Language Design Talk

Coroutines in Kotlin

Andrey.Breslav@JetBrains.com

This talk could have been named…

- async/await/yield
- fibers
- [stackless] continuations

Suspendable Computations

Outline

- Motivation/Examples
- Solutions in other languages
- Kotlin’s Solution
  - Client code
  - Library code
- Compiling Coroutines
- Exception Handling
- Appendix. Serializable Coroutines?

“Legal”

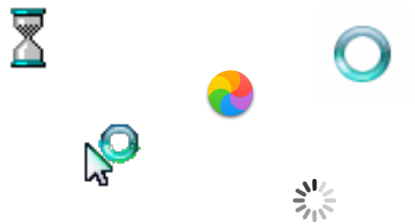
- All I’m saying is no more final than Valhalla ☺

Motivation

UI Thread

Time-consuming operation

```
val image = loadImage(url)
myUI.setImage(image)
```



Latency!

- Blocking bad. Very bad. 

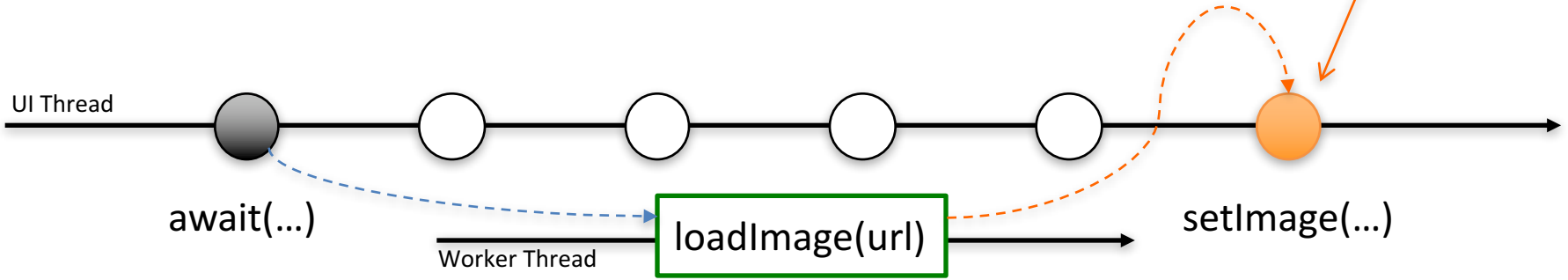
Suspendable Computation

```
asyncUI {
    val image = await(loadImage(url))
    myUI.setImage(image)
}
```

Suspending call

Time-consuming operation

Continuation



```
49  // get all the original jpegs, and create the edited jpegs.
50  function reserveWork()
51  {
52    beanstalkd.reserve(function(err, jobid, payload){ reportError(err);
53      beanstalkd.bury(jobid, 1024, function(err){ reportError(err);
54        var spin = JSON.parse(payload.toString());
55        console.dir(spin);
56        spin.shortid = spin.short_id;
57        var s3key = spin.shortid+"/spin.zip";
58
59        console.log(("["+spin.shortid+"] STARTED (beanjob #"+jobid+")"));
60
61
62        s3.getObject({Bucket: s3bucket, Key: s3key}, function(err, data) { if(err) console.error("Could not get "+s3key); reportError(err);
63          fs.mkdirs(path.dirname(s3key),function(err){ reportError(err);
64            fs.writeFile(s3key,data.Body,function(err){ reportError(err);
65              // spin.zip is on the file system now
66              var cmd = "unzip -o "+s3key+" -d "+path.dirname(s3key);
67              exec(cmd,function(){
68                console.log("Stuff is unzipped!");
69
70                fs.mkdirs(path.dirname(s3key)+"/orig",function(){
71                  var vfs = ["null"];
72                  var rots = [null, "transpose=2", "transpose=2,transpose=2", "transpose=2,transpose=2,transpose=2"];
73                  var rotidx = parseInt(spin.rotation_angle,10)/90;
74                  if(rotidx) vfs.push(rots[rotidx]);
75                  var vf = "-vf "+vfs.join(",");
76                  var ffmpeg_cmd = "ffmpeg -i "+path.dirname(s3key)+"/cap.mp4 -q:v 1 "+vf+" -pix_fmt yuv420p "+path.dirname(s3key)+"/orig/%03d.jpg";
77                  exec(ffmpeg_cmd,function(){
78                    console.log("Done with ffmpeg");
79                    // Upload everything to S3
80                    Step( function(){
81                      for (var i=1; i<=spin.frame_count; i++)
82                      {
83                        var s3key = spin.shortid + "/orig/" + ("00"+i).substr(-3) + ".jpg";
84                        uploadOrig(s3key, this.parallel());
85                      }
86                    },
87                    function(){
88                      fs.readFile(spin.shortid+"/labels.txt",function(err,data){
89                        if(err || !data)
90                          data = new Buffer("{}");
91                        s3.putObject({Bucket: s3bucket, Key: spin.shortid+"/labels.json", ACL: "public-read", ContentType: "text/plain", Body: data}, function(err, data){ reportError(err);
92                          console.log("All files are uploaded");
93                          beanstalkd.use("editor",function(err,tube){ reportError(err,jobid);
94                            beanstalkd.put(1024,0,300,JSON.stringify(spin), function(err,new_jobid){ reportError(err,jobid);
95                              console.log("Added new job to beanstalkd.");
96                              beanstalkd.destroy(jobid, function(){
97                                console.log(("["+spin.shortid+"] FINISHED (beanjob #"+jobid+")"));
98                                reserveWork();
99                              });
100                           });
101                         });
102                       });
103                     });
104                   });
105                 });
106               });
107
108             });
109           });
110         });
111       });
112     });
113
114   });
115 }
```

“Callback Hell”

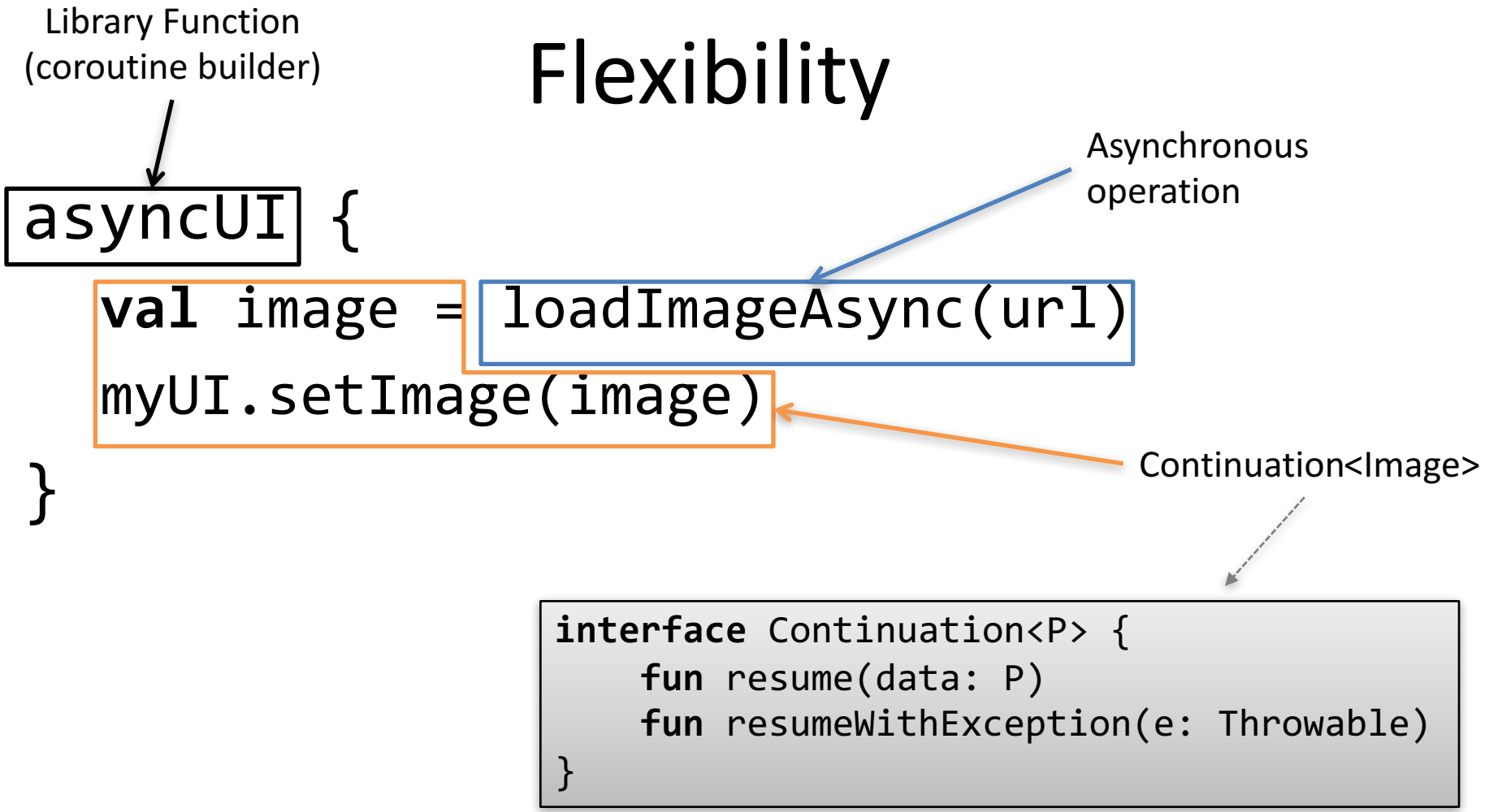
Combining Futures

- `CompletableFuture`

```
.supplyAsync { loadImage(url) }
.thenAccept(myUI::setImage)
```

- so veeery functional ☺

Flexibility



Runtime Support

```
asyncUI {
    val image = await(loadImage(url))
    myUI.setImage(image)
}
```

Continuation<Image>

```
interface Continuation<P> {
    fun resume(data: P)
    fun resumeWithException(e: Throwable)
}
```

Summary: Goals

- **Asynchronous programing** (and more)
  - without explicit callbacks
  - without explicit Future combinators

- **Maximum flexibility** for library designers
  - with minimal runtime support
  - and no macros ☺

Flavors of Coroutines

	Stackless	Stackful
Language restrictions	Use in special contexts ☹	Use anywhere ☺
Implemented in	C#, Scala, Kotlin, …	Quasar, Javaflow, …
Code transformation	Local (compiler magic) ☺	All over the place ☹
Runtime support	Little ☺	Substantial ☹

The C# Way

```
async Task<String> work() {
    await Task.delay(200);
    return “done”;
}

async Task moreWork() {
    Console.WriteLine(“Work started”);
    var str = await work();
    Console.WriteLine($“Work completed: {str}”);
}
```

Example: async/await (I)

type is optional

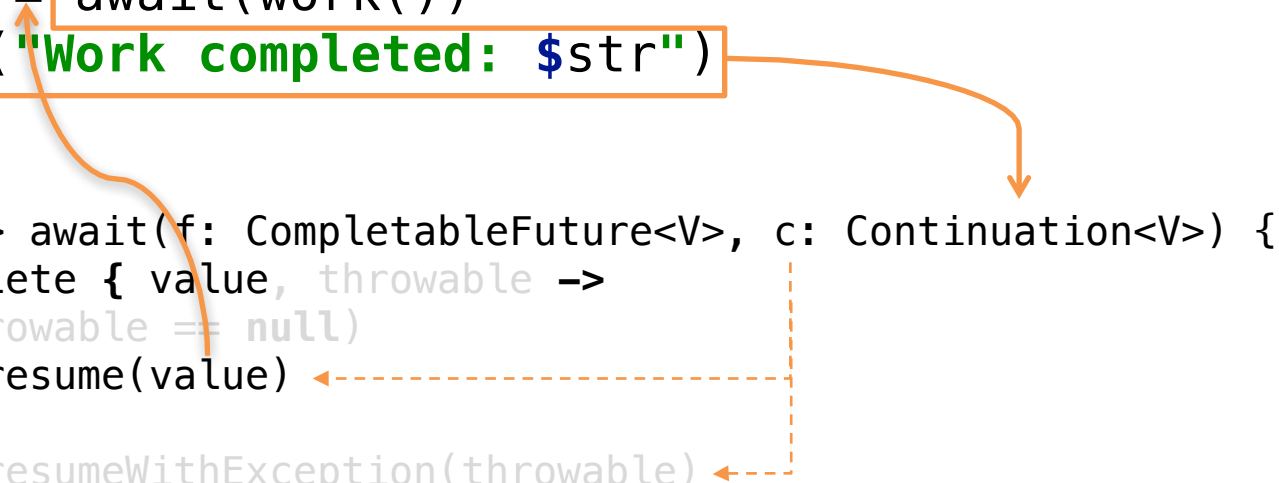
```
fun work(): CompletableFuture<String> = async {
    Thread.sleep(200)
    "done"
}

fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}
```

Example: async/await (I)

```
fun work() = async {
    Thread.sleep(200)
    "done"
}

fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}
```

await()

```
fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}

suspend fun <V> await(f: CompletableFuture<V>, c: Continuation<V>) {
    f.whenComplete { value, throwable ->
        if (throwable == null)
            c.resume(value)
        else
            c.resumeWithException(throwable)
    }
}
```



async()

```
fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}

fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c(controller).resume(Unit)
    return controller.future
}
```

implicit receiver

λ has no params

Controller

```
@AllowSuspendExtensions
class FutureController<T> {
    internal val future = CompletableFuture<T>()

    suspend fun <V> await(f: CompletableFuture<V>, c: Continuation<V>) {
        f.whenComplete { value, throwable ->
          if (throwable == null)
                c.resume(value)
            else
                c.resumeWithException(throwable)
        }
    }

    operator fun handleResult(value: T, c: Continuation<Nothing>) {
        future.complete(value)
    }

    operator fun handleException(t: Throwable, c: Continuation<Nothing>) {
        future.completeExceptionally(t)
    }
}
```

```
fun work() = async {
    Thread.sleep(200)
    "done"
}
```

Extensibility

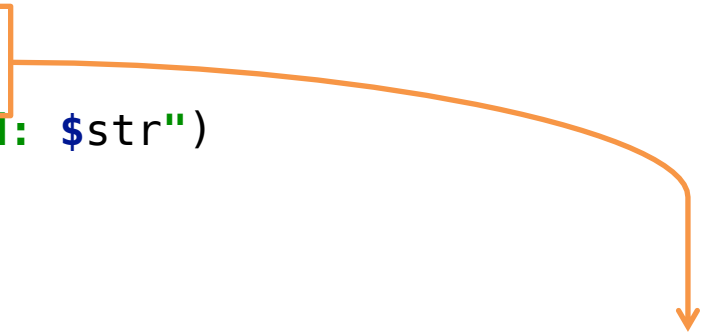
```
suspend fun <V> FutureController<*>.await(
        lf: ListenableFuture<V>, c: Continuation<V>
) {
    Futures.addCallback(lf, object : FutureCallback<V> {
        override fun onSuccess(value: V) {
            c.resume(value)
        }
        override fun onFailure(e: Throwable) {
            c.resumeWithException(throwable)
        }
    })
}

// Example
async {
    val res1 = await(getCompletableFuture())
    val res2 = await(getListeableFuture())
}
```

Summary: Coroutine Libraries

- `fun async(coroutine c: …)`
  - function with a `coroutine` parameter
- `suspend fun await(…, c: Continuation<…>)`
  - function marked `suspend`
  - continuation is implicitly passed in at the call site
- `class Controller`
  - declares `suspend` functions
    - may allow `suspend` extensions
  - declares return/exception handlers

How Suspension Works

```
fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}
```



```
controller.await(
    work(),
    current_continuation
)
return
```

.

Yield (The C# Way)

```
IEnumerable<int> Fibonacci() {
    var cur = 1;
    var next = 1;

    yield return 1;

    while (true) {
        yield return next;

        var tmp = cur + next;
        cur = next;
        next = tmp;
    }
}
```

Infinite (lazy) sequence of Fibonacci numbers

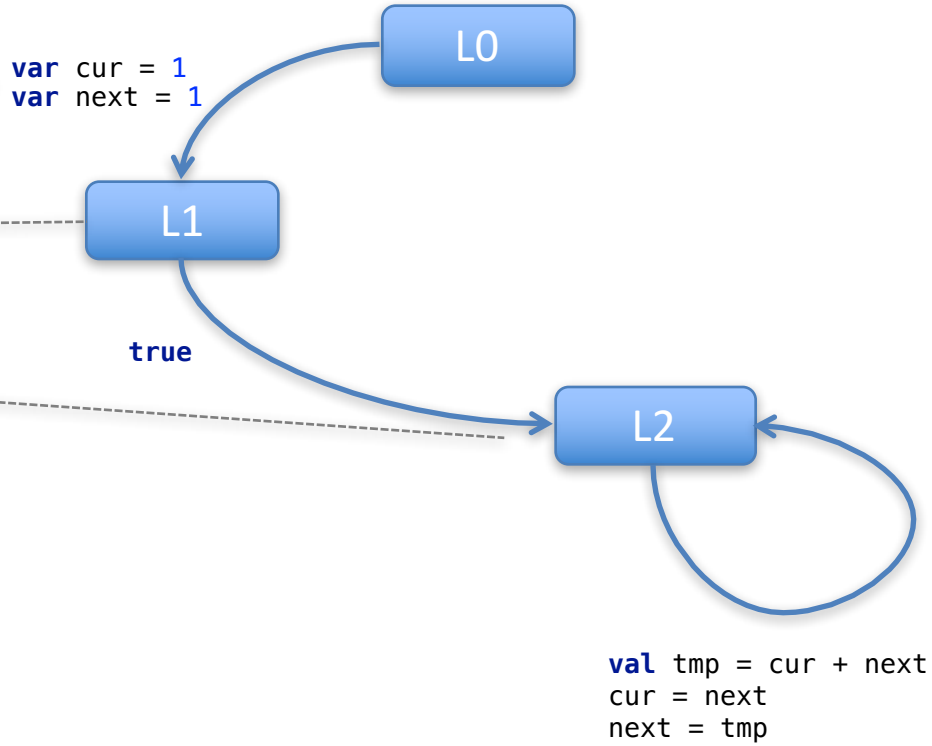
Example: Lazy Fibonacci

```
val fibonacci: Sequence<Int> = generate {
    var cur = 1
    var next = 1

    yield(1)

    while (true) {
        yield(next)

        val tmp = cur + next
        cur = next
        next = tmp
    }
}

assertEquals("1, 1, 2, 3, 5", fibonacci.take(5).joinToString())
```

```
fun <T> generate(
        coroutine c: GeneratorController<T>.() -> Continuation<Unit>
): Sequence<T> =
        object : Sequence<T> {
            override fun iterator(): Iterator<T> {
                val iterator = GeneratorController<T>()
                iterator.setNextStep(c(iterator))
                return iterator
            }
        }

class GeneratorController<T> : AbstractIterator<T>() {
    ...
    suspend fun yield(value: T, c: Continuation<Unit>) {
        setNext(value)
        setNextStep(c)
    }
    ...
}
```

Compiling to State Machines

```
generate {
    var cur = 1
    var next = 1

    yield(1)

    while (true) {
        yield(next)

        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```



Compiling Coroutines (I)

```
val fibonacci = generate {
    var cur = 1
    var next = 1

    yield(1)

    while (true) {
        yield(next)

        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

```
class fibonacci$1 implements Function1,
                             Continuation {

    private GeneratorController controller
    private int label = -2

    private int cur
    private int next
```

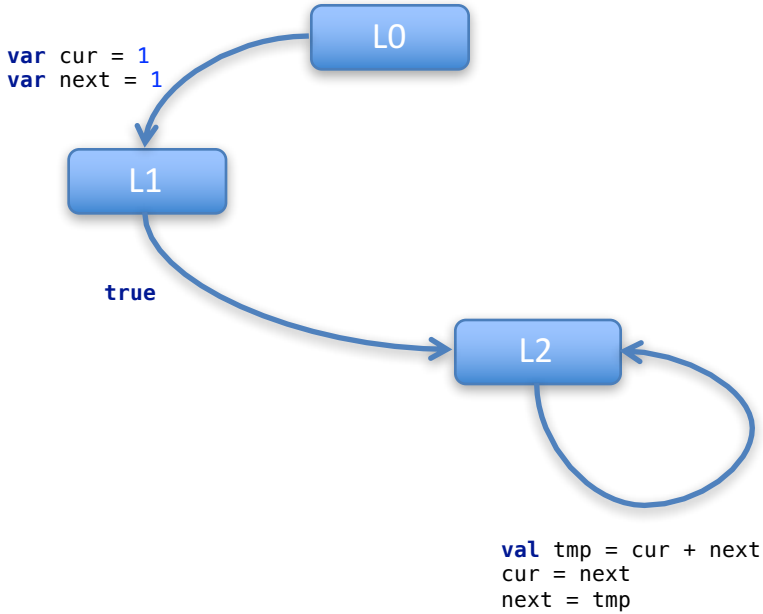
} for shared local variables

```
    public Continuation<Unit> invoke(
                    GeneratorController controller)

    public void resume(Object param)
    public void resumeWithException(Throwable e)
    private void doResume(Object param, Throwable e)
}
```

Compiling Coroutines (II)

- Fields:
  - GeneratorController controller
  - int label

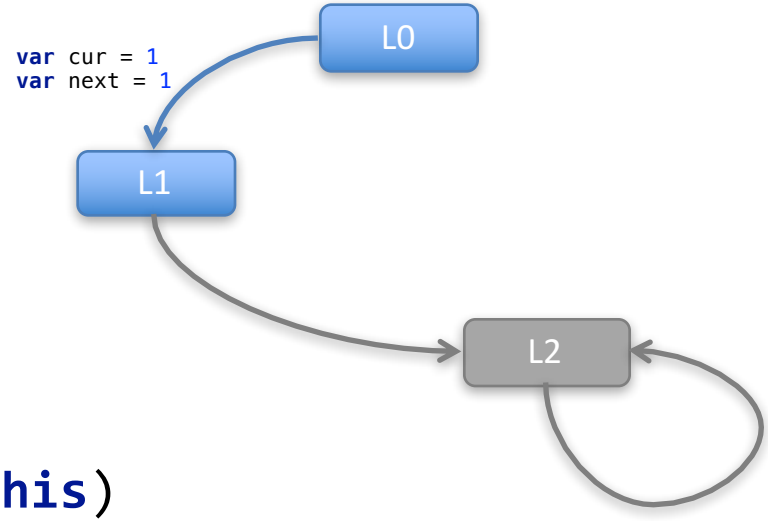
- void doResume(Object param, Throwable e)
  - tableswitch (label)
    case 0: L0
    case 1: L1
    case 2: L2
  - L0:
    ...
    label = 1
    controller.yield(1, /* continuation = */ this)
    return
  - L1:
    ...
    label = 2
    controller.yield(next, /* continuation = */ this)
    return
  - L2:
    ...



Compiling Coroutines (III)

- **L0:**

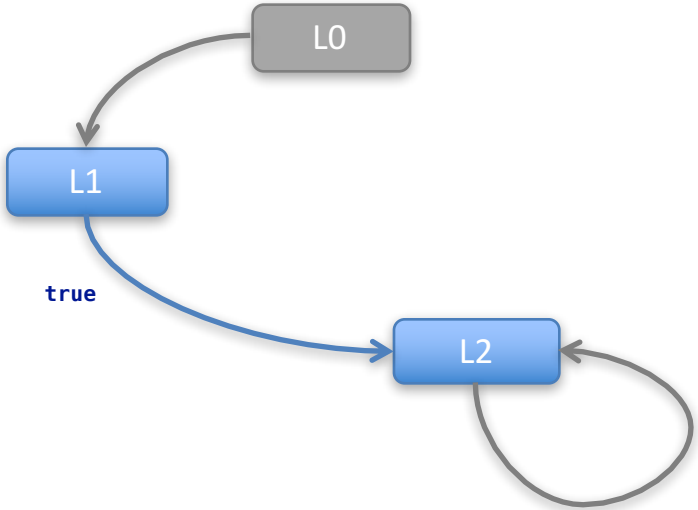
```
var cur = 1
var next = 1

this.cur = cur
this.next = next

this.label = 1
this.controller.yield(1, this)
return
```



Compiling Coroutines (IV)

- `void doResume(Object param, Throwable e)`
  - **L1:**

```
cur = this.cur
next = this.next

if (e != null) throw e

// while (true) {

this.cur = cur
this.next = next

this.label = 2
this.controller.yield(next, this)
return
```



Summary: Compiling Coroutines

- Note: generate()/yield() can be expressed
  - flexibility: ☑

- Coroutine body is compiled to a **state machine**
- Only **one instance** is allocated at runtime

Exception Handling

```
asyncUI {
   val image = await(loadImage(url))
   myUI.setImage(image)
}
```

Throwing and Catching

- Who can **throw**
  - **Synchronous** code (inside a coroutine)
  - **Asynchronous** operations (called from coroutine)
  - Library code (that manages the corouitne)
- Who can **catch**
  - The coroutine itself (user code)
  - Library code

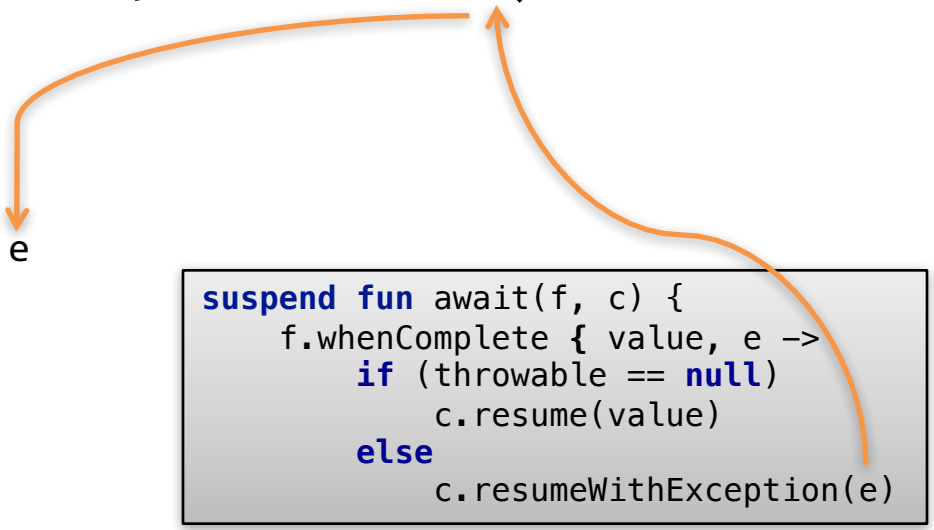
Controller.handleException(e)

```
void doResume(Object param, Throwable e)

      tableswitch (label)
            case 0: L0
            case 1: L1
            case 2: L2
      try {
          L0:
                ...
                label = 1
                controller.await(..., /* continuation = */ this)
                return
          L1:
                ...
                label = 2
                controller.await(..., /* continuation = */ this)
                return
          L2:
                ...

      } catch (Throwable e) {
            controller.handleException(e)
      }
```

Routing Asynchronous Exceptions

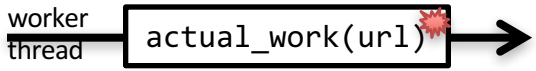
- `void doResume(Object param, Throwable e)`

```
...
– L1:
        // fields -> locals

        if (e != null) throw e

        ...

        // locals -> fields

        this.label = 2
        this.controller.yield(next, this)
        return
```



Example: Exception Handling

```
asyncUI {
    val image = try {
        await(
            loadImage(url)
        )
    } catch(e: Exception) {
        log(e)
        throw e
    }

    myUI.setImage(image)
}
```



Operation order:

- loadImage(url)
  - -> tmp_future
  - -> actual_work()
- await(tmp_future)
- <suspend>
- actual_work() completes
- <resume>
- myUI.setImage(image)

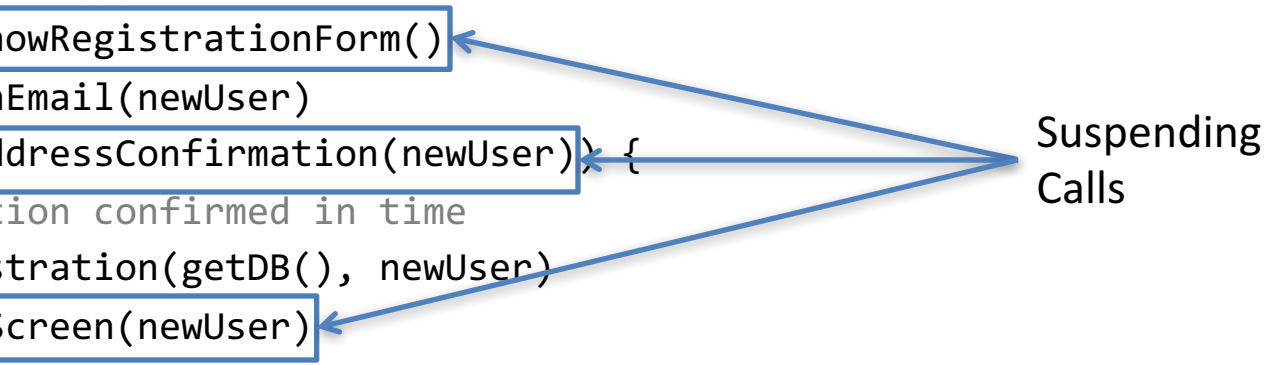
Summary: Exception Handling

- **Uniform** treatment of all exceptions
  - both sync and async
- Default handler: `controller.handleException(e)`

- Not covered in this talk
  - Suspending in `finally` blocks
  - Calling finally blocks through `Continuation<T>`

Appendix. Serializable Coroutines?

```
serializableAsync(getDB()) {
    val newUser = showRegistrationForm()
    sendConfirmationEmail(newUser)
    if (awaitEmailAddressConfirmation(newUser)) {
        // registration confirmed in time
        confirmRegistration(getDB(), newUser)
        showWelcomeScreen(newUser)
    } else {
        // registration not confirmed
        cancelRegistration(getDB(), newUser)
    }
}
```



Data to be serialized:

- label (state of the SM)
- newUser

References

- Language Design Proposal (KEEP)
  - https://github.com/Kotlin/kotlin-coroutines
  - Give your feedback in GitHub Issues
- Example libraries
  - https://github.com/Kotlin/kotlinx.coroutines