Kotlin Coroutines Workshop

Appendix / Extra / Advanced

Composition

Beyond sequential

```
val post = createPost(token, item)
```

Higher-order functions

```
val post = retryIO {
    createPost(token, item)
}
```

Higher-order functions

```
val post = retryIO { createPost(token, item) }

suspend fun <T> retryIO(block: suspend () -> T): T {
    var curDelay = 1000L // start with 1 sec
    while (true) {
        try {
            return block()
        } catch (e: IOException) {
            e.printStackTrace() // log the error
        }
        delay(curDelay)
        curDelay = (curDelay * 2).coerceAtMost(60000L)
    }
}
```

Higher-order functions

```
val post = retryIO { createPost(token, item) }

suspend fun <T> retryIO(block: suspend () -> T): T {
    var curDelay = 1000L // start with 1 sec
    while (true) {
        try {
            return block()
        } catch (e: IOException) {
            e.printStackTrace() // log the error
        }
        delay(curDelay)
        curDelay = (curDelay * 2).coerceAtMost(60000L)
    }
}
```

Higher-order functions

```
val post = retryIO { createPost(token, item) }

suspend fun <T> retryIO(block: suspend () -> T): T {
    var curDelay = 1000L // start with 1 sec
    while (true) {
        try {
            return block()
        } catch (e: IOException) {
            e.printStackTrace() // log the error
        }
        delay(curDelay)
        curDelay = (curDelay * 2).coerceAtMost(60000L)
    }
}
```

suspending lambda

Higher-order functions

```
val post = retryIO { createPost(token, item) }

suspend fun <T> retryIO(block: suspend () -> T): T {
    var curDelay = 1000L // start with 1 sec
    while (true) {
        try {
            return block()
        } catch (e: IOException) {
            e.printStackTrace() // log the error
        }
        delay(curDelay)
        curDelay = (curDelay * 2).coerceAtMost(60000L)
    }
}
```

Higher-order functions

```
val post = retryIO { createPost(token, item) }

suspend fun <T> retryIO(block: suspend () -> T): T {
    var curDelay = 1000L // start with 1 sec
    while (true) {
        try {
            return block()
        } catch (e: IOException) {
            e.printStackTrace() // log the error
        }
        delay(curDelay)
        curDelay = (curDelay * 2).coerceAtMost(60000L)
    }
}
```

Everything like in blocking code

Direct to CPS

How compiler transforms coroutines to callbacks?

Direct code

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Continuations

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Initial continuation

Continuations

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Continuation

Continuations

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Continuation

Convert to CPS?

Callbacks?

```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

Labels

```
suspend fun postItem(item: Item) {
// LABEL 0
    val token = requestToken()
// LABEL 1
    val post = createPost(token, item)
// LABEL 2
    processPost(post)
}
```

Labels

```
suspend fun postItem(item: Item) {
    switch (label) {
      case 0:
          val token = requestToken()
      case 1:
          val post = createPost(token, item)
      case 2:
          processPost(post)
   }
}
```

State

```
suspend fun postItem(item: Item) {
    val sm = object : CoroutineImpl { … }
    switch (sm.label) {
      case 0:
          val token = requestToken()
      case 1:
          val post = createPost(token, item)
      case 2:
          processPost(post)
   }
}
```

CPS Transform

```
fun postItem(item: Item, cont: Continuation) {
    val sm = object : CoroutineImpl { … }
    switch (sm.label) {
      case 0:
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      case 2:
          processPost(post)
   }
}
```

Save state

```
fun postItem(item: Item, cont: Continuation) {
    val sm = …
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      case 2:
          processPost(post)
   }
}
```

Callback

```
fun postItem(item: Item, cont: Continuation) {
    val sm = object : CoroutineImpl { … }
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      case 2:
          processPost(post)
   }
}
```

State Machine as Continuation

Callback

```
fun postItem(item: Item, cont: Continuation) {
    val sm = object : CoroutineImpl {
        fun resume(…) {
            postItem(null, this)
        }
    }
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      …
}
```

Callback

```
fun postItem(item: Item, cont: Continuation) {
    val sm = cont as? ThisSM ?: object : ThisSM {
        fun resume(…) {
            postItem(null, this)
        }
    }
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      …
}
```

Restore state

```
fun postItem(item: Item, cont: Continuation) {
    val sm = …
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          val item = sm.item
          val token = sm.result as Token
          sm.label = 2
          createPost(token, item, sm)
      …
}
```

Continue

```
fun postItem(item: Item, cont: Continuation) {
    val sm = …
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          val item = sm.item
          val token = sm.result as Token
          sm.label = 2
          createPost(token, item, sm)
      …
}
```

State Machine vs Callbacks

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

State Machine vs Callbacks

Reuse closure / state object

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Create new closure

```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

State Machine vs Callbacks

```
suspend fun postItems(items: List<Item>) {
    for (item in items) {
        val token = requestToken()
        val post = createPost(token, item)
        processPost(post)
    }
}
```

Easy loops and higher-order functions

State Machine vs Callbacks

```
suspend fun postItems(items: List<Item>) {
    for (item in items) {
        val token = requestToken()
        val post = createPost(token, item)
        processPost(post)
    }
}
```

Easy loops and higher-order functions

A horrid callback mess

```
fun postItems(items: List<Item>) {
    …
}
```

Dispatcher internals

Continuation Interceptor

```
interface ContinuationInterceptor : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    fun <T> interceptContinuation(continuation: Continuation<T>):
                                                        Continuation<T>

    // …
}
```

Continuation Interceptor

```
interface ContinuationInterceptor : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    fun <T> interceptContinuation(continuation: Continuation<T>):
                                                        Continuation<T>

    // …
}
```

Continuation Interceptor

```
interface ContinuationInterceptor : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    fun <T> interceptContinuation(continuation: Continuation<T>):
                                                        Continuation<T>

    // …
}
```

Dispatched continuation

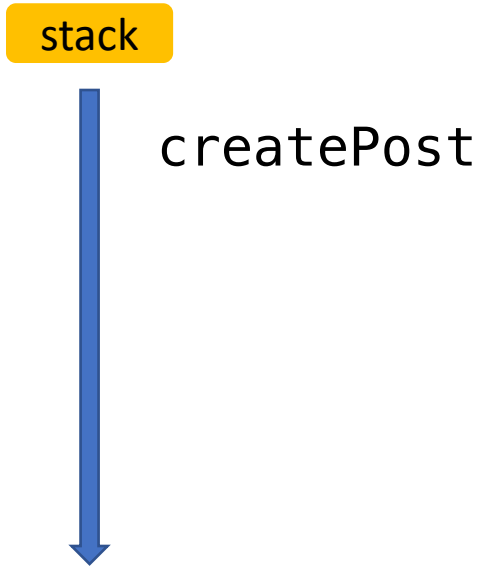
```
class DispatchedContinuation<in T>(
    val dispatcher: CoroutineDispatcher,
    val continuation: Continuation<T>
): Continuation<T> by continuation {

    override fun resume(value: T) {
        dispatcher.dispatch(context, DispatchTask(…))
    }

    …
}
```

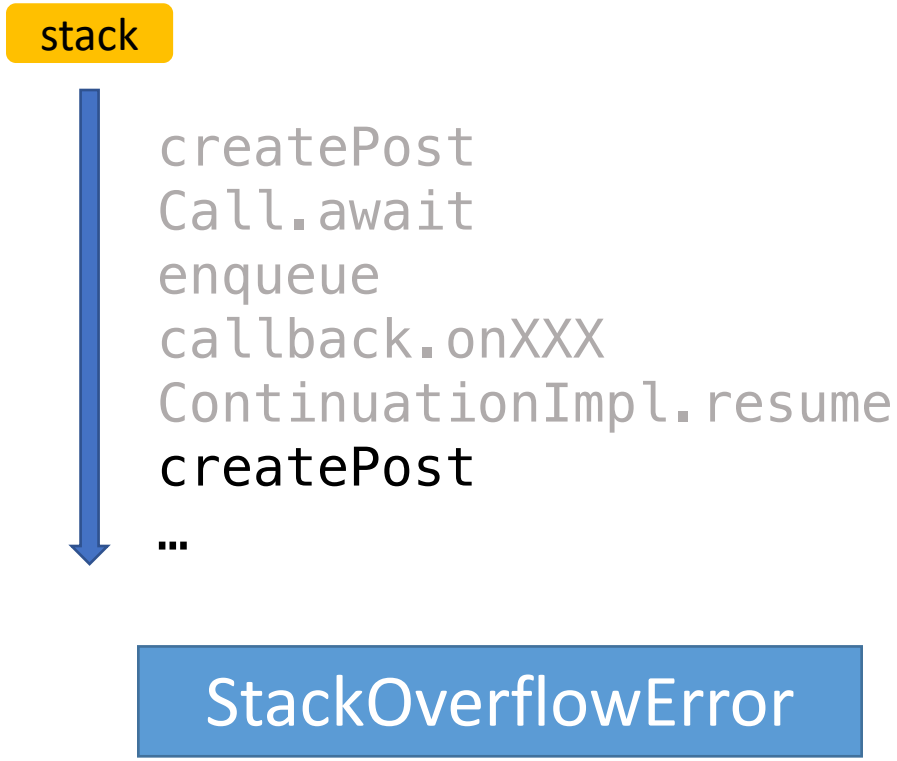
Dispatches execution to another thread

The stack

```
void enqueue(Callback<T> callback) {
  …
  if (failure != null) {
    callback.onFailure(this, failure);
    return;
  }
  …
}
```

Synchronous callback


stack
createPost


stack
createPost
Call.await
enqueue
callback.onXXX
ContinuationImpl.resume
createPost
…
StackOverflowError

```kotlin
suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T =
    suspendCoroutineOrReturn { c: Continuation<T> ->
        val safe = SafeContinuation(c)
        block(safe)
        safe.getResult()
    }
```

```kotlin
package kotlin.coroutines.experimental.intrinsics

suspend fun <T> suspendCoroutineOrReturn(
    block: (Continuation<T>) -> Any?): T
```

```kotlin
package kotlin.coroutines.experimental.intrinsics

suspend fun <T> suspendCoroutineOrReturn(
    block: (Continuation<T>) -> Any?): T
```

CPS Transformation revisited

```
suspend fun createPost(token: Token, item: Item): Post { … }
```

Java/JVM

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

Post | COROUTINE_SUSPENDED

CPS Transformation revisited

```
suspend fun createPost(token: Token, item: Item): Post { … }
```

Java/JVM

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

Return COROUTINE_SUSPENDED and invoke continuation *later*
Return result and don’t invoke continuation

```kotlin
package kotlin.coroutines.experimental.intrinsics

suspend fun <T> suspendCoroutineOrReturn(
    block: (Continuation<T>) -> Any?): T
```

T | COROUTINE_SUSPENDED

```
suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T =
    suspendCoroutineOrReturn { c: Continuation<T> ->
        val safe = SafeContinuation(c)
        block(safe)
        safe.getResult()
    }
```

T | COROUTINE_SUSPENDED

Starting coroutines

Writing your own coroutine builders

Coroutine builder

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

A regular function

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

```kotlin
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

```kotlin
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

suspending lambda

```kotlin
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(…)
    return future
}
```

```kotlin
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(…)
    return future
}
```

```kotlin
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(completion = object : Continuation<T> {
        …
    })
    return future
}
```

```kotlin
fun <T> future(…): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(completion = object : Continuation<T> {
        override val context: CoroutineContext get() = context

        override fun resume(value: T) {
            future.complete(value)
        }

        override fun resumeWithException(exception: Throwable) {
            future.completeExceptionally(exception)
        }
    })
    return future
}
```

```kotlin
fun <T> future(…): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(completion = object : Continuation<T> {
        override val context: CoroutineContext get() = context

        override fun resume(value: T) {
            future.complete(value)
        }

        override fun resumeWithException(exception: Throwable) {
            future.completeExceptionally(exception)
        }
    })
    return future
}
```

That’s all, folks!