Why do we need Promises?

Kacper Kaliński @ Miquido

What is a Promise?

- The term *promise* was proposed in 1976
- It is a proxy for delayed / long / asynchronous computation result
- It can have many names like *future*, *promise*, *delayed*, *deferred*
- It is common to have Future (read only) and associated Promise or Task (read/write)

Promise example

```
let promise = Promise<Int>()
async { // it might be in parallel or just at the end of this scope
    let value = heavyComputation()
    // or network call or even disk read
    promise.fulfill(value)
}
promise.callback { value in
    // access the value when ready
}
let value = promise.wait() // or wait here until it is ready
```

Why do we use Promise?

- ✦ Resolves callback hell (flattens callbacks and builds transformation chains)
- ✦ Can simplify reasoning about asynchronous, concurrent and even paralell code
- ✦ Removes or simplifies a lot of synchronization problems
- ✦ May improve testability and modularity
- ✦ Allows some implementations of async/await

How about async/await?

- ✦ It often replaces Promise usage in codebase...
- ✦ ... but can be complementary / extended by it
- ✦ It simplifies usage and reasoning even more...
- ✦ ...but requires cooperative multitasking (coroutines)
- ✦ It removes some properties of Promise (if there is no conversion available)
- ✦ It might harden ability to execute things in parallel

Concurrency vs Parallelism

"Concurrency is dealing with a lot of things at once.
Parallelism is doing a lot of things at once."

— Rob Pike (co-author of go)

async/await example

```
func heavyComputation() async -> Int { ... }

let value = await heavyComputation()
```

Functional note - Monad

If Promise has defined a function which allows us to apply functions...

... one thing to note here - if we have monadic Promise (and a little support from language) we might not need async/await at all 🤯

(but it is probably not best solution in Swift - see Async/Await proposal)

Monad example

```
let intFuture: Future<Int> = ...
let doubleFuture: Future<Double> = ...

let resultFuture: Future<String> =
doMonad {
    // access wrapped value with special syntax
    let intVal: Int <- intFuture
    let doubleVal: Int <- doubleFuture
    // do normal stuff without locking
    // and return final value
    // it will be wrapped in Future again at the end
    return "\(intVal) + \(doubleVal)"
}
```

Monad explained

```
let intFuture: Future<Int> = ...
let doubleFuture: Future<Double> = ...

let resultFuture: Future<String> =
intFuture.flatMap { (intVal) -> (Future<String>) in
    doubleFuture.flatMap({ (doubleVal) -> (Future<String>) in
        Future(succeededWith:"\(intVal) + \(doubleVal)")
    })
}
```

Implementation

```
class Promise<Value> {
    var value: Value? = nil
    var callbacks: [(Value) -> Void] = []

    func fulfill(_ value: Value) {
        guard result == nil else { return }
        result = value
        callbacks.forEach { $0(value) }
        callbacks = []
    }
    func callback(_ callback: @escaping (Value) -> Void) {
        if let value == value {
            callback(value)
        } else {
            callbacks.append(callback)
        }
    }
}
```

Implementation - Locking

```
let lock: Lock
var value: Value? {
    get {
        lock.lock()
        defer { lock.unlock() }
        return _value
    }
    set {
        lock.lock()
        defer { lock.unlock() }
        _value = newValue
    }
}
private var _value: Value? = nil
```

SwiftNIO

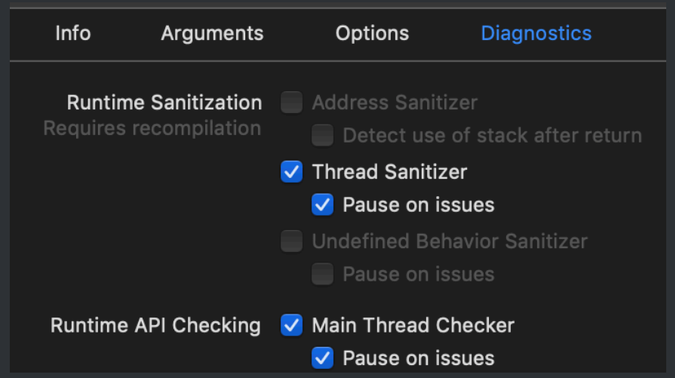
SwiftNIO project contains implementation of Promise without locks 🤯

Synchronization is done by context switching - tasks are automatically switched to associated threads removing need of synchronization.

Safe multithreading

If you develop any concurrent code you should always have thread sanitizer enabled.
Disable it only for performance checks or to use other debug tools.

```
swift test -c release -v --sanitize=thread
```



Making promise might look easy but... there are some pitfalls if you are no familliar with concurrency, locking etc.
Fortunately my company gave me some time to do open source 😎
(and there are some other implementations out there 😅)

Example - basic api

```
let promise: Promise<Int> = Promise() // write only
let future: Future<Int> = promise.future // read only
future.value { value in
    // add a callback for value
}
future.error { error in
    // add a callback for error
}
promise.fulfill(with: 0)
promise.break(with: Error())
```

Example - transformations api

```
let integerFuture: Future<Int> = ...
let stringFuture: Future<String> = integerFuture.map(String.init)
let throwingFuture: Future<Data>
    = stringFuture.map { value in //we get only value here
        throw Error.unimplemented
}
let recoveredFuture: Future<Data>
    = throwingFuture.recover { error in
        if case Error.unimplemented = error {
            return Data()
        } else {
            throw error
        }
}
```

Example - cancelation api

```
let future: Future<Int> = ...
let mapped: Future<String> = future.map(String.init)
mapped.cancel() // value or error will be ignored - finishes now
// cancelation is propagated up but not down
future.value { value in
    // migt still be there
}
mapped.always {
    // on value, error and cancel - use for celanup
}
```

Example - threading api

```
let future: Future<Int> = ...
    future.always {} // on finishig queue / immediately on current queue
let dq = future.switch(to: DispatchQueue.main)
dq.always {} // on DispatchQueue.main
let oq = future.switch(to: OperationQueue.main)
oq.always {} // on OperationQueue.main
let own = future.switch(to: MyOwnWorker())
own.always {} // on MyOwnWorker
```

Example - joining api

```
let future_one: Future<Int> = ...
let future_two: Future<Int> = ...
let zippedTuple: Future<(Int, Int)> = zip(future_one, future_two) // first error or all values
let zippedArray: Future<[Int]> = zip([future_one, future_two, future_three...]) // first error or all
    values
let flattened: Future<Int> // error from future_one or combined value / error from future_two
    = future_one.flatMap { value in
        return future_two.map { $0 + value }
}
```

Example - debug

```
let future: Future<Int> = ...
    future.debug(.single) // log events from this instance
future.debug(.propagated) // log evetns from this and all further instances
```

Example - tests sync

```
let testWorker: TestWorker = TestWorker()
let testPromise: Promise<Int> = Promise()
let testFuture: Future<Int>
    = testPromise.future.switch(to: testWorker) // change execution to manual

testPromise.fulfill(with: 0) // nothing will be done in chain
testFuture.value { // it will wait for manual execution
    XCTAssert($0 == 0)
}

testWorker.execute() // executes all scheduled tasks
```

Example - tests async

```
let asyncFuture: Future<Int> = ...
let anyExpectation = asyncFuture.expectValue(timeout: 3)
let concreteExpectation = asyncFuture.expectValue({ value in value == 1 })

anyExpectation.waitForExpectation()
concreteExpectation.waitForExpectation()
```

DEMO

Useful links

Swift Functors, Applicatives, and Monads in Pictures http://www.mokacoding.com/blog/functor-applicative-monads-in-pictures/

Futura https://github.com/miquido/futura

SwiftNIO https://github.com/apple/swift-nio

Async/Await for Swift https://gist.github.com/lattner/429b9070918248274f25b714dcfc7619

Concurrency Is Not Parallelism https://vimeo.com/49718712

Thank you!

Any questions?