

**Jack be nimble,  
Jack be quick,  
Jack jump over  
The candlestick.**



# Unit Testing with Quick

Presented at [Cocoaheads Denver](#)

2020-03-10 at Galvanize

by @phatblat



... and Nimble

QuicK

# Slides & Examples

## phatblat/UnitTestingWithQuick



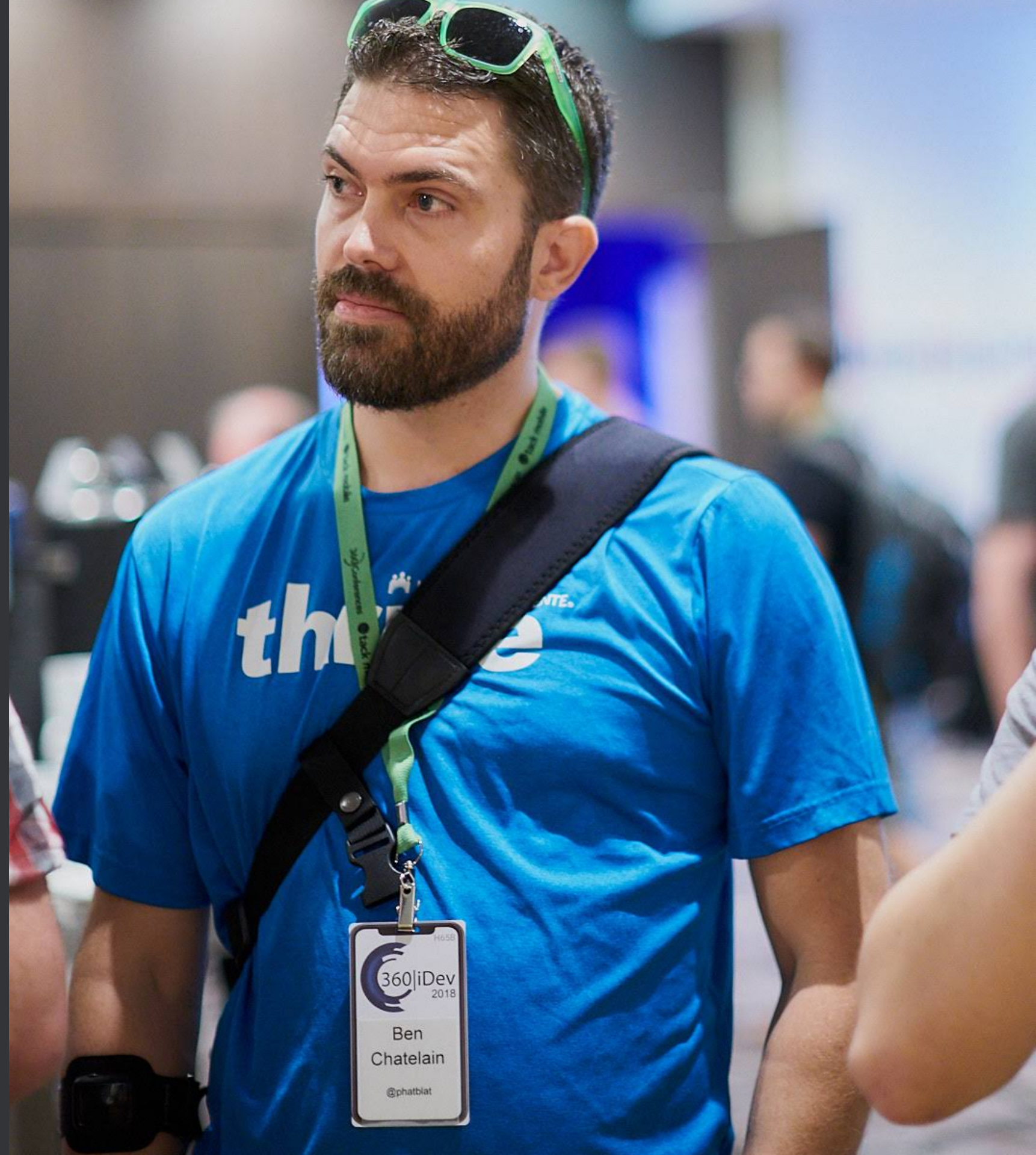
```
1 //
2 DolphinSpec.swift
3
4 import Quick
5 import Nimble
6 import Nimble
7 import Sea
8
9 // MARK: - DolphinSpec
10
11 describe("a dolphin") {
12
13     var dolphin: Dolphin!
14     beforeEach {
15         let position = Position(longitude: 78.304129, latitude: 28.291769, depth: 20.0)
16         dolphin = Dolphin(position: position)
17         Ocean.sharedOcean.add(dolphin)
18     }
19
20     describe("click") {
21         context("when it's not near anything interesting") {
22             it("emits only one click") {
23                 expect(dolphin.click()).to(equal("Click!"))
24             }
25         }
26     }
27
28     context("when it's near something interesting") {
29         beforeEach {
30             let position = Position(longitude: 78.304129, latitude: 28.291769, depth: 20.0)
31             Ocean.sharedOcean.add(SunkenShip(position: position))
32         }
33         it("emits a series of clicks") {
34             expect(dolphin.click()).to(equal("Click, click, click!"))
35         }
36     }
37 }
38 }
```

✖ failed - expected <Click!> to equal <Click, click, click!>



# @phatblat

- Ben Chatelain
- Chief iOS Engineer
- Kaiser Permanente
- Manage suite of ~30 iOS & Android libs
- open source: Quick, mas, Objective-Git



# Quick & Nimble

Open Source

- Quick/[Quick](#)
  - BDD-style testing framework, used to define examples.
- Quick/[Nimble](#)
  - Matcher framework used to express expectations.

# BDD

- *Behavior-Driven Development*
- Don't test code
- Verify behavior
- Semi-formal format for behavior spec
- Similar to user story
- Object-oriented design
- Gherkin: Scenario, Given, When, Then

# User Story

As a store owner,  
I want to add items back to inventory when they are returned or  
exchanged,  
so that I can track inventory.



# Gherkin

**Scenario 1:** Items returned for refund should be added to inventory.  
**Given** that a customer previously bought a black sweater from me  
**and** I have three black sweaters in inventory,  
**when** they return the black sweater for a refund,  
**then** I should have four black sweaters in inventory.

**Quick**

quick

# RSpec

- Behaviour Driven Development for Ruby.
- "Making TDD Productive and Fun."

```
1 require 'bowling'↵
2 ↵
3 RSpec.describe Bowling, "#score" do↵
4   context "with no strikes or spares" do↵
5     it "sums the pin count for each roll" do↵
6       bowling = Bowling.new↵
7       20.times { bowling.hit(4) }↵
8       expect(bowling.score).to eq 80↵
9     end↵
10  end↵
11 end↵
```

# QuickSpec

```
class TableOfContentsSpec: QuickSpec {
  override fun spec() {
    describe("the 'Documentation' directory") {
      it("has everything you need to get started") {
        let sections = Directory("Documentation").sections
        expect(sections).to(contain("Organized Tests with Quick Examples and Example Groups"))
        expect(sections).to(contain("Installing Quick"))
      }

      context("if it doesn't have what you're looking for") {
        it("needs to be updated") {
          let you = You(awesome: true)
          expect{you.submittedAnIssue}.toEventually(beTruthy())
        }
      }
    }
  }
}
```

# describe

```
describe("the thing") { /* closure */ }
```

- Describes the thing being tested.
- Groups examples.
- Serves as a prefix for the actual test name.
- Analogous to **XCTestCase**

# context

```
context("when dark mode is enabled") { /* closure */ }
```

- Describes a condition.
- Optional alternate 2nd-Nth level of grouping for examples.



# it

```
it("calculates an average score") { /* closure */ }
```

- Describes an example behavior.
- Contains assertions (expectations).
- *One expectation per example.*

# Setup & Teardown

```
var dolphin: Dolphin!
```

```
beforeEach { dolphin = Dolphin() }
```

```
afterEach { dolphin = nil }
```

- Contains logic to be run before/after each function *in the same scope*.
- ✨ Can be placed inside any/every **describe** and **context**.

# ✨ Nesting FTW!

```
describe("dolphin") {  
  beforeEach { dolphin = Dolphin() }  
  context("when out of water") {  
    beforeEach { dolphin.airborne = true }  
    context("and making noise") {  
      beforeEach { dolphin.vocalizationLevel = 5 }  
      it("is loud") { /* closure */ }  
      it("can be heard from 100m away") { /* closure */ }    }  
  }  
}
```

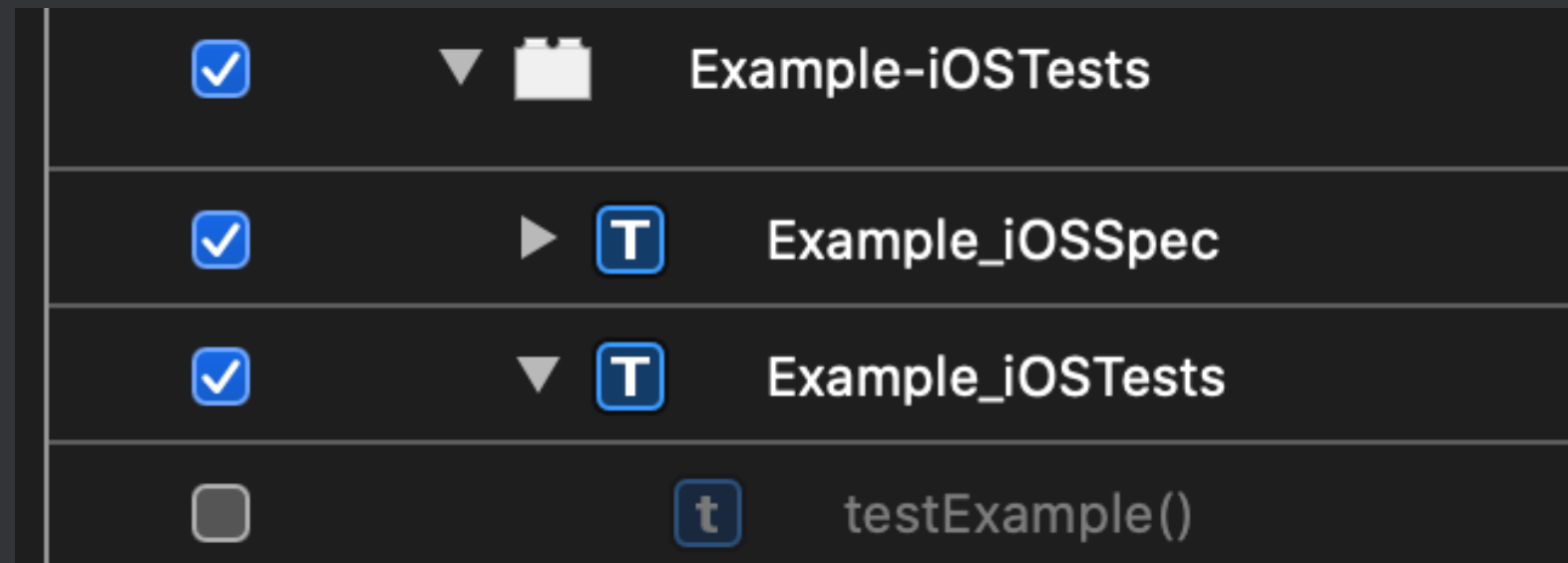
# Suite Setup & Teardown

```
override func spec() {  
    beforeSuite {  
        OceanDatabase.createDatabase(name: "test.db")  
        OceanDatabase.connectToDatabase(name: "test.db")  
    }  
    afterSuite {  
        OceanDatabase.tearDownDatabase(name: "test.db")  
    }  
    describe("a dolphin") {}  
}
```

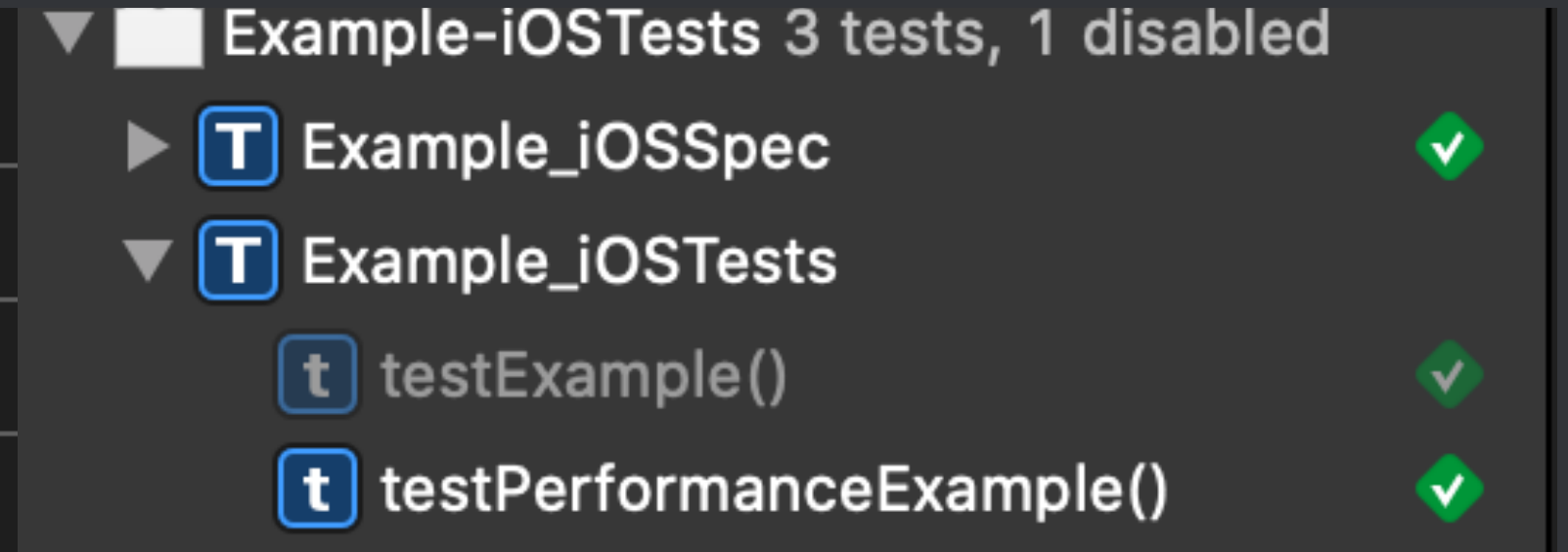
# Disabled Test (Xcode)

- Must edit scheme to disable tests.
- Shows test method as disabled.

Scheme > Test action



Test Navigator



# Disabled Test (Quick)

```
xit("this example is disabled") { code.compiles() == yes }
```

- Prefix any example with `x` to disable.
- Controlled in code.
- Disabled examples do not show in Test Navigator



# ✨ Disabled Test Suite

```
xdescribe("the thing") {  
  xcontext("when dark mode is enabled") {  
    xit("this example is disabled") { code.compiles() == yes }  
  }  
}
```

- Prefix any Quick function(s) with **x** to disable everything under that scope.
- Any combination of disabled functions will be skipped.

# ✨ Focused Test

```
fit("is focused") { expect(example).toRun(true) }  
it("will be ignored") { code.compiles() == yes }
```

- Prefix any example with `f` to focus.
- Only the focused example(s) will be run.

# ✨ Focused Test Suites

```
fdescribe("the thing") {  
  fcontext("when dark mode is enabled") {  
    fit("is focused") { expect(example).toRun(true) }  
  }  
}
```

- Prefix any Quick function(s) with **f** to focus everything under that scope.
- Only the focused example(s) will be run.
- All focused examples will be run.

# Test Readability

## XCTest

```
func testDolphin_click_whenTheDolphinIsNearSomethingInteresting_isEmittedThreeTimes() {  
    // ...  
}
```


## Quick

```
describe("a dolphin") {  
    describe("its click") {  
        context("when the dolphin is near something interesting") {  
            it("is emitted three times") {  
                // ...  
            }  
        }  
    }  
}
```















# Quick Test Names

▼ DolphinSpec › Example-iOSTests 1 passed (100%) in 0.0011s

✓  a\_dolphin\_its\_click\_when\_the\_dolphin\_is\_near\_something\_interesting\_is\_emitted\_three\_times()



# More Test Names

Tests	Status
ViewControllerSpec > OutletActionAssertionTests	
 view_controller__has_a_leftButton_outlet()	
 view_controller__has_a_rightButton_outlet()	
 view_controller__has_a_segmentedControl_outlet()	
 view_controller__receives_a_didTapLeftButton__action_from_leftButton()	
 view_controller__receives_a_didTapRightButton__action_from_rightButton()	
 view_controller__receives_a_segmentedControlValueChanged__action_from_segmentedControl()	

Names are automatically built from describe/context/it descriptions



# Don't Use Properties

```
class TableOfContentsSpec: QuickSpec {  
    var dolphin: Dolphin!  
    override func spec() {  
        describe("dolphin") {  
            beforeEach { self.dolphin = Dolphin() }  
        }  
    }  
}
```

# Use Local Variables

```
class TableOfContentsSpec: QuickSpec {  
  override func spec() {  
    var dolphin: Dolphin!  
    describe("dolphin") {  
      beforeEach { dolphin = Dolphin() }  
    }  
  }  
}
```

# Nimble

# Nimble

- Matcher framework
- Swift and Objective-C
- used to test your **expect**-ations

```
expect(1).to(beAnInstanceOf(Int.self))  
expect("turtle").to(beAnInstanceOf(String.self))
```

# Equality

```
expect(1 + 1).toEqual(2))
```

```
expect(1 + 1) == 2
```

```
expect(1 + 1).toNot(toEqual(3))
```

```
expect(1 + 1) != 3
```



# Decimal Precision

```
expect(1.2).to(beCloseTo(1.1, within: 0.1))
```

```
expect(1.2) == (1.1, 0.1)
```

```
expect(1.2) ≈ 1.1999
```

# Comparison

```
expect(2).to(toBeLessThan(3))
```

```
expect(2) < 3
```

```
expect(3).to(toBeLessThanOrEqualTo(3))
```

```
expect(3) <= 3
```

```
expect(5).to(toBeGreaterThan(3))
```

```
expect(5) > 3
```

```
expect(5).to(toBeGreaterThanOrEqualTo(5))
```

```
expect(5) >= 5
```

# Nilability

```
var dog: Dog? = nil  
expect(dog).to(beNil())  
expect(dog) == nil
```

# Identity

```
expect(actual).to(beIdenticalTo(expected))
```

```
expect(actual) === expected
```

```
expect(actual).toNot(beIdenticalTo(expected))
```

```
expect(actual) !== expected
```

# String Contents

```
expect("seahorse").to(contain("sea"))
```

# Collection Contents

```
expect(["Atlantic", "Pacific"]).toNot(contains("Mississippi"))
```

# Type Checking

## Type Membership

```
expect(1).to(beAKindOf(Int.self))
```

```
expect("turtle").to(beAKindOf(String.self))
```

## Exact Type

```
expect(1).to(beAnInstanceOf(Int.self))
```

```
expect("turtle").to(beAnInstanceOf(String.self))
```

# Type Safety

## Nimble

```
it("does not compile") {  
  expect(1 + 1).to(equal("Squee!"))  
  // Cannot convert value of type 'Int' to expected argument type 'String?'
```

## XCTest

```
func testComparingDifferentTypes() throws {  
  XCTAssertEqual("Squee!", 1 + 1)  
  // Cannot convert value of type 'String' to expected argument type 'Int'
```



# Custom Failure Message

## Nimble

```
expect(1 + 1).to(equal(3))  
// failed - expected to equal <3>, got <2>
```

```
expect(1 + 1).to(equal(3), description: "Make sure libKindergartenMath is loaded")  
// failed - Make sure libKindergartenMath is loaded  
// expected to equal <3>, got <2>
```

## XCTest

```
XCTAssertEqual(1 + 1, 3, "Make sure libKindergartenMath is loaded")  
// XCTAssertEqual failed: ("2") is not equal to ("3") - Make sure libKindergartenMath is loaded
```

# Async Test

```
expect(ocean.isClean).toEventually(beTruthy())
```





# Custom Matcher

```
expect(result).to(beFailure { error in  
    expect(error) == .searchFailed  
})
```

```
func beFailure(test: @escaping (MASError) -> Void = { _ in }) -> Predicate<Result<(), MASError>> {  
    return Predicate.define("be <failure>") { expression, message in  
        if let actual = try expression.evaluate(),  
            case let .failure(error) = actual {  
            test(error)  
            return PredicateResult(status: .matches, message: message)  
        }  
        return PredicateResult(status: .fail, message: message)  
    }  
}
```

# Quick Caveats

- Easy to forget disabled/focused tests.
- Cannot run single example using  or  in editor gutter.
- Clicking on Quick example in Test Navigator doesn't navigate to code.
- Quick tests don't display in Test Navigator until tests have been run.
- No support for performance tests.
- External dependency.

# iOS Examples

# macOS Examples



# Tests that take too long to run end up not being run.

— **Michael Feathers**



# References

- Unit Testing With Quick - Slides & Code Samples
- Quick
  - Quick Docs
- Nimble
- Jon Reid
  - Quality Coding
  - iOS Unit Testing By Example 