{ this is Kotlin }

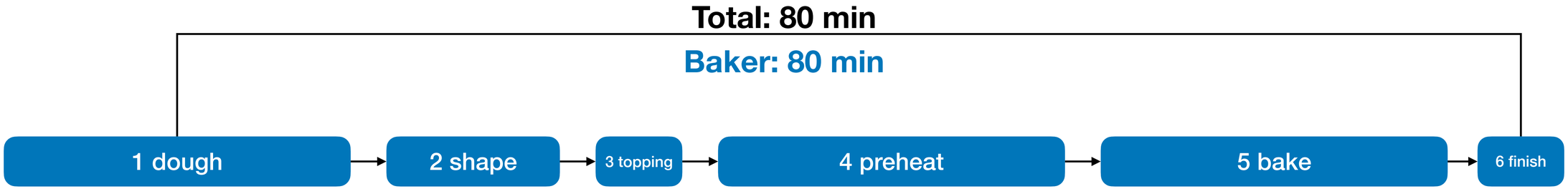
Coroutines

Tiberiu Tofan

Sync/Async | Concurrent | Parallel

synchronous

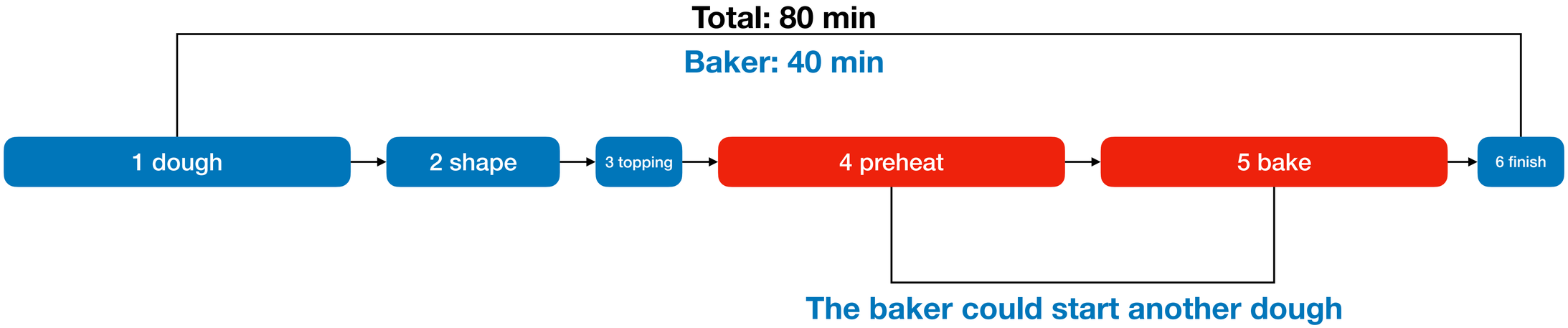
Pretzel recipe:
1.prepare the dough (20 min)
2.shape the pretzels (10 min)
3.prepare the topping (5 min)
4.preheat the oven (20 min)
5.bake the pretzels (20 min)
6.add the topping (5 min)



Sync/Async|Concurrent/Parallel

asynchronous & sequential

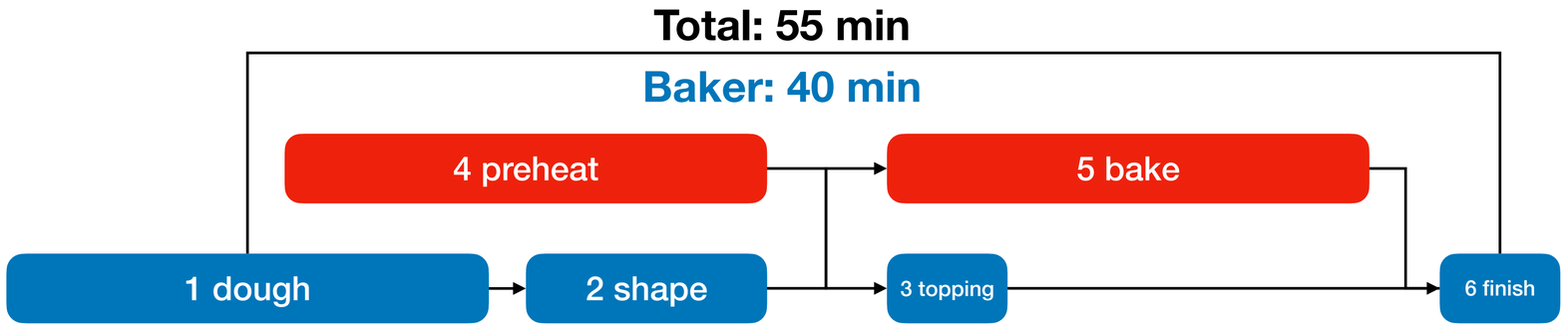
Pretzel recipe:
1.prepare the dough (20 min)
2.shape the pretzels (10 min)
3.prepare the topping (5 min)
4.preheat the oven (20 min)
5.bake the pretzels (20 min)
6.add the topping (5 min)



Sync/Async|Concurrent/Parallel

asynchronous & concurrent

Pretzel recipe:
1.prepare the dough (20 min)
2.shape the pretzels (10 min)
3.prepare the topping (5 min)
4.preheat the oven (20 min)
5.bake the pretzels (20 min)
6.add the topping (5 min)

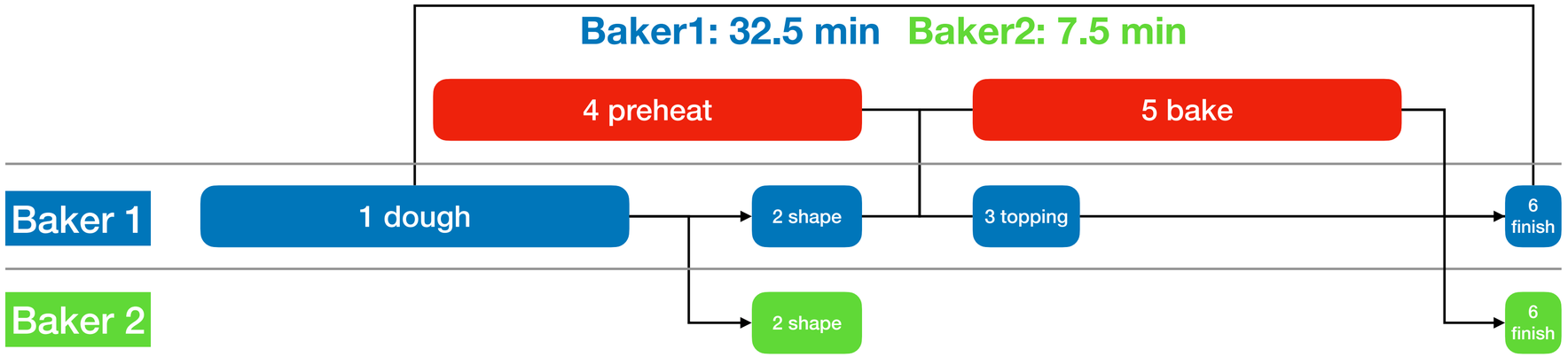


Sync/Async|Concurrent/Parallel

parallel

Pretzel recipe:
1.prepare the dough (20 min)
2.shape the pretzels (10 min)
3.prepare the topping (5 min)
4.preheat the oven (20 min)
5.bake the pretzels (20 min)
6.add the topping (5 min)

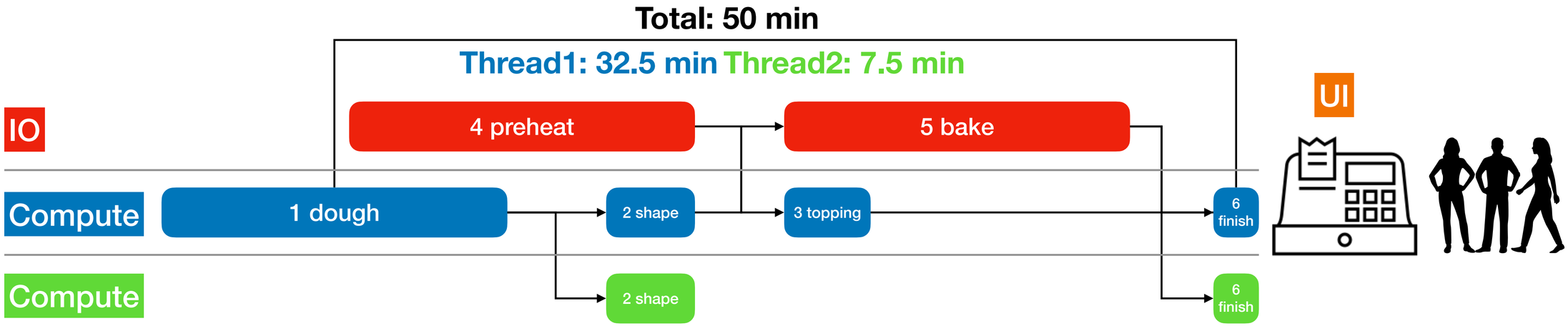
Total: 50 min



Sync/Async|Concurrent/Parallel

parallel

Pretzel recipe:
1.prepare the dough (20 min)
2.shape the pretzels (10 min)
3.prepare the topping (5 min)
4.preheat the oven (20 min)
5.bake the pretzels (20 min)
6.add the topping (5 min)



Suspend Functions

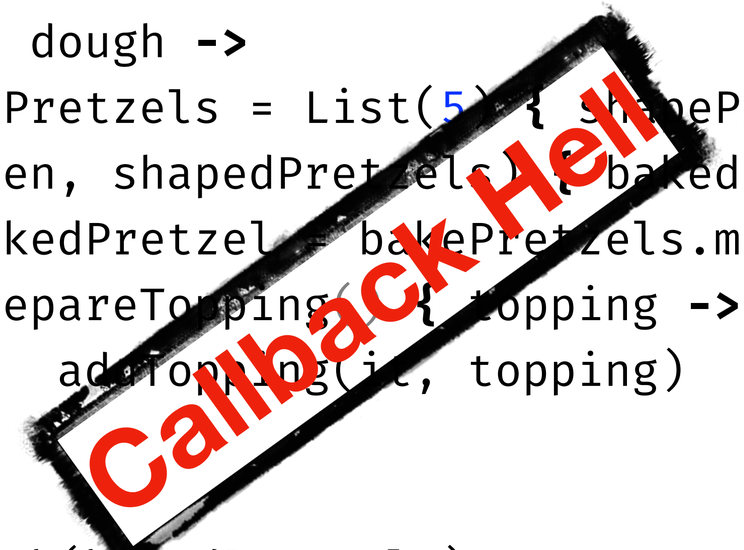
how can we make this code async?

```
fun bakePretzels(): List<FinishedPretzel> {
    val oven = preheatOven(ColdOven)
    val dough = prepareDough()
    val shapedPretzels: List<UncookedPretzel> = List(5) { shapePretzel(dough) }
    val bakedPretzels: List<CookedPretzel> = bake(oven, shapedPretzels)
    val topping: Topping = prepareTopping()
    return bakedPretzels.map { finishPretzel(it, topping) }
}
```

Suspend Functions

how can we make this code async?

```
fun bakePretzels(callback: (List<FinishedPretzel>) -> Unit) {
    preheatOven(ColdOven) { hotOven ->
        prepareDough { dough ->
            val shapedPretzels = List(5) { shapePretzel(dough) } //*this is tricky
            bake(hotOven, shapedPretzels) { bakedPretzels ->
                val bakedPretzel = bakePretzels.map { //*this is tricky
                    prepareTopping { topping ->
                        addTopping(it, topping)
                    }
                }
                callback(bakedPretzels)
            }
        }
    }
}
```



* pseudocode - it wouldn't actually work because handling collections is a lot more complicated

Suspend Functions

how can we make this code async?

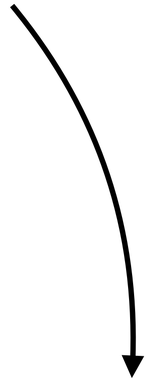
Add **suspend** modifier to all the functions marked in green

```
suspend fun bakePretzels(): List<FinishedPretzel> {
    val oven = preheatOven(ColdOven)
    val dough = prepareDough()
    val shapedPretzels: List<UncookedPretzel> = List(5) { shapePretzel(dough) }
    val bakedPretzels: List<CookedPretzel> = bake(oven, shapedPretzels)
    val topping: Topping = prepareTopping()
    return bakedPretzels.map { finishPretzel(it, topping) }
}
```

Continuation Passing Style

just another way of saying callback

```
suspend fun bakePretzels(): List<FinishedPretzel>
```



```
fun bakePretzels(continuation: Continuation<List<FinishedPretzel>>): Any
```

`suspend` functions are Asynchronous

Concurrency
with threads

```
fun main() {
    val time = measureTimeMillis {
        thread {
            val pretzels = bakePretzels()
            println("Baked ${pretzels.size} pretzels")
        }.join()
    }
    println("finished baking in $time ms")
}
```

```
Baked 5 pretzels
finished baking in 792 ms
```

Concurrency
with coroutines

alternative:
`suspend fun main`

starts a coroutine and blocks the main thread until the coroutine finishes

coroutine builder that fires and forgets (launches a coroutine and returns a job)

```
fun main() = runBlocking {
    val time = measureTimeMillis {
        launch {
            val pretzels = bakePretzels()
            println("Baked ${pretzels.size} pretzels")
        }.join()
    }
    println("finished baking in $time ms")
}
```

```
Baked 5 pretzels
finished baking in 812 ms
```

Concurrency
with coroutines

```
fun main() = runBlocking {
    val time = measureTimeMillis {
        List(100_000) {
            launch(Dispatchers.Default) {
                bakePretzels()
            }
        }.joinAll()
    }
    println("finished baking 500.000 pretzels in $time ms")
}
```

do not try this with threads!

`bakePretzels` is still sequential, but it can serve a lot of concurrent calls

```
finished baking 500.000 pretzels in 7829 ms
```

Coroutines

- an instance of a suspendable computation
- it takes a block of code that runs concurrently with the rest of the code
- not bound to a thread - after a suspend can resume execution in another thread
- it can suspend without blocking a thread
- can be thought of as light-weight threads

Composing suspend Functions
sequential by default

```
suspend fun bakePretzels(): List<FinishedPretzel> {
    val oven = preheatOven(ColdOven)
    val dough = prepareDough()
    val shapedPretzels: List<UncookedPretzel> = List(5) { shapePretzel(dough) }
    val bakedPretzels: List<CookedPretzel> = bake(oven, shapedPretzels)
    val topping: Topping = prepareTopping()
    return bakedPretzels.map { finishPretzel(it, topping) }
}
```

Composing suspend Functions
sequential by default

```
suspend fun bakePretzels(): List<FinishedPretzel> {
    val oven = preheatOven(ColdOven)
    val dough = prepareDough()
    val shapedPretzels: List<UncookedPretzel> = List(5) { shapePretzel(dough) }
    val bakedPretzels: List<CookedPretzel> = bake(oven, shapedPretzels)
    val topping: Topping = prepareTopping()
    return bakedPretzels.map { finishPretzel(it, topping) }
}
```

Composing suspend Functions
sequential by default

```
suspend fun bakePretzels(): List<FinishedPretzel> {
    val oven = preheatOven(ColdOven)
    val dough = prepareDough()
    val shapedPretzels: List<UncookedPretzel> = List(5) { shapePretzel(dough) }
    val bakedPretzels: List<CookedPretzel> = bake(oven, shapedPretzels)
    val topping: Topping = prepareTopping()
    return bakedPretzels.map { finishPretzel(it, topping) }
}
```

Composing suspend Functions
sequential by default

```
suspend fun bakePretzels(): List<FinishedPretzel> {
    val oven = preheatOven(ColdOven)
    val dough = prepareDough()
    val shapedPretzels: List<UncookedPretzel> = List(5) { shapePretzel(dough) }
    val bakedPretzels: List<CookedPretzel> = bake(oven, shapedPretzels)
    val topping: Topping = prepareTopping()
    return bakedPretzels.map { finishPretzel(it, topping) }
}
```

Composing suspend Functions

explicit concurrency via async

another coroutine builder that returns a `Deferred<T>`

```
suspend fun bakePretzels(): List<FinishedPretzel> = coroutineScope {
    val oven = async { preheatOven(ColdOven) }
    val dough = async { prepareDough() }
    val uncookedPretzels = List(5) { async { shapePretzel(dough.await()) } }
    val bakedPretzels = async { bake(oven.await(), uncookedPretzels.awaitAll()) }
    val topping = async { prepareTopping() }
    bakedPretzels.await().map { finishPretzel(it, topping.await()) }
}
```

`async` launches concurrently: we need to call `await` to wait for the result

Coroutine Context

Coroutine Context

dispatcher

- dispatchers determine which threads are used for coroutine execution
  - `Dispatchers.Main` (Android specific) - interact with the UI
  - `Dispatchers.IO` - optimized for disk/network IO and blocking operations
  - `Dispatchers.Default` - optimized for CPU intensive work
  - `Dispatchers.Unconfined` - starts the coroutine in the current thread, resumes after suspension depending on the suspension function (might change thread)

Coroutine Context

choosing a dispatcher

```
launch(Dispatchers.Default) {
    val pretzels = bakePretzels()
    //rest of the code
}
```

Coroutine Context

choosing a dispatcher

```
val pretzels = async(Dispatchers.Default) {
    bakePretzels()
}
```

Coroutine Context

choosing a dispatcher

changes the context of execution for this suspend function

```
suspend fun preheatOven(oven: Oven): HotOven = withContext(Dispatchers.IO) {
    when (oven) {
        is ColdOven -> {
            delay(PREHEAT_OVEN) //network call or file operations
            HotOven
        }
        is HotOven -> oven
    }
}
```

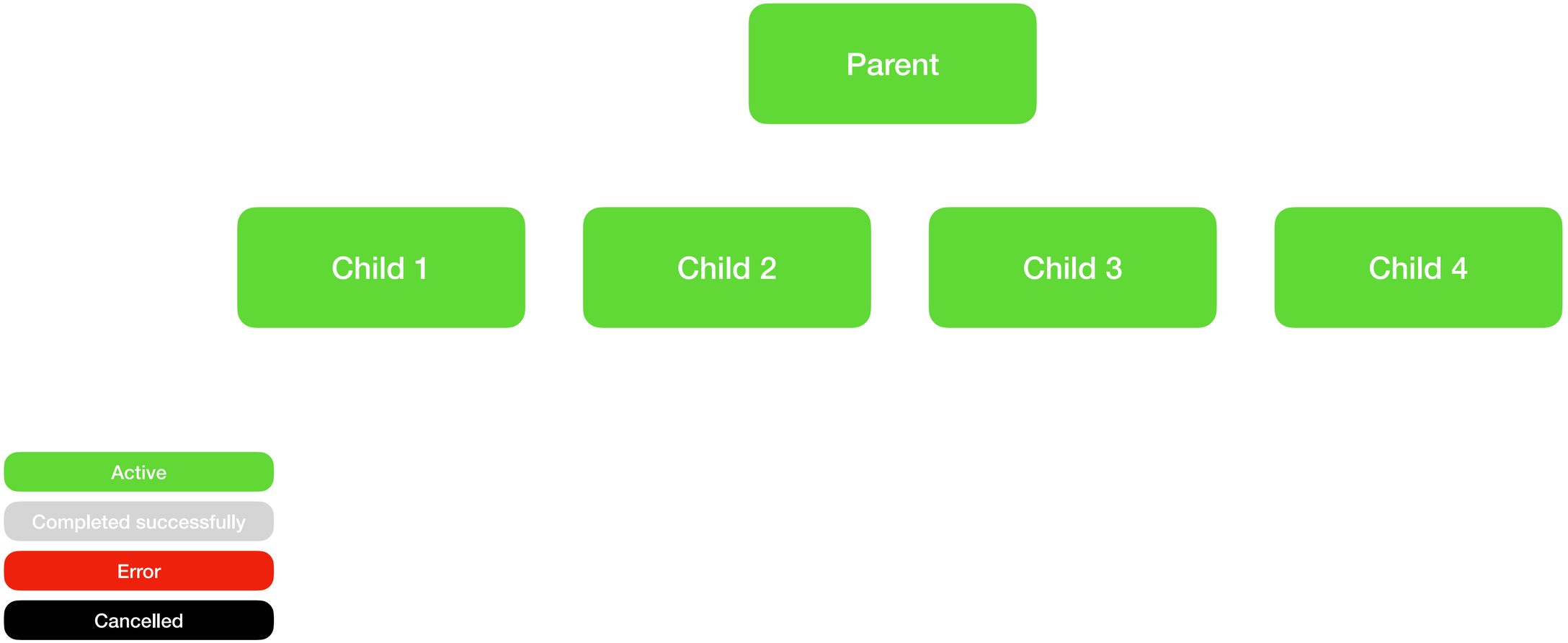
Coroutine Scope

structured concurrency

- coroutines can be launched only in a `CoroutineScope`
- `coroutineScope` builder create a coroutine scope that does not complete until all the launched children complete
- `coroutineScope` builder is suspended - does not block a thread while waiting
- a coroutine launched in the scope of another coroutine inherits the context and the job (the job of the new coroutine becomes the child of the job of the old coroutine)
- if the parent job is canceled all the children of that job are cancelled, recursively
- if an exception is thrown and not cough, the job is cancelled
- prevents coroutine leaks

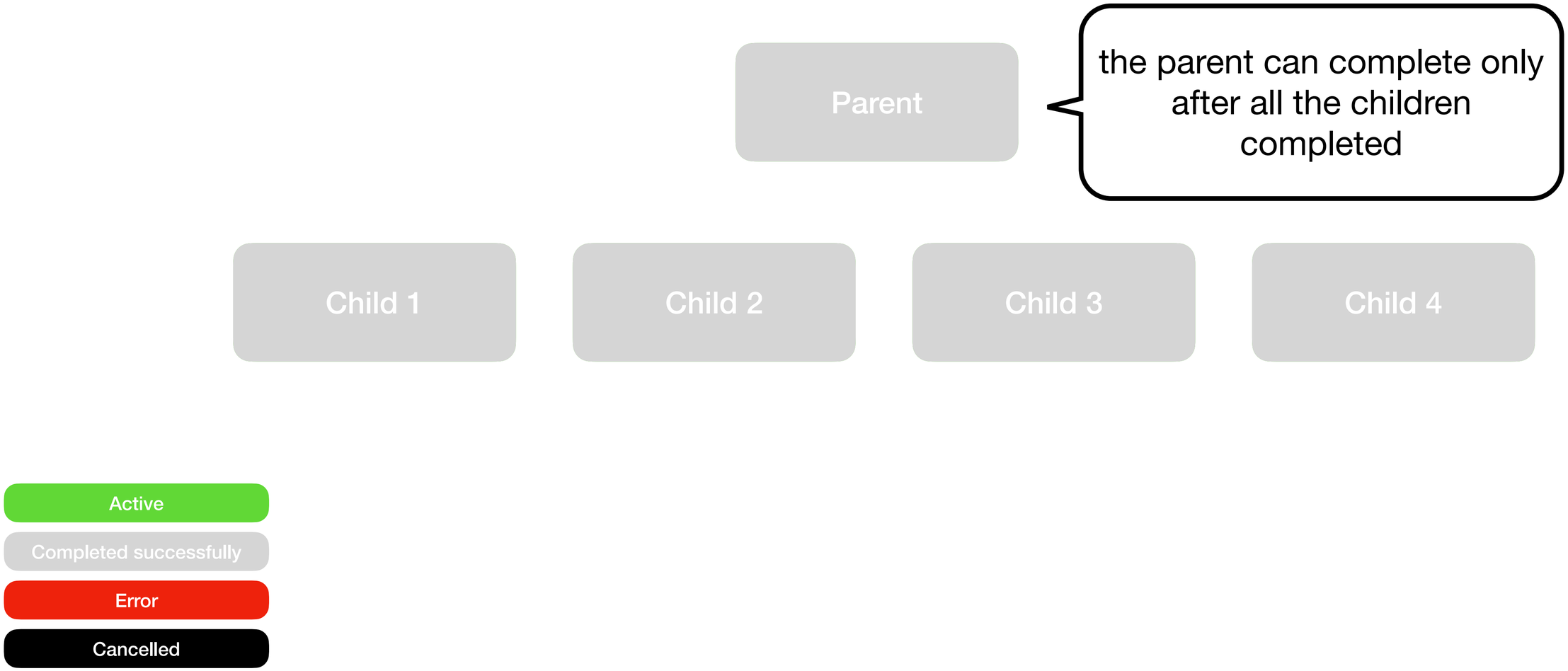
Coroutine Scope

structured concurrency



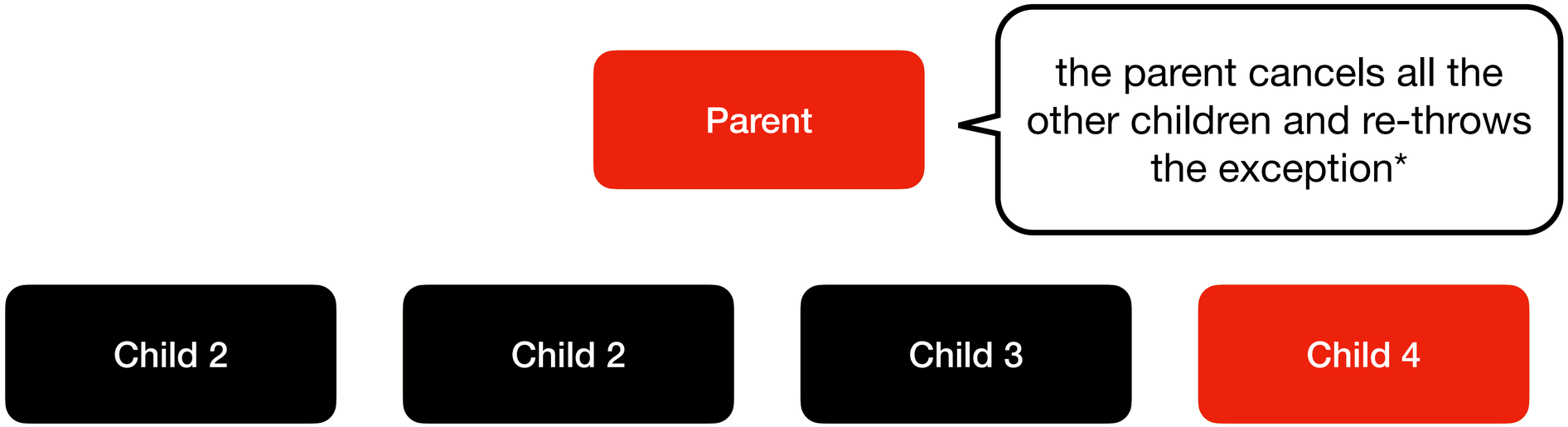
Coroutine Scope

structured concurrency



Coroutine Scope

structured concurrency



Active

Completed successfully

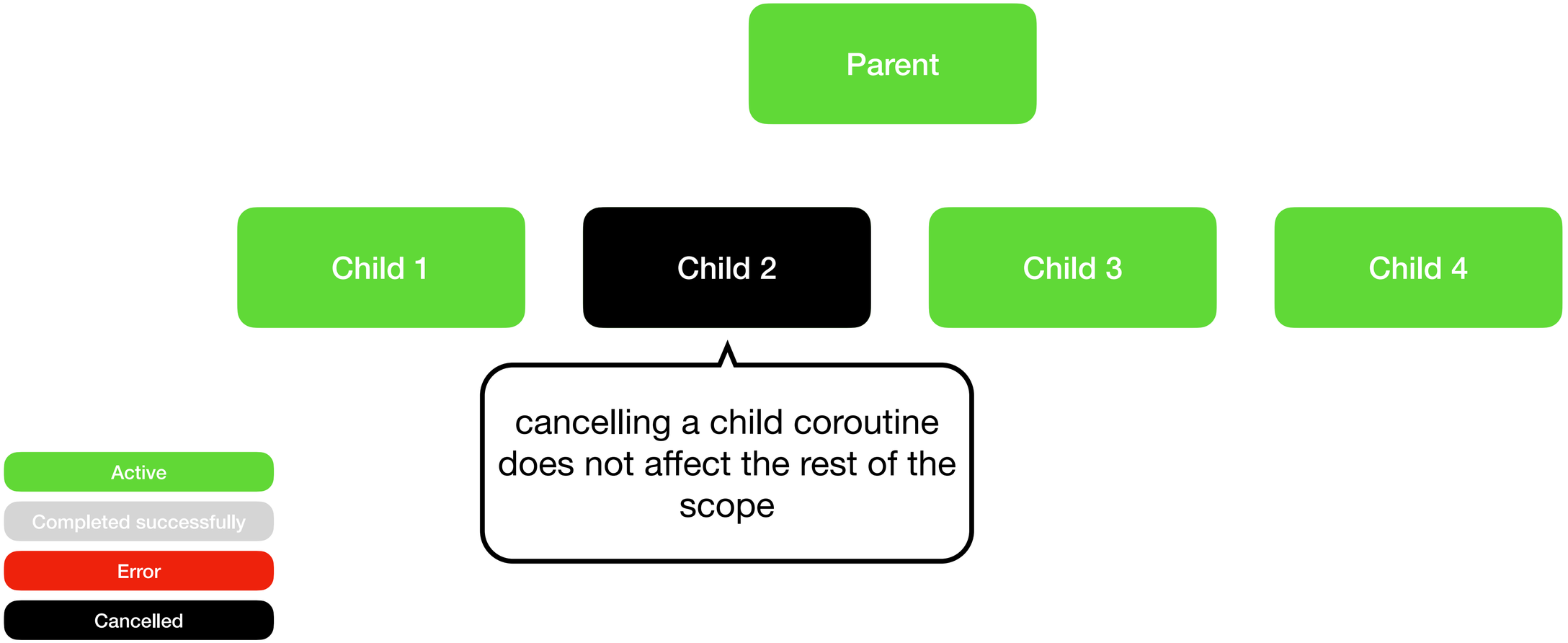
Error

Cancelled

* only if the exception is not dealt with programatically

Coroutine Scope

structured concurrency



Coroutine Scope

structured concurrency

we need a scope to be able to create new concurrent coroutines via async or launch

```
suspend fun bakePretzels(): List<FinishedPretzel> = coroutineScope {
    val oven = async { preheatOven(ColdOven) }
    val dough = async { prepareDough() }
    val uncookedPretzels = List(5) { async { shapePretzel(dough.await()) } }
    val bakedPretzels = async { bake(oven.await(), uncookedPretzels.awaitAll()) }
    val topping = async { prepareTopping() }
    bakedPretzels.await().map { finishPretzel(it, topping.await()) }
}
```

Coroutine Scope

launching long running coroutines

```
fun CoroutineScope.bakeryPipeline() = launch {
        while (true) {
            bakePretzels().forEach {
                println("produced $it")
            }
        }
    }
```

it will print “produced FinishedPretzel” until the job is canceled

Cooperative Cancelation

main suspends here forever

```
fun main() = runBlocking {
    val job = cashDesk()
    delay(1000)
    job.cancelAndJoin()
}
```

let’s close the shop

we didn’t change our dispatcher, so we’re still on the main thread

```
fun CoroutineScope.cashDesk() = launch {
    while (true) {
        println("How many pretzels?")
        val count = readLine()?.toIntOrNull()

        //receive pretzel
    }
}
```

this is blocking - so the main thread is blocked

Cooperative Cancelation

```
fun main() = runBlocking {
    val job = cashDesk()
    delay(1000)
    job.cancelAndJoin()
}
```

```
fun CoroutineScope.cashDesk() = launch {
    while (true) {
        println("How many pretzels?")
        val count = readLine()?.toIntOrNull()
        yield()
        //receive pretzel
    }
}
```

let's give other coroutines a chance to run on the main thread

yield is also a suspending function: it will throw CancelationException after the job is canceled

Cooperative Cancelation

```
fun main() = runBlocking {
    val job = cashDesk()
    delay(1000)
    job.cancelAndJoin()
}
```

cooperate in the cancelation, by stopping the loop if our job is no longer active

by using a multi-threaded dispatcher yield isn't needed

```
fun CoroutineScope.cashDesk() = launch(IO) {
    while (isActive) {
        println("How many pretzels?")
        val count = readLine()?.toIntOrNull()
        //receive pretzel
    }
}
```

Calling Blocking Code

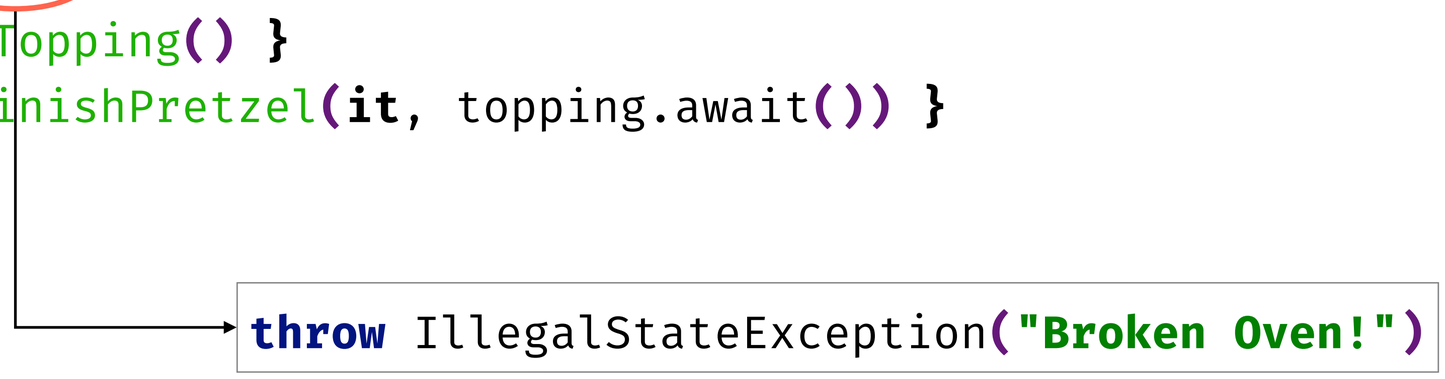
```
val job = launch(Dispatchers.IO) {
    Thread.sleep(100) //actual blocking code
}
```

never use `Main` dispatcher for blocking code: you don't want to block the UI

```
val result: Deferred<String> = async(Dispatchers.IO) {
    Thread.sleep(100) //actual blocking code
    "finished"
}
```

Exception Handling

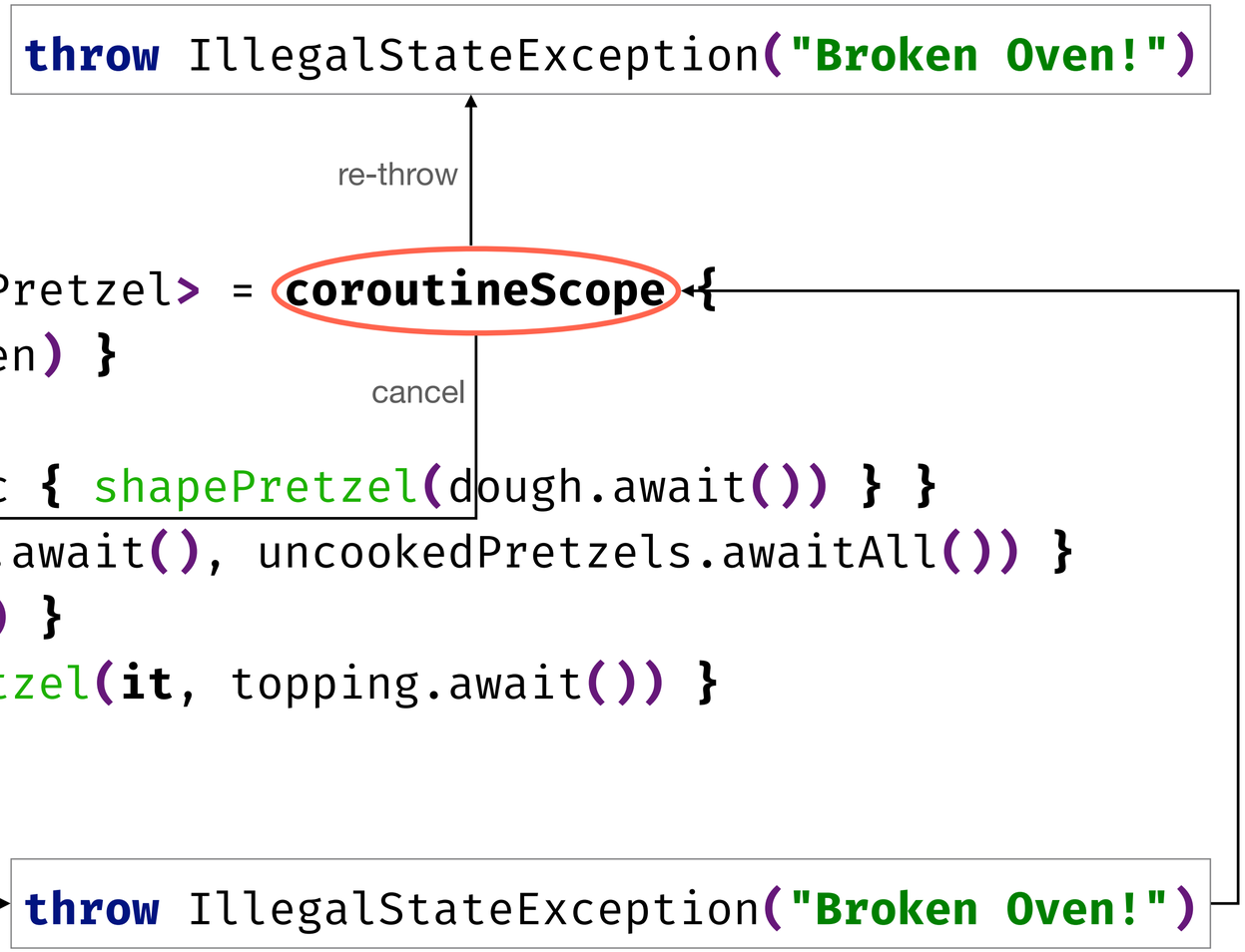
```
suspend fun bakePretzels(): List<FinishedPretzel> = coroutineScope {
    val oven = async { preheatOven(ColdOven) }
    val dough = async { prepareDough() }
    val uncookedPretzels = List(5) { async { shapePretzel(dough.await()) } }
    val bakedPretzels = async { bake(oven.await(), uncookedPretzels.awaitAll()) }
    val topping = async { prepareTopping() }
    bakedPretzels.await().map { finishPretzel(it, topping.await()) }
}
```



Exception Handling

structured concurrency: no coroutine leaks

```
suspend fun bakePretzels(): List<FinishedPretzel> = coroutineScope {
    val oven = async { preheatOven(ColdOven) }
    val dough = async { prepareDough() }
    val uncookedPretzels = List(5) { async { shapePretzel(dough.await()) } }
    val bakedPretzels = async { bake(oven.await(), uncookedPretzels.awaitAll()) }
    val topping = async { prepareTopping() }
    bakedPretzels.await().map { finishPretzel(it, topping.await()) }
}
```



Exception Handling

```
fun main() = runBlocking {
    val pretzels = bakePretzels()
}
```

Exception in thread "main" java.lang.IllegalStateException: Broken Oven!

```
fun main() = runBlocking {
    val pretzels = try {
        bakePretzels()
    } catch (e: Exception) {
        println("There was a problem preparing the pretzels: $e")
        emptyList()
    }
}
```

There was a problem preparing the pretzels: java.lang.IllegalStateException: Broken Oven!

Exception Handling

```
fun main() = runBlocking {
    val pretzels = runCatching {
        bakePretzels()
    }.
```

```
getOrDefault(defa…  List<FinishedPretzel>
getOrElse {...} (…  List<FinishedPretzel>
getOrThrow() for …  List<FinishedPretzel>
map {...} (transform: (List<F…  Result<R>
mapCatching {...} (transform:…  Result<R>
onFailure…  Result<List<FinishedPretzel>>
onSuccess…  Result<List<FinishedPretzel>>
recover {…  Result<List<FinishedPretzel>>
recoverCatching  Result<List<FinishedPre…
```

Communication Between Coroutines

channels

```
fun main() = runBlocking {
    val shelf = bakeryPipeline()
    val job = cashDesk(shelf)
    delay(hours(8))
    job.cancelAndJoin()
    shelf.cancel()
}
```

Communication Between Coroutines

channels

```
fun CoroutineScope.bakeryPipeline() =
    produce<Pretzel>(context = Dispatchers.Default, capacity = 10) {
        while (true) {
            println("Start producing")
            bakePretzels().forEach {
                send(it)
                println("$it sent to shelf")
            }
        }
    }
```

launches a coroutine that produces values in a channel

Communication Between Coroutines

channels

```
fun CoroutineScope.cashDesk(shelf: ReceiveChannel<Pretzel>) = launch(IO) {
    while (isActive) {
        println("How many pretzels?")
        val count = readLine()?.toIntOrNull()
        if (count != null) {
            (1..count).map { shelf.receive() }
                .forEach {
                    print("Here's your pretzel: $it")
                }
        } else {
            println("Command not recognized")
        }
    }
}
```

Communication Between Coroutines

- channels
- flows
- synchronized mutable types (Atomic* types, synchronized collections, etc)

❗ Avoid mutable shared state and never use unsynchronized mutable shared state

Testing Coroutines
kotlinx-coroutines-test

How long will the following test take to run?

```
@Test
fun `registration should close automatically after timeout`() = runBlockingTest {
    val register = AutoCloseableRegistration(60.seconds)

    delay(55.seconds)
    register(playerOne)

    delay(5.1.seconds)
    shouldThrow<RegistrationClosedException> {
        register(playerTwo)
    }
}
```

Testing Coroutines
kotlinx-coroutines-test

How long will the following test take to run?

```
@Test
fun `registration should close automatically after timeout`() = runBlockingTest {
    val register = AutoCloseableRegistration(60.seconds)

    delay(55.seconds)
    register(playerOne)

    delay(5.1.seconds)
    shouldThrow<RegistrationClosedException> {
        register(playerTwo)
    }
}
```

Testing Coroutines
kotlinx-coroutines-test

How long will the following test take to run? A few milliseconds.

```
@Test
fun `registration should close automatically after timeout`() = runBlockingTest {
    val register = AutoCloseableRegistration(60.seconds)

    delay(55.seconds)
    register(playerOne)

    delay(5.1.seconds)
    shouldThrow<RegistrationClosedException> {
        register(playerTwo)
    }
}
```

Coroutines
recap

- suspend functions are asynchronous
- coroutines are cheap
- when calling blocking code, use a different dispatcher
- use structured concurrency to avoid resource leaks (coroutines)
- use structured concurrency to avoid useless resource usage (CPU)
- avoid mutable shared state
- if you must use mutable shared state, make sure it’s synchronized

Exercise

tic-tac-toe

- Create a tic-tac-toe game where two players (Red and Blue) can play concurrently
- Start 10000 concurrent games and measure the execution times
- How many games did each player won? How many draws were there?