COROUTINES

CoroutineScope

To start coroutine scope you can:
Use GlobalScope that has empty coroutine context.
Implement CoroutineScope interface.
Create a scope from a context:
with(CoroutineScope(context = context)) { ... }

Coroutine builders

launch - Launches new coroutine without blocking current thread and returns a reference to the coroutine as a Job.
runBlocking - Runs new coroutine and blocks current thread interruptible until its completion.
async - Creates new coroutine and returns its future result as an implementation of Deferred.
withContext - Change a coroutine context for some block.

Coroutine context

It is an indexed set of Element instances where every element in this set has a unique Key.

EmptyCoroutineContext - Does not change coroutine behavior at all. Like an empty map.
CoroutineName - Sets a name of a coroutine for debugging purposes.
Job - Lifecycle of a coroutine. Can be used to cancel coroutine. A coroutine is responsible for all children with the same Job. It waits for them and cancels all of them if any had an error (To make children independent use SupervisorJob).
CoroutineExceptionHandler - Used to set exception handling for uncaught exceptions.
ContinuationInterceptor - Intercepts continuation. Mainly used by dispatchers.

Channels

fun CoroutineScope.produceSquares():
    ReceiveChannel<Int> = produce {
        for (x in 1..5) send(x * x)
    }

val squares = produceSquares()
repeat(5) { println(squares.receive()) } // 1, 4, 9, 16, 25
val squares2 = produceSquares()
for(square in squares2) print(square) // 1, 4, 9, 16, 25

Coroutine dispatchers

Dispatchers.Default - Different thread (if possible)
It is backed by a shared pool of threads on JVM.
Dispatchers.Main - Platform specific main thread (if exists).
Dispatchers.IO - Thread designed for offloading blocking IO tasks to a shared pool of threads.
Dispatchers.Unconfined - Always uses first available thread (most performant dispatcher).
newSingleThreadContext - Creates a new coroutine execution context using a single thread with built-in yield support.
newFixedThreadPoolContext - Creates new coroutine execution context with the fixed-size thread-pool and built-in yield support.

Sequence builder

val childNumbers = sequence {
    yield(1)
    print("AAA")
    yieldAll(listOf(2, 3))
}
childNumbers.forEach { println(it) } // 1AAA23

val nums = childNumbers.joinToString() // AAA print(nums) // 1, 2, 3

Deal with shared state

AtomicInteger - There are atomics for primitives.
AtomicReference<V> - Atomic reference.
Mutex - Does not let more than one thread at the same time.
private val mutex = Mutex()
mutex.withLock { /* */ }

Actors

sealed class Msg
object IncCounter: Msg()
object PrintCounter: Msg()
class GetCounter(val resp: CompletableDeferred<Int>): Msg()

fun CoroutineScope.counterActor() = actor<Msg> {
    var counter = 0 // Actor state
    for (msg in channel) {
        when (msg) {
            is IncCounter -> counter++
            is PrintCounter -> print(counter)
            is GetCounter -> msg.resp.complete(counter)
        }
    }
}