Kotlin

Isaac Hennessey | Daniel Abbott
Dean Maynard | Isaac Letcher

OVERVIEW

1. What is Kotlin?
2. Runnable Interface
3. Unsynchronized Run
4. Synchronized Run
5. Priorities Run

What is Kotlin?

- Development started in 2010 by JetBrains – first stable release in 2016 and announced as preferred android language in 2019
- Runs on the JVM – transforms Kotlin source code into Java bytecode
- Amazon, Google, Cash App all implement Kotlin in their backend development

Kotlin pros

- Statically Typed
- Concise
- Open Source
- IntelliJ IDEA is made for Kotlin

Kotlin Cons

- Slower compilation
- Steep learning curve
- High potential complexity

RUNNABLE INTERFACE

- The runnable interface gives each thread a task to complete
- Each task will print the thread name, priority, and an iteration of 0 - 5

- Runnable helps specify the code a thread should run via its run() method
- We MUST override the run function
- Separates task logic from thread management

```
3   class Twentylines(val name : String) : Runnable {  3 Usages
4       override fun run() {  3 Usages
5           println("$name running")
6           println("Thread priority: ${Thread.currentThread().priority}")
7           for (i in 0 ≤ .. ≤ 5) { println(i)
8           }
9       }
10  }
```

UNSYNCHRONIZED RUN & LAMBDA FUNCTIONS

Unsynchronized Run

- No order to which thread enters CPU
- Each runs concurrently
- Threads start in the NEW state and then get moved to the RUNNABLE state
- When the threads are finished, they are TERMINATED

Lambda Functions

- Instead of writing a full class or function, lambdas provide a shortcut
- It's an anonymous function
- Result implicitly returned

```
println("=========UNSYNC=========")
val tasks = ArrayList<Thread>()
repeat(5) { tasks.add(thread { Twentylines("Thread$it").run() }) }
```

Nested lambda Function

SYNCHRONIZED RUN

Reentrant Lock

- Is a mutual exclusion lock
- Thread holding the lock is allowed to execute the critical section
- Thread unlocks when finished
- When a thread acquires a lock, it remains in the RUNNABLE state and stays in the JVM Ready Queue

Conditions

- Conditions set the state of the threads
- await() puts the threads in a WAITING state
- .signalAll() moves the threads from WAITING --> BLOCKED

```
println("====SYNC====")
val lock = ReentrantLock()
val condition = lock.newCondition()
var count = 0
repeat( times = 5) { i ->
    tasks.add(thread {
        lock.lock()
        try {
            while (count != i) {
                condition.await()
            }; Twentylines( name = "task $i").run(); count++; condition.signalAll()
        } finally {
            lock.unlock()
        }
    })
}
```

Try / Finally

- If an exception occurs in our critical section, the lock is always released
- This prevents deadlock since some threads would be blocked forever

CRITICAL SECTION

```
lock.lock()
try {
    while (count != i) {
        condition.await()
    }; Twentylines("task $i").run(); count++; condition.signalAll()
} finally {
    lock.unlock()
}
```

- The mutual exclusion lock ensures that only one thread can access the critical section at a time
- Checking the count, running our task, incrementing count, and signaling the threads all happen one at a time
- Without a mutual exclusion lock we are exposed to a race condition

PRIORITY RUN

- Every thread has a set priority
- Priorities can be a range of integers from 1 – 10
- Default priority of any thread is 5
- Threads are in the NEW state until they are started, which puts them in the RUNNABLE state

```
35      tasks.clear()
36      println("====priorities====")
37      repeat( times = 5) { i ->
38          tasks.add(thread(start = false) { Twentylines( name = "Thread$i").run() })
39
40      }
41      tasks[0].priority = Thread.MIN_PRIORITY
42      tasks[1].priority = 3
43      tasks[2].priority = 5
44      tasks[3].priority = 8
45      tasks[4].priority = Thread.MAX_PRIORITY
46
47      tasks.forEach{ it.start() }
48      tasks.forEach{ it.join() }
49  }
```

- Priorities can be set using x.priority = y
- To obtain the current priority you can call, Thread.currentThread().*priority*
- *MAX_PRIORITY = 10 , NORM_PRIORITY = 5 , MIN_PRIORITY = 1*

Does setting priorities make a difference?

```
3   class Twentylines(val name : String) : Runnable {  3 Usages
4       override fun run() {  3 Usages
5           println("$name running")
6           println("Thread priority: ${Thread.currentThread().priority}")
7           for (i in 0 ≤ .. ≤ 5) { println(i)
8           }
9       }
10  }
```

OUTPUTS

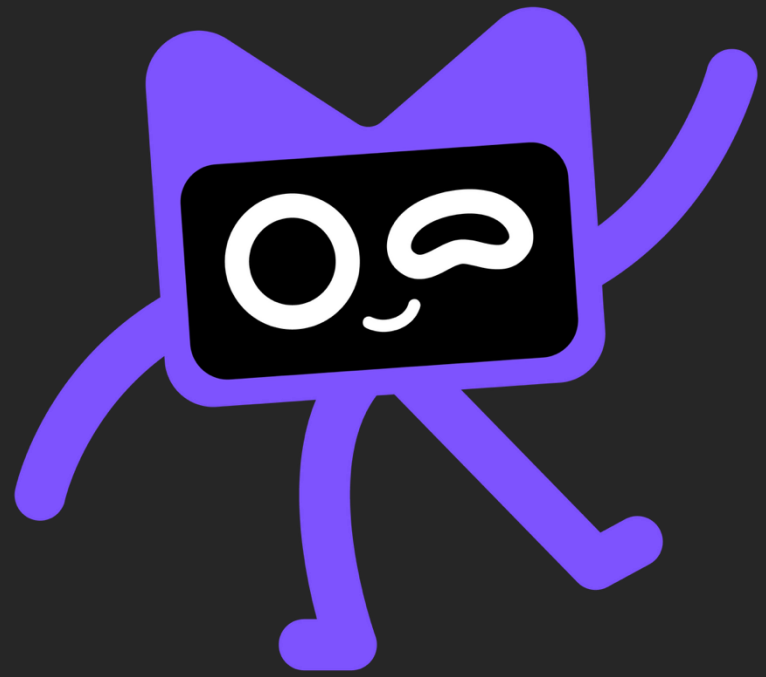
```
====UNSYNC====
Thread0 running
Thread2 running
Thread priority: 5
Thread priority: 5
Thread1 running
Thread priority: 5
Thread3 running
Thread priority: 5
Thread4 running
Thread priority: 5
```

```
====SYNC====
task 0 running
Thread priority: 5
task 1 running
Thread priority: 5
task 2 running
Thread priority: 5
task 3 running
Thread priority: 5
task 4 running
Thread priority: 5
```

```
====priorities====
Thread0 running
Thread priority: 1
Thread1 running
Thread priority: 3
Thread3 running
Thread priority: 8
Thread2 running
Thread priority: 5
Thread4 running
Thread priority: 10
```

Problems & What We Learned

- Kotlin basics
- MacOS
- Lambda Functions
- Priorities
- Critical Section
- Race Condition



CONCLUSION

1. What is Kotlin?
2. Runnable Interface
3. Unsynchronous Run
4. Synchronized Run
5. Priorities Run

QUESTIONS?

Isaac Hennessey | Daniel Abbott
Dean Maynard | Isaac Letcher