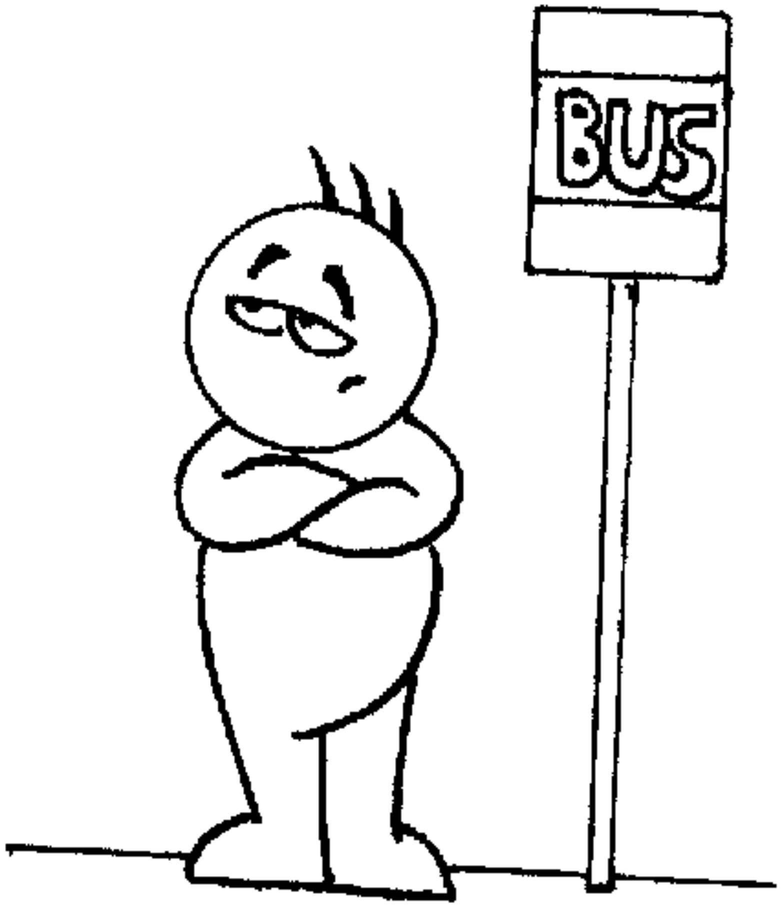
KotlinConf

Introduction to Coroutines

Roman Elizarov
elizarov at JetBrains

Asynchronous programming



How do we write code that waits for something most of the time?

A toy problem

Kotlin

1

```kotlin
fun requestToken(): Token {
    // makes request for a token & waits
    return token // returns result when received
}
```

A toy problem

Kotlin

```
fun requestToken(): Token { … }
2 fun createPost(token: Token, item: Item): Post {
      // sends item to the server & waits
      return post // returns resulting post
  }
```

A toy problem

Kotlin

```
fun requestToken(): Token { … }
fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) {
    // does some local processing of result
}
```

3

A toy problem

Kotlin

```
fun requestToken(): Token { … }
fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) { … }
```

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

Can be done with threads!

Threads

Is anything wrong with it?

```
fun requestToken(): Token {
    // makes request for a token
    // blocks the thread waiting for result
    return token // returns result when received
}
fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) { … }

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

How many threads we can have?

100 🙂

How many threads we can have?

1000 😅

How many threads we can have?

10 000 😩

How many threads we can have?

100 000 😵

Callbacks to the rescue

Sort of …

Callbacks: before

1

```
fun requestToken(): Token {
    // makes request for a token & waits
    return token // returns result when received
}
```

Callbacks: after

callback

1
```
fun requestTokenAsync(cb: (Token) -> Unit) {
    // makes request for a token, invokes callback when done
    // returns immediately
}
```

Callbacks: before

```
  fun requestTokenAsync(cb: (Token) -> Unit) { … }
2 fun createPost(token: Token, item: Item): Post {
      // sends item to the server & waits
      return post // returns resulting post
  }
```

Callbacks: after

```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
2 fun createPostAsync(token: Token, item: Item,
            callback cb: (Post) -> Unit) {
    // sends item to the server, invokes callback when done
    // returns immediately
}
```

Callbacks: before

```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
fun createPostAsync(token: Token, item: Item,
                    cb: (Post) -> Unit) { … }
fun processPost(post: Post) { … }
```

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

requestTokenAsync{
}

Callbacks: after

```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
fun createPostAsync(token: Token, item: Item,
                    cb: (Post) -> Unit) { … }
fun processPost(post: Post) { … }
```

This is simplified. Handling exceptions makes it a real mess

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

aka “callback hell”

Futures/Promises/Rx to the rescue

Sort of …

Futures: before

1

```
fun requestTokenAsync(cb: (Token) -> Unit) {
    // makes request for a token, invokes callback when done
    // returns immediately
}
```

Futures: after

future

1
```
fun requestTokenAsync(): Promise<Token> {
    // makes request for a token
    // returns promise for a future result immediately
}
```

Futures: before

```
  fun requestTokenAsync(): Promise<Token> { … }
2 fun createPostAsync(token: Token, item: Item,
                      cb: (Post) -> Unit) {
      // sends item to the server, invokes callback when done
      // returns immediately
  }
```

Futures: after

future

2
```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> {
    // sends item to the server
    // returns promise for a future result immediately
}
```

Futures: before

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

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

Futures: after

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

Composable &
propagates exceptions

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

No nesting indentation

Futures: after

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

But all those combinators…

Kotlin coroutines to the rescue

Let’s get real

Coroutines: before

1

```
fun requestTokenAsync(): Promise<Token> {
    // makes request for a token
    // returns promise for a future result immediately
}
```

Coroutines: after

1

```
suspend fun requestToken(): Token {
    // makes request for a token & suspends
    return token // returns result when received
}
```

Coroutines: after

natural signature

1

```
suspend fun requestToken(): Token {
    // makes request for a token & suspends
    return token // returns result when received
}
```

Coroutines: before

```
  suspend fun requestToken(): Token { … }
2 fun createPostAsync(token: Token, item: Item): Promise<Post> {
      // sends item to the server
      // returns promise for a future result immediately
  }
```

Coroutines: after

Continuation

2

```
suspend fun requestToken(): Token { … }
suspend fun createPost(token: Token, item: Item): Post {
    // sends item to the server & suspends
    return post // returns result when received
}
```

Coroutines: before

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

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

Coroutines: after

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

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

Coroutines: after

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

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

Like *regular* code

Coroutines: after

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

suspension points

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

Bonus features

- *Regular* loops

```
for ((token, item) in list) {
    createPost(token, item)
}
```

Bonus features

- *Regular* exception handing

```
try {
    createPost(token, item)
} catch (e: BadTokenException) {
    …
}
```

Bonus features

- *Regular* higher-order functions

```
file.readLines().forEach { line ->
    createPost(token, line.toItem())
}
```

- forEach, let, apply, repeat, filter, map, use, etc

Everything like in blocking code

KotlinConf

Suspending functions

Retrofit async

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}
```

Retrofit async

future

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}
```

Retrofit async

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}

suspend fun createPost(token: Token, item: Item): Post =
    serviceInstance.createPost(token, item).await()
```

Retrofit async

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}
```

natural signature

```
suspend fun createPost(token: Token, item: Item): Post =
    serviceInstance.createPost(token, item).await()
```

Retrofit async

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}

suspend fun createPost(token: Token, item: Item): Post =
    serviceInstance.createPost(token, item).await()
```

Suspending extension function from integration library

Composition

Beyond sequential

```kotlin
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) }
```

suspending lambda

```
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

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)
    }
}
```

Coroutine builders

Coroutines revisited

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

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

Coroutines revisited

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

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

Coroutines revisited

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

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

Error: Suspend function 'requestToken' should be called only from a coroutine or another suspend function

Coroutines revisited

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

Can *suspend* execution

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

Coroutines revisited

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

A regular function *cannot*

Can *suspend* execution

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

Coroutines revisited

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

A regular function *cannot*

Can *suspend* execution

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

One cannot simply invoke
a suspending function

Launch

coroutine builder

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

Returns immediately, coroutine works in **background thread pool**

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

一般用于后台

Fire and forget!

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

UI Context

Just specify the context

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

UI Context

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

And it gets executed on UI thread

Where’s the magic of launch?

A regular function

```
fun launch(
    context: CoroutineContext = ~~DefaultDispatcher~~ EmptyCoroutineContext,
    block: suspend () -> Unit
): Job { … }
```

```
fun launch(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> Unit
): Job { … }
```

suspending lambda

```kotlin
fun launch(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> Unit
): Job { … }
```

async / await

Kotlin-way

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

Kotlin

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

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

C# approach to the same problem
(also Python, TS, Dart, coming to JS)

C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

mark with async

C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

use await to suspend

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

returns a future

C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

Why no **await** keyword in Kotlin?

The problem with async

C# `requestToken()` **VALID** –> produces `Task<Token>`

concurrent behavior — default

C# `await requestToken()` **VALID** –> produces `Token`

sequential behavior

Kotlin **suspending functions**
are designed to imitate 仿效；仿制；模仿
sequential behavior
by default

Concurrency is hard
Concurrency has to be explicit

Kotlin approach to async

Concurrency where you need it

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }
```

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }
```

```
var promise1 = loadImageAsync(name1);
var promise2 = loadImageAsync(name2);
```

Start multiple operations **concurrently**

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }

var promise1 = loadImageAsync(name1);
var promise2 = loadImageAsync(name2);

var image1 = await promise1;
var image2 = await promise2;
```

and then wait for them

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }

var promise1 = loadImageAsync(name1);
var promise2 = loadImageAsync(name2);

var image1 = await promise1;
var image2 = await promise2;

var result = combineImages(image1, image2);
```

Kotlin async function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    async { … }
```

Kotlin async function

A regular function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    async { … }
```

Kotlin async function

Kotlin's future type

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    async { … }
```

Kotlin async function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    async { … }
```

async coroutine builder

Kotlin async function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    async { … }
```

```
val deferred1 = loadImageAsync(name1)
val deferred2 = loadImageAsync(name2)
```

Start multiple operations **concurrently**

Kotlin async function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    async { … }

val deferred1 = loadImageAsync(name1)
val deferred2 = loadImageAsync(name2)

val image1 = deferred1.await()
val image2 = deferred2.await()
```

await *function*

and then wait for them

Suspends until deferred is complete

Kotlin async function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    async { … }

val deferred1 = loadImageAsync(name1)
val deferred2 = loadImageAsync(name2)

val image1 = deferred1.await()
val image2 = deferred2.await()

val result = combineImages(image1, image2)
```

Using async function when needed

Is defined as suspending function, not async

```
suspend fun loadImage(name: String): Image { … }
```

Using async function when needed

```
suspend fun loadImage(name: String): Image { … }
```

```
suspend fun loadAndCombine(name1: String, name2: String): Image {
    val deferred1 = async { loadImage(name1) }
    val deferred2 = async { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

Using async function when needed

```
suspend fun loadImage(name: String): Image { … }

suspend fun loadAndCombine(name1: String, name2: String): Image {
    val deferred1 = async { loadImage(name1) }
    val deferred2 = async { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

Using async function when needed

```kotlin
suspend fun loadImage(name: String): Image { … }
```

```kotlin
suspend fun loadAndCombine(name1: String, name2: String): Image {
    val deferred1 = async { loadImage(name1) }
    val deferred2 = async { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

Using async function when needed

```
suspend fun loadImage(name: String): Image { … }


suspend fun loadAndCombine(name1: String, name2: String): Image {
    val deferred1 = async { loadImage(name1) }
    val deferred2 = async { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

Kotlin approach to async

Kotlin `requestToken()` **VALID** –> produces `Token`

sequential behavior

default

Kotlin `async { requestToken() }` **VALID** –> produces `Deferred<Token>`

concurrent behavior

KotlinConf

Coroutines

What are coroutines conceptually?

What are coroutines conceptually?

Coroutines are like ***very*** light-weight threads

Example

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Example

This coroutine builder runs coroutine in the context of invoker thread

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Example

```kotlin
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Example

```kotlin
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Example

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Suspends for 1 second

Example

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

We can join a job just like a thread

Demo

Example

```kotlin
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Prints 100k dots after one second delay

Try that with 100k threads!

Example

```kotlin
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

提倡这样做不会阻塞线程的执行

Example

```kotlin
fun main(args: Array<String>) {
    val jobs = List(100_000) {
        thread {
            Thread.sleep(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

千万不要这样做会阻塞线程的执行

Demo

Example

Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread

```
fun main(args: Array<String>) {
    val jobs = List(100_000) {
        thread {
            Thread.sleep(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Java interop

Java

```
CompletableFuture<Image> loadImageAsync(String name) { … }
```

Java

```
CompletableFuture<Image> loadImageAsync(String name) { … }

CompletableFuture<Image> loadAndCombineAsync(String name1,
                                             String name2)
```

Imagine implementing it in Java…

Java

```
CompletableFuture<Image> loadImageAsync(String name) { … }

CompletableFuture<Image> loadAndCombineAsync(String name1,
                                             String name2)
{
    CompletableFuture<Image> future1 = loadImageAsync(name1);
    CompletableFuture<Image> future2 = loadImageAsync(name2);
    return future1.thenCompose(image1 ->
        future2.thenCompose(image2 ->
            CompletableFuture.supplyAsync(() ->
                combineImages(image1, image2))));
}
```

Java

```java
CompletableFuture<Image> loadImageAsync(String name) { … }

CompletableFuture<Image> loadAndCombineAsync(String name1,
                                             String name2)
{
    CompletableFuture<Image> future1 = loadImageAsync(name1);
    CompletableFuture<Image> future2 = loadImageAsync(name2);
    return future1.thenCompose(image1 ->
        future2.thenCompose(image2 ->
            CompletableFuture.supplyAsync(() ->
                combineImages(image1, image2))));
}
```

Java

```java
CompletableFuture<Image> loadImageAsync(String name) { … }
```

Kotlin

```kotlin
fun loadAndCombineAsync(
    name1: String,
    name2: String
): CompletableFuture<Image> =
    future {
        val future1 = loadImageAsync(name1)
        val future2 = loadImageAsync(name2)
        combineImages(future1.await(), future2.await())
    }
```

Java

```java
CompletableFuture<Image> loadImageAsync(String name) { … }
```

Kotlin

```kotlin
fun loadAndCombineAsync(
    name1: String,
    name2: String
): CompletableFuture<Image> =
    future {
        val future1 = loadImageAsync(name1)
        val future2 = loadImageAsync(name2)
        combineImages(future1.await(), future2.await())
    }
```

Java

```java
CompletableFuture<Image> loadImageAsync(String name) { … }
```

Kotlin

```kotlin
fun loadAndCombineAsync(
    name1: String,
    name2: String
): CompletableFuture<Image> =
    future {
        val future1 = loadImageAsync(name1)
        val future2 = loadImageAsync(name2)
        combineImages(future1.await(), future2.await())
    }
```

future coroutine builder

Java

```java
CompletableFuture<Image> loadImageAsync(String name) { … }
```

Kotlin

```kotlin
fun loadAndCombineAsync(
    name1: String,
    name2: String
): CompletableFuture<Image> =
    future {
        val future1 = loadImageAsync(name1)
        val future2 = loadImageAsync(name2)
        combineImages(future1.await(), future2.await())
    }
```

Java

```
CompletableFuture<Image> loadImageAsync(String name) { … }
```

Kotlin

```
fun loadAndCombineAsync(
    name1: String,
    name2: String
): CompletableFuture<Image> =
    future {
        val future1 = loadImageAsync(name1)
        val future2 = loadImageAsync(name2)
        combineImages(future1.await(), future2.await())
    }
```

Extension for Java's CompletableFuture

KotlinConf

Beyond asynchronous code

Fibonacci sequence

```
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

sequence{}

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

A coroutine builder with restricted suspension

```
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

A suspending function

The same building blocks

```
fun <T> buildSequence(
    builderAction: suspend SequenceBuilder<T>.() -> Unit
): Sequence<T> { … }
```

```kotlin
fun <T> buildSequence(
    builderAction: suspend SequenceBuilder<T>.() -> Unit
): Sequence<T> { … }
```

Result is a *synchronous* sequence

```kotlin
fun <T> buildSequence(
    builderAction: suspend SequenceBuilder<T>.() -> Unit
): Sequence<T> { … }
```

Suspending lambda with receiver

```kotlin
fun <T> buildSequence(
    builderAction: suspend SequenceBuilder<T>.() -> Unit
): Sequence<T> { … }


@RestrictsSuspension
abstract class SequenceBuilder<in T> {
    abstract suspend fun yield(value: T)
}
```

Coroutine is restricted only to suspending functions defined here

Synchronous

```
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
```

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next())
```

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next())
```

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
```

```
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next())
```

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next())
```

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next()) // 1
```

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next()) // 1
println(iter.next()) // 2
```

```kotlin
val fibonacci = buildSequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next()) // 1
println(iter.next()) // 2
```

Synchronous with invoker

Library vs Language

Classic async

`async/await`
`generate/yield`

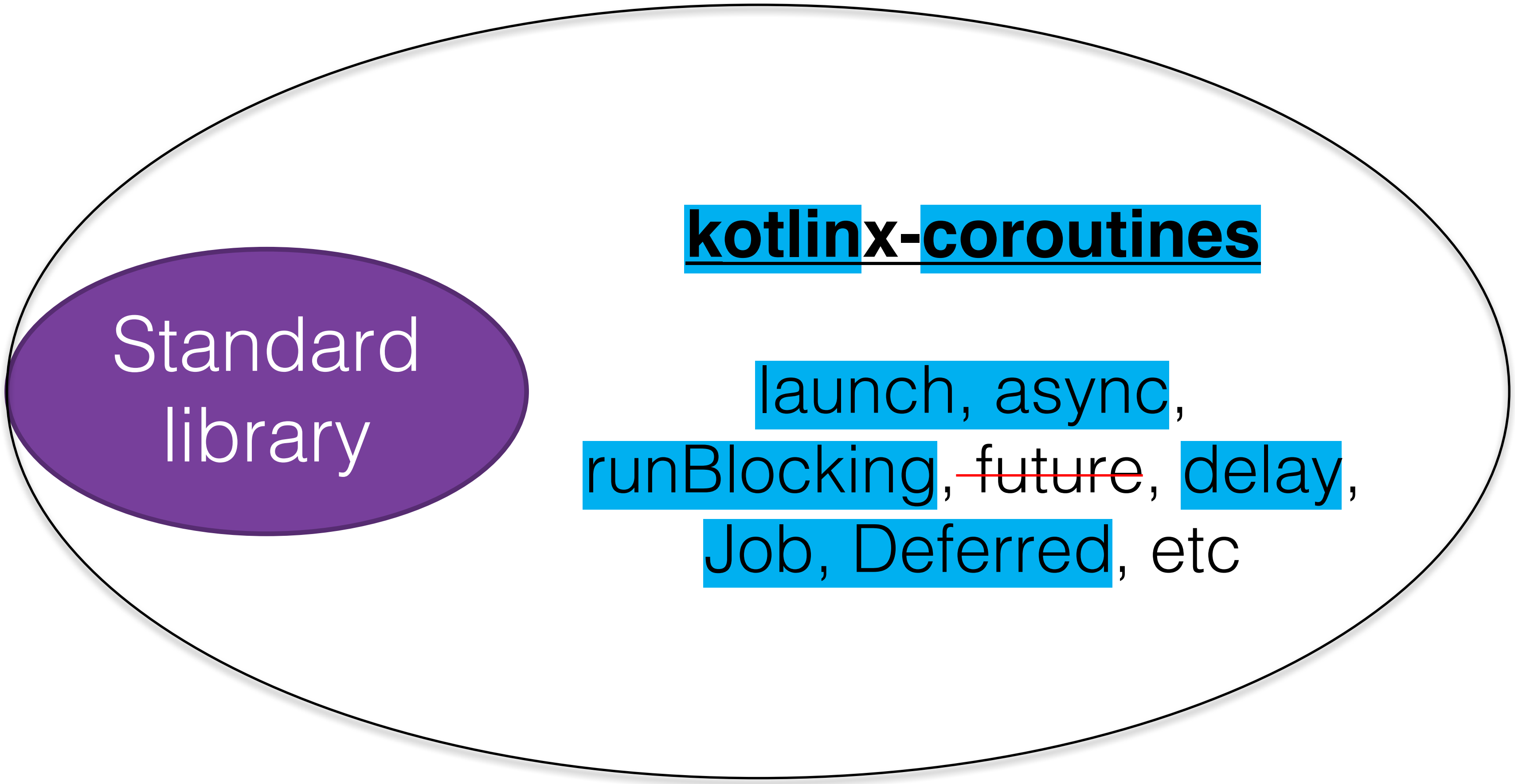
Keywords

Kotlin coroutines

`suspend` — Modifier

Kotlin coroutines

Kotlin coroutines



http://github.com/kotlin/kotlinx.coroutines

Experimental status

Coroutines are here to stay

Backwards compatible inside 1.1 & 1.2

To be finalized in the future

Thank you

Roman Elizarov
elizarov at JetBrains
relizarov

Any questions?