Практическая работа. Корутины

Прежде чем начать

В предыдущем уроке вы узнали о **coroutines**. Вы использовали Kotlin для написания параллельного кода с помощью coroutines. В этом уроке вы примените свои знания о корутинах в приложении для Android и его жизненном цикле. Вы добавите код для одновременного запуска новых coroutines и узнаете, как их тестировать.

Предварительные условия

- Знание основ языка Kotlin, включая функции и лямбды
- Умение создавать макеты в Jetpack Compose
- Уметь писать модульные тесты на Kotlin (см. урок «Написание модульных тестов для ViewModel»).
- Как работают потоки и параллелизм
- Базовые знания о корутинах и CoroutineScope

Что вы будете создавать

- Приложение `Race Tracker`, которое симулирует ход гонки между двумя игроками. Рассматривайте это приложение как возможность поэкспериментировать и узнать больше о различных аспектах корутинов.

Что вы узнаете

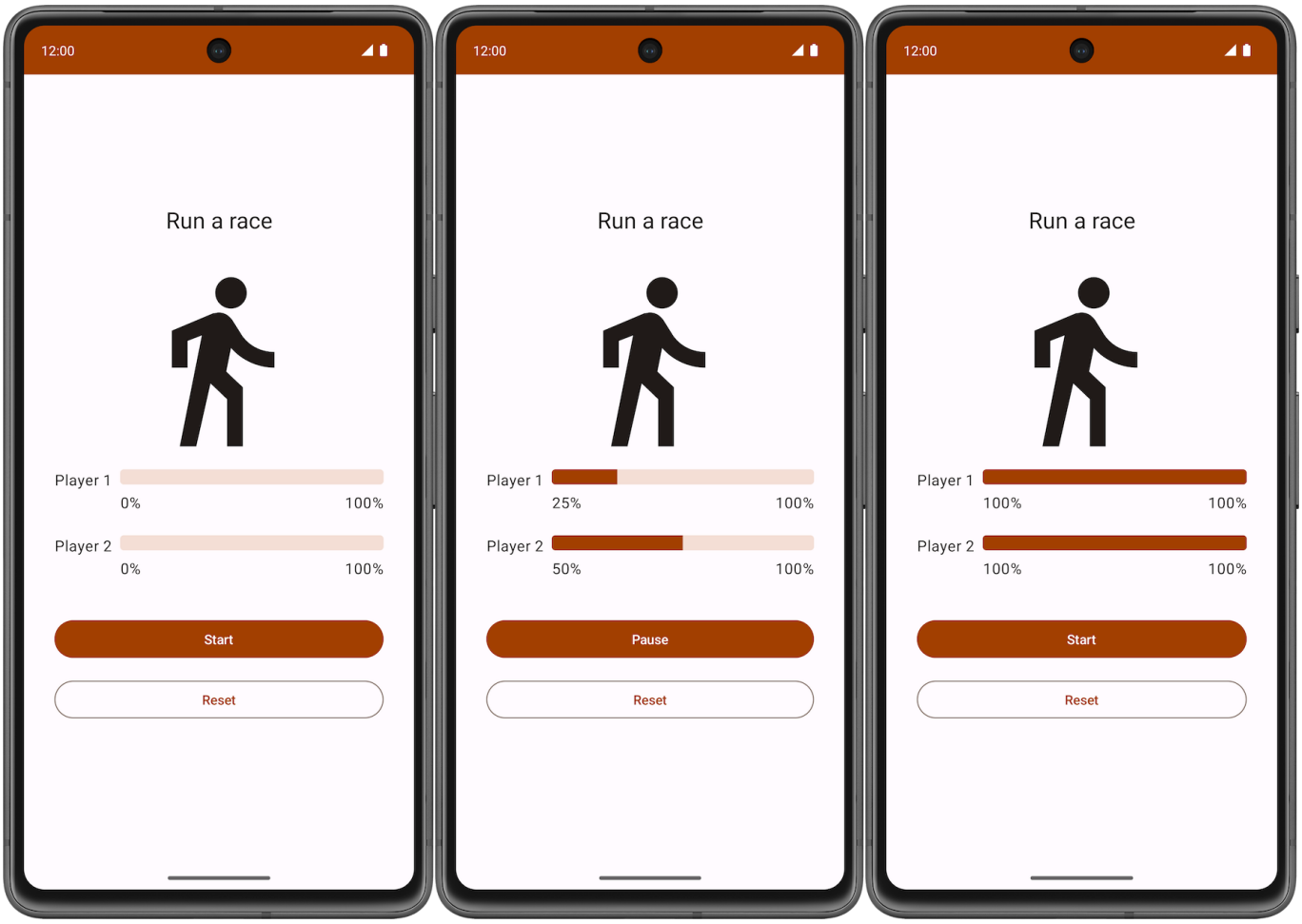
- Использование корутинов в жизненном цикле приложений для Android.
- Принципы структурированного параллелизма.
- Как писать юнит-тесты для проверки корутинов.

Что вам понадобится

- Последняя стабильная версия Android Studio

2. Обзор приложения

Приложение `Race Tracker` имитирует забег двух игроков. Пользовательский интерфейс приложения состоит из двух кнопок, `Start/Pause` и `Reset`, и двух прогресс-баров, показывающих прогресс гонщиков. Игроки 1 и 2 должны «бежать» на разных скоростях. Когда гонка начинается, игрок 2 продвигается в два раза быстрее, чем игрок 1.



В приложении вы будете использовать корутины, чтобы обеспечить:

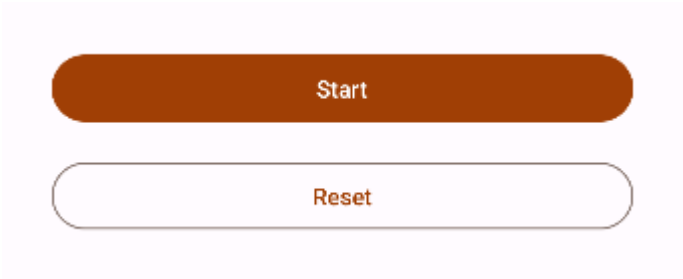
Оба игрока «бегут наперегонки» одновременно. Пользовательский интерфейс приложения отзывчив, а индикаторы прогресса увеличиваются во время гонки. Стартовый код содержит готовый код пользовательского интерфейса для приложения Race Tracker. Основная цель этой части коделаба - познакомить вас с корутинами Kotlin в приложении для Android.

Получите стартовый код

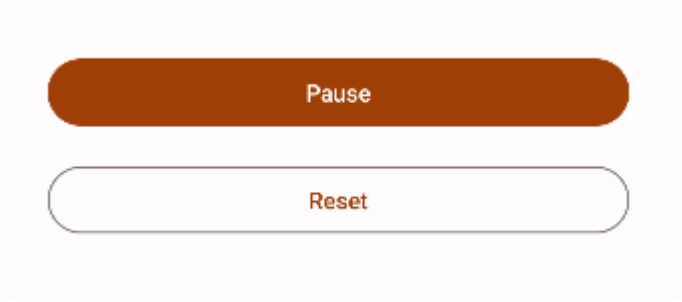
- Чтобы начать работу, склонируйте репозиторий для этого кода:
- скачайте из ресурсов race-tracker.7z
- откройте через cmd
- cd race-tracker
- git checkout starter

Прохождение стартового кода

Вы можете начать гонку, нажав на кнопку `Start`. Во время гонки текст кнопки «Старт» меняется на «Пауза».



В любой момент с помощью этой кнопки можно приостановить или продолжить гонку.



Когда гонка начинается, вы можете видеть прогресс каждого игрока с помощью индикатора состояния. Составная функция `StatusIndicator` отображает статус прогресса каждого игрока. Для отображения индикатора прогресса она использует составную функцию `LinearProgressIndicator`. Для обновления значения прогресса вы будете использовать корутины.

`RaceParticipant` предоставляет данные для приращения прогресса. Этот класс является держателем состояния для каждого из игроков и хранит имя участника, максимальный прогресс, которого необходимо достичь для завершения гонки, длительность задержки между приращениями прогресса, текущий прогресс в гонке и начальный прогресс.

В следующем разделе вы будете использовать корутины для реализации функциональности, позволяющей имитировать прогресс гонки без блокировки пользовательского интерфейса приложения.

Реализация прогресса гонки

Вам нужна функция `run()`, которая сравнивает текущий прогресс игрока с максимальным прогрессом (maxProgress), отражающим общий прогресс гонки, и использует функцию приостановки `delay()`, чтобы добавить небольшую задержку между приращениями прогресса. Эта функция должна быть приостанавливающей функцией, поскольку она вызывает другую приостанавливающую функцию `delay()`. Кроме того, позже в уроке вы будете вызывать эту функцию из coroutine. Выполните следующие шаги для реализации функции:

- Откройте класс `RaceParticipant`, который является частью стартового кода. Внутри класса `RaceParticipant` определите новую функцию приостановки с именем `run()`.

```
class RaceParticipant(
    ...
) {
    var currentProgress by mutableStateOf(initialProgress)
        private set

    suspend fun run() {

    }
    ...
}
```

- Чтобы смоделировать ход гонки, добавьте цикл `while`, который выполняется до тех пор, пока `currentProgress` не достигнет значения `maxProgress`, которое установлено в 100.

```
class RaceParticipant(
    ...
    val maxProgress: Int = 100,
    ...
) {
    var currentProgress by mutableStateOf(initialProgress)
        private set

    suspend fun run() {
        while (currentProgress < maxProgress) {

        }
    }
    ...
}
```

Значение `currentProgress` устанавливается в `initialProgress`, который равен 0. Чтобы смоделировать прогресс участника, увеличьте значение `currentProgress` на значение свойства `progressIncrement` внутри цикла `while`. Обратите внимание, что по умолчанию значение `progressIncrement` равно 1.

```
class RaceParticipant(
    ...
    val maxProgress: Int = 100,
    ...
    private val progressIncrement: Int = 1,
    private val initialProgress: Int = 0
) {
    ...
    var currentProgress by mutableStateOf(initialProgress)
        private set

    suspend fun run() {
```

```
        while (currentProgress < maxProgress) {
            currentProgress += progressIncrement
        }
    }
}
```

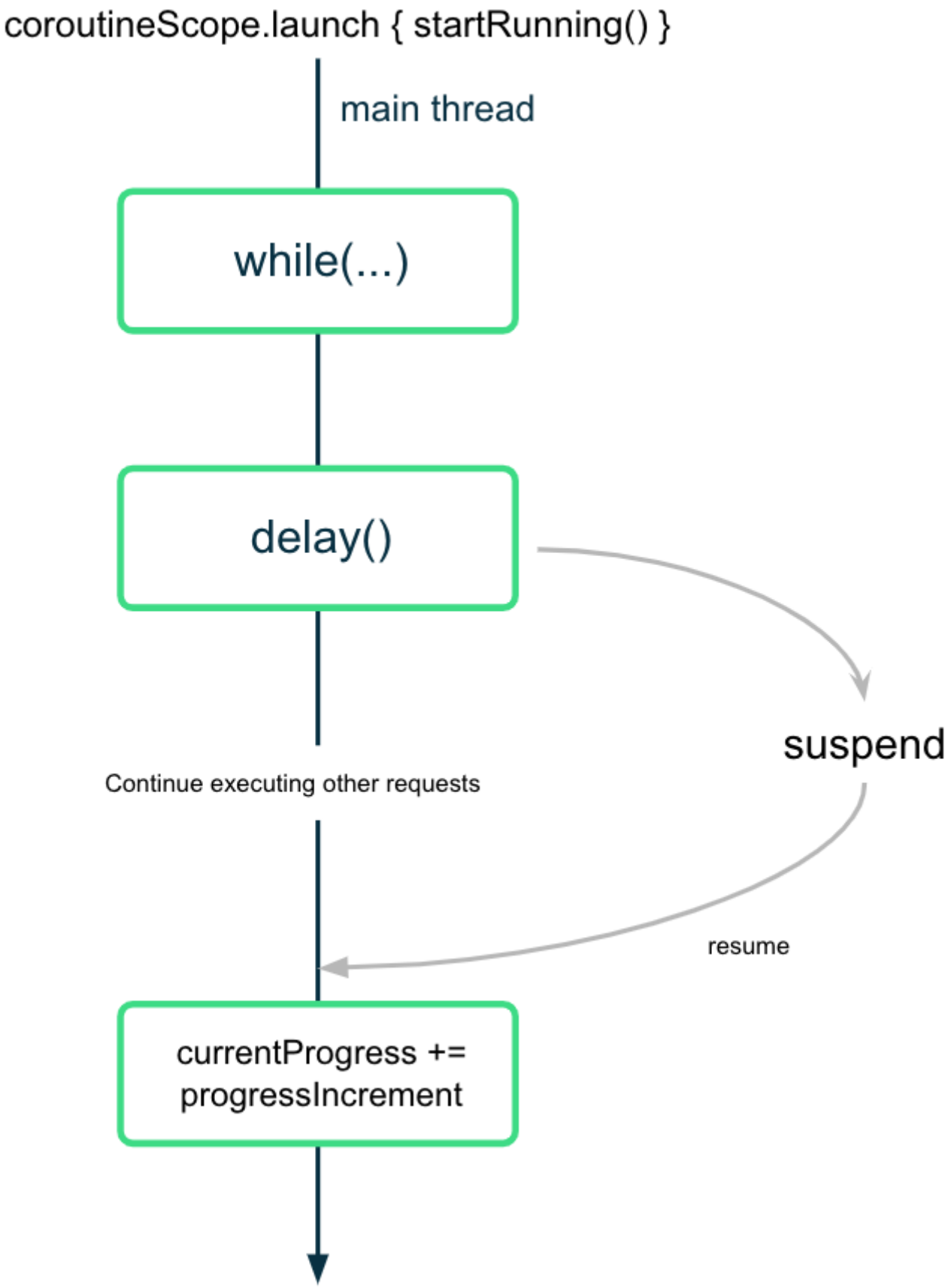
- Чтобы смоделировать различные интервалы прогресса в гонке, используйте функцию приостановки `delay()`. В качестве аргумента передайте значение свойства `progressDelayMillis`.

```
suspend fun run() {
    while (currentProgress < maxProgress) {
        delay(progressDelayMillis)
        currentProgress += progressIncrement
    }
}
```

Когда вы посмотрите на только что добавленный код, вы увидите значок слева от вызова функции `delay()` в Android Studio, как показано на скриншоте ниже:

Этот значок указывает на точку приостановки, когда функция может приостановиться и возобновиться позже.

Главный поток не блокируется, пока coroutine ожидает завершения длительности задержки, как показано на следующей диаграмме:



Корутина приостанавливает (но не блокирует) выполнение после вызова функции `delay()` с нужным значением интервала. По завершении задержки корутина возобновляет выполнение и обновляет значение свойства `currentProgress`.

Запуск гонки

Когда пользователь нажимает кнопку «Старт», необходимо «начать гонку», вызвав функцию `run()` suspend для каждого из двух экземпляров игрока. Для этого нужно запустить корутину для вызова функции `run()`.

Когда вы запускаете корутину для запуска гонки, вам необходимо обеспечить следующие аспекты для обоих участников:

- Они начинают бежать, как только нажимается кнопка «Старт» - то есть запускаются корутины.
- Они приостанавливают или прекращают выполнение при нажатии кнопки Pause или Reset, соответственно, то есть отменяют выполнение корутин.
- Когда пользователь закрывает приложение, отмена управляется должным образом, то есть все корутины отменяются и привязываются к жизненному циклу.
-

В первом уроке про корутины вы узнали, что вызывать приостановленную функцию можно только из другой приостановленной функции. Чтобы безопасно вызывать приостанавливаемые функции из композита, нужно использовать композит `LaunchedEffect()`. Композит `LaunchedEffect()` запускает предоставленную приостанавливающую функцию до тех пор, пока она остается в композиции. С помощью функции `LaunchedEffect()` composable можно выполнить все следующие действия:

- Функция `LaunchedEffect()` composable позволяет безопасно вызывать приостанавливающие функции из композиций.
- Когда функция `LaunchedEffect()` входит в состав Composition, она запускает корутину с блоком кода, переданным в качестве параметра. Он выполняет предоставленную функцию приостановки до тех пор, пока остается в композиции. Когда пользователь нажимает кнопку «Старт» в приложении `RaceTracker`, функция `LaunchedEffect()` входит в композицию и запускает корутину для обновления прогресса. Корутина отменяется, когда `LaunchedEffect()` выходит из композиции. В приложении, если пользователь нажимает кнопку Reset/Pause, LaunchedEffect() удаляется из композиции, а лежащие в ее основе корутины отменяются. Для приложения `RaceTracker` вам не нужно явно предоставлять диспетчера, поскольку об этом позаботится `LaunchedEffect()`.
- Чтобы начать гонку, вызовите функцию `run()` для каждого участника и выполните следующие действия:
- Откройте файл `RaceTrackerApp.kt`, расположенный в пакете `com.example.racetracker.ui`.
- Перейдите к композиту `RaceTrackerApp()` и добавьте вызов композита `LaunchedEffect()` в строку после определения `raceInProgress`.

```
@Composable
fun RaceTrackerApp() {
    ...
    var raceInProgress by remember { mutableStateOf(false) }

    LaunchedEffect {
```

```
    }
    RaceTrackerScreen(...)
}
```

Чтобы гарантировать, что если экземпляры `playerOne` или `playerTwo` будут заменены на другие экземпляры, то `LaunchedEffect()` должен отменить и заново запустить нижележащие корутины, добавьте объекты `playerOne` и `playerTwo` в качестве ключа к `LaunchedEffect`. Подобно тому, как композит Text() перекомпонуется при изменении значения текста, при изменении любого из ключевых аргументов `LaunchedEffect()` базовая программа отменяется и запускается заново.

```
LaunchedEffect(playerOne, playerTwo) {
}
```

- Добавьте вызов функций `playerOne.run()` и `playerTwo.run()`.

```
@Composable
fun RaceTrackerApp() {
    ...
    var raceInProgress by remember { mutableStateOf(false) }

    LaunchedEffect(playerOne, playerTwo) {
        playerOne.run()
        playerTwo.run()
    }
    RaceTrackerScreen(...)
}
```

- Оберните блок `LaunchedEffect()` условием `if`. Начальное значение для этого состояния - false. Значение состояния `raceInProgress` обновляется до true, когда пользователь нажимает кнопку Start и выполняется `LaunchedEffect()`.

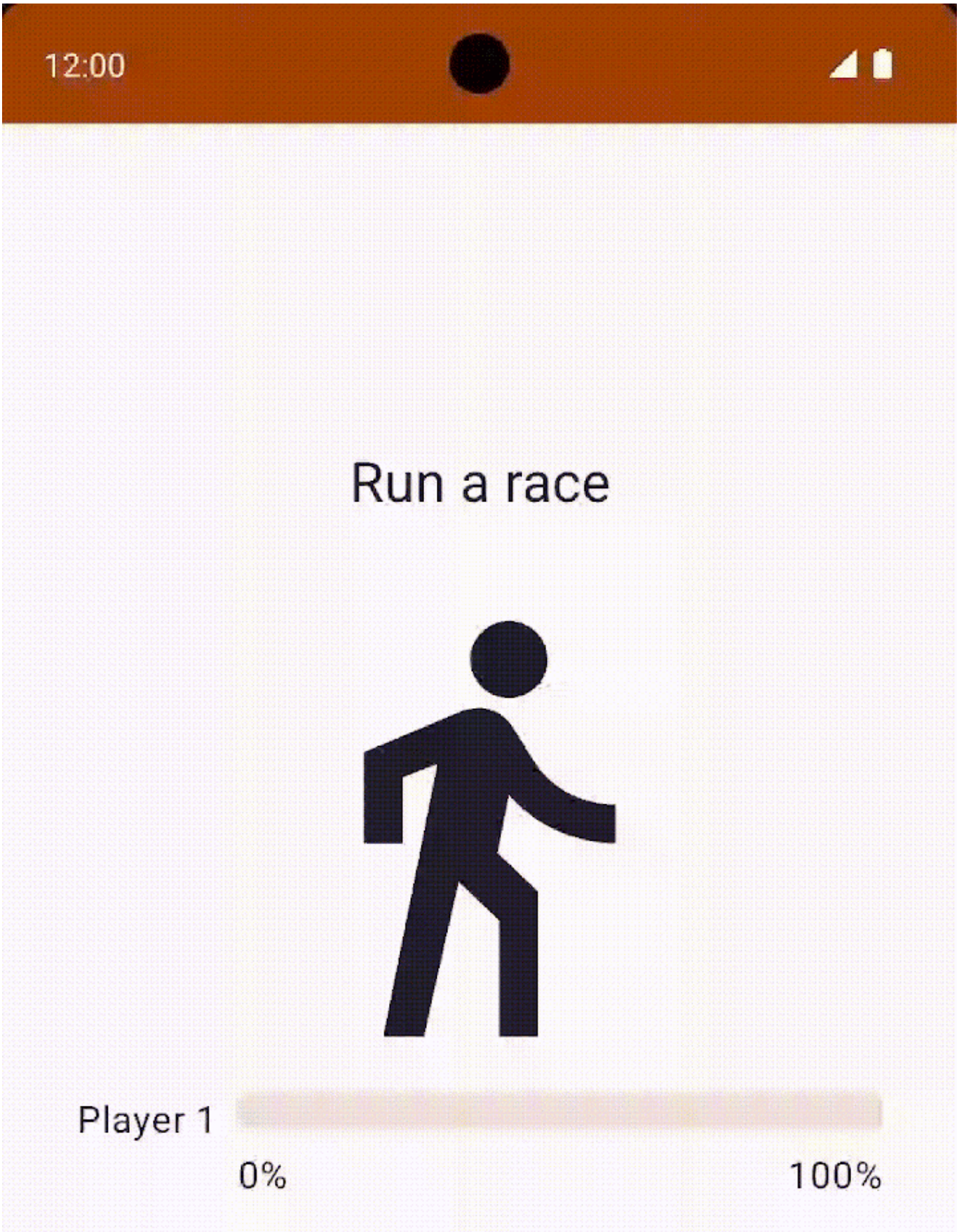
```
if (raceInProgress) {
    LaunchedEffect(playerOne, playerTwo) {
        playerOne.run()
        playerTwo.run()
    }
}
```

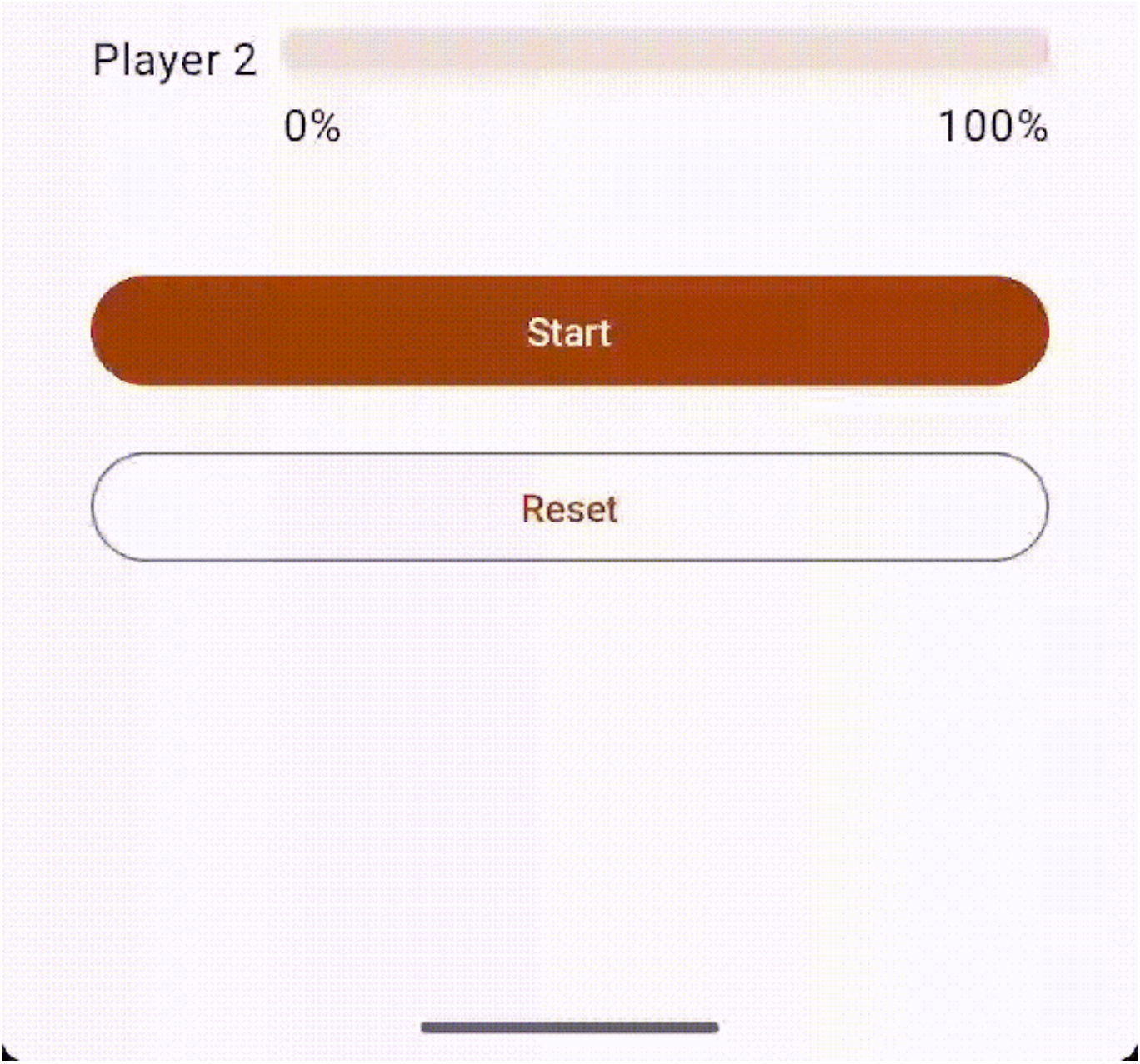
Обновите флаг `raceInProgress` на false, чтобы завершить гонку. Это значение устанавливается в false, когда пользователь также нажимает на Pause. Когда это значение установлено в false, `LaunchedEffect()` гарантирует, что все запущенные корутины будут отменены.

```
LaunchedEffect(playerOne, playerTwo) {
    playerOne.run()
```

```
    playerTwo.run()
    raceInProgress = false
}
```

- Запустите приложение и нажмите кнопку «Старт». Вы должны увидеть, как игрок 1 завершает забег до того, как игрок 2 начинает бежать, как показано на следующем видео:





Это не похоже на честную гонку! В следующем разделе вы узнаете, как запускать параллельные задачи, чтобы оба игрока могли работать одновременно, разберетесь с концепциями и реализуете это поведение.

Структурированный параллелизм

Способ написания кода с использованием coroutines называется **структурированным параллелизмом**. Этот стиль программирования улучшает читаемость и время разработки вашего кода. Идея структурированного параллелизма заключается в том, что у coroutines есть иерархия - задачи могут запускать подзадачи, которые в свою очередь могут запускать подзадачи. Единица этой иерархии называется **областью действия корутины**. Области действия корутин всегда должны быть связаны с жизненным циклом.

API-интерфейсы Coroutines придерживаются этой структурированной схемы параллелизма. Вы не можете вызвать приостановленную функцию из функции, которая не помечена как приостановленная. Это ограничение гарантирует, что вы будете вызывать функции приостановки из конструкторов корутин, таких как `launch`. Эти построители, в свою очередь, привязаны к `CoroutineScope`.

Запуск параллельных задач

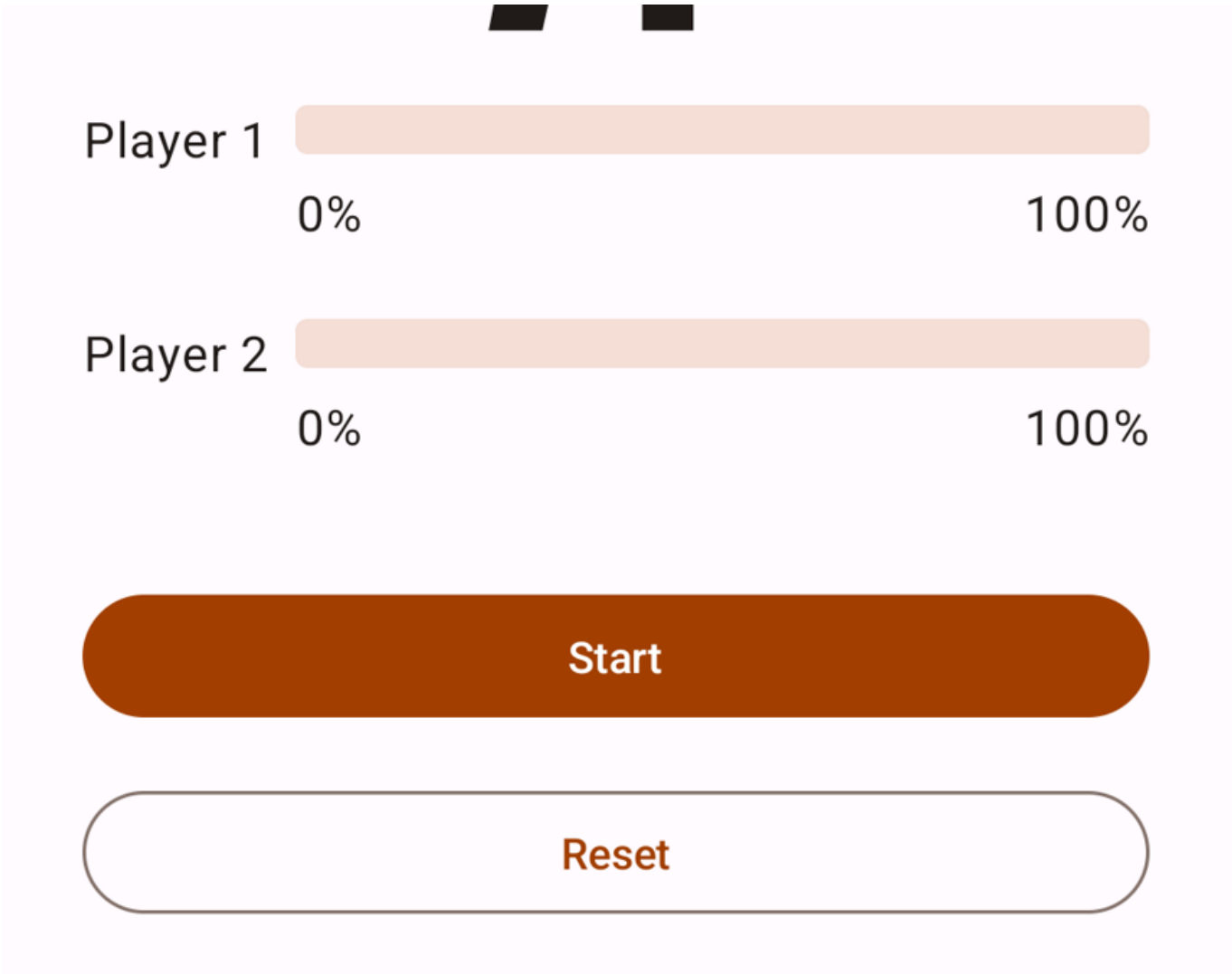
- Чтобы оба участника могли работать одновременно, необходимо запустить две отдельные курутины и переместить каждый вызов функции `run()` внутрь этих курутин. Оберните вызов `playerOne.run()` с помощью конструктора запуска.

```
LaunchedEffect(playerOne, playerTwo) {
    launch { playerOne.run() }
    playerTwo.run()
    raceInProgress = false
}
```

- Аналогичным образом оберните вызов функции `playerTwo.run()` с помощью конструктора запуска. Благодаря этому изменению приложение запускает две корутины, которые выполняются одновременно. Теперь оба игрока могут работать одновременно.

```
LaunchedEffect(playerOne, playerTwo) {
    launch { playerOne.run() }
    launch { playerTwo.run() }
    raceInProgress = false
}
```

- Запустите приложение и нажмите кнопку Start. В то время как вы ожидаете начала гонки, текст кнопки тут же неожиданно меняется на Start.



Когда оба игрока завершат забег, приложение `Race Tracker` должно вернуть текст кнопки «Пауза» на «Старт». Однако сейчас приложение обновляет `raceInProgress` сразу после запуска корутинов, не дожидаясь, пока игроки завершат забег:

```
LaunchedEffect(playerOne, playerTwo) {
    launch {playerOne.run() }
    launch {playerTwo.run() }
    raceInProgress = false // This will update the state immediately, without
waiting for players to finish run() execution.
}
```

Флаг `raceInProgress` обновляется немедленно, потому что:

- Функция построения запуска запускает корутину для выполнения функции `playerOne.run()` и сразу же возвращается для выполнения следующей строки в блоке кода. Такой же поток выполнения происходит со второй функцией построения запуска, которая выполняет функцию `playerTwo.run()`. Как только второй конструктор запуска возвращается, флаг `raceInProgress` обновляется. Это немедленно меняет текст кнопки на Start, и гонка не начинается.

Область действия корутины Функция приостановки `coroutineScope` создает `CoroutineScope` и вызывает указанный блок приостановки с текущей областью видимости. Scope наследует свой

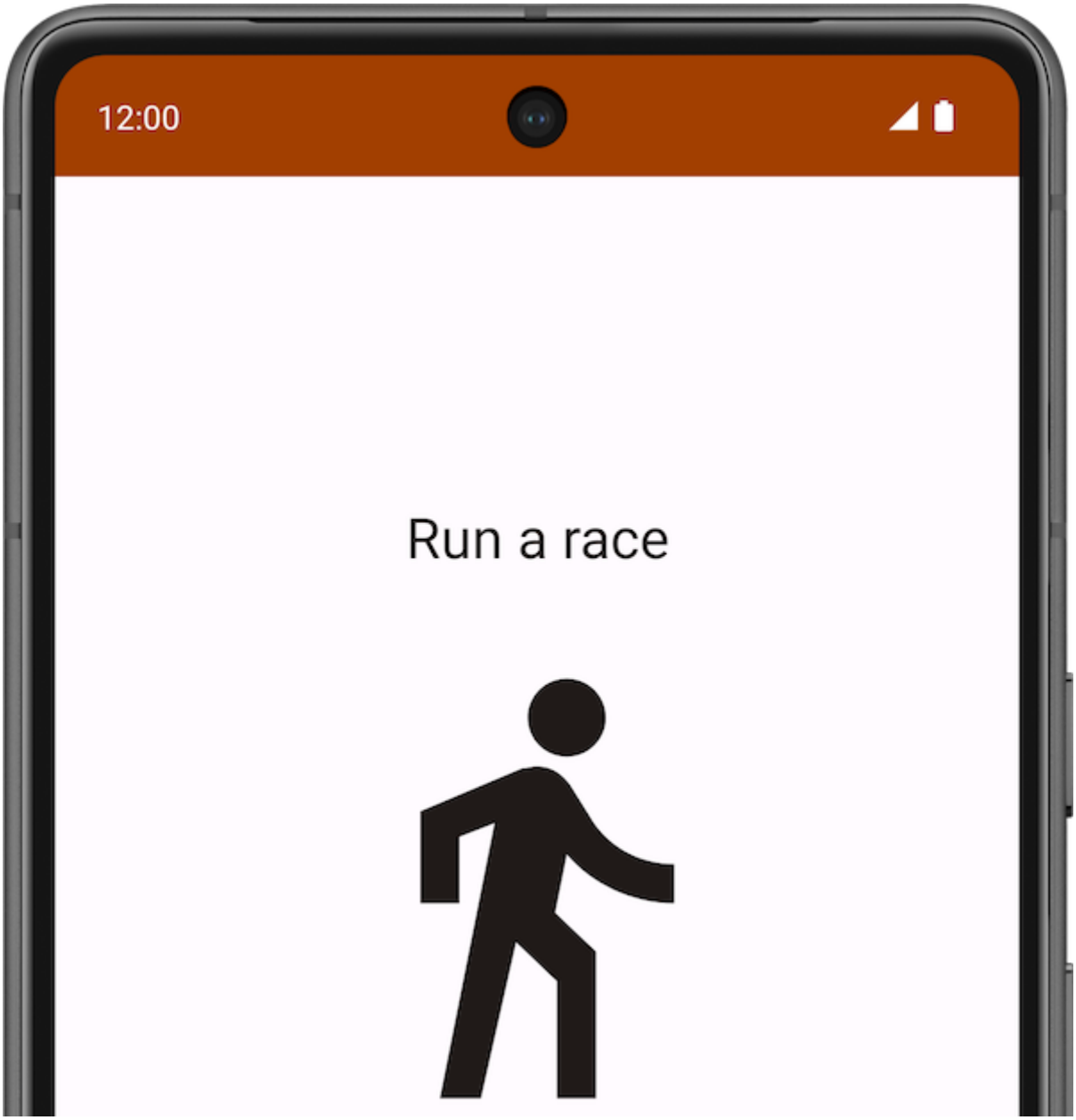
`coroutineContext` от scope `LaunchedEffect()`.

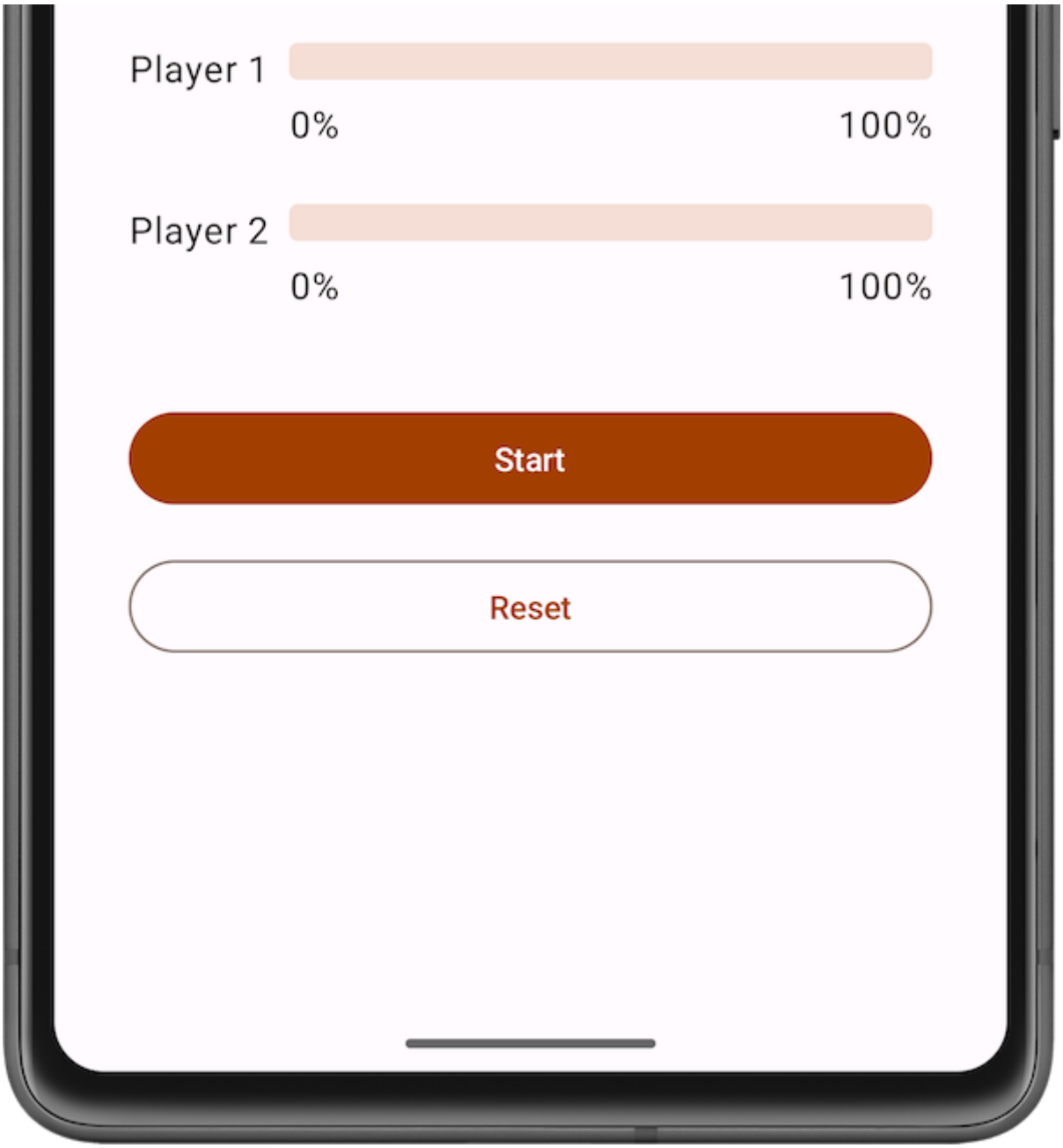
Scope возвращается, как только данный блок и все его дочерние корутины будут завершены. Для приложения `RaceTracker` он возвращается, когда оба объекта-участника завершают выполнение функции `run()`.

- Чтобы убедиться, что функция `run()` для `playerOne` и `playerTwo` завершит выполнение до обновления флага `raceInProgress`, оберните оба конструктора запуска блоком `coroutineScope`.

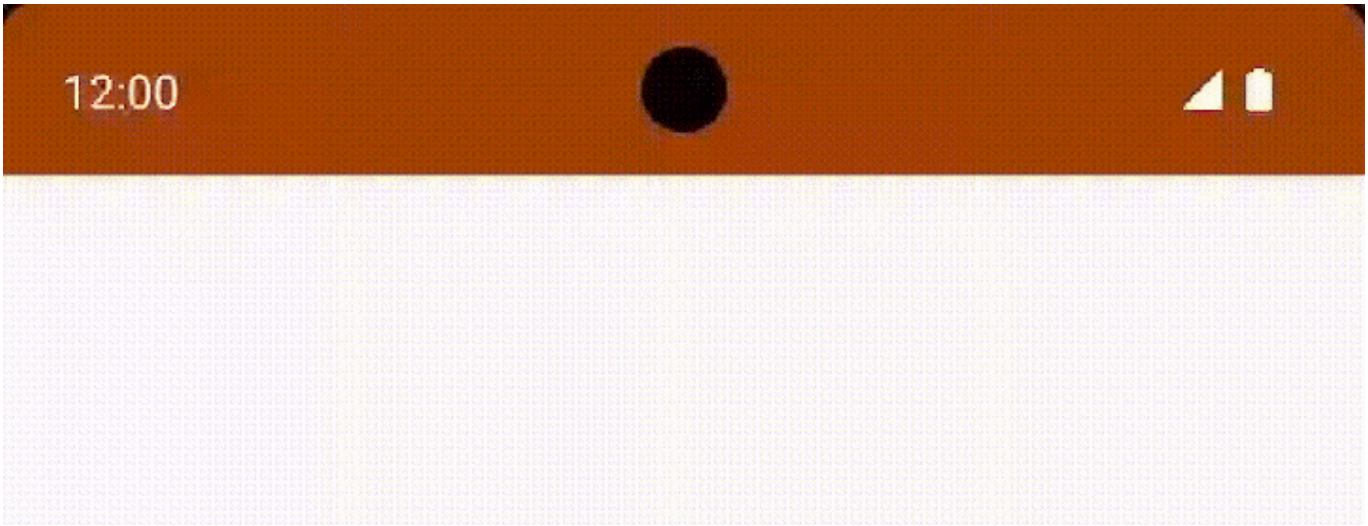
```
LaunchedEffect(playerOne, playerTwo) {
    coroutineScope {
        launch { playerOne.run() }
        launch { playerTwo.run() }
    }
    raceInProgress = false
}
```

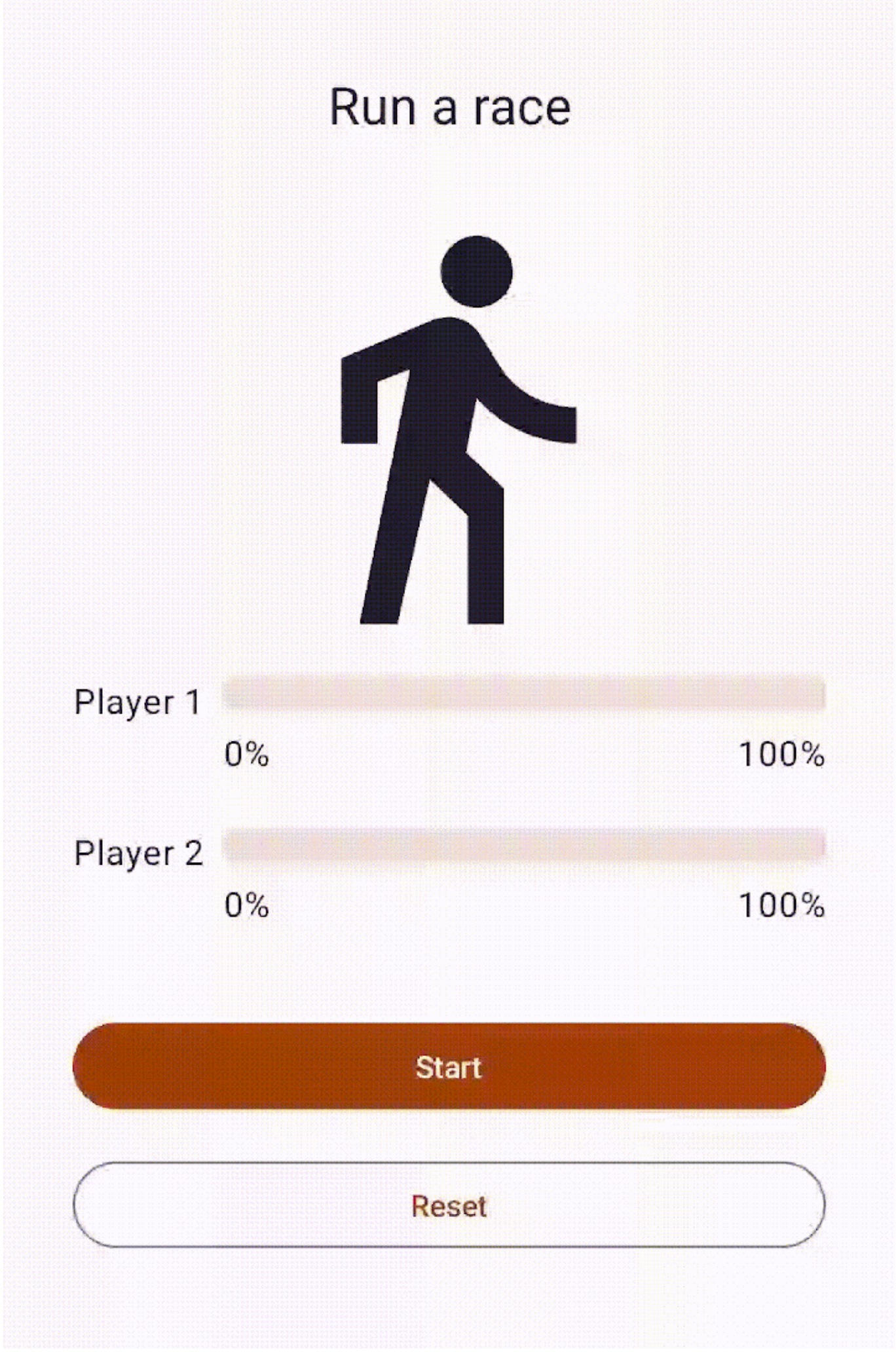
- Запустите приложение на эмуляторе/устройстве Android. Вы должны увидеть следующий экран:



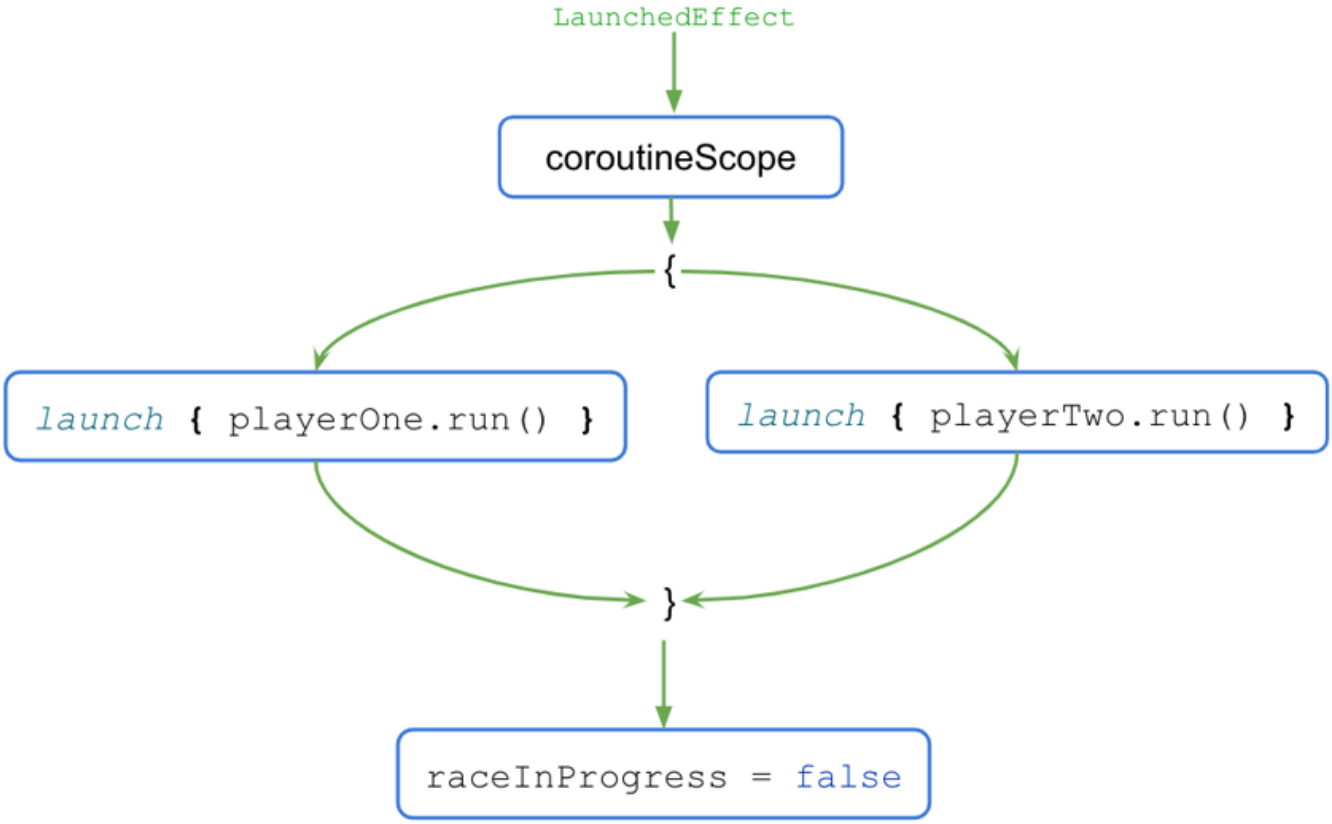


- Нажмите кнопку «Старт». Игрок 2 бежит быстрее, чем Игрок 1. После завершения гонки, когда оба игрока достигнут 100% прогресса, надпись на кнопке «Пауза» изменится на «Старт». Вы можете нажать кнопку Reset, чтобы сбросить забег и заново выполнить симуляцию. Гонка показана на следующем видео.




Run a race
Player 1
0%
100%
Player 2
0%
100%
Start
Reset

Поток выполнения показан на следующей диаграмме:



После выполнения блока `LaunchedEffect()` управление передается блоку `coroutineScope{...}`. Блок `coroutineScope` запускает обе коротины одновременно и ждет, пока они завершат выполнение. После завершения выполнения флаг `raceInProgress` обновляется. Блок `coroutineScope` возвращается и двигается дальше только после того, как весь код внутри блока завершит выполнение. Для кода вне блока наличие или отсутствие параллелизма становится просто деталью реализации. Такой стиль кодирования обеспечивает структурированный подход к параллельному программированию и называется структурированным параллелизмом.

- Когда вы нажимаете кнопку `Reset` после завершения забега, корутины отменяются, а прогресс для обоих игроков обнуляется до 0.

Чтобы увидеть, как отменяются корутины, когда пользователь нажимает кнопку `Reset`, выполните следующие действия:

- Заверните тело метода `run()` в блок `try-catch`, как показано в следующем коде:

```
suspend fun run() {
    try {
        while (currentProgress < maxProgress) {
            delay(progressDelayMillis)
            currentProgress += progressIncrement
        }
    } catch (e: CancellationException) {
        Log.e("RaceParticipant", "$name: ${e.message}")
        throw e // Always re-throw CancellationException.
    }
}
```

- Запустите приложение и нажмите кнопку Start. После нескольких приращений прогресса нажмите кнопку Сброс. Убедитесь, что в Logcat выводится следующее сообщение:

```
Player 1: StandaloneCoroutine was cancelled
Player 2: StandaloneCoroutine was cancelled
```

Напишите модульные тесты для тестирования корутинов

Юнит-тестирование кода, использующего корутины, требует повышенного внимания, поскольку их выполнение может быть асинхронным и происходить в нескольких потоках.

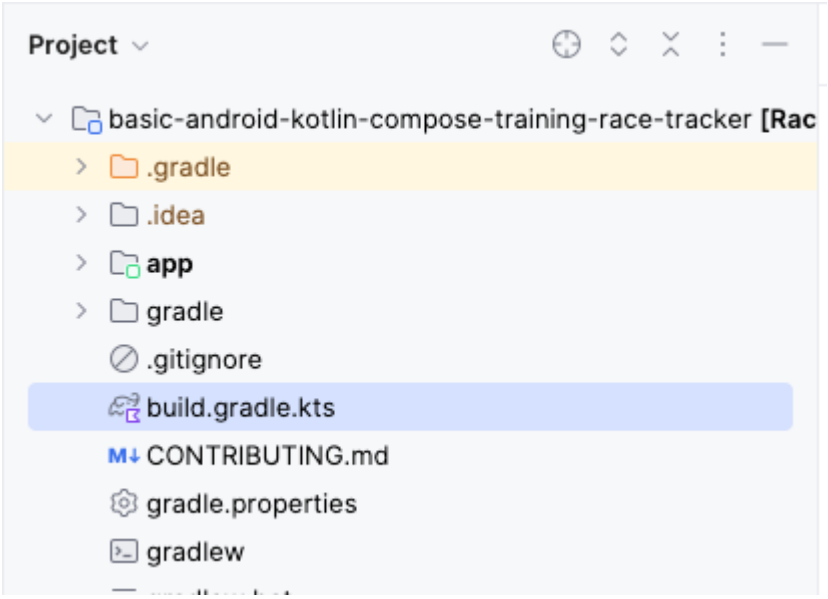
Чтобы вызывать приостанавливающие функции в тестах, необходимо находиться в корутине. Поскольку тестовые функции `JUnit` сами по себе не являются приостанавливающими функциями, вам нужно использовать конструктор корутин `runTest`. Этот конструктор входит в библиотеку `kotlinx-coroutines-test` и предназначен для выполнения тестов. Построитель выполняет тело теста в новой коретине.

> Примечание: Корутины могут быть запущены не только непосредственно в теле теста, но и объектами, используемыми в тесте, с помощью `runTest`.

Поскольку `runTest` является частью библиотеки `kotlinx-coroutines-test`, вам необходимо добавить ее зависимость.

Чтобы добавить зависимость, выполните следующие действия:

- Откройте файл `build.gradle.kts` модуля `app`, расположенный в каталоге app на панели `Project`.



- Внутри файла прокрутите вниз, пока не найдете блок `dependencies{}`.

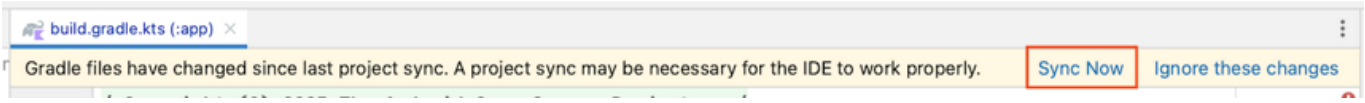
Добавьте зависимость с помощью конфигурации `testImplementation` к библиотеке `kotlinx-coroutines-test`.

```
plugins {
    ...
}

android {
    ...
}

dependencies {
    ...
    testImplementation('org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.4')
}
```

- В панели уведомлений в верхней части файла `build.gradle.kts` нажмите `Sync Now` (Синхронизировать сейчас), чтобы позволить импорту и сборке завершиться, как показано на следующем снимке экрана:



После завершения сборки можно приступать к написанию тестов.

Реализуйте модульные тесты для начала и завершения гонки

Чтобы прогресс гонки корректно обновлялся на разных ее этапах, ваши модульные тесты должны охватывать разные сценарии. В этом примере рассматриваются два сценария:

- Прогресс после начала гонки.
- Прогресс после завершения гонки.

Чтобы проверить, правильно ли обновляется прогресс гонки после ее начала, нужно утверждать, что текущий прогресс устанавливается в 1 после того, как пройдет время `raceParticipant.progressDelayMillis`.

Чтобы реализовать тестовый сценарий, выполните следующие действия:

- Перейдите к файлу `RaceParticipantTest.kt`, расположенному в наборе исходных текстов теста.

Чтобы определить тест, после определения `raceParticipant` создайте функцию `raceParticipant_RaceStarted_ProgressUpdated()` и аннотируйте ее аннотацией `@Test`. Поскольку тестовый блок должен быть помещен в конструктор `runTest`, используйте синтаксис выражения, чтобы вернуть блок `runTest()` в качестве результата теста.

```
class RaceParticipantTest {
    private val raceParticipant = RaceParticipant(
        ...
    )

    @Test
    fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    }
}
```

- Добавьте переменную `expectedProgress`, доступную только для чтения, и установите ее в 1.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
}
```

- Чтобы смоделировать старт гонки, с помощью конструктора запуска запустите новую корутину и вызовите функцию `raceParticipant.run()`.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
}
```

> Примечание: Вы можете напрямую вызвать `raceParticipant.run()` в конструкторе `runtTest`, но реализация теста по умолчанию игнорирует вызов `delay()`. В результате `run()` завершает выполнение до того, как вы сможете проанализировать прогресс.

Значение свойства `raceParticipant.progressDelayMillis` определяет продолжительность обновления прогресса гонки. Чтобы проверить прогресс после истечения времени `progressDelayMillis`, необходимо добавить в тест некоторую форму задержки.

- Используйте вспомогательную функцию `advanceTimeBy()`, чтобы сдвинуть время на величину `raceParticipant.progressDelayMillis`. Функция `advanceTimeBy()` помогает сократить время выполнения теста.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.progressDelayMillis)
}
```

- Поскольку функция `advanceTimeBy()` не запускает задачу, запланированную на заданное время, необходимо вызвать функцию `runCurrent()`. Эта функция выполняет все ожидающие задачи в текущем времени.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.progressDelayMillis)
    runCurrent()
}
```

- Чтобы убедиться, что прогресс обновляется, добавьте вызов функции `assertEquals()`, чтобы проверить, совпадает ли значение свойства `raceParticipant.currentProgress` со значением переменной `expectedProgress`.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.progressDelayMillis)
    runCurrent()
    assertEquals(expectedProgress, raceParticipant.currentProgress)
}
```

- Запустите тест, чтобы убедиться в его прохождении.

Чтобы проверить, правильно ли обновляется прогресс гонки после ее завершения, нужно утверждать, что когда гонка заканчивается, текущий прогресс устанавливается на 100.

- Выполните следующие шаги для реализации теста:

После тестовой функции `raceParticipant_RaceStarted_ProgressUpdated()` создайте функцию `raceParticipant_RaceFinished_ProgressUpdated()` и аннотируйте ее с помощью аннотации `@Test`. Функция должна возвращать результат теста из блока `runTest{}`.

```
class RaceParticipantTest {
    ...

    @Test
    fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
        ...
    }

    @Test
    fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    }
}
```

- С помощью `launch` builder запустите новую coroutine и добавьте в нее вызов функции `raceParticipant.run()`.

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
}
```

- Чтобы имитировать финиш гонки, используйте функцию `advanceTimeBy()`, чтобы продвинуть время диспетчера на `raceParticipant.maxProgress * raceParticipant.progressDelayMillis`:

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.maxProgress *
raceParticipant.progressDelayMillis)
}
```

- Добавьте вызов функции `runCurrent()` для выполнения всех ожидающих заданий.

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.maxProgress *
raceParticipant.progressDelayMillis)
    runCurrent()
}
```

- Чтобы убедиться, что прогресс обновляется корректно, добавьте вызов функции `assertEquals()`, чтобы проверить, равно ли значение свойства `raceParticipant.currentProgress` 100.

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.maxProgress *
raceParticipant.progressDelayMillis)
    runCurrent()
    assertEquals(100, raceParticipant.currentProgress)
}
```

- Запустите тест, чтобы убедиться, что он пройден.

Попробуйте решить эту задачу Примените стратегии тестирования, рассмотренные в практической работе «Написание модульных тестов для ViewModel». Добавьте тесты, чтобы охватить счастливый путь, случаи ошибок и пограничные случаи.

Сравните написанные вами тесты с теми, что есть в коде решения.

Получение кода решения

```
git clone https://github.com/google-developer-training/basic-android-kotlin-
compose-training-race-tracker.git
cd basic-android-kotlin-compose-training-race-tracker
```

Заключение

Вы только что узнали, как использовать корутины для обработки параллелизма. Корутины помогают управлять длительными задачами, которые в противном случае могут заблокировать основной поток и привести к тому, что ваше приложение перестанет реагировать на запросы. Вы также узнали, как писать модульные тесты для проверки корутинов.

К преимуществам корутин можно отнести следующие особенности:

- Удобство чтения: Код, который вы пишете с помощью coroutines, обеспечивает четкое понимание последовательности выполнения строк кода.
- Интеграция с Jetpack: Многие библиотеки Jetpack, такие как Compose и ViewModel, содержат расширения, обеспечивающие полную поддержку корутинов. Некоторые библиотеки также предоставляют собственную область видимости корутинов, которую можно использовать для структурированного параллелизма.
- Структурированный параллелизм: Корутины делают одновременный код безопасным и простым в реализации, избавляют от ненужного шаблонного кода и гарантируют, что запущенные приложением корутины не будут потеряны или продолжат тратить ресурсы.

Резюме

- Корутины позволяют писать долго выполняющийся код, который работает параллельно, не изучая новый стиль программирования. По своей конструкции выполнение корутин является последовательным.

- Ключевое слово `suspend` используется для обозначения функции или типа функции, чтобы указать на ее доступность для выполнения, приостановки и возобновления набора инструкций кода.
- Приостановленная функция может быть вызвана только из другой приостановленной функции.
- Запустить новую корутину можно с помощью функции `launch` или `async builder`.
- `Контекст корутины`, `построители корутин`, `job`, `область видимости корутины` и `диспетчер` - основные компоненты для реализации корутин.
- Корутины используют диспетчеры для определения потоков, которые необходимо использовать для их выполнения.
- `Job` играет важную роль в обеспечении структурированного параллелизма, управляя жизненным циклом короутинов и поддерживая отношения «родитель-ребенок».
- `CoroutineContext` определяет поведение корутины с помощью `Job` и диспетчера корутин.
- `CoroutineScope` контролирует время жизни корутинов с помощью своего задания и рекурсивно применяет правила отмены и другие правила к своим дочерним элементам и их дочерним элементам.
- Запуск, завершение, отмена и отказ - это четыре общие операции в процессе выполнения корутины.
- Корутины следуют принципу структурированного параллелизма.