Department of Computing
COVENTRY UNIVERSITY

304CEM

Promises & Unit Testing

Promises

Coventry University

Callbacks

Remember that functions are *first class objects* in JS. So they can be used as parameter values (“callbacks”) for input to other functions.

```
functionsome_function(param, callback) {
  // async code here - may take a while
  callback(param);
}
some_function("yay",function(value) {
  console.log("callback called! "+ value);
});
```

- Second input to some_function() is a callback function

Nested Callbacks

But what if some_function()'s *callback* uses a callback as one of its arguments?

- We get a *nested callback*
- Unfortunately this pattern can continue for several layers

```
some_function(param,function(err, res) {
  some_function2(param,function(err, res) {
    some_function3(param,function(err, res) {
      some_function4(param,function(err, res) {
        some_function5(param,function(err, res) {
          some_function6(param,function(err, res) {
            // do something useful
          });
        });
      });
    });
  });
});
```

The “Problem”

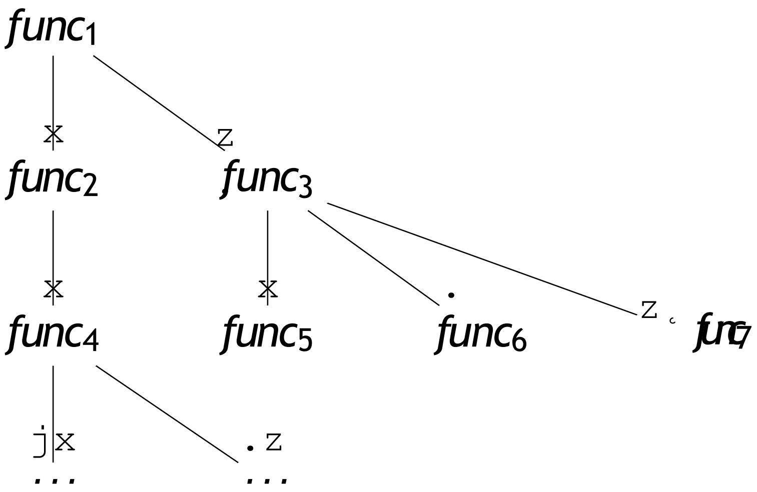
- JS is an event-based language
- So even in moderately complex programs various *chains of events* need to be handled
- Using callbacks to do this (with nesting) makes the code very hard to read
  - Code that is hard to read is hard to debug
  - Code that is hard to read is hard to refactor
  - Code that is hard to read is hard to collaborate on with colleagues

Best case: the function calls are *linear* down the chain - i.e. each function has at most one callback.

$$func_1 \to func_2 \to func_3 \to \ldots \to func_n$$

The Real Problem

Common case: the function calls are *branched* down the chain - i.e. at least one function has two or more callbacks.



Branching Callbacks

Real problems!

- If the callbacks branch, we can’t know the order of the function calls
- It can be very complex to “reassemble” data returned from the various branches correctly
- Scoping becomes a challenge

Possible solutions?

1. Avoid *anonymous* function callbacks. Replace them with (un-nested) named functions defined in their own blocks. But:
   - It is easier but still difficult to read the “meaning” or “intention” of the code
   - Branching and scoping are still challenges
2. Use event listeners when possible

```
varimg1 =document.querySelector(’ .img-1’ );
img1.addEventListener(’ load’ ,function() {
  // woo yey image loaded
});
```

- But what about events that happen before binding?
- What about *combinations of events* happening??

A Better Solution

- Use callbacks in very simple (one or two nested layers) situations
- Use listeners mainly for events that can happen *multiple times on the same object*:
    - `keyup`
    - `click`
    - etc.
- BUT otherwise use *JavaScript Promises* to handle multiple asynchronous event chains
    - in particular success/failure chains arising in AJAX calls!

Promise

An object that represents the result of an asynchronous function call

Represents a value which may be available now, or in the future, or never

How Promises Work

An executor function immediately runs and is passed with function arguments resolve and reject

The executor initiates an async task

Once completed either the resolve or reject function is called

Promise States

pending: initial state, not fulfilled or rejected.
fulfilled: meaning that the operation completed successfully.
rejected: meaning that the operation failed.

Benefits of Promises

Relief from 'callback hell'

Can use a series of simple steps

Simplified exception handling

Ability to defer the computation of the right answer until a more convenient time

Functions That Return Promises

```
const getData = base => new Promise( (resolve, reject) => {
        const url = `http://api.fixer.io/latest?symbols=${base}`
        console.log(url)
        request.get(url, (err, res, body) => {
                if (err) reject(new Error(`could not get conversion rate for ${base}`))
                resolve(JSON.parse(body))
        })
})

const cleanData = data => new Promise( resolve => resolve(data.rates.USD))
```

Promise Chain

```
getData('USD')
  .then( data => cleanData(data))
  .then( data => console.log(data))
  .catch(err => console.log(data))
```

Promise Chain with Callback

```
exports.search = (request, callback) => {
  extractParam(request, 'q')
    .then( query => searchByString(query))
      .then( data => cleanArray(request, data))
    .then( data => callback(null, data))
    .catch( err => callback(err))
}
```

Concurrent Promises

`Promise.all()`

- Takes an iterable (array) of promises
- Resolves if ALL the promises resolve
- Rejects if any of the promises reject

`Promise.race()`

- Resolves as soon as the first promise is returned

Concurrent Promises (All)

```
Promise.all(itemPromises) // from earlier slide
      .then( results => {
            results.forEach( item => {
                  console.log(item)
            })
      }).catch( err => {
           console.log(`error: ${err.message}`)
      })
```

Concurrent Promises (Race)

```
Promise.race(itemPromises)
      .then( result => {
                  console.log(result)
      }).catch( err => {
           console.log(`error: ${err.message}`)
      })
```

Department of Computing

COVENTRY UNIVERSITY

304CEM

Automated Testing

Coventry University

Outcomes

Unit Testing

Code Coverage

Acceptance Testing

Traditional Development

we write code until it 'works'

follow an agreed architectural design

Why Test

Adding features often introduces bugs

Testing needs to take place on a regular basis

- Check that our new code works
- Check we have not broken earlier code!

Unfortunately manual testing is time consuming

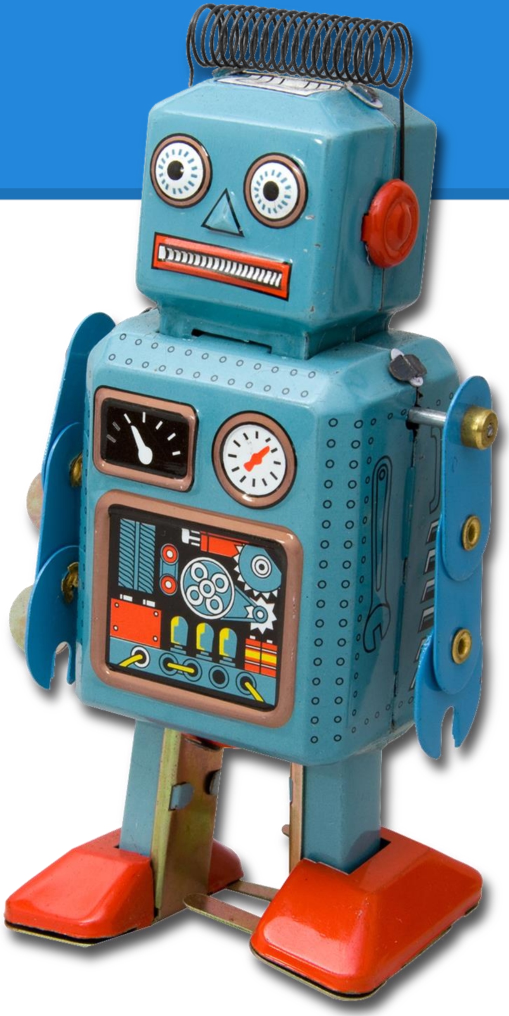
So it is often left out…

Automated Testing

Involves writing code to test your code

This code can be quickly run whenever you add a new feature

Like a robot to do all the boring stuff



Automated Testing

Run same tests repeatedly/accurately

Run a large 'suite' of tests

No human time wasted

Benefits of Automated Testing

Quick one-click testing of all the code

Prevents human error creeping in

Ensure bugs have not crept into existing code

Check that there are no bugs in new code

When all the tests pass, your software is complete!

Types of Automated Testing

Unit Testing

- Testing code logic directly (functions)
- Sometimes called ‘white-box’ testing

Acceptance Testing

- Testing the application as a ‘user’
- Sometimes called ‘black-box’ testing

How Many Tests?

Cover all routes in code
Test median, edge and bad data cases
If in doubt add more tests…

Check code coverage

Unit Testing

Components:

- An Assert Module
  - defines the tests to carry out
    - The built-in NodeJS Assert module
- A Test Runner
  - Carries out the tests and reports the results
    - The Jasmine package

Test Runner

A program that runs the tests

In these examples we will be using Jasmine

a nodejs package

Regression Testing

Are we introducing bugs in previously working code?

All existing code should be supported by exhaustive tests

Even if a test is passed we should always carry it out

All tests run every time code changes

Fixing Bugs

Write a test to replicate the bug
Test should of course fail every time it runs

Try to fix the bug
Bug is fixed when the test(s) pass

Test name should match bug name/code

Test-Driven Development

start by defining the goal

unit tests are both specification and documentation

we define our functionality through the tests

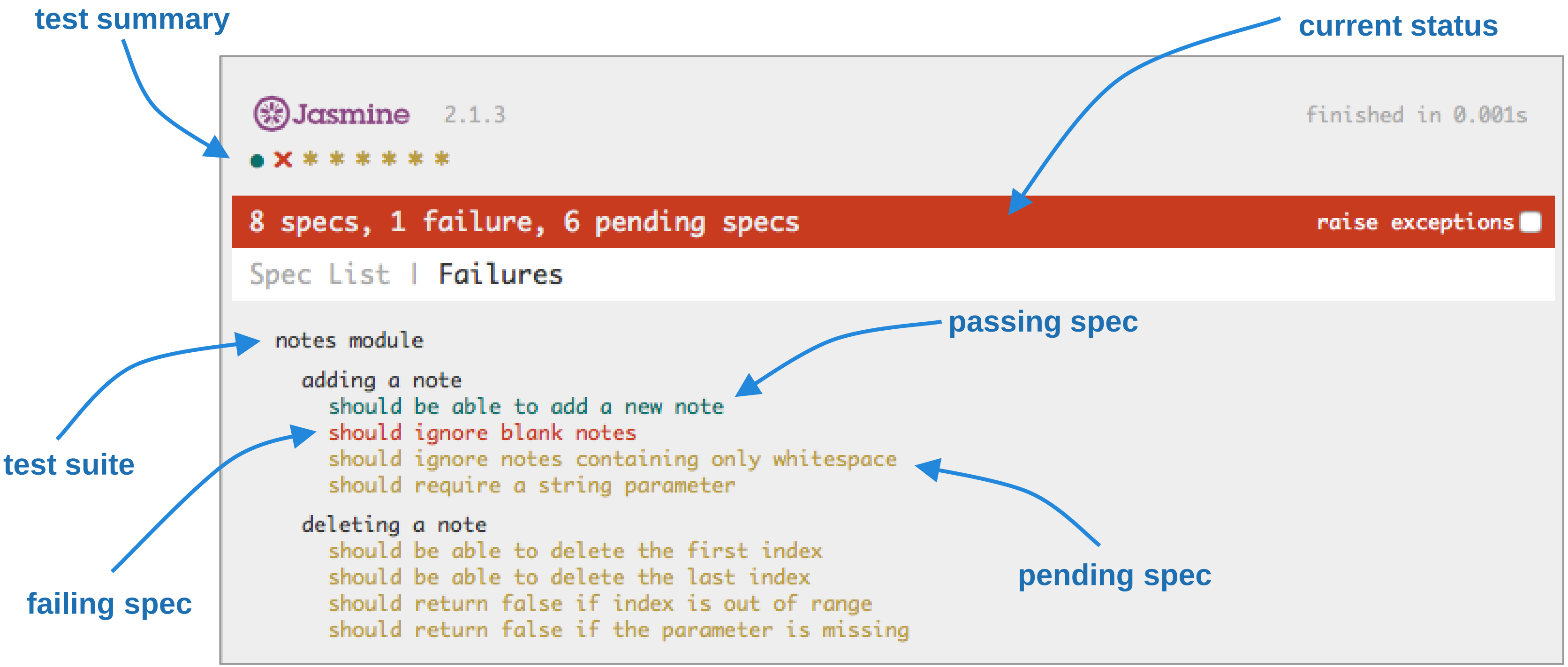
we write our tests before starting to 'code'

forces us to think about *what* we are writing

Test-Driven Development

Test runner
Test suites
Specs
Expectations
Matchers
Setup & teardown
Pending specs
Nesting suites

Jasmine Test Runner



Jasmine 2.1.3 finished in 0.001s

● ✕ * * * * * *

8 specs, 1 failure, 6 pending specs raise exceptions □

Spec List | Failures

```
notes module
  adding a note
    should be able to add a new note
    should ignore blank notes
    should ignore notes containing only whitespace
    should require a string parameter
  deleting a note
    should be able to delete the first index
    should be able to delete the last index
    should return false if index is out of range
    should return false if the parameter is missing
```

test summary

current status

passing spec

test suite

failing spec

pending spec

Suites

Describe Your Tests

Begins with a call to the global Jasmine function **describe**

```
describe('notes module', function () {
 // specs go here
});
```

Specs

Defined by calling the global Jasmine function `it`, which takes a string and a function.

Contains one or more expectations that test the state of the code

```
it('should be able to add a new note', function () {
    // expectations go here
});
```

All expectations must pass for the spec to pass

Expectations & Matchers

An expectation in Jasmine is an assertion that is either true or false

- Built with the function `expect()` that takes a value called an *actual*
- Chained with a *matcher function* that takes an *expected*

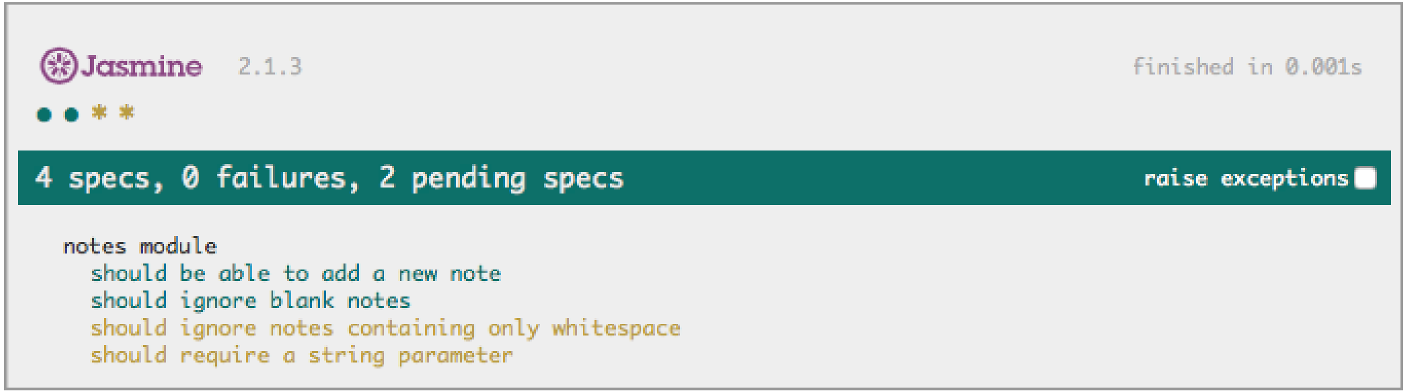
```
expect(notes.add('sixth note')).toBe(true);

expect(notes.count()).toBe(6);
```

Setup & Teardown

Ensures system is in a known state before running tests

```
    beforeAll()
    afterAll()
    beforeEach()
    afterEach()
```

Pending Specs



```
4 specs, 0 failures, 2 pending specs

  notes module
    should be able to add a new note
    should ignore blank notes
    should ignore notes containing only whitespace
    should require a string parameter
```

Flagging Pending Specs

Means they don’t get run:

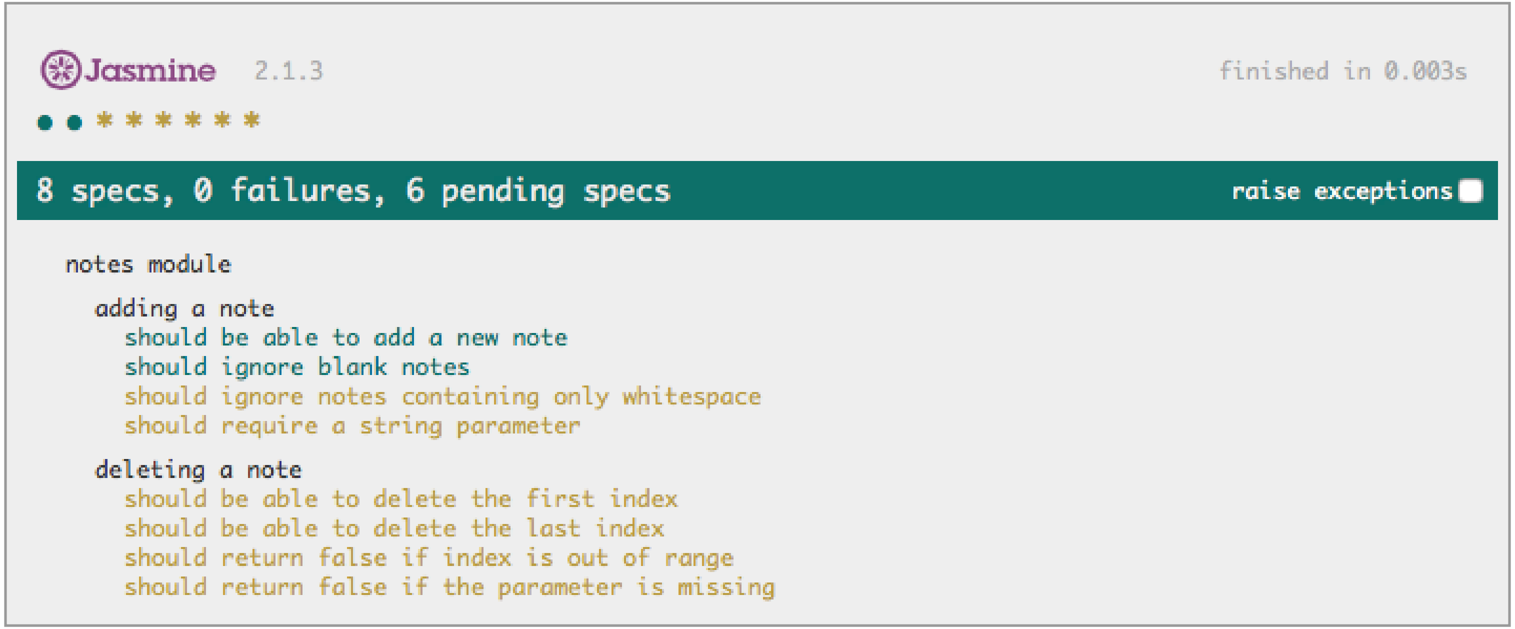
- Any spec without a second parameter

```
it('should ignore blank notes');
```

- Any spec calling the `pending()` function

```
it('should ignore blank notes', function() {
    expect(notes.add('')).toBe(false);
    expect(notes.count()).toBe(5);
    pending();
});
```

Nesting Suites



```
8 specs, 0 failures, 6 pending specs

notes module
  adding a note
    should be able to add a new note
    should ignore blank notes
    should ignore notes containing only whitespace
    should require a string parameter
  deleting a note
    should be able to delete the first index
    should be able to delete the last index
    should return false if index is out of range
    should return false if the parameter is missing
```

Nesting Testing Suites

```
describe('notes module', function () {

    beforeEach(function() {});

    describe('adding a note', function() {
        it('should be able to add a new note');
        it('should ignore blank notes');
    });
    describe('deleting a note', function() {
        it('should be able to delete the first index');
        it('should be able to delete the last index');
    });
});
```

Common Matchers

```
expect(a).toBe(b);
expect(a).toEqual(12);
expect(a).toMatch(/bar/);                      (for regular expressions)
expect(a).toBeDefined();
expect(a).toBeUndefined();
expect(a).toBeNull();
expect(a).toBeTruthy();
expect(a).toBeFalsy();
expect(a).toContain("bar");                    (finding an item in an array)
expect(e).toBeLessThan(4);
expect(a).toBeGreaterThan(2);
expect(a).toBeCloseTo(40, 2);
```

Code Coverage

How much of the code is being tested?
Are all code branches being tested?
How many times are each line tested?

Helps identify any blind-spots

also consider range of test data...

Coverage Criteria

Function coverage

Has each function in the program been called?

Statement coverage

Has each statement in the program been executed?

Branch coverage

Has each branch of each control structure been executed?

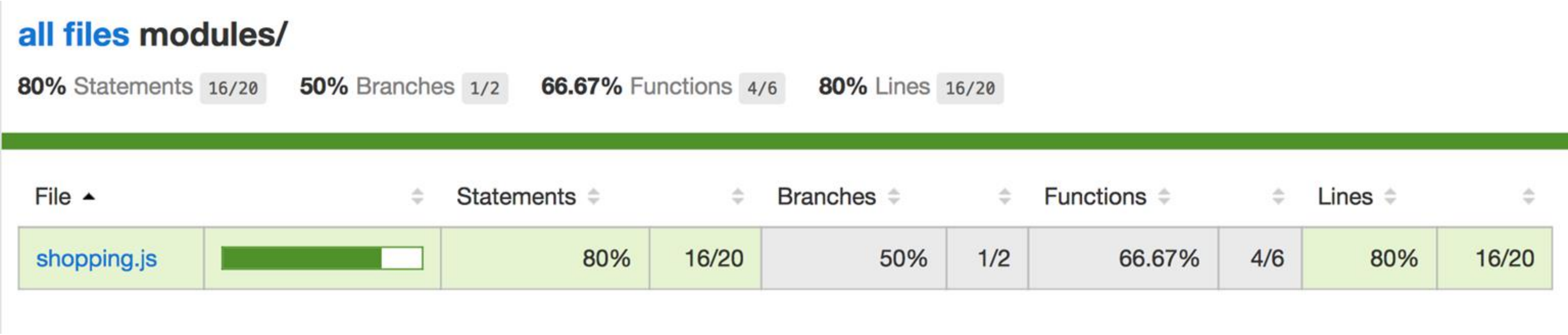
Condition coverage (or predicate coverage)

Has each Boolean sub-expression evaluated both to true and false?

Code Coverage

```
=============================== Coverage summary ===============================
Statements   : 80% ( 16/20 )
Branches     : 50% ( 1/2 )
Functions    : 66.67% ( 4/6 )
Lines        : 80% ( 16/20 )
================================================================================
```

Code Coverage Summary

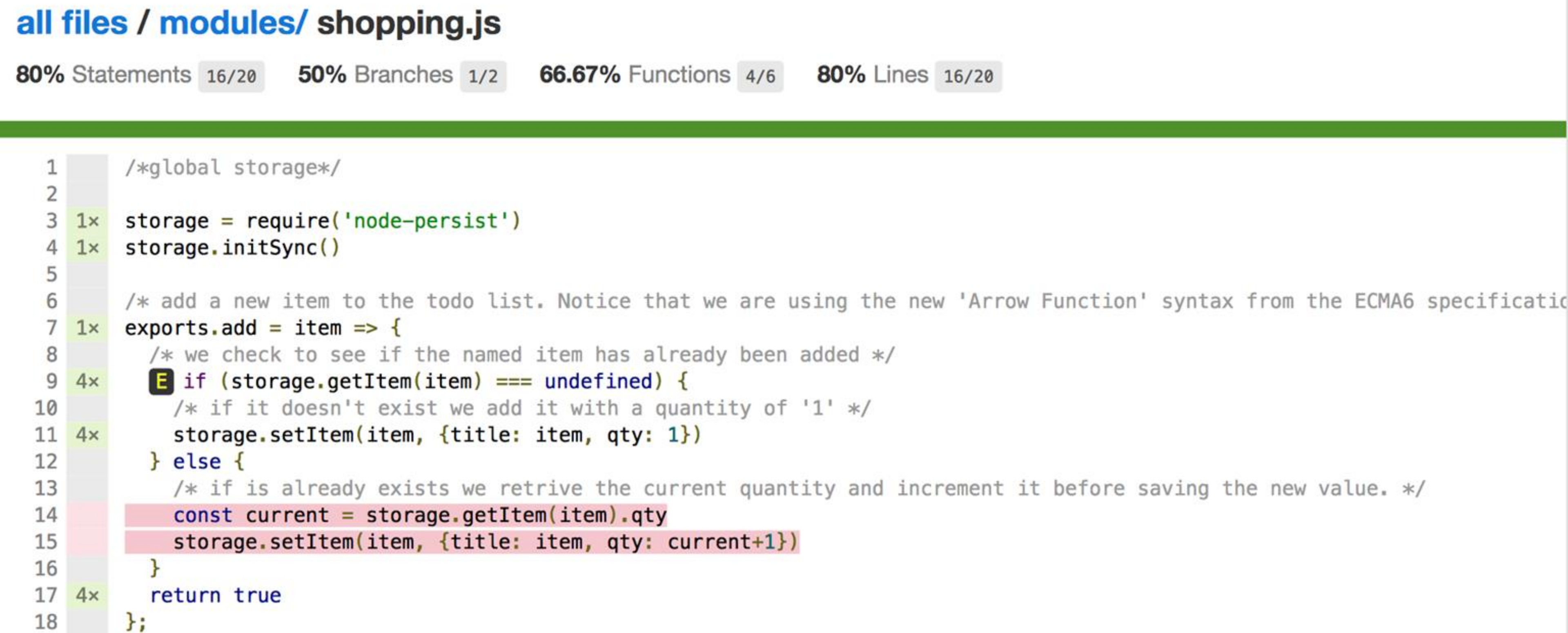


all files modules/

80% Statements `16/20` **50%** Branches `1/2` **66.67%** Functions `4/6` **80%** Lines `16/20`

File ▲		Statements		Branches		Functions		Lines	
shopping.js		80%	16/20	50%	1/2	66.67%	4/6	80%	16/20

Code Coverage Details



all files / modules/ shopping.js

80% Statements 16/20   50% Branches 1/2   66.67% Functions 4/6   80% Lines 16/20

```
      /*global storage*/

1×    storage = require('node-persist')
1×    storage.initSync()

      /* add a new item to the todo list. Notice that we are using the new 'Arrow Function' syntax from the ECMA6 specificati
1×    exports.add = item => {
        /* we check to see if the named item has already been added */
4×      E if (storage.getItem(item) === undefined) {
          /* if it doesn't exist we add it with a quantity of '1' */
4×        storage.setItem(item, {title: item, qty: 1})
        } else {
          /* if is already exists we retrive the current quantity and increment it before saving the new value. */
          const current = storage.getItem(item).qty
          storage.setItem(item, {title: item, qty: current+1})
        }
4×      return true
      };
```

Workflow

Three distinct steps:

1. write an automated test to define an improvement or new function
2. produce enough code to pass the test
3. refactor (clean up) the code without breaking the tests.

TDD Workflow

Write the tests first (they will fail)

Write enough code to pass the tests

- simplest solution that could possibly work

Commit the working code

Refactor the code to improve

Comments/documentation

Commit (again)

TDD Benefits

Working code
Enforcing single responsibility
Conscious development
Improved productivity

Testing in Assignment

- Demonstrate you have used testing
  - Unit testing
  - UI Testing (more advanced)
- Code coverage
- Used in debugging