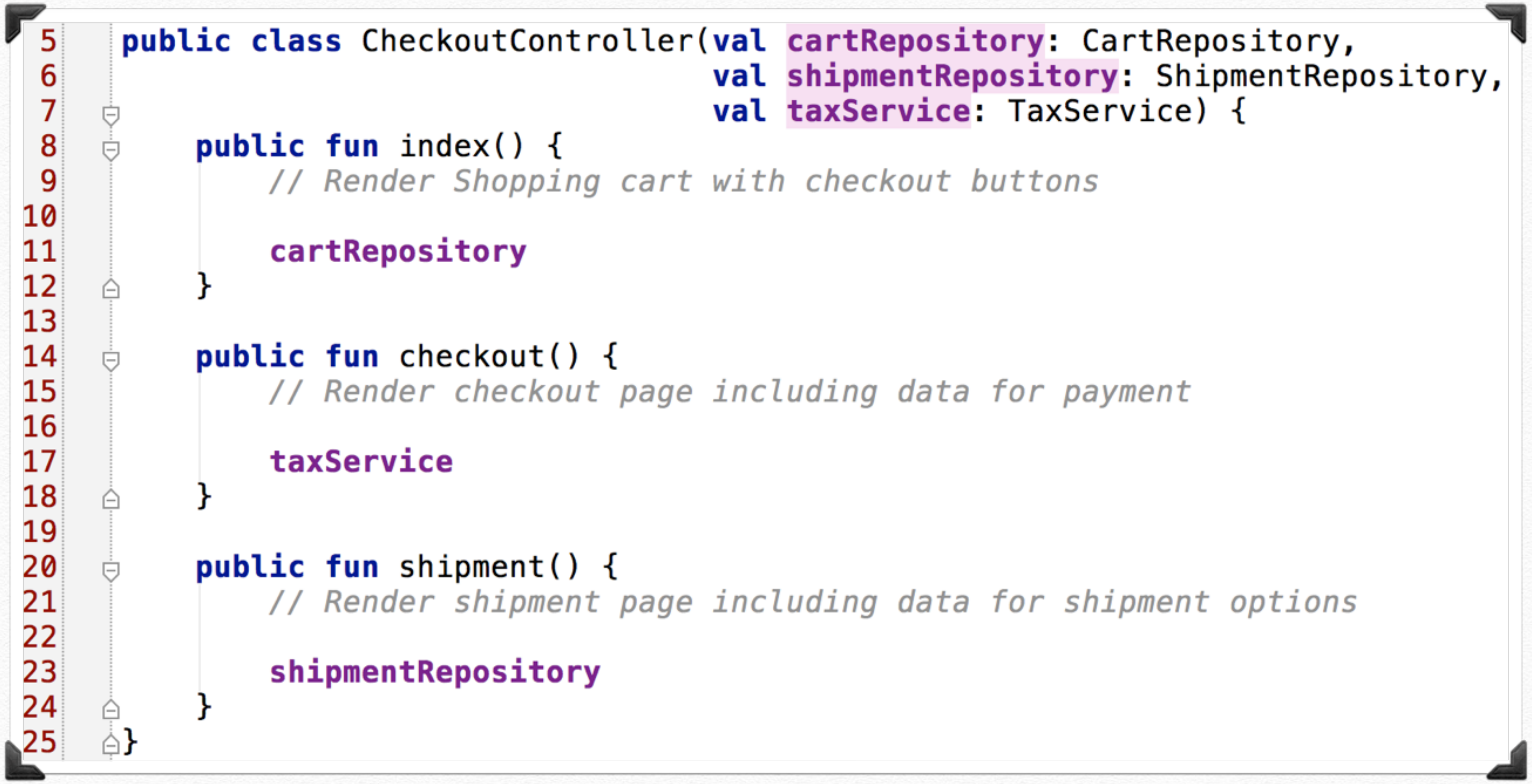
What about dependencies?



```kotlin
public class CustomerRepository(val dataAccess: DataAccess) {
    public fun getById(id: Int): Customer {...}
    public fun getByName(name: String): List<Customer> {...}
    public fun getByEmail(email: String): List<Customer> {...}
}
```

Let's think about dependencies in classes…

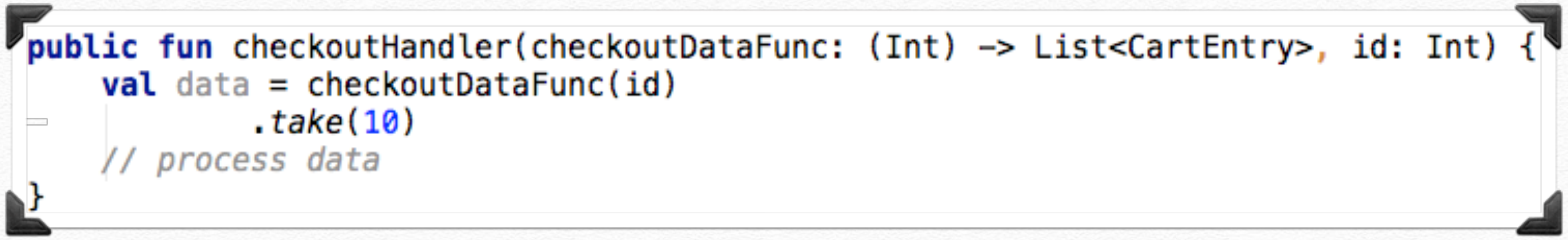
When dependencies grow



```kotlin
public class CheckoutController(val cartRepository: CartRepository,
                                val shipmentRepository: ShipmentRepository,
                                val taxService: TaxService) {
    public fun index() {
        // Render Shopping cart with checkout buttons

        cartRepository
    }

    public fun checkout() {
        // Render checkout page including data for payment

        taxService
    }

    public fun shipment() {
        // Render shipment page including data for shipment options

        shipmentRepository
    }
}
```

How do we deal with dependencies?

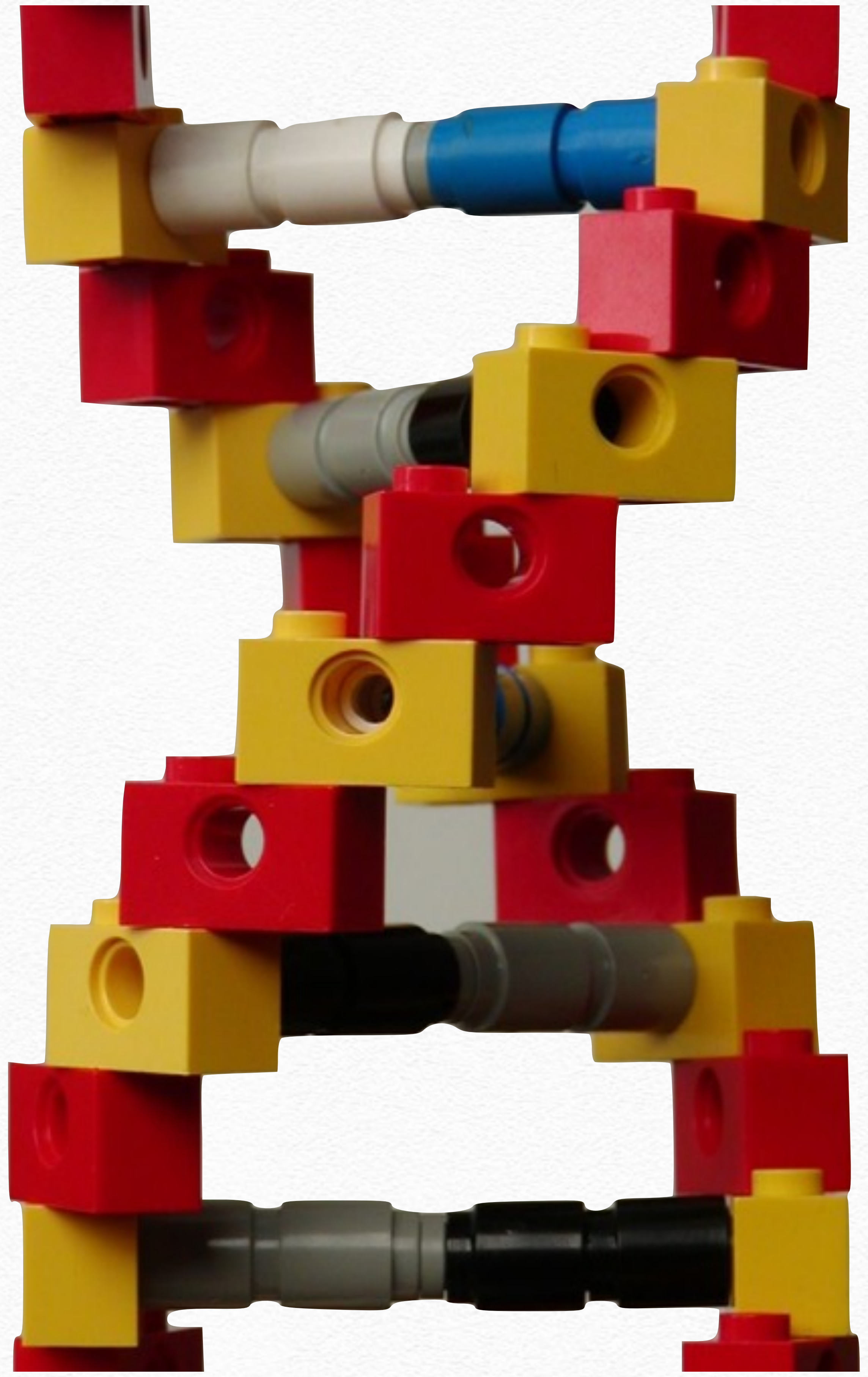
We pass in the behaviour



```
public fun checkoutHandler(checkoutDataFunc: (Int) -> List<CartEntry>, id: Int) {
    val data = checkoutDataFunc(id)
            .take(10)
    // process data
}
```

Thinking Functionally

Functions are our primitives



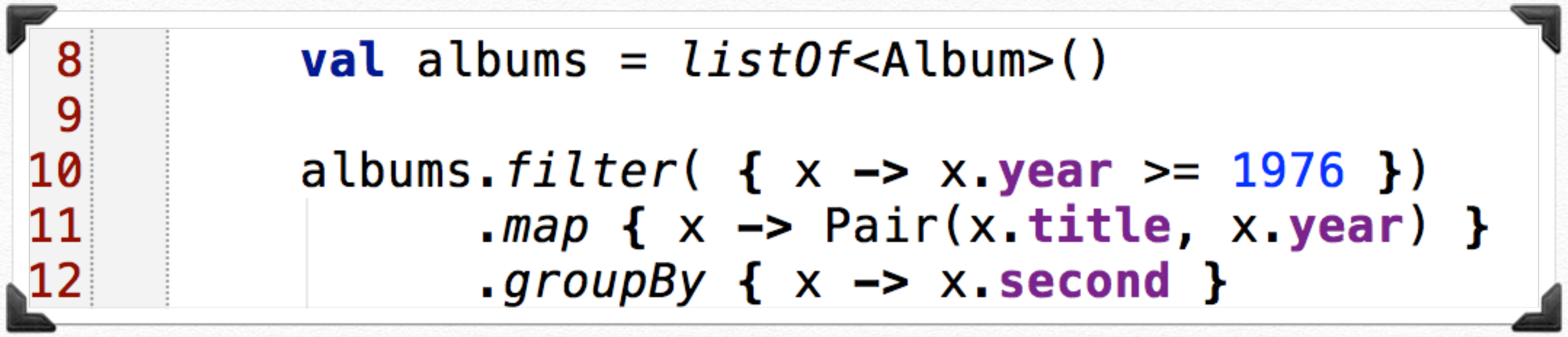
http://www.flickr.com/photos/mknowles/47457221/

We’re merely combining functions

Functions taking functions

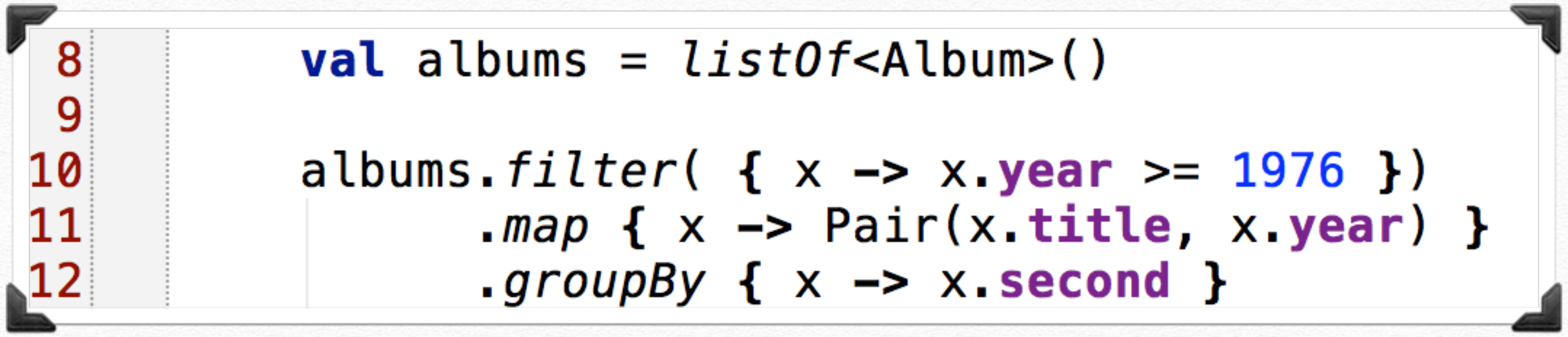
- filter - takes a predicate
- map - takes a mapping function
- groupBy - takes a key lookup function

Functions returning functions



```
val albums = listOf<Album>()

albums.filter( { x -> x.year >= 1976 })
        .map { x -> Pair(x.title, x.year) }
        .groupBy { x -> x.second }
```

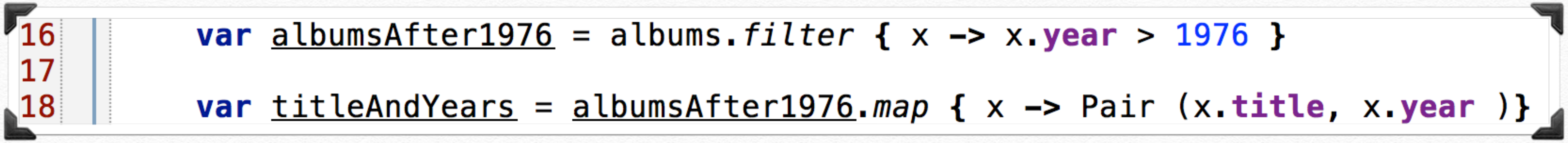
Pipelining



```
val albums = listOf<Album>()

albums.filter( { x -> x.year >= 1976 })
        .map { x -> Pair(x.title, x.year) }
        .groupBy { x -> x.second }
```

Keeping it readable

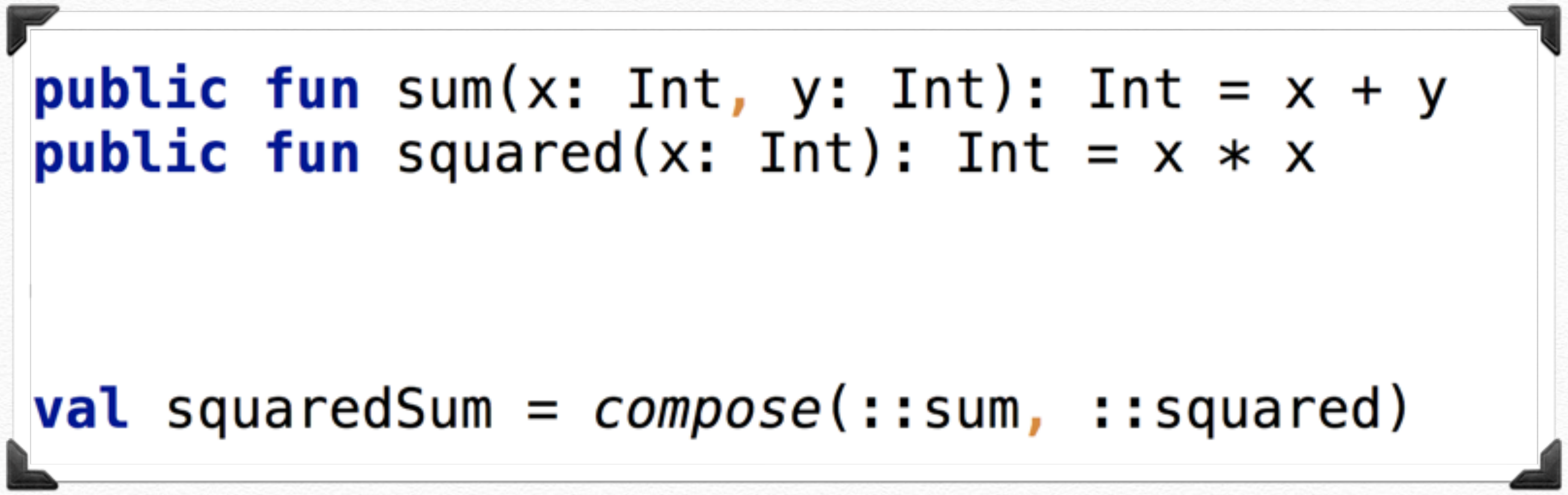
- Abstract multiple pipeline calls into descriptive functions
- Intermediate variables don't always help



```
    var albumsAfter1976 = albums.filter { x -> x.year > 1976 }

    var titleAndYears = albumsAfter1976.map { x -> Pair (x.title, x.year )}
```

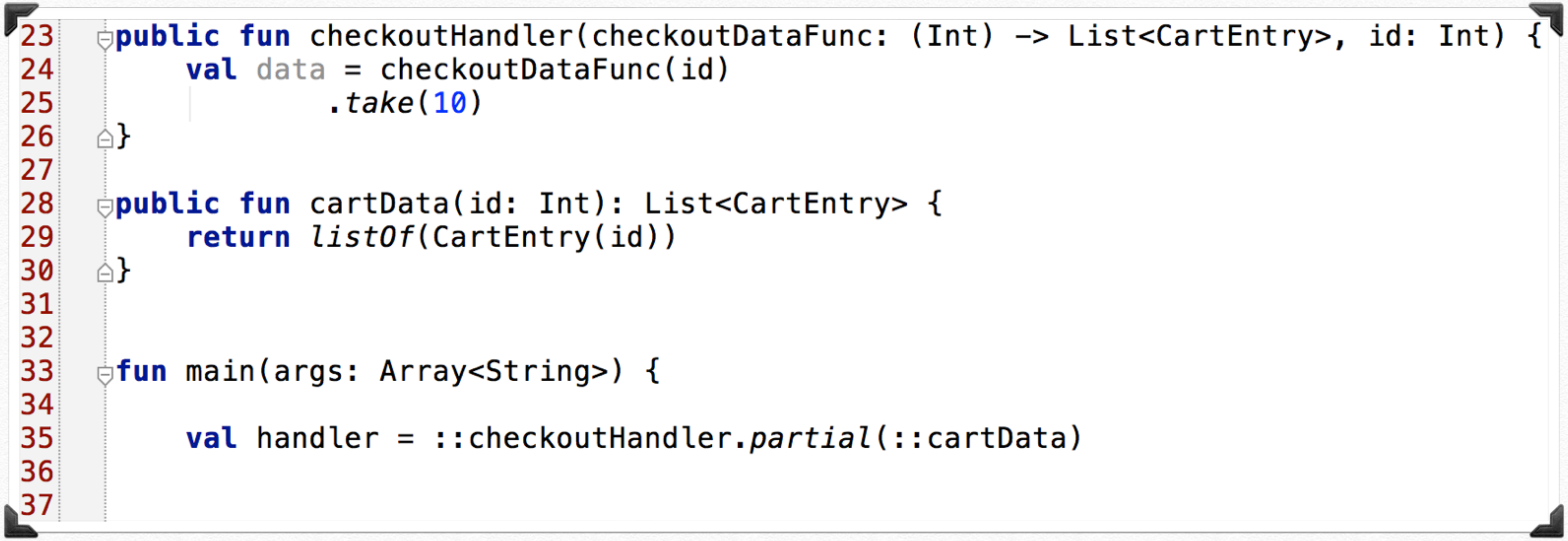
How do we re-use?

Composing functions



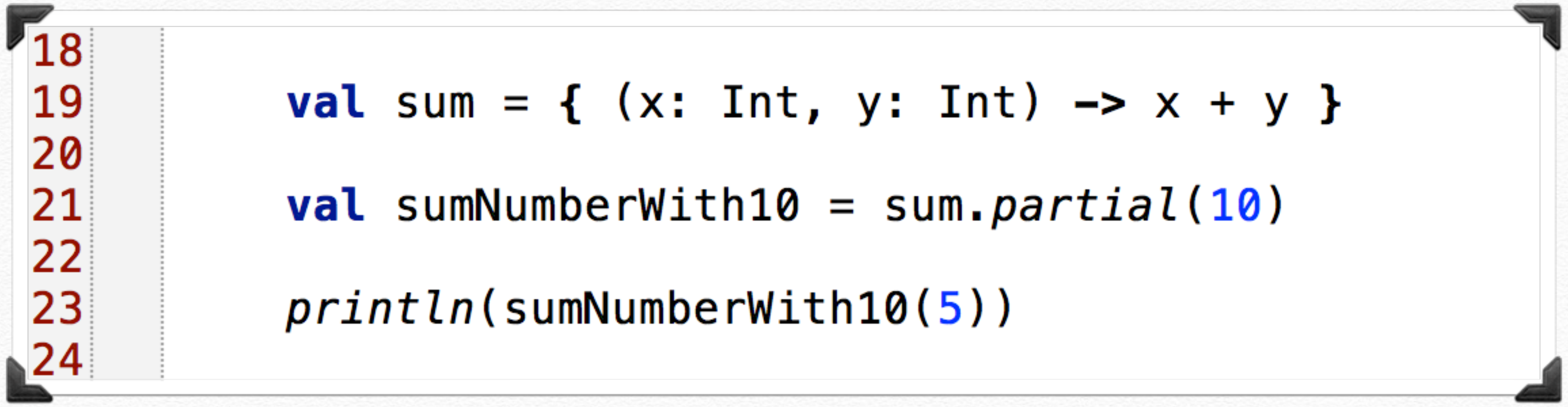
```
public fun sum(x: Int, y: Int): Int = x + y
public fun squared(x: Int): Int = x * x



val squaredSum = compose(::sum, ::squared)
```

Partial Function Applications



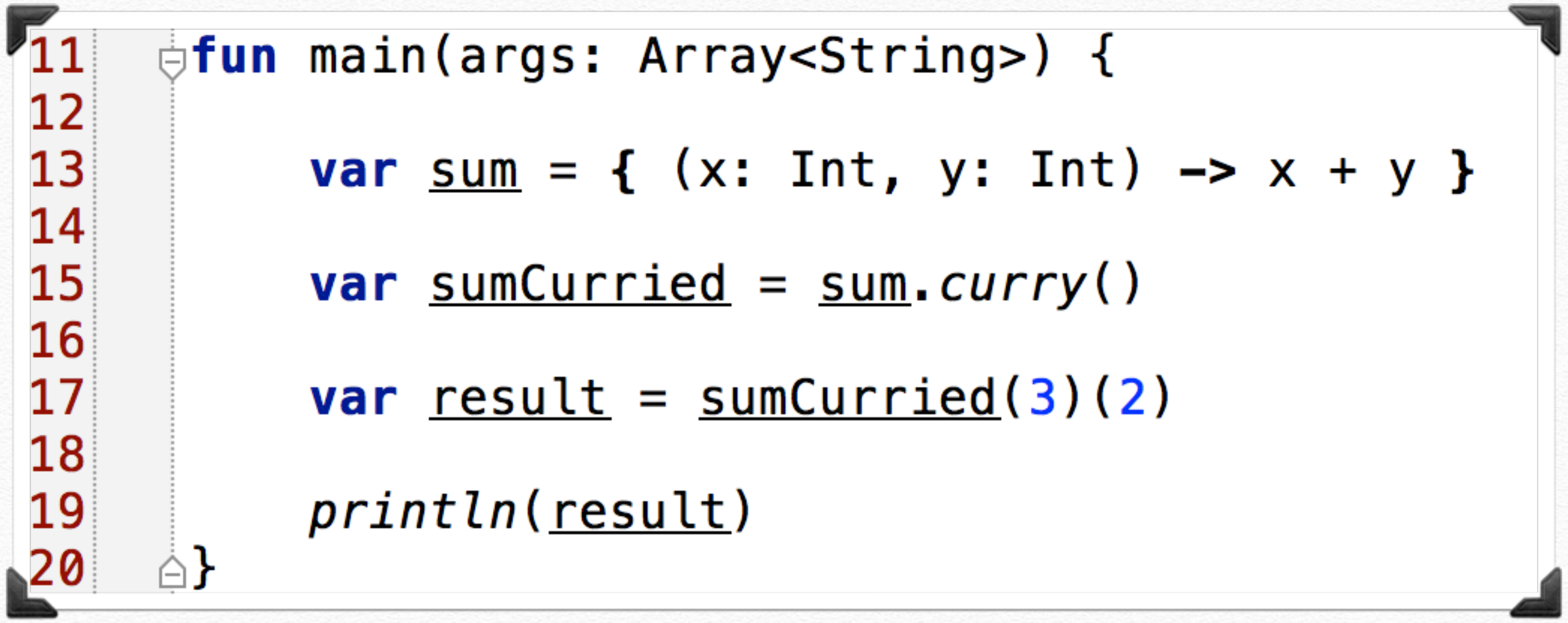
```
public fun checkoutHandler(checkoutDataFunc: (Int) -> List<CartEntry>, id: Int) {
    val data = checkoutDataFunc(id)
            .take(10)
}

public fun cartData(id: Int): List<CartEntry> {
    return listOf(CartEntry(id))
}


fun main(args: Array<String>) {

    val handler = ::checkoutHandler.partial(::cartData)

```

Partial Function Applications



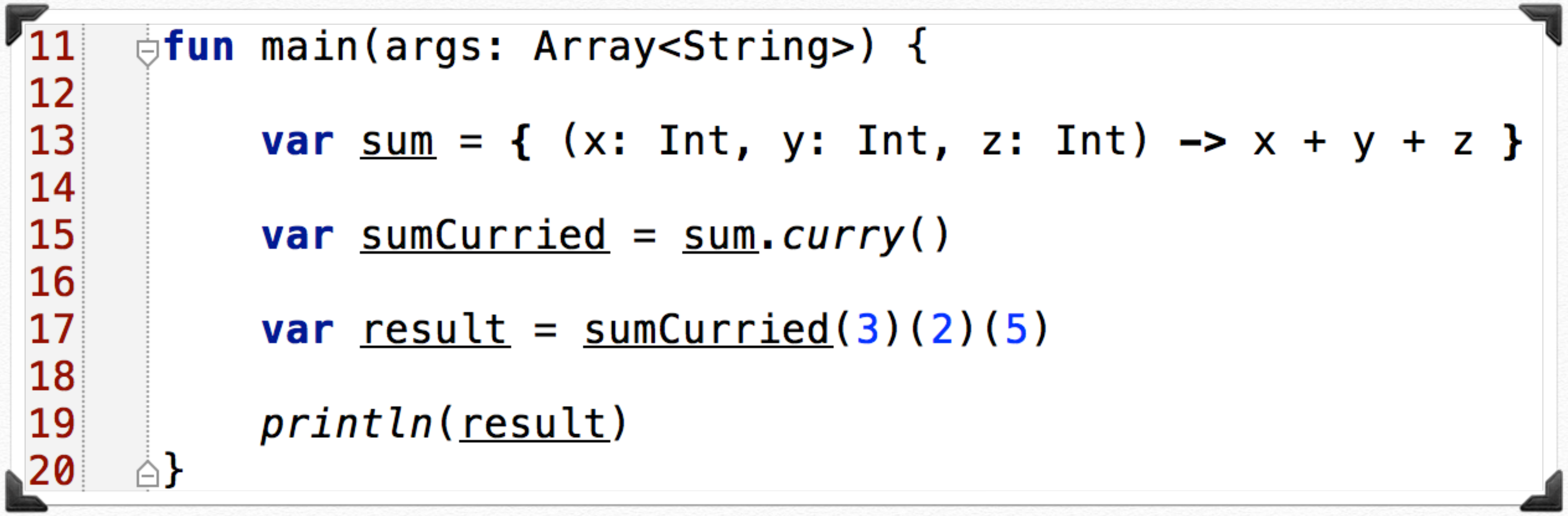
```
    val sum = { (x: Int, y: Int) -> x + y }

    val sumNumberWith10 = sum.partial(10)

    println(sumNumberWith10(5))

```

Currying



```
fun main(args: Array<String>) {

    var sum = { (x: Int, y: Int) -> x + y }

    var sumCurried = sum.curry()

    var result = sumCurried(3)(2)

    println(result)
}
```

Currying



```
fun main(args: Array<String>) {

    var sum = { (x: Int, y: Int, z: Int) -> x + y + z }

    var sumCurried = sum.curry()

    var result = sumCurried(3)(2)(5)

    println(result)
}
```

And then there's the data…

We're taking data and processing it in stages

What is our data?

Customer

Invoices

Point

Age

And what is this data?

- Scalars: Age, Date of Birth, Amount
- Collections of ATD's: List of Customers, List of Invoices, Set of Pairs

Take as input lists and output lists

Common List Operations

- map, flatMap, filter, findAll, merge, zip…..

And scalars can be obtained from lists…

Common Scalar Operations

- first, last, find, aggregate.….

Many functions boil down to folds

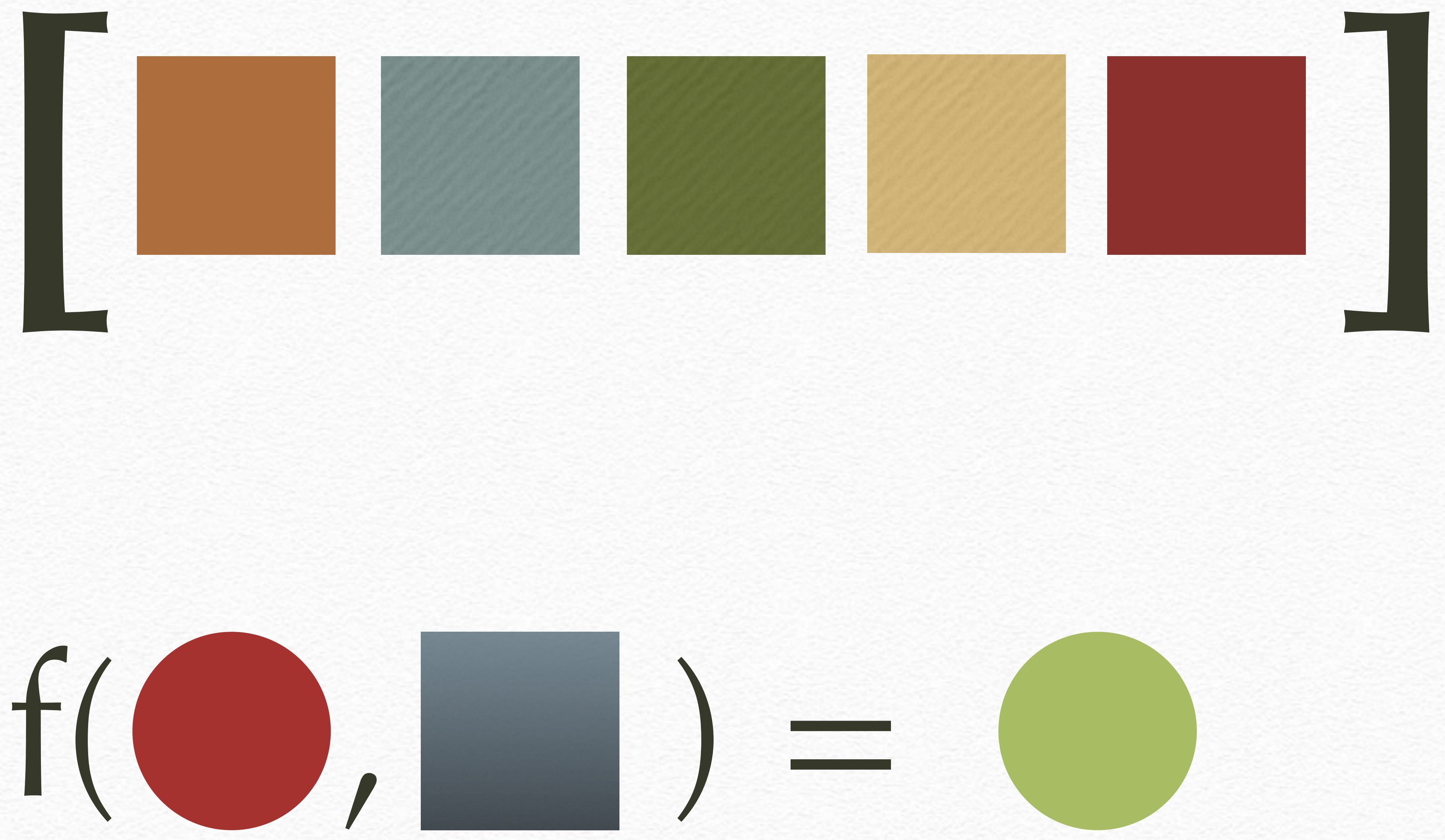
- fold, reduce, aggregate…same thing.

What is fold?

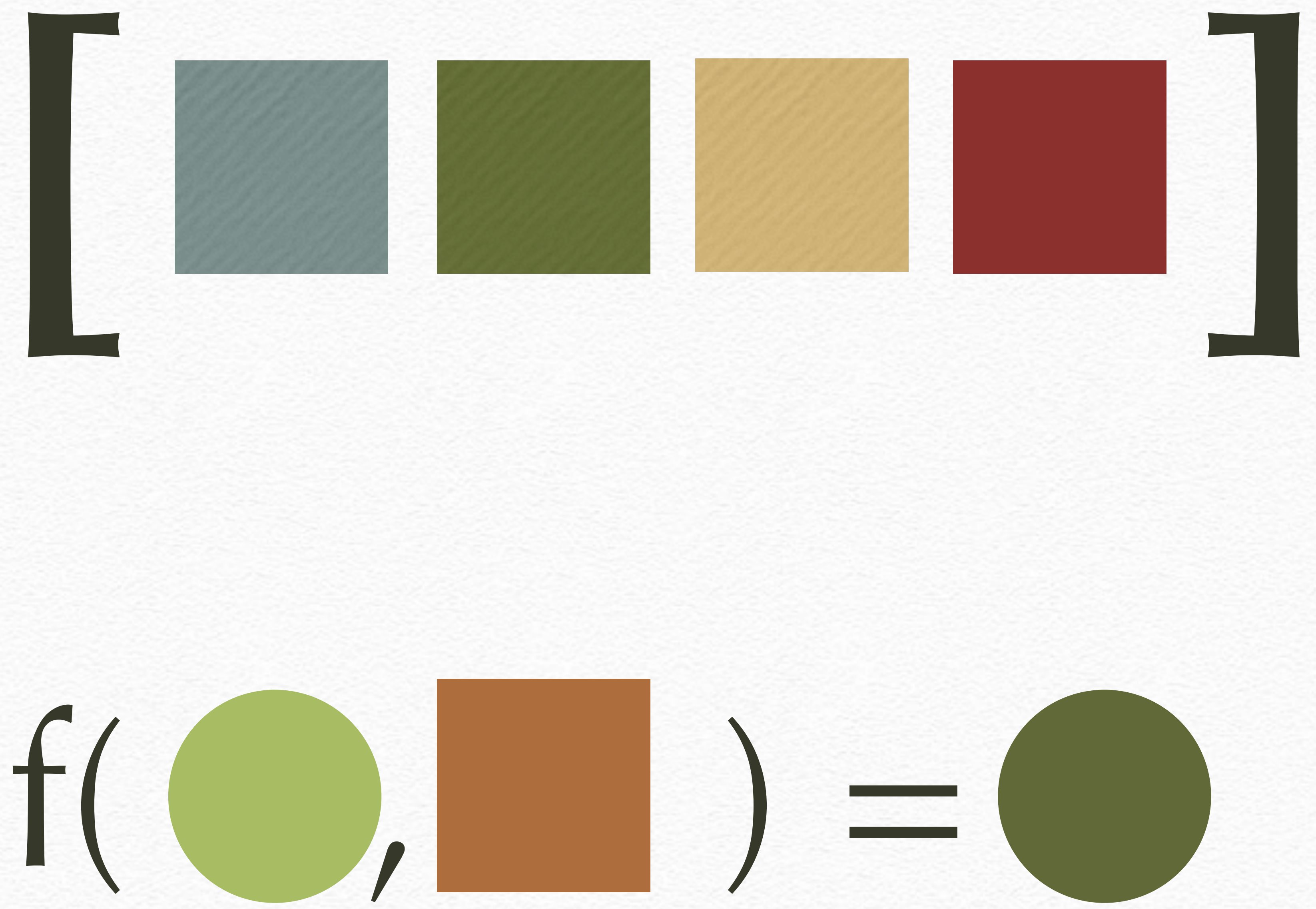
$f(x,y)$ where

x	accumulator
y	element in list

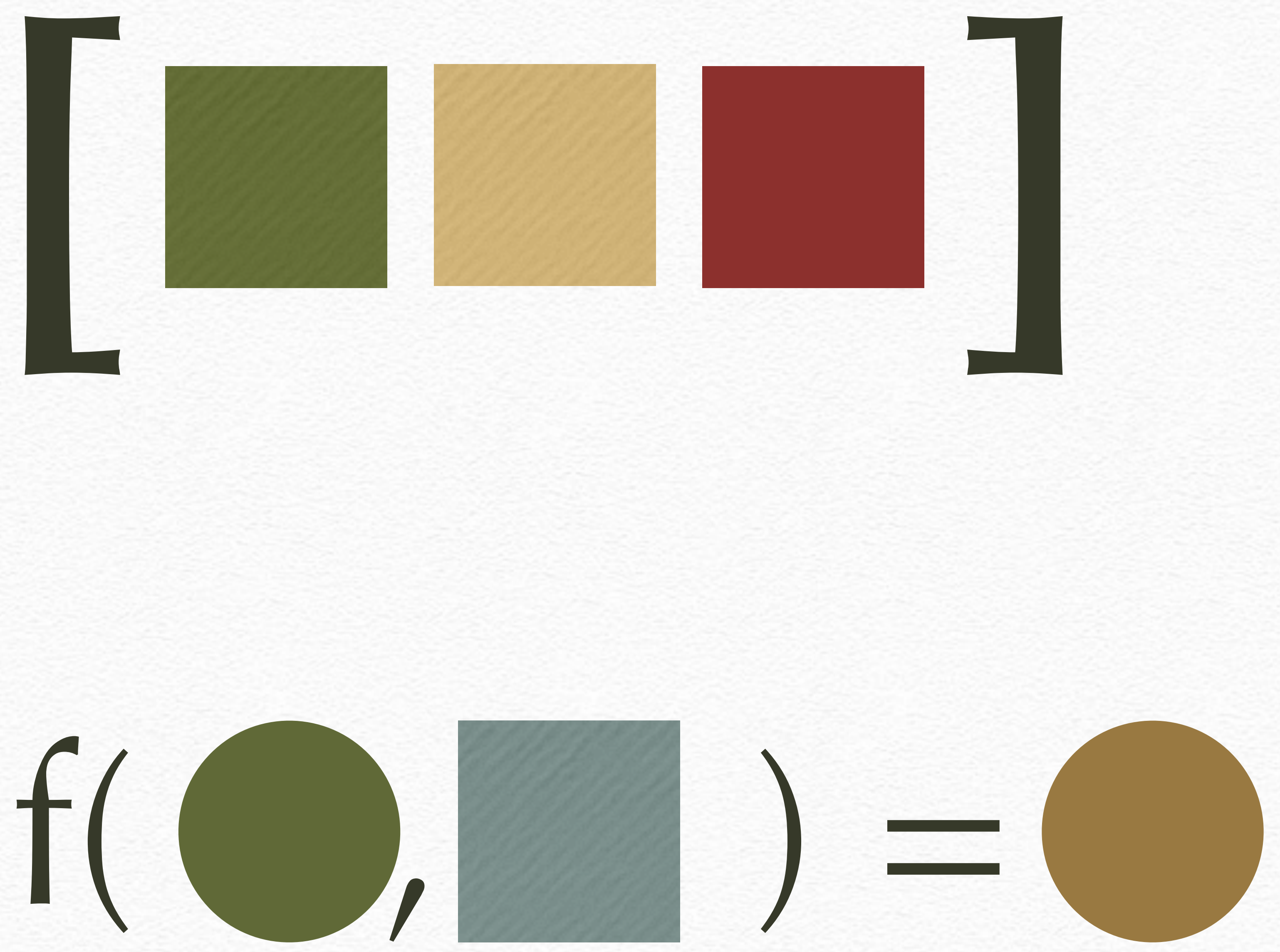
What is fold?



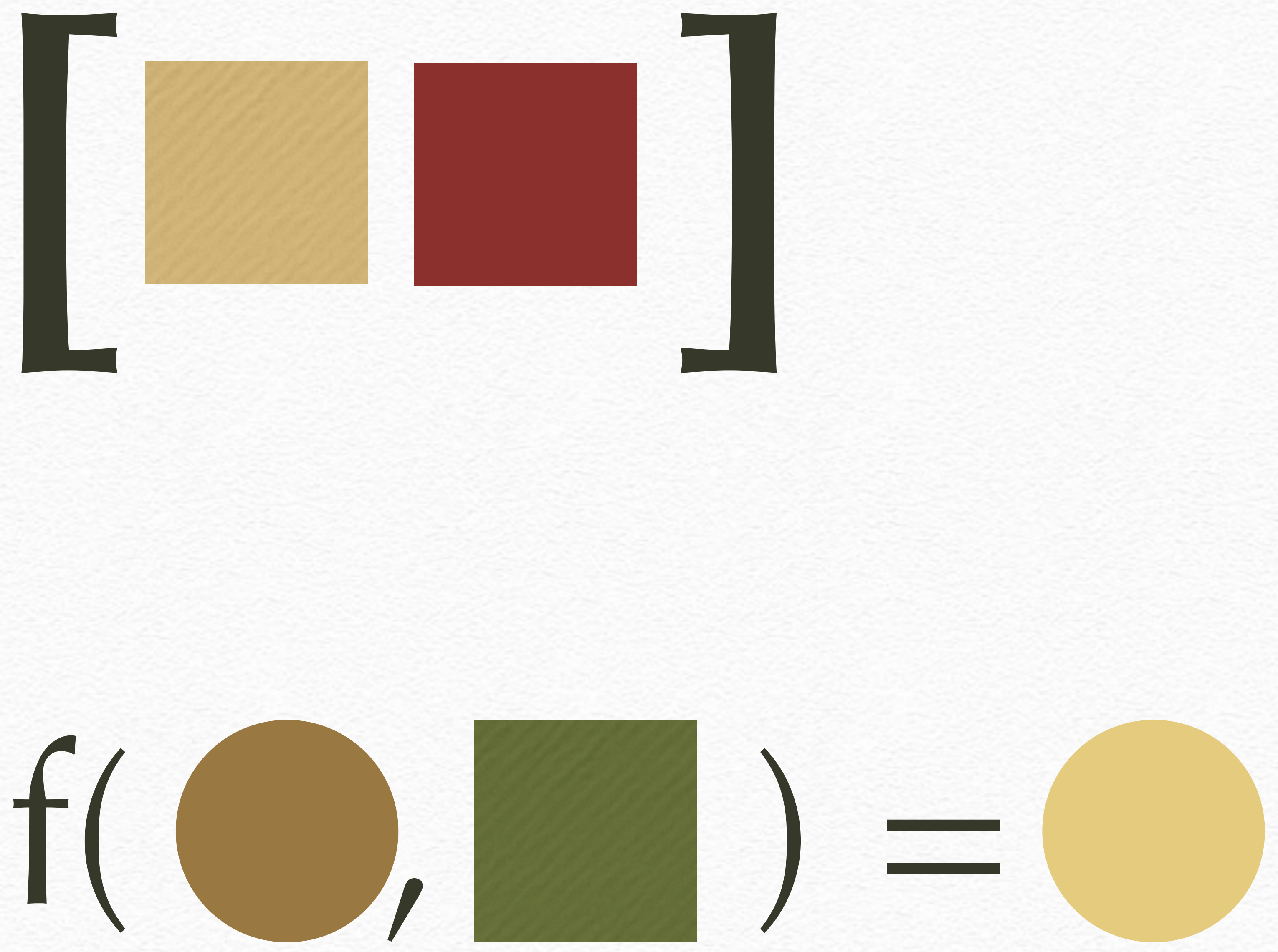
What is fold?



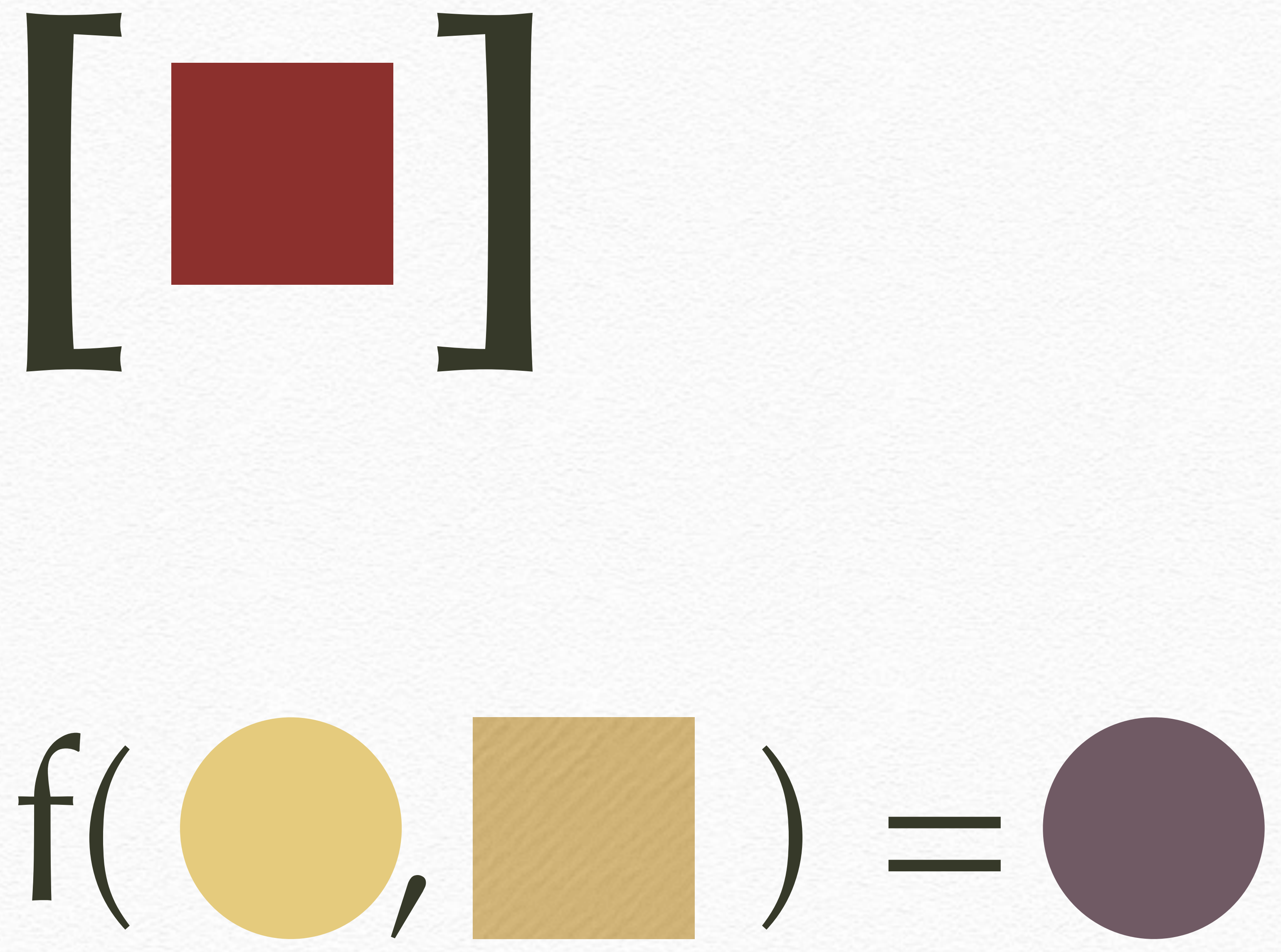
What is fold?



What is fold?



What is fold?



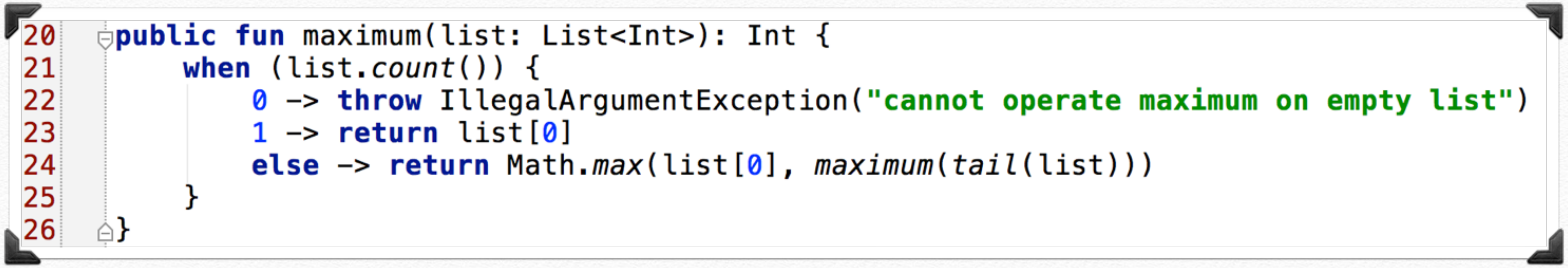
What is fold?

[ ]

f(●, ■) = ●

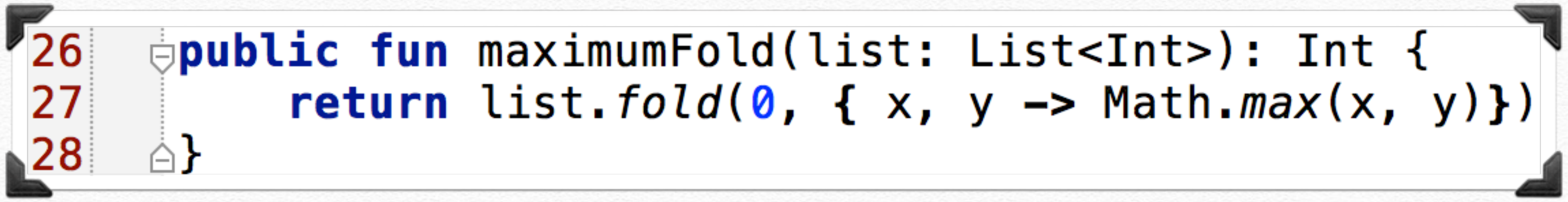
Recursion, Pattern Matching and Folds

The Recursive Maximum



```kotlin
public fun maximum(list: List<Int>): Int {
    when (list.count()) {
        0 -> throw IllegalArgumentException("cannot operate maximum on empty list")
        1 -> return list[0]
        else -> return Math.max(list[0], maximum(tail(list)))
    }
}
```

The Base is the accumulator



```
public fun maximumFold(list: List<Int>): Int {
    return list.fold(0, { x, y -> Math.max(x, y)})
}
```

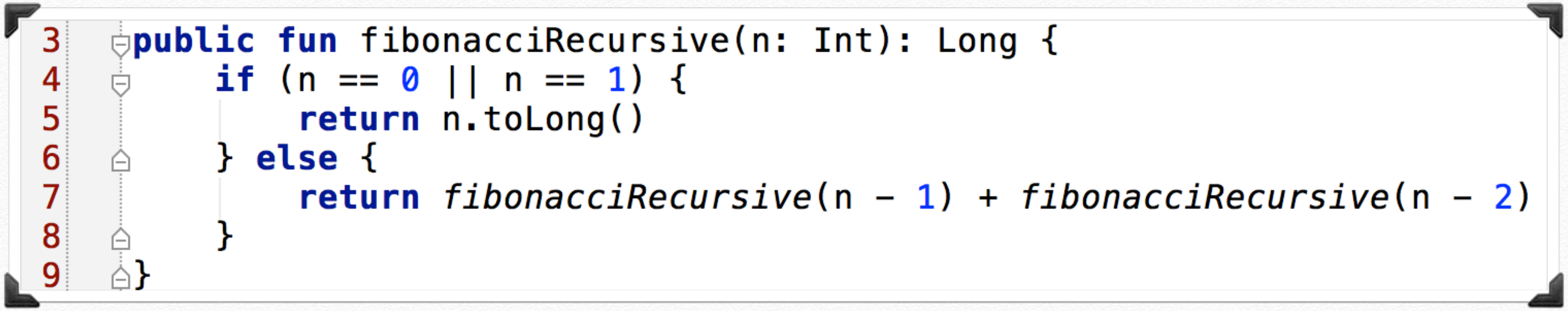
Immutability of the data

We need to avoid state

- Treat lists as infinite
  - Allows for lazy evaluation, reactive programming
- Aim for immutability
  - Create. Don't modify.
  - There's a difference between ReadOnly and Immutable

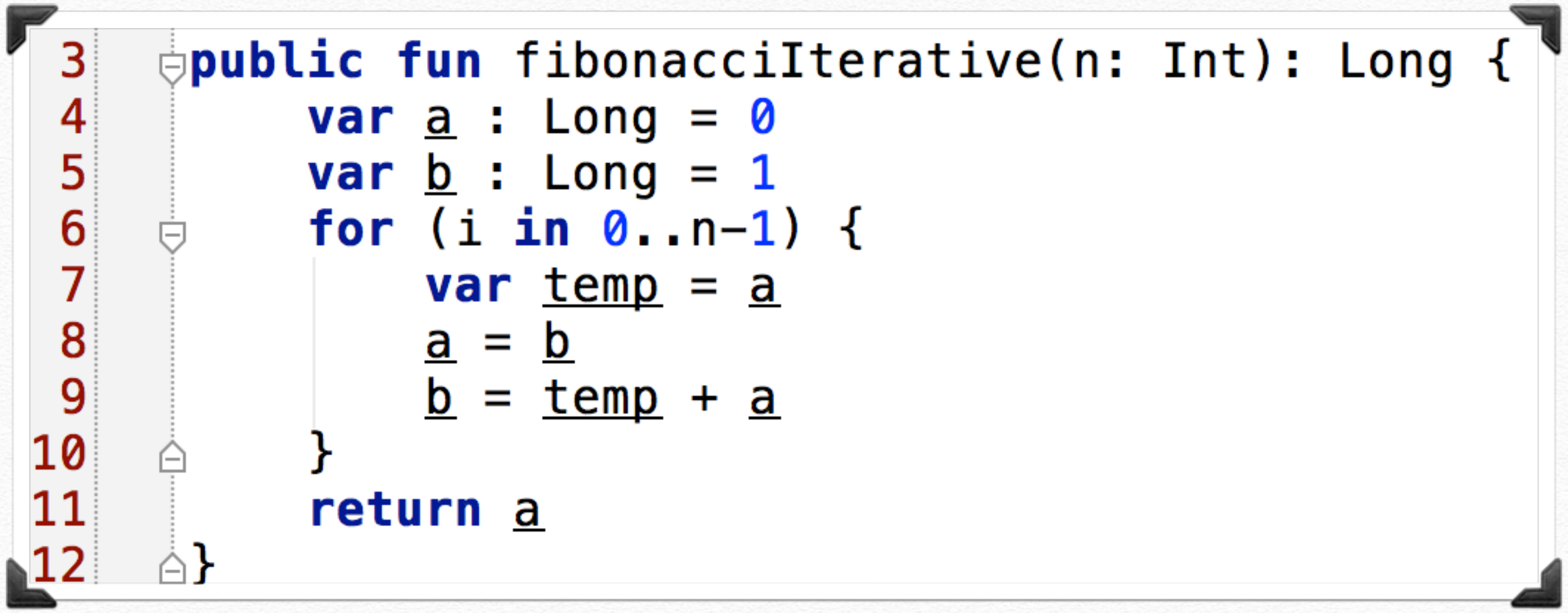
What about Performance?

That Annoying Fibonacci



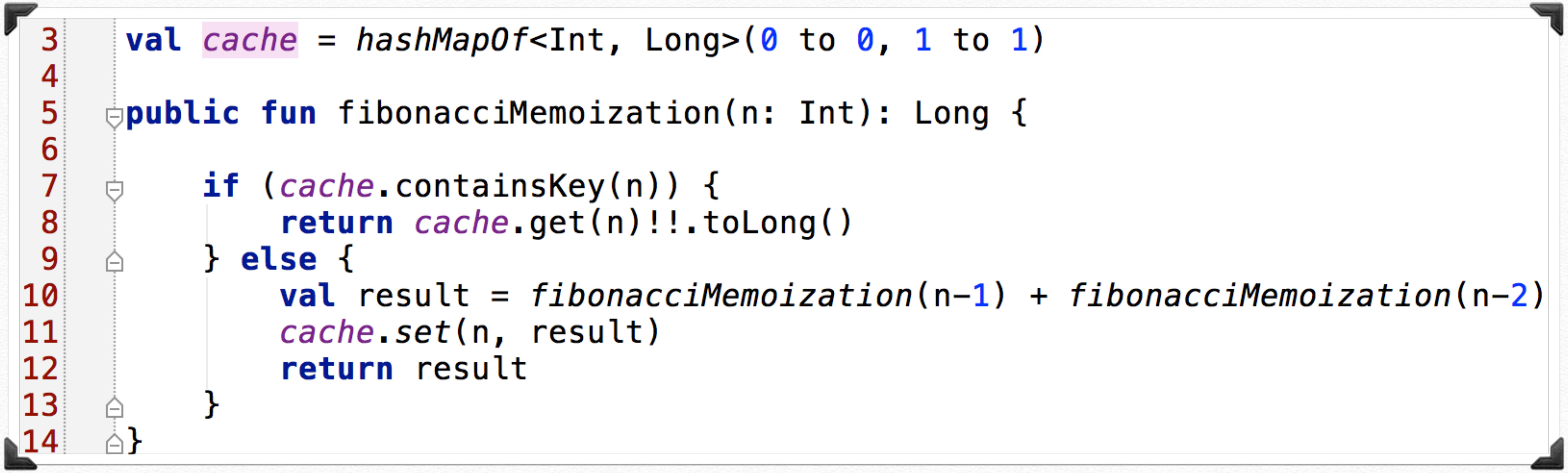
```kotlin
public fun fibonacciRecursive(n: Int): Long {
    if (n == 0 || n == 1) {
        return n.toLong()
    } else {
        return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2)
    }
}
```

The Iterative Approach



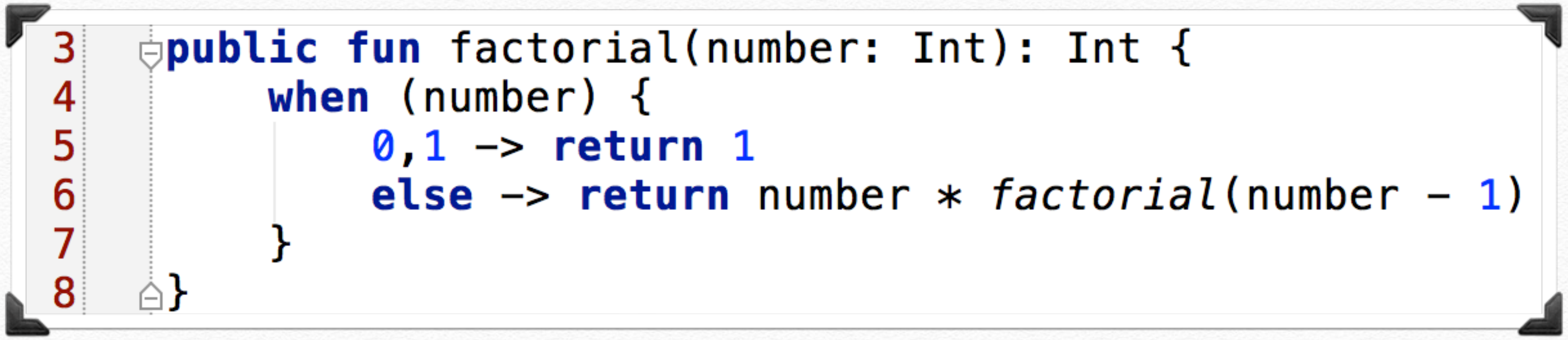
```kotlin
public fun fibonacciIterative(n: Int): Long {
    var a : Long = 0
    var b : Long = 1
    for (i in 0..n-1) {
        var temp = a
        a = b
        b = temp + a
    }
    return a
}
```

Memoization



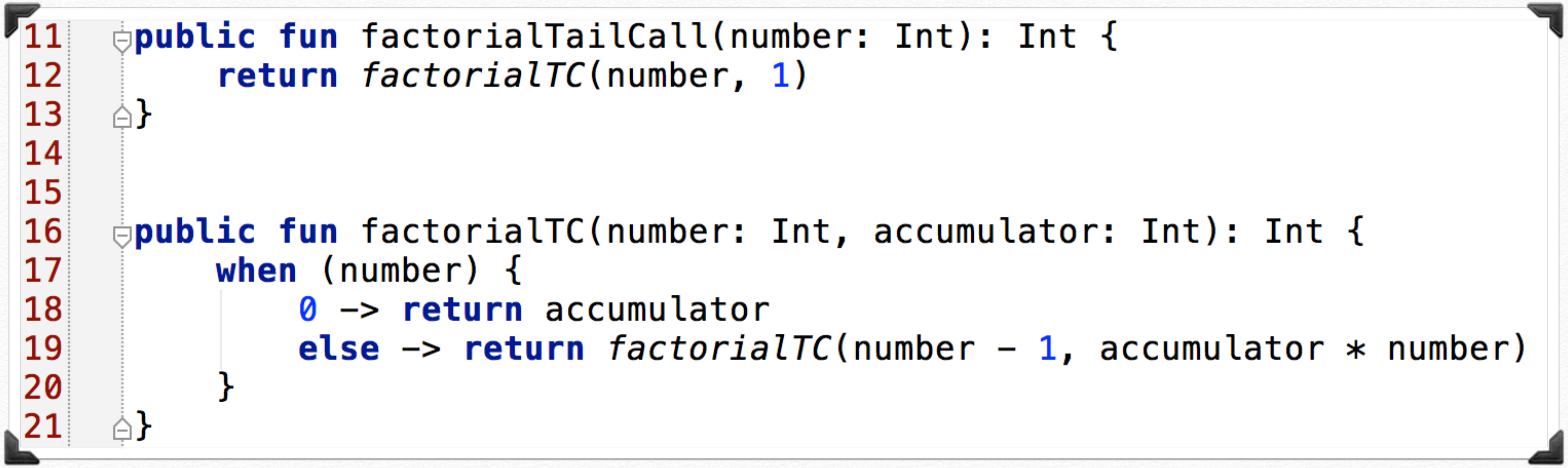
```kotlin
val cache = hashMapOf<Int, Long>(0 to 0, 1 to 1)

public fun fibonacciMemoization(n: Int): Long {

    if (cache.containsKey(n)) {
        return cache.get(n)!!.toLong()
    } else {
        val result = fibonacciMemoization(n-1) + fibonacciMemoization(n-2)
        cache.set(n, result)
        return result
    }
}
```

Recursion and Tail Calls



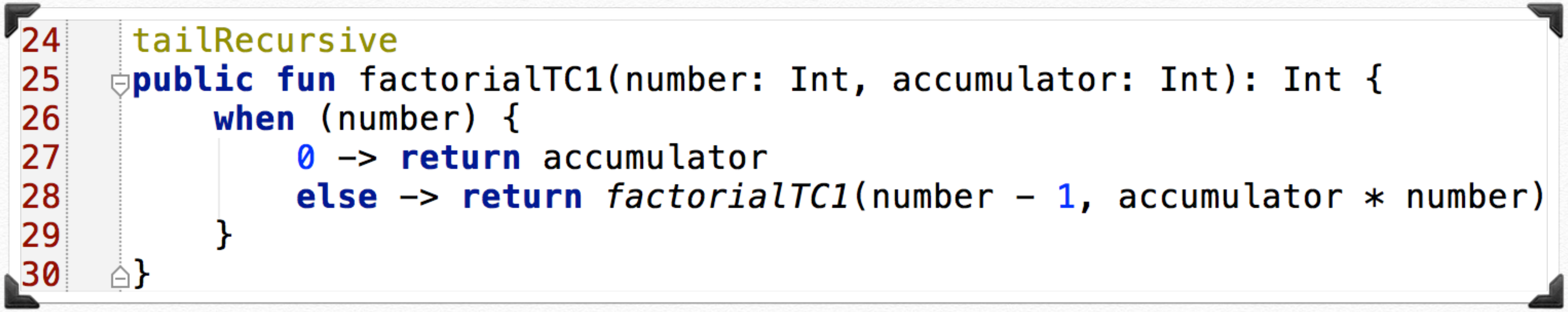
```
public fun factorial(number: Int): Int {
    when (number) {
        0,1 -> return 1
        else -> return number * factorial(number - 1)
    }
}
```

Converting to Tail Calls



```kotlin
public fun factorialTailCall(number: Int): Int {
    return factorialTC(number, 1)
}


public fun factorialTC(number: Int, accumulator: Int): Int {
    when (number) {
        0 -> return accumulator
        else -> return factorialTC(number - 1, accumulator * number)
    }
}
```

Tail Call Optimization



```
tailRecursive
public fun factorialTC1(number: Int, accumulator: Int): Int {
    when (number) {
        0 -> return accumulator
        else -> return factorialTC1(number - 1, accumulator * number)
    }
}
```

Inlining

- Higher-order functions can have performance impact
- Inlining functions can minimise performance impact

What about all that scary stuff?

Functors

- Collection of **a** whereby you can apply a function **a -> b**, returning collection of **b**
- ***map*** is a an example of Functor

Monads

- Abstract Data Type, following a series of rules
  - Maybe Monad -> Option
  - List Monad -> IEnumerable<T> (Iterable<T>)
  - Promises

In Summary

- Embrace Functions as Primitive Elements
- Focus on writing less your own code
- Focus on writing more descriptive code
- Focus on writing deterministic code
- Rocket Science

Recommended Books

- The Little Schemer
- Learn you a Haskell
- Functional Programming in Java

Thank you