Kotlin/Native concurrency model

nikolay igotti@JetBrains

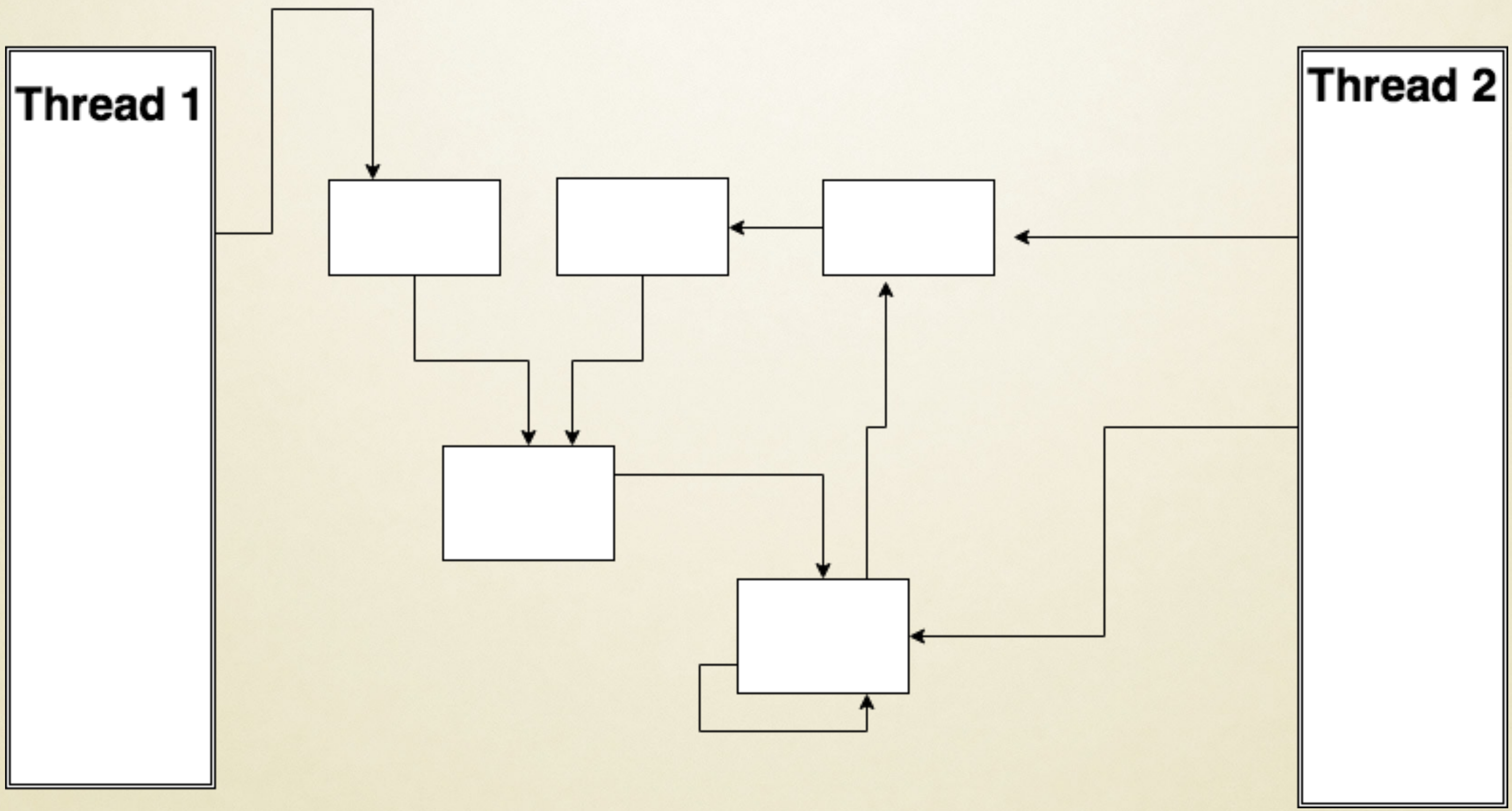
What do we want from concurrency?

- Do many things concurrently
- Easily offload tasks
- Get notified once task a task is done
- Share state safely
- Mutate state safely
- Avoid races and deadlocks

Concurrency in kotlin

- Kotlin as a language has no default concurrency primitives
- Kotlin/JVM uses JVM concurrency
- Kotlin/JS doesn’t have shared object heaps at all
- Threads are clumsy and error-prone
- Still concurrency is important on the modern hardware
- Kotlin/Native got a chance to do better!

Shared heap on JVM



The curse of shared object heap

- JVM is designed to make objects accessible from many mutators simultaneously
- Tracing GC requires complicated memory management algorithms
  - root marking — STW == global GC pauses
  - reachability analysis — STW (or complex algorithms) == GC pauses
  - STW — barriers on JNI borders == heavyweight native interop
- Reference counting is hard to use on the shared heap
  - Tricky to collect cycles
  - Requires atomic counter update
- Programmers can make concurrency errors and runtime doesn't help

Do we really need object sharing?

- For immutable objects - definitively
- For mutable objects - better object and its transitive closure be only accessible to the single mutator at the moment, i.e. having reference works as a lock
- This is better than mutex coming from `synchronized` keyword: no locks on access, no way to make concurrent update errors
- It also simplifies memory manager logic

Kotlin/Native at large

- Kotlin source code to the self-contained machine code, no VM or support libs
- For iOS, macOS, Linux, Windows, WebAssembly targets
- Automated memory management, collect cycles
- Fully automated interoperability with C/Objective-C/Swift
- Access to platform APIs (POSIX, Foundation, AppKit, Win32, etc.)

Kotlin/Native memory manager

- Simple local reference-counter based algorithm
- Cycle collector based on the trial deletion
- Storage containers separated from the objects
- Different container classes (normal, concurrent, permanent, arena)
- No object moving
- Interoperates with Objective-C runtime reference counter
- No cross-thread/worker interactions on memory manager

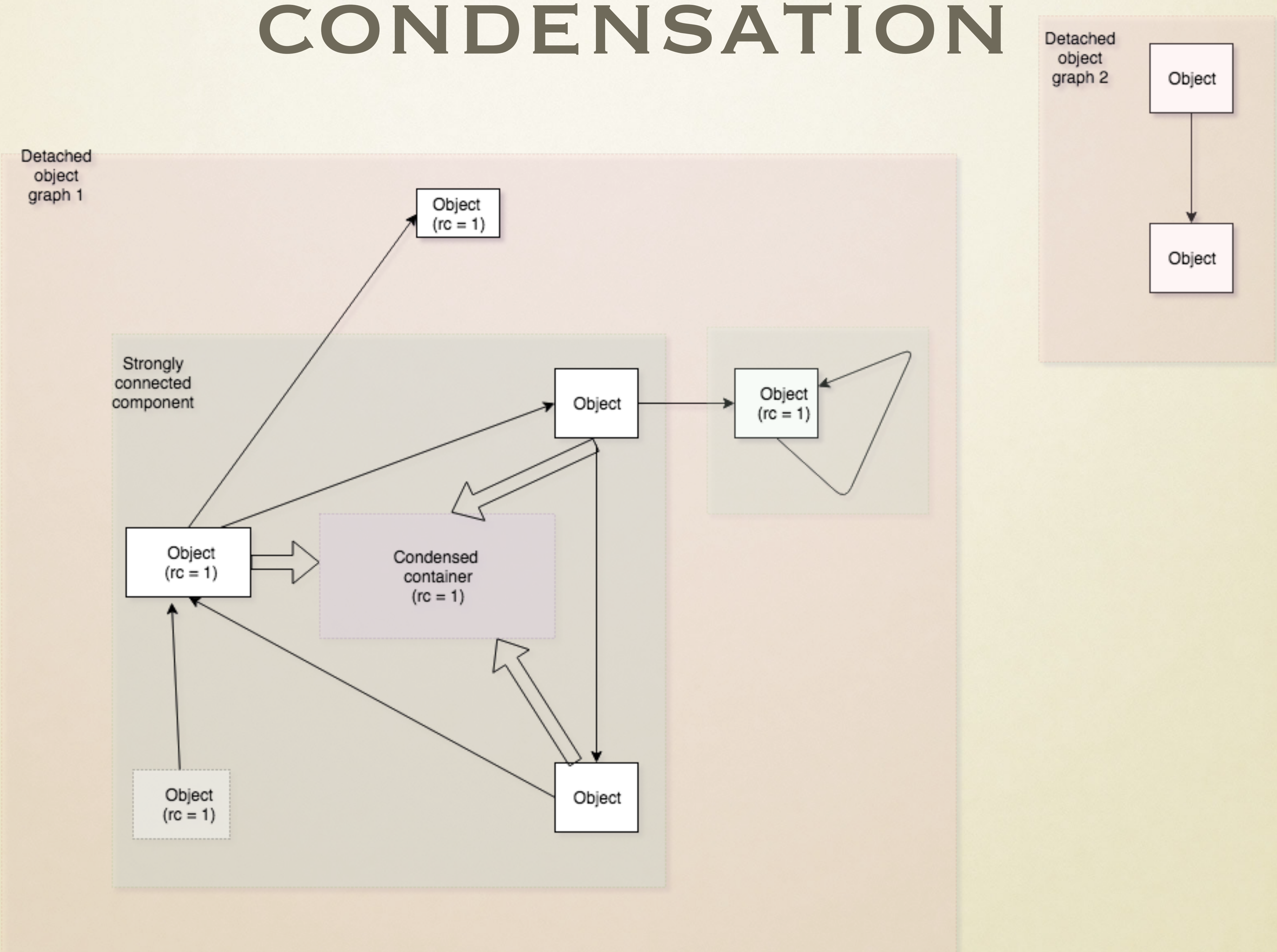
Kotlin got no 'const'

- Immutability is not part of the type system (yet)
- Let's start with the runtime property (like with nullability)
- Immutability is contagious, so propagates to the transitive closure
- Immutability is the one way road
- So welcome `Any.freeze()` (`kotlin.native.concurrent`) extension function!

Freezing

- Makes transitive closure of objects reachable from the given one immutable
- Aggregate strongly connected component to the single storage container, thus make any object graph a DAG
- On mutation attempt a runtime exception is thrown
- Frozen objects can be safely shared across workers
- Some carefully designed classes (i.e. `AtomicInt`) are marked as frozen, but could be mutated via concurrent-safe APIs
- System classes (like primitives boxes and `kotlin.String`) are frozen by default

Object graphs condensation



Sharing

- Frozen object can be safely shared
- Kotlin singleton objects (and companion objects) are frozen automatically after creation and shared
- Top level variables can be marked with the special annotation `@SharedImmutable`
- Default behavior of top level variables of non-value types is that they available from the main thread only
- Annotation `@ThreadLocal` marks top level variable as having private copy for each thread

CONCURRENT EXECUTORS - WORKERS

- Kotlin/Native has workers for computation offload
- Workers can only share immutable objects
- Mutable objects are owned by a single execution context (main thread or worker)
- Every worker has a job queue
- Main thread does not have a job queue (but there's UI queue)
- Workers are built on top of the OS threads

Object transfer

- Sometimes we need to pass data to the concurrent executor
- Along with data itself we could pass the ownership
- We cannot pass only object itself, we have to pass what it refers to
- In reference-counted runtime we could easily ensure object subgraph has no incoming references from the outside world (trial deletion)
- So welcome `kotlin.native.concurrent.Worker.execute`

Worker.execute

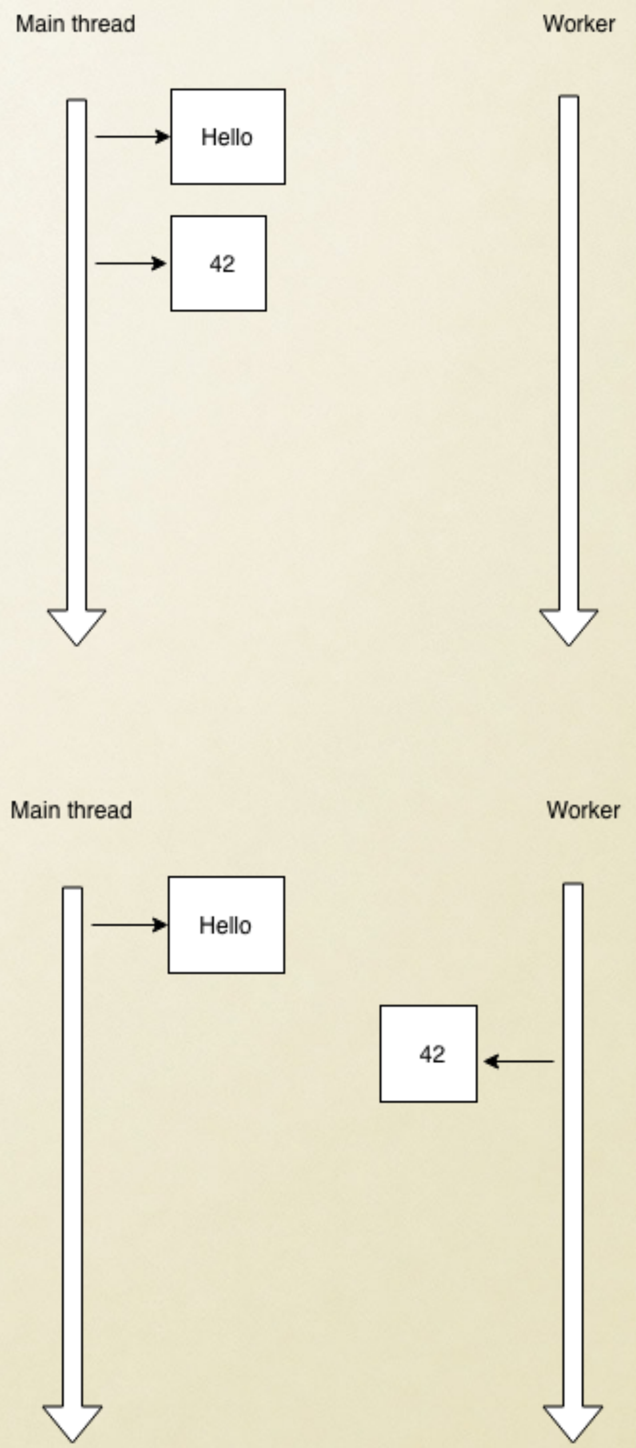
- ```
  public fun <T1, T2>
  execute(mode: TransferMode,
        producer: () -> T1,
       @VolatileLambda job: (T1) -> T2):
  Future<T2>
  ```
- `TransferMode` controls reachability check
- `producer` creates an object graph to detach and give to the worker
- `job` is special non-capturing lambda taking only result of producer and executed in worker context
- returned object is a future, which could be checked for execution status or consumed (on any worker), once ready
  ```

Worker sample

```kotlin
fun factorize(value: UInt): Pair<UInt, List<UInt>> {
    val result = mutableListOf<UInt>()
    var current = value
    outer@while (current > 1u) {
        for (candidate in 2u .. current) {
            if (current % candidate == 0u && isPrime(candidate)) {
                result += candidate
                current = current / candidate
            }
            if (current == 1u) break@outer
        }
    }
    return Pair(value, result.sorted())
}

fun workerSample() {
    val toFactorize = uintArrayOf(
        Random.nextUInt(),
        Random.nextUInt(),
        Random.nextUInt(),
        Random.nextUInt()
    )
    val COUNT = toFactorize.size
    val workers = Array(COUNT, { _ -> Worker.start()})
    val futures = Array(workers.size,
            { i -> workers[i].execute(TransferMode.SAFE, { toFactorize[i] })
            { input -> factorize(input) }
            })
    futures.forEach { future ->
        future.consume { result ->
            println("${result.first} is factored out to ${result.second.toString()}")
        }
    }

    workers.forEach { it: Worker
        it.requestTermination().result
    }
}
```

Object ping-pong example

```kotlin
data class Data(var int: Int)

fun pingPong() {
    val hello = "Hello"
    val worker = Worker.start()
    val future = worker.execute(TransferMode.SAFE, { Data( int: 42)}) {
        it -> it.int++
        println("1: in worker $it")
        ^execute it
    }
    val future2 = worker.execute(TransferMode.SAFE, {
        val result = future.result
        result.int++
        println("2: in main $result")
        ^lambda result
    }) {
        it -> it.int++
        println("3: in worker $it")
        ^execute it
    }
    println("4: in main ${future2.result}")
}
```



Why object graph detachment?

- Some objects are related
- They usually point each to another
- So if we want safe concurrency — they shall go together
- `DetachedObjectGraph` is the container for such structure
- Once detached — can be attached in another worker/thread safely
- Fully concurrent-safe, only one context can have access to objects in isolated object subgraph

Global variables

- Singleton objects (object and enum keyword)
- Top level variables
- Source of the (implicit) state sharing
- Singletons are frozen after creation
- Most top level variables are only accessible from the main thread
- Some immutable top level variables are accessible everywhere
- Can be controlled with `@ThreadLocal` and `@ImmutableShared` annotations

Important cases

- Shared cache: atomic reference for immutable elements, detached object graphs for mutable elements
- Job queue: use worker's queue
- Global constants/configuration: use singleton object or mark with `@SharedImmutable`, see below

Shared cache example

```
data class CacheEntry(var string: String = "name")

@SharedImmutable
val sharedImmutableCache = Array( size: 10) { _ -> AtomicReference<CacheEntry?>( value_: null) }

fun immutableCacheSample() {
    val workers = Array( size: 10, { _ -> Worker.start()})
    val futures = Array(workers.size) { i ->
        workers[i].execute(TransferMode.SAFE, { i }) { workerIndex ->
            for (attempt in 1..100) {
                val modifyIndex = Random.nextInt( from: 0, sharedImmutableCache.size)
                val value = sharedImmutableCache[modifyIndex].value
                if (value == null) {
                    val candidate = CacheEntry(
                        string: "attempt #$attempt of worker $workerIndex at index $modifyIndex"
                    ).freeze()
                    sharedImmutableCache[modifyIndex].compareAndSwap( expected: null, candidate)
                }
            }
        }
    }
    // Ensure all operations are done.
    futures.forEach { future -> future.consume { } }
    println(sharedImmutableCache.map {it -> it.value })
}
```

Concurrency and interop

- Kotlin/Native is tightly tied with the C/Objective-C world
- This world assumes threads/queues as a concurrency primitives
- Let’s play nice!
- Detached object graphs can be passed as `void*` anywhere
- Stable reference from any object can be passed as `void*` (only same thread for mutable, any for immutable)
- Objects can be pinned and pointer to object’s data can be passed as C pointer — no hard boundary with C world

Conclusions

- Kotlin/Native allows fine grained runtime mutability control with `freeze()` operation
- Kotlin/Native enforces good practices of immutable singleton objects and top level variables
- Kotlin/Native provides safe concurrency mechanisms (workers, detachable object graphs, atomics)
- Kotlin/Native can interoperate with C and Objective-C using concurrency-safe primitives
- **Kotlin/Native helps with writing safe concurrent code!**