TDD Example

This is to introduce you to the world of Test Driven Development using a simple example in javascript & the jasmine framework. For this example, we will be building a calculator!!

Start with a failing test (RED)

If you find yourself writing implementation code without a test, you're doing it wrong!

THINK
What is the simplest feature we know a calculator should perform that we can write a test for?

Addition! I'm pretty sure a calculator should be able to add together 1 and 1 and that the result should be equal to 2 so lets write a test for using the jasmine framework.

```
describe("a calculator", function() {
    it("can add 1 and 1 together", function () {
        expect(add(1, 1)).toEqual(2);
    });
});
```

When the jasmine web runner executes this test, we'll see it fail because no function named *add* exists. COOL!

Write the code to make the test pass (GREEN)

If you're pairing, now would be a good time to pass the keyboard!

Next step is to write the simplest implementation code to make the test pass. Here's how that might look for our calculator example:

```
function add(numberA, numberB) {
    return 2;
}
```

This time when the jasmine web runner executes the test, we'll see it pass because the *add* function is returning 2 which is what the test expects.

Refactor

At this point in the TDD cycle we should be thinking about refactoring. At the moment I can't see any interesting duplication in the test or implementation code that would be a good candidate for refactoring so I'm going to skip this step.

Write a failing test (RED)

What's the next test we could write that will drive out some functionality?

I think we should see if our add function can tell us what 2 plus 2 equals… I'm expecting it to be 4 so I'll write a test for it.

```
describe("a calculator", function() {
    it("can add 1 and 1 together", function () {
        expect(add(1, 1)).toEqual(2);
    });

    it("can add 2 and 2 together", function () {
        expect(add(2, 2)).toEqual(4);
    });
});
```

Alright! Another red test!! This time it looks like it's failing because our add function always returns 2.

Write the code to make the test pass (GREEN)

Remember to keep passing that keyboard!

OK, lets go ahead and write the simplest code that will make both our tests pass.

```
function add(numberA, numberB) {
    return numberA + numberB;
}
```

And we're green again!

Refactor

Hmm, I'm still not sure we've got enough interesting code to refactor. Lets keep going and see what happens.

Write a failing test (RED)

What's the next test we could write that will drive out some functionality?

I think addition is done, lets start looking at other functions a calculator should perform. How about subtract? Lets write a test to see if our calculator can subtract two numbers.

```
describe("a calculator", function() {

    // ... omit previous tests for brevity ...

    it("can subtract 1 from 1", function () {
        expect(subtract(1, 1)).toEqual(0);
    });
});
```

When we run the test suite, our 2 previous tests pass but the new one fails because the *subtract* function doesn't exist yet.

Write the code to make the test pass (GREEN)

Lets get to green ASAP!!

Quick, get to green now!!! GO GO GO!!!

```
function subtract(numberA, numberB) {
    return 0;
}
```

And we're back to all green!

Refactor

Really? Not sure we need to…

Write a failing test (RED)

What's the next test we could write that will drive out some functionality?

I wonder if our *subtract* function can take 2 from 2? Lets find out!

```
describe("a calculator", function() {

    // ... omit previous tests for brevity ...

    it("can subtract 1 from 1", function () {
        expect(subtract(1, 1)).toEqual(0);
    });

    it("can subtract 2 from 2", function () {
        expect(subtract(2, 2)).toEqual(0);
    });
});
```

Hey, that's interesting, the new test doesn't fail, it passes! But we're supposed to be writing a failing test… Oh yeah, the next test we write should drive out some more functionality, we should probably choose a test that will yield a different result. Lets try again:

```
describe("a calculator", function() {

    // ... omit previous tests for brevity ...

    it("can subtract 1 from 1", function () {
        expect(subtract(1, 1)).toEqual(0);
    });

    it("can subtract 2 from 4", function () {
        expect(subtract(2, 4)).toEqual(2);
    });
});
```

Great, it's failing! On to the next step.

Write the code to make the test pass (GREEN)

Lets get to green ASAP!!

Alright, lets implement the *subtract* function so that it will pass for both tests.

```
function subtract(numberA, numberB) {
    return numberA - numberB;
}
```

What?! Why is this failing?! Oh yeah, we've implemented it to subtract numberB from numberA but the test is expecting us to subtract numberA from numberB. We'll make a quick change and get the test passing:

```
function subtract(numberA, numberB) {
    return numberB - numberA;
}
```

Good job partner! We're back to green. What was the next step again?

Refactor

So, I can see some duplication in our test suite but at the moment I think by refactoring it we'd lose some of its expressiveness. Lets leave the test suite and take a look at the implementation. In the previous step we switched the order of the integers to be subtracted and I'm not sure how obvious this is. I think we can apply the rename refactoring here to make our intent more obvious and our code more expressive.

Old code:

```
function subtract(numberA, numberB) {
    return numberB - numberA;
}
```

Refactored code:

```
function subtract(amount, fromNumber) {
    return fromNumber - amount;
}
```

That's better! Our tests are all green and the *subtract* function makes more sense!

REPEAT REPEAT REPEAT!!!

If you want to play along at home keep going and implement the remaining two basic calculator functions; multiply and divide! Remember to keep passing the keyboard if you're pairing and always refactor to make the code more readable and to remove duplication **(from both the implementation and the tests!!)**. Enjoy!