TESTING / TDD

GOALS FOR THIS SECTION

1. Testing
    - Test After
    - Test First
    - Test-Driven Development
2. TDD by Example
3. Real-World Testing

TESTING

SOFTWARE TESTING

Software Testing is the process of ensuring that the code works the way the developers intended it to without bugs.

It is critical because:

- It is better for developers to find a bug rather than their customers.
- Buggy software decreases brand reputation.
- Fixing old bugs distracts from advancing new features.

MANUAL TESTING

One way to test is for humans to test it manually.

- Pros: humans are observant and flexible
- Cons: humans are expensive, slow, imperfect, and bore easily

AUTOMATED TESTING

Developers can use programs to test for us.

This is called automated testing.

- Pros: repeated testing is quick and low cost (always knowing the code still works)
- Cons: take time & effort to write, rigid, don't notice side problems

TEST AFTER

Often programmers write the main program first (the "production" code).

Afterward, they write test programs to find any bugs they missed.

- Pros: Developers get to write the exciting code first
- Cons: It can be hard to write tests for production code that wasn't designed to be tested.

TEST FIRST

Another approach is to create the test *before* developing the software solution.

- Pros: Enforce writing tests. Helps think through solutions. Makes sure solutions are written in a testable way.
- Cons: It requires some discipline.

TEST FIRST

This may not seem intuitive at first because, *of course*, any test written prior to the solution is doomed to fail.

That turns out to be the point.

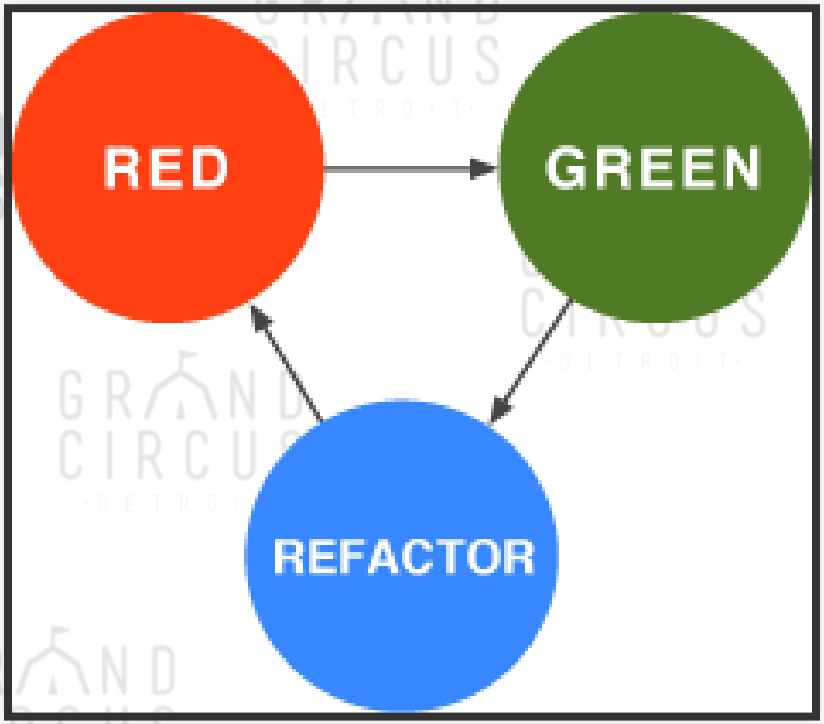
RED, GREEN, REFACTOR

This cycle of testing is most often referred to as Test-Driven Development (TDD).

The core of this method is summarized as "Red, Green, Refactor".

It goes like this:

1. Write a failing test - Red
2. Make that test pass - Green
3. Clean up the code - Refactor


RED
GREEN
REFACTOR

RED, GREEN, REFACTOR

What's with the colors?

The short answer is it has become a convention that test frameworks will mark failing tests as red and passing tests as green. S

o Red = failing test, Green = passing test.

Refactor has been made blue because who knows...

WHY REFACTOR?

When writing tests using the TDD method, it is done in minimal steps.

Only enough code is written to make that test pass.

Then another test is written for the next smallest step to take.

The solution must pass all tests in order to be successful.

The solution would quickly devolve into a real mess if we didn't refactor.

After each time a test passes, stop to refactor.

Look for ways to improve the solution *without changing its behavior*.

EFFECTIVE REFACTORING

We already know it works (green) so after any refactoring the tests must be passed again.

If tests are passing still, the refactoring has not changed the code's behavior.

TEST FRAMEWORKS

TEST FRAMEWORKS

To run tests, a testing framework is needed.

Testing frameworks provide syntax for tests and check the results.

Most popular languages have several popular testing frameworks.

JASMINE

Jasmine is a very nice testing framework that is easy to use and complete out of the box.

It is not as configurable as some of the other frameworks but its completeness makes up for that.

JASMINE

Frameworks like Jasmine can help us define tests.

Often they are used in conjunction with *test runners* which actually run the tests.

Karma is the test runner is choice.

KARMA

Karma will take care of several things...

- Finding JavaScript files and tests.
- Run them in a browser.
- Print out the results on the terminal.
- Run the tests automatically every time a file is saved.

A SAMPLE JASMINE TEST

```
// describe contains a suite of tests for a certain feature or unit
describe("answer", function() {
  // each test case is an `it`. You can have multiple of these.
  // The string here is only for humans to read; it does not affect the test.
  it("to life the universe and everything", function() {
    // Call the function we want to test and check the result.
    expect(answer()).toEqual(42);
  });
});
```

This test says that the `answer()` function is *supposed* to return 42.

SETTING UP JASMINE

SETTING UP JASMINE

Without us setting up your testing environment, here are some simple steps to get started.

First: Initialize the directory:

```
npm init --yes
```

Second: Install Karma and the proper plugins.

```
npm i -g karma-cli
npm i karma jasmine jasmine-core karma-chrome-launcher karma-jasmine karma-spec-reporte
```

SETTING UP JASMINE

Third: run Karma Init (accept all defaults by pressing Enter).

```
karma init
```

Fourth: Assign the files that will be used in karma.conf.js.

```
files: [ "src/*.js", "test/*.js" ]
```

SETTING UP JASMINE

Fifth: Assign the plugins that will be used in karma.conf.js.

Add this as a property to karma.conf.js (the order doesn't matter).

```
plugins: [
  require("karma-chrome-launcher"),
  require("karma-jasmine"),
  require("karma-spec-reporter")
]
```

SETTING UP JASMINE

You are now able to start writing code in the "src" folder and tests in the "tests" folder.

To test your code, run the command `karma start`.

```
karma start
```

JEST

JEST

Jest is gaining a lot of steam.

Just was built on top of Jasmine, so any experience with Jasmine should translate to Jest.

Jest's main mission was to deliver a non-configurable testing kit.

A SAMPLE JEST TEST

```
// describe contains a suite of tests for a certain feature or unit
describe("answer", function() {
  // each test case is an `it`. You can have multiple of these.
  // The string here is only for humans to read; it does not affect the test.
  test("to life the universe and everything", function() {
    // Call the function we want to test and check the result.
    expect(answer()).toEqual(42);
  });
});
```

SETTING UP JEST

SETTING UP JEST

First: Initialize the directory:

```
npm init --yes
```

Install Jest locally.

```
npm install jest --save-dev
```

JEST

Adjust `package.json` to look similar to this:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Once Jest has been installed, two files should be created to start.

```
touch script.js script.test.js
```

JEST

To run tests, simply just run:

```
npm run test
```

If you want to have the tests run each time you make an adjustment, add a watch flag to the test command.

```
npm run test --watchAll
```

If you want to check the coverage (how much of your code is actually tested), add a coverage flag.

```
npm run test --coverage
```

EXERCISE TIME

TEMP CONVERTER

We'll build and test a function that takes a temperature and a unit (either "C" or "F") and converts it.

TEST CASES

To test, we have to brainstorm some *test cases*.

Each test case is an example to run with our code and what we expect our code to do in that case.

TEMP CONVERTER

A function that takes temperature and a unit (either "C" or "F") and converts it.

TEST CASES:

Input temp	Input targetUnit	Output temp
32 (F)	"C"	0 (C)
68 (F)	"C"	20 (C)
100 (C)	"F"	212 (F)
-40 (C)	"F"	-40 (F)

YOU TRY IT

Work in pairs.

WHAT SHOULD I WEAR?

Write a script that will tell you what to wear based on the temperature and type of event.

Event type	suggestion
casual	"something comfy"
semi-formal	"a polo"
formal	"a suit"

Temperature	suggestion
Less than 54 degrees	"a coat"
54 - 70 degrees	"a jacket"
more than 70 degrees	"no jacket"

WHAT SHOULD I WEAR

Display the results for a given set of decisions.

Some sample output:

```
"Since it is 33 degrees and you are going to a formal event, you should wear a suit and a coat."
"Since it is 55 degrees and you are going to a semi-formal event, you should wear a polo and a jacket."
"Since it is 85 degrees and you are going to a casual event, you should wear something comfy and no jacket."
```

TESTING PART 2

1. Project Setup
2. Real-world Testing
3. Testing Techniques
    - Various Assertions
    - Arrange, Act, Assert
    - Spies & Mocks

PROJECT SETUP

1. Open the `battle-game` demo in an editor and the terminal.
2. In the terminal, type `npm install` and hit Enter
3. Run `npm run test` to kick off the tests. It will re-run them every time you save a file.

THE BATTLE GAME

We will be adding tests to this project. However, first, let's have a look at it.

1. Open index.html and play with it.
2. Have a look at `character.js` and `random.js`.

BATTLE GAME DEMO

Let's add some tests.

ARRANGE, ACT, ASSERT

This is a good pattern to follow for each of the non-trivial test cases.

1. *Arrange* - Many times developers will need to get a few things in order before tests can run.
2. *Act* - Run the code.
3. *Assert* - Verify that the code did what you expected.

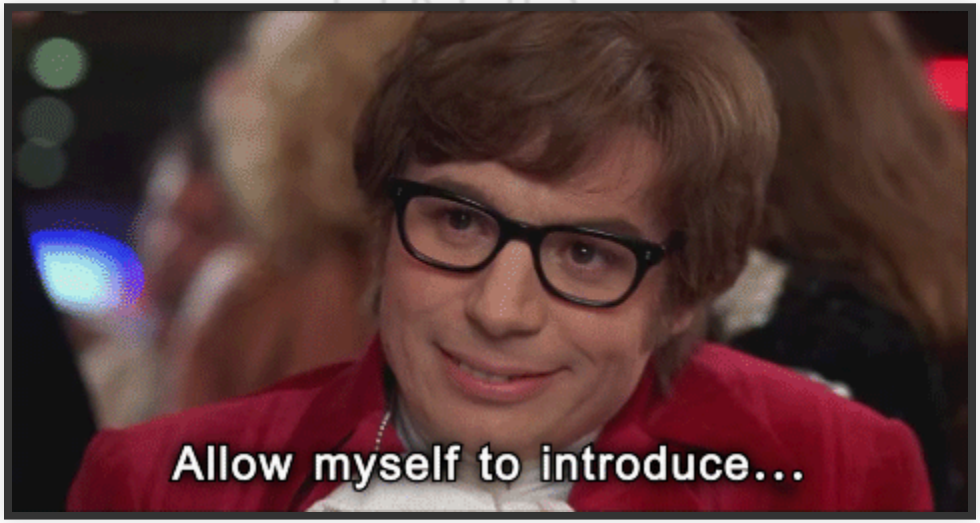
ARRANGE, ACT, ASSERT

```
it("receiveAttackDamage removes health", function() {
  // Arrange
  var player = new Character({
      health: 20
  });

  // Act
  player.receiveAttackDamage(5);

  // Assert
  expect(player.health).toBe(15);
});
```

TESTING WITH SPIES & MOCKS



SPIES & MOCKS

Spies are fake functions with special abilities to track how they are called.

Mocks are fake versions of real modules, services, and other dependencies.

SPIES

Karma can test whether a spy function has been called and what arguments were passed to it.

```
var fakeFunction = jasmine.createSpy("fakeFunction");

fakeFunction("Hello");

expect(fakeFunction).toHaveBeenCalled();
expect(fakeFunction).toHaveBeenCalledWith("Hello");
expect(fakeFunction).not.toHaveBeenCalledWith("Goodbye");
```

SPIES

It is also possible to tell a spy what it should do when it is called.

```
var fakeFunction = jasmine.createSpy("fakeFunction").and.returnValue("Boo!");

var result = fakeFunction("Hello");

expect(fakeFunction).toHaveBeenCalled();
expect(fakeFunction).toHaveBeenCalledWith("Hello");
expect(result).toBe("Boo!");
```

SPIES

Often times, a spy is needed to spy on a method of an object.

```
// Start spying on console.log!!
const logSpy = spyOn(console, "log");

console.log("Hello World!");

expect(logSpy).toHaveBeenCalled();
expect(logSpy).toHaveBeenCalledWith("Hello World!");
```

SPIES

There's a lot that can be done with spies.

See docs.

EXPECTATIONS

So many options...

```
expect(answer).toBe(42); // ===
expect(answer).toEqual(42); // ==
expect(answer).toBeGreaterThan(100); // >
expect(answer).toBeLessThanOrEqual(100); // <=
expect(answer).toBeTruthy();
expect(answer).toBeNull();
expect(answer).toMatch(/cat|dog/); // regular expression
```

Jasmine matchers Jest matchers

TDD LAB