Lua Workshop 2016

Programming iOS in Lua
A bridge story

Jean-Luc Jumpertz
@JLJump

October 14, 2016

CodeFlow

Live Application Development Environment
for iOS, tvOS & macOS

CodeFlow

Live Application Development Environment for iOS, tvOS & macOS

instant feedback

on real devices

native OS SDK

Lua language

native project APIs

live code

live assets

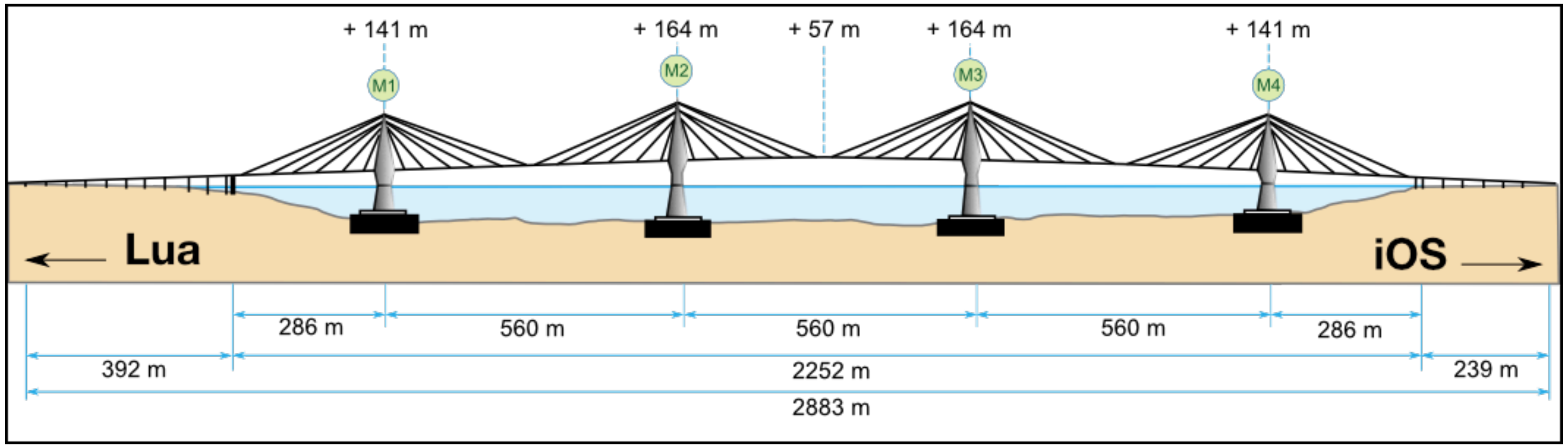
true debugger

live storyboards

A Bridge? What for?

Transparent development of iOS code in Lua

Goals of the iOS bridge



- Enable the development of iOS apps in Lua using the native OS SDK
- Make the use of the native SDK feel natural in Lua
- Make it easy for a Swift or ObjC developer to move to Lua

⇒ **Transparent integration between Lua and iOS**

- Not the same objective as some other bridges
  - Exposing Lua-specific features to the iOS native world was not in the scope, nor was the definition of a Swift / ObjC version of the Lua C API.
  - Low-level aspects of the native world had to be hidden from the Lua code

The foundations

Dealing with type conversions, memory management, and threads

Mixing Lua and native types

- Different typing systems
  - Lua: typed values; untyped function parameters
  - C world: typed variables and parameters; ABI
- Calling native from Lua: convert parameters to the expected types
  - Easy for base types, more complex for structured types, objects, collections…
  - Doing this conversion is the first role of a bridge
- Example: expose a struct to Lua
  - Pseudo-object with constructor, accessors, … and methods
  - Automatic Lua table → struct conversion in function calls

```lua
local CGPoint = struct.CGPoint:_structInterface { x = 0.0, y = 0.0 }
```

```c
struct CGPoint {
    CGFloat x;
    CGFloat y;
};
```

C

```lua
local aPoint = struct.CGPoint (100, 50)
aPoint.x = 200
self.view.center = aPoint
-- ...
self.view.center = { x = 150, y = aPoint.y + 20 }
```

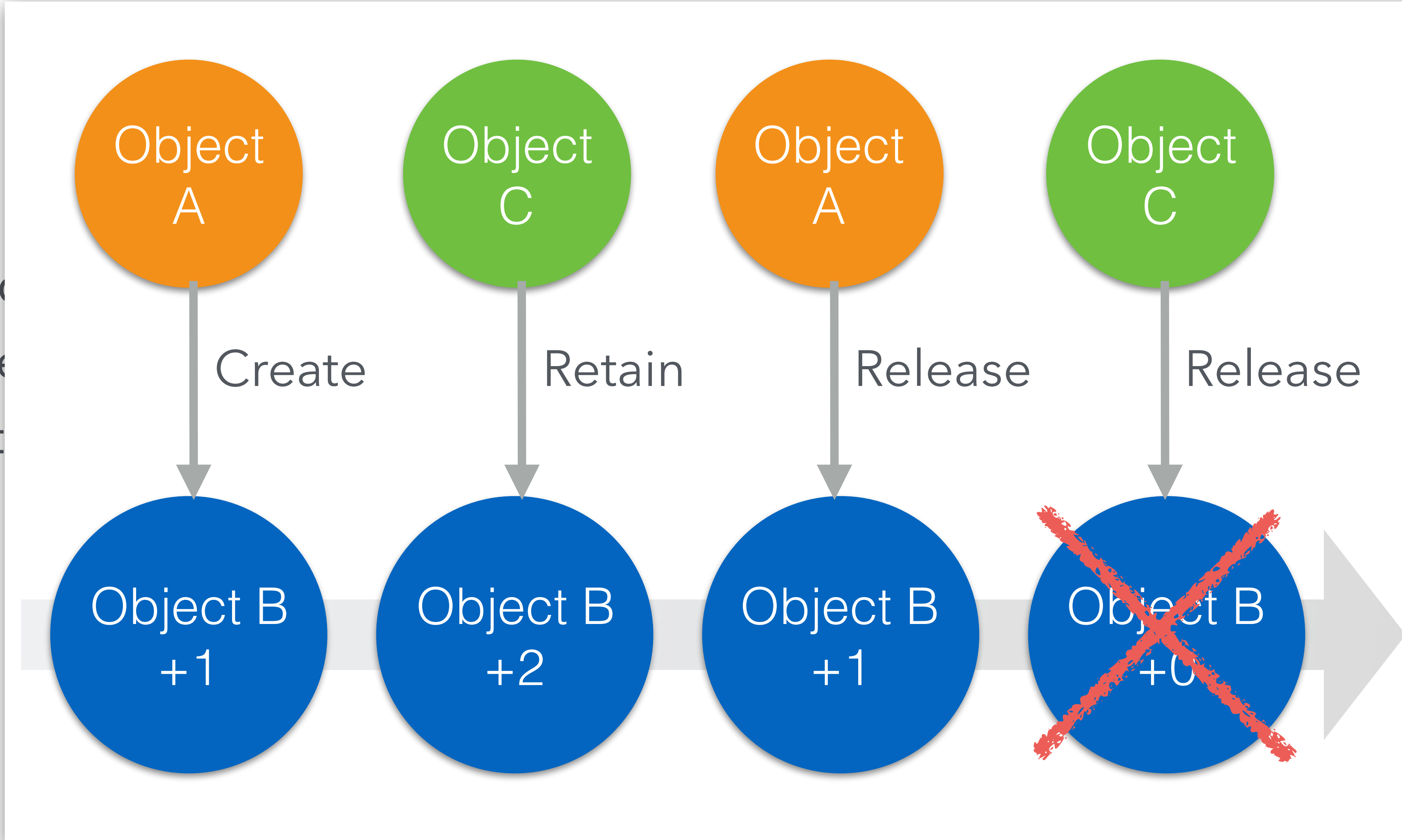
Lua

Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting

Making

- Different memory mod
  - Lua: garbage colle
  - ObjC runtime: aut



Making Memory Models Coexist

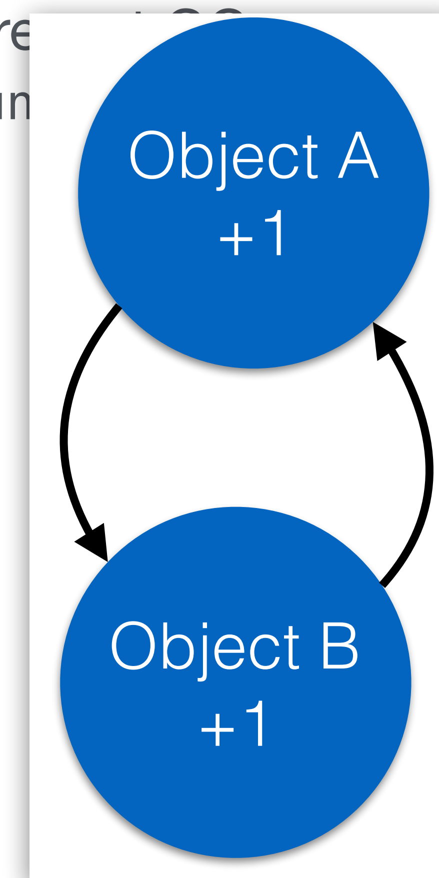
- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to prevent GC (`luaL_ref`) and remove this reference when not used anymore. (`luaL_unref`)

Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to prevent GC (`luaL_ref`) and remove this reference when not used anymore. (`luaL_unref`)
- The retain cycle problem
  - It is possible from Lua, to create a retain cycle between native objects

Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to pre (`luaL_ref`) and remove this reference when not used anymore. (`luaL_un`
- The retain cycle problem
  - It is possible from Lua, to create a retain cycle between native objects



Making Memory Models Coexist

- Different memory models
  - Lua: garbage collector
  - ObjC runtime: automatic reference counting
- Managing objects lifecycle
  - A native object passed to the Lua runtime is retained until GC-ed, and released by its finalizer metamethod
  - A Lua value passed to the native world maintains a Lua reference to prevent GC (`luaL_ref`) and remove this reference when not used anymore. (`luaL_unref`)
- The retain cycle problem
  - It is possible from Lua, to create a retain cycle between native objects

    ⇒ memory leak!
  - Weak object references are the solution
    - Object reference getters: `weakRef` and `strongRef`

      ```
      local weakSelf = self.weakRef
      ```
    - A weak reference become an *all-nil* object when the referenced object is deallocated

Running Lua in a Threaded World

- Lua runs as a single thread, while the host OS is heavily multi-threaded
- In an iOS app, code execution is triggered by user or external events

  ⇒ We can not control in which thread our Lua methods are called!

- The iOS bridge has to make Lua work in a multi-threaded environment

Running Lua in a Threaded World

- Lua runs as a single thread, while the host OS is heavily multi-threaded
- In an iOS app, code execution is triggered by user or external events

  ⇒ We can not control in which thread our Lua methods are called!
- The iOS bridge has to make Lua work in a multi-threaded environment

- Our solution:
  - Every top-level Lua code invocation runs in its own Lua thread (i.e. lua_State)
  - A simple scheduler allows to execute only one Lua thread at a given time, with well-defined deschedule points
- Looks simple but works great in practice!

Design Patterns Translation

Making native design patterns
feel natural in Lua

About Native Design Patterns

- An API is not just about types and function: how to use it is even more important.
- Typical design patterns define the expected way to use the APIs.
- The iOS / macOS SDKs rely on strong design patterns and conventions: MVC, delegation, observing, target-action…
- Making these design patterns feel natural in Lua is key for the bridge usability!

Now, a few examples of design patterns adaptation to Lua:

Pattern 1: Subclass to Customize

This is how Controllers work in iOS.

⇒ We need the possibility to subclass native classes in Lua!

```
local ViewController = class.createClass ("ViewController", objc.UIViewController)

function ViewController:loadView ()
    -- Create a view programmatically.
    self.view = objc.UIView:new()
end

function ViewController:viewDidLoad ()
    self[ViewController.superclass]:viewDidLoad()
    self:configureView ()
    self:addMessageHandler (ViewController, "refreshView")
end

function ViewController:configureView ()
    -- Put here the code configuring the controller's view
    self.view.backgroundColor = objc.UIColor.whiteColor
end

function ViewController:refreshView()
    -- Update the controller's view
    self:configureView()
    -- Other refresh actions
    -- ...
end

return ViewController
```

This creates a Lua subclass of native UIViewController

Two native methods overriden in Lua

Two Lua methods not visible from the native code

Pattern 2: delegation

- A *delegate* object is used to customize or control the actions of a SDK object, by implementing a well-defined API contract declared as a *protocol.* A delegate object can be of any class, provided it implements the expected protocol.
- A Lua object can be declared as the delegate of a native object.
- *Publishing* a protocol makes the protocol's methods defined by a Lua class callable from the native code

This creates a Lua class (with no native superclass)

```
local TableDataSource = class.createClass("TableDataSource")

function TableDataSource:setTableView (tableView)
    self.tableView = tableView
    tableView.datasource = self
end
```

Instances of this class are used as 'data source of a native UITableView object

```
TableDataSource:publishObjcProtocols "UITableViewDataSource"

function TableDataSource:tableView_numberOfRowsInSection (tableView, section)
    local objects = self.objects
    return objects and #objects or 0
end
```

Implement mandatory methods of protocol UITableViewDataSource

```
function TableDataSource:tableView_cellForRowAtIndexPath (tableView, indexPath)
    local cell = tableView:dequeueReusableCellWithIdentifier_forIndexPath("Cell", indexPath)
    local object = self.objects [indexPath.row + 1]
    cell.textLabel.text = object.description
    return cell
end
```

Pattern 3: closure parameters

- Closure (aka ObjC *blocks*) parameters are used for synchronous or asynchronous callback in many places of the iOS / macOS SDKs
- Lua functions are a perfect match for closure parameters!

This native NSString method takes a closure parameter.

```
function CollectionController:setCollectionText(text)
    local words = {}
    local wordsCount = 0
    text:enumerateSubstringsInRange_options_usingBlock
                                (NSRange(0, text.length),
                                 NsString.Enumeration.ByWords,
                                 function(word, range, effectiveRange)
                                     wordsCount = wordsCount + 1
                                     words[wordsCount] = word
                                 end)
    self.textWords = words
    self.collectionView:reloadData()
end
```

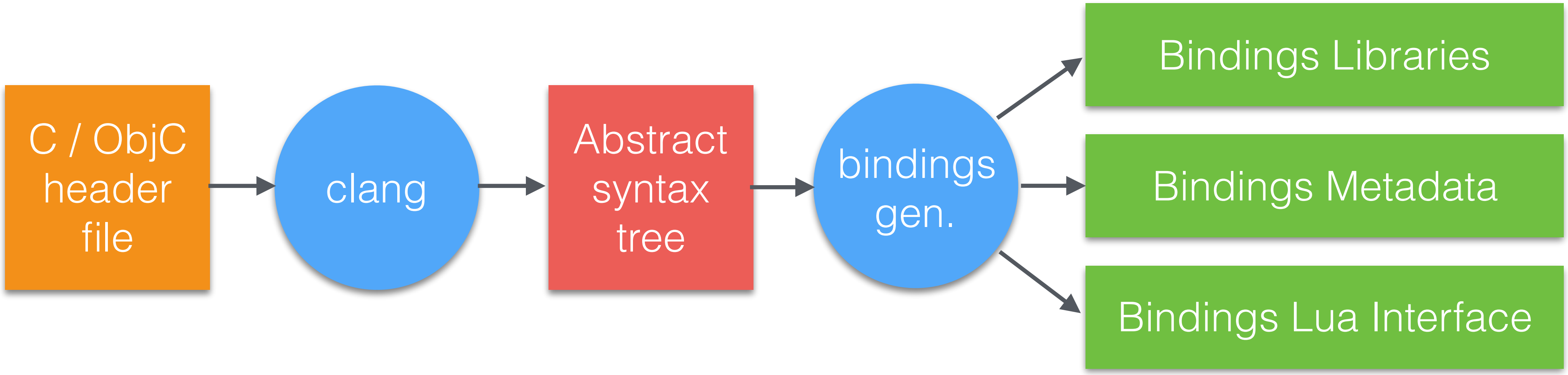
You simply pass a Lua function for this closure parameter


Bindings Generation
Supporting large OS SDKs thanks to automation

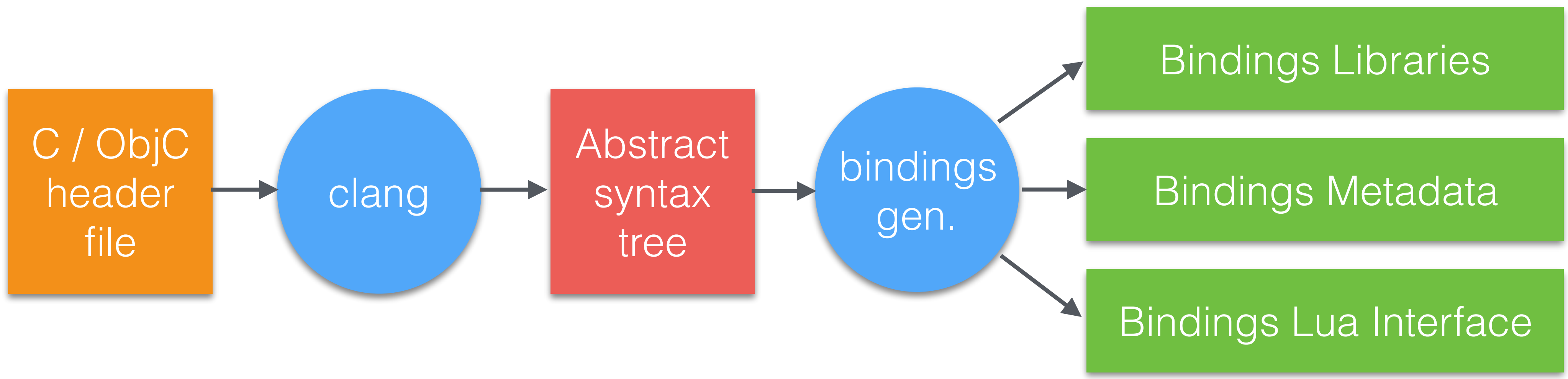
SDK Bindings Generation

- Two main components in the bridge
  - Generic bridge library: memory & threads management, OO framework, generic type conversion and function call bridging
  - *Bindings*: the specific code that makes the bridge work for a given SDK or API
- iOS / macOS SDKs are quite big (~1900 header files for iOS, 2300 for macOS)

  ⇒ Bindings generation has to be automated
- Use clang (llvm) for parsing C / Objective-C headers
- Bindings generation is based on the AST generated by clang



SDK Bindings Generation



- Bindings Libraries
  - Mix of generated code and declarative typing information
  - Linked with the target application
  - Include: constants, enums, structs, C functions, classes with methods and properties, protocols …
  - Loaded as Lua modules

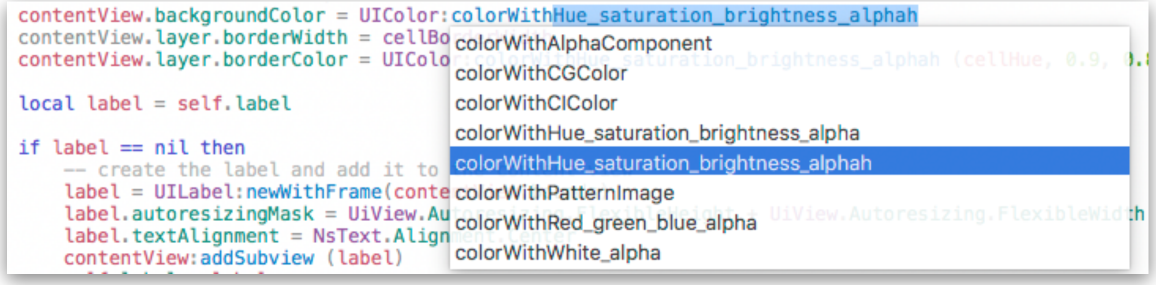
    ```
    local UiGestureRecognizer = require "UIKit.UIGestureRecognizer"
    ```
- Bindings Metadata
  - Used by the IDE
- Bindings Lua Interface
  - A user-readable Lua version of the SDK

IDE Integration

Supporting native SDKs in the IDE for a better coding experience

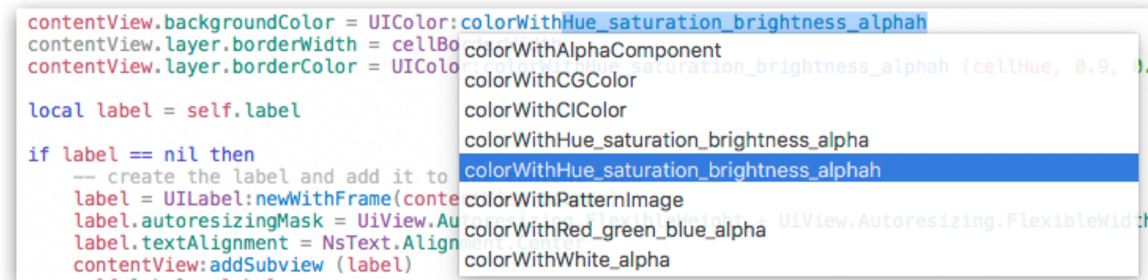
Bridge - IDE Integration

- Goal: help the developer to use the native SDK(s) in Lua
- In the Lua source code editor
    - auto-completion of SDK symbols defined in Bindings Libraries

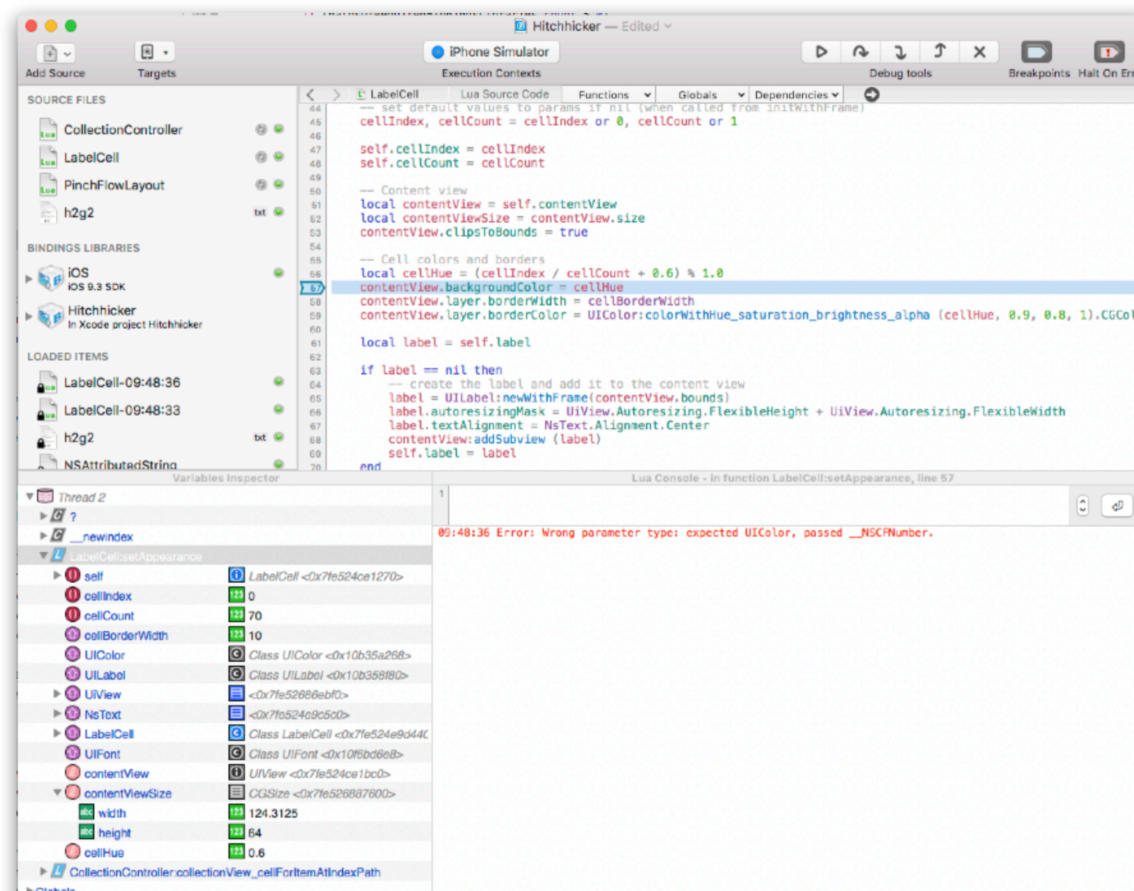


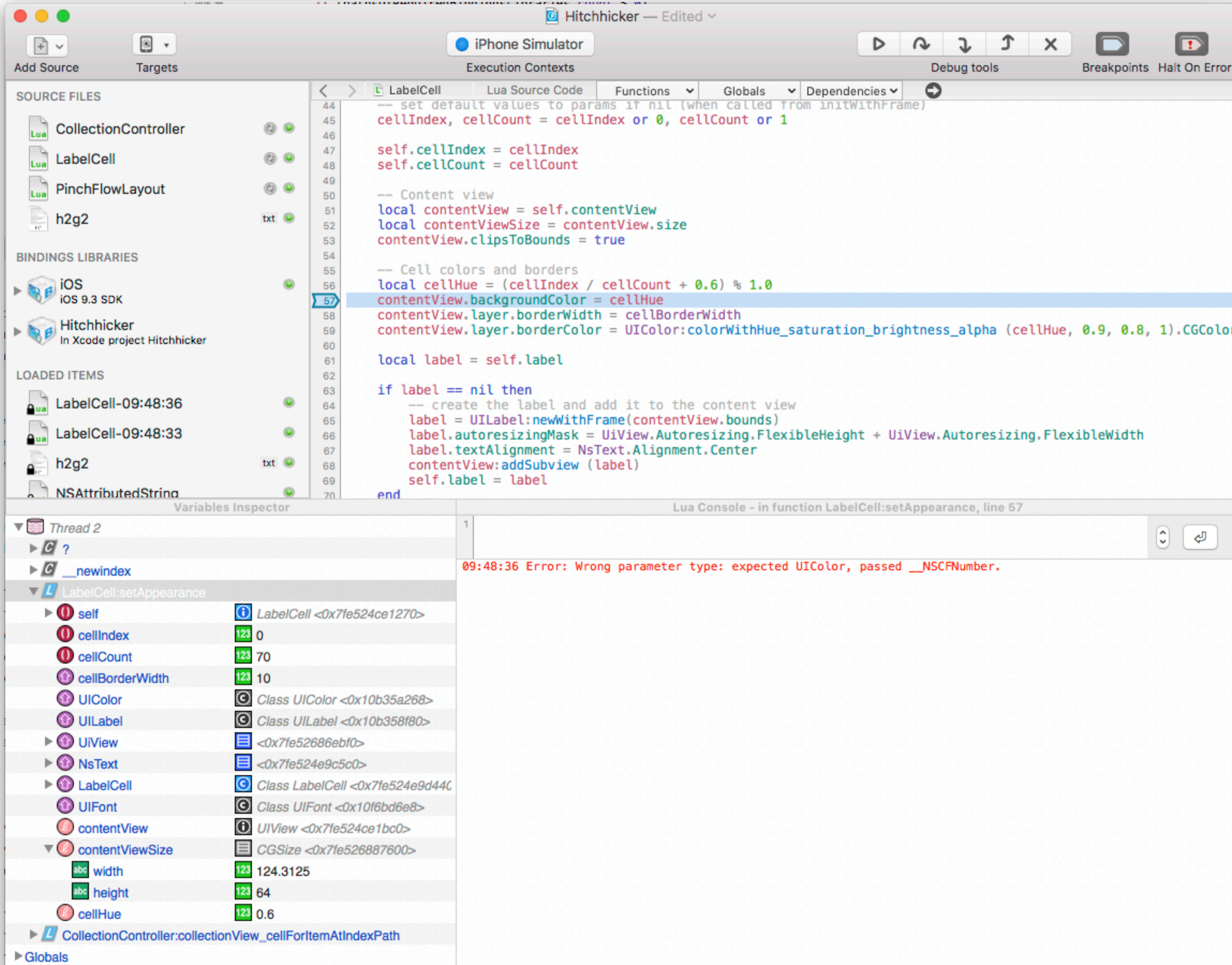
Bridge - IDE Integration

- Goal: help the developer to use the native SDK(s) in Lua
- In the Lua source code editor
  - auto-completion of SDK symbols defined in Bindings Libraries



- For build configuration of target app
  - by computing bindings-related dependencies in Lua modules
- In the Lua debugger
  - inspect native types in the Variables Inspector
  - interrupt on error in case of failed type conversion or wrong nullability … and continue execution after fixing the issue!




Hitchhicker — Edited
Add Source
Targets
iPhone Simulator
Execution Contexts
Debug tools
Breakpoints
Halt On Error
SOURCE FILES
CollectionController
LabelCell
PinchFlowLayout
h2g2
txt
BINDINGS LIBRARIES
iOS
iOS 9.3 SDK
Hitchhicker
In Xcode project Hitchhicker
LOADED ITEMS
LabelCell-09:48:36
LabelCell-09:48:33
h2g2
txt
NSAttributedString
LabelCell
Lua Source Code
Functions
Globals
Dependencies
-- set default values to params if nil (when called from initWithFrame)
cellIndex, cellCount = cellIndex or 0, cellCount or 1
self.cellIndex = cellIndex
self.cellCount = cellCount
-- Content view
local contentView = self.contentView
local contentViewSize = contentView.size
contentView.clipsToBounds = true
-- Cell colors and borders
local cellHue = (cellIndex / cellCount + 0.6) % 1.0
contentView.backgroundColor = cellHue
contentView.layer.borderWidth = cellBorderWidth
contentView.layer.borderColor = UIColor:colorWithHue_saturation_brightness_alpha (cellHue, 0.9, 0.8, 1).CGColor
local label = self.label
if label == nil then
-- create the label and add it to the content view
label = UILabel:newWithFrame(contentView.bounds)
label.autoresizingMask = UiView.Autoresizing.FlexibleHeight + UiView.Autoresizing.FlexibleWidth
label.textAlignment = NsText.Alignment.Center
contentView:addSubview (label)
self.label = label
end
Variables Inspector
Lua Console - in function LabelCell:setAppearance, line 57
09:48:36 Error: Wrong parameter type: expected UIColor, passed __NSCFNumber.
Thread 2
?
__newindex
LabelCell:setAppearance
self
LabelCell <0x7fe524ce1270>
cellIndex
0
cellCount
70
cellBorderWidth
10
UIColor
Class UIColor <0x10b35a268>
UILabel
Class UILabel <0x10b358f80>
UiView
<0x7fe52686ebf0>
NsText
<0x7fe524e9c5c0>
LabelCell
Class LabelCell <0x7fe524e9d44(
UIFont
Class UIFont <0x10f6bd6e8>
contentView
UIView <0x7fe524ce1bc0>
contentViewSize
CGSize <0x7fe526887600>
width
124.3125
height
64
cellHue
0.6
CollectionController:collectionView_cellForItemAtIndexPath
Globals


Tour completed
What have we seen?

Recap

Needed for this bridge:

- A well-defined goal for the iOS bridge.
- Solid low-level foundations: types, memory and threads.
- Careful transposition of the SDK’s main design patterns.
- Bindings generation tools to support large SDKs.
- IDE integration to brings additional value to the user.

For More Information

- About CodeFlow and live-coding on iOS
  - Explore https://www.celedev.com
  - Play with live-coding iOS with Lua: https://www.celedev.com/en/download/
  - Follow the project: @celedev

- About the iOS bridge
  - Read our *Get Started with Lua* series
    https://www.celedev.com/en/documentation/get-started/get-started-with-lua
  - Part 2: CodeFlow object framework
  - Part 3: CodeFlow native bridge

Thank You!

Questions?

Jean-Luc Jumpertz
@JLJump

www.celedev.com
@celedev