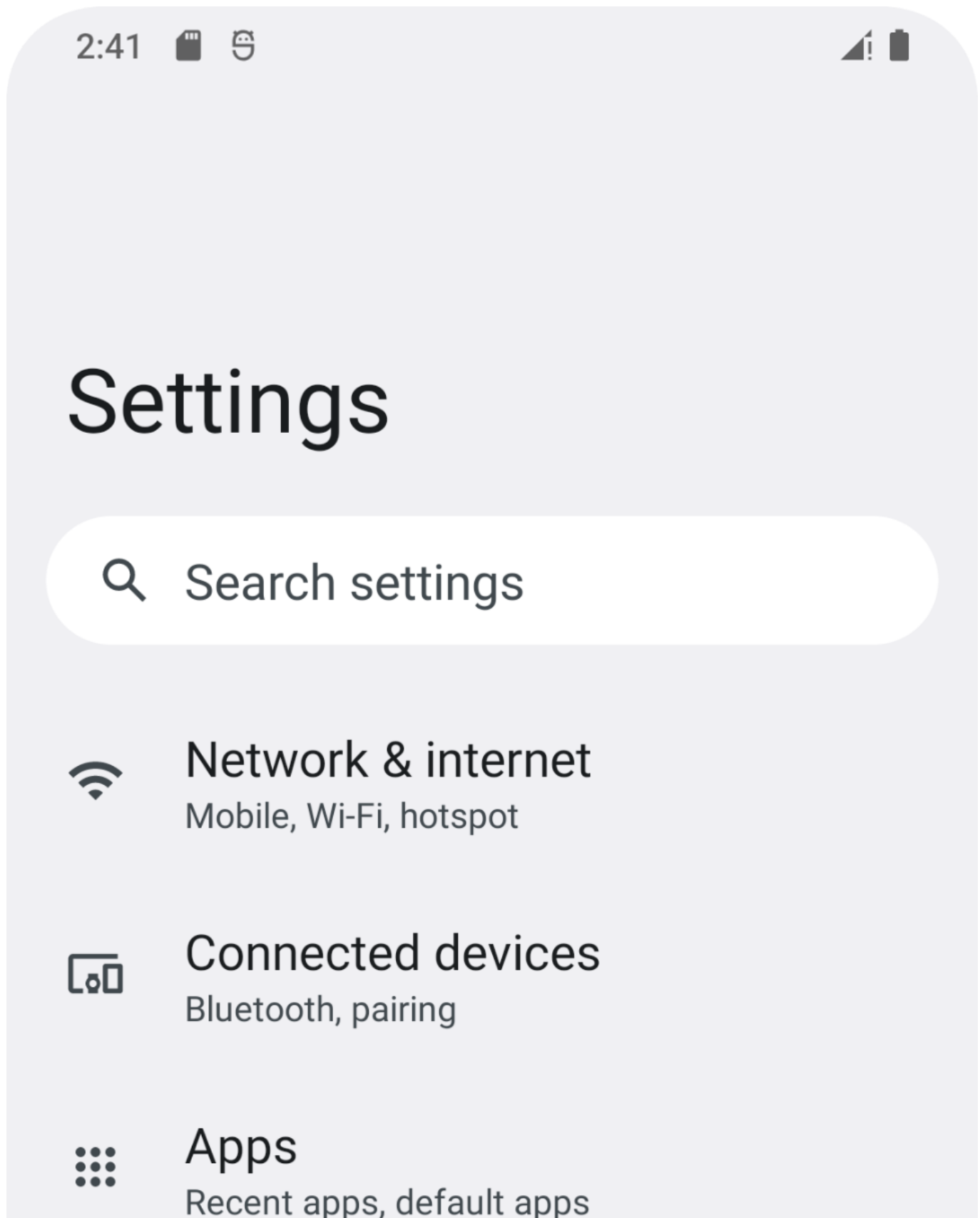
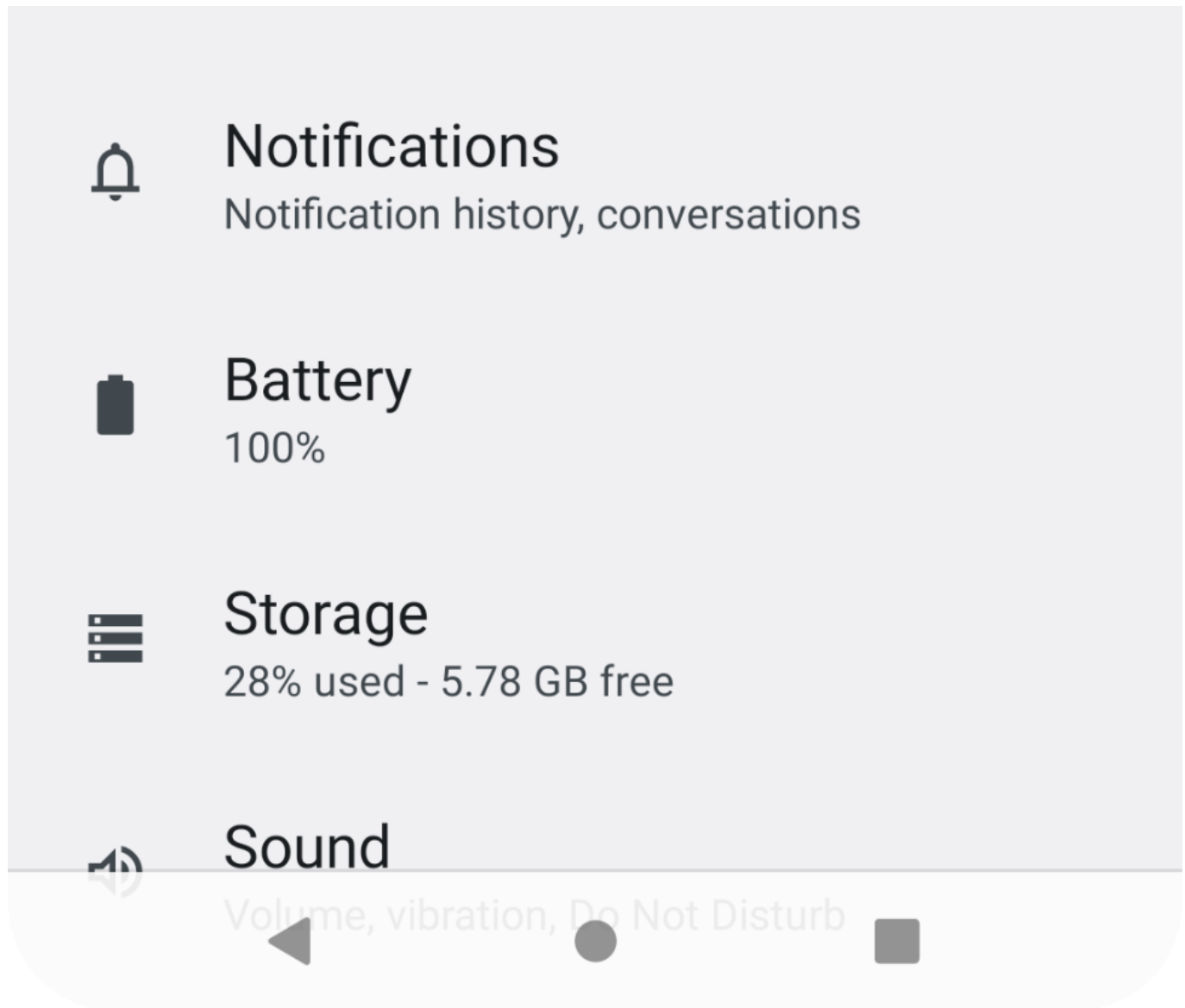


Использование коллекций в Kotlin

Введение

Во многих приложениях вы наверняка видели данные, отображаемые в виде списка: контакты, настройки, результаты поиска и т. д.





Однако в коде, который вы написали до сих пор, вы в основном работали с данными, состоящими из одного значения, например, числа или куска текста, выводимого на экран. Чтобы создавать приложения с произвольными объемами данных, вам нужно научиться использовать **коллекции**.

Типы коллекций (иногда называемые структурами данных) позволяют хранить несколько значений, обычно одного типа, в упорядоченном виде.

Коллекция может представлять собой упорядоченный список, группу уникальных значений или отображение значений одного типа данных на значения другого. Умение эффективно использовать коллекции позволяет реализовать такие распространенные функции приложений для Android, как прокручивающиеся списки, а также решать различные реальные задачи программирования, связанные с произвольными объемами данных.

В этом уроке мы рассмотрим, как работать с несколькими значениями в коде, и познакомимся с различными структурами данных, включая массивы, списки, наборы и карты.

Необходимые условия

- Знакомство с объектно-ориентированным программированием на Kotlin, включая классы, интерфейсы и дженерики.

Что вы узнаете

- Как создавать и изменять массивы.
- Как использовать List и MutableList.
- Как использовать Set и MutableSet.
- Как использовать Map и MutableMap.

Что вам понадобится

- IntelliJ IDEA

Массивы в Kotlin

Что такое массив? Массив - это простейший способ группировки произвольного количества значений в ваших программах.

Подобно тому, как группа солнечных батарей называется солнечным массивом, или как изучение Kotlin открывает множество возможностей для вашей карьеры программиста, массив представляет собой более чем одно значение. Точнее, массив - это последовательность значений, которые имеют один и тот же тип данных.

Single value



8

Array



8 6 7 5 3 0 9

Массив содержит множество значений, называемых элементами

Элементы в массиве упорядочены, и доступ к ним осуществляется с помощью индекса.

Что такое индекс? Индекс - это целое число, соответствующее элементу в массиве. Индекс определяет расстояние элемента от начального элемента массива. Это называется нулевой индексацией. Первый элемент массива находится под индексом 0, второй - под индексом 1, потому что он находится на одном месте от первого элемента, и так далее.

Array



В памяти устройства элементы массива хранятся рядом друг с другом. Хотя детали, лежащие в основе, выходят за рамки данного урока, это имеет два важных последствия:

- Доступ к элементу массива по его индексу - это быстро. Вы можете обратиться к любому произвольному элементу массива по его индексу и ожидать, что это займет примерно столько же времени, сколько и обращение к любому другому произвольному элементу. Именно поэтому говорят, что массивы имеют произвольный доступ.
- Массив имеет фиксированный размер. Это означает, что вы не можете добавлять в массив элементы, превышающие этот размер. Попытка получить доступ к элементу с индексом 100 в массиве из 100 элементов приведет к исключению, потому что наибольший индекс равен 99 (помните, что первый индекс равен 0, а не 1). Однако вы можете изменять значения по индексам в массиве.

Примечание: В этом учебнике под памятью понимается краткосрочная память с произвольным доступом (RAM) устройства, а не долгосрочное постоянное хранилище. Она называется «Случайный доступ», потому что позволяет быстро получить доступ к любому произвольному месту в памяти.

Чтобы объявить массив в коде, используется функция **arrayOf()**.

```
val variable name = arrayOf<data type> ( element1 , element2 , ... )
```

Функция **arrayOf()** принимает в качестве параметров элементы массива и возвращает массив того типа, который соответствует переданным параметрам. Это может немного отличаться от других функций, которые вы видели, потому что `arrayOf()` имеет разное количество параметров. Если передать функции `arrayOf()` два аргумента, то результирующий массив будет содержать два элемента, проиндексированных 0 и 1. Если передать три аргумента, то результирующий массив будет содержать 3 элемента, проиндексированных от 0 до 2.

Давайте посмотрим на массивы в действии с помощью небольшого исследования Солнечной системы!

- Перейдите в IntelliJ Idea
- В `main()` создайте переменную `rockPlanets`. Вызовите `arrayOf()`, передав тип `String`, а также четыре строки - по одной для каждой из планет Солнечной системы.

```
val rockPlanets = arrayOf<String>("Mercury", "Venus", "Earth", "Mars")
```

Поскольку в Kotlin используется вывод типов, при вызове `arrayOf()` имя типа можно опустить. Ниже переменной `rockPlanets` добавьте еще одну переменную `gasPlanets`, не указывая тип в угловых скобках.

```
val gasPlanets = arrayOf("Jupiter", "Saturn", "Uranus", "Neptune")
```

С массивами можно делать разные вещи. Например, как и в случае с числовыми типами `Int` или `Double`, вы можете сложить два массива. Создайте новую переменную `solarSystem` и установите ее равной результату `rockPlanets` и `gasPlanets`, используя оператор `plus (+)`. В результате получится новый массив, содержащий все элементы массива `rockPlanets` и элементы массива `gasPlanets`.

```
val solarSystem = rockPlanets + gasPlanets
```

Запустите свою программу, чтобы убедиться, что она работает. Вы не должны увидеть никакого вывода.

Доступ к элементу массива

Вы можете получить доступ к элементу массива по его индексу.



Это называется синтаксисом субскриптов. Он состоит из трех частей:

Имя массива. Открывающая ([]) и закрывающая (]) квадратные скобки. Индекс элемента массива в квадратных скобках.

Давайте обратимся к элементам массива `solarSystem` по их индексам.

- В `main()` обратитесь к каждому элементу массива `solarSystem` и выведите его на печать. Обратите внимание, что первый индекс равен 0, а последний - 7.

```
println(solarSystem[0])
println(solarSystem[1])
println(solarSystem[2])
println(solarSystem[3])
println(solarSystem[4])
println(solarSystem[5])
println(solarSystem[6])
println(solarSystem[7])
```

- Запустите свою программу. Элементы расположены в том же порядке, в котором вы их перечислили при вызове `arrayOf()`.

```
Mercury  
Venus  
Earth  
Mars  
Jupiter  
Saturn  
Uranus  
Neptune
```

Вы также можете задать значение элемента массива по его индексу.

array name

[

index

]

=

value

Доступ к индексу осуществляется так же, как и раньше: имя массива, затем открывающая и закрывающая квадратные скобки, содержащие индекс. Затем следует оператор присваивания (=) и новое значение.

Давайте попрактикуемся в изменении значений в массиве `solarSystem`.

- Давайте дадим Марсу новое имя для его будущих поселенцев. Обратимся к элементу с индексом 3 и установим его равным «Маленькая Земля».

```
solarSystem[3] = "Little Earth"
```

- Выведите элемент с индексом 3.

```
println(solarSystem[3])
```

- Запустите свою программу. Четвертый элемент массива (с индексом 3) обновлен.

```
...  
Little Earth
```

Допустим, ученые обнаружили, что за Нептуном есть девятая планета, называемая Плутоном. Ранее мы упоминали, что нельзя изменить размер массива. Что произойдет, если вы попытаетесь это сделать? Давайте попробуем добавить Плутон в массив `SolarSystem`. Добавьте Плутон по индексу 8, так как это 9-й элемент в массиве.

```
solarSystem[8] = "Pluto"
```

- Запустите свой код. Он выбрасывает исключение `ArrayIndexOutOfBoundsException`. Поскольку массив уже содержит 8 элементов, как и ожидалось, вы не можете просто добавить девятый элемент.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 8 out of bounds for length 8
```

- Удалите добавление Плутона в массив.

Код для удаления

```
solarSystem[8] = "Pluto"
```

Если вы хотите сделать массив больше, чем он есть, вам нужно создать новый массив. Определите новую переменную `newSolarSystem`, как показано на рисунке. В этом массиве может храниться не восемь, а девять элементов.

```
val newSolarSystem = arrayOf("Mercury", "Venus", "Earth", "Mars", "Jupiter",  
"Saturn", "Uranus", "Neptune", "Pluto")
```

- Теперь попробуйте напечатать элемент с индексом 8.

```
println(newSolarSystem[8])
```

- Запустите свой код и убедитесь, что он работает без каких-либо исключений.

```
...  
Pluto
```

Отличная работа! С вашими знаниями о массивах вы можете делать с коллекциями практически все.

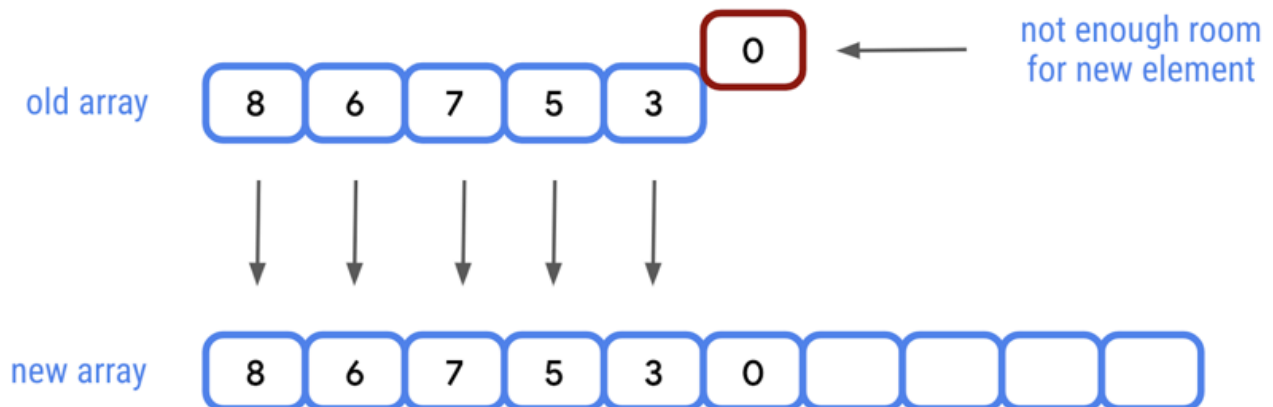
Подождите, не так быстро! Хотя массивы являются одним из фундаментальных аспектов программирования, использование массива для задач, требующих добавления и удаления элементов, уникальности коллекции или сопоставления объектов с другими объектами, не совсем просто и понятно, и код вашего приложения быстро превратится в беспорядок.

Именно поэтому в большинстве языков программирования, включая Kotlin, реализованы специальные типы коллекций для обработки ситуаций, которые часто возникают в реальных приложениях. В следующих разделах вы узнаете о трех распространенных коллекциях: `List`, `Set` и `Map`. Вы также

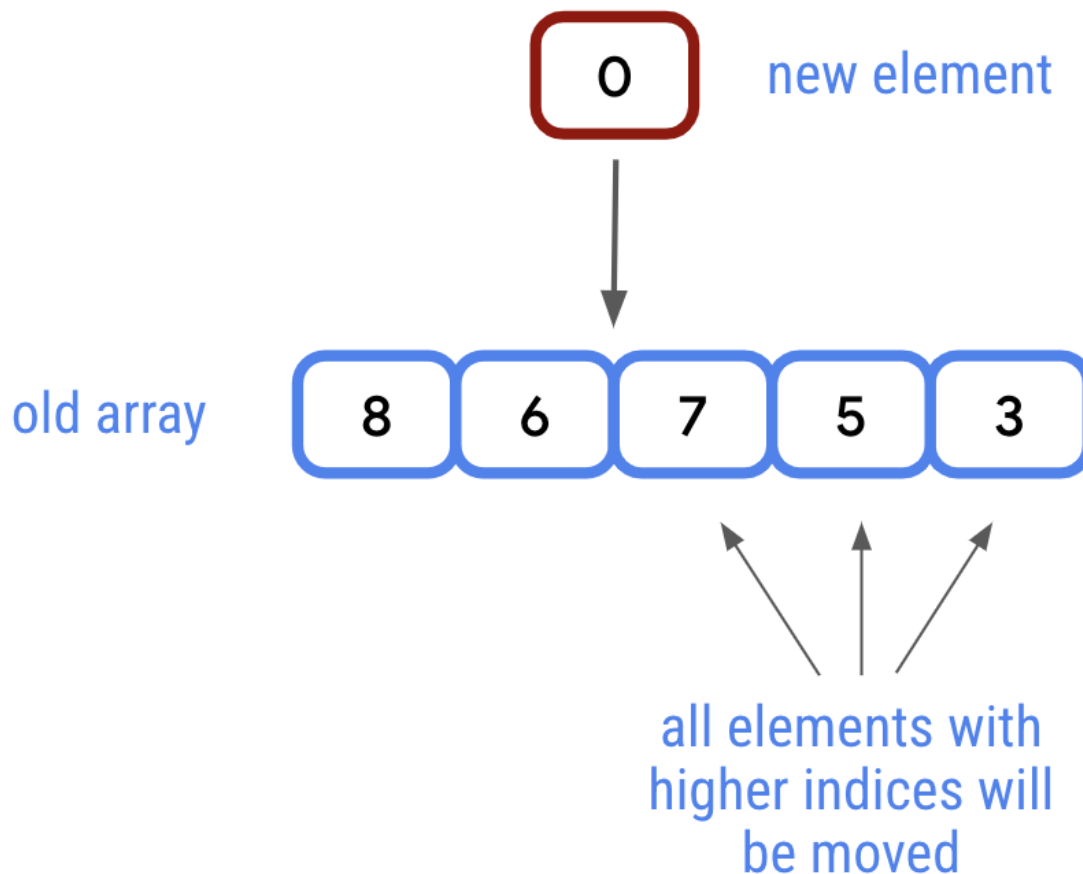
узнаете об общих свойствах и методах, а также о ситуациях, в которых можно использовать эти типы коллекций.

Списки List

Список - это упорядоченная, изменяемая по размеру коллекция, обычно реализуемая в виде изменяемого по размеру массива. Когда массив заполняется до отказа и вы пытаетесь вставить новый элемент, массив копируется в новый массив большего размера.



В списке можно также вставлять новые элементы между другими элементами по определенному индексу.



Именно так списки могут добавлять и удалять элементы. В большинстве случаев добавление любого элемента в список занимает одинаковое количество времени, независимо от того, сколько элементов в списке. Время от времени, если добавление нового элемента приведет к тому, что массив превысит свой определенный размер, элементы массива придется переместить, чтобы освободить место для новых элементов. Списки делают все это за вас, но за кулисами это просто массив, который при необходимости заменяется на новый.

List и MutableList

Типы коллекций, с которыми вы столкнетесь в Kotlin, реализуют один или несколько интерфейсов. Как вы узнали в лекции Generics, objects и extensions, интерфейсы предоставляют стандартный набор свойств и методов для реализации классом. Класс, реализующий интерфейс List, предоставляет реализацию всех свойств и методов интерфейса List. То же самое справедливо и для MutableList.

Так что же делают List и MutableList?

List - это интерфейс, определяющий свойства и методы, связанные с упорядоченной коллекцией элементов, доступной только для чтения.

MutableList расширяет интерфейс List, определяя методы для модификации списка, например, добавления и удаления элементов.

Эти интерфейсы определяют только свойства и методы List и/или MutableList. Класс, который их расширяет, сам определяет, как реализовать каждое свойство и метод. Вышеописанная реализация на основе массива - это то, что вы будете использовать чаще всего, если не всегда, но Kotlin позволяет расширять List и MutableList другими классами.

Функция listOf()

Как и arrayOf(), функция listOf() принимает элементы в качестве параметров, но возвращает не массив, а список.

- Удалите существующий код из main(). В main() создайте список планет под названием solarSystem, вызвав функцию listOf().

```
fun main() {  
    val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars", "Jupiter",  
        "Saturn", "Uranus", "Neptune")  
}
```

- Список имеет свойство size для получения количества элементов в списке. Выведите размер списка solarSystem.

```
println(solarSystem.size)
```

- Запустите свой код. Размер списка должен быть равен 8.

8

Доступ к элементам из списка

Как и в случае с массивом, вы можете получить доступ к элементу с определенным индексом из списка, используя синтаксис `subscript`. То же самое можно сделать с помощью метода `get()`. Синтаксис `subscript` и метод `get()` принимают `Int` в качестве параметра и возвращают элемент с этим индексом. Как и `Array`, `ArrayList` имеет нулевую индексацию, поэтому, например, четвертый элемент будет находиться по индексу 3.

- Выведите планету с индексом 2, используя синтаксис `subscript`.

```
println(solarSystem[2])
```

- Выведите элемент с индексом 3, вызвав функцию `get()` для списка `solarSystem`.

```
println(solarSystem.get(3))
```

- Запустите ваш код. Элемент с индексом 2 - «Земля», а элемент с индексом 3 - «Марс».

```
...  
Earth  
Mars
```

Помимо получения элемента по его индексу, вы также можете искать индекс конкретного элемента с помощью метода `indexOf()`. Метод `indexOf()` ищет в списке заданный элемент (переданный в качестве аргумента) и возвращает индекс первого вхождения этого элемента. Если элемент не встречается в списке, возвращается -1.

- Выведите результат вызова `indexOf()` для списка `solarSystem`, передав в качестве аргумента «Земля».

```
println(solarSystem.indexOf("Earth"))
```

- Вызовите `indexOf()`, передав «Pluto», и выведите результат.

```
println(solarSystem.indexOf("Pluto"))
```

- Запустите свой код. Есть элемент, соответствующий слову «Земля», поэтому выводится индекс 2. Элемента, соответствующего «Плутону», нет, поэтому выводится -1.

```
...  
2  
-1
```

Итерация элементов списка с помощью цикла for

Когда вы изучали типы функций и лямбда-выражения, вы видели, как можно использовать функцию `repeat()` для многократного выполнения кода.

Частой задачей в программировании является выполнение задачи один раз для каждого элемента списка. В Kotlin есть функция, называемая циклом `for`, которая позволяет выполнить эту задачу с помощью краткого и удобного синтаксиса. Часто это называют циклом по списку или итерацией по списку.

```
for ( element name in collection name ) {  
  
    body  
  
}
```

Чтобы выполнить цикл по списку, используйте ключевое слово `for`, за которым следует пара открывающих и закрывающих круглых скобок. Внутри круглых скобок укажите имя переменной, затем ключевое слово `in`, после которого следует имя коллекции. После закрывающей круглой скобки следует пара открывающих и закрывающих фигурных скобок, в которые вы включаете код, выполняемый для каждого элемента коллекции. Это называется телом цикла. Каждый раз, когда выполняется этот код, называется **итерацией**.

Переменная перед ключевым словом `in` не объявляется с помощью `val` или `var` - предполагается, что она будет только `get`. Вы можете назвать ее как угодно. Если список имеет имя во множественном числе, например `planets`, принято называть переменную в форме единственного числа, например `planet`. Также принято называть переменную `item` или `element`.

Это будет использоваться как временная переменная, соответствующая текущему элементу коллекции - элемент с индексом 0 для первой итерации, элемент с индексом 1 для второй итерации и так далее, и доступ к ней можно получить внутри фигурных скобок.

- Чтобы увидеть это в действии, выведите каждое название планеты в отдельной строке с помощью цикла `for`.
- В `main()`, под последним вызовом `println()`, добавьте цикл `for`. В круглых скобках назовите переменную `planet` и пройдите по списку `solarSystem`.

```
for (planet in solarSystem) {  
}
```

- Внутри фигурных скобок выведите значение planet с помощью функции println().

```
for (planet in solarSystem) {  
    println(planet)  
}
```

- Запустите свой код. Код в теле цикла выполняется для каждого элемента коллекции.

```
...  
Mercury  
Venus  
Earth  
Mars  
Jupiter  
Saturn  
Uranus  
Neptune
```

Добавление элементов в список

Возможность добавлять, удалять и обновлять элементы в коллекции доступна только классам, реализующим интерфейс **MutableList**. Если бы вы вели учет вновь открытых планет, вам, скорее всего, понадобилась бы возможность часто добавлять элементы в список. При создании списка, из которого вы хотите добавлять и удалять элементы, необходимо специально вызывать функцию `mutableListOf()`, а не `listOf()`.

Существует две версии функции `add()`:

- Первая функция `add()` имеет единственный параметр типа элемента в списке и добавляет его в конец списка.
- Другая версия функции `add()` имеет два параметра. Первый параметр соответствует индексу, в который должен быть вставлен новый элемент. Второй параметр - это элемент, добавляемый в список.

Давайте посмотрим на оба варианта в действии.

- Измените инициализацию `solarSystem`, чтобы вызывать `mutableListOf()` вместо `listOf()`. Теперь вы можете вызывать методы, определенные в `MutableList`.

```
val solarSystem = mutableListOf("Mercury", "Venus", "Earth", "Mars", "Jupiter",  
                                "Saturn", "Uranus", "Neptune")
```

- Опять же, возможно, мы захотим классифицировать Плутон как планету. Вызовите метод `add()` на `solarSystem`, передав в качестве единственного аргумента «Плутон».

```
solarSystem.add("Pluto")
```

- Некоторые ученые предполагают, что планета под названием Тея существовала до того, как столкнулась с Землей и образовала Луну. Вставьте слово «Тея» под индексом 3, между словами «Земля» и «Марс».

```
solarSystem.add(3, "Theia")
```

Обновление элементов по определенному индексу

Вы можете обновить существующие элементы с помощью синтаксиса субскриптов:

- Обновите значение с индексом 3 до «Future Moon».

```
solarSystem[3] = "Future Moon"
```

- Выведите значение с индексами 3 и 9, используя синтаксис подстрочных символов.

```
println(solarSystem[3])  
println(solarSystem[9])
```

- Запустите свой код, чтобы проверить результат.

```
Future Moon  
Pluto
```

Удаление элементов из списка

Элементы удаляются с помощью метода `remove()` или `removeAt()`. Вы можете удалить элемент, передав его в метод `remove()` или по его индексу с помощью `removeAt()`.

Давайте посмотрим оба метода удаления элемента в действии.

- Вызовите `removeAt()` для `solarSystem`, передав индекс 9. Это должно удалить «Плутон» из списка.

```
solarSystem.removeAt(9)
```

- Вызовите `remove()` для `solarSystem`, передав в качестве элемента для удаления «Future Moon». Это должно произвести поиск в списке, и если будет найден подходящий элемент, он будет удален.

```
solarSystem.remove("Future Moon")
```

List предоставляет метод `contains()`, который возвращает булево значение, если элемент существует в списке. Выведите результат вызова `contains()` для «Плутона».

```
println(solarSystem.contains("Pluto"))
```

Еще более лаконичный синтаксис - использование оператора `in`. Вы можете проверить, находится ли элемент в списке, используя элемент, оператор `in` и коллекцию. Используйте оператор `in`, чтобы проверить, содержит ли `solarSystem` «Future Moon».

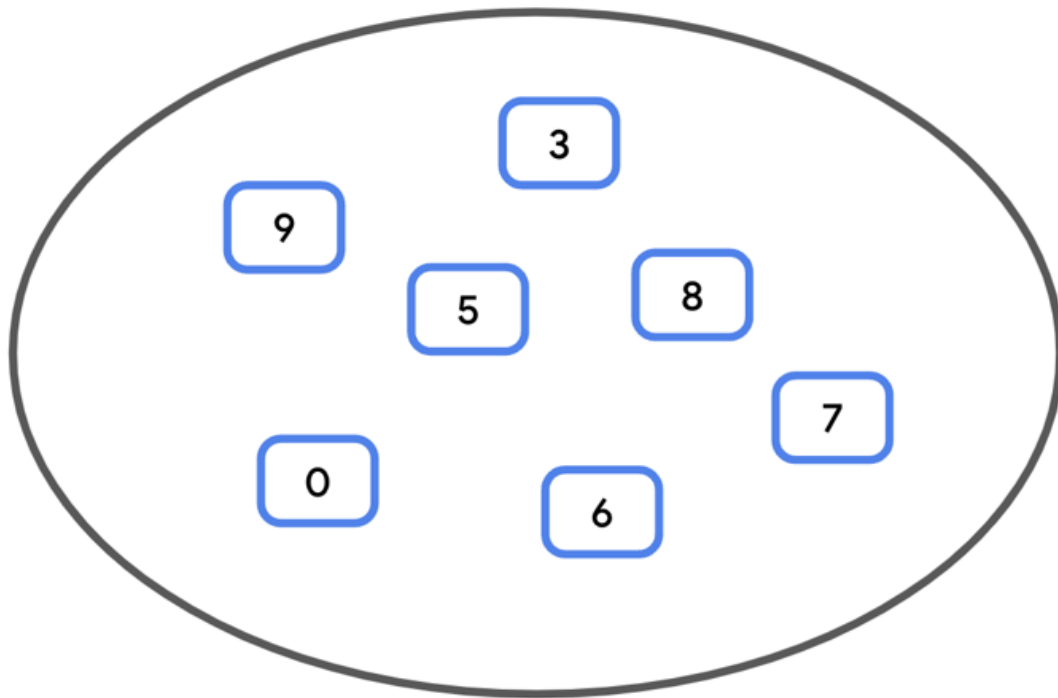
```
println("Future Moon" in solarSystem)
```

- Выполните свой код. Оба утверждения должны вывести `false`.

```
...  
false  
false
```

Наборы Set

Набор Set - это коллекция, которая не имеет определенного порядка и не допускает дублирования значений.

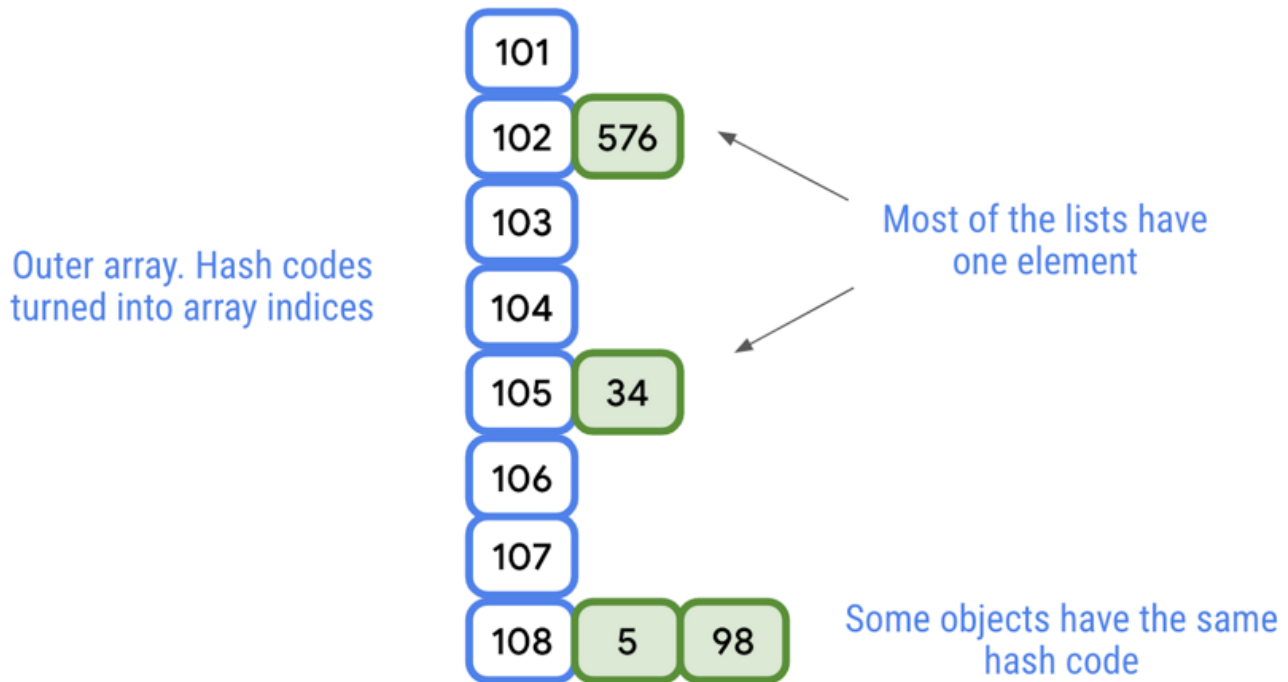


Как можно собрать такую коллекцию? Секрет заключается в хэш-коде. **Хэш-код** - это `Int`, получаемый методом `hashCode()` любого класса Kotlin. Его можно рассматривать как полууникальный идентификатор для объекта Kotlin. Небольшое изменение объекта, например, добавление одного символа в строку, приводит к значительному изменению хэш-значения. Хотя два объекта могут иметь одинаковый хэш-код (это называется хэш-коллизией), функция `hashCode()` обеспечивает определенную степень уникальности, и в большинстве случаев два разных значения имеют уникальный хэш-код.

```
"Kotlin".hashCode() // -204170231
```

```
"Kotlin!".hashCode() // 1131585312
```

Примечание: `Set` использует хэш-коды в качестве индексов массива. Конечно, может существовать около 4 миллиардов различных хэш-кодов, так что набор - это не просто один гигантский массив. Вместо этого можно представить набор как массив списков.



Внешний массив - числа, выделенные синим цветом слева, - каждый из них соответствует диапазону (также известному как bucket) возможных хэш-кодов. Каждый внутренний список, заштрихованный зеленым цветом справа, представляет собой отдельные элементы набора. Поскольку хэш-коллизии относительно редки, даже при ограничении потенциальных индексов внутренние списки в каждом индексе массива будут содержать только один или два элемента, если только не будут добавлены десятки или сотни тысяч элементов.

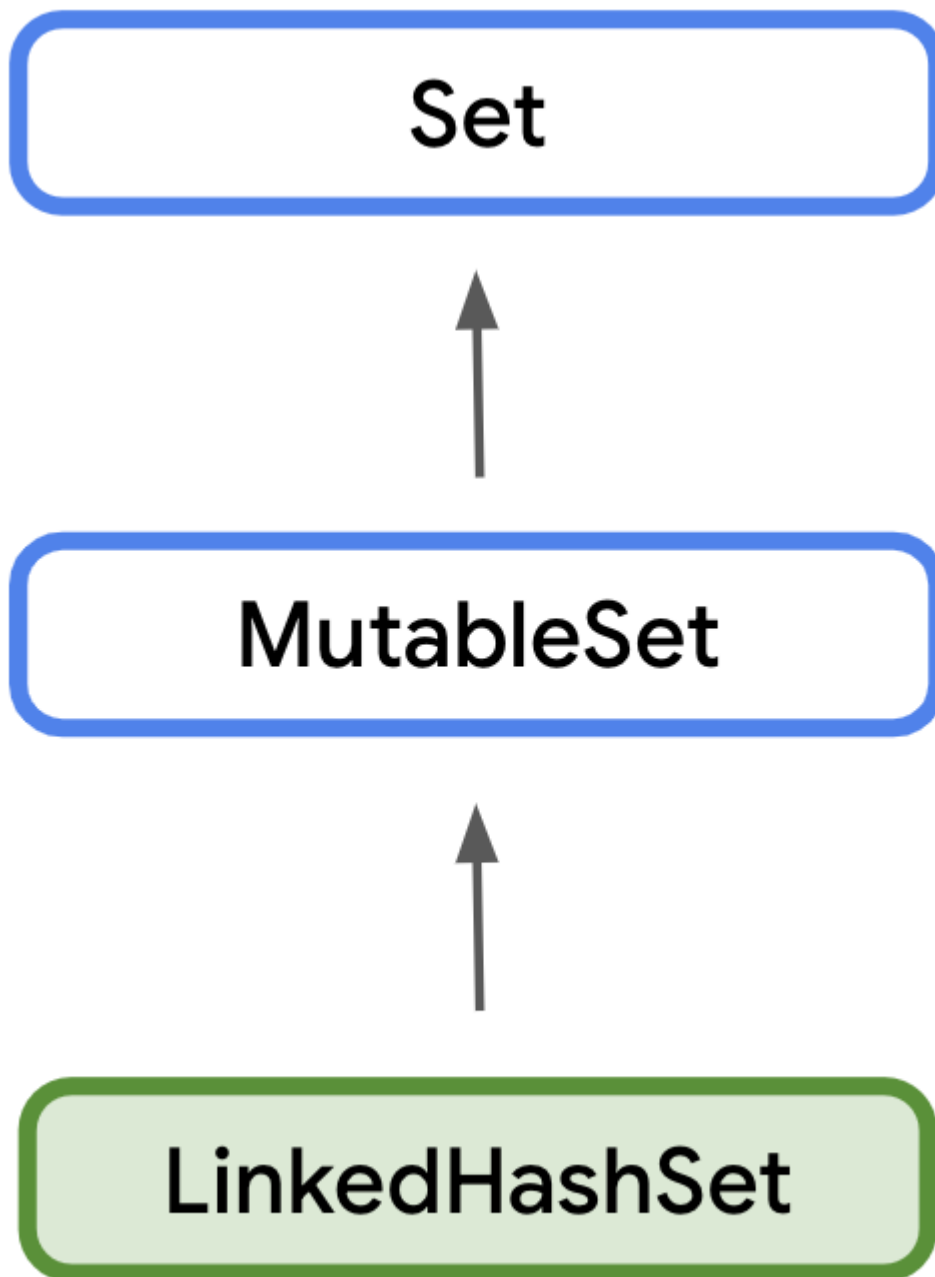
Наборы обладают двумя важными свойствами:

- Поиск определенного элемента в наборе происходит быстро по сравнению со списками, особенно для больших коллекций. В то время как функция `indexOf()` списка требует проверки каждого элемента с самого начала, пока не будет найдено совпадение, в среднем проверка наличия элемента в наборе занимает одинаковое количество времени, независимо от того, первый это элемент или сотысячный.
- Наборы, как правило, используют больше памяти, чем списки, при одинаковом объеме данных, поскольку часто требуется больше индексов массива, чем данных в наборе.

Примечание: Вопреки распространенному мнению, время, необходимое для проверки наличия элемента в наборе, не является фиксированным и зависит от количества данных в наборе. Однако, поскольку коллизий в хэше обычно немного, количество элементов, которые нужно проверить, все равно на порядки меньше, чем при поиске элемента в списке.

Преимущество наборов заключается в обеспечении уникальности. Если вы пишете программу для отслеживания новых открытых планет, набор предоставляет простой способ проверить, была ли планета уже открыта. При больших объемах данных это часто предпочтительнее, чем проверка наличия элемента в списке, которая требует итерации по всем элементам.

Как и `List` и `MutableList`, существуют `Set` и `MutableSet`. `MutableSet` реализует `Set`, поэтому любой класс, реализующий `MutableSet`, должен реализовать оба.



Использование MutableSet в Kotlin

В примере мы будем использовать `MutableSet`, чтобы продемонстрировать, как добавлять и удалять элементы.

- Удалите существующий код из `main()`.
- Создайте набор планет под названием `solarSystem` с помощью функции `mutableSetOf()`. Эта функция возвращает `MutableSet`, реализацией которого по умолчанию является `LinkedHashSet()`.

```
val solarSystem = mutableSetOf("Mercury", "Venus", "Earth", "Mars", "Jupiter",  
                                "Saturn", "Uranus", "Neptune")
```

- Выведите размер набора с помощью свойства `size`.

```
println(solarSystem.size)
```

Как и у `MutableList`, у `MutableSet` есть метод `add()`. Добавьте «Плутон» в набор `solarSystem` с помощью метода `add()`. Он принимает только один параметр для добавляемого элемента. Элементы в наборах не обязательно располагаются по порядку, поэтому индекса не существует!

```
solarSystem.add("Pluto")
```

- Выведите размер набора после добавления элемента.

```
println(solarSystem.size)
```

Функция `contains()` принимает один параметр и проверяет, содержится ли указанный элемент в наборе. Если да, то возвращается `true`. В противном случае возвращается `false`.

- Вызовите `contains()`, чтобы проверить, содержится ли «Плутон» в `SolarSystem`.

```
println(solarSystem.contains("Pluto"))
```

- Запустите свой код. Размер увеличился, и теперь `contains()` возвращает `true`.

```
8
9
true
```

Примечание: В качестве альтернативы можно использовать оператор `in`, чтобы проверить, находится ли элемент в коллекции, например: `"Плутон" in solarSystem` эквивалентно `solarSystem.contains("Плутон")`.

- Как уже говорилось, наборы не могут содержать дубликаты. Попробуйте добавить «Плутон» еще раз.

```
solarSystem.add("Pluto")
```

- Снова напечатайте размер набора.

```
println(solarSystem.size)
```

- Выполните код еще раз. «Плутон» не добавляется, так как он уже есть в наборе. На этот раз размер не должен увеличиваться.

```
...  
9
```

- Функция `remove()` принимает один параметр и удаляет указанный элемент из множества. Используйте функцию `remove()`, чтобы удалить «Плутон».

```
solarSystem.remove("Pluto")
```

Примечание: Помните, что множества - это неупорядоченные коллекции. Невозможно удалить значение из набора по его индексу, поскольку у наборов нет индексов.

- Выведите размер коллекции и снова вызовите `contains()`, чтобы проверить, находится ли «Плутон» еще в наборе.

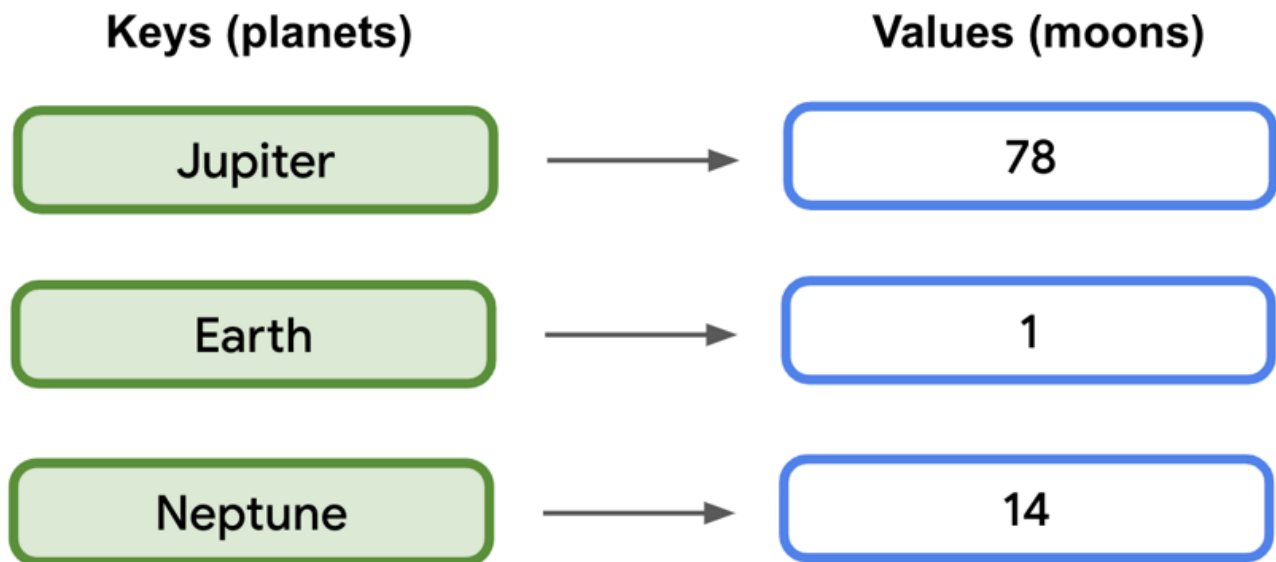
```
println(solarSystem.size)  
println(solarSystem.contains("Pluto"))
```

- Выполните свой код. «Плутон» больше не входит в набор, а его размер теперь равен 8.

```
...  
8  
false
```

Map Collection

Map - это коллекция, состоящая из ключей и значений. Она называется картой, потому что уникальные ключи сопоставляются с другими значениями. Ключ и сопутствующее ему значение часто называют парой ключ-значение.



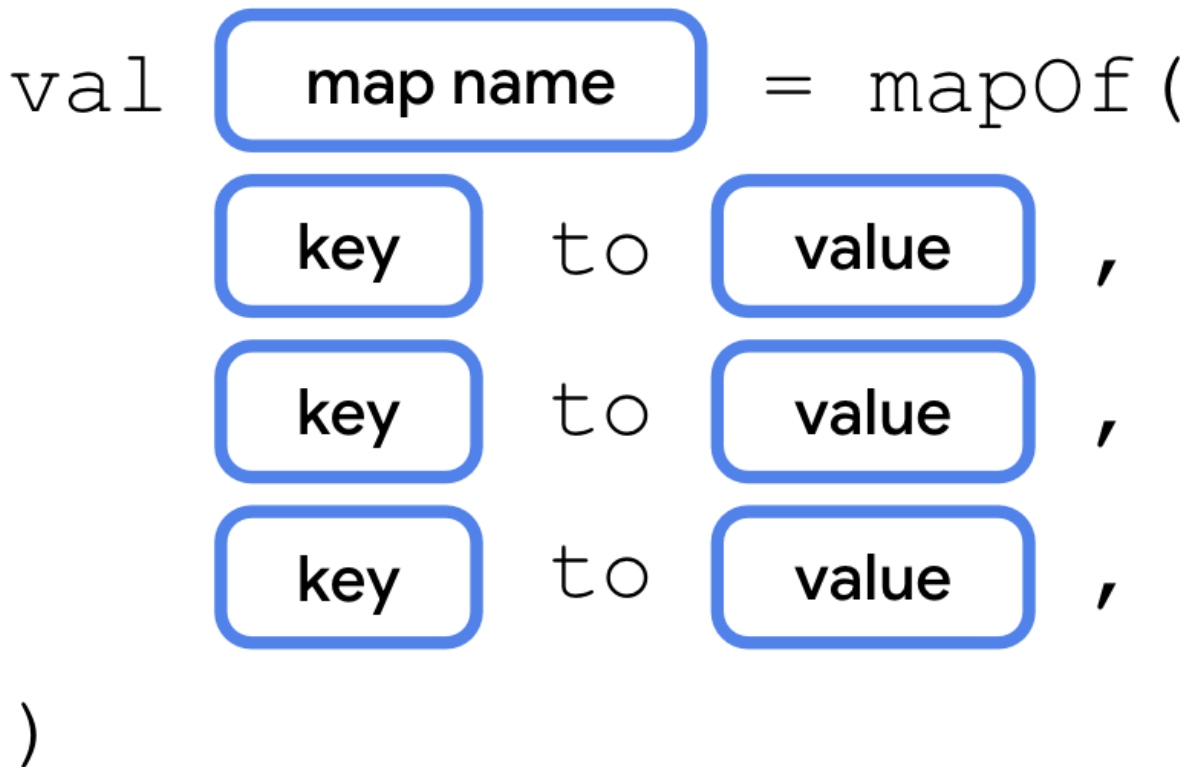
Ключи карты уникальны. Значения карты, однако, не уникальны. Два разных ключа могут соответствовать одному и тому же значению. Например, у «Меркурия» 0 лун, а у «Венеры» - 0 лун.

Доступ к значению из карты по его ключу обычно быстрее, чем поиск по большому списку, например, с помощью `indexOf()`.

Карты могут быть объявлены с помощью функции `mapOf()` или `mutableMapOf()`. Карты требуют двух общих типов, разделенных запятой - один для ключей, другой для значений.

```
mutableMapOf<key type, value type>()
```

Карта также может использовать вывод типа, если у нее есть начальные значения. Чтобы заполнить карту начальными значениями, каждая пара ключ-значение состоит из ключа, за которым следует оператор `to`, а затем значение. Каждая пара разделяется запятой.



The diagram illustrates the syntax for creating a map in Kotlin using the `mapOf` function. It shows the following structure:

```
val map name = mapOf (
    key to value ,
    key to value ,
    key to value ,
)
```

Each `key` and `value` is enclosed in a blue rounded rectangle, and the entire map definition is enclosed in a larger blue rounded rectangle.

Давайте подробнее рассмотрим, как использовать карты, а также некоторые полезные свойства и методы.

- Удалите существующий код из `main()`.
- Создайте карту `solarSystem` с помощью функции `mutableMapOf()` с начальными значениями, как показано на рисунке.

```
val solarSystem = mutableMapOf(
    "Mercury" to 0,
    "Venus" to 0,
    "Earth" to 1,
    "Mars" to 2,
    "Jupiter" to 79,
    "Saturn" to 82,
    "Uranus" to 27,
    "Neptune" to 14
)
```

- Как и списки и наборы, `Map` предоставляет свойство `size`, содержащее количество пар ключ-значение. Выведите размер карты `solarSystem`.

```
println(solarSystem.size)
```

- Для задания дополнительных пар ключ-значение можно использовать синтаксис подстрочных символов. Установите для ключа «Плутон» значение 5.

```
solarSystem["Pluto"] = 5
```

- Выведите размер еще раз, после вставки элемента.

```
println(solarSystem.size)
```

- Для получения значения можно использовать синтаксис подстрочных символов. Выведите количество лун для ключа «Плутон».

```
println(solarSystem["Pluto"])
```

Вы также можете получить доступ к значениям с помощью метода `get()`. Независимо от того, используете ли вы синтаксис subscript или вызываете `get()`, существует вероятность, что переданного вами ключа не окажется в карте. Если пары ключ-значение нет, метод `get()` вернет `null`. Выведите количество лун для планеты «Тея».

```
println(solarSystem.get("Theia"))
```

- Запустите свой код. Должно быть выведено количество лун для Плутона. Однако, поскольку Тейи нет на карте, вызов `get()` возвращает `null`.

```
8  
9  
5  
null
```

Метод `remove()` удаляет пару ключ-значение с указанным ключом. Он также возвращает удаленное значение или `null`, если указанного ключа нет в карте.

- Выведите результат вызова `remove()` и передачи значения «Pluto».

```
solarSystem.remove("Pluto")
```

- Чтобы убедиться, что элемент был удален, напечатайте размер еще раз.

```
println(solarSystem.size)
```

- Запустите ваш код. Размер карты после удаления записи равен 8.

```
...  
8
```

- Синтаксис subscript, или метод put(), также может изменить значение уже существующего ключа. С помощью синтаксиса subscript обновите число лун Юпитера до 78 и выведите новое значение.

```
solarSystem["Jupiter"] = 78  
println(solarSystem["Jupiter"])
```

- Запустите свой код. Значение существующего ключа «Jupiter» будет обновлено.

```
...  
78
```

Заключение

Поздравляем! Вы узнали об одном из самых фундаментальных типов данных в программировании - **массиве**, а также о нескольких удобных типах коллекций, построенных на основе массивов, включая **List**, **Set** и **Map**. Эти типы коллекций позволяют группировать и упорядочивать значения в коде. Массивы и списки обеспечивают быстрый доступ к элементам по их индексу, а наборы и карты используют хэш-коды для облегчения поиска элементов в коллекции. Эти типы коллекций будут часто использоваться в будущих приложениях, и знание того, как их использовать, поможет вам в вашей будущей карьере программиста.

Резюме

- **Array** хранят упорядоченные данные одного типа и имеют фиксированный размер.
- Массивы используются для реализации многих других типов коллекций.
- **List** - это упорядоченные коллекции с изменяемым размером.
- **Set** - это неупорядоченные коллекции, которые не могут содержать дубликаты.
- **Map** работают аналогично наборам и хранят пары ключей и значений определенного типа.