**Корутины (сопрограммы) и потоки.**

Корутина - представляет блок кода, который может выполнятся параллельно с остальным кодом. А базовая функциональность, связанная с корутинами, сосредоточена в библиотеке kotlinx.coroutines.

**1.Добавление kotlinx.coroutines.**

- 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 &quot;Project Settings&quot; 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.

- 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\IdeaIC[номер\_верси

и]\plugins\Kotlin\kotlinc\lib Далее в этой папке выберем библиотеку kotlinx-coroutines-core-jvm.jar и нажмем на OK для ее добавления.

**2.Определение suspend-функции**

Сначала рассмотрим пример, который не использует корутины:

import kotlinx.coroutines.\*

suspend fun main(){

for(i in 0..5){

delay(800L)

println(i)

}

println(&quot;привет, корутины&quot;)

}

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

800 миллисекунд перед выводом на консоль текущего элемента

последовательности.

После выполнения работы цикла выводим на консоль строку «привет, корутины»

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.

**3.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(800L)

println(i)

}

}

println(«привет, корутины»

}

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** – метод, запускающий определенную в лямбде сопрограмму

без блокировки главного потока. Она создает корутину в виде блока кода - в

данном случае это:

{

for(i in 0..5){

delay(500L)

println(i)

}

}

и запускает эту корутину параллельно с остальным кодом. То есть данная

корутина выполняется независимо от прочего кода, определенного в

функции main.

**4.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{ doCorut() }

println(&quot;привет, корутины &quot;)

}

suspend fun doCorut(){

for(i in 0..5){

println(i)

delay(500L)

}

}

В данном случае основной код корутины вынесен в функцию doCorut().

Поскольку в этой функции применяется функция delay(), то doCorut()

определена с модификатором suspend. Сама корутина создается также с

помощью функции launch(), которая вызывает функцию doCorut().

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(500L)

}

}

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)

}

}

}

**GlobalScope** – объект для запуска сопрограмм верхнего уровня.

**runBlocking** – блокировка потока, чтобы оставить программу на выполнение.

**5.Корутины и потоки**

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.