Казанский федеральный университет

СУНЦ КФУ
IT-лицей

НАУКА И УНИВЕРСИТЕТЫ
НАЦИОНАЛЬНЫЕ ПРОЕКТЫ РОССИИ

ПРОЕКТИРОВАНИЕ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ НА IOS И ANDROID

МЕТОДИЧЕСКАЯ БИБЛИОТЕКА

КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

**Специализированный учебный научный центр –
общеобразовательная школа-интернат «IT-лицей»**

ПРОЕКТИРОВАНИЕ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ НА IOS И ANDROID

Учебное пособие

КАЗАНЬ

2022

УДК 004.9
ББК –018*32.973
П79

Печатается по рекомендации ученого совета
СУНЦ IT-лицей КФУ
(протокол № 2 от 25.09.2022)

Авторы:
зав. сектором высокоуровневой разработки Института вычислительной математики
и информационных технологий КФУ **Ф.М. Мусин;**
зам. директора Института вычислительной математики и информационных технологий КФУ,
кандидат технических наук **А.А. Егорчев;**
директор Института вычислительной математики и информационных технологий КФУ,
доктор технических наук **Д.Е. Чикрин;**
зам. директора СУНЦ IT-лицей КФУ,
учитель информатики высшей квалификационной категории **Ш.Б. Азимуратов**

Рецензент
кандидат технических наук, доцент **М.П. Шлеймович**
(Казанский национальный исследовательский технический университет
им. А.Н. Туполева – КАИ)

П79 **Проектирование мобильных приложений на IOS и Android** [Электронный ресурс]: учебное пособие / Ф.М. Мусин, А.А. Егорчев, Д.Е. Чикрин, Ш.Б. Азимуратов. – Электрон. текстовые дан. (1 файл: 3,84 Мб). – Казань: Издательство Казанского университета, 2022. – 66 с. – Систем. требования: Adobe Acrobat Reader. – URL: https://kpfu.ru/portal/docs/F_951336359/ANDROID.i.IOS.pdf. – Загл. с титул. экрана.

ISBN 978-5-00130-648-1

Учебное пособие посвящено разработке приложений для мобильных устройств, под мобильные платформы iOS и Android. Пособие предназначено для обучающихся старших классов, изучающих информатику углубленно, позволит усвоить базовые понятия разработки приложений.

УДК 004.9
ББК –018*32.973

ISBN 978-5-00130-648-1

© Издательство Казанского университета, 2022

ОГЛАВЛЕНИЕ

Предисловие 4
Глава 1. Мобильная разработка на Андроид. 6
1.1. Введение в разработку на Котлин для Андроид 6
1.2. Базовые понятия языка Котлин: объявление переменной, типы данных, преобразования типов, понятие оператора, понятие функции 7
1.3. Понятие значения Null safety. 12
1.4. Комментарии в Котлин 12
1.5. Основные операторы: условные операторы if, when, операторы цикла for и while. . 14
1.6. Классы 17
1.7. Базовые коллекции в Котлин. 19
1.8. Установка инструментов разработчика, первая программа на Андроид. 21
1.9. Структура Андроид проекта. 26
1.10 Макет приложения. 28
1.11. Взаимодействие с интерфейсом 36
Контрольные вопросы и задачи 37
Глава 2. Мобильная разработка на iOS. 39
2.1. Введение в iOS 39
2.2. Введение в разработку на Swift для iOS 40
2.3. Базовые понятия языка Swift: объявление переменной, типы данных, преобразования типов, понятие оператора, понятие функции 41
2.4. Понятие Optional. 46
2.5. Комментарии в Swift. 47
2.6. Основные операторы: условные операторы if, switch, операторы цикла for и while… 47
2.7. Классы. 50
2.8. Базовые коллекции в Swift 53
2.9. Установка инструментов разработчика, первая программа iOS. 54
2.10. Структура iOS проекта. 56
2.11. Макет приложения. 57
2.12. Взаимодействие с интерфейсом. 62
Контрольные вопросы и задачи 63
ЛИТЕРАТУРА 65

Предисловие

Разработка приложений для мобильных устройств – быстрорастущая область информационных технологий. Этому есть несколько причин:

- появление очень большого количества различных мобильных телефонов и смартфонов;
- активное увеличение покрытия мобильного Интернета;
- простота и дешевизна мобильных устройств.

Большая часть мобильных устройств работает под управлением двух операционных систем (ОС) – Андроид и iOS, занимая приблизительно 74 и 25 % соответственно, доля всех остальных ОС составляет не более 1 %. Если говорить об абсолютных числах, количество активных устройств на базе ОС Андроид достигает 2,5 млрд устройств, на базе ОС iOS – более 1,4 млрд.

У разработки приложений для Андроид и iOS есть как общие принципы, так и существенные различия. Рассмотрим сначала общие принципы.

Вне зависимости от ОС любому мобильному разработчику нужно уметь создавать интерфейсы на основе макета, подстраивающиеся под разные экраны. Не менее важно также уметь создавать сетевые запросы и получать ответы от сервера, уметь работать с базами данных. Умение обработать данные тоже будет нелишним, прежде всего это касается формата JSON *(текстовый формат обмена данными, основанный на JavaScript)*, который де-факто является стандартным форматом для работы клиент-серверных мобильных приложений. **Клиент-серверное приложение** – это такое приложение, которое состоит из двух частей: клиентской, работающей на устройстве пользователя, например на смартфоне, и серверной, работающей на удаленном компьютере; взаимодействие обоих частей происходит по компьютерной сети, чаще всего через Интернет. Также в тексте будет встречаться понятие **компилятора**. Компилятор – это программа, написанная разработчиками языка, чтобы переводить текст кода программы в машинный код, который в дальнейшем выполняется компьютером. Кроме знания синтаксиса языков программирования также обязательно знание стандартной библиотеки языка и наиболее популярных библиотек (или по-другому – фреймворков).

Из особенностей разработки для Андроид и iOS прежде всего хочется отметить:

1. Разнообразие устройств. На базе ОС Андроид выпускается огромное количество различных устройств – с различными размерами экранов, различными техническими характеристиками. Более того, многие производители

смартфонов при добавлении Андроид на свое устройство также добавляют огромное количество своих приложений, начиная от текстовых редакторов и заканчивая системными процессами отправки уведомлений. Все это накладывает определенные сложности при отладке и настройке своего приложения на базе ОС Андроид. С iOS с этой стороны проще: количество версий смартфонов, размеров экрана и самой операционной системы ограниченно, поэтому сделать приложение, которое одинаково хорошо работает на всех устройствах, намного проще, чем на других платформах.

2. Обновления ОС. Для iOS всегда есть список нескольких версий ОС, которые официально поддерживаются, все устройства получают фиксированное количество обновлений, поэтому разработчику гораздо проще поддерживать свое приложение – старые версии ОС со временем можно просто отключать от поддержки.

Глава 1. Мобильная разработка на Андроид

1.1. Введение в разработку на Котлин для Андроид

На конференции Google I/O в 2019 году компанией «Гугл» было объявлено об ускоренном переходе разработки приложений для Андроид на язык программирования Котлин. Сегодня подтверждением этому являются следующие факты: все больше сторонних разработчиков переходят на новый язык, официальная документация Андроид с приоритетом выходит для Котлин, официальные инструменты разработки по умолчанию ориентируются на разработку на Котлин и т. д. У самой компании «Гугл» более 60 приложений созданы на Котлин или уже полностью переведены на него, например Google Maps, Home, Google Play, Google Drive и т. д. Почему же было принято решение переходить с языка программирования Джава на Котлин? Котлин – это выразительный и краткий язык программирования, который позволяет избежать многих распространенных ошибок и легко интегрировать новые функции в имеющиеся проекты. Решение компанией «Гугл» было принято на основании множественных отзывов от других разработчиков, по мнению которых Котлин:

1. Выразительный и краткий. Вы можете делать больше, написав меньше кода. Язык содержит много так называемого синтаксического сахара, т. е. многие стандартные обороты, которые в Джаве пришлось бы реализовывать самому разработчику, уже изначально есть в Котлине. Из всех профессиональных разработчиков, работавших с Котлин, примерно $^{2}/_{3}$ утверждают, что переход на Котлин позволил им увеличить свою производительность.

2. Более безопасный код. В Котлине есть множество конструкций, которые позволяют избежать многих распространенных ошибок, таких, например, как ошибки с null pointer exceptions (этот термин используется для случаев, когда вместо ожидаемого экземпляра объекта обнаруживается пустой объект, что может привести к неожиданному завершению работы программы – так называемый crash). По статистике в приложениях, написанных на Котлин, встречается на 20 % меньше случаев crash, чем в приложениях на Джава.

3. Обратная совместимость. Котлин полностью совместим с Джава. Это значит, что из кода, написанного на Котлин, можно вызывать код на Джава и, наоборот, из Джава-кода можно спокойно вызывать код на Котлин. Таким образом, если имеется старый проект, написанный на Джава, необязательно полностью его переписывать на Котлин, можно лишь добавлять новые модули и функционал, а уже по мере возможности или необходимости переводить и старые модули.

4. Структурный параллелизм. В Котлин имеется специальный набор функций, так называемая библиотека – Coroutines (Корутины), которая значительно облегчает работу параллельных вычислений, или асинхронного кода. Асинхронный код – это код, который выполняется не подряд, а параллельно, в Андроид это прежде всего касается работы интерфейса и фоновых задач (доступ к локальным данным, сетевой запрос, какие-то сложные расчеты и т. д.), фоновые задачи не должны блокировать интерфейс. Обычно отслеживание работы асинхронного кода весьма трудоемко, но Корутины многие мелкие задачи берут на себя, что позволяет разработчику сосредоточиться о логике проекта, не задумываясь об этих мелочах.

1.2. Базовые понятия языка Котлин: объявление переменной, типы данных, преобразования типов, понятие оператора, понятие функции

Изучение разработки мобильных приложений для Андроид стоит начать с основ языка Котлин. Наиболее простым способом проверить работоспособность кода можно в онлайн-тренажере от разработчиков языка по адресу: https://play.kotlinlang.org. Этот тренажер содержит текстовый редактор, в который можно вводить свой код и компилятор. Чтобы проверить работоспособность своей программы, ее надо **запустить (Run).**

Первым пунктом будет изучение ключевого слова для обозначения переменной – это **val** или **var**. При обозначении переменной далее следует имя переменной и по желанию можно указать тип переменной. Ключевое слово **val** используется для обозначения переменной, которая в дальнейшем не будет изменяться, этой переменной нельзя будет присвоить новое значение. Ключевое слово **var** используется, соответственно, для обозначения переменных, которые могут изменяться.

```
val dog = "Шарик"
var age = 17
// необязательно, но укажем тип
val height: Int = 165
var dogName: String = "Барбос"
```

В примерах выше переменные dog, height объявлены как неизменяемые, а переменные `age, dogName` объявлены как изменяемые.

Иногда тип указывать обязательно.

```
val a: Any = 12
val context: Context = activity
```

Во всех случаях, когда позволяет логика программы, лучше использоваться **val**.

Котлин является языком со статической типизацией, т. е. переменной одного типа нельзя присваивать переменную другого типа, только явным образом преобразовав из одного типа в другой. Выше мы рассмотрели несколько примеров с объявлением переменных, где у некоторых, как можно было заметить, указан тип переменной, например для `dogName` это `String` – тип строки, для `height` это `Int` – тип для целого числа. Базовыми типами в Котлин являются **Boolean, Char**, **Short**, **Int, Long, Float** и **Double.**

Тип **Boolean** может принимать два значения – **true** или **false**, истина или ложь соответственно. Основные операции над этим типом:

```
&& // логическое AND
|| // логическое OR
! // логическое NOT
```

Тип **Char** используется для объявления символов, в коде символ обрамляется одиночными кавычками:

```
val letter: Char = 'c'
```

Short, **Int, Long** являются типами для натуральных чисел, каждый из которых используется в зависимости от величины числа, которое нужно хранить в переменной. **Float** и **Double** – для чисел с плавающей точкой, которые используются также в зависимости от величины переменной.

```
val i: Int = 9
val d: Double = 1598101.99
val f: Float = 10.99
```

Если тип явно не указывать, то он будет определен исходя из присвоенного значения, в данном случае типом переменной будет **Double.**

```
var weight = 10.99
```

Следующий пример приведет к ошибке при попытке запустить код:

```
val i: Int = 9
val d: Double = i// будет ошибка
```

Ошибка произойдет из-за того, что переменной типа **Double** попытались присвоить переменную типа **Int,** чего в Котлин делать нельзя из-за упомянутой статической типизации. Но можно провести явное преобразование типов:

```
val i: Int = 9
val d: Double = Double(i)
```

В этом случае никакой ошибки не будет и в переменную `d` запишется значение переменной `i`, равное 9.

Еще одним часто используемым типом в Котлин является тип **String**, который уже упоминался в примерах.

```
var city = "Kazan"
```

В данном случае тип переменной, пусть и не объявленный явно, будет определен исходя из присвоенного значения `"Kazan"`, т. е. **String**.

Перечисляя базовые типы любого языка программирования, также стоит отметить особый тип данных – **массивы**. Массив не является типом, который отвечает за определенный числовой тип или строковый, массив – это такой тип данных, который позволяет содержать в переменной множество определенного типа. Например, 5 чисел, или 10 строк, или 15 символов и т. д.

Например, массив из 4 строк с названиями фруктов:

```
val fruits = arrayOf("Apple", "Mango", "Banana", "Orange")
```

Или массив целых чисел:

```
val num = arrayOf(1, 2, 3, 4, 5)
```

Обратиться к элементу массива можно, используя квадратные скобки **[],** по номеру элемента в массиве, начиная с 0:

```
fruits[0]
```

В этом примере мы получим в качестве результата значение первого элемента массива, равное `"Apple"`.

Еще одним понятием любого языка программирования является понятие «**оператор**». Один из операторов мы уже рассмотрели выше – оператор присваивания, который обозначается знаком «=». Это не знак равенства из математики, как можно было бы подумать, а оператор, обозначающий, что переменной слева от знака равенства мы *присвоили* значение справа от знака равенства, т. е. после выполнения этого оператора при обращении к этой переменной мы получим то самое значение:

```
var city = "Kazan"
```

В данном случае переменной `city` мы присвоили значение `"Kazan"`.

В Котлин имеется множество и других операторов, основные из которых: операторы арифметических действий (сложение, вычитание, умножение, деление и т. д.), оператор проверки равенства «==», условные операторы **if, when**, операторы цикла **for, while**. Позднее мы рассмотрим эти операторы более подробно.

Множество операторов можно объединить в одну общность – **функцию.** Функция объявляется при помощи ключевого слова **fun,** далее указывается наименование функции, далее в круглых скобках перечисляются **аргументы функции** (функция также может не иметь никаких аргументов), далее после знака **«:»** может указываться тип **возвращаемого значения функции –** т. е. по окончании выполнения кода внутри функции она *возвращает* какое-то значение в место своего вызова (либо у функции может не быть возвращаемого значения, тогда тип не указывается). И наконец, в фигурных скобках пишется код – так называемое **тело функции**.

```
fun multiply(a: Int, b: Int): Int {
val result = a * b
return result
}
```

В примере описана функция с названием `multiply`, которая, как можно догадаться из названия, служит для умножения чисел. Она принимает в себя два целочисленных аргумента – `a` и `b`. Далее видно, что функция возвращает результат в виде целого числа `Int` (что логично, поскольку Котлин является языком со статической типизацией, и, следовательно, результатом умножения двух целых чисел будет целое число). В теле функции имеется два оператора – оператор умножения «`*`» и оператор `return`, который возвращает полученный ответ в виде числа `result`.

Функцию можно вызвать с помощью **оператора вызова функции,** он обозначается круглыми скобками **«()».** Вызов функции из примера выше будет выглядеть следующим образом:

```
val answer = multiply(15, 42)
```

При вызове функции внутри круглых скобок указываются все аргументы, которые были объявлены при создании функции, в примере выше функция `multiply` принимает в себя два целых числа. Результат выполнения функции запишется в переменную `answer`, которую можно использовать далее в программе.

Кроме обычных функций в Котлин есть так называемые **анонимные функции**. У таких функций, как можно догадаться, нет названия, и объявление выглядит немного по-другому:

```
val multiply: (Int, Int) -> Int = { a, b ->
a * b
}
```

В примере выше нет вызова функции, лишь объявление анонимной функции. Эта функция принимает в себя два целочисленных аргумента `a, b` и возвращает в качестве результата выполнения целое число. В `multiply` запишется ссылка на функцию, по которой в дальнейшем эту функцию можно будет вызвать:

```
val answer = multiply(23, 189)
```

Напоследок стоит рассмотреть одну из стандартных функций Котлин, которая будет часто использоваться – функцию вывода `println()`. Эта функция

принимает в себя строковое значение, которое выводит в консоль (про консоль вывода будет рассказано позднее).

```
println(«Эта строка будет выведена на консоль»)
```

1.3. Понятие значения Null safety

В предыдущем разделе мы рассмотрели базовые типы языка Котлин, эти типы еще часто называют примитивными типами. Кроме примитивных типов многие языки программирования поддерживают еще так называемые ссылочные типы, и Котлин в данном случае не исключение. Как можно было видеть из примеров выше, переменную можно объявить без присваивания ей начального значения. Если так объявить переменную ссылочного типа, то во многих языках значением по умолчанию у переменной станет **null** – пустое значение, или отсутствие какого-либо значения. В Котлин обычная переменная не может иметь значение null, т. е., например, приведенное ниже объявление переменной будет неверным:

```
val country: String = null
```

Чтобы переменная могла иметь значение null, она должна быть *nullable*, т. е. необходимо явно указать с помощью знака **«?»**, что переменная может содержать в себе null:

```
Val country: String? = null
```

Теперь эта переменная может содержать как значения типа **String**, так и значение null. При обращении к такой переменной нужно приписывать знак **«?» (**в случае, когда мы не знаем точно, содержит в себе переменная какое-либо значение или null) либо знак **«!»** (если мы уверены, что переменная на данный момент точно не содержит null).

1.4. Комментарии в Котлин

При написании кода очень полезным бывает оставлять **комментарии** – другим разработчикам, которые будут в дальнейшем работать с этим кодом, самому себе, когда появится необходимость вернуться к проекту через некото-

рое время, для написания документации и т. д. Kotlin, как и большинство других языков программирования, позволяет это делать. При проверке и выполнении кода комментарии компьютером игнорируются.

В Котлин комментарии бывают нескольких видов.

Первый способ – так называемый комментарий в одну строку, он выглядит следующим образом:

```
// Это комментарий. Он не выполняется.
```

Для комментирования нескольких строк имеется другой способ написания **многострочных комментариев в Kotlin**, к примеру:

```
/* Это также комментарий.
   На много..
   много...
   много строчек. */
```

Последним видом являются **вложенные комментарии**:

```
/* Это комментарий.

/* И внутри него
находится
другой комментарий.
*/

Вернемся к первому.
*/
```

Комментарии очень полезны, они позволяют документировать код, пояснять сложные моменты в алгоритме для других программистов, в конце концов, ими можно оставлять шутки для коллег.

1.5. Основные операторы: условные операторы if, when, операторы цикла for и while

Котлин предоставляет большое количество возможностей для реализации условной логики. Самый часто используемый и простой – условный оператор **if else.** Логика работы этого оператора очень проста: если значение выражения в круглых скобках после ключевого слова **if** будет истинным **(true)**, то выполнится код, следующий сразу после круглых скобок внутри фигурных скобок. В противном случае выполнится код, расположенный после ключевого слова **else.** Часть оператора, начинающаяся с ключевого слова **else**, необязательна – в этом случае если выражение после **if** равно false, то ничего выполнено не будет, оператор будет просто пропущен.

```
if (result = 100) {
 println("Результат 100%!")
} else {
 println("Есть куда стремиться!")
}
```

Один оператор может содержать в себе несколько веток **if else.** Это позволяет реализовывать более сложную логику:

```
var x = 5
… // тут могут быть какие-то преобразования с переменной x
if (x > 0) {
    println("x больше чем 0")
} else if (x == 0) {
    println("x равен 0")
} else {
    println("x меньше чем 0")
}
```

Код из примера выше проверяет, какой знак у значения переменной *x*: больше 0, равно 0 или меньше 0.

В Котлин условный оператор также может работать как выражение (так называемое **условное выражение**), которое можно присвоить переменной:

```
var x = 5
… // тут могут быть какие-то преобразования с переменной x
val result: String = if(x > 0) {
 “x больше чем 0”
} else if (x == 0) {
 “x равен 0”
} else {
 “x меньше чем 0”
}
```

Каждое из выражений внутри фигурных скобок должно содержать в конце блока выражение одного и того же типа, в примере выше это строки. В этот блок можно добавить и другие выражения, но если они не будут использоваться для вычисления конечного выражения, то будут проигнорированы. В итоге в переменную `result` запишется одно из трех строковых значений.

Для более сложных условий вместо оператора **if else** можно использовать другой условный оператор – **when.**

```
val grade = 5
val compliment = when (percent) {
    5 -> "Отлично!"
    4 -> "Хорошо!"
    3 -> "Удовлетворительно"
    2 -> "Удовлетворительно"
    else -> "Неверная оценка"
}


println(compliment)
```

В примере выше по значению оценки (в примере это 5) получили соответствующую ей строку (`"Отлично!"`). Оператор **when** проверяет переменную `grade` на равенство разным значениям – 5, 4, 3, 2, а ключевым словом

`else` проверяются все остальные значения, для нашего примера любая оценка, отличная от 2, 3, 4 и 5, является неверной. При совпадении какого-то из значений в теле оператора со значением проверяемой переменной оператор возвращает ответ, который присваивается переменной `compliment`. Изменяя значения `grade`, попробуйте получить разные значения строки `compliment`.

Операторы цикла используются в том случае, когда необходимо одно и то же действие выполнить некоторое количество раз. Одним из наиболее частых примеров использования цикла является работа с массивами. Как отмечалось ранее, массив является множеством элементов определенного типа, и обращение к элементам массива осуществляется путем обращения по индексу. Однако в Котлин имеется возможность использовать цикл для работы с массивом без обращения к индексам. Для этого используется оператор цикла **for:**

```
    val cars = arrayOf("Kalina", "Vesta", "Largus",
"Granta")
    for (car in cars) {
      println(car)
    }
```

В первой строке примера объявляется массив автомобилей `cars`, который состоит из четырех элементов типа String. Далее объявляется оператор `for`, который позволяет обойти все элементы массива и распечатать их в консоль. Действие `car in cars` определяет, что цикл будет проходить по массиву, доставая из него по одному элементу – и присваивая значение этого элемента в переменную `car`.

При работе с оператором **for** полезным также может оказаться оператор **range** – множество значений одного типа, аналогичное массиву. Например:

```
    for (num in 0..20) {
      println(num)
    }
```

В этом примере оператор позволяет пройти по множеству чисел от 0 до 20 включительно. Этот же способ можно использовать для прохождения по массиву, когда необходимо явным образом получить доступ к индексу каждого

элемента. Добавлением ключевого слова **step** можно управлять, с каким шагом циклу проходить множество.

```
for (num in 0..20 step 2) {
  println(num)
}
```

В данном случае num будет присваиваться каждый второй элемент множества.

1.6. Классы

Класс представляет собой шаблон, определяющий свойства и функции, связанные с объектами этого типа. Свойство – это то, что объект знает о себе. Android уже содержит множество готовых классов, но все предусмотреть невозможно и любому программисту приходится создавать собственные классы. Для создания класса используется ключевое слово `class`. Например, объявим класс «велосипед»:

```
class Bicycle
```

Как было указано выше, класс обладает определенными свойствами, например, у велосипеда есть колеса:

```
class Bicycle{
var wheels = arraysOf<Wheel>()
}
```

В примере выше к классу *велосипед* добавлен массив *с колесами* (пустой). В качестве самостоятельного упражнения напишите реализацию класса `Wheel`.

После того как класс создан, можно создать **экземпляр** этого класса:

```
val bicycle = Bicycle()
```

Теперь через этот экземпляр класса *велосипед* можно получить доступ к колесам:

```
val wheels = bicycle.wheels
```

Точно так же можно, наоборот, изменить колеса у велосипеда, если уже имеется созданный массив колес `wheels`:

```
bicycle.wheels = wheels
```

В Котлин у класса кроме свойств еще может быть **конструктор.** Конструктор – это специальная конструкция, которая позволяет задать определенные свойства класса сразу при создании экземпляра. Например, если уже имеется массив колес `wheels`, то можно его задать сразу при создании экземпляра класса. Для начала нужно объявить конструктор и затем при создании экземпляра можно сразу передать массив внутрь класса:

```
class Bicycle(val wheels: List<Wheel>) {
}

val bicycle = Bicycle(wheels)
```

У класса также есть **функции**. Они позволяют моделировать поведение класса. Здесь также стоит обратить внимание на такое понятие, как **инкапсуляция.** Инкапсуляция – это понятие в программировании, при котором не все свойства и функции класса доступны при обращении к экземпляру класса. В Котлин основные уровни доступа определяются ключевыми словами **private** и **public**. Обычно свойства класса определяют **private**, не позволяя изменять свойства класса напрямую.

```
class Bicycle(val wheels: List<Wheel>) {

private val lock: Lock = Lock()

fun unlock(key: Key): Boolean {
// функция пытается открыть замок, и в зависимости от
того, получилось или нет, возвращает true/false
}
}
```

В качестве самостоятельного упражнения напишите реализацию классов `Lock` и `Key`, а также метода `unlock`.

1.7. Базовые коллекции в Котлин

Кроме базовых примитивных типов и массивов любой солидный язык старается предоставить более сложные конструкции. И Котлин не исключение – стандартная библиотека языка предоставляет огромное количество разных объектов на разные случаи жизни. Самыми часто используемыми из этих конструкций являются **коллекции**. Коллекция – это структура данных, которая содержит какое-то количество объектов (напоминает массивы, не правда ли?). Чаще всего эти объекты называются «элементами».

В Котлин есть три вида самых часто используемых коллекций:

1. **Списки List.** Упорядоченная коллекция с доступом к элементам по индексу (целое число, определяющее позицию элемента в списке). Один и тот же элемент может входить в список более одного раза. Примером списка может служить множество цифр телефонного номера: множество состоит из элементов одного типа (целое число), оно упорядочено и порядок цифр являются определяющим для каждого множества.

2. **Множество Set.** Это коллекция уникальных элементов (ни один элемент не может входить в коллекцию более одного раза). Является аналогом множеств из области математики. Порядок элементов не определен и не важен. В качестве примера можно взять школьный класс: ни один ученик в классе не повторяется, порядок не определен.

3. **Map –** карта, ассоциативный массив или словарь. Принцип ее понятен из названия, это коллекция, которая содержит пары «ключ-значение». Ключи должны быть уникальными, соответственно каждый ключ однозначно соотносится с одним-единственным значением. Сами значения могут повторяться. Карты наиболее полезны для случаев, когда нужно в коллекции хранить объекты с логической привязкой, например данные граждан – номер паспорта и личные данные.

Коллекции в Котлин позволяют манипулировать данными независимо от конкретного типа элементов в них, т. е. добавление, удаление элементов типа String будет таким же, как и манипуляции с элементами типа Int, Double и т. д.

Еще стоит отметить, что в Котлин, в отличие от многих других языков программирования, коллекции делятся на две большие группы:

1. **Неизменяемые (read-only)** – коллекции данного типа предоставляют операции по доступу к элементам коллекции.

2. **Изменяемые (mutable)** – к операциям предыдущего типа добавляются операции с возможностью изменения коллекции: добавление, удаление элемента и обновление его свойств. Для того что изменять элементы коллекции, не обязательно объявлять ее через ключевое слово **var**, но вот для переопределения всей коллекции целиком все же потребуется.

Изменяемые коллекции можно создавать с помощью ключевого слова – приставки `mutable`:

```
val numbers = mutableListOf(1, 2, 3, 4, 5)// список
numbers.add(6)
println(numbers)

val numbers = mutableSetOf(1, 2, 3, 4, 5) // множество
numbers.add(6)
println(numbers)

val numbers = mutableMapOf("one" to 1, "two" to 2,
"three" to 3)
numbers.put("four", 4)
println(numbers)
```

В качестве самостоятельного задания попробуйте изменить код так, чтобы можно было списку, множеству и карте присвоить другое значение.

Для создания неизменяемого списка достаточно воспользоваться ключевым словом `listOf`:

```
val numbers = listOf(1, 2, 3, 4, 5)
numbers.add(6)
```

В примере выше будет ошибка, поскольку `numbers` объявлен неизменяемым, а ему попытались добавить новый элемент.

Аналогично создаются неизменяемые множество и карта:

```
val numbers = setOf(1, 2, 3, 4, 5) // множество
println(numbers)

val numbers = mapOf("one" to 1, "two" to 2, "three"
to 3)
println(numbers)
```

1.8. Установка инструментов разработчика, первая программа на Андроид

До сих пор мы изучали теоретические основы Котлин, теперь же посмотрим, как это все применимо к разработке на Андроид. Для полноценной разработки приложений онлайн-тренажера будет уже недостаточно, нужно скачать и поставить на компьютер специальную программу. Программы для разработки называются IDE – integrated development environment (интегрированная среда разработки). Для разработки на Андроид есть специальная IDE – Android Studio (Андроид Студио), скачать ее можно с официального сайта разработчика: https://developer.android.com/studio/.

После того как вы скачали Андроид Студию, ее нужно установить. Для этого запустите только что скачанный установочник.

Важный нюанс: перед установкой убедитесь, что имя пользователя компьютера не содержит никаких символов, кроме латинских букв и обычных цифр (кириллица недопустима!). При необходимости создайте нового пользователя, в имени которого не будет лишних символов. При установке следуйте инструкциям установочника.

По окончании установки запустите Андроид Студию, перед вами откроется стартовое окно с возможностью создать новый проект, открыть уже имеющийся или изменить какие-либо настройки. Нажмите кнопку *New Project*, у вас откроется окно настройки проекта, промотайте до пункта *Empty Activity* и выберите его (окно может немного отличаться от приведенного ниже) (рис. 1).

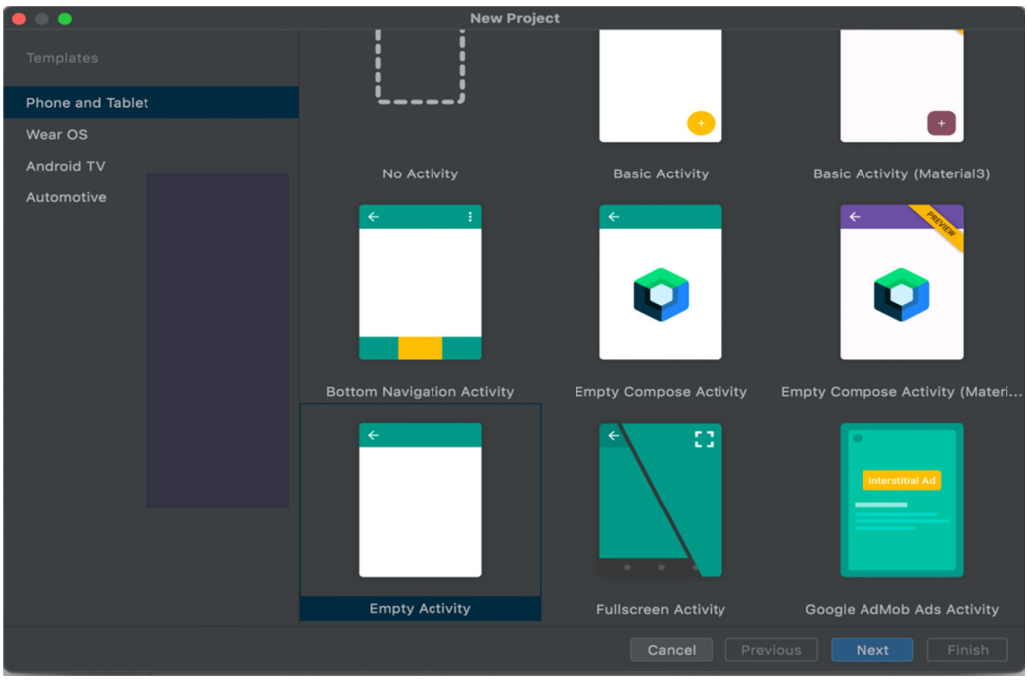

Рис. 1. Окно настройки проекта

Нажмите кнопку *Next*, в появившемся окне в поле *Name* введите название вашего проекта, например Hello World. В пункте *Package name* введите название пакета для вашего проекта, например ru.school.helloworld. В пункте *Save location* настройте путь к вашему проекту, убедитесь, что никаких лишних символов в пути нет, допустимы только латинские буквы и цифры. Пункты *Language* и *Minimum SDK* оставьте без изменений, нажмите *Finish*. Дождитесь, пока Андроид Студия настроит ваш проект, она скачает недостающие элементы для разработки, синхронизирует их, и вскоре можно будет приступить к непосредственной разработке на Андроид. После завершения настройки окно Андроид Студии будет выглядеть примерно так, как на рис. 2.

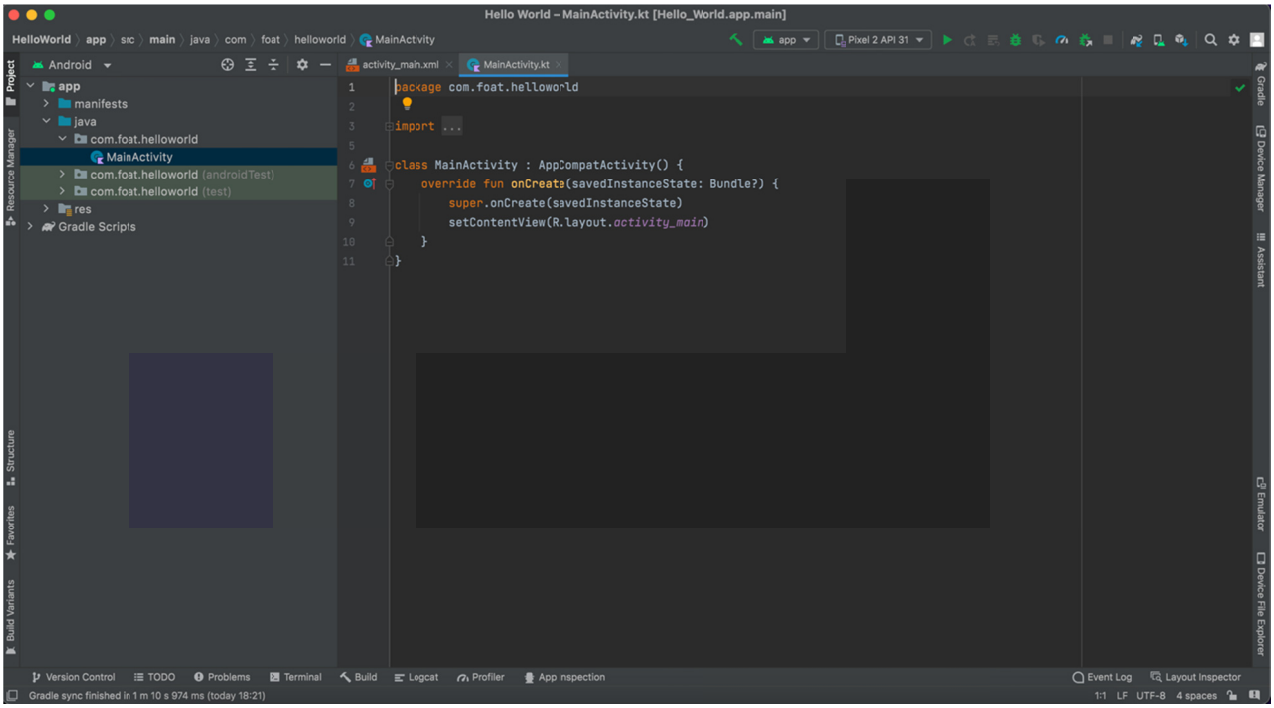

Рис. 2. Окно Андроид Студии с созданным проектом

Далее необходимо создать виртуальное устройство, если такого еще нет. Для этого перейдите в пункт меню Tools --> Device Manager. В открывшемся окне отобразится список имеющихся на данный момент виртуальных устройств (рис. 3).

Если ни одного устройства нет либо нужно создать еще одно, то нажмите кнопку *Create device*. В открывшемся окне необходимо выбрать модель устройства, которую нужно создать (на рис. 4 выбрана модель Pixel 5). Нажмите кнопку *Next*, в новом открывшемся окне выберите версию операционной системы Андроид, устройство, с которым вы хотите создать (рис. 5), рекомендуется выбирать первый или второй вариант сверху списка (на рис. 5 выбрана версия

API 32, которая соответствует версии Андроид 12.1). Нажимая кнопку *Next*, переходим в последнее окно настройки виртуального устройства. Здесь необходимо задать название устройства, которое будет отображаться в списке устройств, а также имеется возможность изменить некоторые другие свойства. Оставьте все настройки без изменений и нажмите кнопку *Finish*. После создания виртуального устройства можно наконец-то попробовать запустить наш проект и посмотреть, что получится. Для этого необходимо в меню с кнопками найти пункт, где указано виртуальное устройство (на рис. 7 это Pixel 2 API 31), и нажатием кнопки рядом (зеленый треугольник) запустить наше приложение на устройстве. Результат выполнения будет примерно такой же, как на рис. 8, отличия могут быть во внешнем виде устройства, по умолчанию оно запускается в виде прикрепленного окна внутри окна Device Manager (рис. 3).

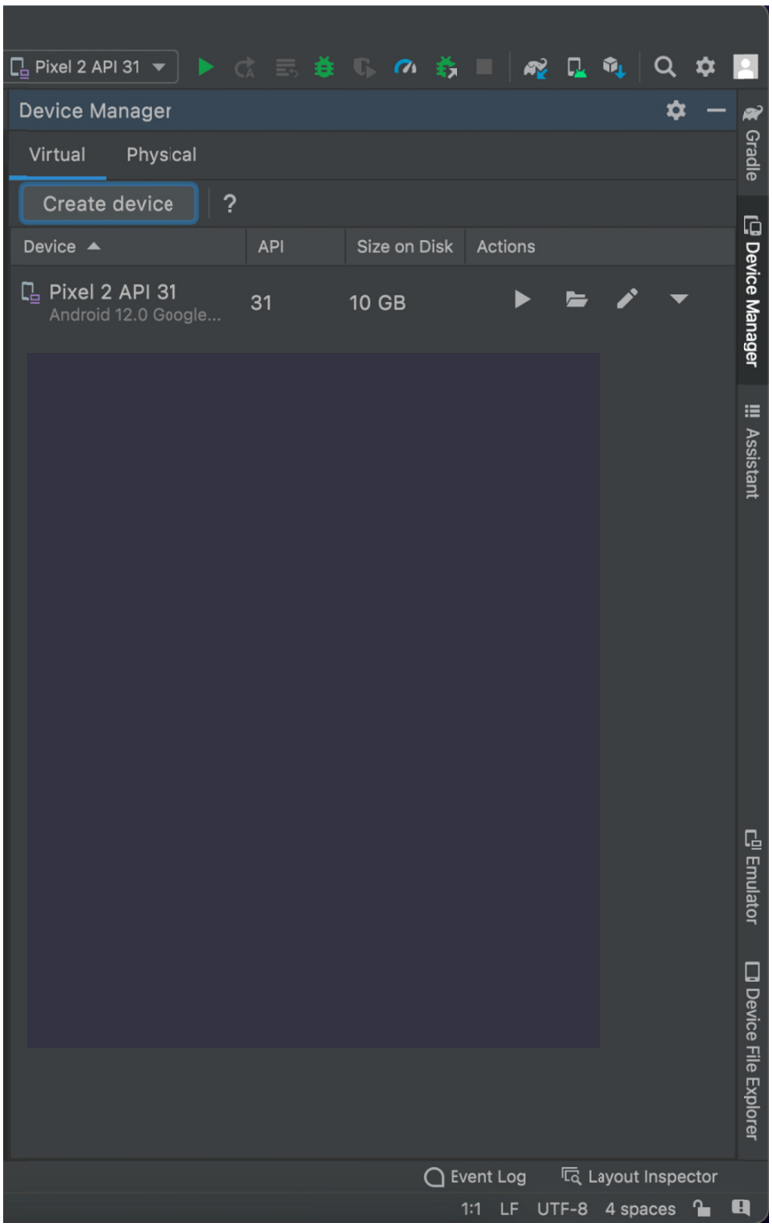

Рис. 3. Окно со списком виртуальных устройств

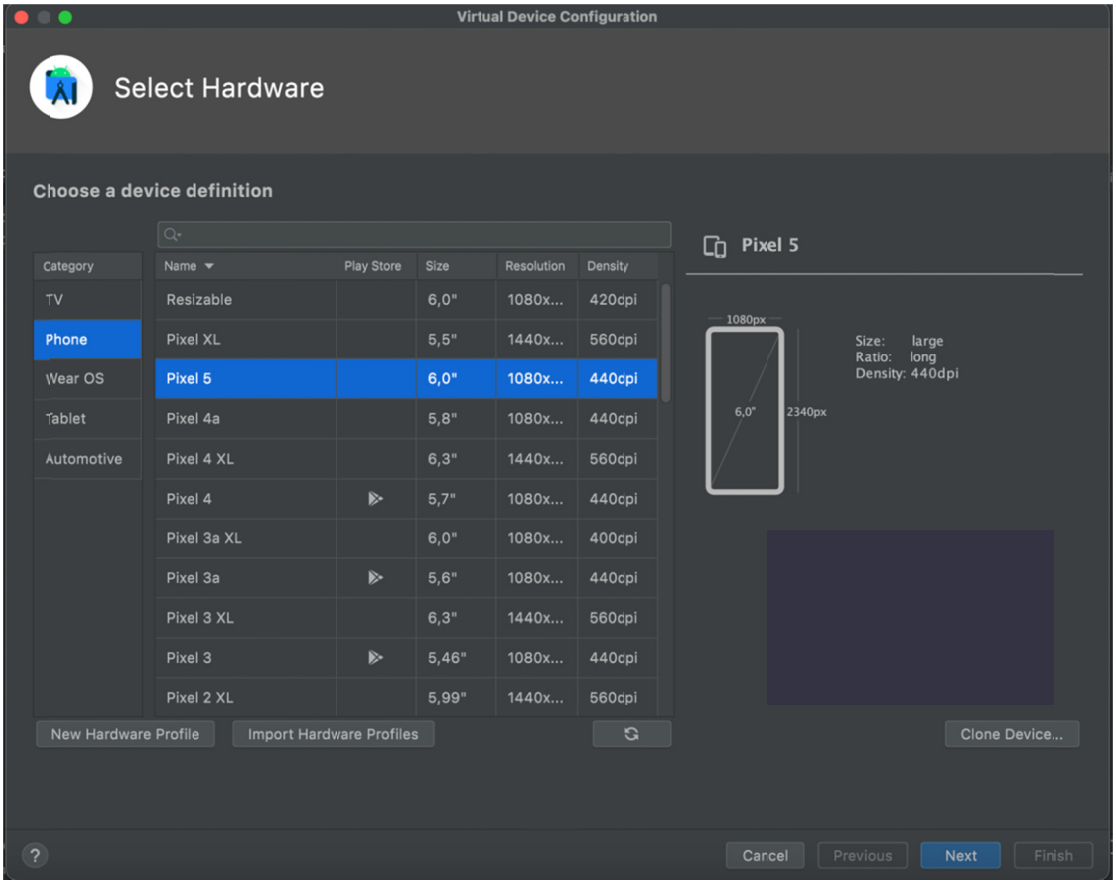

Рис. 4. Окно выбора модели устройства

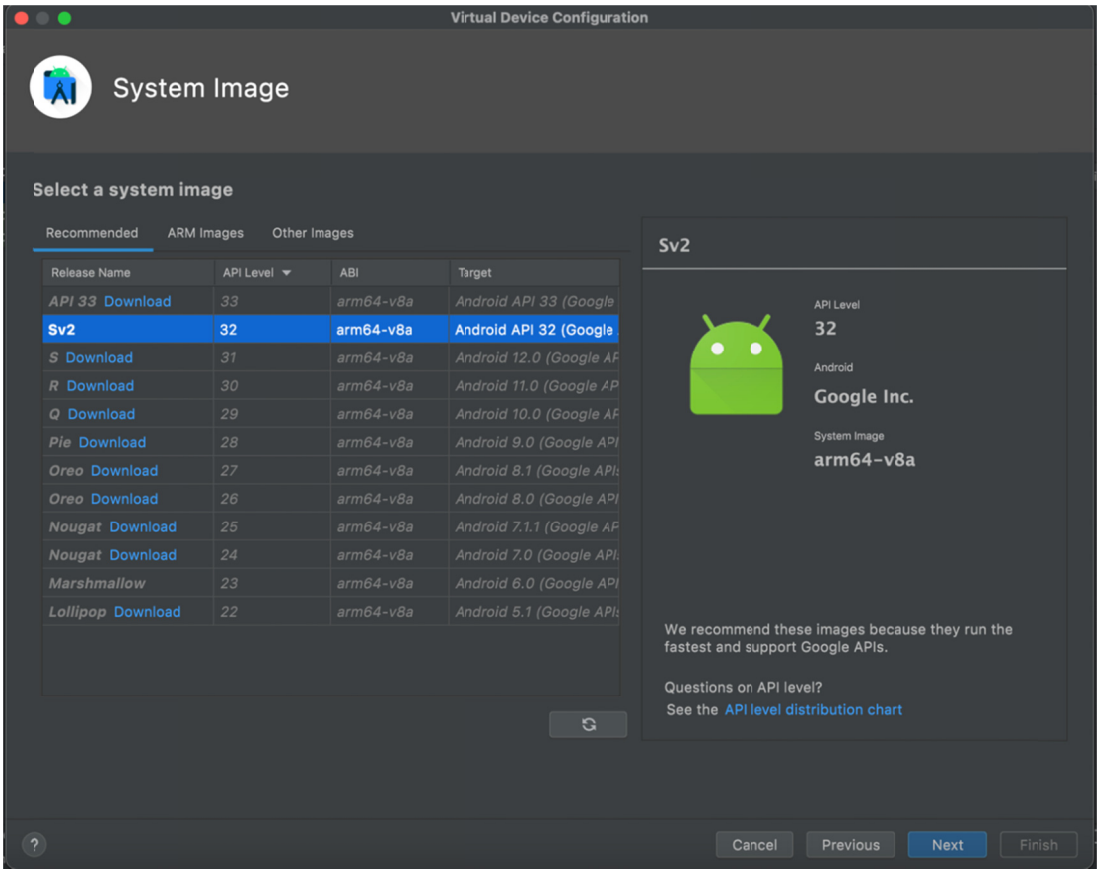

Рис. 5. Окно выбора версии Андроид

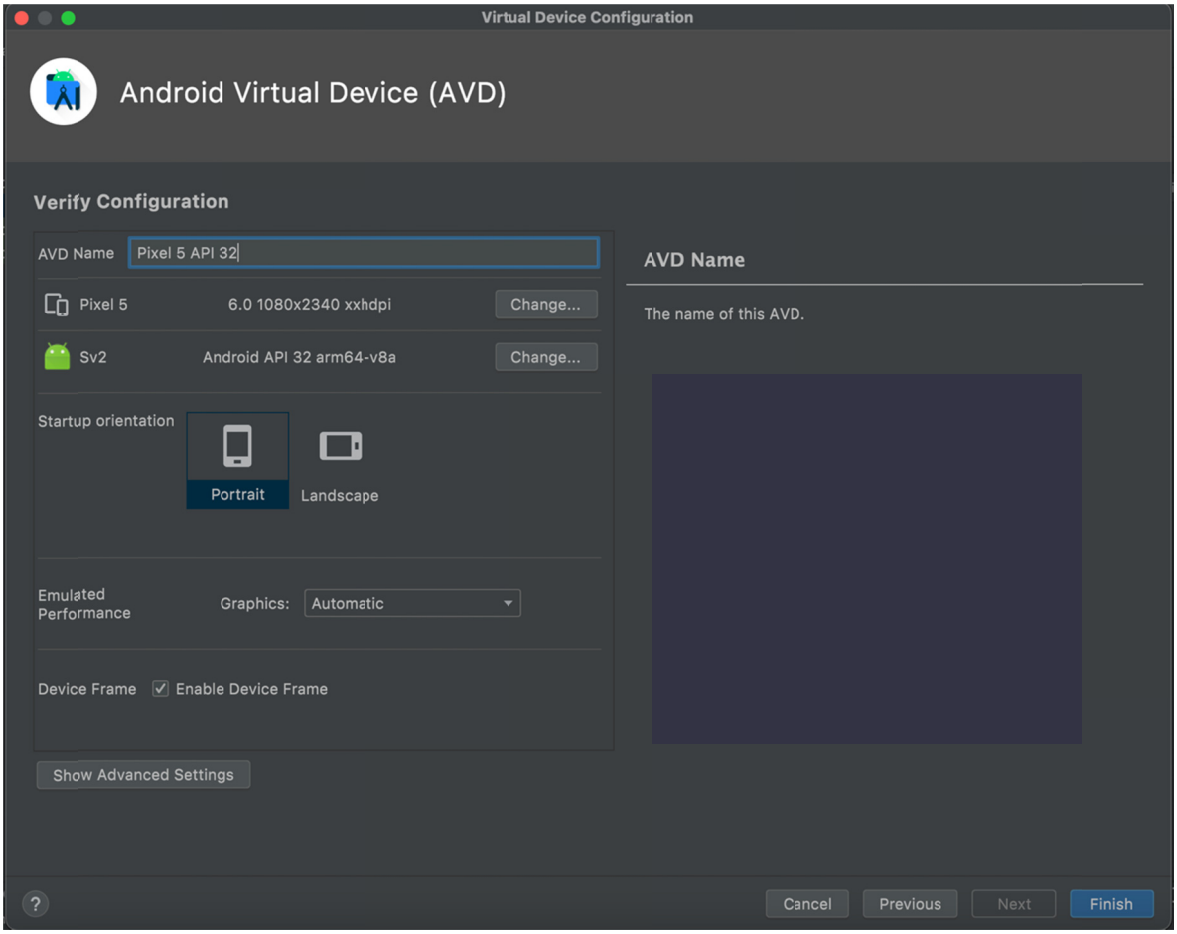

Рис. 6. Окно завершения создания устройства

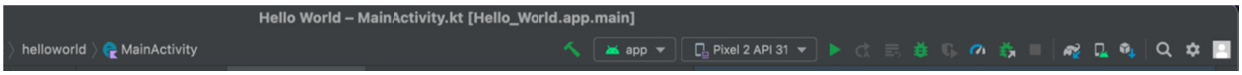

Рис. 7. Меню с кнопками

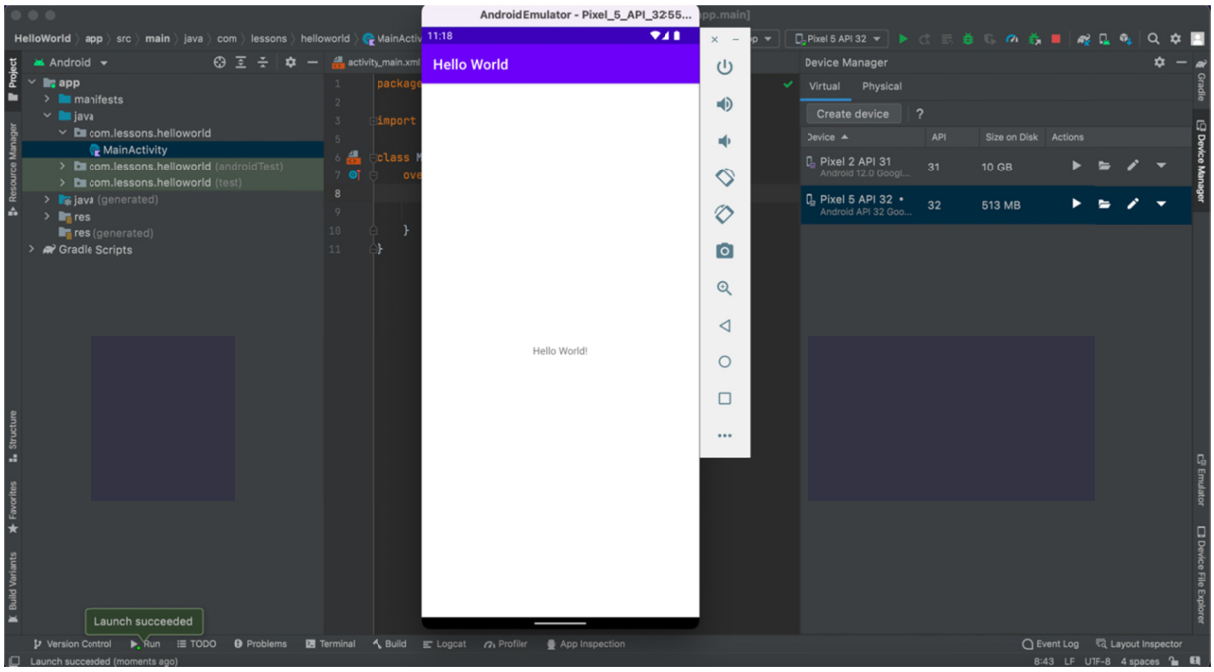

Рис. 8. Виртуальное устройство с запущенной программой

1.9. Структура Андроид проекта

Первая программа успешно запущена, результат виден на устройстве, теперь пришло время разобраться, что же добавлено в код и как это все работает.

В левом верхнем окне отображается структура всего нашего проекта – можно попробовать рассмотреть разные виды отображения:

– Android – режим по умолчанию, отображает основные «кирпичики» проекта;

– Project – режим отображения, как в файловой системе нашего компьютера;

– packages – режим отображения пакетов;

– другие.

В основном самым полезным будет режим отображения “Android”.

При открытии самого верхнего элемента в этом окне – app – мы увидим общую структуру проекта. Самый первый элемент в этом списке – папка **manifests**, которая содержит всего один элемент – **AndroidManifest.xml**. Этот файл будет содержать в себе описание всех элементов нашего приложения. На начальном этапе нам будет достаточно того, что уже содержится в файле, прежде всего – информация о главном окне нашего будущего приложения **MainActivity** (рис. 9).

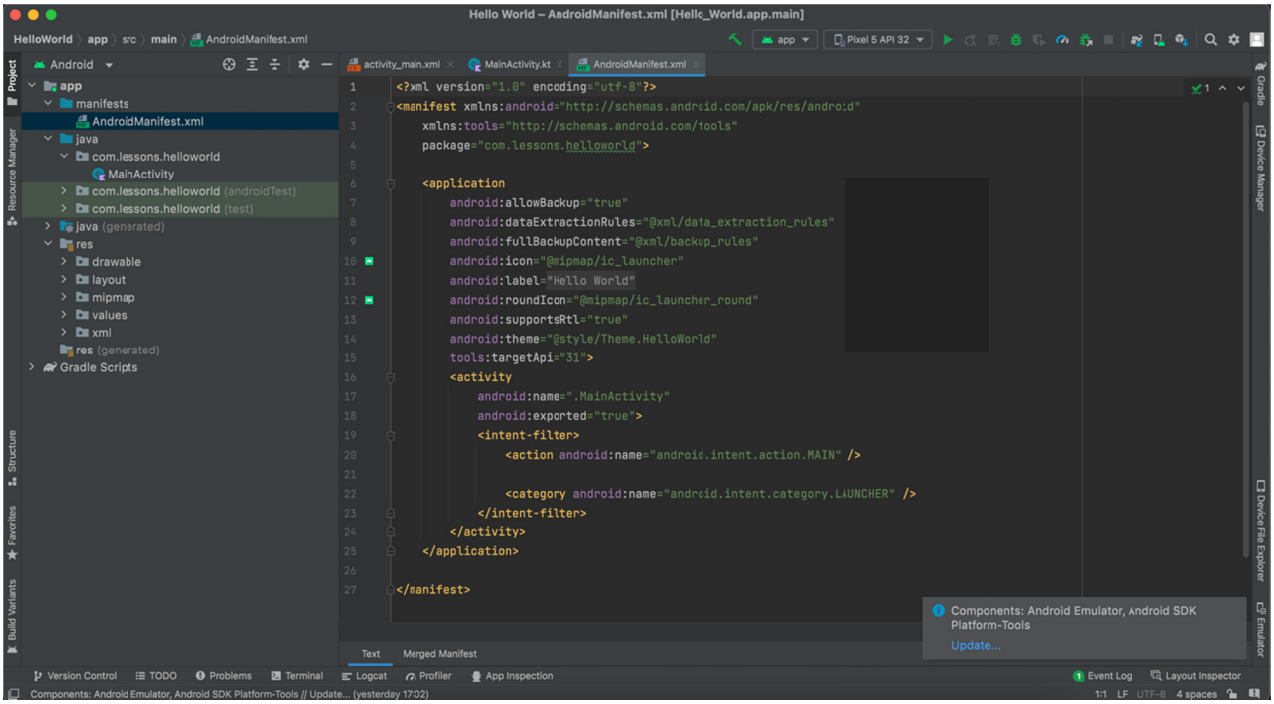

*Рис. 9. Открытый файл **AndroidManifest.xml***

Далее раскройте папку **java**, эта папка в любом проекте содержит основную часть кода приложения, в нашем случае написанного на Котлин. На данный момент в папке один файл – **MainActivity.kt**. Если он не отображается на главном окне, то его можно открыть, найдя в **пакете ru.school.helloworld** (здесь должно быть название пакета, который вы указали при создании проекта). Пакет в Котлин – это логическая общность, которая объединяет в себе определенный функционал (чаще всего это классы и функции), используемый для решения близкой по смыслу задачи. Традиционно пакет начинается с крупной общности типа «ru», «com» и т. д., далее добавляется название компании или организации разработчика (у нас это lessons), далее название проекта без пробелов и маленькими буквами (в нашем случае helloworld). В стандартной библиотеке Андроида часто пакеты начинаются со слов android/android. Две другие подпапки (подписанные соответственно **androidTest** и **test**) с таким же названием содержат тесты (это особый вид кода, который позволяет проверять наш код на корректность) – пока что созданные автоматически.

Далее раскройте папку **res**, эта папка в любом проекте содержит все ресурсы приложения. Так называются любые файлы с контентом – будь то изображения, звуки, текстовые данные, файлы интерфейсов нашего приложения или различные другие вспомогательные файлы. Например:

− в подпапке **drawable** будут содержаться все изображения;

− в подпапке **layout** – файлы интерфейсов приложения, сейчас в ней находится единственный файл **activity_main.xml** с описанием главного окна нашего приложения;

− **mipmap** содержит главную иконку приложения, по умолчанию это изображение зеленого робота;

− **values** содержит строковые ресурсы, цвета, темы со стилями приложения и т. д.

В текущем пособии мы не будем углубляться в сложные проекты с несколькими окнами, рассмотрим только случай приложения с одним главным окном, код которого будет в **MainActivity.kt**, а описание интерфейса – в **activity_main.xml.**

Далее откройте файл **activity_main.xml**.

Справа от панели Project отображаются панели, общее название которых **Редактор макета** (Layout Editor) (рис. 10). В правой части редактора сверху находится панель с палитрой, которая содержит элементы интерфейса, необходимые для приложения. В самом верхнем пункте палитры находятся самые популярные и нужные элементы – **TextView** (для отображения текста), **Button**

(обычная кнопка), **ImageView** (контейнер для вставки изображений), **RecyclerView** (для отображения списков), **ScrollView** (контейнер для отображения проматываемого контента – используется, если контент может не уместиться на экране устройства), **Switch** (элемент с возможностью выбора вариантов).

Снизу от палитры располагается панель со структурой макета – **Дерево макета** (Component tree). Здесь отображаются элементы, которые на данный момент добавлены в наш файл макета.

Далее справа находится основное окно редактора – собственно, панель редактора макета (Design editor). На этой панели в явном виде отображается дизайн нашего макета – либо в визуальном виде, либо в текстовом виде формата XML, либо как комбинация их обоих. Вид отображения можно сменить нажатием одной из кнопок в правом верхнем углу – Code, Split, Design. Попробуйте изменить вид отображения макета. В текстовом режиме отображения панели палитра и дерево макета пропадают. Обычно удобнее всего пользоваться режимом Split, чтобы описывать макет явно в текстовом виде и сразу видеть изменения в дизайне.

1.10. Макет приложения

Теперь посмотрим, какие элементы добавлены в нашем макете. Для начала перейдем в режим отображения макета Split.

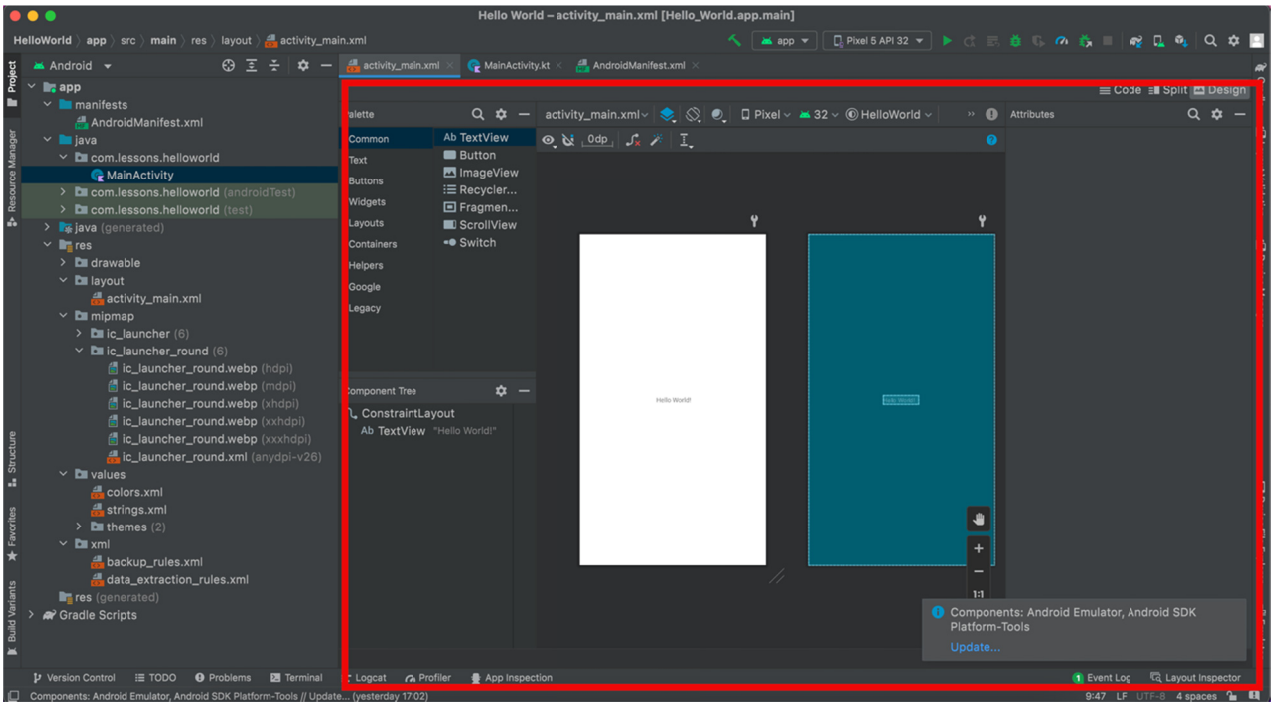

Рис. 10. Редактор макета

На самой верхней строчке файла указан так называемый корневой элемент (в терминологии формата XML) – `androidx.constraintlayout.widget.ConstraintLayout`, где `ConstraintLayout` – собственно, название элемента, а `androidx.constraintlayout.widget` – название пакета, где находится этот элемент. Вообще все элементы в XML выделяются знаками «<» и «>», либо конец элемента может обозначаться, как и начало, но с добавлением знака «/». А Андроид Студия любезно подсвечивает нам границы каждого элемента. Например, для того, чтобы увидеть границы корневого элемента, нажмите в любом месте на название элемента (рис. 11). Кроме `ConstraintLayout` файл содержит еще один элемент – `TextView`, который используется для отображения текста “Hello World”.

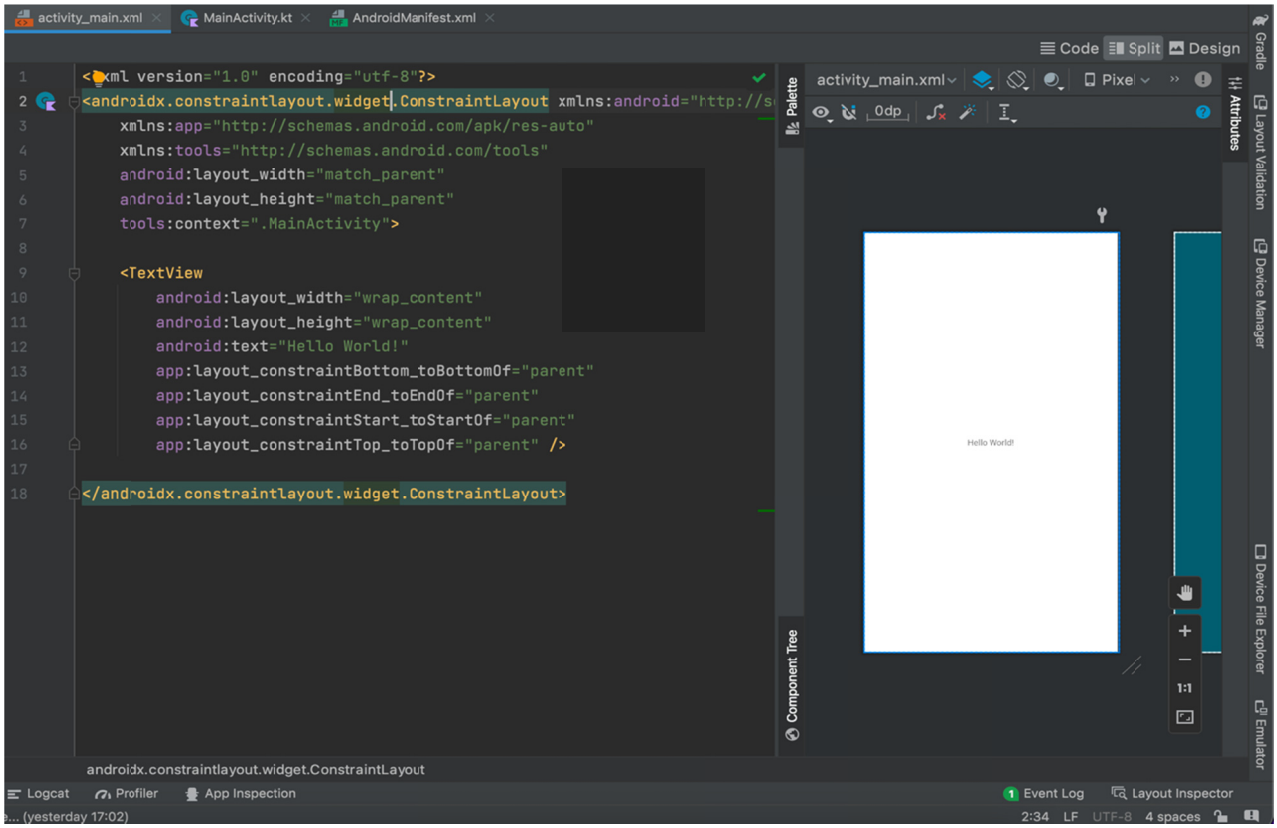

Рис. 11. Подсветка корневого элемента

Давайте рассмотрим подробнее атрибуты `TextView`. Основной атрибут этого элемента – `android:text="Hello World!"`, здесь указывается текст для отображения. Попробуйте изменить текст и запустить приложение.

Также важными и необходимыми атрибутами являются:

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
```

Как можно понять из названия, эти атрибуты используются для задания размеров нашего элемента (длинны и высоты соответственно), это стандартные атрибуты для большинства элементов – и для текстовых полей, и для кнопок, и для контейнеров (например, у элемента `ConstraintLayout` они тоже указаны). Значение `wrap_content` означает, что размер элемента будет подстроен под контент, который в нем содержится, т. е. в случае текстового поля размер элемента будет определен размером текста. Другим возможным значением этого атрибута является `match_parent`, который означает, что элемент займет все свободное пространство по высоте или длине.

Оставшиеся элементы используются для расположения элемента относительно других элементов, в нашем случае – внутри контейнера `ConstraintLayout`:

```
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
```

Как можно понять из названия, первый из них определяет положение нижней части элемента относительно нижней части элемента из правой части выражения (в нашем случае это элемент верхнего уровня, так называемый родительский элемент – `parent`). Следующий описывает положение правой части нашего элемента относительно правой части родительского элемента, далее – положение левой части относительно левой родительского элемента, и, наконец, последний – верхнюю часть нашего элемента относительно верхней родительского. Важное замечание: части ключевого слова `End` и `Start` ориентируются на язык приложения, в котором задаются все тексты, например, для русского языка они будут соответствовать правой и левой частям элемента, а для языков с направлением текста «справа налево» (арабский) они будут соответствовать левой и правой частям.

В правом верхнем углу панели редактора можно найти панель атрибутов выделенного в редакторе элемента (вызывается нажатием кнопки **Attributes**) (рис. 12), прокрутите панель до самого низа и найдите выпадающий список **All attributes** (рис. 13).

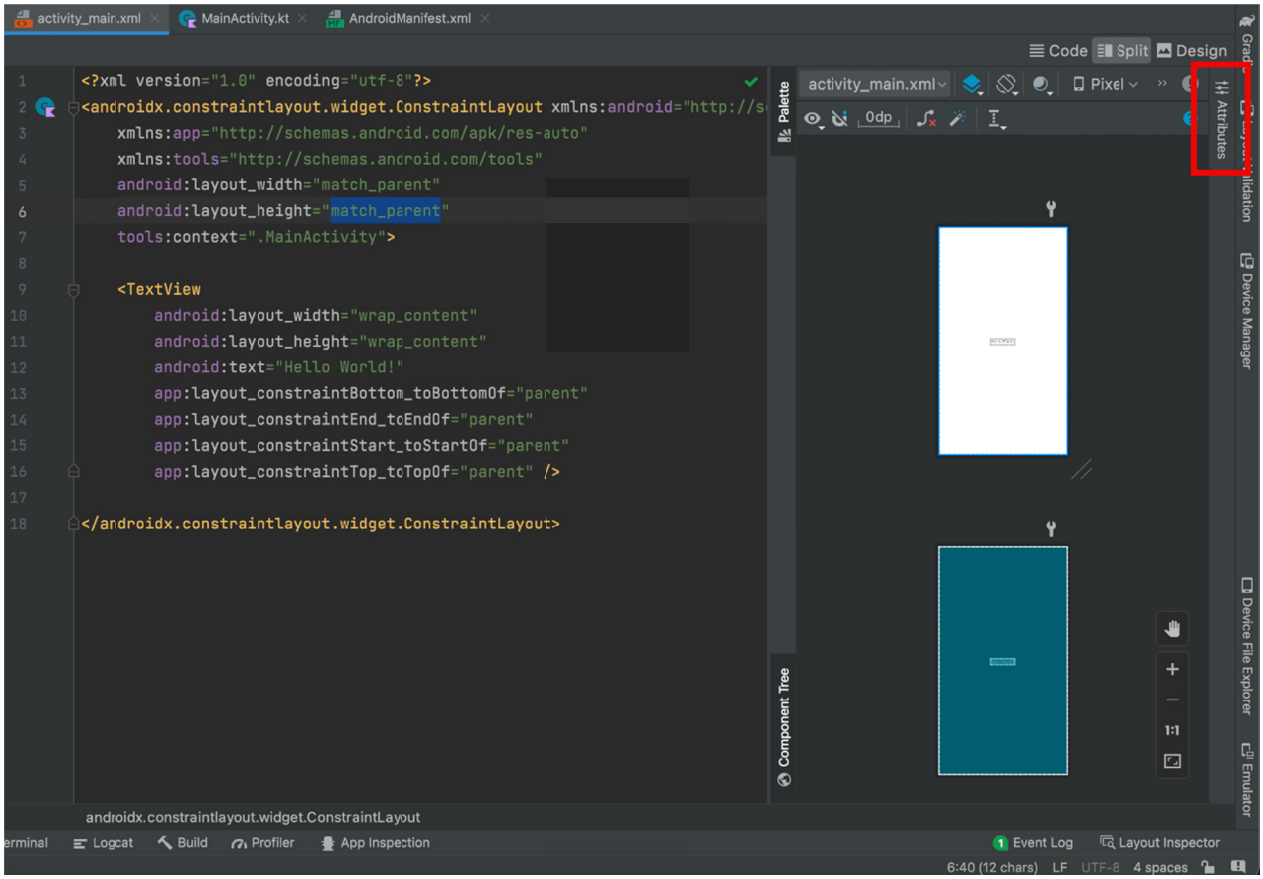

Рис. 12. Кнопка для вызова панели атрибутов

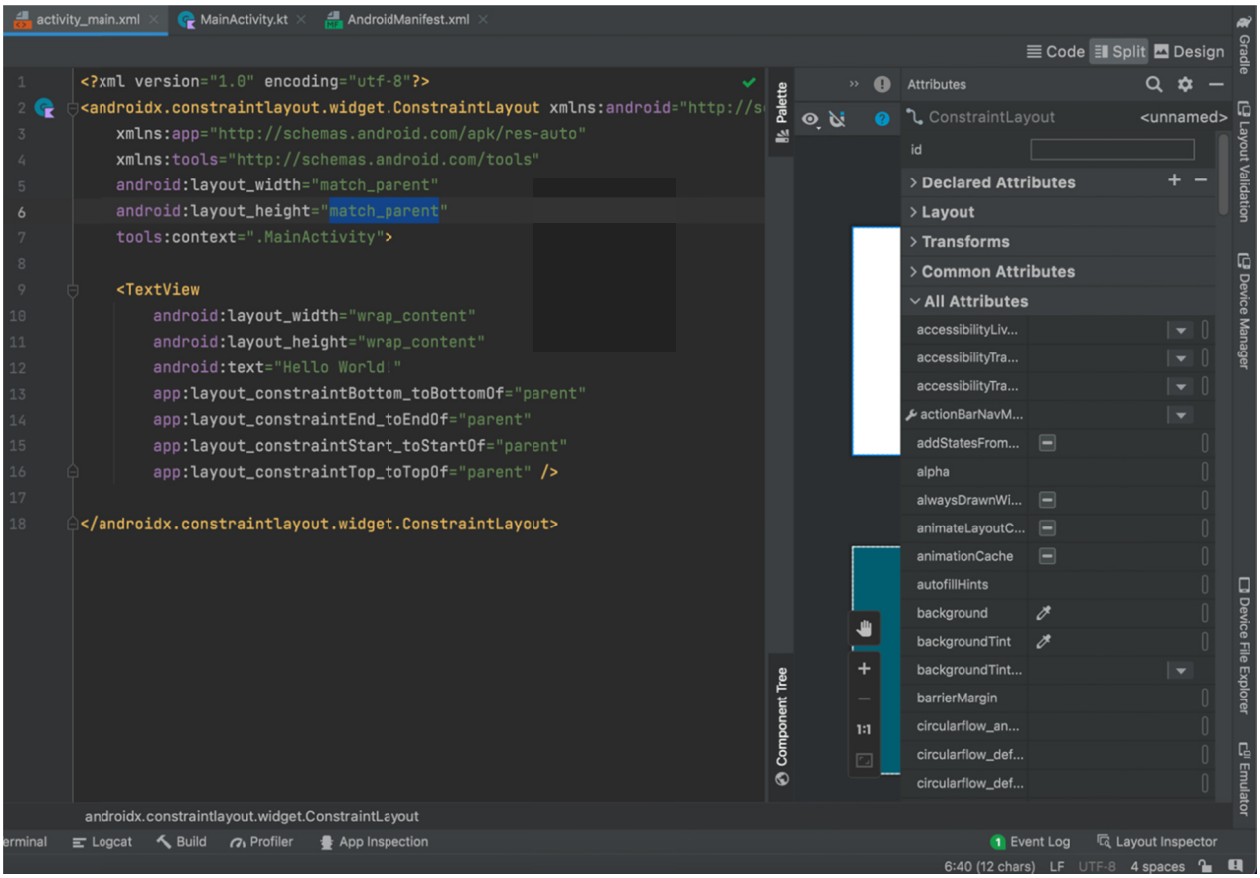

Рис. 13. Панель с атрибутами выделенного в редакторе элемента

Через эту панель можно менять свойства элементов, не внося изменений в формате XML. У разных элементов атрибуты могут отличаться, но некоторые являются общими для всех. Например, `background` отвечает за фоновую заливку элемента, он есть и у контейнера `ConstraintLayout`, и у `TextView`. А вот атрибут `textColor` есть только у элементов, работающих с текстом. Самостоятельно исследуйте панель атрибутов и попробуйте изменить цвет, стиль, размер текста у `TextView`, фоновую заливку у `TextView` и `ConstraintLayout` (Андроид Студия предоставляет инструмент для выбора цвета из палитры), попробуйте задать длинный текст в `TextView`. Как можно заметить, при изменении атрибутов, которых нет в текстовом виде, они сразу же добавляются и в редакторе, это происходит автоматически.

При изменении текста в `TextView` можно было заметить, что Андроид Студия подсвечивает это текстовое поле, при наведении на него курсора появится подсказка: `Hardcoded string "тут_ваш_текст", should use @string resource`. Это означает, что текстовые значения лучше писать не прямо в редакторе макета, а в специальном файле **strings.xml**, который содержится в папке **res**, подпапке **values**. Для поддержки нескольких языков в приложении нужно использовать разные файлы **strings.xml**, каждый из которых должен находиться в своей подпапке **values** с суффиксом языка (например, суффикс **-ru** для русского языка, **-es** – для испанского и т. д.). Папка без суффиксов используется по умолчанию, там располагают файл, из которого берутся все строковые ресурсы, если нет отдельных языковых папок. Поскольку в нашем приложении не будет поддержки разных языков, мы будем использовать эту папку по умолчанию для строк на русском языке. Откройте файл **strings.xml**, пока в этом файле только одна строка `<string name="app_name">Hello World</string>`, она используется для названия приложения (`app_name` – это имя строки, по этому имени к ней можно будет обратиться с любой другой точки проекта, а текст между квадратными скобками – значение этой строки). Создайте еще один элемент с названием radius_label и значением «Введите значение радиуса:». Вернитесь в файл макета и измените значение текста у `TextView` следующим образом:

```
"@string/radius_label"
```

Ключевое слово `@string` означает, что значение нужно будет взять из строковых ресурсов. Далее после знака «слеш» указывается имя строки. Запустите приложение и посмотрите, изменился ли текст.

Добавим еще несколько элементов на макет. Для этого перейдем в режим Design, в левом верхнем углу в панели палитры найдем элемент **Plain Text** (он находится в группе Text) и перетащим его курсором на макет, оставив пониже предыдущего TextView. В группе Buttons найдем элемент **Button** и перетащим его на макет пониже предыдущего элемента Plain Text (рис. 14).

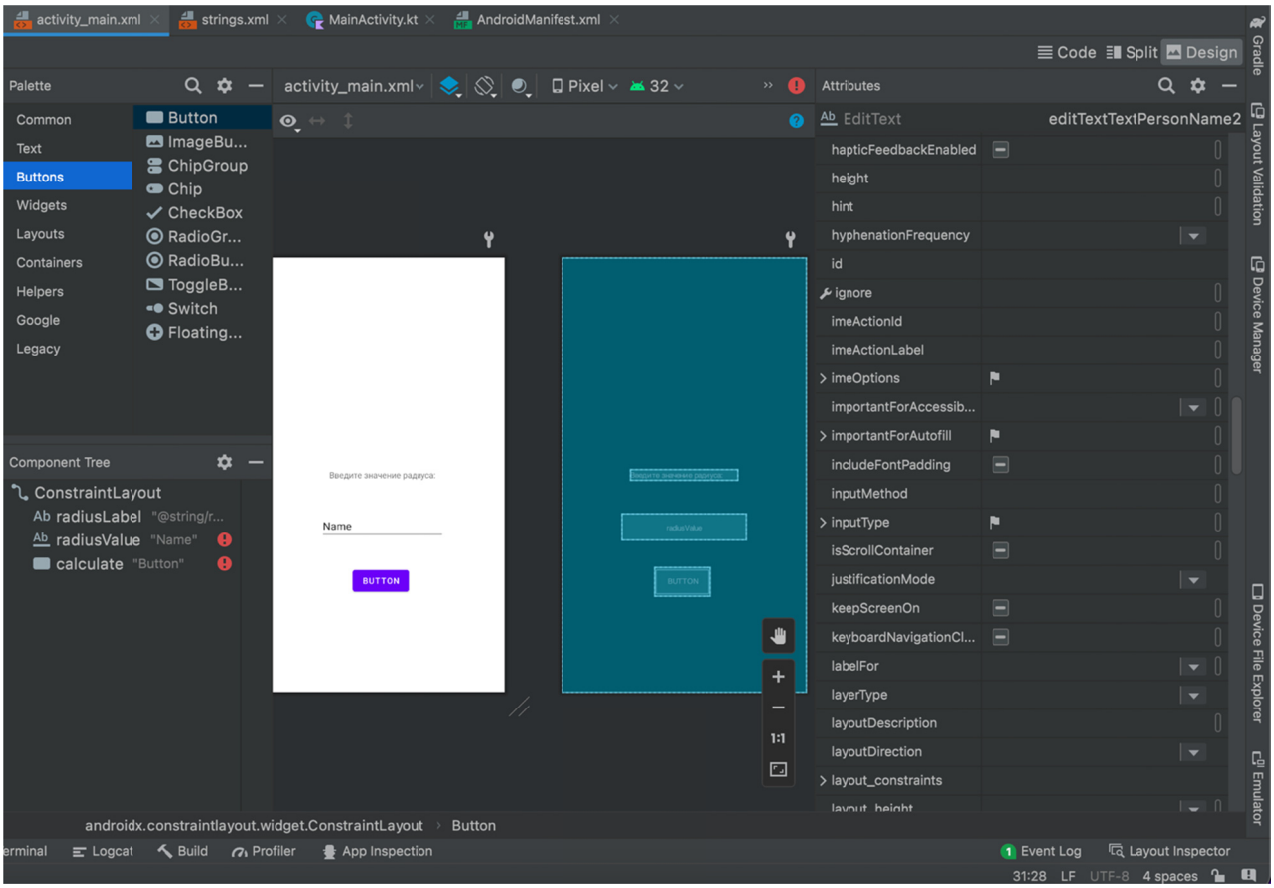

Рис. 14. Окно редактора макета с новыми добавленными элементами

Перейдем в режим Split и откроем панель Attributes. Андроид Студия подсвечивает наши новые добавленные элементы, подсказывая, что им не выставлено положение относительно соседних элементов. Для того чтобы расположить их в нужном нам порядке, зададим каждому элементу `id` – название элемента, по которому мы будем к нему обращаться (по аналогии с именами строковых ресурсов). По очереди выделяя каждый элемент в макете, найдите для каждого атрибут `id` в списке атрибутов и задайте нужные значения, например, для TextView – `radiusLabel`, EditText – `radiusValue`, Button – `calculate` (рис. 15). Далее каждый элемент будет обозначаться его именем.

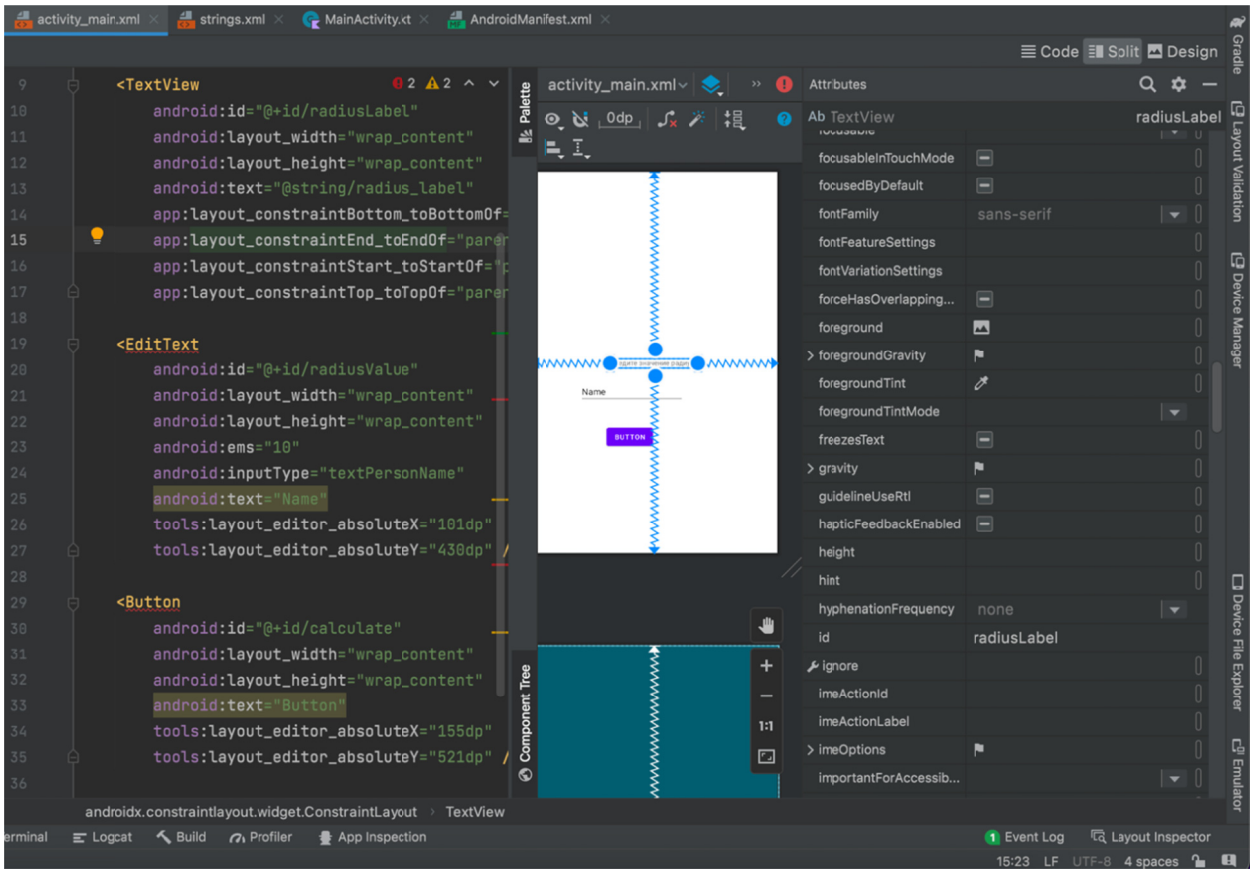

Рис. 15. Окно редактора макета с заданными id

Теперь можно определить для каждого элемента его положение.

Выделим элемент `radiusLabel` и найдем среди его атрибутов выпадающий список **layout_constraints**, раскроем его двойным нажатием и найдем в нем атрибут `layout_constraintBottom_toBottomOf` – изменим его значение на `@id/radiusValue`, это можно сделать, выбрав в выпадающем списке значений нужное имя.

Выделим элемент `radiusValue`, найдем среди его атрибутов `layout_constraintEnd_toEndOf` и `layout_constraintStart_toStartOf` – зададим им значение `parent`, т. е. левый и правый концы элемента будут привязаны к левому и правому краю родительского элемента. Атрибуту `layout_constraintTop_toBottomOf` установим значение `@id/radiusLabel`, это расположит наш элемент сверху по центру родительского элемента, чуть ниже элемента `radiusLabel`. Далее нам нужно, чтобы пользователь мог вводить в это поле только числа, для этого изменим атрибут `inputType`: `android:inputType="numberDecimal"`.

Далее аналогичным образом установите расположение для кнопки `calculate` – сверху по центру и чуть ниже элемента `radiusValue`, а также

удалите текст у элемента `radiusValue`, а текст у кнопки задайте «Вычислить площадь круга».

Самостоятельно добавьте так же еще два TextView:

1. С текстом «Площадь круга =», расположите его между кнопкой `calculate` и полем ввода `radiusValue`, размеры задайте `wrap_content` (не забудьте, что текстовое значение нужно задать в strings.xml).

2. С текстом «Калькулятор площади круга», расположите его на самый верх окна, выше элемента `radiusLabel`. Задайте элементу размер текста `textSize = «24sp», textStyle = «bold»`.

После всех действий у вас должно получиться так же, как на рис. 16. Запустите приложение на эмуляторе и проверьте, что получилось, убедитесь, что в поле ввода можно вводить только числа (целые и дробные).

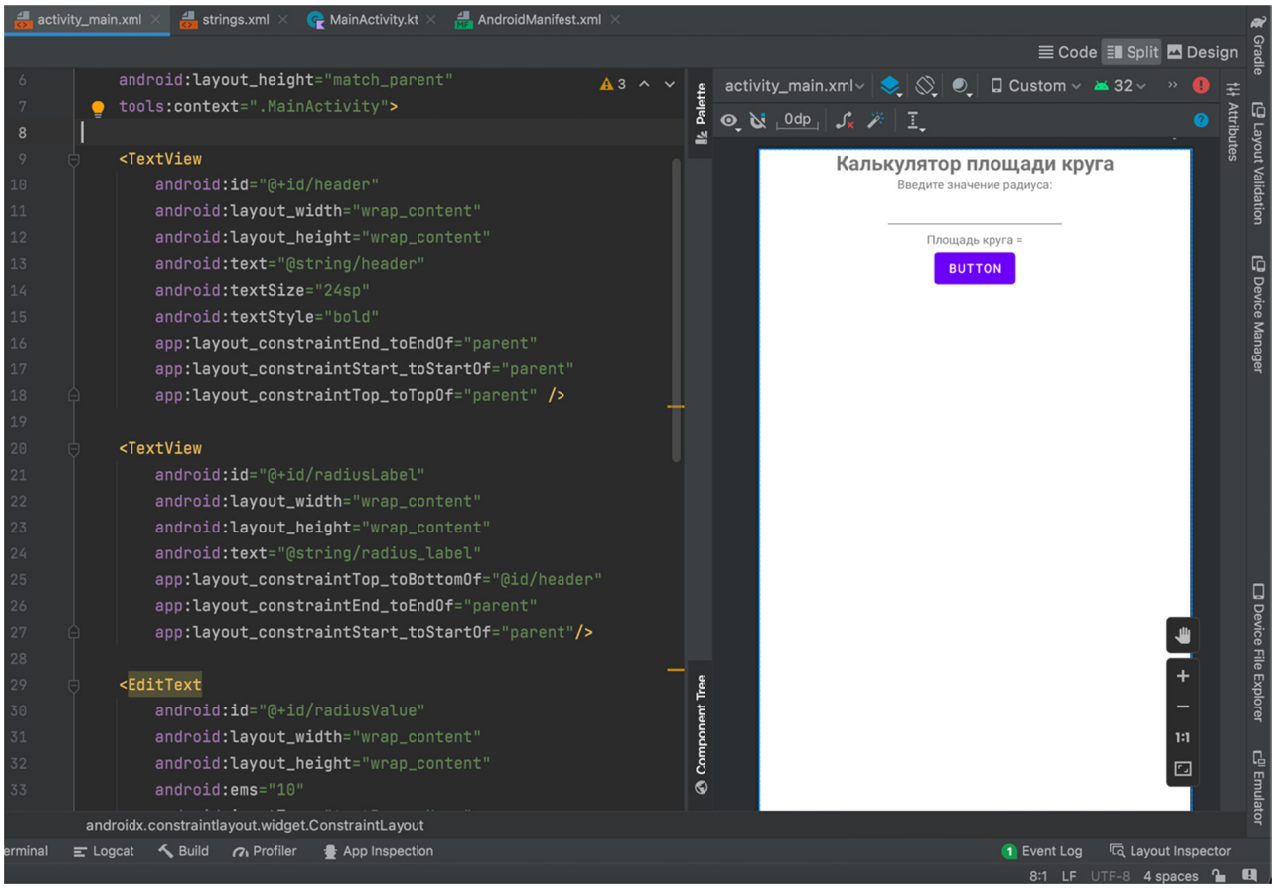

Рис. 16. Готовый макет приложения

1.11. Взаимодействие с интерфейсом

Макет нашего приложения готов, но приложение еще ничего не делает. Для того чтобы приложение реагировало на действия пользователя, нужно добавить обработчики действий – прежде всего, нажатие кнопки `calculate`. Перейдем в файл MainActivity.kt. Немного о классе, который находится в этом файле – **активность MainActivity**. Активность – основной класс структуры Андроид-приложения. Этот класс отвечает за отображение и взаимодействие с главным экраном приложения, его же имя указано в манифесте как точка запуска. В этом классе реализована одна функция `onCreate`, которая вызывается автоматически при запуске приложения и инициализации активности. В этом же методе активности задается макет окна, с помощью метода `setContentView(R.layout.activity_main)`. Все изменения в нашем тестовом проекте мы будем делать внутри этого метода, но вообще хорошим тоном считается создавать отдельные функции с отдельным функционалом.

Далее определим, как в этом файле получить доступ к элементам макета, которые мы создали в `activity_main`. Андроид предоставляет стандартную функцию получения любого элемента из макета путем обращения к ее имени `findViewById`:

```
val button = findViewById<Button>(R.id.calculate)
```

В квадратных скобках функции указывается тип элемента, который нужно достать из макета (в примере выше это `Button`), а в круглых скобках – имя элемента (`R.id.calculate`). Это имя образуется путем добавления приставки `R.id` к идентификатору `calculate`, который был задан в макете (рис. 15). Приставка `R.id` используется для всех идентификаторов макетов. После добавления этого кода Андроид Студия будет подчеркивать строку красным, подсказывая, что здесь есть какая-то ошибка – это ошибка отсутствия пакета с нужным элементом. При наведении курсора на строку выйдет подсказка, что можно импортировать нужный пакет, нажмите Import, после чего добавится строка `import android.widget.Button` наверху файла. Аналогичным образом добавьте переменные для доступа к `radiusValue` и `circleArea`.

Далее создадим обработчик нажатия на кнопку. Для этого вызовем функцию `setOnClickListener` у кнопки:

```
button.setOnClickListener { }
```

Этот метод добавляет обработчик нажатия для кнопки, внутри которой будут описаны действия, которые произведутся при нажатии на кнопку. Для начала получим данные из строки ввода:

```
val radius = input.text.toString().toDoubleOrNull()
```

`input` – название переменной для поля ввода (у вас может отличаться), `text` – стандартная переменная у этого объекта, которую далее нужно преобразовать в `String`, который, в свою очередь, уже преобразуется в `Double`. Если поле ввода содержит данные, которые невозможно преобразовать в `Double`, то в переменную `radius` запишется значение `null`. Проверяем значение `radius` на `null`, и если оно не `null`, то вычисляем размер площади круга по известной формуле и записываем полученное значение в текстовое поле `result`:

```
if (radius != null) {
val res = pi * radius * radius
val label = getString(R.string.circle_area)
result.text = "$label$res"
}
```

Чтобы получить строковый ресурс, воспользовались методом `getString`, который принимает в себя имя строки из strings.xml. Кроме того, для присвоения текстовому полю значения использовали **строку с аргументами –** в Котлин можно прямо в строковое значение вписывать значения других переменных, поставив предварительно перед ними знак доллара «$».

Общий код активности должен получиться примерно таким, как на рис. 17.

Запустите приложение и попробуйте вычислить площадь круга для разных значений радиуса, убедитесь, что все работает, площадь вычисляется верно.

Контрольные вопросы и задачи

1. Какие основные преимущества языка Котлин?

2. Чем отличаются объекты, объявленные с использованием ключевых слов val и var?

3. Назовите основные операции над типом Boolean.

4. В чем отличие анонимных функций от обычных?

5. Что подразумевается, когда говорят, что переменная nullable?

6. Дайте определение оператора range.

7. Что такое конструктор класса?

8. Назовите три основных вида коллекцией в Котлин.

9. Установите все инструменты разработчика – Андроид Студию и эмулятор, пользуясь разделом 1.8.

10. Создайте приложение «Калькулятор площади круга», используя в качестве руководства разделы 1.10 и 1.11.

11. В качестве самостоятельного задания реализуйте новое приложение – простой калькулятор для арифметических операций:

- *10 цифр, от 0 до 9, все можно реализовать через Button;*
- *4 арифметические операции – «+, -, *, /», реализовать через Button;*
- *2 поля ввода – для первого аргумента и для второго, EditText;*
- *одно текстовое поле TextView, для вывода результата;*
- *кнопка очистки полей ввода.*

activity_main.xml | strings.xml | MainActivity.kt | AndroidManifest.xml

```
1  package com.lessons.helloworld
2
3  import androidx.appcompat.app.AppCompatActivity
4  import android.os.Bundle
5  import android.widget.Button
6  import android.widget.EditText
7  import android.widget.TextView
8
9  class MainActivity : AppCompatActivity() {
10
11     private val pi = 3.1415
12
13     override fun onCreate(savedInstanceState: Bundle?) {
14         super.onCreate(savedInstanceState)
15         setContentView(R.layout.activity_main)
16
17         val input = findViewById<EditText>(R.id.radiusValue)
18         val result = findViewById<TextView>(R.id.circleArea)
19         val button = findViewById<Button>(R.id.calculate)
20         button.setOnClickListener { it: View!
21             val radius = input.text.toString().toDoubleOrNull()
22             if (radius != null) {
23                 val res = pi * radius * radius
24                 val label = getString(R.string.circle_area)
25                 result.text = "$label$res"
26             }
27         }
28     }
29 }
```

erminal | Logcat | Build | Profiler | App Inspection

*Рис. 17. Полный код активности **MainActivity***

Глава 2. Мобильная разработка на iOS

2.1. Введение в iOS

iOS – это мобильная операционная система (ОС), созданная и поддерживаемая компанией Apple Inc. для своих устройств – iPhone, iPod Touch и iPad (несмотря на то что, начиная с 2019 года, на iPad устанавливается своя операционная система iPadOS, разработка приложений ведется для обоих ОС одновременно, более того, с появлением ноутбуков Apple на базе процессоров Arm одни и те же приложения с iOS теперь можно запускать и на этих ноутбуках). Это вторая по распространенности мобильная ОС в мире после Андроид. Главным преимуществом ее является то, что одна и та же компания Apple разрабатывает и саму ОС, и устройства на базе этой ОС, что способствует бесперебойной работе этих устройств, более тесной интеграции программных решений и физического устройства, небольшой набор моделей (это является неплохим преимуществом при разработке приложений, поскольку программист может протестировать свое приложение буквально на всех устройствах модельного ряда, для Андроид это сделать затруднительно). Несмотря на плюсы, у iOS, а вернее у разработки для iOS, есть один существенный минус: средства разработки доступны только на MacOS, операционной системе от Apple, которая поставляется только на компьютерах и ноутбуках самой компании разработчика, что сильно повышает порог вхождения в профессию, ведь устройства от Apple относятся не к самой дешевой ценовой категории. С момента создания iOS основную долю разработки для нее вели на языке программирования Objective-C – языке семейства C, очень мощном, но довольно сложном для разработки, который ведет свою историю с 1980-х годов. К 2010 году iOS уже стала очень большой системой, приложения для нее писали десятки и сотни тысяч разработчиков, и у многих из них попадались одни и те же характерные ошибки. Проанализировав эту проблему, в Apple пришли к выводу, что Objective-C является слишком сложным для рядовых программистов, позволяет легко допускать ошибки. В Apple учли эти ошибки и начали разработку нового языка программирования, который был бы проще для изучения, но в то же время не терял бы мощных возможностей Objective-C. Так к 2014 году компанией был представлен новый инструмент для разработки для iOS – язык программирования Swift (Swift), гораздо более надежный и дружелюбный для новичков. На нем сложнее допускать ошибки, он краткий, позволяет писать меньше кода, и – что немаловажно – читать его гораздо легче и приятнее. И в то же время он остался

таким же мощным и быстрым. Кроме, собственно, iOS, Swift позволяет разрабатывать приложения и для других систем Apple – MacOS (ОС для персональных компьютеров и ноутбуков), tvOS (ОС для телевизоров), watchOS (ОС для часов), iPadOS.

2.2. Введение в разработку на Swift для iOS

Swift является языком строгой статической типизации, что позволяет избежать многих ошибок, связанных с неверным использованием разных типов данных. Swift полностью поддерживает объектно-ориентированное программирование (ООП), а также имеет в своем арсенале немало инструментов из функционального программирования. Одной из основных идей языка является понятие «**протокол» (protocol)**, которое призвано помочь разработчику в создании сложных программ и в то же время которое позволило бы избежать многих характерных для ООП проблем. Поскольку Swift создавался очень простым и кратким, его можно считать достаточно хорошим языком для старта как программиста. Для новичков есть возможность скачивать Swift Playgrounds – приложение для iPad для самостоятельного изучения языка. Также Swift является Open Source, т. е. любой желающий может скачать и посмотреть исходный код самого языка и даже поучаствовать в его разработке. Кроме того, есть множество онлайн-тренажеров для проверки работоспособности своего кода, например: https://www.onlinegdb.com/online_swift_compiler. Можно отметить несколько важных особенностей языка перед тем, как перейти к более подробному изучению:

− переменные всегда инициализируются до использования;

− индексы массивов всегда проверяются на попадание в диапазон элементов массива;

− целые числа проверяются на переполнение;

− специальная конструкция языка (так называемый **Optional**) дает возможность избежать ошибок с пустым значением (nil);

− память контролируются автоматически.

Swift динамично развивается, и разработчики постоянно добавляют в язык новые возможности.

2.3. Базовые понятия языка Swift: объявление переменной, типы данных, преобразования типов, понятие оператора, понятие функции

Для простых значений в Swift используется два ключевых слова: **let** и **var**. Первый используется для создания констант, а второй – для изменяемой переменной.

```
var weight = 42.5
let pi = 3.1415
```

И константа, и переменная должны иметь такой же тип, как у значения, которое им присваивается. При объявлении переменной тип указывать не всегда обязательно в случае, если при объявлении происходит и инициализация (в примерах выше компилятор автоматически определит тип переменной и константы, исходя из присвоенного значения). Если же изначально значение не присваивается (или нельзя однозначно определить тип, исходя только из значения, например, если переменная нужна с плавающей точкой, а инициализирована как целое число), то нужно явно указывать тип данных, разделенный двоеточием:

```
var exampleDouble = 70
```

В примере выше мы попытались создать переменную типа Double, но присвоили целое число, поэтому компилятор автоматически определит типом переменной Int, нужно указать явно, какой тип мы хотели:

```
var exampleDouble: Double = 70
```

Swift является языком со строгой статической типизацией, а значит, нельзя переменной с одним типом присваивать значения другого типа, нужно явным образом их конвертировать:

```
let simpleDouble: Double = 41.0
let simpleInt = 12
let sum = simpleInt + simpleDouble
```

Если попытаться запустить код, представленный выше, то компилятор выдаст ошибку при попытке суммировать два значения с разными типами. Явное приведение типа выглядит следующим образом:

```
let a = 42
let simpleDouble = Double(a)
```

В качестве самостоятельного задания попробуйте исправить ошибку из предыдущего примера путем явного приведения одной из констант к типу другой константы в строке суммирования.

Все типы данных в Swift делятся на две большие группы – именованные типы и составные (compound). Именованные типы – это типы с конкретным названием, например Int, Double, String и т. д., большинство типов являются именованными. Все именованные типы можно **расширять**, т. е. добавлять новое поведение через **Extensions** (их мы рассмотрим позже). Составные типы не имеют какого-то обозначения и выглядят следующим образом:

```
(какой-то тип, еще какой-то тип, …)
```

То есть внутри круглых скобок перечисляются все нужные типы.

```
let position: (Double, Double) = (54.42, 63.17)
```

Список примитивных типов в Swift вполне стандартный: **Character, String, Int, Float, Double, Bool**. **Character** используется для одиночных символов, **String** – для строковых данных, **Int** – для целых чисел, **Float** – для чисел с плавающей точкой в диапазоне от $\mathbf{1.2 \times 10^{-38}}$ до $\mathbf{3.4 \times 10^{38}}$, **Double** – для чисел с плавающей точкой с большим диапазоном – от $\mathbf{2.3 \times 10^{-308}}$ до $\mathbf{1.7 \times 10^{308}}$, **Bool** – для значений типа истина/ложь.

```
let character = “s”
var language: String = “Swift”
let age = 17
let pi: Float = 3.1415
let latitude: Double = 27.7007697012432
let isValid: Bool = true
```

Еще одним типом является тип **массив**. В Swift переменные такого типа объявляются с использованием квадратных скобок:

```
let someArray = ["Cat", "Dog", "Cow", "Chicken"]
let otherArray: [String] = ["Cat", "Dog", "Cow",
"Chicken"]
```

Как и с другими типами, если из начального значения можно однозначно сделать вывод о типе переменной/константы, то тип указывать необязательно. Доступ к элементам осуществляется также с помощью квадратных скобок, нумерация индексов начинается с 0:

```
let first = someArray[0]
```

В примере в константу `first` запишется значение первого элемента массива – `"Cat"`.

Если попытаться передать индекс со значением больше, чем количество элементов в массиве, то при выполнении кода будет выдана ошибка переполнения массива. В качестве самостоятельного задания создайте массивы с другими примитивными типами, попробуйте получить доступ к пятому элементу массивов из примеров выше.

Еще в Swift есть такой тип, как **словарь** (**dictionary**). В большинстве языков программирования словари представляют собой вид коллекций, которые реализованы в стандартной библиотеке языка, здесь же словари являются отдельным типом, конкретный вариант которого программист создает сам. Синтаксис словарей довольно прост и похож на синтаксис массивов:

```
[тип_ключа: тип_значения]
```

Например:

```
let someDictionary: [String: Int] = ["Misha": 172,
"Vova": 165, "Lena": 159]
```

В примере выше мы определили словарь с типом ключа `String` и типом значений `Int`. Значения словаря могу быть получены с помощью квадратных скобок: `["Misha"]` вернет `172`. Если попытаться обратиться по несуществу-

ющему ключу, то результатом будет `nil`. Ключом в словаре может быть любой тип, который реализует протокол `Hashable` из стандартной библиотеки Swift. В качестве самостоятельного задания создайте несколько словарей с разными примитивными типами.

И еще один тип, который не является примитивным, но довольно важный в Swift – тип **Any**. Переменная/константа этого типа может содержать *любое значение,* будь то примитивный тип, любой другой именованный тип, составной тип и функция. Сам по себе этот тип используется чаще всего для приведения к другому типу:

```
var str: Any
…// где-то в коде переменной присвоили значение
let result = str as? String
```

В примере объявляется переменная типа `Any`, затем ей присваивается какое-то строковое значение и, наконец, ее значение присваивается другой константе с попыткой явного приведения к типу String.

В Swift, как и в Котлин, есть такое понятие **оператора**. Операторы в Swift ничем особым не отличаются, самый простой – оператор присваивания, который обозначается знаком «=». Также имеются операторы арифметических действий (сложение, вычитание, умножение, деление и т. д.), оператор проверки равенства **«==»**, условные операторы **if, switch**, операторы цикла **for, while**.

Функции в Swift объявляется при помощи ключевого слова **func,** далее указывается наименование функции, далее в круглых скобках перечисляются **аргументы функции** (функция так же может не иметь никаких аргументов), далее после знака **«->»** может указываться тип **возвращаемого значения** (если возвращаемого значения нет, то после круглых скобок ничего не указывается). И наконец, в фигурных скобках пишется код – **тело функции**.

```
func sum(a: Int, b: Int)-> Int {
let result = a + b
return result
}

let c = sum(a: 12, b: 13)
print(c)
```

В примере описана функция с названием `sum`, которая, как можно догадаться из названия, служит для сложения чисел. Она принимает в себя два целочисленных аргумента – `a` и `b`. Далее видно, что функция возвращает результат в виде целого числа `Int`. В теле функции имеется два оператора – сложения «+» и `return`, который возвращает полученный ответ в виде числа `result`. При вызове функции внутри круглых скобок указываются все аргументы, которые были объявлены при создании функции (в примере выше это два целых числа). Результат выполнения функции запишется в переменную `c`, которую можно использовать далее в программе. В качестве самостоятельного задания реализуйте функцию вычитания по примеру функции сложения выше.

Одна из стандартных функций, которая нам нужна будет постоянно в работе – функция вывода в консоль **print().**

```
    print("Привет, Мир!!!")
    print("вот так можно распечатать выражение:
\(12+15)")
```

При необходимости распечатать какое-то выражение или значение из переменной нужно воспользоваться конструкцией **\()**, как видно из примера выше.

Кроме функций с именами в Swift есть еще один вид конструкций похожего типа – **Closures** (так называемые блоки). Это просто блок кода, который ограничен фигурными скобками, может принимать какие-то аргументы, может возвращать результат и имеет доступ к переменным и константам, которые доступны в этой области видимости.

```
    let numbers = [1, 2, 3, 4]
    let newNumbers = numbers.map({ (number: Int) -> Int in
    let result = 2 * number
    return result
    })
```

В примере выше показано, как используется стандартная функция `map`, которая для каждого элемента массива `numbers` производит умножение на 2, результатом будет новый массив. Можно видеть, что на каждом шаге блок кода принимает целое значение (элемент массива `numbers`) и возвращает целое число – результат умножения на 2. Попробуйте самостоятельно изменить пример так, чтобы для четных элементов возвращался 0.

2.4. Понятие Optional

В стандартной библиотеке языка Swift есть конструкция так называемого синтаксического сахара, которая позволяет безопасно работать с пустыми объектами – **nil**. Эта конструкция носит название **Optional** и обозначается знаком вопроса «?», приставленным вплотную к имени типа:

```
var optionalDouble: Double?
```

Подобная запись подразумевает, что такая переменная может содержать как нормальное значение типа **Double**, так и пустое значение, **nil**. Инициализировать такую переменную необязательно, ей автоматически присвоится значение nil. Если мы уверены, что переменная содержит значение, то можно явно попытаться получить это значение:

```
optionalDouble = 135.5
optionalDouble! // 135.5
```

В примере выше мы сначала присвоили переменной `optionalDouble` значение `135.5`, а затем с помощью восклицательного знака явно получили это значение (эта операция называется операцией **разворачивания**, или unwrap). Если же уверенности в том, содержит переменная значение или **nil**, нет, то обращаться к переменной следует с помощью вопросительного знака:

```
optionalDouble?
```

При попытке выполнить разворачивание к переменной со значением nil возникнет ошибка, поэтому во всех случаях, когда уверенности в значении переменной нет, следует обращаться к ней с вопросительным знаком.

Таким образом, Optional помогает избежать ошибок при попытке обратиться к пустому объекту (ошибки вида null pointer exception).

2.5. Комментарии в Swift

В Swift комментарии практически не отличаются от комментариев в Котлин. Здесь их два вида – однострочные и многострочные. Однострочные комментарии выделяются двумя знаками слеша:

```
// это однострочный комментарий в Swift
```

Многострочный комментарий начинается и заканчивается там, где поставлены знаки «/*» и «*/»:

```
/* Это
Многострочный
Комментарий
*/

// Инициализируем переменную
var str = «Swift»
// распечатаем значение переменной
print(str)

/* Функция умножения двух чисел
*/
func multiply(a: Int, b: Int) -> Int {
return a * b
}
```

2.6. Основные операторы: условные операторы if, switch, операторы цикла for и while

Swift предоставляет вполне стандартный набор операторов. Условный оператор **if** выглядит следующим образом:

```
let answer = "Yes"
if answer == "Yes" {
    print("Correct")
} else {
    print("Incorrect")
}
```

Точно так же, как в других языках, оператор **if** может быть составным:

```
let yes = "Yes"
let no = "No"
var answer = "Maybe"
if answer == yes {
    print("Correct")
} else if answer == no {
    print("Incorrect")
} else {
    print("Wrong answer")
}
```

Более сложным условным оператором является оператор **switch**, синтаксис которого тоже простой и понятный:

```
let yes = "Yes"
let no = "No"
let maybe = "Maybe"
var answer = "Maybe"

switch answer {
    case yes: print("Accepted")
    case no: print("Discarded")
    case maybe: print("Choose one more time")
    default: print("Undefined answer...")
}
```

После ключевого слова `switch` идет переменная, значение которой нужно проверить, далее внутри фигурных скобок перечисляются возможные варианты, помеченные ключевым словом `case`, последний вариант отмечается ключевым словом `default` – этот вариант сработает, если ни один из перечисленных «кейсов» не подойдет. Попробуйте самостоятельно изменить код выше, создав массив цветов (colors) и разобрав каждый цвет в отдельный `case` оператора `switch`.

Ни одна программ не обходится без операторов цикла, особенно полезен оператор цикла **for** для прохождения по коллекциям. Есть несколько разновидностей этого оператора, отличающиеся синтаксисом:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

for number in numbers {
    print("Current number = \(number)")
}
```

Самый простой и удобный вариант, когда после ключевого слова объявляется константа, которая на каждом шаге цикла будет соотноситься с текущим элементом коллекции, в примере выше `number` – это такая константа, `in` – ключевое слово, означающее, что константа будет проходить по коллекции, `numbers` – собственно коллекция, по которой осуществляется проход. В теле цикла можно производить различные манипуляции с полученной константой. Такой вариант цикла `for` часто еще называют циклом **foreach**. Самостоятельно попробуйте в теле цикла присвоить какое-то значение константе `number`, убедитесь, что получится ошибка компилятора.

Второй вариант предполагает явное объявление индекса и получение элемента коллекции внутри цикла по этому индексу:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

for index in 0..<numbers.count {
    let number = numbers[index]
    print("Current number = \(number)")
}
```

Как можно видеть, мы определили константу, которая будет проходить по индексам коллекции от 0 до `numbers.count` не включительно. Конструкция `..<` как раз определяет диапазон, не включающий элемент справа. Если нужно в диапазон включить указанный справа элемент, то конструкция будет `0...`, в общем случае, разумеется, слева тоже может быть другое значение. Пользуясь этим способом, стоит помнить о границах коллекции и не выходить за ее пределы. Попробуйте изменить код из примера выше таким образом, чтобы при выполнении кода программа завершилась ошибкой выхода за границы массива.

Оператор цикла **while** обычно используется, когда условие окончания цикла не такое простое, как достижение последнего элемента цикла:

```
let numbers = [1, 3, 5, 7, 9, 11, 13]

var sum = 0
var index = 0
while sum < 15 && index < numbers.count {
    sum += numbers[index]
    index += 1
}
print("Total sum = \(sum)")
```

Так, в примере выше нужно было посчитать сумму первых элементов массива таким образом, чтобы итоговая сумма едва превысила указанное значение. Обратите внимание, что при таком проходе по коллекции нужно самому увеличивать индекс и отслеживать, чтобы индекс не вышел за границы массива.

2.7. Классы

Классы в Swift аналогичны классам в других языках программирования, ключевое слово для определения класса – class. Переменные, константы и функции в классе определяются точно так же, как и в других местах.

```
class Animal {
// хранит количество конечностей животного
var pawsCount = 4

func description() -> String {
return "У этого животного \(pawsCount) конечностей"
    }
}

let animal = Animal()

print(animal.description())
```

В примере выше мы определили класс `Animal`, с переменной, которая хранит количество лап у животного, и простой функцией, которая возвращает в виде строки описание с количеством лап. Затем мы создали экземпляр этого

класса и вывели на печать количество лап у него. Попробуйте самостоятельно создать новый экземпляр класса и изменить ему количество лап (например, задать равным двум).

Говоря о классах, стоит упомянуть про **конструктор** класса. В Swift он обозначается ключевым словом **init:**

```
init(paws: 2) {
pawsCount = paws
}
```

Например, этот конструктор подойдет для созданного ранее нами класса Animal, он принимает в себя количество конечностей, которое затем присваивается в поле класса. Попробуйте самостоятельно добавить в класс новое поле, например возраст животного, имя, и затем дополните конструктор новыми значениями.

Иногда в методах класса нужно явно обратиться к полям или функциям этого класса, это можно сделать, используя ключевое слово **self**:

```
init(pawsCount: 2) {
self.pawsCount = pawsCount
}
```

Так, в примере выше в конструктор передается аргумент с таким же именем, как поле класса, и если обратиться без приставки `self`, то компилятор посчитает, что вы хотите присвоить этот аргумент самому себе, и выдаст ошибку (аргументы, переданные внутрь функции, в теле функции менять нельзя, они константы).

Кроме классов, в Swift есть **протоколы**, обозначаются ключевым слово protocol:

```
protocol Walkable {
    var position: (Double, Double) {get}
mutating func walk()
}
```

Протокол – это некий интерфейс объекта, но без конкретной реализации. Точно так же, как класс, он может содержать переменные/константы, функции, но все должны быть без инициализации и реализации, переменные должны

быть помечены `{get}`. Ключевое слово `mutating` обозначает, что метод может изменять структуру протокола. Экземпляр протокола напрямую создавать нельзя (ведь у такого объекта не будет реализации методов и не будет значений у полей), поэтому от протоколов наследуются классы, реализуя их методы и инициализируя поля:

```
class Dog: Walkable {

    var position: (Double, Double) = (0.0, 0.0)

    func walk() {
        position = (position.0 + 1, position.1 + 1)
    }
}
```

Также обратите внимание, как происходит вызов переменной составного типа: к первой части обращение идет через `position.0`, ко второй – через `position.1`.

Теперь попробуем создать экземпляр созданного класса, вызвать его метод и распечатать новое положение объекта:

```
let dog = Dog()
dog.walk()
dog.walk()

print(dog.position)
```

В качестве самостоятельного задания добавьте в протокол метод run(), а в реализации напишите изменение позиции на +10.

Для добавления новых функций уже имеющимся классам и протоколам (чаще всего это касается объектов стандартной библиотеки) на помощь приходят **расширения** (extensions):

```
extension Int {
    func isOdd() -> Bool {
        return self % 2 != 0
    }
}

print(13.isOdd())
```

В примере выше мы расширили стандартный класс `Int`, добавив новый метод проверки, является ли число нечетным. Попробуйте добавить новый метод, который проверяет делимость числа на 3.

2.8. Базовые коллекции в Swift

В стандартной библиотеке Swift нет коллекций в том виде, как в других языках. Здесь они представлены в виде массивов, множеств и словарей. Множество – это коллекция, которая в неупорядоченном виде содержит уникальные элементы (т. е. элементы не могут быть одинаковыми и их порядок не определен). Про массивы и словари мы уже говорили ранее, отметим лишь дополнительно некоторые их особенности.

Все три типа коллекций в Swift реализуют шаблон generics collections. Это означает, что в качестве элементов в этих коллекциях можно использовать любые типы, важно лишь их указывать при объявлении переменных или констант. Про Generics можно почитать самостоятельно более подробно, здесь лишь укажем, что это позволяет абстрагироваться от конкретных реализаций для конкретных типов, что в свою очередь дает большую гибкость и мощь в написании кода. Если коллекция объявлена в виде переменной, то ее можно изменять – добавлять, удалять, изменять элементы, а если коллекция объявлена константой, то количество элементов в ней изменяться не может.

Приведем пример множества:

```
var letters = Set<Character>()
print("letters is of type Set<Character> with
\(letters.count) items.")
```

В использовании множества аналогичны массивам, по ним можно проходить в цикле, можно удалять и добавлять элементы, но не по индексу, а по значению.

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip
hop"]
for genre in favoriteGenres {
print("\(genre)")
}
```

2.9. Установка инструментов разработчика, первая программа iOS

После изучения теоретических основ Swift перейдем к непосредственной разработке на iOS. Для установки программы разработки на iOS в первую очередь понадобится компьютер с MacOS, ни на каких других ОС данный инструмент не имеется. Программа для разработки на iOS называется XCode, скачать ее можно с официального сайта разработчика: https://apps.apple.com/us/app/xcode/id497799835.

После скачивания установочника XCode установите ее на компьютер.

По окончании установки запустите XCode, перед вами откроется стартовое окно с возможностью создать новый проект, открыть уже имеющийся или изменить какие-либо настройки. Нажмите **Create a new XCode project**, у вас откроется окно выбора типа проекта, выберите вариант App и нажмите Next (рис. 18).

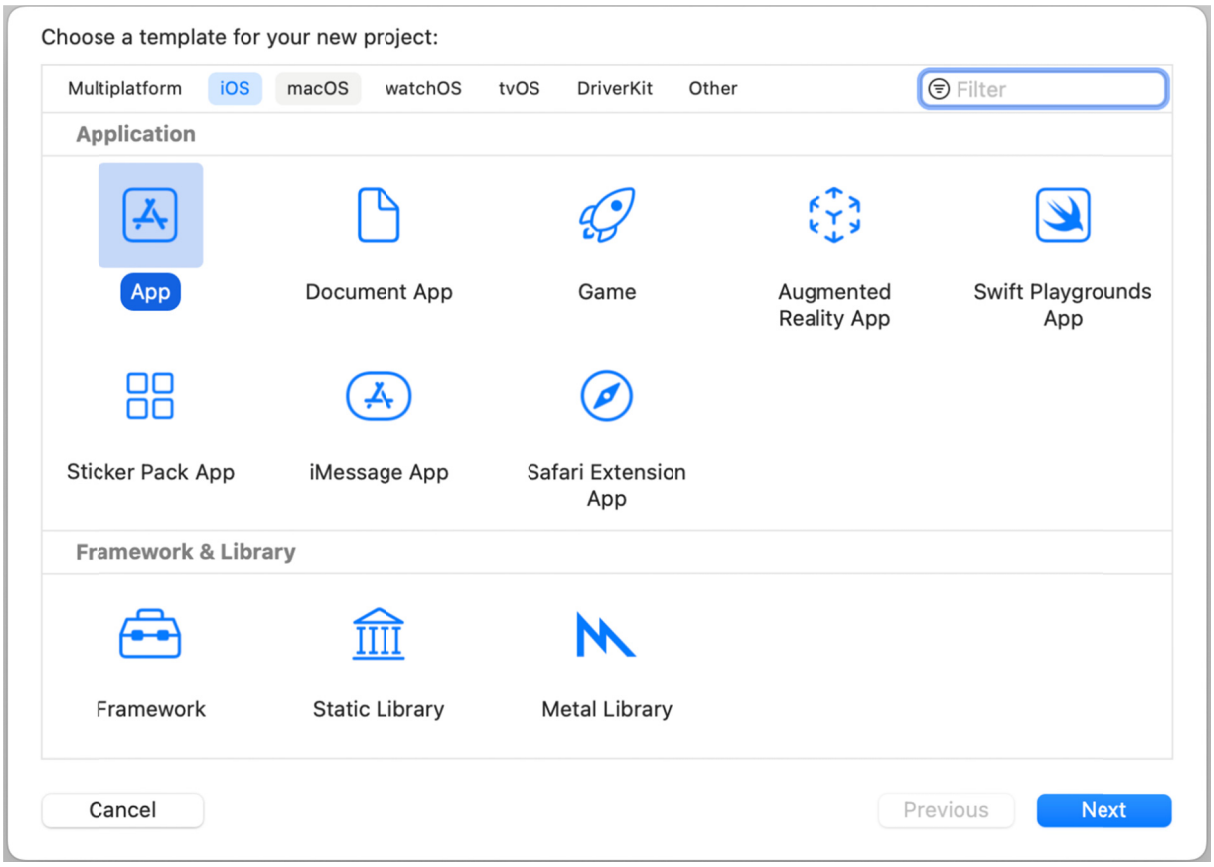

Рис. 18. Окно выбора типа проекта

Далее введите название, например “Hello World”, нажмите Next, подтвердите место сохранения проекта и дождитесь, пока XCode настроит проект. Окно только что созданного проекта выглядит примерно так, как на рис. 19.

Вверху окна можно увидеть пункт выбора эмулятора, на рис. 19 это iPod touch (7th generation), при нажатии на который появится список устройств, эмуляторы которых установились вместе с XCode, выберите любой, который вам по душе. После того как эмулятор выбран, нажатием кнопки черного треугольника в левой части экрана запустите приложение. Окно эмулятора будет выглядеть примерно так, как на рис. 20.

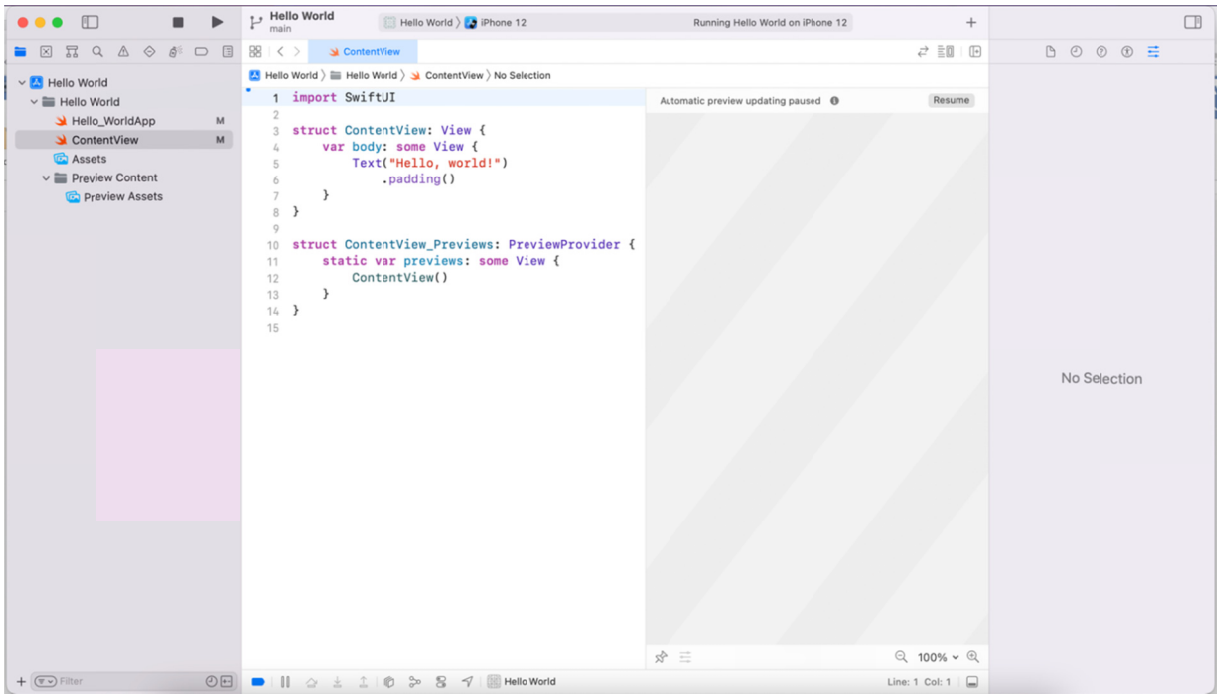

Рис. 19. Окно только что созданного проекта

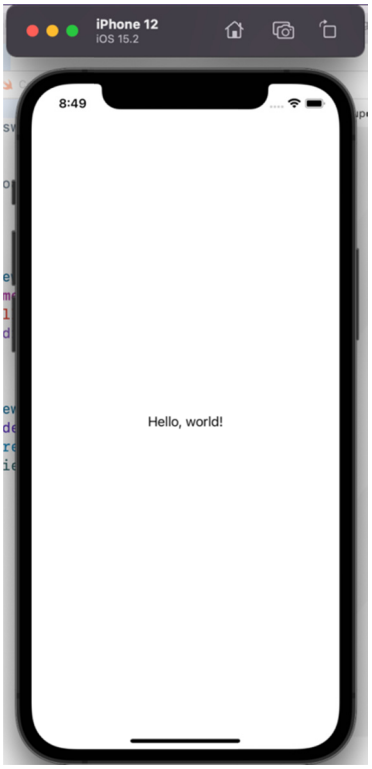

Рис. 20. Окно эмулятора с запущенным первым проектом

2.10. Структура iOS проекта

Все приготовления завершены, программа успешно запущена, теперь разберемся, что там под капотом. В левой части экрана находится панель Навигатора с разными вкладками. По умолчанию на панели Навигатора отображается структура проекта. Из полезных на начальном этапе вкладок можно выделить вкладку поиска (иконки лупы) и вкладку с ошибками компиляции/выполнения (иконка с восклицательным знаком внутри треугольника).

По центру экрана располагается окно редактора. В случае если в нем открыт файл с версткой, то окно делится еще пополам, в левой части которой код верстки, а в правой – визуальное представление. При первом запуске XCode часто не показывает визуального представления, выдавая ошибку Automatic preview updating pause, попробуйте нажатием кнопки Resume в верхнем правом углу окна редактора обновить визуальное отображение, и если все пройдет успешно, то вы увидите окно эмулятора с одной строкой “Hello World”.

В правой части экрана находится окно Инспектора с разными вкладками, каждая из которых содержит определенный набор различных настроек для файла, который на данный момент открыт в редакторе. На начальном этапе эти свойства нам не понадобятся.

Вернемся к структуре проекта. У главного элемента Hello World есть подпапка с таким же названием, это сделано для случая, если в проект добавятся другие библиотеки и подпроекты. В папке Hello World находятся Swift-файлы, элемент Assets (это не файл, а папка, внутри которой будут лежать все иконки и цвета нашего приложения) и папка Preview Content (на начальном этапе эта папка нам неинтересна). Из Swift-файлов первый – это Hello_WorldApp, который является точкой входа в наше приложение, этот класс автоматически вызовется при запуске приложения. Второй – уже упоминавшийся ContentView, который содержит верстку главного экрана приложения. Логика взаимодействия этих двух классов очень простая: при запуске начинает свою работу Hello_WorldApp, внутри которого есть вызов ContentView (откройте файл Hello_WorldApp и найдите там объявление ContentView). Можно обратить внимание, что в коде имеются элементы, не очень похожие на ключевые слова Swift – это из-за того, что здесь используется библиотека (фреймворк) SwiftUI. Он прежде всего используется для верстки и частично для реализации логики приложения. Заглянем в Assets, сейчас там всего 2 элемента – AccentColor и AppIcon. Первый – это элемент цвета, значение которого можно задать самому и далее использовать в программе. Второй –

это главная иконка приложения, при клике на которую можно увидеть, что для нее требуется довольно большое количество картинок разного размера.

2.11. Макет приложения

Начнем разработку нашего приложения созданием верстки, для этого снова откроем файл ContentView. Самой первой строчкой идет `import SwiftUI` – она указывает, что в данном файле будет использоваться библиотека `SwiftUI`. Конструкция

```
struct ContentView: View {
}
```

объявляет о создании объекта **представления** (View), который будет содержать все элементы верстки главного окна приложения. Такой объект представления в SwiftUI должен содержать в себе переменную с названием `body` типа `some View` – эта ключевая связка означает, что переменная будет использоваться в строительстве интерфейса.

```
var body: some View {
}
```

И уже внутри этой переменной описывается весь интерфейс окна, на данный момент там один-единственный элемент – `Text()` с текстом «`Hello, world!`», у элемента так же вызывается функция `padding()` – такие функции **называются модификатором представления** (View Modifier), `padding` задает автоматические отступы между краями элемента и текстом внутри.

Добавим в интерфейс еще элементов, сделаем приложение, которое считает периметр прямоугольника по двум заданным сторонам. Для начала добавим контейнер для всех наших представлений, поскольку `body` должна непосредственно содержать только одно представление. Возьмем контейнер `VStack` – для вертикального расположения всех элементов. Аналогичен ему `HStack` – для горизонтального расположения элементов.

```
VStack {
}
.padding()
```

Зададим контейнеру тоже стандартный отступ. Добавим заголовок для нашего окна, обычный `Text()` с надписью `"Калькулятор периметра прямоугольника"`, добавим отступ.

Заранее создадим переменные внутри нашего объекта, которые будут использоваться для хранения значений полей вводов и для результата:

```
@State var width: String = ""
@State var height: String = ""
@State var perimeter = 0.0
```

В SwiftUI у объекта представления (каковым в нашем случае является `ContentView`) переменные объявляются с помощью ключевого слова `@State` – оно подсказывает, что при изменении переменной наш объект представления тоже будет меняться. В примере выше мы объявили три переменные: для хранения ширины прямоугольника, его высоты и для хранения результата вычисления периметра.

Далее нам нужно два поля для ввода данных. Можно расположить их рядом, для этого используем `HStack`. Внутрь горизонтального контейнера добавим два элемента `TextField` – этот элемент используется в SwiftUI для ввода данных.

Синтаксис элемента `TextField` несколько сложнее, чем у `Text`, здесь так же есть строковое значение (используется для отображения подсказки при вводе), но еще здесь есть так называемый биндинг (Binding) – особый тип данных, который позволяет отслеживать изменение переменной этого типа и реагировать нужным образом. В нашем случае это будет использоваться для того, чтобы отслеживать изменение значений в полях ввода и сохранение их в переменную типа биндинг. Вот так выглядит конструктор этого элемента:

```
public init(_ titleKey: LocalizedStringKey, text:
Binding<String>)
```

Обычную переменную, объявленную через `@State` тоже можно передать в качестве биндинга, помечая знаком доллара:

```
$width
```

Теперь мы можем наконец-то создать два элемента с полями ввода параметров нашего прямоугольника:

```
TextField("Длинна", text: $width)
                    .padding()
```

Самостоятельно добавьте элемент для высоты (не забудьте поместить оба поля в `HStack`).

Далее добавим кнопку, при нажатии на которую будет вычисляться периметр и отображаться. У кнопки интерфейс довольно простой: сначала в круглых скобках задается заголовок кнопки, затем в фигурных скобках расписывается код, который выполнится при нажатии на эту кнопку (не забываем добавить `padding`):

```
Button("Вычислить периметр") {
}
.padding()
```

Осталось добавить простое текстовое поле для отображения результатов вычислений. Единственный нюанс: результат можно отобразить как часть строки, подставляя внутрь текста:

```
"Периметр = \(perimeter)",
```

где переменная `perimeter` – вычисленное значение периметра, которое мы объявляли ранее. Попробуйте запустить полученный код и посмотреть, что получилось (рис. 21).

Попробуем немного разнести интерфейс, заголовок уберем вверх, а остальные элементы оставим по центру. Для этого добавим два элемента `Spacer()` между заголовком и `HStack` и между нижним текстовым элементом и нижним краем `VStack`. `Spacer` – это элемент, не имеющий никакого внешнего отображения, но влияющий на расположение других элементов, основным свойством является возможность растягиваться на все свободное пространство между элементами. Также изменим стиль текста заголовка:

```
font(.system(size: 25))
```

Этот модификатор позволяет установить любой шрифт текстовому элементу и задать его размер. В нашем случае это системный шрифт с размером 25. Попробуем изменить в заголовке еще и цвет текста:

```
foregroundColor(Color.red)
```

В SwiftUI имеется довольно большое количество заранее определенных цветов, можете выбрать любой (XCode подскажет вам их). Запустите приложение и посмотрите, что получилось (ср.: рис. 22).

Попробуем еще изменить фон у контейнера `VStack`, это делается легко через модификатор `background`:

```
.background(Color.gray)
```

Наличия стандартных цветов не всегда бывает достаточно, поэтому добавим свой цвет. Для этого перейдем в Assets, нажатием на правую клавишу мыши вызовем меню и выберем пункт New color set. Назовем новый цвет headerTitle, в дальнейшем будем его использовать для задания цвета тексту заголовка. Выберем в центральном окне квадрат с надписью Any appearance – здесь задается цвет для темы айфона по умолчанию (можно отдельно задать цвет для ночной темы – в элементе с надписью Dark). Нажатием кнопки Show Color Panel в правом нижнем углу вызовем панель цветов и подберем понравившийся цвет. Он присвоится автоматически при нажатии (рис. 23).

Напрямую использовать цвета из Assets в верстке SwiftUI не очень удобно, постоянно обращаясь по имени цвета можно легко ошибиться, поэтому нужно воспользоваться обходным решением. Для этого создадим новый файл Swift (правой клавишей мыши на папку Hello World -> New File…) с названием **Colors+Extensions**. Здесь мы реализуем расширение стандартного класса `Colors` путем добавления новых констант, которые будут соответствовать нашим цветам из Assets.

```
import SwiftUI

extension Color {
    static let headerTitle = Color("headerTitle")
}
```

Теперь этот цвет можно получить по имени константы `Color.headerTitle`.

Самостоятельно создайте несколько разных цветов: для текстовых элементов, фона контейнеров, кнопки и текстовых элементов.

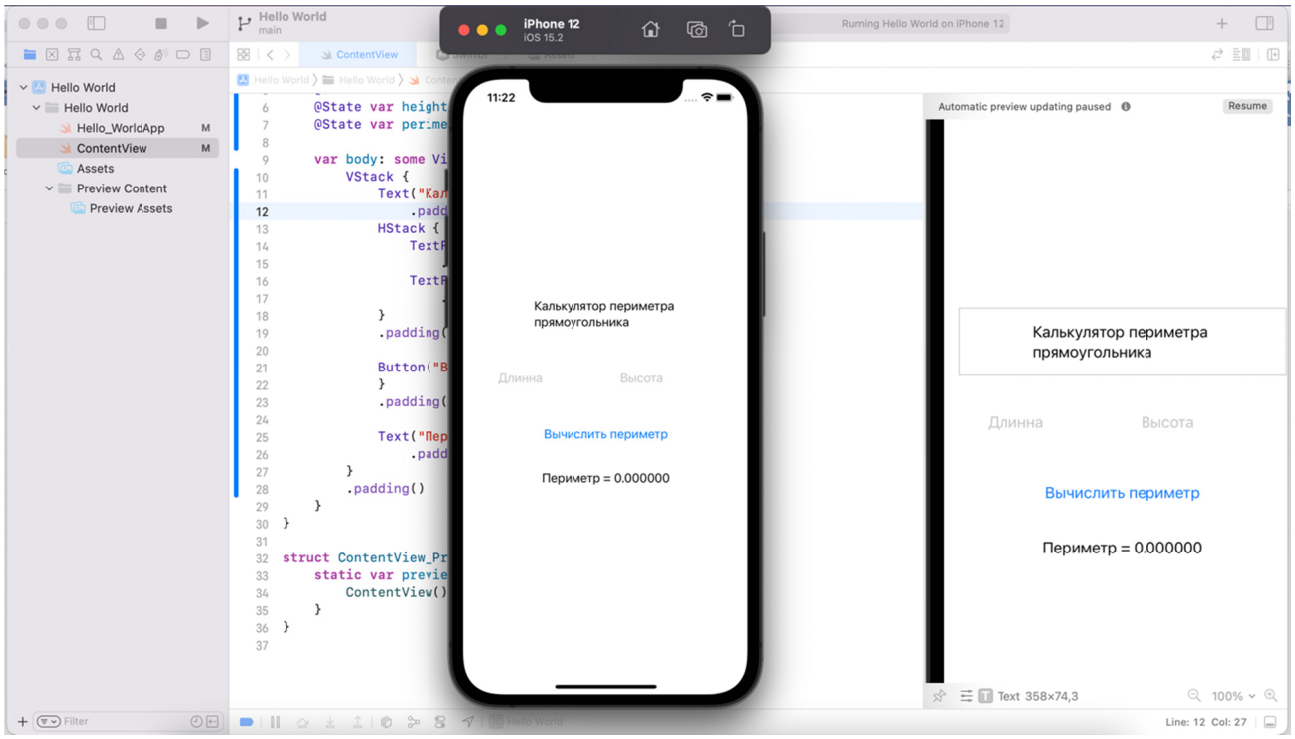

Рис. 21. Окно приложения со всеми добавленными элементами

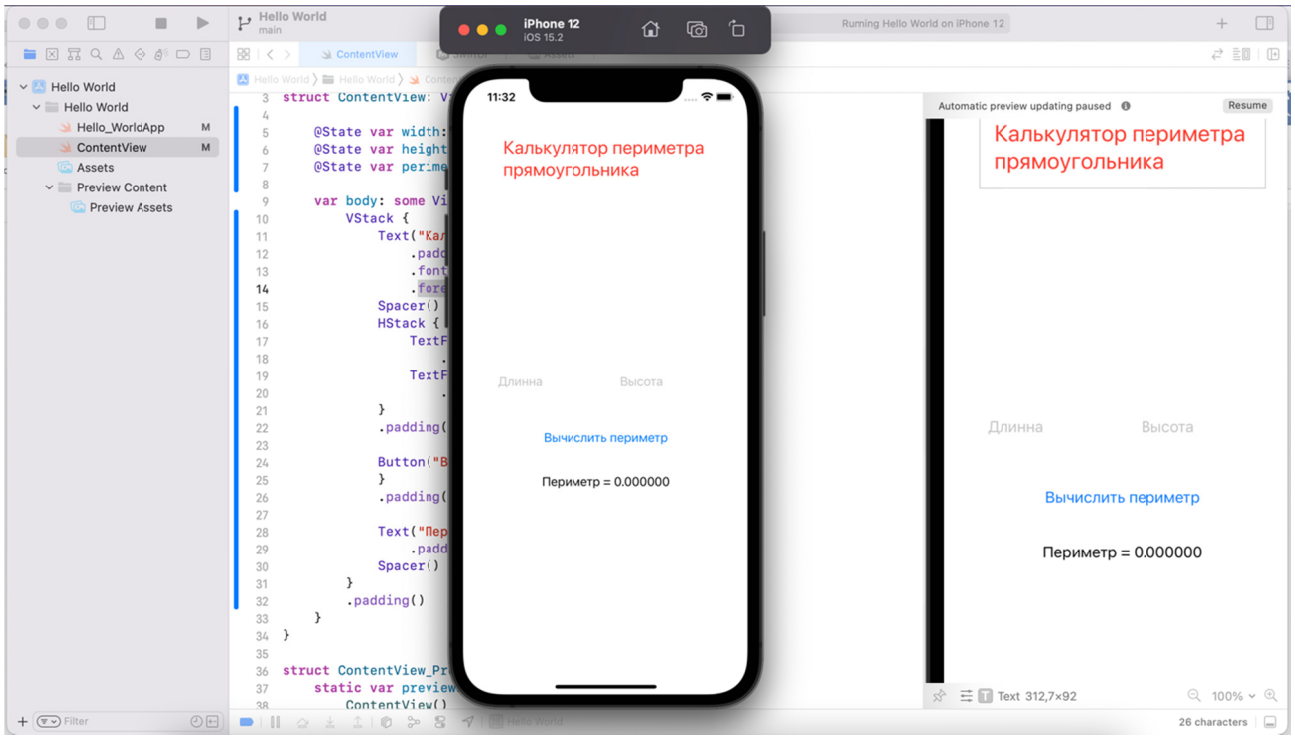

Рис. 22. Окно приложения с измененными заголовком и расположением элементов

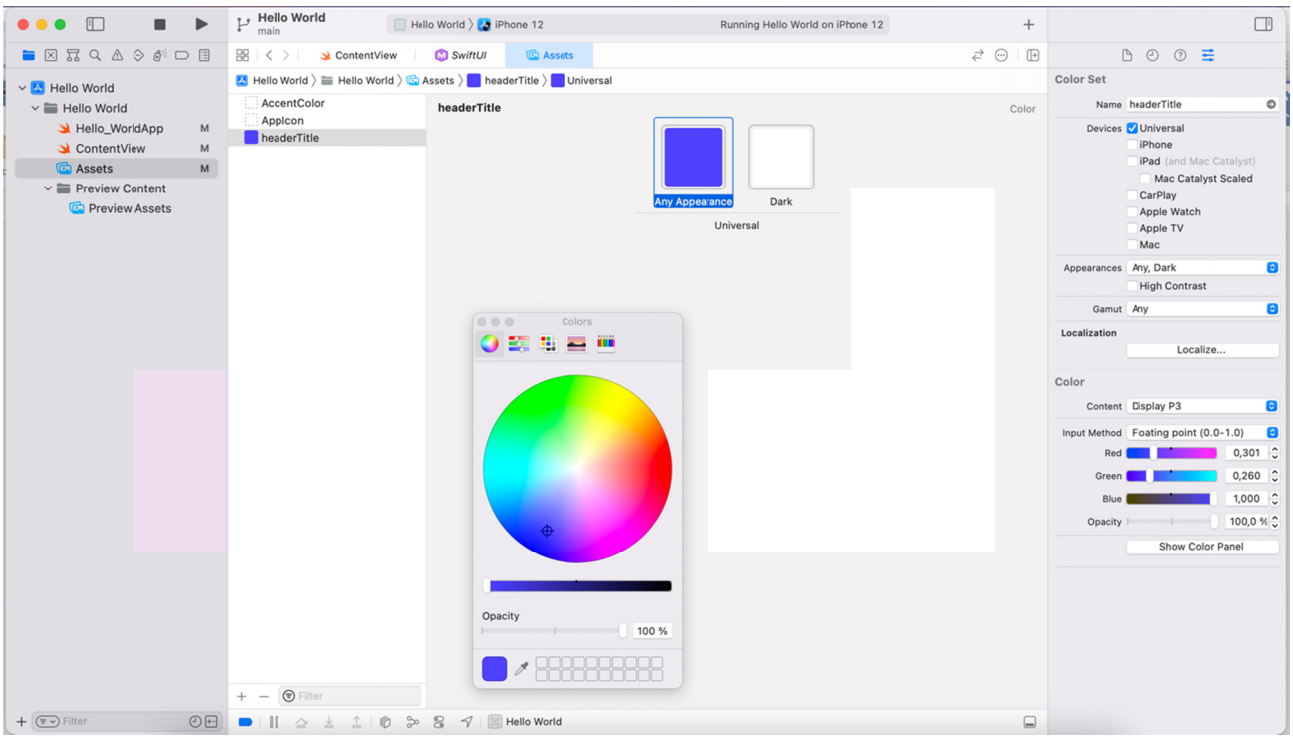

Рис. 23. Окно XCode с новым созданным цветом

2.12. Взаимодействие с интерфейсом

Реализовать функционал обработчиков в SwiftUI очень просто, добавим обработку нажатия на кнопку «Вычислить периметр», которая была определена в пункте 2.11. Для начала нужно получить значения, введенные пользователем, это можно сделать через конструктор типа Double, который принимает в себя строковое значение, пытается перевести в число и возвращает **Optional(Double)**. Если перевести не удалось, то в переменную запишется **nil**, т. е. нам нужно оба значения проверить на пустоту:

```
let w = Double(width)
let h = Double(height)
if w != nil && h != nil {

}
```

И в случае, если оба числа были успешно получены, мы вычисляем периметр и записываем его в переменную `perimeter`. В итоге код кнопки будет выглядеть следующим образом:

```
Button("Вычислить периметр") {
    let w = Double(width)
```

```
        let h = Double(height)
        if w != nil && h != nil {
            perimeter = 2 * (w! + h!)
        }
}
.padding()
```

Замените код кнопки с пустым телом на код, указанный выше, и запустите приложение (рис. 24).

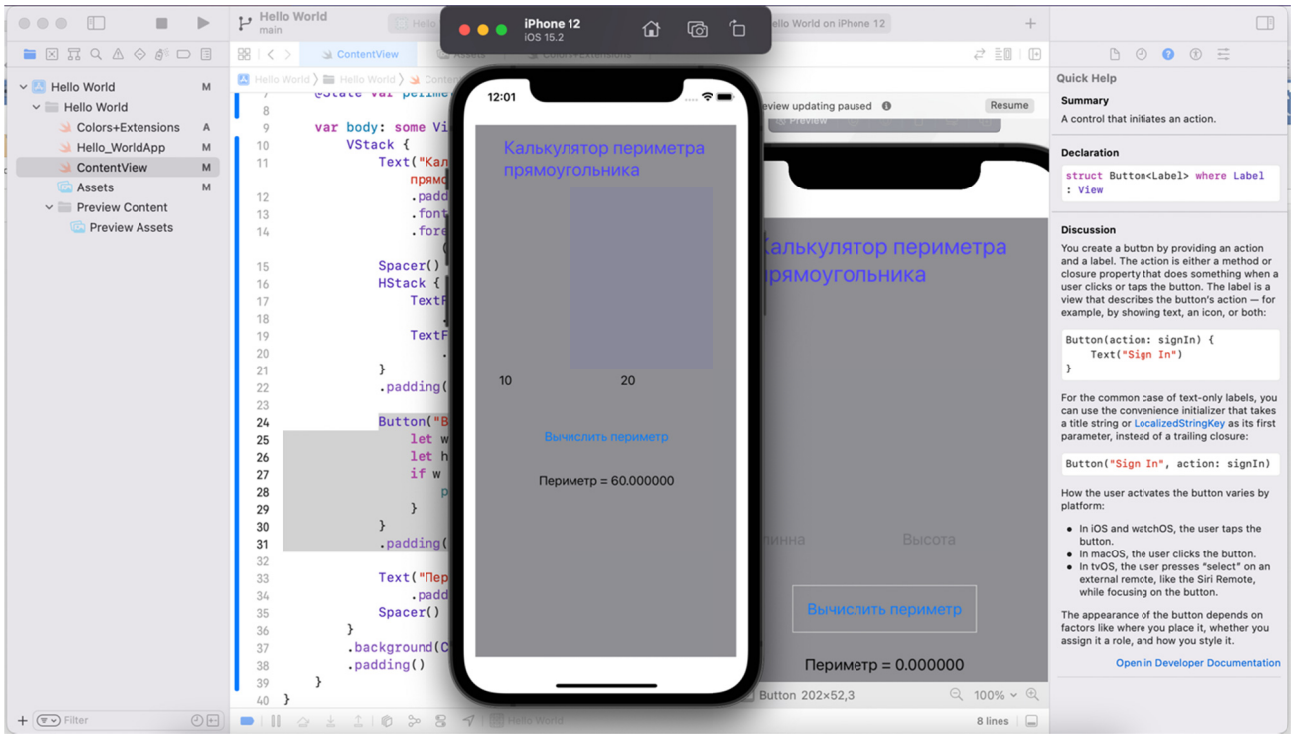

Рис. 24. Окно приложения с вычисленными значениями

Вот и все, калькулятор периметра прямоугольника готов.

Контрольные вопросы и задачи

*1. Можно ли в **Swift** присваивать переменной одного типа значение другого типа? Почему?*

*2. Что такое **словарь** (dictionary)?*

*3. В чем особенность типа **Any**?*

*4. В каком случае можно безопасно использовать операцию **разворачивания** (unwrap)?*

5. Как явно обратиться к полям и функциям класса?

6. Что такое протокол и расширение?

7. Что произойдет, если обратиться к словарю по несуществующему ключу?

*8. Какой модификатор используется для задания отступа между элементами в **SwiftUI**?*

9. Установите средство разработки XCode и запустите тестовый проект на эмуляторе, руководствуясь разделом 2.9.

10. Создайте приложение «Калькулятор периметра прямоугольника», руководствуясь разделами 2.11 и 2.12.

11. В качестве самостоятельной работы создайте калькулятор для арифметических операций:

- *10 цифр, от 0 до 9, все можно реализовать через **Button**;*
- *4 арифметические операции – **«+, -, *, /»,** реализовать через **Button**;*
- *2 поля ввода – для первого аргумента и для второго, **TextField**;*
- *одно текстовое поле **Text**, для вывода результата;*
- *кнопка очистки полей ввода.*

ЛИТЕРАТУРА

1. Официальная страница Котлин. – URL: https://kotlinlang.org/.

2. Официальная страница Swift. – URL: https://www.swift.org/.

3. Обучающие материалы по Котлин для Андроид от Google. – URL: https://developer.android.com/kotlin/learn.

4. Обучающие материалы по Swift для iOS от Apple. – URL: https://developer.apple.com/tutorials/app-dev-training.

5. Официальный онлайн-тренажер для Котлин. – URL: https://play.kotlinlang.org.

6. Онлайн-тренажер для Swift, использованный в пособии. – URL: https://www.onlinegdb.com/online_swift_compiler.

7. Официальный набор обучающих материалов по SwiftUI/Swift. – URL: https://developer.apple.com/swift/resources/.

Электронное учебное издание
сетевого распространения

Мусин Фоат Минегереевич
Егорчев Антон Александрович
Чикрин Дмитрий Евгеньевич
Азимуратов Шухрат Батырович

ПРОЕКТИРОВАНИЕ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ НА IOS И ANDROID

Учебное пособие

Корректор
А.Н. Егорова

Верстка
Т.В. Уточкиной

Дизайн обложки
М.А. Ахметова

Подписано к использованию 25.11.2022.
Формат 60×84 1/16. Гарнитура «Times New Roman». Усл. печ. л. 3,84.
Заказ 117/11

Издательство Казанского университета

420008, г. Казань, ул. Профессора Нужина, 1/37
тел. (843) 206-52-14 (доп. 1705), 206-52-14 (доп. 1704)