5. Ассоциативные массивы aka Maps и множества aka Sets

Введение

В прошлом уроке мы познакомились с вами с различными структурами данных, которые предназначены для хранения упорядоченной последовательности элементов. Понятно, что это далеко не единственный тип структур данных, которые существуют, и сегодня мы с вами познакомимся с двумя новыми структурами данных.

Чтобы понять первую структуру данных --- **ассоциативный массив** --- далеко ходить не надо, достаточно вспомнить о такой штуке как толковый словарь. Он связывает элементы отношением "ключ"-"значение": для определенных слов (ключей) он содержит их описание (значения), для всех остальных --- не содержит ничего. Подобной структурой обладают, на самом деле, многие вещи: набор товаров с их ценами, список контактов в телефоне, рестораны и рейтинги, и т.д. Основная операция, которую они поддерживают, --- это достать значение, соответствующее интересующему нас ключу, т.е. то, что вы делаете, когда ищете значение неизвестного вам слова в словаре.

Ассоциативный массив является обобщенным способом представить подобное отношение. Давайте на следующем примере посмотрим, как с ним можно работать. Представим, что нам необходимо посчитать стоимость нашего списка покупок для заданного набора товаров. Сделать это можно при помощи следующей функции.

```
fun shoppingListCost(
        shoppingList: List<String>,
        costs: Map<String, Double>): Double {
    var totalCost = 0.0

    for (item in shoppingList) {
        val itemCost = costs[item]
        if (itemCost != null) {
            totalCost += itemCost
        }
    }

    return totalCost
}
```

Что мы здесь видим? Наша функция принимает на вход список покупок: параметр `shoppingList` типа `List<String>` --- и набор цен для товаров: параметр `costs` типа `Map<String, Double>`. Данный **параметризованный тип** `Map<Key, Value>` и является типом ассоциативного массива, у которого типовой параметр `Key` задает тип ключей, а `Value` --- тип значений. В нашем случае набор товаров с ценами имеет тип `Map<String, Double>`, т.е. для названия товара содержит его цену в виде действительного числа.

Для того, чтобы считать общую стоимость выбранного набора товаров, мы заводим новую изменяемую переменную `totalCost`, которая изначально равна нулю и которую мы возвращаем как результат в конце функции при помощи `return`. После этого мы проходимся по списку покупок при помощи цикла `for` и для каждой покупки пытаемся достать ее цену из нашего ассоциативного массива при помощи операции индексирования. В отличии от индексирования для списка, операция **индексирования** `map[key]` для ассоциативного массива пытается достать элемент не по какому-то целочисленному индексу, а по ключу соответствующего типа --- в нашем случае, по названию товара, т.е. строке.

А вот дальше мы знакомимся с такой очень интересной вещью как `null`. Как мы отметили раньше, ассоциативный массив содержит пары "ключ"-"значение", однако для некоторых ключей соответствующего им значения может не быть. Вместе с тем, просто так вернуть **"ничего"** мы не можем. Как раз для таких ситуаций и необходим объект `null` --- операция индексирования для ассоциативного массива возвращает `null` в случае, если для заданного ключа нет значения. После того, как мы проверили, что для товара есть его стоимость (`itemCost != null`), мы добавляем ее к общей стоимости набора; в противном случае мы считаем, что данная покупка просто игнорируется.

Попробуем написать тесты для нашей функции.

```
@Test
fun shoppingListCostTest() {
    val itemCosts = mapOf(
            "Хлеб" to 50.0,
            "Молоко" to 100.0
    )
    assertEquals(
            150.0,
            shoppingListCost(
                    listOf("Хлеб", "Молоко"),
                    itemCosts
            )
    )
    assertEquals(
            150.0,
            shoppingListCost(
                    listOf("Хлеб", "Молоко", "Кефир"),
                    itemCosts
            )
    )
    assertEquals(
            0.0,
            shoppingListCost(
                    listOf("Хлеб", "Молоко", "Кефир"),
                    mapOf()
            )
    )
}
```

Как видно из тестов, для **создания** ассоциативного массива может использоваться функция `mapOf()`, которая принимает на вход набор **пар** "ключ"-"значение" типа `Pair<A, B>` (в нашем случае, `Pair<String, Double>`). Для создания пары можно использовать либо конструкцию `Pair(a, b)`, либо запись `a to b`, обе из которых создадут пару из `a` и `b`. Для того, чтобы обратиться к первому или второму элементу пары `pair`, следует использовать запись `pair.first` или `pair.second` соответственно.

В нашем случае мы создаем пары из названия товара и его стоимости (хлеб за 50.0 и молоко за 100.0), после чего собираем из них ассоциативный массив. Затем мы проверяем три случая:

- Список покупок содержит хлеб и молоко, и общая стоимость должна быть равна 150.0
- Список покупок, кроме хлеба и молока, содержит еще кефир, но --- так как его стоимости мы не знаем --- мы его игнорируем, и общая стоимость все равно должна быть равна 150.0
- Для какого-то списка покупок с **пустым** ассоциативным массивом (мы не знаем ни одной цены товара) общая стоимость должна быть равна 0.0

В третьем случае мы создаем пустой ассоциативный массив при помощи функции `mapOf()` без аргументов. Типовые параметры в данном случае компилятор Котлина понимает из того, какой тип должен быть у второго аргумента функции `shoppingListCost`, поэтому их можно не указывать.

Распространённые операции над ассоциативными массивами

Рассмотрим основные операции, доступные над ассоциативными массивами.

- `map[key] / map.get(key)` возвращает значение для ключа `key` или `null` в случае, если значения нет
- `map.size / map.count()` возвращает количество пар "ключ"-"значение" в ассоциативном массиве
- `map + pair` возвращает новый ассоциативный массив на основе `map`, в который добавлено (или изменено) значение ключа, соответствующее паре "ключ"-"значение" из `pair`
- `map - key` возвращает новый ассоциативный массив на основе `map`, из которого, наоборот, удалено значение ключа `key`
- `map1 + map2` собирает два ассоциативных массива в один, причем пары "ключ"-"значение" из `map2` вытесняют значения из `map1`
- `map - listOfKeys` возвращает новый ассоциативный массив на основе `map`, в котором нет ключей из списка `listOfKeys`
- `map.getOrDefault(key, defaultValue)` является расширенной версией операции индексирования. В случае, если в `map` есть значение для ключа `key`, данное выражение вернет его; если значения нет, то будет возвращено значение по умолчанию `defaultValue`.
- `key in map / map.contains(key) / map.containsKey(key)` возвращает `true`, если `map` содержит

значение для ключа `key` и `false` в противном случае

- `map.containsValue(value)` возвращает `true`, если `map` содержит значение `value` для хотя бы одного ключа и `false` в противном случае

Изменяемый ассоциативный массив

Как и в случае со списками, обычный ассоциативный массив (или `Map`) нельзя изменить; если вы хотите иметь такую возможность, то следует использовать изменяемый ассоциативный массив (или `MutableMap`) типа `MutableMap<Key, Value>`. Аналогично `List` и `MutableList`, `MutableMap` расширяет `Map`, т.е. объект `MutableMap` может использоваться везде, где нужен `Map`, --- в подобных местах вы просто не будете использовать его возможности по модификации.

`MutableMap` предоставляет следующие основные возможности по модификации.

- `map[key] = value` **изменяет** имеющееся значение для заданного ключа или **добавляет** новую пару "ключ"-"значение" в случае, если ключ `key` не был связан в `map`
- `map.remove(key)` **удаляет** пару, связанную с ключом `key`

Давайте, как и раньше, попробуем изучить возможности `MutableMap` на следующем примере: из телефонной книги (набора пар "ФИО"-"телефон") следует убрать все записи, не относящиеся к заданному коду страны. Сделать это можно, например, следующим образом.

```
fun filterByCountryCode(
        phoneBook: MutableMap<String, String>,
        countryCode: String) {
    val namesToRemove = mutableListOf<String>()

    for ((name, phone) in phoneBook) {
        if (!phone.startsWith(countryCode)) {
            namesToRemove.add(name)
        }
    }

    for (name in namesToRemove) {
        phoneBook.remove(name)
    }
}
```

Данную функцию можно разбить на две логические части. Первым делом мы проходимся по всем записям в нашей записной книжке и отбираем те из них, у которых телефон начинается с кода, отличного от заданного в параметре `countryCode`. Для этого мы используем функцию `str.startsWith(prefix)`, которая возвращает `true` в случае, если строка `str` **начинается** со строки `prefix`, и `false`, если это не так. Все имена, которые следует удалить, мы записываем в изменяемый список `namesToRemove`.

Обратите внимание на форму заголовка цикла: так как ассоциативный массив является

набором пар, при помощи такого синтаксиса мы можем сразу разбить элемент-пару на две отдельные переменные, доступные в теле цикла. Такое разбиение называется **разрушающим присваиванием** и может применяться к различным объектам, представляющим собой набор элементов, в частности, к спискам или парам. Два альтернативных (и более многословных) способа написать данный цикл приведены ниже.

```
fun filterByCountryCode(
        phoneBook: MutableMap<String, String>,
        countryCode: String) {
    val namesToRemove = mutableListOf<String>()

    for (entry in phoneBook) {
        val (name, phone) = entry
        if (!phone.startsWith(countryCode)) {
            namesToRemove.add(name)
        }
    }

    for (name in namesToRemove) {
        phoneBook.remove(name)
    }
}
```

```
fun filterByCountryCode(
        phoneBook: MutableMap<String, String>,
        countryCode: String) {
    val namesToRemove = mutableListOf<String>()

    for (entry in phoneBook) {
        val name = entry.key
        val phone = entry.value
        if (!phone.startsWith(countryCode)) {
            namesToRemove.add(name)
        }
    }

    for (name in namesToRemove) {
        phoneBook.remove(name)
    }
}
```

После того, как мы собрали все имена, которые следует удалить, мы это и делаем при помощи функции `map.remove(key)`. В итоге, после вызова нашей функции с побочным эффектом в переданном `MutableMap` останутся только записи, у которых номер телефона начинается с нужного нам кода страны.

Тесты для нашей функции выглядят следующим образом.

```
@Test
@Tag("Example")
fun filterByCountryCode() {
    val phoneBook = mutableMapOf(
            "Quagmire" to "+1-800-555-0143",
            "Adam's Ribs" to "+82-000-555-2960",
            "Pharmakon Industries" to "+1-800-555-6321"
    )

    filterByCountryCode(phoneBook, "+1")
    assertEquals(2, phoneBook.size)

    filterByCountryCode(phoneBook, "+1")
    assertEquals(2, phoneBook.size)

    filterByCountryCode(phoneBook, "+999")
    assertEquals(0, phoneBook.size)
}
```

Сперва мы при помощи функции `mutableMapOf` создаем `MutableMap`, в котором есть три записи, две для кода +1 и одна для кода +82. Затем мы пробуем отфильтровать записи по указанным кодам, корректность же проверяем, сравнивая получившийся размер `MutableMap` с ожидаемым. Более правильно, конечно же, было бы сравнивать обновленное значение с эталонным `MutableMap`, однако инициализация эталона занимает дополнительные 3-4 строчки для каждой проверки, поэтому мы немножечко схитрили таким образом. Если запустить наши тесты, то мы увидим, что они успешно проходят. При желании вы можете попробовать модифицировать тесты так, чтобы сравнивать результат с эталоном, и посмотреть, изменится ли результат тестирования.

К этому моменту у вас, скорее всего, возник следующий, довольно очевидный вопрос --- а зачем наша реализация такая сложная? Почему нельзя сразу убирать записи из `MutableMap` внутри цикла, который перебирает его записи? Давайте попробуем и посмотрим, что получится в таком случае.

```
fun filterByCountryCode(
        phoneBook: MutableMap<String, String>,
        countryCode: String) {
    for ((name, phone) in phoneBook) {
        if (!phone.startsWith(countryCode)) {
            phoneBook.remove(name)
        }
    }
}
```

Код стал короче и понятнее, вот только при попытке запустить тесты они упадут с ошибкой `java.util.ConcurrentModificationException`. Название ошибки тонко намекает нам, в чем проблема, --- мы пытаемся перебирать элементы структуры данных и **одновременно** изменять эту самую структуру данных. В этом мы подобны дровосеку, который решил

забраться на сук и срубить его таким образом, --- "ну а что такого, удобно же!" К сожалению, как и в жизни, в программировании подобные чудеса не работают --- очень многие структуры данных (в том числе, и `MutableMap`) не позволяют вам одновременно перебирать и изменять свои элементы. Именно поэтому наша реализация состояла из двух отдельных частей: мы сперва собрали те элементы, что требуется удалить, а потом их удалили.

Распространённые операции над изменяемым ассоциативными массивами

Рассмотрим основные операции, доступные над изменяемыми ассоциативными массивами.

- `map.clear()` удаляет все записи из данного `MutableMap`
- `map[key] = value / map.put(key, value)` добавляет или изменяет соответствующую пару "ключ"-"значение"
- `map.putAll(otherMap)` добавляет в `MutableMap` `map` все пары из `otherMap`, в случае одинаковых ключей значения из `otherMap` перезаписывают значения из `map`
- `map.remove(key)` удаляет пару для ключа `key`

Преобразования между `Map` и `MutableMap`

Аналогично обычным и изменяемым спискам, обычные и изменяемые ассоциативные массивы могут быть преобразованы друг в друга при помощи функций конвертации `mutableMap.toMap()` или `map.toMutableMap()`. Каждая из функций создает **новый** объект на основе имеющегося --- значения остаются одни и те же, но тип у нового объекта будет соответствовать типу для фукции конвертации.

Множества

Теперь давайте рассмотрим такой вид структур данных как множества, которые представляют собой абстрацию, крайне близкую математическим множествам. Вспомним парочку определений.

В математике множеством называется набор каких-либо однотипных элементов, каждый из которых является **уникальным**, --- то есть, во множестве не может быть двух одинаковых элементов. Основная операция, которая представляет интерес для множеств, --- это операция вхождения одного множества в другое.

В программировании множества используются аналогичным образом. Множество `Set<T>` является набором уникальных с точки зрения равенства на `==` элементов типа `T`, и основная доступная операция --- включает или нет множество какой-то элемент.

Рассмотрим, как множества могут использоваться на практике. Пусть нам необходимо

взять какой-то текст (в виде списка слов) и убрать из него определенные слова-паразиты (например, "типа", "как бы", "короче"). Это можно сделать вот так.

```
fun removeFillerWords(
        text: List<String>,
        vararg fillerWords: String): List<String> {
    val fillerWordSet = setOf(*fillerWords)

    val res = mutableListOf<String>()
    for (word in text) {
        if (word !in fillerWordSet) {
            res += word
        }
    }
    return res
}
```

Наша функция принимает на вход текст `text` в виде списка строк и, в виде параметра переменной длины `fillerWords`, --- набор тех слов-паразитов, которые мы хотим из текста удалить. Первым делом мы строим из нашего параметра переменной длины множество слов-паразитов при помощи функции `setOf`. Обратите внимание, что здесь мы пользуемся оператором раскрытия `*` для передачи массива в эту функцию, чтобы итоговое множество было построено из переданных в функцию элементов.

После получения этого множества мы проходимся по всем элементам текста и, если элемент не является словом-паразитом (`word !in fillerWordSet`), мы добавляем его в список результата. Когда мы перебрали все элементы, мы возвращаем результат обратно.

NB: данная задача очень хорошо ложится в концепцию функций высших порядков, которую мы с вами обсуждали в прошлом уроке. С использованием функции `filter` наше решение будет выглядеть совсем просто:

```
fun removeFillerWords(
        text: List<String>,
        vararg fillerWords: String): List<String> {
    val fillerWordSet = setOf(*fillerWords)
    return text.filter { it !in fillerWordSet }
}
```

Попробуем написать тесты для нашей функции `removeFillerWords`.

```
@Test
@Tag("Example")
fun removeFillerWords() {
    assertEquals(
        "Я люблю Котлин".split(" "),
        removeFillerWords(
            "Я как-то люблю Котлин".split(" "),
            "как-то"
        )
    )
    assertEquals(
        "Я люблю Котлин".split(" "),
        removeFillerWords(
            "Я как-то люблю таки Котлин".split(" "),
            "как-то",
            "таки"
        )
    )
    assertEquals(
        "Я люблю Котлин".split(" "),
        removeFillerWords(
            "Я люблю Котлин".split(" "),
            "как-то",
            "таки"
        )
    )
}
```

При написании тестов мы используем функцию `str.split(delim1, delim2, ⋯)`, которая разбивает строку-получатель `str` на список строк по указанным строкам-разделителям `delimN`, как раз для получения списка строк, соответствующего какому-либо тексту. Если запустить наши тесты, то они --- ура-ура --- успешно пройдут.

Основной вопрос, который возникает при взгляде на наше решение, --- а зачем здесь множества? Почему нельзя было работать с оригинальным массивом слов-паразитов `fillerWords`? И действительно, если поменять решение соответствующим образом, то тесты все также будут проходить.

```
fun removeFillerWords(
        text: List<String>,
        vararg fillerWords: String): List<String> {
    val res = mutableListOf<String>()
    for (word in text) {
        if (word !in fillerWords) {
            res += word
        }
    }
    return res
}
```

Если подумать, то станет понятно, что массив или список с элементами, --- это тоже практически множество, необходимо только каким-либо образом обеспечить уникальность элементов. Тогда наш вопрос еще более актуален --- зачем вообще иметь отдельный, специальный тип для множества?

Тут мы с вами впервые знакомимся с таким понятием, как **эффективность** структуры данных. Решение на основе списка, конечно, работает, но **сложность** проверки того, входит или нет какой-то элемент в список, значительно больше, чем аналогичная сложность для множества `Set`. Это связано именно с тем, что `Set` специализирован для того, чтобы представлять множество; и все типичные для множества операции реализованы как можно более эффективно.

Более подробно вопросы эффективности вы будете изучать дальше на вашем пути обучения программированию, пока что можно запомнить очень простую идею --- множество элементов лучше представлять как множество типа `Set`.

Распространенные операции над множествами

Рассмотрим основные операции, доступные над множествами.

- `set.size / set.count()` возвращает количество элементов в множестве
- `e in set / set.contains(e)` проверяет, содержится ли элемент `e` во множестве `set`
- `set.intersect(otherSet)` осуществляет пересечение множеств
- `set.union(otherSet)` объединяет два множества
- `set + e / set + array / set + list` создает новое множество с добавленным элементом или элементами
- `set - e / set - array / set - list` возвращает множество, из которого удалены указанные элементы

Все операции поддерживают уникальность элементов в результирующем множестве автоматически.

Изменяемые множества

"И в третий раз..." мы с вами вспомнили о том, что иногда нам хочется **изменять** объекты в программе, в том числе, множества. По аналогии с предыдущими случаями, тип изменяемого множества --- это `MutableSet<T>`, и он расширяет тип обычного множества, добавляя операции по добавлению и удалению из множества элементов.

Представим, что вам нужно решить следующую задачу, опять же над текстом в виде списка строк: построить набор уникальных слов, которые встречаются в тексте. Одно из возможных решений выглядит следующим образом.

```
fun buildWordSet(text: List<String>): MutableSet<String> {
    val res = mutableSetOf<String>()
    for (word in text) res.add(word)
    return res
}
```

Для добавления новых слов в изменяемое множество, которое является результатом, мы используем функцию `set.add(word)`. Поддержание уникальности содержащихся в множестве элементов выполняется автоматически. В остальном, наше решение должно быть достаточно понятным для вас без дополнительных объяснений.

Посмотрим на тесты для нашей функции.

```
@Test
@Tag("Example")
fun buildWordSet() {
    assertEquals(
        buildWordSet("Я люблю Котлин".split(" ")),
        mutableSetOf("Я", "люблю", "Котлин")
    )
    assertEquals(
        buildWordSet("Я люблю люблю Котлин".split(" ")),
        mutableSetOf("Котлин", "люблю", "Я")
    )
    assertEquals(
        buildWordSet(listOf()),
        mutableSetOf<String>()
    )
}
```

На что можно обратить здесь внимание? В отличие от списков, для равенства множеств порядок элементов **не является важным**; причем это верно как для изменяемых, так и для неизменяемых множеств. Это напрямую следует из свойств множеств из математики, где равенство множеств работает именно так.

Подобный **перенос** свойств объекта из какой-либо предметной области в

программирование является одним из ключевых его (программирования) моментов. Когда вы используете или создаете абстракции (например, структуры данных) для решения задачи, вы **переводите** предметную область задачи в язык, понятный компьютеру; вместе с тем этот перенос должен сохранять свойства, важные для предметной области. Умение сделать это наиболее просто и эффективно придет с опытом.

Распространенные операции над изменяемыми множествами

Рассмотрим основные операции, доступные над изменяемыми множествами.

- `set.add(element)` добавляет элемент в множество
- `set.addAll(listOrSet)` добавляет все элементы из заданного набора элементов
- `set.remove(element)` удаляет элемент из множества
- `set.removeAll(listOrSet)` удаляет все элементы из заданного набора элементов
- `set.retainAll(listOrSet)` оставляет в множестве только элементы, которые есть в заданном наборе элементов
- `set.clear()` удаляет из множества все элементы

Как и раньше, поддержание уникальности элементов выполняется автоматически.

Операции над `null`

Напоследок давайте чуть ближе познакомимся с объектом `null` --- тем самым специальным значением, которое означает **отсутствие** чего-то в ассоциативном массиве. Данная "пустота" в Котлине не может появиться и использоваться просто так; если вы попробуете, например, присвоить `null` в переменную типа `Int`, то у вас ничего не получится. Дело в том, что значение `null` является допустимым только для специальных `nullable` типов; все обычные типы по умолчанию являются `non-nullable`.

Каким образом можно сделать `nullable` тип? Очень просто --- если вы хотите сделать `nullable` версию `Int`, то нужно написать `Int?`. Знак вопроса, обычно выражающий сомнение, в данном контексте делает то же самое --- сигнализирует, что этот тип может как иметь нормальное значение, так и значение `null`.

Есть ли еще какая-либо разница между типами `Int` и `Int?`, кроме того, что во втором может храниться `null`? Да, разница есть, и она заключается в том, что многие операции, возможные над `Int`, нельзя выполнить просто так над `Int?`. Представим, что мы хотим сложить два `Int?`.

```
fun addNullables(a: Int?, b: Int?): Int = a + b  // ERROR
```

Данный код **не будет работать** аж с целыми двумя ошибками: "Operator call corresponds to a dot-qualified call 'a.plus(b)' which is not allowed on a nullable receiver 'a'" и "Type mismatch:

inferred type is Int? but Int was expected". Эти ошибки вызваны как раз тем, что в переменной с типом `Int?` может храниться `null`, а как сложить что-то с тем, чего нет?

Так как операции с `nullable` типами являются потенциально опасными, в Котлине для работы с ними есть специальные **безопасные** операции и операторы, которые учитывают возможность появления `null`. Одним из таких операторов является элвис-оператор `?:`, названный так в честь схожести с прической короля рок-н-ролла Элвиса Пресли. Рассмотрим, как он работает.

Выражение `a ?: valueIfNull` возвращает `a` в случае, если `a` не равно `null`, и `valueIfNull` в противном случае. Это позволяет предоставить "значение по умолчанию" для случая, когда в переменной хранится `null`. В нашем случае сложения двух чисел мы можем считать, что если какого-то числа нет (`null`), то оно равно нулю.

```
fun addNullables(a: Int?, b: Int?): Int = (a ?: 0) + (b ?: 0)
```

Еще один `null`-специфичный оператор --- это оператор безопасного вызова `?.`. Он используется в случаях, когда необходимо **безопасно** вызвать функцию над объектом, который может быть `null`. Выражение `a?.foo(b, c)` возвращает результат вызова функции `foo` с аргументами `b` и `c` над получателем `a`, если `a` не равен `null`; в противном случае возвращается `null`. Пусть нам нужно вернуть сумму элементов в `nullable` списке.

```
fun sumOfNullableList(list: List<Int>?): Int = list?.sum()  // ERROR
```

Такой код не будет работать, потому что `list?.sum()` может вернуть `null`. Если подсмотреть в IntelliJ IDEA, то можно увидеть, что тип такого выражения, --- `Int?`; чтобы исправить ситуацию с типом возвращаемого значения, можно воспользоваться элвис-оператором.

```
fun sumOfNullableList(list: List<Int>?): Int = list?.sum() ?: 0
```

Третий оператор, относящийся к `null`, но не являющийся безопасным, --- это оператор `!!`. Его смысл очень прост --- он делает из `nullable` выражения `non-nullable` выражение. В случае, если выражение имеет нормальное значение, эта операция завершается успешно. А вот если в выражении был `null`, это приводит к ошибке `NullPointerException`; по этой причине использовать этот оператор можно только тогда, когда вы уверены в том, что выражение не содержит `null`. Например, пусть вы работаете с ассоциативным массивом следующим образом.

```
val map = getMapOfNumbers()
if (map[key] != null) {
    val correctedNumber = map[key] + correction // ERROR
    // ...
}
```

Несмотря на то, что мы проверили значение в `if`, Котлин считает, что `map[key]` может

вернуть `null` и выдает ошибку компиляции. Если мы считаем, что значение действительно не может поменяться, то можно воспользоваться `!!`.

```
val map = getMapOfNumbers()
if (map[key] != null) {
    val correctedNumber = map[key]!! + correction
    // ...
}
```

Кто-то может спросить: подождите, мы в самом начале этого урока делали ровно такую же операцию, и никакого оператора `!!` там не было. Вспомним, о чем идет речь.

```
fun shoppingListCost(
        shoppingList: List<String>,
        costs: Map<String, Double>): Double {
    var totalCost = 0.0

    for (item in shoppingList) {
        val itemCost = costs[item]
        if (itemCost != null) {
            totalCost += itemCost // No `!!` operator
        }
    }

    return totalCost
}
```

Что здесь происходит? Тут нам помогает такая вещь как "умные приведения типов" или смарт-касты. Компилятор Котлина, увидев, что **неизменяемое** выражение `itemCost` проверили на неравенство `null`, "стирает" с его типа знак вопроса внутри `if`; именно поэтому `itemCost` можно использовать без каких-либо безопасных операторов. Если присмотреться, то IntelliJ IDEA специальным образом подсвечивает подобные ситуации в редакторе кода.

Почему это не работает для `map[key]`? Именно потому что выражение `map[key]` не является неизменяемым, то есть результат его вычисления может быть разным в разные моменты времени; для того, чтобы сохранить безопасность кода, компилятор не делает никаких опасных предположений и отдает всю ответственность вам.

Если попробовать описать правила работы с `null` в компактном виде, то они могут выглядеть следующим образом.

- Если у вас **нет** никакого осмысленного значения по умолчанию для объекта, проверьте на `null` в `if` или `when` и воспользуйтесь смарт-кастами
- Если у вас **есть** какое-либо значение по умолчанию, можно применить элвис-оператор
- Если вы хотите вызвать функции над `nullable` объектом, воспользуйтесь оператором безопасного вызова

- Если вы точно-точно знаете, что `nullable` объект на самом деле не может содержать `null`, можете применить оператор `!!`

Этими правилами покрываются 99 из 100 ситуаций, с которыми вы можете столкнуться при программировании на Котлине. К тому моменту, как вы окажетесь в той самой "1 из 100" ситуации, вы уже будете разбираться в программировании достаточно, чтобы справиться с ней самостоятельно.

Упражнения

Откройте файл `src/lesson5/task1/Map.kt` в проекте `KotlinAsFirst`. Выберите любую из задач в нём. Придумайте её решение и запишите его в теле соответствующей функции.

Откройте файл `test/lesson5/task1/Tests.kt`, найдите в нём тестовую функцию — её название должно совпадать с названием написанной вами функции. Запустите тестирование, в случае обнаружения ошибок исправьте их и добейтесь прохождения теста. Подумайте, все ли необходимые проверки включены в состав тестовой функции, добавьте в неё недостающие проверки.

После этого решите еще одну или несколько задач из урока 5. Обратите внимание, что некоторые задачи (`propagateHandshakes`, `findSumOfTwo` и `bagPacking`) действительно являются сложными и могут для своего решения потребовать от вас знакомства с дополнительными материалами. Если вы очень хотите решить эти задачи, но самостоятельно у вас это не получается, попробуйте поискать возможные подходы к их решению в Интернете. Убедительная просьба не слепо копировать готовое решение, а постараться разобраться в нем и понять его основную идею.

Отдельно отметим, что, как мы обсуждали с вами в уроке, многие из задач урока 5 могут быть решены без использования множеств или ассоциативных массивов, однако постарайтесь все же использовать именно них. Подумайте, какие вычисления вы можете не делать при использовании множеств или ассоциативных массивов? Как вам кажется, делает ли это ваше решение более эффективным?

Убедитесь в том, что можете решать задачи с использованием множеств и/или ассоциативных массивов уверенно и без посторонней помощи. После этого переходите к следующему разделу.

Дополнительное чтение

- Википедия. "Динамическое программирование"