Корутины

Введение в корутины

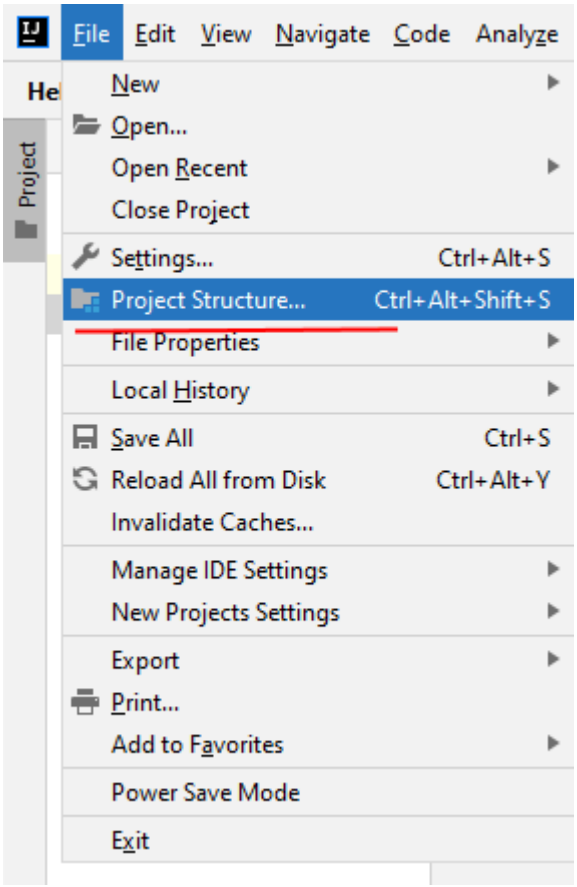
В последнее время поддержка асинхронности и параллельных вычислений стала неотъемлимой чертой многих языков программирования. И Kotlin не является исключением. Зачем нужны асинхронность и параллельные вычисления? Параллельные вычисления позволяют выполнять несколько задач одновременно, а асинхронность позволяет не блокировать основной ход приложения во время выполнения задачи, которая занимает продолжительное время. Например, мы создаем графическое приложение для десктопа или мобильного устройства. И нам надо по нажатию на кнопку отправлять запрос к интернет-ресурсу. Однако подобный запрос может занять продолжительное время. И чтобы приложение не зависало на период отправки запроса, подобные запросы к интернет-ресурсам следует отправлять асинхронно. При асинхронных запросах пользователь не ждет пока придет ответ от интернет-ресурса, а продолжает работу с приложением, а при получении ответа получит соответствующее уведомление.

В языке Kotlin поддержка асинхронности и параллельных вычислений воплощена в виде корутин (coroutine). По сути корутина представляет блок кода, который может выполняться параллельно с остальным кодом. А базовая функциональность, связанная с корутинами, сосредоточена в библиотеке kotlinx.coroutines.

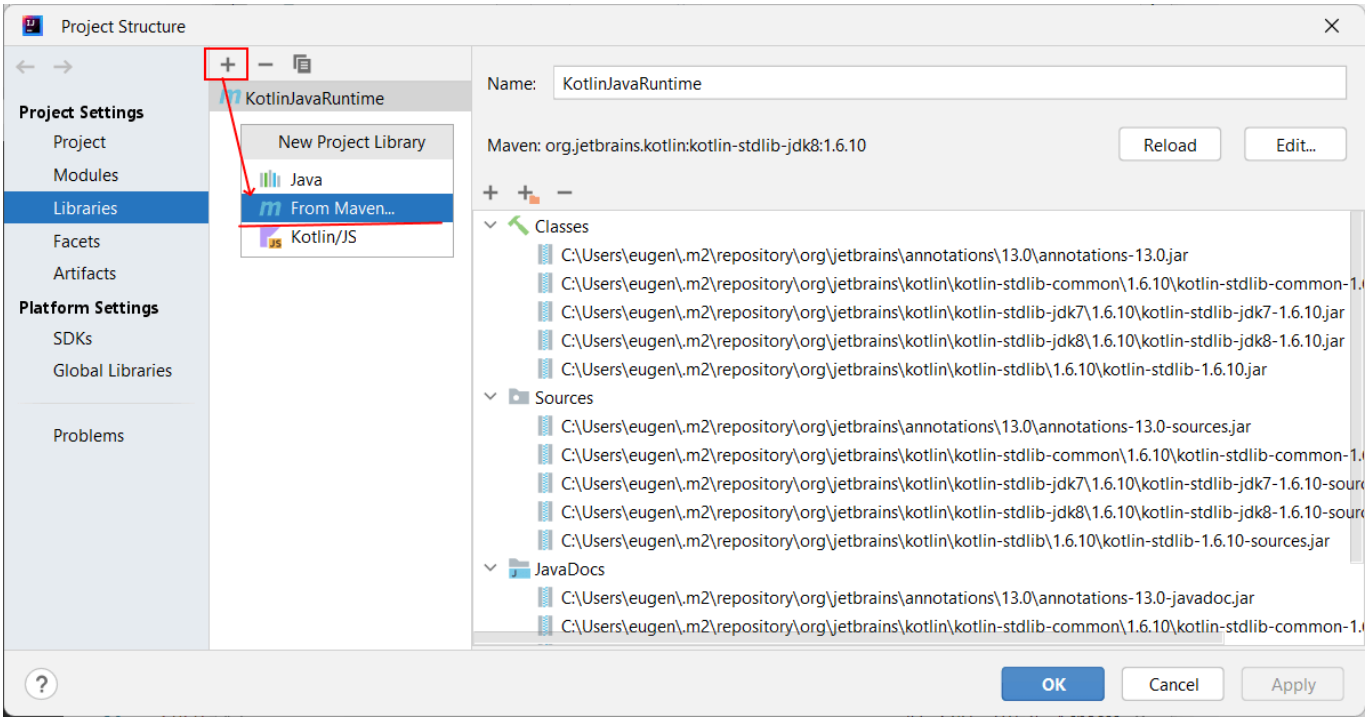
Рассмотрим определение и применение корутины на простейшем примере.

Добавление kotlinx.coroutines

Прежде всего стоит отметить, что функциональность корутин (библиотека kotlinx.coroutines) по умолчанию не включена в проект. И нам ее надо добавить. Если мы создаем проект консольного приложения в IntelliJ IDEA, то мы можем добавить соответствующую библиотеку в проект. Для этого в меню File перейдем к пункту Project Structure..

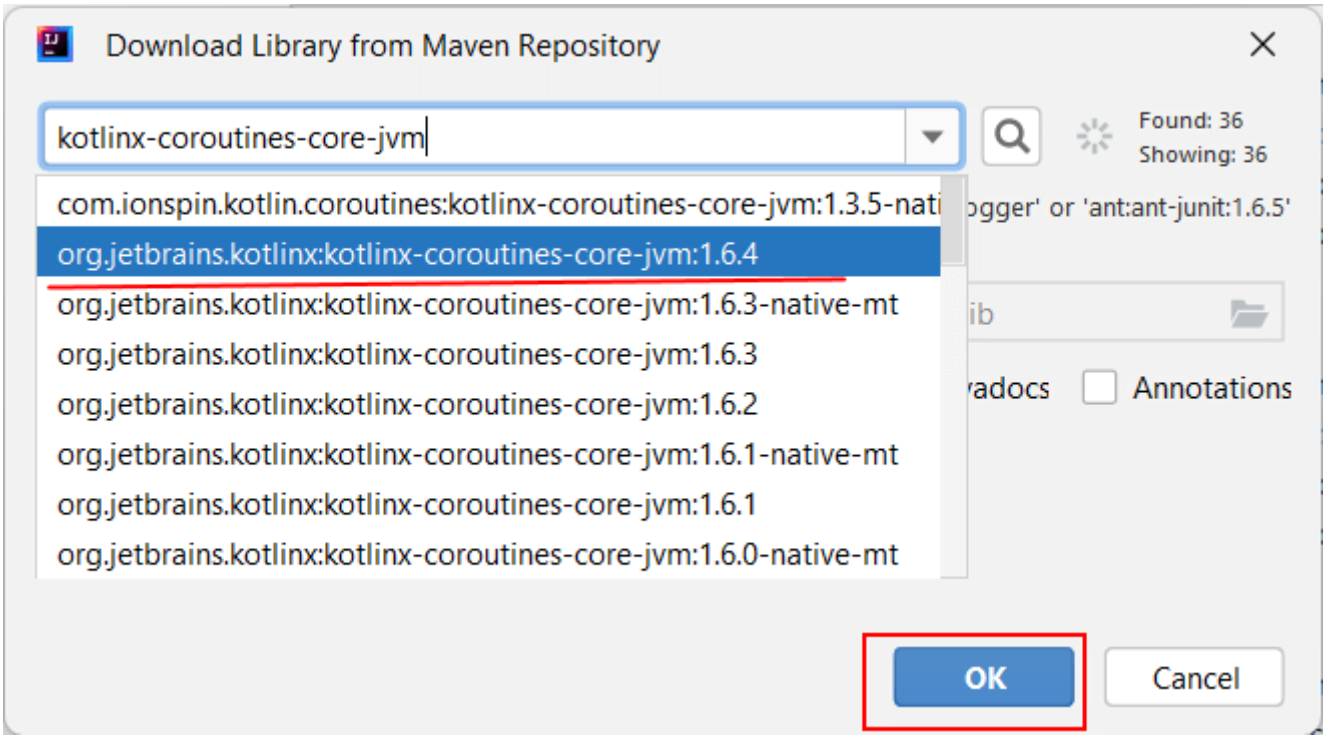


Далее на вкладке "Project Settings" перейдем к пункту Libraries. В центральном поле отобразятся библиотеки, добавленные в проект.



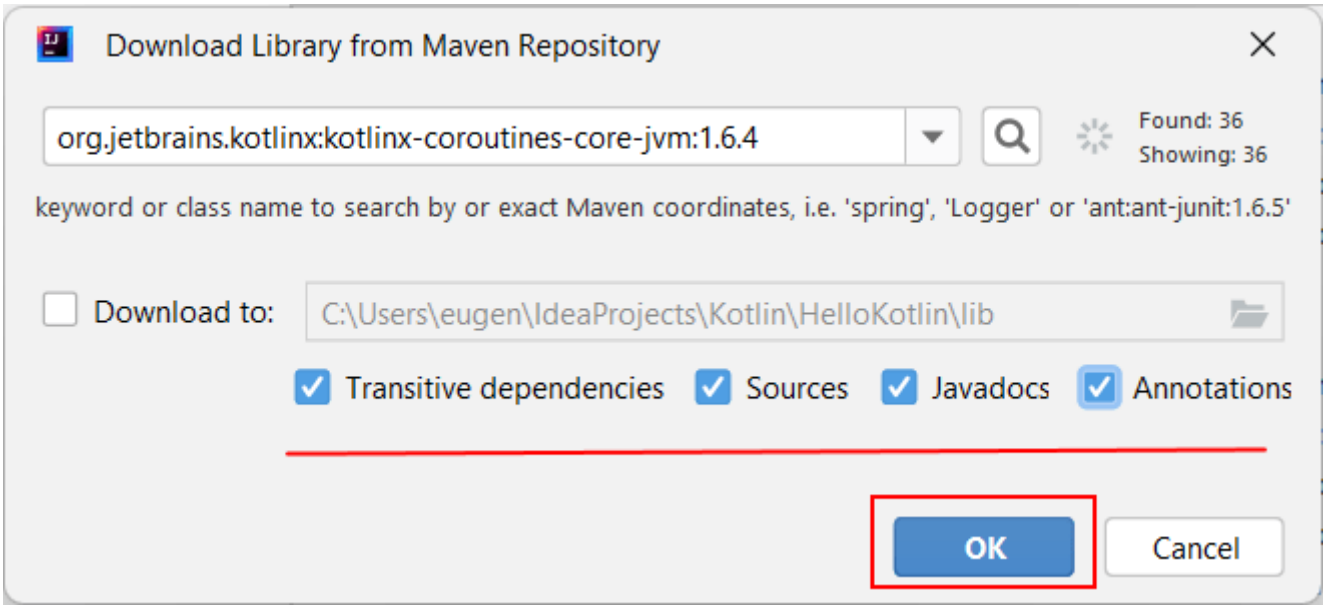
И для добавления новой библиотеки нажмем на знак плюса и в контекстном меню выберем пункт From Maven...

После этого нам откроется окно для добавления библиотеки через Maven. В этом окне в поле ввода введем название нужной нам библиотеки - kotlinx-coroutines-core-jvm и нажмем на кнопку поиска. Если соответствующая библиотека найдена, то нам отобразится выпадающий список с результатами

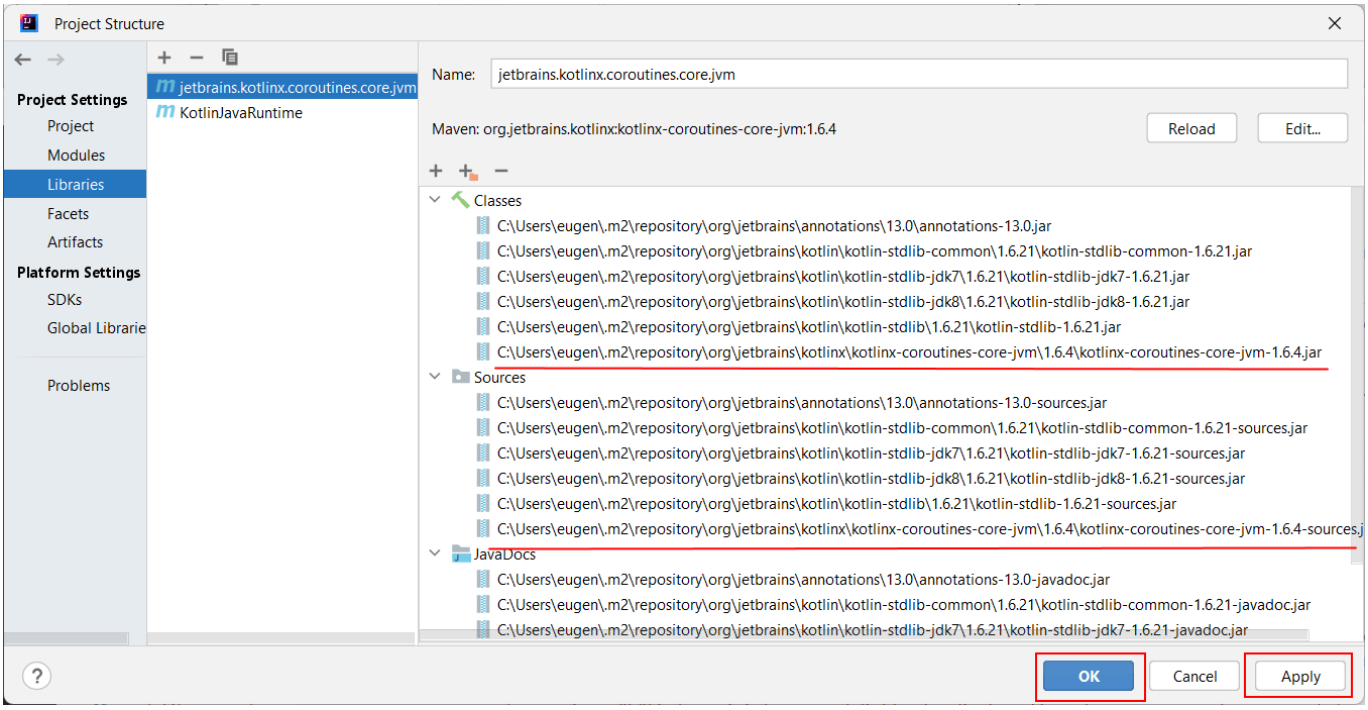


Выберем из него последнюю версию, которая называется наподобие org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.6.4 - в данном случае используется версия 1.6.4, но конкретный номер версии может отличаться.

Отметим все необходимые флажки и нажмем на кнопку OK

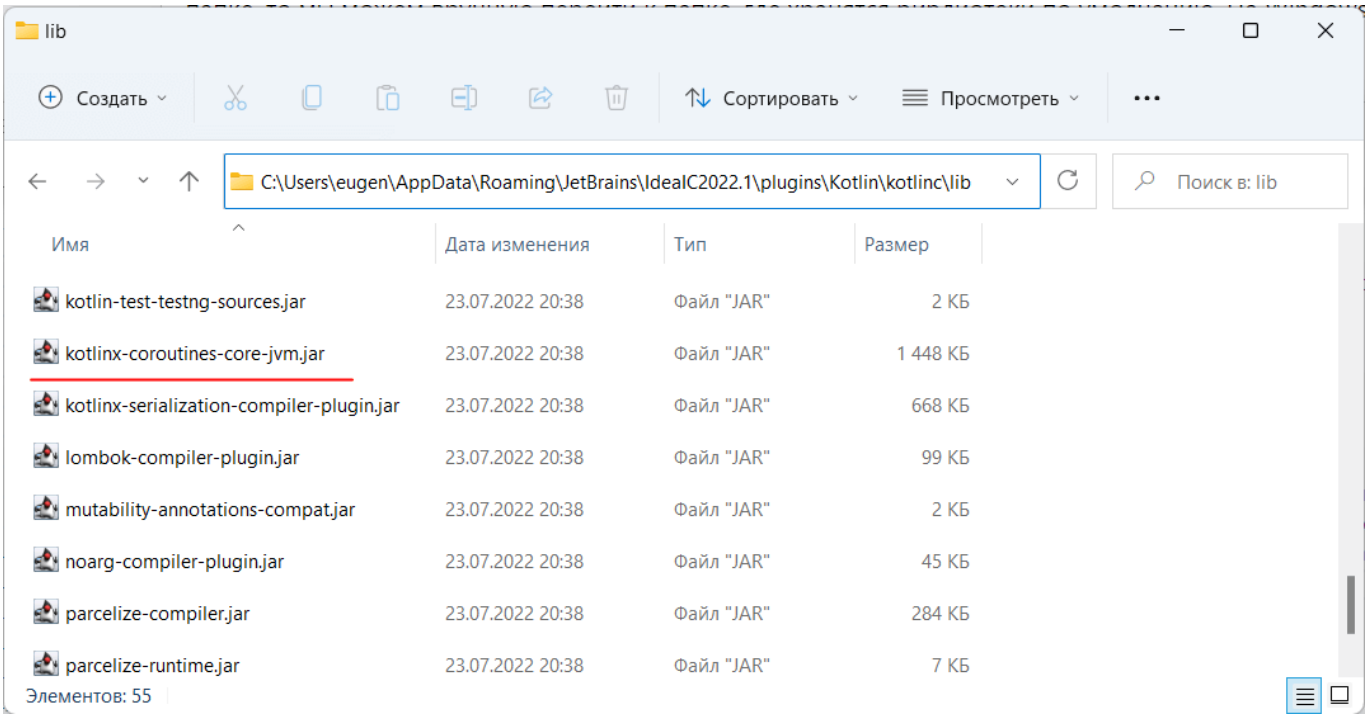


После установки библиотеки мы сможем найти ее файл в списке библиотек



В качестве альтернативы мы могли бы вручную подключить нужную библиотеку из локального хранилища. Так, на Windows это будет папка C:\Users[Имя_пользователя]\AppData\Roaming\JetBrains\IdealC[номер_версии]\plugins\Kotlin\kotlinc\lib

Далее в этой папке выберем библиотеку kotlinx-coroutines-core-jvm.jar и нажмем на OK для ее добавления:



Определение suspend-функции

Сначала рассмотрим пример, который не использует корутины:

```
import kotlinx.coroutines.*
```

```
suspend fun main(){
    for(i in 0..5){
        delay(400L)
        println(i)
    }

    println("Hello Coroutines")
}
```

Здесь в функции main перебираем последовательность от 0 до 5 и выводит текущий элемент последовательности на консоль. Для имитации продолжительной работы внутри цикла вызываем специальную функцию delay() из пакета kotlinx.coroutines. В эту функцию передается количество миллисекунд, на которое выполняется задержка. Передаваемое значение должно иметь тип Long. То есть здесь функция будет выполнять задержку в 400 миллисекунд перед выводом на консоль текущего элемента последовательности.

После выполнения работы цикла выводим на консоль строку "Hello Coroutines".

И чтобы использовать внутри функции main функцию delay(), функция main предваряется модификатром suspend. Модификатор suspend определяет функцию, которая может приостановить свое выполнение и возобновить его через некоторый период времени.

Сама функция delay() тоже является подобной функцией, которая определена с модификатором suspend. А любая функция с модификатором suspend может вызываться либо из другой функции, которая тоже имеет модификатор suspend, либо из корутины.

Если мы запустим приложение, то мы увидим следующий консольный вывод:

```
0
1
2
3
4
5
Hello Coroutines
```

Здесь мы видим, что строка "Hello Coroutines" выводится после выполнения цикла. Но вместо цикла у нас могла бы быть более содержательная, но и более продолжительная работа, например, обращение к интернет-ресурсу, к удаленой базе данных, какие-то операции с файлами и т.д. И в этом случае все определенные после этой работы действия ожидали бы завершения этой продолжительной работы, как в данном случае строка "Hello Coroutines" ждет завершения цикла.

Определение корутины

Теперь вынесем продолжительную работу - то есть цикл в корутину:

```
import kotlinx.coroutines.*
```

```
suspend fun main() = coroutineScope{
    launch{
            for(i in 0..5){
            delay(400L)
            println(i)
        }
    }

    println("Hello Coroutines")
}
```

Прежде всего для определения и выполнения корутины нам надо определить для нее контекст, так как корутина может вызываться только в контексте корутины (coroutine scope). Для этого применяется функция coroutineScope() - создает контекст корутины. Кроме того, эта функция ожидает выполнения всех определенных внутри нее корутин. Стоит отметить, что coroutineScope() может применяться только в функции с модификатором suspend, коей является функция main.

Сама корутина определяется и запускается с помощью построителя корутин - функции launch. Она создает корутину в виде блока кода - в данном случае это:

```
{
    for(i in 0..5){
        delay(400L)
        println(i)
    }
}
```

и запускает эту корутину параллельно с остальным кодом. То есть данная корутина выполняется независимо от прочего кода, определенного в функции main.

В итоге при выполнении программы мы увидим несколько другой консольный вывод:

```
Hello Coroutines
0
1
2
3
4
5
```

Теперь строка "Hello Coroutines" не ожидает, пока завершится цикл, а выполняется параллельно с ним.

Вынесение кода корутин в отдельную функцию

Выше код корутины располагался непосредственно в функции main. Но также можно определить его в виде отдельной функции и вызывать в корутине эту функцию:

```
import kotlinx.coroutines.*

suspend fun main()= coroutineScope{
    launch{ doWork() }

    println("Hello Coroutines")
}
suspend fun doWork(){
    for(i in 0..5){
        println(i)
        delay(400L)
    }
}
```

В данном случае основной код корутины вынесен в функцию doWork(). Поскольку в этой функции применяется функция delay(), то doWork() определена с модификатором suspend. Сама корутина создается также с помощью функции launch(), которая вызывает функцию doWork().

Обратите внимание, что в примере выше в конце функции main вызывается функция println(), которая выводит строку на консоль. Если мы ее удалим, то мы столкнемся с ошибкой - функция main должна возвращать значение Unit. В этом случае мы можем либо явным образом возвратить значение Unit:

```
import kotlinx.coroutines.*

suspend fun main()= coroutineScope{
    launch{
        for(i in 0..5){
            println(i)
            delay(400L)
        }
    }
    Unit
}
```

Либо можно типизировать функцию coroutineScope типом Unit:

```
import kotlinx.coroutines.*

suspend fun main()= coroutineScope<Unit>{
    launch{
        for(i in 0..5){
            println(i)
            delay(400L)
        }
    }
}
```

Корутины и потоки

В ряде языков программирования есть такие структуры, которые позволяют использовать потоки. Однако между корутинами и потоками нет прямого соответствия. Корутина не привязана к конкретному потоку. Она может быть приостановить выполнение в одном потоке, а возобновить выполнение в другом.

Когда корутина приостанавливает свое выполнение, например, как в случае выше при вызове задержки с помощью функции delay(), эта корутина освобождает поток, в котором она выполнялась, и сохраняется в памяти. А освобожденный поток может быть зайдествован для других задач. А когда завершается запущенная задача (например, выполнение функции delay()), корутина возобновляет свою работу в одном из свободных потоков.

Область корутины

Корутина может выполняться только в определенной области корутины (coroutine scope). Область корутин представляет пространство, в рамках которого действуют корутины, она имеет определенный жизненный цикл и сама управляет жизненным циклом создаваемых внутри нее корутин.

И для создания области корутин в Kotlin может использоваться ряд функций, которые создают объект интерфейса CoroutineScope. Одной из функций является coroutineScope. Она может применяться к любой функции, например:

```
import kotlinx.coroutines.*

suspend fun main(){

    doWork()

    println("Hello Coroutines")
}
suspend fun doWork()= coroutineScope{
    launch{
        for(i in 0..5){
            println(i)
            delay(400L)
        }
    }
}
```

Запуск нескольких корутин

Подобным образом можно запускать в одной функции сразу несколько корутин. И они будут выполняться одновременно. Например:

```
import kotlinx.coroutines.*

suspend fun main()= coroutineScope{
```

```
    launch{
        for(i in 0..5){
            delay(400L)
            println(i)
        }
    }
    launch{
        for(i in 6..10){
            delay(400L)
            println(i)
        }
    }

    println("Hello Coroutines")
}
```

Функция coroutineScope(), которая создает область корутин, будет ожидать завершения всех определенных в этой области корутин. То есть функция main завршит выполнение, когда будут завершены обе корутины.

И в моем случае я получу следующий консольный вывод (данный вывод строго не детерминирован):

```
Hello Coroutines
6
0
7
1
8
2
9
3
10
4
5
```

runBlocking

Кроме функции coroutineScope для создания контекста корутины может применяться функция runBlocking.

```
import kotlinx.coroutines.*

fun main() = runBlocking{
    launch{
        for(i in 0..5){
            delay(400L)
            println(i)
        }
```

```
    }

    println("Hello Coroutines")
}
```

Функция runBlocking блокирует вызывающий поток, пока все корутины внутри вызова runBlocking { ... } не завершат свое выполнение. В этом собственно основное отличие runBlocking от coroutineScope: coroutineScope не блокирует вызывающий поток, а просто приостанавливает выполнение, освобождания поток для использования другими ресурсами.

Вложенные корутины

Одна корутина может содержать другие корутины. Например:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    launch{
        println("Outer coroutine")
        launch{
            println("Inner coroutine")
            delay(400L)
        }
    }

    println("End of Main")
}
```

И подобным образом внешние корутины определяют область для вложенных корутин и управляют их жизненным циклом.

launch и Job

Для создания корутины нужен построитель корутин. И одним из построителей корутин в пакете kotlinx.coroutines является функция launch. В принципе в прошлых темах уже было рассмотрено, как с помощью launch создавать корутины. Сейчас же рассмотрем некоторые аспекты подробнее.

Прежде всего, launch(), как правило, применяется, когда нам не надо возвращать результат из корутины и когда нам ее надо выполнять одновременно с другим кодом.

Job

Построитель корутин launch возвращает объект Job, с помощью которого можно управлять запущеной корутиной:

```
val job: Job = launch{
    println("Some coroutin")
```

```
    delay(400L)
}
```

Например, его метод join() позволяет ожидать, пока корутина не завершится. Например:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    launch{
        for(i in 1..5){
            println(i)
            delay(400L)
        }
    }

    println("Start")
    println("End")
}
```

В данном случае мы получим следующий консольный вывод:

```
Start
End
1
2
3
4
5
```

Теперь явным образом применим интерфейс Job:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    val job = launch{
        for(i in 1..5){
            println(i)
            delay(400L)
        }
    }

    println("Start")
    job.join() // ожидаем завершения корутины
    println("End")
}
```

Здесь корутина также запускается с помощью launch, однако благодаря методу join() полученного объекта Job функция main остановит выполнение и будет ожидать завершения корутины и только после ее завершения продолжит работу. Соответственно в данном случае консольный вывод будет иным:

```
Start
1
2
3
4
5
End
```

Отложенное выполнение

По умолчанию построитель корутин launch создает и сразу же запускает корутину. Однако Kotlin также позволяет применять технику отложенного запуска корутины (lazy-запуск), при котором корутина запускается при вызове метода start() объекта Job.

Для установки отложенного запуска в функцию launch() передается значение start = CoroutineStart.LAZY

Чтобы увидеть разницу, сначала возьмем корутину со стандартным выполнением:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    // корутина создана и запущена
    launch( ) {
        delay(200L)
        println("Coroutine has started")
    }

    delay(1000L)
    println("Other actions in main method")
}
```

Чтобы позволить корутины выполниться до остальных действий в методе main, после определения корутины установлена задержка в 1 секунду. В итоге здесь получим следующий консольный вывод:

```
Coroutine has started
Other actions in main method
```

Теперь применим отложенное выполнение:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    // корутина создана, но не запущена
    val job = launch(start = CoroutineStart.LAZY) {
        delay(200L)
        println("Coroutine has started")
    }

    delay(1000L)
    job.start() // запускаем корутину
    println("Other actions in main method")
}
```

Теперь корутина только создается с помощью функции launch, но непосредственно она запускается только при вызове метода job.start(), соответственно мы получим другой результат программы:

```
Other actions in main method
Coroutine has started
```

Async, await и Deferred

Наряду с launch в пакете kotlinx.coroutines есть еще один построитель корутин - функция async. Эта функция применяется, когда надо получить из корутины некоторый результат.

async запускает отдельную корутину, которая выполняется параллельно с остальными корутинами. Например:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    async{ printHello()}
    println("Program has finished")
}
suspend fun printHello(){
    delay(500L)  // имитация продолжительной работы
    println("Hello work!")
}
```

Консольный вывод программы:

```
Program has finished
Hello work!
```

Кроме того, async-корутина возвращает объект Deferred, который ожидает получения результата корутины. (Интерфейс Deferred унаследован от интерфейса Job, поэтому для также доступны весь функционал, определенный для интефейса Job)

Для получения результата из объекта Deferred применяется функция await(). Рассмотрим на примере:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    val message: Deferred<String> = async{ getMessage()}
    println("message: ${message.await()}")
    println("Program has finished")
}
suspend fun getMessage() : String{
    delay(500L)  // имитация продолжительной работы
    return "Hello"
}
```

В данном случае для имитации продолжительной работы определена функция getMessage(), которая возвращает строку.

С помощью функции async запускаем корутину, которая выполняет эту функцию.

```
async{ getMessage()}
```

Поскольку функция getMessage() возвращает объект типа String, то возвращаемый корутиной объект представляет тип Deferred (объект Deferred типизируется возвращаемым типом функции, то есть в данном случае типом String).

```
val message: Deferred<String> = async{ getMessage()}
```

Далее у объекта Deferred для получения результата функции getMessage() вызываем метод await(). Он ожидает, пока не будет получен результат. Консольный вывод программы:

```
message: Hello
Program has finished
```

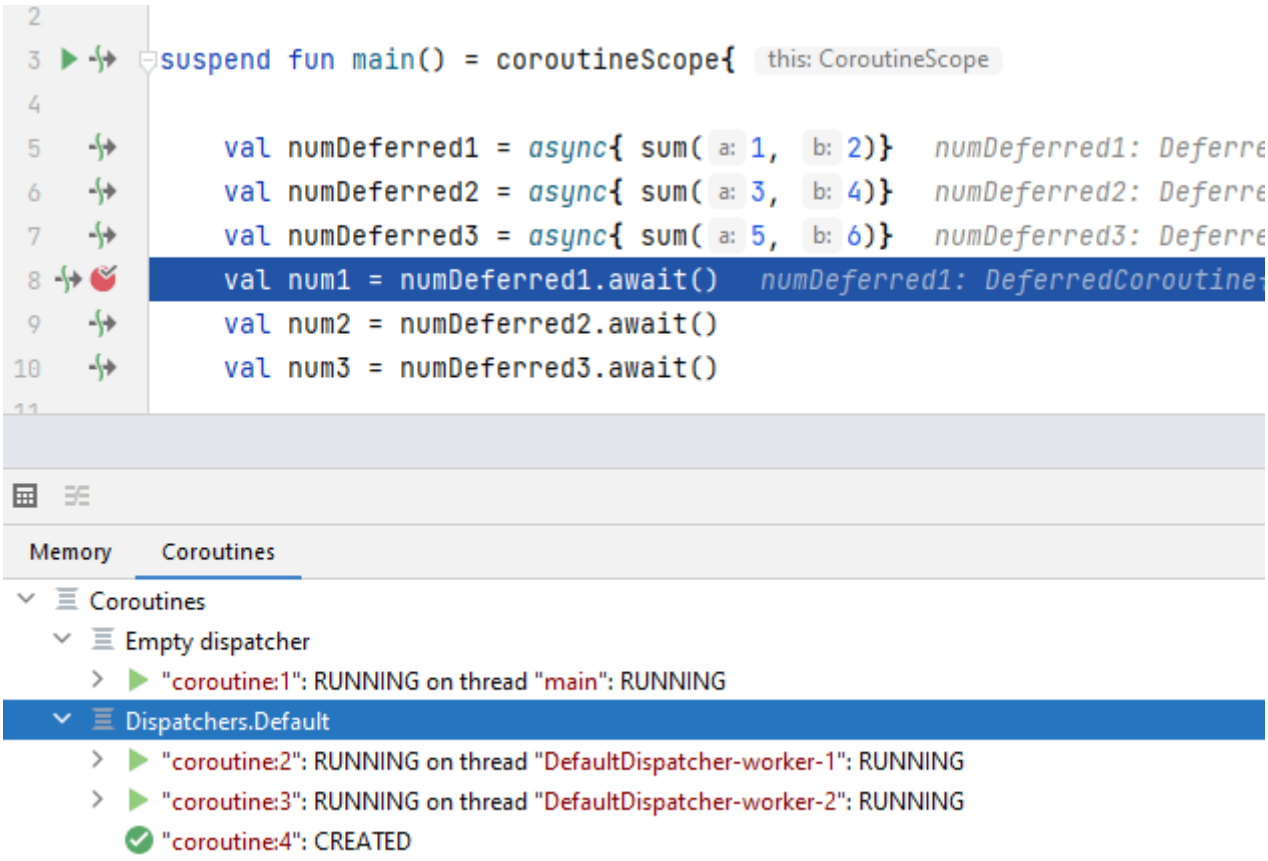
Поскольку функция getMessage() возвращает объект типа String, то метод await() в данном случае также будет возвращать строку, которую мы могли бы, например, присвоить переменной:

```
val text: String = message.await()
```

При этом мы можем с помощью async запустить несколько корутин, которые будут выполняться параллельно:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    val numDeferred1 = async{ sum(1, 2)}
    val numDeferred2 = async{ sum(3, 4)}
    val numDeferred3 = async{ sum(5, 6)}
    val num1 = numDeferred1.await()
    val num2 = numDeferred2.await()
    val num3 = numDeferred3.await()

    println("number1: $num1  number2: $num2  number3: $num3")
}
suspend fun sum(a: Int, b: Int) : Int{
    delay(500L) // имитация продолжительной работы
    return a + b
}
```

Здесь запускается три корутины, каждая из которых выполняет функцию sum(). Эта функция складывает два числа и возвращает их сумму в виде объекта Int. Поэтому корутины возвращают объект Deferred. Соответственно вызов метода await() у этого объекта возвратит объект Int, то есть сумму двух чисел. При этом все три корутины будет запущены одновременно. Например, ниже на скриншоте отладчика корутин видно, что две корутины уже работают (или находятся в состоянии Running), и еще одна корутина только создана и ожидает запуска (состояние Created)



Отложенный запуск

По умолчанию построитель корутин async создает и сразу же запускает корутину. Но как и при создании корутины с помощью launch для async-корутин можно применять технику отложенного запуска. Только в данном случае корутина запускается не только при вызове метода start объекта Deferred (который усналедован от интерфейса Job), но также и с помощью метода await() при обращении к результу корутины. Например:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    // корутина создана, но не запущена
    val sum = async(start = CoroutineStart.LAZY){ sum(1, 2)}

    delay(1000L)
    println("Actions after the coroutine creation")
    println("sum: ${sum.await()}")   // запуск и выполнение корутины
}
fun sum(a: Int, b: Int) : Int{
    println("Coroutine has started")
    return a + b
}
```

Консольный вывод программы:

```
Actions after the coroutine creation
Coroutine has started
sum: 3
```

Если необходимо, чтобы корутина еще до метода await() начала выполняться, то можно вызвать метод start():

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    // корутина создана, но не запущена
    val sum = async(start = CoroutineStart.LAZY){ sum(1, 2)}

    delay(1000L)
    println("Actions after the coroutine creation")
    sum.start()                         // запуск корутины
    println("sum: ${sum.await()}")      // получаем результат
}
fun sum(a: Int, b: Int) : Int{
    println("Coroutine has started")
    return a + b
}
```

Диспетчер корутины

Контекст корутины включает себя такой элемент как диспетчер корутины. Диспетчер корутины определяет какой поток или какие потоки будут использоваться для выполнения корутины.

Все построители корутины, в частности, функции launch и async в качестве необязательного параметра принимают объект типа CoroutineContext, который может использоваться для определения диспетчера создаваемой корутины.

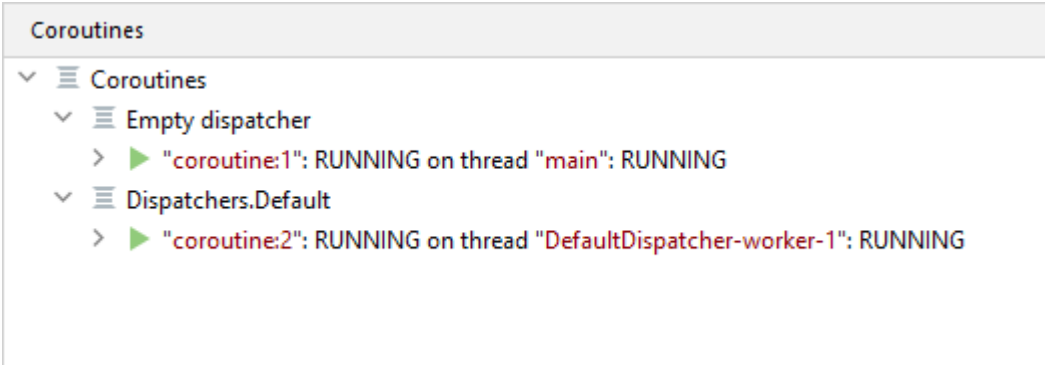
Когда функция launch вызывается без параметров, она перенимает контекст, в котором она создается и запускается. Например, используем метод Thread.currentThread(), который предоставляет JDK, чтобы получить данные потока корутины:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    launch {
        println("Корутина выполняется на потоке: ${Thread.currentThread().name}")
    }
    println("Функция main выполняется на потоке: ${Thread.currentThread().name}")
}
```

Здесь с помощью переменной Thread.currentThread().name мы можем получить имя потока. И в данном случае мы получим консольный вывод наподобие следующего:

```
Функция main выполняется на потоке: main
Корутина выполняется на потоке: DefaultDispatcher-worker-1
```

Мы видим, что функция main выполняется на потоке под названием "main" (который собственно и отведен для выполнения этой функции), а корутина выполняется на другом потоке с названием DefaultDispatcher-worker-1. Если мы обратимся к отладчику корутин, то мы сможем увидеть применяемый корутиной диспетчер:



Здесь мы видим, что корутина, которая выполняется в потоке "DefaultDispatcher-worker-1", применяет диспетчер Dispatcher.Default.

Поскольку контекст корутин в функции main создается в данном случае с помощью функции coroutineScope, которая устанавливает для создаваемых корутин по умолчанию диспетчер типа Dispatcher.Default. И, корутина, определенная в примере выше перенимает этот контекст вместе с данным типом диспетчера.

Что это значит? Рассмотрим доступные типы диспетчеров:

- Dispatchers.Default: применяется по умолчанию, если тип диспетчера не указан явным образом. Этот тип использует общий пул разделяемых фоновых потоков и подходит для вычислений, которые не работают с операциями ввода-вывода (операциями с файлами, базами данных, сетью) и которые требуют интенсивного потребления ресурсов центрального процессора.
- Dispatchers.IO: использует общий пул потоков, создаваемых по мере необходимости, и предназначен для выполнения операций ввода-вывода (например, операции с файлами или сетевыми запросами).
- Dispatchers.Main: применяется в графических приложениях, например, в приложениях Android или JavaFX.
- Dispatchers.Unconfined: корутина не закреплена четко за определенным потоком или пулом потоков. Она запускается в текущем потоке до первой приостановки. После возобновления работы корутина продолжает работу в одном из потоков, который сторого не фиксирован. Разработчики языка Kotlin в обычной ситуации не рекомендуют использовать данный тип.

- newSingleThreadContext и newFixedThreadPoolContext: позволяют вручную задать поток/пул для выполнения корутины

И мы можем сами задать для корутины диспетчер, передав в функцию launch (а также async) соответствующее значение:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    launch(Dispatchers.Default) {   // явным образом определяем диспетчер
Dispatcher.Default
        println("Корутина выполняется на потоке: ${Thread.currentThread().name}")
    }
    println("Функция main выполняется на потоке: ${Thread.currentThread().name}")
}
```

Dispatchers.Unconfined

Тип Dispatchers.Unconfined запускает корутину в текущем вызывающем потоке до первой приостановки. После возобновления корутина продолжает работу в одном из потоков, который строго не фиксирован. Подобный тип подходит для корутин, которым не требуется интенсивно потреблять время CPU или работать с общими данными, наподобие объектов пользовательского интерфейса. Применим данный тип:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    launch(Dispatchers.Unconfined) {
        println("Поток корутины (до остановки): ${Thread.currentThread().name}")
        delay(500L)
        println("Поток корутины (после остановки):
${Thread.currentThread().name}")
    }

    println("Поток функции main: ${Thread.currentThread().name}")
}
```

Консольный вывод:

```
Поток корутины (до остановки): main
Поток функции main: main
Поток корутины (после остановки): kotlinx.coroutines.DefaultExecutor
```

newSingleThreadContext

newSingleThreadContext вручную запускает поток с указанным именем:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    launch(newSingleThreadContext("Custom Thread")) {
        println("Поток корутины: ${Thread.currentThread().name}")
    }

    println("Поток функции main: ${Thread.currentThread().name}")
}
```

В данном случае для выполнения корутины будет запускаться поток с именем "Custom Thread". Консольный вывод:

```
Поток функции main: main
Поток корутины: Custom Thread
```

В то же время выделенный поток является довольно затратным ресурсом. И в реальном приложении подобый поток следует либо освобождать с помощью функции close(), если он больше не нужен, либо хранить в глобальной переменной и использовать его повторно для подобных задач на протяжении работы приложения.

Отмена выполнения корутин

При работе приложения может сложиться необходимость отменить выполнение корутины. Например, в мобильном приложении запущена корутина для загрузки данных с некоторого интернет-ресуса, но пользователь решил перейти к другой странице приложения, и ему больше не нужны эти данные. В этом случае чтобы зря не тратить ресурсу системы, мы можем предусмотреть отмену выполнения корутины.

Для отмены выполнения корутины у объекта Job может применяться метод cancel():

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    val downloader: Job = launch{
        println("Начинаем загрузку файлов")
        for(i in 1..5){
            println("Загружен файл $i")
            delay(500L)
        }
    }
    delay(800L)     // установим задержку, чтобы несколько файлов загрузились
```

```
    println("Надоело ждать, пока все файлы загрузятся. Прерву-ка я загрузку...")
    downloader.cancel()    // отменяем корутину
    downloader.join()      // ожидаем завершения корутины
    println("Работа программы завершена")
}
```

В данном случае определена корутина, которая имитирует загрузку файлов. В цикле пробегаемся от 1 до 5 и условно загружаем пять файлов.

Далее вызов метода downloader.cancel() сигнализирует корутине, что надо прервать выполнение. Затем с помощью метода join() ожидаем завершения корутина, которая прервана. В итоге получим консольный вывод наподобие следующего:

```
Начинаем загрузку файлов
Загружен файл 1
Загружен файл 2
Надоело ждать, пока все файлы загрузятся. Прерву-ка я загрузку...
Работа программы завершена
```

Также вместо двух методов cancel() и join() можно использовать один сборный метод cancelAndJoin():

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    val downloader: Job = launch{
        println("Начинаем загрузку файлов")
        for(i in 1..5){
            println("Загружен файл $i")
            delay(500L)
        }
    }
    delay(800L)
    println("Надоело ждать, пока все файлы загрузятся. Прерву-ка я загрузку...")
    downloader.cancelAndJoin()    // отменяем корутину и ожидаем ее завершения
    println("Работа программы завершена")
}
```

Обработка исключения CancellationException

Все suspend-функции в пакете kotlinx.coroutines являются прерываемыми (cancellable). Это значит, что они проверяют, прервана ли корутина. И если ее выполнение прервано, они генерируют исключение типа CancellationException. И в самой корутине мы можем перехватить это исключение, чтобы обработать отмену корутины. Например:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    val downloader: Job = launch{
        try {
            println("Начинаем загрузку файлов")
            for(i in 1..5){
                println("Загружен файл $i")
                delay(500L)
            }
        }
        catch (e: CancellationException ){
            println("Загрузка файлов прервана")
        }
        finally{
            println("Загрузка завершена")
        }
    }
    delay(800L)
    println("Надоело ждать. Прерву-ка я загрузку...")
    downloader.cancelAndJoin()    // отменяем корутину и ожидаем ее завершения
    println("Работа программы завершена")
}
```

Здесь код выполнения корутины обернут в конструкцию try. Если корутина будет прервана извне, то с помощью блока catch и перехвата исключения CancellationException мы сможем обработать отмену корутины.

И если нам надо выполнить некоторые завершающие действия, например, освободить используемые в корутине ресурсы - закрыть файлы, различные подключения к внешним ресурсам, то это можно сделать в блоке finally. Но в данном случае в этом блоке просто выводим диагностическое сообщение.

В итоге при вызове метода downloader.cancel() производейт отмена корутины. Будет сгенерировано исключение, и в корутине в блоке catch мы сможем ее обработать. В итоге получим следующий консольный вывод:

```
Начинаем загрузку файлов
Загружен файл 1
Загружен файл 2
Надоело ждать. Прерву-ка я загрузку...
Загрузка файлов прервана
Загрузка завершена
Работа программы завершена
```

Отмена выполнения async-корутины

Подобным образом можно отменять выполнение и корутин, создаваемых с помощью функции async(). В этом случае обычно вызов метода await() помещается в блок try:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope{

    // создаем и запускаем корутину
    val message = async {
        getMessage()
    }
    // отмена корутины
    message.cancelAndJoin()

    try {
        // ожидаем получение результата
        println("message: ${message.await()}")
    }
    catch (e:CancellationException){
        println("Coroutine has been canceled")
    }
    println("Program has finished")
}

suspend fun getMessage() : String{
    delay(500L)
    return "Hello"
}
```

Консольный вывод программы:

```
Coroutine has been canceled
Program has finished
```

Каналы

Каналы позволяют передавать потоки данных. В Kotlin каналы представлены интерфейсом Channel, у которого следует выделить два основных метода:

- abstract suspend fun send(element: E): Unit

Отправляет объект element в канал

- abstract suspend fun receive(): E

Получает данные из канала

Определим простейший канал, через который будем передавать числа типа Int:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel

suspend fun main() = coroutineScope{

    val channel = Channel<Int>()
    launch {
        for (n in 1..5) {
            // отправляем данные через канал
            channel.send(n)
        }
    }

    // получаем данные из канала
    repeat(5) {
        val number = channel.receive()
        println(number)
    }
    println("End")
}
```

Основная функциональность каналов сосредочена в пакете kotlinx.coroutines.channels, соответственно импортируем из него тип Channel:

```
import kotlinx.coroutines.channels.Channel
```

В программе определяем переменную, которая будет представлять канал:

```
val channel = Channel<Int>()
```

Поскольку через канал будут передаваться значения типа Int, то соответственно объект Channel типизирован типом Int.

Затем с помощью функции launch создаем корутину, в которой в цикле передаем в канал числа от 1 до 5:

```
launch {
    for (n in 1..5) {
        // отправляем данные через канал
        channel.send(n)
    }
}
```

В метод send() собственно передается то значение, которое мы хотим отправить через канал. Особенностью этого метода является то, что мы можем его запустить только в корутине.

Для получения данных из канала с помощью функции repeat() определяем функцию, которая будет выполнятся 5 раз - так как мы передаем в канал пять чисел:

```
repeat(5) {
    val number = channel.receive()  // получаем значения из канала
    println(number)
}
```

Метод receive() возвращает извлекаемый из канала объект.

Консольный вывод программы:

```
1
2
3
4
5
End
```

Другой пример - отправка через канал строк:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel

suspend fun main() = coroutineScope{

    val channel = Channel<String>()
    launch {
        val users = listOf("Tom", "Bob", "Sam")
        for (user in users) {
            println("Sending $user")
            channel.send(user)
        }
    }

    repeat(3) {
        val user = channel.receive()
        println("Received: $user")
    }
    println("End")
}
```

В даном случае в канал передаем три строки, соответственно функция repeat() три раза запускает получение данных из канала. Консольный вывод программы:

```
Sending Tom
Received: Tom
Sending Bob
Received: Bob
Sending Sam
Received: Sam
End
```

Закрытие канала

Чтобы указать, что в канале больше нет данных, его можно закрыть с помощью метода close(). Если для получения данных из канала применяется цикл for, то, получив сигнал о закрытии канала, данный цикл получит все ранее посланные объекты до закрытия и завершит выполнение:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel

suspend fun main() = coroutineScope{

    val channel = Channel<String>()
    launch {
        val users = listOf("Tom", "Bob", "Sam")
        for (user in users) {
            channel.send(user)  // Отправляем данные в канал
        }
        channel.close()  // Закрытие канала
    }

    for(user in channel) {  // Получаем данные из канала
        println(user)
    }
    println("End")
}
```

Паттерн producer-consumer

Рассмотренный выше пример по сути является распростаненным способом передачи данных от одной корутины к другой. И чтобы упростить написание подобного кода, Kotlin предоставляет ряд дополнительных функций. Так, функция produce() представляет построитель корутины, который создает корутину, в которой передаются данные в канал. Например, с помощью функции produce() мы можем определить новую функцию-корутину, которая будет отправлять определенные данные:

```
fun CoroutineScope.getUsers(): ReceiveChannel<String> = produce{
    val users = listOf("Tom", "Bob", "Sam")
    for (user in users) {
        send(user)
```

```
    }
}
```

Здесь определяется функция getUsers(). Причем она определяется как функция интерфейса CoroutineScope. Функция должна возвращать объект ReceiveChannel, типизированный типом передаваемых данных (в данном случае передаем значения типа String).

Функция getUsers() представляет корутину, создаваемую построителем корутин produce. В корутине опять же проходим по списку строк и с помощью функции send передаем в канал данные.

Для потребления данных из канала может применяться метод consumeEach() объекта ReceiveChannel, который по сути заменяет цикл for. Он принимает функцию, в которую в качестве параметра передается получаемый из канала объект:

```
val users = getUsers()
users.consumeEach { user -> println(user) }
```

Полный код программы:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

suspend fun main() = coroutineScope{

    val users = getUsers()
    users.consumeEach { user -> println(user) }
    println("End")
}

fun CoroutineScope.getUsers(): ReceiveChannel<String> = produce{
    val users = listOf("Tom", "Bob", "Sam")
    for (user in users) {
        send(user)
    }
}
```

Консольный вывод программы:

```
Tom
Bob
Sam
End
```