A.STEFANCHUK IT STEP

UNIT TESTING

WHAT IS UNIT TEST?

WHAT IS UNIT TEST?

- Unit testing is method to verify that single unit of code (function, class, etc.) is working correctly

WHAT IS UNIT TEST?

- Unit testing is method to verify that single unit of code (function, class, etc.) is working correctly
- Unit tests are executed very quickly (in milliseconds)

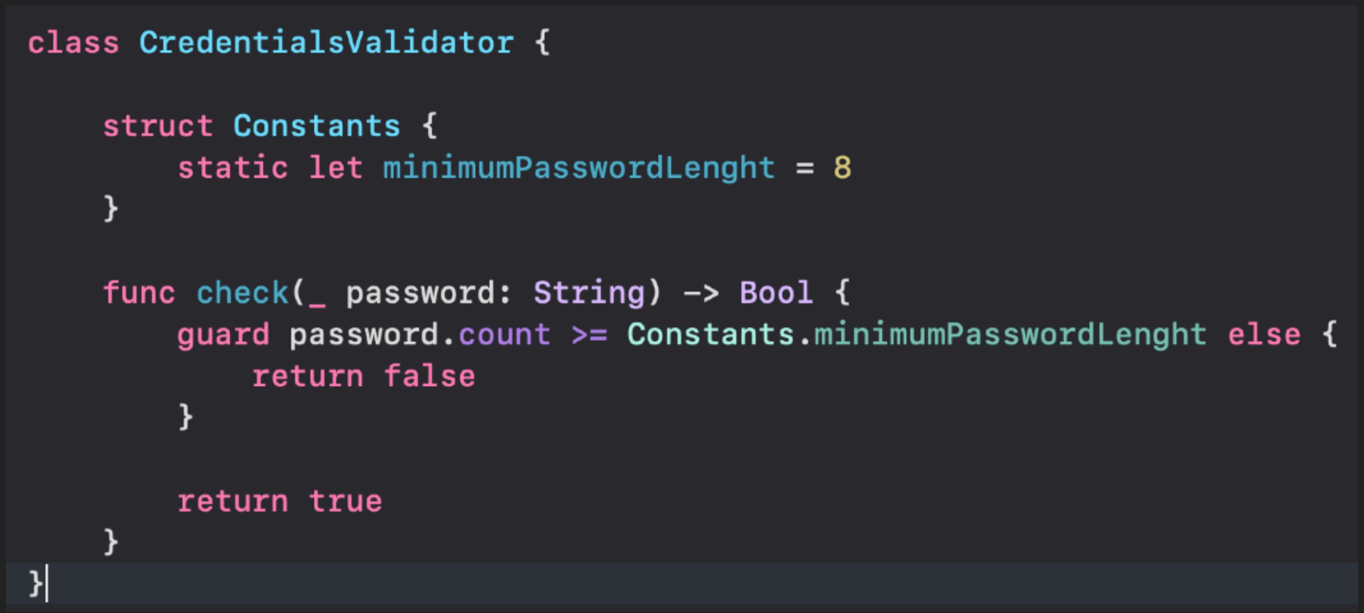
WHAT IS UNIT TEST?

- Unit testing is method to verify that single unit of code (function, class, etc.) is working correctly
- Unit tests are executed very quickly (in milliseconds)
- We test units of code in isolation - if class is dependent on some other classes -> we will control that (about that later)

WHAT IS UNIT TEST?

- Unit testing is method to verify that single unit of code (function, class, etc.) is working correctly
- Unit tests are executed very quickly (in milliseconds)
- We test units of code in isolation - if class is dependent on some other classes -> we will control that (about that later)
- Failing unit test should clearly point out the problem in code

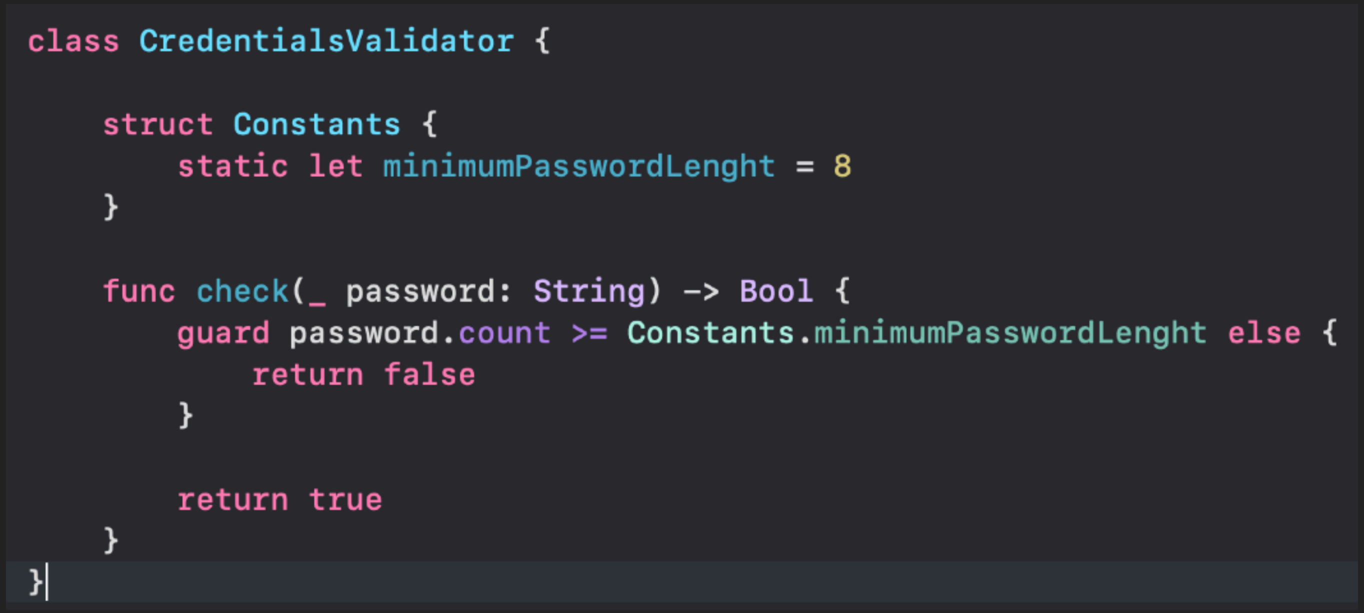
UNIT TEST EXAMPLE



```
class CredentialsValidator {

    struct Constants {
        static let minimumPasswordLenght = 8
    }

    func check(_ password: String) -> Bool {
        guard password.count >= Constants.minimumPasswordLenght else {
            return false
        }

        return true
    }
}
```

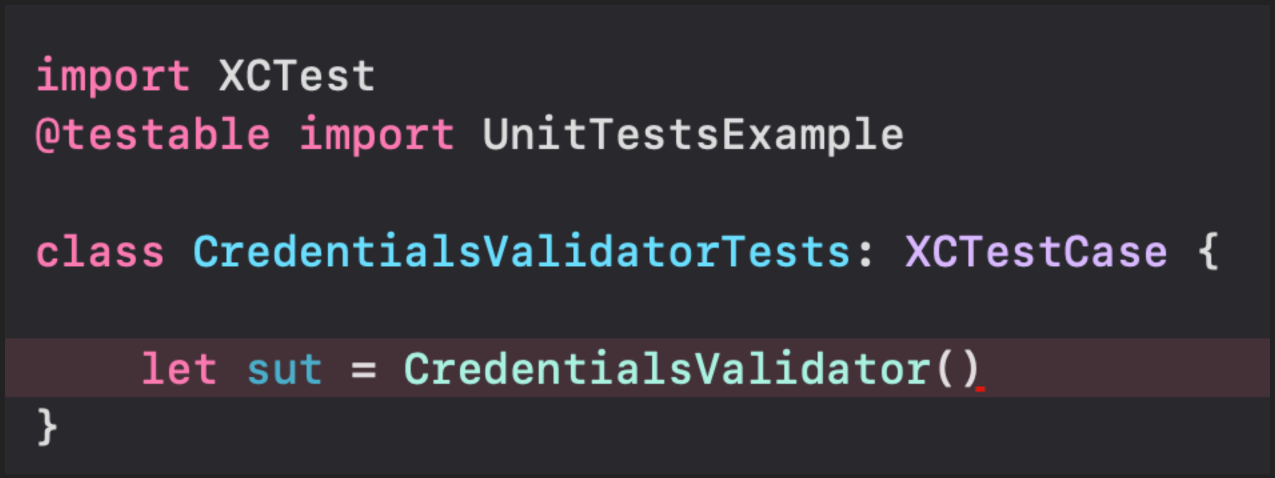
Class to be tested

UNIT TEST EXAMPLE



```swift
class CredentialsValidator {

    struct Constants {
        static let minimumPasswordLenght = 8
    }

    func check(_ password: String) -> Bool {
        guard password.count >= Constants.minimumPasswordLenght else {
            return false
        }

        return true
    }
}
```

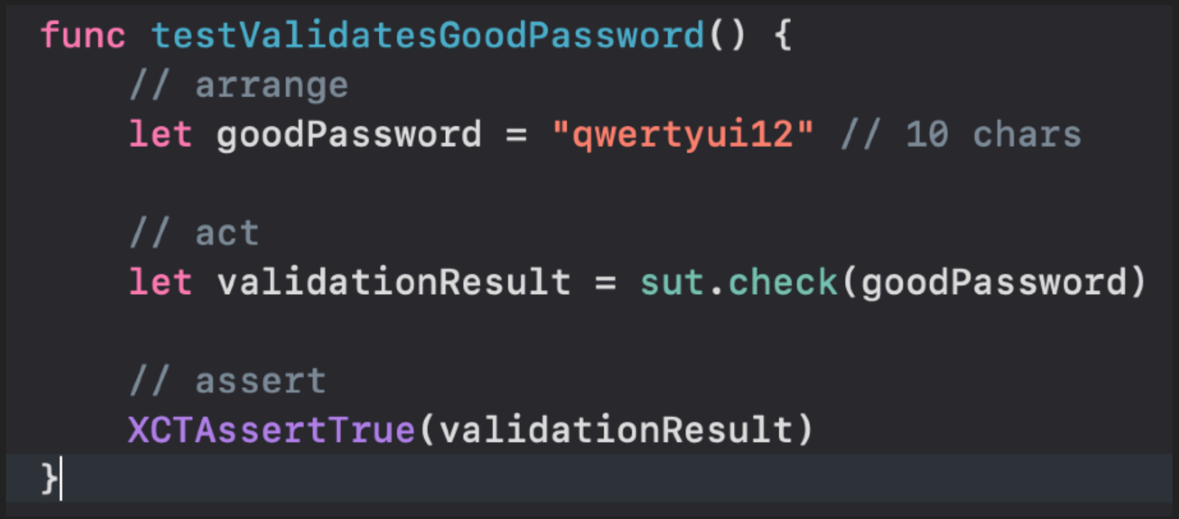
Class to be tested



```swift
import XCTest
@testable import UnitTestsExample

class CredentialsValidatorTests: XCTestCase {

    let sut = CredentialsValidator()
}
```

Unit test base

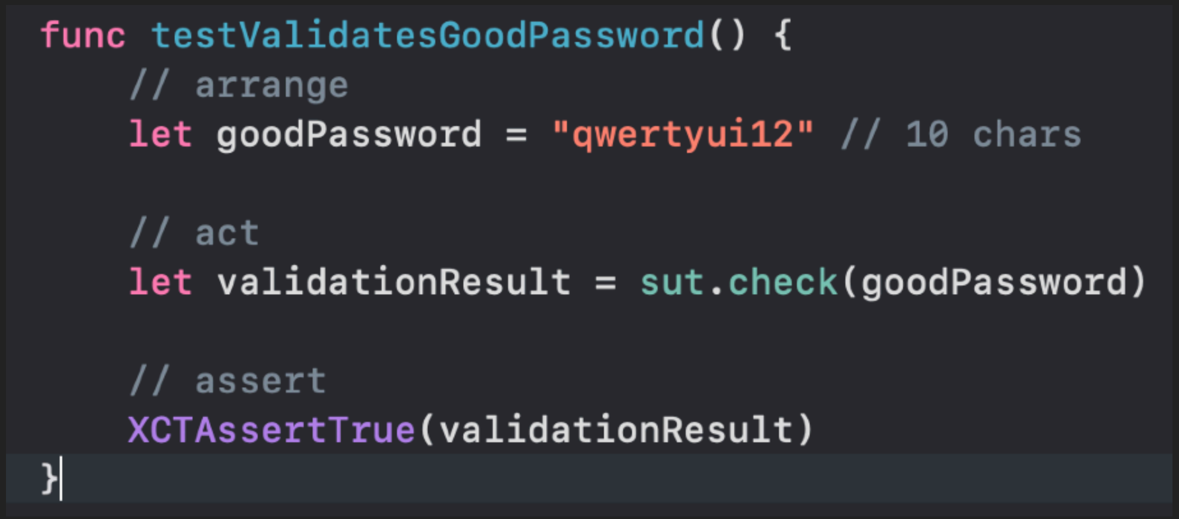
UNIT TEST EXAMPLE



```swift
func testValidatesGoodPassword() {
    // arrange
    let goodPassword = "qwertyui12" // 10 chars

    // act
    let validationResult = sut.check(goodPassword)

    // assert
    XCTAssertTrue(validationResult)
}
```

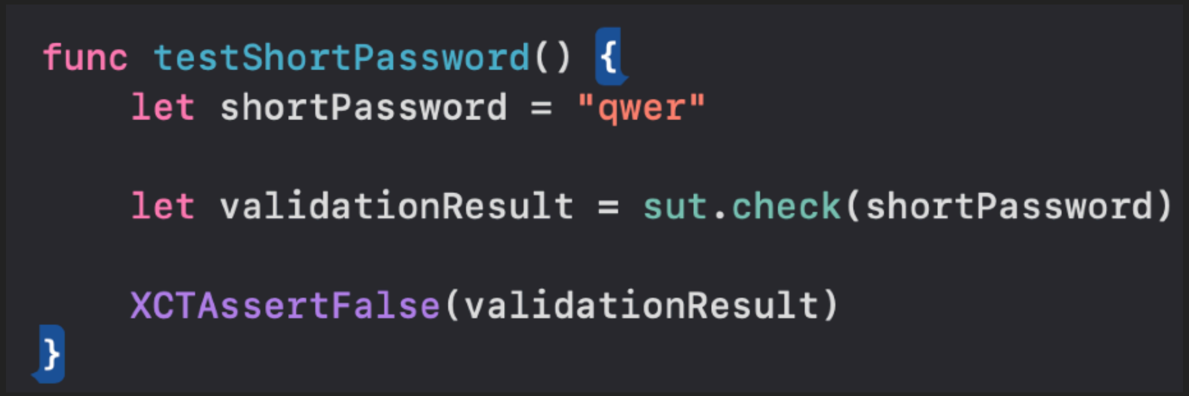
Happy path test

UNIT TEST EXAMPLE



```swift
func testValidatesGoodPassword() {
    // arrange
    let goodPassword = "qwertyui12" // 10 chars

    // act
    let validationResult = sut.check(goodPassword)

    // assert
    XCTAssertTrue(validationResult)
}
```

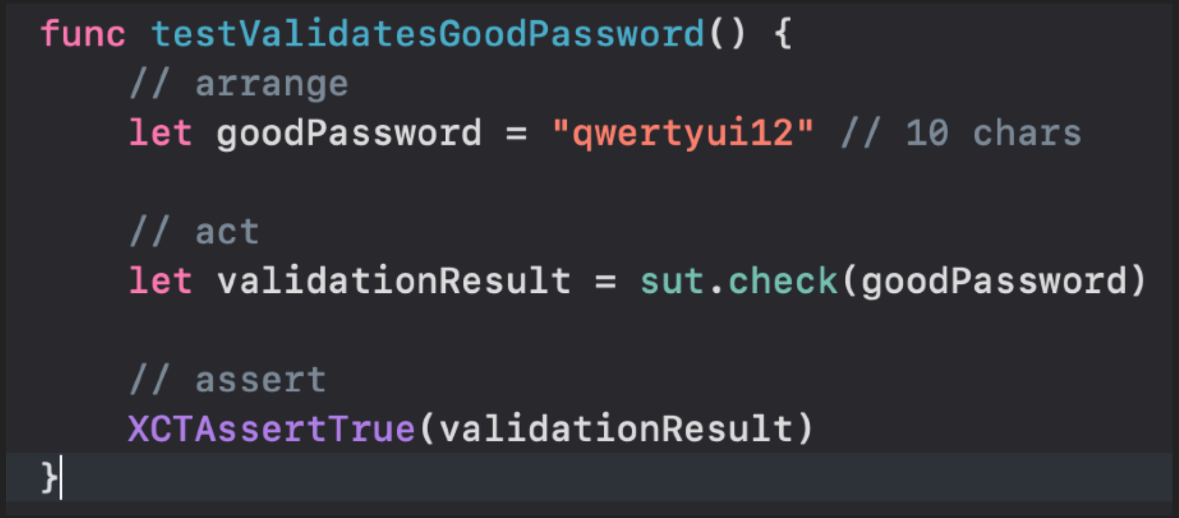
Happy path test



```swift
func testShortPassword() {
    let shortPassword = "qwer"

    let validationResult = sut.check(shortPassword)

    XCTAssertFalse(validationResult)
}
```

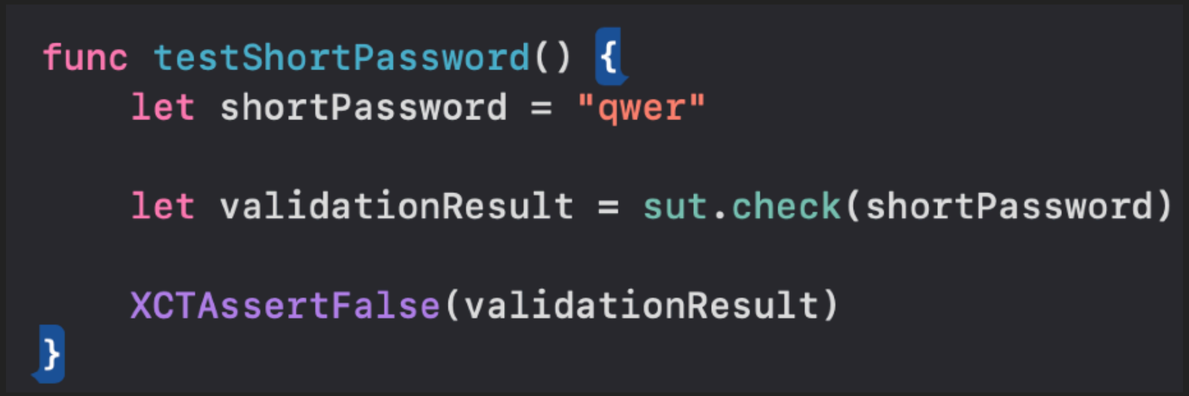
Negative test

UNIT TEST EXAMPLE



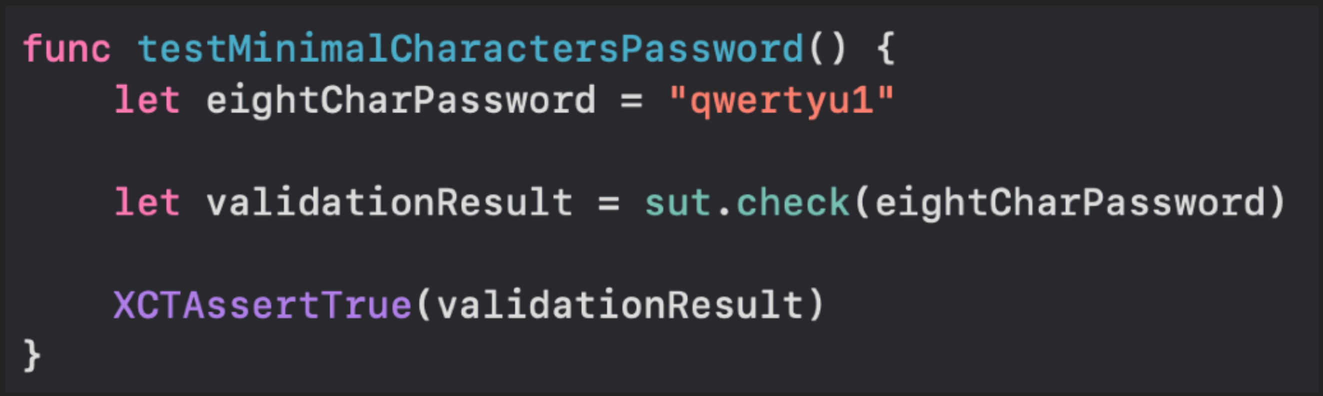
```swift
func testValidatesGoodPassword() {
    // arrange
    let goodPassword = "qwertyui12" // 10 chars

    // act
    let validationResult = sut.check(goodPassword)

    // assert
    XCTAssertTrue(validationResult)
}
```

Happy path test



```swift
func testShortPassword() {
    let shortPassword = "qwer"

    let validationResult = sut.check(shortPassword)

    XCTAssertFalse(validationResult)
}
```

Negative test



```swift
func testMinimalCharactersPassword() {
    let eightCharPassword = "qwertyu1"

    let validationResult = sut.check(eightCharPassword)

    XCTAssertTrue(validationResult)
}
```

Boundary test

UNIT TEST STRUCTURE

UNIT TEST STRUCTURE

- 1. Prepare initial state of tested class and prepare input data (arrange or given part)

UNIT TEST STRUCTURE

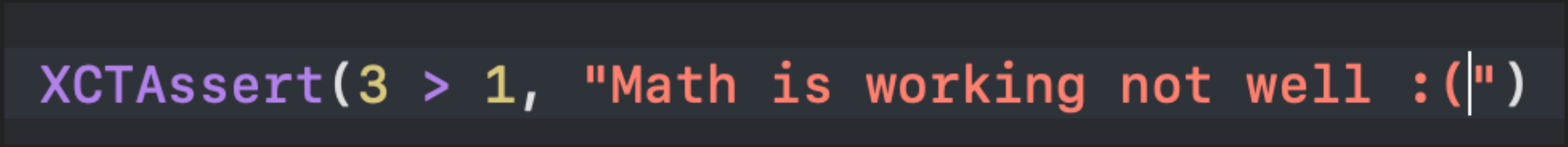
- 1. Prepare initial state of tested class and prepare input data (arrange or given part)
- 2. Call function that you want to test and see its results (act or when part)

UNIT TEST STRUCTURE

- 1. Prepare initial state of tested class and prepare input data (arrange or given part)
- 2. Call function that you want to test and see its results (act or when part)
- 3. Compare result with expectations and verify that function worked well (assert or then part)

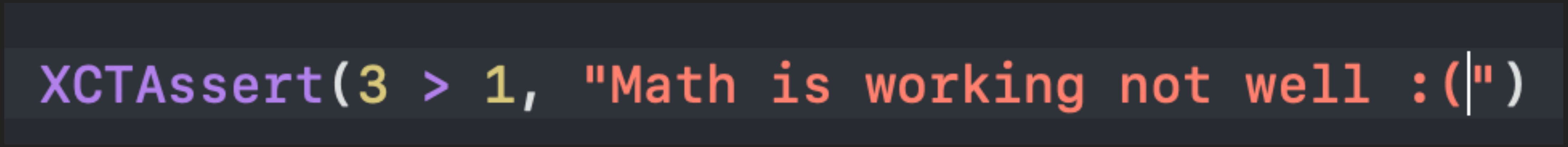
ASSERTION FUNCTIONS

ASSERTION FUNCTIONS

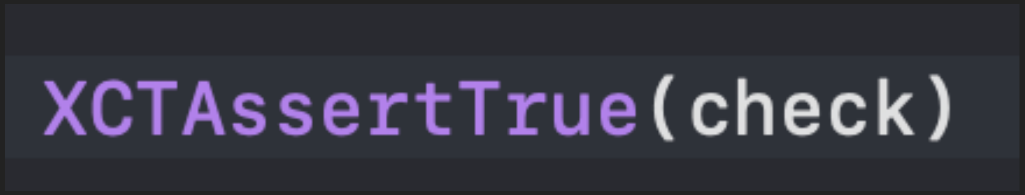


```swift
XCTAssert(3 > 1, "Math is working not well :(")
```

ASSERTION FUNCTIONS



```
XCTAssert(3 > 1, "Math is working not well :(")
```



```
XCTAssertTrue(check)
```

ASSERTION FUNCTIONS

```
XCTAssert(3 > 1, "Math is working not well :(")
```

```
XCTAssertTrue(check)
```

```
XCTAssertFalse(check)
```

ASSERTION FUNCTIONS

```
XCTAssert(3 > 1, "Math is working not well :(")
```

```
XCTAssertTrue(check)
```

```
XCTAssertFalse(check)
```

```
XCTAssertEqual(expectedResult, actualResult)
```

ASSERTION FUNCTIONS

```
XCTAssert(3 > 1, "Math is working not well :(")
```

```
XCTAssertTrue(check)
```

```
XCTAssertFalse(check)
```

```
XCTAssertEqual(expectedResult, actualResult)
```

```
XCTAssertNil(actualResult)
```

ASSERTION FUNCTIONS

```
XCTAssert(3 > 1, "Math is working not well :(")
```

```
XCTAssertTrue(check)
```

```
XCTAssertFalse(check)
```

```
XCTAssertEqual(expectedResult, actualResult)
```

```
XCTAssertNil(actualResult)
```

And others

WHAT TO TEST?

WHAT TO TEST?

- Main logic (business)

WHAT TO TEST?

- Main logic (business)
- Model classes

WHAT TO TEST?

- Main logic (business)
- Model classes
- Model transformations

WHAT TO TEST?

- Main logic (business)
- Model classes
- Model transformations
- Edge cases (very long input, invalid input)

WHAT TO TEST?

- Main logic (business)
- Model classes
- Model transformations
- Edge cases (very long input, invalid input)
- Legacy code

WHY WE SHOULD WRITE UNIT TESTS?

WHY WE SHOULD WRITE UNIT TESTS?

- It gives us confidence that code is working correctly

WHY WE SHOULD WRITE UNIT TESTS?

- It gives us confidence that code is working correctly
- We will have less bugs

WHY WE SHOULD WRITE UNIT TESTS?

- It gives us confidence that code is working correctly
- We will have less bugs
- It allows refactoring existing code easily without fear of breaking anything

WHY WE SHOULD WRITE UNIT TESTS?

- It gives us confidence that code is working correctly
- We will have less bugs
- It allows refactoring existing code easily without fear of breaking anything
- It can serve as documentation to other developers

WHY WE SHOULD WRITE UNIT TESTS?

- It gives us confidence that code is working correctly
- We will have less bugs
- It allows refactoring existing code easily without fear of breaking anything
- It can serve as documentation to other developers
- Force us to write code with good quality

WHY WE SHOULD WRITE UNIT TESTS?

- It gives us confidence that code is working correctly
- We will have less bugs
- It allows refactoring existing code easily without fear of breaking anything
- It can serve as documentation to other developers
- Force us to write code with good quality
- Easy to test a lot of cases and save time on manual testing

HOW TO SELL TO CUSTOMER?

HOW TO SELL TO CUSTOMER?

- Yes, it requires additional time, but it is perfect investment in future

HOW TO SELL TO CUSTOMER?

- Yes, it requires additional time, but it is perfect investment in future
- It makes improving quality of code really easy, because we can refactor code safely + adding new features will produces less bugs

HOW TO SELL TO CUSTOMER?

- Yes, it requires additional time, but it is perfect investment in future
- It makes improving quality of code really easy, because we can refactor code safely + adding new features will produces less bugs
- It saves time for manual testing and can give instant feedback about broken code

FIRST TESTING PRINCIPLES

FIRST TESTING PRINCIPLES

- Fast - unit tests should be very fast, so you can run thousands of them in second

FIRST TESTING PRINCIPLES

- Fast - unit tests should be very fast, so you can run thousands of them in second
- Isolated / Independent - class should be tested without any external dependencies (such as database, networking and others). Those should be controlled with special techniques. Also order of tests execution shouldn’t affect results - each test should have clear state

FIRST TESTING PRINCIPLES

- Fast - unit tests should be very fast, so you can run thousands of them in second
- Isolated / Independent - class should be tested without any external dependencies (such as database, networking and others). Those should be controlled with special techniques. Also order of tests execution shouldn't affect results - each test should have clear state
- Repeatable - test should give same result after each execution and always succeed or fail

FIRST TESTING PRINCIPLES

- Self validating - we shouldn’t analyse result of test executing - it should only show if code works as expected or it has some problem

FIRST TESTING PRINCIPLES

- Self validating - we shouldn’t analyse result of test executing - it should only show if code works as expected or it has some problem
- Thorough - you should test happy path, negative and edge cases, so code behaves well in any conditions

WHAT IS TDD?

WHAT IS TDD?

- Test driven development - is technique when you write tests first and only then main code

WHAT IS TDD?

- Test driven development - is technique when you write tests first and only then main code
- Main idea is that you write failing test first, then you write code to make test pass and then you refactor it to have good code

WHAT IS TDD?

- Test driven development - is technique when you write tests first and only then main code
- Main idea is that you write failing test first, then you write code to make test pass and then you refactor it to have good code
- It is also connected to “Red->Green->Refactor” scheme

WHAT ARE MOCKS?

WHAT ARE MOCKS?

- Mocks are fake implementation of class dependencies in order to have control over them

WHAT ARE MOCKS?

- Mocks are fake implementation of class dependencies in order to have control over them
- You can mock database and use in-memory implementation instead of using writing and reading form file which will slow down unit tests execution a lot

WHAT ARE MOCKS?

- Mocks are fake implementation of class dependencies in order to have control over them
- You can mock database and use in-memory implementation instead of using writing and reading form file which will slow down unit tests execution a lot
- You can mock networking layer and not make network calls in unit tests only returning predefined responses

WHAT ARE MOCKS?

- Mocks are fake implementation of class dependencies in order to have control over them
- You can mock database and use in-memory implementation instead of using writing and reading form file which will slow down unit tests execution a lot
- You can mock networking layer and not make network calls in unit tests only returning predefined responses
- You can not rely on system frameworks and return only predefined values

WHEN NOT TO UNIT TEST?

WHEN NOT TO UNIT TEST?

- Unit tests are not good idea when you are working in start-up and you want to release your product as soon as possible. In that conditions code quality won’t be the best and will be changing a lot. So you’ll need to rewrite your unit tests every time as well which is total waste of time