Kotlin

Coroutines Labs

Nội dung

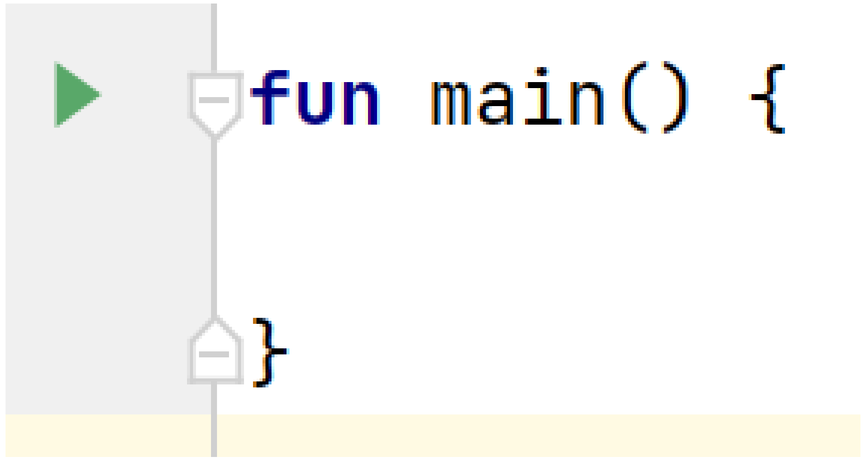
Bước 1

- Tạo Project Kotlin coroutine example
- Thêm các dependencies vào app/build.gradle.kt

implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")

Bước 2.

- Tạo package firstcoroutines, tạo file BuildFirstCoroutines.kt trong package này.
- Tạo hàm main.
- Ấn nút mũi tên xanh và chạy Run (hoặc ấn Ctrl+Shift+F10).



```
fun main() {


}
```

Lab 1. Chương trình coroutine đầu tiên

```
package
vn.edu.hust.soict.gv.quangnh.coroutineexample.firstcoroutines

import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

fun main() {
    GlobalScope.launch {
        delay(1000)
        print("Hello ")
    }
    print("World ")
    Thread.sleep(2000)
}
```

Lab 2

- Tạo coroutine dùng runBlocking: tạo ra coroutine và block thread hiện tại

```
fun main() {
    runBlocking {
        delay(1000)
        println("Hello ")
        delay(1000)
        println("World ")
    }
    println("After runBlocking")
}
```

a. Xác định kết quả của chương trình này

b. Hãy hiển thị tên tiến trình của từng đoạn code trong chương trình

```
println("Current Thread: ${Thread.currentThread().name}")
```

Bài 2. Coroutine Context

2.1. Dispatchers

- Dispatchers: quyết định Thread mà Coroutine chạy
  - Dispatchers.Default
  - Dispatchers.IO: đọc Database, Networking
  - Dispatchers.Main: update UI
  - Dispatchers.Unconfined

Lab 3: Dispatcher

- Tạo package coroutinecontext
- Tạo file TestDispatchers:

```
package
vn.edu.hust.soict.gv.quangnh.coroutineexample.coroutinecontext

import android.util.Log
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.launch
import
vn.edu.hust.soict.gv.quangnh.coroutineexample.MainActivity
```

```
object TestDispatchers {
    fun runMyFirstCoroutines() {
        GlobalScope.launch(Dispatchers.Default) {
            Log.d(MainActivity::class.java.simpleName, "Dispatchers Default run on ${Thread.currentThread().name}")
        }
        GlobalScope.launch(Dispatchers.IO) {
            Log.d(MainActivity::class.java.simpleName, "Dispatchers IO run on ${Thread.currentThread().name}")
        }
        GlobalScope.launch(Dispatchers.Unconfined) {
            Log.d(MainActivity::class.java.simpleName, "Dispatchers Unconfined run on ${Thread.currentThread().name}")
        }
        GlobalScope.launch(Dispatchers.Main) {
            Log.d(MainActivity::class.java.simpleName, "Dispatchers Main run on ${Thread.currentThread().name}")
        }
    }
}
```

File MainActivity.kt

```
package vn.edu.hust.soict.gv.quangnh.coroutineexample

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import
vn.edu.hust.soict.gv.quangnh.coroutineexample.coroutineconte
xt.TestDispatchers

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        TestDispatchers.runMyFirstCoroutines()
    }
}
```

Nhận xét

- Các luồng chạy bất đồng bộ, không theo đúng thứ tự code
- Dispatchers Unconfined và Main đều chạy trên Main thread. Tuy nhiên nếu Dispatchers Unconfined chạy quá lâu thì sẽ được chuyển sang Thread mới.

Ví dụ 4

```
object TestDispachers {
    fun runMyFirstCoroutines() {
        GlobalScope.launch(Dispatchers.Unconfined) {
            Log.d(MainActivity::class.java.simpleName, "Before delay - 
Dispachers Unconfined run on ${Thread.currentThread().name}")
            delay(1000)
            Log.d(MainActivity::class.java.simpleName, "Dispachers 
Unconfined run on ${Thread.currentThread().name}")
        }
        GlobalScope.launch(Dispatchers.Main) {
            Log.d(MainActivity::class.java.simpleName, "Dispachers Main run 
on ${Thread.currentThread().name}")
        }
    }
}
```

2.2. withContext Ví dụ 5a

```
fun testMySecondWithContext() {
    GlobalScope.launch(Dispatchers.IO) {
        // Run long time task
        Log.d("myLog", "Run long time task - Thread:
${Thread.currentThread().name}")
        delay(2000)
        withContext(Dispatchers.Main) {
            // Update UI here
            Log.d("myLog", "Update UI - Thread:
${Thread.currentThread().name}")
        }
    }
}
```

2.2. withContext. Ví dụ 5b

```
fun testMySecondWithContext() {
    var n : Int = 10
    GlobalScope.launch(Dispatchers.IO) {
        // Run long time task
        Log.d("myLog", "Run long time task - Thread: ${Thread.currentThread().name}")
        delay(2000)
        n += 20
        withContext(Dispatchers.Main) {
            // Update UI here
            Log.d("myLog", "Update UI - Thread: ${Thread.currentThread().name}  n = $n")
        }
    }
}
```

2.3. Job. Ví dụ 6

```
package
vn.edu.hust.soict.gv.quangnh.coroutineexample.coroutinecontext

import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.Job
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
```

```kotlin
fun main() {
    val job1: Job = GlobalScope.launch {
        delay(2000)
        println("Hello Kotlin")
    }

    val job2: Job = GlobalScope.launch {
        // job2 chờ đợi công việc của job1 hoàn thành rồi mới thực hiện
        job1.join()
        delay(1000)
        println("I'm Coroutine")
    }
    Thread.sleep(4000)
}
```

Cancel coroutine. Ví dụ 7

```
fun main() {
    runBlocking {
        val job = launch(Dispatchers.Default) {
            repeat(1000) {
                delay(500)
                println("I'm sleeping $it ...")
            }
        }
        delay(1500)
        job.cancel()
        print("Cancelled coroutines")
    }
}
```

Hàm cancelAndJoin(). Ví dụ 8

```
fun main() {
    runBlocking {
        val startTime = System.currentTimeMillis()
        val job = launch(Dispatchers.Default) {
            var nextPrintTime = startTime
            var i = 0
            while (i < 5) { // computation loop, just waste CPU
                // print a message twice a second
                if (System.currentTimeMillis() >= nextPrintTime) {
                    println("job: I'm sleeping ${i++} ...")
                    nextPrintTime += 500
                }
            }
        }
        delay(1400) // delay a bit
        println("main: I'm tired of waiting")
        job.cancelAndJoin() // cancel the job and waits for its completion
        println("main: Now I can quit")
    }
```

Biến isActive. Ví dụ 9

```
fun main() {
    runBlocking {
        val startTime = System.currentTimeMillis()
        val job = launch(Dispatchers.Default) {
            var nextPrintTime = startTime
            var i = 0
            while (isActive) { // computation loop, just waste CPU
                // print a message twice a second
                if (System.currentTimeMillis() >= nextPrintTime) {
                    println("job: I'm sleeping ${i++} ...")
                    nextPrintTime += 500
                }
            }
        }
        delay(1400) // delay a bit
        println("main: I'm tired of waiting")
        job.cancelAndJoin() // cancel the job and waits for its completion
        println("main: Now I can quit")
    }
```

Coroutine với try … catch … finally

Ví dụ 10

```
fun main() {
    runBlocking {
        val job = launch {
            try {
                repeat(1000) {
                    delay(100)
                    println("Hello Coroutine")
                }
            } finally {
                println("Print from finally")
            }

        }
        delay(300)
        println("I want stop coroutine")
        job.cancel()
    }
}
```

Hàm delay trong khối finally Ví dụ 11

```
fun main() {
    runBlocking {
        val job = launch {
            try {
                repeat(1000) {
                    delay(100)
                    println("Hello Coroutine")
                }
            } finally {
                println("Print from finally")
                delay(100)
                println("Please print me last times")
            }

        }
        delay(300)
        println("I want stop coroutine")
        job.cancel()
    }
}
```

Hàm withContext(NonCancellable) Ví dụ 12

```
fun main() {
    runBlocking {
        val job = launch {
            try {
                repeat(1000) {
                    delay(100)
                    println("Hello Coroutine")
                }
            } finally {
                println("Print from finally")
                withContext(NonCancellable) {
                    repeat(2) {
                        delay(100)
                        println("Print from NonCancellable")
                    }
                }
            }

        }
        delay(300)
        println("I want stop coroutine")
        job.cancel()
    }
}
```

2.4. Timeouts. Ví dụ 13

```
fun main() {
    runBlocking {
        withTimeout(1800) {
            repeat(1000) {
                println("I'm sleeping $it")
                delay(500)

            }
        }
    }
}
```

Nghĩa là coroutine này chỉ chạy tối đa 1800 ms.

Xử lý Exception bằng withTimeoutOrNull Ví dụ 14

```
fun main() {
    runBlocking {
        val result = withTimeoutOrNull(1800) {
            repeat(1000) {
                println("I'm sleeping $it")
                delay(500)
            }
            "Done"
        }
        println("Result = $result")
    }
}
```

Nếu thời gian chạy coroutine ít hơn thời gian Timeout. Ví dụ 15

```
fun main() {
    runBlocking {
        val result = withTimeoutOrNull(1800) {
            repeat(2) {
                println("I'm sleeping $it")
                delay(500)
            }
            "Done"
        }
        println("Result = $result")
    }
}
```

3. Async và Await. Ví dụ 16

```
package
vn.edu.hust.soict.gv.quangnh.coroutineexample.async_await

import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking
import kotlin.system.measureTimeMillis
```

```kotlin
fun main() {
    runBlocking {
        val time = measureTimeMillis {
            val a = doSomethingFunny1()
            val b = doSomethingFunny2()
            println("a + b = ${a + b}")
        }
        println("Time = $time")
    }
}
suspend fun doSomethingFunny1(): Int {
    delay(1000)
    return 10
}

suspend fun doSomethingFunny2(): Int {
    delay(1000)
    return 20
}
```

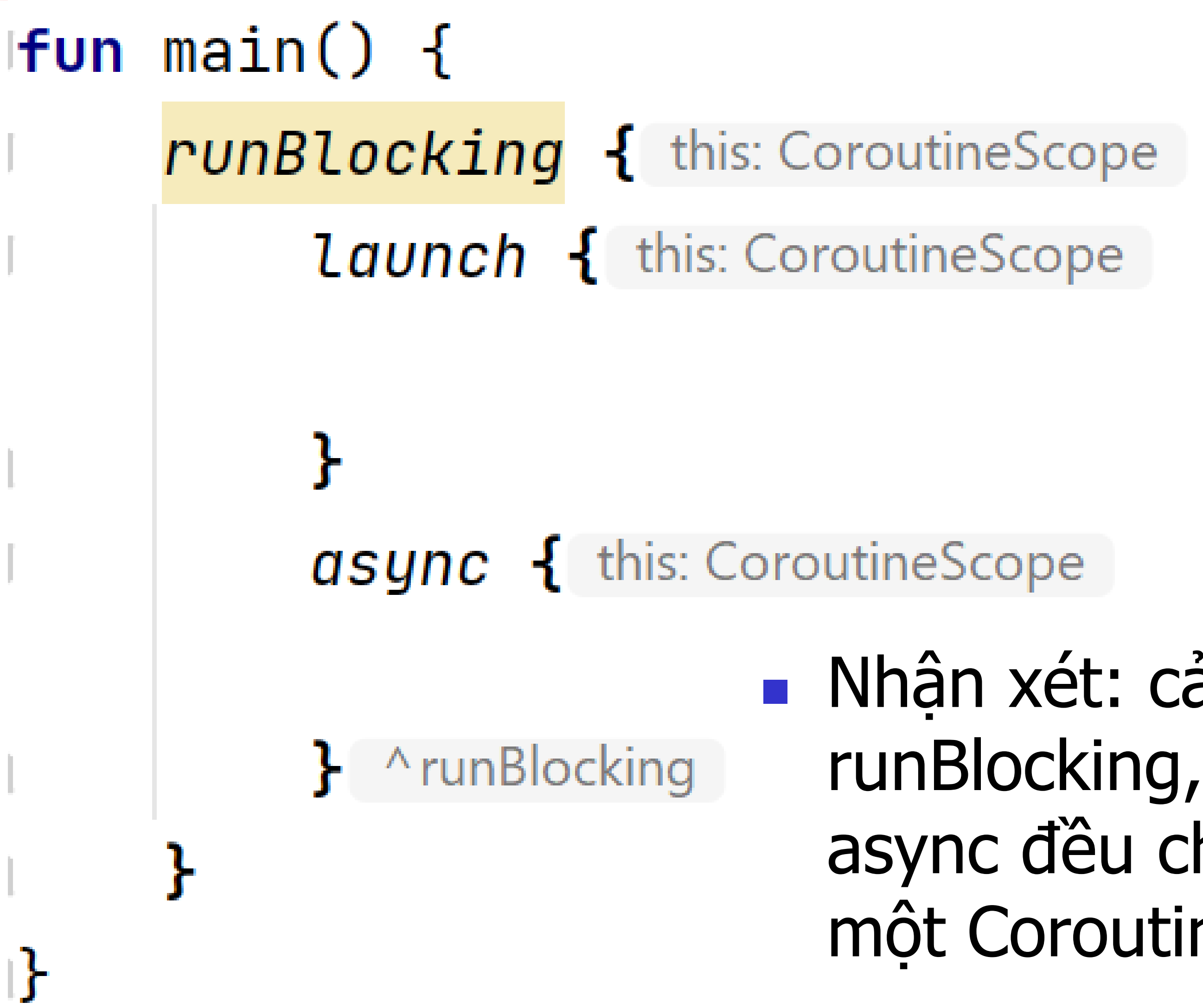
Ví dụ 17: async - await

```
import kotlinx.coroutines.Deferred
import kotlinx.coroutines.async
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking
import kotlin.system.measureTimeMillis
```

```
fun main() {
    runBlocking {
        val time = measureTimeMillis {
            val a: Deferred<Int> = async { doSomethingFunny1() }
            val b: Deferred<Int> = async { doSomethingFunny2() }
            println(a.await() + b.await())
        }
        println("Time = $time")
    }
}
suspend fun doSomethingFunny1(): Int {
    delay(1000)
    return 10
}

suspend fun doSomethingFunny2(): Int {
    delay(1000)
    return 20
}
```

4. CoroutineScope



```
fun main() {
    runBlocking { this: CoroutineScope
        launch { this: CoroutineScope


        }
        async { this: CoroutineScope


        } ^runBlocking
    }
}
```

- Nhận xét: cả runBlocking, launch và async đều chạy trong một CoroutineScope

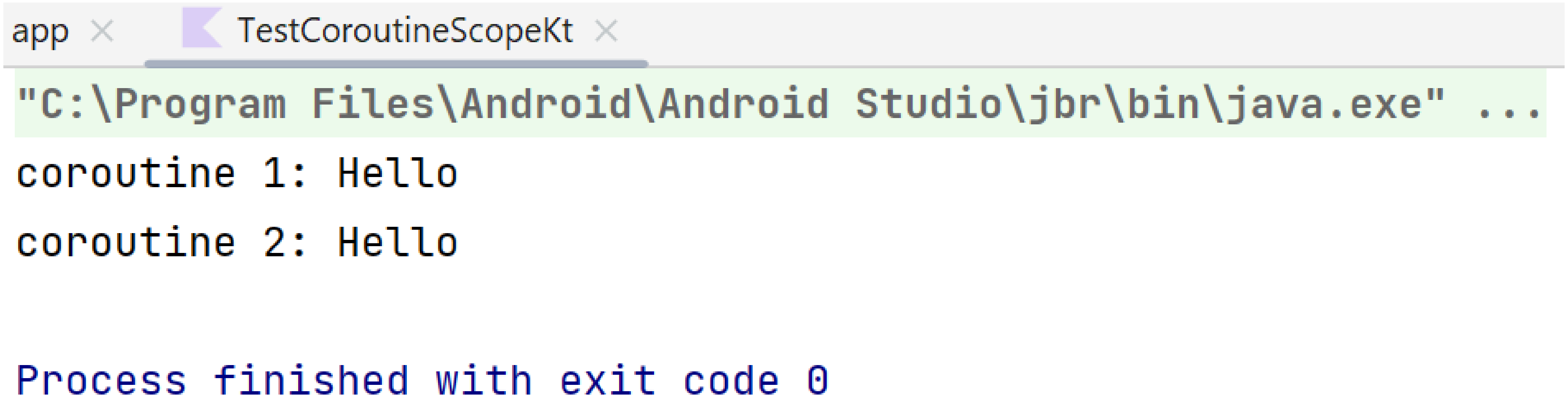
Ví dụ 18

```
fun main() {
  runBlocking {

    val job1 = launch {
      launch {
        delay(100)
        println("coroutine 1: Hello")
        delay(1000)
        println("coroutine 1: Goodbye")
      }
      launch {
        delay(100)
        println("coroutine 2: Hello")
        delay(1000)
        println("coroutine 2: Goodbye")
      }
    }
    delay(500)
    job1.cancel()

  }
}
```

Kết quả



app × TestCoroutineScopeKt ×

```
"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" ...
coroutine 1: Hello
coroutine 2: Hello


Process finished with exit code 0
```

- Nhận xét: coroutine cha bị cancel thì coroutine con cũng bị hủy theo.
- Nếu tác vụ nhất thiết phải hoàn thành kể cả khi coroutine cha bị hủy thì dùng GlobalScope.

Ví dụ 19

```
fun main() {
    runBlocking {

        val job1 = launch {
            launch {
                delay(100)
                println("coroutine 1: Hello")
                delay(1000)
                println("coroutine 1: Goodbye")
            }
            launch {
                delay(100)
                println("coroutine 2: Hello")
                delay(1000)
                println("coroutine 2: Goodbye")
            }
            GlobalScope.launch {
                delay(100)
                println("coroutine 3: Hello")
                delay(1000)
                println("coroutine 3: Goodbye")
            }
        }
        delay(500)
        job1.cancel()
        delay(2500)

    }
}
```

Ví dụ 20

```
fun main() {
    runBlocking {
        val job = launch {
            repeat(3) {
                delay(100)
                println("coroutine: $it")
            }
            println("Print from parent")
        }
        job.join()
        delay(1000)
    }
}
```

Ví dụ 21

```
fun main() {
    runBlocking {
        val job = launch {
            repeat(3) {
                launch {
                    delay(100)
                    println("coroutine: $it")
                }
            }
            println("Print from parent")
        }
        job.join()
        delay(1000)
    }
}
```

5. Xử lý Exception và Supervision trong Coroutine

Ví dụ 22

```
fun main() {
    runBlocking {
        val job = GlobalScope.launch {
            println("Throw Exception from Launch")
            throw NullPointerException()
        }
        // chờ đợi coroutine hoàn thành
        job.join()
        val deferred = GlobalScope.async {
            println("Throw Exception from Async")
            throw IndexOutOfBoundsException()
        }
    }
}
```

Khi thêm câu lệnh await() Ví dụ 23

```
fun main() {
    runBlocking {
        val job = GlobalScope.launch {
            println("Throw Exception from Launch")
            throw NullPointerException()
        }
        // chờ đợi coroutine hoàn thành
        job.join()
        val deferred = GlobalScope.async {
            println("Throw Exception from Async")
            throw IndexOutOfBoundsException()
        }
        deferred.await()
    }
}
```

Xử lý lỗi trong coroutine dùng try ... catch Ví dụ 24

```
fun main() {
    runBlocking {
        val job = GlobalScope.launch {
            try {
                println("Throw Exception from Launch")
                throw NullPointerException()
            } catch (e: NullPointerException) {
                println(e.toString())
            }
        }
        // chờ đợi coroutine hoàn thành
        job.join()
        val deferred = GlobalScope.async {
            println("Throw Exception from Async")
            throw IndexOutOfBoundsException()
        }
        try {
            deferred.await()
        } catch (e: IndexOutOfBoundsException) {
            println(e.toString())
        }
```

Bắt lỗi với CoroutineExceptionHandler Ví dụ 25

```
fun main() {
  runBlocking {
      val handler = CoroutineExceptionHandler { _, exception ->
          println("Error here: ${exception.toString()}")
      }
      val job = GlobalScope.launch(handler) {
          println("Throw Exception from Launch")
          throw NullPointerException()
      }
      // chờ đợi coroutine hoàn thành
      job.join()
      val deferred = GlobalScope.async {
          println("Throw Exception from Async")
          throw IndexOutOfBoundsException()
      }
      try {
          deferred.await()
      }catch (e: IndexOutOfBoundsException) {
          println(e.toString())
      }
  }
}
```

Bắt lỗi + chỉ định context

```
val job = GlobalScope.launch(handler + Dispatchers.Default) {
    println("Throw Exception from Launch")
    throw NullPointerException()
}
```

CoroutineExceptionHandler không bắt được lỗi với async Ví dụ 26

```
fun main() {
    runBlocking {
        val handler = CoroutineExceptionHandler { _, exception ->
            println("Error here: ${exception.toString()}")
        }
        val job = GlobalScope.launch(handler + Dispatchers.Default) {
            println("Throw Exception from Launch")
            throw NullPointerException()
        }
        // chờ đợi coroutine hoàn thành
        job.join()
        val deferred = GlobalScope.async(handler) {
            println("Throw Exception from Async")
            throw IndexOutOfBoundsException()
        }
        deferred.await()
    }
}
```

Nếu trong Coroutine cha có nhiều coroutine con, và các coroutine con có khả năng tạo ra các lỗi. Ví dụ 27

- Khi Coroutine thứ 2 throw Exception thì các coroutine khác sẽ dừng.

```
fun main() {
    runBlocking {
        val handle = CoroutineExceptionHandler {_, exception ->
            println("Exception: $exception")
        }
        val job = GlobalScope.launch(handle) {
            launch {
                println("Coroutine 1")
                delay(300)
                println("Coroutine 1 continue")
                throw IndexOutOfBoundsException("Coroutine 1")
            }
launch {
            println("Coroutine 2")
            delay(200)
            throw NullPointerException("Coroutine 2")
        }
        launch {
            println("Coroutine 3")
            delay(400)
            println("Coroutine 3 continue")
            throw ArithmeticException("Coroutine 3")
        }
    }
    job.join()
    delay(1000)
  } // end of runBlocking
}
```

Bắt lỗi với suppressed. Ví dụ 28

```
fun main() {
    runBlocking {
        val handle = CoroutineExceptionHandler {_, exception ->
            println("Exception: $exception with suppressed
${exception.suppressed.contentToString()}")
        }
        val job = GlobalScope.launch(handle) {
            launch {
                println("Coroutine 1")
                delay(300)
                println("Coroutine 1 continue")
                throw IndexOutOfBoundsException("Coroutine 1")
            }
launch {
        try {
            delay(Long.MAX_VALUE)
        } finally {
            throw ArithmeticException("Coroutine 2")
        }
    }

    launch {
        println("Coroutine 3")
        delay(400)
        println("Coroutine 3 continue")
        throw ArithmeticException("Coroutine 3")
    }
  }
  job.join()
  delay(1000)
} // end of runBlocking
}
```

SupervisorJob và SupervisorScope Ví dụ 29

```
fun main() {
    runBlocking {
        val supervisorJob = SupervisorJob()
        with(CoroutineScope(coroutineContext +
supervisorJob)) {
            val firstChild = launch {
                println("Print from First Child")
                throw NullPointerException()
            }
val secondChild = launch {
            firstChild.join()
            println("print from second Child. First Child is Active: ${firstChild.isActive}")
            try {
                delay(1000)
            } finally {
                println("Second Child Cancelled")
            }
        }
        firstChild.join()
        println("Cancelling SupervisorJob")
        supervisorJob.cancel()
        secondChild.join()
    }
  }
}
```

SupervisorScope Ví dụ 30

```
fun main() {
  runBlocking {
    supervisorScope {
      val firstChild = launch {
        println("Print from First Child")
        throw NullPointerException()
      }
      val secondChild = launch {
        firstChild.join()
        println("print from second Child. First Child is Active:
${firstChild.isActive}")
        try {
          delay(1000)
        } finally {
          println("Second Child Cancelled")
        }
      }
      firstChild.join()
      secondChild.join()
    }
  }
}
```

6. Sequence trong Kotlin Ví dụ 31

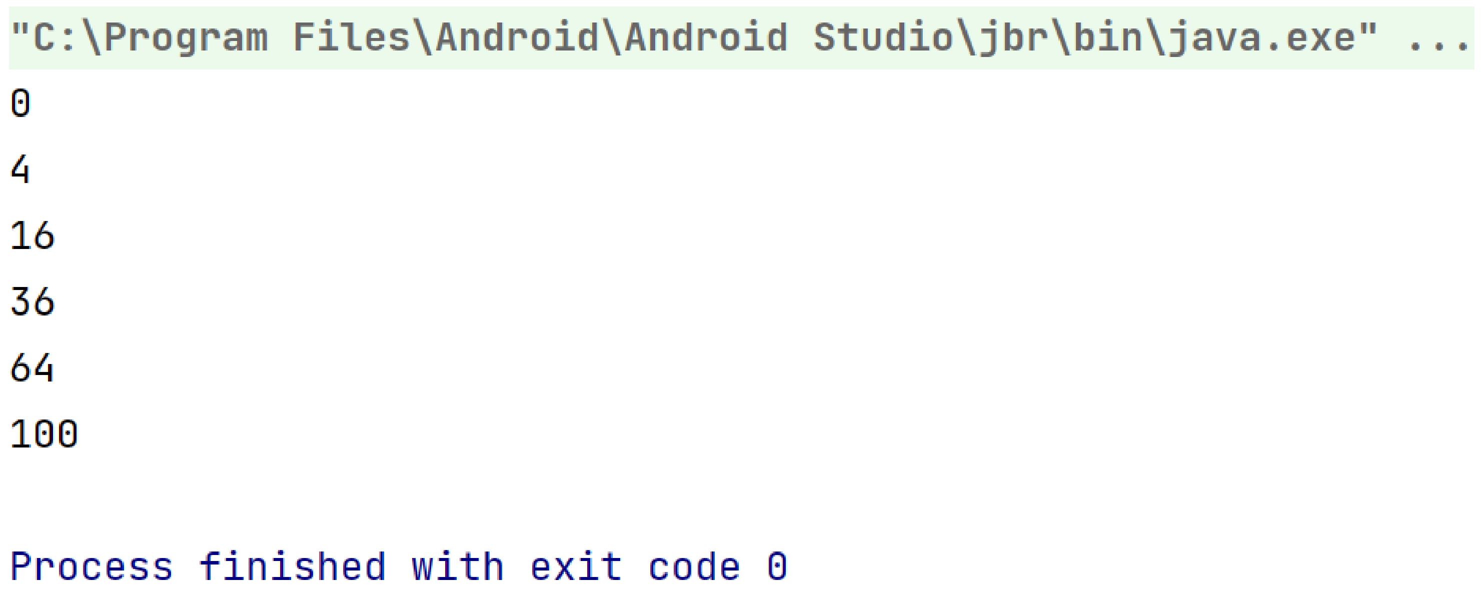
```
fun foo(n: Int) : Sequence<Int>  = sequence {
    for (i in 0..n) {
        if (i % 2 == 0)
            yield(i)
    }
}

fun main() {
    foo(10).forEach {
        println(it)
    }
}
```

Kết hợp sequence với map

```
fun main() {
    foo(10).map{it * it}.forEach {
        println(it)
    }
}
```

- Kết quả



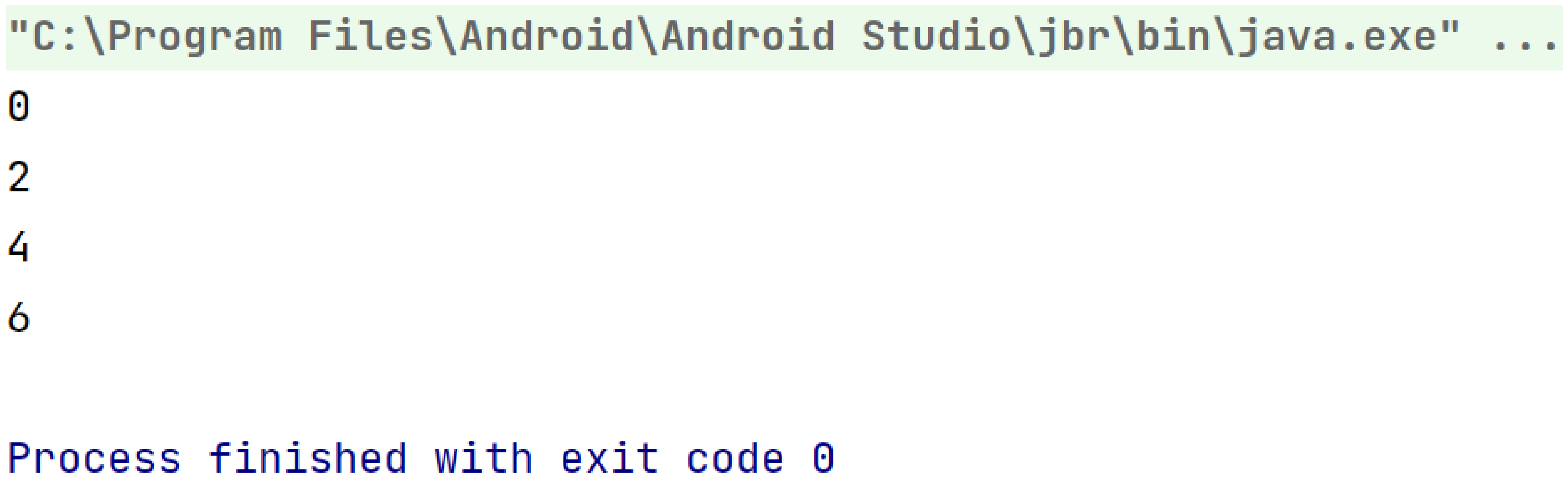
```
"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" ...
0
4
16
36
64
100

Process finished with exit code 0
```

Kết hợp sequence với filter

```kotlin
fun main() {
    foo(10).filter { it < 8 }.forEach {
        println(it)
    }
}
```

- Kết quả



```
"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" ...
0
2
4
6

Process finished with exit code 0
```

7. Giới thiệu về Flow trong Kotlin Coroutines Ví dụ 32

```
fun main() {
    runBlocking {
        val foo = foo(200)
        foo(5).collect {
            println("i = $it")
        }
    }
}

fun foo(n : Int): Flow<Int> = flow {
    for(i in 0..n) {
        delay(1000)
        emit(i)
    }
}
```

Flow Cancellation - Ví dụ 33

```
fun main() {
    runBlocking {
        withTimeoutOrNull(3000) {
            foo(10).collect {
                println("i = $it")
            }
        }
    }
}fun foo(n : Int) : Flow<Int> = flow {
    for (i in 0..n) {
        delay(1000)
        emit(i)
    }
}
```

8. Các toán tử trong Flow

- Hàm transform - Ví dụ 34

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        // transform
        // biến đổi giá trị trước khi thêm vào list kết quả
        list.asFlow().transform {
            emit(it * it)
        }.collect {
            println("value = $it")
        }

    }
}
```

Transform có thể emit nhiều giá trị Ví dụ 35

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        // transform
        // biến đổi giá trị trước khi thêm vào list kết quả
        list.asFlow().transform {
            emit(it * it)
            emit(it * it * it)
        }.collect {
            println("value = $it")
        }
    }
}
```

Hàm map - Ví dụ 36

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        // map: chỉ biến đổi được một giá trị
        list.asFlow().map {
            it * it
        }.collect {
            println("value = $it")
        }
    }
}
```

Hàm take - Ví dụ 37

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        // take: lấy số lượng giá trị nhất định. Ví dụ như lấy 3 giá
trị đầu tiên
        list.asFlow().take(3).collect {
            println("value = $it")
        }
    }
}
```

Hàm filter - Ví dụ 38

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        // filter: lọc Flow theo các điều kiện. Ví dụ: lọc lấy các
số chẵn
        list.asFlow().filter {
            it % 2 == 0
        }.collect {
            println("value = $it")
        }
    }
}
```

Hàm reduce - Ví dụ 39

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        // reduce: Tính cộng dồn các phần tử của Flow.
        // Ví dụ: tính tổng các phần tử trong list
        val sum: Int = list.asFlow().reduce { accumulator, value ->
            println("accumulatior = $accumulator and value = 
$value")
            accumulator + value
        }
        println("sum = $sum")
    }
}
```

Hàm fold - Ví dụ 40

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        // fold: giống như reduce nhưng có thể đặt giá trị khởi đầu
        val sum: Int = list.asFlow().fold(5) { accumulator, value ->
            println("accumulatior = $accumulator and value = 
$value")
            accumulator + value
        }
        println("sum = $sum")
    }
}
```

Kết hợp nhiều hàm với nhau
Ví dụ 41

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        list.asFlow().filter {
            it % 2 == 0
        }.map {it * 2 }
            .take(2)
            .collect {
                println("value = $it")
            }

    }
}
```

Hàm single() and singleOrNull()

- Kiểm tra nếu Flow có đúng một giá trị. Nếu Flow có số lượng giá trị emit > 1 hay < 1 thì sẽ throw Exception.
- Ví dụ 42

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        (1..10).asFlow().single()
    }
}
```

singleOrNull(): nếu list rỗng thì trả về null
Ví dụ 43

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        //(1..10).asFlow().single()
        //listOf<Int>().asFlow().single()
        val a = listOf<Int>().asFlow().singleOrNull()
        println(a)
    }
}
```

Hàm zip: Kết hợp hai Flow với nhau Ví dụ 44

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        val nums = (1..3).asFlow()
        val strs = listOf("one", "two", "three").asFlow()
        nums.zip(strs) { num, str ->
            "(num = $num and str = $str)"
        }.collect {
            println(it)
        }
    }
}
```

Hàm combine - Ví dụ 45

```
fun main() {
  val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
  runBlocking {
    val nums = (1..3).asFlow().onEach {
      delay(100)
    }
    val strs = listOf("one", "two", "three").asFlow().onEach {
      delay(400)
    }
    val startTime = System.currentTimeMillis()
    nums.zip(strs) { num, str ->
      "(num = $num and str = $str)"
    }.collect {
      println("value = $it at ${System.currentTimeMillis() - startTime} ")
    }
  }
}
```

Hàm combine - Ví dụ 46

```
startTime = System.currentTimeMillis()
nums.combine(strs) { num, str ->
    "(num = $num and str = $str)"
}.collect {
    println("value = $it at ${System.currentTimeMillis() -
startTime} ")
}
```

```
fun main() {
    val list: List<Int> = listOf<Int>(1, 8, 9, 3, 6, 7, 2)
    runBlocking {
        val nums = (1..3).asFlow().onEach {
            delay(100)
        }
        val strs = listOf("one", "two", "three").asFlow().onEach {
            delay(400)
        }

        var startTime = System.currentTimeMillis()
        nums.zip(strs) { num, str ->
            "(num = $num and str = $str)"
        }.collect {
            println("value = $it at ${System.currentTimeMillis() - startTime} ")
        }
        println("==============================")
        startTime = System.currentTimeMillis()
        nums.combine(strs) { num, str ->
            "(num = $num and str = $str)"
        }.collect {
            println("value = $it at ${System.currentTimeMillis() - startTime} ")
        }
    }
}
```