The Future of Swift

The Future of Swift

I bet Apple regrets neglecting to annotate @clattner_llvm noescape.

— Daniel Jalkut (@danielpunkass)

(Very) Quick History Lesson

- Jul/10 - Internal Development at Apple
- WWDC14 - Public unveiling
- Sep/14 - Swift 1.0
- Sep/15 - Swift 2.0 *(try/catch, guard, defer)*
- Dec/15 - Swift Open-Source 🍾
- Sep/16 - Swift 3.0 *(Swifty API)*

Our goal for Swift has always been for it to take over the world.

— Chris Lattner (Tesla Motors, Inc.)

Swift 3.1 (Spring 2017)

New Sequence Methods (3.1)

```
// dropLast()
[12, 30, 5, 11].dropLast()
// [12, 30, 5]
```

🆕

```
// drop(while:)
[12, 30, 5, 11].drop(while: { $0 > 10 })
// [5, 11]

// prefix(while:)
[12, 30, 5, 11].prefix(while: { $0 > 10 })
// [12, 30]
```

Confidence: ✅

New API Annotations (3.1)

```
// Relative to OS version
@available(iOS 10.0, *)
```

🆕

```
// Relative to Swift version
@available(swift 3)
@available(swift, introduced: 3)
@available(swift, obsoleted: 3.1)
```

Confidence: ✅

New Numeric Initializers (3.1)

```
// Rounding errors
let integer = Int(3.9)
// integer: 3
```

🆕

```
// Failing initializers
let integer = Int(exactly: 3.9)
// integer: nil

let integer = Int(exactly: 3.0)
// integer: 3
```

Confidence: 🔜

Protocol-oriented integers (3.1)

```
// Compare error
let firstInteger = Int8(42), secondInteger = 4
if firstInteger > secondInteger {}
// error: binary operator '>' cannot be applied to operands of type 'Int8' and 'Int'
```

🆕

```
// Works now
let firstInteger = Int8(42), secondInteger = 4
if firstInteger > secondInteger {}
// true
```

Confidence: 🔜

Swift 4 (Late 2017)

Swift 4 (Late 2017)

- Stabilization of the ABI
  - Usage of frameworks not compiled for your Swift version
- Improved compile time, compiler reliability, and error messages
- New generics features needed by standard library
- Standard library API improvements, e.g. String

Memory Ownership Model (4.0)

- Opt-in memory ownership typing for references
  - Fast: Unique ownership guarantees no ARC 🏎️
  - Safe: Correctness is enforced statically 🎉
- Type system enhancements
  - `owned`: Have responsibility for value
  - `borrowed`: Just using it temporarily

Memory Ownership Model (4.0)

- Low burden, but still more than most users care about
  - Aiming at users who want C++ level performance
  - Everyone else should be able to ignore it

Memory Ownership Model (4.0)

Potential syntax:

```
extension Collection {
    func map<T>(_ f: (borrowed Element) -> T) -> [T] {
        var result = [T]()
        for element in self {
            result.append(f(element))
        }
        return result
    }
}
```

Confidence: ✅

Swift 5 (2018)

Swift 5 (2018)

- Tackling Task Concurrency
- Goal: Start discussions in Spring/Summer 2017
- Aim to have a “manifesto” design sketch by Fall 2017
- First deliverables in Swift 5

Task Concurrency (5.0)

- `async/await` for elegant async operations
  - Native concurrency at language level
  - Solves completion handler „pyramid of doom“
  - C#, Javascript, Python, Kotlin etc.

Confidence: ✅

Task Concurrency (5.0)

Potential syntax:

```
async func downloadImage(from url: URL) -> Task<UIImage?> {
    do {
        let dataTask: Task<Data> = URLSession.shared.dataTask(with: url)
        let data: Data = try await dataTask
        return UIImage(data: data)
    } catch {
        return nil
    }
}

let image: UIImage? = await downloadImage(from: url)
```

Actor Model (5.0)

- Actor model to define tasks, along with managed state
- Each Actor is effectively
  - A `DispatchQueue`
  - State it manages
  - Operations that act on it
- Erlang, Akka (JVM)

Actor Model (5.0)

Potential syntax:

```
actor NetworkRequestHandler {
    private var userID: UserID

    async func processRequest(_ connection: Connection) {
        // send messages to other actors
        // create new actors
        // modify local state
    }
}

let requestHandler = NetworkRequestHandler()
await requestHandler.processRequest(connection)
```

Actor Model (5.0)

- Better reliability model
  - Terminate failing Actor instead of entire process 💣
  - All `awaits` on an Actor's method throw an error
  - Enables custom failure recovery
  - Runtime cleans up resources owned by that Actor

Confidence: 

Cycle Collector (5.0)

- Commonly requested, technically feasible to implement
  - But: Thinking (a little bit) about memory is good!
- “Leaks” are possible with any memory management model
- Code is read/maintained far more than it is written
- ARC provides an explicit model for memory management
  - `weak`, `unowned`

Cycle Collector (5.0)

- Availability of a cycle collector would partition the community
  - Some packages would rely on it, some would not
  - If on by default, almost everything would rely on it

🆕

- Automatically suggest weak in the right places

Confidence: ➡️ SOON

Not gonna happen ⛔

Tracing Garbage Collection

Drawbacks

- Native interoperability with unmanaged code
  - Possible but would introduce significant complexity (JNI)
- Non-deterministic object destruction
  - ARC gets you deterministic destruction of objects
  - Eliminates “finalizers” as a concept (resurrection, threading)

Tracing Garbage Collection

Drawbacks

- Performance
  - May run at unfortunate times (stutter)
  - Uses ~3-4x more memory than ARC to achieve good performance
  - Memory usage is very important for mobile and cloud apps

Open-Source Anarchy

```
func parse(input: (Int | String)?) -> String?
    unless input != nil
        fatalError(“No input“)

    let intInput: Int? = Int(input)

    if let intInput =?
        return if intInput % 2 == 0
            "x is even"
        else
            "x is odd"
    else
        return nil
```

Thank you

patrick.gaissert@maibornwolff.de