Making Swift Mock Objects More Powerful 💪

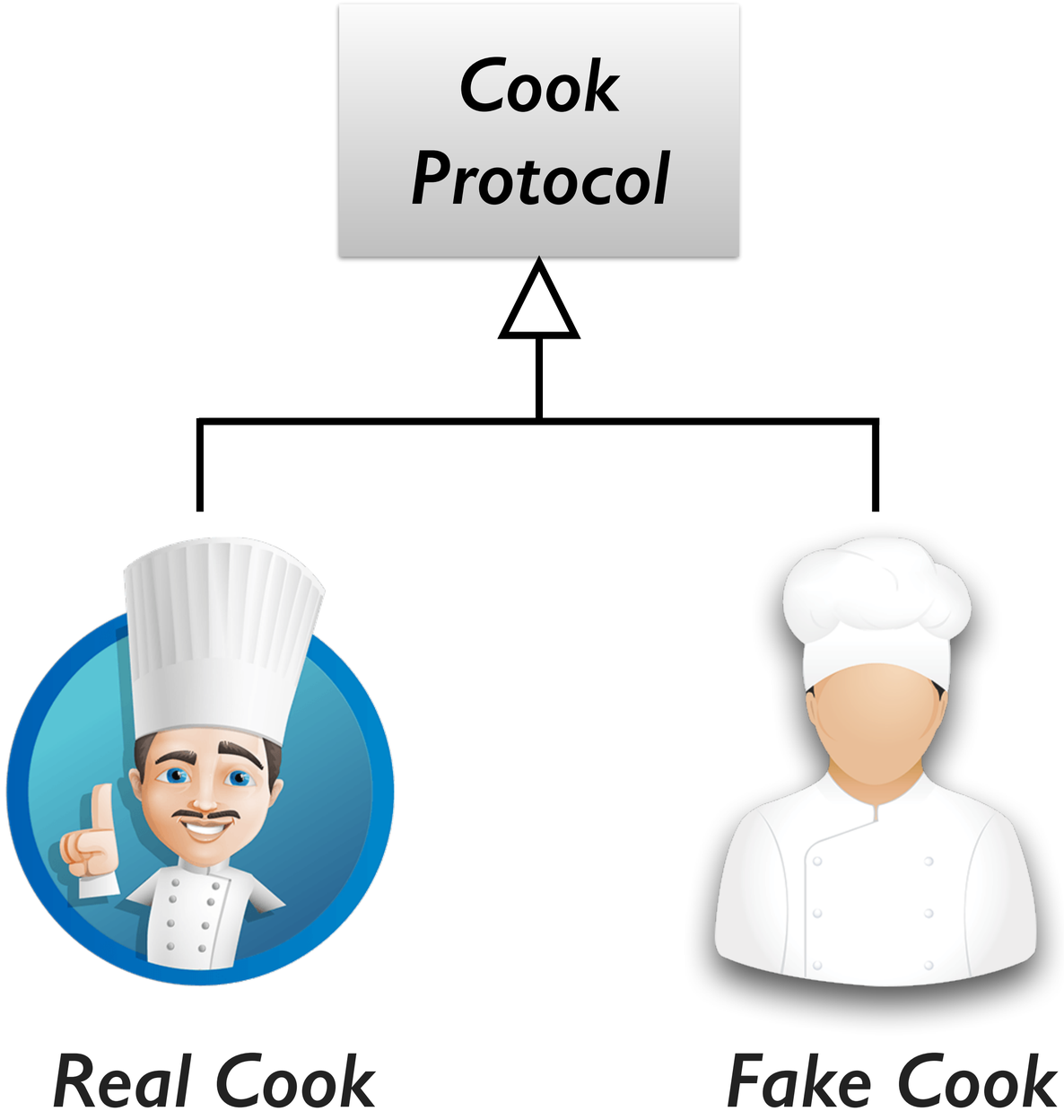
Jon Reid (American Express)
@qcoding

Interaction Test

Table *Waiter* *Cook*

Interaction Test

Table *Waiter* *Cook*


Cook
Protocol
Real Cook
Fake Cook

```
enum RamenSoup {
    case shio      // salt base
    case shoyu     // soy sauce base
    case miso      // miso paste base
    case tonkotsu // creamy pork base
}

protocol CookProtocol {
    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String],
            completionHandler: @escaping (Result<🍜>) -> Void
        ) -> Void
}
```

```
struct Waiter {
    let cook: CookProtocol

    func order() {
        cook.cookRamen(
                bowls: 2,
                soup: .miso,
                extras: ["wakame", // seaweed
                         "tamago"]) // egg
    }
}
```

```
class MockCook: CookProtocol {
    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        // ??? capture information
    }
}

class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        // ??? assert something
    }
}
```

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        XCTAssertTrue(mockCook.cookRamenWasCalled)
    }
}
```

```
class MockCook: CookProtocol {
    var cookRamenWasCalled = false

    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        cookRamenWasCalled = true
    }
}
```

```
class MockCook: CookProtocol {
    var cookRamenCallCount = 0

    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        cookRamenCallCount += 1
    }
}
```

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        XCTAssertEqual(mockCook.cookRamenCallCount, 1)
    }
}
```

```
class MockCook: CookProtocol {
    var cookRamenCallCount = 0

    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        cookRamenCallCount += 1
    }
}
```

```
class MockCook: CookProtocol {
    var cookRamenCallCount = 0
    var cookRamenBowls = 0

    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        cookRamenCallCount += 1
        cookRamenBowls = bowls
    }
}
```

```
class MockCook: CookProtocol {
    var cookRamenCallCount = 0
    var cookRamenBowls = 0
    var cookRamenSoup: RamenSoup?

    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        cookRamenCallCount += 1
        cookRamenBowls = bowls
        cookRamenSoup = soup
    }
}
```

```
class MockCook: CookProtocol {
    var cookRamenCallCount = 0
    var cookRamenBowls = 0
    var cookRamenSoup: RamenSoup?
    var cookRamenExtras: [String] = []

    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        cookRamenCallCount += 1
        cookRamenBowls = bowls
        cookRamenSoup = soup
        cookRamenExtras = extras
    }
}
```

```
class MockCook: CookProtocol {
    var cookRamenCallCount = 0
    var cookRamenLastBowls = 0
    var cookRamenLastSoup: RamenSoup?
    var cookRamenLastExtras: [String] = []

    func cookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        cookRamenCallCount += 1
        cookRamenLastBowls = bowls
        cookRamenLastSoup = soup
        cookRamenLastExtras = extras
    }
}
```

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        XCTAssertEqual(mockCook.cookRamenCallCount, 1)
        XCTAssertEqual(mockCook.cookRamenLastBowls, 2)
        XCTAssertEqual(mockCook.cookRamenLastSoup, RamenSoup.miso)
        XCTAssertEqual(mockCook.cookRamenLastExtras, ["wakame", "tamago"])
    }
}
```

```
    func verifyCookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        XCTAssertEqual(cookRamenCallCount, 1)
        XCTAssertEqual(cookRamenLastBowls, bowls)
        XCTAssertEqual(cookRamenLastSoup, soup)
        XCTAssertEqual(cookRamenLastExtras, extras)
    }
}

class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2,
            soup: .miso,
            extras: ["wakame", "tamago"])
    }
}
```

```
    func verifyCookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String]) -> Void {
        XCTAssertEqual(cookRamenCallCount, 1)
        XCTAssertEqual(cookRamenLastBowls, bow
        XCTAssertEqual(cookRamenLastSoup, soup)
        XCTAssertEqual(cookRamenLastExtras, extras)
    }
}

class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2, 3
            soup: .miso,
            extras: ["wakame", "tamago"])
    }
}
```

XCTAssertEqual failed: ("2") is not equal to ("3")

```
func verifyCookRamen(
        bowls: Int,
        soup: RamenSoup,
        extras: [String],
        file: StaticString = #file,
        line: UInt = #line) -> Void {
    XCTAssertEqual(cookRamenCallCount, 1, file: file, line: line)
    XCTAssertEqual(cookRamenLastBowls, bowls, file: file, line: line)
    XCTAssertEqual(cookRamenLastSoup, soup, file: file, line: line)
    XCTAssertEqual(cookRamenLastExtras, extras, file: file, line: line)
}
```

```
    func verifyCookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String],
            file: StaticString = #file,
            line: UInt = #line) -> Void {
        XCTAssertEqual(cookRamenCallCount, 1, file: file, line: line)
        XCTAssertEqual(cookRamenLastBowls, bowls, file: file, line: line)
        XCTAssertEqual(cookRamenLastSoup, soup, file: file, line: line)
        XCTAssertEqual(cookRamenLastExtras, extras, file: file, line: line)
    }
}

class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2, 3
            soup: .miso,
            extras: ["wakame", "tamago"])
    }
}
```

XCTAssertEqual failed: ("2") is not equal to ("3")

```
    func verifyCookRamen(
            bowls: Int,
            soup: RamenSoup,
            extras: [String],
            file: StaticString = #file,
            line: UInt = #line) -> Void {
        XCTAssertEqual(cookRamenCallCount, 1, "call count", file: file, line: line)
        XCTAssertEqual(cookRamenLastBowls, bowls, "bowls", file: file, line: line)
        XCTAssertEqual(cookRamenLastSoup, soup, "soup", file: file, line: line)
        XCTAssertEqual(cookRamenLastExtras, extras, "extras", file: file, line: line)
    }
}

class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2, 3
            soup: .miso,
            extras: ["wakame", "tamago"])
    }
}
```

XCTAssertEqual failed: ("2") is not equal to ("3") - bowls

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2,
            soup: .miso,
            extras: ["wakame", "tamago"])
    }
}
```

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2,
            soup: .miso,
            extras: ["tamago", "wakame"])
    }
}
```

XCTAssertEqual failed: ("["wakame", "tamago"]") is not equal to ("["tamago", "wakame"]") - extras

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2,
            soup: .miso,
            extrasMatcher: { extras in
                extras.count == 2 &&
                extras.contains("wakame") &&
                extras.contains("tamago") })
    }
}
```

```
func verifyCookRamen(
        bowls: Int,
        soup: RamenSoup,
        extrasMatcher: (([String]) -> Bool),
        file: StaticString = #file,
        line: UInt = #line) -> Void {
    XCTAssertEqual(cookRamenCallCount, 1, "call count", file: file, line: line)
    XCTAssertEqual(cookRamenLastBowls, bowls, "bowls", file: file, line: line)
    XCTAssertEqual(cookRamenLastSoup, soup, "soup", file: file, line: line)
    XCTAssertTrue(extrasMatcher(cookRamenLastExtras),
            "extras was \(cookRamenLastExtras)",
            file: file, line: line)
}
```

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2,
            soup: .miso,
            extras: ["tamago", "wakame"])
    }
}
```

XCTAssertTrue failed - extras was ["tamago", "nori"]

Hamcrest Matchers

- Predicate: match(value) ⇒ true/false
- Describes itself (expectations)
- Describes mismatch precisely
- Collection matchers composed of other matchers
- Framework for writing new matchers

```
func applyMatcher<T>(_ matcher: Matcher<T>, label: String, toValue value: T,
                     _ file: StaticString, _ line: UInt) -> Void {
    switch matcher.matches(value) {
    case .match:
        return
    case let .mismatch(mismatchDescription):
        XCTFail(label + " " +
                    describeMismatch(value, matcher.description, mismatchDescription),
                file: file, line: line)
    }
}
```

```
func verifyCookRamen(
        bowls: Int,
        soup: RamenSoup,
        extras: Matcher<Array<String>>,
        file: StaticString = #file,
        line: UInt = #line) -> Void {
    XCTAssertEqual(cookRamenCallCount, 1, "call count", file: file, line: line)
    XCTAssertEqual(cookRamenLastBowls, bowls, "bowls", file: file, line: line)
    XCTAssertEqual(cookRamenLastSoup, soup, "soup", file: file, line: line)
    applyMatcher(extras, label: "extras", toValue: cookRamenLastExtras, file, line)
}
```

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2,
            soup: .miso,
            extrasMatcher: { extras in
                extras.count == 2 &&
                extras.contains("wakame") &&
                extras.contains("tamago") })
    }
}
```

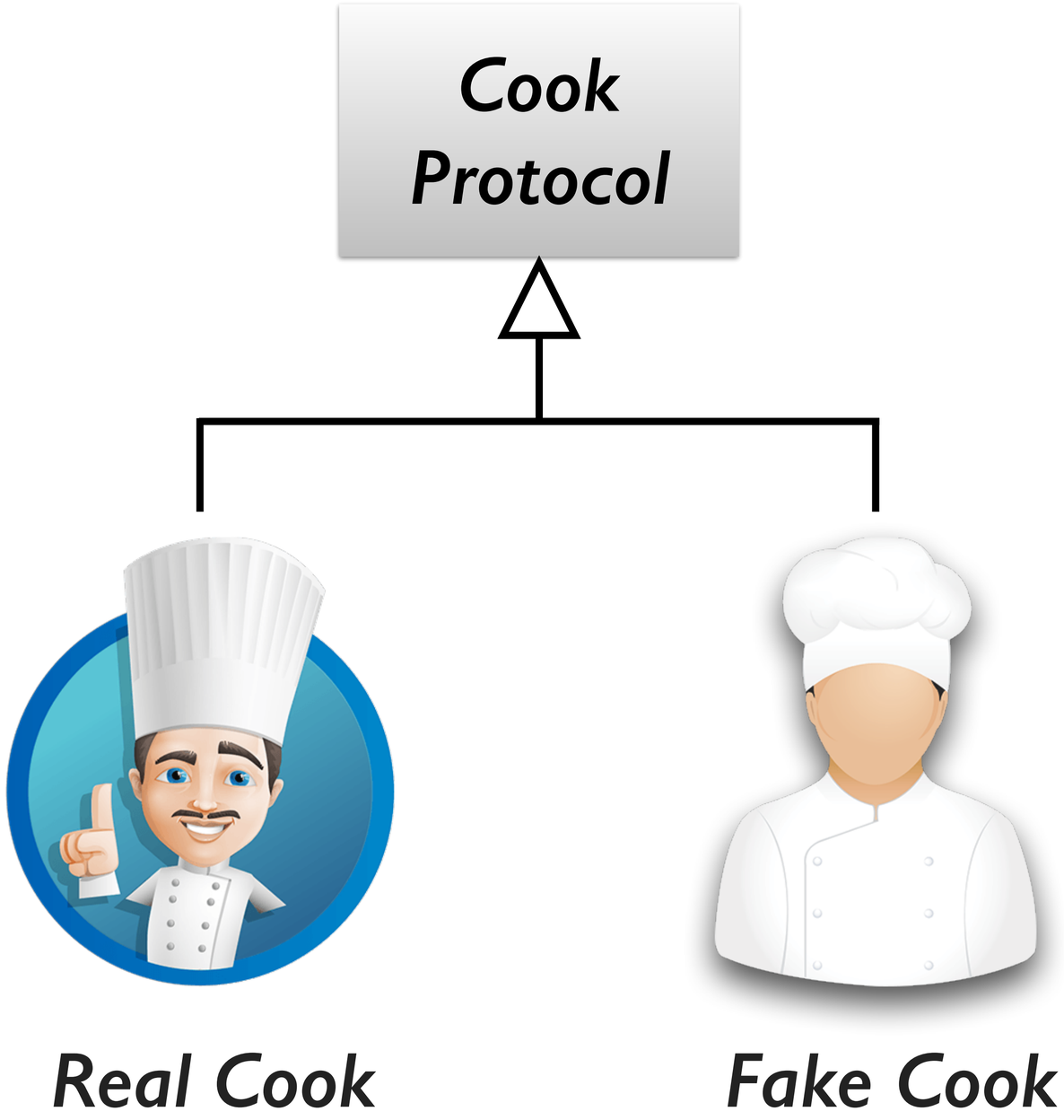
```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: 2,
            soup: .miso,
            extras: containsInAnyOrder("tamago", "chashu"))
    }
}
```

failed: extras GOT ["wakame", "tamago"] (mismatch: GOT: "wakame", EXPECTED: "chashu"),
EXPECTED: a sequence containing in any order ["tamago", "chashu"]

```
func verifyCookRamen(,
        bowls: Matcher<Int>,
        soup: Matcher<RamenSoup>,
        extras: Matcher<Array<String>>,
        file: StaticString = #file,
        line: UInt = #line) -> Void {
    applyMatcher(equalTo(1), label: "call count", toValue: cookRamenCallCount, file, line)
    applyMatcher(bowls, label: "bowls", toValue: cookRamenLastBowls, file, line)
    applyMatcher(soup, label: "soup", toValue: cookRamenLastSoup!, file, line)
    applyMatcher(extras, label: "extras", toValue: cookRamenLastExtras, file, line)
}
```

```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: equalTo(2),
            soup: equalTo(.miso),
            extras: containsInAnyOrder("tamago", "chashu"))
    }
}
```

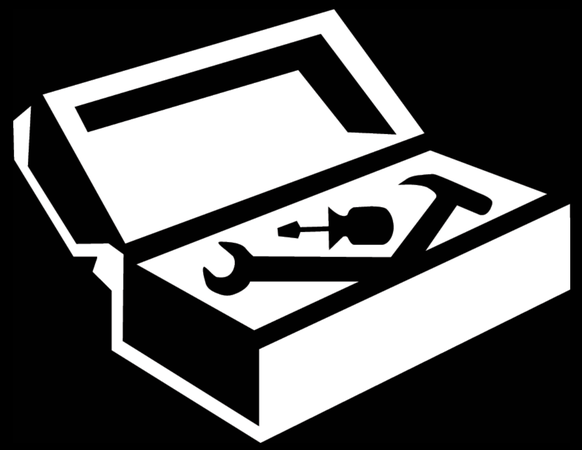
```
class WaiterTests: XCTestCase {
    func testOrder_ShouldCookRamen() {
        let mockCook = MockCook()
        let waiter = Waiter(cook: mockCook)

        waiter.order()

        mockCook.verifyCookRamen(
            bowls: greaterThanOrEqualTo(2),
            soup: anything(),
            extras: containsInAnyOrder(equalTo("tamago"),
                                       hasPrefix("chashu")))
    }
}
```


Cook
Protocol
Real Cook
Fake Cook

💪 Mock Recommendations 💪

- Capture "call count" (Int not Bool)
- Multiple asserts? Extract a helper method
- Pass file/line to asserts in helper
- Add messages to each assert
- Assert using predicates, not equality
- Consider using Swift Hamcrest matchers

Thank you!



Slides and code:
qualitycoding.org/tryswift2017