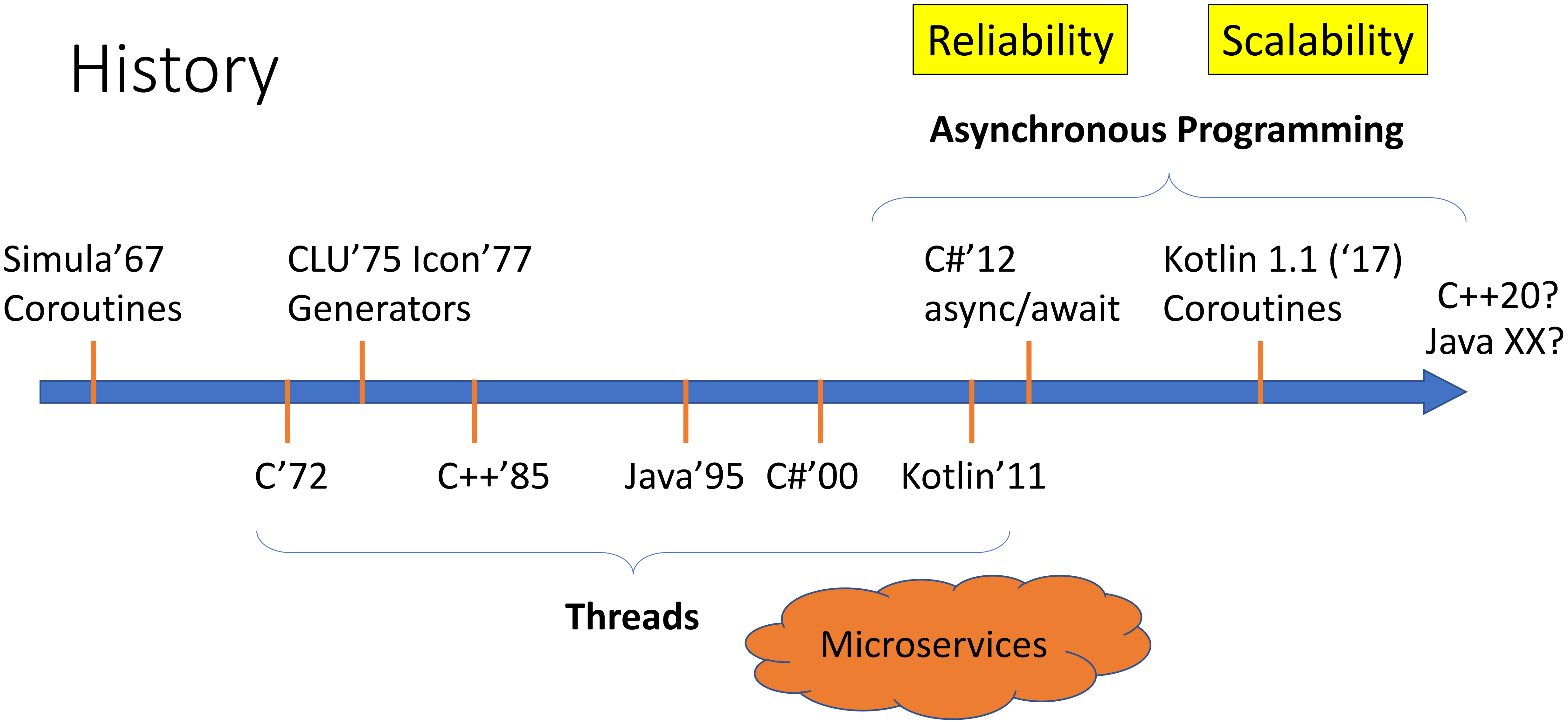
Kotlin Coroutines

History



Reliability
Scalability
Asynchronous Programming
Simula'67
Coroutines
CLU'75 Icon'77
Generators
C#'12
async/await
Kotlin 1.1 ('17)
Coroutines
C++20?
Java XX?
C'72
C++'85
Java'95
C#'00
Kotlin'11
Threads
Microservices

Asynchronous Programming with **Callbacks**

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

2
```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
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)
}
```

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

```
    fun requestTokenAsync(): Promise<Token> { … }
2   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) { … }
```

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

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) }
}
```

Callbacks are still here

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

Callbacks are still here

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 operators…

Kotlin coroutines to the rescue

For real

Coroutines: before

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

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

Coroutines: after

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

2

natural signature

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

```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)
}
```

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())
}
```

Bonus features

- *Custom* higher-order functions

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

Everything like in blocking code

How does it work?

A quick peek behind the scenes

Kotlin suspending functions

Kotlin

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

Java/JVM

callback

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

Kotlin suspending functions

Kotlin

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

Java/JVM

callback

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

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

Continuation is a generic callback interface

Kotlin suspending functions

Kotlin

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

Java/JVM

callback

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

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

Kotlin suspending functions

Kotlin

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

Java/JVM

callback

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

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

Kotlin suspending functions

Kotlin

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

Java/JVM

callback

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

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

Code with suspension points

Kotlin

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

Java/JVM

```
switch (cont.label) {
    case 0:
        cont.label = 1;
        requestToken(cont);
        break;
    case 1:
        Token token = (Token) prevResult;
        cont.label = 2;
        createPost(token, item, cont);
        break;
    case 2:
        Post post = (Post) prevResult;
        processPost(post);
        break;
}
```

Compiles to *state machine* (simplified code shown)

Code with suspension points

Kotlin

```kotlin
val token = requestToken()
val post = createPost(token, item)
processPost(post)
```

Java/JVM

```java
switch (cont.label) {
    case 0:
        cont.label = 1;
        requestToken(cont);
        break;
    case 1:
        Token token = (Token) prevResult;
        cont.label = 2;
        createPost(token, item, cont);
        break;
    case 2:
        Post post = (Post) prevResult;
        processPost(post);
        break;
}
```

Reuses the same callback (state machine) instance

Integration

Zoo of futures on JVM

Retrofit async

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

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

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

natural signature

```kotlin
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

```
suspend fun <T> Call<T>.await(): T {
    …
}
```

Callbacks everywhere

```
suspend fun <T> Call<T>.await(): T {
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            // todo
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            // todo
        }
    })
}
```

```kotlin
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

```kotlin
suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T
```

```kotlin
suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T
```

```
suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T
```

Regular function

Inspired by **call/cc** from **Scheme**

```kotlin
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Install callback

```
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Install callback

```
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Analyze response

```
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

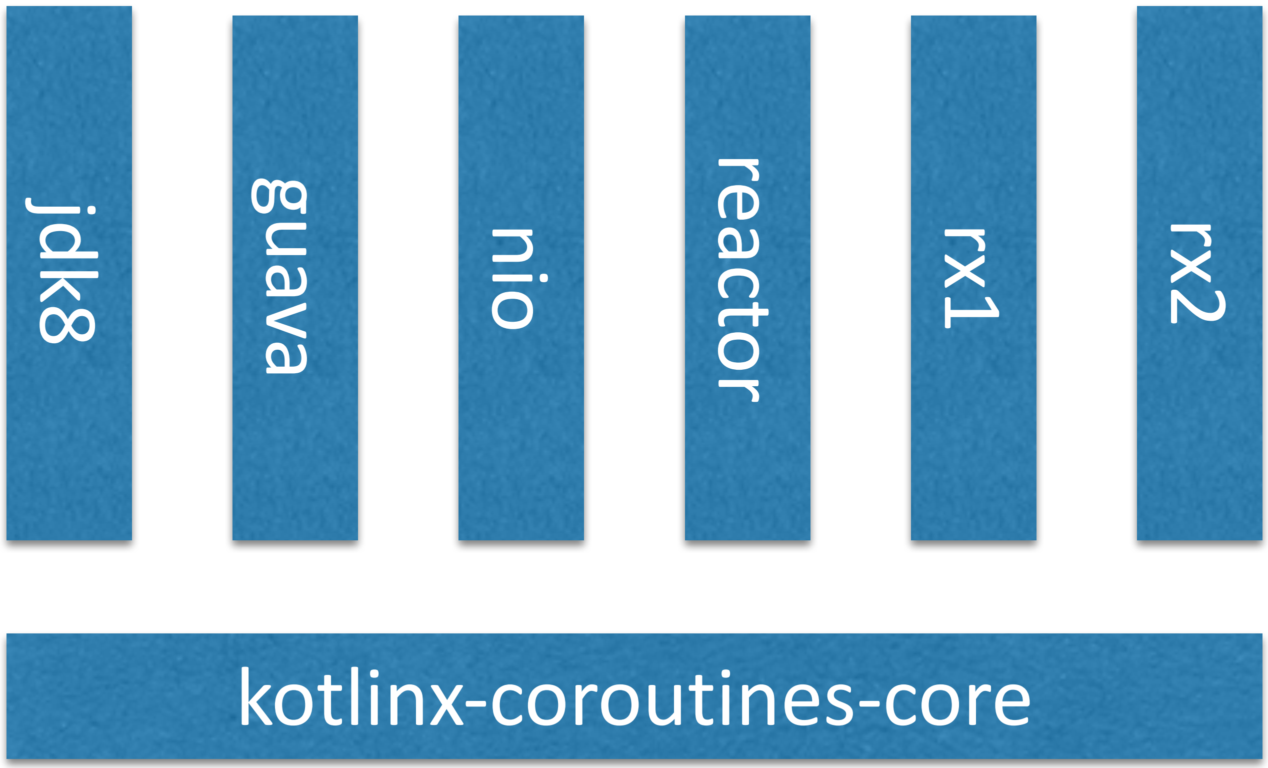
Analyze response

```
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

That's all

Out-of-the box integrations



Coroutine builders

How can we start a coroutine?

Coroutines revisited

```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 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**

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

Fire and forget!

```
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,
    block: suspend () -> Unit
): Job { … }
```

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

suspending lambda

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

async / await

The classic approach

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 in Python, TS, Dart, 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?

<u>The problem with async</u>

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

concurrent behavior — default

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

sequential behavior

Concurrency is hard
Concurrency has to be explicit

Kotlin **suspending functions** are
designed to imitate <u>sequential</u> 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

```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()
```

and then wait for them

await *function*

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)
```

Idiomatic: async when needed

Is defined as suspending function, not async

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

Idiomatic: async 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())
}
```

Idiomatic: async 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())
}
```

Idiomatic: async 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())
}
```

Idiomatic: async 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	

What are coroutines conceptually?

What are coroutines conceptually?

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

Example

```kotlin
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

Example

```
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

```
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>) {
    val jobs = List(100_000) {
        thread {
            Thread.sleep(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Example

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

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

Java interop

Can we use Kotlin coroutines with Java code?

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

```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

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

Kotlin

```
fun loadAndCombineAsync(
    name1: String,
    name2: String
): CompletableFuture<Image> =
    …
```

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

```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())
    }
```

Beyond asynchronous code

Kotlin’s approach to generate/yield – synchronous coroutines

Fibonacci sequence

```
val fibonacci: Sequence<Int> = …
```

Fibonacci sequence

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

Fibonacci sequence

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

println(fibonacci.take(10).toList())
```

Fibonacci sequence

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

println(fibonacci.take(10).toList())

>> [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Fibonacci sequence

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
    }
}
```

Fibonacci sequence

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

A suspending function in the scope of **buildSequence**

Fibonacci sequence

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

println(fibonacci.take(10).toList())
```

Synchronous

Fibonacci sequence

```
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()
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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())
```

Fibonacci sequence

```
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
```

Fibonacci sequence

```
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
```

etc ad infinum

Communicating Sequential Processes (CSP)

Shared Mutable State

Concurrent Fibonacci sequence

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

Asynchronous
(like a light-weight thread)

Concurrent Fibonacci sequence

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

fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive())
}
```

Asynchronous
(can be in different thread)

Another coroutine

Concurrent Fibonacci sequence

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

fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive())
}
```

Asynchronous
(can be in different thread)

Another coroutine

Receives from **channel**

Concurrent Fibonacci sequence

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

fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive()) // 1
}
```

Concurrent Fibonacci sequence

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

fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive()) // 1
    println(fibonacci.receive()) // 1
}
```

etc ad infinum

Library vs Language

Keeping the core language small

Classic async

$$\left.\begin{array}{l}\texttt{async/await}\\\texttt{generate/yield}\end{array}\right\}\text{Keywords}$$

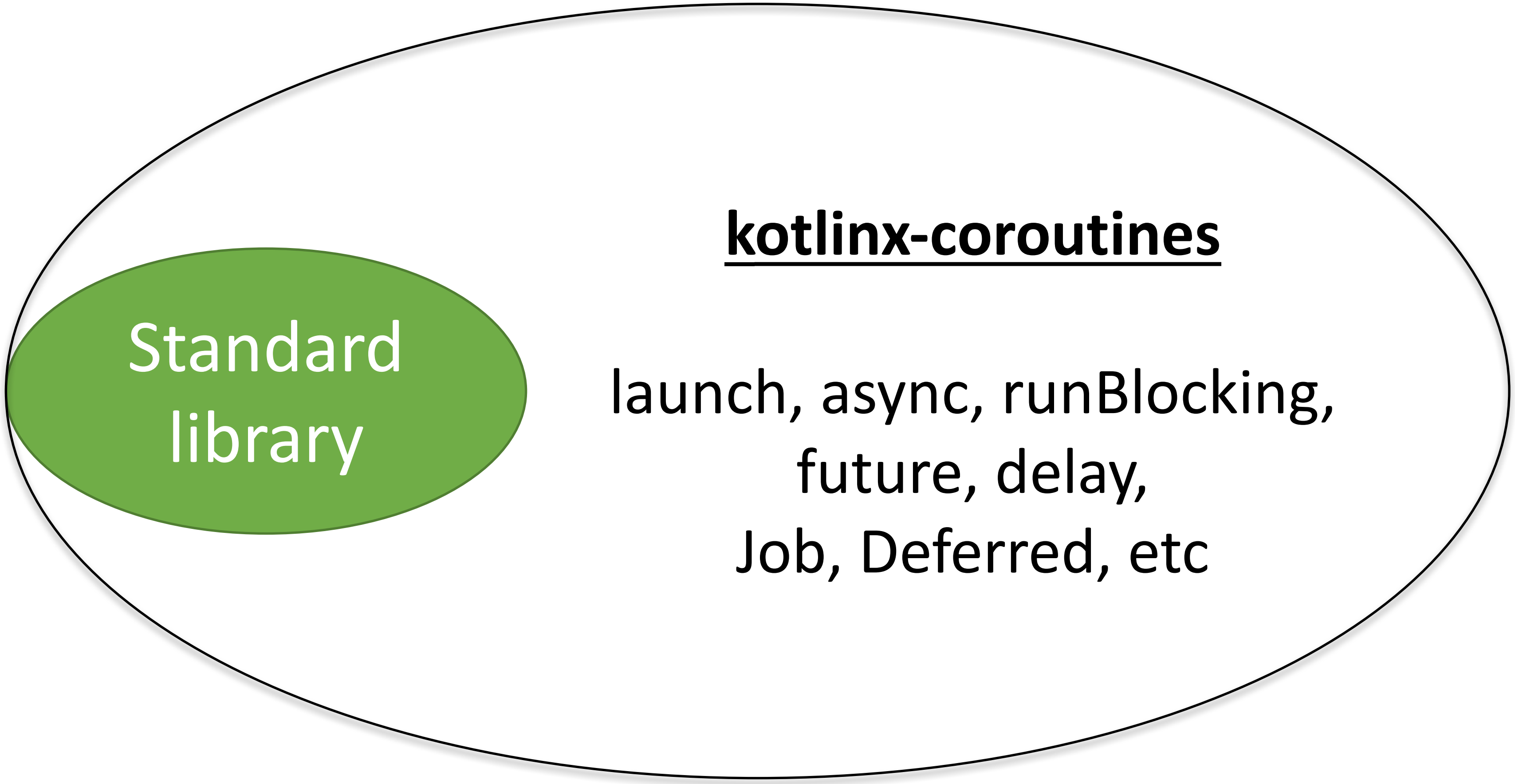
Kotlin coroutines

`suspend`

Modifier

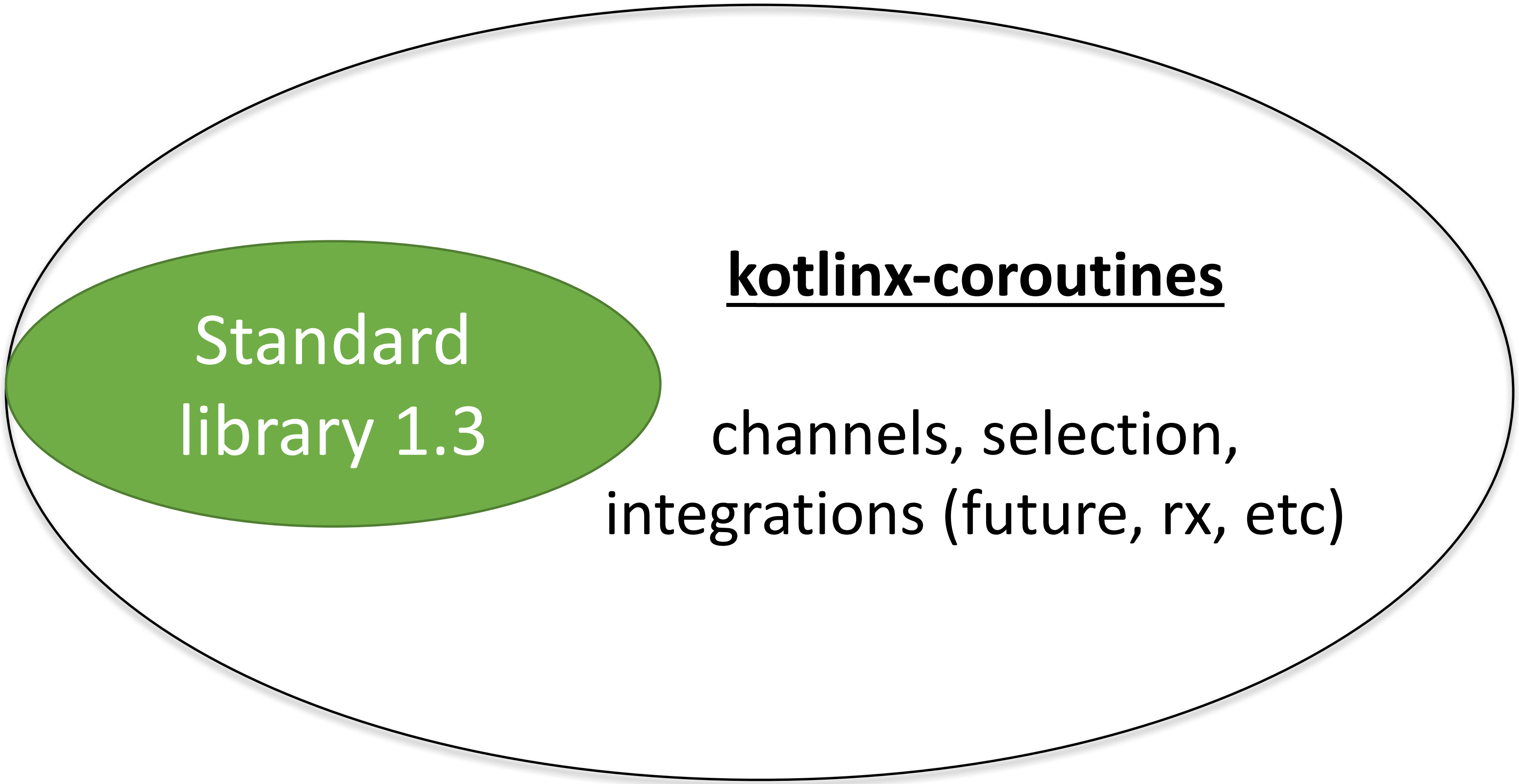
Kotlin coroutines

Kotlin coroutines



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

Kotlin coroutines Roadmap



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

There is more

- Communicating Sequential Processes (CSP) Style
  - Channels and Actors
  - Selection and synchronization
  - Job hierarchies and cancellation

- Learn more in Guide to kotlinx.coroutines by example

- Visit **KotlinConf 2018** (3-5 Oct) in Amsterdam