May 2019

Diving deep into the Coroutines API

FIVE

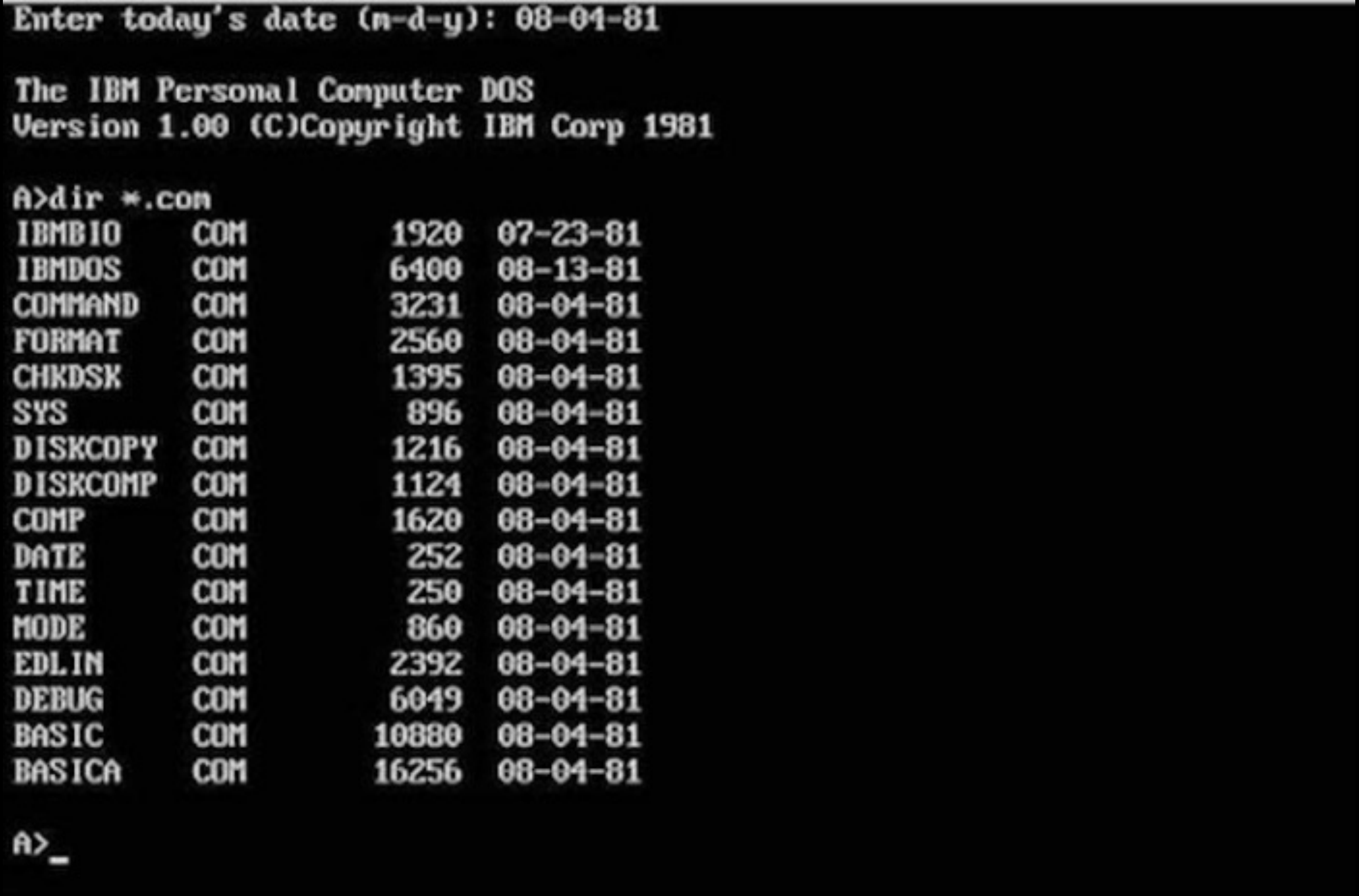
Filip Babić

About me

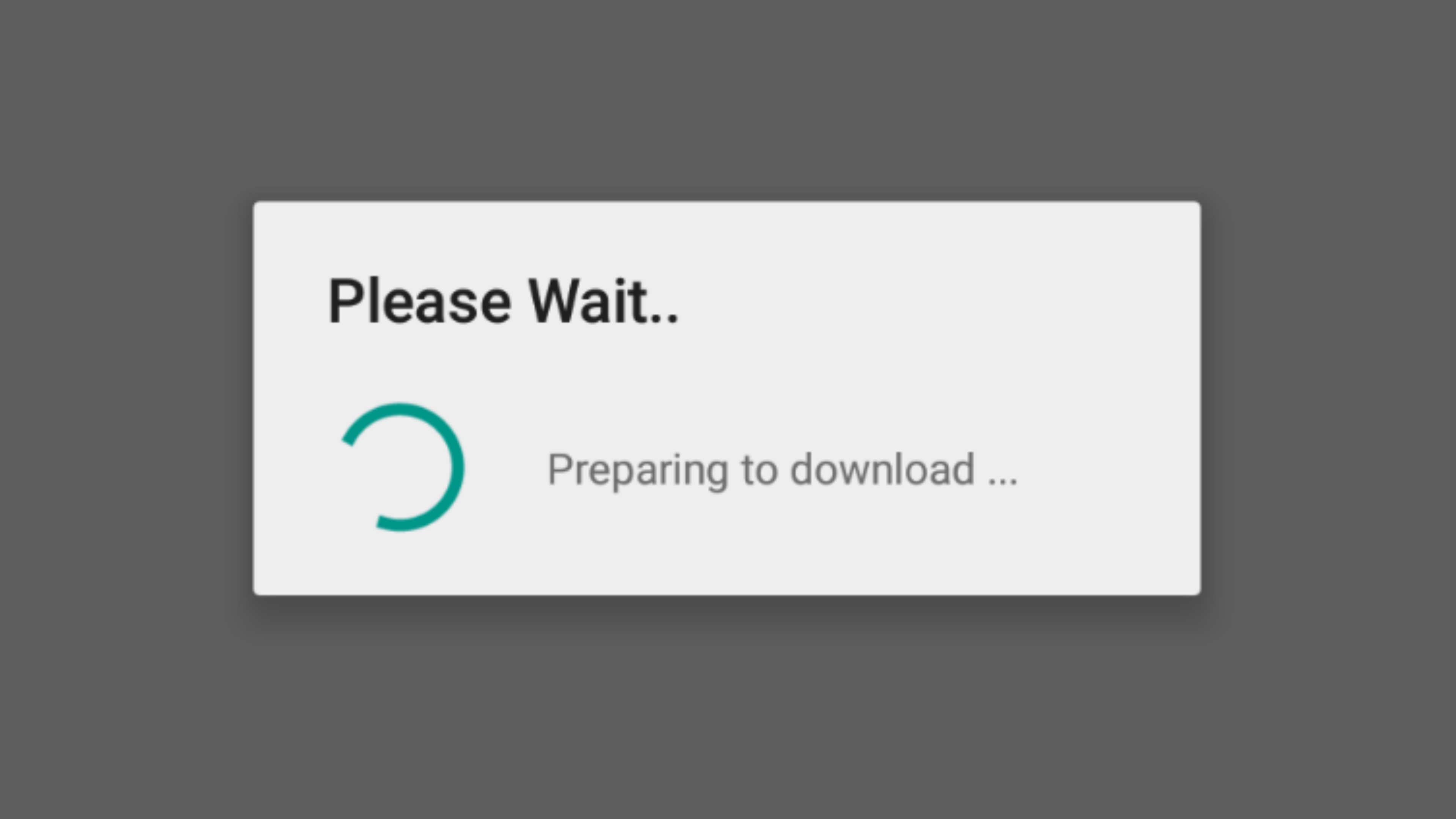
- Android developer @Five
- Android Author and Tech editor for RayWenderlich
- Speaking, writing, teaching…

The flow

- Brief history lesson of **async. programming**
- Introducing **coroutines** & **suspending** functions
- The **inner works** of the Coroutines API
- Writing **quality** concurrency code



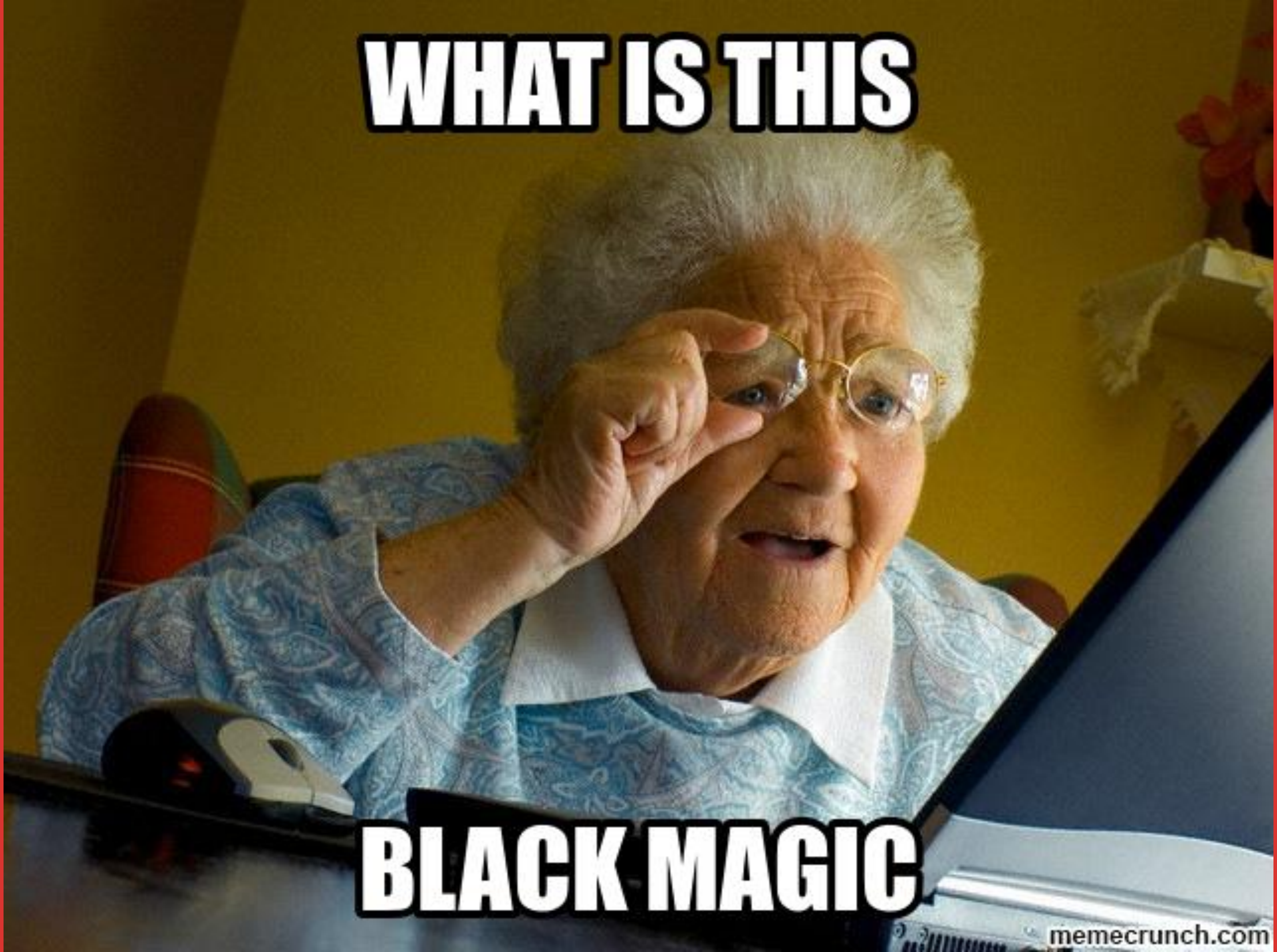
```
Enter today's date (m-d-y): 08-04-81

The IBM Personal Computer DOS
Version 1.00 (C)Copyright IBM Corp 1981

A>dir *.com
IBMBIO    COM       1920  07-23-81
IBMDOS    COM       6400  08-13-81
COMMAND   COM       3231  08-04-81
FORMAT    COM       2560  08-04-81
CHKDSK    COM       1395  08-04-81
SYS       COM        896  08-04-81
DISKCOPY  COM       1216  08-04-81
DISKCOMP  COM       1124  08-04-81
COMP      COM       1620  08-04-81
DATE      COM        252  08-04-81
TIME      COM        250  08-04-81
MODE      COM        860  08-04-81
EDLIN     COM       2392  08-04-81
DEBUG     COM       6049  08-04-81
BASIC     COM      10880  08-04-81
BASICA    COM      16256  08-04-81

A>_
```



Please Wait..

Preparing to download ...

λ


WHAT IS THIS
BLACK MAGIC
memecrunch.com

Kotlin Coroutines

(~~Black magic~~)

Coroutines

Coroutines

Coroutine Builders

Coroutines

Coroutine Builders

Suspend Functions

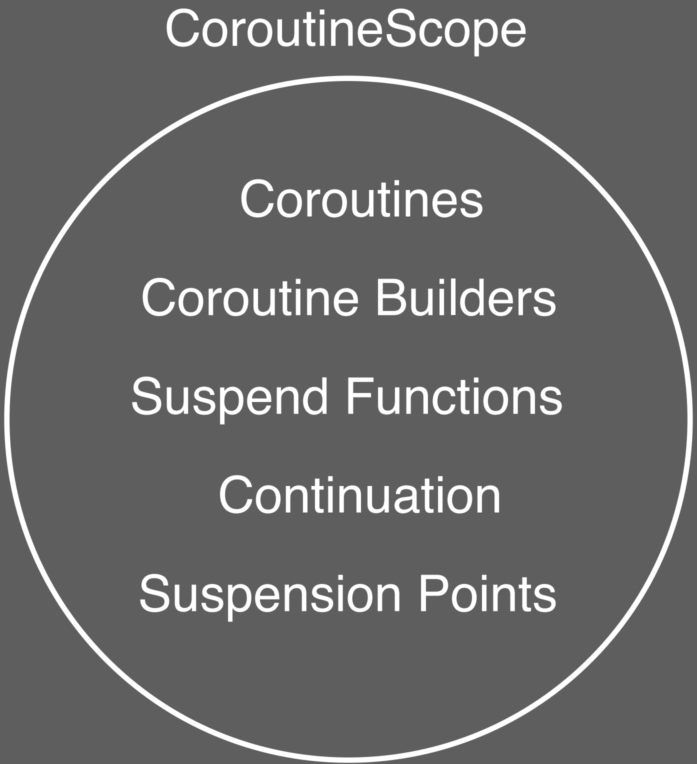
Coroutines

Coroutine Builders

Suspend Functions

Continuation

Suspension Points



CoroutineScope

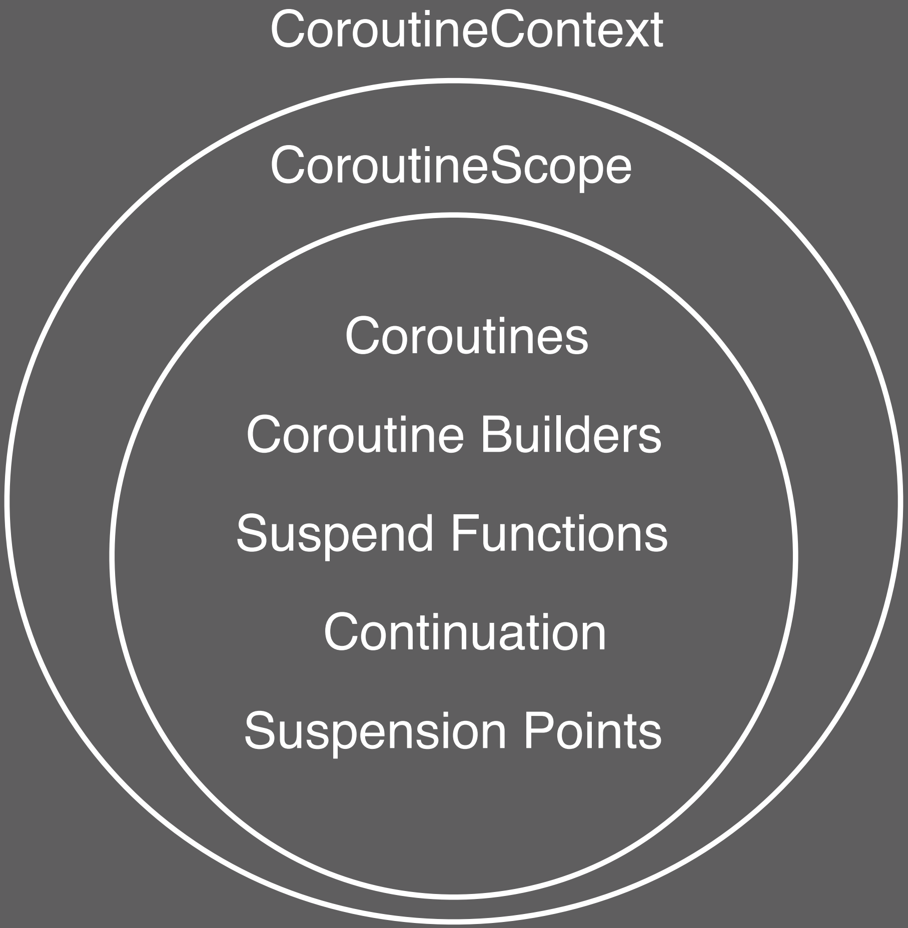
Coroutines

Coroutine Builders

Suspend Functions

Continuation

Suspension Points



CoroutineContext

CoroutineScope

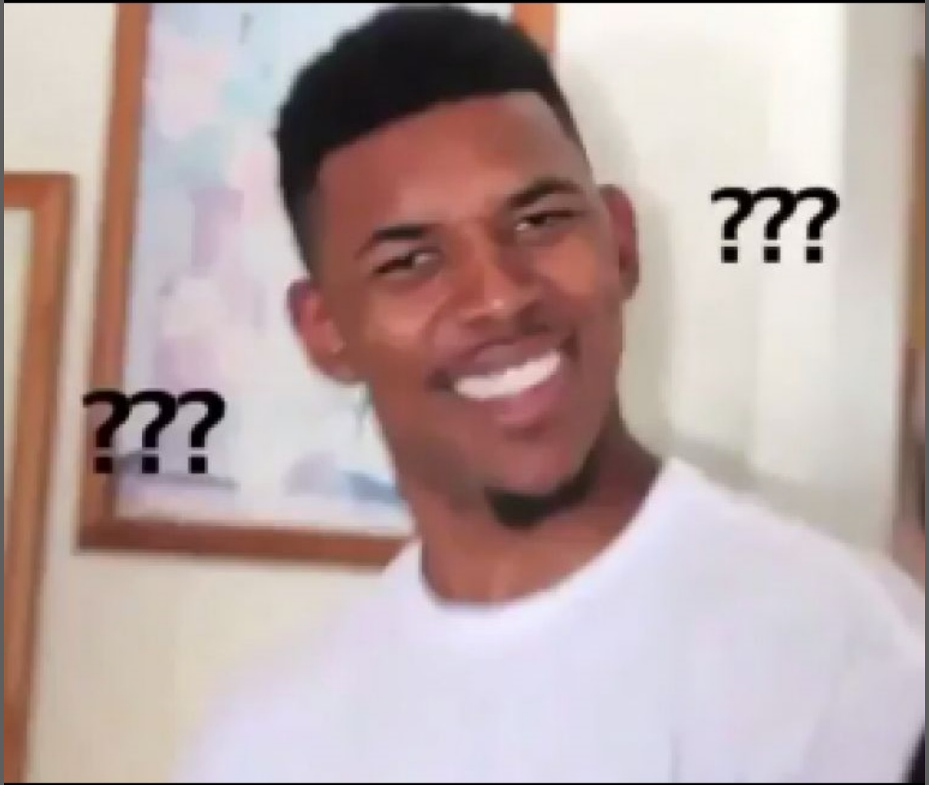
Coroutines

Coroutine Builders

Suspend Functions

Continuation

Suspension Points



Well, that’s a lot :]

Coroutine builders

Launch

```
launch {
  println("This is a coroutine")
}
```

Launch

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

Launch

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

Launch

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

Launch

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

Launch

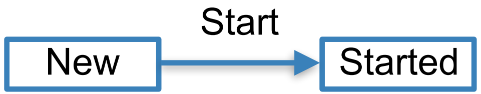
```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

Job

Job

- Cancellable piece of work
- Has lifecycle states
- Parent-child job relations

Job

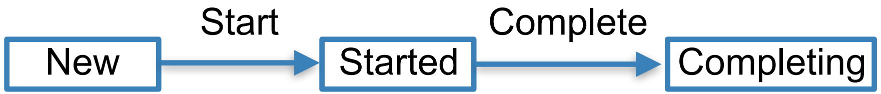


New

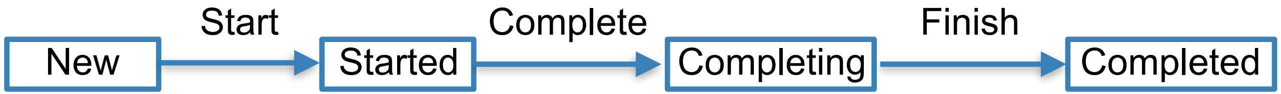
Start

Started

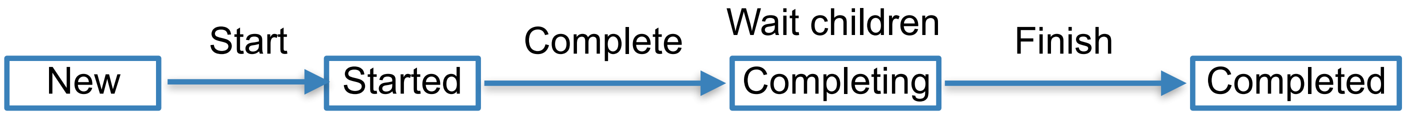
Job



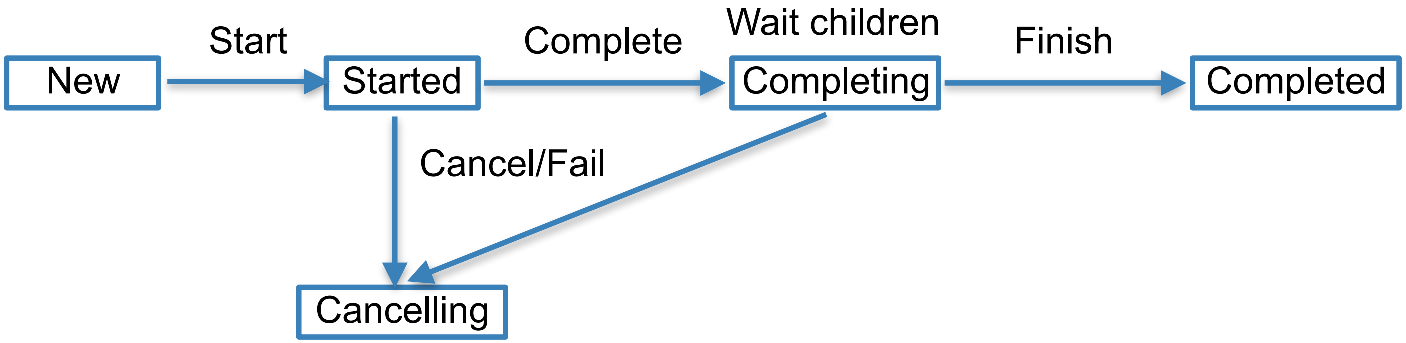
Job



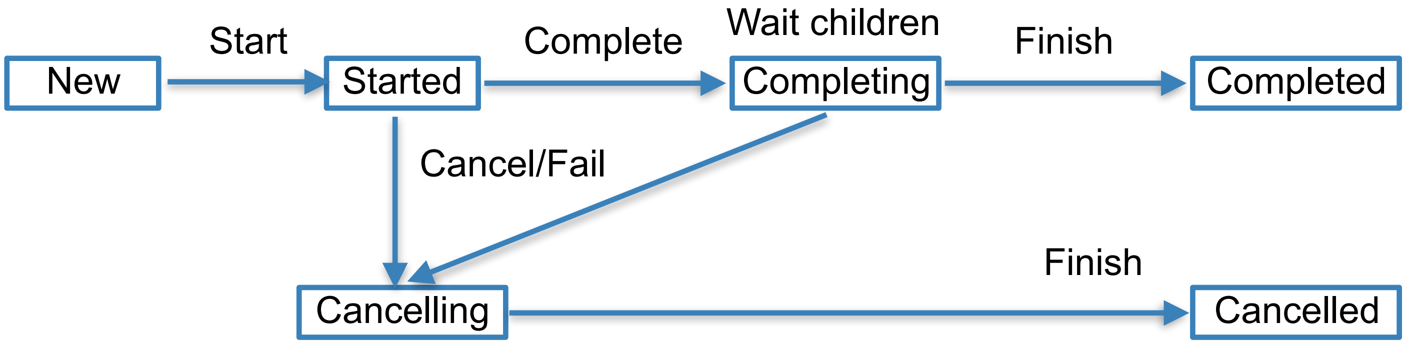
Job


New
Start
Started
Complete
Wait children
Completing
Finish
Completed

Job


Start
Complete
Wait children
Finish
New
Started
Completing
Completed
Cancel/Fail
Cancelling

Job



New

Start

Started

Complete

Wait children

Completing

Finish

Completed

Cancel/Fail

Cancelling

Finish

Cancelled

CoroutineScope

CoroutineScope

```
launch {
  println("This is a coroutine")
}
```

CoroutineScope

```
with(GlobalScope) {

  launch {
    println("This is a coroutine")
  }
}
```

CoroutineScope

```kotlin
abstract class BasePresenterImpl<View : BaseView> : BasePresenter<View>, CoroutineScope {

    private lateinit var view: View

    protected var parentJob = Job()

    override fun setView(view: View) {
        this.view = view
    }

    fun onDestroy() = cancel()

    override val coroutineContext: CoroutineContext
        get() = Dispatchers.Default + parentJob
}
```

CoroutineScope

```
// Somewhere in the presenter
launch {
  println("This is a coroutine")
}
```

CoroutineContext

N
OO

CoroutineContext

- A **set** of CoroutineContext elements
- Each element dictates one important piece of the puzzle
- **Lifecyle, Threading, Exception handling**

CoroutineContext

```
override val coroutineContext: CoroutineContext
    get() = Dispatchers.Default + parentJob
```

CoroutineContext

```
/**
 * Persistent context for the coroutine. It is an indexed set of [Element] instances.
 * An indexed set is a mix between a set and a map.
 * Every element in this set has a unique [Key]. Keys are compared _by reference_.
 */
@SinceKotlin("1.3")
public interface CoroutineContext {
    /**
     * Returns the element with the given [key] from this context or `null`.
     * Keys are compared _by reference_, that is to get an element from the context the
reference to its actual key
     * object must be presented to this function.
     */
    public operator fun <E : Element> get(key: Key<E>): E?

..
```

Where to find Contexts

- Jobs implement the Context interface
- ContinuationInterceptor (revolves around threading)
- CoroutineExceptionHandler (pretty self-explanatory)

Suspension functions

Suspension functions

- Functions which don’t have to be executed linearly
- Can be paused and resumed at any point in time, as many times as needed
- Rely on continuations
- **Suspend** modifier

CoroutineContext

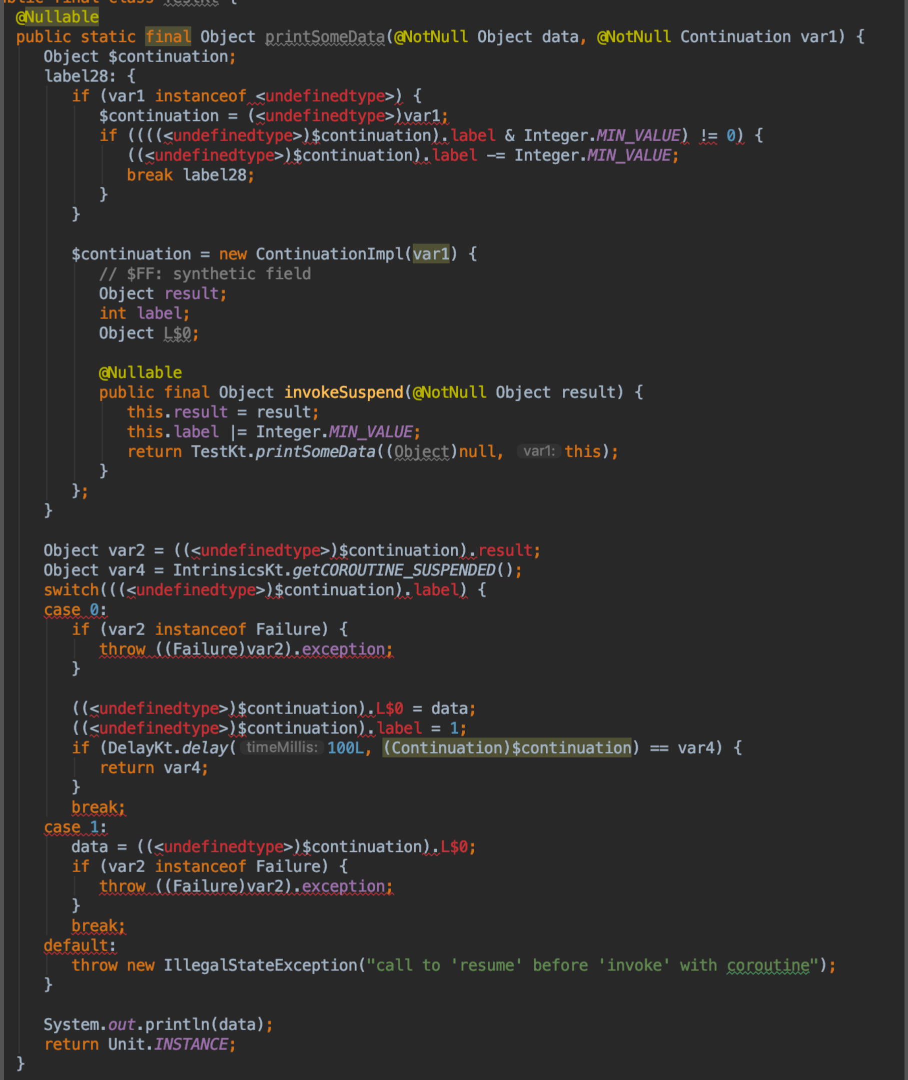
```
fun printSomeData(data: Any) {
  println(data)
}

public final class TestKt {
   public static final void printSomeData(@NotNull Object data) {
      Intrinsics.checkParameterIsNotNull(data, "data");
      System.out.println(data);
   }
}
```

CoroutineContext

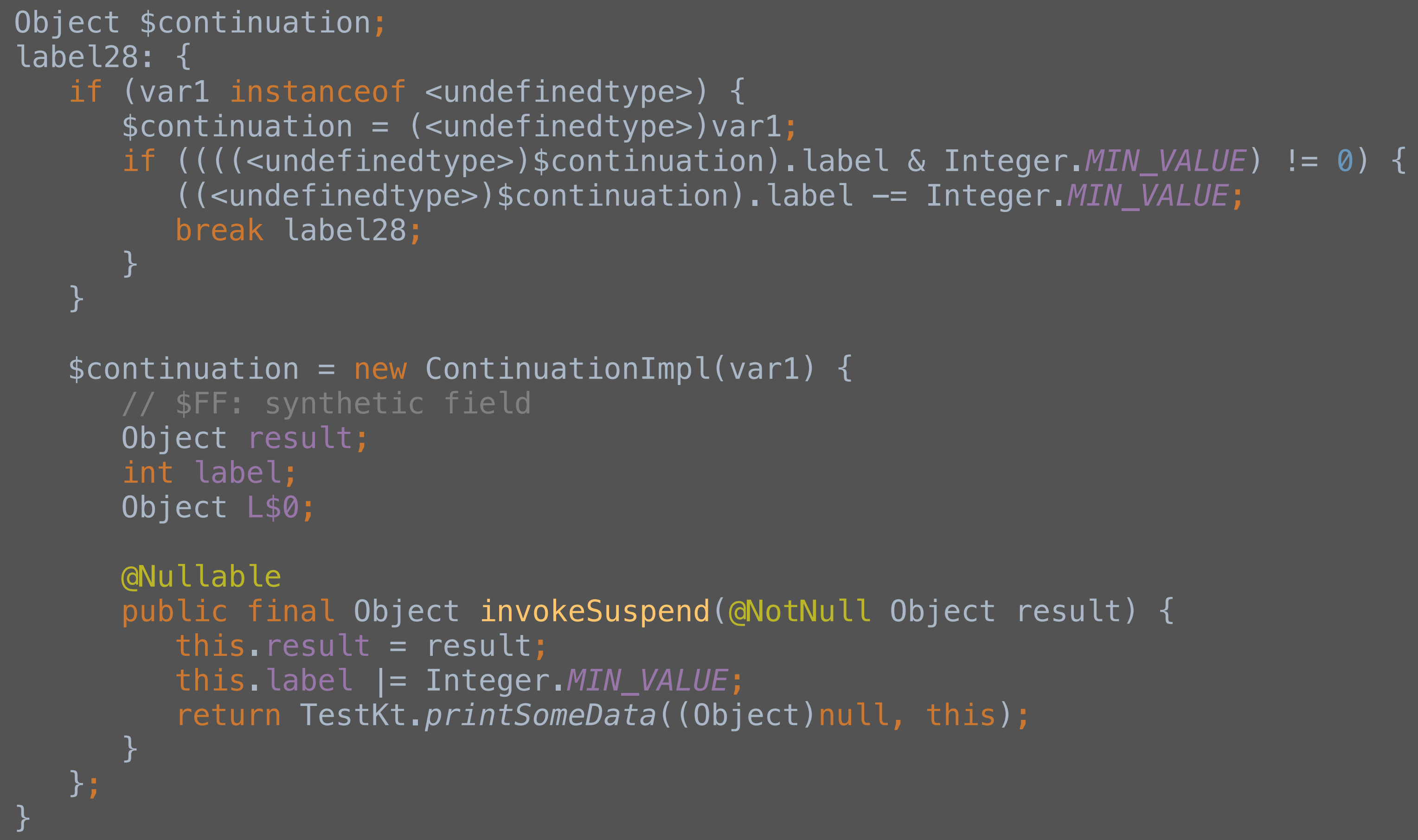
```
suspend fun printSomeData(data: Any) {
  println(data)
}

public final class TestKt {
  @Nullable
  public static final Object printSomeData(@NotNull Object data,
  @NotNull Continuation var1) {
    System.out.println(data);
    return Unit.INSTANCE;
  }
}
```

CoroutineContext

```kotlin
suspend fun printSomeData(data: Any) {
  delay(100)
  println(data)
}
```



```
ublic final class TestKt {
  @Nullable
  public static final Object printSomeData(@NotNull Object data, @NotNull Continuation var1) {
     Object $continuation;
     label28: {
        if (var1 instanceof <undefinedtype>) {
           $continuation = (<undefinedtype>)var1;
           if ((((<undefinedtype>)$continuation).label & Integer.MIN_VALUE) != 0) {
              ((<undefinedtype>)$continuation).label -= Integer.MIN_VALUE;
              break label28;
           }
        }

        $continuation = new ContinuationImpl(var1) {
           // $FF: synthetic field
           Object result;
           int label;
           Object L$0;

           @Nullable
           public final Object invokeSuspend(@NotNull Object result) {
              this.result = result;
              this.label |= Integer.MIN_VALUE;
              return TestKt.printSomeData((Object)null,  var1: this);
           }
        };
     }

     Object var2 = ((<undefinedtype>)$continuation).result;
     Object var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
     switch(((<undefinedtype>)$continuation).label) {
     case 0:
        if (var2 instanceof Failure) {
           throw ((Failure)var2).exception;
        }

        ((<undefinedtype>)$continuation).L$0 = data;
        ((<undefinedtype>)$continuation).label = 1;
        if (DelayKt.delay( timeMillis: 100L, (Continuation)$continuation) == var4) {
           return var4;
        }
        break;
     case 1:
        data = ((<undefinedtype>)$continuation).L$0;
        if (var2 instanceof Failure) {
           throw ((Failure)var2).exception;
        }
        break;
     default:
        throw new IllegalStateException("call to 'resume' before 'invoke' with coroutine");
     }

     System.out.println(data);
     return Unit.INSTANCE;
  }
```



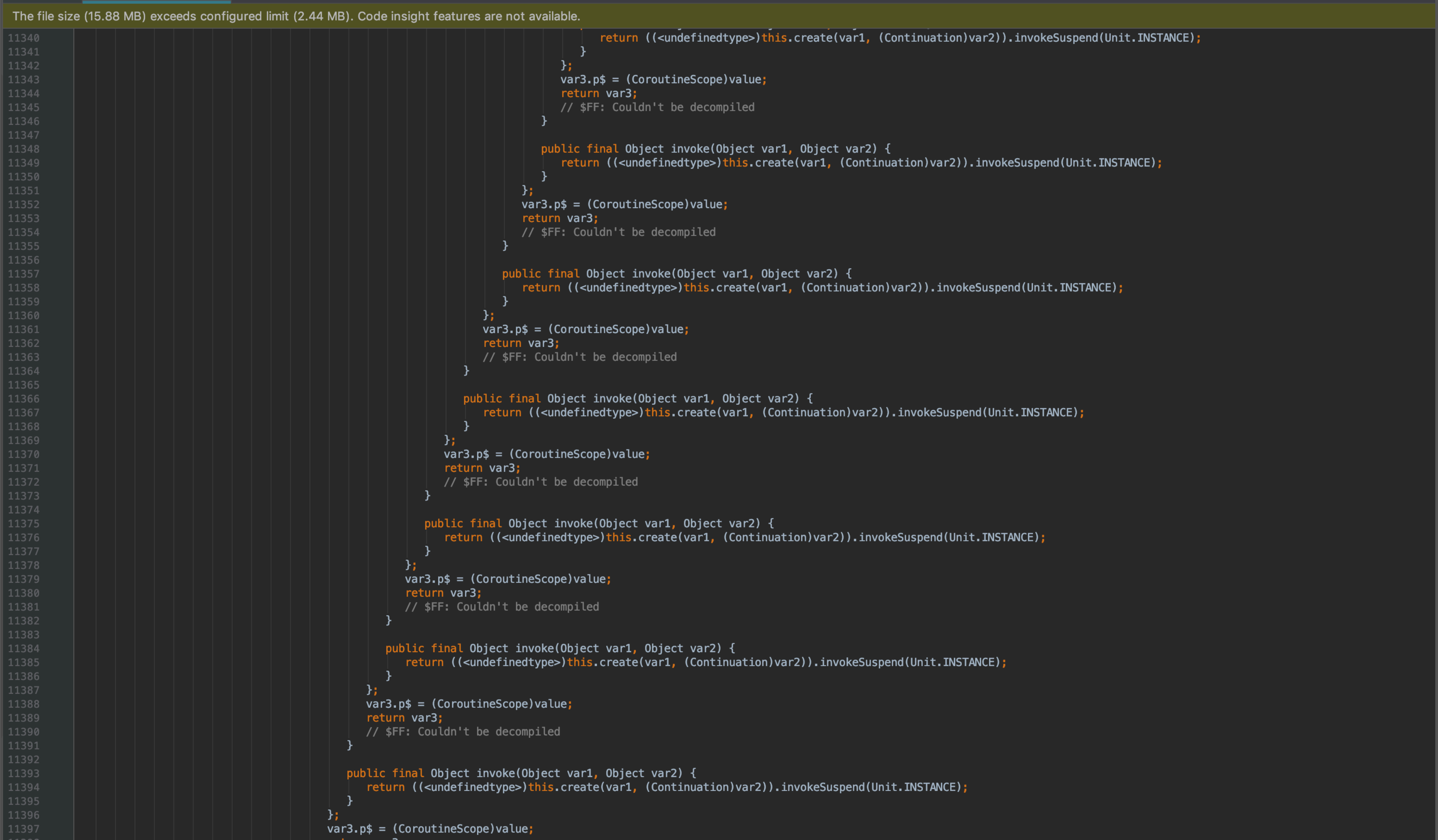
```
Object $continuation;
label28: {
   if (var1 instanceof <undefinedtype>) {
      $continuation = (<undefinedtype>)var1;
      if ((((<undefinedtype>)$continuation).label & Integer.MIN_VALUE) != 0) {
         ((<undefinedtype>)$continuation).label -= Integer.MIN_VALUE;
         break label28;
      }
   }

   $continuation = new ContinuationImpl(var1) {
      // $FF: synthetic field
      Object result;
      int label;
      Object L$0;

      @Nullable
      public final Object invokeSuspend(@NotNull Object result) {
         this.result = result;
         this.label |= Integer.MIN_VALUE;
         return TestKt.printSomeData((Object)null, this);
      }
   };
}
```

```
Object var2 = ((<undefinedtype>)$continuation).result;
Object var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
switch(((<undefinedtype>)$continuation).label) {
case 0:
   if (var2 instanceof Failure) {
      throw ((Failure)var2).exception;
   }

   ((<undefinedtype>)$continuation).L$0 = data;
   ((<undefinedtype>)$continuation).label = 1;
   if (DelayKt.delay(100L, (Continuation)$continuation) == var4) {
      return var4;
   }
   break;
case 1:
   data = ((<undefinedtype>)$continuation).L$0;
   if (var2 instanceof Failure) {
      throw ((Failure)var2).exception;
   }
   break;
default:
   throw new IllegalStateException("call to 'resume' before 'invoke' with coroutine");
}
```

```
…
System.out.println(data);
return Unit.INSTANCE;
```

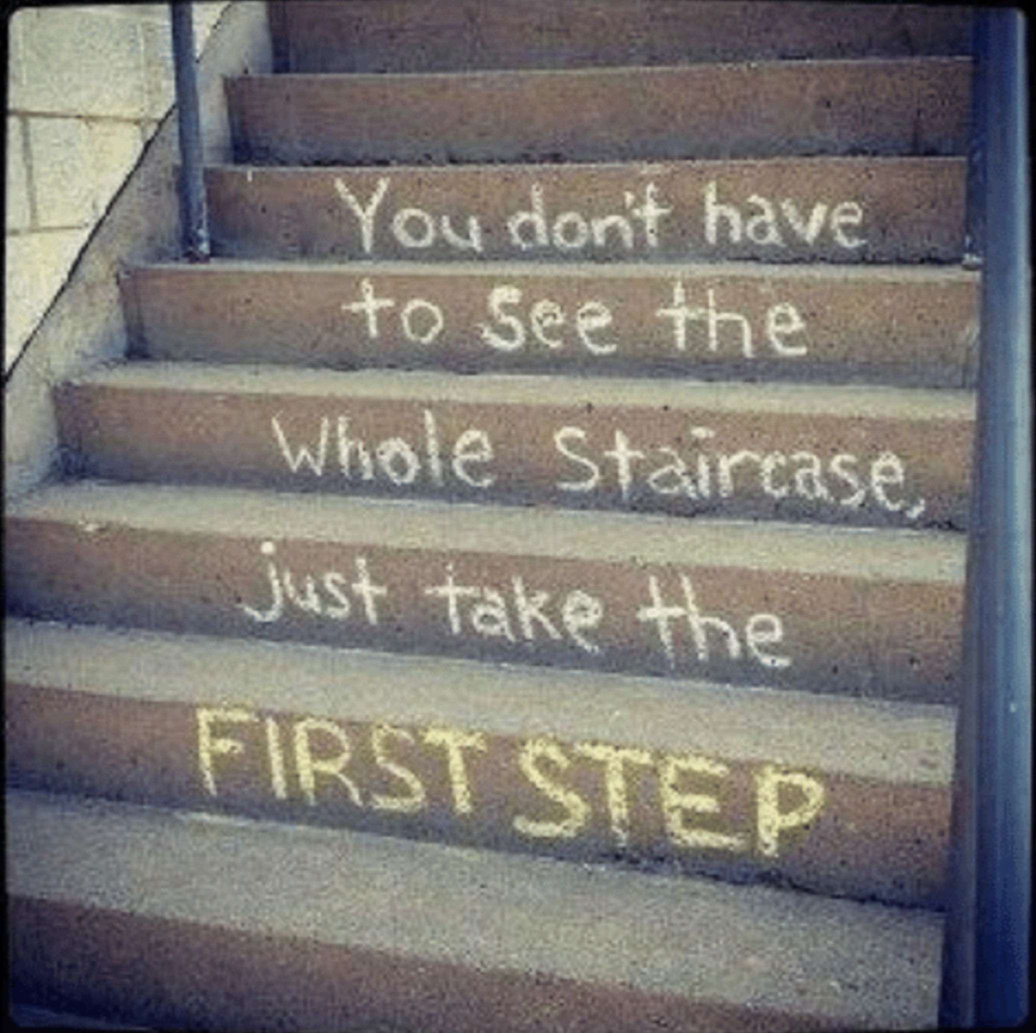
CoroutineContext

```
suspend fun printSomeData(data: Any) {
  delay(100)
  println(data)

  val something = GlobalScope.async {
    ""
  }

  something.await()
}
```



The file size (15.88 MB) exceeds configured limit (2.44 MB). Code insight features are not available.

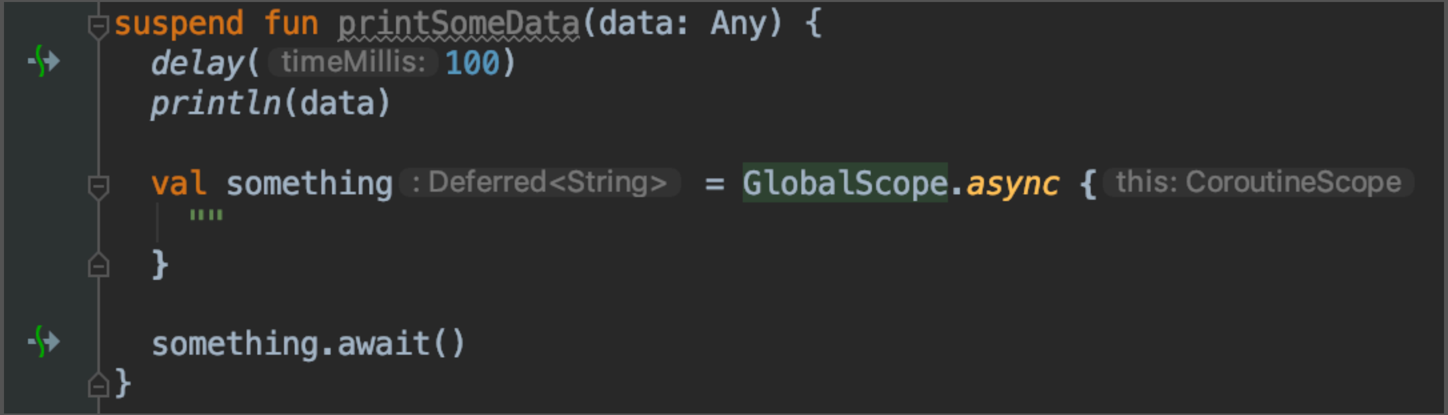
```
                                                        return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                                                    }
                                                };
                                                var3.p$ = (CoroutineScope)value;
                                                return var3;
                                                // $FF: Couldn't be decompiled
                                            }

                                            public final Object invoke(Object var1, Object var2) {
                                                return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                                            }
                                        };
                                        var3.p$ = (CoroutineScope)value;
                                        return var3;
                                        // $FF: Couldn't be decompiled
                                    }

                                    public final Object invoke(Object var1, Object var2) {
                                        return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                                    }
                                };
                                var3.p$ = (CoroutineScope)value;
                                return var3;
                                // $FF: Couldn't be decompiled
                            }

                            public final Object invoke(Object var1, Object var2) {
                                return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                            }
                        };
                        var3.p$ = (CoroutineScope)value;
                        return var3;
                        // $FF: Couldn't be decompiled
                    }

                    public final Object invoke(Object var1, Object var2) {
                        return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                    }
                };
                var3.p$ = (CoroutineScope)value;
                return var3;
                // $FF: Couldn't be decompiled
            }

            public final Object invoke(Object var1, Object var2) {
                return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
            }
        };
        var3.p$ = (CoroutineScope)value;
        return var3;
        // $FF: Couldn't be decompiled
    }

    public final Object invoke(Object var1, Object var2) {
        return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
    }
};
var3.p$ = (CoroutineScope)value;
```


You don't have
to See the
Whole Staircase,
just take the
FIRST STEP

So why does this happen?

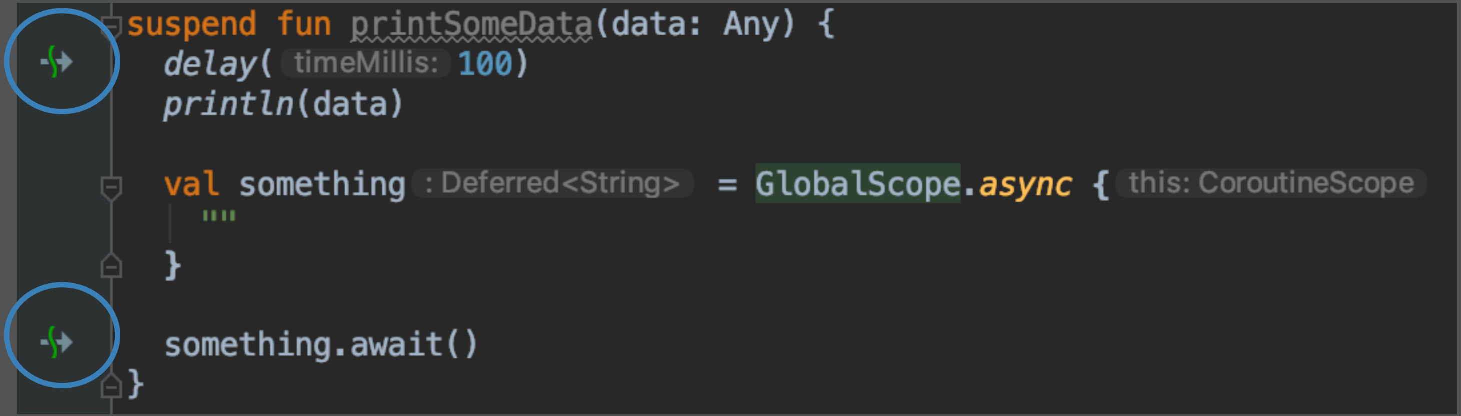
Suspension points

Suspension points

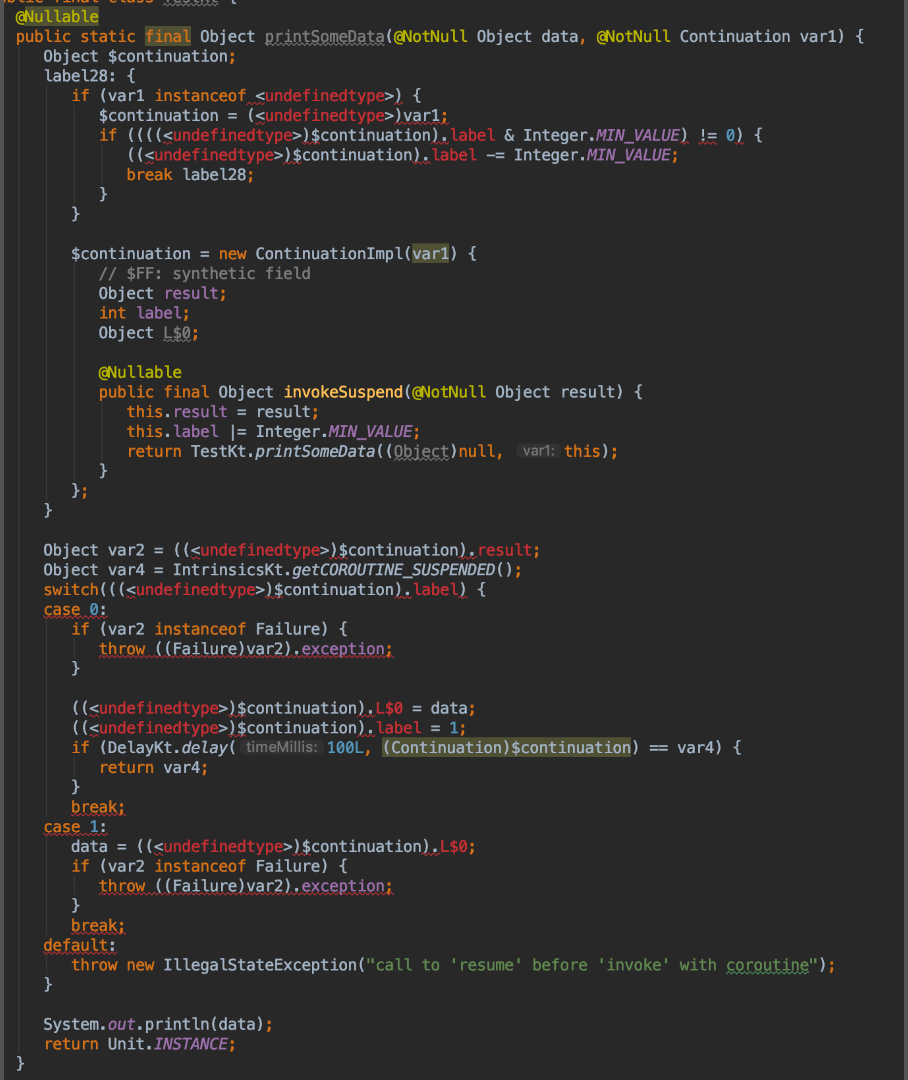


```
suspend fun printSomeData(data: Any) {
    delay( timeMillis: 100)
    println(data)

    val something : Deferred<String>  = GlobalScope.async { this: CoroutineScope
        ""
    }

    something.await()
}
```

Suspension points



```
suspend fun printSomeData(data: Any) {
    delay( timeMillis: 100)
    println(data)

    val something : Deferred<String> = GlobalScope.async { this: CoroutineScope
        ""
    }

    something.await()
}
```

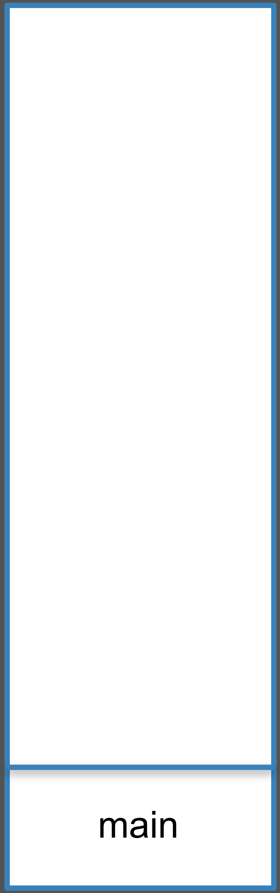


```
ublic final class TestKt {
    @Nullable
    public static final Object printSomeData(@NotNull Object data, @NotNull Continuation var1) {
       Object $continuation;
       label28: {
          if (var1 instanceof <undefinedtype>) {
             $continuation = (<undefinedtype>)var1;
             if ((((<undefinedtype>)$continuation).label & Integer.MIN_VALUE) != 0) {
                ((<undefinedtype>)$continuation).label -= Integer.MIN_VALUE;
                break label28;
             }
          }

          $continuation = new ContinuationImpl(var1) {
             // $FF: synthetic field
             Object result;
             int label;
             Object L$0;

             @Nullable
             public final Object invokeSuspend(@NotNull Object result) {
                this.result = result;
                this.label |= Integer.MIN_VALUE;
                return TestKt.printSomeData((Object)null,  var1: this);
             }
          };
       }

       Object var2 = ((<undefinedtype>)$continuation).result;
       Object var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
       switch(((<undefinedtype>)$continuation).label) {
       case 0:
          if (var2 instanceof Failure) {
             throw ((Failure)var2).exception;
          }

          ((<undefinedtype>)$continuation).L$0 = data;
          ((<undefinedtype>)$continuation).label = 1;
          if (DelayKt.delay( timeMillis: 100L, (Continuation)$continuation) == var4) {
             return var4;
          }
          break;
       case 1:
          data = ((<undefinedtype>)$continuation).L$0;
          if (var2 instanceof Failure) {
             throw ((Failure)var2).exception;
          }
          break;
       default:
          throw new IllegalStateException("call to 'resume' before 'invoke' with coroutine");
       }

       System.out.println(data);
       return Unit.INSTANCE;
    }
```

Continuations

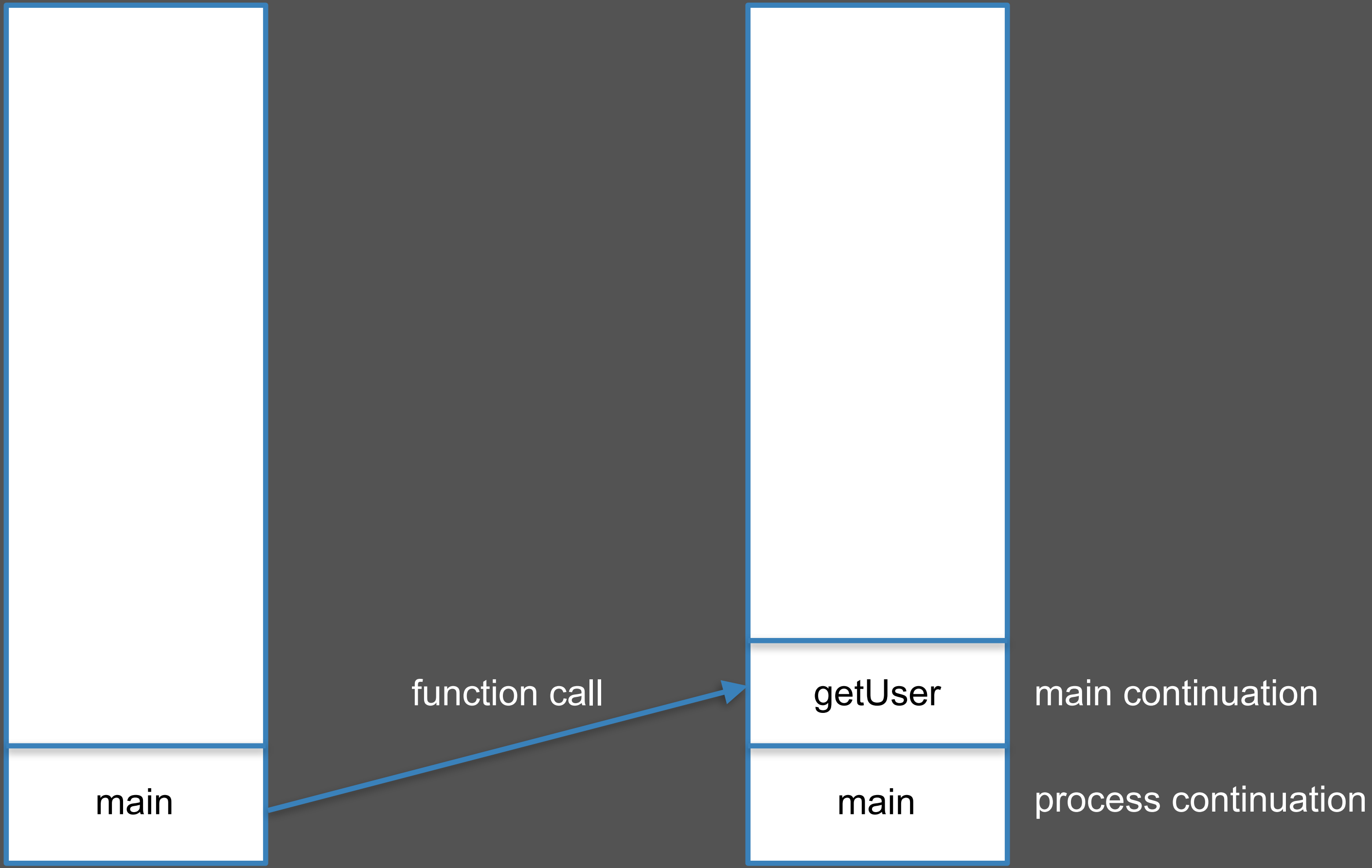
Continuations

- Low level callbacks, which devise and manipulate the execution flow
- The system already has continuations implemented
- Hold the program state



main

The process continuation



main

function call

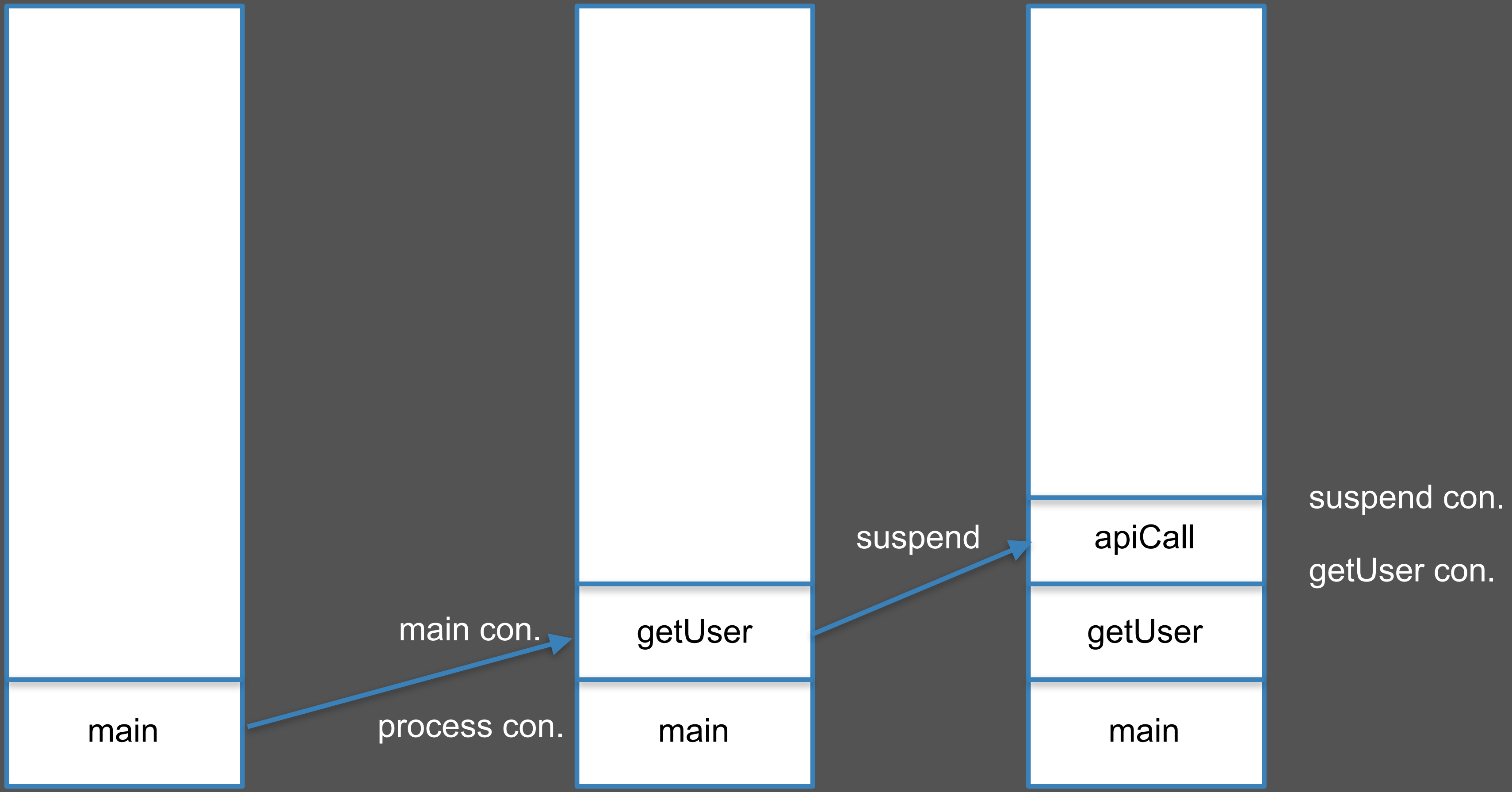
getUser

main continuation

main

process continuation

The process continuation



main

main con.

process con.

getUser

main

suspend

apiCall

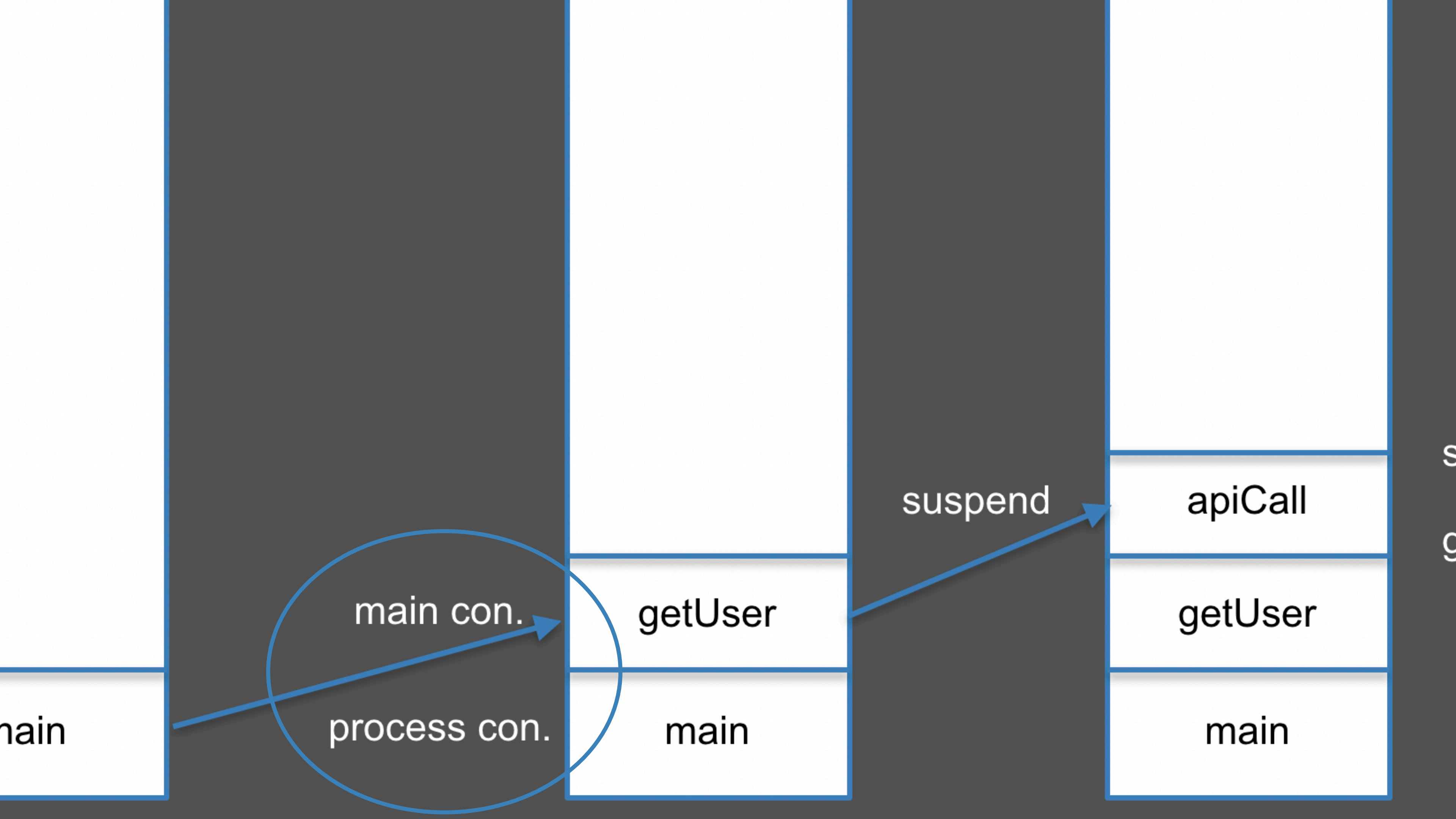
getUser

main

suspend con.

getUser con.

The process continuation



nain

main con.

process con.

getUser

main

suspend

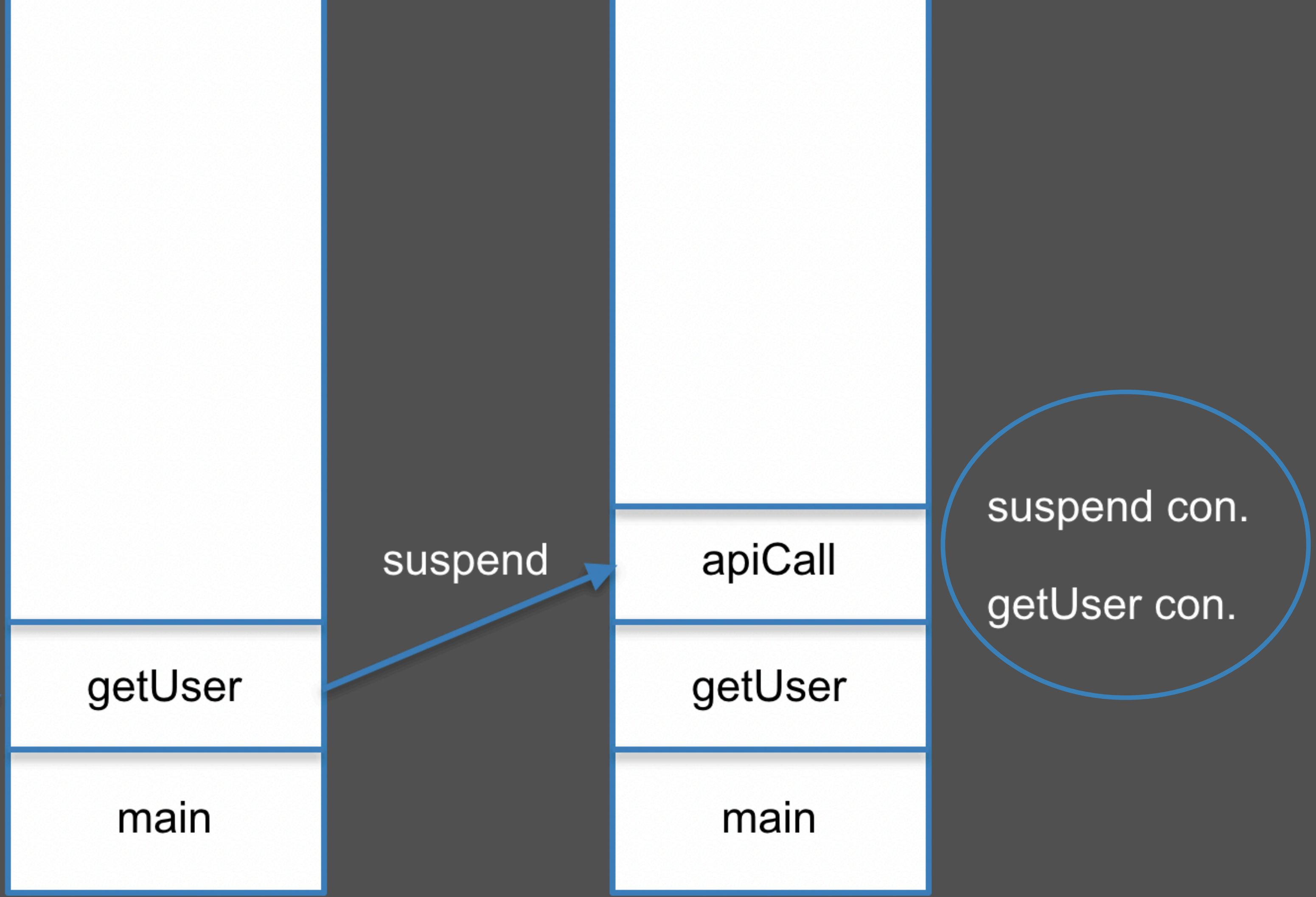
apiCall

getUser

main

s

g



getUser

main

suspend

apiCall

getUser

main

suspend con.

getUser con.

Continuations

- Can be cascade -> exceptions, returns
- Do not create multiple stack entries
- Single stack entry, multiple execution flows

Interceptors and handlers

ContinuationInterceptor

- Handle the internal threading of coroutines
- Dispatchers class - Main, IO, Unconfined, Default
- Really just a relay for execution flows

CoroutineExceptionHandler

```
GlobalScope.launch(Dispatchers.IO) {
  val result = getExpensiveResult()

  launch(Dispatchers.Main) {
    updateUi(result)
  }
}
```

CoroutineExceptionHandler

- Handle exceptions within coroutines!
- Provide you with the context, so you can restart a coroutine, or create a new one

CoroutineExceptionHandler

```kotlin
fun main(args: Array<String>) {

  GlobalScope.launch(context = handler) {
    throw IllegalArgumentException()
  }

  while (true) {

  }
}

val handler = CoroutineExceptionHandler { coroutineContext, throwable ->
  println(coroutineContext)

  if (throwable is IllegalArgumentException) {
    println("R.I.P. coroutine")
  }
}
```

Let’s go back to the hands on concepts

λ

How to share values

- Shared data -> easiest, most volatile
- Queues, polling mechanisms -> cool, but can be complex
- Futures, promises -> really good, safe, cheap but they can scale bad

async/await

async/await

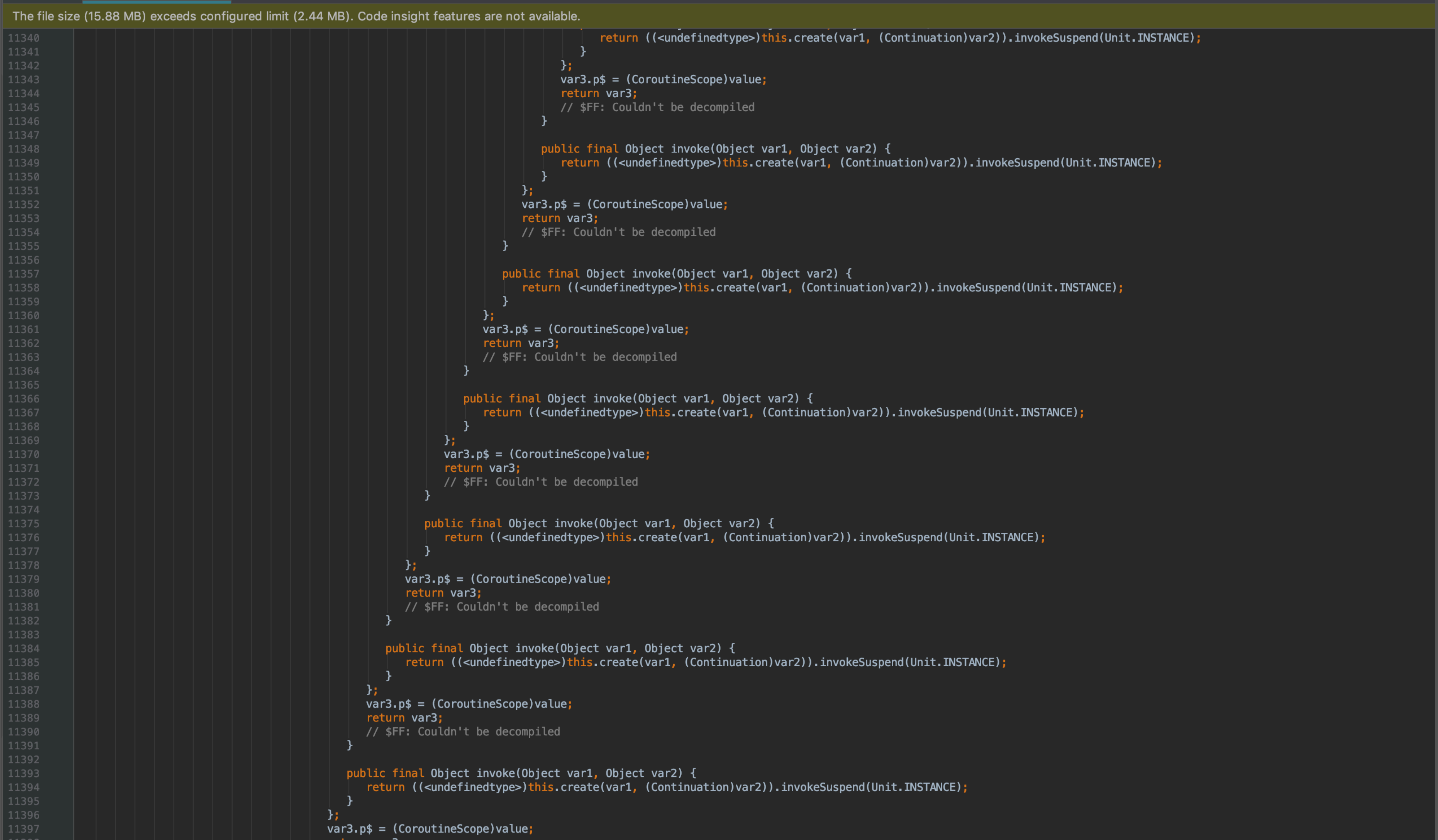
- Provide an asynchronous construct, which can return values
- Make the syntax sequential, and understandable
- Rely on coroutines

async/await

```
with(GlobalScope) {

  launch {
    val expensiveResult = async { getExpensiveResult() }

  }
}
```

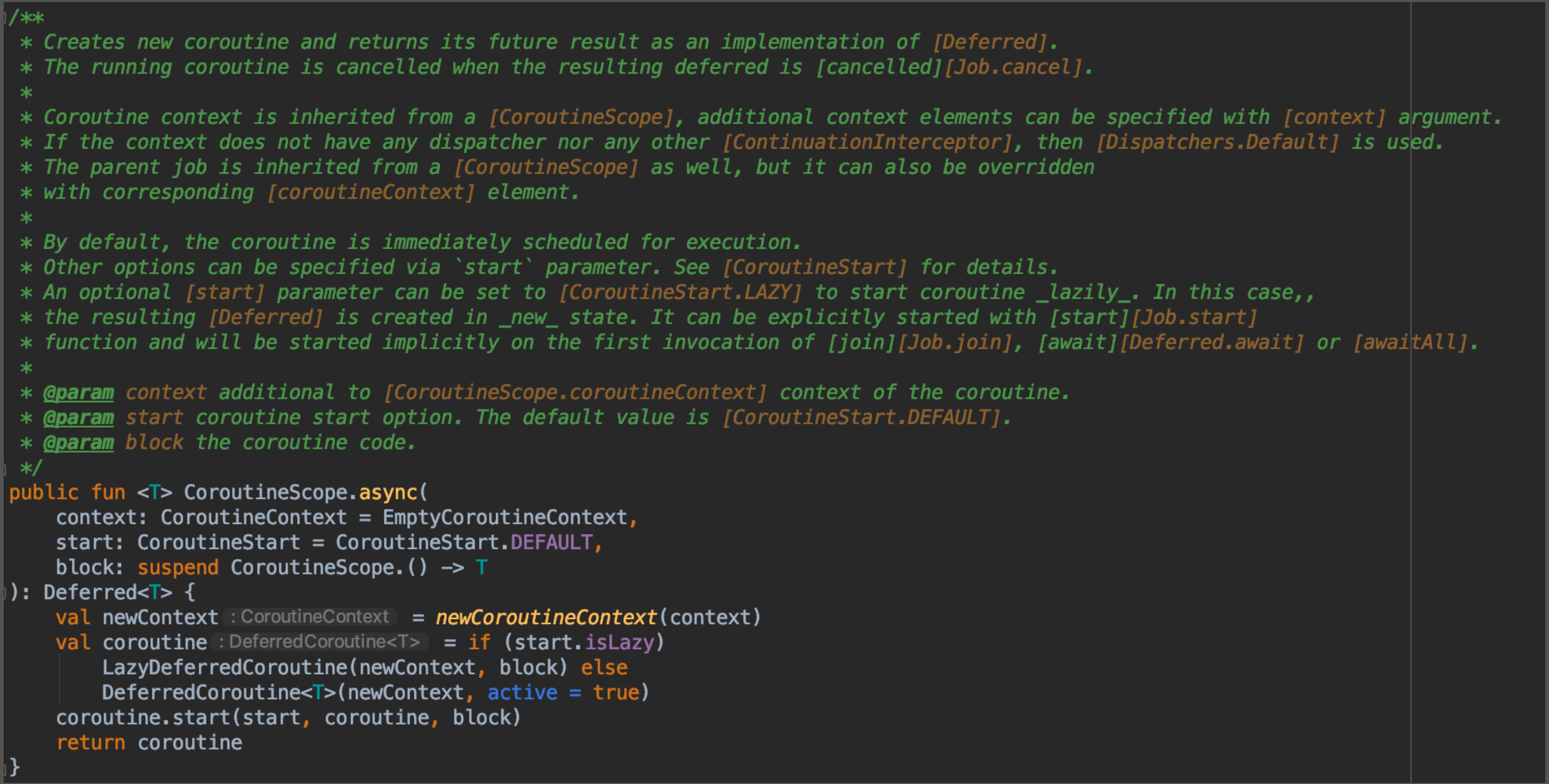
async/await

```
launch {
    val expensiveResultDeferred = async { getExpensiveResult() }

    val actualResult = expensiveResultDeferred.await()
    println(actualResult)
  }
}
```



The file size (15.88 MB) exceeds configured limit (2.44 MB). Code insight features are not available.

```
                                          return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                                       }
                                    };
                                    var3.p$ = (CoroutineScope)value;
                                    return var3;
                                    // $FF: Couldn't be decompiled
                                 }

                                 public final Object invoke(Object var1, Object var2) {
                                    return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                                 }
                              };
                              var3.p$ = (CoroutineScope)value;
                              return var3;
                              // $FF: Couldn't be decompiled
                           }

                           public final Object invoke(Object var1, Object var2) {
                              return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                           }
                        };
                        var3.p$ = (CoroutineScope)value;
                        return var3;
                        // $FF: Couldn't be decompiled
                     }

                     public final Object invoke(Object var1, Object var2) {
                        return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
                     }
                  };
                  var3.p$ = (CoroutineScope)value;
                  return var3;
                  // $FF: Couldn't be decompiled
               }

               public final Object invoke(Object var1, Object var2) {
                  return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
               }
            };
            var3.p$ = (CoroutineScope)value;
            return var3;
            // $FF: Couldn't be decompiled
         }

         public final Object invoke(Object var1, Object var2) {
            return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
         }
      };
      var3.p$ = (CoroutineScope)value;
      return var3;
      // $FF: Couldn't be decompiled
   }

   public final Object invoke(Object var1, Object var2) {
      return ((<undefinedtype>)this.create(var1, (Continuation)var2)).invokeSuspend(Unit.INSTANCE);
   }
};
var3.p$ = (CoroutineScope)value;
```



```kotlin
/**
 * Creates new coroutine and returns its future result as an implementation of [Deferred].
 * The running coroutine is cancelled when the resulting deferred is [cancelled][Job.cancel].
 *
 * Coroutine context is inherited from a [CoroutineScope], additional context elements can be specified with [context] argument.
 * If the context does not have any dispatcher nor any other [ContinuationInterceptor], then [Dispatchers.Default] is used.
 * The parent job is inherited from a [CoroutineScope] as well, but it can also be overridden
 * with corresponding [coroutineContext] element.
 *
 * By default, the coroutine is immediately scheduled for execution.
 * Other options can be specified via `start` parameter. See [CoroutineStart] for details.
 * An optional [start] parameter can be set to [CoroutineStart.LAZY] to start coroutine _lazily_. In this case,,
 * the resulting [Deferred] is created in _new_ state. It can be explicitly started with [start][Job.start]
 * function and will be started implicitly on the first invocation of [join][Job.join], [await][Deferred.await] or [awaitAll].
 *
 * @param context additional to [CoroutineScope.coroutineContext] context of the coroutine.
 * @param start coroutine start option. The default value is [CoroutineStart.DEFAULT].
 * @param block the coroutine code.
 */
public fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T> {
    val newContext : CoroutineContext  = newCoroutineContext(context)
    val coroutine : DeferredCoroutine<T>  = if (start.isLazy)
        LazyDeferredCoroutine(newContext, block) else
        DeferredCoroutine<T>(newContext, active = true)
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

async/await

```
launch {
  val expensiveResultDeferred = async { getExpensiveResult() }
  val userDeferred = async { getUser() }

  printResults(expensiveResultDeferred.await(), userDeferred.await())
}
```

What about safety?

Safety

```
val launch = GlobalScope.launch {
  val result = async { getExpensiveResult() }

  println(result.await())
}

Thread.sleep(50)
launch.cancel()
```

Safety

```
fun getExpensiveResult(): Int {
  var someCondition = false

  while (true) {
    //do something

    println("Running")
    if (someCondition) {
      break
    }
  }
  return 100
}
```

Structured and explicit code

- Write clear and expressive concurrency code
- Rely on **isActive** parent flags, and **finite** CoroutineScopes
- You don’t have to know everything

Sum it up

- Coroutines use thread pools, and smart low level callbacks
- They do not block threads, as they can be suspended, and navigated with continuations
- Using a set of context elements, you can decorate coroutines
- Coroutines are very safe and clean, but you can still write crappy code

Should I use coroutines?

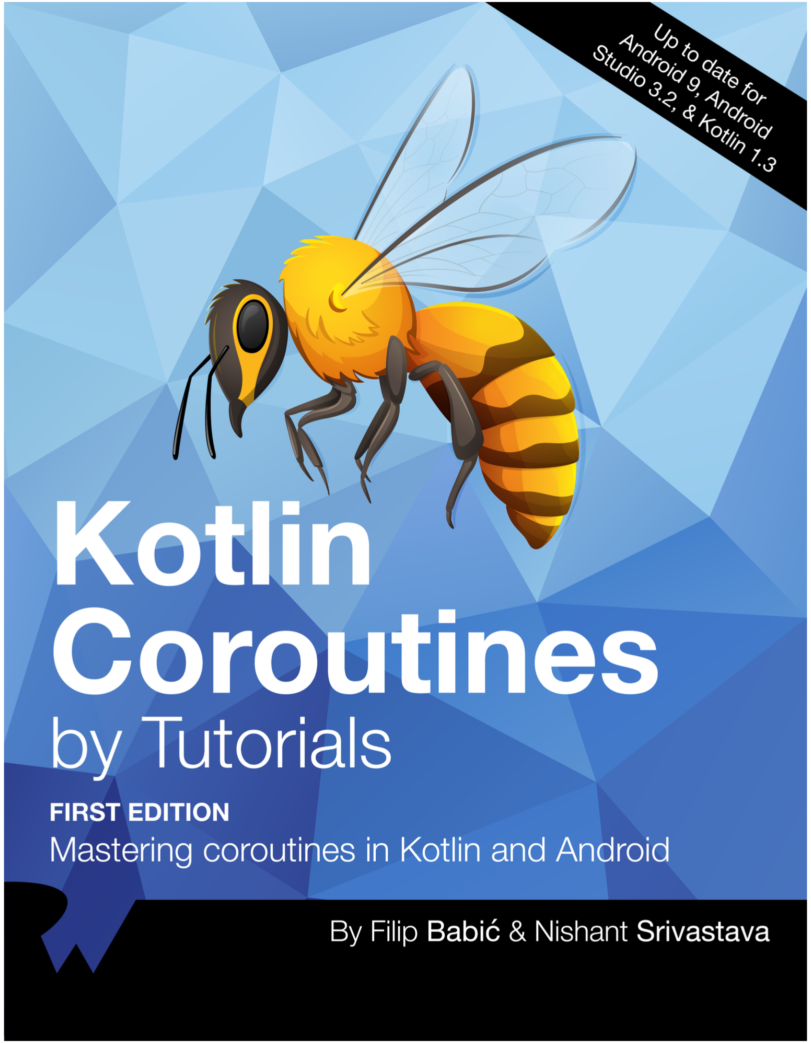
Resources

- The project:

  https://github.com/filbabic/CoroutinesExpoTutorialWorkshop

- Kotlin Coroutines by Tutorials

Kotlin Coroutines by Tutorials


Up to date for Android 9, Android Studio 3.2, & Kotlin 1.3
Kotlin Coroutines
by Tutorials
FIRST EDITION
Mastering coroutines in Kotlin and Android
By Filip Babić & Nishant Srivastava

Questions? :]