# Тема: Введение в корутины

## Прежде чем начать

Этот урок познакомит вас с параллелизмом, который является критически важным навыком для разработчиков Android, чтобы обеспечить отличный пользовательский опыт. Одновременность подразумевает выполнение нескольких задач в вашем приложении одновременно. Например, ваше приложение может получать данные с веб-сервера или сохранять пользовательские данные на устройстве, одновременно реагируя на события пользовательского ввода и соответствующим образом обновляя пользовательский интерфейс.

Для одновременного выполнения работы в вашем приложении вы будете использовать **корутины** Kotlin. **Корутины** позволяют приостановить выполнение блока кода и возобновить его позже, чтобы в это время можно было выполнить другую работу. Корутины облегчают написание асинхронного кода, что означает, что одна задача не должна полностью завершаться перед запуском следующей задачи, что позволяет выполнять несколько задач одновременно.

В этом уроке вы рассмотрите несколько базовых примеров в Idea, где вы получите практические навыки работы с корутинами, чтобы лучше освоить асинхронное программирование.

## Необходимые условия

- Уметь создать базовую программу на Kotlin с функцией main()
- Знание основ языка Kotlin, включая функции и лямбды

## Что вы создадите

- Короткая программа на Kotlin для изучения и экспериментирования с основами корутинов

## Что вы узнаете

- Как корутины Kotlin могут упростить асинхронное программирование
- Назначение структурированного параллелизма и почему он имеет значение

## Синхронный код

Простая программа В синхронном коде одновременно выполняется только одна концептуальная задача. Вы можете представить это как последовательный линейный путь. Одна задача должна полностью завершиться, прежде чем будет начата следующая. Ниже приведен пример синхронного кода.

- Откройте Idea.
- Замените код на следующий код для программы, которая показывает прогноз погоды на солнечную погоду. В функции `main()` сначала выводим текст: Прогноз погоды. Затем выводим: Солнечно.

```
fun main() {
    println("Weather forecast")
```

```
    println("Sunny")
}
```

- Запустите код. Результат выполнения приведенного выше кода должен быть следующим:

```
Weather forecast
Sunny
```

`println()` - это синхронный вызов, потому что задача печати текста на выходе завершается до того, как выполнение перейдет к следующей строке кода. Поскольку каждый вызов функции в `main()` является синхронным, вся функция `main()` является **синхронной**. Является ли функция синхронной или асинхронной, определяется тем, из каких частей она состоит.

Синхронная функция возвращается только тогда, когда ее задача полностью выполнена. Поэтому после выполнения последнего оператора print в `main()` вся работа завершена. Функция main() возвращается, и программа завершается.

**Добавьте задержку**.

Теперь представим, что для получения прогноза солнечной погоды требуется сетевой запрос к удаленному веб-серверу.

- Имитируйте сетевой запрос, добавив в код задержку перед выводом сообщения о том, что прогноз погоды солнечный.

> Установите пакет implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")

Во-первых, добавьте `import kotlinx.coroutines.*` в верхней части кода перед функцией `main()`. Это импортирует функции, которые вы будете использовать, из библиотеки Kotlin coroutines. Измените свой код, добавив вызов `delay(1000)`, который задерживает выполнение оставшейся части функции `main()` на 1000 миллисекунд, или 1 секунду. Вставьте этот вызов `delay()` перед оператором `print` для Sunny.

```
import kotlinx.coroutines.*

fun main() {
    println("Weather forecast")
    delay(1000)
    println("Sunny")
}
```

- delay() - это специальная функция приостановки, предоставляемая библиотекой Kotlin coroutines. Выполнение функции `main()` приостановится в этот момент, а затем возобновится по истечении указанной продолжительности задержки (в данном случае - одна секунда).

Если вы попытаетесь запустить свою программу в этот момент, произойдет ошибка компиляции: Функция приостановки 'delay' должна вызываться только из корутины или другой функции приостановки.

Для изучения корутинов в Kotlin вы можете обернуть существующий код вызовом функции `runBlocking()` из библиотеки coroutines. runBlocking() запускает цикл событий, который может обрабатывать несколько задач одновременно, продолжая выполнение каждой задачи с того места, где она остановилась, когда она будет готова к возобновлению.

Переместите существующее содержимое функции `main()` в тело вызова `runBlocking {}`. Тело runBlocking{} выполняется в новой корутине.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        delay(1000)
        println("Sunny")
    }
}
```

`runBlocking()` является **синхронным**; он не вернется, пока не завершится вся работа внутри его лямбда-блока. Это означает, что она будет ждать завершения работы в вызове `delay()` (пока не пройдет одна секунда), а затем продолжит выполнение оператора `print("Sunny")`. Когда вся работа в функции `runBlocking()` завершена, функция возвращается, что завершает работу программы.

- Запустите программу. Вот вывод:

```
Weather forecast
Sunny
```

На выходе получаем то же самое, что и раньше. Код по-прежнему синхронный - он выполняется по прямой линии и делает только одно действие за раз. Однако разница теперь в том, что он выполняется в течение более длительного периода времени благодаря задержке.

Буква «co-» в слове **co**routine означает кооператив. Код сотрудничает для совместного использования основного цикла событий, когда он приостанавливается для ожидания чего-то, что позволяет выполнять другую работу в это время. (Частица «-**routine**» в слове «coroutine» означает набор инструкций, подобный функции). В данном примере coroutine приостанавливается, когда достигает вызова `delay()`. За ту секунду, когда приостанавливается работа, можно выполнить другую работу (хотя в данной программе никакой другой работы нет). Как только длительность задержки истечет, корутина возобновит выполнение и сможет приступить к печати Sunny на вывод.

> Примечание: В общем случае используйте `runBlocking()` внутри функции `main()`, подобной этой, только в учебных целях. В коде приложений Android функция `runBlocking()` не нужна, поскольку Android предоставляет цикл событий для обработки возобновленной работы приложения, когда оно становится готовым. Однако `runBlocking()` может быть полезна в тестах, позволяя тестам ожидать определенных условий в приложении перед вызовом тестовых утверждений.

**Отложенные функции suspend**.

Если фактическая логика выполнения сетевого запроса для получения данных о погоде становится более сложной, вы можете захотеть вынести ее в отдельную функцию. Давайте рефакторим код, чтобы увидеть эффект.

- Извлеките код, имитирующий сетевой запрос на получение данных о погоде, и перенесите его в собственную функцию `printForecast()`. Вызовите `printForecast()` из кода `runBlocking()`.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
    }
}

fun printForecast() {
    delay(1000)
    println("Sunny")
}
```

Если вы запустите программу сейчас, то увидите ту же ошибку компиляции, что и раньше.

> Приостанавливающая функция может быть вызвана только из корутины или другой приостанавливающей функции, поэтому определите `printForecast()` как приостанавливающую функцию.

- Добавьте модификатор **suspend** непосредственно перед ключевым словом `fun` в объявлении функции `printForecast()`, чтобы сделать ее приостанавливающей функцией.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}
```

Помните, что `delay()` - это приостанавливающая функция, а теперь вы сделали `printForecast()` тоже приостанавливающей функцией.

Приостанавливающая функция похожа на обычную функцию, но ее можно приостановить и возобновить позже. Для этого приостанавливающие функции можно вызывать только из других приостанавливающих функций, которые предоставляют такую возможность.

Приостанавливающая функция может содержать ноль или более точек приостановки. **Точка приостановки** - это место внутри функции, где выполнение функции может быть приостановлено. Как только выполнение возобновится, она подхватит то место в коде, на котором остановилась в последний раз, и продолжит работу с остальной частью функции.

Потренируйтесь, добавив в код еще одну приостанавливающую функцию, расположенную ниже объявления функции `printForecast()`. Назовите эту новую приостанавливающую функцию `printTemperature()`. Вы можете сделать вид, что это сетевой запрос на получение данных о температуре для прогноза погоды.

- Внутри функции также отложите выполнение на 1000 миллисекунд, а затем напечатайте на выходе значение температуры, например 30 градусов Цельсия. Для вывода символа градуса, °, можно использовать управляющую последовательность «`\u00b0`».

```
suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

- Вызовите новую функцию `printTemperature()` из кода `runBlocking()` в функции `main()`.

Вот полный код:

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
        printTemperature()
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

- Запустите программу. На выходе должно получиться:

```
Прогноз погоды
Солнечно
30°C
```

В этом коде корутина сначала приостанавливается с задержкой в функции `printForecast()` suspend, а затем возобновляется после этой односекундной задержки. Текст Sunny печатается на выходе. Функция `printForecast()` возвращается обратно к вызывающей стороне.

Далее вызывается функция `printTemperature()`. Эта корутина приостанавливается, когда достигает вызова `delay()`, а затем возобновляется через одну секунду и завершает печать значения температуры на выходе. Функция `printTemperature()` завершила всю работу и возвращается.

В теле `runBlocking()` больше нет задач для выполнения, поэтому функция `runBlocking()` возвращается, и программа завершается.

Как упоминалось ранее, `runBlocking()` является синхронной, и каждый вызов в теле будет вызываться последовательно. Обратите внимание, что хорошо спроектированная приостанавливающая функция возвращается только после того, как вся работа будет завершена. В результате эти приостанавливающие функции выполняются одна за другой.

(Необязательно) Если вы хотите посмотреть, сколько времени занимает выполнение этой программы с задержками, то можете обернуть свой код в вызов `measureTimeMillis()`, который вернет время в миллисекундах, необходимое для выполнения переданного блока кода. Добавьте оператор импорта (`import kotlin.system.*`), чтобы получить доступ к этой функции. Выведите время выполнения и разделите на 1000.0, чтобы преобразовать миллисекунды в секунды.

```
import kotlin.system.*
import kotlinx.coroutines.*

fun main() {
    val time = measureTimeMillis {
        runBlocking {
            println("Weather forecast")
            printForecast()
            printTemperature()
        }
    }
    println("Execution time: ${time / 1000.0} seconds")
}
suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

Выход:

```
Weather forecast
Sunny
30°C
Execution time: 2.128 seconds
```

Вывод показывает, что выполнение заняло ~ 2,1 секунды. (Точное время выполнения может быть немного другим) Это кажется разумным, потому что каждая из приостанавливающих функций имеет задержку в одну секунду.

До сих пор вы видели, что код в `coroutine` по умолчанию вызывается последовательно. Если вы хотите, чтобы все выполнялось параллельно, вам придется явно указать на это, и в следующем разделе вы узнаете, как это сделать. Вы воспользуетесь `coroutine` циклом событий для одновременного выполнения нескольких задач, что ускорит время выполнения программы.

## Асинхронный код

### launch()

Используйте функцию `launch()` из библиотеки `coroutines`, чтобы запустить новую coroutine. Для одновременного выполнения задач добавьте в код несколько функций `launch()`, чтобы несколько coroutines могли выполняться одновременно.

Корутины в Kotlin следуют ключевой концепции, называемой **структурированным параллелизмом**, когда ваш код по умолчанию является последовательным и взаимодействует с базовым циклом событий, если вы явно не попросите о параллельном выполнении (например, с помощью `launch()`). Предполагается, что если вы вызываете функцию, она должна полностью завершить свою работу к моменту возвращения, независимо от того, сколько корутинов она могла использовать в деталях своей реализации. Даже если она завершает работу с исключением, после того как исключение выброшено, больше нет никаких ожидающих заданий от функции. Следовательно, вся работа завершается, как только поток управления возвращается из функции, независимо от того, выбросила ли она исключение или успешно завершила свою работу.

- Начните с кода из предыдущих шагов. Используйте функцию `launch()`, чтобы переместить каждый вызов `printForecast()` и `printTemperature()` соответственно в свои собственные корутины.

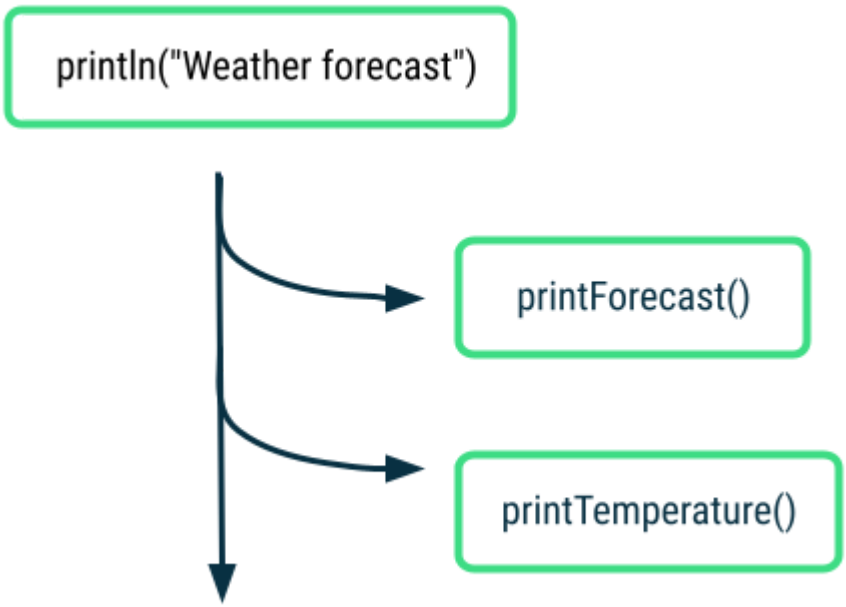
```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
        launch {
            printTemperature()
```

```
        }
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

- Запустите программу. Вот вывод:

```
Weather forecast
Sunny
30°C
```

Результат тот же, но вы, возможно, заметили, что программа стала выполняться быстрее. Раньше нужно было дождаться полного завершения приостановленной функции `printForecast()`, прежде чем переходить к функции `printTemperature()`. Теперь функции `printForecast()` и `printTemperature()` могут выполняться одновременно, поскольку они находятся в отдельных корутинах.




Вызов запуска { printForecast() } может вернуться до того, как вся работа в printForecast() будет завершена. В этом и заключается прелесть корутинов. Вы можете перейти к следующему вызову `launch()`, чтобы запустить следующую корутину. Аналогично, запуск { printTemperature() } также возвращается еще до завершения всей работы.

(Необязательно) Если вы хотите посмотреть, насколько быстрее стала работать программа, вы можете добавить код `measureTimeMillis()`, чтобы проверить время выполнения.

```
import kotlin.system.*
import kotlinx.coroutines.*

fun main() {
    val time = measureTimeMillis {
        runBlocking {
            println("Weather forecast")
            launch {
                printForecast()
            }
            launch {
                printTemperature()
            }
        }
    }
    println("Execution time: ${time / 1000.0} seconds")
}

...
```

Выходные данные:

```
Прогноз погоды
Солнечно
30°C
Время выполнения: 1.122 секунды
```

Вы видите, что время выполнения уменьшилось с ~ 2,1 секунды до ~ 1,1 секунды, так что после добавления параллельных операций программа стала выполняться быстрее! Вы можете удалить этот код измерения времени, прежде чем переходить к следующим шагам.

Как вы думаете, что произойдет, если добавить еще один оператор печати после второго вызова launch(), до конца кода `runBlocking()`? Где появится это сообщение в выводе?

- Измените код `runBlocking()`, чтобы добавить дополнительный оператор печати до конца этого блока.

```
...

fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
```

```
        launch {
            printTemperature()
        }
        println("Have a good day!")
    }
}

...
```

- Запустите программу, и вот результат:

```
Weather forecast
Have a good day!
Sunny
30°C
```

Из этого вывода видно, что после запуска двух новых coroutines для `printForecast()` и `printTemperature()` можно переходить к следующей инструкции, которая печатает `Have a good day!`. Это демонстрирует природу `launch()` по принципу «запустить и забыть». Вы запускаете новую корутину с помощью `launch()`, и вам не нужно беспокоиться о том, когда ее работа будет завершена.

Позже корутины завершат свою работу и выведут оставшиеся сообщения. Как только вся работа (включая все корутины) в теле вызова `runBlocking()` будет завершена, runBlocking() вернется, и программа завершится.

Теперь вы превратили синхронный код в асинхронный. Когда асинхронная функция возвращается, задача может быть еще не завершена. Именно это вы видели в случае с `launch()`. Функция вернулась, но ее работа еще не была завершена. Благодаря использованию `launch()` в вашем коде могут одновременно выполняться несколько задач, что является мощной возможностью для использования в разрабатываемых вами приложениях для Android.

**async()**

В реальном мире вы не можете знать, сколько времени займут сетевые запросы на прогноз и температуру. Если вы хотите вывести единый прогноз погоды, когда обе задачи будут выполнены, то текущего подхода с `launch()` будет недостаточно. Вот тут-то и приходит на помощь `async()`.

Используйте функцию `async()` из библиотеки coroutines, если вам небезразлично, когда завершится работа coroutine, и вам нужно вернуть значение.

Функция `async()` возвращает объект типа `Deferred`, который как бы обещает, что результат будет в нем, когда он будет готов. Вы можете получить доступ к результату на объекте `Deferred` с помощью `await()`.

- Сначала измените функции suspend, чтобы они возвращали строку вместо печати данных о прогнозе и температуре. Измените названия функций `printForecast()` и `printTemperature()` на `getForecast()` и `getTemperature()`.

```
...

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

- Измените код `runBlocking()` так, чтобы он использовал `async()` вместо `launch()` для двух coroutines. Храните возвращаемое значение каждого вызова `async()` в переменных `forecast` и `temperature`, которые являются объектами `Deferred`, хранящими результат типа String. (Указание типа необязательно, поскольку в Kotlin используется вывод типов, но оно включено ниже, чтобы вы могли лучше понять, что возвращается в результате вызовов `async()`).

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        val forecast: Deferred<String> = async {
            getForecast()
        }
        val temperature: Deferred<String> = async {
            getTemperature()
        }
        ...
    }
}

...
```

Позже в корутине, после двух вызовов `async()`, вы можете получить доступ к результату этих корутин, вызвав `await()` на объектах `Deferred`. В этом случае вы можете вывести значение каждой программы с помощью `forecast.await()` и `temperature.await()`.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        val forecast: Deferred<String> = async {
            getForecast()
        }
        val temperature: Deferred<String> = async {
```

```
            getTemperature()
        }
        println("${forecast.await()} ${temperature.await()}")
        println("Have a good day!")
    }
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

- Запустите программу и получите результат:

```
Weather forecast
Sunny 30°C
Have a good day!
```

> Замечательно! Вы создали две параллельно выполняющиеся корутины для получения данных о прогнозе и температуре. После завершения каждая из них возвращала значение. Затем вы объединили два возвращаемых значения в один оператор печати: Sunny 30°C.

## Параллельная декомпозиция

Мы можем пойти еще дальше и посмотреть, как coroutines могут быть полезны при параллельной декомпозиции работы. **Параллельная декомпозиция** предполагает разбиение задачи на более мелкие подзадачи, которые можно решать параллельно. Когда результаты подзадач готовы, вы можете объединить их в конечный результат.

В своем коде извлеките логику составления прогноза погоды из тела `runBlocking()` в одну функцию `getWeatherReport()`, которая возвращает комбинированную строку Sunny 30°C.

- Определите в коде новую suspend функцию `getWeatherReport()`.
- Установите эту функцию равной результату вызова функции `coroutineScope{}` с пустым лямбда-блоком, который в итоге будет содержать логику для получения прогноза погоды.

```
...

suspend fun getWeatherReport() = coroutineScope {

}

...
```

`coroutineScope{}` создает локальную область видимости для этой задачи прогноза погоды. Коротины, запущенные в этой области, группируются в ней, что имеет последствия для отмены и исключений, о которых вы скоро узнаете.

В теле функции `coroutineScope()` создайте две новые функции с помощью `async()` для получения данных о прогнозе и температуре, соответственно. Создайте строку с прогнозом погоды, объединив результаты этих двух программ. Для этого вызовите `await()` на каждом из объектов `Deferred`, возвращенных вызовами `async()`. Это гарантирует, что каждая из программ завершит свою работу и вернет результат до того, как мы вернемся из этой функции.

```
...

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

...
```

- Вызовите эту новую функцию `getWeatherReport()` из `runBlocking()`. Вот полный код:

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

- Запустите программу и увидите следующее сообщение:

```
Weather forecast
Sunny 30°C
Have a good day!
```

Результат тот же, но есть несколько интересных моментов. Как уже упоминалось, функция `coroutineScope()` вернется только после того, как вся ее работа, включая запущенные ею корутины, будет завершена. В данном случае обе корутины `getForecast()` и `getTemperature()` должны завершиться и вернуть соответствующие результаты. Затем текст Sunny и 30°C объединяются и возвращаются из области видимости. Сообщение о погоде с температурой 30°C будет выведено на экран, и вызывающая программа сможет перейти к последнему оператору печати «Хорошего дня!».

При использовании функции `coroutineScope()`, несмотря на то что внутри функция выполняет работу параллельно, для вызывающей стороны она выглядит как синхронная операция, поскольку `coroutineScope` не вернется, пока не будет выполнена вся работа.

> Ключевой момент для структурированного параллелизма заключается в том, что вы можете взять несколько параллельных операций и поместить их в одну синхронную операцию, где параллелизм - это деталь реализации. Единственное требование к вызывающему коду - находиться в приостанавливающей функции или корутине. В остальном структура вызывающего кода не должна учитывать детали параллелизма.

## Исключения и отмена

Теперь поговорим о ситуациях, когда может возникнуть ошибка или отмениться работа.

### Введение в исключения

**Exception** - это неожиданное событие, которое происходит во время выполнения вашего кода. Вы должны реализовать соответствующие способы обработки этих исключений, чтобы предотвратить аварийное завершение работы вашего приложения и негативное влияние на работу пользователей.

Вот пример программы, которая завершается раньше времени из-за исключения. Программа предназначена для вычисления количества пицц, которые получит каждый человек, путем деления `NumberOfPizzas / NumberOfPeople`. Допустим, вы случайно забыли установить значение `numberOfPeople` в фактическое значение.

```
fun main() {
    val numberOfPeople = 0
    val numberOfPizzas = 20
    println("Slices per person: ${numberOfPizzas / numberOfPeople}")
}
```

Когда вы запустите программу, она завершится с арифметическим исключением, потому что вы не можете разделить число на ноль.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at FileKt.main (File.kt:4)
 at FileKt.main (File.kt:-1)
 at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0 (:-2)
```

Эта проблема имеет простое решение: вы можете изменить начальное значение `NumberOfPeople` на ненулевое. Однако по мере усложнения кода возникают случаи, когда невозможно предусмотреть и предотвратить все исключения.

Что произойдет, если одна из ваших корутин завершится с исключением? Измените код программы погоды, чтобы узнать это.

**Исключения в корутинах**

Начните с программы погоды из предыдущего раздела.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

В одной из приостанавливающих функций намеренно бросьте исключение, чтобы посмотреть, какой будет эффект. Это имитирует непредвиденную ошибку при получении данных с сервера, что вполне правдоподобно.

В функцию `getTemperature()` добавьте строку кода, которая выбрасывает исключение. Напишите выражение `throw`, используя ключевое слово `throw` в Kotlin, а затем новый экземпляр исключения,

которое расширяется от `Throwable`. Например, вы можете бросить `AssertionError` и передать строку сообщения, которая описывает ошибку более подробно: `throw AssertionError(«Temperature is invalid»)`. Выброс этого исключения останавливает дальнейшее выполнение функции `getTemperature()`.

```
...
suspend fun getTemperature(): String {
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

Вы также можете изменить задержку на 500 миллисекунд для метода `getTemperature()`, чтобы знать, что исключение произойдет до того, как другая функция `getForecast()` сможет завершить свою работу.

- Запустите программу, чтобы увидеть результат.

```
Weather forecast
Exception in thread "main" java.lang.AssertionError: Temperature is invalid
 at FileKt.getTemperature (File.kt:24)
 at FileKt$getTemperature$1.invokeSuspend (File.kt:-1)
 at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith
(ContinuationImpl.kt:33)
```

Чтобы понять это поведение, вам нужно знать, что между корутинами существуют отношения «родитель-ребенок». Вы можете запустить корутину (так называемую дочернюю) из другой корутины (родительской). По мере того как вы будете запускать другие корутины от этих корутин, вы сможете построить целую иерархию корутин.

Корутина, выполняющая `getTemperature()`, и корутина, выполняющая `getForecast()`, являются дочерними корутинами одной и той же родительской корутины. Поведение, которое вы наблюдаете с исключениями в корутинах, связано со структурированным параллелизмом. Когда одна из дочерних программ терпит неудачу с исключением, оно передается наверх. Родительская корутина отменяется, что, в свою очередь, отменяет все остальные дочерние корутины (например, корутину, выполняющую `getForecast()` в данном случае). Наконец, ошибка распространяется вверх, и программа завершается с ошибкой `AssertionError`.

## Исключения с функцией Try-catch

Если вы знаете, что некоторые части вашего кода могут вызвать исключение, то вы можете окружить этот код блоком `try-catch`. Вы можете поймать исключение и обработать его более изящно в своем приложении, например, показав пользователю полезное сообщение об ошибке. Вот фрагмент кода, в котором показано, как это может выглядеть:

```
try {
    // Some code that may throw an exception
} catch (e: IllegalArgumentException) {
    // Handle exception
}
```

Этот подход работает и для асинхронного кода с короутинами. Вы все еще можете использовать выражение `try-catch` для перехвата и обработки исключений в коротинах. Причина в том, что при структурированном параллелизме последовательный код все равно остается синхронным кодом, поэтому блок `try-catch` будет работать так же, как и ожидалось.

```
...

fun main() {
    runBlocking {
        ...
        try {
            ...
            throw IllegalArgumentException("No city selected")
            ...
        } catch (e: IllegalArgumentException) {
            println("Caught exception $e")
            // Handle error
        }
    }
}

...
```

Чтобы лучше освоить работу с исключениями, измените программу погоды, чтобы перехватить исключение, которое вы добавили ранее, и вывести его на экран.

В функции `runBlocking()` добавьте блок `try-catch` вокруг кода, вызывающего `getWeatherReport()`. Выведите пойманную ошибку, а также сообщение о том, что прогноз погоды недоступен.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        try {
            println(getWeatherReport())
        } catch (e: AssertionError) {
            println("Caught exception in runBlocking(): $e")
            println("Report unavailable at this time")
        }
        println("Have a good day!")
    }
```

```
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

- Запустите программу, и теперь ошибка будет обработана изящно, а программа сможет успешно завершить выполнение.

```
Weather forecast
Caught exception in runBlocking(): java.lang.AssertionError: Temperature is
invalid
Report unavailable at this time
Have a good day!
```

Из вывода видно, что `getTemperature()` выбрасывает исключение. В теле функции `runBlocking()` вы окружаете вызов `println(getWeatherReport())` блоком `try-catch`. Вы ловите тип исключения, который ожидался (AssertionError в данном примере). Затем вы выводите исключение в виде «Caught exception», за которым следует строка сообщения об ошибке. Чтобы справиться с ошибкой, вы сообщаете пользователю, что прогноз погоды недоступен, с помощью дополнительного оператора `println()`: Report unavailable at this time.

Обратите внимание, что такое поведение означает, что если произошел сбой при получении температуры, то прогноза погоды не будет вообще (даже если был получен достоверный прогноз).

В зависимости от того, как вы хотите, чтобы вела себя ваша программа, существует альтернативный способ обработки исключения в программе погоды.

- Переместите обработку ошибок так, чтобы поведение `try-catch` фактически происходило внутри корутины, запущенной `async()` для получения температуры. Таким образом, прогноз погоды все еще может быть напечатан, даже если температура не была получена. Вот код:

```
import kotlinx.coroutines.*

fun main() {
```

```
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async {
        try {
            getTemperature()
        } catch (e: AssertionError) {
            println("Caught exception $e")
            "{ No temperature found }"
        }
    }

    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

- Запустите программу.

```
Weather forecast
Caught exception java.lang.AssertionError: Temperature is invalid
Sunny { No temperature found }
Have a good day!
```

Из вывода видно, что вызов `getTemperature()` завершился неудачей с исключением, но код в `async()` смог поймать это исключение и изящно его обработать, заставив coroutine вернуть String, в которой говорится, что температура не найдена. Отчет о погоде все еще может быть распечатан, с успешным прогнозом «Солнечно». Температура в прогнозе погоды отсутствует, но вместо нее выдается сообщение, объясняющее, что температура не найдена. Это лучше для пользователя, чем падение программы с ошибкой.

Этот подход к обработке ошибок можно представить так: `async()` является производителем, когда с его помощью запускается coroutine. `await()` является потребителем, потому что он ждет результата от coroutine. Производитель выполняет работу и выдает результат. Потребитель потребляет результат.

Если в продюсере возникнет исключение, то потребитель получит это исключение, если оно не будет обработано, и корутина завершится неудачей. Однако если производитель сможет поймать и обработать исключение, то потребитель не увидит этого исключения и увидит корректный результат.

Вот код `getWeatherReport()` для справки:

```
suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async {
        try {
            getTemperature()
        } catch (e: AssertionError) {
            println("Caught exception $e")
            "{ No temperature found }"
        }
    }

    "${forecast.await()} ${temperature.await()}"
}
```

В этом случае производитель (`async()`) смог перехватить и обработать исключение и по-прежнему вернуть строковый результат «{ Температура не найдена }». Потребитель (`await()`) получает этот строковый результат и даже не должен знать, что произошло исключение. Это еще один вариант изящной обработки исключения, которое, как вы ожидаете, может произойти в вашем коде.

> Примечание: исключения распространяются по-разному для сопрограмм, запущенных с помощью `launch()`, по сравнению с `async()`. Внутри сопрограммы, запущенной с помощью `launch()`, исключение выбрасывается немедленно, поэтому вы можете окружить код блоком try-catch, если ожидается, что он выдаст исключение. См. пример.

> Предупреждение: Внутри оператора `try-catch` в коде корутины избегайте перехвата общего `Exception`, поскольку он включает очень широкий спектр исключений. Вы можете случайно поймать и подавить ошибку, которая на самом деле является ошибкой и должна быть исправлена в вашем коде. Другая важная причина заключается в том, что отмена корутинов, которая обсуждается далее в этом разделе, зависит от `CancellationException`. Поэтому если вы будете ловить любые типы исключений, включая `CancellationExceptions`, не перебрасывая их, то поведение отмены в ваших coroutines может вести себя не так, как ожидалось. Вместо этого поймайте определенный тип исключения, которое, как вы ожидаете, будет выброшено из вашего кода.

Теперь вы узнали, что исключения распространяются вверх по дереву coroutine, если они не обрабатываются. Также важно быть осторожным, если исключение распространяется вплоть до корня иерархии, что может привести к краху всего приложения.

## Отмена

Схожая с исключениями тема - **отмена корутин**. Этот сценарий обычно зависит от пользователя, когда какое-либо событие заставляет приложение отменить работу, которую оно ранее начало.

Например, допустим, пользователь выбрал в приложении предпочтение, согласно которому он больше не хочет видеть в приложении значения температуры. Они хотят знать только прогноз погоды (например, солнечную), но не точную температуру. Следовательно, отмените корутину, которая в данный момент получает данные о температуре.

- Сначала начните с исходного кода, приведенного ниже (без отмены).

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

- После некоторой задержки отмените работу корутины, которая получала информацию о температуре, чтобы в прогнозе погоды отображался только прогноз. Измените возвращаемое значение блока `coroutineScope` так, чтобы оно было только строкой прогноза погоды.

```
...
suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }

    delay(200)
    temperature.cancel()

    "${forecast.await()}"
}
...
```

- Запустите программу. Теперь вывод выглядит следующим образом. Отчет о погоде состоит только из прогноза погоды Sunny, но не температуры, потому что эта корутина была отменена.

```
Weather forecast
Sunny
Have a good day!
```

Из этого вы узнали, что корутину можно отменить, но это не повлияет на другие корутины в той же области видимости, а родительская корутина не будет отменена.

> Отмена должна быть кооперативной, поэтому вы должны реализовать свою coroutine так, чтобы ее можно было отменить.

В этом разделе вы увидели, как `отмена` и `исключения` ведут себя в короутинах и как это связано с иерархией короутинов. Давайте узнаем больше о формальных концепциях, лежащих в основе короутинов, чтобы вы могли понять, как все важные части собираются вместе.

## Концепции корутин

При асинхронном или параллельном выполнении работы возникают вопросы о том, как будет выполняться работа, как долго должна существовать coroutine, что произойдет, если она будет отменена или завершится с ошибкой, и многое другое. Корутины следуют принципу структурированного параллелизма, который заставляет вас отвечать на эти вопросы, когда вы используете корутины в своем коде с помощью комбинации механизмов.

## Job

Когда вы запускаете корутину с помощью функции `launch()`, она возвращает экземпляр **Job**. В Job хранится `хэндл`, или ссылка, на coroutine, чтобы вы могли управлять ее жизненным циклом.

```
val job = launch { ... }
```

> Примечание: объект `Deferred`, который возвращается из корутины, запущенной с помощью функции `async()`, также является `Job` и хранит будущий результат корутины.

`Job` можно использовать для управления жизненным циклом, или тем, как долго будет жить корутина, например, для отмены корутины, если задание больше не нужно.

```
job.cancel()
```

С помощью `job` можно проверить, активно ли оно, отменено или завершено. `Job` считается завершенным, если эта корутина и все запущенные ею корутины выполнили всю свою работу. Заметим, что работа могла завершиться по другой причине, например, из-за отмены или исключения, но в этом случае job все равно считается завершенным.

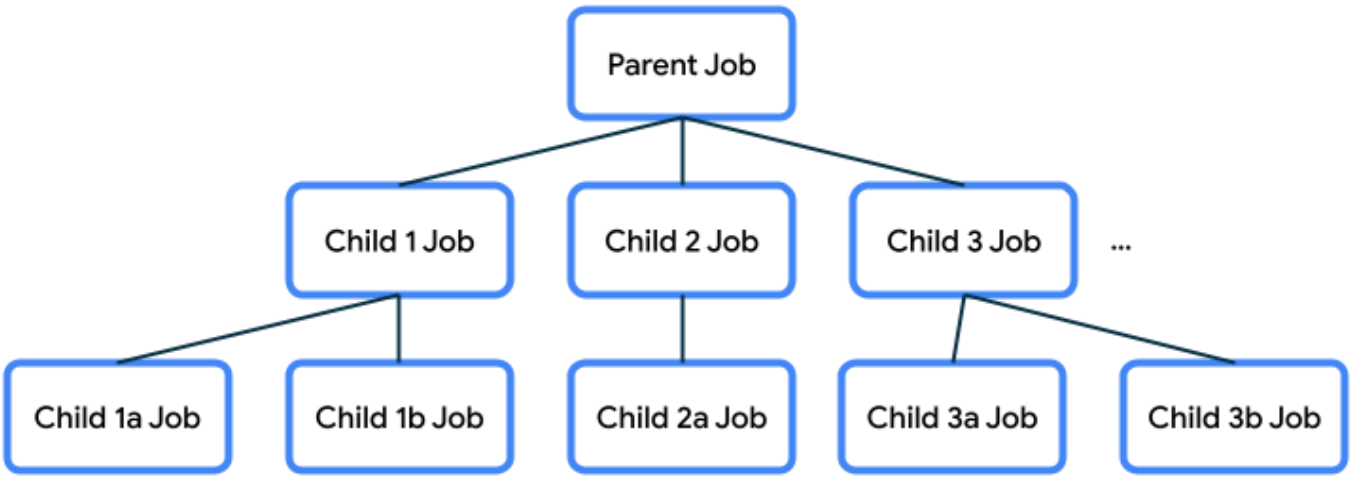
job также отслеживают отношения между родительскими и дочерними программами.

## Иерархия job

Когда корутина запускает другую корутину, job, которое возвращается из новой корутины, называется дочерним по отношению к исходному родительскому job.

```
val job = launch {
    ...

    val childJob = launch { ... }

    ...
}
```

Эти отношения между родителями и детьми образуют иерархию job, в которой каждое job может запускать job и так далее.




Эти отношения между родителем и ребенком важны, потому что они диктуют определенное поведение ребенка и родителя, а также других детей, принадлежащих тому же родителю. Вы видели это поведение в предыдущих примерах с программой погоды.

Если родительское job отменяется, то отменяются и его дочерние job. Когда дочернее job отменяется с помощью `job.cancel()`, оно завершается, но не отменяет свое родительское.

Если job терпит неудачу с исключением, оно отменяет свое родительское job с этим исключением. Это называется распространением ошибки вверх (к родителю, родителю родителя и так далее). .

## CoroutineScope

Корутины обычно запускаются в `CoroutineScope`. Это гарантирует, что у нас не будет неуправляемых и потерянных корутинов, которые могут тратить ресурсы.

`launch()` и `async()` - это функции расширения `CoroutineScope`. Вызовите `launch()` или `async()` для области видимости, чтобы создать новую корутину в этой области.

`CoroutineScope` привязан к жизненному циклу, который устанавливает границы того, как долго будут жить корутины внутри этого диапазона. Если область видимости отменяется, то отменяется ее job, и отмена этого job распространяется на ее дочерние job. Если дочернее job в области видимости не выполнилось с исключением, то другие дочерние job отменяются, родительское job отменяется, а исключение повторно передается вызывающему.

В этом уроке вы использовали функцию `runBlocking()`, которая предоставляет `CoroutineScope` для вашей программы. Вы также узнали, как использовать `coroutineScope { }` для создания новой области видимости в функции `getWeatherReport()`.

## CoroutineScope в приложениях Android

Android обеспечивает поддержку области видимости в сущностях с четко определенным жизненным циклом, таких как Activity (lifecycleScope) и ViewModel (viewModelScope). Корутины, запущенные в этих областях, будут придерживаться жизненного цикла соответствующей сущности, такой как Activity или ViewModel.

Например, допустим, вы запускаете корутину в Activity с предоставленной областью видимости корутины под названием `lifecycleScope`. Если активность будет уничтожена, то `lifecycleScope` будет отменен, и все его дочерние корутины также будут автоматически отменены. Вам просто нужно решить, является ли поведение корутины, следующей за жизненным циклом активности, тем поведением, которое вы хотите.

В приложении `Race Tracker` для Android, над которым вы будете работать, вы узнаете, как привязать свои корутины к жизненному циклу компонент.

## Детали реализации CoroutineScope

Если посмотреть исходный код реализации `CoroutineScope.kt` в библиотеке Kotlin coroutines, то можно увидеть, что `CoroutineScope` объявлен как интерфейс и содержит `CoroutineContext` в качестве переменной.

Функции `launch()` и `async()` создают новую дочернюю корутину в этой области, и она также наследует контекст от области. Что содержится в контексте? Давайте обсудим это далее.

## CoroutineContext

CoroutineContext предоставляет информацию о контексте, в котором будет выполняться coroutine. По сути, `CoroutineContext` - это карта, хранящая элементы, где каждый элемент имеет уникальный ключ. Это не обязательные поля, но вот некоторые примеры того, что может содержаться в контексте:

- имя - имя корутины для ее уникальной идентификации
- job - управляет жизненным циклом корутины
- диспетчер - отправляет работу в соответствующий поток
- обработчик исключений - обрабатывает исключения, вызванные кодом, выполняемым в корутине

> Примечание: Это значения по умолчанию для `CoroutineContext`, которые будут использоваться, если вы не укажете их значения:

- «coroutine» для имени программы
- нет родительского задания
- Dispatchers.Default для диспетчера корутины
- нет обработчика исключений

Каждый из элементов контекста может быть дополнен оператором +. Например, один `CoroutineContext` может быть определен следующим образом:

```
Job() + Dispatchers.Main + exceptionHandler
```

Поскольку имя не указано, используется имя корутины по умолчанию.

Если внутри корутины запустить новую корутину, то дочерняя корутина унаследует `CoroutineContext` от родительской корутины, но заменит задание специально для только что созданной корутины. Вы также можете переопределить любые элементы, которые были унаследованы от родительского контекста, передав аргументы в функции `launch()` или `async()` для тех частей контекста, которые вы хотите изменить.

```
scope.launch(Dispatchers.Default) {
    ...
}
```

Вы неоднократно встречали упоминание о **диспетчере**. Его роль заключается в диспетчеризации или назначении работы потоку. Давайте обсудим потоки и диспетчеры более подробно.

## Диспетчер

Корутины используют диспетчеры для определения потока, который будет использоваться для их выполнения. Поток может быть запущен, выполнить определенную работу (исполнить некоторый код), а затем завершиться, когда больше не будет никакой работы.

Когда пользователь запускает ваше приложение, система Android создает новый процесс и единственный поток выполнения для вашего приложения, который называется **главным потоком**. Главный поток выполняет множество важных операций для вашего приложения, включая системные события Android, отрисовку пользовательского интерфейса на экране, обработку событий пользовательского ввода и многое другое. В результате большая часть кода, который вы пишете для своего приложения, скорее всего, будет выполняться в главном потоке.

Есть два термина, которые необходимо понимать, когда речь заходит о потоковом поведении вашего кода: **блокирующий** и **неблокирующий**. Обычная функция блокирует вызывающий поток до тех пор, пока ее работа не будет завершена. Это означает, что она не отдает вызывающий поток до тех пор, пока работа не будет завершена, и никакая другая работа не может быть выполнена в это время. И наоборот, неблокирующий код выдает вызывающий поток до тех пор, пока не будет выполнено определенное условие, и вы можете в это время выполнять другую работу. Вы можете использовать асинхронную функцию для выполнения неблокирующей работы, потому что она возвращается до завершения работы.

В приложениях для Android **блокирующий** код следует вызывать в главном потоке только в том случае, если он будет выполняться достаточно быстро. Цель - держать главный поток незаблокированным, чтобы он мог выполнить работу немедленно, если сработает новое событие. Этот основной поток является потоком пользовательского интерфейса для вашей деятельности и отвечает за отрисовку пользовательского интерфейса и события, связанные с пользовательским интерфейсом. Когда на экране происходят изменения, пользовательский интерфейс должен быть перерисован. Например, при анимации на экране пользовательский интерфейс должен перерисовываться часто, чтобы это выглядело как плавный переход. Если главному потоку необходимо выполнить длительный блок работы, то экран будет обновляться не так часто, и пользователь увидит резкий переход (известный как «джанк») или приложение может зависнуть или медленно реагировать.

Следовательно, нам нужно переместить все длительные рабочие элементы из главного потока и обрабатывать их в другом потоке. Ваше приложение начинается с одного главного потока, но вы можете создать несколько потоков для выполнения дополнительной работы. Эти дополнительные потоки можно назвать **рабочими**. Совершенно нормально, если долго выполняющаяся задача блокирует рабочий поток на длительное время, потому что в это время главный поток разблокирован и может активно отвечать пользователю.

В Kotlin есть несколько встроенных диспетчеров:

- **Dispatchers.Main**: Используйте этот диспетчер для запуска корутины на главном потоке Android. Этот диспетчер используется в основном для обработки обновлений и взаимодействия с пользовательским интерфейсом, а также для выполнения быстрой работы.
- **Dispatchers.IO**: Этот диспетчер оптимизирован для выполнения дисковых или сетевых операций ввода-вывода вне основного потока. Например, чтение из файлов или запись в них, а также выполнение любых сетевых операций.
- **Dispatchers.Default**: Это диспетчер по умолчанию, используемый при вызове `launch()` и `async()`, когда в их контексте не указан диспетчер. Вы можете использовать этот диспетчер для выполнения трудоемкой вычислительной работы вне основного потока. Например, обработка файла растрового изображения.

> Примечание: Существуют также расширения `Executor.asCoroutineDispatcher()` и `Handler.asCoroutineDispatcher()`, если вам нужно создать `CoroutineDispatcher` из уже имеющегося `Handler` или `Executor`.

Попробуйте следующий пример в Kotlin, чтобы лучше понять диспетчеры корутин.

- Замените любой код в Kotlin следующим кодом:

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        launch {
            delay(1000)
            println("10 results found.")
        }
        println("Loading...")
```

```
    }
}
```

- Теперь оберните содержимое запущенной корутины вызовом функции `withContext()`, чтобы изменить `CoroutineContext`, в котором выполняется корутина, и, в частности, переопределить диспетчер. Переключитесь на использование `Dispatchers.Default` (вместо `Dispatchers.Main`, который в настоящее время используется для остального кода корутины в программе).

```
...

fun main() {
    runBlocking {
        launch {
            withContext(Dispatchers.Default) {
                delay(1000)
                println("10 results found.")
            }
        }
        println("Loading...")
    }
}
```

Переключение диспетчеров возможно потому, что функция withContext() сама по себе является приостанавливающей функцией. Она выполняет предоставленный блок кода с использованием нового контекста `CoroutineContext`. Новый контекст берется из контекста родительского job (внешнего блока `launch()`), за исключением того, что он переопределяет диспетчер, используемый в родительском контексте, на тот, который указан здесь: `Dispatchers.Default`. Таким образом, мы можем перейти от выполнения работы с `Dispatchers.Main` к использованию `Dispatchers.Default`.

- Запустите программу. На выходе должно получиться:

```
Loading...
10 results found.
```

- Добавьте операторы печати, чтобы узнать, в каком потоке вы находитесь, вызвав `Thread.currentThread().name`.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("${Thread.currentThread().name} - runBlocking function")
                launch {
            println("${Thread.currentThread().name} - launch function")
            withContext(Dispatchers.Default) {
                println("${Thread.currentThread().name} - withContext function")
```

```
                delay(1000)
                println("10 results found.")
            }
            println("${Thread.currentThread().name} - end of launch function")
        }
        println("Loading...")
    }
}
```

- Запустите программу. На выходе должно получиться:

```
main @coroutine#1 - runBlocking function
Loading...
main @coroutine#2 - launch function
DefaultDispatcher-worker-1 @coroutine#2 - withContext function
10 results found.
main @coroutine#2 - end of launch function
```

Из этого вывода видно, что большая часть кода выполняется в корутинах в главном потоке. Однако часть кода в блоке `withContext(Dispatchers.Default)` выполняется в корутине на рабочем потоке `Default Dispatcher` (который не является основным потоком). Обратите внимание, что после возврата функции `withContext()` coroutine возвращается к выполнению в основном потоке (об этом свидетельствует оператор вывода: main @coroutine#2 - конец функции запуска). Этот пример демонстрирует, что вы можете переключать диспетчер, изменяя контекст, используемый для короутина.

Если у вас есть корутины, которые были запущены в главном потоке, и вы хотите перенести определенные операции из главного потока, то вы можете использовать `withContext` для переключения диспетчера, используемого для этой работы. Выберите подходящий из доступных диспетчеров: `Main`, `Default` и `IO`, в зависимости от типа операции. Затем эта работа может быть назначена потоку (или группе потоков, называемой пулом потоков), предназначенному для этой цели. Корутины могут самостоятельно приостанавливаться, и диспетчер также влияет на то, как они возобновляются.

Обратите внимание, что при работе с такими популярными библиотеками, как `Room` и `Retrofit` (в этом и следующем блоке), вам может не потребоваться явно переключать диспетчер, если код библиотеки уже выполняет эту работу с помощью альтернативного диспетчера корутин, например `Dispatchers.IO`. В таких случаях функции приостановки, открываемые этими библиотеками, могут быть безопасными для основного потока и вызываться из корутины, запущенной в основном потоке. Библиотека сама справится с переключением диспетчера на тот, который использует рабочие потоки.

Теперь вы получили высокоуровневое представление о важных частях короутинов и роли, которую `CoroutineScope`, `CoroutineContext`, `CoroutineDispatcher` и `Jobs` играют в формировании жизненного цикла и поведения короутина.

## Заключение

Вы отлично поработали над этой сложной темой о корутинах! Вы узнали, что корутины очень полезны, потому что их выполнение можно приостановить, освободив основной поток для выполнения другой

работы, а затем возобновить работу корутины. Это позволяет выполнять в коде параллельные операции.

Корутинный код в Kotlin следует принципу **структурированного параллелизма**. По умолчанию он является последовательным, поэтому вам необходимо явно указать, что вы хотите получить параллельность (например, используя `launch()` или `async()`). При структурированном параллелизме вы можете взять несколько параллельных операций и поместить их в одну синхронную операцию, где параллелизм - это деталь реализации. Единственное требование к вызывающему коду - находиться в suspend функции или корутине. В остальном структура вызывающего кода не должна учитывать детали параллелизма. Это делает ваш асинхронный код более легким для чтения и рассуждений.

Структурированный параллелизм отслеживает все запущенные в вашем приложении корутины и гарантирует, что они не будут потеряны. Корутины могут иметь иерархию - job могут запускать подзадачи, которые, в свою очередь, могут запускать подзадачи. Job поддерживают отношения «родитель-ребенок» между короутинами и позволяют вам контролировать жизненный цикл короутины.

`Запуск`, `завершение`, `отмена` и `отказ` - это четыре общие операции, выполняемые в процессе выполнения коротины. Чтобы облегчить сопровождение параллельных программ, структурированный параллелизм определяет принципы, на основе которых осуществляется управление общими операциями в иерархии:

- **Запуск**: Запуск coroutine в область видимости, которая имеет определенную границу времени жизни.
- **Завершение**: Job не завершено, пока не завершены его дочерние job.
- **Отмена**: Эта операция должна распространяться вниз. Когда отменяется короутин, дочерние короутины также должны быть отменены.
- **Exception**: Эта операция должна распространяться вверх. Если какая-либо корутина выбрасывает исключение, то родительская отменяет все свои дочерние корутины, отменяет себя и распространяет исключение до своей родительской. Это продолжается до тех пор, пока исколючение не будет поймано и обработано. Это гарантирует, что о любых ошибках в коде будет сообщено должным образом, и они никогда не будут потеряны. Благодаря практической работе с coroutines и пониманию концепций, лежащих в основе coroutines, вы теперь лучше подготовлены к написанию параллельного кода в вашем приложении для Android. Используя корутины для асинхронного программирования, ваш код будет проще для чтения и рассуждений, более надежным в ситуациях отмены и исключений, а также обеспечит более оптимальный и отзывчивый опыт для конечных пользователей.

## Резюме

- Корутины позволяют писать долго выполняющийся код, который работает параллельно, без изучения нового стиля программирования. Выполнение корутины является последовательным по своей природе.
- Корутины следуют принципу **структурированного параллелизма**, который помогает гарантировать, что работа не будет потеряна и привязана к области видимости с определенными границами ее жизни. Ваш код по умолчанию является последовательным и взаимодействует с базовым циклом событий, если только вы явно не попросите о параллельном выполнении (например, используя `launch()` или `async()`). Предполагается, что если вы вызываете функцию,

то она должна полностью завершить свою работу (если не произойдет исключения) к моменту возвращения, независимо от того, сколько корутинов она могла использовать в деталях своей реализации.

- Модификатор `suspend` используется для обозначения функции, выполнение которой может быть приостановлено и возобновлено в более поздний момент.
- Приостановленная функция может быть вызвана только из другой приостанавливающей функции или из корутины.
- Запустить новую корутину можно с помощью функций расширения `launch()` или `async()` на `CoroutineScope`.
- `Jobs` играет важную роль в обеспечении структурированного параллелизма, управляя жизненным циклом `coroutine` и поддерживая отношения «родитель-ребенок».
- `CoroutineScope` контролирует время жизни корутинов с помощью своего `Job` и рекурсивно применяет правила отмены и другие правила к своим дочерним процессам и их дочерним процессам.
- Контекст `CoroutineContext` определяет поведение программы и может включать ссылки на job и диспетчер программы.
- Корутины используют `CoroutineDispatcher` для определения потоков, которые будут использоваться для их выполнения.