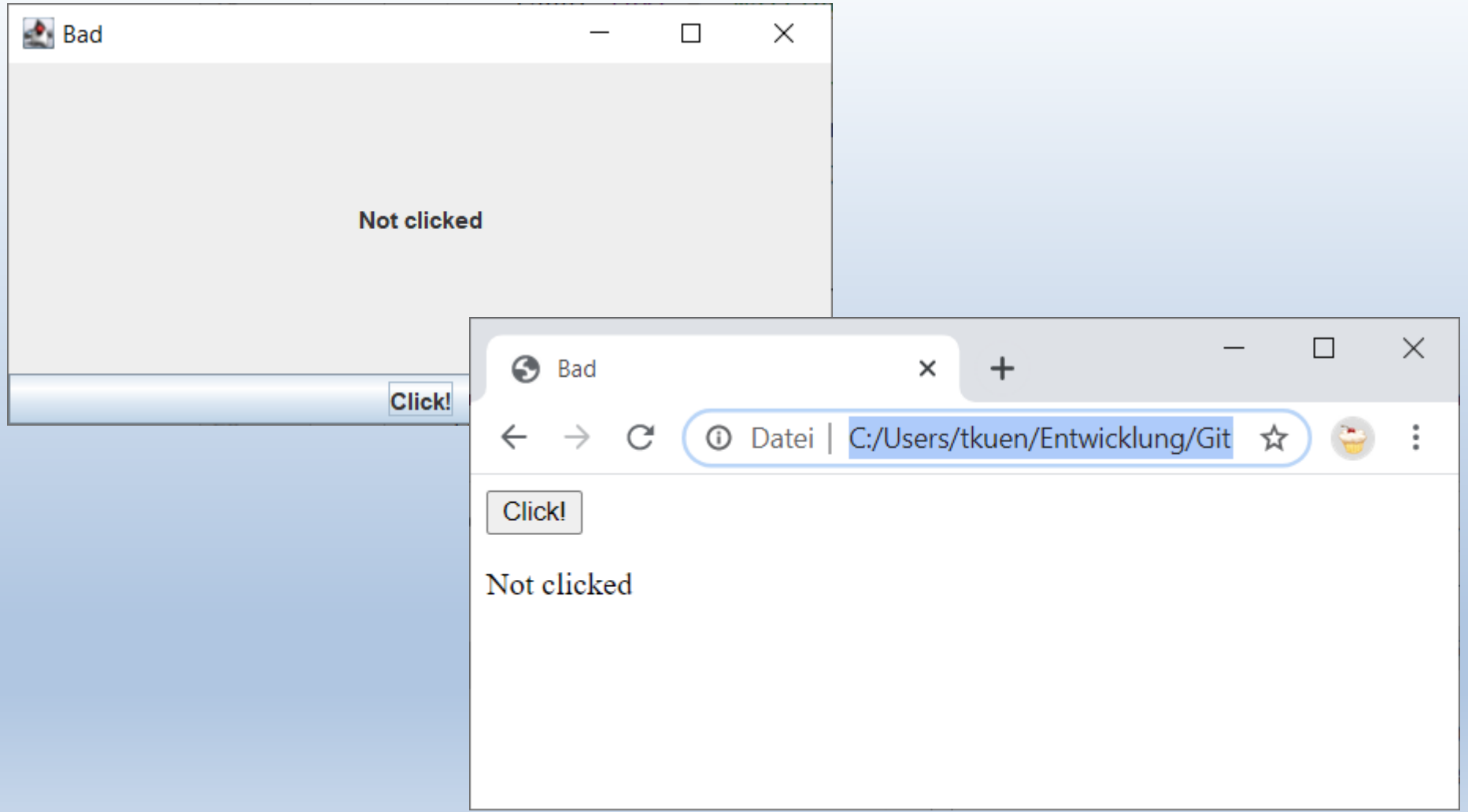
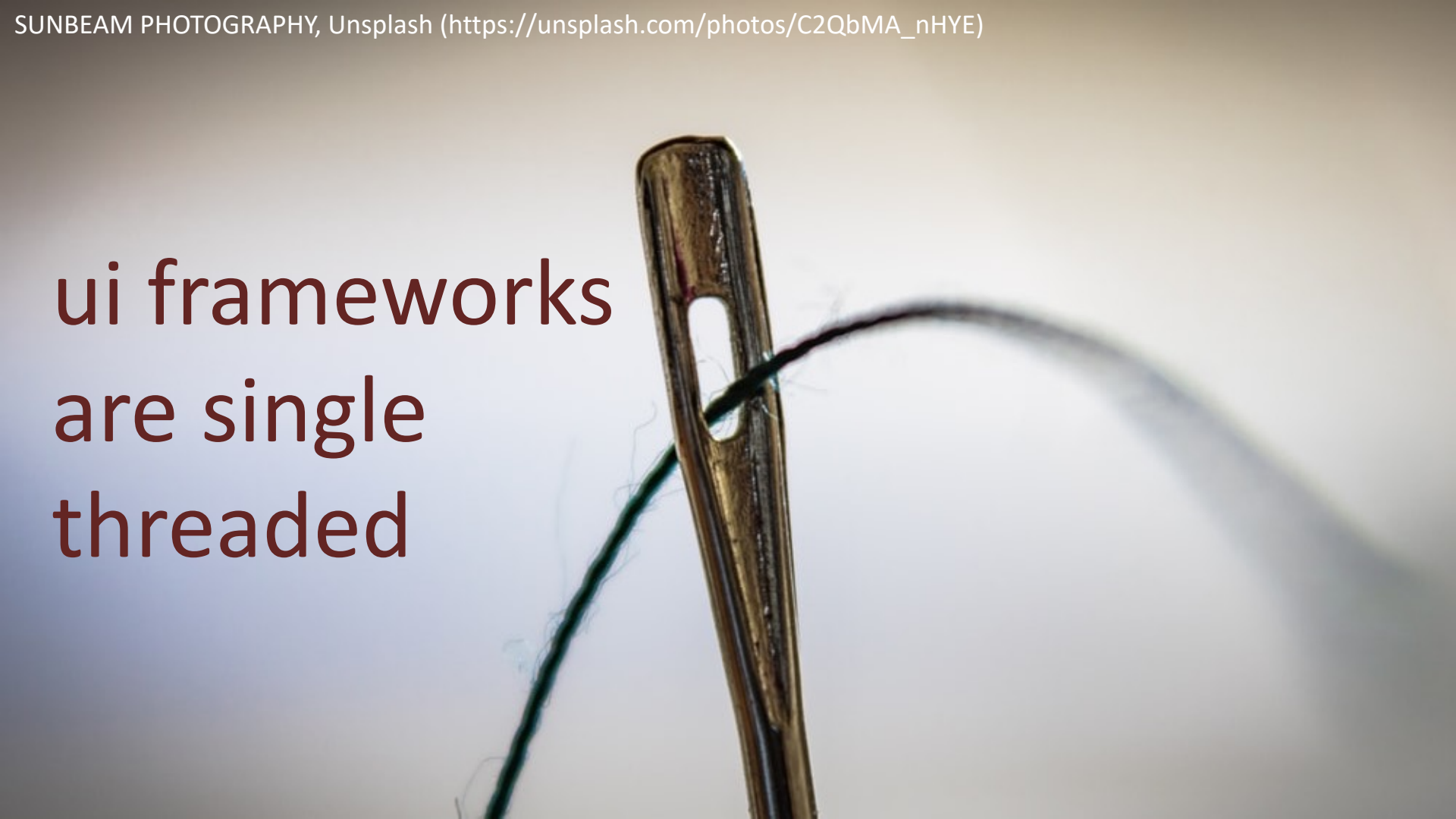
# Waiting for Godot

Thomas Künneth, MATHEMA GmbH

Alberto Bigoni, Unsplash (https://unsplash.com/photos/wmrHsSL7Io4)


Bad
Not clicked
Click!
Bad
Datei | C:/Users/tkuen/Entwicklung/Git
Click!
Not clicked


SUNBEAM PHOTOGRAPHY, Unsplash (https://unsplash.com/photos/C2QbMA_nHYE)
ui frameworks
are single
threaded

Better.kt

```
fun main() {
    invokeLater {
        val frame = JFrame( title: "Better")
        val label = JLabel( text: "Not clicked")
        label.horizontalAlignment = JLabel.CENTER
        frame.contentPane.add(label, BorderLayout.CENTER)
        val button = JButton( text: "Click!")
        button.addActionListener {  it: ActionEvent!
            label.text = "waiting"
            thread {
                val result =
                        URL( spec: "http://127.0.0.1:8080/hello/10000")
                                .openStream()
                                .bufferedReader()
                                .use { it.readText() }
                invokeLater {
                    label.text = result
                }
            }
        }
        frame.contentPane.add(button, BorderLayout.SOUTH)
        frame.pack()
        frame.setLocationRelativeTo(null)
        frame.isVisible = true
    }
}
```

```
fetch("http://127.0.0.1:8080/hello/10000")
    .then(function (response) {
        return response.text();
    })
    .then(function (text) {
        document.getElementById("message").innerHTML = text;
    })
    .catch(function (error) {
        console.log("Error: " + error);
    })
    .finally(function () {
        document.getElementById("button").disabled = false;
    });
    document.getElementById("message").innerHTML =
    "waiting";
```

Even_better.kt

```
import ...

fun main() {
    invokeLater {
        val frame = JFrame( title: "Better")
        val label = JLabel( text: "Not clicked")
        label.horizontalAlignment = JLabel.CENTER
        frame.contentPane.add(label, BorderLayout.CENTER)
        val button = JButton( text: "Click!")
        button.addActionListener {  it: ActionEvent!
            label.text = "waiting"
            val future = CompletableFuture.supplyAsync {
                URL( spec: "http://127.0.0.1:8080/hello/10000")
                        .openStream()
                        .bufferedReader()
                        .use { it.readText() }
            }
            future.thenAccept {  it: String!
                label.text = it
            }
        }
        frame.contentPane.add(button, BorderLayout.SOUTH)
        frame.pack()
        frame.setLocationRelativeTo(null)
        frame.isVisible = true
    }
}
```

BetterCoroutine.kt

```
1  import ...
11
12 fun main() = invokeLater {
13     val frame = JFrame( title: "Better Coroutine")
14     val label = JLabel( text: "Not clicked")
15     label.horizontalAlignment = JLabel.CENTER
16     frame.contentPane.add(label, BorderLayout.CENTER)
17     val button = JButton( text: "Click!")
18     button.addActionListener {  it: ActionEvent!
19         label.text = "waiting"
20         GlobalScope.launch() {
21             val result = GlobalScope.async(Dispatchers.IO) {
22                 URL( spec: "http://127.0.0.1:8080/hello/10000")
23                         .openStream()
24                         .bufferedReader()
25                         .use { it.readText() }
26             }
27             label.text = result.await()
28         }
29     }
30     frame.contentPane.add(button, BorderLayout.SOUTH)
31     frame.pack()
32     frame.setLocationRelativeTo(null)
33     frame.isVisible = true
34 }
```

A **coroutine** is an instance of [a] suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run and has a similar life-cycle — it is created and started, but it is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. Moreover, like a future or promise, it may complete with some result (which is either a value or an exception).

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

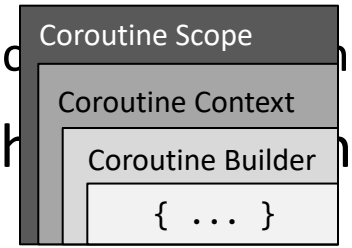

Matt Walsh, Unsplash (https://unsplash.com/photos/tVkdGtEe2C4)

# TL;DR;

- Coroutines can be suspended and resumed
- Are like lightweight threads
- Kotlin somehow uses threads for their execution

# CoroutineScope

- Controls and manages one or more coroutines
  - Can start and cancel them
  - Is notified upon c nd failures
- Defines when/wh ng coroutines exist



- Usually a scope corresponds to the lifecycle of some entity
- Coroutine-aware frameworks provide specific scopes
- `GlobalScope` is valid during lifetime of the app

GlobalScopeDemo.kt

```kotlin
package godot

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

fun main() {
    GlobalScope.launch { this: CoroutineScope
        println("started")
        delay( timeMillis: 3000)
        println("finished")
    }
    println("waiting")
    Thread.sleep( millis: 4000)
}
```

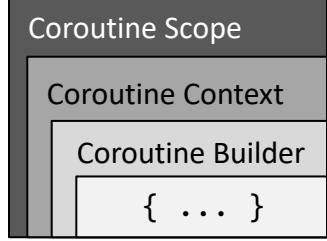
RunBlockingJoinDemo.kt

```kotlin
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    print("Hello, ")
    val job = GlobalScope.launch { this: CoroutineScope
        delay( timeMillis: 3000L)
        println("World!")
    }
    runBlocking { this: CoroutineSc
        job.join()
    }
}
```

RunBlockingDemo.kt

```kotlin
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking<Unit> { this: CoroutineScope
    print("Hello, ")
    launch { this: CoroutineScope
        delay( timeMillis: 3000L)
        println("World!")
    }
}
```

# CoroutineContext




- Each coroutine is executed in a specific context (`CoroutineContext`)
- Is provided through `CoroutineScope.coroutineContext`
- Implemented as an index based set of elements (mixture of a set and a map)

- A new coroutine inherits the parent context
- parent context = Defaults + inherited context + arguments (for example from `launch`)
- new `CoroutineContext` = parent context + `Job()`
- Important elements: `CoroutineDispatcher`, `Job`, `CoroutineExceptionHandler`, `CoroutineName`

```kotlin
package godot

import kotlinx.coroutines.CoroutineName
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    runBlocking { this: CoroutineScope
        launch(CoroutineName( name: "Hello")) { this: CoroutineScope
            println(coroutineContext.toString())
            launch(CoroutineName( name: "world")) { this: CoroutineScope
                println(coroutineContext.toString())
            }
        }
```

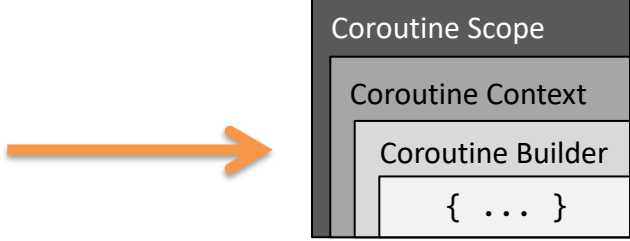
Run: CoroutineContextDemoKt × BetterKt ×

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...
[CoroutineName(Hello), StandaloneCoroutine{Active}@4fccd51b, BlockingEventLoop@44e81672]
[CoroutineName(world), StandaloneCoroutine{Active}@46f5f779, BlockingEventLoop@44e81672]

Process finished with exit code 0
```

# Coroutine Dispatcher

- Define which threads are used to execute a coroutine
- Can confine a coroutine to one thread, a thread pool, or pose no restrictions

- `Dispatchers.Default`
  execution in a thread pool based on the number of cores
- `Dispatchers.Main`
  Execution only on the Main Thread
- `Dispatchers.IO`
  For long running/blocking I/O operations
- `Dispatchers.Unconfined`
  no restrictions (shouldn‘t be used)




A **coroutine builder** [is] a function that takes some suspending lambda as an argument, creates a coroutine, and, optionally, gives access to its result in some form. For example, `launch{}`, `future{}`, and `sequence{}` [..] are coroutine builders. The standard library provides primitive coroutine builders that are used to define all other coroutine builders.

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

# Coroutine Builder: TL;DR;

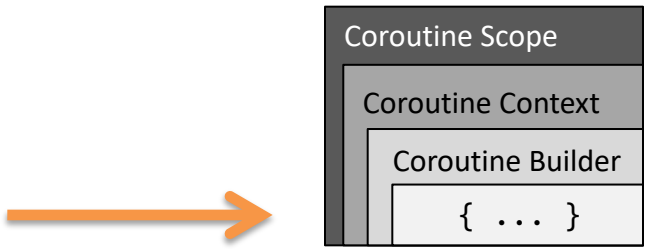
- launch coroutines
- examples: `launch`, `async`, `runBlocking`, …
- extension functions to `CoroutineScope`
- Can receive parameters, for example a launch mode


Matt Walsh, Unsplash (https://unsplash.com/photos/tVkdGtEe2C4)

# Launch modes

- `DEFAULT`
  begin execution immediately
- `ATOMIC`
  similar to `DEFAULT`; can be cancelled only after execution has started
- `LAZY`
  launch only if needed (when result is accessed)
- `UNDISPATCHED`
  Immediate execution on the current thread

```kotlin
package godot

import kotlinx.coroutines.*

fun main() = runBlocking {  this: CoroutineScope
    print("Hello, ")
    print(async(Dispatchers.Default, start = CoroutineStart.LAZY) {
        wait()
    }.await())
}

suspend fun wait(): String {
    delay( timeMillis: 3000)
    return "world!"
}
```

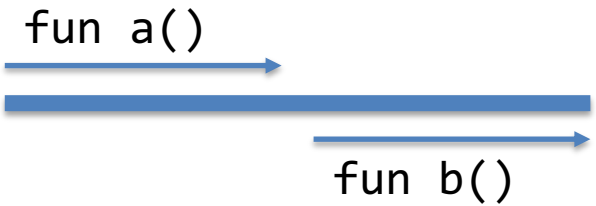

Coroutine Scope
Coroutine Context
Coroutine Builder
{ ... }

A **suspending lambda** [is a] a block of code that have to run in a coroutine. It looks exactly like an ordinary lambda expression but its functional type is marked with `suspend` modifier. Just like a regular lambda expression is a short syntactic form for an anonymous local function, a suspending lambda is a short syntactic form for an anonymous suspending function. It may suspend execution of the code without blocking the current thread of execution by invoking suspending functions. For example, blocks of code in curly braces following `launch`, `future`, and `sequence` functions [...] are suspending lambdas.

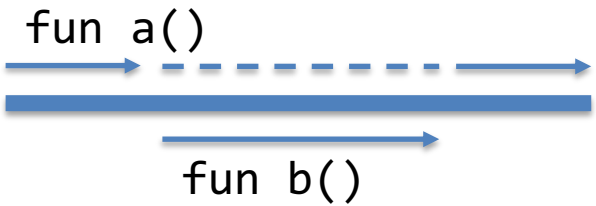
https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology

A **suspending function** is a function that is marked with `suspend` modifier. It may suspend execution of the code without blocking the current thread of execution by invoking other suspending functions. A suspending function cannot be invoked from a regular code, but only from other suspending functions and from suspending lambdas [...]. For example, `await()` and `yield()` [...] are suspending functions that may be defined in a library. The standard library provides primitive suspending functions that are used to define all other suspending functions.

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology




normal, blocking function




suspendable function

A **suspension point** is a point during coroutine execution where the execution of the coroutine may be suspended. Syntactically, a suspension point is an invocation of suspending function, but the actual suspension happens when the suspending function invokes the standard library primitive to suspend the execution.

A **continuation** is a state of the suspended coroutine at suspension point. It conceptually represents the rest of its execution after the suspension point.

https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology


Matt Walsh, Unsplash (https://unsplash.com/photos/tVkdGtEe2C4)

# TL;DR;

- suspend functions are basic building blocks of coroutines
- are paused at some time or have to wait for something to finish
- do not block the current thread
- are called from other coroutines or suspending functions
- like normal functions return something

SimpleSuspending
FunctionsDemo.kt

SuspendDemo2.kt

```
5   import kotlinx.coroutines.launch
6
7   suspend fun main() {
8       print("Hello, ")
9       val job = GlobalScope.launch { hello() }
10      while (job.isActive) {
11          print("+")
12          delay( timeMillis: 1000)
13      }
14  }
15
16  suspend fun hello() {
17      delay( timeMillis: 5000)
18      println(" World!")
19  }
```

# Orchestrating coroutines

Manuel Nägeli, Unsplash (https://unsplash.com/photos/6DBZqMe2c5U)

SeveralCoroutines.kt

```
1  import kotlinx.coroutines.delay
2  import kotlinx.coroutines.launch
3  import kotlinx.coroutines.runBlocking
4
5  fun main() {
6      runBlocking {  this: CoroutineScope
7          for (i in 1..10) {
8              launch {  this: CoroutineScope
9                  delay( timeMillis: i * 1000
10                 println("$i finished")
11             }
12         }
13         println("Already there")
14     }
15     println("All done")
16 }
```

```
/Library/Java/JavaVirtualMachines/j
Already there
1 finished
2 finished
3 finished
4 finished
5 finished
6 finished
7 finished
8 finished
9 finished
10 finished
All done

Process finished with exit code 0
```

CoroutineScope.kt

```kotlin
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking<Unit> {  this: CoroutineScope
    coroutineScope {  this: CoroutineScope
        for (i in 1..10) {
            launch {  this: CoroutineScope
                delay( timeMillis: i * 1000L)
                println("$i finished")
            }
        }
        println("Already there")
    }
    println("All done")
}
```

- `runBlocking` and `coroutineScope` wait for the execution of its block and children
- `runBlocking` blocks the current thread
- `coroutineScope` suspends its execution
- Difference: coroutine builder vs. suspending function


Justus Menke, Unsplash (https://unsplash.com/photos/Uwecr7Su3dU)

# Cancel execution

- `launch` returns a `Job` instance
  - controls the lifecycle
  - allows for hierarchies
- `cancel` initiates the cancellation
- `join` waits for the cancellation to take effect

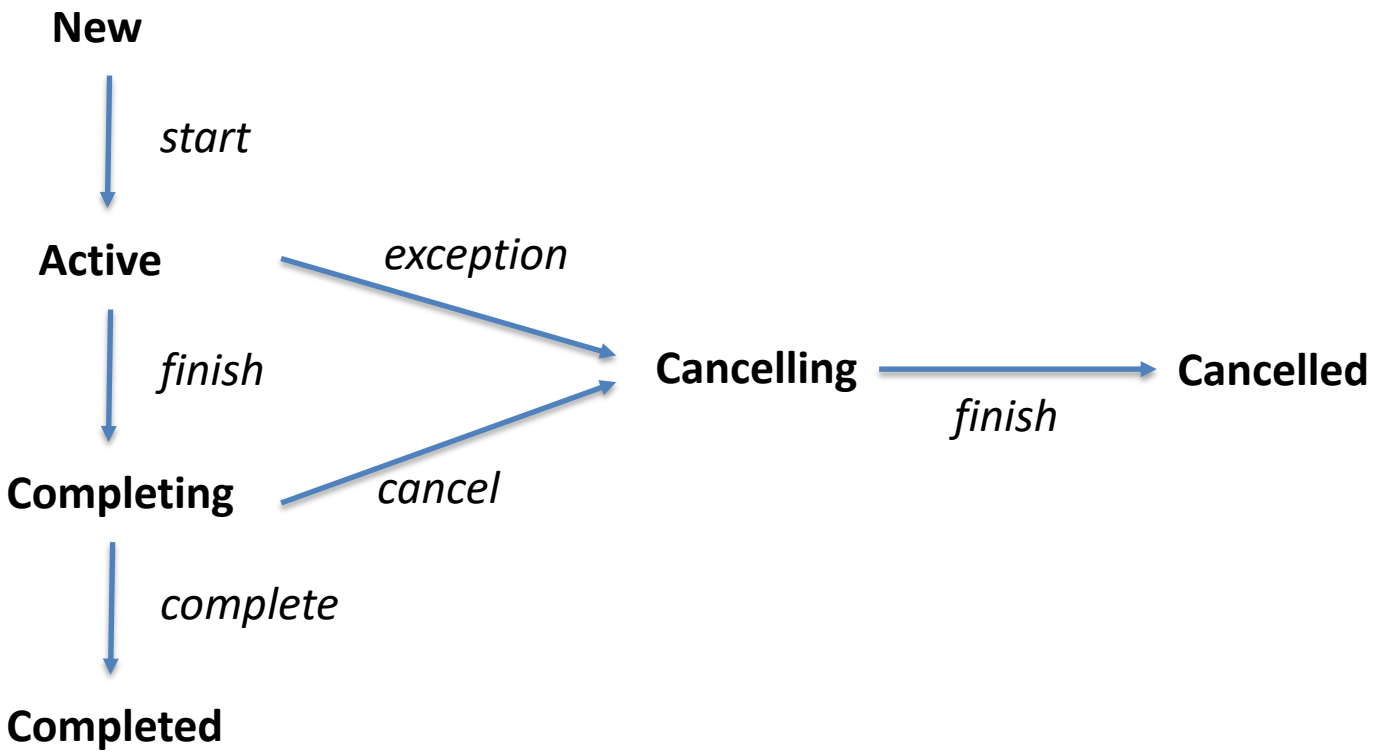
CancelDemo.kt

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

fun main() = runBlocking { this: CoroutineScope
    println(measureTimeMillis {
        val job = launch { this: CoroutineScope
            println("Enter")
            delay( timeMillis: 10000)
            println("Exit")
        }
        delay( timeMillis: 3000)
        job.cancelAndJoin()
        println("Cancelled")
    })
}
```

```
/Library/Java/JavaVirtualMachines/jdk
Enter
Cancelled
3013

Process finished with exit code 0
```

- Cancelling a scope with `cancel()` cancels all children
- Cancelling a child with `cancel()` does not cancel other children

# Job Lifecycle



CancelNotWorkingDemo.kt

```
1  import ...
6
7  fun main() = runBlocking {  this: CoroutineScope
8      println(measureTimeMillis {
9          val job = launch {  this: CoroutineScope
10             println("Enter")
11             var count = 0
12             while (true) {
13                 println("${++count}")
14             }
15             println("Exit")
16         }
17         delay( timeMillis: 3000)
18         job.cancelAndJoin()
19         println("Cancelled")
20     })
21 }
```


Agê Barros, Unsplash (https://unsplash.com/photos/rBPOfVqROzY)

# Properties

- isActive
- isCancelled
- isCompleted

Artem Sapegin, Unsplash (https://unsplash.com/photos/b18TRXc8UPQ)

- Detect cancellation requests with `isActive`
- Regularly invoke `delay()`, `yield()` or `ensureActive()`

```
...
val job = launch {
    println("Enter")
    var count = 0
    while (isActive) {
        println("${++count}")
        yield()
    }
    println("Exit")
}
...
```

- All suspending functions in `kotlinx.coroutines` are cancellable
- Your suspend functions should be cancellable, too

# Coroutines must finish when they are no longer needed


Kyle Glenn, Unsplash (https://unsplash.com/photos/dGk-qYBk4OA)

# CancellationException

- Is thrown by suspend functions to signal a cancellation
- Useful to distinguish between cancellations and other exceptions

TimeoutDemo.kt

```
import ...

fun main(): Unit = runBlocking<Unit> {  this: CoroutineScope
    println(measureTimeMillis {
        val delay = 2000
        try {
            withTimeout( timeMillis: delay + 1000L) {  this: CoroutineScope
                val current = System.currentTimeMillis()
                while ((System.currentTimeMillis() - current) < delay) {
                    yield()
                }
            }
        } catch (e: TimeoutCancellationException) {
            println("I timed out")
        } finally {
            println
        }
    })
}
```

```
internal fun CoroutineContext.checkCompletion() {
    val job = get(Job)
    if (job != null && !job.isActive) throw job.getCancellationException()
}
```

# Returning values

Nakota Wagner, Unsplash (https://unsplash.com/photos/picnLtOBcbY)

ReturnAValueDemo.kt

```
1   import ...
4
5   fun main() = runBlocking<Unit> {  this: CoroutineScope
6       println("Took ${measureTimeMillis {
7           println("result: ${myFun1() + myFun2()}")
8       }} ms")
9   }
10
11  suspend fun myFun1(): Int {
12      delay( timeMillis: 1000)
13      return 1
14  }
15
16  suspend fun myFun2(): Int {
17      delay( timeMillis: 2000)
18      return 2
19  }
```

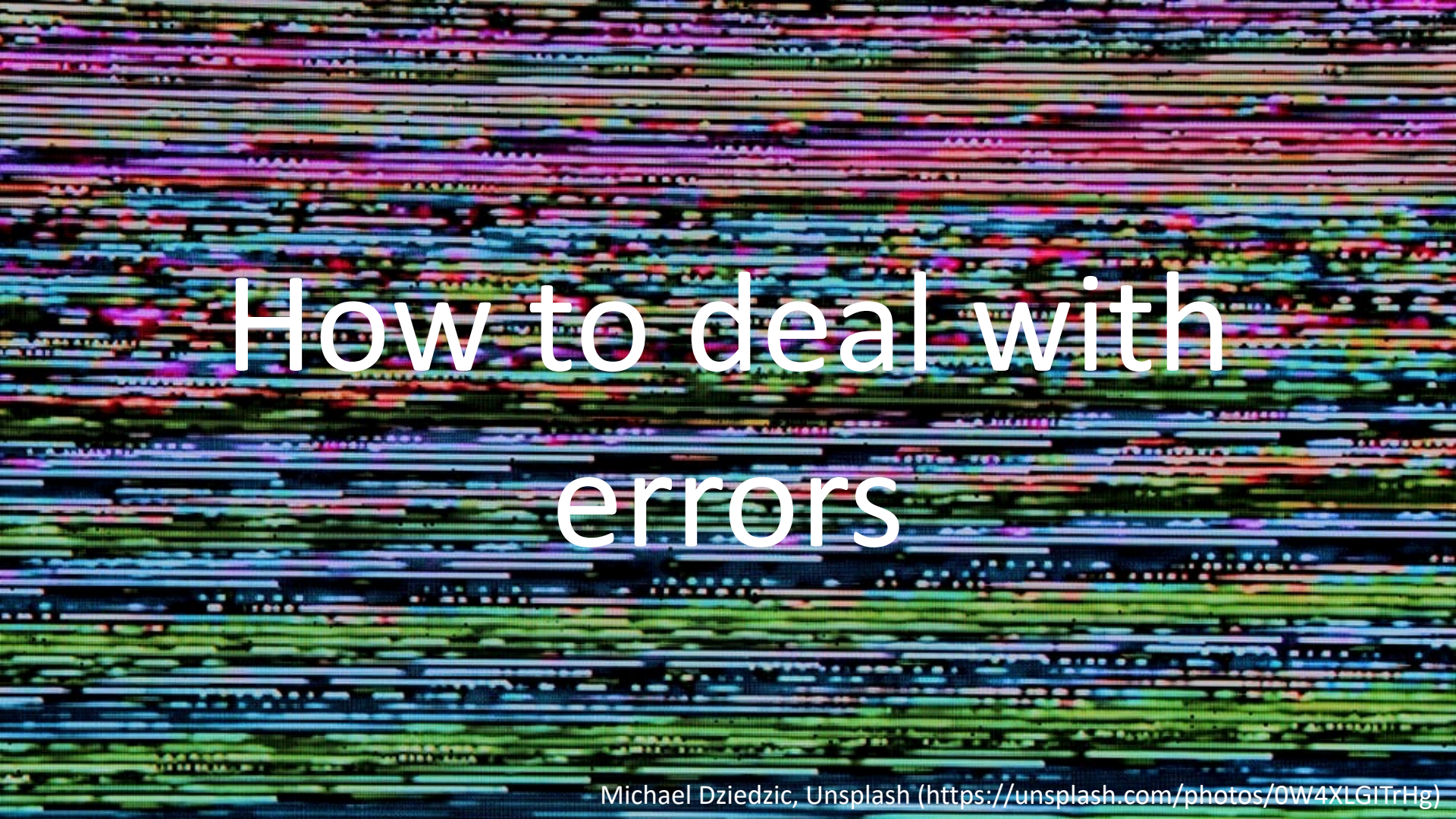
```
ReturnAValueDemoKt
/Library/Java/JavaVirtualMachines/jdk-14
result: 3
Took 3007 ms

Process finished with exit code 0
```

# Deferred

- `launch` is fire and forget
- What if somethig should be done when a result is available?
  - `Deferred` is a non blocking, cancellable Future
  - Created with `async` (Builder) or `CompletableDeferred()`
  - Same states like `Job`
  - Get result with `await()` (in case of an error an exception is thrown)

AsyncDemo2.kt

```
fun main() = runBlocking<Unit> {  this: CoroutineScope
    println("Took ${
        measureTimeMillis {
            val looper = launch {  this: CoroutineScope
                while (isActive) {
                    println("+")
                    delay( timeMillis: 500)
                }
            }
            println("result: ${
                async { myFun1() }.await()
                        + async { myFun2() }.await()
            } ")
            looper.cancelAndJoin()
        }
    } ms")
}
```




Michael Dziedzic, Unsplash (https://unsplash.com/photos/0W4XLGITrHg)

- Exceptions after `launch` are treated like uncaught exceptions in threads
- Crashed children cancel the parent with this exception
- Exceptions should be be handeled explicitly when using `async`

- Uncaught exceptions can be tracked with `CoroutineExceptionHandler`
- Alternative: `runCatching()`

ExceptionDemo.kt

```
1   import ...
3
4   fun main() = runBlocking<Unit> {  this: Coroutine
5       val job = launch() {  this: CoroutineScope
6           println("Hello launch")
7           throw RuntimeException("Hello crash")
8       }
9       job.join()
10      // not printed!!!
11      println("join() finished")
12  }
```

ExceptionDemo2.kt

```
1   import ...
5
6   fun main() = runBlocking {  this: CoroutineScope
7       val handler = CoroutineExceptionHandler { _, excep
8           println("Caught $exception")
9       }
10      val job = GlobalScope.launch(handler) {  this: Corouti
11          println("Hello launch")
12          throw RuntimeException("Hello crash")
13      }
14      job.join()
15      println("join() finished")
16  }
```

ExceptionDemo3.kt

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {  this: CoroutineScope
    val deferred: Deferred<Unit> = GlobalScope.async {  this:
        println("Started")
        delay( timeMillis: 1000)
        throw IllegalArgumentException("I want to crash")
    }
    try {
        deferred.await()
    } catch (e: Exception) {
        println(e.message)
    } finally {
        println("I am finally here")
    }
    println("Finished")
}
```

# SupervisorJob

- Errors or cancellations do not terminate other children:

```
val scope =
CoroutineScope(SupervisorJob())
```

- Uncaught exceptions are propagated upward
- `SupervisorJob` works only when being the immediate parent of a coroutine

Kolleen Gladden, Unsplash (https://unsplash.com/photos/ij5_qCBpIVY)

- Coroutines reside in their own library (`kotlinx.coroutines`)
- On language level only one keyword (`suspend`)

- Coroutines look like threads
- But behave differently
  - *Blocking* vs. *suspending*
  - Require cooperation

- API feels not always intuitive
  - When which Dispatcher?
  - When which Start Mode?

# Vielen Dank!

@tkuenneth

thomas.kuenneth@mathema.de

https://github.com/tkuenneth/mind_the_thread