# Asynchronous Computing in Kotlin

## About me

**Trayan Iliev**

– CEO of **IPT – Intellectual Products & Technologies** [http://www.iproduct.org](http://www.iproduct.org)

– Oracle® certified programmer 15+ Y

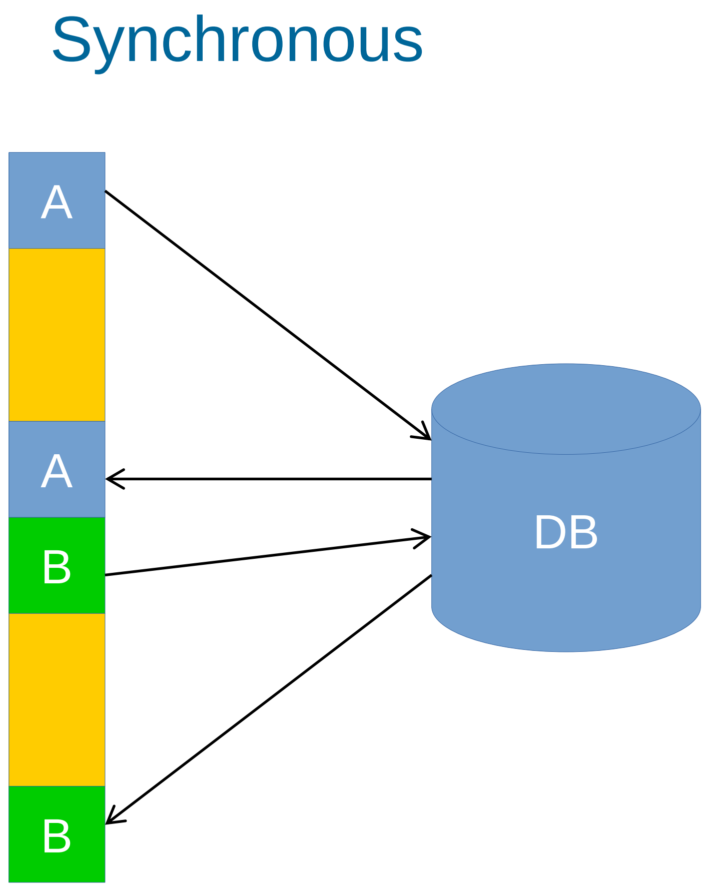
– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

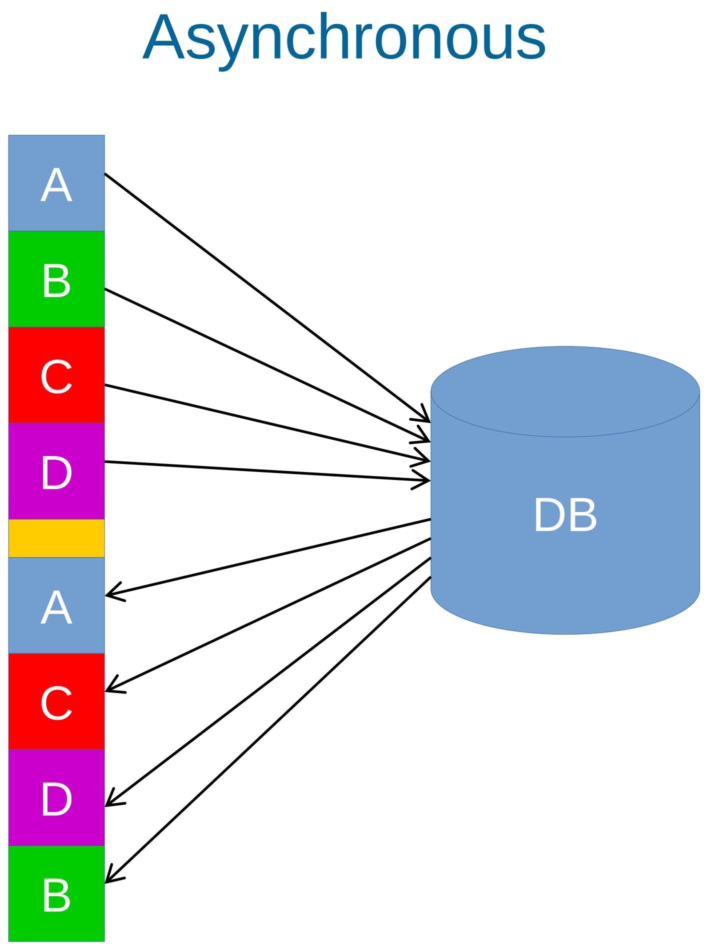
– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer  RoboLearn hackathons and  IoT enthusiast

# Synchronous vs. Asynchronous IO





## Blocking vs. Non-blocking

- Blocking concurrency – uses **Mut**ual **Ex**clusion primitives (aka **Locks**) to prevent threads from simultaneously accessing/modifying the same resource
- Non-blocking concurrency does not make use of locks.
- One of the most advantageous feature of non-blocking vs. blocking is that, threads does not have to be suspended/waken up by the OS. Such overhead can amount to 1ms to a few 10ms, so removing this can be a big performance gain. In java for example, it also means that you can choose to use non-fair locking, which can have much more system throughput than fair-locking.

## Non-blocking Concurrency

- In computer science, an algorithm is called **non-blocking** if failure or suspension of any thread cannot cause failure or suspension of another thread;[1] for some operations, these algorithms provide a useful alternative to traditional blocking implementations. A non-blocking algorithm is **lock-free** if there is guaranteed system-wide progress, and **wait-free** if there is also guaranteed per-thread progress. "Non-blocking" was used as a synonym for "lock-free" in the literature until the introduction of obstruction-freedom in 2003.
- It has been shown that widely available atomic *conditional* primitives, CAS and LL/SC, cannot provide starvation-free implementations of many common data structures without memory costs growing linearly in the number of threads.

[Wikipedia]

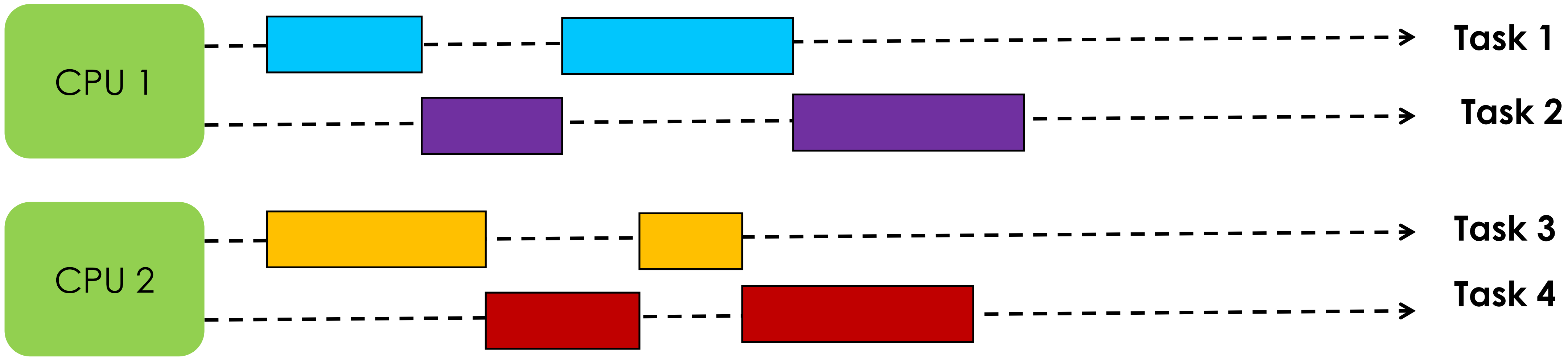
## Concurrency vs. Parallelism

Question:

***What is the difference between concurrency and parallelism?***

## Concurrency vs. Parallelism

- Concurrency refers to how a single CPU can make progress on multiple tasks seemingly at the same time (AKA concurrently).
- Parallelism allows an application to parallelize the execution of a single task - typically by splitting the task up into subtasks which can be completed in parallel.



## Asynchronous programming techniques

- For decades, as developers we are confronted with a problem to solve - how to prevent our applications from blocking. Whether we're developing desktop, mobile, or even server-side applications, we want to avoid having the user wait or what's worse cause bottlenecks that would prevent an application from scaling.
- There have been many approaches to solving this problem, including:
  - Threading
  - Callbacks
  - Futures, promises, and others
  - Reactive Extensions
  - Coroutines

## Example problem

```
class Item
class Token
class Post
fun submitPost(token: Token, item: Item): Post {
    return Post()
}
fun processPost(post: Post) {}
fun preparePost(): Token {
    // makes a request and consequently blocks the main thread
    val token = Token()
    return token
}

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

## Threads Drawbacks [https://kotlinlang.org/docs/async-programming.html]

- Threads aren't cheap. Threads require context switches which are costly.
- Threads aren't infinite. The number of threads that can be launched is limited by the underlying operating system. In server-side applications, this could cause a major bottleneck.
- Threads aren't always available. Some platforms, such as JavaScript do not even support threads.
- Threads aren't easy. Debugging threads, avoiding race conditions are common problems we suffer in multi-threaded programming.

## Callbacks Drawbacks

- Difficulty of nested callbacks. Usually a function that is used as a callback, often ends up needing its own callback. This leads to a series of nested callbacks which lead to incomprehensible code. The pattern is often referred to as the titled christmas tree (braces represent branches of the tree).
- Error handling is complicated. The nesting model makes error handling and propagation of these somewhat more complicated.
- Callbacks are quite common in event-loop architectures such as JavaScript, but even there, generally people have moved away to using other approaches such as promises or reactive extensions.

## Futures, promises, and others

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token: Token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post: Post ->
            processPost(post)
        }

}

fun preparePostAsync(): Promise<Token> {
    // makes request and returns a promise that is completed later
    val promise = Promise<Token>()
    return promise
}
```

## Futures, promises, and others

- Different programming model. Similar to callbacks, the programming model moves away from a top-down imperative approach to a compositional model with chained calls. Traditional program structures such as loops, exception handling, etc. usually are no longer valid in this model.
- Different APIs. Usually there's a need to learn a completely new API such as thenCompose or thenAccept, which can also vary across platforms.
- Specific return type. The return type moves away from the actual data that we need and instead returns a new type Promise which has to be introspected.
- Error handling can be complicated. The propagation and chaining of errors aren't always straightforward.

## Reactive Extensions - Rx

- Reactive Extensions (**Rx**) were introduced to C# by Erik Meijer. While it was definitely used on the .NET platform it really didn't reach mainstream adoption until Netflix ported it over to Java, naming it RxJava. From then on, numerous ports for many platforms including JavaScript (RxJS).
- The idea behind Rx is to move towards what's called observable streams whereby we now think of data as streams (infinite amounts of data) and these streams can be observed. In practical terms, Rx is simply the Observer Pattern with a series of extensions.
- **"everything is a stream, and it's observable"**
- One benefit as opposed to Futures is that given it's ported to so many platforms, generally we can find a consistent API experience no matter what we use, be it C#, Java, JavaScript, or other langusges Rx is available.
- In addition, Rx introduces nicer approach to error handling.

## Coroutines

```
fun postItem(item: Item) {
    runBlocking {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // makes a request and suspends the coroutine
    return suspendCoroutine { /* ... */ }
}
```

## Scalability Problem

- Scalability is the ability of a program to handle growing workloads.
- One way in which programs can scale is parallelism: if we want to process a large chunk of data, we describe its processing as a sequence of transforms on a stream, and by setting it to parallel we ask multiple processing cores to process the parts of the task simultaneously.
- The problem is that the processes and threads, the OS supported units of concurrency, cannot match the scale of the application domain's natural units of concurrency — a session, an HTTP request, or a database transactional operation.
- A server can handle upward of a million concurrent open sockets, yet the operating system cannot efficiently handle more than a few thousand active threads. So it becomes a mapping problem - M:N

## Why are OS Threads Heavy?

- Universal – represent all languages and types of workloads
- Can be suspended and resumed - this requires preserving its state, which includes the instruction pointer, as well as all of the local computation data, stored on the stack.
- The stack should be quite large, because we can not assume constraints in advance.
- Because the OS kernel must schedule all types of threads that behave very differently it terms of processing and blocking — some serving HTTP requests, others playing videos => its scheduler must be all-encompassing, and not optimized.

## Solutions To Thread Scalability Problem

- Because threads are costly to create, we pool them -> but we must pay the price: leaking thread-local data and a complex cancellation protocol.
- Thread pooling is coarse grained – not enough threads for all tasks.
- So instead of blocking the thread, the task should return the thread to the pool while it is waiting for some external event, such as a response from a database or a service, or any other activity that would block it.
- The task is no longer bound to a single thread for its entire execution.
- Proliferation of asynchronous APIs, from Noide.js to NIO in Java, to the many “reactive” libraries (Reactive Extensions - Rx, etc.) => intrusive, all-encompassing frameworks, even basic control flow, like loops and try/catch, need to be reconstructed in “reactive” DSLs, supporting classes with hundreds of methods.

## What Color is Your Function? [http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/]

- Synchronous functions return values, async ones do not and instead invoke callbacks.
- Synchronous functions give their result as a return value, async functions give it by invoking a callback you pass to it.
- You can’t call an async function from a synchronous one because you won’t be able to determine the result until the async one completes later.
- Async functions don’t compose in expressions because of the callbacks, have different error-handling.
- Node’s whole idea is that the core libs are all asynchronous. (Though they did dial that back and start adding ___Sync() versions of a lot functions.)

## Async/Await in JS (and some other languages)

- Cooperative scheduling points are marked explicitly with await => scalable synchronous code – but we mark it as async – a bit of confusing!
- Solves the context issue by introducing a new kind of context that is like thread but is incompatible with threads – one blocks and the other returns some sort of Promise - you can not easily mix sync and async code.

## Right-Sized Concurrency

- If we could make threads lighter, we could have more of them, and can use them as intended:
    1. to directly represent domain units of concurrency;
    2. by virtualizing scarce computational resources;
    3. hiding the complexity of managing those resources.
- Example: Kotlin coroutines, goroutines, Erlang
- creating and blocking goroutines is cheap
- Managed by the Kotlin runtime, and scheduled cooperatively unlike the existing threads of OS.

## How Are Coroutines Better?

- The coroutines make use of a stack, so it can represent execution state more compactly.
- Control over execution by optimized scheduler;
- Millions of coroutines => every unit of concurrency in the application domain can be represented by its own coroutine
- Just spawn a new coroutine, one per task.
- Example: Ktor HTTP request - a new coroutine is already spawned to handle it, but now, in the course of handling the request, you want to simultaneously query a database, and issue outgoing requests to three other services? No problem: spawn more coroutines.

## How Are Coroutines Better?

- You need to wait for something to happen without wasting precious resources – forget about callbacks or reactive stream chaining – just block!
- Write straightforward, boring code.
- Coroutines preserve all the benefits threads give us are preserved by : control flow, exception context; only the runtime cost in footprint and performance is gone!

## Callbacks

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post: Post ->
            processPost(post)
        }
    }
}

fun submitPostAsync(token: Token, item: Item, any: Any) {
}

fun preparePostAsync(callback: (Token) -> Unit) {
    // make request and return immediately
    // arrange callback to be invoked later
}
```

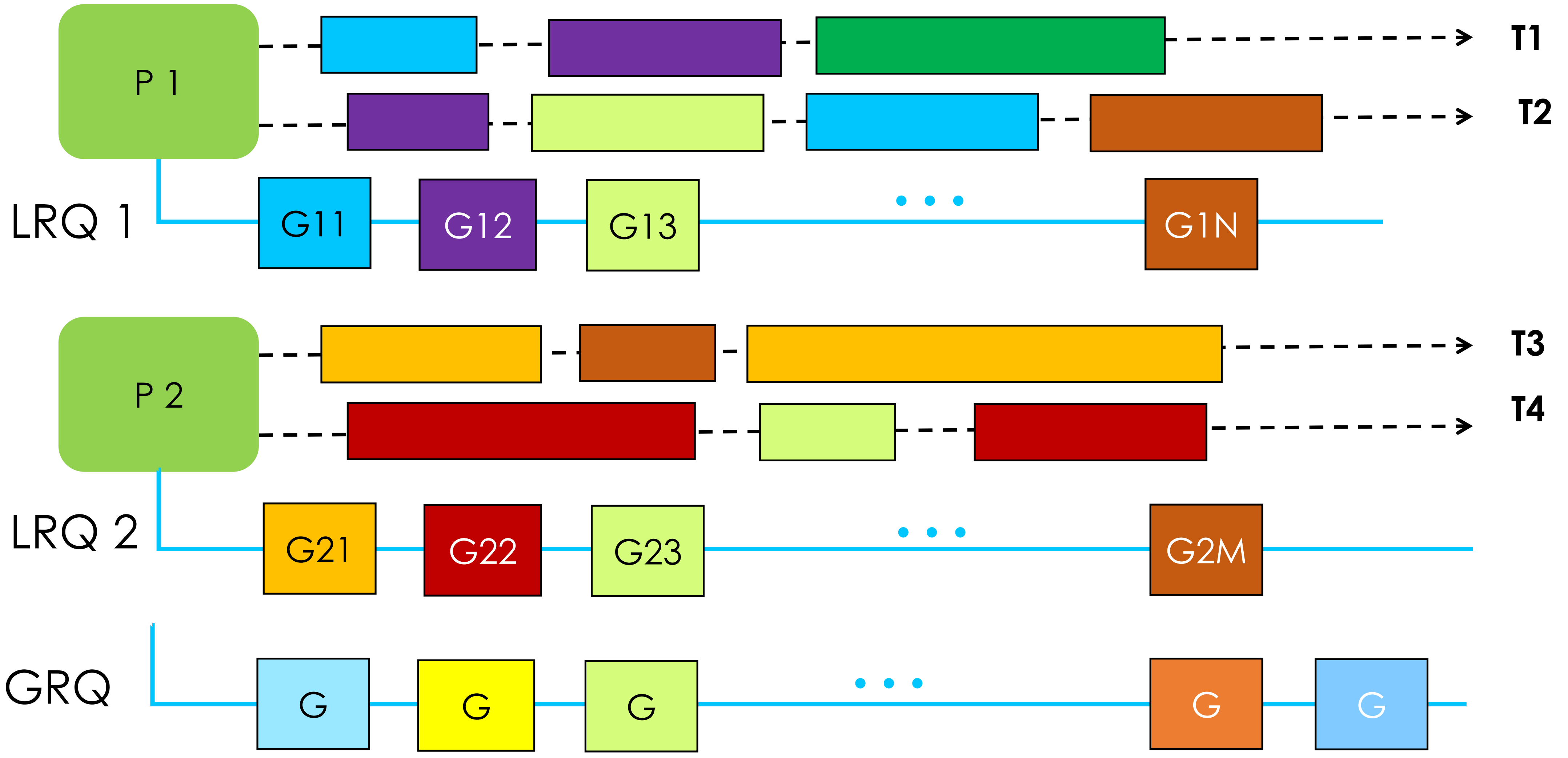
# Welcome Coroutines!

Goroutines and channels

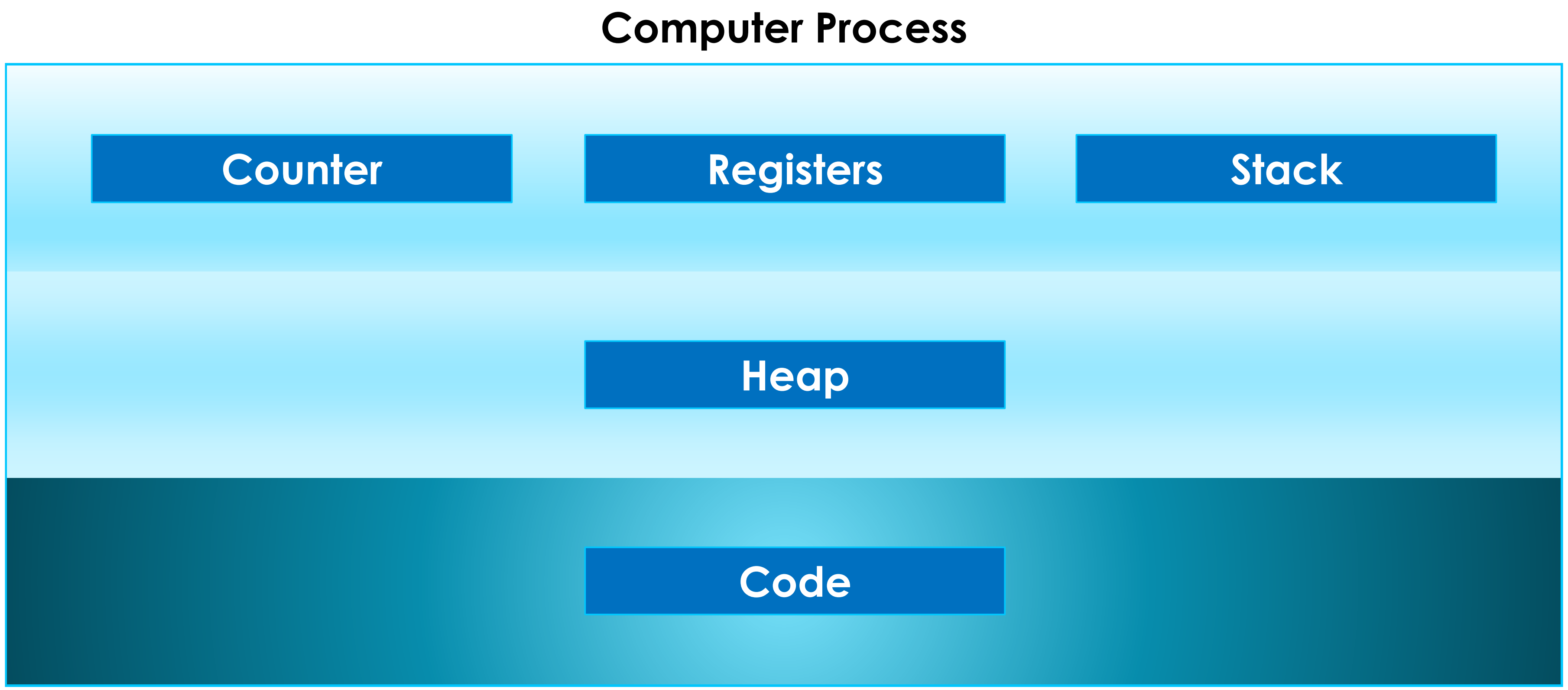
## Coroutines Scheduling

**GRQ – Global Run Queue** **LRQ – Local Run Queue**

P – Processor T – Thread G – Coroutine

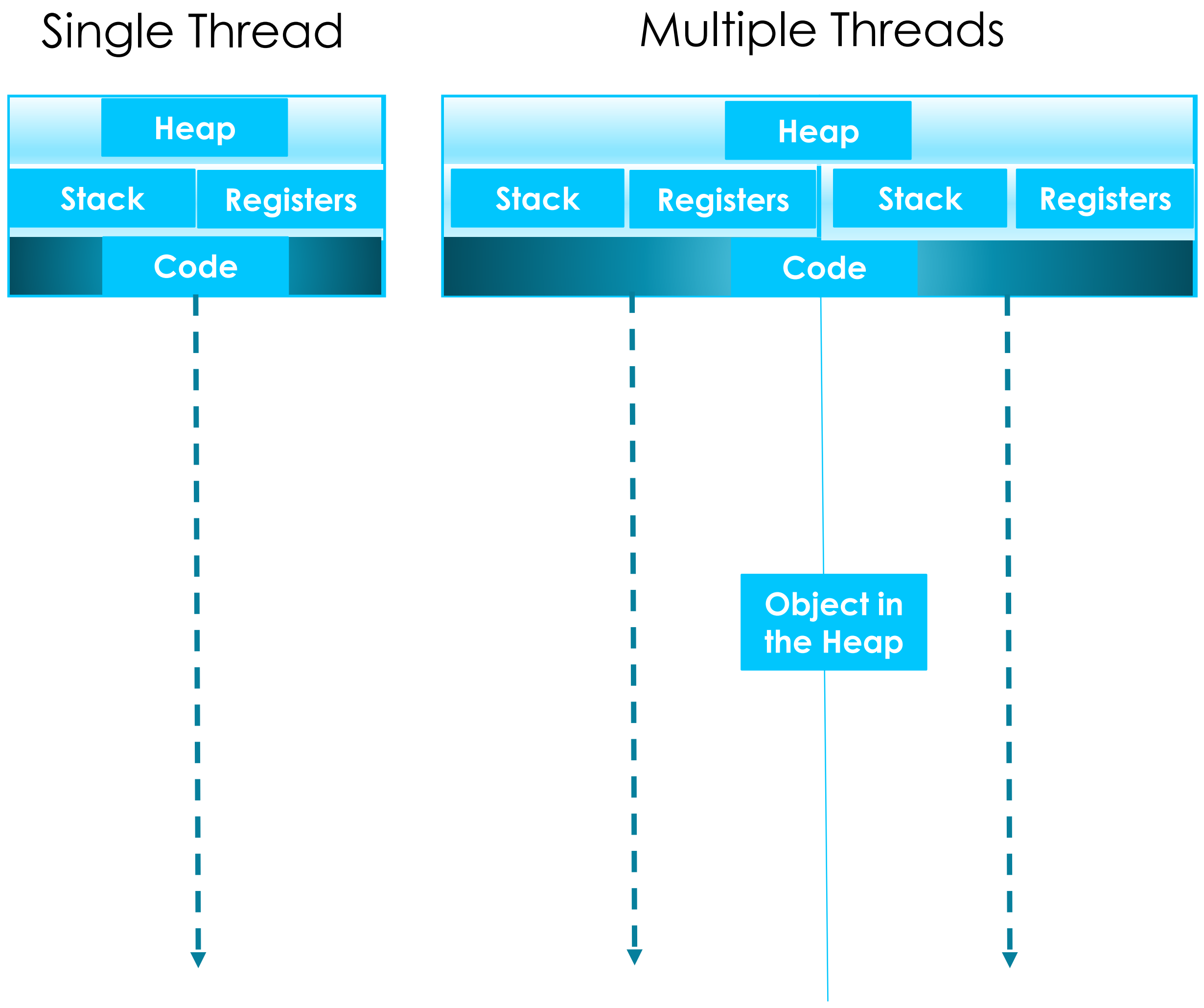


## Concurrency Approaches: Processes



# Threads

- There can be many threads in the same process
- The threads can access the shared memory
- This means that the global objects can be accessed by all threads
- Provided by the OS
- Cheaper to create than processes
- Some languages expose them directly other hide them behind a level of abstraction



## Coroutines

- They are executed independently from the main function
- The coroutines stack can grow dynamically as needed
- In Kotlin there is a smart scheduler that can map coroutines to OS threads
- Coroutines follow the idea of Communicating sequential processes of Hoare

## But why should we bother: concurrency problems

If we access the same memory from two threads/ goroutines, than we have a race! Lets see this pseudo-code:

```
int i = 0

thread1 { i++ }
thread2 { i++ }

wait { thread1 } { thread2 }
print i
```

What will be the result of the computation?

## Critical Sections

- We provide Mutual Exclusion between different threads accessing the same resource concurrently
- There are many ways to implement Mutual Exclusion
- And the message passing using Channels of course :)

## Share by communicating vs. Communicate by sharing

We can use Mutex / Atomic primitives for mutual exclusion between coroutines as in other languages, but in most cases it happens to be simpler and more obvious to handle data to other coroutines using channels.

## Sequences

- Along with collections, the Kotlin standard library contains another container type – sequences (**Sequence<T>**). Sequences offer the same functions as **Iterable** but implement another approach to multi-step collection processing.
- When the processing of an **Iterable** includes multiple steps, they are executed eagerly: each processing step completes and returns its result – an intermediate collection. The following step executes on this collection. In turn, multi-step processing of **sequences** is executed lazily when possible: actual computing happens only when the result of the whole processing chain is requested.
- The order of operations execution is different as well: **Sequence** performs all the processing steps one-by-one for every single element. In turn, **Iterable** completes each step for the whole collection and then proceeds to the next step.

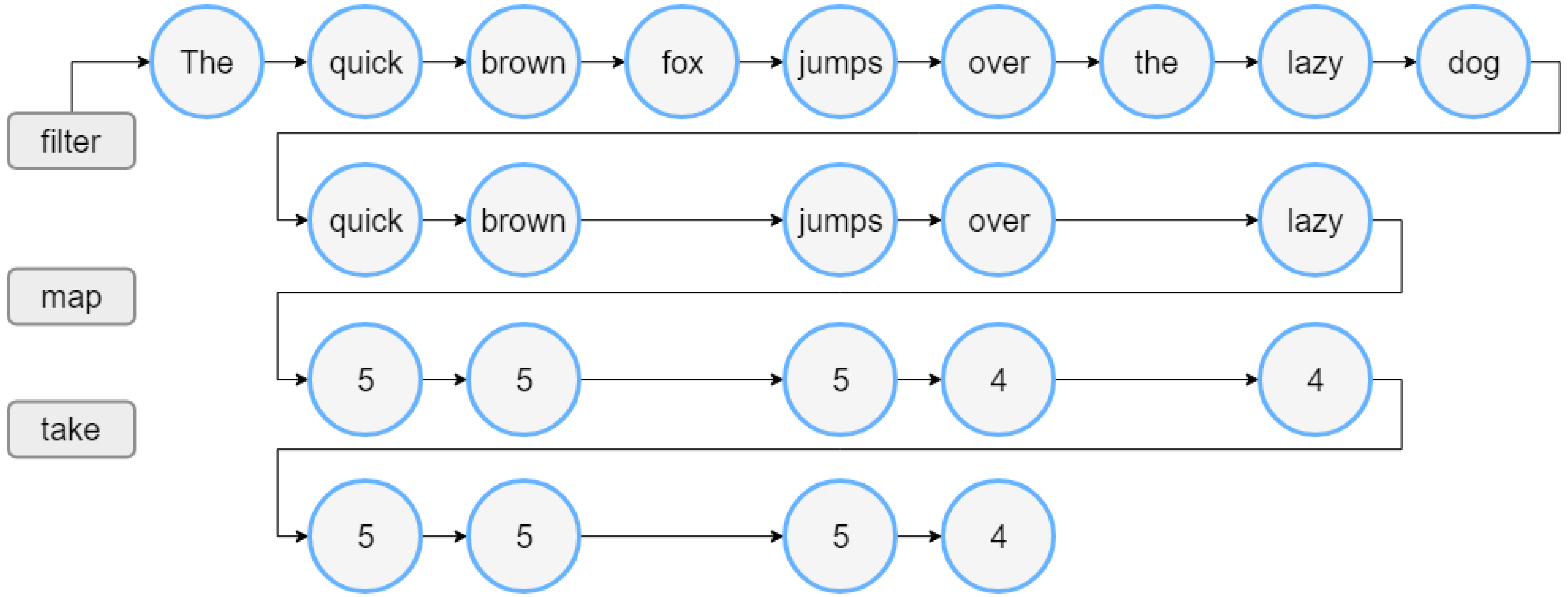
## Sequence Generation

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
val numbers = listOf("one", "two", "three", "four")
val numbersSequence2 = numbers.asSequence()

val oddNumbers = generateSequence(1) { it + 2 } // `it` is the previous element
println(oddNumbers.take(5).toList())
//println(oddNumbers.count())     // error: the sequence is infinite

val oddNumbersLessThan10 = generateSequence(1) { if (it < 8) it + 2 else null }
println(oddNumbersLessThan10.count())

val oddNumbers2 = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
```

## Iterable

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```

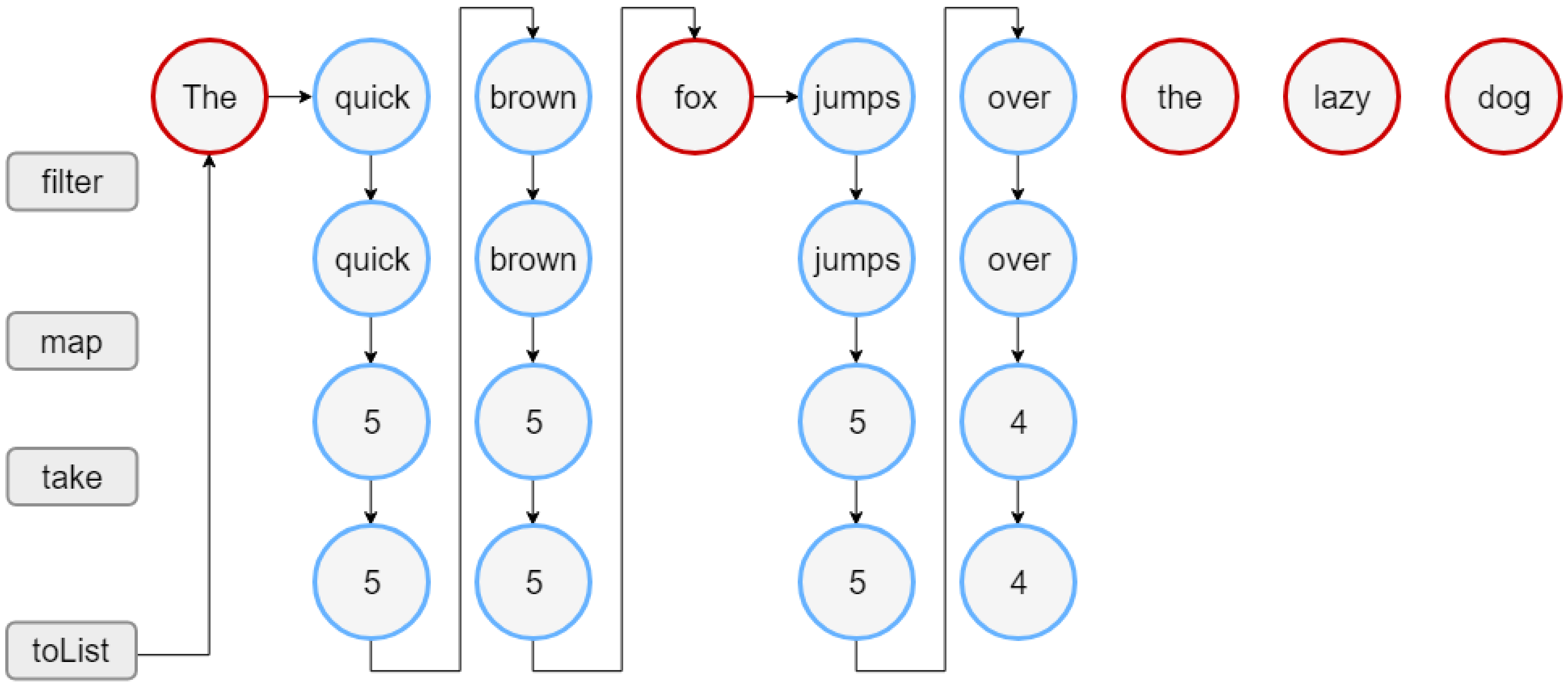
# Iterable


filter
map
take
The
quick
brown
fox
jumps
over
the
lazy
dog
quick
brown
jumps
over
lazy
5
5
5
4
4
5
5
5
4

## Sequence

```
 val words = "The quick brown fox jumps over the lazy dog".split(" ")
//convert the List to a Sequence
   val wordsSequence = words.asSequence()

   val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
       .map { println("length: ${it.length}"); it.length }
       .take(4)

   println("Lengths of first 4 words longer than 3 chars")
// terminal operation: obtaining the result as a List
   println(lengthsSequence.toList())
```

## Sequence


filter
map
take
toList
The
quick
brown
fox
jumps
over
the
lazy
dog
quick
brown
jumps
over
5
5
5
4
5
5
5
4

## Learn Kotlin by Example & Kotlin idioms

https://play.kotlinlang.org/byExample/

https://kotlinlang.org/docs/idioms.html

## Thank’s for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD