

Introduction to LLDB and the Swift REPL

Session 409
Sean Callanan
Debugger Engineer

LLDB Is Better Than Ever with Swift

The screenshot shows the Xcode interface with the LLDB debugger open. The title bar reads "DispatchTest — main.swift". The left sidebar displays the project structure under "DispatchTest" with a PID of 88574, Paused. Below this are metrics for CPU (0%), Memory (1.7 MB), Energy Impact (Zero), Disk (Zero KB/s), and Network (Zero KB/s). The bottom section shows two threads: "Thread 1" (Queue: com.apple.main-thread) and "Thread 2" (Queue: hello (serial)). Thread 2 is currently selected, showing stack frames for "0 DispatchTest.(dispatch...)". The right pane shows the source code for "main.swift" with a breakpoint at line 14. The code includes imports from Foundation, a function "dispatch_onto_queue", and a variable "a" set to 2. The status bar at the bottom indicates "Auto" mode.

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t, block : dispatch_block_t) {
    dispatch_async(queue) {
        println("Hello \(a)")
    }
}

var queue = dispatch_queue_create("hello", nil)
dispatch_onto_queue(queue, 2)
dispatch_sync(queue, {})
```

A a = (Swift.Int) 2

LLDB Is Better Than Ever with Swift

It helps you fix problems in your code!

The screenshot shows the Xcode interface with the LLDB debugger open. The title bar says "DispatchTest — main.m". The left sidebar shows the project structure with "DispatchTest" selected. The main area displays the following Swift code:

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t) {
    dispatch_async(queue) {
        println("Hello \(a)")
    }
}

var queue = dispatch_queue_create("helloQueue", nil)
dispatch_onto_queue(queue, 2)
dispatch_sync(queue, {})
```

The code is stepped through, with line 14 highlighted in green. The variable "a" is being inspected, showing its value as 2. The bottom status bar indicates "Auto ⌘ | ⌂ i | ⌂ Search".

LLDB Is Better Than Ever with Swift

It helps you fix problems in your code!

LLDB has a variety of bug-fixing tools

- The **stack** diagnoses a stopped app
- **Breakpoints** stop an app when needed
- The **expr** command inspects data

The screenshot shows the Xcode debugger window for a Swift application named "DispatchTest". The application is listed in the top left with "PID 88574, Paused". Below it, resource monitoring sections show "CPU" at 0%, "Memory" at 1.7 MB, "Energy Impact" at Zero, "Disk" at Zero KB/s, and "Network" at Zero KB/s. In the bottom left, the "Threads" section shows "Thread 1" (Queue: com.apple.main-thread) and "Thread 2" (Queue: hello (serial)). Thread 2 is expanded to show stack frames: 0 DispatchTest.(dispatch...), 1 reabstraction thunk..., 2 _dispatch_call_blo..., and 9 start_wqthread. The main pane displays the source code for "main.swift" in the "DispatchTest" folder. Line 14 is highlighted with a green background, showing the code: "println(\"Hello \\\(a)\")". A blue selection bar is visible above the code area. At the bottom right, there's a search bar with the placeholder "Search".

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t, block : dispatch_block_t) {
    dispatch_async(queue) {
        block()
    }
}

var queue = dispatch_queue_create("hello", nil)
dispatch_onto_queue(queue, 2) {
    println("Hello \(a)")
}
dispatch_sync(queue, {})

A a = (Swift.Int) 2
```

LLDB Is Better Than Ever with Swift

It helps you fix problems in your code!

LLDB has a variety of bug-fixing tools

- The **stack** diagnoses a stopped app
- **Breakpoints** stop an app when needed
- The **expr** command inspects data

These tools work great with Swift!

- Use them to find familiar bugs...
- ...and some new ones!

The screenshot shows the Xcode interface with the LLDB debugger open. The title bar says "DispatchTest — main.m". The left sidebar shows the project structure with "DispatchTest" selected. The main area displays the LLDB command-line interface. At the top, there are sections for CPU (0%), Memory (1.7 MB), Energy Impact (Zero), Disk (Zero KB/s), and Network (Zero KB/s). Below these are sections for Thread 1 and Thread 2. Thread 1 is for the main queue, and Thread 2 is for a serial queue named "hello". A stack trace is shown for Thread 2, starting with "0 DispatchTest.(dispatch_main + 0)". The source code for "main.swift" is visible on the right, showing a function "dispatch_onto_queue" that prints "Hello \a". The variable "a" is set to the value 2. The bottom of the window has search and filter controls.

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t) {
    dispatch_async(queue) {
        println("Hello \\"a\"")
    }
}

var queue = dispatch_queue_create("hello", nil)
dispatch_onto_queue(queue, 2)
dispatch_sync(queue, {})
```

...and Now There's a REPL, Too!

The screenshot shows the Xcode IDE with the Instruments tool open. The main window displays performance metrics for a process named "DispatchTest" (PID 88574, Paused). The metrics shown are CPU (0%), Memory (1.7 MB), Energy Impact (Zero), Disk (Zero KB/s), and Network (Zero KB/s). Below these metrics, two threads are listed: "Thread 1" (Queue: com.apple.main-thread) and "Thread 2" (Queue: hello (serial)). Thread 2 is currently selected, showing stack frames for "0 DispatchTest.(dispatch_main+0x10)", "1 reabstraction thunk... (inlined)", "2 _dispatch_call_blo...", and "9 start_wqthread".

The right side of the interface shows the code editor for "DispatchTest.m". The file contains Swift code related to dispatch queues and a function "twice". A command-line interface (CLI) window titled "(lldb)" is open, showing the results of running the "twice" function.

```
// Copyright (c) 2014 Apple Inc. All rights reserved.
import Foundation
func dispatch_onto_queue(var queue : dispatch_queue_t, block: dispatch_block_t) {
    dispatch_async(queue) {
        block()
    }
}
var queue = dispatch_queue_create("hello", nil)
dispatch_onto_queue(queue, 2)

(lldb) repl
1> func twice(x : Int) -> Int {
2.     return x * 2
3.
4.
5. }
6> twice(3)
$R1: (Int) = 6
7>
```

...and Now There's a REPL, Too!

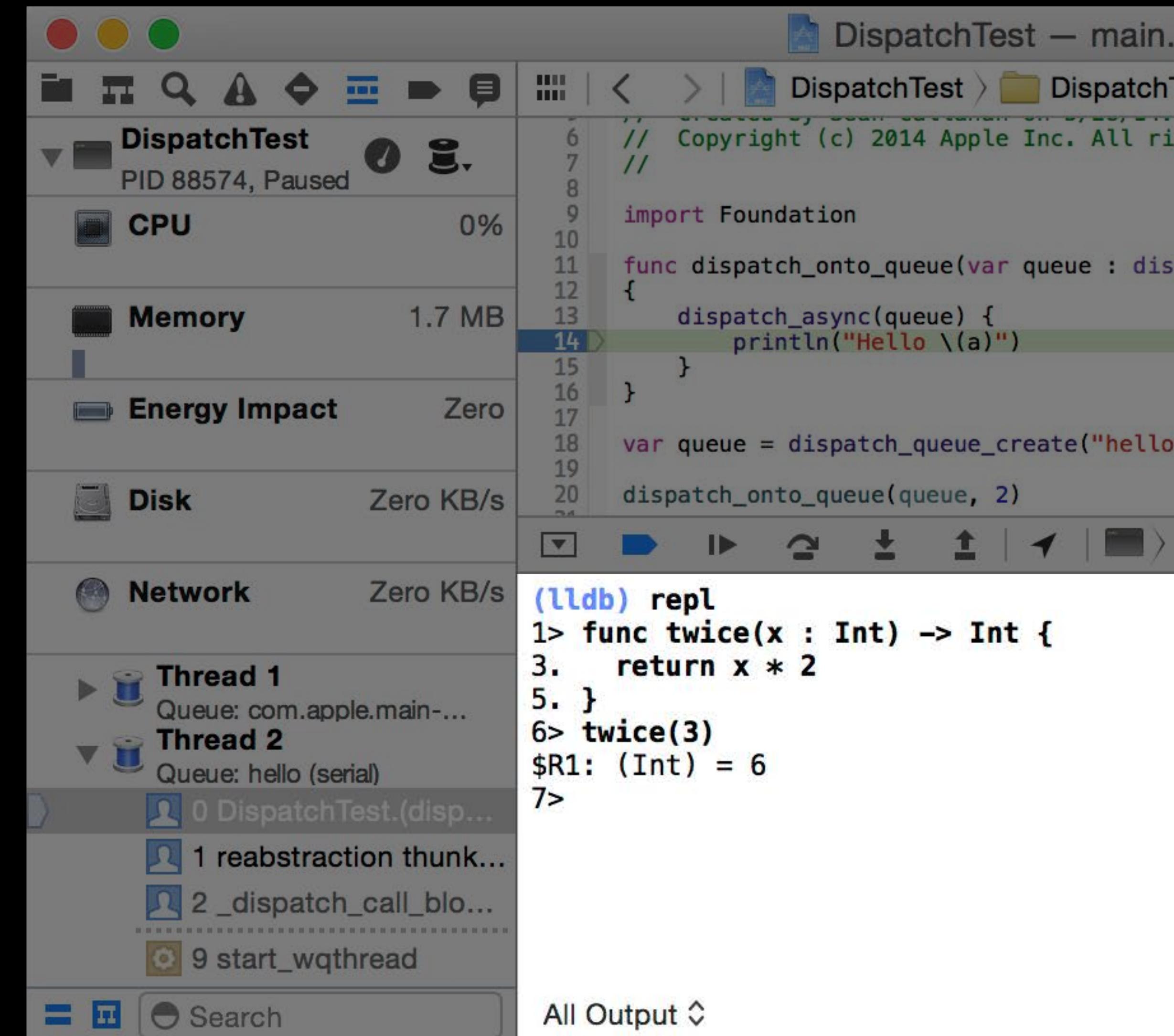
The screenshot shows the Xcode interface with the lldb window open. The top status bar indicates "DispatchTest — main.m". The left sidebar displays various system metrics: CPU (0%), Memory (1.7 MB), Energy Impact (Zero), Disk (Zero KB/s), and Network (Zero KB/s). Below these are two threads: "Thread 1" (Queue: com.apple.main-thread) and "Thread 2" (Queue: hello (serial)). Thread 2 has several stack frames visible: 0 DispatchTest.(dispatch...) at the top, followed by 1 reabstraction thunk..., 2 _dispatch_call_blo..., and 9 start_wqthread at the bottom. The right pane shows the source code for main.m and a REPL session. The source code includes imports for Foundation and Dispatch, defines a dispatch_onto_queue function, and creates a serial queue named "hello" to which it dispatches a print message. The REPL session starts with "(lldb) repl", followed by a definition of a "twice" function that returns twice its input. When "twice(3)" is run, the response "\$R1: (Int) = 6" is shown. The bottom right corner of the lldb window has a "All Output" button.

```
// Copyright (c) 2014 Apple Inc. All rights reserved.
import Foundation
func dispatch_onto_queue(var queue : dispatch_queue_t, block : dispatch_block_t) {
    dispatch_async(queue) {
        block()
    }
}
var queue = dispatch_queue_create("hello", DISPATCH_QUEUE_SERIAL)
dispatch_onto_queue(queue, 2)

(lldb) repl
1> func twice(x : Int) -> Int {
2.     return x * 2
3.
4.
5. }
6> twice(3)
$R1: (Int) = 6
7>
```

...and Now There's a REPL, Too!

The Read-Eval-Print-Loop is built on LLDB



The screenshot shows the Xcode interface with the LLDB debugger open. The top status bar indicates "DispatchTest — main.m". The left sidebar displays various metrics: CPU (0%), Memory (1.7 MB), Energy Impact (Zero), Disk (Zero KB/s), Network (Zero KB/s), and Thread information (Thread 1 and Thread 2). The bottom pane shows the REPL session:

```
(lldb) repl
1> func twice(x : Int) -> Int {
2.     return x * 2
3.
4.
5. }
6> twice(3)
$R1: (Int) = 6
7>
```

...and Now There's a REPL, Too!

The Read-Eval-Print-Loop is built on LLDB

You can use it anywhere

- (lldb) **repl** when your app is running
- \$ **xcrun swift** for a clean slate

The screenshot shows the Xcode interface with the LLDB debugger open. The top status bar indicates "DispatchTest — main.m". The left sidebar shows the project structure with "DispatchTest" selected. The main area displays the LLDB command-line interface. A Swift script named "main.swift" is being run, showing code related to dispatch queues and a function named "twice". The bottom part of the screen shows the output of the "repl" command, where the user defines a "twice" function and evaluates it with the value 3, resulting in the output \$R1: (Int) = 6.

```
// Copyright (c) 2014 Apple Inc. All rights reserved.
import Foundation
func dispatch_onto_queue(var queue : dispatch_queue_t, block: dispatch_block_t) {
    dispatch_async(queue) {
        block()
    }
}
var queue = dispatch_queue_create("hello", nil)
dispatch_onto_queue(queue, 2)

(lldb) repl
1> func twice(x : Int) -> Int {
2.     return x * 2
3.
4.
5. }
6> twice(3)
$R1: (Int) = 6
7>
```

...and Now There's a REPL, Too!

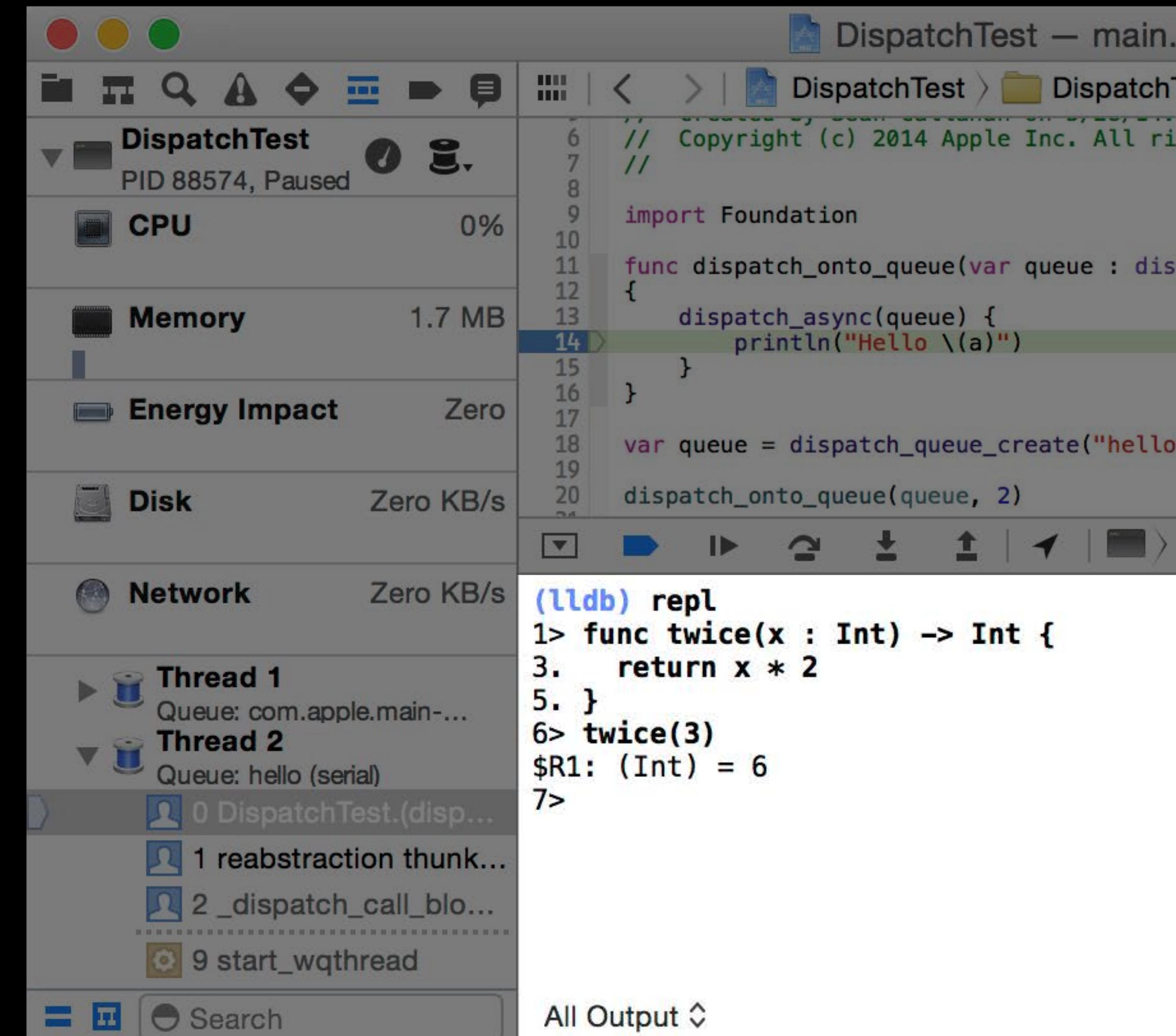
The Read-Eval-Print-Loop is built on LLDB

You can use it anywhere

- (lldb) **repl** when your app is running
- \$ **xcrun swift** for a clean slate

We'll show you some of the ways to use it

- Test your app interactively
- Try out new code in your app



The screenshot shows the Xcode debugger interface with the LLDB window open. The title bar says "DispatchTest — main.m". The left sidebar shows the project structure with "DispatchTest" selected. The main area displays the source code of "main.m" with line 14 highlighted: "println("Hello \\\(a)")". Below the code, the LLDB REPL is active, showing the following session:

```
(lldb) repl
1> func twice(x : Int) -> Int {
2.     return x * 2
3.
4.
5. }
6> twice(3)
$R1: (Int) = 6
7>
```

The bottom status bar shows "All Output".

Where the REPL Fits In

Interactive development tools and their uses

The **expression** command

Call functions, edit data in scope

LLDB Python scripting

Create new debugger features

Playgrounds

Prototype from scratch

The LLDB REPL

Add code to an existing session

Overview of Today's Session

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

REPL-enabled debugging workflows

- Validating existing code
- Trying out new code

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

REPL-enabled debugging workflows

- Validating existing code
- Trying out new code

Summing up

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

REPL-enabled debugging workflows

- Validating existing code
- Trying out new code

Summing up

Basic Debugging Survival Skills

Interacting with LLDB

The screenshot shows the Xcode interface during a debugging session. The top navigation bar indicates the project is "DispatchTest" and the file is "main.swift". The left sidebar displays various system metrics: CPU (0%), Memory (1.7 MB), Energy Impact (Zero), Disk (Zero KB/s), and Network (Zero KB/s). The bottom-left pane shows two threads: "Thread 1" (Queue: com.apple.main-thread) and "Thread 2" (Queue: hello (serial)). The bottom-right pane is the "Output" tab, which contains an LLDB repl session. The repl session starts with a function definition:

```
(lldb) repl
1> func twice(x : Int) -> Int {
2.     return x * 2
3.
4> twice(3)
$R1: (Int) = 6
5>
```

In the main editor area, the code for "main.swift" is shown. A breakpoint is set at line 14, which contains the line `println("Hello \(a)")`. The status bar at the bottom right of the editor indicates "Thread 2: breakpoint 1.1".

```
6 // Copyright (c) 2014 Apple Inc. All rights reserved.
7 //
8
9 import Foundation
10
11 func dispatch_onto_queue(var queue : dispatch_queue_t, a : Int)
12 {
13     dispatch_async(queue) {
14         println("Hello \(a)")
15     }
16 }
17
18 var queue = dispatch_queue_create("hello", nil)
19
20 dispatch_onto_queue(queue, 2)
```

Basic Debugging Survival Skills

Interacting with LLDB

The screenshot shows the Xcode interface during a debugging session. The top status bar indicates "DispatchTest — main.swift". The left sidebar displays various system metrics: CPU (0%), Memory (1.7 MB), Energy Impact (Zero), Disk (Zero KB/s), and Network (Zero KB/s). The bottom sidebar lists threads: Thread 1 (Queue: com.apple.main-thread) and Thread 2 (Queue: hello (serial)). The main area shows the source code for `main.swift`:

```
// Copyright (c) 2014 Apple Inc. All rights reserved.  
import Foundation  
func dispatch_onto_queue(var queue : dispatch_queue_t, a : Int)  
{  
    dispatch_async(queue) {  
        println("Hello \(a)")  
    }  
}  
var queue = dispatch_queue_create("hello", nil)  
dispatch_onto_queue(queue, 2)
```

A breakpoint is set at line 14, which is highlighted in green. The status bar at the bottom right shows "Thread 2: breakpoint 1.1". In the bottom right corner of the main window, there is a small icon with a blue border.

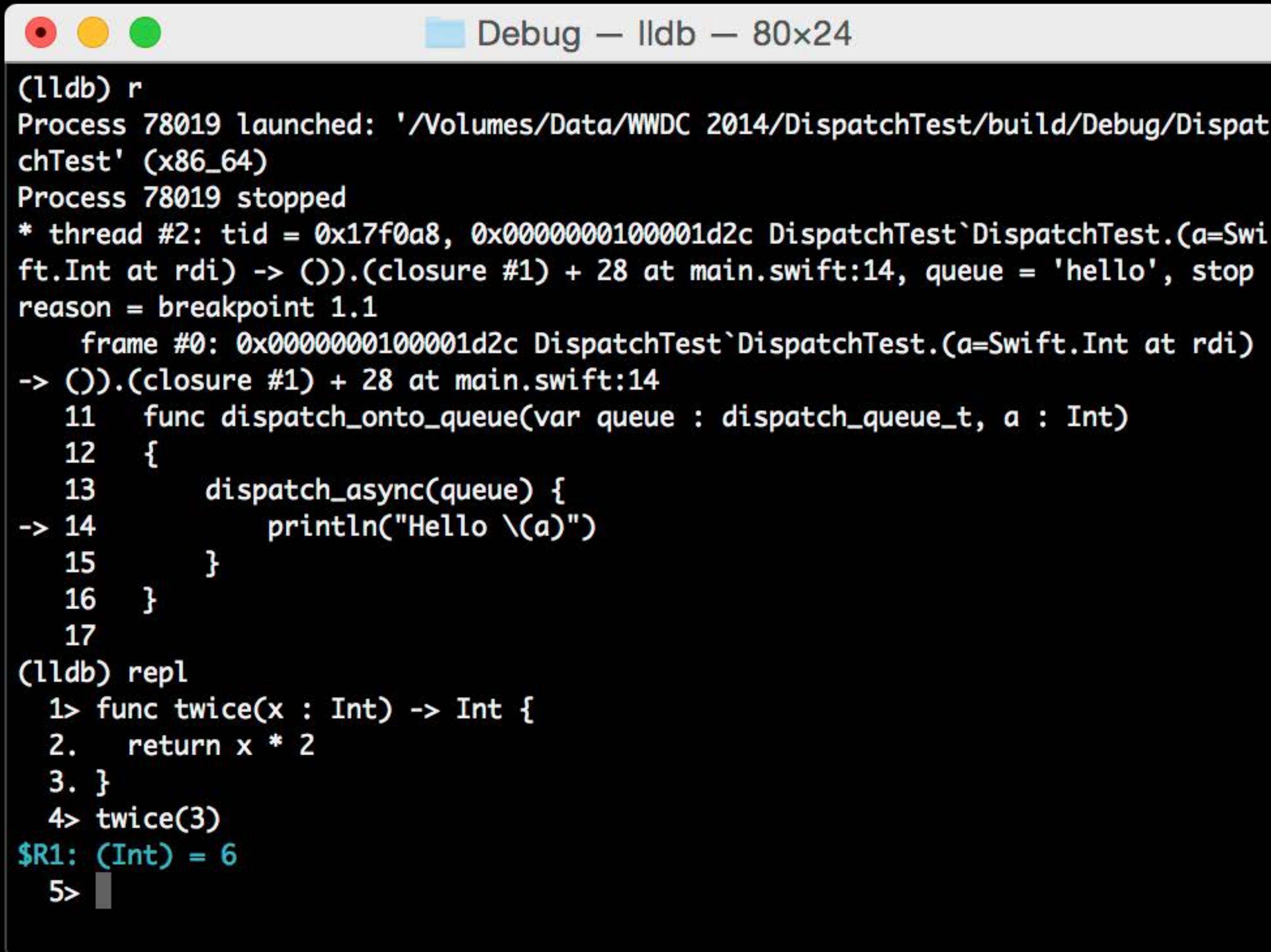
In the bottom left, the LLDB repl window shows:

```
(lldb) repl  
1> func twice(x : Int) -> Int {  
3.     return x * 2  
5. }  
6> twice(3)  
$R1: (Int) = 6  
7>
```

The bottom center of the screen has a "Search" field and a "All Output" dropdown.

Basic Debugging Survival Skills

Interacting with LLDB



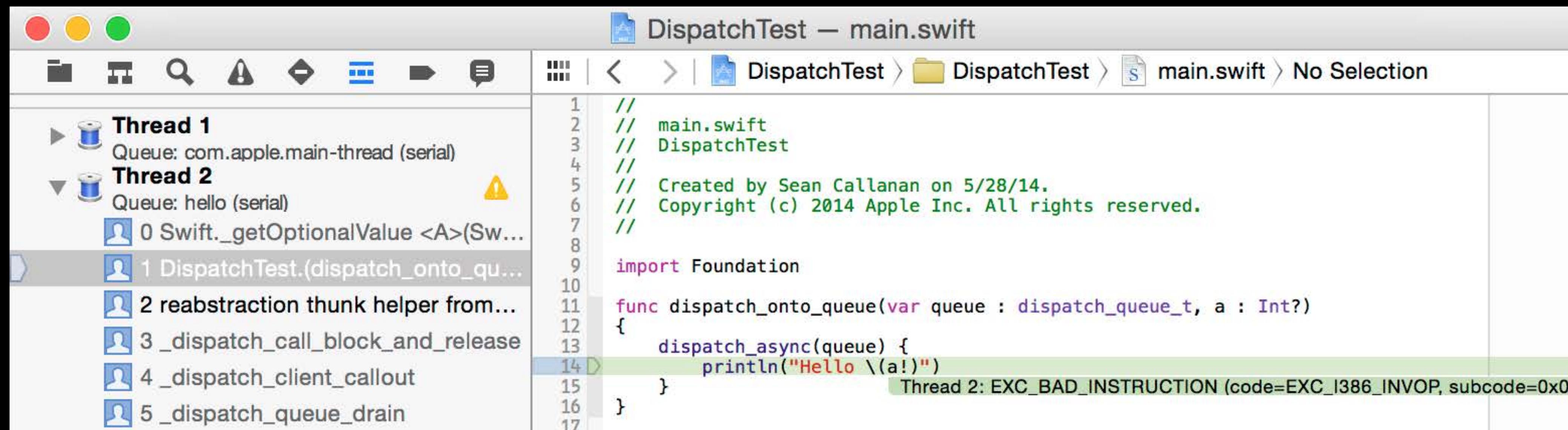
The screenshot shows the Xcode debugger interface with the title bar "Debug — lldb — 80x24". The main window displays LLDB commands and Swift code. The session starts with "(lldb) r", launching a process named "DispatchTest". The process stops at a breakpoint, showing frame #0 with a call to "dispatch_onto_queue". The code being debugged is:

```
(lldb) r
Process 78019 launched: '/Volumes/Data/WWDC 2014/DispatchTest/build/Debug/DispatchTest' (x86_64)
Process 78019 stopped
* thread #2: tid = 0x17f0a8, 0x0000000100001d2c DispatchTest`DispatchTest.(a=Swift.Int at rdi) -> ()).(closure #1) + 28 at main.swift:14, queue = 'hello', stop reason = breakpoint 1.1
    frame #0: 0x0000000100001d2c DispatchTest`DispatchTest.(a=Swift.Int at rdi) -> ()).(closure #1) + 28 at main.swift:14
    11     func dispatch_onto_queue(var queue : dispatch_queue_t, a : Int)
    12 {
    13     dispatch_async(queue) {
-> 14         println("Hello \(a)")
    15     }
    16 }
    17

(lldb) repl
1> func twice(x : Int) -> Int {
2.   return x * 2
3. }
4> twice(3)
$R1: (Int) = 6
5>
```

Reading a Stopped App

Questions to ask



The screenshot shows the Xcode interface with a stopped application. The left sidebar displays two threads: Thread 1 (com.apple.main-thread) and Thread 2 (hello). Thread 2 is currently selected, showing a stack trace:

- 0 Swift._getOptionalValue <A>(Sw...
- 1 DispatchTest.dispatch_onto_qu...
- 2 reabstraction thunk helper from...
- 3 _dispatch_call_block_and_release
- 4 _dispatch_client_callout
- 5 _dispatch_queue_drain

The main editor window shows the source code for `main.swift`:

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

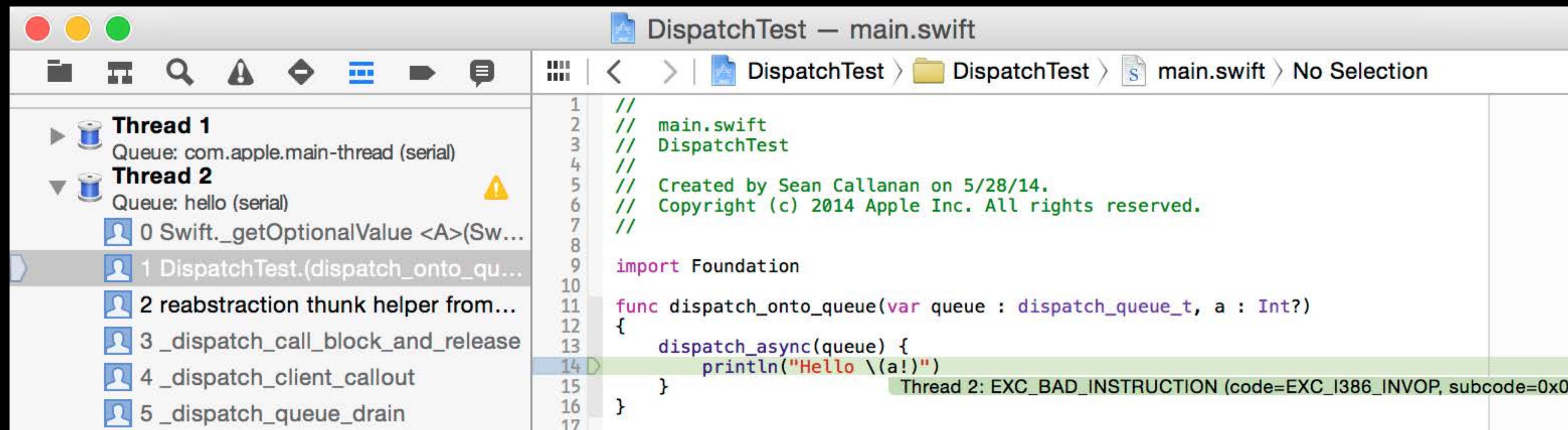
import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t, a : Int?) {
    dispatch_async(queue) {
        println("Hello \(a!)")
    }
}
```

A tooltip for the final line of the stack trace indicates a thread error: "Thread 2: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)".

Reading a Stopped App

Questions to ask



The screenshot shows the Xcode interface with a stopped application. The top navigation bar displays "DispatchTest — main.swift". The left sidebar lists two threads: "Thread 1" (com.apple.main-thread) and "Thread 2" (hello). Thread 2 has a yellow warning icon. The stack trace for Thread 2 is visible, showing frames from Swift._getOptionalValue down to _dispatch_queue_drain. The main editor area shows the source code for main.swift, which includes imports for Foundation and a function named dispatch_onto_queue. The line 14, which contains the println statement, is highlighted in green and shows the error message "Thread 2: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)".

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

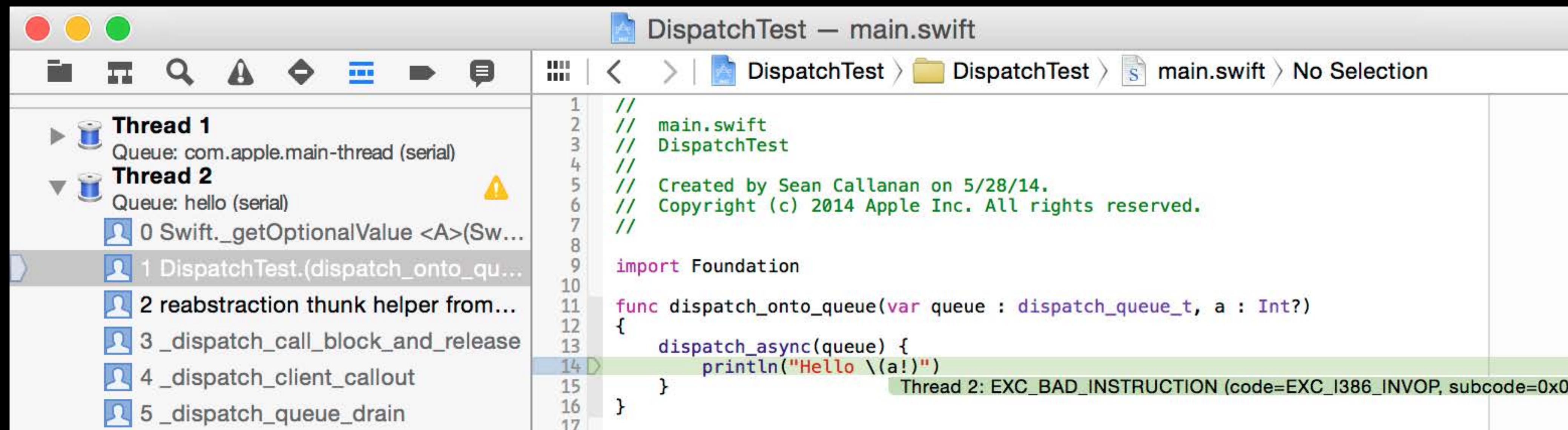
import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t, a : Int?) {
    dispatch_async(queue) {
        println("Hello \(a!)")
    }
}
```

What is the stop reason?

Reading a Stopped App

Questions to ask



The screenshot shows the Xcode interface with a stopped application. The left sidebar displays two threads: Thread 1 (com.apple.main-thread) and Thread 2 (hello). Thread 2 is expanded, showing a stack trace with frames 0 through 5. Frame 14 is highlighted, corresponding to the code in the main.swift file on the right.

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t, a : Int?) {
    dispatch_async(queue) {
        println("Hello \(a!)" Thread 2: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
    }
}
```

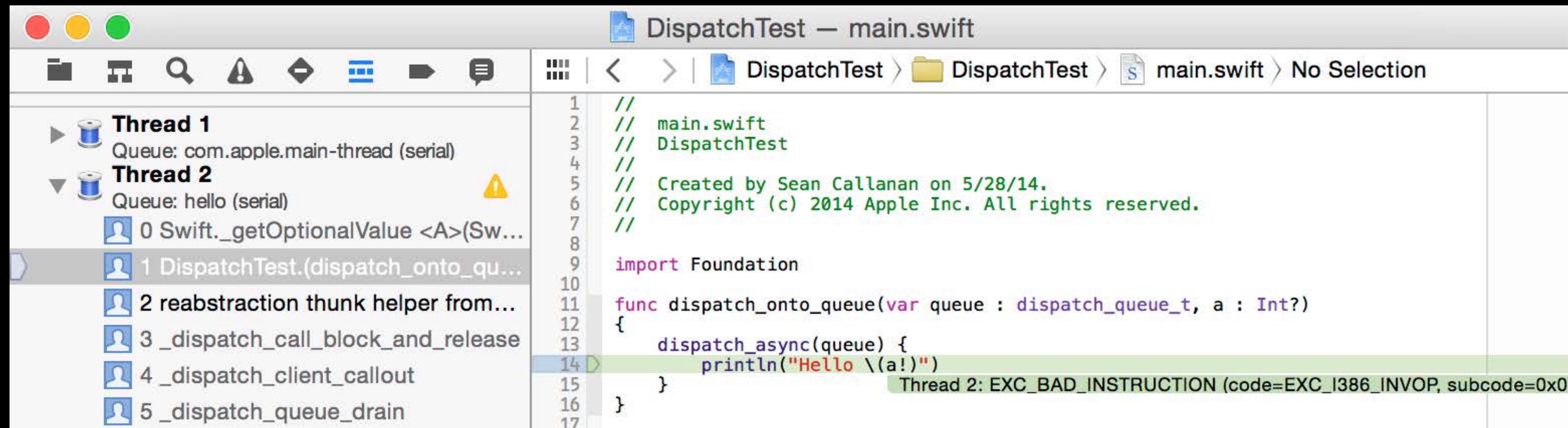
What is the stop reason?

How did this code get called?

- Look up the stack
- Find where things went bad

Reading a Stopped App

Questions to ask



The screenshot shows the Xcode interface with a stopped application. The top navigation bar displays "DispatchTest — main.swift". The left sidebar shows two threads: "Thread 1" on the com.apple.main-thread (serial) and "Thread 2" on the hello (serial) queue. Thread 2 has a yellow warning icon. The stack trace for Thread 2 is visible, showing frames from Swift._getOptionalValue, DispatchTest.dispatch_onto_queue, reabstraction thunk helper, _dispatch_call_block_and_release, _dispatch_client_callout, and _dispatch_queue_drain. The main editor area shows the source code for main.swift:

```
// main.swift
// DispatchTest
//
// Created by Sean Callanan on 5/28/14.
// Copyright (c) 2014 Apple Inc. All rights reserved.

import Foundation

func dispatch_onto_queue(var queue : dispatch_queue_t, a : Int?) {
    dispatch_async(queue) {
        println("Hello \(a!)")
    }
}
```

A tooltip for the last line of the stack trace indicates "Thread 2: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)".

What is the stop reason?

How did this code get called?

- Look up the stack
- Find where things went bad

What were the failure conditions?

- Find the problematic function
- Inspect variables to find bad data

Reading a Stopped App

Example one: The basics

What is the stop reason?

```
(lldb) t i |thread info |
* thread #1:
  tid = 0x91a7fa,
  0x100002208 accessing-nil`Swift._getOptionalValue <A>(Swift.Optional<A>) ->
A + 696 at accessing-nil.swift:0,
  queue = 'com.apple.main-thread',
  stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
```

Reading a Stopped App

Example one: The basics

What is the stop reason?

```
(lldb) t i |thread info | We're in standard library code
* thread #1:
  tid = 0x91a7fa,
  0x100002208 accessing-nil`Swift._getOptionalValue <A>(Swift.Optional<A>) ->
  A + 696 at accessing-nil.swift:0,
  queue = 'com.apple.main-thread',
  stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
```



Reading a Stopped App

Example one: The basics

What is the stop reason?

```
(lldb) t i |thread info | We're in standard library code
* thread #1:
  tid = 0x91a7fa,
  0x100002208 accessing-nil`Swift._getOptionalValue <A>(Swift.Optional<A>) ->
  A + 696 at accessing-nil.swift:0,
  queue = 'com.apple.main-thread',
  stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
```

Reading a Stopped App

Example one: The basics

What is the stop reason?

```
(lldb) t i |thread info | We're in standard library code
* thread #1:
  tid = 0x91a7fa,
  0x100002208 accessing-nil`Swift._getOptionalValue <A>(Swift.Optional<A>) ->
  A + 696 at accessing-nil.swift:0,
  queue = 'com.apple.main-thread',
  stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
```

Bad instructions are typically used in assertions

Reading a Stopped App

Example one: The basics

What is the stop reason?

```
(lldb) t i |thread info | We're in standard library code
* thread #1:
  tid = 0x91a7fa,
  0x100002208 accessing-nil`Swift._getOptionalValue <A>(Swift.Optional<A>) ->
  A + 696 at accessing-nil.swift:0,
  queue = 'com.apple.main-thread',
  stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
```

Bad instructions are typically used in assertions

This means that an assertion failed in the standard library!

Reading a Stopped App

Example one: The basics

How did this code get called?

```
(lldb) bt      |thread backtrace|  
...  
* frame #0: 0x100002208 accessing-nil`Swift._getOptionalValue  
<A>(Swift.Optional<A>) -> A + 696 at accessing-nil.swift:0  
    frame #1: 0x100001b02 accessing-nil`top_level_code + 114 at accessing-  
nil.swift:6  
frame #2: 0x100001c8a accessing-nil`main + 42 at accessing-nil.swift:0  
frame #3: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example one: The basics

How did this code get called?

```
(lldb) bt      |thread backtrace|  
...  
* frame #0: 0x100002208 accessing-nil`Swift._getOptionalValue  
<A>(Swift.Optional<A>) -> A + 696 at accessing-nil.swift:0  
    frame #1: 0x100001b02 accessing-nil`top_level_code + 114 at accessing-  
nil.swift:6  
frame #2: 0x100001c8a accessing-nil`main + 42 at accessing-nil.swift:0  
frame #3: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example one: The basics

How did this code get called?

```
(lldb) bt      |thread backtrace|  
...  
* frame #0: 0x100002208 accessing-nil`Swift._getOptionalValue  
<A>(Swift.Optional<A>) -> A + 696 at accessing-nil.swift:0  
    frame #1: 0x100001b02 accessing-nil`top_level_code + 114 at accessing-  
nil.swift:6  
    frame #2: 0x100001c8a accessing-nil`main + 42 at accessing-nil.swift:0  
    frame #3: 0x7fff873995c9 libdyld.dylib`start + 1
```

Our code called directly into the standard library

Reading a Stopped App

Example one: The basics

What were the failure conditions?

```
(lldb) f 1      |frame select 1|  
frame #1: 0x100001b02 accessing-nil`top_level_code + 114 at accessing-  
nil.swift:6  
3 }  
4  
5 var foo : AnyObject? = ReturnsNil()  
-> 6 var bar : AnyObject = foo!
```

Reading a Stopped App

Example one: The basics

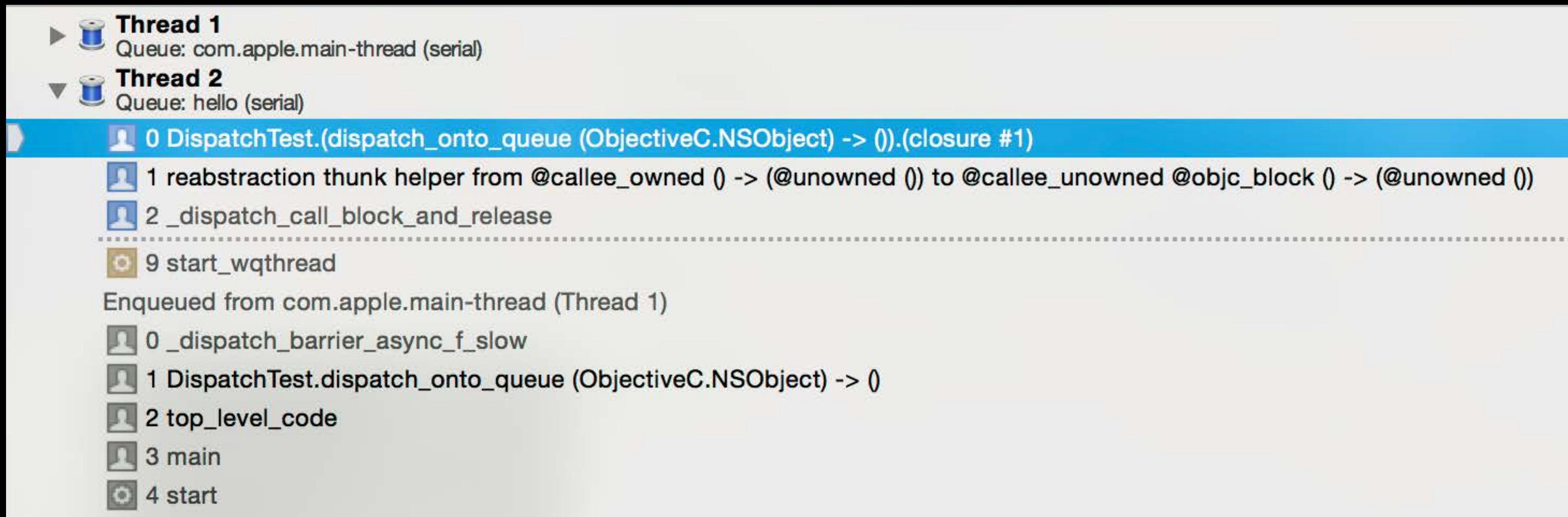
What were the failure conditions?

```
(lldb) f 1      |frame select 1|  
frame #1: 0x100001b02 accessing-nil`top_level_code + 114 at accessing-  
nil.swift:6  
3 }  
4  
5 var foo : AnyObject? = ReturnsNil()  
-> 6 var bar : AnyObject = foo!  
(lldb) p foo      |expression foo|  
(AnyObject?) $R1 = None
```

Reading a Stopped App

Stacks and libdispatch

If you're using Xcode, then you can now see where blocks got dispatched!



The screenshot shows the Xcode Call Graph tool interface. It displays two threads:

- Thread 1**: Queue: com.apple.main-thread (serial)
- Thread 2**: Queue: hello (serial)

The call stack for Thread 2 is highlighted with a blue bar and shows the following frames:

- 0 DispatchTest.(dispatch_onto_queue (ObjectiveC.NSObject) -> ()).(closure #1)
- 1 reabstraction thunk helper from @callee_owned () -> (@unowned ()) to @callee_unowned @objc_block () -> (@unowned ())
- 2 _dispatch_call_block_and_release
- 9 start_wqthread

Below the stack, it says "Enqueued from com.apple.main-thread (Thread 1)".

The full stack trace for Thread 2 is as follows:

- 0 _dispatch_barrier_async_f_slow
- 1 DispatchTest.dispatch_onto_queue (ObjectiveC.NSObject) -> ()
- 2 top_level_code
- 3 main
- 4 start

Reading a Stopped App

Example two: More Swift assertions

What is the stop reason? How did the code get called?

(lldb) bt

```
* ... stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
 * frame #0: 0x10003cb57
libswift_stdlib_core.dylib`Swift.Array.subscript.getter (Swift.Int) -> A +
167
    frame #1: 0x100001047 out-of-bounds`main.FindElement <A>($return_value=T?
at 0x00007fff5fbfffb18, array=T[] at 0x00007fff5fbfffb40,
matches=<unavailable>) -> Swift.Bool) -> Swift.Optional<A> + 679 at out-of-
bounds.swift:3
    frame #2: 0x100000d9a out-of-bounds`top_level_code + 362 at out-of-
bounds.swift:11
    frame #3: 0x10000136a out-of-bounds`main + 42 at out-of-bounds.swift:0
frame #4: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example two: More Swift assertions

What is the stop reason? How did the code get called?

(lldb) bt

```
* ... stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
* frame #0: 0x10003cb57
libswift_stdlib_core.dylib`Swift.Array.subscript.getter (Swift.Int) -> A +
167
    frame #1: 0x100001047 out-of-bounds`main.FindElement <A>($return_value=T?
at 0x00007fff5fbfffb18, array=T[] at 0x00007fff5fbfffb40,
matches=<unavailable>) -> Swift.Bool) -> Swift.Optional<A> + 679 at out-of-
bounds.swift:3
    frame #2: 0x100000d9a out-of-bounds`top_level_code + 362 at out-of-
bounds.swift:11
    frame #3: 0x10000136a out-of-bounds`main + 42 at out-of-bounds.swift:0
frame #4: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example two: More Swift assertions

What is the stop reason? How did the code get called?

(lldb) bt

```
* ... stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
* frame #0: 0x10003cb57
libswift_stdlib_core.dylib`Swift.Array.subscript.getter (Swift.Int) -> A +
167
    frame #1: 0x100001047 out-of-bounds`main.FindElement <A>($return_value=T?
at 0x00007fff5fbfffb18, array=T[] at 0x00007fff5fbfffb40,
matches=<unavailable>) -> Swift.Bool) -> Swift.Optional<A> + 679 at out-of-
bounds.swift:3
    frame #2: 0x100000d9a out-of-bounds`top_level_code + 362 at out-of-
bounds.swift:11
    frame #3: 0x10000136a out-of-bounds`main + 42 at out-of-bounds.swift:0
frame #4: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example two: More Swift assertions

What is the stop reason? How did the code get called?

(lldb) bt

```
* ... stop reason = EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
* frame #0: 0x10003cb57
libswift_stdlib_core.dylib`Swift.Array.subscript.getter (Swift.Int) -> A +
167
    frame #1: 0x100001047 out-of-bounds`main.FindElement <A>($return_value=T?
at 0x00007fff5fbfffb18, array=T[] at 0x00007fff5fbfffb40,
matches=<unavailable>) -> Swift.Bool) -> Swift.Optional<A> + 679 at out-of-
bounds.swift:3
    frame #2: 0x100000d9a out-of-bounds`top_level_code + 362 at out-of-
bounds.swift:11
    frame #3: 0x10000136a out-of-bounds`main + 42 at out-of-bounds.swift:0
frame #4: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example two: More Swift assertions

What were the failure conditions?

(lldb) f 1

```
1 func FindElement<T>(var array : Array<T>, var match : T -> Bool) -> T? {  
2     for index in 0...array.count {  
-> 3         if (match(array[index])) {  
4             return array[index]  
5         }  
6     }  
7     return nil
```

Reading a Stopped App

Example two: More Swift assertions

What were the failure conditions?

(lldb) f 1

```
1 func FindElement<T>(var array : Array<T>, var match : T -> Bool) -> T? {  
2     for index in 0...array.count {  
-> 3         if (match(array[index])) {  
4             return array[index]  
5         }  
6     }  
7     return nil
```

(lldb) p index

(RangeGenerator<Int>) \$R1 = (startIndex = 4, endIndex = 4)

Reading a Stopped App

Example two: More Swift assertions

What were the failure conditions?

(lldb) f 1

```
1 func FindElement<T>(var array : Array<T>, var match : T -> Bool) -> T? {  
2     for index in 0...array.count {  
-> 3         if (match(array[index])) {  
4             return array[index]  
5         }  
6     }  
7     return nil
```

(lldb) p index

(RangeGenerator<Int>) \$R1 = (startIndex = 4, endIndex = 4)

(lldb) p array.count

(Int) \$R2 = 3

Reading a Stopped App

Example two: More Swift assertions

What were the failure conditions?

(lldb) f 1

```
1 func FindElement<T>(var array : Array<T>, var match : T -> Bool) -> T? {  
2     for index in 0...array.count { ← Use ... to avoid the last index  
-> 3         if (match(array[index])) { Or just use for element in array  
4             return array[index]  
5         }  
6     }  
7     return nil
```

(lldb) p index

(RangeGenerator<Int>) \$R1 = (startIndex = 4, endIndex = 4)

(lldb) p array.count

(Int) \$R2 = 3

Reading a Stopped App

Example three: Exceptions in Objective-C

(lldb) bt

```
* thread #1: tid = (...), 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10, queue = 'com.apple.main-thread', stop reason = signal SIGABRT
* frame #0: 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10
  frame #1: 0x7fff876699bb libsystem_pthread.dylib`pthread_kill + 90
  frame #2: 0x7fff876345ff libsystem_c.dylib`abort + 129
  frame #3: 0x7fff95203b31 libc++abi.dylib`abort_message + 257
  frame #4: 0x7fff9522b9d1 libc++abi.dylib`default_terminate_handler() + 267
  frame #5: 0x7fff8f49350d libobjc.A.dylib`_objc_terminate() + 103
  frame #6: 0x7fff952290a1 libc++abi.dylib`std::__terminate(void (*)()) + 8
  frame #7: 0x7fff95228b2c libc++abi.dylib`__cxa_throw + 121
  frame #8: 0x7fff8f48f6a7 libobjc.A.dylib`objc_exception_throw + 341
  frame #9: 0x7fff93616cce CoreFoundation`-[__NSArrayI objectAtIndex:] + 190
  frame #10: 0x100001baf nsarray`nsarray.FindElement <A>(...)
  frame #11: 0x100001855 nsarray`top_level_code + 693 at nsarray.swift:15
  frame #12: 0x10000225a nsarray`main + 42 at nsarray.swift:0
  frame #13: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example three: Exceptions in Objective-C

(lldb) bt

```
* thread #1: tid = (...), 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10, queue = 'com.apple.main-thread', stop reason = signal SIGABRT
* frame #0: 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10
frame #1: 0x7fff876699bb libsystem_pthread.dylib`pthread_kill + 90
frame #2: 0x7fff876345ff libsystem_c.dylib`abort + 129
frame #3: 0x7fff95203b31 libc++abi.dylib`abort_message + 257
frame #4: 0x7fff9522b9d1 libc++abi.dylib`default_terminate_handler() + 267
frame #5: 0x7fff8f49350d libobjc.A.dylib`_objc_terminate() + 103
frame #6: 0x7fff952290a1 libc++abi.dylib`std::__terminate(void (*)()) + 8
frame #7: 0x7fff95228b2c libc++abi.dylib`__cxa_throw + 121
frame #8: 0x7fff8f48f6a7 libobjc.A.dylib`objc_exception_throw + 341
frame #9: 0x7fff93616cce CoreFoundation`-[__NSArrayI objectAtIndex:] + 190
frame #10: 0x100001baf nsarray`nsarray.FindElement <A>(...)
frame #11: 0x100001855 nsarray`top_level_code + 693 at nsarray.swift:15
frame #12: 0x10000225a nsarray`main + 42 at nsarray.swift:0
frame #13: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example three: Exceptions in Objective-C

(lldb) bt

```
* thread #1: tid = (...), 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10, queue = 'com.apple.main-thread', stop reason = signal SIGABRT
* frame #0: 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10
frame #1: 0x7fff876699bb libsystem_pthread.dylib`pthread_kill + 90
frame #2: 0x7fff876345ff libsystem_c.dylib`abort + 129
frame #3: 0x7fff95203b31 libc++abi.dylib`abort_message + 257
frame #4: 0x7fff9522b9d1 libc++abi.dylib`default_terminate_handler() + 267
frame #5: 0x7fff8f49350d libobjc.A.dylib`_objc_terminate() + 103
frame #6: 0x7fff952290a1 libc++abi.dylib`std::__terminate(void (*)()) + 8
frame #7: 0x7fff95228b2c libc++abi.dylib`__cxa_throw + 121
frame #8: 0x7fff8f48f6a7 libobjc.A.dylib`objc_exception_throw + 341
frame #9: 0x7fff93616cce CoreFoundation`-[__NSArrayI objectAtIndex:] + 190
frame #10: 0x100001baf nsarray`nsarray.FindElement <A>(...)
frame #11: 0x100001855 nsarray`top_level_code + 693 at nsarray.swift:15
frame #12: 0x10000225a nsarray`main + 42 at nsarray.swift:0
frame #13: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example three: Exceptions in Objective-C

(lldb) bt

```
* thread #1: tid = (...), 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10, queue = 'com.apple.main-thread', stop reason = signal SIGABRT
* frame #0: 0x7fff8995337a libsystem_kernel.dylib`__pthread_kill + 10
frame #1: 0x7fff876699bb libsystem_pthread.dylib`pthread_kill + 90
frame #2: 0x7fff876345ff libsystem_c.dylib`abort + 129
frame #3: 0x7fff95203b31 libc++abi.dylib`abort_message + 257
frame #4: 0x7fff9522b9d1 libc++abi.dylib`default_terminate_handler() + 267
frame #5: 0x7fff8f49350d libobjc.A.dylib`_objc_terminate() + 103
frame #6: 0x7fff952290a1 libc++abi.dylib`std::__terminate(void (*)()) + 8
frame #7: 0x7fff95228b2c libc++abi.dylib`__cxa_throw + 121
frame #8: 0x7fff8f48f6a7 libobjc.A.dylib`objc_exception_throw + 341
frame #9: 0x7fff93616cce CoreFoundation`-[__NSArrayI objectAtIndex:] + 190
frame #10: 0x100001baf nsarray`nsarray.FindElement <A>(...)
frame #11: 0x100001855 nsarray`top_level_code + 693 at nsarray.swift:15
frame #12: 0x10000225a nsarray`main + 42 at nsarray.swift:0
frame #13: 0x7fff873995c9 libdyld.dylib`start + 1
```

Reading a Stopped App

Example three: Exceptions in Objective-C

(lldb) f 10

```
2
3 func FindElement<T>(var array : NSArray, var matches : T -> Bool) -> T? {
4     for index in 0...array.count {
->5         if let elementAsT = array[index] as? T {
6             if matches(elementAsT) {
7                 return elementAsT
8 }
```

Reading a Stopped App

Example three: Exceptions in Objective-C

(lldb) f 10

```
2
3 func FindElement<T>(var array : NSArray, var matches : T -> Bool) -> T? {
4     for index in 0...array.count {
->5         if let elementAsT = array[index] as? T {
6             if matches(elementAsT) {
7                 return elementAsT
8 }
```

Reading a Stopped App

Example four: The more things change...

(lldb) bt

```
* thread #1: tid = (...), 0x1001ef7cd libswift_stdlib_core.dylib`initializeWithCopy value
witness for Swift.Character + 13, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=1, address=0x0)
* frame #0: 0x1001ef7cd libswift_stdlib_core.dylib`
    initializeWithCopy value witness for Swift.Character + 13
frame #1: 0x100001e9a null-pointer`
    Swift.UnsafePointer.memory.getter : A + 26
frame #2: 0x100001966 null-pointer`
    Swift.UnsafePointer.subscript.getter (Swift.Int) -> A + 54
frame #3: 0x10000167b null-pointer`
    top_level_code + 299 at null-pointer.swift:4
frame #4: 0x10000171a
    null-pointer`main + 42 at null-pointer.swift:0
```

Reading a Stopped App

Example four: The more things change...

(lldb) bt

```
* thread #1: tid = (...), 0x1001ef7cd libswift_stdlib_core.dylib`initializeWithCopy value
witness for Swift.Character + 13, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=1, address=0x0)
* frame #0: 0x1001ef7cd libswift_stdlib_core.dylib`
    initializeWithCopy value witness for Swift.Character + 13
frame #1: 0x100001e9a null-pointer`
    Swift.UnsafePointer.memory.getter : A + 26
frame #2: 0x100001966 null-pointer`
    Swift.UnsafePointer.subscript.getter (Swift.Int) -> A + 54
frame #3: 0x10000167b null-pointer`
    top_level_code + 299 at null-pointer.swift:4
frame #4: 0x10000171a
    null-pointer`main + 42 at null-pointer.swift:0
```

Reading a Stopped App

Example four: The more things change...

(lldb) bt

```
* thread #1: tid = (...), 0x1001ef7cd libswift_stdlib_core.dylib`initializeWithCopy value
witness for Swift.Character + 13, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=1, address=0x0)
* frame #0: 0x1001ef7cd libswift_stdlib_core.dylib`
    initializeWithCopy value witness for Swift.Character + 13
frame #1: 0x100001e9a null-pointer`
    Swift.UnsafePointer.memory.getter : A + 26
frame #2: 0x100001966 null-pointer`
    Swift.UnsafePointer.subscript.getter (Swift.Int) -> A + 54
frame #3: 0x10000167b null-pointer`
    top_level_code + 299 at null-pointer.swift:4
frame #4: 0x10000171a
    null-pointer`main + 42 at null-pointer.swift:0
```

Reading a Stopped App

Example four: The more things change...

```
(lldb) f 3  
frame #3: 0x000000010000167b null-pointer`top_level_code + 299 at null-  
pointer.swift:4  
1 import Foundation  
2 let empty_stuff =  
    NSData(contentsOfURL:NSURL(string:"http://non.existent.web.site.com"))  
3 let chars = UnsafePointer<Character>(empty_stuff.bytes)  
->4 let char = chars[0]  
5 println(char)
```

Reading a Stopped App

Example four: The more things change...

```
(lldb) f 3
```

```
frame #3: 0x000000010000167b null-pointer`top_level_code + 299 at null-
pointer.swift:4
1 import Foundation
2 let empty_stuff =
    NSData(contentsOfURL:NSURL(string:"http://non.existent.web.site.com"))
3 let chars = UnsafePointer<Character>(empty_stuff.bytes)
->4 let char = chars[0]
5 println(char)
```

```
(lldb) p chars
```

```
(UnsafePointer<Character>) $R1 = (value = Builtin.RawPointer =
0x0000000000000000)
```

Reading a Stopped App

Example four: The more things change...

(lldb) f 3

```
frame #3: 0x000000010000167b null-pointer`top_level_code + 299 at null-
pointer.swift:4
1 import Foundation
2 let empty_stuff =
    NSData(contentsOfURL:NSURL(string:"http://non.existent.web.site.com"))
3 let chars = UnsafePointer<Character>(empty_stuff.bytes)
->4 let char = chars[0]
5 println(char)
```

(lldb) p chars

```
(UnsafePointer<Character>) $R1 = (value = Builtin.RawPointer =
0x0000000000000000)
```

Swift code is safe by default, but be careful with APIs that use unsafe pointers!

Reading a Stopped App

Summing up

The stop reason tells you what happened

`EXC_BAD_INSTRUCTION`

Assertion

`SIGABRT`

Exception (usually Objective-C)

`EXC_BAD_ACCESS`

Memory error

The stack tells you how it happened

The expression command helps you inspect variables

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

REPL-enabled debugging workflows

- Validating existing code
- Trying out new code

Summing up

Stopping an App at the Right Time

Breakpoint concepts

Stopping an App at the Right Time

Breakpoint concepts

Use a breakpoint if any of these are true

- The problem isn't a crash
- The cause of a crash isn't on the stack
- You want to step through failing code

Stopping an App at the Right Time

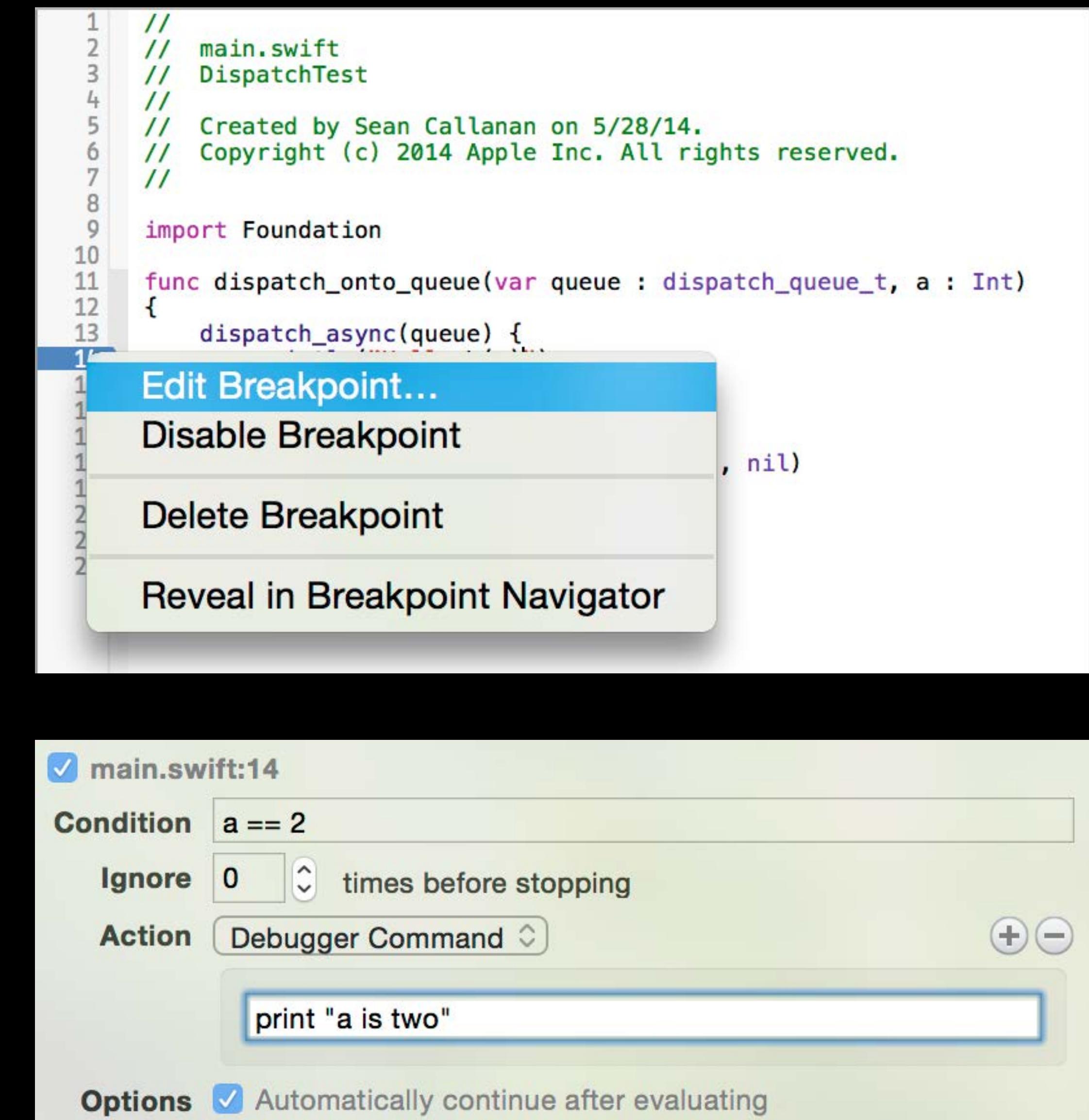
Breakpoint concepts

Use a breakpoint if any of these are true

- The problem isn't a crash
- The cause of a crash isn't on the stack
- You want to step through failing code

Important parts of a breakpoint

- Specification of where to stop
- Locations matching your specification
- Condition (optional)—Stop only if true
- Actions (optional) to perform when hit



Stopping an App at the Right Time

Example one: File-and-line breakpoints

```
1 var overdraftFee : UInt = 10
2 class Account {
3     var valueInCents : Int = 0
4     func withdraw(var amountInCents:UInt) {
5         if (Int(amountInCents) > self.valueInCents) {
6             valueInCents -= Int(overdraftFee)
7         }
8         valueInCents -= Int(amountInCents)
9     }
10    func deposit(var amountInCents:UInt) {
11        valueInCents += Int(amountInCents)
12    }
13 }
```

Stopping an App at the Right Time

Example one: File-and-line breakpoints

```
1 var overdraftFee : UInt = 10
2 class Account {
3     var valueInCents : Int = 0
4     func withdraw(var amountInCents:UInt) {
5         if (Int(amountInCents) > self.valueInCents) {
6             valueInCents -= Int(overdraftFee)
7         }
8         valueInCents -= Int(amountInCents)
9     }
10    func deposit(var amountInCents:UInt) {
11        valueInCents += Int(amountInCents)
12    }
13 }
```

Stopping an App at the Right Time

Example one: File-and-line breakpoints

```
1 var overdraftFee : UInt = 10
2 class Account {
3     var valueInCents : Int = 0
4     func withdraw(var amountInCents:UInt) {
5         if (Int(amountInCents) > self.valueInCents) {
6             valueInCents -= Int(overdraftFee)
7         }
8         valueInCents -= Int(amountInCents)
9     }
10    func deposit(var amountInCents:UInt) {
11        valueInCents += Int(amountInCents)
12    }
13 }
```

(lldb) b Account.swift:6

```
breakpoint set --file Account.swift --line 6
```

Stopping an App at the Right Time

Example one: File-and-line breakpoints

```
(lldb) br l |breakpoint list|
```

Current breakpoints:

```
1: file = 'file-and-line.swift', line = 6, locations = 1
  1.1: where = file-and-line`main.Account.withdraw (main.Account)(Swift.UInt)
        -> () + 79 at file-and-line.swift:6, address = file-and-line
          [0x000000010000123f], unresolved, hit count = 0
```

Stopping an App at the Right Time

Example one: File-and-line breakpoints

```
Specification  
(lldb) br l      |breakpoint list|  
Current breakpoints:  
1: file = 'file-and-line.swift', line = 6, locations = 1  
  1.1: where = file-and-line`main.Account.withdraw (main.Account)(Swift.UInt)  
        -> () + 79 at file-and-line.swift:6, address = file-and-line  
        [0x000000010000123f], unresolved, hit count = 0
```

Stopping an App at the Right Time

Example one: File-and-line breakpoints

```
Specification  
(lldb) br l      |breakpoint list|  
Current breakpoints:  
1: file = 'file-and-line.swift', line = 6, locations = 1  
  1.1: where = file-and-line`main.Account.withdraw (main.Account)(Swift.UInt)  
        -> () + 79 at file-and-line.swift:6, address = file-and-line  
        [0x000000010000123f], unresolved, hit count = 0  
Location
```

Stopping an App at the Right Time

Example one: File-and-line breakpoints

```
Specification  
(lldb) br l      |breakpoint list|  
Current breakpoints:  
1: file = 'file-and-line.swift', line = 6, locations = 1  
  1.1: where = file-and-line`main.Account.withdraw (main.Account)(Swift.UInt)  
        -> () + 79 at file-and-line.swift:6, address = file-and-line  
        [0x000000010000123f], unresolved, hit count = 0  
Location
```

If you want to stop hitting the breakpoint for a while, you can disable the breakpoint

```
(lldb) br dis 1    |breakpoint disable 1|
```

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
1 func timestwo(var a : Int) -> Int {  
2     return a * 2  
3 }  
4 func timestwo(var a : Double) -> Double {  
5     return a * 2.0  
6 }  
7 func timestwo(var a : String) -> String {  
8     return a + a  
9 }
```

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
1 func timestwo(var a : Int) -> Int {  
2     return a * 2  
3 }  
4 func timestwo(var a : Double) -> Double {  
5     return a * 2.0  
6 }  
7 func timestwo(var a : String) -> String {  
8     return a + a  
9 }
```

(lldb) **b timestwo**

Breakpoint 1: 3 locations.

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br l
```

Current breakpoints:

```
1: name = 'timestwo', locations = 3
  1.1: where = symbolic`symbolic.timestwo (Swift.Int) -> Swift.Int + 8 at
symbolic.swift:2, address = symbolic[0x0000000100001548], unresolved, hit
count = 0
  1.2: where = symbolic`symbolic.timestwo (Swift.Double) -> Swift.Double + 24
at symbolic.swift:5, address = symbolic[0x0000000100001578], unresolved, hit
count = 0
  1.3: where = symbolic`symbolic.timestwo (Swift.String) -> Swift.String + 20
at symbolic.swift:8, address = symbolic[0x0000000100001594], unresolved, hit
count = 0
```

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br l
```

Current breakpoints:

```
1: name = 'timestwo', locations = 3
   1.1: where = symbolic`symbolic.timestwo (Swift.Int) -> Swift.Int + 8 at
symbolic.swift:2, address = symbolic[0x0000000100001548], unresolved, hit
count = 0
   1.2: where = symbolic`symbolic.timestwo (Swift.Double) -> Swift.Double + 24
at symbolic.swift:5, address = symbolic[0x0000000100001578], unresolved, hit
count = 0
   1.3: where = symbolic`symbolic.timestwo (Swift.String) -> Swift.String + 20
at symbolic.swift:8, address = symbolic[0x0000000100001594], unresolved, hit
count = 0
```

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br l
```

Current breakpoints:

```
1: name = 'timestwo', locations = 3
   1.1: where = symbolic`symbolic.timestwo (Swift.Int) -> Swift.Int + 8 at
symbolic.swift:2, address = symbolic[0x0000000100001548], unresolved, hit
count = 0
   1.2: where = symbolic`symbolic.timestwo (Swift.Double) -> Swift.Double + 24
at symbolic.swift:5, address = symbolic[0x0000000100001578], unresolved, hit
count = 0
   1.3: where = symbolic`symbolic.timestwo (Swift.String) -> Swift.String + 20
at symbolic.swift:8, address = symbolic[0x0000000100001594], unresolved, hit
count = 0
```

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br l
```

Current breakpoints:

```
1: name = 'timestwo', locations = 3
   1.1: where = symbolic`symbolic.timestwo (Swift.Int) -> Swift.Int + 8 at
symbolic.swift:2, address = symbolic[0x0000000100001548], unresolved, hit
count = 0
   1.2: where = symbolic`symbolic.timestwo (Swift.Double) -> Swift.Double + 24
at symbolic.swift:5, address = symbolic[0x0000000100001578], unresolved, hit
count = 0
   1.3: where = symbolic`symbolic.timestwo (Swift.String) -> Swift.String + 20
at symbolic.swift:8, address = symbolic[0x0000000100001594], unresolved, hit
count = 0
```

If you only care about strings, you can disable the other locations

```
(lldb) breakpoint disable 1.1 1.2
```

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br s -r timestwo.*String  
breakpoint set --func-regex timestwo.*String
```

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br s -r timestwo.*String  
breakpoint set --func-regex timestwo.*String
```

Breakpoint 2: where = symbolic`symbolic.timestwo (Swift.String) -> Swift.String + 20 at symbolic.swift:8, address = 0x0000000100001594

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br s -r timestwo.*String  
breakpoint set --func-regex timestwo.*String
```

Breakpoint 2: where = symbolic`symbolic.timestwo (Swift.String) -> Swift.String + 20 at symbolic.swift:8, address = 0x0000000100001594

You can use regular-expression breakpoints for many types of categories

- Methods of a class:**-r Account\.\.**
- Functions in a module:**-r main\.\.**

Stopping an App at the Right Time

Example two: Symbolic breakpoints

```
(lldb) br s -r timestwo.*String  
breakpoint set --func-regex timestwo.*String
```

Breakpoint 2: where = symbolic`symbolic.timestwo (Swift.String) -> Swift.String + 20 at symbolic.swift:8, address = 0x0000000100001594

You can use regular-expression breakpoints for many types of categories:

- Methods of a class:`-r Account\.\.`
- Functions in a module:`-r main\.\.`

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

(lldb) b Account.swift:4

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

(lldb) b Account.swift:4

Where would we charge an overdraft fee?

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

(lldb) b Account.swift:4

Where would we charge an overdraft fee?

(lldb) br m -c "valueInCents < Int(amountInCents)"
breakpoint modify --condition "..."

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

(lldb) b Account.swift:4

(lldb) br co a |breakpoint command add

Enter your debugger command(s). Type 'DONE' to end.

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

(lldb) b Account.swift:4

(lldb) br co a |breakpoint command add

Enter your debugger command(s). Type 'DONE' to end.

> p valueInCents

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

(lldb) b Account.swift:4

(lldb) br co a |breakpoint command add

Enter your debugger command(s). Type 'DONE' to end.

> p valueInCents

> continue

Stopping an App at the Right Time

Example three: Smart breakpoints

```
1 class Account {  
2     var valueInCents : Int = 0  
3     func withdraw(var amountInCents:UInt) {  
4         valueInCents -= Int(amountInCents)  
5     }  
6     func deposit(var amountInCents:UInt) {  
7         valueInCents += Int(amountInCents)  
8     }  
9 }
```

(lldb) b Account.swift:4

(lldb) br co a |breakpoint command add

Enter your debugger command(s). Type 'DONE' to end.

```
> p valueInCents  
> continue  
> DONE
```

Stopping an App at the Right Time

Example three: Smart breakpoints

(lldb) run

Stopping an App at the Right Time

Example three: Smart breakpoints

```
(lldb) run
```

```
(Int) $R1 = 30
```

```
Process 9257 resuming
```

```
Command #2 'continue' continued the target.
```

```
(Int) $R2 = 15
```

```
Process 9257 resuming
```

```
Command #2 'continue' continued the target.
```

```
(Int) $R3 = 10
```

```
Process 9257 resuming
```

```
Command #2 'continue' continued the target.
```

```
(Int) $R4 = -5
```

```
Process 9257 resuming
```

```
Command #2 'continue' continued the target.
```

Stopping an App at the Right Time

Smart breakpoints in Swift and Objective-C

You must set separate conditions and actions for Objective-C and Swift locations

```
(lldb) br m —c "a[3] == 2" 3.1
```

Stopping an App at the Right Time

Smart breakpoints in Swift and Objective-C

You must set separate conditions and actions for Objective-C and Swift locations

```
(lldb) br m —c“a[3] == 2” 3.1
```

```
(lldb) br m —c“[a[3] isEqual:@2]” 3.2
```

Stopping an App at the Right Time

Smart breakpoints in Swift and Objective-C

You must set separate conditions and actions for Objective-C and Swift locations

```
(lldb) br m -c "a[3] == 2" 3.1
```

```
(lldb) br m -c "[a[3] isEqual:@2]" 3.2
```

How to tell breakpoint locations apart

Objective-C Function 1.1: where = test`**main** + 126 (...)

Objective-C Method 2.1: where = test`-[MyString initWithNSString:] + 24 (...)

Swift Function 3.1: where = test`**test.ConformsDirectly.protocol_func**
(a.ConformsDirectly)(Swift.Int) -> Swift.Int + 35 (...)

Stopping an App at the Right Time

Summing up

Breakpoints are a powerful way to stop your program

You can filter based on:

Source location	<code>--file --line</code>
Function name	<code>--name</code>
Module or class	<code>--func-regex</code>
Variable values	<code>breakpoint modify --condition</code>

You can even set automated actions so you don't have to stop!

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

REPL-enabled debugging workflows

- Validating existing code
- Trying out new code

Summing up

REPL Concepts

The REPL and the LLDB command line

REPL Concepts

The REPL and the LLDB command line

You can launch the REPL from a shell with an empty target

```
$ xcrun swift
```

REPL Concepts

The REPL and the LLDB command line

You can launch the REPL from a shell with an empty target

```
$ xcrun swift
```

You can break into the REPL whenever your program is stopped

```
(lldb) repl
```

```
1>
```

REPL Concepts

The REPL and the LLDB command line

You can launch the REPL from a shell with an empty target

```
$ xcrun swift
```

You can break into the REPL whenever your program is stopped

```
(lldb) repl
```

```
1>
```

You can break into LLDB from the REPL

```
1> :
```

```
(lldb)
```

REPL Concepts

The REPL and the LLDB command line

You can launch the REPL from a shell with an empty target

```
$ xcrun swift
```

You can break into the REPL whenever your program is stopped

```
(lldb) repl
```

```
1>
```

You can break into LLDB from the REPL

```
1> :
```

```
(lldb)
```

You can issue LLDB commands directly from the REPL

```
1> :help
```

Validating Existing Code

Example one: Array partition

```
func partition(input: Int[] ) -> (Int[], Int[]) {...}
```

Validating Existing Code

Example one: Array partition

```
func partition(input: Int[]) -> (Int[], Int[]) {...}
```

(lldb) repl

Validating Existing Code

Example one: Array partition

```
func partition(input: Int[]) -> (Int[], Int[]) {...}
```

```
(lldb) repl
```

```
1> partition([3,4,5])
```

Validating Existing Code

Example one: Array partition

```
func partition(input: Int[]) -> (Int[], Int[]) {...}  
(lldb) repl  
1> partition([3,4,5])  
$R1: (Int[], Int[]) = {  
    0 = size=1 {  
        [0] = 3  
    }  
    1 = size=2 {  
        [0] = 4  
        [1] = 5  
    }  
}
```

Validating Existing Code

Example one: Array partition

```
2> func ispartition(l: Int[], r: Int[]) -> Bool {  
3.     for le in l {  
4.         for re in r {  
5.             if le > re { return false }  
6.         }  
7.     }  
8.     return true  
9. }
```

Validating Existing Code

Example one: Array partition

```
2> func ispartition(l: Int[], r: Int[]) -> Bool {  
3.    for le in l {  
4.        for re in r {  
5.            if le > re { return false }  
6.        }  
7.    }  
8.    return true  
9. }  
  
10> ispartition(partition([6,2,1,5,3,8]))
```

Validating Existing Code

Example one: Array partition

```
2> func ispartition(l: Int[], r: Int[]) -> Bool {  
3.    for le in l {  
4.        for re in r {  
5.            if le > re { return false }  
6.        }  
7.    }  
8.    return true  
9. }  
  
10> ispartition(partition([6,2,1,5,3,8]))  
$R2: Bool = true
```

Validating Existing Code

Example two: Sorting

```
func mysort(input: Int[]) -> Int[] {...}
```

Validating Existing Code

Example two: Sorting

```
func mysort(input: Int[]) -> Int[] {...}  
(lldb) repl
```

Validating Existing Code

Example two: Sorting

```
func mysort(input: Int[]) -> Int[] {...}  
(lldb) repl  
1> mysort([5,1,4,2,3])  
$R1: (Int[]) = size=5 {  
    [0] = 1  
    [1] = 2  
    [2] = 3  
    [3] = 4  
    [4] = 5  
}
```

Validating Existing Code

Example two: Sorting

```
2> func issorted(a: Int[]) -> Bool {  
3.     var last : Int? = nil  
4.     for ae in a {  
5.         if (last && (last! > ae)) { return false }  
6.         last = ae  
7.     }  
8.     return true  
9. }
```

Validating Existing Code

Example two: Sorting

```
2> func issorted(a: Int[]) -> Bool {  
3.     var last : Int? = nil  
4.     for ae in a {  
5.         if (last && (last! > ae)) { return false }  
6.         last = ae  
7.     }  
8.     return true  
9. }  
  
10> issorted(mysort([6,2,1,5,3,8]))
```

Validating Existing Code

Example two: Sorting

```
2> func issorted(a: Int[]) -> Bool {  
3.     var last : Int? = nil  
4.     for ae in a {  
5.         if (last && (last! > ae)) { return false }  
6.         last = ae  
7.     }  
8.     return true  
9. }  
  
10> issorted(mysort([6,2,1,5,3,8]))  
$R2: Bool = true
```

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

REPL-enabled debugging workflows

- Validating existing code
- Trying out new code

Summing up

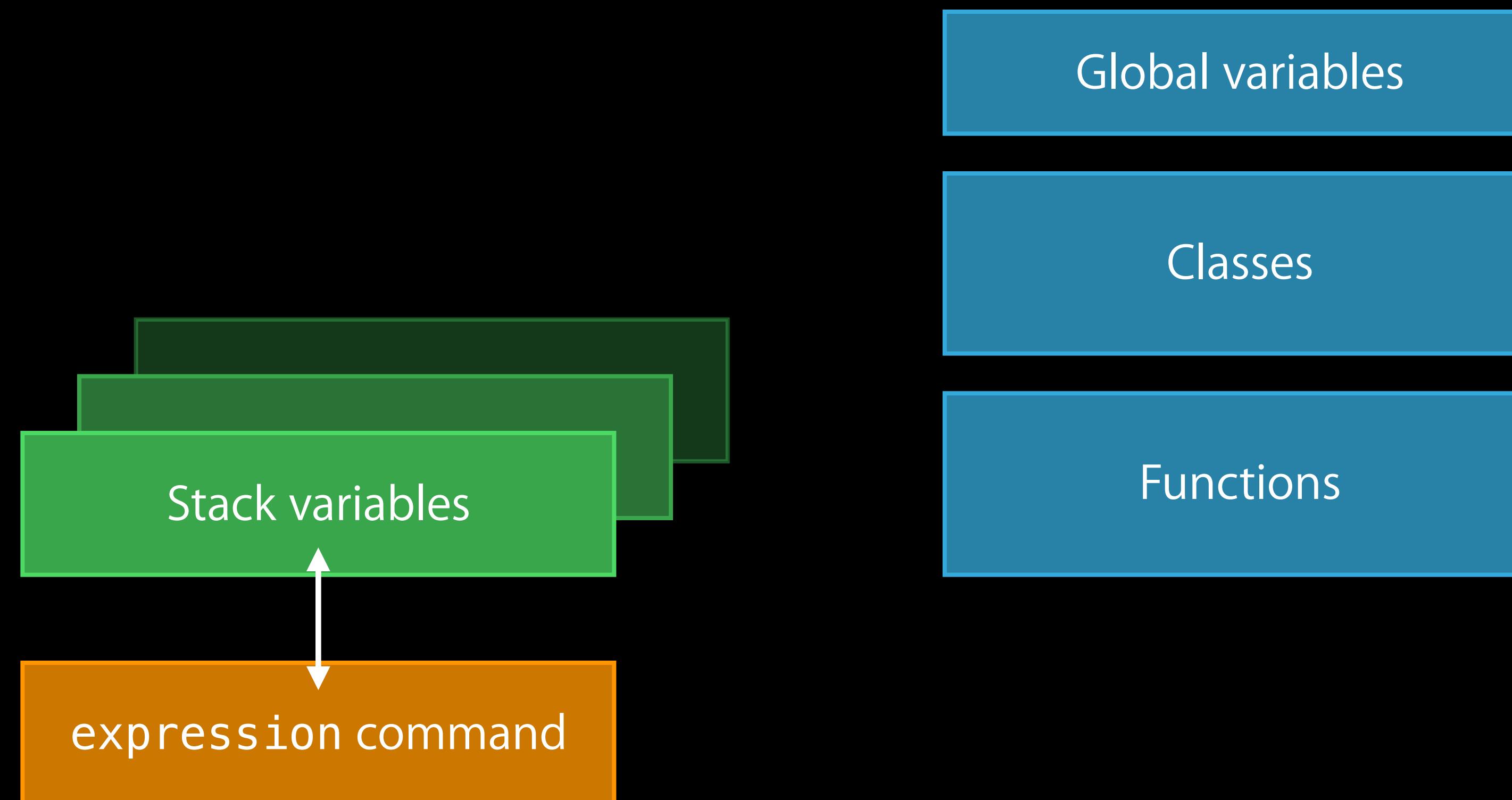
REPL Concepts

The REPL adds global code



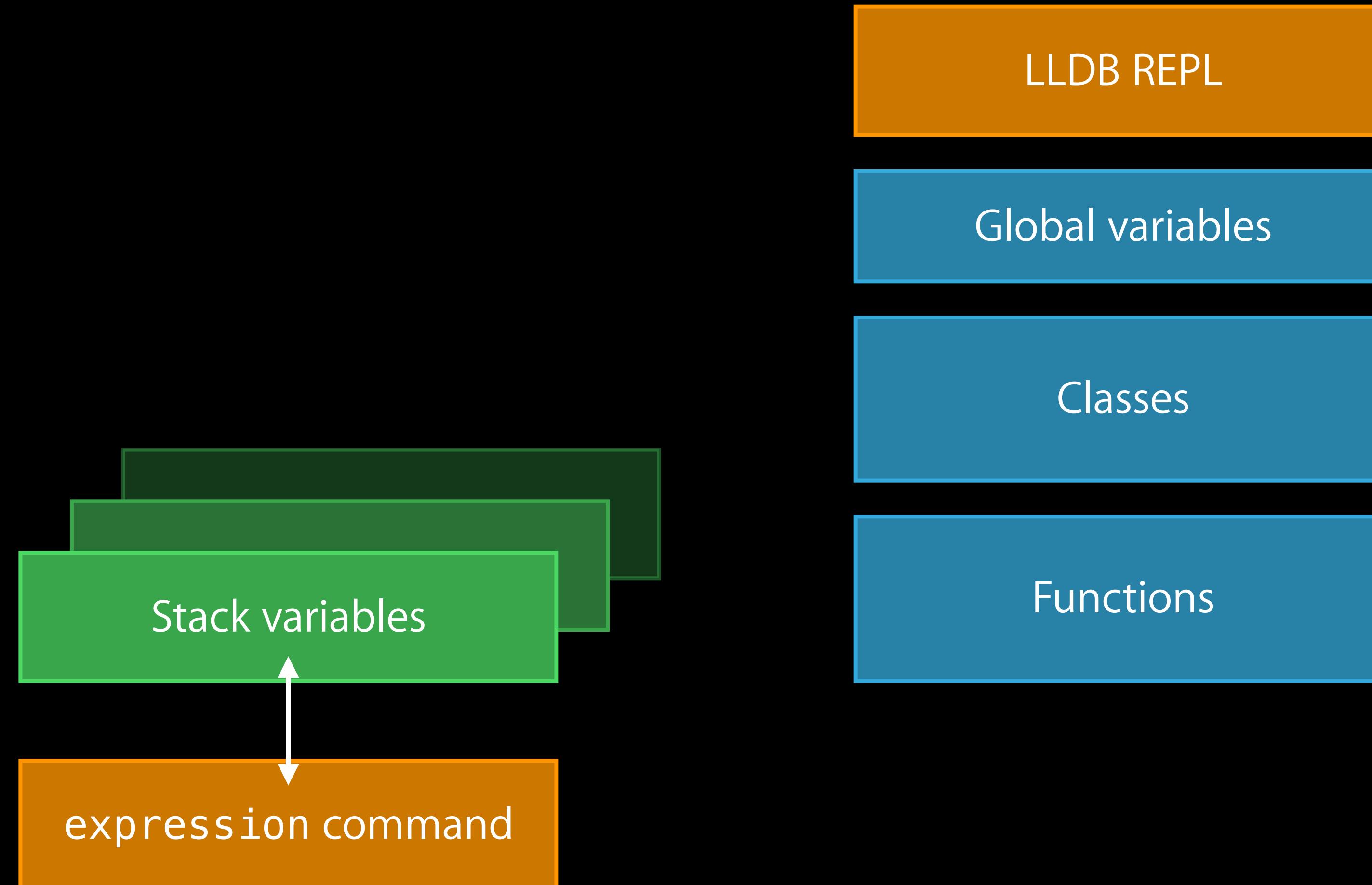
REPL Concepts

The REPL adds global code



REPL Concepts

The REPL adds global code



Trying Out New Code

Example one: Dispatching on a queue

```
import Foundation

var queue : dispatch_queue_t =
    dispatch_queue_create("my_queue", nil)
```

Trying Out New Code

Example one: Dispatching on a queue

```
import Foundation

var queue : dispatch_queue_t =
    dispatch_queue_create("my_queue", nil)

(lldb) repl
1> dispatch_sync(queue) {println("world")}
```

Trying Out New Code

Example one: Dispatching on a queue

```
import Foundation

var queue : dispatch_queue_t =
    dispatch_queue_create("my_queue", nil)
```

```
(lldb) repl
1> dispatch_sync(queue) {println("world")}
world
2>
```

Trying Out New Code

Example two: Implementing a protocol

```
protocol Doublable { func twice () -> Self }
extension Int : Doublable {
    func twice () -> Int { return 2 * self }
}
func fourtimes<T:Doublable>(a : T) -> T {
    return a.twice().twice()
}
```

Trying Out New Code

Example two: Implementing a protocol

```
protocol Doublable { func twice () -> Self }
extension Int : Doublable {
    func twice () -> Int { return 2 * self }
}
func fourtimes<T:Doublable>(a : T) -> T {
    return a.twice().twice()
}
(lldb) repl
```

Trying Out New Code

Example two: Implementing a protocol

```
protocol Doublable { func twice () -> Self }
extension Int : Doublable {
    func twice () -> Int { return 2 * self }
}
func fourtimes<T:Doublable>(a : T) -> T {
    return a.twice().twice()
}
(lldb) repl
1> extension String : Doublable {
2.   func twice() -> String { return self + self }
3. }
```

Trying Out New Code

Example two: Implementing a protocol

```
protocol Doublable { func twice () -> Self }
extension Int : Doublable {
    func twice () -> Int { return 2 * self }
}
func fourtimes<T:Doublable>(a : T) -> T {
    return a.twice().twice()
}
(lldb) repl
1> extension String : Doublable {
2.   func twice() -> String { return self + self }
3. }
4> println(fourtimes("three"))
```

Trying Out New Code

Example two: Implementing a protocol

```
protocol Doublable { func twice () -> Self }
extension Int : Doublable {
    func twice () -> Int { return 2 * self }
}
func fourtimes<T:Doublable>(a : T) -> T {
    return a.twice().twice()
}
(lldb) repl
1> extension String : Doublable {
2.   func twice() -> String { return self + self }
3. }
4> println(fourtimes("three"))
threethreethreethree
```

Overview of Today's Session

Basic debugging survival skills

- Reading a stopped app
- Stopping an app at the right time

REPL-enabled debugging workflows

- Validating existing code
- Trying out new code

Summing up

Summary

LLDB provides tools to diagnose bugs in your program

- Stop reasons and the stack tell you what happened and how
- The print command tells you why it happened
- Breakpoints stop your program when you want to stop

With the REPL you can debug your program in Swift

- Validate your existing code with ad-hoc unit tests
- Add new functionality to your program on the fly

More Information

Dave DeLong
Developer Tools Evangelist
delong@apple.com

Documentation
LLDB Quick Start Guide
<http://developer.apple.com>

Apple Developer Forums
<http://devforums.apple.com>

Related Sessions

