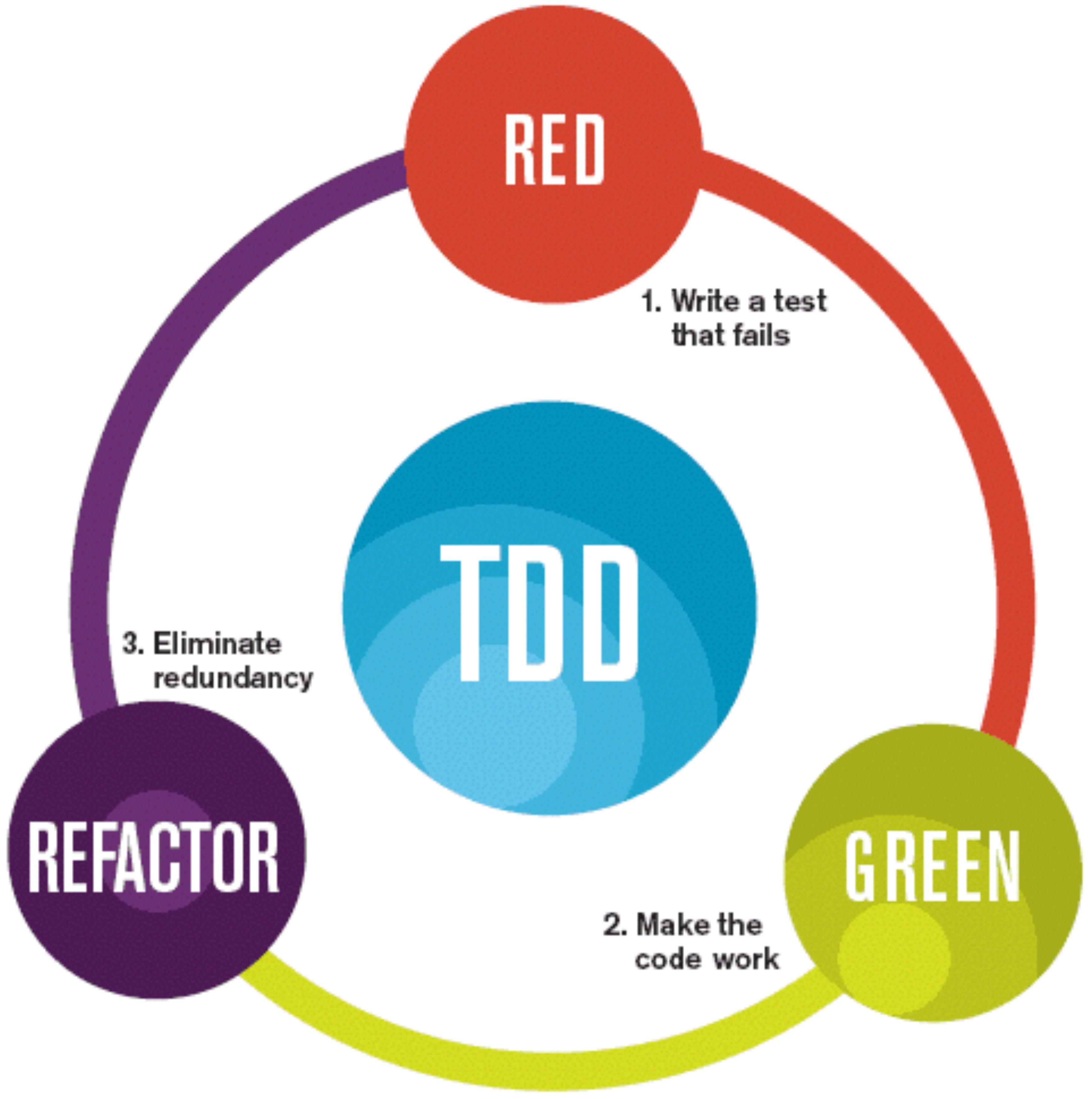


The mantra of Test-Driven Development (TDD) is "red, green, refactor."

RUTGERS | CODING BOOTCAMP

Session 16.1 - Test-driven development (TDD)

Test-driven development (TDD)

Objectives

- Students will understand why we test our codebase
- Students will understand what Test-driven development (TDD) is
- Students will understand unit testing in TDD

Why Test Our Code?

Tests Improve Design

- Writing tests forces you to make your code testable. Specifically, you tend to rely less on bad habits like using global variables, instead making your code more component based and easier to use.

Tests Defend Against Other Programmers

- Larger projects and applications are touched by many programmers. With code tests, you can ensure that when another programmer changes the code, the test breaks and informs the programmer of the problem. This behavior avoids breaking unknown sections of the code that are relying on that code

Why Test Our Code?

Testing Forces You to Slow Down and Think

- TDD design requires programmers to think about function inputs and outputs

Tests Reduce Bugs in Existing Features

- With proper unit testing we can ensure that functions won't be broken during refactoring of code

Testing Makes Development Faster

- TDD helps you to realize when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing

Tests Reduce Fear

- One of the biggest fears that programmers encounter is making a change to a piece of code and not knowing what is going to break. Having a complete test suite allows programmers to remove the fear of making changes or adding new features.

Test Driven Development (TDD)

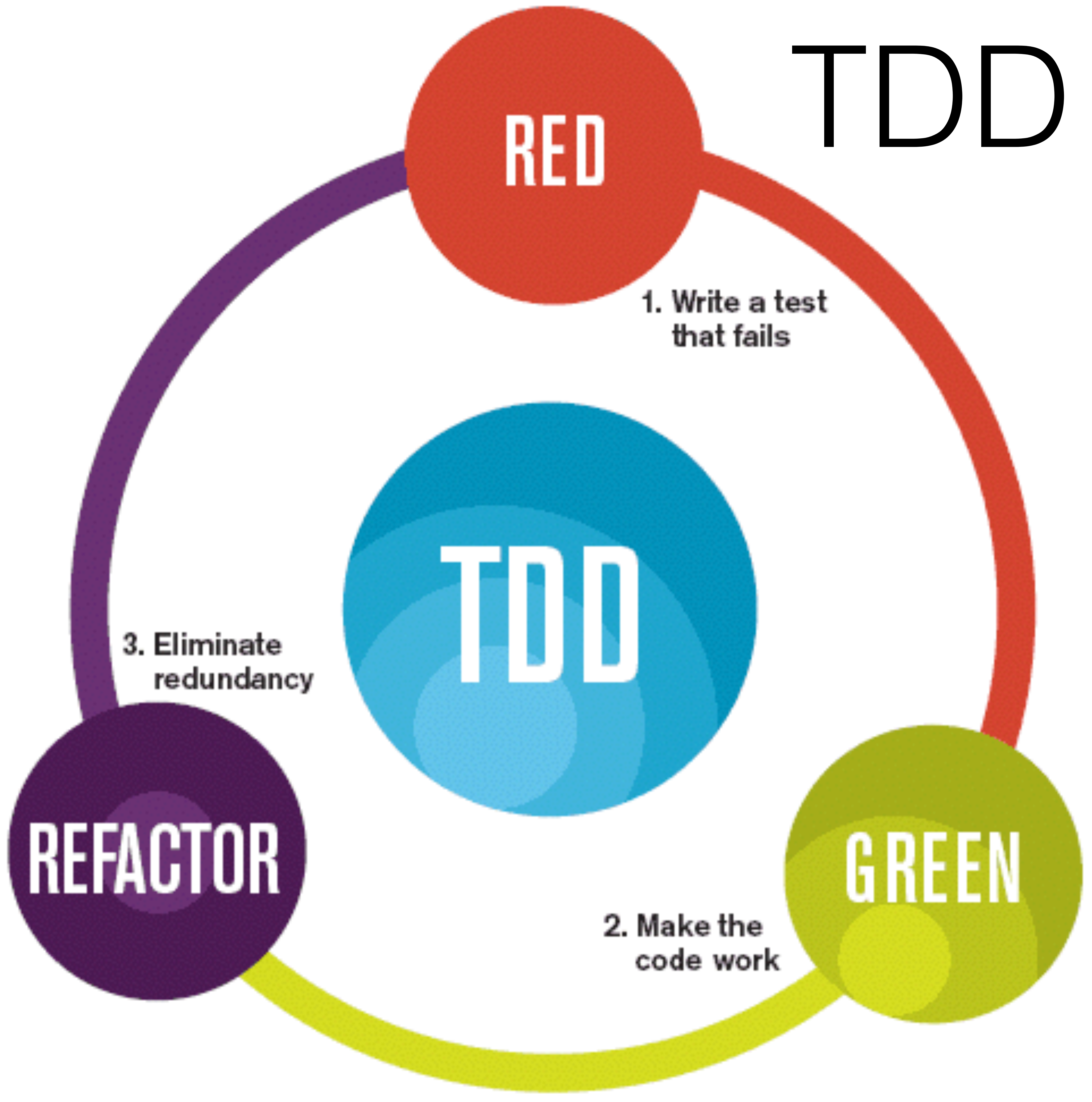
TDD tests small sections of code in an automated fashion in order to make sure that when small critical sections of the code work we can rely on larger sections that rely on the smaller sections to work also.

Unit Tests - testing individual units of code in isolation.

TDD Method

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

TDD Method



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

1. Write function with no code inside but knowing the desired inputs and output to accomplish the task.
   This will fail the test.
2. Write the most basic code inside your function to make it work and pass the test.
   This will pass the test.
3. When time or resources permit, revisit the function and refactor, or improve the code.
   Refactor.

Testing Tools For Javascript

- Jasmine
- Karma
- Selenium
- Selenium Webdriver

Writing Tests

- Tests are defined in spec files in your application that test functions in your code.

- Each time the code is being deployed, all tests defined in your specs should be run on your code to ensure that with the addition of your new code, previous tests still pass.

Jasmine

- A JavaScript testing library
- With an easy to use syntax
- That can be used in the browser (with Selenium)
- Or on the command line (TDD)

Jasmine uses the term "specs" to emphasize that they should be written before you start coding.

Example Test(Spec) of add()

```
describe("Addition function", function() {
  it("should add numbers", function() {
    expect(add(1, 1)).toBe(2);
  });
});
```

We are testing that our add() function gives us the desired result of what we expect from an addition function.

Example Test(Spec) of add()

```
describe("Calculator", function() {
  describe("Addition function", function() {
    it("should add numbers", function() {
      expect(add(1, 1)).toBe(2);
      expect(add(2, 2)).toBeGreaterThan(3);
    });
  });
});
```

Multiple tests can be performed in a describe block to ensure that the function behaves correctly in multiple scenarios

Spec Requirements

```
describe("Calculator", function() {
  describe("Addition function", function() {
    it("should add numbers", function() {
      expect(add(1, 1)).toBe(2);
      expect(add(2, 2)).toBeGreaterThan(3);
    });
  });
});
```

- At least one **`describe`** block (they can be nested)
- At least one **`it`** block which contains a spec
- At least one expectation, which consists of:
  - **`expect`** which is passed an expression (called the "actual" value)
  - A matcher with the expected value (**`toBe`** and **`toBeGreaterThan`**)

Using Jasmine

- **The Plan**
- We're going to write a mini string library which has the following functions.
- A "first word" function which returns the first word of a string.
- An "nth word" function which returns any word n of a string.

Using Jasmine

First we describe it
We're testing the **`stringUtil`** library.
Specifically we're testing the **`firstWord`** function.

```
describe("stringUtil", function() {
  describe("firstWord", function() {
    // Specs go here.
  });
});
```

Using Jasmine

Then we spec it

```
describe("stringUtil", function() {
  describe("firstWord", function() {
    it("should return the first word of a string", function () {
      expect(stringUtil.firstWord("one two")).toBe("one");
    });
  });
});
```

Using Jasmine

Then we run the specs

FAIL

Using Jasmine

Remember the TDD process.
Red -> Green -> Refactor
If you don't get red after writing your specs first, then there is something wrong with your specs.
Basically, that first red means you're doing the right thing.

```
stringUtil isn't defined.
```

Using Jasmine

So let's define it (in our code)

```
var stringUtil = {};
```

Using Jasmine

Test again and **FAIL** : Now we get a different error (from our spec)
`stringUtil` doesn't have a `firstWord` method

Let's fix that (in our code)

```
var stringUtil = {
  firstWord: function() {
    // Not doing anything yet.
  }
};
```

Using Jasmine

Test again and **FAIL** : Our function isn't returning anything

Now for some actual logic

```
var stringUtil = {
  firstWord: function(text) {
    var textWords = text.split(" ");
    return textWords[0];
  }
};
```

Using Jasmine

We got to green!

(But that code wasn't the greatest)

Time to refactor

```
var stringUtil = {
  firstWord: function(text) {
    return text.split(" ")[0];
  }
};
```

Using Jasmine

Green again!

Now lets write the specs for **`nthWord`**

```
describe("stringUtil", function() {
  describe("firstWord", function() {
    it("should return the first word of a string", function () {
      expect(stringUtil.firstWord("one two")).toBe("one");
    });
  });

  describe("nthWord", function() {
    it("should return the nth word of a string", function () {
      expect(stringUtil.nthWord("one two", 1)).toBe("one");
      expect(stringUtil.nthWord("one two", 2)).toBe("two");
    });
  });
});
```

Using Jasmine

We get our first **failing** spec because `stringUtil` doesn't have a `nthWord` method

Let's write `nthWord`

```
var stringUtil = {
  firstWord: function(text) {
    return text.split(" ")[0];
  },

  nthWord: function(text, i) {
    return text.split(" ")[i - 1];
  }
};
```

Using Jasmine

All Green! TDD Unit Test Complete

```
describe("stringUtil", function() {
  describe("firstWord", function() {
    it("should return the first word of a string", function () {
      expect(stringUtil.firstWord("one two")).toBe("one");
    });
  });

  describe("nthWord", function() {
    it("should return the nth word of a string", function () {
      expect(stringUtil.nthWord("one two", 1)).toBe("one");
      expect(stringUtil.nthWord("one two", 2)).toBe("two");
    });
  });
});
```