Marcin Moskała

Kotlin exercises

Kotlin Exercises

Marcin Moskała

This book is available at http://leanpub.com/kotlin_exercises

This version was published on 2024-11-21

Contents

Introduction

This is not a standard book, but rather a collection of exercises that I am using when I am teaching Kotlin. I collected them all into one place, to make them easier to use.

Most of those exercises are also included in the following books:

- Kotlin Essentials
- Functional Kotlin
- Kotlin Coroutines: Deep Dive
- Advanced Kotlin
- Effective Kotlin: Best Practices

They include all the knowledge needed to solve them. If you want to learn Kotlin, I recommend you to read them. Though if you are only interested in solving exercises, feel free to use those presented in this book.

Kotlin Essentials exercises

This is a collection of exercises that were designed to practice the knowledge presented in the Kotlin Essentials book. They are not very difficult, but they will help you to learn and practice some essential knowledge. Some might seem a bit boring, but practicing basic knowledge in separation is sometimes like that. Keep in mind that soon this knowledge will be used to do things that are much more interesting.

Exercise: Your first program

Your task is to create a program that will print “Hello, World” to the console and then check what was the code compiled to on JVM.

1. Install IntelliJ IDEA if you do not have it yet.
2. Create a new Kotlin project.
3. Create a new Kotlin file.
4. Create a `main` function using Live Template “main”.
5. Use the `println` function to print “Hello, World” to the console.
6. Run the program using the Run button (gutter icon).
7. Check the output in the Run tool window.
8. Select “Show Kotlin Bytecode” from the Tools > Kotlin menu.
9. Click “Decompile” in the Kotlin Bytecode window.
10. Check the output, and compare this generated Java file with the original Kotlin code.

Exercise: Basic values operations

What will be the result of the following expressions?

```
fun main() {
    println(1 + 2 * 3) // ?
    println(10 % 3) // ?
    println(-1 % 3) // ?

    println(8.8 / 4) // ?
    println(10 / 3) // ?

    println(11.toFloat()) // ?
    println(10.10.toInt()) // ?

    var a = 10
    a += 5
    println(a) // ?
    a -= 3
    println(a) // ?
    a++
    println(a) // ?
    println(a++) // ?
    println(a) // ?
    println(--a) // ?
    println(a) // ?

    println(true && false) // ?
    println(true || false) // ?
    println(!!!!true) // ?

    println('A'.code) // 65
    println('A' + 1) // ?
    println('C'.code) // ?

    println("A + B") // ?
    println("A" + "B") // ?
    println("A" + 1) // ?
    println("A" + 1 + 2) // ?
}
```

Exercise: Using when

What will be the result of the following expressions?

```
private val magicNumbers = listOf(7, 13)

fun name(a: Any?): String = when (a) {
    null -> "Nothing"
    1, 2, 3 -> "Small number"
    in magicNumbers -> "Magic number"
    in 4..100 -> "Big number"
    is String -> "String: $a"
    is Int, is Long -> "Int or Long: $a"
    else -> "No idea, really"
}

fun main() {
    println(name(1)) // ?
    println(name("A")) // ?
    println(name(null)) // ?
    println(name(5)) // ?
    println(name(100)) // ?
    println(name('A')) // ?
    println(name("1")) // ?
    println(name(-1)) // ?
    println(name(101)) // ?
    println(name(1L)) // ?
    println(name(7)) // ?
    println(name(3)) // ?
    println(name(3.0)) // ?
    println(name(100L)) // ?
}
```

Exercise: Pretty time display

Implement the `secondsToPrettyTime` function that takes an integer number of seconds and returns a string representation of the time in the following format: "X h Y min Z sec", where X, Y, and Z are the number of hours, minutes, and seconds respectively. If a value is zero, return "Now". If the input is negative, return "Invalid input".

```
fun secondsToPrettyTime(seconds: Int): String {
    return ""
}
```

Example usage:

```
println(secondsToPrettyTime(-1)) // Invalid input
println(secondsToPrettyTime(0)) // Now
println(secondsToPrettyTime(45)) // 45 sec
println(secondsToPrettyTime(60)) // 1 min
println(secondsToPrettyTime(150)) // 2 min 30 sec
println(secondsToPrettyTime(1410)) // 23 min 30 sec
println(secondsToPrettyTime(60 * 60)) // 1 h
println(secondsToPrettyTime(3665)) // 1 h 1 min 5 sec
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/conditions/PrettyTime.kt`. You can clone this project and solve this exercise locally.

Hint: You can use `trim` function on a string to remove leading and trailing whitespace characters.

Exercise: Person details display

Your task is to implement `formatPersonDisplay` function: It should have `String` result type and the following parameters:

- `name` of type `String?` and default value `null`.
- `surname` of type `String?` and default value `null`.
- `age` of type `Int?` and default value `null`.

> Beware! Parameter types should include `?`, so those should be `String?` and `Int?` instead of `String` and `Int`. This is because we want to allow passing `null` as a parameter value. This will be explained in the chapter *Nullability*.

Function should return a string in the following format: `"{name} {surname} ({age})"`. If any of the parameters is `null`, it should be omitted from the result. If all parameters are `null`, it should return an empty string.

Here are some examples of how the function should work:

```
println(formatPersonDisplay("John", "Smith", 42))
// John Smith (42)
println(formatPersonDisplay("Alex", "Simonson"))
// Alex Simonson
println(formatPersonDisplay("Peter", age = 25))
// Peter (25)
println(formatPersonDisplay(surname="Johnson", age=18))
// Johnson (18)
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/functions/PersonDisplay.kt`. You can clone this project and solve this exercise locally.

In this project, example usage and unit tests are commented not to prevent other files from compilation. To uncomment them, select commented lines and use `command + /` on Mac (`Ctrl + /` on Windows)

Hint: You can use `trim` function on a string to remove leading and trailing whitespace characters.

Exercise: Range Operations

Implement the following functions using range operations:

- `calculateSumOfSquares` function that takes an integer `n` as an argument and returns the sum of the squares of all positive integers from `0` to `n`.
- `calculateSumOfEven` function that takes an integer `n` as an argument and returns the sum of all even numbers from `0` to `n`.
- `countDownByStep` function that takes three integers: `start`, `end`, and `step`, and returns a string representation of the numbers from `start` down to `end`, with each step being `step` units apart.

```
fun calculateSumOfSquares(n: Int): Int = TODO()

fun calculateSumOfEven(n: Int): Int = TODO()

fun countDownByStep(
    start: Int,
    end: Int,
    step: Int
): String = TODO()
```

Example usage:

```
// Examples for calculateSumOfSquares
println(calculateSumOfSquares(0)) // 0
println(calculateSumOfSquares(1)) // 1
println(calculateSumOfSquares(2)) // 5 (1 + 4)
println(calculateSumOfSquares(3)) // 14 (1 + 4 + 9)
println(calculateSumOfSquares(4)) // 30 (1 + 4 + 9 + 16)

// Example for calculateSumOfEven
println(calculateSumOfEven(0)) // 0
println(calculateSumOfEven(1)) // 0
println(calculateSumOfEven(2)) // 2
println(calculateSumOfEven(3)) // 2
println(calculateSumOfEven(5)) // 6 (2 + 4)
println(calculateSumOfEven(10))
// 30 (2 + 4 + 6 + 8 + 10)
println(calculateSumOfEven(12))
// 42 (2 + 4 + 6 + 8 + 10 + 12)
println(calculateSumOfEven(20))
```

```
// 110 (2 + 4 + 6 + 8 + 10 + 12 + 14 + 16 + 18 + 20)

// Example for countDownByStep
println(countDownByStep(1, 1, 1)) // 1
println(countDownByStep(5, 1, 2)) // 5, 3, 1
println(countDownByStep(10, 1, 3)) // 10, 7, 4, 1
println(countDownByStep(15, 5, 5)) // 15, 10, 5
println(countDownByStep(20, 2, 3))
// 20, 17, 14, 11, 8, 5, 2
println(countDownByStep(10, 4, 3)) // 10, 7, 4
println(countDownByStep(-1, -1, 1)) // -1
println(countDownByStep(-5, -9, 2)) // -5, -7, -9
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/loops/Loops.kt`. You can clone this project and solve this exercise locally.

Exercise: User Information Processor

Implement the `processUserInformation` function according to the following rules:

- If the input `user` is `null`, return `"Missing user information"`.
- If `name` from `user` is `null`, throw an `IllegalArgumentException`.
- If `age` from `user` is `null`, treat it as 0.
- If `email` from `user` is `null` or contains a blank email address, return `"Missing email"`.
- Otherwise, return information about user in the following format: "User {name} is {age} years old, email: {email}".

Starting code:

```
fun processUserInformation(user: User?): String {
    return ""
}

data class EmailAddress(val email: String?)

data class User(
    val name: String?,
    val age: Int?,
    val email: EmailAddress?
)
```

Example usage:

```
println(processUserInformation(null))
// Missing user information

val user1 = User(
    "John",
    30,
    EmailAddress("john@example.com")
)
println(processUserInformation(user1))
// User John is 30 years old, email: john@example.com

val user2 = User(
    "Alice",
    null,
    EmailAddress("alice@example.com")
```

```
)
println(processUserInformation(user2))
// User Alice is 0 years old, email: alice@example.com

val user3 = User(
    "Bob",
    25,
    EmailAddress("") // or EmailAddress(null) or null
)
println(processUserInformation(user3))
// Missing email

val user6 = User(
    null,
    40,
    EmailAddress("jake@example.com")
)
println(processUserInformation(user6))
// IllegalArgumentException
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/nullability/UserInformation.kt`. You can clone this project and solve this exercise locally.

Exercise: Implementing the Product class

Define the `Product` class with properties:

- `name` (String): The product's name.
- `price` (Double): The product's price.
- `quantity` (Int): The initial quantity of the product.

Implement a custom setter for the `quantity` property that ensures that negative values are set to zero.

Add a method named `calculateTotalValue` that returns the product's total value.

Implement a method named `restock` that increases the product's quantity by a specified positive amount. If a negative value is specified, the method should do nothing.

Instructions:

1. Define the `Product` class with primary constructor properties `name`, `price`, and parameter `quantity`.
2. Implement a property `quantity` with a custom setter.
3. Implement the `calculateTotalValue` method to calculate and return the total value.
4. Implement the `restock` method to increase the quantity.
5. Test your implementation by creating instances of the `Product` class and using its methods.

Example usage:

```
val laptop = Product("Laptop", 999.99, 5)

println(laptop.name) // Laptop
println(laptop.quantity) // 5
println(laptop.calculateTotalValue()) // 4999.95

laptop.restock(3)

println(laptop.quantity) // 8
println(laptop.calculateTotalValue()) // 7999.92

laptop.quantity = -2

println(laptop.quantity) // 0
println(laptop.calculateTotalValue()) // 0.0
```

```
laptop.quantity = 10

println(laptop.quantity) // 10
println(laptop.calculateTotalValue()) // 9999.9
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/classes/Product.kt`. You can clone this project and solve this exercise locally.

Hints:

- Use the `field` keyword in the custom setter to modify the backing field.
- Calculate the total value by multiplying `price` and `quantity`.

Exercise: GUI View Hierarchy Simulation

Implement the following classes and methods, that will represent a view hierarchy in a GUI system:

1. Create an open class `View` with primary constructor properties: read-only `id` of type `String` and read-write `isVisible` of type `Boolean`.
2. Implement methods `show` and `hide` in the `View` class, that will change the `isVisible` property to `true` and `false` respectively.
3. Create two subclasses of `View`: `TextView` and `Toggle`.
4. `TextView` should have a constructor with parameter `id` and property `text`, that will represent the text displayed in the view. It should set superclass `isVisible` property to `true`.
5. `Toggle` should have a constructor with parameter `id`. It should set superclass `isVisible` property to `true`.
6. `Toggle` should have an additional property `isOn`, that will represent the state of the toggle. It should be initially set to `false`.
7. `Toggle` should have a method `click`, that will change the `isOn` property to the opposite state.

Example usage:

```
val view = View(
    id = "v1",
    isVisible = false,
)
println(view.id) // v1
println(view.isVisible) // false

val textView = TextView(
    id = "tv1",
    text = "Hello, World!",
)
println(textView.id) // tv1
println(textView.text) // Hello, World!
println(textView.isVisible) // true

textView.text = "Welcome to Kotlin!"
println(textView.text) // Welcome to Kotlin!

textView.hide()
println(textView.isVisible) // false
```

```
val toggle = Toggle(
    id = "toggle1",
)
println(toggle.id) // toggle1

println(toggle.isOn) // false
toggle.click()
println(toggle.isOn) // true

println(toggle.isVisible) // true
toggle.hide()
println(toggle.isVisible) // false
toggle.show()
println(toggle.isVisible) // true
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/classes/Gui.kt`. You can clone this project and solve this exercise locally.

Exercise: Data class practice

1. Create a data class for a `Person` with a `name` and `age` property of types `String` and `Int`.
2. Create a `Person` instance with name "John" and age 30.
3. Print the `Person` instance.
4. Create a copy of the `Person` instance with name "Jane".
5. Create a new `Person` instance with name "Jane" and age 30.
6. Check if the two `Person` instances are equal.
7. Print the `hashCode` of all the `Person` instances.
8. Destructure the `Person` instance created using `copy` (so the one with name "Jane") into two variables, and print values of those variables.

Exercise: Pizza factory

Your task is to define a companion object in the `Pizza` class that will contain factory functions for creating pizzas. The functions should be named `hawaiian` and `margherita` and should return pizzas with the following toppings:

- Hawaiian: ham, pineapple
- Margherita: tomato, mozzarella

```
class Pizza(
    val toppings: List<String>,
) {
    // Class body
}
```

> In the starting code, there is an empty class body with a comment. I added it because otherwise, people doing the exercise often got confused, and instead of in the class body, they defined the companion object in the constructor.

The following code should work:

```
val hawaiian = Pizza.hawaiian()
println(hawaiian.toppings) // [ham, pineapple]
val margherita = Pizza.margherita()
println(margherita.toppings) // [tomato, mozzarella]
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/objects/Pizza.kt`. You can clone this project and solve this exercise locally.

Hint: To create a list of strings, you can use the `listOf` function. For example, `listOf("a", "b", "c")` creates a list with three elements: `"a"`, `"b"`, and `"c"`.

Exercise: Catching exceptions

In the below `handleInput` function, different kind of user inputs can lead to exceptions that can break our program execution. Your task is to catch all possible exceptions and display appropriate messages to the user.

- If the user input for a number is not a correct number, then `toInt` throws `NumberFormatException`. In such case, you should print "Invalid input: ", and error message.
- If the user input is a number, but the second number is zero, then `ArithmeticException` is thrown. In such case, you should print "Division by zero".
- If the user input is an operator that is not supported by our calculator, then `IllegalOperatorException` is thrown. In such case, you should print "Illegal operator: ", and the operator that was entered by the user.

Starting code:

```
fun main() {
    while (true) {
        // Wrap below function call with try-catching block,
        // and handle possible exceptions.
        handleInput()
    }
}

fun handleInput() {
    print("Enter the first number: ")
    val num1 = readln().toInt()
    print("Enter an operator (+, -, *, /): ")
    val operator = readln()
    print("Enter the second number: ")
    val num2 = readln().toInt()

    val result = when (operator) {
        "+" -> num1 + num2
        "-" -> num1 - num2
        "*" -> num1 * num2
        "/" -> num1 / num2
        else -> throw IllegalOperatorException(operator)
    }

    println("Result: $result")
}
```

```
class IllegalOperatorException(val operator: String) :
    Exception("Unknown operator: $operator")
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/exceptions/HandleExceptions.kt`. You can clone this project and solve this exercise locally.

Exercise: Days of the week enum

Your task is to create an enum that represents the days of the week and contains a few useful properties and functions. Here are the exact instructions:

1. Define an enum class named `DayOfWeek` with the following days: `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, `SUNDAY`.
2. Enum elements should be defined in the order they appear in the week.
3. Enum should define a primary constructor with two properties: `isWeekend` (of type `Boolean`) indicating whether the day is a weekend day (Saturday or Sunday) and `dayName` (of type `String`) containing the full name of the day.
4. Implement a function named `nextDay` that takes a `DayOfWeek` as input and returns the next day in the sequence. For example, if the input is `MONDAY`, the function should return `TUESDAY`.

The following code should work:

```
val friday: DayOfWeek = DayOfWeek.FRIDAY
println(friday.dayName) // Friday
println(friday.isWeekend) // false
val saturday: DayOfWeek = friday.nextDay()
println(saturday.dayName) // Saturday
println(saturday.isWeekend) // true
```

Example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/enums/DayOfWeek.kt`. You can clone this project and solve this exercise locally.

Exercise: Conversion and measurement unit creation

Your task is to implement two kinds of extension functions. First, you need functions to transform between `User` and `UserJson` classes. `User` represents our domain object, while `UserJson` represents the structure that is sent to the client. The conversion should be done according to the following rules:

1. `username` should be copied as is,
2. `email` should be converted to/from `String` in the `Email` class,
3. `registrationDate` should be converted to/from `String` in ISO format. You can use `parse` and `toString` fron `LocalDateTime` for that,
4. `height` should be converted to/from `Int` in `Centimeters` class.

Then implement `cm` extension property for `Int` that will create a `Centimeters` object.

```
data class User(
    val username: String,
    val email: Email,
    val registrationDate: LocalDateTime,
    val height: Centimeters,
)

data class Email(val value: String)

data class Centimeters(val value: Int)

data class UserJson(
    val username: String,
    val email: String,
    val registrationDate: String,
    val heightCm: Int,
)
```

Once your solution is ready, the following code should work:

```
val user = User(
    username = "alex",
    email = Email("alex@example.com"),
    registrationDate = LocalDateTime
        .of(1410, 7, 15, 10, 13),
    height = 170.cm,
)
val userJson = user.toUserJson()
println(userJson)
// UserJson(username=alex, email=alex@example.com,
// registrationDate=1410-07-15T10:13, heightCm=170)
val user2 = userJson.toUser()
println(user2) // User(username=alex,
// email=Email(value=alex@example.com),
// registrationDate=1410-07-15T10:13,
// height=Centimeters(value=170))
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/extensions/User.kt`. You can clone this project and solve this exercise locally.

Exercise: Inventory management

Your task is to implement the missing functions in the `Inventory` class. This is how those methods should work:

- `addProduct` should add a product to the inventory and assign a producer to it,
- `removeProduct` should remove a product from the inventory and remove the producer,
- `addSeller` should add a seller to the sellers' collection,
- `removeSeller` should remove a seller from the sellers' collection,
- `getProductsCount` should return the number of products in the inventory,
- `hasProduct` should return `true` if the inventory contains the given product,
- `hasProducts` should return `true` if the inventory contains any products,
- `getProducer` should return the producer of the given product, or `null` if the product is not in the inventory,
- `produceInventoryDisplay` should return a string containing the inventory display in the following format:

```
Inventory:
{name} ({category}) - ${price}
Produced by: {producer}
Sellers: {sellers}
```

Starting code:

```
class Inventory {
    private val products = mutableListOf<Product>()
    private val productIdToProducer =
        mutableMapOf<String, String>()
    private val sellers = mutableSetOf<String>()

    fun addProduct(product: Product, producer: String) {
        // TODO: Add product and assign producer
    }

    fun removeProduct(product: Product) {
        // TODO: Remove product and producer
    }

    fun getProductsCount(): Int = TODO()

    fun hasProduct(product: Product): Boolean = TODO()
```

```
    fun hasProducts(): Boolean = TODO()

    fun getProducer(product: Product): String? = TODO()

    fun addSeller(seller: String) {
        // TODO: Add seller
    }

    fun removeSeller(seller: String) {
        // TODO: Remove seller
    }

    fun produceInventoryDisplay(): String {
        var result = "Inventory:\n"
        // TODO: For each product, print name, category, price
        // in the format "{name} ({category}) - ${price}"
        // and print the producer in the format
        // "Produced by: {producer}"
        // TODO: Print sellers in the format
        //  "Sellers: {sellers}"
        return result
    }
}

class Product(
    val id: String,
    val name: String,
    val price: Double,
    val category: String,
)
```

Once your solution is ready, the following code should work:

```
val inventory = Inventory()
println(inventory.hasProducts()) // false

val p1 = Product("P1", "Phone", 599.99, "Electronics")
val p2 = Product("P2", "Laptop", 1199.99, "Electronics")
val p3 = Product("P3", "Shirt", 29.99, "Clothing")

inventory.addProduct(p1, "TechCompany")
inventory.addProduct(p2, "TechCompany")
inventory.addProduct(p3, "ClothingCompany")
```

```
inventory.addSeller("Seller1")
inventory.addSeller("Seller2")

println(inventory.getProductsCount()) // 3
println(inventory.hasProduct(p1)) // true
println(inventory.hasProducts()) // true
println(inventory.getProducer(p1)) // TechCompany

println(inventory.produceInventoryDisplay())
// Inventory:
// Phone (Electronics) - $599.99
// Produced by: TechCompany
// Laptop (Electronics) - $1199.99
// Produced by: TechCompany
// Shirt (Clothing) - $29.99
// Produced by: ClothingCompany
// Sellers: [Seller1, Seller2]

inventory.removeProduct(p2)
inventory.addSeller("Seller1")
inventory.removeSeller("Seller2")

println(inventory.getProductsCount()) // 2
println(inventory.hasProduct(p1)) // true
println(inventory.hasProduct(p2)) // false
println(inventory.hasProducts()) // true
println(inventory.getProducer(p2)) // null

println(inventory.produceInventoryDisplay())
// Inventory:
// Phone (Electronics) - $599.99
// Produced by: TechCompany
// Shirt (Clothing) - $29.99
// Produced by: ClothingCompany
// Sellers: [Seller1]
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/collections/Inventory.kt`. You can clone this project and solve this exercise locally.

Exercise: Money operations

Implement appropriate operator methods in the class `Money` to allow:

- adding two `Money` objects using `+` operator, (in case of different currencies, throw `IllegalArgumentException`),
- subtracting two `Money` objects using `-` operator, (in case of different currencies, throw `IllegalArgumentException`),
- negating a `Money` object using unary `-` operator,
- multiplying a `Money` object by an integer number using `*` operator.

Starting code:

```
data class Money(
    val amount: BigDecimal,
    val currency: Currency
) {
    // TODO: Implement operators overloading here

    companion object {
        fun eur(amount: String) =
            Money(BigDecimal(amount), Currency.EUR)
    }
}

enum class Currency {
    EUR, USD, GBP
}
```

The following code should work:

```
val money1 = Money.eur("10.00")
val money2 = Money.eur("29.99")

println(money1 + money2) // Money(amount=39.99, currency=EUR)
println(money2 - money1) // Money(amount=19.99, currency=EUR)
println(-money1) // Money(amount=-10.00, currency=EUR)
println(money1 * 3) // Money(amount=30.00, currency=EUR)
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/operators/Money.kt`. You can clone this project and solve this exercise locally.

Exercise: The closest supertype of types

What is the closest supertype of the following types?

- `Int` and `Double`
- `Double` and `Number`
- `String` and `Nothing`
- `Float` and `Double?`
- `String` and `Float`
- `Char` and `Nothing?`
- `Nothing` and `Any`
- `Nothing?` and `Any`
- `Char?` and `Nothing?`
- `Nothing?` and `Any?`

Exercise: Stock

Your task is to implement a generic stack class. It should be capable of storing elements of any data type and supporting standard stack operations, including:

1. `push(item: T)`: This method adds an item of type `T` to the top of the stack. It takes one parameter, `item`, which represents the element to be added to the stack.
2. `pop(): T?`: The `pop` method removes and returns the item from the top of the stack. It returns an item of type `T`, or `null` if the stack is empty.
3. `peek(): T?`: The `peek` method returns the item from the top of the stack without removing it. It returns an item of type `T`, or `null` if the stack is empty.
4. `isEmpty(): Boolean`: The `isEmpty` method checks if the stack is empty and returns `true` if it is, or `false` if it contains elements.
5. `size(): Int`: The `size` method returns the number of elements currently in the stack as an integer value.

You can keep a mutable list as a property of the class to store the elements. To pop an element, use `removeAt` method.

The following code should work:

```
val intStack = Stack<Int>()
intStack.push(1)
intStack.push(2)
intStack.push(3)

val stringStack = Stack<String>()
stringStack.push("A")
stringStack.push("B")
stringStack.push("C")

println(intStack.peek()) // 3
while (!intStack.isEmpty()) { // 3, 2, 1
    println(intStack.pop())
}
println(intStack.peek()) // null
println(intStack.isEmpty()) // true

println(stringStack.size()) // 3
while (!stringStack.isEmpty()) { // C, B, A
    println(stringStack.pop())
}
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/generics/Stack.kt`. You can clone this project and solve this exercise locally.

Final Project: Workout manager

In this exercise, you will be creating a console-based push-up workout manager in Kotlin. The goal of this manager is to guide users through a series of push-up sets, recording their performance, and adjusting future workout suggestions based on their results. The program will also store the suggested repetitions for the user's next workout in a file.

This is how you should start the code:

```
val interactor = WorkoutInteractor(
    manager = WorkoutManager(),
    nextRepetitionsRepository = FileNextRepetitionsRepository(),
)
interactor.start()
```

Expected Components:

- `Repetitions` data class, that stores the number of push-ups to be done in the first and second rounds. It has two properties: `firstRound` and `secondRound`, both of type `Int`.
- `NextRepetitionsRepository` interface, defining the contract for reading and writing the next set of repetitions. It has two methods: `read()`, that returns the Repetitions from storage (or `null` if not found), and `write(repetitions: Repetitions)`, that writes the given Repetitions to storage.
- `FileNextRepetitionsRepository` class, implementing the `NextRepetitionsRepository` interface. It reads and writes the `Repetitions` object to a file named `next_repetitions.txt`. If the file does not exist, it should create it.
- `WorkoutManager` class that handles the logic for determining the next set of repetitions based on the user's performance. It has a method `nextRepetitions(firstRoundDone: Int, secondRoundDone: Int, thirdRoundDone: Int, repetitions: Repetitions): Repetitions`, which returns a new `Repetitions` object with adjusted values for the next workout based on the user's performance.
- `WorkoutInteractor` class that serves as the main orchestrator of the application. It interacts with the user, guides them through the workout, gets their input, and provides feedback. It has a method `start()` that starts the workout routine, guides the user through the push-up sets, records their performance, and provides feedback for the next workout. It also has a helper method `getValidIntInput(prompt: String)` to get valid integer input from the user. The constructor of this class takes a `WorkoutManager` and a `NextRepetitionsRepository`.

To read and write file, use `File` class. To check that a file exists, use `exist` method. To read the content of a file, use `readText` method. To write to a file, use `writeText` method. To create a new file, use `createNewFile` method.

To read user input, use `readlnOrNull` function. It returns `null` if the input is not a valid integer. To convert a string to an integer, use `toIntOrNull` method. To write to the console, use `println` function.

The algorithm for updating repetitions should be as follows:

- If the user could not do enough push-ups in the first round, decrease the repetitions for both rounds by 1.
- If the user could not do enough push-ups in the second round, decrease the repetitions for the second round by 1.
- If the user could do more push-ups in the third round than in the first round, increase the repetitions for both rounds by 1.
- If the user could do more push-ups in the third round than in the second round, increase the repetitions for the second round by 1.
- Otherwise, keep the repetitions the same.

Here is what an example conversations with the user might look like:

```
Hello, I am your push-ups workout assistant!
Now do 5 push-ups
How many push-ups did you do?
5
Now rest for 1 minute
Now do 5 push-ups
How many push-ups did you do?
4
Now rest for 1 minute
Now do as many push-ups as you can!
How many push-ups did you do?
3
Your next repetitions will be: 5 and 4

Hello, I am your push-ups workout assistant!
Now do 5 push-ups
How many push-ups did you do?
5
Now rest for 1 minute
Now do 4 push-ups
How many push-ups did you do?
5
Now rest for 1 minute
Now do as many push-ups as you can!
How many push-ups did you do?
6
```

```
Your next repetitions will be: 6 and 5

Hello, I am your push-ups workout assistant!
Now do 6 push-ups
How many push-ups did you do?
-1
Please enter a valid number.
How many push-ups did you do?
0
Now rest for 1 minute
Now do 5 push-ups
How many push-ups did you do?
10
Now rest for 1 minute
Now do as many push-ups as you can!
How many push-ups did you do?
0
Your next repetitions will be: 5 and 4
```

Instructions:

1. Start by defining the `Repetitions` data class with its properties.
2. Define the `NextRepetitionsRepository` interface with the expected methods.
3. Implement the `FileNextRepetitionsRepository` class. Ensure you handle potential file-related exceptions.
4. Implement the `WorkoutManager` class. Here, you'll decide how to adjust the repetitions for the next workout based on the user's performance.
5. Finally, implement the `WorkoutInteractor` class. This class will interact with the user, guide them through the workout, get their input, and provide feedback based on the logic in the `WorkoutManager`.
6. Once all components are implemented, run the starting code to test your application. Adjust your workout recommendations, perform the push-ups, and see how the program suggests adjustments for your next session!

Functional Kotlin exercises

Time for the collection of exercises that I find most fun. Those exercises are about functional programming in Kotlin, so they will cover a lot of collection processing, lambdas, and other functional features. They are related to the book Functional Kotlin. I am sure you will enjoy them.

Exercise: Function types and literals

The following class shows example implementations of methods `add`, `printNum`, `triple`, `produceName` and `longestOf`:

```
class FunctionsClassic {

    fun add(num1: Int, num2: Int): Int = num1 + num2

    fun printNum(num: Int) {
        print(num)
    }

    fun triple(num: Int): Int = num * 3

    fun produceName(name: String): Name = Name(name)

    fun longestOf(
        str1: String,
        str2: String,
        str3: String,
    ): String = maxOf(str1, str2, str3, compareBy { it.length })
}

data class Name(val name: String)
```

Your task is to write similar classes, but instead of defining functions, they should define properties with function types. Those properties should represent the same functions as in the `FunctionsClassic` class. For instance, the `add` function should be represented by a property named `add` of type `(Int, Int) -> Int`. The behavior of those properties should also be identical to the behavior of functions from `FunctionsClassic` Implement classes with the following implementation of functional properties:

- `AnonymousFunctionalTypeSpecified` - should define properties with explicit function types and define their values using anonymous functions. The types of parameters in those anonymous functions should be inferred.
- `AnonymousFunctionalTypeInferred` - should define properties with inferred function types from anonymous function definitions that should be used to define values. The parameters of those anonymous functions should be explicitly typed.
- `LambdaFunctionalTypeSpecified` - should define properties with explicit function types and define their values using lambda expressions. The types of parameters in those lambda expressions should be inferred. You should not use the implicit parameter `it` in this class.

- `LambdaFunctionalTypeInferred` - should define properties with inferred function types from lambda expression definitions that should be used to define values. The parameters of those lambda expressions should be explicitly typed.
- `LambdaUsingImplicitParameter` - should define properties with explicit function types and define their values using lambda expressions, just like `LambdaFunctional-TypeSpecified`, but it should use implicit parameter convention whenever possible.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/base/Functional.kt`. You can clone this project and solve this exercise locally.

Exercise: Observable value

Your task is to implement the `Observable` class, which should hold a value and allow observing its changes. It should have a `value` property, which should be settable. It should also have a `observe` function, which should take a function that will be called whenever the value changes.

```
val observable = Observable(1)
println(observable.value) // 1
observable.observe { println("Changed to $it") }
observable.value = 2 // Changed to 2
println(observable.value) // 2
observable.observe { println("now it is $it") }
observable.value = 3
// Changed to 3
// now it is 3
```

Starting code:

```
class Observable<T>(initial: T) {
    var value: T = initial
}
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/base/Observable.kt`. You can clone this project and solve this exercise locally.

Exercise: Inferred function types

Consider the following code:

```
class Centimeter(val value: Double) {
    fun plus(other: Centimeter): Centimeter =
        Centimeter(value + other.value)

    fun times(other: Double): Centimeter =
        Centimeter(value * other)

    override fun toString(): String = "$value cm"
}

val Int.cm get() = Centimeter(this.toDouble())

fun distance(from: Centimeter, to: Centimeter): Centimeter =
    Centimeter(abs(to.value - from.value))
```

For the below function references, predict what will be the result function type:

- `Centimeter::plus`
- `Centimeter::times`
- `Centimeter::value`
- `Centimeter::toString`
- `Centimeter(1.0)::plus`
- `Centimeter(2.0)::times`
- `Centimeter(3.0)::value`
- `Centimeter(4.0)::toString`
- `Int::cm`
- `123::cm`
- `::distance`

Exercise: Function references

This is a continuation of the exercise “Function types and literals”. This time, your task is to implement the following classes:

- `FunctionReference` - that defines properties `printNum`, `triple` and `produceName` with explicit function types, and define their values using references to functions from the Kotlin standard library or from `Name` class.
- `FunctionMemberReference` that defines properties `printNum`, `triple`, `produceName` and `longestOf` with explicit function types, and define their values using references to methods from its own body.
- `BoundedFunctionReference` that defines properties `printNum`, `triple`, `produceName` and `longestOf` with explicit function types, and define their values using references to methods `classic` object.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/base/References.kt`. You can clone this project and solve this exercise locally.

Exercise: Inline functions

Implement the following inline functions:

- `anyOf` extension on `Iterable<*>`, that returns `true` if any element is of the given type.
- `firstOfOrNull` extension on `Iterable<*>`, that returns first element of the given type or `null` if there is no such element.
- `filterValuesInstanceOf` extension on `Map<*, *>` that returns a map with only entries that have specified types of both keys and values.

Example usage:

```
val list = listOf(1, "A", 3, "B")
println(list.anyOf<Int>()) // true
println(list.anyOf<String>()) // true
println(list.anyOf<Double>()) // true

println(list.firstOfOrNull<String>()) // A
println(list.firstOfOrNull<Int>()) // 1
println(list.firstOfOrNull<Double>()) // null

val map = mapOf(1 to 2, 2 to "A", 3 to 4, "B" to "C")
println(map.filterValuesInstanceOf<Int, String>()) // {2=A}
println(map.filterValuesInstanceOf<String, String>()) // {B=C}
println(map.filterValuesInstanceOf<Int, Int>()) // {1=2, 3=4}
println(map.filterValuesInstanceOf<String, Int>()) // {}
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/base/Inline.kt`. You can clone this project and solve this exercise locally.

Hint: This exercise can be solved using `any`, `firstOrNull`, and `filter` functions, which will be discussed in the next chapter. If you do not know them yet, you can solve this exercise using the `for` loop, or you can leave it for later.

Exercise: Implement map

In this chapter, you've seen already how `map` can be implemented, but this time it will be your task to implement it yourself. So make your own implementation, without looking at what was presented in this chapter. As a help, you can take a look at implementations of `filter` and `flatMap`. Here you have a couple of examples how `map` can be used:

```
val list = listOf(1, 2, 3)
println(list.map { it * 2 }) // [2, 4, 6]
println(list.map { "$it!" }) // [1!, 2!, 3!]
println(list.map { it % 2 == 0 }) // [false, true, false]
```

This function should be an extension function on `Iterable`, that returns `List`. Consider what is the size of the result collection. If the actual receiver type is `Collection`, you can take its `size`, otherwise you can use `10` as the initial size for the result list.

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/Map.kt`. You can clone this project and solve this exercise locally.

You can also find there simplified implementations of `onEach`, `filter` and `flatMap`. Use them as inspiration. You can also notice, that to prevent you from using `map` from the Kotlin standard library, I used a feature called import alias, so I imported this function under a different name.

```
import kotlin.collections.map as `implement it yourself`
```

Exercise: Optimize collection processing

Function `getPassingSurnames` has 5 processing steps, but we can easily transform it into 2 simple steps using the functions you know already. Do it. Here is how the function looks like:

```
fun List<StudentJson>.getPassingSurnames(): List<String> =
    this.filter { it.result >= 50 }
        .filter { it.pointsInSemester >= 15 }
        .map { it.surname }
        .filter { it != null }
        .map { it!! }
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/PassingSurnames.kt`. You can clone this project and solve this exercise locally.

Exercise: Adding element at position

We can add an element at a specific position to a mutable list using the `add` method. For instance:

```
fun main() {
    val list = mutableListOf(1, 2, 3)
    list.add(1, 4)
    println(list) // [1, 4, 2, 3]
}
```

Unfortunately, there is no similar function that would allow us to add an element at a specific position to an immutable list. Your task is to define it, and name it `plusAt`.

```
fun <T> List<T>.plusAt(index: Int, element: T): List<T> {
    TODO()
}
```

It should be used in the following way:

```
val list = listOf(1, 2, 3)
println(list.plusAt(1, 4)) // [1, 4, 2, 3]
println(list.plusAt(0, 5)) // [5, 1, 2, 3]
println(list.plusAt(3, 6)) // [1, 2, 3, 6]

val list2 = listOf("A", "B", "C")
println(list2.plusAt(1, "D")) // [A, D, B, C]
```

This function should check if the index is correct. If it is not, it should throw an `IllegalArgumentException`. Should consider `0` as a correct index, and add this element to the beginning of the list. Should also consider `size` as a correct index, and add this element to the end of the list.

I know at least three significantly different ways to implement this function. One uses a mutable collection, while others use collection processing functions and + operator. Try to implement it in all three ways.

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/PlusAt.kt`. You can clone this project and solve this exercise locally.

Exercise: Implement shop functions

Implement the following functions:

- `getWaitingCustomers` that returns the list of customers who ordered products that have not been delivered yet.
- `countProductSales` that returns the number of times a given product was ordered.
- `getCustomers` that returns the list of customers who made orders with total price greater than or equal to the given amount.

Starting code:

```
fun Shop.getWaitingCustomers(): List<Customer> = TODO()

fun Shop.countProductSales(product: Product): Int = TODO()

fun Shop.getCustomers(minAmount: Double): List<Customer> = TODO()

data class Shop(
    val name: String,
    val customers: List<Customer>
)
data class Customer(
    val name: String,
    val city: City,
    val orders: List<Order>
)
data class Order(
    val products: List<Product>,
    val isDelivered: Boolean
)
data class Product(
    val name: String,
    val price: Double
)
data class City(
    val name: String
)
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/Shop.kt`. You can clone this project and solve this exercise locally.

Exercise: Prime access list

Associating elements to a map can be an important performance optimization. Finding an element by key in a map is a lot faster than iterating over a list and comparing each element to the searched value. To see the different, implement methods of the `PrimeAccessRepository` class:

- `isOnAllowList` should return `true` if the user is on the allowlist (entry with this user id has `allowList` set to `true`), and `false` otherwise,
- `isOnDenyList` should return `true` if the user is on the denylist (entry with this user id has `denyList` set to `true`), and `false` otherwise.

```
class PrimeAccessRepository(
    private val primeAccessList: PrimeAccessList
) {
    fun isOnAllowList(userId: String): Boolean = TODO()
    fun isOnDenyList(userId: String): Boolean = TODO()
}

class PrimeAccessList(
    val entries: List<PrimeAccessEntry>
)

class PrimeAccessEntry(
    val userId: String,
    val allowList: Boolean,
    val denyList: Boolean,
)
```

Implement two kinds of solutions:

- iterating over entries to find the searched user id,
- associating entries to a map by user id in the class body, and in methods finding the entry by user id.

For each of those solutions, check its efficiency using the following code:

```
val entries = List(200_000) {
    PrimeAccessEntry(
        userId = it.toString(),
        allowList = Random.nextBoolean(),
        denyList = Random.nextBoolean()
    )
}.shuffled()
val accessList = PrimeAccessList(entries)

val repo: PrimeAccessRepository
measureTimeMillis {
    repo = PrimeAccessRepository(accessList)
}.also { println("Class creation took $it ms") }

measureTimeMillis {
    for (userId in 1L..10_000L) {
        repo.isOnAllowList(userId.toString())
    }
}.also { println("Operation took $it ms") }

measureTimeMillis {
    for (userId in 1L..10_000L) {
        repo.isOnDenyList(userId.toString())
    }
}.also { println("Operation took $it ms") }
```

> Beware! Such measurements are not precise. They are only to show the difference between the two solutions. For precise measurements, you should use a benchmarking library, like JMH.

Starting code and example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/collections/PrimeAccess.kt`. You can clone this project and solve this exercise locally.

Exercise: Top articles

Your task is to implement a function; that will return the top articles from the given list. The top articles are those that have the highest number of views. The returned list should have the same order as the given list. The function should return a list of the given size. If the given list is smaller than the given size, it should return all the articles. If the given list is empty, it should return an empty list.

Starting code:

```
class TopArticlesGenerator(
    private val articles: List<ArticleStatistics>,
) {
    fun topArticles(n: Int): List<ArticleStatistics> = TODO()
}

data class ArticleStatistics(
    val title: String,
    val views: Long,
)
```

Example usage:

```
val generator = TopArticlesGenerator(
    listOf(
        ArticleStatistics("Article 1", 400),
        ArticleStatistics("Article 2", 100),
        ArticleStatistics("Article 3", 200),
        ArticleStatistics("Article 4", 300),
        ArticleStatistics("Article 5", 500),
        ArticleStatistics("Article 6", 0),
    )
)
val topArticles = generator.topArticles(3)
topArticles.onEach { println(it) }
// ArticleStatistics(title=Article 1, views=400)
// ArticleStatistics(title=Article 4, views=300)
// ArticleStatistics(title=Article 5, views=500)
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/TopArticles.kt`. You can clone this project and solve this exercise locally.

Hint: The trick to be able to sort by views and then return the original order is to use `withIndex` function.

Exercise: Refactor collection processing

In your code, you found a complex collection processing implemented using classic Java-like techniques. Your task is to refactor this code to use Kotlin collection processing functions. The function is responsible for finding the best students for internships. This is what is looks like:

```
fun List<StudentGrades>.getBestForScholarship(
    semester: String
): List<StudentGrades> {
    val students = this
    var candidates = mutableListOf<StudentGrades>()
    for (s in students) {
        var ectsPointsGained = 0
        for (g in s.grades) {
            if (g.semester == semester && g.passing) {
                ectsPointsGained += g.ects
            }
        }
        if (ectsPointsGained > 30) {
            candidates.add(s)
        }
    }
    Collections.sort(candidates, { s1, s2 ->
        val difference =
            averageGradeFromSemester(s2, semester) -
                    averageGradeFromSemester(s1, semester)
        if (difference > 0) 1 else -1
    })
    val best = mutableListOf<StudentGrades>()
    for (i in 0 until 10) {
        val next = candidates.getOrNull(i)
        if (next != null) {
            best.add(next)
        }
    }
    return best
}

private fun averageGradeFromSemester(
    student: StudentGrades,
    semester: String
): Double {
    var sum = 0.0
```

```
    var count = 0
    for (g in student.grades) {
        if (g.semester == semester) {
            sum += g.grade
            count++
        }
    }
    return sum / count
}

data class Grade(
    val passing: Boolean,
    var ects: Int,
    var semester: String,
    var grade: Double
)

data class StudentGrades(
    val studentId: String,
    val grades: List<Grade>
)
```

Without changing behavior, refactor those functions.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/StudentGradesIntenship.kt`. You can clone this project and solve this exercise locally.

Hint: To calculate an average of all the numbers in a list, you can use the `average` function.

Exercise: Passing students list

Implement `makePassingStudentsList` function to display a list of students who have got more than 15 points in the semester and a result of at least 50. Display them in a single string, presenting each student in a new line, in alphabetical order (sorting by surname, and for equals surnames by name), in the format: "{name} {surname}, {result}". Starting code:

```
fun List<Student>.makePassingStudentsList(): String = TODO()

data class Student(
    val name: String,
    val surname: String,
    val result: Double,
    val pointsInSemester: Int
)
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/PassingStudents.kt`. You can clone this project and solve this exercise locally.

Exercise: Best students list

Implement `makeBestStudentsList` function to display the best 10 students, so they can get an internship. You should compare them by their result (higher is better). To get an internship, students need to have got at least 30 points in the semester and a result of at least 80. The best student gets 5000 USD, the next 3 get 3000 USD and the next 6 get 1000 USD. Display them in a single string, presenting each student in a new line, in alphabetical order (sorting by surname, and for equals surnames by name), in the format "{name} {surname}, ${internship size}". Starting code:

```
fun List<Student>.makeBestStudentsList(): String = TODO()

data class Student(
    val name: String,
    val surname: String,
    val result: Double,
    val pointsInSemester: Int
)
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/BestStudents.kt`. You can clone this project and solve this exercise locally.

Exercise: Functional Quick Sort

There is a popular exercise in the functional programming community, to implement a simplified version of the Quick Sort algorithm. It is a recursive algorithm, so it is a good exercise to practice recursion. It is also a good exercise to practice functional programming, because it can be implemented in a functional way. The algorithm is as follows: Quick sort should take the first element (pivot), then split the rest to bigger and smaller than pivot, to finally return smaller elements sorted recursively, then pivot, and then bigger elements sorted recursively. Also, if list size is 0 or 1, it is already sorted, so you should return the list itself. The complete implementation should take just a couple of lines.

```
fun <T : Comparable<T>> List<T>.quickSort(): List<T> {
    TODO()
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/QuickSort.kt`. You can clone this project and solve this exercise locally.

Exercise: Powerset

Your task is to implement the `powerset` function, which returns a set of all subsets of a given set. Here are a few examples of how it should work:

```
powerset(setOf()) == setOf(setOf())
powerset(setOf(1)) == setOf(setOf(), setOf(1))
powerset(setOf(1, 2)) == setOf(
    setOf(), setOf(1), setOf(2), setOf(1, 2)
)
powerset(setOf(1, 2, 3)) == setOf(
    setOf(), setOf(1), setOf(2), setOf(3), setOf(1, 2),
    setOf(1, 3), setOf(2, 3), setOf(1, 2, 3)
)
```

Here is the starting point:

```
fun <T> Collection<T>.powerset(): Set<Set<T>> {
    TODO()
}
```

It is easiest to implement this function using recursion. Notice, that a powerset of a set with n elements is a powerset of a set with n-1 elements, plus all elements from the powerset of a set with n-1 elements with the n-th element added.

```
powerset(setOf()) == setOf(setOf())

powerset(setOf(1)) ==
    powerset(setOf()) + powerset(setOf()).eachPlus(1)

powerset(setOf(1,2)) ==
    powerset(setOf(1)) + powerset(setOf(1)).eachPlus(2)

powerset(N) == powerset(N/m) + powerset(N/m).eachPlus(m)
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/Powerset.kt`. You can clone this project and solve this exercise locally.

Exercise: All possible partitions of a set

Your task is to implement the `partitions` function, which returns a set of all possible partitions of a given set. A partition of a set is a grouping of its elements into non-empty subsets, in such a way that every element is included in exactly one subset. Here are a few examples of how it should work:

```
partitions(setOf(1)) == setOf(setOf(setOf(1)))
partitions(setOf(1, 2)) == setOf(
    setOf(setOf(1), setOf(2)),
    setOf(setOf(1, 2))
)
partitions(setOf(1, 2, 3)) == setOf(
    setOf(setOf(1), setOf(2), setOf(3)),
    setOf(setOf(1), setOf(2, 3)),
    setOf(setOf(1, 2), setOf(3)),
    setOf(setOf(1, 2, 3))
)
```

Here is the starting point:

```
fun <T> Collection<T>.partitions(): Set<Set<Set<T>>> = TODO()
```

It is easiest to implement this function using recursion.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/collections/Partitions.kt`. You can clone this project and solve this exercise locally.

Exercise: Understanding sequences

What will be printed by the following code?

```
fun m(i: Int): Int {
    print("m$i ")
    return i * i
}
fun f(i: Int): Boolean {
    print("f$i ")
    return i % 2 == 0
}

fun main() {
    val list = listOf(1, 2, 3, 4)
    list.map(::m).filter(::f) // ?
    list.filter(::f).map(::m) // ?
    val sequence = sequenceOf(1, 2, 3, 4)
    sequence.map(::m).filter(::f).toList() // ?
    sequence.map(::m).filter(::f) // ?
    sequence.map(::m).filter(::f).first() // ?
    sequence.filter(::f).map(::m).toList() // ?

    val sequence2 = list.asSequence().map(::m) // ?
    sequence2.toList() // ?
    sequence2.filter(::f).toList() // ?
}
```

Exercise: HTML table DSL

You are working on a project that requires you to generate HTML tables.

```
fun createTable(): TableBuilder {
    val td1 = TdBuilder()
    td1.text = "A"
    val td2 = TdBuilder()
    td2.text = "B"

    val tr1 = TrBuilder()
    tr1.tds += td1
    tr1.tds += td2

    val td3 = TdBuilder()
    td3.text = "C"
    val td4 = TdBuilder()
    td4.text = "D"

    val tr2 = TrBuilder()
    tr2.tds += td3
    tr2.tds += td4

    val html = TableBuilder()
    html.trs += tr1
    html.trs += tr2
    return html
}
```

You want to make it more readable and easier to use. You decide to create a DSL for it. You want to be able to create a table in the following way:

```
fun createTable(): TableBuilder = table {
    tr {
        td { +"A" }
        td { +"B" }
    }
    tr {
        td { +"C" }
        td { +"D" }
    }
}
```

To make it possible, you need to four functions: `table`, `tr`, `td` and `text`. Some of them need to be added inside builder classes.

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/dsl/Table.kt`. You can clone this project and solve this exercise locally.

Exercise: Creating user table row

As a continuation of the previous exercise, you decided to define a function that generates a table with user details. This is how you implemented it:

```
fun userTable(users: List<User>): TableBuilder = table {
    tr {
        td { +"Id" }
        td { +"Name" }
        td { +"Points" }
        td { +"Category" }
    }
    for (user in users) {
        userRow(user)
    }
}
```

The only thing left is to define `userRow` function. This is your task. It should generate a table row with user details. This is how it should work:

```
val users = listOf(
    User("1", "Randy", 2, "A"),
    User("4", "Andy", 4, "B"),
    User("3", "Mandy", 1, "C"),
    User("5", "Cindy", 5, "A"),
    User("2", "Lindy", 3, "B"),
)
val table = userTable(users)
println(table)
// <table>
// <tr><td>Id</td><td>Name</td>
// <td>Points</td><td>Category</td></tr>
// <tr><td>1</td><td>Randy</td><td>2</td><td>A</td></tr>
// <tr><td>4</td><td>Andy</td><td>4</td><td>B</td></tr>
// <tr><td>3</td><td>Mandy</td><td>1</td><td>C</td></tr>
// <tr><td>5</td><td>Cindy</td><td>5</td><td>A</td></tr>
// <tr><td>2</td><td>Lindy</td><td>3</td><td>B</td></tr>
// </table>
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/dsl/UsersTable.kt`. You can clone this project and solve this exercise locally.

Exercise: Using scope functions

See the below implementation of `StudentService`. Modify it to use scope functions. As a result, all the methods should be single-expression functions.

```
class StudentService(
    private val studentRepository: StudentRepository,
    private val studentFactory: StudentFactory,
    private val logger: Logger,
) {
    fun addStudent(addStudentRequest: AddStudentRequest):Student?{
        val student = studentFactory
            .produceStudent(addStudentRequest)
            ?: return null
        studentRepository.addStudent(student)
        return student
    }

    fun getStudent(studentId: String): ExposedStudent? {
        val student = studentRepository.getStudent(studentId)
            ?: return null

        logger.log("Student found: $student")
        return studentFactory.produceExposed(student)
    }

    fun getStudents(semester: String): List<ExposedStudent> {
        val request = produceGetStudentsRequest(semester)
        val students = studentRepository.getStudents(request)
        logger.log("${students.size} students in $semester")
        return students
            .map { studentFactory.produceExposed(it) }
    }

    private fun produceGetStudentsRequest(
        semester: String,
    ): GetStudentsRequest {
        val request = GetStudentsRequest()
        request.expectedSemester = semester
        request.minResult = 3.0
        return request
    }
}
```

> Beware! The form after transformation is shorter but not necessarily better. It is less readable for less experienced Kotlin developers, and it is harder to debug. Outside this exercise, use scope functions with caution.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/scope/Scope.kt`. You can clone this project and solve this exercise locally.

Exercise: orThrow

In my everyday practice, I've noticed that I sometimes need a function that in the middle of a multiline expression can be used to throw an exception if a value is `null`. So I defined it and called it `orThrow`. This is how its usage looks like:

```
fun getUser(userId: String) = userRepository
    .getUser(userId)
    .orThrow { UserNotFoundException(userId) }
    .also { log("Found user: $it") }
    .toUserJson()
```

Your task is to implement `orThrow` function. It should throw the exception specified in its lambda argument if the value is `null`. Otherwise, it should return the value typed as non-nullable.

Unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/scope/orThrow.kt`. You can clone this project and solve this exercise locally.

Exercise: Logger

You are working on a pet store project. You implemented a function that is responsible for adding new pets to the database. Now your task is to add logging to this function. You want to log the following messages:

- Always start with info “Adding pet with name {name}”
- If the pet was added successfully, log info “Added pet with id {id}”
- If the pet was not added because of a conflict, log warning “There already is pet named {name}”
- If the pet was not added because of database error, log error “Failed to add pet with name {name}”

You want to add logger **using context receiver** of `addPet` function. The type of this receiver should be `Logger`, and inside `addPet` you should use its methods to log messages.

```
class PetStore(
    private val database: Database,
) {
    fun addPet(
        addPetRequest: AddPetRequest,
    ): Pet? {
        return try {
            database.addPet(addPetRequest)
        } catch (e: InsertionConflictException) {
            null
        } catch (e: Exception) {
            null
        }
    }
}
```

This is what used classes and interfaces look like:

```
data class AddPetRequest(val name: String)
data class Pet(val id: Int, val name: String)
class InsertionConflictException : Exception()

interface Database {
    fun addPet(addPetRequest: AddPetRequest): Pet
}

interface Logger {
    fun logInfo(message: String)
    fun logWarning(message: String)
    fun logError(message: String)
}
```

This is an example usage, showing what should be logged in different cases:

```
fun main(): Unit = with(ConsoleLogger()) {
    val database = RandomDatabase()
    val petStore = PetStore(database)
    petStore.addPet(AddPetRequest("Fluffy"))
    // [INFO] - Adding pet with name Fluffy
    // [INFO] - Added pet with id -81731626
    // or
    // [WARNING] - There already is pet named Fluffy
    // or
    // [ERROR] - Failed to add pet with name Fluffy
}

class RandomDatabase : Database {
    override fun addPet(addPetRequest: AddPetRequest): Pet =
        when {
            Random.nextBoolean() ->
                Pet(1234, addPetRequest.name)
            Random.nextBoolean() ->
                throw InsertionConflictException()
            else -> throw Exception()
        }
}

class ConsoleLogger : Logger {
    override fun logInfo(message: String) {
        println("[INFO] - $message")
    }
```

```
    override fun logWarning(message: String) {
        println("[WARNING] - $message")
    }

    override fun logError(message: String) {
        println("[ERROR] - $message")
    }
}
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/context/PetStore.kt`. You can clone this project and solve this exercise locally.

Final Project: UserService

The project will emulate a real-life task, typical to backend development, but similar to what is done on the domain layer in Android. We will create a service that will allow us to manage users. It will be a simple service, but it will be a good example of how to use Kotlin to create a domain layer.

All the files for solving this exercise can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the `functional/project` package. You should clone this project and solve this exercise locally.

Your task is to implement methods from classes `UserService`, `UserDtoFactory`, and `RealUserKeyGenerator`.

`UserService` is a service that will allow us to manage users. It should have the following properties and methods:

- `userByIdCache` and `userByKeyCache` - properties that define this class caching mechanism. They should use `cache` DSL builder to configure a cache with 1 minute of expiration time after both read and write, and it should configure how data should be loaded into the cache. The `userByIdCache` should load data using the `userRepository.getUser` method, and the `userByKeyCache` should load data using the `userRepository.getUserByKey` method. In both cases, when data is loaded, it should also be stored in the other cache.
- `getUser(id: String): User?` - returns user with the given id or `null` if there is no such user.
- `getUserByKey(key: String): User?` - returns user with the given key or `null` if there is no such user.
- `getToken(email: String, passwordHash: String): String` - returns a token for the user with the given email if the password hash is correct. If there is no such user or the password hash is incorrect, it throws an exception with the message "Wrong email or password".
- `updateUser(token: String, userPatch: UserPatch): User` - updates the user with the given token using the given patch. If there is no such user, it throws an exception with message "User not found".
- `addUser(token: String, addUser: AddUser): User` - adds a new user using the given data. Only admin can add users, so if the user with the given token is not an admin, it throws an exception.
- `userStatistics(token: String): UserStatistics` - returns statistics about users. If the user with the given token is not an admin, it throws an exception.

`UserDtoFactory` is a factory that creates `UserDto` objects. It should have the method `produceUserDto(addUser: AddUser): UserDto` that creates a `UserDto` object using the given `AddUser` object. It should use the `timeProvider` to get the current time, the `uuidGenerator` to generate an id and a key, and the `userKeyGenerator` to generate a key.

`RealUserKeyGenerator` is a generator that creates keys for users. It should have the method `findPublicKey(name: String, surname: String): String?` to return a key for the given name and surname. If there is no such key, it shoudld return `null`. It should use the `userRepository` to check if the key is available. This function should try different combinations of name and surname in a form transformed to a correct key value. Key should be lowercase and only include characters or digits. It should not include any special characters. It should also not be shorter than `4` characters. If the key is not available, it should return `null`. It should try to generate a key in the following order, but if none of those keys is available, it should generate a random key using the `uuidGenerator`:

- `"$name$surname"`
- `"$surname$name"`
- `"$name${surname.first()}"`
- `"${name.first()}$surname"`
- `"$surname${name.first()}"`
- `"${surname.first()}$name"`

More detailed explanation for each of those functions can be found on their comments. You can also find unit tests for those functions in their files. Remember to run them to check if your solution is correct.

Kotlin Coroutines exercises

Exercise: Factorial sequence

Implement a program that calculates and displays the factorial of a given number. The factorial of a number is the product of all positive integers up to that number. So, the first 5 numbers in this sequence should be:

- `0! = 1`
- `1! = 1`
- `2! = 1 * 2 = 2`
- `3! = 1 * 2 * 3 = 6`
- `4! = 1 * 2 * 3 * 4 = 24`

This sequence should be infinite. The result should be of type `BigInteger`.

```
val factorial: Sequence<BigInteger> = sequence {
    TODO()
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/sequences/Factorial.kt`. You can clone this project and solve this exercise locally.

Exercise: Prime numbers sequence

Implement a program that calculates and displays prime numbers. A prime number is a number that is divisible only by itself and 1. So, the first 5 numbers in this sequence should be 2, 3, 5, 7, 11...

This sequence should be infinite. The result should be of type `BigInteger`. The first prime number is 2.

The easiest way to check if a number is prime is to check if it is divisible by any of the previous prime numbers. If it is not divisible by any of them, then it is prime. This is not the most efficient way to check if a number is prime, but it is good enough for this exercise.

```
val primes: Sequence<BigInteger> = sequence {
    TODO()
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/sequences/Prime.kt`. You can clone this project and solve this exercise locally.

Exercise: Callback function wrappers

In the `FetchTasksUseCase` class, implement the following functions that should each fetch tasks using `fetchTasks` method from `callbackUseCase`, but return it using different types:

- `fetchTasks`, which should return tasks or throw an exception in the case of an error.
- `fetchTasksResult`, which should return `Result` representing either a success or a failure.
- `fetchTasksOrNull`, which should return tasks or `null` in the case of an error.

```
class FetchTasksUseCase(
    private val callbackUseCase: FetchTasksCallbackUseCase
) {
    @Throws(ApiException::class)
    suspend fun fetchTasks(): List<Task> =
        TODO()
    suspend fun fetchTasksResult(): Result<List<Task>> =
        TODO()
    suspend fun fetchTasksOrNull(): List<Task>? =
        TODO()
}

interface FetchTasksCallbackUseCase {
    fun fetchTasks(
        onSuccess: (List<Task>) -> Unit,
        onError: (Throwable) -> Unit
    ): Cancellable
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/suspension/FetchTasksUseCase.kt`. You can clone this project and solve this exercise locally.

Exercise: Continuation storage

Implement the `continuationSteal` function, which should print "Before" and "After" in the console; in between them, it should suspend the coroutine this function is executed on and store the continuation in the `continuation` property. After resuming, it should print the string this continuation was resumed with.

```
var continuation: Continuation<String>? = null

suspend fun continuationSteal(console: Console) {
    console.println("Before")
    // TODO
    console.println("After")
}

interface Console {
    fun println(text: Any?)
}
```

Example usage:

```
fun main(): Unit = runBlocking<Unit> {
    launch {
        continuationSteal(object : Console {
            override fun println(text: Any?) {
                kotlin.io.println(text)
            }
        })
    }
    delay(1000)
    continuation?.resume("This is some text")
}
// Before
// (1 sec)
// This is some text
// After
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/suspension/ContinuationStorage.kt`. You can clone this project and solve this exercise locally.

Hint: Use `suspendCancellableCoroutine` instead of `suspendCoroutine`.

Exercise: What is stored by a continuation?

Consider you started the following code in debug mode, and you are currently stopped at the line marked with `// HERE`. What is stored in the `continuation` variable?

```
import kotlin.coroutines.*

suspend fun a() {
    val c = 12345
    suspendCoroutine { it.resume(Unit) }
    suspendCoroutine { continuation ->
        continuation.resume(c) // HERE
    }
    println(c)
}

suspend fun main() {
    val a = "ABC"
    val b = listOf(1, 2, 3)
    println(b)
    a()
    println(a)
}
```

Exercise: UserDetailsRepository

In the `UserDetailsRepository` class, implement the `getUserDetails` function, which return details of the current user. It should:

- Check if the user is available in the database. If so, it should return this data.
- If the user is not available in the database, it should asynchronously fetch the user details from the client, use them to create a `UserDetails` object, and return it.
- After fetching new user data, it should asynchronously save it in the database. `getUserDetails` function should be able to complete its execution without waiting for the data to be saved.

Starting code:

```
class UserDetailsRepository(
    private val client: UserDataClient,
    private val userDatabase: UserDetailsDatabase,
    private val backgroundScope: CoroutineScope,
) {
    suspend fun getUserDetails(): UserDetails {
        TODO()
    }
}

interface UserDataClient {
    suspend fun getName(): String
    suspend fun getFriends(): List<Friend>
    suspend fun getProfile(): Profile
}

interface UserDetailsDatabase {
    suspend fun load(): UserDetails?
    suspend fun save(user: UserDetails)
}

data class UserDetails(
    val name: String,
    val friends: List<Friend>,
    val profile: Profile
)

data class Friend(val id: String)
data class Profile(val description: String)
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/starting/UserDetailsRepository.kt`. You can clone this project and solve this exercise locally.

Exercise: BestStudentUseCase

In the `BestStudentUseCase` class, implement the `getBestStudent` function, which fetches all students from the given semester and then finds the one with the best result. For this, it first needs to use `StudentsRepository` to find the IDs of students in the semester, after which it needs to fetch these users' details. Fetching details should be done asynchronously. The best user is the one with the highest `result` value. If there are no students in the given semester, the function should throw `IllegalStateException`.

```
class BestStudentUseCase(
    private val repo: StudentsRepository
) {
    suspend fun getBestStudent(semester: String): Student = TODO()
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/starting/BestStudent.kt`. You can clone this project and solve this exercise locally.

Exercise: CommentService

Implement the following functions from the `CommentService` class:

- `addComment` - should read a user id from a token, transform the body to a `CommentDocument`, and add it to the comment repository.
- `getComments` - should get comments with users and return them as a collection.

```
class CommentService(
    private val commentRepository: CommentRepository,
    private val userService: UserService,
    private val commentFactory: CommentFactory
) {
    suspend fun addComment(
        token: String,
        collectionKey: String,
        body: AddComment
    ) {
        TODO()
    }

    suspend fun getComments(
        collectionKey: String
    ): CommentsCollection = TODO()
}
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/comment/CommentService.kt`. You can clone this project and solve this exercise locally.

You can assume that `findUserById` from `userService` can be called multiple times for the same user id because it is cached. Alternatively, you can refactor this service to make sure it is not called more than once for the same id by the same `getComments` call. The second option is more complex.

Exercise: mapAsync

Practice shows that many projects require asynchronous mapping of elements in a collection. To avoid repeating this pattern, implement an `mapAsync` function, which should map all elements in a collection asynchronously.

```
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/recipes/MapAsync.kt`. You can clone this project and solve this exercise locally.

Exercise: Understanding context propagation

Fill the gaps in the following code. Both the `log` and `slog` functions should print the name of the coroutine they are called from, and message from argument.

```
suspend fun log(msg: String) {
    val name = ___[CoroutineName]?.name
    println("[$name] $msg")
}

fun CoroutineScope.slog(msg: String) {
    val name = ___[CoroutineName]?.name
    println("[$name] $msg")
}

suspend fun main() = withContext(CoroutineName("Outer")) {
    log("Starting") // [___] Starting
    launch(CoroutineName("Inner")) {
        slog("A") // [___] A
        launch {
            log("B") // [___] B
        }
    }
    launch {
        slog("C") // [___] C
    }
    GlobalScope.launch {
        log("D") // [___] D
    }
    slog("Ending") // [___] Ending
}
```

Exercise: CounterContext

Implement a `CounterContext` class that is a mutable counter and can always be asked for a new value by using the `next` function. It should be possible to create multiple instances of `CounterContext`, each of which should have its own counter.

Example usage:

```
fun main(): Unit = runBlocking(CounterContext()) {
    println(coroutineContext[CounterContext]?.next()) // 0
    println(coroutineContext[CounterContext]?.next()) // 1
    launch {
        println(coroutineContext[CounterContext]?.next())// 2
        println(coroutineContext[CounterContext]?.next())// 3
    }
    launch(CounterContext()) {
        println(coroutineContext[CounterContext]?.next())// 0
        println(coroutineContext[CounterContext]?.next())// 1
    }
}
```

The simplest way to create a custom context is by creating a class that extends the `AbstractCoroutineContextElement` class. This class implements the `CoroutineContext.Element` interface, and it is a good starting point for creating custom contexts. It requires the value that is used as a key for this context. It is recommended to use a companion object for this purpose. Here is an example of a simple context:

```
class MyContext : AbstractCoroutineContextElement(MyContext){
   companion object : CoroutineContext.Key<MyContext>
}
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/context/CounterContext.kt`. You can clone this project and solve this exercise locally.

Exercise: Using dispatchers

For each of the functions below, decide if you need to set a dispatcher, and if so, which one:

- `createInvoice` - prepares an invoice and sends it to the SaaS API using the Retrofit library. In this library, we define suspending functions. The `toFakturowniaInvoiceData` function is a simple mapping.
- `sendEmail` - prepares an email and sends it using the `api` method of the `SendGrid` class. This is a blocking operation that returns a `Response` object.
- `getUserOrders` - gets orders made by the user from the database using the MongoDB Kotlin driver. `toList` is a suspending function that returns a list of orders.
- `upscaleImage` - a function that uses the TensorFlow library to upscale an image.

```
override suspend fun createInvoice(
    invoiceData: InvoiceData,
) {
    invoiceApi.postInvoice(
        invoiceData.toFakturowniaInvoiceData()
    )
}

interface InvoiceApi {
    @POST("invoices.json")
    suspend fun postInvoice(
        @Body invoice: SendInvoiceData
    ): InvoiceCreationResponse
}

private val invoiceApi = retrofit2.Retrofit.Builder()
    .baseUrl(invoiceBaseUrl)
    .build()
    .create(InvoiceApi::class.java)

private val sendGrid = SendGrid(API_KEY)

suspend fun sendEmail(email: Email) {
    val request = Request().apply {
        method = Method.POST
        endpoint = "mail/send"
        body = email.toSendGridEmail()
    }
    sendGrid.api(request)
}
```

```
private val orderCollection = mongoClient
    .getDatabase("shop")
    .getCollection<Order>("orders")

suspend fun getUserOrders(userId: String): List<Order> =
    orderCollection
        .find(eq("userId", userId))
        .toList()

val model = TensorFlowModel()

suspend fun upscaleImage(image: Image): Image =
    model.upscale(image)
```

Exercise: DiscNewsRepository

`DiscNewsRepository` is a simple class that uses `DiscReader` to read stored news from disk. The problem is that reading data from disk is a blocking operation, therefore the current implementation of `DiscNewsRepository` blocks threads in suspending functions, which leads to performance problems in your application. Fix this problem! Assume that `getNews` is used intensively and you want to support 200 parallel reads. Starting code:

```
class DiscNewsRepository(
    private val discReader: DiscReader
) : NewsRepository {
    override suspend fun getNews(newsId: String): News {
        val (title, content) = discReader.read("user/$newsId")
        return News(title, content)
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/dispatcher/DiscNewsRepository.kt`. You can clone this project and solve this exercise locally.

Exercise: Experiments with dispatchers

Experiment with how dispatcher choice influences execution time when you start 100 coroutines, each of which performs the following operations:

- CPU-intensive `cpuHeavy`
- Blocking `Thread.sleep(1000)`
- Suspending `delay(1000)`

```
val dispatcher = Dispatchers.IO.limitedParallelism(1)
//val dispatcher = Dispatchers.Default
//val dispatcher = Dispatchers.IO
//val dispatcher = Dispatchers.IO.limitedParallelism(100)

val operation = ::cpu1
//val operation = ::blocking
//val operation = ::suspending

fun cpu1() {
    var i = Int.MAX_VALUE
    while (i > 0) i -= if (i % 2 == 0) 1 else 2
}

fun blocking() {
    Thread.sleep(1000)
}

suspend fun suspending() {
    delay(1000)
}
```

Test the following dispatchers:

- Dispatcher limited to 1 thread
- `Dispatchers.Default`
- `Dispatchers.IO`
- Dispatcher limited to 100 threads

Test all combinations and fill the table below with the results. Then, try to explain the results.

Dispatcher	CPU-intensive	Blocking	Suspending
1 thread			
Dispatchers.Default			
Dispatchers.IO			
100 threads			

You can use the following code to measure execution time:

```
suspend fun main() = measureTimeMillis {
    coroutineScope {
        repeat(100) {
            launch(dispatcher) {
                operation()
                println("Done $it")
            }
        }
    }
}.let { println("Took $it") }
```

Starting code and example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/dispatcher/Experiments.kt`. You can clone this project and solve this exercise locally.

Exercise: Correct mistakes with cancellation

All below code snippets contain mistakes related to cancellation. Name them and explain why they are mistakes. Write a corrected version of the code.

Function `updateUser` contains **four** mistakes. Three are related to cancellation.

```
suspend fun updateUser() {
    val user = readUser() // blocking
    val userSettings = readUserSettings(user.id) // blocking

    try {
        updateUserInDatabase(user, userSettings) // suspending
    } catch (e: CancellationException) {
        revertUnfinishedTransactions() // suspending
    }
}
```

Function `sendSignature` contains **three** mistakes. Two are related to cancellation.

```
suspend fun sendSignature(file: File) {
    try {
        val content = file.readText()
        val signature = calculateSignature(content)
        sendSignature(signature) // suspending
    } catch (e: Exception) {
        println("Error while sending signature: ${e.message}")
        e.printStackTrace()
    } finally {
        file.delete()
    }
}
```

Function `trySendUntilSuccess` contains **one** mistake related to cancellation.

```
suspend fun trySendUntilSuccess() {
    var success = false
    do {
        try {
            send()
            success = true
        } catch (e: Exception) {
            println("Error while sending: ${e.message}")
            e.printStackTrace()
        }
    } while (!success)
}
```

Exercise: NotificationSender

Implement `NotificationSender` so that it sends notifications concurrently using `NotificationClient`. Notifications should be sent asynchronously using the `scope` property. This scope should include `dispatcher`. When an exception occurs while sending a notification, it should be collected using `ExceptionCollector` and it should not prevent other notifications from being sent. When `cancel` method is called, it should cancel all coroutines that are sending notifications, but it should not prevent future notifications being sent.

```
class NotificationSender(
    private val client: NotificationClient,
    private val exceptionCollector: ExceptionCollector,
    dispatcher: CoroutineDispatcher,
) {
    val scope: CoroutineScope = TODO()

    fun sendNotifications(notifications: List<Notification>) {
        // TODO
    }

    fun cancel() {
        // TODO
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/scope/NotificationSender.kt`. You can clone this project and solve this exercise locally.

Exercise: BaseViewModel

Implement a `BaseViewModel` class that is a base class for view models in MVVM architecture. It should have a `scope` property that provides a scope for starting coroutines. It should also have an `onCleared` function that cancels all coroutines started in this scope. In the case of an exception in a coroutine, other coroutines should not be interrupted, but this error should be emitted from `exceptions` using `trySendBlocking` function. All coroutines should run on the main thread by default.

```
abstract class BaseViewModel : ViewModel() {
    private val _exceptions = Channel<Throwable>(Channel.UNLIMITE\
D)
    val exceptions: Flow<Throwable> = _exceptions.receiveAsFlow()

    val scope: CoroutineScope = TODO()
}
```

Example usage:

```
class MainViewModel(
    private val userRepo: UserRepository,
    private val newsRepo: NewsRepository
) : BaseViewModel() {
    private val _userData = MutableStateFlow<UserData?>(null)
    val userData: StateFlow<UserData?> = _userData

    private val _news = MutableStateFlow(emptyList<News>())
    val news: StateFlow<List<News>> = _news

    init {
        scope.launch {
            _userData.value = userRepo.getUser()
        }
        scope.launch {
            _news.value = newsRepo.getNews()
                .sortedByDescending { it.date }
        }
    }
}
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/scope/BaseViewModel.kt`. You can clone this project and solve this exercise locally.

Exercise: UserDownloader

This `UserDownloader` class is used to fetch users from an API, followed by a list of all downloaded users. The problem is that it is not thread-safe; therefore, when you use the `getUser` function from multiple coroutines, you will have concurrency problems and the list of downloaded users will not be complete. Fix this problem using the following techniques:

- Use a dispatcher limited to a single thread and use a read-only list to store users.
- Use a dispatcher limited to a single thread and keep using a mutable list to store users.
- Use a synchronized block to protect the shared state and keep using a mutable list to store users.
- Use a concurrent collection to store users.

```
class UserDownloader(private val api: NetworkService) {
    private val users = mutableListOf<User>()

    fun downloaded(): List<User> = users.toList()

    suspend fun getUser(id: Int) {
        val newUser = api.getUser(id)
        users += newUser
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/Downloader.kt`. You can clone this project and solve this exercise locally.

Exercise: CompanyDetailsRepository

The `CompanyDetailsRepository` class is used to fetch company details from an API. The problem is that the current implementation is not thread-safe and exposes internal collection. Fix those problems using the following techniques:

- Use `synchronized` block to protect the shared state and keep using a mutable map to store company details.
- Use `synchronized` block to protect the shared state and use a read-only map to store company details.
- Use a dispatcher limited to a single thread and keep using a mutable map to store company details.
- Use a dispatcher limited to a single thread and use a read-only map to store company details.
- Use `AtomicReference` and read-only map to store company details (that should work, but that is certainly not a good solution).
- Use a concurrent collection to store company details.
- Use a mutex and keep using a mutable map to store company details.

None of those solutions can guarantee that one element is fetched only once. To achieve this, implement solutions using the following techniques:

- Use a concurrent collection with suspending lazy objects to store company details (before doing this one, first finish *Suspended lazy* exercise).
- Use caching library to cache company details (Aedile is a good choice).

Starting code:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher
) {
    private val details = mutableMapOf<Company, CompanyDetails>()

    suspend fun getDetails(company: Company): CompanyDetails {
        val current = getDetailsOrNull(company)
        if (current == null) {
            val companyDetails = client.fetchDetails(company)
            details[company] = companyDetails
            return companyDetails
        }
        return current
    }
```

```
    fun getDetailsOrNull(company: Company): CompanyDetails? =
        details[company]

    fun getReadyDetails(): Map<Company, CompanyDetails> =
        details

    fun clear() {
        details.clear()
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/CompanyDetailsRepository.kt`. You can clone this project and solve this exercise locally.

On the project, you can also find performance tests that can help you test the performance of each solution. Use them, and note down the results.

Exercise: CancellingRefresher

You want to make sure that if a new refresh is started, the previous one is cancelled. You have the following code:

```
class CancellingRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend () -> Unit,
) {
    private var refreshJob: Job? = null

    fun refresh() {
        refreshJob?.cancel()
        refreshJob = scope.launch {
            refreshData()
        }
    }
}
```

The problem is that this implementation is not correct, because if started concurrently, `refresh` function might not cancel some coroutines. Your task is to fix it.

> This is a popular pattern on Android, but there in most cases there is no need for synchronization, because UI handlers are always called on the main thread.

Make sure that the `refresh` function is thread-safe using the following techniques:

- Using `synchronized` block.
- Using `Mutex`.
- Using a dispatcher limited to a single thread.

Compare how much time your unit tests take for each solution.

Check if you can solve this problem using a concurrent set of jobs, just remember to remove jobs that are cancelled already.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/CancellingRefresher.kt`. You can clone this project and solve this exercise locally.

Exercise: TokenRepository

Correct `TokenRepository` to make sure that only one call to `fetchToken` is made at a time. At the same time, make sure that `invalidateToken` does not invalidate the token that is being fetched, only the one that is already stored.

```
class TokenRepository(
    private val client: TokenClient,
    private val timeProvider: TimeProvider
) {
    private var token: Token? = null

    suspend fun getToken(): Token {
        val currentToken = token
        if (currentToken != null &&
            currentToken.expiration > timeProvider.now()) {
            return currentToken
        }
        val newToken = client.fetchToken()
        token = newToken
        return newToken
    }

    fun invalidateToken() {
        token = null
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/TokenRepository.kt`. You can clone this project and solve this exercise locally.

Exercise: Suspended lazy

Implement `suspendLazy`, which creates a lazy value that is computed when it is needed for the first time. The main difference between `suspendLazy` and `lazy` is that `suspendLazy` has a suspending initializer and a function to get a value.

```
val config: suspend () -> Config = suspendLazy {
    service.fetchConfig()
}

suspend fun getConfig(): UserData = config()
```

Starting code:

```
fun <T> suspendLazy(initializer: suspend () -> T):SuspendLazy<T> =
    TODO()

interface SuspendLazy<T> : suspend () -> T {
    val isInitialized: Boolean
    fun valueOrNull(): T?
    override suspend operator fun invoke(): T
}
```

Make sure that `initializer` is called only once, even if `get` is called from multiple coroutines.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/recipes/SuspendLazy.kt`. You can clone this project and solve this exercise locally.

Exercise: mapAsync with concurrency limit

Implement a `mapAsync` function that should work like the `map` function but should be able to run multiple coroutines concurrently. The number of concurrent coroutines should be limited by the `concurrency` parameter. If `concurrency` is `1`, then it should work like the `map` function. If `concurrency` is `2`, then it should run `2` coroutines concurrently, and so on. If `concurrency` is `0` or less, then it should throw `IllegalArgumentException`.

```
suspend fun <T, R> Iterable<T>.mapAsync(
    concurrency: Int,
    transformation: suspend (T) -> R
): List<R> = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/recipes/MapAsyncLimited.kt`. You can clone this project and solve this exercise locally.

Exercise: Test UserDetailsRepository

In a previous exercise, we implemented a `UserDetailsRepository` class. Here is a possible implementation:

```
class UserDetailsRepository(
    private val client: UserDataClient,
    private val userDatabase: UserDetailsDatabase,
    private val backgroundScope: CoroutineScope,
) {
    suspend fun getUserDetails(): UserDetails =
        coroutineScope {
            val stored = userDatabase.load()
            if (stored != null) {
                return@coroutineScope stored
            }
            val name = async { client.getName() }
            val friends = async { client.getFriends() }
            val profile = async { client.getProfile() }
            val details = UserDetails(
                name = name.await(),
                friends = friends.await(),
                profile = profile.await(),
            )
            backgroundScope.launch { userDatabase.save(details) }
            details
        }
}
```

Your task is to implement a single unit test, that verifies the following behavior:

- should fetch name, friends, and profile asynchronously,
- should return fetched details,
- should save details to the database asynchronously,
- should load details from the database if they are already there.

Here is your starting code to modify:

Starting code:

```
@Test
fun `should fetch details asynchronously`() = runTest {
    // given
    val client = object : UserDataClient {
        override suspend fun getName(): String {
            // TODO
            return "Ben"
        }

        override suspend fun getFriends(): List<Friend> {
            // TODO
            return listOf(Friend("friend-id-1"))
        }

        override suspend fun getProfile(): Profile {
            // TODO
            return Profile("Example description")
        }
    }
    var savedDetails: UserDetails? = null
    val database = object : UserDetailsDatabase {
        override suspend fun load(): UserDetails? {
            // TODO
            return savedDetails
        }

        override suspend fun save(user: UserDetails) {
            // TODO
            savedDetails = user
        }
    }

    val repo: UserDetailsRepository = TODO()

    // when
    val details = repo.getUserDetails()

    // then data are fetched asynchronously
    // TODO

    // when all children are finished
    // TODO

    // then data are saved to the database
```

```
    // TODO

    // when getting details again
    // TODO

    // then data are loaded from the database
    // TODO
}
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/test/TestUserDetailsRepository.kt`. You can clone this project and solve this exercise locally.

Exercise: Testing mapAsync

Your task is to test the `mapAsync` function, which maps elements of a collection asynchronously.

```
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> = coroutineScope {
    map { async { transformation(it) } }
        .awaitAll()
}
```

You should test the following cases:

- should behave like a regular map for lists and sets
- should map asynchronously
- should keep elements' order
- should support context propagation
- should support cancellation
- should immediately throw exception it it occurs in transformation

To verify that your unit tests are correct, check if the following implementations fail the relevant tests:

```
// does not map asynchronously
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> =
    map { transformation(it) }

// does not keep elements order
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> = this
    .asFlow()
    .flatMapMerge { flow { emit(transformation(it)) } }
    .toList()

// does support context propagation or cancellation
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> = coroutineScope {
```

```
    map { GlobalScope.async { transformation(it) } }
        .awaitAll()
}

// does not immediately throw an exception from transformation
// (waits until the failing coroutine is awaited)
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> = supervisorScope {
    map { async { transformation(it) } }
        .map { it.await() }
}
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/test/MapAsyncTest.kt`. You can clone this project and solve this exercise locally.

Exercise: Testing the NotificationSender class

Your task is to test the following implementation of `NotificationSender`:

```
class NotificationSender(
    private val client: NotificationClient,
    private val exceptionCollector: ExceptionCollector,
    dispatcher: CoroutineDispatcher,
) {
    private val exceptionHandler =
        CoroutineExceptionHandler { _, throwable ->
            exceptionCollector.collectException(throwable)
        }
    val scope: CoroutineScope = CoroutineScope(
        SupervisorJob() + dispatcher + exceptionHandler
    )

    fun sendNotifications(notifications: List<Notification>) {
        notifications.forEach { notification ->
            scope.launch {
                client.send(notification)
            }
        }
    }

    fun cancel() {
        scope.coroutineContext.cancelChildren()
    }
}
```

You should test the following cases:

- should send notifications concurrently
- should cancel all coroutines when `cancel` is called
- should not cancel other sending processes when one of them fails
- should collect exceptions from all coroutines

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/test/NotificationSenderTest.kt`. You can clone this project and solve this exercise locally.

Exercise: Testing a View Model

Clone MarcinMoskala/testing-viewmodel project from GitHub. It contains a simple Android app that uses `UserListViewModel` to manage its main screen state. Implement test functions for the `UserListViewModel` class using fakes in `UserListViewModelTest` or using mocks in `UserListViewModelMocksTest`.

Exercise: UserRefresher

You want to make sure that refreshing is done one at a time. You want to make sure that if there are multiple refreshes, they will be executed synchronously, so one after another. You have the following code:

```
class UserRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend (Int) -> Unit,
) {
    private var refreshJob: Job? = null

    suspend fun refresh(userId: Int) {
        refreshJob?.join()
        refreshJob = scope.launch {
            refreshData(userId)
        }
    }
}
```

The problem is that this implementation is not correct, because if it is started concurrently, you might have two coroutines running at the same time.

There are two ways to solve this problem:

- Using `Channel` and a single coroutine to handle data refresh.
- Using `Mutex` to synchronize refreshes.

Solve this problem using both techniques.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/UserRefresher.kt`. You can clone this project and solve this exercise locally.

Exercise: Cafeteria simulation

Your task is to simulate a cafeteria. Your cafeteria should have four baristas: Alice, Bob, Celine and Dave. All the baristas can work at the same time. They can prepare latte or espresso. Preparing an espresso requires grinding coffee beans and making espresso. Preparing a latte requires grinding coffee beans, making espresso, and adding milk. Each of these processes takes 3 seconds. You have only one cashier, who can announce that a coffee is ready but cannot announce it more than once per second. Each barista can prepare only one coffee at a time. If a customer orders a coffee and all baristas are busy, they have to wait until one of them is free. You should communicate coffee requests using the terminal. As a result, you should be able to see when each coffee is ready and what each barista is doing at a given moment. Here is what an example simulation should look like:

```
Welcome to Dream Coffee!
Press E to get espresso, L to get latte.
E
Order for ESPRESSO sent
Alice: Grinding coffee...
(3 seconds later)
Alice: Making espresso...
(3 seconds later)
Coffee Espresso is ready
L
Order for LATTE sent
Bob: Grinding coffee...
L
Order for LATTE sent
Celine: Grinding coffee...
(3 seconds later)
Bob: Making espresso...
Celine: Making espresso...
(3 seconds later)
Bob: Brewing milk...
Celine: Brewing milk...
(3 seconds later)
Coffee Latte is ready
(1 second later)
Coffee Latte is ready
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/channel/Cafeteria.kt`. You can clone this project and solve this exercise locally.

Exercise: raceOf

You implement an application that can take data from two sources and should always choose the faster response. For that, you decide to implement a `raceOf` function that starts two or more asynchronous computations and returns the result of the first one that completes. This is an example of how it could work:

```
suspend fun fetchUserData(): UserData = raceOf(
    { service1.fetchUserData() },
    { service2.fetchUserData() }
)
```

Implement a `raceOf` function that takes a list of suspending functions and returns the result of the first one that completes. Use the `select` expression to implement it.

```
suspend fun <T> raceOf(
    racer: suspend CoroutineScope.() -> T,
    vararg racers: suspend CoroutineScope.() -> T
): T = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/recipes/RaceOf.kt`. You can clone this project and solve this exercise locally.

Exercise: Flow utils

Implement the following elements:

- an `infiniteFlow` property that produces an infinite flow of `Unit` elements.
- a `neverFlow` property that produces a flow that never emits anything.
- an `everyFlow` function that produces a flow that emits `Unit` every `timeMillis` parameter milliseconds.
- a `flowOf` function that produces a flow that emits the result of a `lambda` parameter of type `suspend () -> T`.
- a `flowOfFlatten` function that produces a flow that emits all the elements emitted by the result of a `lambda` parameter of type `suspend () -> Flow<T>`.

```
val infiniteFlow: Flow<Unit> = TODO()

val neverFlow: Flow<Nothing> = TODO()

fun everyFlow(timeMillis: Long): Flow<Unit> = TODO()

fun <T> flowOf(lambda: suspend () -> T): Flow<T> = TODO()

fun <T> flowOfFlatten(
    lambda: suspend () -> Flow<T>
): Flow<T> = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/FlowUtils.kt`. You can clone this project and solve this exercise locally.

Exercise: All users flow

You fetch users from an API that uses pagination. You want to expose a flow that emits all users. Implement a `getAllUsers` function that returns a flow of all users. Use a `fetchUsers` function to fetch users from the API. You can assume that the `fetchUsers` function is already implemented. It takes a page number as a parameter and returns a list of users from that page. If there are no more pages, it returns an empty list. You should fetch pages on demand.

Starting code:

```
class AllUsers(private val repository: UserRepository) {
    fun getAllUsers(): Flow<User> = TODO()
}

interface UserRepository {
    fun fetchUsers(pageNumber: Int): List<User>
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/AllUsers.kt`. You can clone this project and solve this exercise locally.

Exercise: distinct

Implement a `distinct` function that returns a flow of distinct elements. It should keep all the previously emitted elements in memory and emit only those that were not emitted before.

```
fun <T> Flow<T>.distinct(): Flow<T> = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/Distinct.kt`. You can clone this project and solve this exercise locally.

Exercise: TemperatureService

You are working on a `TemperatureService` class that should provide temperature updates for a given city. The service should use a `TemperatureDataSource` to get temperature updates. The service should:

- Provide a `Flow` of `Fahrenheit` temperatures for a given city (updates filtered by city, and converted from Celsius).
- Store the last known temperature for each city (at this point, you can store temparature for only those cities that are currently observed).
- Provide the last known temperature for a given city.
- Provide all last known temperatures.

```
class TemperatureService(
    private val temperatureDataSource: TemperatureDataSource,
    backgroundScope: CoroutineScope,
) {
    private val lastKnownTemperature =
        ConcurrentHashMap<String, Fahrenheit>()

    fun observeTemperature(city: String): Flow<Fahrenheit> =
        TODO()

    fun getLastKnown(city: String): Fahrenheit? =
        lastKnownTemperature[city]

    fun getAllLastKnown(): Map<String, Fahrenheit> =
        lastKnownTemperature.toMap()

    private fun celsiusToFahrenheit(celsius: Double) =
        Fahrenheit(celsius * 9 / 5 + 32)
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/TemperatureService.kt`. You can clone this project and solve this exercise locally.

Exercise: NewsViewModel

You are implementing a news application. You have a `NewsRepository` that exposes a `fetchNews` function that returns a `Flow` of news. You want to implement a `NewsViewModel` that should start fetching news when initialized. Each fetched news item should be emitted to `_newsToShow` using the `update` function (`_newsToShow.update { it + news }`). When fetching starts, `_progressVisible` should be set to `true` using `value` property; when fetching ends, `_progressVisible` should be set to `false`. When an error occurs, it should be sent to `_errors` using the `send` function. However, if the error is `ApiException`, this flow should always start again. Start this flow in the `init` block in the `viewModelScope` scope.

Starting code:

```
class NewsViewModel(
    newsRepository: NewsRepository,
) : BaseViewModel() {
    private val _progressVisible = MutableStateFlow(false)
    val progressVisible = _progressVisible.asStateFlow()

    private val _newsToShow = MutableStateFlow(emptyList<News>())
    val newsToShow = _newsToShow.asStateFlow()

    private val _errors = Channel<Throwable>(Channel.UNLIMITED)
    val errors = _errors.receiveAsFlow()

    init {
        // TODO
    }
}

class ApiException : Exception()

interface NewsRepository {
    fun fetchNews(): Flow<News>
}

abstract class BaseViewModel {
    protected val viewModelScope = CoroutineScope(
        Dispatchers.Main.immediate + SupervisorJob()
    )
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on

GitHub in the file `coroutines/flow/NewsViewModel.kt`. You can clone this project and solve this exercise locally.

Exercise: ProductService

Implement the observeProducts function in the ProductService class, which should return a flow of Product objects based on the given categories parameter. The function should meet the following requirements:

- Observe product updates from the `productRepository`.
- Do not emit the same product twice in a row.
- Map each product update to `Product` object using the `productRepository`. Products should be fetched concurrently.
- Filter the products based on the provided categories set, emitting only the products whose category is present in the set.
- Increment the `activeObservers` count when the flow starts and decrement it when the flow completes.

Starting code:

```
class ProductService(
    private val productRepository: ProductRepository,
    backgroundScope: CoroutineScope,
) {
    private val activeObservers = AtomicInteger(0)

    fun observeProducts(categories: Set<String>): Flow<Product> =\
 TODO()

    fun activeObserversCount(): Int = activeObservers.get()
}

interface ProductRepository {
    // Emits ids of the products that got updated
    fun observeProductUpdates(): Flow<String>
    suspend fun fetchProduct(id: String): Product
}

data class Product(
    val id: String,
    val category: String,
    val name: String,
    val price: Double,
)
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/ProductService.kt`. You can clone this project and solve this exercise locally.

Exercise: Flow Kata

To practice all the different flow operators, implement the following functions:

- `producingUnits` - produces a flow of `Unit` with `num` elements.
- `delayEach` - adds a delay of time `timeMillis` between elements.
- `mapIndexed` - should transform values, where the transformation value should have the index of the element.
- `toNextNumbers` - should transform elements to the next numbers starting from 1. So `Unit`, `Unit`, `Unit` should be transformed to 1, 2, 3.
- `withHistory` - produces not only elements but the whole history until now.
- `makeLightSwitch` - based on two light switches; it should decide if the light should be switched on. It should emit true after a change if one switch is true and the other is false. The first emitted state should be false, and then a new state should be emitted with each event from any switch.
- `makeLightSwitchToggle` - based on two light switches; it should decide if the light should be switched on. It should emit true if the number of events from both light switches is odd. Each event from a light switch emits a new state. The first emitted state is true; each new state is the previous state, but toggled.
- `polonaisePairing` - Should pair two flows of people so that each person from one flow is paired with the person from the other flow, which is emitted at the same time. A person without a pair should be ignored.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/Kata.kt`. You can clone this project and solve this exercise locally.

Exercise: MessageService

Implement a `MessageService` class that has three functions:

- `threadsSearch` - takes a flow of queries and returns a flow of threads. Each query should be used to observe threads from the repository. If a new query is emitted, the previous one should be cancelled.
- `subscribeThreads` - takes a flow of threads and returns a flow of thread updates. Each thread should be subscribed to in the repository. Should observe all threads concurrently.
- `sendMessages` - takes a flow of messages lists and returns a flow of responses that are the result of sending those messages. Each message should be synchronously sent to the repository, and its result should be the response.

```
class MessageService(
    private val messageRepository: MessageRepository
) {
    fun threadsSearch(
        query: Flow<String>
    ): Flow<MessageThread> = TODO()

    fun subscribeThreads(
        threads: Flow<MessageThread>
    ): Flow<MessageThreadUpdate> = TODO()

    fun sendMessages(
        messages: Flow<List<Message>>
    ): Flow<MessageSendingResponse> = TODO()
}

interface MessageRepository {
    fun searchThreads(
        query: String
    ): Flow<MessageThread>

    fun subscribeThread(
        threadId: String
    ): Flow<MessageThreadUpdate>

    fun sendMessages(
        messages: List<Message>
    ): Flow<MessageSendingResponse>
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/MessageService.kt`. You can clone this project and solve this exercise locally.

Exercise: Update ProductService

Get back to the `ProductService` exercise, and improve its implementation so that there is only one observer of product updates, and the same product is not fetched more than once per update. Observation should start when the first observer is added and stop when the last observer completes.

Exercise: Update TemperatureService

Get back to the `TemperatureService` exercise, and improve its implementation so that there is only one observer for the temperature updates, that observes since this class was created for as long as it exists.

Exercise: LocationService

Implement a `LocationService` class that eagerly observes location updates and exposes them as a flow. It should also expose the last-known location as a function (or `null` if there are no known locations). No matter how many observers there are, there should be only one location observer from `LocationRepository`. When new observers are added, they should receive the last known location. This class should not send the same location multiple times in a row. If observer is slower than the location updates, it should receive only the last known location (we are not interested in intermediate locations).

```
class LocationService(
    locationRepository: LocationRepository,
    backgroundScope: CoroutineScope,
) {
    fun observeLocation(): Flow<Location> = TODO()

    fun currentLocation(): Location? = TODO()
}
```

There are three ways how this problem can be solved:

- Using `stateIn`.
- Using `shareIn` and `replayCache`.
- Using `shareIn` and a property to store the last known location.

Try to implement all those solutions.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/LocationService.kt`. You can clone this project and solve this exercise locally.

Exercise: PriceService

In your application, you need to store the prices of a large number of products. You keep them in the `PriceService` class. You want to expose a flow of prices that first emits the current prices of all the products, and then only the prices that have changed. Should not send empty updates. This service should also expose the current prices as a function. No matter how many observers there are, there should be only one price observer from `PriceRepository`.

```
class PriceService(
    priceRepository: PriceRepository,
    backgroundScope: CoroutineScope,
) {
    fun observePrices(): Flow<Map<ProductId, PriceConfig>> =TODO()

    fun currentPrices(): Map<ProductId, PriceConfig> = TODO()
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/PriceService.kt`. You can clone this project and solve this exercise locally.

Exercise: NewsViewModel using stateIn

Exercise *NewsViewModel* can be solved more easily using the `stateIn` function. Change its implementation now. Compare both solutions. Which one do you prefer? What are adventages and disadvantages of each solution?

Exercise: Flow testing

Test the `observeAppointments` function from the `ObserveAppointmentsUseCase` class. Starting code:

```
class ObserveAppointmentsUseCase(
    private val appointmentRepository: AppointmentRepository
) {
    fun observeAppointments(): Flow<List<Appointment>> =
        appointmentRepository
        .observeAppointments()
        .filterIsInstance<AppointmentUpdate>()
        .map { it.appointments }
        .distinctUntilChanged()
        .retry { it is ApiException && it.code in 500..599 }
}

interface AppointmentRepository {
    fun observeAppointments(): Flow<AppointmentEvent>
}

sealed class AppointmentEvent
data class AppointmentUpdate(
    val appointments: List<Appointment>
) : AppointmentEvent()
data object AppointmentConfirmed : AppointmentEvent()
data class Appointment(val title: String, val time: Instant)
data class ApiException(val code: Int) : Throwable()
```

You should test the following cases:

- should receive only appointment lists from appointment updates
- should not receive non-distinct values
- should retry exceptions of type `ApiException` with code 5XX

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/flow/ObserveAppointmentsUseCase.kt`. You can clone this project and solve this exercise locally.

Advanced Kotlin exercises

This is a collection of exercises that I use in my workshop Advanced Kotlin, and that I later included in the book of the same name. Those are not easy exercises, some require a lot of time to solve. They will teach you not only some advanced Kotlin features, but they will also teach you, how popular libraries work under the hood.

Exercise: Usage of generic types

The code below will not compile due to a type mismatch. Which lines will show compilation errors?

```
fun takeIntList(list: List<Int>) {}
takeIntList(listOf<Any>())
takeIntList(listOf<Nothing>())

fun takeIntMutableList(list: MutableList<Int>) {}
takeIntMutableList(mutableListOf<Any>())
takeIntMutableList(mutableListOf<Nothing>())

fun takeAnyList(list: List<Any>) {}
takeAnyList(listOf<Int>())
takeAnyList(listOf<Nothing>())

class BoxOut<out T>
fun takeBoxOutInt(box: BoxOut<Int>) {}
takeBoxOutInt(BoxOut<Int>())
takeBoxOutInt(BoxOut<Number>())
takeBoxOutInt(BoxOut<Nothing>())

fun takeBoxOutNumber(box: BoxOut<Number>) {}
takeBoxOutNumber(BoxOut<Int>())
takeBoxOutNumber(BoxOut<Number>())
takeBoxOutNumber(BoxOut<Nothing>())

fun takeBoxOutNothing(box: BoxOut<Nothing>) {}
takeBoxOutNothing(BoxOut<Int>())
takeBoxOutNothing(BoxOut<Number>())
takeBoxOutNothing(BoxOut<Nothing>())

fun takeBoxOutStar(box: BoxOut<*>) {}
takeBoxOutStar(BoxOut<Int>())
takeBoxOutStar(BoxOut<Number>())
takeBoxOutStar(BoxOut<Nothing>())

class BoxIn<in T>
fun takeBoxInInt(box: BoxIn<Int>) {}
takeBoxInInt(BoxIn<Int>())
takeBoxInInt(BoxIn<Number>())
takeBoxInInt(BoxIn<Nothing>())
takeBoxInInt(BoxIn<Any>())
```

Exercise: Generic Response

You needed to model a response from a server that can be represented as a success or a failure, both of which contain data of a generic type. This is how you modeled it:

```
sealed class Response<R, E>
class Success<R, E>(val value: R) : Response<R, E>()
class Failure<R, E>(val error: E) : Response<R, E>()
```

However, you found that this implementation is problematic. To create a `Success` object, you need to provide two generic types, but you only need one. To create a `Failure` object, you need to provide two generic types, but you only need one. Your task is to fix this problem.

```
val rs1 = Success(1) // Compilation error
val rs2 = Success("ABC") // Compilation error
val re1 = Failure(Error()) // Compilation error
val re2 = Failure("Error") // Compilation error
```

You need to define `Success` and `Failure` in a way that each can be created with only one generic type argument. You want to be able to use `Success` and `Failure` without specifying generic types, so just use `Success(1)` or `Failure("Error")`.

You also want to allow `Success<Int>` to be upcast to `Success<Number>` or to `Success<Any>`, and `Failure<Error>` to be upcast to `Failure<Throwable>` or to `Failure<Any>`. You want to be able to use `Success<Int>` as `Response<Int, Throwable>`.

```
val rs1 = Success(1)
val rs2 = Success("ABC")
val re1 = Failure(Error())
val re2 = Failure("Error")

val rs3: Success<Number> = rs1
val rs4: Success<Any> = rs1
val re3: Failure<Throwable> = re1
val re4: Failure<Any> = re1

val r1: Response<Int, Throwable> = rs1
val r2: Response<Int, Throwable> = re1
```

Starting code and example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/generics/Response.kt`. You can clone this project and solve this exercise locally.

Exercise: Generic Consumer

In your project, you use a class that represents a consumer of some type. You have two implementations of this class: `Printer` and `Sender`. A printer that can accept `Number` should also accept `Int` and `Double`. A sender that can accept `Int` should also accept `Number` and `Any`. In general, a consumer that can accept `T` should also accept `S` if it is a subtype of `T`. Update the `Consumer`, `Printer` and `Sender` classes to achieve this.

```
abstract class Consumer<T> {
    abstract fun consume(elem: T)
}

class Printer<T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}

class Sender<T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}
```

Example usage:

```
val p1 = Printer<Number>()
val p2: Printer<Int> = p1
val p3: Printer<Double> = p1

val s1 = Sender<Any>()
val s2: Sender<Int> = s1
val s3: Sender<String> = s1

val c1: Consumer<Number> = p1
val c2: Consumer<Int> = p1
val c3: Consumer<Double> = p1
```

Starting code and example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/generics/Consumer.kt`. You can clone this project and solve this exercise locally.

Exercise: ApplicationScope

You need to create an `ApplicationScope` class that implements the `CoroutineScope`, `ApplicationControlScope` and `LoggingScope` interfaces. It should expect primary constructor properties of the same types and use them as delegates. Use interface delegation to implement this.

```
interface ApplicationControlScope {
    val application: Application
    fun start()
    fun stop()
    fun isRunning(): Boolean
}

data class Application(val name: String)

interface LoggingScope {
    fun logInfo(message: String)
    fun logWarning(message: String)
    fun logError(message: String)
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/delegates/ApplicationScope.kt`. You can clone this project and solve this exercise locally.

Exercise: Lateinit delegate

Implement a `Lateinit` delegate that makes a property behave like a `lateinit` property so that the delegate can keep set values, but it should not require an initial value. If the getter is called before the property is set, it should throw an `IllegalStateException` exception with the message "Uninitialized lateinit property {name}".

```
val a by Lateinit<Int>()
a = 1
println(a) // 1

val b by Lateinit<String>()
b = "ABC"
println(b) // ABC

val c by Lateinit<String>()
println(c) // IllegalStateException:
// Uninitialized lateinit property c
```

This delegate should support nullable types.

```
val a by Lateinit<Int?>()
a = 1
println(a) // 1
a = null
println(a) // null
```

Unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/delegates/Lateinit.kt`. You can clone this project and solve this exercise locally.

Hint: You can make your class implement the `ReadWriteProperty<Any?, T>` interface to make it a property delegate.

Exercise: Blog Post Properties

In your project, you're working with a `BlogPost` class that represents a blog post, including its title, content, and author details. Your task is to enhance the `BlogPost` class by adding and implementing the following properties:

- `authorName` - a string that combines the author's name and surname. For example, if the author's name is "John" and the surname is "Smith", the value of this property should be "John Smith". You can assume that you will need to use this property many times per blog post object.
- `wordCount` - an integer that represents the number of words in the blog post. You can assume that this property is expensive to calculate and you might need it more than once per blog post object
- `isLongRead` - a boolean that indicates whether the blog post is longer than 1000 characters. You can assume that you will need to use this property at most once per blog post object.
- `summary` - a string that is calculated using the `generateSummary` function. This object is very heavy to calculate, so you don't want to calculate it more than once.

You need to decide how to implement each of these properties. Your options are:

- A `val` property defined by a value
- A `val` property defined by a getter
- A lazy property

Starting code:

```
data class BlogPost(
    val title: String,
    val content: String,
    val author: Author,
) {
    // TODO: Add properties here

    private fun generateSummary(content: String): String =
        content.take(100) + "..."
}

data class Author(
    val key: String,
    val name: String,
    val surname: String,
)
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/delegates/Lazy.kt`. You can clone this project and solve this exercise locally.

Exercise: Mutable lazy delegate

The `lazy` delegate can only be used to `val` variables. Implement a `MutableLazy` delegate that can be used to `var` variables. It should behave like a `lazy` delegate but supporting read-write properties. If the property getter is called before the setter, the `MutableLazy` delegate should initialize the property using its lambda expression. If the property getter is called after the setter, it should return the value that was set.

```
fun calculate(): Int {
    print("Calculating... ")
    return 42
}

var a by mutableLazy { calculate() }
println(a) // Calculating... 42
println(a) // 42
a = 1
println(a) // 1

var b by mutableLazy { calculate() }
b = 2
println(b) // 2


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

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/delegates/MutableLazy.kt`. You can clone this project and solve this exercise locally.

Exercise: Coroutine time measurement

In order to measure a block execution time, you defined the following function, that can measure real time on production, and virtual time in unit tests:

```
suspend fun measureCoroutine(
    body: suspend () -> Unit
): Duration {
    val dispatcher = coroutineContext[ContinuationInterceptor]
    return if (dispatcher is TestDispatcher) {
        val before = dispatcher.scheduler.currentTime
        body()
        val after = dispatcher.scheduler.currentTime
        after - before
    } else {
        measureTimeMillis {
            body()
        }
    }.milliseconds
}
```

However, you found that it is not very convenient to use because it lacks a contract. Define a contract to make the following code compile:

```
runTest {
    val result: String
    val duration = measureCoroutine {
        delay(1000)
        result = "OK"
    }
    println(duration) // 1000 ms
    println(result) // OK
}

runBlocking {
    val result: String
    val duration = measureCoroutine {
        delay(1000)
        result = "OK"
    }
    println(duration) // 1000 ms
    println(result) // OK
}
```

Starting code and example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/contract/MesureCoroutineTime.kt`. You can clone this project and solve this exercise locally.

Exercise: Adjust Kotlin for Java usage

Consider the following Kotlin elements:

```
package advanced.java

data class Money(
    val amount: BigDecimal = BigDecimal.ZERO,
    val currency: Currency = Currency.EUR,
) {
    companion object {
        fun eur(amount: String) =
            Money(BigDecimal(amount), Currency.EUR)

        fun usd(amount: String) =
            Money(BigDecimal(amount), Currency.USD)

        val ZERO_EUR = eur("0.00")
    }
}

fun List<Money>.sum(): Money? {
    if (isEmpty()) return null
    val currency = this.map { it.currency }.toSet().single()
    return Money(
        amount = sumOf { it.amount },
        currency = currency
    )
}

operator fun Money.plus(other: Money): Money {
    require(currency == other.currency)
    return Money(amount + other.amount, currency)
}

enum class Currency {
    EUR, USD
}
```

This is how they can be used in Kotlin:

```
fun main() {
    val money1 = Money.eur("10.00")
    val money2 = Money.eur("29.99")

    println(listOf(money1, money2, money1).sum())
    // Money(amount=49.99, currency=EUR)

    println(money1 + money2)
    // Money(amount=39.99, currency=EUR)

    val money3 = Money.usd("10.00")
    val money4 = Money()
    val money5 = Money(BigDecimal.ONE)
    val money6 = Money.ZERO_EUR
}
```

However, Java usage is not so convenient. Your task is to add appropriate annotations so it is more Java-friendly and can be used like this:

```
package advanced.java;

import java.math.BigDecimal;
import java.util.List;

public class JavaClass {

    public static void main(String[] args) {
        Money money1 = Money.eur("10.00");
        Money money2 = Money.eur("29.99");

        List<Money> moneyList =
                List.of(money1, money2, money1);

        System.out.println(MoneyUtils.plus(money1, money2));
        // Money(amount=39.99, currency=EUR)

        Money money3 = Money.usd("10.00");
        Money money4 = new Money();
        Money money5 = new Money(BigDecimal.ONE);
        Money money6 = Money.ZERO_EUR;
    }
}
```

Exercise: Multiplatform LocalDateTime

In your multiplatform project, you need to define a common type to represent time. You decided that you want it to behave just like `LocalDateTime` from the `java.time` library. In fact, you decided that on Kotlin/JVM you want to use `LocalDateTime` directly as an actual type. Define a common type `LocalDateTime`; then, define an actual typealias for Kotlin/JVM and an actual class for Kotlin/JS that wraps over `Date` from JavaScript.

These are the expected elements that you need to provide for each platform:

```
expect class LocalDateTime {
    fun getSecond(): Int
    fun getMinute(): Int
    fun getHour(): Int
    fun plusSeconds(seconds: Long): LocalDateTime
}

expect fun now(): LocalDateTime

expect fun parseLocalDateTime(str: String): LocalDateTime
```

Starting code and unit tests for this exercise can be found in the project:

https://github.com/MarcinMoskala/kmp-exercise

Exercise: Migrating a Kotlin/JVM project to KMP

Clone the following project:

https://github.com/MarcinMoskala/sudoku-generator-exercise

This is a Kotlin/JVM project that implements a logic that can generate and solve sudoku puzzles. However, you need to use these capabilities in a React project written in TypeScript. Transform your project to a Kotlin Multiplatform project and generate a JavaScript library named "sudoku-generator" from it. Then, use it in your React project.

In the `web-client` folder, you can find a React project. It is already set to use the generated library, assuming it is going to be named "sudoku-generator" and located in the default path "build/productionLibrary". You can run it using the `npm start` command. It is a simple project that displays an unsolved and a solved version of a sudoku. To make it work, you need to transform the Kotlin parts to Kotlin multiplatform and define a Kotlin/JS wrapper over the sudoku generator that is exported to JavaScript and exposes objects that can be used in TypeScript.

The exported class should be named `SudokuGenerator`; it should have no package, an empty constructor, and a single `generateSudoku` method that takes no arguments and returns an object `Sudoku` with a random sudoku and its solution. The `Sudoku` object should have two properties: `sudoku` and `solved`. Both should be of type `Array<Array<Int?>>`. The sudoku should be generated using `SudokuGenerator` and `SudokuSolver`. Both these classes can be created using the `SudokuGenerator()` and `SudokuSolver()` constructors. The sudoku should be generated using the `generate` method from the `SudokuGenerator` class, with the solver from the `SudokuSolver` class used as an argument.

To transform `SudokuState` to `Array<Array<Int?>>`, you can use the following function:

```
fun SudokuState.toJs(): Array<Array<Int?>> = List(9) { row ->
    List(9) { col ->
        val cell = this.cells[SudokuState.Position(row, col)]
        when (cell) {
            is SudokuState.CellState.Filled -> cell.value
            is SudokuState.CellState.Empty, null -> null
        }
    }.toTypedArray()
}.toTypedArray()
```

Hints:

- IntelliJ often has trouble with recognizing a change from Kotlin/JVM to Kotlin Multiplatform. If you encounter this problem, try to invalidate the cache and restart IntelliJ. Also, give it a moment.
- To set the name of the generated module, you need to set the `moduleName` property in the `js(IR)` block.

- If you have problems with generating TypeScript definitions, try to call `generateTypeScriptDefinitions` in the `js(IR)` block.
- Generate the library using the Gradle

`jsBrowserProductionLibraryDistribution` task, then remember to use `npm install` in `web-client`.

Exercise: Function caller

Your task is to implement methods of a class that is used to call function references with constant values specified by type. This class should have the following methods:

- `setConstant` - sets a constant value for a given type.
- `call` - calls a function reference with constant values specified by type.

If a constant value for a given type is not specified, an exception should be thrown. Unless the parameter with this type is optional, then its default argument should be used.

```
class FunctionCaller {
    inline fun <reified T> setConstant(value: T) {
        setConstant(typeOf<T>(), value)
    }

    fun setConstant(type: KType, value: Any?) {
        TODO()
    }

    fun <T> call(function: KFunction<T>): T {
        TODO()
    }
}
```

Example usage:

```
fun printStrIntNum(str: String, int: Int, num: Number) {
    println("str: $str, int: $int, num: $num")
}

fun printWithOptionals(l: Long = 999, s: String) {
    println("l: $l, s: $s")
}

fun main() {
    val caller = FunctionCaller()
    caller.setConstant("ABC")
    caller.setConstant(123)
    caller.setConstant(typeOf<Number>(), 3.14)
    caller.call(::printStrIntNum)
    // str: ABC, int: 123, num: 3.14
    caller.call(::printWithOptionals)
    // l: 999, s: ABC
}
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/reflection/FunctionCaller.kt`. You can clone this project and solve this exercise locally.

Hint: To support optional parameters, you should use `callBy` instead of `call`.

Exercise: Object serialization to JSON

Your task is to implement a function to serialize a Kotlin object to JSON. The resulting text should include all class member properties. You should support primitive types, iterables (present as an array), and map type (present as an object). You should also support nested objects. Do not use any external libraries. Kotlin's reflection is all you need.

You should support the following annotations:

- `@SerializationName` - can be applied to a property to change its name in the resulting JSON.
- `@SerializationIgnore` - can be applied to a property to ignore it in the resulting JSON.
- `@SerializationNameMapper` - can be applied to a class or property to specify a custom name mapper. The mapper should implement the `NameMapper` interface. This mapper can be an object declaration or a class with a no-arg constructor.
- `@SerializationIgnoreNulls` - can be applied to a class to ignore all null properties in the resulting JSON.

```
@Target(AnnotationTarget.PROPERTY)
annotation class SerializationName(val name: String)

@Target(AnnotationTarget.PROPERTY)
annotation class SerializationIgnore

@Target(AnnotationTarget.PROPERTY, AnnotationTarget.CLASS)
annotation class SerializationNameMapper(
    val mapper: KClass<out NameMapper>
)

@Target(AnnotationTarget.CLASS)
annotation class SerializationIgnoreNulls

interface NameMapper {
    fun map(name: String): String
}


fun serializeToJson(value: Any): String = TODO()
```

Example usage:

```
@SerializationNameMapper(SnakeCaseName::class)
@SerializationIgnoreNulls
class Creature(
    val name: String,
    @SerializationName("att")
    val attack: Int,
    @SerializationName("def")
    val defence: Int,
    val traits: List<Trait>,
    val elementCost: Map<Element, Int>,
    @SerializationNameMapper(LowerCaseName::class)
    val isSpecial: Boolean,
    @SerializationIgnore
    var used: Boolean = false,
    val extraDetails: String? = null,
)

object LowerCaseName : NameMapper {
    override fun map(name: String): String = name.lowercase()
}

class SnakeCaseName : NameMapper {
    val pattern = "(?<=.)[A-Z]".toRegex()

    override fun map(name: String): String =
        name.replace(pattern, "_$0").lowercase()
}

enum class Element {
    FOREST, ANY,
}

enum class Trait {
    FLYING
}

fun main() {
    val creature = Creature(
        name = "Cockatrice",
        attack = 2,
        defence = 4,
        traits = listOf(Trait.FLYING),
        elementCost = mapOf(
            Element.ANY to 3,
```

```
            Element.FOREST to 2
        ),
        isSpecial = true,
    )
    println(serializeToJson(creature))
    // {"att": 2, "def": 4,
    // "element_cost": {"ANY": 3, "FOREST": 2},
    // "isspecial": true, "name": "Cockatrice",
    // "traits": ["FLYING"]}
}
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/reflection/JsonSerializer.kt`. You can clone this project and solve this exercise locally.

Exercise: Object serialization to XML

Your task is to implement a function that serializes a Kotlin object to XML. The resulting text should include all class member properties. You should also support nested objects. Do not use any external libraries. Kotlin's reflection is all you need.

You should support the following annotations:

- `@SerializationName` - can be applied to a property to change its name in the resulting XML.
- `@SerializationIgnore` - can be applied to a property to ignore it in the resulting XML.
- `@SerializationNameMapper` - can be applied to a class or property to specify a custom name mapper. The mapper should implement the `NameMapper` interface. This mapper can be an object declaration or a class with a no-arg constructor.
- `@SerializationIgnoreNulls` - can be applied to a class to ignore all null properties in the resulting XML.

```
@Target(AnnotationTarget.PROPERTY)
annotation class SerializationName(val name: String)

@Target(AnnotationTarget.PROPERTY)
annotation class SerializationIgnore

@Target(AnnotationTarget.PROPERTY, AnnotationTarget.CLASS)
annotation class SerializationNameMapper(
    val mapper: KClass<out NameMapper>
)

@Target(AnnotationTarget.CLASS)
annotation class SerializationIgnoreNulls

interface NameMapper {
    fun map(name: String): String
}


fun serializeToXml(value: Any): String = TODO()
```

Usage example (the resulting XML should not include indentation – it was added here for readability):

```
fun main() {
    data class SampleDataClass(
        val externalTxnId: String,
        val merchantTxnId: String,
        val reference: String
    )

    val data = SampleDataClass(
        externalTxnId = "07026984141550752666",
        merchantTxnId = "07026984141550752666",
        reference = "MERCHPAY"
    )

    println(serializeToXml(data))
    // <SampleDataClass>
    //     <externalTxnId>07026984141550752666<externalTxnId>
    //     <merchantTxnId>07026984141550752666<merchantTxnId>
    //     <reference>MERCHPAY<reference>
    // </SampleDataClass>

    @SerializationNameMapper(UpperSnakeCaseName::class)
    @SerializationIgnoreNulls
    class Book(
        val title: String,
        val author: String,
        @SerializationName("YEAR")
        val publicationYear: Int,
        val isbn: String?,
        @SerializationIgnore
        val price: Double,
    )

    @SerializationNameMapper(UpperSnakeCaseName::class)
    class Library(
        val catalog: List<Book>
    )

    val library = Library(
        catalog = listOf(
            Book(
                title = "The Hobbit",
                author = "J. R. R. Tolkien",
                publicationYear = 1937,
                isbn = "978-0-261-10235-4",
```

```
                price = 9.99,
            ),
            Book(
                title = "The Witcher",
                author = "Andrzej Sapkowski",
                publicationYear = 1993,
                isbn = "978-0-575-09404-2",
                price = 7.99,
            ),
            Book(
                title = "Antifragile",
                author = "Nassim Nicholas Taleb",
                publicationYear = 2012,
                isbn = null,
                price = 12.99,
            )
        )
    )

    println(serializeToXml(library))
    // <LIBRARY>
    //     <CATALOG>
    //         <BOOK>
    //             <AUTHOR>J. R. R. Tolkien<AUTHOR>
    //             <ISBN>978-0-261-10235-4<ISBN>
    //             <YEAR>1937<YEAR>
    //             <TITLE>The Hobbit<TITLE>
    //         </BOOK>
    //         <BOOK>
    //             <AUTHOR>Andrzej Sapkowski<AUTHOR>
    //             <ISBN>978-0-575-09404-2<ISBN>
    //             <YEAR>1993<YEAR>
    //             <TITLE>The Witcher<TITLE>
    //         </BOOK>
    //        <BOOK>
    //            <AUTHOR>Nassim Nicholas Taleb<AUTHOR>
    //            <YEAR>2012<YEAR>
    //            <TITLE>Antifragile<TITLE>
    //        </BOOK>
    //     <CATALOG>
    // </LIBRARY>
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on

GitHub in the file `advanced/reflection/XmlSerializer.kt`. You can clone this project and solve this exercise locally.

Exercise: DSL-based dependency injection library

Your task is to implement a simple dependency injection library. It should be based on the `Registry` class, which should be used to register dependencies. It should have the following methods:

- `register` - registers a normal dependency that is created every time it is needed. It should take a type and a lambda expression that returns an instance of that type. In the scope of this lambda expression, you should be able to use `Registry` to get other dependencies. This function should have both an inline version with reified type, and a non-inline version with the `KClass` parameter.
- `singleton` - registers a singleton dependency that is created only once and then reused. It should take a type and a lambda expression that returns an instance of that type. In the scope of this lambda expression, you should be able to use `Registry` to get other dependencies. This function should have both an inline version with a reified type, and a non-inline version with the `KClass` parameter.
- `get` - returns an instance of a given type. If the type is registered as a singleton, it should return the same instance every time. If the type is registered as a normal dependency, it should return a new instance every time it is called. This function should have both an inline version with reified type, and a non-inline version with the `KClass` parameter.
- `exists` - returns true if a given type is registered, otherwise it returns false. This function should have both an inline version with a reified type, and a non-inline version with the `KClass` parameter.

You should also implement a `registry` function to create a `Registry` instance in DSL style. It should take a lambda expression with `Registry` as a receiver, and it should return a `Registry` instance. In the scope of this lambda expression, you should be able to use `Registry` to register dependencies.

Example usage:

```
data class UserConfiguration(val url: String)

interface UserRepository {
    fun get(): String
}
class RealUserRepository(
    private val userConfiguration: UserConfiguration,
) : UserRepository {
    override fun get(): String =
        "User from ${userConfiguration.url}"
}
```

```
class UserService(
    private val userRepository: UserRepository,
    private val userConfiguration: UserConfiguration,
) {
    fun get(): String = "Got ${userRepository.get()}"
}

fun main() {
    val registry: Registry = registry {
        singleton<UserConfiguration> {
            UserConfiguration("http://localhost:8080")
        }
        normal<UserService> {
            UserService(
                userRepository = get(),
                userConfiguration = get(),
            )
        }
        singleton<UserRepository> {
            RealUserRepository(
                userConfiguration = get(),
            )
        }
    }

    val userService: UserService = registry.get()
    println(userService.get())
    // Got User from http://localhost:8080

    val ur1 = registry.get<UserRepository>()
    val ur2 = registry.get<UserRepository>()
    println(ur1 === ur2) // true

    val uc1 = registry.get<UserService>()
    val uc2 = registry.get<UserService>()
    println(uc1 === uc2) // false
}
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/reflection/DependencyInjection.kt`. You can clone this project and solve this exercise locally.

Exercise: Mocking library

Your task is to implement a simple library for interface mocking. You should only support mocking interface methods that do not throw exceptions. This is how you should use it:

```
data class User(val name: String)

interface UserRepository {
    fun getUser(userId: String): User?
    fun allUsers(): List<User>
}

interface UserService {
    fun getUser(): User
}

fun main() {
    val registry = MockRegistry()
    val userRepository = registry.mock<UserRepository>()
    val userService = registry.mock<UserService>()

    registry.setReturnValue(
        { userRepository.getUser("alex") },
        User("Alex Smith")
    )
    registry.setReturnValue(
        { userRepository.getUser("bell") },
        User("Bell Rogers")
    )
    registry.setReturnValue(
        { userRepository.getUser("dan") },
        null
    )
    registry.setBody({ userRepository.allUsers() }) {
        listOf(User("James Bond"), User("Jane Doe"))
    }
    registry.setBody({ userService.getUser() }) {
        User(userRepository.getUser("dan")?.name ?: "Unknown")
    }

    println(userRepository.getUser("alex"))
    // User(name=Alex Smith)
    println(userRepository.allUsers())
    // [User(name=James Bond), User(name=Jane Doe)]
```

```
    println(userRepository.getUser("bell"))
    // User(name=Bell Rogers)
    println(userService.getUser())
    // User(name=Unknown)
    registry.setReturnValue(
        { userRepository.getUser("dan") },
        User("Dan Brown")
    )
    println(userService.getUser())
    // User(name=Dan Brown)
}
```

Your implementation should use `Proxy` from the `java.lang.reflect` package.

You can find unit tests and usage examples in the MarcinMoskala/kotlin-exercises project on GitHub in the file `advanced/java/Mocking.kt`. You can clone this project and solve this exercise locally.

Hints:

- A mocked method can be called to both record and use the mock. To know what situation it is, define a `recording` flag in your class.
- Each call is identified by three things: mock object identifier, method name, and method arguments. You can define a data class to keep those three things.

Exercise: Annotation Processing execution measurement wrapper

Your task is to implement a custom Annotation Processor, that will generate a wrapper for measuring the execution time of public methods annotated with `Measured` annotation.

This processor should generate a wrapper class for each class that contains at least one method annotated with `Measured` annotation.
This wrapper class should be a Java class under the same package, and be named just like the wrapped class but with `Measured` prefix.
It should have two constructors: one that takes an instance of the original class as a parameter, and one that creates an instance of the original class based on the default constructor.
It should have the same methods as the original class, but each method with `Measured` annotation should measure the time of its execution and print it in the format "{method name} from {wrapped class name} took {execution time} ms".

For example, for the following class:

```
package academy.kt

class TokenService {

    @Measured
    fun getToken(): String {
        Thread.sleep(1000)
        return "ABCD"
    }
}
```

The following wrapper class should be generated:

```
package academy.kt;

import java.lang.String;
import org.jetbrains.annotations.NotNull;

class MeasuredTokenService {
  private TokenService wrapper;

  MeasuredTokenService(TokenService wrapper) {
    this.wrapper = wrapper;
  }
```

```
  MeasuredTokenService() {
    this.wrapper = new TokenService();
  }

  @NotNull
  public final String getToken() {
    long before = System.currentTimeMillis();
    java.lang.String value = wrapper.getToken();
    long after = System.currentTimeMillis();
    System.out.println("getToken from TokenService took " + (afte\
r - before) + " ms");
    return value;
  }
}
```

For the following class:

```
package academy.kt

class UserService(
    private val tokenService: TokenService
) {

    @Measured
    fun findUser(id: Int): User {
        tokenService.getToken()
        Thread.sleep(1000)
        return User("$id")
    }

    @Measured
    fun findUsers(): User {
        Thread.sleep(1000)
        return User("")
    }
}
```

The following wrapper class should be generated:

```
package academy.kt;

import org.jetbrains.annotations.NotNull;

class MeasuredUserService {
  private UserService wrapper;

  MeasuredUserService(UserService wrapper) {
    this.wrapper = wrapper;
  }

  MeasuredUserService(@NotNull TokenService tokenService) {
    this.wrapper = new UserService(tokenService);
  }

  @NotNull
  public final User findUser(int id) {
    long before = System.currentTimeMillis();
    academy.kt.User value = wrapper.findUser(id);
    long after = System.currentTimeMillis();
    System.out.println("findUser from UserService took " + (after\
 - before) + " ms");
    return value;
  }

  @NotNull
  public final User findUsers() {
    long before = System.currentTimeMillis();
    academy.kt.User value = wrapper.findUsers();
    long after = System.currentTimeMillis();
    System.out.println("findUsers from UserService took " + (afte\
r - before) + " ms");
    return value;
  }
}
```

Example usage of generated classes:

```
fun main() {
    val tokenService = TokenService()
    val userService = UserService(tokenService)
    val measuredService = MeasuredUserService(tokenService)
    val user = measuredService.findUser(12)
    // findUser from UserService took 200Xms
    val user2 = measuredService.findUsers()
    // findUser from UserService took 100Xms

    val measuredTokenService = MeasuredTokenService(tokenService)
    val token = measuredTokenService.getToken()
    // getToken from TokenService took 100Xms
}
```

For generating Kotlin code use Java Poet. Assume that wrapped class has no type parameters.

Here is a project, that provides configuration and sample use, and only requires processor implementation:

https://github.com/MarcinMoskala/measured-wrapper-ap

Exercise: KSP execution measurement wrapper

Your task is to implement a custom Kotlin Symbol Processor, that will generate a wrapper for measuring the execution time of public methods annotated with `Measured` annotation.

This processor should generate a wrapper class for each class that contains at least one method annotated with `Measured` annotation.
This wrapper class should be under the same package, and be named just like the wrapped class but with `Measured` prefix.
It should have two constructors: one that takes an instance of the original class as a parameter, and one that creates an instance of the original class based on the default constructor.
It should have the same methods as the original class, but each method with `Measured` annotation should measure the time of its execution and print it in the format "{method name} from {wrapped class name} took {execution time} ms".

For example, for the following class:

```
package academy.kt

class TokenService {

    @Measured
    fun getToken(): String {
        Thread.sleep(1000)
        return "ABCD"
    }
}
```

The following wrapper class should be generated:

```
package academy.kt

class MeasuredTokenService(
    val wrapper: TokenService,
) {
    constructor() : this(TokenService())

    fun getToken(): String {
        val before = System.currentTimeMillis()
        val value = wrapper.getToken()
        val after = System.currentTimeMillis()
        println("getToken from TokenService took ${after - before\
} ms")
```

```
        return value
    }
}
```

For the following class:

```
package academy.kt

class UserService(
    private val tokenService: TokenService
) {

    @Measured
    fun findUser(id: Int): User {
        tokenService.getToken()
        Thread.sleep(1000)
        return User("$id")
    }

    @Measured
    fun findUsers(): User {
        Thread.sleep(1000)
        return User("")
    }
}
```

The following wrapper class should be generated:

```
package academy.kt

class MeasuredUserService(
    val wrapper: UserService,
) {
    constructor(tokenService: TokenService) : this(
        UserService(tokenService)
    )

    fun findUser(id: Int): User {
        val before = System.currentTimeMillis()
        val value = wrapper.findUser(id)
        val after = System.currentTimeMillis()
        println("findUser from UserService took ${after - before}\
 ms")
```

```
        return value
    }

    fun findUsers(): User {
        val before = System.currentTimeMillis()
        val value = wrapper.findUsers()
        val after = System.currentTimeMillis()
        println("findUsers from UserService took ${after - before\
} ms")
        return value
    }
}
```

Example usage of generated classes:

```
fun main() {
    val tokenService = TokenService()
    val userService = UserService(tokenService)
    val measuredService = MeasuredUserService(tokenService)
    val user = measuredService.findUser(12)
    // findUser from UserService took 200Xms
    val user2 = measuredService.findUsers()
    // findUser from UserService took 100Xms

    val measuredTokenService = MeasuredTokenService(tokenService)
    val token = measuredTokenService.getToken()
    // getToken from TokenService took 100Xms
}
```

For generating Kotlin code use Kotlin Poet. Assume that wrapped class has no type parameters.

Here is a project, that provides configuration, tests and sample use, and only requires processor implementation:

https://github.com/MarcinMoskala/measured-wrapper-ksp

Here is a clean template for a KSP project, that you can use as a more demanding starting point:

https://github.com/MarcinMoskala/ksp-template

For inspiration of how a KSP project can be implemented, see the following projects:

- https://github.com/MarcinMoskala/generateinterface-ksp - generates interfaces for classes
- https://github.com/MarcinMoskala/DependencyInjection-KSP - generates class for simple dependency injection

Effective Kotlin exercises

This is a collection of exercises that I use in my workshop Effective Kotlin. I decided to not include them in the book, to keep the original formula of the book. Nevertheless, they might help you to learn and practice some essential knowledge, especially about safety and performance in Kotlin.

Exercise: A mutability problem

What operation on `name` will make the following code print `false`?

```
fun main() {
    val set = mutableSetOf<Name>()
    val name = Name("AAA")
    set.add(name)
    // ???
    println(set.contains(name)) // should print false
    println(set.first() == name) // should print true
}

data class Name(var name: String)
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/NameAndSet.kt`. You can clone this project and solve this exercise locally.

Exercise: Flow with history

As an exercise in my Kotlin Coroutines workshop, I ask participants to implement a `withHistory` function that returns a flow of all the values that have been emitted. As a result, I often see the following implementation:

```
fun <T> Flow<T>.withHistory(): Flow<List<T>> = flow {
    val history = mutableListOf<T>()
    emit(history)
    collect {
        history += it
        emit(history)
    }
}
```

This implementation is not correct, as is illustrated by the usage example below. Your task is to fix it.

```
suspend fun main() {
    flowOf(1, 2, 3)
        .withHistory()
        .toList()
        .let(::println)
    // [[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/FlowHistory.kt`. You can clone this project and solve this exercise locally.

Exercise: Correct InMemoryUserRepository

Fix `InMemoryUserRepository`. Before you start, can you identify all the problems? Analyze the code and note what might go wrong.

```
class InMemoryUserRepository {
    private val users = mutableSetOf<User>()

    fun addUser(user: User) {
        users.add(user)
    }

    fun getUsers() = users

    fun hasUser(user: User): Boolean = user in users

    fun changeSurname(userId: Int, newSurname: String) {
        users.find { it.id == userId }?.surname = newSurname
    }

    fun changeAllSurnames(newSurname: String) {
        users.forEach { it.surname = newSurname }
    }

    data class User(
        val id: Int,
        val name: String,
        var surname: String
    )
}
```

Here are concrete problems that you should fix:

- the code should not expose a mutation point.
- should allow concurrent user addition.
- should allow concurrent surname changes.
- a surname change should not cause problems with finding this user.
- should allow parallel write and read.
- when we set new surnames, they should all be the same.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/InMemoryUserRepository.kt`. You can clone this project and solve this exercise locally.

Exercise: Chunking a flow

Your task is to implement `chunked` extension function for `Flow<T>` that splits the flow into chunks of `T` items and emits them every `duration` time interval. It should emit the last chunk immediately when the flow is completed.

```
fun <T> Flow<T>.chunked(
    duration: Duration
): Flow<List<T>> = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/Chunked.kt`. You can clone this project and solve this exercise locally.

Things to consider about your solution:

- Is it thread-safe? Does it use a coroutine-friendly mechanism to assure thread-safety?
- Which collection should you use under the hood? What are the pros and cons of different collections?
- Do you stop collecting the source flow when this flow is cancelled or completed? Do you complete this flow when the source flow is completed? Do you pass exceptions from the source flow?

Exercise: Specify expectations

Implement a `notifyUser` function from safe/Requirements.kt that notifies the user using notifier unless:

- The `user` property is `null`, in which case do nothing.
- The `id` property is incorrect according to the `checkId` function from Notifier, in which case throw an `IncorrectId` exception.
- `user.name` or `user.surname` is `null`, in which case you should throw `IllegalArgumentException`.
- The notifier is not initialized, which is an incorrect state, in which case you should throw `IllegalStateException`.
- `notifyPerson` does not return `true` when you send a notification, in which case throw `AssertionError`.

Starting code:

```
fun Notifier.notifyUser(user: User?) {
    TODO()
}

data class User(
    val id: Int,
    val name: String?,
    var surname: String?
)

class IncorrectId : Error()

interface Notifier {

    /**
     * Indicate instance readiness to notify users
     */
    val initialized: Boolean

    /**
     * Notifies person
     * @param id Is an id of user we want to notify
     * @return Was the operation successful
     */
    fun notifyPerson(id: Int): Boolean
```

```
    /**
     * Checks if we can send message to the following id
     */
    fun checkId(id: Int): Boolean
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/safe/NotifyUser.kt`. You can clone this project and solve this exercise locally.

Exercise: Unit testing

Implement unit tests for the `SynchronizeUserUseCase` class in `SynchronizeUserUseCaseTest` using fake objects.

```
class CorrectCardsUseCase(
    private val view: AnkiView,
    private val cardsRepository: AnkiCardsRepository,
) {

    suspend fun start() {
        val progressBar = AnkiProgressBar(size = Small)
        view.show(progressBar)

        try {
            cardsRepository.correctCards()
        } finally {
            view.hide(progressBar)
        }

        val dialog = AnkiDialog(
            title = "Success",
            text = "Cards correction successful",
            okButton = AnkiDialog.Button("OK"),
        )
        view.show(dialog)
    }
}
```

Use `CorrectCardsUseCaseTest` and `SynchronizeCardsUseCaseTest` as inspiration. You can find those classes in the repository MarcinMoskala/kotlin-exercises in the package `effective.anki`.

Exercise: Unit testing using mocks

Implement tests for the class `SynchronizeUserUseCase` in `SynchronizeCardsUseCaseMockTest` using mocks.

```
class CorrectCardsUseCase(
    private val view: AnkiView,
    private val cardsRepository: AnkiCardsRepository,
) {

    suspend fun start() {
        val progressBar = AnkiProgressBar(size = Small)
        view.show(progressBar)

        try {
            cardsRepository.correctCards()
        } finally {
            view.hide(progressBar)
        }

        val dialog = AnkiDialog(
            title = "Success",
            text = "Cards correction successful",
            okButton = AnkiDialog.Button("OK"),
        )
        view.show(dialog)
    }
}
```

Introduce a mocking library of your choice. You can find those classes in the repository MarcinMoskala/kotlin-exercises in the package `effective.anki`.

Exercise: Composition vs inheritance

The `CorrectCardsUseCase`, `SynchronizeCardsUseCase` and `SynchronizeUserUseCase` classes share some functionalities:

- Showing a progress bar (only `CorrectCardsUseCase` and `SynchronizeCardsUseCase`),
- Handling network exceptions (only `SynchronizeCardsUseCase` and `SynchronizeUserUseCase`)
- Showing a success dialog (only `CorrectCardsUseCase` and `SynchronizeCardsUseCase`)

Extract this repeating code using:

- Inheritance
- Composition

Compare these two approaches.

You can find those classes in the repository MarcinMoskala/kotlin-exercises in the package `effective.anki`.

Exercise: Make DeckConnector comparable

Change the `DeckConnector` class:

- Implement a `compareTo` method so it first compares values based on the `deckName` property; if they are equal, then compare them based on the `state` property ordinal.
- Implement `equals` and `hashCode` methods so that two `DeckConnector` instances are equal if their `deckName` and `state` properties are equal.

```
class DeckConnector(
    val deckName: String
) : Comparable<DeckConnector> {
    var state: ConnectionState = ConnectionState.Initial

    override fun compareTo(other: DeckConnector): Int {
        TODO("Not yet implemented")
    }

    enum class ConnectionState {
        Initial,
        Connected,
        Disconnected
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/design/DeckConnector.kt`. You can clone this project and solve this exercise locally.

Exercise: Correct EventListenerRegistry

What is the potential memory leak in `EventListenerRegistry`? Fix it.

```
class EventListenerRegistry<E> {
    private val listeners = ConcurrentHashMap
        .newKeySet<EventListener<E>>()

    fun addEventListener(
        event: E,
        handler: () -> Unit
    ): EventListener<E> {
        val listener = EventListener(event, handler)
        listeners += listener
        return listener
    }

    fun invokeListeners(event: E) {
        listeners
            .filter { it.event == event && it.isActive }
            .forEach { it.handleEvent() }
    }
}

class EventListener<E>(
    val event: E,
    val handler: () -> Unit,
) {
    var isActive: Boolean = true
        private set

    fun handleEvent() {
        handler()
    }

    fun cancel() {
        isActive = false
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/efficient/EventListenerRegistry.kt`. You can clone this project and solve this exercise locally.

Exercise: Market data processing optimization

Your task is to optimize the code used to observe market data, and to display events that fulfill the given filters. The code is already working, but it is not optimized. The `Filter` class is used to filter events. The `MarketRepository` class is used to observe market data and to emit events. The `MarketRepository` class is used to emit market events. `TradeService` is used to observe market data and to display events that fulfill the given filters. Other classes are used to model market data, and to provide fake data for tests. Check how much time this code takes before and after each optimization.

```
import effective.efficient.Filter.*
import effective.efficient.Filter.Relation.*
import effective.efficient.Filter.SnapshotPart.*
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch
import java.util.concurrent.ConcurrentHashMap
import kotlin.random.Random
import kotlin.system.measureTimeMillis

data class TickerSnapshot(
    val ticker: Ticker,
    val snapshot: Snapshot,
)

data class Snapshot(
    val bid: PriceSizeTime?,
    val ask: PriceSizeTime?,
    val last: PriceSizeTime?,
)

data class PriceSizeTime(
    val price: Price,
    val size: Int? = null,
    val time: Long? = null,
)

data class Ticker(val value: String)
data class Price(val value: Float?)

sealed interface Event { val ticker: String }
data class BidEvent(override val ticker: String, val price: Float\
?, val size: Int?, val time: Long?) : Event
```

```
data class AskEvent(override val ticker: String, val price: Float\
?, val size: Int?, val time: Long?) : Event
data class TradeEvent(override val ticker: String, val price: Flo\
at?, val size: Int?, val time: Long?) : Event

val tickers = List(1000) { Ticker("Ticker$it") }

// Do not touch this one
class MarketClient {
    fun observe() = flow {
        val random = Random(123456789)
        while (true) {
            val event = when ((0..2).random(random)) {
                0 -> BidEvent(
                    tickers.random(random).value,
                    if (random.nextInt(100) == 1) null else (0..1\
00).random(random).toFloat(),
                    if (random.nextInt(100) == 1) null else (0..1\
00).random(random),
                    if (random.nextInt(100) == 1) null else Syste\
m.currentTimeMillis()
                )

                1 -> AskEvent(
                    tickers.random(random).value,
                    if (random.nextInt(100) == 1) null else (0..1\
00).random(random).toFloat(),
                    if (random.nextInt(100) == 1) null else (0..1\
00).random(random),
                    if (random.nextInt(100) == 1) null else Syste\
m.currentTimeMillis()
                )

                else -> TradeEvent(
                    tickers.random(random).value,
                    if (random.nextInt(100) == 1) null else (0..1\
00).random(random).toFloat(),
                    if (random.nextInt(100) == 1) null else (0..1\
00).random(random),
                    if (random.nextInt(100) == 1) null else Syste\
m.currentTimeMillis()
                )
            }
            emit(event)
```

```
        }
    }
}

class MarketRepository(
    private val client: MarketClient,
    backgroundScope: CoroutineScope,
) {
    private val snapshots = ConcurrentHashMap<Ticker, Snapshot>()
    private val updates = MutableSharedFlow<TickerSnapshot>()

    fun observeUpdates() = updates
        .onStart { snapshots.forEach { emit(TickerSnapshot(it.key\
, it.value)) } }

    init {
        backgroundScope.launch {
            client.observe().collect {
                when (it) {
                    is BidEvent -> {
                        val snapshot = snapshots.getOrPut(Ticker(\
it.ticker)) { Snapshot(null, null, null) }
                            .copy(bid = PriceSizeTime(Price(it.pr\
ice), it.size, it.time))
                        snapshots[Ticker(it.ticker)] = snapshot
                        updates.emit(TickerSnapshot(Ticker(it.tic\
ker), snapshot))
                    }

                    is AskEvent -> {
                        val snapshot = snapshots.getOrPut(Ticker(\
it.ticker)) { Snapshot(null, null, null) }
                            .copy(ask = PriceSizeTime(Price(it.pr\
ice), it.size, it.time))
                        snapshots[Ticker(it.ticker)] = snapshot
                        updates.emit(TickerSnapshot(Ticker(it.tic\
ker), snapshot))
                    }

                    is TradeEvent -> {
                        val snapshot = snapshots.getOrPut(Ticker(\
it.ticker)) { Snapshot(null, null, null) }
                            .copy(last = PriceSizeTime(Price(it.p\
rice), it.size, it.time))
```

```
                        snapshots[Ticker(it.ticker)] = snapshot
                        updates.emit(TickerSnapshot(Ticker(it.tic\
ker), snapshot))
                    }
                }
            }
        }
    }
}

sealed class Filter {
    data object All : Filter()
    class Or(val filters: List<Filter>) : Filter()
    class And(val filters: List<Filter>) : Filter()
    class PrizeCondition(
        val snapshotPart: SnapshotPart,
        val relation: Relation,
        val value: Float,
    ) : Filter()

    class TickerIs(val tickers: List<Ticker>) : Filter()
    class Not(val filter: Filter) : Filter()

    enum class SnapshotPart {
        Ask, Bid, Last, Spread
    }

    enum class Relation {
        GreaterThan, LessThan, Equal
    }

    fun check(tickerSnapshot: TickerSnapshot): Boolean = when (th\
is) {
        All -> true
        is Or -> filters.any { it.check(tickerSnapshot) }
        is And -> filters.all { it.check(tickerSnapshot) }
        is PrizeCondition -> run {
            val snapshotPrize = when (snapshotPart) {
                Ask -> tickerSnapshot.snapshot.ask?.price?.value \
?: return@run false
                Bid -> tickerSnapshot.snapshot.bid?.price?.value \
?: return@run false
                Last -> tickerSnapshot.snapshot.last?.price?.valu\
e ?: return@run false
```

```
                Spread -> {
                    val bid = tickerSnapshot.snapshot.bid?.price?\
.value ?: return@run false
                    val ask = tickerSnapshot.snapshot.ask?.price?\
.value ?: return@run false
                    ask - bid
                }
            }
            when (relation) {
                GreaterThan -> snapshotPrize > value
                LessThan -> snapshotPrize < value
                Equal -> snapshotPrize == value
            }
        }

        is TickerIs -> tickers.contains(tickerSnapshot.ticker)
        is Not -> !filter.check(tickerSnapshot)
    }
}

class TradeService(
    private val repository: MarketRepository,
) {
    fun observeUpdates(
        filter: Filter,
        tickers: List<Ticker>? = null,
    ) = repository.observeUpdates()
        .filter { tickers == null || it.ticker in tickers }
        .filter { filter.check(it) }
}

suspend fun main() {
    val client = MarketClient()
    val repository = MarketRepository(client, backgroundScope = C\
oroutineScope(SupervisorJob()))
    val service = TradeService(repository)
    val filter = Or(
        listOf(
            And(listOf(TickerIs(tickers.take(1)), PrizeCondition(\
Ask, GreaterThan, 99f))),
            And(listOf(PrizeCondition(Spread, GreaterThan, 99f))),
        )
    )
```

```
    measureTimeMillis {
        service.observeUpdates(
            filter = filter,
            tickers = tickers.take(70)
        ).take(1_000)
            .collect { println(it) }
    }.let { println("Took $it") }
}
```

Exercise: Prime access list

Associating elements to a map can be an important performance optimization. Finding an element by key in a map is a lot faster than iterating over a list and comparing each element to the searched value. To see the difference, implement methods for the `PrimeAccessRepository` class:

- `isOnAllowList` should return `true` if the user is on the allowlist (the entry with this user id has `allowList` set to `true`); otherwise, it should return `false`,
- `isOnDenyList` should return `true` if the user is on the denylist (the entry with this user id has `denyList` set to `true`); otherwise, it should return `false`.

```
class PrimeAccessRepository(
    private val primeAccessList: PrimeAccessList
) {
    fun isOnAllowList(userId: String): Boolean = TODO()
    fun isOnDenyList(userId: String): Boolean = TODO()
}

class PrimeAccessList(
    val entries: List<PrimeAccessEntry>
)

class PrimeAccessEntry(
    val userId: String,
    val allowList: Boolean,
    val denyList: Boolean,
)
```

Implement two kinds of solutions:

- iterate over entries to find the desired user id,
- associate entries to a map by user id in the class body, and find the entry by user id in the methods.

Check the efficiency of each of these solutions using the following code:

```
val entries = List(200_000) {
    PrimeAccessEntry(
        userId = it.toString(),
        allowList = Random.nextBoolean(),
        denyList = Random.nextBoolean()
    )
}.shuffled()
val accessList = PrimeAccessList(entries)

val repo: PrimeAccessRepository
measureTimeMillis {
    repo = PrimeAccessRepository(accessList)
}.also { println("Class creation took $it ms") }

measureTimeMillis {
    for (userId in 1L..10_000L) {
        repo.isOnAllowList(userId.toString())
    }
}.also { println("Operation took $it ms") }

measureTimeMillis {
    for (userId in 1L..10_000L) {
        repo.isOnDenyList(userId.toString())
    }
}.also { println("Operation took $it ms") }
```

> Beware! Such measurements are not precise. They are only to show the difference between the two solutions. For precise measurements, you should use a benchmarking library, like JMH.

Starting code and example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `effective/collections/PrimeAccess.kt`. You can clone this project and solve this exercise locally.

Exercise: Crime analysis

Your task is to analyze crime data from Chicago in order to count the number of crimes per primary cause. You've implemented the solution, but it fails due to `OutOfMemoryError`. Fix it.

```
fun main() {
    measureTimeMillis {
        File("Crimes_-_2001_to_Present.csv")
            .readLines()
            .drop(1)
            .map { Crime.parse(it) }
            .groupBy { it.primaryType }
            .mapValues { it.value.size }
            .toList()
            .sortedByDescending { (_, num) -> num }
            .joinToString(separator = "\n") { (type, num) ->
                "$num $type"
            }
            .let(::println)
    }.let { println("Took $it") }
}
```

Download data in CSV format from the following link:

https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2/data

Compare execution time of the following solutions that use the `groupBy` function to group elements, and the same solutions that use `groupingBy` function to group elements.

Exercise solutions

Solution: Basic values operations

```
fun main() {
    println(1 + 2 * 3) // 7
    println(10 % 3) // 1
    println(-1 % 3) // -1

    println(8.8 / 4) // 2.2
    println(10 / 3) // 3

    println(11.toFloat()) // 11.0
    println(10.10.toInt()) // 10

    var a = 10
    a += 5
    println(a) // 15
    a -= 3
    println(a) // 12
    a++
    println(a) // 13
    println(a++) // 13
    println(a) // 14
    println(--a) // 13
    println(a) // 13

    println(true && false) // false
    println(true || false) // true
    println(!!!!true) // true

    println('A'.code) // 65
    println('A' + 1) // B
    println('C'.code) // 67

    println("A + B") // A + B
    println("A" + "B") // AB
    println("A" + 1) // A1
    println("A" + 1 + 2) // A12
}
```

Solution: Using when

```
private val magicNumbers = listOf(7, 13)

fun name(a: Any?): String = when (a) {
    null -> "Nothing"
    1, 2, 3 -> "Small number"
    in magicNumbers -> "Magic number"
    in 4..100 -> "Big number"
    is String -> "String: $a"
    is Int, is Long -> "Int or Long: $a"
    else -> "No idea, really"
}

fun main() {
    println(name(1)) // Small number
    println(name("A")) // String: A
    println(name(null)) // Nothing
    println(name(5)) // Big number
    // (because 5 is in 4..100)
    println(name(100)) // Big number
    // (because 100 is in 4..100)
    println(name('A')) // No idea, really
    // (because 'A' is Char)
    println(name("1")) // String: 1
    println(name(-1)) // Int or Long: -1
    println(name(101)) // Int or Long: 101
    // (because 101 is greater than 100, so not in 4..100)
    println(name(1L)) // Int or Long: 1
    // (because 1L is Long)
    println(name(7)) // Magic number
    // (because 7 is in magicNumbers collection)
    println(name(3)) // Small number
    println(name(3.0)) // No idea, really
    // (because 3.0 is Double)
    println(name(100L)) // Int or Long: 100
    // (because 100L is Long)
}
```

Solution: Pretty time display

```
fun secondsToPrettyTime(seconds: Int): String {
    if (seconds < 0) return "Invalid input"
    if (seconds == 0) return "Now"
    // or
    // when {
    //     seconds < 0 -> return "Invalid input"
    //     seconds == 0 -> return "Now"
    // }

    val hours = seconds / SECONDS_IN_HOUR
    val minutes = (seconds % SECONDS_IN_HOUR)/MINUTES_IN_HOUR
    val secondsLeft = seconds % SECONDS_IN_MINUTE

    var result = ""
    if (hours > 0) {
        result += "$hours h"
    }
    if (minutes > 0) {
        result += " $minutes min"
    }
    if (secondsLeft > 0) {
        result += " $secondsLeft sec"
    }
    return result.trim()
}

// Constants should be defined outside of the function
// we often name them using UPPER_SNAKE_CASE convention
// const modifier is a low-level optimization
private const val SECONDS_IN_MINUTE = 60
private const val MINUTES_IN_HOUR = 60
private const val SECONDS_IN_HOUR = 3600
```

This is not the only possible solution to this exercise. Especially transforming `hours`, `minutes` and `secondsLeft` into a string can be done in many different ways. I will show you a few popular solutions to this problem. Some of them use functions like `listOfNotNull`, `joinToString`, `takeIf` or `buildString` that are presented in the Functional Kotlin book (the next book I recommend you reading after Kotlin Essentials). Have patience, you will learn about them later, for now you can just treat them as curious examples of Kotlin code.

```
// Option 1 - This option is similar to the original,
// but uses if as an expression, and adds strings with +
val hoursString = if (hours > 0) "$hours h " else ""
val minutesString = if (minutes > 0) "$minutes min " else ""
val secondsString =
    if (secondsLeft > 0) "$secondsLeft sec " else ""
return (hoursString + minutesString + secondsString).trim()

// Option 2 - This option uses listOfNotNull to construct
// a list of non-null elements, and then transforms those
// elements into a single string using joinToString
return listOfNotNull(
    if (hours > 0) "$hours h" else null,
    if (minutes > 0) "$minutes min" else null,
    if (secondsLeft > 0) "$secondsLeft sec" else null
).joinToString(separator = " ")

// Option 3 - This option is similar to the previous one,
// but uses takeIf to filter out null elements
return listOfNotNull(
    "$hours h".takeIf { hours > 0 },
    "$minutes min".takeIf { minutes > 0 },
    "$secondsLeft sec".takeIf { secondsLeft > 0 },
).joinToString(separator = " ")

// Option 4 - This option uses buildString function
// to construct a string by appending it with next parts
// inside lambda expression block
return buildString {
    if (hours > 0) append("$hours h ")
    if (minutes > 0) append("$minutes min ")
    if (secondsLeft > 0) append("$secondsLeft sec ")
}.trim()

// Option 5 - This is the most complicated option,
// because it covers all possible cases in a when-expression.
// I do not recommend using this solution, but I wanted to
// show it as an example of how when-expression can be used.
return when {
    hours == 0 && minutes == 0 && secondsLeft == 0 -> ""
    hours == 0 && minutes == 0 -> "$secondsLeft sec"
    hours == 0 && secondsLeft == 0 -> "$minutes min"
    minutes == 0 && secondsLeft == 0 -> "$hours h"
    minutes == 0 -> "$hours h $secondsLeft sec"
```

```
    secondsLeft == 0 -> "$hours h $minutes min"
    hours == 0 -> "$minutes min $secondsLeft sec"
    else -> "$hours h $minutes min $secondsLeft sec"
}
```

Solution: Person details display

This is the expected solution for this exercise:

```
fun formatPersonDisplay(
    name: String? = null,
    surname: String? = null,
    age: Int? = null,
): String {
    var result = ""
    if (name != null) {
        result += name
    }
    if (surname != null) {
        result += " $surname"
    }
    if (age != null) {
        result += " ($age)"
    }
    return result.trim()
}
```

This exercise can also be solved using functions that are presented in the Functional Kotlin book (the next book I recommend you reading after Kotlin Essentials). This is a solution using `listOfNotNull` and `joinToString` functions:

```
fun formatPersonDisplay(
    name: String? = null,
    surname: String? = null,
    age: Int? = null,
): String = listOfNotNull(
    name,
    surname,
    if (age != null) "($age)" else null
    // or age?.let { "($it)" }
).joinToString(separator = " ")
```

This is a solution using `buildString` function (it allows constructing a string by appending it with next parts inside lambda expression block):

```
fun formatPersonDisplay(
    name: String? = null,
    surname: String? = null,
    age: Int? = null,
): String = buildString {
    if (name != null) append("$name")
    if (surname != null) append(" $surname")
    if (age != null) append(" ($age)")
}.trim()
```

Solution: Range Operations

This is the expected solution:

```
fun calculateSumOfSquares(n: Int): Int {
    var sum = 0
    for (i in 1..n) {
        sum += i * i
    }
    return sum
}

fun calculateSumOfEven(n: Int): Int {
    var sum = 0
    for (i in 2..n step 2) {
        sum += i
    }
    return sum
}

fun countDownByStep(start: Int, end: Int, step: Int): String {
    var result = ""
    for (i in start downTo end step step) {
        result += i
        if (i != end) {
            result += ", "
        }
    }
    return result
}
```

This exercise can also be solved using functions that are presented in the Functional Kotlin book (the next book I recommend you reading after Kotlin Essentials). This is a solution using `fold` function:

```
fun calculateSumOfSquares(n: Int): Int =
    (1..n).fold(0) { sum, i -> sum + i * i }

fun calculateSumOfEven(n: Int): Int =
    (2..n step 2).fold(0) { sum, i -> sum + i }

fun countDownByStep(start: Int, end: Int, step: Int): String =
    (start downTo end step step).fold("") { result, i ->
        result + i + if (i != end) ", " else ""
    }
```

This is a solution using `sumOf`, `sum` and `joinToString` functions:

```
fun calculateSumOfSquares(n: Int): Int = (1..n).sumOf { it * it }

fun calculateSumOfEven(n: Int): Int = (0..n step 2).sum()

fun countDownByStep(start: Int, end: Int, step: Int): String =
    (start downTo end step step).joinToString(separator = ", ")
```

Solution: User Information Processor

```
fun processUserInformation(user: User?): String {
    if (user == null) {
        return "Missing user information"
    }
    val name = requireNotNull(user.name)
    val age = user.age ?: 0
    val email = user.email?.email
    if (email.isNullOrBlank()) {
        return "Missing email"
    }

    return "User $name is $age years old, email: $email"
}
```

Solution: Implementing the Product class

```
class Product(
    val name: String,
    val price: Double,
    initialQuantity: Int
) {
    var quantity: Int = initialQuantity
        set(value) {
            field = if (value >= 0) value else 0
        }

    fun calculateTotalValue(): Double {
        return price * quantity
    }

    fun restock(additionalQuantity: Int) {
        if (additionalQuantity > 0) {
            quantity += additionalQuantity
        }
    }
}
```

Solution: GUI View Hierarchy Simulation

```
open class View(
    val id: String,
    var isVisible: Boolean
) {
    fun show() {
        isVisible = true
    }

    fun hide() {
        isVisible = false
    }
}

class TextView(
    id: String,
    var text: String
) : View(id, true)

class Toggle(
    id: String,
) : View(id, true) {
    var isOn: Boolean = false

    fun click() {
        isOn = !isOn
    }
}
```

Solution: Data class practice

```
data class Person(val name: String, val age: Int)

fun main() {
    val person1 = Person("John", 30)
    println(person1) // Person(name=John, age=30)
    val person2 = person1.copy(name = "Jane")
    val person3 = Person("Jane", 30)
    println(person2 == person3) // true
    println(person1.hashCode()) // 71750739
    println(person2.hashCode()) // 71339152
    println(person3.hashCode()) // 71339152
    val (name, age) = person2
    println(name) // Jane
    println(age) // 30
}
```

Solution: Pizza factory

```
class Pizza(
    val toppings: List<String>,
) {
    companion object {
        fun hawaiian() = Pizza(listOf("ham", "pineapple"))

        fun margherita() = Pizza(listOf("tomato", "mozzarella"))
    }
}
```

Solution: Catching exceptions

```
fun main() {
    while (true) {
        try {
            handleInput()
        } catch (e: NumberFormatException) {
            println("Invalid input: ${e.message}")
        } catch (e: ArithmeticException) {
            println("Division by zero")
        } catch (e: IllegalOperatorException) {
            println("Illegal operator: ${e.operator}")
        }
    }
}
```

Solution: Days of the week enum

```
enum class DayOfWeek(
    val isWeekend: Boolean,
    val dayName: String,
) {
    MONDAY(false, "Monday"),
    TUESDAY(false, "Tuesday"),
    WEDNESDAY(false, "Wednesday"),
    THURSDAY(false, "Thursday"),
    FRIDAY(false, "Friday"),
    SATURDAY(true, "Saturday"),
    SUNDAY(true, "Sunday");

    fun nextDay(): DayOfWeek {
        val days = DayOfWeek.entries //or DayOfWeek.values()
        val currentIndex = days.indexOf(this)
        val nextIndex = (currentIndex + 1) % days.size
        return days[nextIndex]
    }
}
```

Solution: Conversion and measurement unit creation

```
fun User.toUserJson(): UserJson = UserJson(
    username = username,
    email = email.value,
    registrationDate = registrationDate.toString(),
    heightCm = height.value,
)

fun UserJson.toUser(): User = User(
    username = username,
    email = Email(email),
    registrationDate = LocalDateTime.parse(registrationDate),
    height = heightCm.cm,
)

val Int.cm: Centimeters get() = Centimeters(this)
```

Solution: Inventory management

```
class Inventory {
    private val products = mutableListOf<Product>()
    private val productIdToProducer =
        mutableMapOf<String, String>()
    private val sellers = mutableSetOf<String>()

    fun addProduct(product: Product, producer: String) {
        products.add(product)
        productIdToProducer[product.id] = producer
    }

    fun removeProduct(product: Product) {
        products.remove(product)
        productIdToProducer.remove(product.id)
    }

    fun addSeller(seller: String) {
        sellers.add(seller)
    }

    fun removeSeller(seller: String) {
        sellers.remove(seller)
    }

    fun getProductsCount() = products.size

    fun hasProduct(product: Product) =
        products.contains(product)

    fun hasProducts() = products.isNotEmpty()

    fun getProducer(product: Product) =
        productIdToProducer[product.id]

    fun produceInventoryDisplay(): String {
        var result = "Inventory:\n"
        for (product in products) {
            val name = product.name
            val category = product.category
            val price = product.price
            result += "$name ($category) - $price\n"
            val producer = productIdToProducer[product.id]
```

```
            result += "Produced by: $producer\n"
        }
        result += "Sellers: $sellers"
        return result
    }
}
```

Solution: Money operations

```
data class Money(
    val amount: BigDecimal,
    val currency: Currency
) {
    operator fun plus(other: Money): Money {
        require(currency == other.currency) {
            "Cannot add money of different currencies"
        }
        return Money(amount + other.amount, currency)
    }

    operator fun minus(other: Money): Money {
        require(currency == other.currency) {
            "Cannot subtract money of different currencies"
        }
        return Money(amount - other.amount, currency)
    }

    operator fun unaryMinus(): Money = Money(-amount, currency)

    operator fun times(times: Int): Money =
        Money(amount * times.toBigDecimal(), currency)

    companion object {
        fun eur(amount: String) =
            Money(BigDecimal(amount), Currency.EUR)
    }
}
```

Solution: The closest supertype of types

- `Int` and `Double` -> `Number`
- `Double` and `Number` -> `Number`
- `String` and `Nothing` -> `String`
- `Float` and `Double?` -> `Number?`
- `String` and `Float` -> `Any`
- `Char` and `Nothing?` -> `Char?`
- `Nothing` and `Any` -> `Any`
- `Nothing?` and `Any` -> `Any?`
- `Char?` and `Nothing?` -> `Char?`
- `Nothing?` and `Any?` -> `Any?`

Solution: Stock

```
class Stack<T> {
    private val elements: MutableList<T> = mutableListOf()

    fun push(item: T) {
        elements.add(item)
    }

    fun pop(): T? =
        if (isEmpty()) null
        else elements.removeAt(elements.size - 1)

    fun peek(): T? = elements.lastOrNull()

    fun isEmpty(): Boolean = elements.isEmpty()

    fun size(): Int = elements.size
}
```

Solution: Workout manager

```
import java.io.File

interface NextRepetitionsRepository {
    fun read(): Repetitions?
    fun write(repetitions: Repetitions)
}

class FileNextRepetitionsRepository: NextRepetitionsRepository {
    private val store = File("next_repetitions.txt")

    override fun read(): Repetitions? {
        if (!store.exists()) return null
        val content = store.readLines()
        val firstRound = content.getOrNull(0)?.toIntOrNull()
            ?: return null
        val secondRound = content.getOrNull(1)?.toIntOrNull()
            ?: return null
        return Repetitions(
            firstRound = firstRound,
            secondRound = secondRound,
        )
    }

    override fun write(repetitions: Repetitions) {
        if (!store.exists()) store.createNewFile()
        store.writeText("${repetitions.firstRound}\n"+
            "${repetitions.secondRound}")
    }
}

data class Repetitions(val firstRound: Int, val secondRound: Int)

class WorkoutManager {
    fun nextRepetitions(
        firstRoundDone: Int,
        secondRoundDone: Int,
        thirdRoundDone: Int,
        repetitions: Repetitions,
    ): Repetitions = when {
        firstRoundDone < repetitions.firstRound -> {
            // Could not do enough push-ups in the first round
            repetitions.copy(
```

```
                firstRound = repetitions.firstRound - 1,
                secondRound = repetitions.secondRound - 1
            )
        }
        secondRoundDone < repetitions.secondRound -> {
            // Could not do enough push-ups in the second round
            repetitions.copy(
                secondRound = repetitions.secondRound - 1
            )
        }
        thirdRoundDone > repetitions.firstRound -> {
            // The third round is above the first round,
            // so we increase the repetitions for both rounds
            repetitions.copy(
                firstRound = repetitions.firstRound + 1,
                secondRound = repetitions.secondRound + 1,
            )
        }
        thirdRoundDone > repetitions.secondRound -> {
            // The third round is below the first round,
            // but above the second round, so we increase
            // the repetitions for the second round
            repetitions.copy(
                secondRound = repetitions.secondRound + 1,
            )
        }
        else -> repetitions
    }
}

class WorkoutInteractor(
    val manager: WorkoutManager,
    val nextRepetitionsRepository: NextRepetitionsRepository
) {
    fun start() {
        println("Hello, I am your push-ups workout assistant!")
        val repetitions = nextRepetitionsRepository.read()
            ?: Repetitions(5, 5)
        println("Now do ${repetitions.firstRound} push-ups")
        val firstRoundDone = getValidIntInput(
              "How many push-ups did you do?"
        )
        println("Now rest for 1 minute")
        Thread.sleep(1000 * 60)
```

```
        println("Now do ${repetitions.secondRound} push-ups")
        val secondRoundDone = getValidIntInput(
              "How many push-ups did you do?"
        )
        println("Now rest for 1 minute")
        Thread.sleep(1000 * 60)
        println("Now do as many push-ups as you can!")
        val thirdRoundDone = getValidIntInput(
            "How many push-ups did you do?"
        )
        val next = manager.nextRepetitions(
            firstRoundDone = firstRoundDone,
            secondRoundDone = secondRoundDone,
            thirdRoundDone = thirdRoundDone,
            repetitions = repetitions,
        )
        nextRepetitionsRepository.write(next)
        println("Your next repetitions will be: "+
            "${next.firstRound} and ${next.secondRound}")
    }

    private fun getValidIntInput(prompt: String): Int {
        while (true) {
            println(prompt)
            val input = readlnOrNull()?.toIntOrNull()
            if (input != null && input >= 0) {
                return input
            }
            println("Please enter a valid number.")
        }
    }
}

fun main() {
    val interactor = WorkoutInteractor(
        manager = WorkoutManager(),
        nextRepetitionsRepository =
            FileNextRepetitionsRepository(),
    )
    interactor.start()
}
```

Solution: Function types and literals

```
class AnonymousFunctionalTypeSpecified {
    val add: (Int, Int) -> Int = fun(num1, num2) = num1 + num2
    val printNum: (Int) -> Unit = fun(num) { print(num) }
    val triple: (Int) -> Int = fun(num) = num * 3
    val produceName: (String) -> Name = fun(name) = Name(name)
    val longestOf: (String, String, String) -> String =
        fun(str1, str2, str3) =
            maxOf(str1, str2, str3, compareBy { it.length })
}

class AnonymousFunctionalTypeInferred {
    val add = fun(num1: Int, num2: Int) = num1 + num2
    val printNum = fun(num: Int) { print(num) }
    val triple = fun(num: Int) = num * 3
    val produceName = fun(name: String) = Name(name)
    val longestOf =
        fun(str1: String, str2: String, str3: String) =
            maxOf(str1, str2, str3, compareBy { it.length })
}

class LambdaFunctionalTypeSpecified {
    val add: (Int, Int) -> Int = { num1, num2 -> num1 + num2 }
    val printNum: (Int) -> Unit = { num -> print(num) }
    val triple: (Int) -> Int = { num -> num * 3 }
    val produceName: (String) -> Name = { name -> Name(name) }
    val longestOf: (String, String, String) -> String =
        { str1, str2, str3 ->
            maxOf(str1, str2, str3, compareBy { it.length })
        }
}

class LambdaFunctionalTypeInferred {
    val add = { num1: Int, num2: Int -> num1 + num2 }
    val printNum = { num: Int -> print(num) }
    val triple = { num: Int -> num * 3 }
    val produceName = { name: String -> Name(name) }
    val longestOf =
        { str1: String, str2: String, str3: String ->
            maxOf(str1, str2, str3, compareBy { it.length })
        }
}
```

```
class LambdaUsingImplicitParameter {
    val add: (Int, Int) -> Int = { num1, num2 -> num1 + num2 }
    val printNum: (Int) -> Unit = { print(it) }
    val triple: (Int) -> Int = { it * 3 }
    val produceName: (String) -> Name = { Name(it) }
    val longestOf: (String, String, String) -> String =
        { str1, str2, str3 ->
            maxOf(str1, str2, str3, compareBy { it.length })
        }
}
```

Solution: Observable value

```
class Observable<T>(initial: T) {
    var value: T = initial
        set(value) {
            field = value
            observers.forEach { it(value) }
        }
    private val observers = mutableListOf<(T) -> Unit>()

    fun observe(observer: (T) -> Unit) {
        observers.add(observer)
    }
}
```

Solution: Inferred function types

- `Centimeter::plus` function type is `(Centimeter, Centimeter) -> Centimeter`
- `Centimeter::times` function type is `(Centimeter, Double) -> Centimeter`
- `Centimeter::value` function type is `(Centimeter) -> Double`
- `Centimeter::toString` function type is `(Centimeter) -> String`
- `Centimeter(1.0)::plus` function type is `(Centimeter) -> Centimeter`
- `Centimeter(2.0)::times` function type is `(Double) -> Centimeter`
- `Centimeter(3.0)::value` function type is `() -> Double`
- `Centimeter(4.0)::toString` function type is `() -> String`
- `Int::cm` function type is `(Int) -> Centimeter`
- `123::cm` function type is `() -> Centimeter`
- `::distance` function type is `(Centimeter, Centimeter) -> Centimeter`

Solution: Function references

```
class FunctionReference {
    val add: (Int, Int) -> Int = Int::plus
    val printNum: (Int) -> Unit = ::print
    val triple: (Int) -> Int = 3::times
    val produceName: (String) -> Name = ::Name
}

class FunctionMemberReference {
    val add: (Int, Int) -> Int = this::add
    val printNum: (Int) -> Unit = this::printNum
    val triple: (Int) -> Int = this::triple
    val produceName: (String) -> Name = this::produceName
    val longestOf: (String, String, String) -> String =
        this::longestOf

    private fun add(num1: Int, num2: Int): Int = num1 + num2

    private fun printNum(num: Int) {
        print(num)
    }

    private fun triple(num: Int): Int = num * 3

    private fun produceName(name: String): Name = Name(name)

    private fun longestOf(
        str1: String,
        str2: String,
        str3: String
    ): String =
        maxOf(str1, str2, str3, compareBy { it.length })
}

class BoundedFunctionReference {
    private val classic = FunctionsClassic()

    val add: (Int, Int) -> Int = classic::add
    val printNum: (Int) -> Unit = classic::printNum
    val triple: (Int) -> Int = classic::triple
    val produceName: (String) -> Name = classic::produceName
    val longestOf: (String, String, String) -> String =
        classic::longestOf
```

```
}
```

Solution: Inline functions

Solutions using for-loops:

```
inline fun <reified T> Iterable<*>.anyOf(): Boolean {
    for (element in this) {
        if (element is T) return true
    }
    return false
}

inline fun <reified T> Iterable<*>.firstOfOrNull(): T? {
    for (element in this) {
        if (element is T) return element
    }
    return null
}

inline fun <reified T, reified R> Map<*, *>
        .filterValuesInstanceOf(): Map<T, R> {
    val result = mutableMapOf<T, R>()
    for ((key, value) in this) {
        if (key is T && value is R) {
            result[key] = value
        }
    }
    return result
}
```

Solutions using collection processing functions:

```
inline fun <reified T> Iterable<*>.anyOf(): Boolean =
    any { it is T }

inline fun <reified T> Iterable<*>.firstOfOrNull(): T? =
    firstOrNull { it is T } as? T

inline fun <reified T, reified R> Map<*, *>
        .filterValuesInstanceOf(): Map<T, R> =
    filter { it.key is T && it.value is T } as Map<T, R>
```

Solution: Implement map

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

Solution: Optimize collection processing

```
fun List<StudentJson>.getPassingSurnames(): List<String> = this
    .filter { it.result >= 50 && it.pointsInSemester >= 15 }
    .mapNotNull { it.surname }
```

I needed to break the function definition to fit the page. In actual code, I would not do that.

Solution: Adding element at position

Solution using a mutable collection:

```
fun <T> List<T>.plusAt(index: Int, element: T): List<T> {
    require(index in 0..size)
    val result = toMutableList()
    result.add(index, element)
    return result
}
```

Solution using `take` and `drop`:

```
fun <T> List<T>.plusAt(index: Int, element: T): List<T> {
    require(index in 0..size)
    return take(index) + element + drop(index)
}
```

Solution using `flatMapIndexed`:

```
fun <T> List<T>.plusAt(index: Int, element: T): List<T> {
    require(index in 0..size)
    return when (index) {
        0 -> listOf(element) + this
        size -> this + element
        else -> flatMapIndexed { i, e ->
            if (i == index) listOf(element, e)
            else listOf(e)
        }
    }
}
```

Solution: Implement shop functions

```
fun Shop.getWaitingCustomers(): List<Customer> = customers
    .filter { it.orders.any { !it.isDelivered } }

fun Shop.countProductSales(product: Product): Int = customers
    .flatMap { it.orders }
    .flatMap { it.products }
    .count { it == product }
// or (with better performance)
//  .sumOf {
//      it.orders.sumOf {
//          it.products.count { it == product }
//      }
//  }

fun Shop.getCustomers(minAmount: Double): List<Customer> =
    customers.filter {
        it.orders.sumOf {
            it.products.sumOf { it.price }
        } >= minAmount
    }
```

Solution: Prime access list

This is what the solution that requires iterating over entries to find the searched user id could look like:

```
class PrimeAccessRepository(
    private val primeAccessList: PrimeAccessList,
) {
    fun isOnAllowList(userId: String): Boolean =
        primeAccessList.entries
            .find { it.userId == userId }
            ?.allowList
            ?: false

    fun isOnDenyList(userId: String): Boolean =
        primeAccessList.entries
            .find { it.userId == userId }
            ?.denyList
            ?: false
}
```

On my machine (Apple M2 Pro), performance measurement gives the following results:

```
Class creation took 0 ms
Operation took 2926 ms
Operation took 2831 ms
```

The optimized solution that associates entries to a map by user id in the class body and finds the entry by user id in the methods could look like this:

```
class PrimeAccessRepository(
   primeAccessList: PrimeAccessList,
) {
   private val entries = primeAccessList.entries
        .associateBy { it.userId }

   fun isOnAllowList(userId: String): Boolean =
       entries[userId]?.allowList ?: false

   fun isOnDenyList(userId: String): Boolean =
       entries[userId]?.denyList ?: false
}
```

On my machine (Apple M2 Pro), performance measurement gives the following results:

```
Class creation took 32 ms
Operation took 2 ms
Operation took 2 ms
```

This means that class creation is slightly slower (it requires one iteration over entries to create the map), but operations are much faster (they require only one map access).

Solution: Top articles

```
class TopArticlesGenerator(
    private val articles: List<ArticleStatistics>,
) {
    fun topArticles(n: Int): List<ArticleStatistics> = articles
        .withIndex()
        .sortedByDescending { it.value.views }
        .take(n)
        .sortedBy { it.index }
        .map { it.value }
}
```

Solution: Refactor collection processing

```
fun List<StudentGrades>.getBestForScholarship(
    semester: String
): List<StudentGrades> = this
    .filter { s ->
        s.grades
            .filter { it.semester == semester && it.passing }
            .sumOf { it.ects } > 30
    }
    .sortedByDescending {
        averageGradeFromSemester(it, semester)
    }
    .take(10)

fun averageGradeFromSemester(
    student: StudentGrades,
    semester: String
): Double = student.grades
    .filter { it.semester == semester }
    .map { it.grade }
    .average()
```

Solution: Passing students list

```
fun List<Student>.makePassingStudentsList(): String = this
    .filter { it.pointsInSemester > 15 && it.result >= 50 }
    .sortedWith(compareBy({ it.surname }, { it.name }))
    .joinToString(separator = "\n") {
        "${it.name} ${it.surname}, ${it.result}"
    }
```

Solution: Best students list

```
fun List<Student>.makeBestStudentsList(): String = this
    .filter { it.pointsInSemester >= 30 && it.result >= 80 }
    .sortedByDescending { it.result }
    .zip(INTERNSHIPS)
    .sortedWith(compareBy({ it.first.surname },{ it.first.name }))
    .joinToString(separator = "\n") {(student, internship)->
        "${student.name} ${student.surname}, $$internship"
    }

private val INTERNSHIPS =
    List(1) { 5_000 } + List(3) { 3_000 } + List(6) { 1_000 }
```

Solution: Functional Quick Sort

```
fun <T : Comparable<T>> List<T>.quickSort(): List<T> {
    if (this.size <= 1) return this
    val pivot = this.first()
    val (smaller, bigger) = this.drop(1).partition { it < pivot }
    return smaller.quickSort() + pivot + bigger.quickSort()
}
```

Solution: Powerset

```
fun <T> Collection<T>.powerset(): Set<Set<T>> {
    if (this.isEmpty()) return setOf(setOf())
    val element = this.first()
    val rest = this.drop(1).powerset()
    return rest + rest.map { it + element }
}
```

Solution: All possible partitions of a set

```
fun <T> Collection<T>.partitions(): Set<Set<Set<T>>> {
    if (isEmpty()) return setOf()
    if (size == 1) return setOf(setOf(setOf(first())))
    val head = first()
    val tailPartitions = drop(1).partitions()
    val whereHeadIsAlone: Set<Set<Set<T>>> = tailPartitions
        .map { it + setOf(setOf(head)) }
        .toSet()
    val whereHeadIsNotAlone: Set<Set<Set<T>>> = tailPartitions
        .flatMap { partition: Set<Set<T>> ->
            partition.map { subset: Set<T> ->
                partition
                    .minusElement(subset)
                    .plusElement(subset + head)
            }
        }
        .toSet()
    return whereHeadIsAlone + whereHeadIsNotAlone
}
```

Solution: Understanding sequences

```
fun m(i: Int): Int {
    print("m$i ")
    return i * i
}
fun f(i: Int): Boolean {
    print("f$i ")
    return i % 2 == 0
}

fun main() {
    val list = listOf(1, 2, 3, 4)
    list.map(::m).filter(::f)
    // m1 m2 m3 m4 f1 f4 f9 f16

    list.filter(::f).map(::m)
    // f1 f2 f3 f4 m2 m4
    // (notice that using filter first is more efficient)

    val sequence = sequenceOf(1, 2, 3, 4)
    sequence.map(::m).filter(::f).toList()
    // m1 f1 m2 f4 m3 f9 m4 f16

    sequence.map(::m).filter(::f)
    // (nothing)
    // (a sequence does nothing until a terminal operation)

    sequence.map(::m).filter(::f).first()
    // m1 f1 m2 f4

    sequence.filter(::f).map(::m).toList()
    // f1 f2 m2 f3 f4 m4

    val sequence2 = list.asSequence().map(::m)
    // (nothing)
    // (a sequence does nothing until a terminal operation)

    sequence2.toList()
    // m1 m2 m3 m4

    sequence2.filter(::f).toList()
    // m1 f1 m2 f4 m3 f9 m4 f16
}
```

Solution: HTML table DSL

```
fun table(init: TableBuilder.() -> Unit): TableBuilder =
    TableBuilder().apply(init)

data class TableBuilder(
    var trs: List<TrBuilder> = emptyList()
) {
    fun tr(init: TrBuilder.() -> Unit) {
        trs += TrBuilder().apply(init)
    }

    override fun toString(): String =
        "<table>${trs.joinToString(separator = "")}</table>"
}

data class TrBuilder(
    var tds: List<TdBuilder> = emptyList()
) {
    fun td(init: TdBuilder.() -> Unit) {
        tds += TdBuilder().apply(init)
    }

    override fun toString(): String =
        "<tr>${tds.joinToString(separator = "")}</tr>"
}

data class TdBuilder(var text: String = "") {
    operator fun String.unaryPlus() {
        text += this
    }

    override fun toString(): String = "<td>$text</td>"
}
```

Solution: Creating user table row

```
private fun TableBuilder.userRow(user: User) {
    tr {
        td { +user.id }
        td { +user.name }
        td { +user.points.toString() }
        td { +user.category }
    }
}
```

Solution: Using scope functions

```
class StudentService(
    private val studentRepository: StudentRepository,
    private val studentFactory: StudentFactory,
    private val logger: Logger,
) {
    fun addStudent(addStudentRequest: AddStudentRequest):Student?=
        addStudentRequest
            .let { studentFactory.produceStudent(it) }
            // or .let(studentFactory::produceStudent)
            ?.also { studentRepository.addStudent(it) }
            // or ?.also(studentRepository::addStudent)

    fun getStudent(studentId: String): ExposedStudent? =
        studentRepository
            .getStudent(studentId)
            ?.also { logger.log("Student found: $it") }
            ?.let { studentFactory.produceExposed(it) }
            // or ?.let(studentFactory::produceExposed)

    fun getStudents(semester: String): List<ExposedStudent> =
        produceGetStudentsRequest(semester)
            .let { studentRepository.getStudents(it) }
            // or .let(studentRepository::getStudents)
            .also {logger.log("${it.size} students in $semester")}
            .map { studentFactory.produceExposed(it) }
            // or .map(studentFactory::produceExposed)

    private fun produceGetStudentsRequest(
        semester: String,
    ) = GetStudentsRequest().apply {
        minResult = 3.0
        expectedSemester = semester
    }
}
```

I needed to break some lines to fit the page. In real code, I would not do that.

Solution: orThrow

```
fun <T> T?.orThrow(lazyException: () -> Throwable): T =
    this ?: throw lazyException()
```

Solution: Logger

```
class PetStore(
    private val database: Database,
) {
    context(Logger)
    fun addPet(
        addPetRequest: AddPetRequest,
    ): Pet? {
        logInfo("Adding pet with name ${addPetRequest.name}")
        return try {
            database.addPet(addPetRequest)
              .also { logInfo("Added pet with id ${it.id}") }
        } catch (e: InsertionConflictException) {
            logWarning("There already is " +
                    "pet named ${addPetRequest.name}")
            null
        } catch (e: Exception) {
            logError("Failed to add " +
                    "pet with name ${addPetRequest.name}")
            null
        }
    }
}
```

I needed to break the function definition to fit the page. In actual code, I would not do that.

Solution: UserService

```
class UserService(
    private val userRepository: UserRepository,
    private val userDtoFactory: UserDtoFactory,
    private val tokenRepository: TokenRepository,
    private val logger: Logger,
) {
    private val userByIdCache: Cache<String, User> = cache {
        clearAfterWrite = 1.minutes
        clearAfterRead = 1.minutes
        load { id: String ->
            userRepository.getUser(id)
                ?.toUser()
                ?.also { userByKeyCache.store(it.key, it) }
        }
    }
    private val userByKeyCache: Cache<String, User> = cache {
        clearAfterWrite = 1.minutes
        clearAfterRead = 1.minutes
        load { key: String ->
            userRepository.getUserByKey(key)
                ?.toUser()
                ?.also { userByIdCache.store(it.id, it) }
        }
    }

    fun getUser(id: String): User? = userByIdCache.get(id)

    fun getUserByKey(key: String): User? = userByKeyCache.get(key)

    fun getToken(email: String, passwordHash: String): String =
        userRepository.getUserByEmail(email)
            ?.takeIf { it.passwordHash == passwordHash }
            ?.let { tokenRepository.createToken(it.id, it.isAdmin\
) }
            ?: error("Wrong email or password")

    fun updateUser(token: String, userPatch: UserPatch): User =
        tokenRepository.getUserId(token)
            ?.let { userRepository.getUser(it) }
            ?.let { userDto ->
                userDto.copy(
                    email = userPatch.email ?: userDto.email,
```

```
                    name = userPatch.name ?: userDto.name,
                    surname = userPatch.surname ?: userDto.surnam\
e,
                )
            }
            ?.also {
                userRepository.updateUser(it)
                userByIdCache.remove(it.id)
                userByKeyCache.remove(it.key)
                logger.log("User updated: $it")
            }
            ?.toUser()
            ?: error("User not found")

    fun addUser(token: String, addUser: AddUser): User {
        if (!tokenRepository.isAdmin(token)) {
            error("Only admin can add user")
        }
        return userDtoFactory.produceUserDto(addUser)
            .also(userRepository::addUser)
            .toUser()
            .also { logger.log("User added: $it") }
    }

    fun userStatistics(token: String): UserStatistics {
        if (!tokenRepository.isAdmin(token)) {
            error("Only admin can get statistics")
        }
        return UserStatistics(
            numberOfUsersCreatedEachDay = userRepository
                .getAllUsers()
                .groupingBy { it.creationTime.toLocalDate() }
                .eachCount()
        )
    }

    fun clearCache() {
        userByIdCache.clear()
        userByKeyCache.clear()
    }
}

class UserDtoFactory(
    private val timeProvider: TimeProvider,
```

```
    private val uuidGenerator: UuidGenerator,
    private val userKeyGenerator: UserKeyGenerator,
) {
    fun produceUserDto(addUser: AddUser): UserDto = UserDto(
        id = uuidGenerator.generate(),
        key = userKeyGenerator
            .findPublicKey(addUser.name, addUser.surname)
            ?: uuidGenerator.generate(),
        email = addUser.email,
        name = addUser.name,
        surname = addUser.surname,
        passwordHash = addUser.passwordHash,
        isAdmin = false,
        creationTime = timeProvider.now(),
    )
}

class RealUserKeyGenerator(
    private val userRepository: UserRepository,
): UserKeyGenerator {
    override fun findPublicKey(
        name: String,
        surname: String
    ): String? = sequenceOf(
        "$name$surname",
        "$surname$name",
        "$name${surname.first()}",
        "${name.first()}$surname",
        "$surname${name.first()}",
        "${surname.first()}$name",
    ).map(::properKey)
        .filter { it.length >= 4 }
        .find(userRepository::isAvailableKey)

    private fun properKey(original: String): String = original
        .replace("[^0-9a-zA-Z]".toRegex(), "")
        .lowercase()
        .trim()
}
```

Solution: Factorial sequence

```
val factorial: Sequence<BigInteger> = sequence {
    var i = BigInteger.ZERO
    var factorial = BigInteger.ONE
    while (true) {
        yield(factorial)
        i++
        factorial *= i
    }
}
```

Solution: Prime numbers sequence

```
val primes: Sequence<BigInteger> = sequence {
    var i = BigInteger("2")
    var primes = listOf(i)
    yield(i)
    while (true) {
        i++
        if (primes.none { i % it == BigInteger.ZERO }) {
            primes += i
            yield(i)
        }
    }
}
```

Solution: Callback function wrappers

```
class FetchTasksUseCase(
    private val callbackUseCase: FetchTasksCallbackUseCase
) {
    @Throws(ApiException::class)
    suspend fun fetchTasks(): List<Task> =
        suspendCancellableCoroutine { cont ->
            val cancellable = callbackUseCase.fetchTasks(
                onSuccess = { cont.resume(it) },
                onError = { cont.resumeWithException(it) }
            )
            cont.invokeOnCancellation { cancellable.cancel() }
        }
    suspend fun fetchTasksResult(): Result<List<Task>> =
        suspendCancellableCoroutine { cont ->
            val cancellable = callbackUseCase.fetchTasks(
                onSuccess = { cont.resume(Result.success(it)) },
                onError = { cont.resume(Result.failure(it)) }
            )
            cont.invokeOnCancellation { cancellable.cancel() }
        }
    suspend fun fetchTasksOrNull(): List<Task>? =
        suspendCancellableCoroutine { cont ->
            val cancellable = callbackUseCase.fetchTasks(
                onSuccess = { cont.resume(it) },
                onError = { cont.resume(null) }
            )
            cont.invokeOnCancellation { cancellable.cancel() }
        }
}
```

Solution: Continuation storage

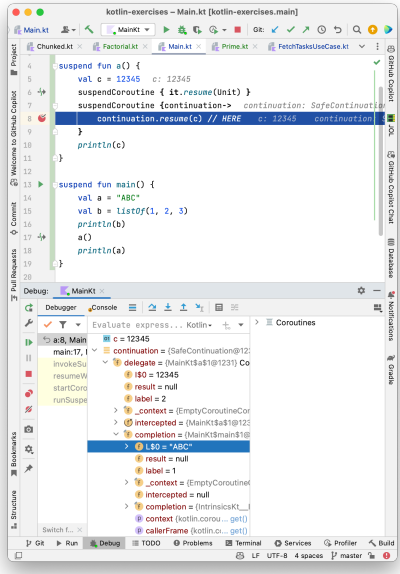
```
var continuation: Continuation<String>? = null

suspend fun continuationSteal(console: Console) {
    console.println("Before")
    val result = suspendCancellableCoroutine {
        continuation = it
    }
    console.println(result)
    console.println("After")
}
```

Solution: What is stored by a continuation?

A continuation must store all the coroutine's variables that are used after the suspension point. So, in this case it needs to store `c` and `a` (it does not need to store the value `b` because it is not used after the suspension). A continuation must also store the label that marks the point at which the coroutine should be resumed. In this case, this is the label 2 for function `a`, and 1 for `main`.

Here is a simplified continuation structure:

```
{
    I$0: 12345,
    label: 2,
    completion: {
        I$0: "ABC"
        label: 1,
        completion: ...
    }
}
```



Solution: UserDetailsRepository

This is a possible solution:

```
class UserDetailsRepository(
    private val client: UserDataClient,
    private val userDatabase: UserDetailsDatabase,
    private val backgroundScope: CoroutineScope,
) {
    suspend fun getUserDetails(): UserDetails =
        coroutineScope {
            val stored = userDatabase.load()
            if (stored != null) {
                return@coroutineScope stored
            }
            val name = async { client.getName() }
            val friends = async { client.getFriends() }
            val profile = async { client.getProfile() }
            val details = UserDetails(
                name = name.await(),
                friends = friends.await(),
                profile = profile.await(),
            )
            backgroundScope.launch { userDatabase.save(details) }
            details
        }
}
```

`coroutineScope` can either wrap whole function, or just `async` and `await`.

This problem can be also solved without the last `async` call. In such a case, the behavior of this function would remain the same, but I would consider it less readable.

```
class UserDetailsRepository(
    private val client: UserDataClient,
    private val userDatabase: UserDetailsDatabase,
    private val backgroundScope: CoroutineScope,
) {
    suspend fun getUserDetails(): UserDetails {
        val stored = userDatabase.load()
        if (stored != null) {
            return stored
        }
        val details = coroutineScope {
```

```
            val name = async { client.getName() }
            val friends = async { client.getFriends() }
            val profile = client.getProfile()
            UserDetails(
                name = name.await(),
                friends = friends.await(),
                profile = profile,
            )
        }
        backgroundScope.launch { userDatabase.save(details) }
        return details
    }
}
```

Solution: BestStudentUseCase

```
class BestStudentUseCase(
    private val repo: StudentsRepository
) {
    suspend fun getBestStudent(
        semester: String
    ): Student = coroutineScope {
        repo.getStudentIds(semester)
            .map { id -> async { repo.getStudent(id) } }
            .awaitAll()
            .maxByOrNull { it.result }
            ?: error("No students in semester $semester")
    }
}
```

It is a common mistake to wrap `getStudentIds` with an `async` call, and await it. The effective behavior of `getBestStudent` would be the same as without this `async` call, but this unnecessary wrapping would make the code less readable and less performant.

Another common mistake is to use `map { it.await() }` instead of `awaitAll()`. In both cases, the effective behavior of `getBestStudent` will be the same. Still, it is best practice to use `awaitAll()` because it is more readable and more performant, and it behaves more appropriately in some other functions if there is an exception in an async call.

Solution: CommentService

Simple solution:

```
class CommentService(
    private val commentRepository: CommentRepository,
    private val userService: UserService,
    private val commentFactory: CommentFactory
) {
    suspend fun addComment(
        token: String,
        collectionKey: String,
        body: AddComment
    ) {
        val userId = userService.readUserId(token)
        val commentDocument = commentFactory
            .toCommentDocument(userId, collectionKey, body)
        commentRepository.addComment(commentDocument)
    }

    suspend fun getComments(
        collectionKey: String
    ) = coroutineScope {
        val commentDocuments = commentRepository
            .getComments(collectionKey)
        CommentsCollection(
            collectionKey = collectionKey,
            elements = commentDocuments
                .map { async { makeCommentElement(it) } }
                .awaitAll()
        )
    }

    private suspend fun makeCommentElement(
        commentDocument: CommentDocument
    ) = CommentElement(
        id = commentDocument._id,
        collectionKey = commentDocument.collectionKey,
        user = userService.findUserById(commentDocument.userId),
        comment = commentDocument.comment,
        date = commentDocument.date,
    )
}
```

This is how this class should look if we want to make sure that `findUserById` is not called more than once for the same user id by the same `getComments` call:

```
class CommentService(
    private val commentRepository: CommentRepository,
    private val userService: UserService,
    private val commentFactory: CommentFactory
) {
    suspend fun addComment(
        token: String,
        collectionKey: String,
        body: AddComment
    ) {
        val userId = userService.readUserId(token)
        val commentDocument = commentFactory
            .toCommentDocument(userId, collectionKey, body)
        commentRepository.addComment(commentDocument)
    }

    suspend fun getComments(
        collectionKey: String
    ) = coroutineScope {
        val commentDocuments = commentRepository
            .getComments(collectionKey)
        val users: Map<String, User> = commentDocuments
            .map { it.userId }
            .toSet()
            .map { async { userService.findUserById(it) } }
            .awaitAll()
            .associateBy { it.id }

        CommentsCollection(
            collectionKey = collectionKey,
            elements = commentDocuments.map {
                val user = users[it.userId]
                makeCommentElement(it, user)
            }
        )
    }

    private fun makeCommentElement(
        commentDocument: CommentDocument,
        user: User?,
    ) = CommentElement(
```

```
        id = commentDocument._id,
        collectionKey = commentDocument.collectionKey,
        user = user,
        comment = commentDocument.comment,
        date = commentDocument.date,
    )
}
```

Solution: mapAsync

```
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> = coroutineScope {
    map { async { transformation(it) } }
        .awaitAll()
}
```

Solution: Understanding context propagation

```
suspend fun log(msg: String) {
    val name = coroutineContext[CoroutineName]?.name
    println("[$name] $msg")
}

fun CoroutineScope.slog(msg: String) {
    val name = coroutineContext[CoroutineName]?.name
    println("[$name] $msg")
}

suspend fun main() = withContext(CoroutineName("Outer")) {
    log("Starting") // [Outer] Starting
    launch(CoroutineName("Inner")) {
        slog("A") // [Inner] A
        launch {
            log("B") // [Inner] B
        }
    }
    launch {
        slog("C") // [Outer] C
    }
    GlobalScope.launch {
        log("D") // [null] D
    }
    slog("Ending") // [Outer] Ending
}
```

Solution: CounterContext

Here is a simple solution:

```
class CounterContext :
    AbstractCoroutineContextElement(CounterContext){

    private var counter = 0

    fun next(): Int = counter++

    companion object : CoroutineContext.Key<CounterContext>
}
```

Here is a thread-safe solution:

```
class CounterContext :
    AbstractCoroutineContextElement(CounterContext){

    private val counter = AtomicInteger(0)

    fun next(): Int = counter.getAndIncrement()

    companion object : CoroutineContext.Key<CounterContext>
}
```

Solution: Using dispatchers

- `createInvoice` does not need to set a dispatcher because `postInvoice` is a suspending function that does not block threads.
- `sendEmail` needs to set the dispatcher to `Dispatchers.IO` or a custom dispatcher with a thread limit because `api` is a blocking function.
- `getUserOrders` does not need to set a dispatcher because `toList` is a suspending function that does not block threads.
- `upscaleImage` needs to set the dispatcher to `Dispatchers.Default` or `Dispatchers.Default` limited to a certain number of threads because upscaling an image is a CPU-intensive operation.

Solution: DiscNewsRepository

You need to wrap the blocking function with `withContext`, which uses the appropriate dispatcher. In other cases, this dispatcher might have been `Dispatchers.IO`, but in this case we expect this function to be used intensively. You want to support 200 parallel reads, so you need to use a dispatcher with a custom thread limit. The best choice is to use `Dispatchers.IO` with the `limitedParallelism` parameter set to 200.

```
class DiscNewsRepository(
    private val discReader: DiscReader
) : NewsRepository {
    private val dispatcher = Dispatchers.IO
        .limitedParallelism(200)

    override suspend fun getNews(
        newsId: String
    ): News = withContext(dispatcher) {
        val (title, content) = discReader.read("user/$newsId")
        News(title, content)
    }
}
```

Solution: Experiments with dispatchers

On my machine (MacBook M2 Pro 2022), the results are as follows:

Dispatcher	CPU-intensive	Blocking	Suspending
1 thread	21 794 ms	100 450 ms	1 030 ms
Dispatchers.Default	3 066 ms	10 058 ms	1 024 ms
Dispatchers.IO	2 973 ms	2 025 ms	1 025 ms
100 threads	3 038 ms	1 023 ms	1 024 ms

Let's discuss these results. Starting from CPU-intensive, it should be clear that 1 thread is the worst choice because it does not use the power of all 10 cores in my machine. All the other dispatchers tested here have similar execution times. In theory, `Dispatchers.Default` should be fastest because it avoids the costs of additional threads; however, when I was running this experiment, there were many other processes running on my machine (IntelliJ, Chrome), and dispatchers with more threads were able to access the CPU more frequently.

For blocking operations, the more threads a dispatcher has, the less time it takes to execute. The formula is linear: the number of coroutines divided by the number of threads, rounded up. However, you need to remember that the more threads you have, the more memory you use, and the more context switches you have; so, you should not use too many threads.

Regarding suspending functions, the dispatcher does not matter because suspending functions do not block threads. So, if we only suspend and do nothing blocking or CPU-intensive, then the dispatcher does not matter.

Solution: Correct mistakes with cancellation

There are two changes that need to be made:

- We should not block suspending function unless we set a dispatcher that can be used for blocking operations. This is because suspending functions should not be blocking (they should be dispatcher-agnostic). To correct it, we should wrap the whole function with `withContext` with a dispatcher that can be used for blocking operations, like `Dispatchers.IO`.
- We should use `yield` between two blocking operations, to allow cancellation and redispatching between them.
- We should wrap `revertUnfinishedTransactions` with the block `withContext(NonCancellable)` so it (a suspending function) can be executed (otherwise a suspension would throw an exception in cancelled coroutine).
- We should rethrow the `CancellationException`, because the functions that call `updateUser` might also want to specify what to do when the operation is cancelled.

Notice, that using `async` here is not useful, because all operations must be called sequentially.

```
suspend fun updateUser() = withContext(Dispatchers.IO) {
    val user = readUser() // blocking
    yield()
    val settings = readUserSettings(user.id) // blocking

    try {
        updateUserInDatabase(user, settings) // suspending
    } catch (e: CancellationException) {
        withContext(NonCancellable) {
            revertUnfinishedTransactions() // suspending
        }
        throw e
    }
}
```

There are three changes that need to be made:

- We should use a dispatcher that can be used for blocking operations, like `Dispatchers.IO`, because we have blocking operations `readText` and `delete`.
- We should make sure we rethrow the `CancellationException`.
- We should use `yield` between a blocking and CPU operations, to allow cancellation between them (though this is not such important, as `calculateSignature` is likely lightweight, and there is a suspending call straight after it).

Notice, that using `async` here is not useful, because all operations must be called sequentially.

```
suspend fun sendSignature(file: File) =
    withContext(Dispatchers.IO) {
        try {
            val content = file.readText()
            yield()
            val signature = calculateSignature(content)
            sendSignature(signature) // suspending
        } catch (e: CancellationException) {
            throw e
        } catch (e: Exception) {
            println("Error while sending signature: ${e.message}")
            e.printStackTrace()
        } finally {
            file.delete()
        }
    }
```

There is one change that needs to be made:

- We should rethrow the `CancellationException`.

```
suspend fun trySendUntilSuccess() {
    var success = false
    do {
        try {
            send()
            success = true
        } catch (e: CancellationException) {
            throw e
        } catch (e: Exception) {
            println("Error while sending: ${e.message}")
            e.printStackTrace()
        }
    } while (!success)
}
```

That example is especially interesting, because if we do not rethrow the `CancellationException`, in case of cancellation, this function will be in an infinite loop. You can use the following code to see it yourself:

```
fun main() = runBlocking {
    val job = launch { trySendUntilSuccess() }
    delay(100)
    job.cancelAndJoin()
}

suspend fun send() {
    println("Sending...")
    delay(1000)
}
```

Solution: NotificationSender

```
class NotificationSender(
    private val client: NotificationClient,
    private val exceptionCollector: ExceptionCollector,
    dispatcher: CoroutineDispatcher,
) {
    private val exceptionHandler =
        CoroutineExceptionHandler { _, throwable ->
            exceptionCollector.collectException(throwable)
        }
    val scope: CoroutineScope = CoroutineScope(
        SupervisorJob() + dispatcher + exceptionHandler
    )

    fun sendNotifications(notifications: List<Notification>) {
        notifications.forEach { notification ->
            scope.launch {
                client.send(notification)
            }
        }
    }

    fun cancel() {
        scope.coroutineContext.cancelChildren()
    }
}
```

Solution: BaseViewModel

```
abstract class BaseViewModel : ViewModel() {
    private val _exceptions = Channel<Throwable>(Channel.UNLIMITE\
D)
    val exceptions: Flow<Throwable> = _exceptions.receiveAsFlow()

    private val exceptionHandler =
        CoroutineExceptionHandler { _, throwable ->
            _exceptions.trySendBlocking(throwable)
        }
    val scope: CoroutineScope = CoroutineScope(
        SupervisorJob() +
                Dispatchers.Main.immediate +
                exceptionHandler
    )

    override fun onCleared() {
        scope.coroutineContext.cancelChildren()
    }
}
```

Solution: UserDownloader

This is the solution using a dispatcher limited to a single thread and a read-only list:

```
class UserDownloader(private val api: NetworkService) {
    private var users = listOf<User>()
    private val dispatcher = Dispatchers.IO
        .limitedParallelism(1)

    fun downloaded(): List<User> = users

    suspend fun getUser(id: Int) = withContext(dispatcher) {
        val newUser = api.getUser(id)
        users += newUser
    }
}
```

This is the solution using a dispatcher limited to a single thread and a mutable list:

```
class UserDownloader(private val api: NetworkService) {
    private val users = mutableListOf<User>()
    private val dispatcher = Dispatchers.IO
        .limitedParallelism(1)

    suspend fun downloaded(): List<User>=withContext(dispatcher){
        users.toList()
    }

    suspend fun getUser(id: Int) = withContext(dispatcher) {
        val newUser = api.getUser(id)
        users += newUser
    }
}
```

This is the solution using a synchronized block and a mutable list:

```
class UserDownloader(private val api: NetworkService) {
    private val users = mutableListOf<User>()
    private val lock = Any()

    suspend fun downloaded(): List<User> = synchronized(lock) {
        users.toList()
    }

    suspend fun getUser(id: Int) {
        val newUser = api.getUser(id)
        synchronized(lock) {
            users += newUser
        }
    }
}
```

This is the solution using a concurrent set:

```
class UserDownloader(private val api: NetworkService) {
    private val users = ConcurrentHashMap.newKeySet<User>()

    fun downloaded(): List<User> = users.toList()

    suspend fun getUser(id: Int) {
        val newUser = api.getUser(id)
        users += newUser
    }
}
```

Solution: CompanyDetailsRepository

The solution using `synchronized` block and a mutable map:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher
) {
    private val details = mutableMapOf<Company, CompanyDetails>()
    private val lock = Any()

    suspend fun getDetails(company: Company): CompanyDetails {
        val current = getDetailsOrNull(company)
        if (current == null) {
            val companyDetails = client.fetchDetails(company)
              synchronized(lock) {
                  details[company] = companyDetails
              }
            return companyDetails
        }
        return current
    }

    fun getDetailsOrNull(company: Company): CompanyDetails? =
        synchronized(lock) {
            details[company]
        }

    fun getReadyDetails(): Map<Company, CompanyDetails> =
        synchronized(lock) {
            details.toMap()
        }

    fun clear() = synchronized(lock) {
        details.clear()
    }
}
```

The solution using `synchronized` block and a read-only map:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher
) {
    private var details = mapOf<Company, CompanyDetails>()
    private val lock = Any()

    suspend fun getDetails(company: Company): CompanyDetails {
        val current = getDetailsOrNull(company)
        if (current == null) {
            val companyDetails = client.fetchDetails(company)
            synchronized(lock) {
                details = details + (company to companyDetails)
            }
            return companyDetails
        }
        return current
    }

    // Access to value is atomic so we don't need to synchronize
    fun getDetailsOrNull(company: Company): CompanyDetails? =
        details[company]

    // Access to value is atomic so we don't need to synchronize
    fun getReadyDetails(): Map<Company, CompanyDetails> = details

    // Setting value is atomic so we don't need to synchronize
    fun clear() {
        details = emptyMap()
    }
}
```

The solution using a dispatcher limited to a single thread and a mutable map:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher,
) {
    private val details = mutableMapOf<Company, CompanyDetails>()
    private val dispatcher = dispatcher.limitedParallelism(1)

    suspend fun getDetails(company: Company): CompanyDetails =
        withContext(dispatcher) {
            val current = getDetailsOrNull(company)
```

```
            if (current == null) {
                val companyDetails = client.fetchDetails(company)
                details[company] = companyDetails
                companyDetails
            } else {
                current
            }
        }

    suspend fun getDetailsOrNull(
        company: Company
    ): CompanyDetails? = withContext(dispatcher) {
        details[company]
    }

    suspend fun getReadyDetails(): Map<Company, CompanyDetails> =
        withContext(dispatcher) {
            details.toMap()
        }

    suspend fun clear() = withContext(dispatcher) {
        details.clear()
    }
}
```

The solution using a dispatcher limited to a single thread and a read-only map:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher,
) {
    private var details = mapOf<Company, CompanyDetails>()
    private val dispatcher = dispatcher.limitedParallelism(1)

    suspend fun getDetails(company: Company): CompanyDetails =
        withContext(dispatcher) {
            val current = details[company]
            if (current == null) {
                val companyDetails = client.fetchDetails(company)
                details += (company to companyDetails)
                companyDetails
            } else {
                current
            }
```

```
        }

    fun getDetailsOrNull(company: Company): CompanyDetails? =
        details[company]

    fun getReadyDetails(): Map<Company, CompanyDetails> = details

    fun clear() {
        details = emptyMap()
    }
}
```

The solution using `AtomicReference` and read-only map:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher
) {
    private var details =
        AtomicReference<Map<Company, CompanyDetails>>(emptyMap())

    suspend fun getDetails(company: Company): CompanyDetails {
        val current = getDetailsOrNull(company)
        if (current == null) {
            val companyDetails = client.fetchDetails(company)
            details.updateAndGet {
                it + (company to companyDetails)
            }
            return companyDetails
        }
        return current
    }

    fun getDetailsOrNull(company: Company): CompanyDetails? =
        details.get()[company]

    fun getReadyDetails(): Map<Company, CompanyDetails> =
        details.get()

    fun clear() {
        details.set(emptyMap())
    }
}
```

This solution is not efficient, because for in case of conflict, `updateAndGet` will recalculate its update. Since adding an element to a read-only collection is heavy, that is not a good solution.

The solution using a concurrent collection from `java.util.concurrent` package:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher,
) {
    private var details =
        ConcurrentHashMap<Company, CompanyDetails>()

    suspend fun getDetails(company: Company): CompanyDetails {
        val current = details[company]
        if (current == null) {
            val companyDetails = client.fetchDetails(company)
            details[company] = companyDetails
            return companyDetails
        }
        return current
    }

    fun getDetailsOrNull(company: Company): CompanyDetails? =
        details[company]

    fun getReadyDetails(): Map<Company, CompanyDetails> =
        details.toMap()

    fun clear() {
        details.clear()
    }
}
```

The solution using a mutex and a mutable map:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher
) {
    private val details = mutableMapOf<Company, CompanyDetails>()
    private val mutex = Mutex()

    suspend fun getDetails(company: Company): CompanyDetails {
        val current = getDetailsOrNull(company)
        if (current == null) {
            val companyDetails = client.fetchDetails(company)
            mutex.withLock {
                details[company] = companyDetails
            }
            return companyDetails
        }
        return current
    }

    suspend fun getDetailsOrNull(
        company: Company
    ): CompanyDetails? = mutex.withLock {
        details[company]
    }

    suspend fun getReadyDetails(): Map<Company, CompanyDetails> =
        mutex.withLock {
            details.toMap()
        }

    suspend fun clear() = mutex.withLock {
        details.clear()
    }
}
```

The solution using concurrent collection with suspending lazy objects (`suspendLazy` implementation from the exercise *Suspended lazy*):

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher,
) {
    private var details =
        ConcurrentHashMap<Company, SuspendLazy<CompanyDetails>>()

    suspend fun getDetails(company: Company): CompanyDetails {
        details.computeIfAbsent(company) {
            suspendLazy { client.fetchDetails(company) }
        }
        return details[company]!!.invoke()
    }

    fun getDetailsOrNull(company: Company): CompanyDetails? =
        details[company]?.valueOrNull()

    fun getReadyDetails(): Map<Company, CompanyDetails> = details
        .toMap()
        .mapNotNull { (k, v) -> v.valueOrNull()?.let { k to it } }
        .toMap()

    fun clear() {
        details.clear()
    }
}
```

The solution using caching library to cache details:

```
class CompanyDetailsRepository(
    private val client: CompanyDetailsClient,
    dispatcher: CoroutineDispatcher,
) {
    private var cache = cacheBuilder<Company, CompanyDetails> {
        this.dispatcher = dispatcher
    }.build()

    suspend fun getDetails(company: Company): CompanyDetails =
        cache.get(company) { client.fetchDetails(company) }

    fun getDetailsOrNull(company: Company): CompanyDetails? =
        cache.getOrNull(company)

    fun getReadyDetails(): Map<Company, CompanyDetails> =
```

```
        cache.asDeferredMap()
            .filter { it.value.isCompleted }
            .mapValues { it.value.getCompleted() }

    fun clear() {
        cache.invalidateAll()
    }
}
```

Here is how much time different operations took on my machine (MacBook M2 Pro 2022, time in ms or 10,000 calls, concurrently on 10,000 coroutines). Only consider the order of magnitude, not the exact values due to error margins.

	getDetail	getDetailOrNull	getReadyDetails
Synchronization block, mutable map	6	2	4417
Synchronization block, read-only map	2	1	1
Mutex, mutable map	11	3	4193
Mutex, read-only map	2	1	1
Single thread dispatcher, mutable map	19	5	7925
Single thread dispatcher, read-only map	12	1	1
ConcurrentHashMap	2	1	1260
Suspended lazy map	2	2	3159
Cache	2	1	4970

The solution that is both efficient, simple, and works best is using caching library. It can also be configured for more advanced use cases.

Regarding simpler solutions, concurrent collection, if it is enough for your use case, is the best choice. It is simple and efficient.

Using a dispatcher is the easiest solution, but it requires making more functions suspending. Synchronized block or mutex requires fine-grained control over the shared state, but it is efficient and helps achieve simpler API.

Using mutable collection is better when you more ofen need to modify the collection than making its copy, what is the most common case.

Solution: CancellingRefresher

This is the solution using `synchronized` block:

```
class CancellingRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend () -> Unit,
) {
    private var refreshJob: Job? = null
    private val refreshLock = Any()

    fun refresh() = synchronized(refreshLock) {
        refreshJob?.cancel()
        refreshJob = scope.launch {
            refreshData()
        }
    }
}
```

This is the solution using `Mutex`:

```
class CancellingRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend () -> Unit,
) {
    private var refreshJob: Job? = null
    private val refreshMutex = Mutex()

    suspend fun refresh() = refreshMutex.withLock {
        refreshJob?.cancel()
        refreshJob = scope.launch {
            refreshData()
        }
    }
}
```

This is the solution using a dispatcher limited to a single thread:

```
class CancellingRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend () -> Unit,
) {
    private var refreshJob: Job? = null
    private val dispatcher = Dispatchers.IO.limitedParallelism(1)

    suspend fun refresh() = withContext(dispatcher) {
        refreshJob?.cancel()
        refreshJob = scope.launch {
            refreshData()
        }
    }
}
```

This problem cannot be solved using a concurrent set of jobs, because cancellation, cleaning and adding elements cannot be one operation, so we would have race conditions between them.

```
// INCORRECT SOLUTION!
class CancellingRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend () -> Unit,
) {
    private var refreshJobs = ConcurrentHashMap.newKeySet<Job>()

    fun refresh() {
        refreshJobs.forEach { it.cancel() }
        refreshJobs.clear()
        refreshJobs += scope.launch {
            refreshData()
        }
    }
}
```

Solution: TokenRepository

```
class TokenRepository(
    private val client: TokenClient,
    private val timeProvider: TimeProvider
) {
    private var token: Token? = null
    private val mutex = Mutex()

    suspend fun getToken(): Token = mutex.withLock {
        val currentToken = token
        if (currentToken != null &&
            currentToken.expiration > timeProvider.now()) {
            return currentToken
        }
        val newToken = client.fetchToken()
        token = newToken
        return newToken
    }

    fun invalidateToken() {
        token = null
    }
}
```

Here we must use `Mutex` in `getToken` to make sure we make only one call to `fetchToken` at a time. We must not use this mutex to synchronize `invalidateToken`, because then cleaning the token would wait for the token to be fetched, and it would invalidate this new token.

We cannot use `synchronized` block, because we need to suspend instead of blocking the thread. We would use it if `fetchToken` would be a blocking operation, and we wanted `getToken` to be a blocking operation as well.

We cannot use atomic reference or a dispatcher limited to a single thread, because we need to make sure that `fetchToken` is called only once at a time.

Solution: Suspended lazy

```
fun <T> suspendLazy(initializer: suspend () -> T): SuspendLazy<T>{
    var innerInitializer: (suspend () -> T)? = initializer
    val mutex = Mutex()
    var holder: Any? = Any()

    return object : SuspendLazy<T> {
        override val isInitialized: Boolean
            get() = innerInitializer == null

        override fun valueOrNull(): T? =
            if (isInitialized) holder as T else null

        @Suppress("UNCHECKED_CAST")
        override suspend fun invoke(): T =
            if (isInitialized) holder as T
            else mutex.withLock {
                innerInitializer?.let {
                    holder = it()
                    innerInitializer = null
                }
                holder as T
            }
    }
}
```

Solution: mapAsync with concurrency limit

```
suspend fun <T, R> Iterable<T>.mapAsync(
    concurrency: Int,
    transformation: suspend (T) -> R
): List<R> = coroutineScope {
    val semaphore = Semaphore(concurrency)
    map {
        async {
            semaphore.withPermit {
                transformation(it)
            }
        }
    }.awaitAll()
}
```

Solution: Test UserDetailsRepository

This is a possible solution:

```
@Test
fun `should fetch details asynchronously`() = runTest {
    // given
    val client = object : UserDataClient {
        override suspend fun getName(): String {
            delay(100)
            return "Ben"
        }

        override suspend fun getFriends(): List<Friend> {
            delay(200)
            return listOf(Friend("friend-id-1"))
        }

        override suspend fun getProfile(): Profile {
            delay(300)
            return Profile("Example description")
        }
    }
    var savedDetails: UserDetails? = null
    val database = object : UserDetailsDatabase {
        override suspend fun load(): UserDetails? {
            delay(500)
            return savedDetails
        }

        override suspend fun save(user: UserDetails) {
            delay(400)
            savedDetails = user
        }
    }

    val repo = UserDetailsRepository(
        client = client,
        userDatabase = database,
        backgroundScope = backgroundScope
    )

    // when
    val details = repo.getUserDetails()
```

```
    // then data are fetched asynchronously
    assertEquals("Ben", details.name)
    assertEquals(listOf(Friend("friend-id-1")), details.friends)
    assertEquals(Profile("Example description"), details.profile)
    assertEquals(350, currentTime) // max(100, 200, 300) + 500
    assertEquals(null, savedDetails)

    // when all children are finished
    backgroundScope.coroutineContext.job.children
        .forEach { it.join() }

    // then data are saved to the database
    assertEquals(750, currentTime) // prev + 400
    assertEquals(details, savedDetails)

    // when getting details again
    val details2 = repo.getUserDetails()

    // then data are loaded from the database
    assertEquals(details, details2)
    assertEquals(1250, currentTime) // prev + 500
}
```

Notice that we assert that:

- the name, friends, and profile are fetched asynchronously by delaying the response, and verify that the overall time is the maximum of the delays,
- the fetched details are returned by asserting the result,
- the details are saved to the database asynchronously by delaying this operation, verifying that after the `getUserDetails` call, the details are not saved yet, and then awaiting the background scope to finish the operation, and checking that the details are saved, and the time is greater by the save delay,
- the details are loaded from the database if they are already there by delaying the load operation and verifying that the details are returned, and that it took only the load delay time.

It would be a better practice to divide this complex test into smaller ones, but the task was to test multiple behaviors in one test.

Solution: Testing mapAsync

```
class MapAsyncTest {
    @Test
    fun `should behave like a regular map`() = runTest {
        val list = ('a'..'z').toList()
        assertEquals(
            list.map { c -> c.inc() },
            list.mapAsync { c -> c.inc() }
        )
        assertEquals(
            list.map { c -> c.code },
            list.mapAsync { c -> c.code }
        )
        assertEquals(
            list.map { c -> c.uppercaseChar() },
            list.mapAsync { c -> c.uppercaseChar() }
        )

        val set = (1..10).toSet()
        assertEquals(
            set.map { i -> i * i },
            set.mapAsync { i -> i * i }
        )
        assertEquals(
            set.map { i -> "A$i" },
            set.mapAsync { i -> "A$i" }
        )
        assertEquals(
            set.map { i -> i.toChar() },
            set.mapAsync { i -> i.toChar() }
        )
    }

    @Test
    fun `should map async`() = runTest {
        val transforms: List<suspend () -> String> = listOf(
            { delay(3000); "A" },
            { delay(2000); "B" },
            { delay(4000); "C" },
            { delay(1000); "D" },
        )

        val res = transforms.mapAsync { it() }
```

```
    assertEquals(4000, currentTime)
}

@Test
fun `should keep elements order`() = runTest {
    val transforms: List<suspend () -> String> = listOf(
        { delay(3000); "A" },
        { delay(2000); "B" },
        { delay(4000); "C" },
        { delay(1000); "D" },
    )

    val res = transforms.mapAsync { it() }
    assertEquals(listOf("A", "B", "C", "D"), res)
}

@Test
fun `should support context propagation`() = runTest {
    var ctx: CoroutineContext? = null

    val name1 = CoroutineName("Name 1")
    withContext(name1) {
        listOf("A").mapAsync {
            ctx = currentCoroutineContext()
            it
        }
    }
    assertEquals(name1, ctx?.get(CoroutineName))

    val name2 = CoroutineName("Some name 2")
    withContext(name2) {
        listOf("B").mapAsync {
            ctx = currentCoroutineContext()
            it
        }
    }
    assertEquals(name2, ctx?.get(CoroutineName))
}

@Test
fun `should support cancellation`() = runTest {
    var job: Job? = null

    val parentJob = launch {
```

```
            listOf("A").mapAsync {
                job = currentCoroutineContext().job
                delay(Long.MAX_VALUE)
            }
        }

        delay(1000)
        parentJob.cancel()
        assertEquals(true, job?.isCancelled)
    }

    @Test
    fun `should propagate exceptions`() = runTest {
        // given
        val e = object : Throwable() {}
        val bodies = listOf(
            suspend { "A" },
            suspend { delay(1000); "B" },
            suspend { delay(500); throw e },
            suspend { "C" }
        )

        // when
        val result = runCatching { bodies.mapAsync { it() } }

        // then
        assertTrue(result.isFailure)
        assertEquals(e, result.exceptionOrNull())
        assertEquals(500, currentTime)
    }
}
```

Solution: Testing the NotificationSender class

```
class NotificationSenderTest {

    @Test
    fun `should send notifications concurrently`() {
        val fakeNotificationsClient = FakeNotificationClient(
            delayTime = 200
        )
        val fakeExceptionCollector = FakeExceptionCollector()
        val testDispatcher = StandardTestDispatcher()
        val sender = NotificationSender(
            client = fakeNotificationsClient,
            exceptionCollector = fakeExceptionCollector,
            dispatcher = testDispatcher,
        )
        val notifications = List(20) { Notification("ID$it") }

        // when
        sender.sendNotifications(notifications)
        testDispatcher.scheduler.advanceUntilIdle()

        // then
        assertEquals(notifications, fakeNotificationsClient.sent)
        assertEquals(
            200,
            testDispatcher.scheduler.currentTime,
            "Notifications should be sent concurrently"
        )
    }

    @Test
    fun `should cancel all coroutines when cancel is called`() {
        val fakeNotificationsClient = FakeNotificationClient(
            delayTime = 1000
        )
        val fakeExceptionCollector = FakeExceptionCollector()
        val testDispatcher = StandardTestDispatcher()
        val sender = NotificationSender(
            client = fakeNotificationsClient,
            exceptionCollector = fakeExceptionCollector,
            dispatcher = testDispatcher,
        )
        val notifications = List(20) { Notification("ID$it") }
```

```
    // when
    sender.sendNotifications(notifications)
    testDispatcher.scheduler.advanceTimeBy(500)
    sender.cancel()

    // then
    val children = sender.scope.coroutineContext.job.children
    assert(children.all { it.isCancelled })

    // and scope should still be active
    assert(sender.scope.isActive)
}

@Test
fun `should not cancel other sending processes`() {
    val fakeNotificationsClient = FakeNotificationClient(
        delayTime = 100,
        failEvery = 10
    )
    val fakeExceptionCollector = FakeExceptionCollector()
    val testDispatcher = StandardTestDispatcher()
    val sender = NotificationSender(
        client = fakeNotificationsClient,
        exceptionCollector = fakeExceptionCollector,
        dispatcher = testDispatcher,
    )
    val notifications = List(100) { Notification("ID$it") }

    // when
    sender.sendNotifications(notifications)
    testDispatcher.scheduler.advanceUntilIdle()

    // then
    assertEquals(90, fakeNotificationsClient.sent.size)
}

@Test
fun `should collect exceptions from all coroutines`() {
    val fakeNotificationsClient = FakeNotificationClient(
        delayTime = 100,
        failEvery = 10
    )
    val fakeExceptionCollector = FakeExceptionCollector()
```

```
        val testDispatcher = StandardTestDispatcher()
        val sender = NotificationSender(
            client = fakeNotificationsClient,
            exceptionCollector = fakeExceptionCollector,
            dispatcher = testDispatcher,
        )
        val notifications = List(100) { Notification("ID$it") }

        // when
        sender.sendNotifications(notifications)
        testDispatcher.scheduler.advanceUntilIdle()

        // then
        assertEquals(10, fakeExceptionCollector.collected.size)
    }
}
```

Solution: UserRefresher

This is the solution using a `Channel`:

```
class UserRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend (Int) -> Unit,
) {
    private val queue = Channel<Int>(Channel.UNLIMITED)

    init {
        scope.launch {
            for (userId in queue) {
                refreshData(userId)
            }
        }
    }

    suspend fun refresh(userId: Int) {
        queue.send(userId)
    }
}
```

This is the solution using `Mutex`:

```
class UserRefresher(
    private val scope: CoroutineScope,
    private val refreshData: suspend (Int) -> Unit,
) {
    private val mutex = Mutex()

    fun refresh(userId: Int) {
        scope.launch {
            mutex.withLock {
                refreshData(userId)
            }
        }
    }
}
```

Solution: Cafeteria simulation

```
suspend fun main() = coroutineScope {
    val orders = Channel<CoffeeType>(Channel.UNLIMITED)
    val coffees = Channel<Coffee>(Channel.UNLIMITED)

    serveOrders(orders, coffees, "Alice")
    serveOrders(orders, coffees, "Bob")
    serveOrders(orders, coffees, "Celine")
    serveOrders(orders, coffees, "Dave")

    launch {
        for (coffee in coffees) {
            println("Coffee $coffee is ready")
        }
    }

    println("Welcome to Dream Coffee!")
    println("Press E to get espresso, L to get latte.")
    while (true) {
        val type = when (readlnOrNull()) {
            "E" -> CoffeeType.ESPRESSO
            "L" -> CoffeeType.LATTE
            else -> continue
        }
        orders.send(type)
        println("Order for $type sent")
    }
}

fun CoroutineScope.serveOrders(
    orders: ReceiveChannel<CoffeeType>,
    servedOrders: SendChannel<Coffee>,
    baristaName: String
) = launch {
    for (order in orders) {
        val coffee = makeCoffee(order, baristaName)
        servedOrders.send(coffee)
    }
}
```

Solution: raceOf

```
suspend fun <T> raceOf(
    racer: suspend CoroutineScope.() -> T,
    vararg racers: suspend CoroutineScope.() -> T
): T = coroutineScope {
    select {
        (listOf(racer) + racers).forEach { racer ->
            async { racer() }.onAwait {
                coroutineContext.job.cancelChildren()
                it
            }
        }
    }
}
```

Solution: Flow utils

```
val infiniteFlow = flow {
    while (true) {
        emit(Unit)
    }
}

val neverFlow = flow<Nothing> {
    suspendCancellableCoroutine { }
}

fun everyFlow(timeMillis: Long) = flow {
    while (true) {
        delay(timeMillis)
        emit(Unit)
    }
}

fun <T> flowOf(lambda: suspend () -> T) = flow {
    emit(lambda())
}

fun <T> flowOfFlatten(lambda: suspend () -> Flow<T>) = flow {
    val flow = lambda()
    flow.collect { emit(it) }
}
```

Solution: All users flow

```
class AllUsers(private val repository: UserRepository) {
    fun getAllUsers(): Flow<User> = flow {
        var pageNumber = 0
        do {
            val users = repository.fetchUsers(pageNumber)
            users.forEach { emit(it) }
            pageNumber++
        } while (users.isNotEmpty())
    }
}
```

Solution: distinct

```
fun <T> Flow<T>.distinct(): Flow<T> = flow {
    val seen = mutableSetOf<T>()
    collect {
        if (seen.add(it)) emit(it)
    }
}
```

Notice that already emitted elements must be kept inside the flow, not in the `distinct` function local variable. This is a common but incorrect implementation:

```
fun <T> Flow<T>.distinct(): Flow<T> {
    val set = mutableSetOf<T>()
    return filter { set.add(it) }
}
```

The problem with this implementation is that it will not work correctly when the same flow is used more than once. It will work only for the first collection.

```
val f = flowOf(1, 1, 3, 1, 2, 3, 1, 2, 3, 1)
    .distinct()

println(f.toList()) // [1, 3, 2]
println(f.toList()) // should be [1, 3, 2], but is []
```

Solution: TemperatureService

Here is a simple solution:

```
class TemperatureService(
    private val temperatureDataSource: TemperatureDataSource,
    backgroundScope: CoroutineScope,
) {
    private val lastKnownTemperature =
        ConcurrentHashMap<String, Fahrenheit>()

    fun observeTemperature(city: String): Flow<Fahrenheit> =
        temperatureDataSource.observeTemperatureUpdates()
            .filter { it.city == city }
            .map { celsiusToFahrenheit(it.temperature) }
            .onEach { lastKnownTemperature[city] = it }
            .onStart { lastKnownTemperature[city]?.let { emit(it)\
 } }

    fun getLastKnown(city: String): Fahrenheit? =
        lastKnownTemperature[city]

    fun getAllLastKnown(): Map<String, Fahrenheit> =
        lastKnownTemperature.toMap()

    private fun celsiusToFahrenheit(celsius: Double) =
        Fahrenheit(celsius * 9 / 5 + 32)
}
```

However, this solution has one efficiency issue: every new observer of `observeTemperature` will start a new flow of product updates. Also, observed temperatures will now be stored only if there is an observer of this temperature. We can overcome both those limitations using `SharedFlow`, which will be introduced later.

Solution: NewsViewModel

```
class NewsViewModel(
    newsRepository: NewsRepository,
) : BaseViewModel() {
    private val _progressVisible = MutableStateFlow(false)
    val progressVisible = _progressVisible.asStateFlow()

    private val _newsToShow = MutableStateFlow(emptyList<News>())
    val newsToShow = _newsToShow.asStateFlow()

    private val _errors = Channel<Throwable>(Channel.UNLIMITED)
    val errors = _errors.receiveAsFlow()

    init {
        newsRepository.fetchNews()
            .retry { error -> error is ApiException }
            .catch { error -> _errors.send(error) }
            .onStart { _progressVisible.value = true }
            .onCompletion { _progressVisible.value = false }
            .onEach { news -> _newsToShow.update { it + news } }
            .launchIn(viewModelScope)
    }
}
```

Solution: ProductService

This is a simple solution:

```
class ProductService(
    private val productRepository: ProductRepository,
    backgroundScope: CoroutineScope,
) {
    private val activeObservers = AtomicInteger(0)

    fun observeProducts(categories: Set<String>): Flow<Product> =
        productRepository
            .observeProductUpdates()
            .distinctUntilChanged()
            .flatMapMerge(concurrency = Int.MAX_VALUE) {
                flow { emit(productRepository.fetchProduct(it)) }
            }
            .filter { it.category in categories }
            .onStart { activeObservers.incrementAndGet() }
            .onCompletion { activeObservers.decrementAndGet() }

    fun activeObserversCount(): Int = activeObservers.get()
}
```

However, this solution has one efficiency issue: every new observer of `observeProducts` will start a new flow of product updates. This can can be optimized using `SharedFlow`, which will be introduced later.

Solution: Flow Kata

```
fun producingUnits(num: Int): Flow<Unit> =
    flow { repeat(num) { emit(Unit) } }
// or
//    List(num) {}.asFlow()
// or
//    (1..num).map { }.asFlow()

fun <T> Flow<T>.delayEach(timeMillis: Long): Flow<T> =
    onEach { delay(timeMillis) }
// or
//    flow { collect { delay(timeMillis); emit(it) } }

fun <T, R> Flow<T>.mapIndexed(
    transformation: suspend (index: Int, T) -> R
): Flow<R> =
    withIndex().map { transformation(it.index, it.value) }
// or
//    flow {
//        var index = 0
//        collect {
//            emit(transformation(index, it))
//            index++
//        }
//    }
// or
//    flow {
//        collectIndexed { index, value ->
//            emit(transformation(index, value))
//        }
//    }

fun Flow<*>.toNextNumbers(): Flow<Int> =
    mapIndexed { index, _ -> index + 1 }
// or
//    withIndex().map { it.index + 1 }
// or
//    scan(0) { acc, _ -> acc + 1 }.drop(1)
// or
//    flow {
//        var value = 1
//        collect {
//            emit(value++)
```

```
//        }
//    }
// or
//    flow {
//        collectIndexed { index, _ ->
//            emit(index + 1)
//        }
//    }

fun <T> Flow<T>.withHistory(): Flow<List<T>> =
    scan(emptyList()) { acc, value -> acc + value }
// or
//    flow {
//        var history = listOf<T>()
//        emit(history)
//        collect {
//            history += it
//            emit(history)
//        }
//    }

fun makeLightSwitch(
    switch1: Flow<Boolean>,
    switch2: Flow<Boolean>
): Flow<Boolean> =
    switch1.onStart { emit(false) }
        .combine(switch2.onStart { emit(false) }) { e1, e2 ->
            e1 xor e2
        }

fun makeLightSwitchToggle(
    switch1: Flow<*>,
    switch2: Flow<*>
): Flow<Boolean> =
    merge(switch1, switch2)
        .mapIndexed { index, _ -> index % 2 == 0 }

fun polonaisePairing(
    track1: Flow<Person>,
    track2: Flow<Person>
): Flow<Pair<Person, Person>> =
    track1.zip(track2) { p1, p2 -> Pair(p1, p2) }
```

Solution: MessageService

```
class MessageService(
    private val messageRepository: MessageRepository
) {
    fun threadsSearch(
        query: Flow<String>
    ): Flow<MessageThread> = query
        .flatMapLatest { messageRepository.searchThreads(it) }

    fun subscribeThreads(
        threads: Flow<MessageThread>
    ): Flow<MessageThreadUpdate> = threads
        .flatMapMerge(concurrency = Int.MAX_VALUE) {
            messageRepository.subscribeThread(it.id)
        }

    fun sendMessages(
        messages: Flow<List<Message>>
    ): Flow<MessageSendingResponse> = messages
        .flatMapConcat { messageRepository.sendMessages(it) }
}
```

Solution: Update ProductService

```
class ProductService(
    private val productRepository: ProductRepository,
    backgroundScope: CoroutineScope,
) {
    private val activeObservers = AtomicInteger(0)
    private val updatedProducts = productRepository
        .observeProductUpdates()
        .distinctUntilChanged()
        .flatMapMerge(concurrency = Int.MAX_VALUE) {
            flow { emit(productRepository.fetchProduct(it)) }
        }
        .shareIn(
            scope = backgroundScope,
            started = SharingStarted.WhileSubscribed(),
        )

    fun observeProducts(categories: Set<String>): Flow<Product> =
        updatedProducts
            .filter { it.category in categories }
            .onStart { activeObservers.incrementAndGet() }
            .onCompletion { activeObservers.decrementAndGet() }

    fun activeObserversCount(): Int = activeObservers.get()
}
```

Solution: Update TemperatureService

```
class TemperatureService(
    private val temperatureDataSource: TemperatureDataSource,
    backgroundScope: CoroutineScope,
) {
    private val lastKnownTemperature =
        ConcurrentHashMap<String, Fahrenheit>()
    private val temperatureUpdates = temperatureDataSource
        .observeTemperatureUpdates()
        .map { it.city to celsiusToFahrenheit(it.temperature) }
        .onEach { (city, temp) -> lastKnownTemperature[city] = te\
mp }
        .shareIn(
            scope = backgroundScope,
            started = SharingStarted.Eagerly,
        )

    fun observeTemperature(city: String): Flow<Fahrenheit> =
        temperatureUpdates
            .filter { (updateCity, _) -> updateCity == city }
            .map { it.second }
            .onStart { lastKnownTemperature[city]?.let { emit(it)\
 } }

    fun getLastKnown(city: String): Fahrenheit? =
        lastKnownTemperature[city]

    fun getAllLastKnown(): Map<String, Fahrenheit> =
        lastKnownTemperature.toMap()

    private fun celsiusToFahrenheit(celsius: Double) =
        Fahrenheit(celsius * 9 / 5 + 32)
}
```

Solution: LocationService

This is how this problem can be solved using `stateIn`:

```
class LocationService(
    locationRepository: LocationRepository,
    backgroundScope: CoroutineScope,
) {
    private val locationFlow = locationRepository
        .observeLocation()
        .stateIn(
            backgroundScope,
            SharingStarted.Eagerly,
            null
        )

    fun observeLocation(): Flow<Location> = locationFlow
        .filterNotNull()

    fun currentLocation(): Location? = locationFlow.value
}
```

`StateFlow` has reply cache of size 1 by default. It also conflates, and eliminates repetitions of the same value in a row.

This is how this problem can be solved using `shareIn` and `replayCache`:

```
class LocationService(
    locationRepository: LocationRepository,
    backgroundScope: CoroutineScope,
) {
    private val location = locationRepository
        .observeLocation()
        .distinctUntilChanged()
        .shareIn(
            scope = backgroundScope,
            started = SharingStarted.Eagerly,
            replay = 1
        )

    fun observeLocation(): Flow<Location> = location
        .conflate()

    fun currentLocation(): Location? = location
```

```
        .replayCache
        .lastOrNull()
}
```

This is how this problem can be solved using `shareIn` and a property to store the last known location:

```
class LocationService(
    locationRepository: LocationRepository,
    backgroundScope: CoroutineScope,
) {
    private var lastKnownLocation: Location? = null
    private val locationFlow = locationRepository
        .observeLocation()
        .distinctUntilChanged()
        .onEach { lastKnownLocation = it }
        .shareIn(
            scope = backgroundScope,
            started = SharingStarted.Eagerly,
        )

    fun observeLocation(): Flow<Location> = locationFlow
        .conflate()

    fun currentLocation(): Location? = lastKnownLocation
}
```

Solution: PriceService

```
class PriceService(
    priceRepository: PriceRepository,
    backgroundScope: CoroutineScope,
) {
    private val prices =
        ConcurrentHashMap<ProductId, PriceConfig>()
    private val pricesObserver = priceRepository
        .observeUpdates()
        .onEach { prices.putAll(it) }
        .shareIn(
            scope = backgroundScope,
            started = SharingStarted.Eagerly
        )

    fun observePrices(): Flow<Map<ProductId, PriceConfig>> =
        pricesObserver
            .onSubscription { emit(currentPrices()) }
            .filter { it.isNotEmpty() }

    fun currentPrices(): Map<ProductId, PriceConfig> = prices
        .toMap()
}
```

Solution: NewsViewModel using stateIn

```
class NewsViewModel(
    newsRepository: NewsRepository,
) : BaseViewModel() {
    private val _progressVisible = MutableStateFlow(false)
    val progressVisible = _progressVisible.asStateFlow()

    private val _errors = Channel<Throwable>(Channel.UNLIMITED)
    val errors = _errors.receiveAsFlow()

    val newsToShow = newsRepository.fetchNews()
        .retry { error -> error is ApiException }
        .catch { error -> _errors.send(error) }
        .onStart { _progressVisible.value = true }
        .onCompletion { _progressVisible.value = false }
        .scan(emptyList<News>()) { acc, news -> acc + news }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.Eagerly,
            initialValue = emptyList<News>()
        )
}
```

Solution: Usage of generic types

The lines that fail are marked with `X`:

```
fun takeIntList(list: List<Int>) {}
takeIntList(listOf<Any>())      X
takeIntList(listOf<Nothing>())

fun takeIntMutableList(list: MutableList<Int>) {}
takeIntMutableList(mutableListOf<Any>())      X
takeIntMutableList(mutableListOf<Nothing>())  X

fun takeAnyList(list: List<Any>) {}
takeAnyList(listOf<Int>())
takeAnyList(listOf<Nothing>())

class BoxOut<out T>
fun takeBoxOutInt(box: BoxOut<Int>) {}
takeBoxOutInt(BoxOut<Int>())
takeBoxOutInt(BoxOut<Number>())  X
takeBoxOutInt(BoxOut<Nothing>())

fun takeBoxOutNumber(box: BoxOut<Number>) {}
takeBoxOutNumber(BoxOut<Int>())
takeBoxOutNumber(BoxOut<Number>())
takeBoxOutNumber(BoxOut<Nothing>())

fun takeBoxOutNothing(box: BoxOut<Nothing>) {}
takeBoxOutNothing(BoxOut<Int>())     X
takeBoxOutNothing(BoxOut<Number>())  X
takeBoxOutNothing(BoxOut<Nothing>())

fun takeBoxOutStar(box: BoxOut<*>) {}
takeBoxOutStar(BoxOut<Int>())
takeBoxOutStar(BoxOut<Number>())
takeBoxOutStar(BoxOut<Nothing>())

class BoxIn<in T>
fun takeBoxInInt(box: BoxIn<Int>) {}
takeBoxInInt(BoxIn<Int>())
takeBoxInInt(BoxIn<Number>())
takeBoxInInt(BoxIn<Nothing>())  X
takeBoxInInt(BoxIn<Any>())
```

Solution: Generic Response

```
sealed class Response<out R, out E>
class Success<out R>(val value: R) : Response<R, Nothing>()
class Failure<out E>(val error: E) : Response<Nothing, E>()
```

Solution: Generic Consumer

```
abstract class Consumer<in T> {
    abstract fun consume(elem: T)
}

class Printer<in T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}

class Sender<in T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}
```

Solution: ApplicationScope

```
class ApplicationScope(
    private val scope: CoroutineScope,
    private val applicationScope: ApplicationControlScope,
    private val loggingScope: LoggingScope,
) : CoroutineScope by scope,
    ApplicationControlScope by applicationScope,
    LoggingScope by loggingScope
```

Solution: Lateinit delegate

The naive solution to this problem would use `null` as a marker for a value that has not been initialized, but this solution would not work correctly for nullable types.

```
// Solution that does not work correctly for nullable types
class Lateinit<T>: ReadWriteProperty<Any?, T> {
    var value: T? = null

    override fun getValue(
        thisRef: Any?,
        prop: KProperty<*>
    ): T =
        value ?: error(
          "Uninitialized lateinit property ${prop.name}"
        )

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

You should replace "..." with "Uninitialized lateinit".

Correct solutions require a different way of keeping information about values that have not been initialized. It can be another property, a special flag, or an object.

```
class Lateinit<T>: ReadWriteProperty<Any?, T> {
    var value: T? = null
    var isInitialized = false

    override fun getValue(
        thisRef: Any?,
        prop: KProperty<*>
    ): T {
        if (!isInitialized) {
            error("Uninitialized lateinit property ${prop.name}")
        }
        return value as T
```

```
    }

    override fun setValue(
        thisRef: Any?,
        prop: KProperty<*>,
        value: T
    ) {
        this.value = value
        this.isInitialized = true
    }
}
```

This solution is not thread safe. If two threads try to set the value at the same time, one thread might overwrite the value set by the other thread. To make it thread safe, we can use a `synchronized` block.

```
class Lateinit<T>: ReadWriteProperty<Any?, T> {
    var value: Any? = NOT_INITIALIZED

    override fun getValue(
        thisRef: Any?,
        prop: KProperty<*>
    ): T {
        if (value == NOT_INITIALIZED) {
            error("Uninitialized lateinit property ${prop.name}")
        }
        return value as T
    }

    override fun setValue(
        thisRef: Any?,
        prop: KProperty<*>,
        value: T
    ) {
        this.value = value
    }

    companion object {
        val NOT_INITIALIZED = Any()
    }
}
```

```
class Lateinit<T> : ReadWriteProperty<Any?, T> {
    var value: ValueHolder<T> = NotInitialized

    override fun getValue(
        thisRef: Any?,
        prop: KProperty<*>
    ): T = when (val v = value) {
        NotInitialized ->
            error("Uninitialized lateinit property ${prop.name}")
        is Value -> v.value
    }

    override fun setValue(
        thisRef: Any?,
        prop: KProperty<*>,
        value: T
    ) {
        this.value = Value(value)
    }

    sealed interface ValueHolder<out T>
    class Value<T>(val value: T) : ValueHolder<T>
    object NotInitialized : ValueHolder<Nothing>
}
```

Solution: Blog Post Properties

```
data class BlogPost(
    val title: String,
    val content: String,
    val author: Author,
) {
    // Since this property is needed on average more than
    // once per blog post, and it is not expensive to
    // calculate, it is best to define it as a value.
    val authorName: String =
        "${author.name} ${author.surname}"

    // Since this property is needed on average more than
    // once per blog post, and it is expensive to
    // calculate, it is best to define it as a lazy
    val wordCount: Int by lazy {
        content.split("\\s+").size
    }

    // Since this property is needed on average less than
    // once per blog post, and it is not expensive to
    // calculate, it is best to define it as a getter.
    val isLongRead: Boolean
        get() = content.length > 1000

    // Since this property is very expensive to calculate,
    // it is best to define it as a lazy
    val summary: String by lazy {
        generateSummary(content)
    }

    private fun generateSummary(content: String): String =
        content.take(100) + "..."
}
```

Solution: Mutable lazy delegate

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

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

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T {
        if (initializer != null) {
            value = initializer?.invoke()
            initializer = null
        }
        return value as T
    }

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

This solution lacks thread safety. The simplest way to make it thread safe is to use a `synchronized` block. A more efficient solution is to use the `AtomicReference` and `compareAndSet` functions.

Solution: Coroutine time measurement

```
@OptIn(ExperimentalContracts::class)
suspend fun measureCoroutine(
    body: suspend () -> Unit
): Duration {
    contract {
        callsInPlace(body, InvocationKind.EXACTLY_ONCE)
    }
    val dispatcher = coroutineContext[ContinuationInterceptor]
    return if (dispatcher is TestDispatcher) {
        val before = dispatcher.scheduler.currentTime
        body()
        val after = dispatcher.scheduler.currentTime
        after - before
    } else {
        measureTimeMillis {
            body()
        }
    }.milliseconds
}
```

Solution: Adjust Kotlin for Java usage

```
@file:JvmName("MoneyUtils")
package advanced.java

import java.math.BigDecimal

data class Money @JvmOverloads constructor(
    val amount: BigDecimal = BigDecimal.ZERO,
    val currency: Currency = Currency.EUR,
) {
    companion object {
        @JvmStatic
        fun eur(amount: String) =
            Money(BigDecimal(amount), Currency.EUR)

        @JvmStatic
        fun usd(amount: String) =
            Money(BigDecimal(amount), Currency.USD)

        @JvmField
        val ZERO_EUR = eur("0.00")
    }
}

@JvmName("sumMoney")
fun List<Money>.sum(): Money? {
    if (isEmpty()) return null
    val currency = this.map { it.currency }.toSet().single()
    return Money(
        amount = sumOf { it.amount },
        currency = currency
    )
}

operator fun Money.plus(other: Money): Money {
    require(currency == other.currency)
    return Money(amount + other.amount, currency)
}

enum class Currency {
    EUR, USD
}
```

Solution: Multiplatform LocalDateTime

Kotlin/JVM code:

```
import java.time.LocalDateTime as JavaLocalDateTime

actual typealias LocalDateTime = java.time.LocalDateTime

actual fun now(): LocalDateTime = JavaLocalDateTime.now()

actual fun parseLocalDateTime(str: String): LocalDateTime =
    JavaLocalDateTime.parse(str)
```

Kotlin/JS code:

```
import kotlin.js.Date

actual class LocalDateTime(
    val date: Date = Date(),
) {
    actual fun getSecond(): Int = date.getSeconds()

    actual fun getMinute(): Int = date.getMinutes()

    actual fun getHour(): Int = date.getHours()

    actual fun plusSeconds(seconds: Long): LocalDateTime =
        LocalDateTime(Date(date.getTime() + seconds * 1000))
}

actual fun now(): LocalDateTime = LocalDateTime()

actual fun parseLocalDateTime(str: String): LocalDateTime =
    LocalDateTime(Date(Date.parse(str)))
```

Solution: Migrating a Kotlin/JVM project to KMP

First, you need to transform the `build.gradle.kts` configuration to Kotlin Multiplatform, and change the "kotlin("jvm")" plugin to "kotlin("multiplatform")". Then, you need to configure the targets. This is what this configuration might look like:

```
plugins {
    kotlin("multiplatform") version "1.8.0"
    application
}

group = "org.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

kotlin {
    jvm {
        withJava()
    }
    js(IR) {
        moduleName = "sudoku-generator"
        browser()
        binaries.library()
    }
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation(
"org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4")
            }
        }
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val jvmMain by getting
        val jvmTest by getting
        val jsMain by getting
        val jsTest by getting
```

```
    }
}
```

Now move all your code from `src/main/kotlin` to `src/commonMain/kotlin`. Then move all your tests from `src/test/kotlin` to `src/commonTest/kotlin`.

In the tests, you need to replace the test annotations with those from the kotlin-test library.

You need to add a `src/jsMain/kotlin` folder and create a `SudokuGeneratorJs.kt` file in it. It could contain the following code:

```
@file:OptIn(ExperimentalJsExport::class)

import generator.SudokuGenerator
import solver.SudokuSolver

@JsExport
@JsName("SudokuGenerator")
class SudokuGeneratorJs {
    private val generator = SudokuGenerator()
    private val solver = SudokuSolver()

    fun generateSudoku() = generator
        .generate(solver)
        .let {
            Sudoku(it.solved.toJs(), it.sudoku.toJs())
        }
}

@JsExport
class Sudoku(
    val solved: Array<Array<Int?>>,
    val sudoku: Array<Array<Int?>>
)

fun SudokuState.toJs(): Array<Array<Int?>> = List(9) { row ->
    List(9) { col ->
        val cell = this.cells[SudokuState.Position(row, col)]
        when (cell) {
            is SudokuState.CellState.Filled -> cell.value
            is SudokuState.CellState.Empty, null -> null
        }
    }.toTypedArray()
}.toTypedArray()
```

Solution: Function caller

```
class FunctionCaller {
    private var values: MutableMap<KType, Any?> =
        mutableMapOf()

    inline fun <reified T> setConstant(value: T) {
        setConstant(typeOf<T>(), value)
    }

    fun setConstant(type: KType, value: Any?) {
        values[type] = value
    }

    fun <T> call(function: KFunction<T>): T {
        val args = function.parameters
            .filter { param ->
                values.containsKey(param.type)
            }
            .associateWith { param ->
                val type = param.type
                val value = values[type]
                require(param.isOptional || value != null) {
                    "No value for $type"
                }
                value
            }
        return function.callBy(args)
    }
}
```

Solution: Object serialization to JSON

```
fun serializeToJson(value: Any): String = valueToJson(value)

private fun objectToJson(any: Any): String {
    val reference = any::class
    val classNameMapper = reference
        .findAnnotation<SerializationNameMapper>()
        ?.let(::createMapper)
    val ignoreNulls = reference
        .hasAnnotation<SerializationIgnoreNulls>()

    return reference
        .memberProperties
        .filterNot { it.hasAnnotation<SerializationIgnore>()}
        .mapNotNull { prop ->
            val annotationName = prop
                .findAnnotation<SerializationName>()
            val mapper = prop
                .findAnnotation<SerializationNameMapper>()
                ?.let(::createMapper)
            val name = annotationName?.name
                ?: mapper?.map(prop.name)
                ?: classNameMapper?.map(prop.name)
                ?: prop.name
            val value = prop.call(any)
            if (ignoreNulls && value == null) {
                return@mapNotNull null
            }
            "\"${name}\": ${valueToJson(value)}"
        }
        .joinToString(
            prefix = "{",
            postfix = "}",
        )
}

private fun valueToJson(value: Any?): String = when (value) {
    null, is Number, is Boolean -> "$value"
    is String, is Char, is Enum<*> -> "\"$value\""
    is Iterable<*> -> iterableToJson(value)
    is Map<*, *> -> mapToJson(value)
    else -> objectToJson(value)
}
```

```
private fun iterableToJson(any: Iterable<*>): String = any
    .joinToString(
        prefix = "[",
        postfix = "]",
        transform = ::valueToJson
    )

private fun mapToJson(any: Map<*, *>) = any.toList()
    .joinToString(
        prefix = "{",
        postfix = "}",
        transform = {
            "\"${it.first}\": ${valueToJson(it.second)}"
        }
    )

private fun createMapper(
    annotation: SerializationNameMapper
): NameMapper =
    annotation.mapper.objectInstance
        ?: createWithNoargConstructor(annotation)
        ?: error("Cannot create mapper")

private fun createWithNoargConstructor(
    annotation: SerializationNameMapper
): NameMapper? =
    annotation.mapper
        .constructors
        .find { it.parameters.isEmpty() }
        ?.call()
```

Solution: Object serialization to XML

```
fun serializeToXml(value: Any): String = valueToXml(value)

private fun objectToXml(any: Any): String {
    val reference = any::class
    val classNameMapper = reference
        .findAnnotation<SerializationNameMapper>()
        ?.let(::createMapper)
    val simpleName = reference.simpleName.orEmpty()
    val className = classNameMapper?.map(simpleName)
        ?: simpleName
    val ignoreNulls = reference
        .hasAnnotation<SerializationIgnoreNulls>()

    return reference
        .memberProperties
        .filterNot { it.hasAnnotation<SerializationIgnore>()}
        .mapNotNull { prop ->
            val annotationName = prop
                .findAnnotation<SerializationName>()
            val mapper = prop
                .findAnnotation<SerializationNameMapper>()
                ?.let(::createMapper)
            val name = annotationName?.name
                ?: mapper?.map(prop.name)
                ?: classNameMapper?.map(prop.name)
                ?: prop.name
            val value = prop.call(any)
            if (ignoreNulls && value == null) {
                return@mapNotNull null
            }
            "<$name>${valueToXml(value)}</$name>"
        }
        .joinToString(
            separator = "",
            prefix = "<$className>",
            postfix = "</$className>",
        )
}

private fun valueToXml(value: Any?): String = when (value) {
    null, is Number, is Boolean, is String,
        is Char, is Enum<*> -> "$value"
```

```
    is Iterable<*> -> iterableToJson(value)
    is Map<*, *> -> mapToJson(value)
    else -> objectToXml(value)
}

private fun iterableToJson(any: Iterable<*>): String = any
    .joinToString(
        separator = "",
        transform = ::valueToXml
    )

private fun mapToJson(any: Map<*, *>) = any.toList()
    .joinToString(
        separator = "",
        transform = { (name, value) ->
            "<$name>${valueToXml(value)}</$name>"
        }
    )

private fun createMapper(
    annotation: SerializationNameMapper
): NameMapper =
    annotation.mapper.objectInstance
        ?: createWithNoargConstructor(annotation)
        ?: error("Cannot create mapper")

private fun createWithNoargConstructor(
    annotation: SerializationNameMapper
): NameMapper? =
    annotation.mapper
        .constructors
        .find { it.parameters.isEmpty() }
        ?.call()
```

Solution: DSL-based dependency injection library

```
class Registry {
    private val creatorsRegistry =
        mutableMapOf<KType, () -> Any?>()
    private val instances =
        mutableMapOf<KType, Any?>()

    inline fun <reified T> singleton(
        noinline creator: Registry.() -> T
    ) {
        singleton(typeOf<T>(), creator)
    }

    fun singleton(type: KType, creator: Registry.() -> Any?){
        creatorsRegistry[type] = {
            instances.getOrPut(type) { creator.invoke(this) }
        }
    }

    inline fun <reified T> register(
        noinline creator: Registry.() -> T
    ) {
        register(typeOf<T>(), creator)
    }

    fun register(type: KType, creator: Registry.() -> Any?) {
        creatorsRegistry[type] = { creator(this) }
    }

    inline fun <reified T> get(): T = get(typeOf<T>()) as T

    fun get(key: KType): Any? {
        require(exists(key)) { "The $key not in registry." }
        return creatorsRegistry[key]?.invoke()
    }

    fun exists(key: KType) =
        creatorsRegistry.containsKey(key)

    inline fun <reified T> exists(): Boolean =
        exists(typeOf<T>())
}
```

```
fun registry(init: Registry.() -> Unit) = Registry()
    .apply(init)
```

Solution: Mocking library

```
import org.junit.Test
import java.lang.reflect.Method
import java.lang.reflect.Proxy
import kotlin.reflect.KClass
import kotlin.test.assertEquals
import kotlin.test.assertNotNull

class MockRegistry {
    inline fun <reified T : Any> mock(): T = mock(T::class)
    private var recording = false
    private var bodyToStore: (() -> Any?)? = null
    private var handlers = mutableMapOf<Record, () -> Any?>()

    inline fun <reified T: Any> mock(): T = mock(T::class.java)

    fun <T : Any> mock(classRef: KClass<T>): T {
        val proxy = Any()
        return Proxy.newProxyInstance(
            classRef.java.classLoader,
            arrayOf(classRef.java)
        ) { _, method, args ->
            val record = Record(proxy, method, args?.toList())
            if (recording) {
                handlers[record] = checkNotNull(bodyToStore)
                throw RecordedCorrectlyException()
            } else {
                checkNotNull(handlers[record]) { "No recording...\
" }
                    .invoke()
            }
        } as T
    }

    fun <T> setBody(functionCall: () -> T, body: () -> T) {
        try {
            recording = true
            this.bodyToStore = body
            functionCall()
        } catch (e: RecordedCorrectlyException) {
            // no-op
        } finally {
            recording = false
```

```
            this.bodyToStore = null
        }
    }

    fun <T> setReturnValue(functionCall: () -> T, value: T) {
        setBody(functionCall) { value }
    }

    class RecordedCorrectlyException : RuntimeException()

    data class Record(
        val proxy: Any,
        val method: Method,
        val arguments: List<Any?>?,
    )
}
```

Solution: Annotation Processing execution measurement wrapper

Here is an example solution code:

```
package academy.kt

import com.squareup.javapoet.*
import javax.annotation.processing.AbstractProcessor
import javax.annotation.processing.RoundEnvironment
import javax.lang.model.AnnotatedConstruct
import javax.lang.model.SourceVersion
import javax.lang.model.element.*
import javax.lang.model.type.TypeMirror


class GenerateMeasuredWrapperProcessor : AbstractProcessor() {

    override fun getSupportedAnnotationTypes(): Set<String> =
        setOf(Measured::class.qualifiedName!!)

    override fun getSupportedSourceVersion(): SourceVersion =
        SourceVersion.latestSupported()

    override fun process(
        annotations: Set<TypeElement>,
        roundEnv: RoundEnvironment
    ): Boolean {
        roundEnv.getElementsAnnotatedWith(Measured::class.java)
            .filterIsInstance<ExecutableElement>()
            .groupBy { it.enclosingElement!! }
            .forEach { (clazz) -> generateMeasuredClass(clazz) }
        return true
    }

    private fun generateMeasuredClass(classElement: Element) {
        val className = classElement.simpleName.toString()
        val measuredName = "Measured$className"
        val measuredPackage = processingEnv.elementUtils
            .getPackageOf(classElement)
            .qualifiedName
            .toString()
        val publicMethods = classElement.enclosedElements
```

```
            .filter { it.kind == ElementKind.METHOD }
            .filter { it.modifiers.contains(Modifier.PUBLIC) }
            .filterIsInstance<ExecutableElement>()
        val constructorParameters = classElement.enclosedElements
            .filter { it.kind == ElementKind.CONSTRUCTOR }
            .filterIsInstance<ExecutableElement>()
            .flatMap { it.parameters }

        JavaFile.builder(
            measuredPackage,
            TypeSpec
                .classBuilder(measuredName)
                .addField(
                    FieldSpec.builder(
                        classElement.asType().toTypeSpec(),
                        "wrapper",
                        Modifier.PRIVATE
                    ).build()
                )
                .addMethod(
                    MethodSpec.constructorBuilder()
                        .addParameter(
                            classElement.asType().toTypeSpec(),
                            "wrapper"
                        )
                        .addStatement("this.wrapper = wrapper")
                        .build()
                )
                .addMethod(
                    MethodSpec.constructorBuilder()
                        .addParameters(constructorParameters
                            .filterIsInstance<VariableElement>()
                            .map {
                                buildMethodParameter(it)
                            })
                        .addStatement(
                            "this.wrapper = new ${
                                classElement.simpleName
                            }(${
                                constructorParameters
                                    .joinToString { it.simpleName\
.toString() }
                            })"
                        )
```

```
                    .build()
            )
            .addMethods(publicMethods.map {
                buildInterfaceMethod(className, it)
            }
            )
            .build()
    ).build()
        .writeTo(processingEnv.filer)
}

private fun buildInterfaceMethod(
    className: String,
    method: ExecutableElement
): MethodSpec {
    val methodName = method.simpleName.toString()
    return MethodSpec
        .methodBuilder(methodName)
        .addModifiers(method.modifiers)
        .addParameters(
            method.parameters
                .map(::buildMethodParameter)
        )
        .returns(
            TypeName.get(method.returnType)
                .annotated(
                    method.returnType.getAnnotationSpecs()
                )
        )
        .addAnnotations(method
            .annotationMirrors
            .filter { !isMeasured(it) }
            .map(AnnotationSpec::get))
        .addCode(run {
            val params = method.parameters
                .joinToString { it.simpleName }

            if (method.annotationMirrors.none(::isMeasured))
                "return wrapper.$methodName($params);"
            else {
                val retType = method.returnType
                """
                long before = System.currentTimeMillis();
                $retType value = wrapper.$methodName($params);
```

```
                    long after = System.currentTimeMillis();
                    System.out.println("$methodName from $className took " + (after - before) + " ms");
                    return value;
                    """.trimIndent()
                }
            })
            .build()
    }

    private fun isMeasured(it: AnnotationMirror) =
        (it.annotationType.asElement() as TypeElement)
            .qualifiedName
            ?.toString() == Measured::class.qualifiedName

    private fun buildMethodParameter(
        variableElement: VariableElement
    ): ParameterSpec {
        val asType = variableElement.asType()
        return ParameterSpec
            .builder(
                TypeName.get(asType)
                    .annotated(asType.getAnnotationSpecs()),
                variableElement.simpleName.toString()
            )
            .addAnnotations(variableElement.getAnnotationSpecs())
            .build()
    }
}

private fun TypeMirror.toTypeSpec() = TypeName.get(this)
    .annotated(this.getAnnotationSpecs())

private fun AnnotatedConstruct.getAnnotationSpecs() =
    annotationMirrors.map(AnnotationSpec::get)
```

Solution: KSP execution measurement wrapper

Here is an example solution code:

```
package academy.kt

import com.google.devtools.ksp.closestClassDeclaration
import com.google.devtools.ksp.getDeclaredFunctions
import com.google.devtools.ksp.isConstructor
import com.google.devtools.ksp.isPublic
import com.google.devtools.ksp.processing.*
import com.google.devtools.ksp.symbol.*
import com.squareup.kotlinpoet.*
import com.squareup.kotlinpoet.ksp.toAnnotationSpec
import com.squareup.kotlinpoet.ksp.toKModifier
import com.squareup.kotlinpoet.ksp.toTypeName
import java.io.OutputStreamWriter
import java.nio.charset.StandardCharsets

class MeasuredWrapperGenerator(
    private val codeGenerator: CodeGenerator,
    private val logger: KSPLogger,
) : SymbolProcessor {
    private val annotationName = Measured::class.qualifiedName!!

    override fun process(resolver: Resolver): List<KSAnnotated> {
        resolver
            .getSymbolsWithAnnotation(
                annotationName
            )
            .filterIsInstance<KSFunctionDeclaration>()
            .groupBy { it.closestClassDeclaration() }
            .forEach { (classDeclaration, _) ->
                if (classDeclaration != null) {
                    generateMeasuredClass(classDeclaration)
                }
            }

        return emptyList()
    }

    private fun generateMeasuredClass(
        classElement: KSClassDeclaration
    ) {
```

```
        val className = classElement.simpleName.getShortName()
        val measuredName = "Measured$className"
        val measuredPackage = classElement.packageName.asString()
        val publicMethods = classElement
            .getDeclaredFunctions()
            .filter { !it.isConstructor() && it.isPublic() }
            .toList()

        val fileSpec = FileSpec.builder(
            measuredPackage,
            "$measuredName.kt"
        )
            .addType(
                TypeSpec.classBuilder(measuredName)
                    .primaryConstructor(
                        FunSpec.constructorBuilder()
                            .addParameter(
                                "wrapper",
                                classElement.asType(emptyList())
                                    .toTypeName()
                            )
                            .build()
                    )
                    .addFunction(
                        FunSpec.constructorBuilder()
                            .addParameters(
                                classElement.primaryConstructor!!\
.parameters
                                    .map { buildInterfaceMethodPa\
rameter(it) }
                            )
                            .callThisConstructor(
                                "$className(${
                                    classElement.primaryConstruct\
or!!.parameters.joinToString {
                                        it.name?.getShortName().o\
rEmpty()
                                    }
                                })"
                            )
                            .build()
                    )
                    .addProperty(
                        PropertySpec.builder(
```

```
                        "wrapper",
                        classElement.asType(emptyList())
                            .toTypeName()
                    ).initializer("wrapper")
                        .build()
                )
                .addFunctions(
                    publicMethods
                        .map { buildMethod(className, it) }
                        .toList()
                )
                .build()
        )
        .build()

    val dependencies = Dependencies(
        aggregating = false,
        classElement.containingFile!!
    )
    val file = codeGenerator.createNewFile(
        dependencies, measuredPackage, measuredName
    )
    OutputStreamWriter(file, StandardCharsets.UTF_8)
        .use(fileSpec::writeTo)
}

private fun buildMethod(
    className: String,
    method: KSFunctionDeclaration
): FunSpec {
    val methodName = method.simpleName.getShortName()
    return FunSpec.builder(methodName)
        .addModifiers(
            method.modifiers
                .mapNotNull { it.toKModifier() }.toList()
        )
        .addParameters(
            method.parameters
                .map { buildInterfaceMethodParameter(it) }
        )
        .returns(method.returnType!!.toTypeName())
        .addAnnotations(method
            .annotations
            .filter { !isMeasured(it) }
```

```
                .map { it.toAnnotationSpec() }
                .toList())
            .addCode(
                if (method.annotations.none { isMeasured(it) })
                    "return wrapper.$methodName(${
                        method.parameters.joinToString {
                            it.name?.getShortName().orEmpty()
                        }
                    })"
                else
                    """
                    val before = System.currentTimeMillis()
                    val value = wrapper.$methodName(${
                        method.parameters.joinToString {
                            it.name?.getShortName().orEmpty()
                        }
                    })
                    val after = System.currentTimeMillis()
                    println("$methodName from $className took ${'\
$'}{after-before} ms")
                    return value
                    """.trimIndent()
            )
            .build()
    }

    private fun isMeasured(annotation: KSAnnotation) = annotation
        .annotationType
        .resolve()
        .declaration
        .qualifiedName
        ?.asString() == annotationName

    private fun buildInterfaceMethodParameter(
        variableElement: KSValueParameter,
    ): ParameterSpec = ParameterSpec
        .builder(
            variableElement.name!!.getShortName(),
            variableElement.type.toTypeName(),
        )
        .addAnnotations(
            variableElement.annotations
                .map { it.toAnnotationSpec() }.toList()
        )
```

```
        .build()
}
```

Solution: A mutability problem

Any operation that changes the `name` property will make the code print `false`. This is because `mutableSetOf` uses `hashCode` and `equals` to check if an element is already in the set. When we add an element to the set, the set calculates its hash code and stores it. When we call `contains`, the set calculates the hash code of the element we pass as an argument and compares it with the hash code of the element stored in the set. Since the hash code of `name` is calculated based on the value of the `name` property, changing the value of this property will change the hash code. Thus, the set will not be able to find the element we are looking for.

```
fun main() {
    val set = mutableSetOf<Name>()
    val name = Name("AAA")
    set.add(name)
    name.name = "BBB" // or any other text
    println(set.contains(name)) // false
    println(set.first() == name) // true
}
```

Solution: Flow with history

The problem is that the `withHistory` function always returns the same list reference; so, when its values are changed, all the previously emitted lists are also changed. The simplest solution is to use a read-only list instead of a mutable list.

```
fun <T> Flow<T>.withHistory(): Flow<List<T>> = flow {
    var history = listOf<T>()
    emit(history)
    collect {
        history += it
        emit(history)
    }
}
```

This function can also be implemented using `scan`:

```
fun <T> Flow<T>.withHistory(): Flow<List<T>> =
    scan(emptyList()) { history, value ->
        history + value
    }
```

Solution: Correct InMemoryUserRepository

Solution using a `synchronized` block and a read-only set:

```
class InMemoryUserRepository {
    private var users = setOf<User>()
    private val lock = Any()

    fun addUser(user: User) = synchronized(lock) {
        users += user
    }

    fun getUsers() = users

    fun hasUser(user: User): Boolean = user in users

    fun changeSurname(
        userId: Int,
        newSurname: String
    ) = synchronized(lock) {
        val oldUser = users
            .find { it.id == userId }
            ?: return@synchronized
        val newUser = oldUser.copy(surname = newSurname)
        users = users - oldUser + newUser
    }

    fun changeAllSurnames(
        newSurname: String
    ) = synchronized(lock) {
        users = users
            .map { it.copy(surname = newSurname) }
            .toSet()
    }

    data class User(
        val id: Int,
        val name: String,
        val surname: String
    )
}
```

Solution using a `synchronized` block and a mutable set:

```
class InMemoryUserRepository {
    private val users = mutableSetOf<User>()
    private val lock = Any()

    fun addUser(user: User) = synchronized(lock) {
        users += user
    }

    // Synchronization is needed here, because making a copy
    // requires iterating over the whole set, and we don't
    // want to have a concurrent modification exception
    fun getUsers() = synchronized(lock) {
        users.toSet()
    }

    fun hasUser(user: User): Boolean = synchronized(lock) {
        user in users
    }

    fun changeSurname(
        userId: Int,
        newSurname: String
    ) = synchronized(lock) {
        val oldUser = users
            .find { it.id == userId }
            ?: return@synchronized
        val newUser = oldUser.copy(surname = newSurname)
        users -= oldUser
        users += newUser
    }

    fun changeAllSurnames(
        newSurname: String
    ) = synchronized(lock) {
        val newUsers = users
            .map { it.copy(surname = newSurname) }
        users.clear()
        users.addAll(newUsers)
    }

    data class User(
        val id: Int,
        val name: String,
        val surname: String
```

```
    )
}
```

This problem cannot be safely solved using `ConcurrentHashMap` because it does not allow compound operations like `changeSurname` or `changeAllSurnames` to be secured. It is possible to use `ConcurrentHashMap` for `addUser` and `hasUser`, but it is not possible to use it for `getUsers` because it does not allow iteration over the whole set.

```
class InMemoryUserRepository {
    private val users = ConcurrentHashMap.newKeySet<User>()

    fun addUser(user: User) {
        users += user
    }

    fun getUsers() = users.toSet()

    fun hasUser(user: User): Boolean = user in users

    data class User(
        val id: Int,
        val name: String,
        val surname: String
    )
}
```

Solution: Chunking a flow

This is my solution to this exercise:

```
fun <T> Flow<T>.chunked(
    duration: Duration
): Flow<List<T>> = channelFlow {
    var chunk = LinkedList<T>()
    val mutex = Mutex()
    val sendIfNotEmpty = suspend {
        if (chunk.isNotEmpty()) {
            mutex.withLock {
                send(chunk)
                chunk = LinkedList<T>()
            }
        }
    }

    val timerJob = launch {
        try {
            while (true) {
                delay(duration)
                sendIfNotEmpty()
            }
        } finally {
            sendIfNotEmpty()
        }
    }

    try {
        collect {
            chunk.add(it)
        }
    } finally {
        timerJob.cancel()
    }
}
```

I will be happy to see other solutions! This is not an easy problem, but it certainly has many solutions.

Solution: Specify expectations

```
fun Notifier.notifyUser(user: User?) {
    user ?: return
    // or
    // if(user == null) return

    if(!checkId(user.id)) throw IncorrectId()

    require(user.name != null) {
        "Name cannot be null"
    }
    require(user.surname != null) {
        "Surname cannot be null"
    }

    check(initialized) {
        "Notifier is not initialized"
    }

    val result = notifyPerson(user.id)

    assert(result)
}
```

Solution: Make DeckConnector comparable

```
class DeckConnector(
    val deckName: String
) : Comparable<DeckConnector> {
    var state: ConnectionState = ConnectionState.Initial

    override fun equals(other: Any?): Boolean =
        this === other ||
                other is DeckConnector &&
                other.deckName == deckName &&
                other.state == state

    override fun hashCode(): Int =
        Objects.hash(deckName, state)

    override fun compareTo(other: DeckConnector): Int =
        compareValuesBy(this, other,
            { it.deckName },
            { it.state }
        )

    enum class ConnectionState {
        Initial,
        Connected,
        Disconnected
    }
}
```

Solution: Correct EventListenerRegistry

The first problem is that the `handler` variable in the `EventListener` class is not cleared when the listener is canceled. It might be a heavy object, which is why we should clear it. We can do this by setting `handler` to `null` in the `cancel` method.

The second, smaller problem is that the `listeners` variable in `EventListenerRegistry` does not eliminate canceled listeners. We can do this by removing canceled listeners from the set when listeners are invoked.

```
class EventListenerRegistry<E> {
    private var listeners = ConcurrentHashMap
        .newKeySet<EventListener<E>>()
    private val lock = Any()

    fun addEventListener(
        event: E,
        handler: () -> Unit
    ): EventListener<E> = synchronized(lock) {
        val listener = EventListener(event, handler)
        listeners += listener
        listener
    }

    fun invokeListeners(event: E) {
        val eventListeners = listeners
            .filter { it.event == event }
        for (listener in eventListeners) {
            if (listener.isActive) {
                listener.handleEvent()
            } else {
                listeners.remove(listener)
            }
        }
    }
}

class EventListener<E>(
    val event: E,
    handler: () -> Unit,
) {
    private var handler: (() -> Unit)? = handler
    val isActive: Boolean get() = handler != null
```

```
    fun handleEvent() {
        handler?.invoke()
    }

    fun cancel() {
        handler = null
    }
}
```

This problem can also be solved by removing canceled listeners from a set immediately after canceling them, but this solution would require event listeners to have access to the repository, which might be problematic in some cases.

Solution: Market data processing optimization

The key optimizations that should be applied are:

- To change domain objects to use primitives and inline classes where possible.
- Optimizing `observeUpdates` to filter only if necessary, and to use a set to store tickers.

In my case, the code initially took around 20 671 ms, after changing domain objects to use primitives and inline classes it took around 18 750 ms, and after optimizing `observeUpdates` it took around 14 128 ms

```
import effective.efficient.Filter.*
import effective.efficient.Filter.Relation.*
import effective.efficient.Filter.SnapshotPart.*
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch
import java.util.concurrent.ConcurrentHashMap
import kotlin.random.Random
import kotlin.system.measureTimeMillis

data class TickerSnapshot(
    val ticker: Ticker,
    val snapshot: Snapshot,
)

data class Snapshot(
    var bid: PriceSizeTime?,
    var ask: PriceSizeTime?,
    var last: PriceSizeTime?,
)

data class PriceSizeTime(
    val price: Price,
    val size: Int = -1,
    val time: Long = -1,
)

@JvmInline
value class Ticker(val value: String)

@JvmInline
```

```
value class Price(val value: Float)

sealed interface Event {
    val ticker: String
}

data class BidEvent(
    override val ticker: String,
    val price: Float = Float.NaN,
    val size: Int = -1,
    val time: Long = -1
) : Event

data class AskEvent(
    override val ticker: String,
    val price: Float = Float.NaN,
    val size: Int = -1,
    val time: Long = -1
) : Event

data class TradeEvent(
    override val ticker: String,
    val price: Float = Float.NaN,
    val size: Int = -1,
    val time: Long = -1
) : Event

val tickers = List(1000) { Ticker("Ticker$it") }

// Do not touch this one
class MarketClient {
    fun observe() = flow {
        val random = Random(123456789)
        while (true) {
            val event = when ((0..2).random(random)) {
                0 -> BidEvent(
                    tickers.random(random).value,
                    if (random.nextInt(100) == 1) Float.NaN else \
(0..100).random(random).toFloat(),
                    if (random.nextInt(100) == 1) -1 else (0..100\
).random(random),
                    if (random.nextInt(100) == 1) -1 else System.\
currentTimeMillis()
                )
```

```
                1 -> AskEvent(
                    tickers.random(random).value,
                    if (random.nextInt(100) == 1) Float.NaN else \
(0..100).random(random).toFloat(),
                    if (random.nextInt(100) == 1) -1 else (0..100\
).random(random),
                    if (random.nextInt(100) == 1) -1 else System.\
currentTimeMillis()
                )

                else -> TradeEvent(
                    tickers.random(random).value,
                    if (random.nextInt(100) == 1) Float.NaN else \
(0..100).random(random).toFloat(),
                    if (random.nextInt(100) == 1) -1 else (0..100\
).random(random),
                    if (random.nextInt(100) == 1) -1 else System.\
currentTimeMillis()
                )
            }
            emit(event)
        }
    }
}


class MarketRepository(
    private val client: MarketClient,
    backgroundScope: CoroutineScope,
) {
    private val snapshots = ConcurrentHashMap<Ticker, TickerSnaps\
hot>()
    private val updates = MutableSharedFlow<TickerSnapshot>()

    fun observeUpdates() = updates
        .onStart { snapshots.forEach { emit(it.value) } }

    init {
        backgroundScope.launch {
            client.observe().collect {
                when (it) {
                    is BidEvent -> {
                        val snapshot = snapshots.getOrPut(Ticker(\
```

```
it.ticker)) { TickerSnapshot(Ticker(it.ticker), Snapshot(null, nu
ll, null)) }
                        snapshot.snapshot.bid = PriceSizeTime(Pri\
ce(it.price), it.size, it.time)
                        val ticker = Ticker(it.ticker)
                        snapshots[ticker] = snapshot
                        updates.emit(snapshot)
                    }

                    is AskEvent -> {
                        val snapshot = snapshots.getOrPut(Ticker(\
it.ticker)) { TickerSnapshot(Ticker(it.ticker), Snapshot(null, nu
ll, null)) }
                        snapshot.snapshot.ask = PriceSizeTime(Pri\
ce(it.price), it.size, it.time)
                        val ticker = Ticker(it.ticker)
                        snapshots[ticker] = snapshot
                        updates.emit(snapshot)
                    }

                    is TradeEvent -> {
                        val snapshot = snapshots.getOrPut(Ticker(\
it.ticker)) { TickerSnapshot(Ticker(it.ticker), Snapshot(null, nu
ll, null)) }
                        snapshot.snapshot.last = PriceSizeTime(Pr\
ice(it.price), it.size, it.time)
                        val ticker = Ticker(it.ticker)
                        snapshots[ticker] = snapshot
                        updates.emit(snapshot)
                    }
                }
            }
        }
    }
}

sealed class Filter {
    data object All : Filter()
    class Or(val filters: List<Filter>) : Filter()
    class And(val filters: List<Filter>) : Filter()
    class PrizeCondition(
        val snapshotPart: SnapshotPart,
        val relation: Relation,
        val value: Float,
```

```
    ) : Filter()

    class TickerIs(val tickers: List<Ticker>) : Filter()
    class Not(val filter: Filter) : Filter()

    enum class SnapshotPart {
        Ask, Bid, Last, Spread
    }

    enum class Relation {
        GreaterThan, LessThan, Equal
    }

    fun check(tickerSnapshot: TickerSnapshot): Boolean = when (th\
is) {
        All -> true
        is Or -> filters.any { it.check(tickerSnapshot) }
        is And -> filters.all { it.check(tickerSnapshot) }
        is PrizeCondition -> run {
            val snapshotPrize = when (snapshotPart) {
                Ask -> tickerSnapshot.snapshot.ask?.price?.value.\
takeUnless { it == Float.NaN } ?: return@run false
                Bid -> tickerSnapshot.snapshot.bid?.price?.value.\
takeUnless { it == Float.NaN } ?: return@run false
                Last -> tickerSnapshot.snapshot.last?.price?.valu\
e.takeUnless { it == Float.NaN } ?: return@run false
                Spread -> {
                    val bid =
                        tickerSnapshot.snapshot.bid?.price?.value\
.takeUnless { it == Float.NaN } ?: return@run false
                    val ask =
                        tickerSnapshot.snapshot.ask?.price?.value\
.takeUnless { it == Float.NaN } ?: return@run false
                    ask - bid
                }
            }
            when (relation) {
                GreaterThan -> snapshotPrize > value
                LessThan -> snapshotPrize < value
                Equal -> snapshotPrize == value
            }
        }

        is TickerIs -> tickers.contains(tickerSnapshot.ticker)
```

```
        is Not -> !filter.check(tickerSnapshot)
    }
}

class TradeService(
    private val repository: MarketRepository,
) {
    fun observeUpdates(
        filter: Filter,
        tickers: List<Ticker>? = null,
    ): Flow<TickerSnapshot> {
        val tickersSet = tickers.orEmpty().toSet()
        return repository.observeUpdates()
            .filerIf(tickers != null) { it.ticker in tickersSet }
            .filter { filter.check(it) }
    }
}

inline fun <T> Flow<T>.filerIf(addFilterIf: Boolean, crossinline \
condition: suspend (T) -> Boolean) =
    if (!addFilterIf) this else this.filter(condition)

suspend fun main() {
    val client = MarketClient()
    val repository = MarketRepository(client, backgroundScope = C\
oroutineScope(SupervisorJob()))
    val service = TradeService(repository)
    val filter = Or(
        listOf(
            And(listOf(TickerIs(tickers.take(1)), PrizeCondition(\
Ask, GreaterThan, 99f))),
            And(listOf(PrizeCondition(Spread, GreaterThan, 99f))),
        )
    )

    measureTimeMillis {
        service.observeUpdates(
            filter = filter,
            tickers = tickers.take(70)
        ).take(1_000)
            .collect { println(it) }
    }.let { println("Took $it") }
}
```

Beyond that, there are some changes that can be applied, but they would mean possible safety issues, that are are only partially possible to secure. Those possibilities include:

- Using `MutableMap` instead of `ConcurrentHashMap`.
- Making domain objects mutable.

Solution: Crime analysis

The problem is that the `readLines` function reads all lines from the file and stores them in memory. The solution is to use the `useLines` function, which reads lines from a file lazily.

The following solution uses the `groupBy` function to group elements, and it uses the `useLines` function to read lines from the file:

```
fun main() {
    measureTimeMillis {
        File("Crimes_-_2001_to_Present.csv")
            .useLines { lines ->
                lines.drop(1)
                    .map { Crime.parse(it) }
                    .groupBy { it.primaryType }
                    .mapValues { it.value.size }
                    .toList()
                    .sortedByDescending { (_, num) -> num }
                    .joinToString(separator = "\n") {
                        (type, num) -> "$num $type"
                    }
                    .let(::println)
                }
    }.let { println("Took $it") }
}
```

The following solution uses the `groupingBy` function to group elements, and it uses the `useLines` function to read lines from the file:

```
fun main() {
    measureTimeMillis {
        File("Crimes_-_2001_to_Present.csv")
            .useLines { lines ->
                lines.drop(1)
                    .map { Crime.parse(it) }
                    .groupingBy { it.primaryType }
                    .eachCount()
                    .toList()
                    .sortedByDescending { (_, num) -> num }
                    .joinToString(separator = "\n") {
                        (type, num) -> "$num $type"
                    }
                    .let(::println)
            }
```

```
    }.let { println("Took $it") }
}
```

On my machine (MacBook Pro M2, data from late 2023), this is what the execution time looks like:

- 6707 ms when using `groupBy`.
- 4455 ms when using `groupingBy`.