Лекция 8. Основы Kotlin

Лекция 9. Основы Kotlin.

1. Представление и обработка выбора: перечисления и конструкция «when»
2. Итерации: циклы «while» и «for»
3. Исключения в Kotlin
4. Создание коллекций в Kotlin

```
enum class Color {
  RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

```
enum class Color(
  val r: Int, val g: Int, val b: Int
) {
  RED(255, 0, 0), ORANGE(255, 165, 0),
  YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
  INDIGO(75, 0, 130), VIOLET(238, 130, 238);

  fun rgb() = (r * 256 + g) * 256 + b

}
>>> println(Color.BLUE.rgb())
255
```

Значения свойств определяются для каждой константы

Объявление свойств констант перечисления

Точка с запятой здесь обязательна

Определение метода класса перечисления

```
fun getMnemonic(color: Color) =
            when (color) {
                Color.RED -> "Каждый"
                Color.ORANGE -> "Охотник"
                Color.YELLOW -> "Желает"
                Color.GREEN -> "Знать"
                Color.BLUE -> "Где"
                Color.INDIGO -> "Сидит"
                Color.VIOLET -> "Фазан"
            }
>>> println(getMnemonic(Color.BLUE))
Где
```

Сразу возвращает выражение «when»

Возвращает соответствующую строку, если цвет совпадает с константой перечисления

```
fun getWarmth(color: Color) = when(color) {
  Color.RED, Color.ORANGE, Color.YELLOW -> "теплый"
  Color.GREEN -> "нейтральный"
  Color.BLUE, Color.INDIGO, Color.VIOLET -> "холодный"
}

>>> println(getWarmth(Color.ORANGE))
теплый
```

```
import ch02.colors.Color
import ch02.colors.Color.*

fun getWarmth(color: Color) = when(color) {
  RED, ORANGE, YELLOW -> "теплый"
```

Обращение к импортированным константам по именам

Импорт класса Color, объявленный в другом пакете

Явный импорт констант перечисления для обращения к ним по именам

```
fun mix(c1: Color, c2: Color) =
  when (setOf(c1, c2)) {
    setOf(RED, YELLOW) -> ORANGE
    setOf(YELLOW, BLUE) -> GREEN
    setOf(BLUE, VIOLET) -> INDIGO
    else -> throw Exception("Грязный цвет")
  }

>>> println(mix(BLUE, YELLOW))
GREEN
```

Перечисление пар цветов, пригодных для смешивания

Аргументом выражения «when» может быть любой объект. Он проверяется условными выражениями ветвей

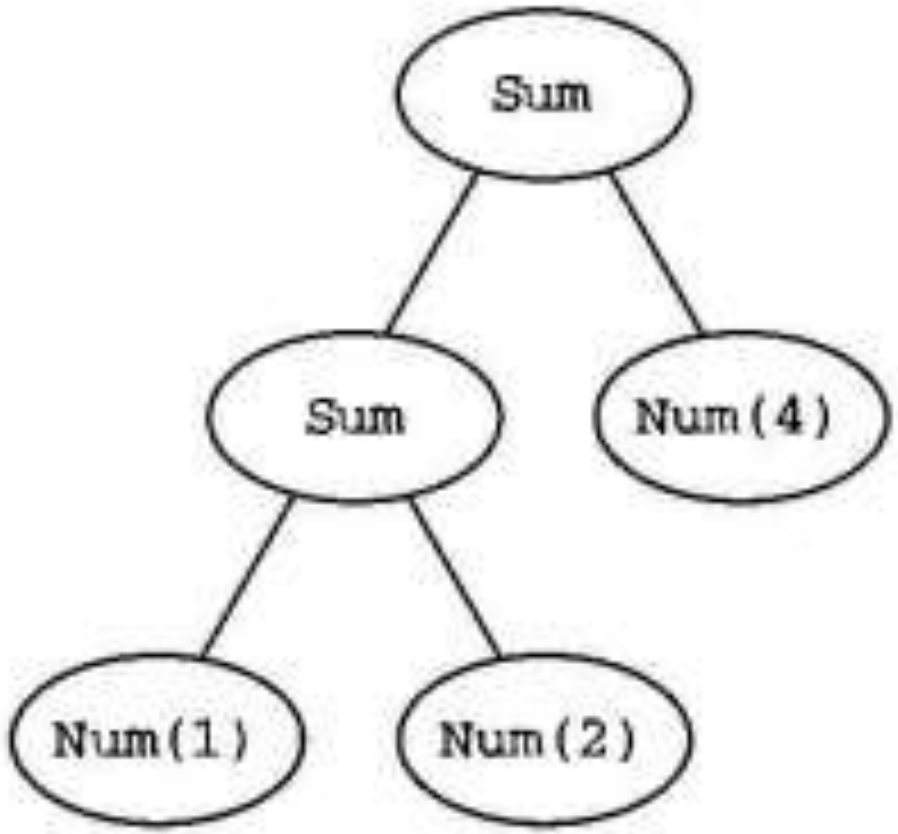
Выполняется, если не соответствует ни одной из ветвей

```
fun mixOptimized(c1: Color, c2: Color) =
    when {                                          <— Выражение «when» без аргумента
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE
        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN
        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO
        else -> throw Exception("Dirty color")
    }
>>> println(mixOptimized(BLUE, YELLOW))
GREEN
```

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr
```

Простой класс объектов-значений с одним свойством value, реализующий интерфейс Expr

Аргументами операции Sum могут быть любые экземпляры Expr: Num или другой объект Sum



```
fun eval(e: Expr): Int {
  if (e is Num) {
    val n = e as Num      <— Явное приведение к типу Num здесь излишне
    return n.value
  }
  if (e is Sum) {
    return eval(e.right) + eval(e.left)  <— Переменная e уже приведена к нужному типу!
  }
  throw IllegalArgumentException("Unknown expression")
}

>>> println(eval(Sum(Sum(Num(1), Num(2)), Num(4))))
7
```

```
fun eval(e: Expr): Int =
  if (e is Num) {
    e.value
  } else if (e is Sum) {
    eval(e.right) + eval(e.left)
  } else {
    throw IllegalArgumentException("Unknown expression")
  }

>>> println(eval(Sum(Num(1), Num(2))))
3
```

```
fun eval(e: Expr): Int =
  when (e) {
    is Num ->                          <— Ветка «when» проверяет тип аргумента
      e.value                          <— Используется автоматическое приведение типов
    is Sum ->                                  <— Ветка «when» проверяет тип аргумента
      eval(e.right) + eval(e.left)             <— Используется автоматическое приведение типов
    else ->
      throw IllegalArgumentException("Unknown expression")
  }
```

```
fun evalWithLogging(e: Expr): Int =
  when (e) {
    is Num -> {
      println("num: ${e.value}")
      e.value
    }
    is Sum -> {
      val left = evalWithLogging(e.left)
      val right = evalWithLogging(e.right)
      println("sum: $left + $right")
      left + right
    }
    else -> throw IllegalArgumentException("Unknown expression")
  }
```

Это последнее выражение в блоке, функция вернет его значение, если e имеет тип Num

Функция вернет значение этого выражения, если e имеет тип Sum

```
while (condition) {
  /*...*/
}

do {
  /*...*/
} while (condition)
```

Тело цикла выполняется, пока условие остается истинным

Тело выполняется первый раз безусловно. После этого оно выполняется, пока условие остается истинным

```
fun fizzBuzz(i: Int) = when {
  i % 15 == 0 -> "FizzBuzz "
  i % 3 == 0 -> "Fizz "
  i % 5 == 0 -> "Buzz "
  else -> "$i "
}

>>> for (i in 1..100) {
...     print(fizzBuzz(i))
...   }

1 2 Fizz 4 Buzz Fizz 7 …
```

Если i делится на 15, вернуть «FizzBuzz». Так же как в Java, % - это оператор деления по модулю (остаток от деления нацело)

Если i делится на 3, вернуть «Fizz»

Если i делится на 5, вернуть «Buzz»

Ветвь else возвращает само число

Выполнить обход диапазона от 1 до 100

```
>>> for (i in 100 downTo 1 step 2) {
    ... print(fizzBuzz(i))
    ... }
Buzz 98 Fizz 94 92 FizzBuzz 88 …
```

```
val binaryReps = TreeMap<Char, String>()
for (c in 'A'..'F') {
  val binary = Integer.toBinaryString(c.toInt())
  binaryReps[c] = binary
}

for ((letter, binary) in binaryReps) {
  println("$letter = $binary")
}
```

Словарь TreeMap хранит ключи в порядке сортировки

Обход диапазона символов от A до F

Преобразует ASCII-код в двоичное представление

Сохраняет в словаре значение с ключом в c

Обход элементов словаря; ключ и значение присваиваются двум переменным

```
val list = arrayListOf("10", "11", "1001")
for ((index, element) in list.withIndex()) {
  println("$index: $element")
}
```

Обход коллекции с сохранением индекса

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'

>>> println(isLetter('q'))
true
>>> println(isNotDigit('x'))
true
```

```
fun recognize(c: Char) = when (c) {
  in '0'..'9' -> "It's a digit!"
  in 'a'..'z', in 'A'..'Z' -> "It's a letter!"
  else -> "I don't know..."
}

>>> println(recognize('8'))
It's a digit!
```

Проверяет вхождение значения в диапазон от 0 до 9

Можно совместить несколько диапазонов

```
if (percentage !in 0..100) {
  throw IllegalArgumentException(
    "A percentage value must be between 0 and 100: $percentage")
}
```

```
fun readNumber(reader: BufferedReader): Int? {
  try {
    val line = reader.readLine()
    return Integer.parseInt(line)
  }
  catch (e: NumberFormatException) {
    return null
  }
  finally {
    reader.close()
  }
}
```

Не требуется явно указывать, какое исключение может возбудить функция

Тип исключения записывается справа

Блок «finally» действует так же, как в Java

```
>>> val reader = BufferedReader(StringReader("239"))
>>> println(readNumber(reader))
239
```

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    }
    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
```

Получит значение выражения «try»

Ничего не выведет

Создание массива:

```
var array = arrayOf(1, 3, 2)
```

Создание массива, инициализированного null:

```
var nullArray: Array<String?> = arrayOfNulls(2)
```

Создает массив с размером 2, инициализированный значениями null. То же самое, что arrayOf(null, null)

Определение размера массива:

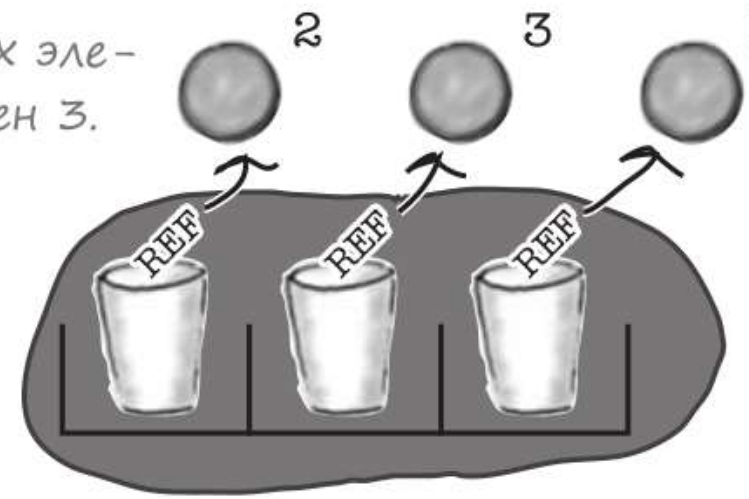
```
val size = array.size
```

Массив содержит место для трех элементов, поэтому его размер равен 3.

Перестановка элементов в обратном порядке:

```
array.reverse()
```

Переставляет элементы массива в обратном порядке.



Проверка присутствия заданного значения:

```
val isIn = array.contains(1)
```

Массив содержит 1, поэтому функция возвращает true.

Вычисление суммы элементов (для числовых массивов):

```
val sum = array.sum()
```
← Возвращает 6, так как 2 + 3 + 1 = 6.

Вычисление среднего значения элементов (для числовых массивов):

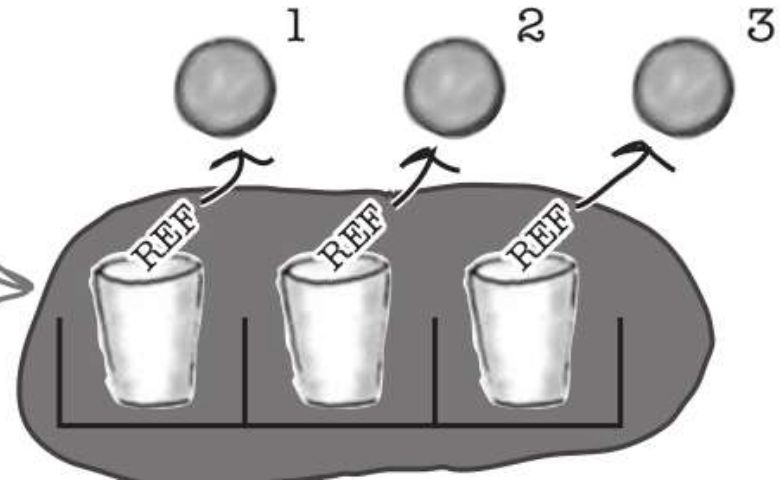
```
val average = array.average()
```
← Возвращает Double — в данном случае (2 + 3 + 1)/3 = 2.0.

Поиск наименьшего или наибольшего элемента (работает для чисел, String, Char и Boolean):

```
array.min()
array.max()
```
} min() возвращает 1, так как это наименьшее значение в массиве. max() возвращает 3 – наибольшее значение.

Сортировка массива в естественном порядке (работает для чисел, String, Char и Boolean):

```
array.sort()
```
← Изменяет порядок элементов в массиве, чтобы они следовали от наименьшего значения к наибольшему, или от false к true.



все же массивы не идеальны.

Размер массива не может изменяться динамически

При создании массива компилятор определяет его размер по количеству элементов, с которыми он инициализируется. Этот размер фиксируется раз и навсегда. Массив не увеличится, если вы захотите добавить в него новый элемент, и не уменьшится, если вы захотите элемент удалить.

Изменяемость массивов

Другое ограничение заключается в том, что после создания массива вы не можете предотвратить его модификацию. Если вы создаете массив кодом следующего вида:

```
val myArray = arrayOf(1, 2, 3)
```

ничто не помешает обновить его содержимое:

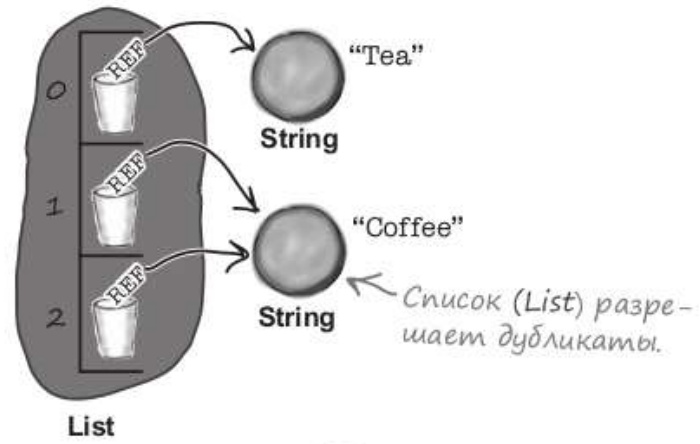
```
myArray[0] = 6
```

List, Set и Map

В Kotlin существуют три основных типа коллекций — **`List`**, **`Set`** и **`Map`**. Каждый тип имеет четко определенное предназначение:

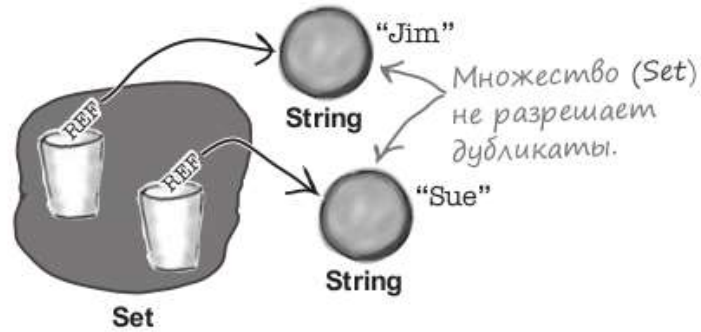
List — когда важен порядок

`List` хранит и отслеживает позицию элементов. Она знает, в какой позиции списка находится тот или иной элемент, и несколько элементов могут содержать ссылки на один объект.



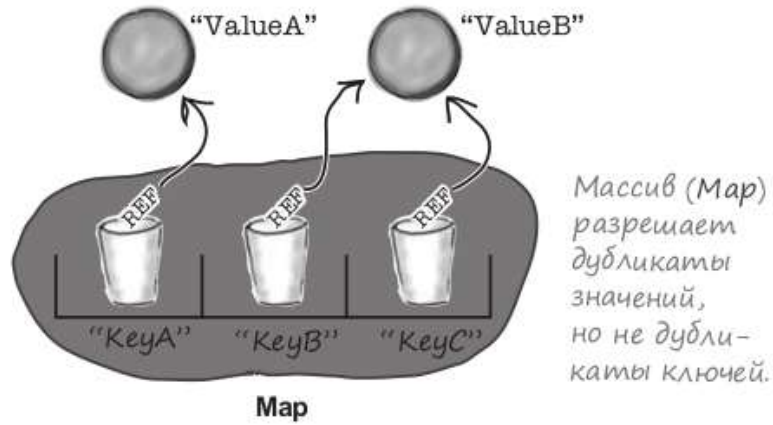
Set — когда важна уникальность

`Set` не разрешает дубликаты и не отслеживает порядок, в котором хранятся значения. Коллекция не может содержать несколько элементов, ссылающихся на один и тот же объект, или несколько элементов, ссылающихся на два объекта, которые считаются равными.



Map — когда важен поиск по ключу

`Map` использует пары «ключ-значение». Этот тип коллекции знает, какое значение связано с заданным ключом. Два ключа могут ссылаться на один объект, но дубликаты ключей невозможны. Хотя ключи обычно представляют собой строковые имена (например, для составления списков свойств в формате «имя-значение»), ключом также может быть произвольный объект.



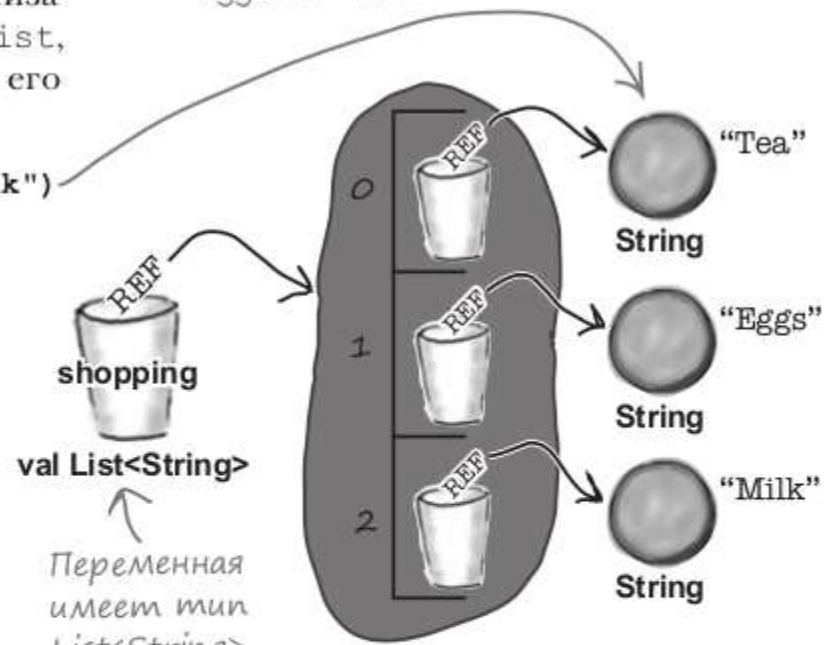
Простые коллекции `List`, `Set` и `Map` *неизменяемы*; это означает, что после инициализации коллекции вы уже не сможете добавлять или удалять элементы. Если вам необходимо добавлять или удалять элементы, Kotlin предоставляет изменяемые версии подклассов: **`MutableList`**, **`MutableSet`** и **`MutableMap`**. Например, если вы хотите пользоваться всеми преимуществами `List` и при этом иметь возможность обновлять его содержимое, выберите `MutableList`.

Итак, теперь вы знаете три основных типа коллекций из стандартной библиотеки Kotlin. Посмотрим, как использовать каждый из них. Начнем с `List`.

Создание списка **List** имеет много общего с созданием массива: в программе вызывается функция с именем **listOf**, которой передаются значения для инициализации элементов. Например, следующий код создает `List`, инициализирует его тремя строками и присваивает его новой переменной с именем `shopping`:

```
val shopping = listOf("Tea", "Eggs", "Milk")
```

Этот фрагмент создает объект List со строковыми значениями «Tea», «Eggs» и «Milk».



Переменная имеет тип List<String>, так что List содержит ссылки на объекты String.

Компилятор определяет тип объекта, который должен содержаться в списке `List`, по типам всех значений, переданных при создании. Например, список в нашем примере инициализируется тремя строками, поэтому компилятор создает `List` с типом `List<String>`. Тип `List` также можно задать явно:

```
val shopping: List<String>
shopping = listOf("Tea", "Eggs", "Milk")
```

...и как ими пользоваться

После того как объект `List` будет создан, вы сможете обращаться к содержащимся в нем элементам функцией **get**. Например, следующий код проверяет, что размер `List` больше 0, после чего выводит элемент с индексом 0:

```
if (shopping.size > 0) {
    println(shopping.get(0))
    //Выводит "Tea"
}
```

Размер List желательно проверять заранее, так как при передаче недействительного индекса get() выдаст исключение ArrayIndexOutOfBoundsException.

Перебор всех элементов List выполняется следующим кодом:

```
for (item in shopping) println (item)
```

Вы также можете проверить, содержит ли List ссылку на конкретный объект, и получить индекс соответствующего элемента:

```
if (shopping.contains("Milk")) {
    println(shopping.indexOf("Milk"))
    //Выводит 2
}
```

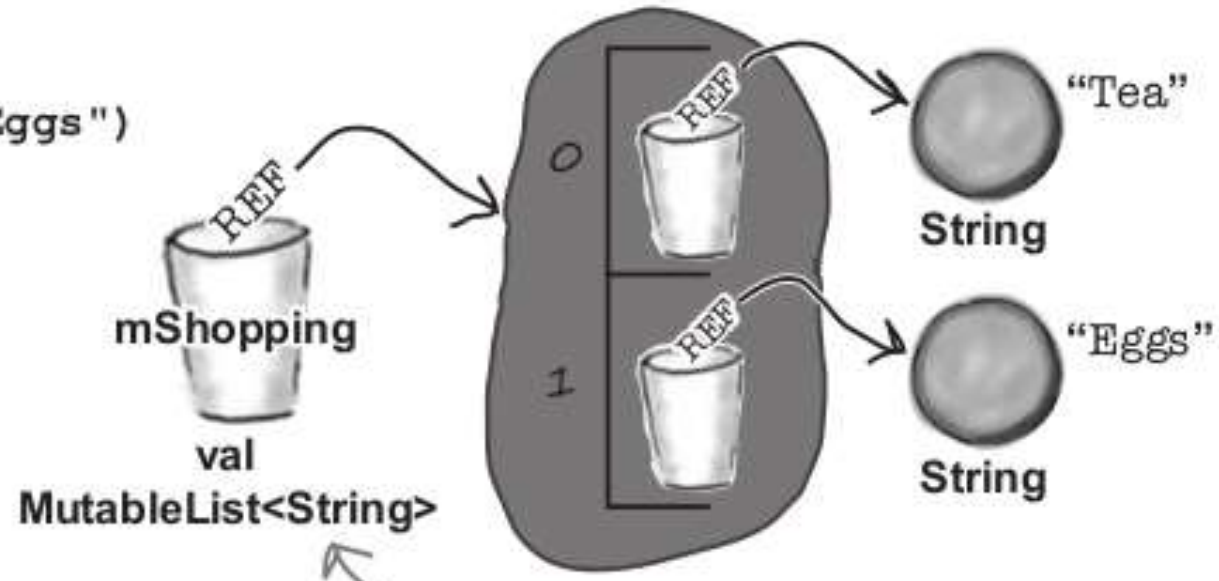
Как видите, в программах списки List используются практически так же, как и массивы. Но между List и массивами также существует серьезное различие: списки List неизменяемы — хранящиеся в них ссылки невозможно обновить.

В List и других коллекциях могут храниться ссылки на объекты любых типов: String, Int, Duck, Pizza и т. д.

Если вам нужен список с возможностью обновления элементов, используйте **MutableList**. Объект MutableList определяется почти так же, как определяется List, но только в этом случае используется функция **mutableListOf**:

```
val mShopping = mutableListOf("Tea", "Eggs")
```

MutableList является подклассом List, поэтому для MutableList можно вызывать те же функции, что и для List. Однако у MutableList есть дополнительные функции, которые используются для добавления, удаления, обновления или перестановки существующих значений.



Если передать функции mutableListOf() строковые значения, компилятор определяет, что вам нужен объект типа MutableList<String> (MutableList для хранения String).

...и добавьте в него значения

Новые элементы добавляются в MutableList функцией **add**. Чтобы добавить новое значение в конец MutableList, передайте значение функции add в единственном параметре. Например, следующий код добавляет строку «Milk» в конец mShopping:

```
mShopping.add("Milk")
```

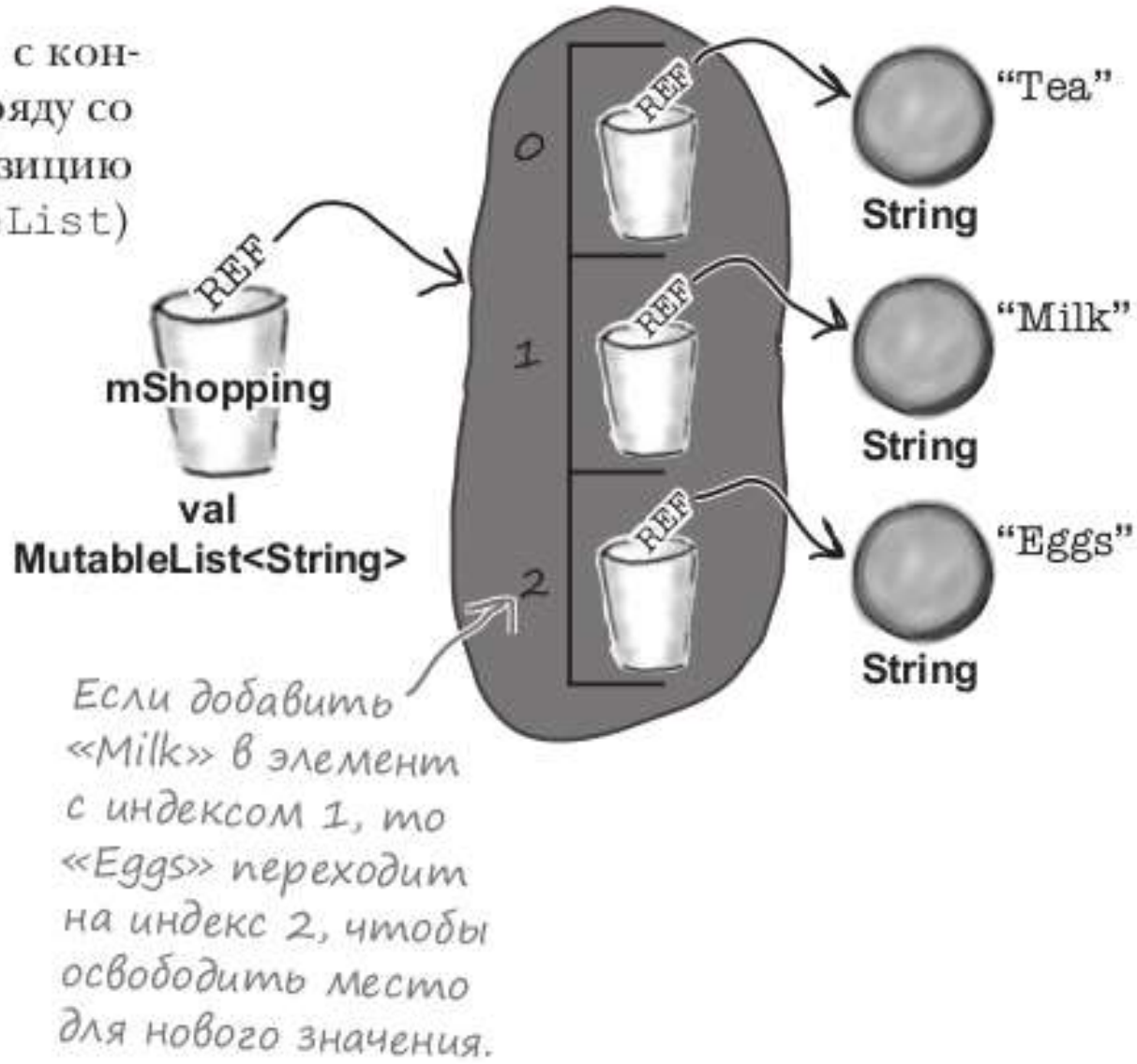
Размер `MutableList` увеличивается, чтобы в списке хранились три значения вместо двух.

Если же вы хотите вставить значение в позицию с конкретным индексом, передайте индекс функции наряду со значением. Например, вставка значения «Milk» в позицию с индексом 1 (вместо добавления в конец `MutableList`) выполняется так:

```
mShopping.add(1, "Milk")
```

При вставке значения с конкретным индексом другие значения сдвигаются и освобождают для него место. Так, в нашем примере значение «Eggs» перемещается из позиции с индексом 1 в позицию с индексом 2, чтобы значение «Milk» можно было вставить в позиции с индексом 1.

Кроме добавления значений в `MutableList`, элементы также можно удалять и заменять. Давайте посмотрим, как это делается.

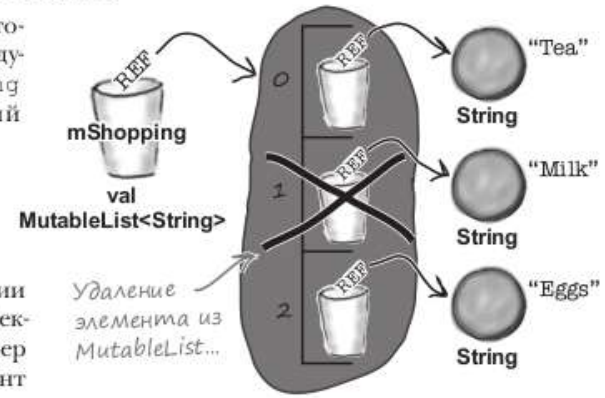


Значения можно удалять...

Существуют два способа удаления значений из `MutableList`.

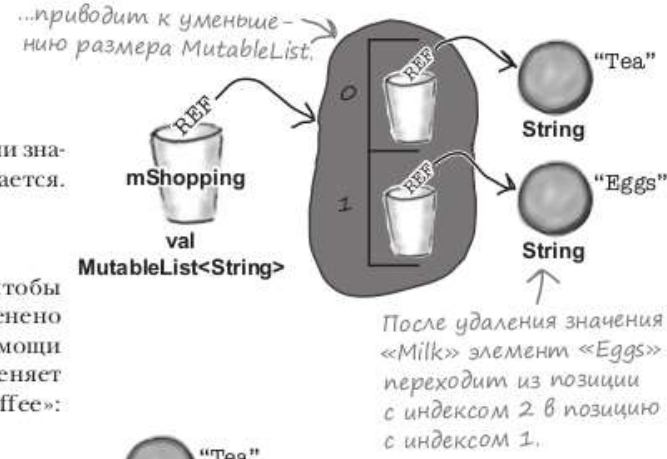
В первом способе вызывается функция **`remove`**, которой передается удаляемое значение. Например, следующий код проверяет, содержит ли список `mShopping` строку «Milk», после чего удаляет соответствующий элемент:

```
if (mShopping.contains("Milk")) {
    mShopping.remove("Milk")
}
```



Второй способ основан на использовании функции **`removeAt`** для удаления значения с заданным индексом. Например, следующий код проверяет, что размер `mShopping` больше 1, после чего удаляет элемент с индексом 1:

```
if (mShopping.size > 1) {
    mShopping.removeAt(1)
}
```

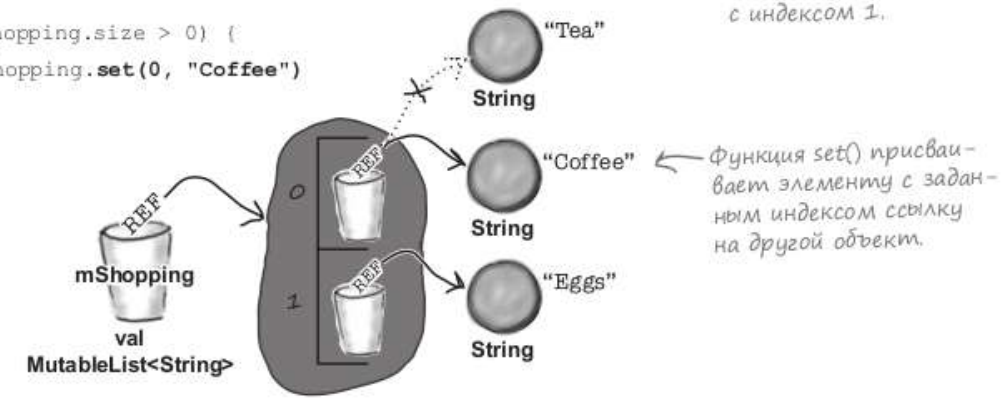


Какой бы способ вы ни выбрали, при удалении значения из `MutableList` размер списка уменьшается.

...и заменять их другими значениями

Если вы хотите обновить `MutableList` так, чтобы значение с конкретным индексом было заменено другим значением, это можно сделать при помощи функции **`set`**. Например, следующий код заменяет значение «Tea» с индексом 0 строкой «Coffee»:

```
if (mShopping.size > 0) {
    mShopping.set(0, "Coffee")
}
```



```
mShopping.sort()
mShopping.reverse()
```
Этот фрагмент сортирует MutableList в обратном порядке.

Или сгенерировать случайную перестановку функцией **shuffle**:

```
mShopping.shuffle()
```

Также существуют полезные функции для внесения массовых изменений в `MutableList`. Например, функция **addAll** добавляет все элементы, хранящиеся в другой коллекции. Следующий код добавляет в `mShopping` значения «Cookies» и «Sugar»:

```
val toAdd = listOf("Cookies", "Sugar")
mShopping.addAll(toAdd)
```

Функция **removeAll** удаляет элементы, входящие в другую коллекцию:

```
val toRemove = listOf("Milk", "Sugar")
mShopping.removeAll(toRemove)
```

Функция **retainAll** оставляет все элементы, входящие в другую коллекцию, и удаляет все остальные:

```
val toRetain = listOf("Milk", "Sugar")
mShopping.retainAll(toRetain)
```

Также можно воспользоваться функцией **clear** для удаления всех элементов:

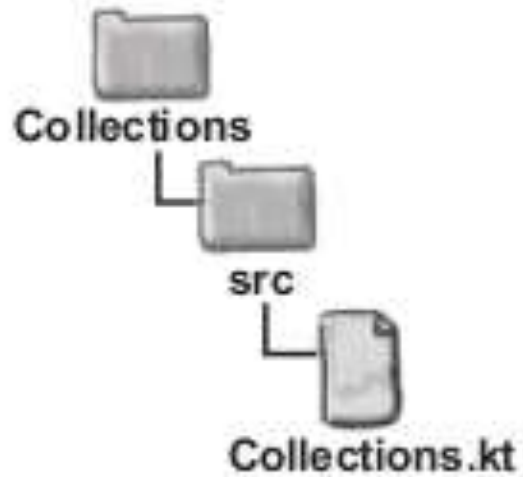
```
mShopping.clear()
```
Уничтожает содержимое mShopping, чтобы размер был равен 0.

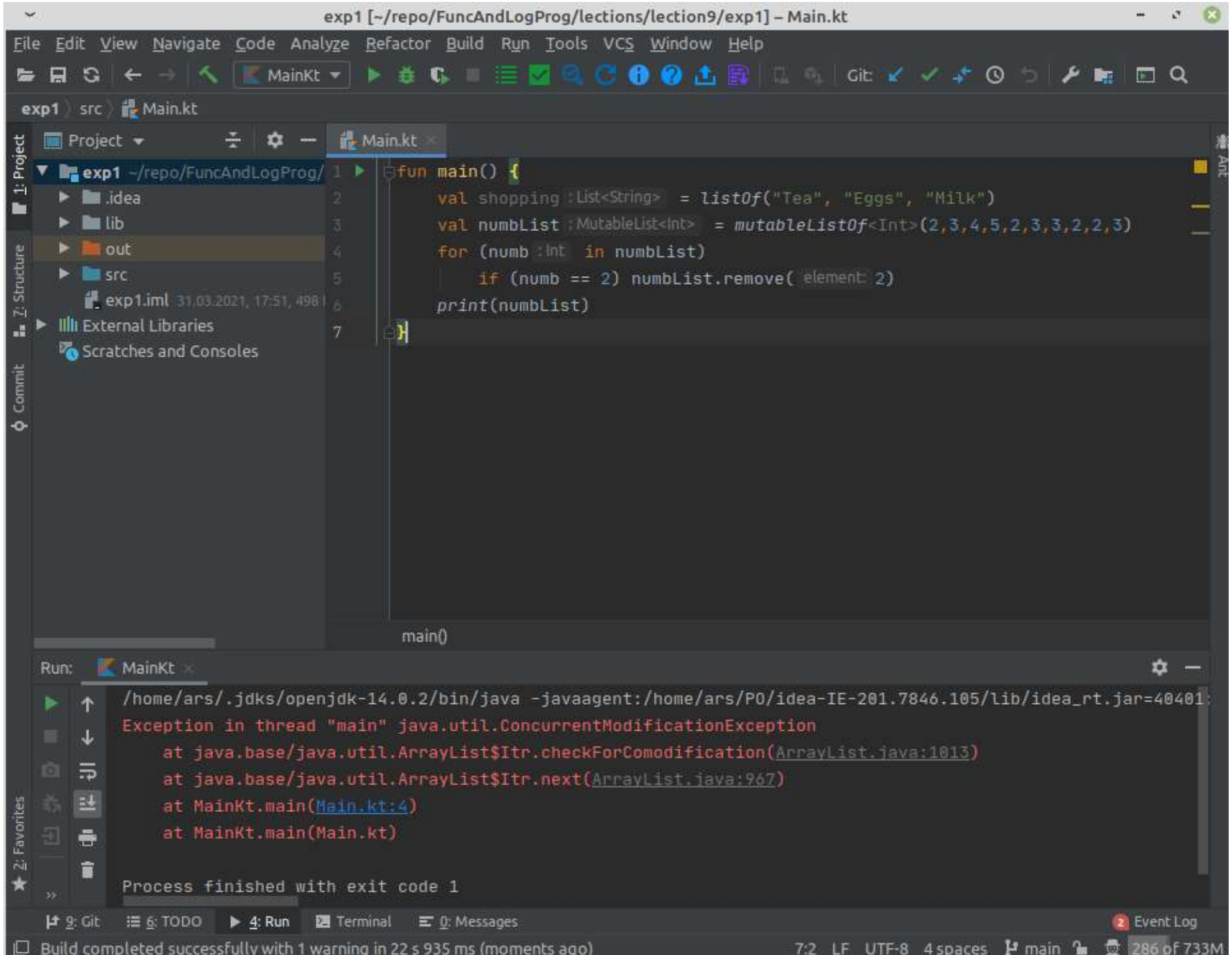
Иногда требуется скопировать `List` или `MutableList`, чтобы сохранить «снимок» содержимого списка. Для этой цели используется функция **`toList`**. Например, следующий код копирует `mShopping` и присваивает копию новой переменной с именем `shoppingSnapshot`:

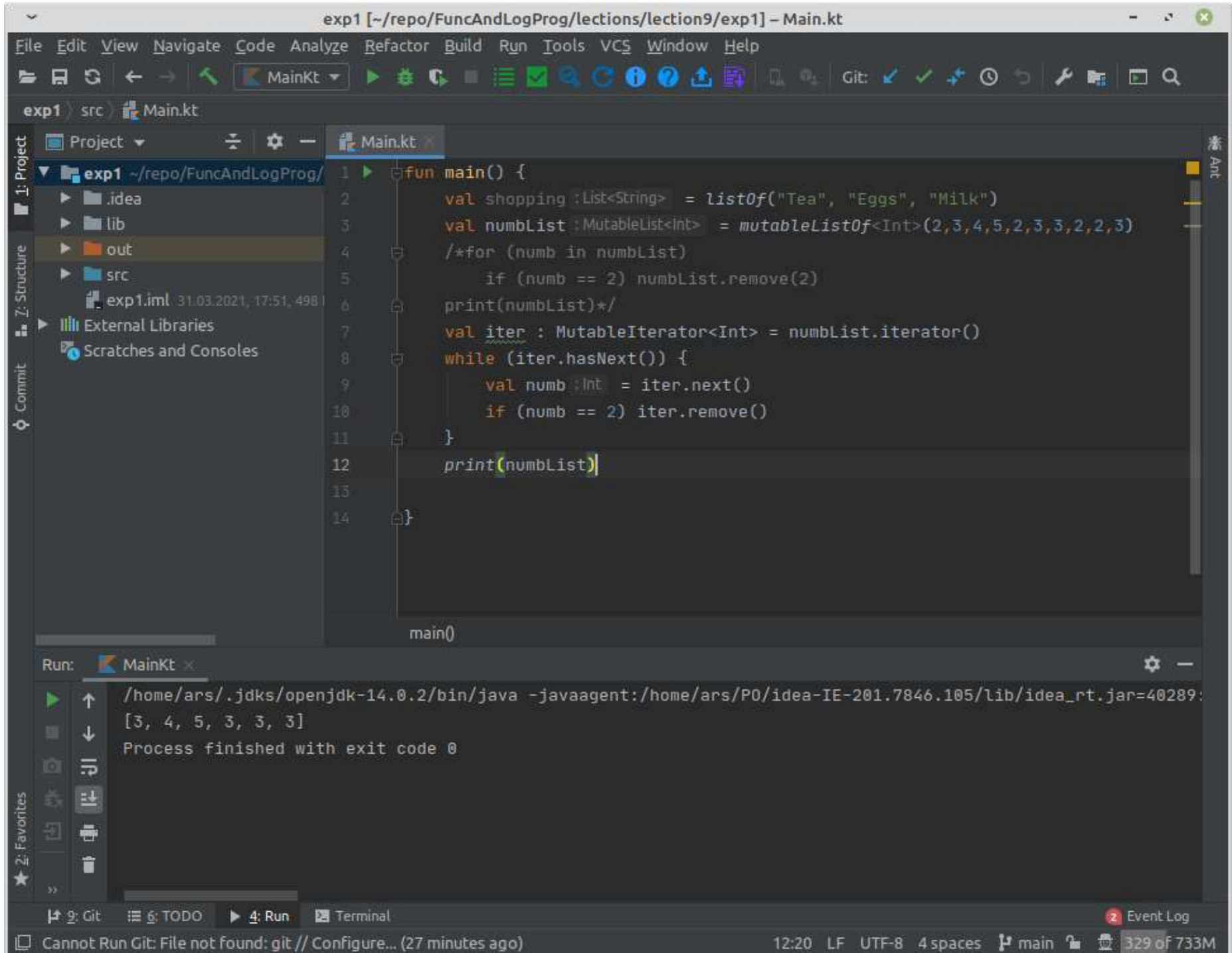
```
val shoppingCopy = mShopping.toList()
```

Функция `toList` возвращает `List`, а не `MutableList`, поэтому содержимое `shoppingCopy` обновить невозможно. Другие полезные функции, которые могут использоваться для копирования `MutableList`, — **`sorted`** (возвращает отсортированный список `List`), **`reversed`** (возвращает список `List` со значениями в обратном порядке) и **`shuffled`** (возвращает список `List` и генерирует случайную перестановку его значений).

```
fun main(args: Array<String>) {
    val mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")
}
```



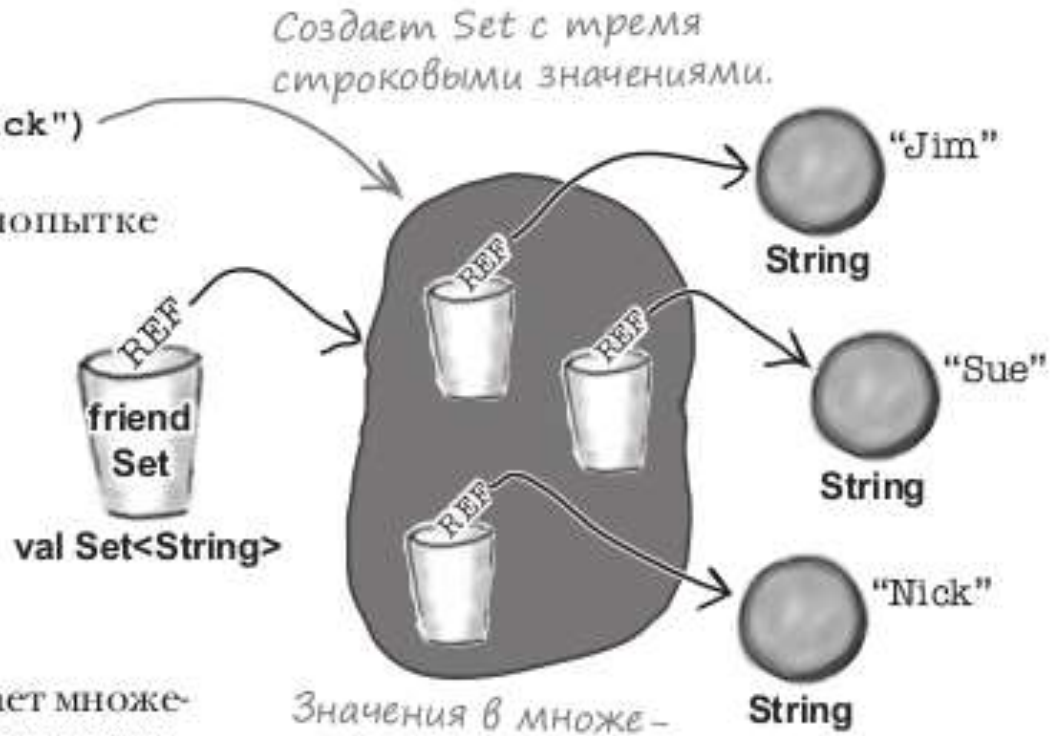

exp1 [~/repo/FuncAndLogProg/lections/lection9/exp1] – Main.kt
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MainKt
Git:
exp1 src Main.kt
Project
exp1 ~/repo/FuncAndLogProg/
.idea
lib
out
src
exp1.iml 31.03.2021, 17:51, 498
External Libraries
Scratches and Consoles
Main.kt
fun main() {
val shopping :List<String> = listOf("Tea", "Eggs", "Milk")
val numbList :MutableList<Int> = mutableListOf<Int>(2,3,4,5,2,3,3,2,2,3)
for (numb :Int in numbList)
if (numb == 2) numbList.remove(element: 2)
print(numbList)
}
main()
Run: MainKt
/home/ars/.jdks/openjdk-14.0.2/bin/java -javaagent:/home/ars/PO/idea-IE-201.7846.105/lib/idea_rt.jar=40401
Exception in thread "main" java.util.ConcurrentModificationException
at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
at MainKt.main(Main.kt:4)
at MainKt.main(Main.kt)
Process finished with exit code 1
9: Git 6: TODO 4: Run Terminal 0: Messages Event Log
Build completed successfully with 1 warning in 22 s 935 ms (moments ago)
7:2 LF UTF-8 4 spaces main 286 of 733M


exp1 [~/repo/FuncAndLogProg/lections/lection9/exp1] – Main.kt
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MainKt
Git:
exp1 src Main.kt
Project
exp1 ~/repo/FuncAndLogProg/
.idea
lib
out
src
exp1.iml 31.03.2021, 17:51, 498 B
External Libraries
Scratches and Consoles
Main.kt
fun main() {
val shopping :List<String> = listOf("Tea", "Eggs", "Milk")
val numbList :MutableList<Int> = mutableListOf<Int>(2,3,4,5,2,3,3,2,2,3)
/*for (numb in numbList)
if (numb == 2) numbList.remove(2)
print(numbList)*/
val iter : MutableIterator<Int> = numbList.iterator()
while (iter.hasNext()) {
val numb :Int = iter.next()
if (numb == 2) iter.remove()
}
print(numbList)
}
main()
Run: MainKt
/home/ars/.jdks/openjdk-14.0.2/bin/java -javaagent:/home/ars/PO/idea-IE-201.7846.105/lib/idea_rt.jar=40289
[3, 4, 5, 3, 3, 3]
Process finished with exit code 0
9: Git
6: TODO
4: Run
Terminal
Event Log
Cannot Run Git: File not found: git // Configure... (27 minutes ago)
12:20 LF UTF-8 4 spaces main
329 of 733M

Если вам нужна коллекция, которая не допускает дублирования, используйте **`Set`**: неупорядоченную коллекцию без повторяющихся значений.

Для создания объекта `Set` вызывается функция с именем **`setOf`**, которой передаются значения для инициализации элементов множества. Например, следующий фрагмент создает множество `Set`, инициализирует его тремя строками и присваивает новой переменной с именем `friendSet`:

```
val friendSet = setOf("Jim", "Sue", "Nick")
```



В `Set` не может быть дубликатов, поэтому при попытке создать множество следующей командой:

```
val friendSet = setOf("Jim",
                      "Sue",
                      "Sue",
                      "Nick",
                      "Nick")
```

`Set` игнорирует дубликаты «Sue» и «Nick». Код создает множество `Set`, которое содержит три разных строковых значения, как и в предыдущем случае.

Компилятор определяет тип элементов множества `Set` по значениям, передаваемым при создании. Например, следующий код инициализирует `Set` строковыми значениями, так что компилятор создает множество `Set` с типом `Set<String>`.

Значения `Set` не упорядочены, поэтому в отличие от `List`, у них нет функции `get` для получения значения с заданным индексом. Впрочем, функция `contains` позволяет проверить, содержит ли множество `Set` конкретное значение:

```
val isFredGoing = friendSet.contains("Fred")
```

Возвращает true, если friendSet содержит значение «Fred», и false в противном случае.

Перебор элементов множества `Set` в цикле выполняется так:

```
for (item in friendSet) println(item)
```

Множество `Set` неизменяемо — в него нельзя добавлять новые или удалять существующие значения. Для выполнения таких операций используется класс `MutableSet`. Но прежде чем мы покажем, как создавать и использовать такой класс, следует ответить на один важный вопрос: **как `Set` определяет, является ли значение дубликатом?**

В отличие от List, множество Set не упорядочено и не может содержать повторяющихся значений.

Как Set проверяет наличие дубликатов

Чтобы ответить на этот вопрос, воспроизведем последовательность действий, которые выполняет `Set` для решения вопроса о том, является ли значение дубликатом.

1. **Set получает хеш-код объекта и сравнивает его с хеш-кодами объектов, уже находящихся в множестве Set.**
   `Set` использует хеш-коды для сохранения элементов способом, заметно ускоряющим обращение к ним. Хеш-код — это своего рода этикетка на «корзине», в которой хранятся элементы, так что все объекты с хеш-кодом 742 (например) хранятся в корзине с меткой 742.

   Если совпадающих хеш-кодов не находится, `Set` считает, что это не дубликат, и добавляет новое значение. Но если совпадающие хеш-коды будут обнаружены, класс `Set` должен выполнить дополнительные проверки — происходит переход к шагу 2.

2. **Set использует оператор === для сравнения нового значения с любыми содержащимися в нем объектами с тем же хеш-кодом.**
   Как было показано в главе 7, оператор `===` используется для проверки того, указывают ли две ссылки на один объект. Таким образом, если оператор `===` возвращает `true` для любого объекта с тем же хеш-кодом, `Set` знает, что новое значение является дубликатом, и отклоняет его. Но если оператор `===` возвращает `false`, то `Set` переходит к шагу 3.

3. **Set использует оператор == для сравнения нового значения со всеми объектами, содержащимися в коллекции, с совпадающими хеш-кодами.**
   Оператор `==` вызывает функцию `equals` значения. Если функция возвращает `true`, то `Set` рассматривает новое значение как дубликат и отвергает его. Если же оператор `==` возвращает `false`, то `Set` считает, что новое значение не является дубликатом, и добавляет его.

Итак, есть две ситуации, в которых `Set` рассматривает новое значение как дубликат: если это *тот же* объект или оно *равно* значению, уже содержащемуся в коллекции. Рассмотрим происходящее более подробно.

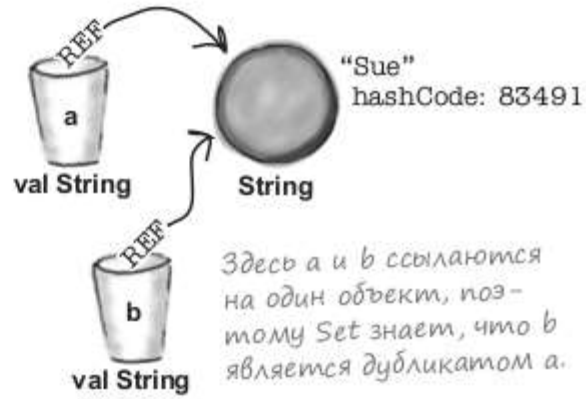
Посмотрим, как это относится к операторам === и ==.

Проверка оператором ===

Если у вас имеются две ссылки, указывающие на один объект, вы получите одинаковые результаты при вызове функции `hashCode` для каждой ссылки. Если функция `hashCode` не переопределена, то в поведении по умолчанию (унаследованном от суперкласса `Any`) каждый класс получит уникальный хеш-код.

При выполнении следующего кода `Set` замечает, что `a` и `b` имеют одинаковые хеш-коды и ссылаются на один объект, поэтому в `Set` добавляется только одно значение:

```
val a = "Sue"
val b = a
val set = setOf(a, b)
```



Проверка оператором ==

Если вы хотите, чтобы класс `Set` рассматривал два разных объекта `Recipe` как равные (или эквивалентные), есть два варианта: сделать `Recipe` классом данных или переопределить функции `hashCode` и `equals`, унаследованные от `Any`. Преобразование `Recipe` в класс данных — самый простой вариант, так как в этом случае обе функции переопределяются автоматически.

Как упоминалось выше, в поведении по умолчанию (из `Any`) каждому объекту предоставляется уникальное значение хеш-кода. Таким образом, вы *должны* переопределить `hashCode`, чтобы быть уверенными в том, что два эквивалентных объекта возвращают один хеш-код. Но вы также должны переопределить `equals`, чтобы оператор == возвращал `true` при сравнении объектов с совпадающими значениями свойств.

В следующем примере в `Set` будет добавлено одно значение, если `Recipe` является классом данных:

```
val a = Recipe("Thai Curry")
val b = Recipe("Thai Curry")
val set = setOf(a, b)
```

Правила переопределения hashCode и equals

Если вы решите вручную переопределить функции `hashCode` и `equals` в своем классе (вместо того, чтобы использовать класс данных), вам придется соблюдать ряд правил. Если эти правила будут нарушены, в мире Kotlin произойдет катастрофа — множества `Set` будут работать некорректно. Непременно соблюдайте эти правила!

Перечислим эти правила:

- Если два объекта равны, они должны иметь одинаковые хеш-коды.
- Если два объекта равны, вызов `equals` для любого из объектов должен возвращать `true`. Другими словами, если `(a.equals(b))`, то `(b.equals(a))`.
- Если два объекта имеют одинаковые хеш-коды, это не значит, что они равны. Но если они равны, то они должны иметь одинаковые хеш-коды.
- При переопределении `equals` вы должны переопределить `hashCode`.
- По умолчанию функция `hashCode` генерирует уникальное целое число для каждого объекта. Таким образом, если вы не переопределите `hashCode` в классе, не являющемся классом данных, два объекта этого типа ни при каких условиях не будут считаться равными.
- По умолчанию функция `equals` должна выполнять сравнение `===`, то есть проверять, что две ссылки относятся к одному объекту. Следовательно, если вы не переопределяете `equals` в классе, не являющемся классом данных, два объекта не будут считаться равными, потому что по ссылкам на два разных объекта всегда будут содержаться разные наборы битов.

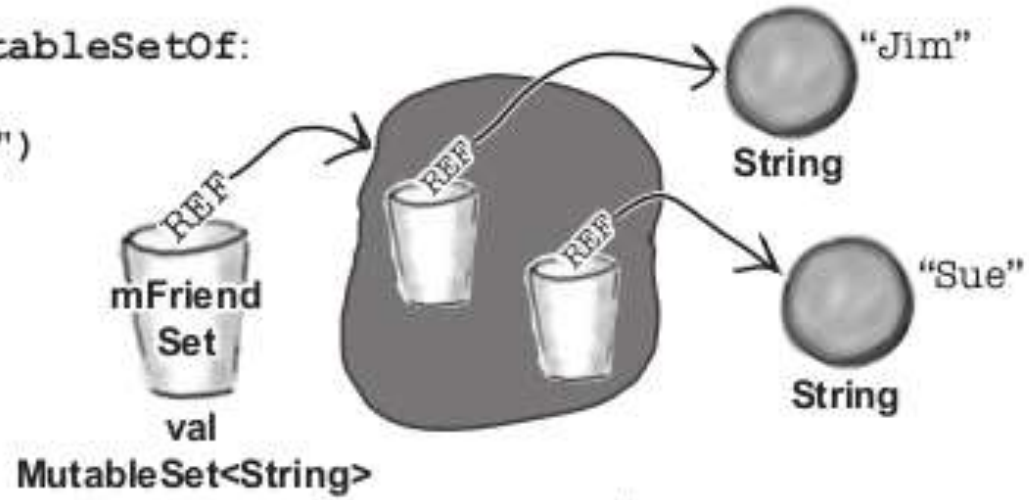
a.equals(b) также должно означать, что a.hashCode() == b.hashCode()

Но a.hashCode() == b.hashCode() не означает, что a.equals(b)

Теперь, когда вы знаете о `Set`, перейдем к **`MutableSet`**. `MutableSet` является подклассом `Set`, но содержит дополнительные функции для добавления и удаления значений.

Объект `MutableSet` создается вызовом функции **`mutableSetOf`**:

```
val mFriendSet = mutableSetOf("Jim", "Sue")
```



Если передать функции mutableSetOf() строковые значения, компилятор определяет, что вам нужен объект типа MutableSet<String> (MutableSet для хранения String).

Здесь `MutableSet` инициализируется двумя строками, поэтому компилятор заключает, что вам нужен объект `MutableSet` с типом `MutableSet<String>`.

Новые значения добавляются в `MutableSet` функцией **`add`**. Например, следующий код добавляет значение "Nick" в `mFriendSet`:

```
mFriendSet.add("Nick")
```

Функция `add` проверяет, встречается ли переданный объект в `MutableSet`. Если дубликат будет найден, возвращается `false`. Но если значение не является дубликатом, оно добавляется в `MutableSet` (с увеличением размера на 1), а функция возвращает `true` — признак успешного выполнения операции.

Для удаления значений из `MutableSet` используется функция `remove`. Например, следующий код удаляет строку «Nick» из `mFriendSet`:

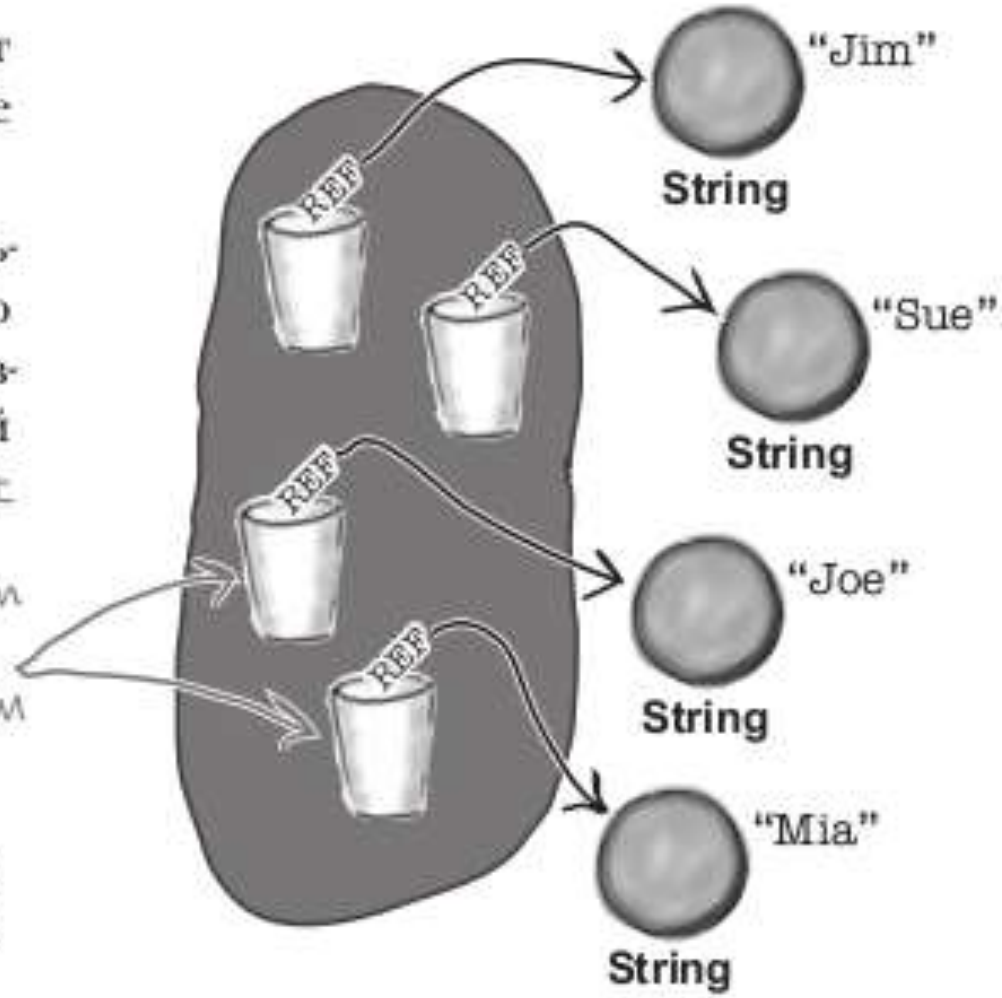
```
mFriendSet.remove("Nick")
```

Если строка «Nick» существует в `MutableSet`, функция удаляет ее и возвращает `true`. Но если подходящий объект найти не удается, функция просто возвращает `false`.

Функции **`addAll`**, **`removeAll`** и **`retainAll`** также могут использоваться для внесения массовых изменений в `MutableSet` (по аналогии с `MutableList`). Например, функция `addAll` добавляет в `MutableSet` все элементы, присутствующие в другой коллекции, так что для добавления «Joe» и «Mia» в `mFriendSet` можно использовать следующий код:

```
val toAdd = setOf("Joe", "Mia")
mFriendSet.addAll(toAdd)
```

addAll() добавляет значения, содержащиеся в другом объекте Set.



Как и в случае с `MutableList`, также можно воспользоваться функцией **`clear`** для удаления всех элементов из `MutableSet`:

```
mFriendSet.clear()
```

Если вы хотите сделать снимок содержимого MutableSet, вы можете сделать это по аналогии с MutableList. Например, при помощи функции **toSet** можно создать неизменяемую копию mFriendSet и присвоить копию новой переменной friendSetCopy:

```
val friendSetCopy = mFriendSet.toSet()
```

Также можно скопировать Set или MutableSet в новый объект List функцией **toList**:

```
val friendList = mFriendSet.toList()
```

А если у вас имеется объект MutableList или List, его можно скопировать в Set функцией **toSet**:

```
val shoppingSet = mShopping.toSet()
```

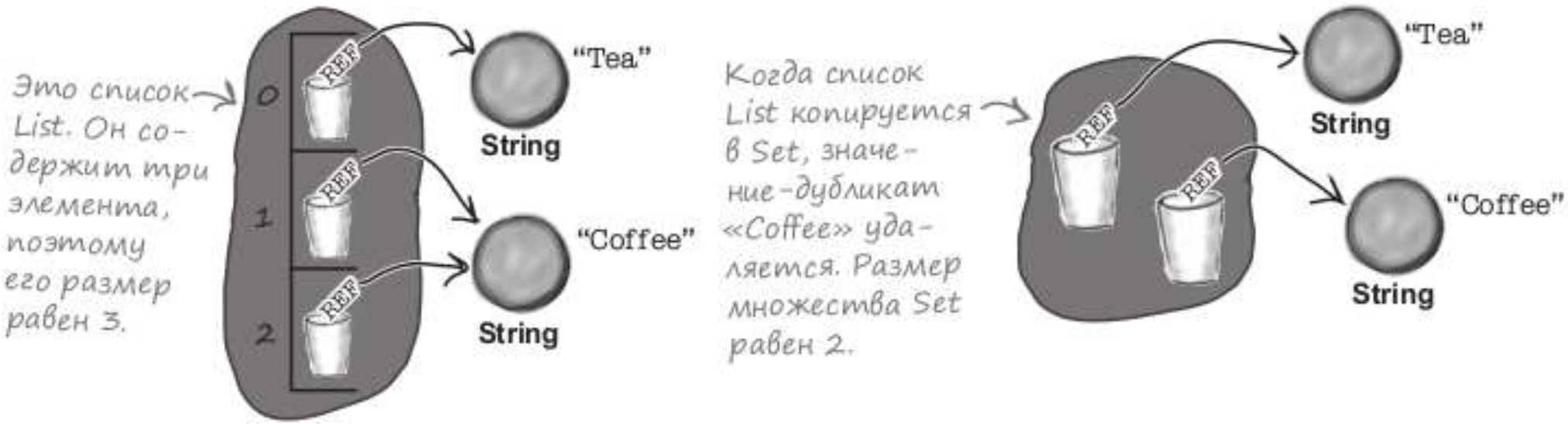
Объект MutableSet также содержит функции toMutableSet() (для копирования в новый объект MutableSet) и toMutableList() (для копирования в новый объект MutableList).

Копирование коллекции в другой тип может быть особенно полезно, если вы хотите выполнить другую операцию, которая без этого была бы неэффективной. Например, чтобы проверить, содержит ли список дубликаты, можно скопировать `List` в `Set` и проверить размер каждой коллекции. В следующем коде этим способом мы проверяем, содержит ли дубликаты список `mShopping` (`MutableList`):

Создает версию mShopping в формате Set и возвращает ее размер.

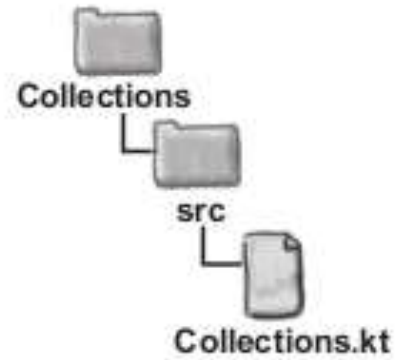
```
if (mShopping.size > mShopping.toSet().size) {
    //mShopping содержит дубликаты
}
```

Если список `mShopping` содержит дубликаты, его размер будет больше, чем после копирования в `Set`, потому что при преобразовании `MutableList` в `Set` дубликаты будут удалены.



Переменная mShoppingList должна объявляться с ключевым словом var, чтобы позднее ее можно было обновить другим значением MutableList<String>.

```
fun main(args: Array<String>) {
    ~~val~~ var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")
```



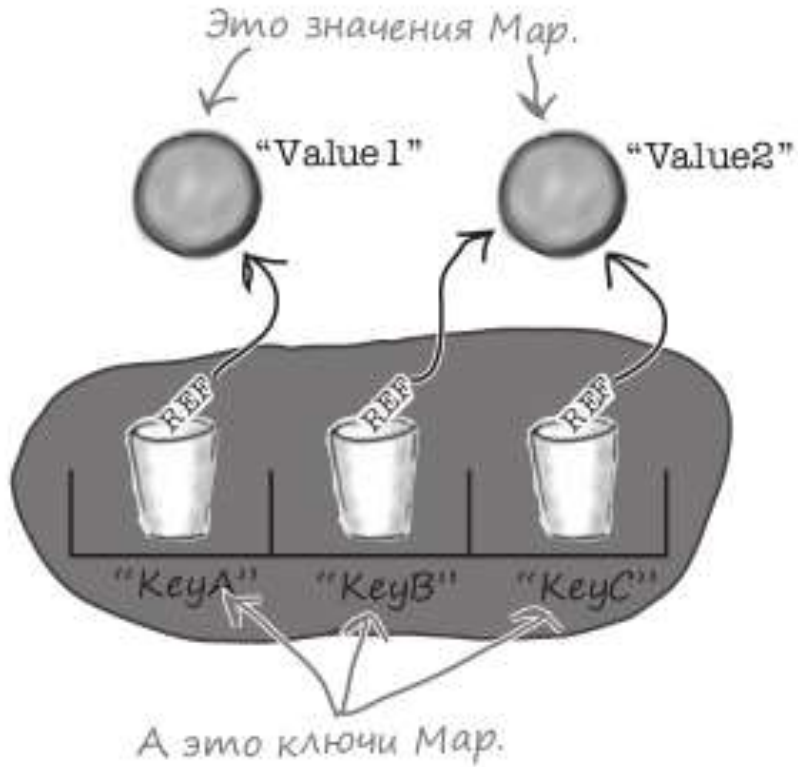
Добавьте этот код.

```
    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")
}
```

Ассоциативные массивы Map

`List` и `Set` прекрасно работают, но есть еще один тип коллекций, с которым мы хотим вас познакомить: **`Map`**. Коллекция `Map` работает как список свойств. Вы передаете ей ключ, а `Map` возвращает значение, связанное с этим ключом. Хотя ключи обычно имеют тип `String`, они могут быть объектами любого типа.

Каждый элемент `Map` состоит из двух объектов — ***ключа*** и ***значения***. С каждым ключом связывается одно значение. В коллекции могут присутствовать повторяющиеся *значения*, но не повторяющиеся *ключи*.



Как создать Map

Чтобы создать `Map`, вызовите функцию с именем **`mapOf`** и передайте ей пары «ключ-значение» для инициализации `Map`. Например, следующий код создает `Map` с тремя элементами. Ключами являются строки («Recipe1», «Recipe2» и «Recipe3»), а значениями — объекты `Recipe`:

```
val r1 = Recipe("Chicken Soup")
val r2 = Recipe("Quinoa Salad")
val r3 = Recipe("Thai Curry")

val recipeMap = mapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
```

Каждый элемент определяется в форме «ключ-значение». Ключи обычно являются строками, как в данном примере.

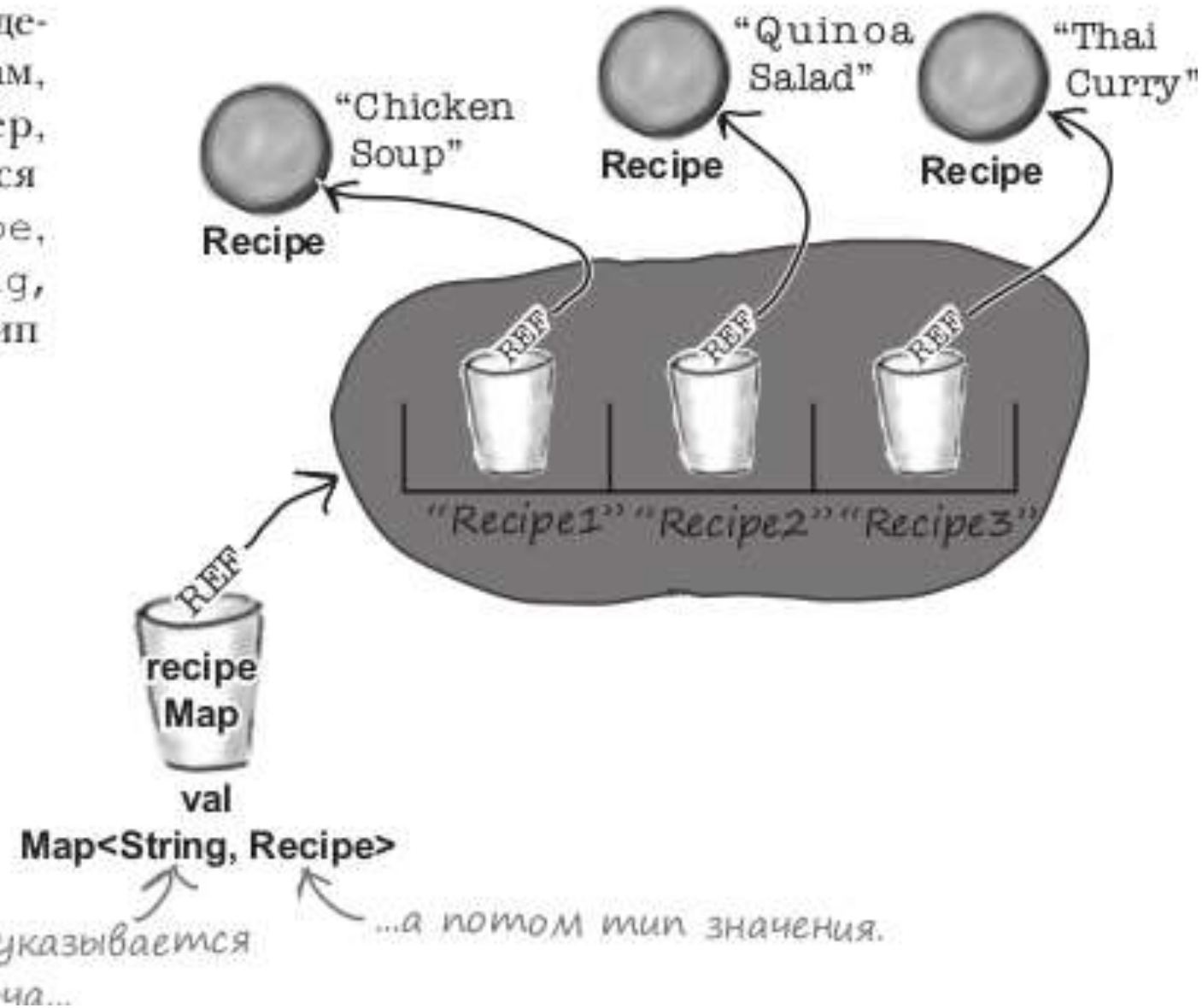
Как нетрудно догадаться, компилятор определяет тип пар «ключ-значение» по элементам, которыми он инициализируется. Например, приведенный выше `Map` инициализируется строковыми ключами и значениями `Recipe`, поэтому будет создан `Map` с типом `Map<String, Recipe>`. Также можно явно определить тип `Map` кодом следующего вида:

```
val recipeMap: Map<String, Recipe>
```

Обычно тип `Map` задается в форме:

```
Map<тип_ключа, тип_значения>
```

Итак, вы научились создавать ассоциативные массивы `Map`. Теперь посмотрим, как их использовать.



С Map чаще всего выполняются три операции: проверка наличия конкретного ключа или значения, выборка значения для заданного ключа и перебор всех элементов Map.

Для проверки наличия конкретного ключа или значения в Map используются его функции **containsKey** и **containsValue**. Например, следующий код проверяет, содержит ли Map с именем recipeMap ключ «Recipe1»:

```
recipeMap.containsKey("Recipe1")
```

Вы можете проверить, содержит ли recipeMap объект Recipe для «Chicken Soup» при помощи функции containsValue:

```
val recipeToCheck = Recipe("Chicken Soup")
if (recipeMap.containsValue(recipeToCheck)) {
    //Код, выполняемый при наличии значения в Map
}
```

Предполагается, что Recipe является классом данных, так что Map может определить, когда два объекта Recipe равны.

Для получения значения, связанного с конкретным ключом, используются функции **get** и **getValue**. Если заданный ключ не существует, `get` возвращает `null`, а `getValue` вызывает исключение. В следующем примере функция `getValue` получает объект `Recipe`, связанный с ключом «Recipe1»:

```
if (recipeMap.containsKey("Recipe1")) {
    val recipe = recipeMap.getValue("Recipe1")
    //Код использования объекта Recipe
}
```

Если в recipeMap нет ключа «Recipe1», эта строка вызовет исключение.

Также вы можете перебрать все элементы `Map`. Например, вот как цикл `for` используется для вывода всех пар «ключ-значение» в `recipeMap`:

```
for ((key, value) in recipeMap) {
    println("Key is $key, value is $value")
}
```

Объект `Map` неизменяем, поэтому вы не сможете добавлять или удалять пары «ключ-значение» или обновлять значение, хранящееся для заданного ключа. Для выполнения такой операции следует использовать класс `MutableMap`. Посмотрим, как он работает.

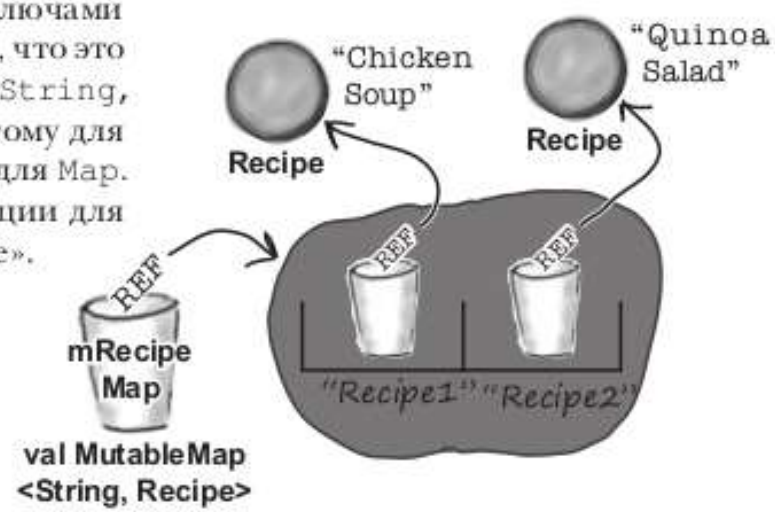
Создание MutableMap

Объекты **MutableMap** определяются практически так же, как Map, если не считать того, что вместо функции mapOf используется функция **mutableMapOf**. Например, следующий код создает массив MutableMap с тремя элементами, как и в предыдущем примере:

```
val r1 = Recipe("Chicken Soup")
val r2 = Recipe("Quinoa Salad")

val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2)
```

Объект MutableMap инициализируется строковыми ключами и значениями Recipe, поэтому компилятор делает вывод, что это должна быть коллекция MutableMap типа MutableMap<String, Recipe>. MutableMap является подклассом Map, поэтому для MutableMap могут вызываться те же функции, что и для Map. Однако MutableMap содержит дополнительные функции для добавления, удаления и обновления пар «ключ-значение».



Если функции mutableMapOf() передаются строковые ключи и значения Recipe, компилятор определяет, что нужно создать объект типа MutableMap<String, Recipe>.

Включение элементов в MutableMap

Для включения элементов в MutableMap используется функция **put**. Например, следующий код добавляет ключ «Recipe3» в mRecipeMap и связывает его с объектом Recipe для «Thai Curry»:

```
val r3 = Recipe("Thai Curry")
mRecipeMap.put("Recipe3", r3)
```

Сначала задается ключ, а потом значение.

Если MutableMap уже содержит заданный ключ, функция put заменяет значение для этого ключа и возвращает исходное значение.

В MutableMap можно добавить сразу несколько пар «ключ-значение» при помощи функции **putAll**. Функция получает один аргумент — Map с добавляемыми элементами. Например, следующий код добавляет объекты «Jambalaya» и «Sausage Rolls» в Map с именем recipesToAdd, после чего добавляет эти элементы в mRecipeMap:

```
val r4 = Recipe("Jambalaya")
val r5 = Recipe("Sausage Rolls")
val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5)
mRecipeMap.putAll(recipesToAdd)
```

Теперь посмотрим, как происходит удаление значений.

Для удаления элементов из `MutableMap` используется функция **`remove`**. Функция перегружена, чтобы ее можно было вызывать двумя разными способами.

В первом способе функции `remove` передается ключ удаляемого элемента. Например, следующий код удаляет из `mRecipeMap` элемент с ключом «Recipe2»:

```
mRecipeMap.remove("Recipe2")
```
← Удаление элемента с ключом «Recipe2».

Во втором варианте функции `remove` передается ключ и значение. Функция удаляет запись только в том случае, если будет найдено совпадение для ключа и для значения. Таким образом, следующий код удаляет элемент для «Recipe2» только тогда, когда он связан с объектом `Recipe` «Quinoa Salad»:

```
val recipeToRemove = Recipe("Quinoa Salad")
mRecipeMap.remove("Recipe2", recipeToRemove)
```
← Удаление элемента с ключом «Recipe2», но только в том случае, если его значением является объект Recipe «Quinoa Salad».

Какой бы способ вы ни выбрали, при удалении элемента из коллекции `MutableMap` размер последней уменьшается.

Наконец, вы можете воспользоваться функцией **`clear`** для удаления всех элементов из `MutableMap` по аналогии с тем, как это делается с `MutableList` и `MutableSet`:

```
mRecipeMap.clear()
```

Итак, вы научились обновлять `MutableMap`. Теперь давайте посмотрим, как их копировать.

Как и в случае с другими типами коллекций, вы можете создать снимок MutableMap. Например, при помощи функции **toMap** можно создать копию mRecipeMap, доступную только для чтения, и присвоить ее новой переменной:

```
val recipeMapCopy = mRecipeMap.toMap()
```

Map или MutableMap можно скопировать в новый объект List, содержащий все пары «ключ-значение», при помощи функции **toList**:

← MutableMap также содержит функции toMutableMap() и toMutableList().

```
val RecipeList = mRecipeMap.toList()
```

И вы также можете получить прямой доступ к парам «ключ-значение», обратившись к свойству **entries** объекта Map. Свойство entries возвращает Set при использовании с Map или MutableSet при использовании с MutableMap. Например, следующий код возвращает объект MutableSet с парами «ключ-значение» из mRecipeMap:

```
val recipeEntries = mRecipeMap.entries
```

Другие полезные свойства — **keys** (возвращает множество `Set` или `MutableSet` с ключами `Map`) и **values** (возвращает обобщенную коллекцию значений `Map`). Например, при помощи этих свойств можно проверить, встречаются ли в `Map` повторяющиеся значения:

Обратите внимание: свойства entries, keys и values фактически хранятся в Map (или MutableMap), это не копии. А если вы работаете с MutableMap, эти свойства могут обновляться.

```
if (mRecipeMap.size > mRecipeMap.values.toSet().size) {
    println("mRecipeMap contains duplicates values")
}
```

Это объясняется тем, что вызов

```
mRecipeMap.values.toSet()
```

копирует значения `Map` в `Set` с удалением всех дубликатов.

Теперь, когда узнали, как работать с массивами `Map` и `MutableMap`, немного доработаем наш проект Collections.

```
data class Recipe(var name: String)
```
Добавьте класс данных Recipe.

```
fun main(args: Array<String>) {
    var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")

    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")
```

Добавьте этот фрагмент.

```
    val r1 = Recipe("Chicken Soup")
    val r2 = Recipe("Quinoa Salad")
    val r3 = Recipe("Thai Curry")
    val r4 = Recipe("Jambalaya")
    val r5 = Recipe("Sausage Rolls")
    val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
    println("mRecipeMap original: $mRecipeMap")
    val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5)
    mRecipeMap.putAll(recipesToAdd)
    println("mRecipeMap updated: $mRecipeMap")
    if (mRecipeMap.containsKey("Recipe1")) {
        println("Recipe1 is: ${mRecipeMap.getValue("Recipe1")}")
    }
}
```

```
data class Recipe(var name: String)
```
Добавьте класс данных Recipe.

```
fun main(args: Array<String>) {
    var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")

    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")
```

Добавьте этот фрагмент.

```
    val r1 = Recipe("Chicken Soup")
    val r2 = Recipe("Quinoa Salad")
    val r3 = Recipe("Thai Curry")
    val r4 = Recipe("Jambalaya")
    val r5 = Recipe("Sausage Rolls")
    val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
    println("mRecipeMap original: $mRecipeMap")
    val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5)
    mRecipeMap.putAll(recipesToAdd)
    println("mRecipeMap updated: $mRecipeMap")
    if (mRecipeMap.containsKey("Recipe1")) {
        println("Recipe1 is: ${mRecipeMap.getValue("Recipe1")}")
    }
}
```