Efficient Kotlin

Speed-up your Kotlin project

Marcin Moskała

About the trainer:

Marcin Moskała
@marcinmoskala
marcinmoskala@gmail.com

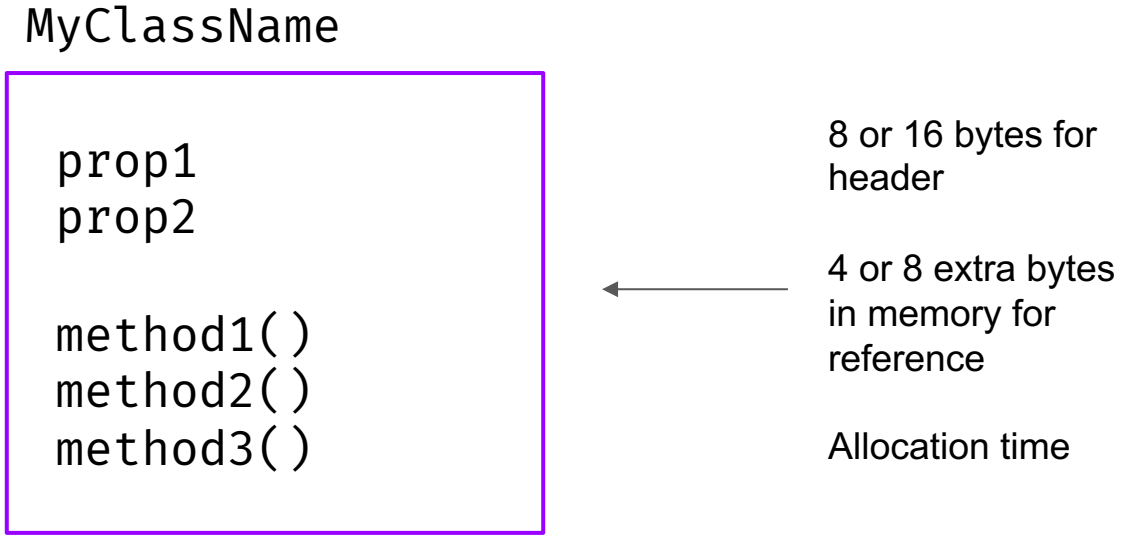
Trainer, consultant, author of
Effective Kotlin &
Android Development in Kotlin

Kt. Academy
www.kt.academy

Part 1: Good code
Chapter 1: Safety
Chapter 2: Readability
Part 2: Code design
Chapter 3: Reusability
Chapter 4: Abstractions design
Chapter 5: Objects creation
Chapter 6: Class design
Part 3: Efficiency
Chapter 7: Make it cheap
Chapter 8: Efficient collection processing

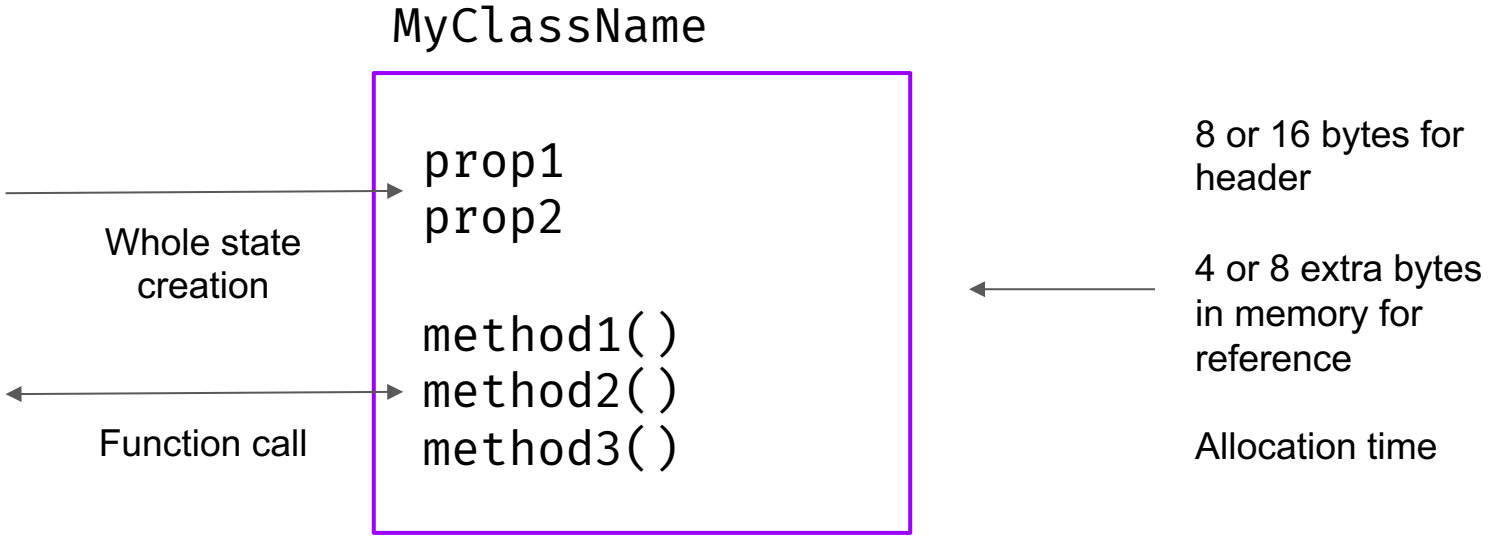
Chapter 7:
Make it cheap

Item 42: Avoid unnecessary object creation



```
class A
private val a = A()

// Benchmark result: 2.698 ns/op
fun accessA(blackhole: Blackhole) {
    blackhole.consume(a)
}

// Benchmark result: 3.814 ns/op
fun createA(blackhole: Blackhole) {
    blackhole.consume(A())
}

// Benchmark result: 3828.540 ns/op
fun createListAccessA(blackhole: Blackhole) {
    blackhole.consume(List(1000) { a })
}

// Benchmark result: 5322.857 ns/op
fun createListCreateA(blackhole: Blackhole) {
    blackhole.consume(List(1000) { A() })
}
```

Item 42: Avoid unnecessary object creation



`int` takes 4 bytes
`Integer` takes 16 bytes

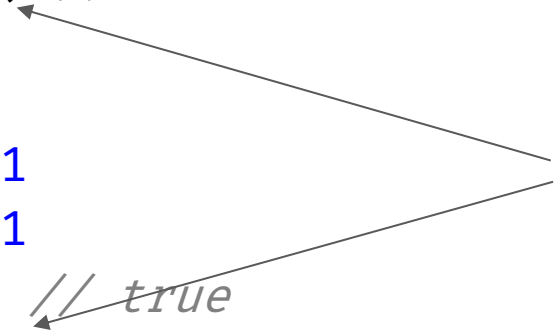
Item 42: Avoid unnecessary object creation

```
val str1 = "Lorem ipsum dolor sit amet"
val str2 = "Lorem ipsum dolor sit amet"
print(str1 == str2) // true
print(str1 === str2) // true
```

Item 42: Avoid unnecessary object creation

```
val j1: Int? = 1234
val j2: Int? = 1234
print(j1 == j2) // true
print(j1 === j2) // false
```

```
val i1: Int? = 1
val i2: Int? = 1
print(i1 == i2) // true
print(i1 === i2) // true, because i2 was taken from cache
```



Object declaration

```
sealed class LinkedList<out T>

class Node<out T>(
    val head: T,
    val tail: LinkedList<T>
) : LinkedList<T>()

object Empty : LinkedList<Nothing>()

// Usage
val list: LinkedList<Int> = Node(1, Node(2, Node(3, Empty)))
val list2: LinkedList<String> = Node("A", Node("B", Empty))
```

Factory function with a cache

```
fun <T> List<T> emptyList() {
    return EMPTY_LIST;
}
```

```
private val connections =
    mutableMapOf<String, Connection>()

fun getConnection(host: String) =
    connections.getOrPut(host) { createConnection(host) }
```

Factory function with a cache

```
private val FIB_CACHE = mutableMapOf<Int, BigInteger>()

fun fib(n: Int): BigInteger = FIB_CACHE.getOrPut(n) {
   if (n <= 1) BigInteger.ONE else fib(n - 1) + fib(n - 2)
}
```

n	100 [ns]	200 [ns]	300 [ns]	400 [ns]
fib iterative	1997	5234	7008	9727
fib with cache (first)	4413	9815	15484	22205
fib with cache (later)	8	8	8	8

Object references

`WeakReference<T>`	do not prevent Garbage Collector from cleaning-up the value. So once no other reference (variable) is using it, the value will be cleaned.
`SoftReference<T>`	not guaranteeing that the value won't be cleaned up by the GC either, but in most JVM implementations, this value won't be cleaned unless memory is needed. Soft references are perfect when we implement a cache.

Heavy object lifting

```
fun <T: Comparable<T>> Iterable<T>.countMax(): Int =
    count { it == this.max() }
```

```
fun <T: Comparable<T>> Iterable<T>.countMax(): Int {
    val max = this.max()
    return count { it == max }
}
```

Heavy object lifting

```
private val IS_VALID_EMAIL_REGEX by lazy { "\\A(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\z".toRegex() }

fun String.isValidIpAddress(): Boolean =
    matches(IS_VALID_EMAIL_REGEX)

// Usage
print("5.173.80.254".isValidIpAddress()) // true
```

Lazy initialization

```
class A {
    val b = B()
    val c = C()
    val d = D()

    //...
}
```

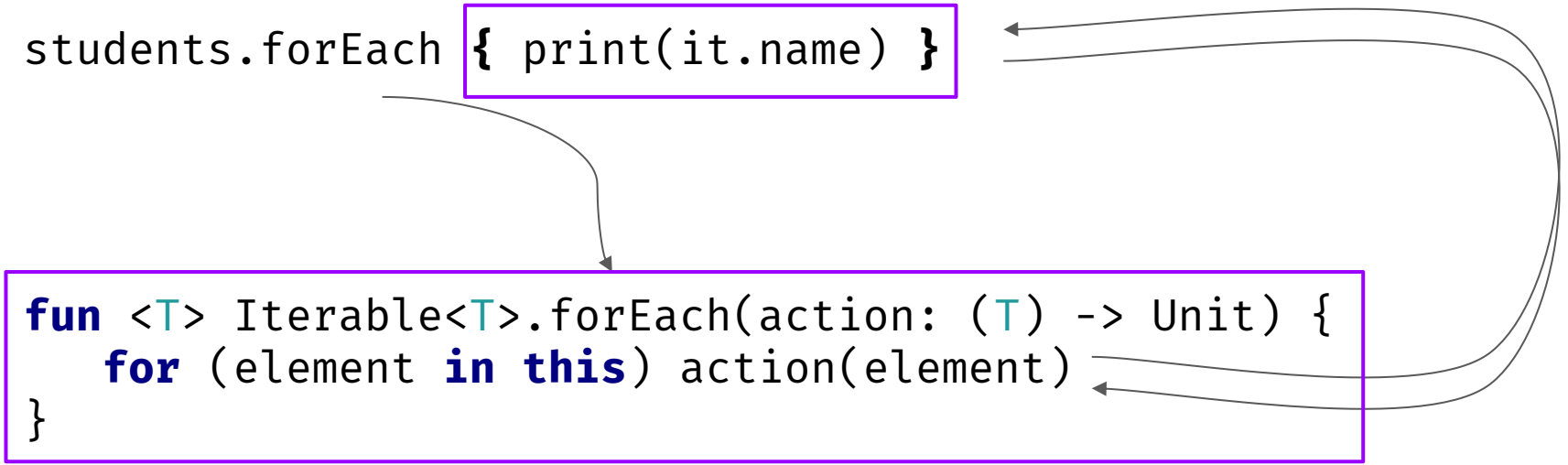
```
class A {
    val b by lazy { B() }
    val c by lazy { C() }
    val d by lazy { D() }

    //...
}
```

Using primitives

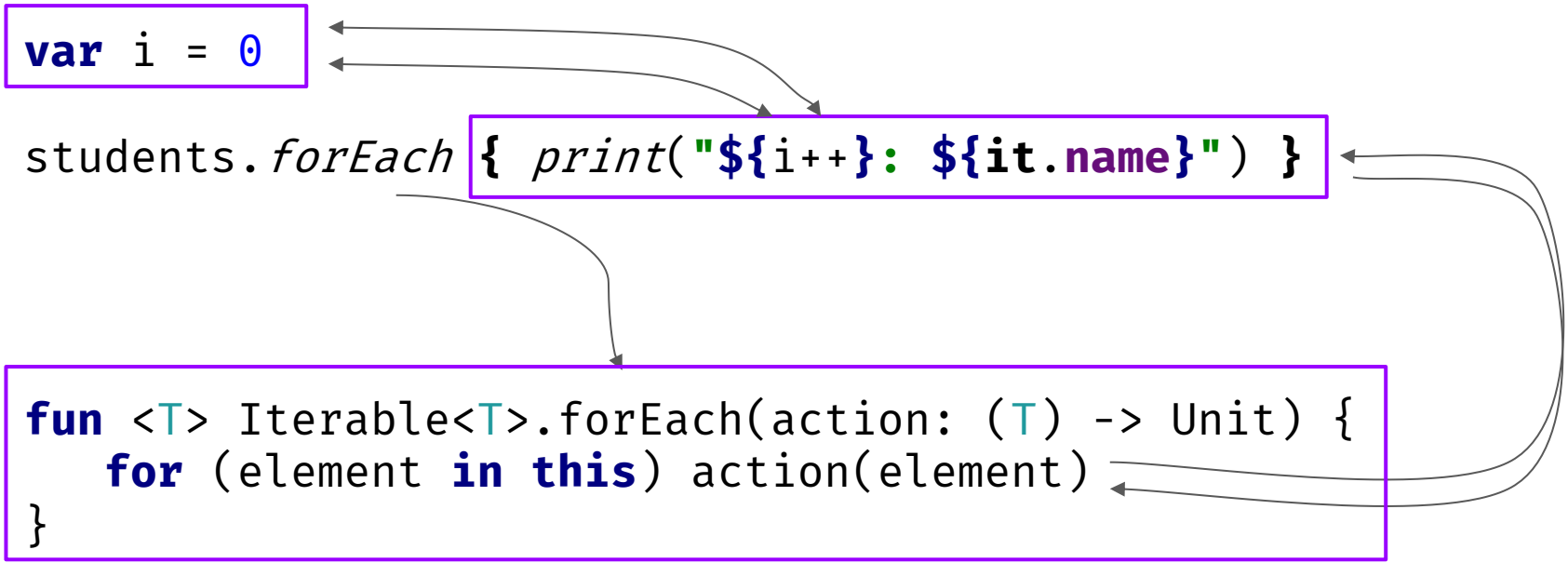
```
Int -> int
Int? -> Integer
List<Int> -> List<Integer>
```

```
fun Iterable<Int>.maxOrNull(): Int? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var max: Int = iterator.next()
    while (iterator.hasNext()) {
        val e = iterator.next()
        if (max < e) max = e
    }
    return max
}
```

Item 43: Use inline modifier for functions with parameters of functional types



Item 43: Use inline modifier for functions with parameters of functional types



Item 43: Use inline modifier for functions with parameters of functional types

```
var i = 0

for (element in students) { print("${i++}: ${it.name}") }
```

During compilation

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

A type argument in inline function can be reified

```
any is List<Int> // Error
any is List<*> // OK


inline fun <reified T> printTypeName() {
    print(T::class.simpleName)
}

// Usage
printTypeName<Int>()    // Int
printTypeName<Char>()   // Char
printTypeName<String>() // String
```

```
print(Int::class.simpleName) // Int
print(Char::class.simpleName) // Char
print(String::class.simpleName) // String
```

A type argument in inline function can be reified

```
class Worker
class Manager

val employees: List<Any> =
    listOf(Worker(), Manager(), Worker())

val workers: List<Worker> =
    employees.filterIsInstance<Worker>()
```

Costs of inline modifier

Inline functions cannot be recursive

Inline functions cannot use elements with more restrictive visibility.

Inline functions make our code grow

```
internal inline fun read() {
    val reader = Reader() // Error
    // ...
}

private class Reader {
    // ...
}
```

Crossinline and noinline

```
fun httpCall(url: String, callback: ()->Unit) {}

inline fun getToken(
    crossinline onRefresh: ()->Unit,
    noinline onGenerate: ()->Unit
) {
    if (Prefs.token == null) {
        httpCall("get-token", onGenerate)
    } else {
        httpCall("refresh-token") {
            onRefresh()
            onGenerate()
        }
    }
}
```

Inline it, but non-local return is not allowed

No not inline at all

Item 44: Consider using inline classes

```
inline class Name(private val value: String) {
    // ...
    fun greet() {
        print("Hello, I am $value")
    }
}

// Code
val name: Name = Name("Marcin")
name.greet()

// During compilation replaced with code similar to:
val name: String = "Marcin"
Name.`greet-impl`(name)
```

Indicate unit of measure

```
interface User {
    fun decideAboutTime(): Int
    fun wakeUp()
}

interface Timer {
    fun callAfter(timeMillis: Int, callback: ()->Unit)
}

fun setUpUserWakeUpUser(user: User, timer: Timer) {
    val time: Int = user.decideAboutTime()
    timer.callAfter(time) { user.wakeUp() }
}
```

```
inline class Minutes(val minutes: Int) {
    fun toMillis(): Millis = Millis(minutes * 60 * 1000)
}

inline class Millis(val milliseconds: Int)

interface User {
    fun decideAboutTime(): Minutes
    fun wakeUp()
}

interface Timer {
    fun callAfter(timeMillis: Millis, callback: ()->Unit)
}

fun setUpUserWakeUpUser(user: User, timer: Timer) {
    val time = user.decideAboutTime()
    timer.callAfter(time.toMillis()) {
        user.wakeUp()
    }
}
```

Indicate unit of measure

```
val Int.min
    get() = Minutes(this)

val Int.ms
    get() = Millis(this)

val timeMin: Minutes = 10.min
```

Protect us from type misuse

```
inline class StudentId(val studentId: Int)
inline class TeacherId(val teacherId: Int)
inline class SchoolId(val studentId: Int)

class Grades(
    @ColumnInfo(name = "studentId")
    val studentId: StudentId,
    @ColumnInfo(name = "teacherId")
    val teacherId: TeacherId,
    @ColumnInfo(name = "schoolId")
    val schoolId: SchoolId,
    // ...
)
```

```
interface TimeUnit {
    val millis: Long
}

inline class Minutes(val minutes: Long): TimeUnit {
    override val millis: Long get() = minutes * 60 * 1000
    // ...
}

inline class Millis(val milliseconds: Long): TimeUnit {
    override val millis: Long get() = milliseconds
}

fun setUpTimer(time: TimeUnit) {
    val millis = time.millis
    //...
}

setUpTimer(Minutes(123))
setUpTimer(Millis(456789))
```

Typealias

```
typealias NewName = Int
val n: NewName = 10


typealias ClickListener = (view: View, event: Event) -> Unit

class View {
    fun addClickListener(listener: ClickListener) {}
    fun removeClickListener(listener: ClickListener) {}
    //...
}
```

Typealias

```
typealias Seconds = Int
typealias Millis = Int

fun getTime(): Millis = 10
fun setUpTimer(time: Seconds) {}

fun main() {
    val seconds: Seconds = 10
    val millis: Millis = seconds // No compiler error

    setUpTimer(getTime())
}
```

Item 45: Eliminate obsolete object references

```
class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //...
        activity = this
    }

    companion object {
        // DON'T DO THIS! It is a huge memory leak
        var activity: MainActivity? = null
    }
}
```

Item 45: Eliminate obsolete object references

```
class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Be careful, we leak a reference to the whole class
        logError = { Log.e(this::class.simpleName, it.message) }
    }

    companion object {
        // DON'T DO THIS! A memory leak
        var logError: ((Throwable)->Unit)? = null
    }
}
```

```
class Stack {
   private var elements: Array<Any?> = arrayOfNulls(DEFAULT_INITIAL_CAPACITY)
   private var size = 0

   fun push(e: Any) {
       if (elements.size == size) {
           elements = elements.copyOf(2 * size + 1)
       }
       elements[size++] = e
   }

   fun pop(): Any? {
      if (size == 0) throw EmptyStackException()
      val elem = elements[--size]
      elements[size] = null
      return elem
   }

   companion object {
       private const val DEFAULT_INITIAL_CAPACITY = 16
   }
}
```

```kotlin
fun <T> mutableLazy(initializer: () -> T): ReadWriteProperty<Any?, T>
        = MutableLazy(initializer)

private class MutableLazy<T>(var initializer: (() -> T)?) : ReadWriteProperty<Any?, T> {
    private var value: T? = null

    override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        synchronized(this) {
            initializer?.let {
                value = it()
                initializer = null
            }
            return value as T
        }
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        synchronized(this) {
            this.value = value
            initializer = null
        }
    }
}
```

Start with profiling

Chapter 8: Efficient collection processing

Collection processing

```
val newsItemAdapters = news
    .filter { it.visible }
    .sortedByDescending { it.publishedAt }
    .map(::NewsItemAdapter)
```

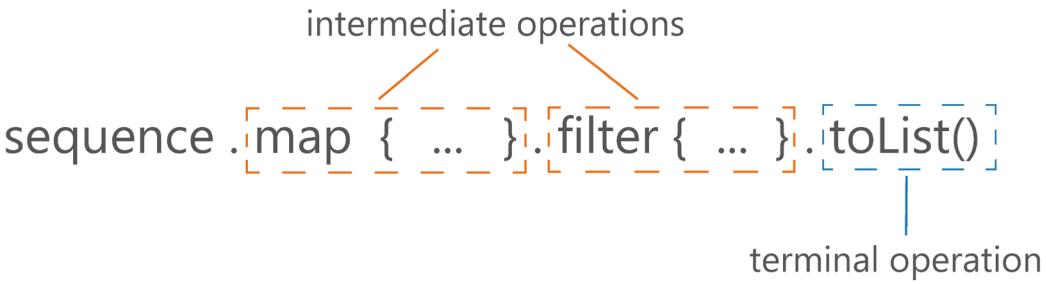
Iterable vs Sequence

```
interface Iterable<out T> {
    operator fun iterator(): Iterator<T>
}

interface Sequence<out T> {
    operator fun iterator(): Iterator<T>
}
```

Iterable vs Sequence

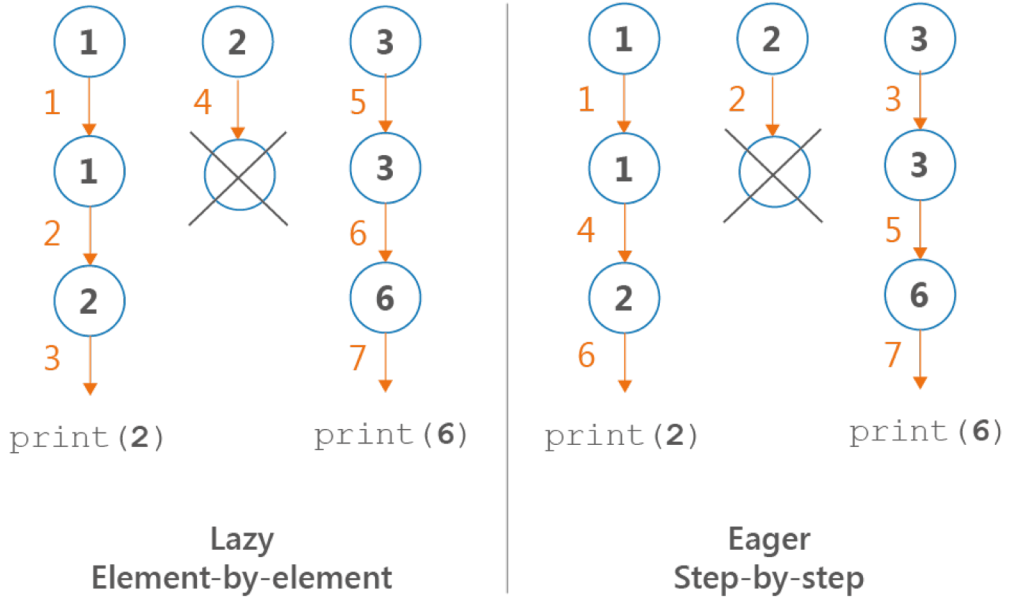
```
inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList(), predicate)
}

fun <T> Sequence<T>.filter(predicate: (T) -> Boolean): Sequence<T> {
    return FilteringSequence(this, true, predicate)
}
```

Iterable vs Sequence

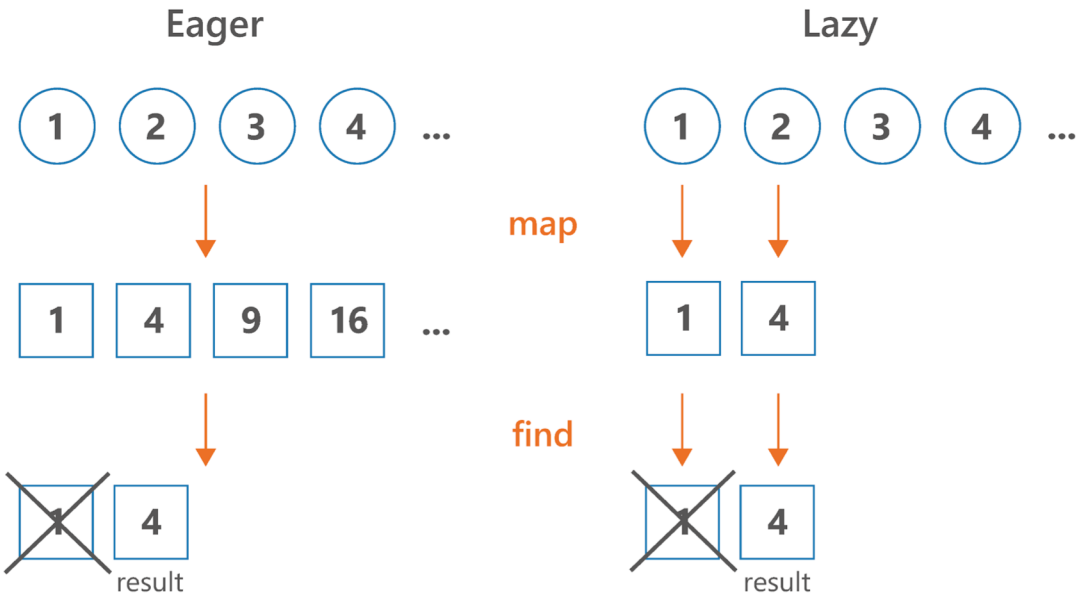


Iterable vs Sequence

```
filter { it % 2 == 1 }

map { it * 2 }

forEach { print(it) }
```



Iterable vs Sequence



Iterable vs Sequence

```
File("ChicagoCrimes.csv").useLines { lines: Sequence<String> ->
    lines.drop(1) // Drop descriptions of the columns
         .mapNotNull { it.split(",").getOrNull(6) }
         .filter { "CANNABIS" in it }
         .count()
         .let { println(it) } // 318185
}
```

Exception in thread “main” java.lang.OutOfMemoryError: Java heap space

Tough business of best practices

Item 46: Prefer Sequence for **big collections** with **more than one processing step**

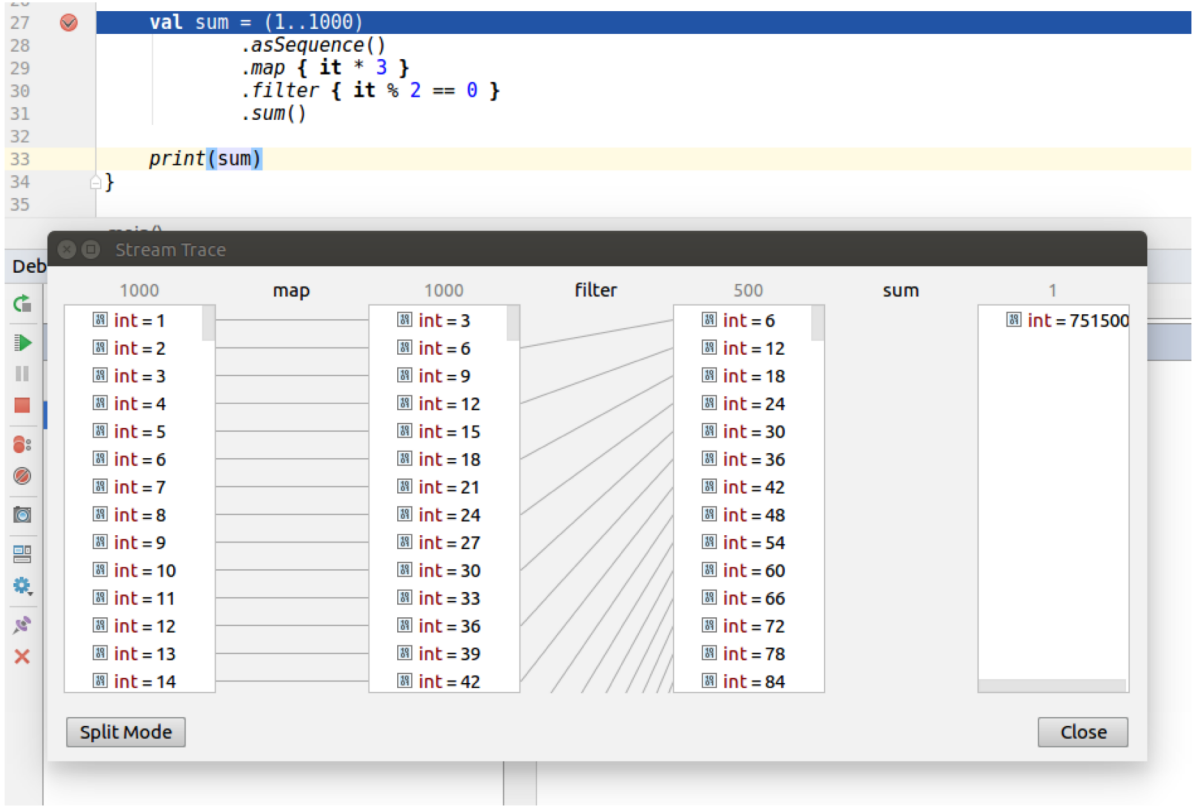
```
val minIndent = lines
    .filter { it.trim().isNotEmpty() }
    .map(St
    .min()
```

Convert call chain into 'Sequence'

Show lambda return expression hints

Add explicit type arguments

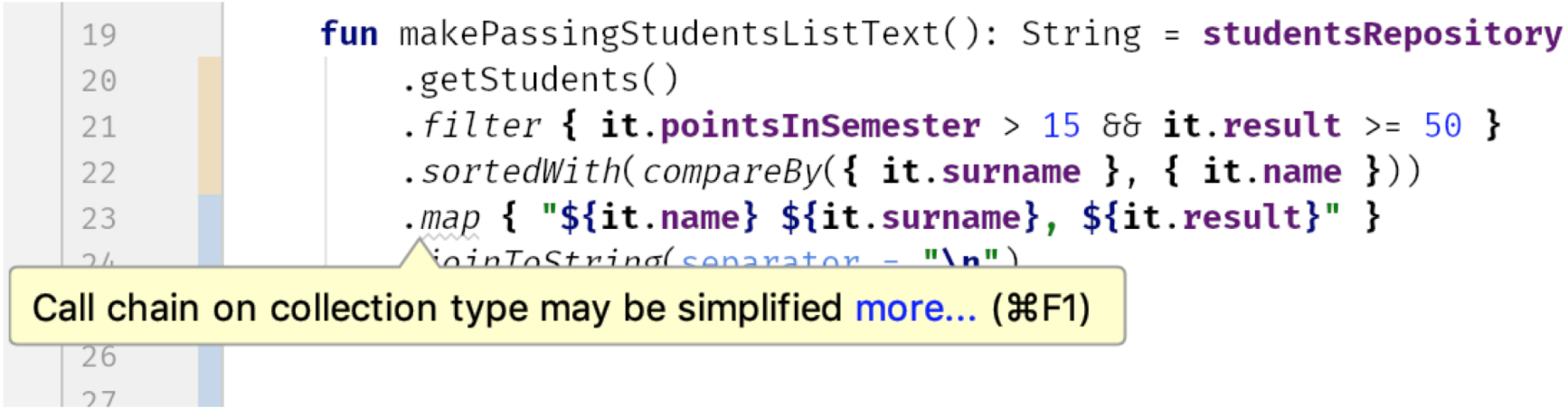
Sequence debugging



Item 47: Limit the number of operations

```
// Best
fun List<Student>.getNames(): List<String> = this
    .mapNotNull { it.name }
```

Item 47: Limit the number of operations



Instead of:	Use:
`.filter { it != null }` `.map { it!! }`	`.filterNotNull()`
`.map { <Transformation> }` `.filterNotNull()`	`.mapNotNull { <Transformation> }`
`.map { <Transformation> }` `.joinToString()`	`.joinToString { <Transformation> }`
`.filter { <Predicate 1> }` `.filter { <Predicate 2> }`	`.filter {` `  <Predicate 1> && <Predicate 2> }`
`.filter { it is Type }` `.map { it as Type }`	`.filterIsInstance<Type>()`
`.sortedBy { <Key 2> }` `.sortedBy { <Key 1> }`	`.sortedWith(` `compareBy({ <Key 1> }, { <Key 2> }))`
`listOf(...)` `.filterNotNull()`	`listOfNotNull(...)`

`.withIndex()`
`.filter { (index, elem) ->`
`<Predicate using index> }`
`.map { it.value }`
	`.filterIndexed { index, elem ->`
	`<Predicate using index> }`
	
	(Similarly for map, forEach, reduce and fold)

Item 48: Consider Arrays with primitives for performance-critical processing

```
List<Int> -> List<Integer>
Array<Int> -> Integer[]
IntArray -> int[]
```

```
val bigIntList = List(10_000) { it }       // 2 000 006 944 bytes
val bigIntArray = bigIntList.toIntArray() //   400 000 016 bytes

bigIntList.average()  // 1 438 909 ns (based on a benchmark)
bigIntArray.average() // 1 206 723 ns (based on a benchmark)
```

Item 49: Consider using mutable collections

```
var list = listOf(1,2,3)
list += 4

operator fun <T> Iterable<T>.plus(element: T): List<T> {
    if (this is Collection) return this.plus(element)
    val result = ArrayList<T>()
    result.addAll(this)
    result.add(element)
    return result
}
```

Item 49: Consider using mutable collections

```
inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    val destination = ArrayList<R>(
        if (this is Collection<*>) this.size else 10
    )
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

Item 49: Consider computational complexity

Efficient Kotlin

Speed-up your Kotlin project

Marcin Moskała