Instructor's Manual for Kotlin for Android App Development

by Peter Sommerhoff

Instructor's Manual

For the things we have to learn before we can do them, we learn by doing them.

Aristotle

This booklet can be used by instructors and students alike. Instructors can use it as in-class exercises or as assignments, and students can use these exercises for self-organized learning. **Example solutions for all exercises can be found at the end of the booklet and on GitHub**[1].

Of course, **these represent just *one* way to solve an exercise**. Many of the exercises in this booklet are fairly open so that many different solutions are possible. The central point is that you try solving them your way and gain programming experience with Kotlin.

You can solve these exercises in both Kotlin scratch files (`.kts` extension) and normal Kotlin files (`.kt` extension) with a main function. But remember that some declarations, like sealed classes, don't work well with scratch files.

The stars are an indicator for the difficulty of an exercise: more stars mean higher difficulty. Exercise additions marked as "bonus" are typically a little harder than the original exercise.

Exercises for Chapter 2

These exercises foster what you learned about Kotlin's basic language features like conditions, loops, functions, and exceptions.

[1] https://github.com/petersommerhoff/kotlin-for-android-app-development/tree/master/exercises/src/main/kotlin

Exercise 2.1 (⋆)

Write a program that produces the following output using loops (without using a standard library function such as `String.repeat`).

```
#
##
###
####
#####
######
```

Exercise 2.2 (⋆)

Write a program that builds upon exercise 2.1 by producing a 'christmas tree' output, where the last line should always have three hash symbols representing the trunk of the tree:

```
     #
    ###
   #####
  #######
 #########
    ###
```

Once this works, allow calling the program with a command-line argument that specifies the height of the christmas tree (the number of rows excluding the trunk). Command-line arguments can be accessed using the `args` parameter of a main function, and you can transform them to an integer using `toInt`.

The height of the tree should be at least 3. You can use Kotlin's `require` function for validations like this. In case of an exception, the program should print a helpful message to the user and proceed the program with a default tree height of 5.

Hint: There are three types of exceptions to handle here:

- Accessing the first element of args results in an `ArrayIndexOutOfBoundsException` if no command-line argument is passed to the program.
- The require function throws an `IllegalArgumentException` if the given predicate is `false`.
- Transforming the command-line argument to an integer causes a `NumberFormatException` if the given argument cannot be parsed to an `Int`. Note that this is a special form of the `IllegalArgumentException` so you must catch the

`NumberFormatException` first; otherwise, it will be handled by the **`catch`** block for `IllegalArgumentException`.

Tool Tip: You can set up the run in IntelliJ so that it passes command-line arguments by going to *Run*, *Edit Configurations*, and then under 'Program Arguments' specify the tree size (for instance 10).

Bonus: Split your program up into functions with well-defined responsibilities.

Exercise 2.3 (⋆)

Play the compiler! Can you beat Kotlin's compiler when it comes to type inference?

Which type does each of the following expressions have?

- `val rating = 9.7`
- `val bugsPerLineOfCode = 0.3f`
- `val moleculeCount = 13L`
- `const val QUIT_SYMBOL = 'q'`
- `val ratio = 17 / 5`
- `val offset = "Hello, World".length`
- `val punctuation = listOf("comma", "period", "semicolon", "dash")`
- ```
  val status = if(finishedWork()) {
    goHome()
     "Going home…"
  } else {
    writeCode()
     "Writing perfectly clean code…"
  }
  ```
- `val grades = 1..6`
- `val lowerCaseLatin = 'a'..'z'`
- ```
  fun findFile(fileName: String): String? { … }
  val fileLocation = findFile("secrets.txt") ?: ""
  ```
- ```
  val fileLocation = try {
    findFile("secrets.txt")
  } catch(e: FileNotFoundException) {
    null
  }
  ```
  ```

- `val fileLocation = findFile("secrets.txt") ?: throw FileNotFoundException("Could not find secret file")`
- `val hasAccess = user.age >= 21`
- ```
  val usersToPurchases = mapOf(
      "john" to mutableListOf("shoes", "football", "gloves"),
      "sarah" to mutableListOf("book", "laptop")
  )
  ```
- `val useless = null`

Exercise 2.4 (⋆)

Compare and contrast if and when conditions by deciding which is superior in the following use cases:

- You want to check if a user's age is in the range 1 to 10, 11 to 20, ..., 41 to 50, or above.
- You want to execute a certain piece of code if a variable is not `null`.
- You want to compare a given variable against each value from a finite set of distinct values, such as a set of several different states an automaton may be in.
- While processing a tree data structure, you want to switch between two different paths of execution depending on whether you're at an inner branch or a leaf of the tree.

Exercise 2.5 (⋆)

Next, examine the equivalence (or non-equivalence) of `for` loops and `while` loops by transforming the following loops to an equivalent loop of the other kind, if you can:

```
for (word in sentence.split(" ")) {
  print("$word. ")
}

for (gradeBoundary in 55..100 step 5) {
  println("Next grade: ${gradeBoundary - 5} - ${gradeBoundary}")
}

while (userInput != ":q") {
  print("> ")
  val userInput = readLine() ?: ""
}
```
```

What did you find? How does this result differ from languages like C++ or Java?

Exercise 2.6 (⋆)

Implement a number guessing game in which the user is prompted to enter a number between 1 and 100 until he or she guesses correctly. After every wrong guess, the user is told whether the guess was too high or too low.

Hints: To generate a random number between 1 and 100, you can use `java.util.Random.nextInt`; and to read a user's input from the command-line, Kotlin offers the `readLine` function.

Exercise 2.7 (⋆)

Write a function that outputs a multiplication table with a given range of rows and columns, then call it to print a table with rows from 1 to 5 and columns from 1 to 10.

The desired output has the form:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50

Tip: You can use `System.out.format("%-8d", value)` to output a number with 8 characters of space (and left-aligned due to the minus).

Exercise 2.8 (⋆)

Implement a function that calculates the Greatest Common Divisor (GCD) of two given integers. Use your GCD function in another function that computes the Least Common Multiple (LCM) of two given integers. Note that the LCM function may return a `Long`.

Bonus: Can you implement both functions as a 'one-liner' (using function shorthand syntax)?

Exercise 2.9 (⋆⋆)

Implement a function that accepts a sentence as input and applies the following transformation to each word:

- The first and last letter in each word remain where they are

- All other letters are shuffled randomly

Make sure to handle common punctuation characters in the given sentence. They should be ignored by the shuffling and still be at the same place in the result where they were in the original sentence.

Note that you should still be able to read most of the jumbled sentence as long as the first and last letters of each word stay in place because the human brain reads visually word by word, not character by character.

Hint: you can use Java's `Collections.shuffle` to shuffle a list of `Char`s for this task.

Exercise 2.10 (★★)

Given a string `s` consisting of only letters `[a–z]`, find the number of anagrams in all substrings of `s`.

Two strings are anagrams of each other if the letters in one can be repositioned to form the other string. For example, the string "peter" has two anagrammatic pairs of substrings, namely "et" and "te". The string "heyyy" has 4 anagrammatic pairs and the string "heyyyy" has 10.

Hint: Which properties must two strings fulfill to possibly be anagrams? Can you partition your substrings using one of these properties?

Exercise 2.11 (★★★)

Let *pd(n)* be the sum of all proper divisors of an integer *n*. Proper divisors include 1 but not *n*: for instance, the proper divisors of 36 are 1, 2, 3, 4, 6, 9, 12, and 18. So *pd*(36) = 55. If two different numbers a ≠ b fulfill *pd*(a) = b and *pd*(b) = a, then a and b are called an *amicable pair*. In other words, if the sum of the proper divisors of each number is equal to the other number.

Find all amicable pairs of numbers up to 2000 (there are only 2).

Bonus: Find all amicable pairs of numbers up to 10,000. Here, the main challenge becomes algorithmic efficiency to reduce computation time.

Super bonus: Is your algorithm fast enough for an upper bound of 100,000? How about 1,000,000?

Hint: Here's the expected output for amicable pairs of numbers up to 1,000,000:

```
[ (220, 284), (1184, 1210), (2620, 2924), (5020, 5564), (6232, 6368), (10744, 10856),
  (12285, 14595), (17296, 18416), (63020, 76084), (66928, 66992), (67095, 71145),
  (69615, 87633), (79750, 88730), (100485, 124155), (122265, 139815), (122368, 123152),
  (141664, 153176), (142310, 168730), (171856, 176336), (176272, 180848),
  (185368, 203432), (196724, 202444), (280540, 365084), (308620, 389924),
```

```
(319550, 430402), (356408, 399592), (437456, 455344), (469028, 486178),
(503056, 514736), (522405, 525915), (600392, 669688), (609928, 686072),
(624184, 691256), (635624, 712216), (643336, 652664), (667964, 783556),
(726104, 796696), (802725, 863835), (879712, 901424), (898216, 980984) ]
```

Exercises for Chapter 3

These exercises will help you master functional-style programming with lambda expressions, higher-order functions, and lazy evaluation.

Exercise 3.1 (⋆)

Implement a higher-order function iterate that accepts a function of type `(Double) -> Double` and an integer, and returns a function that applies the given function the given number of times.

Exercise 3.2 (⋆)

Given a map that maps people's names to a list of their children's names:

```
val parentsToChildren = mapOf(
    "Susan" to listOf("Kevin", "Katie"),
    "Marcus" to listOf("Claire"),
    "Kate" to emptyList(),
    "Mike" to listOf("Jake", "Helen", "John")
)
```

Use appropriate higher-order functions from Kotlin's standard library to find all people who have at least one child and whose names start with an `'M'`, and map them to strings of the form `"M's children are A and B and C"`. Thus, you should get the following output:

```
[Marcus's children are Claire, Mike's children are Jake and Helen and John]
```

Exercise 3.3 (⋆)

For each of the following tasks, which of Kotlin's scoping functions would be most adequate?

- Initializing an object (that requires additional initialization after the constructor call)
- Calling many methods on the same variable
- Performing a block of code only if a given variable is not `null`
- Immediate function application (directly running a given lambda expression)
- Intercepting a function chain for ancillary operations, such as logging, without breaking the function chain
- Transforming an explicit parameter to a receiver object

Bonus: Craft a code snippet that demonstrates each of these use cases with the appropriate scoping function.

Exercise 3.4 (★★)

Show (or at least reason) that the following identities hold for any `List<Int>`:

- `collection.take(5) + collection.drop(5) == collection`
- `collection.none { it > 5 } == collection.all { it <= 5 }`
- `collection.sorted() == collection.sortedDescending().reversed()`
- `collection.reduce { x, y -> x + y } ==`
  `collection.fold(0) { x, y -> x + y }`
- `collection.filter(predicate).filter(predicate) == collection.filter(predicate)`
- `collection.take(5).take(5) == collection.take(5)`

Which of these identities also hold for lists of any type, not just for lists of integers?

Exercise 3.5 (★)

The hailstone sequence is defined as follows:

- Choose any positive integer *n* as the start value.
- If *n* is even, the next element is *n/2*.
- If *n* is odd, the next element is *3n + 1*.

Implement the hailstone sequence as a lazy sequence in Kotlin using `generateSequence`.

Bonus: Implement the sequence again using `sequence`[2] or `buildSequence`. You will need to include the Kotlin coroutines dependency in your Gradle project for this:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:0.25.0"
```

Context: The Collatz Conjecture, phrased by Lothar Collatz in 1937, hypothesizes that every hailstone sequence—irrespective of starting value—converges to 1. Nobody was able to prove or disprove this conjecture yet (since 1937).

---

2 As of Kotlin 1.3, this replaces `buildSequence` but works the same way.

Exercise 3.6 (⋆⋆)

The *look-and-say sequence* can start with any positive single-digit number. The next element is then generated by spelling out the previous element. For instance, starting with 1, the next element would spell out 'one one' (or 11), and then 'two one' (or 21) because there are two ones. Here's how this look-and-say sequence would proceed:

'one', 'one one', 'two one(s)', 'one two, one one', 'one one, one two, two one(s)', ...

In other words (or rather numbers):

1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, 31131211131221, ...

For the implementation, you use the second representation, but you can still represent it as a sequence of strings in your code. Implement this mathematical sequence as a lazy sequence in Kotlin using `generateSequence`.

Bonus: Does your implementation work for arbitrary, but valid, initial values? If not, try to modify so that it supports any value between 1 and 9 as initial value. What do you recognize when starting a sequence with a value larger than 3?

Super bonus: Why can this sequence only go as high as 3 and never reach 4 (unless of course you start with a number larger than 3)?

Exercise 3.7 (⋆)

Write your own implementation of a `map` function. You can restrict it to work only with iterables of integers and only map to integers as well (generics are discussed in Chapter 4).

Use your custom map function to map a list of integers to their squares.

Bonus: Can you call your function on `IntRange`s as well? If not, raise its level of abstraction (without generics) to allow using it on ranges and several other types as well.

Exercise 3.8 (⋆)

You are given the data in the following (less than ideal) format:

```
val nutritionData = mapOf(
        "apple" to mapOf(
                "macros" to listOf("carbs" to 11.4, "protein" to 0.3, "fats" to 0.2),
                "vitamins" to listOf("vitamin c" to 4.6),
                "minerals" to listOf("calcium" to null, "zinc" to 0.1)
```

```
        ),
        "black beans" to mapOf(
                "macros" to listOf("carbs" to 15.0, "protein" to 8.9, "fats" to null),
                "vitamins" to listOf("vitamin c" to 0.0, "vitamin a" to 0.0018),
                "minerals" to listOf("calcium" to 27.0, "zinc" to 2.1)
        ),
        "walnuts" to mapOf(
                "macros" to listOf("carbs" to 13.7, "protein" to 15.2, "fats" to 65.2),
                "vitamins" to listOf("vitamin c" to 1.3, "vitamin a" to 0.006),
                "minerals" to listOf("calcium" to 98.0, "zinc" to 2.9)
        )
)
```

Extract the following data using functional-style programming:

- Which food is richest in calcium?
- What's the average fat content of the foods?
- Display the foods sorted by their protein content in descending order.
- Group the foods into those with at least 5% of fat content, and those with less than 5%.

Take care to handle nullables safely because it just so happens that the data contains some gaps.

Exercise 3.9 (★★)

Implement a lazy infinite sequence that produces all rational numbers less than or equal to 1, i.e., every fraction *p/q* where *p* and *q* are integers, *q* != 0 and *p* <= *q*.

Exercises for Chapter 4

This set of exercises lets you practice Kotlin's object-oriented concepts, namely the different kinds of classes and entities, using properties and methods, and language constructs special to Kotlin.

You'll write your own data structures, implement popular design patterns, and solve some algorithmic challenges as well.

Exercise 4.1 (⋆)

Your local university hired you to model a (very simplified) version of the university as an object-oriented system. The entities you should implement are `University`, `Professor`, `Student`, and `Course`.

These entities have the following properties and capabilities (which can be modeled as properties and methods):

- A university has a name and a founding year. It can hire new professors and enroll new students—which should add them to the university's list of professors or students, respectively.
- A professor has a name, an age, and a yearly salary. He or she can teach courses and test students in an oral exam—which gives the student a grade for that course.
- A student has a name, an age, a matriculation number, a grade for every course he or she completed in the past (by taking the exam), and a likelihood to succeed in their studies that starts out at 50%.

  A student can enroll in courses and take exams for courses. Additionally, he or she can study (which increases the likelihood of succeeding in their studies by 2% up to a maximum of 100%), or party (which has the adverse effect).
- Every course has a title and a description.

Make sure to use adequate data types (and kinds of classes).

This exercise is about creating basic classes with properties and methods. The degree to which you make this a usable app is left to you. Feel free to add additional entities as necessary to model your system.

Exercise 4.2 (⋆)

Model a class that represents a rectangle. It should have all properties that define the rectangle, a method that returns its area, and a method that allows to stretch the rectangle's width by a given factor.

Next, model another class that represents a square. Considering that inheritance is commonly explained as being an "is-a" relationship, how do the rectangle and square class relate to each other?

Use inheritance in your code and try to implement both classes adequately. Which problem occurs? How can you explain it?

Exercise 4.3 (⋆⋆)

Kotlin provides array lists and doubly linked lists based on Java's collections. In this exercise, you'll extend this by implementing a singly linked list. "Singly linked" means that each node only points to the next element but not to the previous. This data structure is illustrated in Figure 4.1.

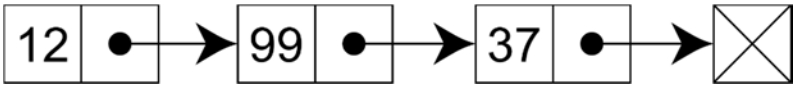

Figure 4.1 Singly linked list data structure; end of a list is signified by a *sentinel* element

First, implement a read-only variant of this data structure, called `SinglyLinkedList`, that accepts an arbitrary number of elements of any type in its constructor. It should offer methods to check whether the list contains an element, get an element by index and it should implement the interface `Iterable<E>` that allows using your `SinglyLinkedList` in for loops and also enables using `filter`, `map`, `reduce` and so forth.

Can you make use of the fact that your list is read-only in some way? You may want to look up Kotlin's `@UnsafeVariance` annotation, for instance to implement the `contains` method (which is a perfectly valid method for a read-only data structure).

Hint: Begin by modeling individual node as shown in Figure 4.1.

Bonus: What are the benefits and drawbacks of this data structure when compared to doubly linked lists and array lists?

Exercise 4.4 (⋆⋆)

You want to implement your own little RPG (role playing game). To prototype the idea, you decide to create a minimal version of the game. It should have two enemy types to start with: goblins and orcs. The player's hero is a warrior that may carry an armor and a sword. You may want to add other playable heros later (the same goes for enemy types).

Both enemies and heros have a name, a number of health points (HP), an attack stat (ATK) and a defense stat (DEF). The initial stats for all game characters are as follows:

- The warrior hero starts off with 50 HP, 10 ATK, and 5 DEF.
- Goblins have 16 HP, 9 ATK, and 3 DEF.
- Orcs are stronger opponents with 30 HP, 11 ATK, and 4 DEF.

Carrying a sword increases the hero's attack by 6, and wearing an armor increases his or her defense by 2. After every fight, there is a 25% chance that the hero finds a sword, a 15% change to find an armor, and a 10% chance to heal 10 HP. If the hero already carries a sword or armor, there's no benefit to finding another one.

The damage made by an attack is equal to the attacker's ATK stat minus the attackee's DEF stat.

This prototype will be a console game. When starting the program, the user should be able to choose whether to fight a goblin or an orc in each round. Beating a goblin earn the player 75 points and beating an orc is worth 200 points. The goal is to earn as many points as possible.

Before jumping into the code, think about how to model the entities in an object-oriented way. What are their relations? How can you make your software design extensible for new types of characters or equipment?

Exercise 4.5 (⋆)

Decide which modifier or language feature to use for each of these use cases:

- A test class' property that is re-initialized before each test case (for instance in a `@BeforeEach` method as in JUnit 5)
- Validating property changes and rejecting invalid changes
- Properties holding UI components that can only be initialized once the necessary context is ready (for instance initializations in `onCreate` on Android)
- Expensive objects that may not be used at runtime because they are not used on all execution paths

Bonus: Create a Kotlin scratch file that demonstrates each scenario listed above.

Exercise 4.6 (⋆)

In this exercise, you'll implement a tree data structure similar to the `BinaryTree<T>` from Chapter 4. However, it will be more general because a node may have any number of children. Implement this as a mutable data type in which each node has a value and a list of children.

Next, implement the Visitor Pattern[3]. This pattern allows traversing a tree structure while performing a given operation on each node. Implement both a depth-first and a breadth-first algorithm.

Depth-first means you first traverse child nodes of child nodes recursively, all the way down to the first leaves, and only then visit the next child of the root node. Breadth-first means you visit all nodes at one hierarchy level first, and only then descend to the next level (the child nodes of the child nodes). Figure 4.2 shows an example:

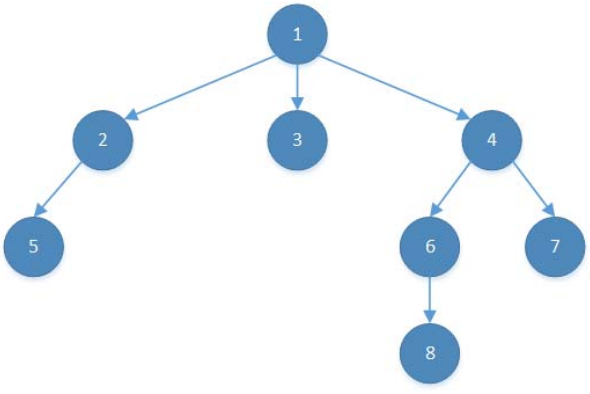

Figure 4.2 Tree data structure with numbered nodes

A depth-first traversal visits the nodes in the order 1, 2, 5, 3, 4, 6, 8, 7 whereas a breadth-first traversal goes 1, 2, 3, 4, 5, 6, 7, 8 with the numbering from Figure 4.2.

After implementing a visitor function, use it to compute the sum of all nodes in a tree of integers, and to transform all names in a tree of strings to lowercase.

Tip: Sealed classes don't play well with Kotlin script files so better use a normal `.kt` file.

[3] https://en.wikipedia.org/wiki/Visitor_pattern

Exercise 4.7 (⋆)

The Strategy Pattern[4] allows to swap a piece of functionality or implementation, traditionally by using inheritance or an interface. In this exercise, you'll combine object-oriented and functional concepts to implement this pattern without inheritance or a strategy interface.

To do so, imagine you want to implement a navigation system that offers two strategies to calculate routes: fastest and shortest. For the purpose of this exercise, let's say you have a class Navigator that calculates routes. How could you model this situation using the traditional Strategy Pattern that makes use of inheritance?

Do you see a way to adapt the traditional design using functional concepts? In any case, your design should be easily extensible for new navigation strategies.

(You don't need to implement the actual navigation logic for this exercise.)

[4] https://en.wikipedia.org/wiki/Strategy_pattern

Exercises for Chapter 5

These exercises will test your knowledge about interoperability issues when using Kotlin and Java together, from platform types to useful JVM annotations and best practices.

Exercise 5.1 (⋆⋆)

Play the compiler! For Java methods with the following signatures, which platform types does the Kotlin compiler infer for the return type when you call them from Kotlin?

- `double calculateAverage()`
- `Integer getAge()`
- `@NonNull List<String> titles()`
- `char[] toCharArray()`
- `Person[] getAttendees()`
- `Object get()`
- `<T> List<? extends Comparable<T>> fetchRatings()`
- `Map<@NonNull Address, @NonNull List<Employee>> locationsToEmployees()`

Exercise 5.2 (⋆)

Improve the following Java code to aid smooth interoperability with Kotlin:

```
class TreasureChest {
    public static final String sealed = "SEALED";
    public static final String open = "OPEN";
    String status = sealed;
    Object object = new Diamond();

    boolean is(Object object) { return this.object == object; }
    void open() { this.status = open; }
    void seal() { this.status = sealed; }
    String get() { return this.status; }

    class Diamond {}
}
```

Exercise 5.3 (⋆)

Which of Kotlin's JVM interoperability annotations would be the right choice for the following scenarios?

- You want to prevent the generation of getters and setters for a property and instead allow direct field access.
- You want to prevent class names such as `CandyKt` or `CookieKt` when using your Kotlin files from Java.
- You want to be able to call a companion object method directly on its containing class from Java.
- You want to at least have some of the overloaded options to call a Kotlin function with optional arguments from Java.

Exercise 5.4 (⋆)

Which of the properties and methods in the following Kotlin code snippets compile to static members in the Java bytecode? Use the *Show Kotlin Bytecode* action and the decompiler tool in IntelliJ to verify your answers.

- ```
  class Constants {
    @JvmField val PI = BigDecimal(3.14159265358979323846264433)
  }
  ```
- ```
  object MainPresenter {
    @JvmStatic fun notifyUser(message: String) { … }
  }
  ```
- ```
  object RobotFactory {
    val defaultRobot = Robot()
  }
  ```
- ```
  object Constants {
    const val PI = 3.14159265358979323
  }
  ```
- ```
  class Person {
    companion object PersonBuilder {
      fun build(): Person = Person()
    }
  }
  ```
  ```

- ```
  class MainUi {
    lateinit var btnOk: Button
  }
  ```

Exercise 5.5 (⋆)

What would be a good return type for functions such as `System.exit`, `fail`, or `error` that cause the program flow to throw an exception or terminate altogether? What does the return type signify and in which ways does the Kotlin compiler make use this?

Exercise 5.6 (⋆⋆)

Let's say you come across the following Kotlin file in a mixed-language project. Thus, the declarations from this file should be usable from Java. Can you modify the code to facilitate working with it from both Java and Kotlin?

```
// File name "Trees.kt"

// Elements in this tree may change but its structure remains.
// Modifying this class is not part of the exercise but feel free to do so.
sealed class Tree<T>
data class Leaf<T>(var value: T) : Tree<T>()
data class Branch<T>(var value: T, val children: List<Tree<T>>) : Tree<T>() {
  init { require(children.isNotEmpty()) }
}

fun <T> applyToAllNodes(transform: (T) -> T, tree: Tree<T>) {
  when (tree) {
    is Leaf -> tree.value = transform(tree.value)  // Modifies the tree in-place
    is Branch -> {
      tree.value = transform(tree.value)
      tree.children.forEach { subtree -> applyToAllNodes(transform, subtree) }
    }
  }
}
```
```

Exercises for Chapter 6

These exercises will let you gain experience with concurrency in Kotlin, from general concurrency issues to coroutines, asynchrony, parallelism, and the actor model.

Exercise 6.1 (⋆)

This exercise lets you explore a basic concurrency scenario and its pitfalls. To this end, create a mutable integer initialized with zero. Then launch three coroutines, each of which adds 1 to the shared mutable state a million times. After all coroutines finish, print out the final value of the variable.

Let this program run multiple times. What do you see? How can you explain it? Do you know the name of this phenomenon?

Bonus: Try to resolve the problem using Kotlin's `Mutex` class from the coroutines dependency.

Exercise 6.2 (⋆⋆)

In this exercise, let's fetch random Chuck Norris jokes from a Chuck Norris API.[5] First, implement a program that fetches a random joke and prints it to the console.

Expand your program to fetch 3 jokes in parallel—each without blocking the other—every time the user hits ENTER. Quit the program when the user enters "q".

Bonus: Add a timeout of 1 second to your asynchronous calls so that the user is shown a message if no joke could be retrieved within that time. (You may want to set the timeout to around 10ms for testing purposes).

Hint: Take a look at the class `java.net.URL` and at Kotlin's extensions for reading content from it.

Exercise 6.3 (⋆⋆)

Suspending functions may feel magic at first but they are not. This exercise is intended to let you explore again how to compiler treats them differently from normal functions.

To do so, first create normal function and explore the (decompiled) Java bytecode, then create a suspending function with the same parameters and return values and compare the (decompiled) Java bytecode. Add a few suspension points to the suspending function as well.

[5] http://www.icndb.com/api/

Which similarities and differences can you spot? How and why does the compiler generate this kind of bytecode for suspending functions? What are continuations and how are they related to coroutines and suspending functions?

Exercise 6.4 (★★★)

Actors are a promising way to model concurrent systems by avoiding shared state from the get-go. This exercise aims to let you implement your first own actors.

First, implement 3 producers that send a random integer between 1 and 100 (roughly) every 300ms, 500ms, and 1000ms, respectively. Next, create an actor that handles values coming from all these producers every 200ms and always selects the first value that is available—but in a fair way, meaning that when multiple values are available at the same time, the actor should not always default to the first producer. Use only rendezvous channels for communication between the actors.

For the purposes of this exercise, it suffices for the actor to print the received values to the console.

Exercise 6.5 (★★)

The Miller's dog Ruffy likes to run out and terrorize the neighbors' dogs while his owners don't look. So he was attached a GPS tag that sends his location every second, and his owners are warned as soon as his distance from the house's center reaches 20m.

Implement this scenario using a producer that emits GPS locations which get increasingly further away from a fixed house location (choose any GPS location for this). A separate actor should process the location data and emit a warning when the distance reaches the threshold (a warning to the console suffices for this exercise).

Note: Besides from using coroutines, for this exercise you'll have to find an algorithm that is adequate to calculate distances between two GPS coordinates that are fairly close to each other.

Exercises for Chapter 9

In these exercises, you can implement your own Kotlin DSL and familiarize yourself with using the existing Kotlin Gradle DSL.

Exercise 9.1 (★★)

Write a DSL, including all classes required, that allows users to easily create recipes. A recipe consists of at least two ingredients and may have an additional description text. Every ingredient is composed of the name of the food and an amount.

Can you make this DSL so easy to use that a chef would be able to write, modify, and validate recipes in your DSL (after minimal training)?

Exercise 9.2 (★★)

Migrate the Kudoo app to the Gradle Kotlin DSL, including all build scripts and the `settings.gradle` file.

Note: At the time of writing, Android Studio still has trouble recognizing the Gradle Kotlin DSL immediately. You may also have to delete your local gradle build cache if you run into an error with that. Also make sure to update to the newest version of Android Studio.

Solutions

These solutions are also available on GitHub.[6] I highly recommend to use this GitHub repository as your primary resource for the solutions – they're up-to-date, searchable, editable, and runnable.

Please note that most solutions shown here use Kotlin scratch files (`.kts` file extension).

Solution for Exercise 2.1

Solution 2.1 Printing More and More "Hashtags"

```
for (count in 1..6) {
  for (i in 1..count) {
    print("#")
  }
  println()
}
```

Solution for Exercise 2.2

Solution 2.2 Printing Christmas Trees

```
// Reads and parses command-line argument, handles possible exceptions
val rows = try {
  require(args[0].toInt() >= 3)
  args[0].toInt()
} catch (e: IllegalArgumentException) {
  System.err.println("Please use a tree size of at least 3. Using default size of
5...")
  5
} catch (e: ArrayIndexOutOfBoundsException) {
  System.err.println("Please specify the tree size as command-line parameter. Using
default size of 5...")
  5
} catch (e: NumberFormatException) {
  System.err.println("Command-line argument must be an integer. Using default size of
5...")
  5
```

[6] https://github.com/petersommerhoff/kotlin-for-android-app-development/tree/master/exercises/src/main/kotlin

```
}

// Extracts relevant data
val treeWidth = (rows * 2) - 1
val treeMiddle = treeWidth / 2

// Prints tree 'leaves'
for (row in 0 until rows) {
  for (i in 1..treeMiddle - row) print(" ")
  for (i in 1..(2 * row + 1)) print("#")
  println()
}

// Prints tree trunk
for (i in 1 until treeMiddle) print(" ")
print("###")
```

...and a more modularized version for the bonus task:

Solution 2.2 (Bonus) Printing Christmas Trees

```
// Uses local helper functions
fun readTreeHeightFromArgs(fallbackHeight: Int): Int {
  return try {
    args[0].toInt().also { require(it > 3) }  // Parses and validates command-line
argument
  } catch (e: NumberFormatException) {
    System.err.println("Command-line argument must be an integer. Using default size of
$fallbackHeight...")
    fallbackHeight  // Uses default value in case of exception
  } catch (e: IllegalArgumentException) {
    System.err.println("Please use a tree size of at least 3. Using default size of
$fallbackHeight...")
    fallbackHeight
  } catch (e: ArrayIndexOutOfBoundsException) {
    System.err.println("Please specify the tree size as command-line parameter. Using
default size of $fallbackHeight...")
    fallbackHeight
  }
}

// Extracts relevant data
```

```
val treeHeight = readTreeHeightFromArgs(fallbackHeight = 5)
val treeWidth = (treeHeight * 2) - 1
val treeMiddle = treeWidth / 2  // The center on the horizontal axis

// Uses local helper functions
fun printTreeRow(row: Int) {
  for (i in 1..treeMiddle - row) print(" ")
  for (i in 1..(2 * row + 1)) print("#")
  println()
}

fun printTreeLeaves() {  // Note that these helpers don't need parameters because they
can access variables from enclosing scope
  for (row in 0 until treeHeight) {
    printTreeRow(row)
  }
}

fun printTreeTrunk() = printTreeRow(1)  // Trunk is same as second row of tree leaves

fun printTree() {
  printTreeLeaves()
  printTreeTrunk()
}

// Actual program
printTree()
```

Solution for Exercise 2.3

The Kotlin compiler will infer the following types for the expressions:

```
rating:             Double

bugsPerLineOfCode:  Float

moleculeCount:      Long

QUIT_SYMBOL:        Char

ratio:              Int (performs integer division)
```

```
offset:             Int

punctuation:        List<String>

status:             String

grades:             IntRange

lowerCaseLatin:     CharRange

fileLocation:       String

fileLocation:       String?

fileLocation:       String

hasAccess:          Boolean (assuming there's a user object with an age property of
type Int)

usersToPurchases:   Map<String, MutableList<String>>

useless:            Nothing?
```

Solution for Exercise 2.4

Solution 2.4 **Conditions**

```
/*
 * 1) You want to check if a user's age is in the range 1 to 10, 11 to 20, ..., 41 to
50, or above.
 *
 * Here, a when expression without argument allows for the most concise and readable
solution:
 */
val age = 42
when {
  age < 1 -> throw IllegalArgumentException("Value must be at least 1")
  age in 1..10 -> println("1 to 10")
  age in 11..20 -> println("11 to 20")
  age in 21..30 -> println("21 to 30")
  age in 31..40 -> println("31 to 40")
  age in 41..50 -> println("41 to 50")
  else -> println("Over 50")
```

```
}

/*
 * 2) You want to execute a certain piece of code if a variable is not null.
 *
 * This is a common use case for a simple if statement:
 */
val response = "200 OK"
if (response != null) {
  println("Received response: $response")
}

/*
 * 3) You want to compare a given variable against each value from a finite set of
distinct values,
 * such as a set of several different states an automaton may be in.
 *
 * This is a prime example for using when statements:
 *
 * (Enums are covered later in Chapter 4 and improve representing states like the
following)
 */
val states = arrayOf("IDLE", "RUNNING", "COMPLETED")
val currentState = "RUNNING"

when (currentState) {
  "IDLE" -> println("Process is idle")
  "RUNNING" -> println("Process is running")
  "COMPLETED" -> println("Process has completed")
}

/*
 * 4) While processing a tree data structure, you want to switch between two different
paths of
 * execution depending on whether you're at an inner branch or a leaf of the tree.
 *
 * Although you could do this with a when statement, choosing between two paths is a
perfect use
 * case for a simple if-else statement:
 */
val isLeaf = false
if (isLeaf) {
  println("Reached a leaf, will go back up the tree...")
```

```
} else {
  println("At an inner node, will continue descending until a leaf is reached...")
}
```

Solution for Exercise 2.5

Solution 2.5 **Loops**

```
// 1)
val sentence = "Why are iPhone chargers not called apple juice?"
val words = sentence.split(" ")
var index = 0
while (index < words.size) {
  print("${words[index]}. ")
  index++
}

// 2)
val steps = 55..100 step 5
index = 0  // Don't usually reuse the same index in another loop!
while (true) {
  val gradeBoundary = steps.elementAtOrNull(index) ?: break
  println("Next grade: ${gradeBoundary - 5} - ${gradeBoundary}")
  index++
}

// 3)
/*
 * This one cannot be expressed as a for loop in Kotlin.
 *
 * In contrast to languages like C++ and Java, while and for loops in Kotlin are not
fundamentally
 * equivalent -- while loops are more expressive. More specifically, for loops in
Kotlin are
 * equivalent to for-each loops from other languages and only allow iterating over
anything that
 * provides an iterator().
 */
```

Solution for Exercise 2.6

Solution 2.6 **Number Guessing Game**

```
val numberToGuess = Random().nextInt(100) + 1
var userGuess = -1
println("Guess the number between 1 and 100!")

do {
  print("> ")
  userGuess = try {
    readLine()!!.toInt()  // `readLine` cannot return null when using the command-line
  } catch (e: Exception) {
    println("Not a valid guess, please enter a number between 1 and 100")
    continue
  }
  when {
    userGuess < numberToGuess -> println("Too low!")
    userGuess == numberToGuess -> println("Congratulations, $userGuess is the correct
number!")
    userGuess > numberToGuess -> println("Too high!")
  }
} while (userGuess != numberToGuess)
```

Solution for Exercise 2.7

Solution 2.7 **Multiplication Table**

```
fun printMultiplicationTable(rows: IntRange, columns: IntRange) {
  for (row in rows) {
    for (col in columns) {
      System.out.format("%-8d", row * col)  // Left-aligned output with 8 chars space
per column
    }
    println()
  }
}
```

```
printMultiplicationTable(rows = 1..5, columns = 1..10)
```

Solution for Exercise 2.8

Solution 2.8 GCD & LCM

```
/*
 * - If one of the numbers is zero, the GCD is defined to be the other number
 * - Otherwise, you can find the GCD using recursion (calling the function inside
itself)
 */
fun greatestCommonDivisor(a: Int, b: Int): Int {
  if (a == 0) {
    return b
  }
  return greatestCommonDivisor(b % a, a)
}

/*
 * The implementation above can easily be translated into a single if-expression
 */
fun greatestCommonDivisorShort(a: Int, b: Int): Int =
    if (a == 0) b else greatestCommonDivisorShort(b % a, a)

/*
 * As a bonus sidenote for the interested, the implementation above is actually *tail-
recursive*
 * because the only recursive call appears as the last statement inside the function
body. Kotlin
 * allows you to mark such function as `tailrec` to let the compiler do additional
optimizations.
 */
tailrec fun greatestCommonDivisorTailrec(a: Int, b: Int): Int =
    if (a == 0) b else greatestCommonDivisorTailrec(b % a, a)

/*
 * For two numbers a and b, it holds that gcd(a,b) * lcm(a,b) = a * b
 * Therefore, lcm(a,b) = a*b / gcd(a,b)
 */
```

```
fun leastCommonMultiple(a: Int, b: Int): Long = a.toLong() * b /
greatestCommonDivisor(a, b)

/*
 * Sanity-checks the two functions using a few sample inputs (this is not a proper
test!)
 */
for (a in arrayOf(6, 8, 15, 27, 34, 120)) {
  for (b in arrayOf(2, 3, 9, 17)) {
    println("GCD($a, $b) = ${greatestCommonDivisorShort(a, b)}")
    println("LCM($a, $b) = ${leastCommonMultiple(a, b)}")
  }
}
```

Solution for Exercise 2.9

Solution 2.9 Shuffled Words

```
// Shuffles all chars in the given string
fun shuffle(str: String): String {
  val chars = str.toList()
  Collections.shuffle(chars)
  return chars.joinToString(separator = "")
}

/*
 * Shuffles only the inner characters of a string, meaning all but the first and last.
 *
 * Note: With generic functions (Chapter 4), you could write a much more general and
thus reusable function
 * that shuffles a given range of some given Iterable<T>.
 */
fun shuffleInnerLetters(word: String): String {
  if (word.length <= 2) {
    return word
  }
  val innerLetters = word.substring(1, word.length - 1)
  val shuffledInner = shuffle(innerLetters)
  val firstLetter = word.first()
```

```
  val lastLetter = word.last()
  return "$firstLetter$shuffledInner$lastLetter"
}

// The list is certainly not complete
val punctuation = listOf('.', ',', ':', '-', '!', '?', ';')

fun memorizePunctuation(str: String): Map<Int, Char> {
  val indexToChar = mutableMapOf<Int, Char>()
  for ((index, char) in str.withIndex()) {
    if (char in punctuation) {
      indexToChar.put(index, char)
    }
  }
  return indexToChar
}

fun String.removePunctuation() = this.filterNot { it in punctuation }

fun shuffleSentence(sentence: String): String {
  val sentencePunctuation = memorizePunctuation(sentence)
  val words = sentence.removePunctuation().split(" ")
  val shuffledWords = mutableListOf<String>()
  for (word in words) {
    shuffledWords += shuffleInnerLetters(word)  // In Chapter 3, you'll learn how to do
transformations like these more easily
  }

  val result = StringBuilder()
  result.append(shuffledWords.joinToString(separator = " "))
  for ((index, punc) in sentencePunctuation) {
    result.insert(index, punc)
  }
  return result.toString()
}

// Simple test run
println(shuffleSentence("You should still be able to read this without too much
trouble. Especially once your brain gets used to this, you can read this almost like a
normal sentence. This works well for simple texts but gets harder for sentences with
more complicated words."))
```

Solution for Exercise 2.10

Solution 2.10 Anagrams of Substrings

```
import kotlin.collections.*

fun anagramsOfSubstrings(str: String): Int {
  var anagramCount = 0
  val lengthToSubstrings = getSubstringsByLength(str)
  for (substrings in lengthToSubstrings.values) {
    for ((i, s1) in substrings.withIndex()) {
      for (s2 in substrings.elementsAfter(i)) {
        if (s1.isAnagramOf(s2)) anagramCount++
      }
    }
  }
  return anagramCount
}

private fun getSubstringsByLength(str: String): Map<Int, List<String>> {
  val lengthToSubstrings = mutableMapOf<Int, List<String>>()
  for (length in 1..(str.length - 1)) {
    lengthToSubstrings[length] = str.windowed(size = length, step = 1)
  }

  return lengthToSubstrings
}

private fun <T> List<T>.elementsAfter(index: Int) = this.subList(index + 1, this.size)

private fun String.isAnagramOf(other: String): Boolean = (letterFrequencies(this) ==
letterFrequencies(other))

private fun letterFrequencies(str: String): Map<Char, Int> =
    str.toCharArray().groupBy { it }.map { it.key to it.value.size }.toMap()

fun main(args: Array<String>) {
  println(anagramsOfSubstrings("abba"))    // 4
  println(anagramsOfSubstrings("abcd"))    // 0
  println(anagramsOfSubstrings("kkkk"))    // 10
  println(anagramsOfSubstrings("abab"))    // 5
  println(anagramsOfSubstrings("peter"))   // 2
  println(anagramsOfSubstrings("heyyy"))   // 4
```

```
  println(anagramsOfSubstrings("heyyyy"))  // 10
}
```

Solution for Exercise 2.11

Solution 2.x **Amicable Pairs**

```
// Inefficient solution that suffices for amicable pairs of numbers up to 2000
fun findAmicablePairs(upperBound: Int): List<Pair<Int, Int>> {
  val amicablePairs = mutableListOf<Pair<Int, Int>>()
  for (n in 1..upperBound) {
    val sumDivisorsOfN = properDivisors(n).sum()
    for (m in n+1..upperBound) {
      val sumDivisorsOfM = properDivisors(m).sum()
      if (sumDivisorsOfN == m && sumDivisorsOfM == n) amicablePairs.add(n to m)
    }
  }
  return amicablePairs
}

private fun properDivisors(n: Int): List<Int> {
  if (n == 0) return emptyList()

  val properDivisors = mutableListOf(1)  // 1 is always a proper divisor

  // It suffices to iterate to sqrt(n) and then add n/j to the divisors (if n/j != j)
  for (j in 2..sqrt(n.toDouble()).toInt()) {
    if (n % j == 0) {
      properDivisors.add(j)
      if (n/j != j) {
        properDivisors.add(n/j)
      }
    }
  }
  return properDivisors
}

// More efficient solution that can find amicable pairs of numbers up to 100,000
quickly and takes only a few seconds
// for an upper bound of 1,000,000
fun findAmicablePairsFast(upperBound: Int): List<Pair<Int, Int>> {
```

```
  val amicablePairs = mutableListOf<Pair<Int, Int>>()
  val sumsOfDivisors = IntArray(upperBound + 1) { properDivisors(it).sum() }  // Cache
the sums of divisors
  for (n in 1..upperBound) {
    val m = sumsOfDivisors[n]
    if (m < upperBound && m > n && sumsOfDivisors[m] == n) {
      amicablePairs.add(n to m)
    }
  }
  return amicablePairs
}

fun main(args: Array<String>) {
  println(findAmicablePairs(2000))
  println(findAmicablePairsFast(1_000_000))
}
```

Solution for Exercise 3.1

Solution 3.1 **Repeating a Function**

```
fun iterate(repetitions: Int, func: (Double) -> Double): (Double) -> Double = { d:
Double ->
  var result = d
  for (i in 1..repetitions) {
    result = func(result)
  }
  result  // Note that this is not the return value of `iterate` -- it's the return
value of the returned lambda
}

val plusThree = iterate(3) { it + 1.0 }
println(plusThree(3.1415))  // Note machine precision issue with floating point numbers

val toThePowerOf8 = iterate(3) { it.pow(2) }
println(toThePowerOf8(2.0))
```

Solution for Exercise 3.2

Solution 3.2 **Common Higher-Order Functions**

```
val parentsToChildren = mapOf(
    "Susan" to listOf("Kevin", "Katie"),
    "Marcus" to listOf("Claire"),
    "Kate" to emptyList(),
    "Mike" to listOf("Jake", "Helen", "John")
)

val result = parentsToChildren.filter { it.value.isNotEmpty() }
    .filter { it.key.toUpperCase().startsWith("M") }
    .map { "${it.key}'s children are ${it.value.joinToString(" and ")}" }

println(result)
```

Solution for Exercise 3.3

Solution 3.3 **Scoping Functions**

```
// 1) Initializing an object (that requires additional initialization after the
constructor call)
// -> This is the ideal use case for `apply`
val button = JButton("OK").apply {
  size = Dimension(200, 60)
  font = Font.getFont("Segoe UI")
  addActionListener { }
}

// 2) Calling many methods on the same variable
// -> This is the most common use case for `with` (which returns the value in the last
line)
// Note: You could use `apply` here as well but we use it rather for object
initialization and
// prefer `with` for using builders and returning the built value
val question = with(StringBuilder()) {
  append("Are you ")
  append(button.text)
  append("?")
  toString()
}

// 3) Performing a block of code only if a given variable is not null
// -> The most typical use case for calling `let` with a safe call
val font: Font? = button.font
font?.let {
  println(it.fontName)
}

// 4) Immediate function application (directly running a given lambda expression)
// -> This can be achieved most easily using `run`
run {
  val scoped = "Invisible to the outside"
  println(scoped)
}

// 5) Intercepting a function chain for ancillary operations, such as logging, without
breaking the
// function chain
// -> We use `also` for ancillary operations like this
val vowelCount = question.filter { it in listOf('a', 'e', 'i', 'o', 'u') }
    .also { println("Found vowels: $it") }
```

```
    .count()

// 6) Transforming an explicit parameter to a receiver object
// -> This is achieved using `run` on the parameter, e.g., to avoid repeating long
variable names
fun validateCredentials(userCredentialsToValidate: Credentials) =
userCredentialsToValidate.run {
  // `this.name`, or just `name`, now refers to the name property of the credentials
object
  name.isNotBlank()
  password.length >= 8
}


class Credentials(val name: String, val password: String)
```

Solution for Exercise 3.4

Solution 3.4 **Working with Collections**

```
val collection = listOf(0, 1, 3, 5, 7, 8, 9)

// 1) First five elements plus everything *but* the first five elements is again the
whole list.
println(collection.take(5) + collection.drop(5) == collection)

// 2) None of the elements in the list are greater than 5 if and only of all are lower
or equal to 5.
println(collection.none { it > 5 } == collection.all { it <= 5 })

// 3) Ascending order is exactly descending order reversed.
println(collection.sorted() == collection.sortedDescending().reversed())

// 4) Reducing all elements with + is the same as folding with an additional zero
because zero does
// not add to the overall sum.
println(collection.reduce { x, y -> x + y } == collection.fold(0) { x, y -> x + y })

// 5) Filtering for the *same* predicate more than once has no use.
// A function f with this property is called *idempotent*: f(f(x)) = f(x)
// From this it follows that f(f(...(f(x))...) = f(x) by repeated simplification using
```

```
f(f(x)) = f(x)
val predicate: (Int) -> Boolean = { it > 5 }
println(collection.filter(predicate).filter(predicate) == collection.filter(predicate))

// 6) Taking the first five elements of the first five elements still gives you the
first five
// elements. This is another example of an idempotent function (if you pass the same
argument).
println(collection.take(5).take(5) == collection.take(5))

// Which of these identities also hold for lists of any type, not just for lists of
integers?
// 1) holds for any list
// 2) only works for lists of integers due to the predicates
// 3) holds for any list
// 4) only works for lists of integers due to the 0 argument in fold but the general
concept applies
// as long as the operation (here +) is commutative and fold's first argument (here 0)
is the
// neutral element w.r.t. to the operation. For example, 1 as initial value and * as
operator works.
// 5) holds for any list
// 6) holds for any list
```

Solution for Exercise 3.5

Solution 3.5 **Hailstone Sequence**

```
fun Int.isEven() = this % 2 == 0

fun generateHailstoneSequence(initial: Int) =
    if (initial <= 0)
      throw IllegalArgumentException("Sequence must start with a positive integer")
    else generateSequence(initial) { n ->
      if (n == 1) return@generateSequence null  // Makes the sequence terminate
      if (n.isEven()) n / 2 else 3 * n + 1  // Defines value of next element via lambda
return value
    }

fun buildHailstoneSequence(initial: Int) =
    if (initial <= 0)
```

```
        throw IllegalArgumentException("Sequence must start with a positive integer")
    else sequence {
        // Use `sequence` instead in Kotlin 1.3
        var next = initial
        while (true) {
            yield(next)
            if (next == 1) return@sequence
            next = if (next.isEven()) next / 2 else 3 * next + 1
        }
    }

// Simple test runs for `generateSequence`
val hailstoneOfSeven = generateHailstoneSequence(7)
val hailstoneOf1337 = generateHailstoneSequence(1337)

hailstoneOfSeven.forEach { print("$it, ") }
println()
hailstoneOf1337.forEach { print("$it, ") }

println("¥n")

// Simple test runs for `buildSequence`
val builtHailstoneOfSeven = buildHailstoneSequence(7)
val builtHailstoneOf1337 = buildHailstoneSequence(1337)

builtHailstoneOfSeven.forEach { print("$it, ") }
println()
builtHailstoneOf1337.forEach { print("$it, ") }
```

Solution for Exercise 3.6

Solution 3.6 **Look-and-Say Sequence**

```
fun generateLookAndSaySequence(initial: Int): Sequence<String> {
    // This is an alternative to validation as done in Exercise 3.5
    require(initial in 1..9) { "Sequence must start with a positive integer" }

    return generateSequence(initial.toString()) {
        val NONE = '-'
        val chars = it.toCharArray()
        val output = StringBuilder()
```

```
    var currentNumber = NONE
    var counter = 0

    for (char in chars) {
      if (char == currentNumber) {
        counter++
      } else {
        if (currentNumber != NONE) output.append("$counter$currentNumber")
        currentNumber = char
        counter = 1
      }
    }
    output.append("$counter$currentNumber")
    output.toString()
  }
}

// Simple test run
val lookAndSay = generateLookAndSaySequence(1)
lookAndSay.take(10).forEach(::println)

println()

val lookAndSayWith9 = generateLookAndSaySequence(9)  // Sequence elements always end
with "19"
lookAndSayWith9.take(10).forEach(::println)

/*
 * Super bonus: with initial value in 1..3, the sequence can never contains a 4.
 *
 * This can easily be proved by contradiction:
 * - Let's assume there was an element in the sequence with a 4.
 * - Without loss of generality, we can assume that the 4 is followed by a 1 in that
element.
 * - Thus, the sequence element containing "41" signifies that its predecessor
contained the substring "1111".
 * - But this contradicts the structure of the sequence because the substring "1111"
would have been "21" instead
 *   ("one one followed by one one are two ones).
 * - Thus, the original element containing 4 cannot exist.
 */
```

Solution for Exercise 3.7

Solution 3.7 Your Own Higher-Order `map` Function

```
// Usually, you would use the signature fun <T, R> Iterable<T>.map(f: (T) -> R) but
this is covered in Chapter 4.

// Uses the abstract interface Iterable so that it works with all collections, ranges
and more
fun Iterable<Int>.mapTo(transform: (Int) -> Int): List<Int> {
  val result = mutableListOf<Int>()
  for (element in this) {
    result.add(transform(element))
  }

  return result.toList()
}

// Simple test runs
val numbers = listOf(2, 7, 6, 11)
val mappedNumbers = numbers.mapTo { it * it }
println(mappedNumbers)

val range = 5..10
val mappedRange = range.mapTo { it + 5 }
println(mappedRange)
```

Solution for Exercise 3.8

Solution 3.8 Working with Nested Collections

```
val nutritionData = mapOf(
    "apple" to mapOf(
        "macros" to mapOf("carbs" to 11.4, "protein" to 0.3, "fats" to 0.2),
        "vitamins" to mapOf("vitamin c" to 4.6),
        "minerals" to mapOf("calcium" to null, "zinc" to 0.1)
    ),
    "black beans" to mapOf(
        "macros" to mapOf("carbs" to 15.0, "protein" to 8.9, "fats" to null),
        "vitamins" to mapOf("vitamin c" to 0.0, "vitamin a" to 0.0018),
        "minerals" to mapOf("calcium" to 27.0, "zinc" to 2.1)
```

```
    ),
    "walnuts" to mapOf(
        "macros" to mapOf("carbs" to 13.7, "protein" to 15.2, "fats" to 65.2),
        "vitamins" to mapOf("vitamin c" to 1.3, "vitamin a" to 0.006),
        "minerals" to mapOf("calcium" to 98.0, "zinc" to 2.9)
    )
)

// 1) Richest in calcium
val richestInCalcium = nutritionData
    .maxBy { it.value["minerals"]?.get("calcium") ?: -1.0 }?.key ?: "No food found"

println("Richest in calcium: $richestInCalcium")

// 2) Average fat content
val averageFat = nutritionData.filter { it.value["macros"]?.get("fats") != null }
    .mapValues { it.value["macros"]?.get("fats") }
    .values
    .filterNotNull()
    .average()

println("Average fat content: $averageFat")

// 3) Sorted by protein content descending
val proteinSources = nutritionData.entries.sortedByDescending {
  it.value["macros"]?.get("protein") ?: 0.0
}.map { it.key.capitalize() }

println("Best protein sources:")
proteinSources.forEach(::println)

// 4) Grouping by fat content
val groupedByFat = nutritionData.entries.groupBy {
  val fat = it.value["macros"]?.get("fats") ?: 0.0
  if (fat < 5.0) "Low fat" else "High fat"  // Specifies group names
}.mapValues { it.value.map { foodToData -> foodToData.key.capitalize() } }

println("Grouped by fat content: $groupedByFat")
```

Solution for Exercise 3.9

Solution 3.9 Generating All Rational Numbers

```
/*
 * To generate these rational numbers, we cannot simply start going through 1, 2, 3,
4, ... because
 * we would never "arrive" at any non-integer numbers. A similar statement goes for any
sequence
 * where only the denominator increases first.
 *
 * The idea is to generate the sequence of elements 1, 1/2, 2, 1/3, 2/3, 3, 1/4, 2/4,
3/4, 4, ...
 *
 * You can visualize this in an infinite triangle where we go through the bottom-left
to top-right
 * diagonals:
 *
 * 1   2   3   4   5
 * 1/2 2/3 3/4 4/5
 * 1/3 2/4 3/5
 * 1/4 2/5
 * 1/5
 *
 * This ensures that every rational numbers leq 1 is theoretically generated at some
point.
 */



fun generateRationals(): Sequence<Double> {
  var denominator = 1
  var numerator = 1
  return sequence {
    while (true) {
      if (numerator == denominator) {  // Switches to next diagonal
        yield(denominator++.toDouble())  // Yields denominator and increments it for
next iteration via ++
        numerator = 1
      } else {  // Iterates through current diagonal
        yield(numerator++.toDouble() / denominator)  // Increments numerator for next
iteration via ++
      }
    }
  }
}
```

```
// Simple test run
val rationals = generateRationals()
rationals.take(20).forEach(::println)
```

Solution for Exercise 4.1

The design in this solution uses 5 classes and 6 different files:

Solution 4.1 **University.kt**

```
class University(val name: String, val foundingYear: Int) {

  val professors = mutableListOf<Professor>()
  val students = mutableListOf<Student>()

  fun hire(prof: Professor) {
    professors.add(prof)
    println("Hired Professor ${prof.name} at $name...")
  }

  fun enroll(newStudent: Student) {
    students.add(newStudent)
    println("Enrolled student ${newStudent.name} at $name...")
  }
}
```

Solution 4.1 **Professor.kt**

```
class Professor(val name: String, val age: Int, val yearlySalary: Int) {

  val coursesTaught = mutableListOf<Course>()

  fun teach(course: Course) {
    coursesTaught.add(course)
    println("Teaching ${course.title}...")
  }

  fun doOralExam(student: Student, course: Course) {
    println("Testing student ${student.name} in ${course.title}")
    student.grade(course, Grade.A)  // Everyone gets an A
  }
}
```

Solution 4.1 **Student.kt**

```
class Student(val name: String, val age: Int, val matriculationNo: Int) {

  val grades = mutableMapOf<Course, Grade>()
  val currentEnrollments = mutableListOf<Course>()

  private var successLikelihoodInPercent: Int = 50

  fun enroll(course: Course) {
    currentEnrollments.add(course)
    println("$name enrolled in ${course.title}...")
  }

  fun takeExam(course: Course) {
    println("$name takes exam in ${course.title}...")
  }

  fun learn() {
    println("$name is learning...")
    successLikelihoodInPercent = min(successLikelihoodInPercent + 2, 100)
  }

  fun party() {
    println("$name is partying! ¥uD83C¥uDF89")
    successLikelihoodInPercent = max(successLikelihoodInPercent - 2, 0)
  }

  fun grade(course: Course, grade: Grade) {
    grades[course] = grade
  }
}
```

Solution 4.1 **Course.kt**

```
data class Course(val title: String, val description: String)
```

Solution 4.1 **Grade.kt**

```
enum class Grade {
  A, B, C, D, FAILED
}
```

Solution 4.1 **Main.kt**

```
fun main(args: Array<String>) {
  val rwth = University("RWTH Aachen University", 1870)
  val professor = Professor("John Doe", 44, 90_000)
  val student = Student("Sarah Keller", 22, 123456)
  val course = Course(
      "Programming I",
      "Learn object-oriented, functional, and logical programming"
  )

  rwth.hire(professor)
  rwth.enroll(student)
  professor.teach(course)
  student.enroll(course)
  student.party()
  student.party()
  student.party()
  student.learn()
  student.takeExam(course)
  professor.doOralExam(student, course)
}
```

Solution for Exercise 4.2

Solution 4.2 **Inheritance Issues**

```
/*
 * Although a square "is-a" rectangle, it requires additional invariants, here width ==
height.
 * In such scenarios, inheritance may be undesirable. It's useful rather when a
subclass really
```

```
 * just adds functionality in addition to its superclass.
 *
 * The problem here boils down to the Liskov Substitution Principle (LSP), one of the
principles
 * of object-oriented software design. It states that a subclass should always be able
to be used
 * in place of its superclass and behave the same way (preserve invariants etc).
 *
 * The takeaway is that object-oriented design, and especially inheritance, are not
always the ideal
 * way to model or solve a problem.
 */
open class Rectangle(var width: Double, var height: Double) {
  open fun area() = width * height
  open fun stretch(factor: Double) {
    width *= factor
  }
}

class Square(width: Double) : Rectangle(width, width) {
  override fun stretch(factor: Double) {
    super.stretch(factor)
    height *= factor  // Violates invariant of Rectangle. But otherwise would violate
invariant of Square.
  }
}

fun main(args: Array<String>) {
  val rect: Rectangle = Square(10.0)  // Rectangle happens to be a Square at runtime
  rect.stretch(2.0)
  println(rect.height)  // Would expect unchanged 10.0 for a rectangle
  println(rect.width)   // Would expect stretched 20.0 for a rectangle
  println(rect.area())  // Would expect 200.0 for a rectangle that was stretched only
in width
}
```

Solution for Exercise 4.3

Solution 4.3 Singly Linked List

```
open class SinglyLinkedList<out E>(vararg elements: E) : Iterable<E> {

  val head: Node

  interface Node
  class SinglyLinkedNode<out E>(val value: E, val tail: Node) : Node
  class Sentinel : Node

  init {
    var nextNode: Node = Sentinel()
    for (element in elements.reversed()) {  // Builds up structure backwards, starting
at the sentinel
      nextNode = SinglyLinkedNode(element, nextNode)
    }
    head = nextNode
  }

  override operator fun iterator(): Iterator<E> = object : Iterator<E> {
    var currentNode: Node = head

    override fun hasNext(): Boolean = currentNode !is Sentinel

    override fun next(): E = (currentNode as SinglyLinkedNode<E>).value.also {
      currentNode = (currentNode as SinglyLinkedNode<*>).tail
    }
  }

  operator fun get(index: Int) = when (head) {
    is SinglyLinkedNode<*> -> this.elementAt(index)
    is Sentinel -> throw IndexOutOfBoundsException("Singly linked list is empty.")
    else -> throw IllegalStateException("This should not happen: Node should either be
Sentinel or SinglyLinkedNode.")
  }

  operator fun contains(element: @UnsafeVariance E) = this.any { it == element }
}

// Simple test run
fun main(args: Array<String>) {

  val list = SinglyLinkedList(1, 2, 3, 4, 5)

  for (element in list) {  // Uses `iterator`
```

```
    println("Next element: $element")
  }

  list.filter { it > 2 }.map { it * 2 }.forEach(::println)  // Only works for
`Iterable`s

  println(3 in list)  // Uses `contains`
  println(9 in list)

  println(list[0])    // Uses `get`
  println(list[3])
  println(list[5])    // Should throw OutOfBoundsException
}

/*
 * Singly linked lists take up less memory than doubly-linked lists since only half as
many pointers are stored (except
 * if some clever optimizations are applied that are possible in some languages).
 * Also, in mutable implementations, fewer links have to be adjusted to add and remove
elements.
 * The read-only version in this exercise enables another interesting feature called
structure sharing, which means that
 * the tail can easily be the tail of multiple heads -- this doesn't work in doubly-
linked lists.
 *
 * On the other hand, removing is easier in doubly-linked lists because you need to get
a reference to the previous
 * element in order to adjust the links (and doubly-linked nodes have a pointer to
their previous element).
 *
 * Array lists allow very efficient (constant-time) access at any index but adding and
removing elements can be hard
 * once the underlying array runs out of space. This is good for lists where you
primarily read from random indices.
 */
```

Solution for Exercise 4.4

There are again a million different ways to model this system. This solution is rather simplistic. It encapsulates commonalities into abstract superclasses.

Note that the concrete game characters wouldn't have to be classes of their own yet in this design but, in a more fleshed-out version, they would likely have to be classes of their own anyway.

Solution 4.4 **GameCharacter.kt**

```
abstract class GameCharacter(val name: String, var healthPoints: Int, var attackPoints:
Int, var defensePoints: Int) {

  fun attack(other: GameCharacter) {
    val damage = max(0, this.attackPoints - other.defensePoints)
    other.healthPoints = max(0, other.healthPoints - damage)
    println("> ${this.name} made $damage damage to ${other.name} (${other.name} has
${other.healthPoints} HP left)")
  }

  fun isDead() = healthPoints <= 0

  fun heal(hp: Int) {
    this.healthPoints += hp  // There's no maximum for HP
    println("> ${this.name} healed $hp HP (now has ${this.healthPoints} HP)")
  }
}
```

Solution 4.4 **Hero.kt**

```
abstract class Hero(
    name: String,
    hp: Int,
    attackPoints: Int,
    defensePoints: Int,
    var hasSword: Boolean = false,
    var hasArmor: Boolean = false,
    var experiencePointsEarned: Int = 0
) : GameCharacter(name, hp, attackPoints, defensePoints) {

  private val swordAttackBonus = 6
  private val armorDefenseBonus = 2

  fun equipSword() {
    if (hasSword) return
```

```
    attackPoints += swordAttackBonus
    hasSword = true
    println("> You found a sword! You now have ${this.attackPoints} ATK!")
  }

  fun equipArmor() {
    if (hasArmor) return

    defensePoints += armorDefenseBonus
    hasArmor = true
    println("> You found an armor! You now have ${this.defensePoints} DEF!")
  }

  fun fight(enemy: Enemy) {
    while (this.healthPoints > 0 && enemy.healthPoints > 0) {
      attack(enemy)
      if (!enemy.isDead()) {
        enemy.attack(this)
      }
    }
    if (!isDead()) {
      experiencePointsEarned += enemy.experiencePoints
    }
  }
}
```

Solution 4.4 **Warrior.kt**

```
class Warrior : Hero("Warrior", 50, 10, 5)
```

Solution 4.4 **Enemy.kt**

```
abstract class Enemy(name: String, hp: Int, attackPoints: Int, defensePoints: Int, val
experiencePoints: Int)
  : GameCharacter(name, hp, attackPoints, defensePoints)
```

Solution 4.4 **Goblin.kt**

```
class Goblin : Enemy("Goblin", 16, 9, 3, 75)
```

Solution 4.4 **Orc.kt**

```
class Orc : Enemy("Orc", 30, 11, 4, 200)
```

Solution 4.4 **Main.kt**

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

  val hero = Warrior()

  loop@ while (true) {  // Main game loop
    println("> Fight [g]oblin or [o]rc?")
    print("> ")
    val input = readLine()!!.toLowerCase()
    when (input) {
      !in arrayOf("g", "o") -> {
        println("> Please enter 'g' to fight a [g]oblin or 'o' to fight an [o]rc!")
        continue@loop
      }
      "g" -> hero.fight(Goblin())
      "o" -> hero.fight(Orc())
    }

    if (!hero.hasSword && Math.random() < 0.25)
      hero.equipSword()

    if (!hero.hasArmor && Math.random() < 0.15)
      hero.equipArmor()

    if (Math.random() < 0.10)
      hero.heal(10)

    if (hero.isDead()) {
      println(">>> You're dead! You earned ${hero.experiencePointsEarned} XP!")
      break
    } else {
      println("> Status: ${hero.healthPoints} HP left (${hero.attackPoints} ATK,
${hero.defensePoints} DEF)")
    }
```

```
  }
}
```

Solution for Exercise 4.5

Solution 4.5 Delegates and Late-Initialized Properties

```
/*
 * 1) A test class' property that is re-initialized before each test case
 *
 * -> This is a typical use case for a lateinit property because a framework or
separate entity (here the test runner)
 *    is responsible for initializing it.
 */
class Person(var age: Int) {
  fun birthday() { age++ }
}

annotation class BeforeEach  // Let's imagine this comes from JUnit

class SomeTest {

  lateinit var person: Person

  @BeforeEach  // In a JUnit test case, this would automatically run before each test
case
  fun setup() {
    person = Person(20)
  }

  fun birthdayShouldIncreaseAgeByOne() {
    // when
    val ageBefore = person.age

    // given
    person.birthday()
```

```
    // then
    // (misusing Kotlin's `assert` as test case asserter for this example...)
    assert(person.age == ageBefore + 1) { "Age should increase by 1 on birthday." }
  }
}

/*
 * 2) Validating property changes and rejecting invalid changes
 *
 * -> Observing property changes is possible with delegated properties, both
`observable` and `vetoable`. To be able to
 *    reject invalid changes, you must use `vetoable`
 */
class Cat {
  var name by Delegates.vetoable("Oscar") { property, oldName, newName ->
    newName.isNotBlank()  // Reject blank names
  }
}

val oscar = Cat()
oscar.name = "  "
println(oscar.name)

/*
 * 3) Properties holding UI components that can only be initialized once the necessary
context is ready
 *
 * -> This is another use case for lateinit which allows declaring all desired
properties at the top of the class and
 *    initializing them once the necessary context is created. Note that there is no
external framework that's
 *    responsible for the initialization here so take care to perform all
initializations before use.
 */
open class ViewModel
class MyViewModel : ViewModel()

class ViewModels {
  companion object {
    fun <T : ViewModel> get(viewModel: Class<T>) = MyViewModel()
  }
}
```

```
class SomeActivity {  // Let's imagine this was an Android activity
  lateinit var someViewModel: MyViewModel  // Cannot be initialized yet

  fun onCreate() {  // On Android, this would be a lifecycle method
    someViewModel = ViewModels.get(MyViewModel::class.java)
  }
}

/*
 * 4) Expensive objects that may not be used at runtime because they are not used on
all execution paths
 *
 * -> To initialize (expensive) objects only if and when it becomes necessary, you can
use a `lazy` delegated property
 */
class Database

class AnotherActivity {
  val db: Database by lazy { buildExpensiveDatabaseObject() }

  fun buildExpensiveDatabaseObject() = Database()  // Let's imagine this was an
expensive object creation
}
```

Solution for Exercise 4.6

Solution 4.6 Visitor Pattern

```
// You may also model this as just a single class `TreeNode` and decide if it's a
branch or leaf via its no. of children
sealed class TreeNode<T>(var value: T, val children: List<TreeNode<T>>)

class Branch<T>(value: T, children: List<TreeNode<T>>) : TreeNode<T>(value, children) {
  override fun toString(): String = "{$value, ${children.map { it.toString() }}}"
}

class Leaf<T>(value: T) : TreeNode<T>(value, emptyList<TreeNode<T>>()) {  //
`emptyList` does not allow adding elements
  override fun toString(): String = "{$value}"
```

```
}

fun <T> visitDepthFirst(root: TreeNode<T>, transform: (TreeNode<T>) -> Unit): Unit =
when (root) {
  is Leaf -> transform(root)
  is Branch -> {
    transform(root)
    root.children.forEach { visitDepthFirst(it, transform) }
  }
}

fun <T> visitBreadthFirst(root: TreeNode<T>, transform: (TreeNode<T>) -> Unit): Unit =
when (root) {
  is Leaf -> transform(root)
  is Branch -> {
    transform(root)
    visit(root, transform)
  }
}

private fun <T> visit(node: TreeNode<T>, transform: (TreeNode<T>) -> Unit): Unit {
  node.children.forEach { transform(it) }
  node.children.forEach { visit(it, transform) }
}

fun main(args: Array<String>) {

  // Builds up the tree from Figure 4.3 in the Instructor's Manual
  val tree = Branch(1, listOf(
      Branch(2, listOf(
          Leaf(5)
      )),
      Leaf(3),
      Branch(4, listOf(
          Branch(6, listOf(
              Leaf(8)
          )),
          Leaf(7)
      ))
  ))

  println("Original: $tree")
  visitDepthFirst(tree) { it.value = it.value * 2 }
```

```
  println("Times two: $tree")
  visitDepthFirst(tree) { it.value = it.value - 1 }
  println("Minus one: $tree")

  visitBreadthFirst(tree) { it.value = (it.value + 1) / 2 }
  println("Back to original: $tree")

  visitBreadthFirst(tree) { print("${it.value}, ") }
  println()
  visitDepthFirst(tree) { print("${it.value}, ") }
}
```

Solution for Exercise 4.7

Solution 4.7 Strategy Pattern

```
// The navigation system can use different strategies
class NavigationSystem {
  fun navigate(destination: Location, strategy: NavigationStrategy =
FastestRouteStrategy())
      = strategy.calculateRoute(destination)

  // In reality, the `strategy` function and thus `navigate` would return a route
  fun navigate(destination: Location, strategy: (Location) -> Unit)
    = strategy(destination)
}

data class Location(val latitude: Double, val longitude: Double)

// Traditional approach: Strategy interface with a subclass for each concrete strategy
interface NavigationStrategy {
  fun calculateRoute(destination: Location)
}

class ShortestRouteStrategy : NavigationStrategy {
  override fun calculateRoute(destination: Location) {
    println("Calculating shortest route...")
  }
}
```

```
class FastestRouteStrategy : NavigationStrategy {
  override fun calculateRoute(destination: Location) {
    println("Calculating fastest route...")
  }
}

// Functional approach: algorithms can be extracted and passed as function instead of
using inheritance
val shortestRouteStrategy = { location: Location ->
  println("Calculating shortest route to $location (via lambda)...")
}

val fastestRouteStrategy = { location: Location ->
  println("Calculating fastest route to $location (via lambda)...")
}

fun main(args: Array<String>) {
  val navi = NavigationSystem()
  val shortestRoute = ShortestRouteStrategy()
  val stonehenge = Location(51.178883, -1.826215)

  navi.navigate(stonehenge, shortestRoute)
  navi.navigate(stonehenge, fastestRouteStrategy)
}

/*
 * This exercise nicely presents these "new ways of modularity" functional programming
enables. Lambda expressions can
 * encapsulate algorithms and make them interchangeable without using inheritance.
 */
```

Solution for Exercise 5.1

Solution 5.1 Platform Types

```
/*
 * The inferred platform types are, in order, the following:
 * 1) Double
 * 2) Int!
 * 3) (Mutable)List<String!>
 * 4) CharArray!
 * 5) Array<(out) Person!>!
 * 6) Any!
 * 7) MutableList<out Comparable<T!>!>!
 * 8) (Mutable)Map<Address, (Mutable)List<Employee!>>!
 */

/*
 * Here are the examples in code to verify. These use the Java class `Exercise5_1`.
 * To see the inferred platform types, let IntelliJ show autocomplete inside the
methods and look at the return types.
 */
val average = calculateAverage()    // Double
val age = getAge()                  // Int!
val titles = titles()               // (Mutable)List<String!>
val chars = toCharArray()           // CharArray!
val people = getAttendees()         // Array<(out) Person!>!
val thing = get()                   // Any!
val ratings = fetchRatings<Int>()   // MutableList<out Comparable<T!>!>!
val locationsToEmployees = locationsToEmployees()  // (Mutable)Map<Address,
(Mutable)List<Employee!>>!
```

Solution 5.1 Java Class with Corresponding Methods for Validation

```
public class Exercise5_1 {

    public static double calculateAverage() { return 1.0; }  // Only method that cannot
return null

    public static Integer getAge() { return null; }
```

```
    public static @NonNull
    List<String> titles() { return null; }  // Can still return null, but with a
warning

    public static char[] toCharArray() { return null; }

    public static Person[] getAttendees() { return null; }

    public static Object get() { return null; }

    public static <T> List<? extends Comparable<T>> fetchRatings() { return null; }

    public static Map<@NonNull Address, @NonNull List<Employee>> locationsToEmployees()
{ return null; }

    public class Address {}
    public class Employee {}
    public class Person {}
}
```

Solution for Exercise 5.2

Solution 5.2 Improved Treasure Chest Class

```
class TreasureChest {

    // You don't have to use an enum here but it's good practice to avoid Kotlin
keywords as field or method names.
    enum Status { OPEN, SEALED }

    private Status status = SEALED;

    // Here, `object` was a hard keyword so it's definitely better to avoid it
    Object treasure = new Diamond();

    // Here we avoid the hard keyword `is`; `contains` is a more conventional for this
anyway
    boolean contains(Object object) { return this.treasure == object; }

    // `open` is a modifier in Kotlin but it's a good method name and unlikely to cause
confusion so we kept it here.
```

```
    void open() { this.status = OPEN; }
    void seal() { this.status = SEALED; }

    // This avoids the soft keyword `get` as there's no need for it, it's a bad method name anyway
    Status status() { return this.status; }

    class Diamond {}
}
```

Solution 5.2 **Usability in Kotlin**

```
// Using the original class
private val chestOriginal = TreasureChestOriginal()
if (chestOriginal.`is`(TreasureChestOriginal().Diamond())) {  // Must escape `is` because it's a hard keyword
  chestOriginal.open()
  println(chestOriginal.`object`)  // Must escape `object` because it's a hard keyword
  chestOriginal.seal()
}

// Using the improved class
private val chest = TreasureChest()
if (chest.contains(TreasureChest().Diamond())) {
  chest.open()
  println(chest.treasure)
  chest.seal()
}
```

Solution for Exercise 5.3

Solution 5.3 **JVM Annotations in Action**

```
/*
 * 1) You want to prevent the generation of getters and setters for a property and instead allow direct field access.
 * -> @JvmField exposes a property directly as a field in the Java bytecode, thus without getters or setters.
 */
class ExposedField {
  @JvmField val exposed = "No getter or setter"
```

```
}

/*
 * 2) You want to prevent class names such as CandyKt or CookieKt when using your
Kotlin files from Java.
 * -> Any renaming of identifiers when compiling to Java bytecode can be done via
@JvmName. To rename the class
 * generated from an entire file as in this scenario is done using @file:JvmName at the
top of the file.
 */
// See top of this file for an example

/*
 * 3) You want to be able to call a companion object method directly on its containing
class from Java.
 * -> Calling a method directly on a Java class is enabled by static methods so you
need the @JvmStatic annotation.
 */
class Accessor {
  companion object {
    @JvmStatic fun access() {}
  }
}

/*
 * 4) You want to at least have some of the overloaded options to call a Kotlin
function with optional arguments from Java.
 * -> This is enabled by adding @JvmOverloads to the function declaration. It only
enabled a constrained form of
 * optional arguments because the order of parameters is preserved -- for n optional
parameters, only n+1 different
 * combinations of calling the method from Java are enabled.
 */
@JvmOverloads fun overloaded(i: Int = 1, s: String = "", d: Double = 0.0) {}
```

Solution 5.3 **Using the Declarations from Java**

```
public class Exercise5_3 {

    public static void main(String[] args) {
```

```
        String field = new ExposedField().exposed;

        Accessor.access();

        JvmAnnotations.overloaded();
        JvmAnnotations.overloaded(42);
        JvmAnnotations.overloaded(42, "Hello");
        JvmAnnotations.overloaded(42, "Hello", 3.14159);
    }
}
```

Solution for Exercise 5.4

Solution 5.4 Generating Static Declarations in the Bytecode

```
/*
 * Generates no static members.
 */
class Constants1 {
  @JvmField val PI = BigDecimal(3.14159265358979323846433)
}

/*
 * Generates a static method due to @JvmStatic.
 * Additional static INSTANCE generated for Singleton pattern
 */
object MainPresenter {
  fun notifyUser(message: String) {}
}

/*
 * Generates a private static member for `defaultRobot` but the public getter for it is
not static.
 * Additional static member INSTANCE generated for Singleton pattern.
 */
object RobotFactory {
  val defaultRobot = Robot()
}
```

```
/*
 * Generates a static field PI.
 * Additional static INSTANCE generated for Singleton pattern
 */
object Constants2 {
  const val PI = 3.14159265358979323
}

/*
 * Generates a static nested class PersonBuilder.
 * The companion object methods are not static.
 */
class Person {
  companion object PersonBuilder {
    fun build(): Person = Person()
  }
}

/*
 * Generates no static members.
 */
class MainUi {
  lateinit var btnOk: Button
}



// -----------
class Robot
```

Solution for Exercise 5.5

Solution 5.5 The Nothing Type in Action

```
/*
 * A good return type for functions such as `fail` and `exit` is Kotlin's special
`Nothing` type.
 */
fun fail(message: String): Nothing = throw IllegalStateException(message)
```

```
/*
 * This return type tells the compiler that the function never terminates, and the
compiler can use this information to
 * infer interesting data such as nullability.
 */
fun fetchMessage(): String? = "Hello, World!"
val nullable: String? = fetchMessage()
val notNullable: String = fetchMessage() ?: fail("Could not fetch a message.")
// `notNullable` cannot be null here because, if it was, the line above wouldn't
terminate so this line wouldn't execute
```

Solution for Exercise 5.6

Solution 5.6 Improved Code and Function Definition

```
/*
 * - Giving the generated class an explicit name prevents ugly "TreesKt" naming and
avoids leaking
 *   the implementation to the outside.
 */
@file:JvmName("TreeUtils")

// ...

/*
 * - Extension function allows calling more naturally (from Kotlin)
 * - The concept of this function is that of a `map` function. Calling it `map` thus
gets across its
 *   functionality effectively and with a concise name (assuming readers know
functional programming)
 * - Making the lambda the last parameter allows calling it without parentheses (from
Kotlin)
 *
 * (Making the Tree immutable is definitely another way to improve the code but not
assumed as part
 * of the solution here).
 */
fun <T> Tree<T>.map(transform: (T) -> T) {
  when (this) {
    is Leaf -> value = transform(value)
    is Branch -> {
```

```
        value = transform(value)
        children.forEach { subtree -> subtree.map(transform) }
    }
  }
}
```

Solution 5.x **Calling the Original and Improved Code from Java**

```
public class CallingClass {

    static void useTrees() {
        List<Tree<Integer>> leafs = Arrays.asList(
            new Leaf<>(2),
            new Leaf<>(3)
        );
        Tree<Integer> tree = new Branch<>(1, leafs);

        // "Original" method
        // Note that the IDE cannot infer the generic type in the argument until the
second one is passed in
        TreesOriginalKt.appIyToAllNodes(node -> node * node, tree);

        // Improved `map` method
        TreeUtils.map(tree, integer -> integer * integer);

        System.out.println(tree);  // Uses Branch.toString() from data class
    }

    public static void main(String[] args) {
        useTrees();
    }
}
```

Solution for Exercise 6.1

Solution 6.1 Synchronization Issues with Shared Mutable State

```
/*
 * These exercise solutions use Kotlin 1.3 along with a newer coroutines version that
uses structured concurrency -- a
 * fascinating concept that makes your concurrent code more predictable and easier to
reason about.  It's used since
 * version 0.26.0: https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0
 *
 * Additionally,  the syntax change a little.  Read about the changes here:
 * 1) Structured concurrency: https://medium.com/@elizarov/structured-concurrency-
722d765aa952
 * 2) Kotlin 1.3: https://kotlinlang.org/docs/reference/whatsnew13.html
 */

runBlocking {

  // This will be our *shared mutable state* (you should already cringe at the sound of
those words)
  var counter = 0

  val coroutines = List(3) {
    // Coroutines are launched from GlobalScope (unless you want children coroutines);
since coroutines version 0.26.0
    launch {
      repeat(1_000_000) {
        counter++
      }
    }
  }

  coroutines.forEach { it.join() }

  // What do you think will be the result?
  println("Final counter: $counter")
}

/*
 * What do you see? How can you explain it? Do you know the name of this phenomenon?
 * -> You get random results but almost certainly not 3_000_000 (although that's what
you would expect without any
```

```
 *    synchronization issues). This can be explained by the fact that the three
coroutines concurrently access the
 *    counter and updates of one may overwritten by another and thus lost. The process
is this:
 *
 *    Coroutine1 reads `counter`: value is 100
 *    Coroutine1 increments value to 101 (but does not yet write the value back to
counter)
 *    Coroutine1 is suspended
 *    Coroutine2 reads `counter`: value is 100
 *    Coroutine2 increments to 101 and writes value: value is now 101
 *    Coroutine2 is suspended
 *    Coroutine1 writes back value: value is still 101
 *    -> Effectively, the counter was increased by 1 but two iterations of incrementing
were run
 *
 * Generally, these types of phenomena can be called synchronization issues because
multiple concurrent units compete
 * for a shared resource so that synchronization becomes necessary. More specifically,
the symptom of random counter
 * results is called a *Heisenbug*. Because it appears randomly and produces
unpredictable outcomes, it is extremely
 * nasty to debug -- so definitely avoid such situations in the first place, e.g., by
avoiding shared mutable state.
 */
```

Solution 6.1 (Bonus) Using Locks for Mutual Exclusion

```
runBlocking {

  var counter = 0
  val lock = Mutex()  // A mutex is the simplest way to implement mutual exclusion

  val coroutines = List(3) {
    launch {
      repeat(1_000_000) {
        lock.withLock {  // Only one coroutine at a time can enter this block to
increment the counter
          counter++
        }
```

```
      }
    }
  }

  coroutines.forEach { it.join() }

  println("Final counter: $counter")
}
```

Solution for Exercise 6.2

Solution 6.2 Fetching Chuck Norris Jokes (Implementing Parallelism)

```
/*
 * These exercise solutions use Kotlin 1.3 along with a newer coroutines version that
uses structured concurrency -- a
 * fascinating concept that makes your concurrent code more predictable and easier to
reason about. It's used since
 * version 0.26.0: https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0
 *
 * Additionally, the syntax change a little. Read about the changes here:
 * 1) Structured concurrency: https://medium.com/@elizarov/structured-concurrency-
722d765aa952
 * 2) Kotlin 1.3: https://kotlinlang.org/docs/reference/whatsnew13.html
 */

private val API_URL = "http://api.icndb.com/jokes/random"
val randomJokeUrl = URL(API_URL)

runBlocking {

  println(fetchJoke())

  do {
    print("> Hit ENTER for 3 more jokes or 'q' to quit")
    val input = readLine()!!
    if (input.isBlank()) {
      fetchJokesParallel(count = 3)
    }
```

```
  } while(input != "q")
}

suspend fun fetchJokesParallel(count: Int) = coroutineScope {
  repeat(count) {
    launch {
      val joke = fetchJoke()
      println(joke)
    }
  }
}

fun fetchJoke(): String {
  val json = randomJokeUrl.readText()
  val regex = Regex("¥"joke¥": ¥"(.*)¥",")
  val joke = regex.find(json)?.groupValues?.get(1) ?: ""  // Returns the match from the
first parentheses
  return joke
}
```

Solution 6.2 Adding a Timeout for Asynchronous Calls

```
/*
 * These exercise solutions use Kotlin 1.3 along with a newer coroutines version that
uses structured concurrency -- a
 * fascinating concept that makes your concurrent code more predictable and easier to
reason about.  It's used since
 * version 0.26.0: https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0
 *
 * Additionally, the syntax change a little. Read about the changes here:
 * 1) Structured concurrency: https://medium.com/@elizarov/structured-concurrency-
722d765aa952
 * 2) Kotlin 1.3: https://kotlinlang.org/docs/reference/whatsnew13.html
 */

private val API_URL = "http://api.icndb.com/jokes/random"
val randomJokeUrl = URL(API_URL)

runBlocking {
```

```
  println(fetchJoke())

  do {
    print("> Hit ENTER for 3 more jokes or 'q' to quit")
    val input = readLine()!!
    if (input.isBlank()) {
      fetchJokesParallel(count = 3)
    }
  } while(input != "q")
}

suspend fun fetchJokesParallel(count: Int) = coroutineScope {
  withTimeoutOrNull(1000) {  // This is the only change required (along with the elvis operator below)
    repeat(count) {
      launch {
        val joke = fetchJoke()
        println(joke)
      }
    }
  } ?: println("Error: Timed out while fetching jokes. Please retry.")
}

fun fetchJoke(): String {
  val json = randomJokeUrl.readText()
  val regex = Regex("¥"joke¥": ¥"(.*)¥",")
  val joke = regex.find(json)?.groupValues?.get(1) ?: ""  // Returns the match from the first parentheses
  return joke
}
```

Solution for Exercise 6.3

Solution 6.3 Compilation of Suspending Functions

```
/*
 * These exercise solutions use Kotlin 1.3 along with a newer coroutines version that uses structured concurrency -- a
 * fascinating concept that makes your concurrent code more predictable and easier to reason about.  It's used since
 * version 0.26.0: https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0
```

```
 *
 * Additionally, the syntax change a little. Read about the changes here:
 * 1) Structured concurrency: https://medium.com/@elizarov/structured-concurrency-
722d765aa952
 * 2) Kotlin 1.3: https://kotlinlang.org/docs/reference/whatsnew13.html
 */

fun normal(s: String, i: Int): Boolean {
  return true
}

suspend fun suspendable(s: String, i: Int): Boolean {
  delay(100)
  val a = 10
  delay(100)

  return true
}

/*
 * For the normal function, there's nothing surprising in the bytecode (or the
decompiled Java code). The Kotlin
 * compiler makes it a `final` method and adds @NotNull annotations where appropriate.
 *
 * For the suspend function, you can see the result of how the Kotlin compiler treats
suspending functions. Before we
 * dive into the method body, let's first recap the transformations regarding the
signature:
 * 1) Every suspending function gets an additional Continuation parameter
 * 2) The return type of the suspend function becomes the generic type parameter of the
Continuation, so here we get
 *    Continuation<Boolean>. You can only see this in a comment shown in the bytecode:
 *
 *    // declaration: suspendable(java.lang.String, int,
kotlin.coroutines.Continuation<? super java.lang.Boolean>)
 *
 * 3) The return type of the compiled method becomes Any? (or rather Object). Remember
that this represents the union
 *    of the original return type (Boolean) and the special COROUTINE_SUSPENDED marker.
 * 4) There's no `suspend` modifier (obviously because it's not something that the JVM
offers)
 *
 * Looking at the generated method body, you can see that it implements a simple state
```

```
machine. There's a `label` that
 * indicates which state you're currently in, and a switch-case statements that runs
the piece of the function that
 * corresponds to the current state. In our example:
 * -> We have 2 suspension points (the calls to `delay`)
 * -> Thus, we have 3 states in the state machine (the three blocks of code separated
by the delays, including the zero
 *    lines of code before the first delay)
 * -> Thus, the switch-case has 3 cases (one for each state), plus a default case. At
the end of each case, the label
 *    is increased by one so that the function knows where to continue later in case it
gets suspended.
 *
 * Depending on the structure of your suspend function, the compiler may perform all
kinds of optimizations, and the
 * details of code generation will likely keep changing with newer compiler versions
but this is the general structure
 * you should see.
 */
```

Solution for Exercise 6.4

Solution 6.4 Actors, Channels, and `select`

```
/*
 * These exercise solutions use Kotlin 1.3 along with a newer coroutines version that
uses structured concurrency -- a
 * fascinating concept that makes your concurrent code more predictable and easier to
reason about. It's used since
 * version 0.26.0: https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0
 *
 * Additionally, the syntax change a little. Read about the changes here:
 * 1) Structured concurrency: https://medium.com/@elizarov/structured-concurrency-
722d765aa952
 * 2) Kotlin 1.3: https://kotlinlang.org/docs/reference/whatsnew13.html
 */

suspend fun doPeriodically(periodInMs: Int, task: suspend () -> Unit) {
  while (true) {
    task()
    delay(periodInMs.toLong())
```

```
  }
}

// With structured concurrency, this is the common pattern to create actors (via
extension function on CoroutineScope)
fun CoroutineScope.produceRandomNumbers(periodInMs: Int = 0): ReceiveChannel<String> =
produce(capacity = RENDEZVOUS) {
  doPeriodically(periodInMs) {
    send("$periodInMs ms: ${Random().nextInt(100) + 1}")
  }
}

// Since Kotlin 1.3, the parameter for the main function is optional
fun main() = runBlocking<Unit> {

  val producer300ms = produceRandomNumbers(periodInMs = 300)
  val producer500ms = produceRandomNumbers(periodInMs = 500)
  val producer1000ms = produceRandomNumbers(periodInMs = 1000)

  actor<Nothing>(capacity = RENDEZVOUS) {
    doPeriodically(periodInMs = 200) {
      selectUnbiased<Unit> {  // Unbiased (fair) select that chooses randomly between
available sources
        producer300ms.onReceive { println(it) }
        producer500ms.onReceive { println(it) }
        producer1000ms.onReceive { println(it) }
      }
    }
  }

  delay(5000)
  System.exit(0)
}
```

Solution for Exercise 6.5

Solution 6.5 More Actors (and Calculating GPS Distances)

```
/*
 * These exercise solutions use Kotlin 1.3 along with a newer coroutines version that
uses structured concurrency -- a
```

```
 * fascinating concept that makes your concurrent code more predictable and easier to
reason about.  It's used since
 * version 0.26.0: https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0
 *
 * Additionally, the syntax change a little.  Read about the changes here:
 * 1) Structured concurrency: https://medium.com/@elizarov/structured-concurrency-
722d765aa952
 * 2) Kotlin 1.3: https://kotlinlang.org/docs/reference/whatsnew13.html
 */

data class Location(val lat: Double, val lng: Double)

val earthRadiusInKm = 6378.137

val houseLocation = Location(38.897675, -77.036530)

/**
 * Uses the haversine algorithm which is numerically stable even for low distances.
 */
fun distanceInMeters(a: Location, b: Location): Double {
  val deltaLat = (b.lat - a.lat).toRadians()
  val deltaLng = (b.lng - a.lng).toRadians()
  val sinDeltaLat = sin(deltaLat / 2)
  val sinDeltaLng = sin(deltaLng / 2)

  val squareOfHalfChordLength = sinDeltaLat.pow(2) + sinDeltaLng.pow(2) *
cos(a.lat.toRadians()) * cos(b.lat.toRadians())
  val angularDistance = 2 * atan2(sqrt(squareOfHalfChordLength), sqrt(1 -
squareOfHalfChordLength))

  return earthRadiusInKm * angularDistance * 1000
}

fun Double.toRadians() = (this * PI / 180.0)

runBlocking {

  val watcher = actor<Location> {
    consumeEach {  dogLocation ->
      val distanceFromHouse = distanceInMeters(houseLocation, dogLocation)
      println("Current distance: ${distanceFromHouse}m ¥t¥t-- Dog is at
(${dogLocation.lat}, ¥t${dogLocation.lng})")
      if (distanceFromHouse >= 20.0) {
```

```
        println("WARNING: Dog may be at neighbors' again!")
      }
    }
  }

  produce<Location> {
    var dogLocation = houseLocation
    while(true) {
      watcher.send(dogLocation)
      dogLocation = Location(dogLocation.lat - 0.00001, dogLocation.lng + 0.00002)
      delay(1000)
    }
  }

  // Lets this simulating run for 15 seconds
  delay(15_000)
  System.exit(0)
}
```

Solution for Exercise 9.1

Solution 9.1 Recipe DSL

```
/*
 * This is a rather minimal DSL, without any additional nesting. Instead, the `add`
function ends up reading like part
 * of a recipe (see [Main.kts]).
 */

fun recipe(name: String, init: RecipeBuilder.() -> Unit) =
RecipeBuilder(name).apply(init).build()

class RecipeBuilder(val recipeName: String) {
  var description: String = ""
  var instructions: String = ""
  private val ingredients: MutableList<Ingredient> = mutableListOf()

  fun add(ingredient: Ingredient) = ingredients.add(ingredient)

  fun build() = when {
    isValid() -> Recipe(recipeName, description, ingredients)
    else -> throw IllegalStateException("Need at least 2 ingredients")
  }

  private fun isValid() = ingredients.size >= 2
}

data class Recipe(
    private val name: String,
    private val description: String,
    private val ingredients: List<Ingredient>
)

data class Ingredient(private val name: String, private val amount: Amount)

data class Amount(private val amount: Number, private val measure: Measure)

// You can use other units that you're used to (cups, oz); note that this list is very
incomplete
enum class Measure(val representation: String) {
  GRAMS("g"),
  MILLILITERS("ml"),
```

```
  PIECES(" pc"),
  PACK(" pack")
}

infix fun Amount.of(food: String) = Ingredient(food, this)

val Number.g: Amount
  get() = Amount(this, GRAMS)

val Number.ml: Amount
  get() = Amount(this, MILLILITERS)

val Number.pc: Amount
  get() = Amount(this, PIECES)

val Number.pack: Amount
  get() = Amount(this, PACK)
```

Solution 9.1 **Using the Recipe DSL**

```
/*
 * There are a million different ways to design this DSL. One extremely useful feature
here is to have extension
 * properties for the units (such as mg and g) and infix function `of` that allows the
code to read almost like a recipe.
 *
 * There are certainly many ways to flesh this DSL out further, and I encourage you to
do so if you want to gather more
 * experience building DSLs in Kotlin.
 */

val bananaBreadRecipe = recipe("Banana & Walnut Bread") {

  description = "A healthy & tasty banana and walnut bread"

  add(2.pc of "Ripe banana")
  add(125.ml of "Milk (any kind)")
  add(10.ml of "Lemon juice")

  add(200.g of "All-purpose flour")
```

```
  add(100.g of "Whole wheat flour")
  add(100.g of "Brown sugar")
  add(1.pack of "Baking powder")
  add(2.g of "Salt")
  add(20.g of "Flaxseeds")
  add(100.g of "Walnuts")

  instructions = """Mash bananas and mix with rest of wet ingredients. Mix all dry
ingredients and stir into
      wet ingredients until you get a (quite wet) dough. Put into baking form,
sprinkle water on top,
      and bake for 45min at 175° C. Cut a line into the top of the bread after around
5min of baking."""
}

println(bananaBreadRecipe)
```

Solution for Exercise 9.2

Solution 9.2 **settings.gradle.kts**

```
include(":app")
```

Solution 9.2 **build.gradle.kts (Root Project)**

```
// Top-level build file where you can add configuration options common to all sub-
projects/modules.

buildscript {
    extra["kotlin_version"] = "1.2.60"
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath("com.android.tools.build:gradle:3.1.4")
```

```
        classpath("org.jetbrains.kotlin:kotlin-gradle-
plugin:${extra["kotlin_version"]}")

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task<Delete>("clean") {
    delete(rootProject.buildDir)
}
```

Solution 9.2 **build.gradle.kts (app module)**

```
import org.jetbrains.kotlin.gradle.internal.AndroidExtensionsExtension

plugins {
    id("com.android.application")
    id("kotlin-android")
    id("kotlin-android-extensions")
    id("kotlin-kapt")
}

android {
    compileSdkVersion(27)
    defaultConfig {
        applicationId = "com.petersommerhoff.kudoofinal"
        minSdkVersion(19)
        targetSdkVersion(27)
        versionCode = 1
        versionName = "1.0"
        testInstrumentationRunner = "android.support.test.runner.AndroidJUnitRunner"
    }
```

```
    buildTypes {
        getByName("release") {
            isMinifyEnabled = false
            proguardFiles("proguard-rules.pro")
        }
    }
}

dependencies {
    val kotlin_version: String by rootProject.extra

    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version")
    implementation("com.android.support:appcompat-v7:27.1.1")
    implementation("com.android.support.constraint:constraint-layout:1.1.2")
    implementation("com.android.support:design:27.1.1")

    val room_version = "1.1.1"
    implementation("android.arch.persistence.room:runtime:$room_version")
    kapt("android.arch.persistence.room:compiler:$room_version")

    val coroutines_version = "0.24.0"
    // Use newest version if you want (might differ)
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutines_version")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
android:$coroutines_version")

    val lifecycle_version = "1.1.1"
    implementation("android.arch.lifecycle:extensions:$lifecycle_version")
    kapt("android.arch.lifecycle:compiler:$lifecycle_version")

    testImplementation("junit:junit:4.12")
    androidTestImplementation("com.android.support.test:runner:1.0.2")
    androidTestImplementation("com.android.support.test.espresso:espresso-core:3.0.2")
}

androidExtensions {
  configure(delegateClosureOf<AndroidExtensionsExtension> {
    isExperimental = true
  })
}
```