**Практическое занятие 12**

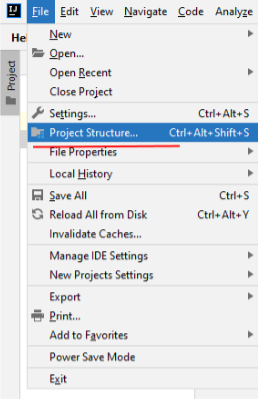
**Корутины(сопрограммы) и потоки.**

**1.Введение в корутины**

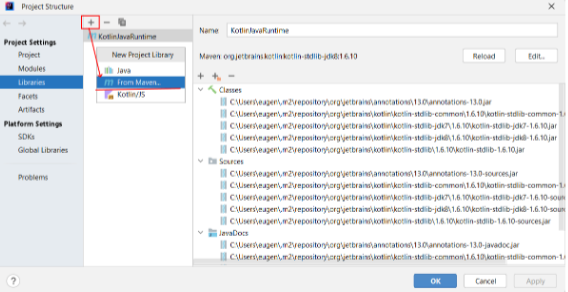
В языке **Kotlin** поддержка асинхронности и параллельных вычислений воплощена в виде корутин (**coroutine**). По сути корутина представляет блок кода, который может выполняться параллельно с остальным кодом. А базовая функциональность, связанная с корутинами, сосредоточена в библиотеке **kotlinx.coroutines**.

**1а.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**..

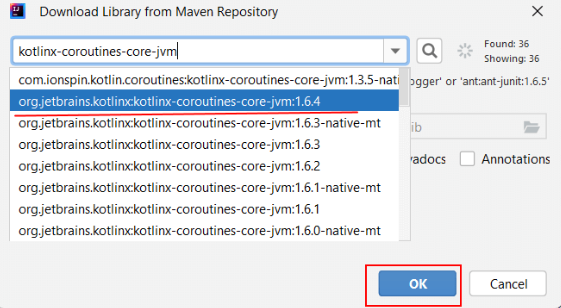


Далее на вкладке “**Project Settings**” перейдем к пункту **Libraries**. В центральном поле отобразятся библиотеки, добавленные в проект.

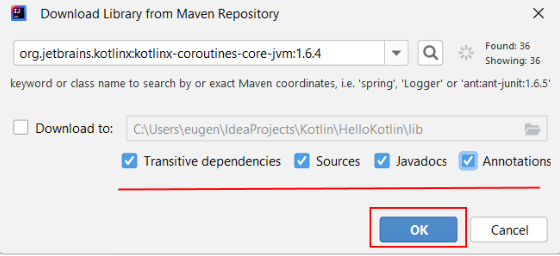


И для добавления новой библиотеки нажмем на знак плюса и в контекстном меню выберем пункт **From Maven**...

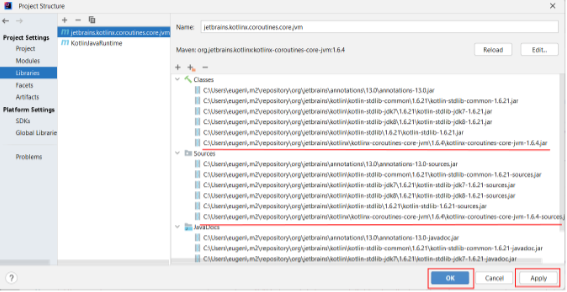
После этого нам откроется окно для добавления библиотеки через **Maven**. В этом окне в поле ввода введем название нужной нам библиотеки - **kotlinx- coroutines-core-jvm** и нажмем на кнопку поиска. Если соответствующая библиотека найдена, то нам отобразится выпадающий список с результатами



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 её файл в списке библиотек



**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)

}

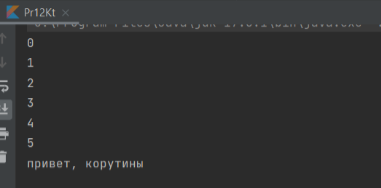
Println(”привет, корутины”)

}

Для имитации продолжительной работы внутри цикла вызываем специальную функцию **delay()** из пакета **kotlinx.coroutines**. В эту функцию передается количество миллисекунд, на которое выполняется задержка.

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.

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



**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**.

**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){

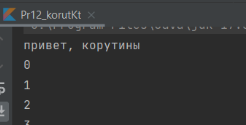
delay(500L)

println(i)

}

}

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



Теперь строка “**привет, корутины**” не ожидает, пока завершится цикл, а выполняется параллельно с ним.

**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(“привет, корутины”)

}

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 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(500L)

}

}

Unit

}

Либо можно типизировать функцию **coroutineScope** типом **Unit**:

import kotlinx.coroutines.\*

suspend fun main() = coroutineScope<Unit>{

launch{

for(i in 0..5){

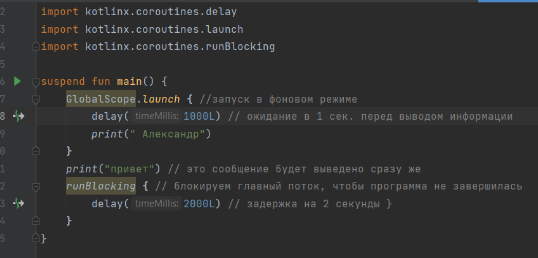
println(i)

delay(400L)

}

}

}



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

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

**5.Корутины и потоки**

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.