Testing

...and the value thereof

Mathematics is the rigor of most engineering disciplines. Structural engineers, mechanical engineers, aerospace engineers all make copious use of calculus, differential equations, linear algebra and so forth. They don't just "throw stuff together".

By contrast, although computer science itself has deep roots in mathematics, the actual act of software engineering rarely makes direct use of math in the same way that other engineering disciplines do.

That given, how do programmers ensure that the systems they build are sound? Typically there aren't a lot of calculations before a programmer builds a sub-system.

The primary practical answer for most kinds of software development is...

Testing is the engineering rigor of software engineering.

When we say "testing" here, we do NOT mean manual testing where a person clicks around and drives the program's user interface with their own mouse and keyboard.

We are talking about AUTOMATED TESTS, in particular the style called a UNIT TEST.

An automated test, as the name suggests, runs automatically.

The point of an automated test is to capture the manual steps and human discernment that would be applied by a human tester and run them at high speed and with perfect fidelity over and over.

In a modern, professionally-built software system, there are typically hundreds or thousands of tests that run routinely, whenever someone checks in their code to the source code management system (nowadays, often Github).

What's a "unit test"?

A reliable system is composed of reliable components. A Lego block doesn't suddenly change shape on you. It maintains a clear and unchanging interface. You can rely on that interface as you snap that Lego block together with other Lego blocks, and thereby build up a very complex block structure.

The components of your program need to behave similarly to Lego blocks. If they honor their contracts with each other, then you can build up a very complex software system with them. If they don't, however, then everything will turn into an unmaintainable mess.

How do you know that each component in your software system (your program) is honoring its interface properly?

That's what unit tests do. Each "unit" is a component of your system. It's a Lego block.

For our purposes here, the most fundamental UNIT in your programs right now is a FUNCTION.

When should I unit test?

You should be unit-testing any non-trivial functions in your programs. You may have to bend this rule somewhat in order to meet time constraints in a programming interview/admissions assessment context, but on the whole it is a dangerous practice to avoid testing. You are just as likely to end up burning more time than you save by not testing, because when you drive at high speed without guardrails, it's easy to plunge off a cliff into debugging hell.

It is extremely easy to confuse ourselves with accidental complexity as we write programs. Testing each component as we go helps to ensure that we keep track of what we are building.

A warning sign that you should be testing

If you ever find yourself sprinkling `console.log` statements all over the place to figure out what on earth your program is doing anymore, you probably could have used some small tests to verify things better as you go.

Assertions

Unit testing is built around the concept of ASSERTIONS.

Programmers ASSERT that such-and-such is TRUE about the unit under consideration.

Let's take a small, real example:

```
function square(x) {

  return x * x;

}
```

A very simple assertion would look like this:

```
console.log(square(5) === 25);
```

- If `square` is given the input `5` and returns the output `25`, then this will print `true`.
- However if `square` is given the input `5` and returns the output `125`, then this will print `false`.

Test frameworks

It's not a problem to write small assertions about your code in this informal manner, via `console.log`, when you are first starting out. It's a lot better than not exercising your code at all, and having to debug a bunch of code when you finally do run your program.

However, professional programmers use a testing framework that formalizes the concept of assertion, and provides well-formatted test output so you can quickly see where problems have arisen in your code.

A typical such assertion might look like this:

```
var output = square(5);
```

```
expect(output).to.equal(25);
```

Right now, you don't need to worry about using a fully-featured testing framework. The above code might not make sense as is; for more information, please feel free to research "Chai Assertion Library".

However, even early in your journey to becoming a professional programmer, there is some value to being able to write an assertion a little more formally than just sprinkling `console.log` into your code. You can make a simple assertion helper function yourself trivially, something like this:

```
function assertEqual(actual, expected, testName) {

  if (actual === expected) {

    console.log('passed');

  } else {

    console.log('FAILED ' + testName + ': Expected ' + expected + ', but got ' +
actual);

  }

}
```

Then you can call that instead of `console.log` directly, inside of each of your tests.

Terminology note: Assertions are not unit tests!

An assertion is a statement that something is true or false about the behavior of your program.

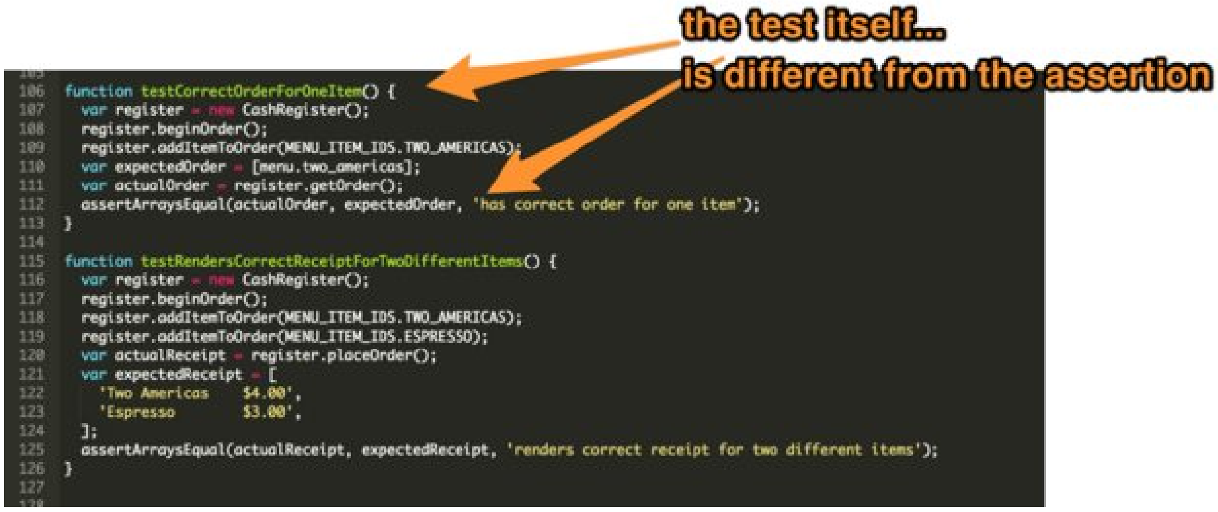
However, in order to be able to make an assertion, you usually have some setup to do.

You prepare some input, and an expected output. If there are classes involved then you usually have to instantiate an instance of that class. You may have other context to set up before you can make the assertion -- let's say you're asserting something about step #3 of a 4-step flow.

Then you have to run through steps 1 and 2 to prepare to evaluate the result of running step 3.

All that setup, outside of the assertion itself, is still part of your test.

That's why we're being careful here to distinguish assertions from the unit tests around those assertions!



```
function testCorrectOrderForOneItem() {
  var register = new CashRegister();
  register.beginOrder();
  register.addItemToOrder(MENU_ITEM_IDS.TWO_AMERICAS);
  var expectedOrder = [menu.two_americas];
  var actualOrder = register.getOrder();
  assertArraysEqual(actualOrder, expectedOrder, 'has correct order for one item');
}

function testRendersCorrectReceiptForTwoDifferentItems() {
  var register = new CashRegister();
  register.beginOrder();
  register.addItemToOrder(MENU_ITEM_IDS.TWO_AMERICAS);
  register.addItemToOrder(MENU_ITEM_IDS.ESPRESSO);
  var actualReceipt = register.placeOrder();
  var expectedReceipt = [
    'Two Americas    $4.00',
    'Espresso        $3.00',
  ];
  assertArraysEqual(actualReceipt, expectedReceipt, 'renders correct receipt for two different items');
}
```

How do I decide what to test?

Here is a fuller set of assertions we might make about the `square` function:

1. Given `-5` as input, it should return `25` as output.
2. Given `0` as input, it should return `0` as output.
3. Given `5` as input, it should return `25` as output.
4. Given `0.25` as input, it should return `0.0625` as output.

This seems like a natural set of tests. Why do they seem like a reasonable set of tests, though? They didn't come out of nowhere, of course. To clarify that decision-making process, let's look at the premises that we have about squaring that underlie this choice of tests:

1. Negative numbers become positive when squared.
2. 0 squared is 0.
3. A number squared is that number times itself.
4. A fractional number becomes smaller when squared, because a fraction of a fraction is smaller.

Whenever you are testing one of your functions, you want to similarly apply CATEGORICAL REASONING to identify test cases that sufficiently explore the full contract of the code under test.

Note: Your tests go *outside* the code that you're testing

You should never put your test *inside* the code that you are testing (aka "the code under test").

Your tests should treat each function as a "black box".

They supply inputs and assert expectations about the resulting outputs.

Bad:

```
function decorateClassListWithAges(classList) {

  var classListWithAges = classList.map(function(student) {

    return {'name': student, 'age': getRandomIntInclusive(10, 15)}

  });
  var checkAge = assertRange(classListWithAges[0].age, 10, 15,

    'check age is between 10 and 15');

  return classListWithAges;

};
```

How do I write unit tests?

There are a few different styles you can use in this phase of your learning, and in an interview/admissions assessment context (later, as you head towards full-on professional programming, you will learn more).

"Inline console logs", no assert helpers style

You can just drop a direct `console.log` statement if you are really pressed for time. No `assertEquals` type of helper functions, just raw logging.

This is not as tidy as using a formal assert function, but will do in a pinch.

This is the most verbose and repetitive and can rapidly get really messy! So, beware. BUT if you have a very small program and very simple assertions, you can use this judiciously to make rapid progress and keep moving.

```
function decorateClassListWithAges(classList) {

  // some implementation here

}
```

```
// TESTS

var classList = ['Brittany', 'Asheley', 'Magee'];

var classListWithAges = decorateClassListWithAges(classList);

var output = classListWithAges[0].age;

var isValid = output >= 10 && output <= 15;

console.log('check age of first student is between 10 and 15: ' + isValid);
```

"Named test functions" style

If you want to be extra-tidy, make a distinct, named test function for each test.

This is very clean and professional, but you need to type fast to make this work well in an interview/admissions assessment context :) Please, please go for it though if you can make it work!

```
function decorateClassListWithAges(classList) {

  // some implementation here

}



function testDecoratesFirstStudentWithValidAge() {
  var classList = ['Jamie', 'Gilbert', 'Nicholas'];

  var classListWithAges = decorateClassListWithAges(classList);

  var checkAge = assertRange(classListWithAges[0].age, 10, 15, 'check age is between 10 and 15');

}




// Execute your test suite

testDecoratesFirstStudentWithValidAge(); // <-- the test we wrote above

testSomethingElse();

testYetAnotherThing(); // etc
```

"Inline but with assertion helpers" style

This is a compromise between the above two styles. The tests themselves do not have function wrappers, but you save some repetition by leveraging assertion helpers like `areArraysEqual` or whatever you need.

Like dropping `console.log` statements, this too can quickly get a little bit messy, so be careful. That said, you also save quite a bit of typing, and in an interview/admissions assessment context, time is of the essence.

```
function decorateClassListWithAges(classList) {

  // some implementation here
}
```

```
// TESTS

var classList = ['Huey', 'Dewey', 'Louis'];

var classListWithAges = decorateClassListWithAges(classList);

var checkAge = assertRange(classListWithAges[0].age, 10, 15, 'check age of first
student is between 10 and 15');
```

Naming Tests

When it comes to naming your unit tests, someone should be able to look at the name of your test and immediately be able to ascertain what the test actually tests.

With a properly named series of tests, it is easy for you, and other programmers, to determine that you have tested all that requires testing.

- No two tests should have the same name. Otherwise, they'd be the same test, right?
- Don't just name your test after the function that it is testing. Otherwise how can you have two tests of the same function? They'd have the same name. (see above) *What* are you testing about that function?
- Don't use the name of the function at all in your test name.

Good:

```
assertEqual(findLongestPalindrome('racecar hannah'), 'racecar', 'finds longest of two
palindromes');
```

Note that one can easily determine what this test does by prefixing the name of the test with `it`; making the sentence that describes the test read as: "It (i.e., the function being tested) finds the longest of two palindromes."

You can explicitly say "it" in your test name, if you want:

```
... 'it finds longest of two palindromes');
```

You can also use the word "should" if that feels really natural to you. Most assertion frameworks force you to use the "it" style above, though.

Okay:

```
... 'should find longest of two palindromes');
```

But don't stray too far from the above patterns. E.g.,

Bad:

```
...'my test');
```

Bad:

```
... 'findLongestPalindrome');
```

A note on testing object equality

We know that trying to compare two arrays, such as `[1, 2, 3] === [1, 2, 3]`, is not going to work. That statement winds up comparing two memory addresses of two array instances, not the contents of those arrays.

You can write a function that compares two arrays properly if you assume that their elements only contain simple values like numbers or strings (i.e., not object values like more arrays). (Simple values like that are called "scalar" values by the way.)

```
function assertArraysEqual(actual, expected, testName) {

  var areEqualValues = actual.every(function(item, i) {

    return item === expected[i];

  });

  var areEqualLength = (actual.length === expected.length);



  if (areEqualLength && areEqualValues) {

    console.log('passed');

  } else {

    console.log('FAILED [' + testName + '] Expected "' + expected + '", but got "' +
actual + '"');

  }

}
```

A handy trick is to convert those arrays to strings first, then compare the strings.

You can use a helper function called `JSON.stringify()` to do this. You can learn more about the JSON format if you want, but it's not necessary just to use this helper function.

```
function assertObjectsEqual(actual, expected, testName) {

  actual = JSON.stringify(actual);

  expected = JSON.stringify(expected);

  if (actual === expected) {

    console.log('passed');

  } else {

    console.log('FAILED [' + testName + '] Expected ' + expected + ', but got ' +
actual);

  }

}
```

HOWEVER this isn't completely reliable in all cases where you might want to compare two objects, because the keys in an object can be the same but in a different order. Key order shouldn't matter for comparing equality, but `stringify()` will consider them to be different, giving a false negative (thinking two objects don't match when really they do).

```
JSON.stringify({foo: 1, bar: 2})
```

```
"{"foo":1,"bar":2}"
```

```
JSON.stringify({bar: 2, foo: 1})
```

```
"{"bar":2,"foo":1}"
```

That said, a full implementation of "deep equality" check is way out of scope of this course.

You can glance here at a real, complete implementation, if you are curious: https://github.com/substack/node-deep-equal/blob/master/index.js .

In a unit test during a technical interview/technical admissions assessment, you completely control the "expected" value that you are comparing against. Therefore in practice it isn't a big deal to use `JSON.stringify` to do a quick comparison during an interview/admissions assessment. Be prepared to justify to your interviewer (or out loud during your admissions assessment) why it's safe for your intended use, despite being unsafe for general comparisons in production code.