iOS Dev Accelerator
Week 7 Day 4

- Instruments & Stack Trace
- Objective-C Runtime
- Run Loops
- Messaging Nil

Instruments

- “Instruments is a performance, analysis, and testing tool for dynamically tracing and profiling OS X and iOS Code”
- Instruments allows to you to view data from largely different aspects of your app and view them side by side, helping you identify trends and bottlenecks.
- Instruments is a set of modules, each one a template that collects and displays different types of information, such as file IO, memory usage, CPU, etc.

Trace Document

- All the data and work done in instruments is done in a trace document.
- The trace documents displays the set of instruments you are using and the data they have collected.
- Each document will typically only hold a single session worth of data and is considered a ‘single trace’. They can be saved for future viewing.
- There are a few instruments that can even simulate user interface interactions.

Trace Document

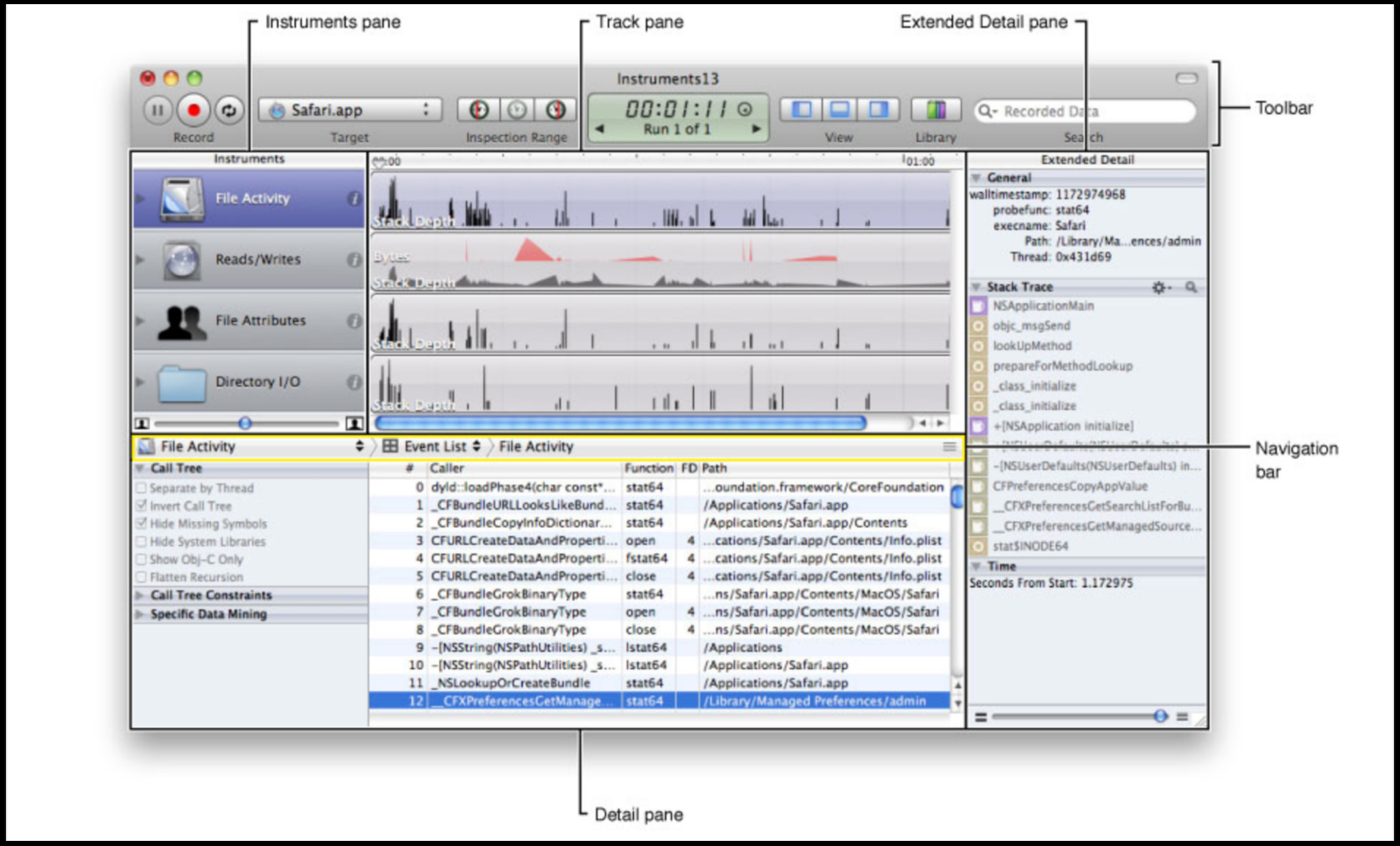


Table 2-3 Trace document toolbar controls

Control	Description
Pause/Resume button	Pauses the gathering of data during a recording. This button does not actually stop recording, it simply stops Instruments from gathering data while a recording is under way. In the track pane, pauses show up as a gap in the trace data.
Record/Stop button	Starts and stops the recording process. Use this button to begin gathering trace data.
Loop button	Sets whether the user interface recorder should loop during play back to repeat the recorded steps continuously. Use this setting to gather multiple runs of a given set of steps.
Target menu	Selects the target for the document. The target is the process (or processes) for which data is gathered.
Inspection Range control	Selects a time range in the track pane. When set, Instruments displays only data collected within the specified time period. Use the buttons of this control to set the start and end points of the inspection range and to clear the inspection range.
Time/Run control	Shows the elapsed time of the current trace. If your trace document has multiple data runs associated with it, you can use the arrow controls to choose the run whose data you want to display in the track pane.
View control	Hides or shows the Instruments pane, Detail pane, and Extended Detail pane. This control makes it easier to focus on the area of interest.
Library button	Hides or shows the instrument library. For information on using the Library window, see "Adding and Configuring Instruments."
Search field	Filters information in the Detail pane based on a search term that you provide. Use the search field's menu to select search options.

Allocations

- The allocations instrument captures and displaying information about memory allocations in your app.
- Allocations can help you identify what the hell in your app is taking up all that memory (wrongly sized images!)
- Demo

Leaks

- The Leaks instrument measures your general memory usage, and marks any leaked memory.
- If a leak is an object, it will be reported by its Class name. If its not an object, it will just say Malloc-size.
- Demo

Zombies

- The Zombies instrument will help you detect overreleased objects, aka zombies aka dangling pointers.
- Zombies works by replacing any object that has its retain count set to 0 with an NSZombie object, and then detects whenever these zombie objects are executed on. Pretty clever.
- For use with pre-arc code.
- Demo

Time Profiler

- Time profiler helps you provide a great user experience. Use it anytime your app is acting ‘choppy’ or sluggish and you aren't sure why.
- It’s goal is to tell you how much time is spent in each method.
- Demo

Printing a stack trace

- If you your execution is ever paused, like for a breakpoint, you can simply type in **bt** in the console to print out a back trace.
- Sometimes this is helpful to send to someone else if you are debugging

Demo

Objective-C Runtime

Sourced from Apple’s runtime guide and CocoaSamurai

Objective-C Runtime

- Objective-C defers as many decisions as it possibly can from compile and link time to runtime.
- This makes Objective-c very dynamic.
- This means Objective-c doesn't just need a compiler, it also needs a runtime system to execute the compiled code
- Think of the Objective-C runtime as the operating system of the Objective-C language.
- The Objective-C library is written in C & Assembler. It is what adds the object oriented capabilities to C, thereby giving us Objectice-C

Objective-C Runtime

- Do you need to understand how the runtime works in order to write awesome apps? Absolutely not
- But having an understanding of how the runtime works is great knowledge to have if you want to further understand how Objective-C/iOS works, which in turn can help you write better code.
- I would expect an iOS/Objective-C ‘Expert’ to be familiar with the topics we are about to discuss. (Interviewers can also ask about it to quickly gauge how much experience you really have)
- The runtime is open source! (http://opensource.apple.com)
- Swift uses the objective-c runtime, since it uses objectice-c built frameworks (UIKit, etc)

Objective-C Versions

- There are two versions of the Objective-C runtime:
  - Modern version (introduced in Objective-C 2.0)
  - Legacy version
- iOS & 64-bit OS X applications use the modern version
- 32-bit OS X applications use the legacy version
- The biggest difference between the two is the addition of “non-fragile” ivars in the modern runtime (google this for an explanation)

Interacting with the runtime

- Your Objective-C apps interact with the runtime on 3 different levels:
  - **Objective-C Source Code**: The runtime system works automatically with the Objective-C code you write and then compile.
  - **NSObject Methods**: Most objects in Cocoa are subclasses of the NSObject class, so they inherit its methods. Many of the methods are directly designed to work with the runtime (class, isMemberofClass,respondsToSelector,etc)
  - **Runtime Functions**: The runtime is a dynamic shared open source library. You can freely modify it to your liking to develop systems and tools to augment the development experience.

Messaging

- Whenever you send a message in Objective-C:
  - [receiver message];
- The compiler converts this message expression into a call of the function objc_msgSend. This function takes the receiver and the name of the method mentioned in the message,aka the selector, as its two principle parameters:
  - objc_msgSend(receiver,selector)
- Any arguments are passed in with the message are also handled by objc_msgSend:
  - objc_msgSend(receiver, selector, arg1, arg2,….)

Class structure

- The key to understanding how messaging work is by understanding the structure that the compiler builds for each class and object
- Every class structure includes two essential elements:
  - a pointer to the superclass
  - a class dispatch table
- The class dispatch table has entries that associate method selectors with the class specific addresses of the methods they identify.

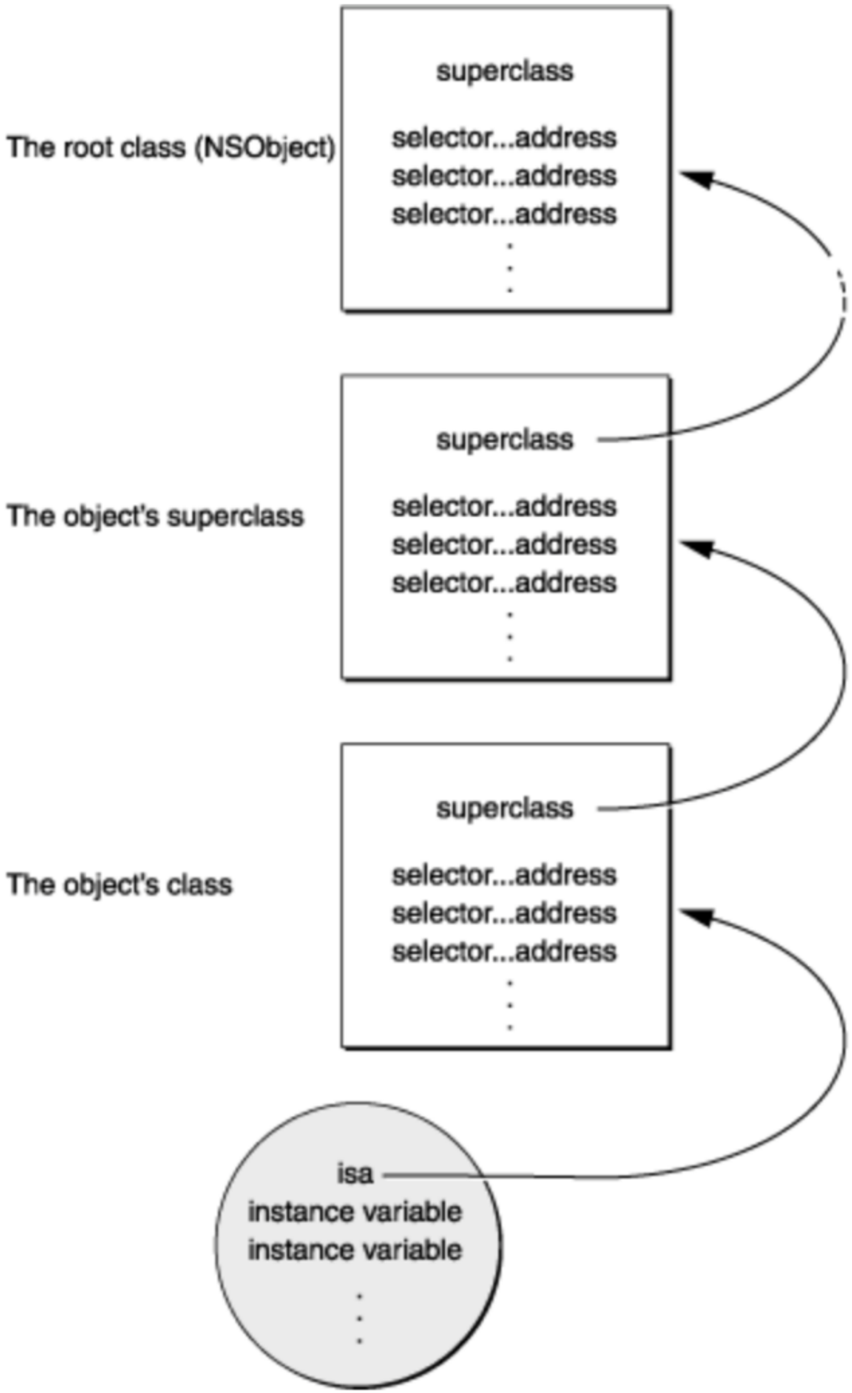
Object Structure

- So we know an objective-c class keeps a reference to its super class, and a list of all its implemented methods.
- When a new object is created, memory for it is allocated, and its instance variables are initialized.
- First among the object's variables is one very important element:
  - A pointer called 'isa'
- The 'isa' pointer is a pointer to an object's class structure.

Message sending

- So when a message is sent to an object, the messaging function (objc_msgSend()) follows the object's isa pointer to the class structure, where it looks up the method selector in the dispatch table.
- If it cant find the method, objc_msgSend() follows the class structure's pointer to its super class and tries to find the method again.
- With each failure to find the method, objc_msgSend() climbs the class hierarchy until it reaches the NSObject class.
- Once the method is found, the function calls the method in the table and passes it the receiving object's data structure.

Message sending



Dynamic binding

- All of this messaging business is how method implementations are chosen at runtime.
- In object oriented jargon, this means the methods are dynamically bound to messages

Hidden arguments

- When objc_msgSend finds the method, it calls the method and passes in all the regular parameters of the method (if it has any)
- It also passes in two hidden arguments:
  - The receiving object
  - The selector for the method
- These arguments give the method implementation information about the two halves of the message expression that invoked it:
  - [receivingObject selectorForMethod]
- These arguments are said to be ‘hidden’ because they aren't declared in your source code that defined the method. They are inserted into the implementation when the code is compiled.

Hidden arguments

- You can actually access these hidden arguments in your source code!

```
-(void)doSomethingCrazy {

  id target = self;
  SEL method = _cmd;
```

- Self is by far the more useful argument, since its what allows you to access that object's ivars/properties

Class Cache

- When the runtime follows an object's isa pointer to its class, it can find many methods (hundreds of them even!)
- Often times, only a small handful of methods are actually called on a regular basis
- So the class implements a cache, whenever a method is called it puts into the cache
- This greatly increases the speed of objc_msgSend(), because it means it usually won't have to search through the entire table of implemented methods

vtable

- Most Objective-C methods are located using hash table lookup inside objc_msgSend
- In the modern Runtime, there is feature called virtual table, which allows the most popular methods that are called to be stored in a virtual table.
- Virtual table allows us to look things up array style, which is faster than a hash table (because of collisions)
- Objective-C’s vtable implementation limit its use to a few selectors that are either called all the time and rarely overriden.

vtable

objc_msgSend_vtable0	allocWithZone:
objc_msgSend_vtable1	alloc
objc_msgSend_vtable2	class
objc_msgSend_vtable3	self
objc_msgSend_vtable4	isKindOfClass:
objc_msgSend_vtable5	respondsToSelector:
objc_msgSend_vtable6	isFlipped
objc_msgSend_vtable7	length
objc_msgSend_vtable8	objectForKey:
objc_msgSend_vtable9	count
objc_msgSend_vtable10	objectAtIndex:
objc_msgSend_vtable11	isEqualToString:
objc_msgSend_vtable12	isEqual:
objc_msgSend_vtable13	retain (non-GC) hash (GC)
objc_msgSend_vtable14	release (non-GC) addObject: (GC)
objc_msgSend_vtable15	autorelease (non-GC) countByEnumeratingWithState:objects:count: (GC)

the run time keeps a vtable for the most NSObject methods. It will also generate vtables for your own classes

Objective-c Runtime

- For further knowledge and practice with the Objective-C runtime, look up the topics method swizzling and associated objects.

Run Loops

Run Loops

- Run loops are what actually make your code run
- They are a messaging mechanism, they schedule work and coordinate the receipt of incoming events
- You can think of it like a post box that waits for messages and delivers them to recipients

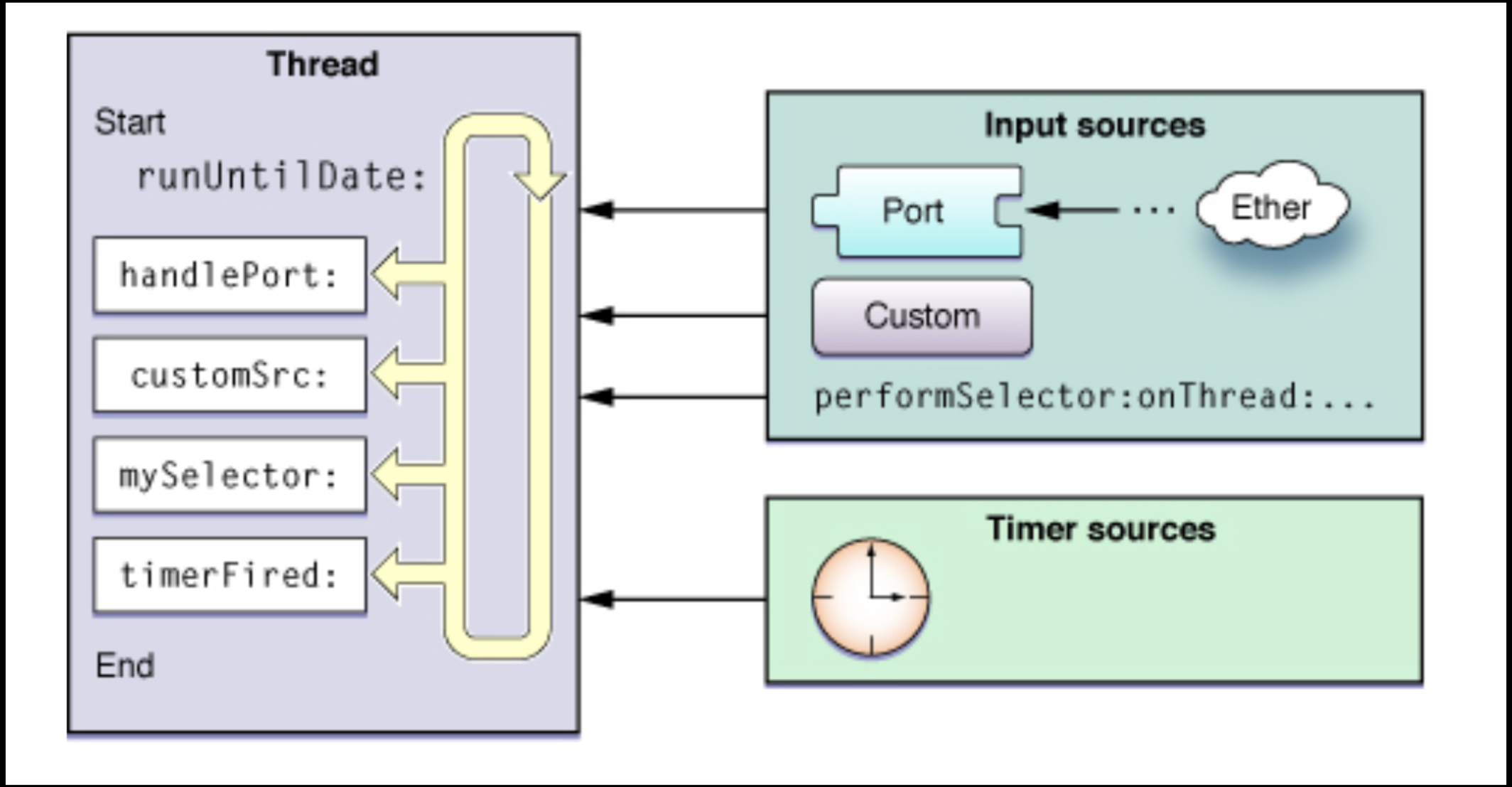
Run Loops

- Run loops do two things:
  - wait until something happens
  - dispatch that message to its receiver
- Run loops are what separates our interactive iOS apps from command line tools.
- (Most) command line tools are launched with parameters, execute their command, and then exit.
- But our apps wait for user input, react, and then go back to waiting.

Run Loop Sources

- A run loop receives events from two types of sources:
  - Input sources deliver asynchronous events, usually messages from other threads or from a different application
  - Timer sources deliver synchronous events, occurring at a scheduled time or repeating interval.

Run Loop Sources



Run Loop Modes

- A run loop mode is a collection of input sources and timers to be monitored and a collection of run loop observers to be notified.
- Each time you run your run loop, a particular mode is specified to be run.
- During that pass of the run loop, only sources associated with that run loop are monitored and allowed to deliver events.
- You can use modes to filter out any unwanted sources during a particular pass.
- Most of the time, your run loops will simply run in the system-defined “default” mode.

Run Loops and Threads

- Every thread has one run loop associated with it
- In OSX/iOS applications, this is an instance of NSRunLoop. For lower level applications, use CFRunLoopRef
- You can also run [NSRunLoop currentRunLoop] to get a reference to the run loop associated with the current thread your code is executing on

Demo

When to use Run Loops?

- The only time you need to work with run loops is if you explicitly create secondary threads (instances of NSThread) for your application
- This is pretty rare now because of GCD and OperationQueues
- Even then, NSThreads automatically create a run loop when the thread itself is created, so you wouldn't need to create the run loop yourself
- It's just good to know generally what a run loop is and what it does.

Messaging Nil

Working with Nil

- In most other OOP languages, trying to interact with a nil reference will raise an exception/crash your application
- In Objective-C, messages can be safely sent to nil references

Nil Messaging Rules

- If the method you are calling on the nil object returns an object, the method returns nil
- If the method you are calling on the nil object returns a number/scalar, the method returns 0
- If the method you are calling on the nil object returns a struct, the method returns a struct with all fields set to 0
- If the method returns anything else, the return value is undefined

sending messages to nil != sending messages to deallocated instances

- If you send a message to a deallocated object, your app will crash every time
- Its rare for this to happen in an ARC environment, since ARC will properly release things for us when things are set to nil
- But where it can happen even in ARC is if you are working with a property with the assign modifier
- We experienced this issue with our animation controller and being our navigation controller's delegate
- A navigation controller's delegate property has the assign modifier. Assign modifier means this property wont be automatically nil'd out when the object it points to is deallocated. A weak property will do that automatically, weak property are duh best.