14.1. Why Test Your Code?

Checking your code is part of the development process. Developers rarely write code without verifying it. You are used to debugging programs as you write them. In fact, we devoted an entire chapter to debugging early in the course.

Your development process probably looks something like this:

1. Write code
2. Run program
3. Notice error and investigate
4. Repeat these steps until there are no more errors

But there's a better way to test your code, using *automated* tests. Automated tests actively test your code and help to remove the burden of manual testing. There are many types of automated tests. This chapter focuses on **unit testing**, which tests the smallest components (or *units*) of code. These are typically individual functions.

Before we dive into the *how* of unit testing, let's discuss the *why*.

14.1.1. Know Your Code *Really* Works

Manual testing can eventually lead you to a complete, error-free program. Unit testing provides a better alternative.

This might sound familiar:

You write a program and manually test it. Thinking it is complete, you turn it in only to find that it has a bug or use case that you didn't consider.

The unit testing process helps avoid this by starting with a list of specific, clearly stated behaviors that the program should satisfy. The behaviors are then converted into automated tests that demonstrate program behavior and provide a framework for writing code that *really* works.

14.1.2. Find Regressions

What about this situation?

You write feature #1 for a program. You then move on to feature #2. After finishing feature #2, you realize that your changes broke feature #1.

This is frustrating, right? Especially with larger programs, adding new features often causes unexpected problems in other parts of the code, potentially breaking the entire program. The introduction of such a bug is known as a **regression**.

If you have a collection of tests that can run quickly and consistently, you will know *right away* when a regression appears in your program. This allows you to identify and fix it more quickly.

14.1.3. Tests as Documentation

One of the most powerful aspects of unit testing is that it allows us to clearly define program expectations. A good collection of unit tests can function as a set of *statements* about *how* the program should behave. You and others can read the tests and quickly get an idea of the specifics of program behavior.

Example

Your coworker gives you a function that validates phone numbers, but doesn't provide much detail. Does it handle country codes? Does it require an area code? Does it allow parentheses around area codes? These details would be easily understood if the function had a collection of unit tests that described its behavior.

Code with a good, descriptive set of unit tests is sometimes called **self-documenting code**.

Remembering what your code does and why you structured it a certain way is easy for small programs. However, as the number of your projects increase and their size grows, the need for documentation becomes critical.

Documentation can be in the form of code comments or external text documents. These can be helpful, but have one major drawback which is that they can get out of date very quickly. Out dated, incorrect documentation is very frustrating for a user.

Properly designed unit tests are runnable documentation for your project. Because unit tests are runnable code that declares and verifies features, they can NEVER get out of sync with the updated code. If feature is added or removed, the tests must be updated in order to make them pass.

Let's go ahead and write our first unit test!

14.2. Hello, Jasmine!

In order to unit test our code, we need to use a module. Such a module is called a **unit-testing framework**, **test runner**, or **test harness**, and there are many to choose from.

We will use Jasmine, a popular JavaScript testing framework.

14.2.1. Using Jasmine

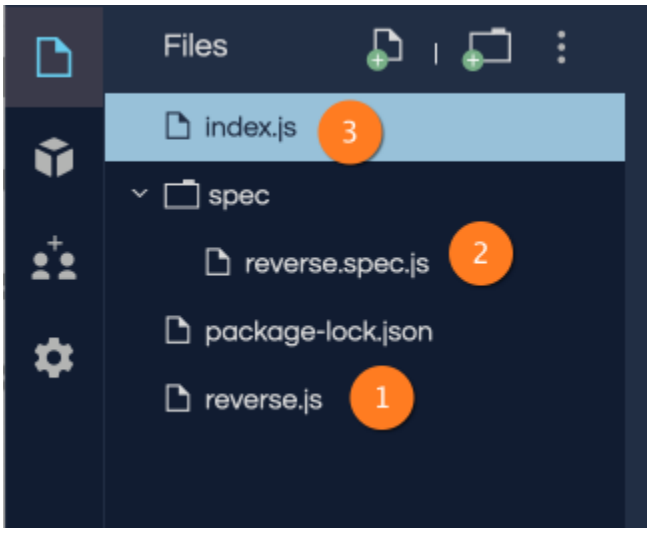
Jasmine is an npm module that can be installed and used in a manner similar to `readline-sync`. Usually, Jasmine must be manually installed into a project, but we do not need to learn how to do this yet.

In this chapter we will continue to use repl.it, which automatically installs npm modules when it runs a program that contains a `require` statement.

Try It!

Run some tests for the reverse function. This is the same `reverse` function that we wrote previously.

Don't worry about understanding the code at this point, just hit *run* to execute the tests. How many total tests are there? How many passed? How many failed?
A project using Jasmine has several components. Here's the project structure:



A Jasmine project

There are three important files:

1. `reverse.js` contains the `reverse` function, which must be exported for use in other files.
2. `spec/reverse.spec.js` contains the tests for `reverse`.
3. `index.js` contains the Jasmine code needed to run the tests. This is the file that executes when you hit *run* in repl.it.

Warning

Jasmine can be set up and used in many different ways. If you are looking for an answer on the Internet,—on Stack Overflow or in the Jasmine documentation—you will see widely varying usages of Jasmine that don't apply to your situation. Rely on this book as your main reference, and you'll be fine.

14.2.2. Hello, Jasmine!

Let's build a "Hello, World!" Jasmine project, to get familiar with the basic components. Open and fork this repl.it project

We will walk you through the steps needed to get a simple Jasmine project up and running. Code along with us throughout this section.

14.2.2.1. `index.js`

This is the main project file. Up until now, **`index.js`** is where you have been writing the code for a given exercise or assignment. Now that we are writing tests for our code, **`index.js`** will contain the Jasmine code to find and execute the tests. Our project-specific code will live in other files.

```
 const Jasmine = require('jasmine');
1const jasmine = new Jasmine();
2
3jasmine.loadConfig({
4   spec_dir: 'spec',
5   spec_files: [
6      "**/*[sS]pec.js"
7   ],
8});
9
10jasmine.execute();
11
```

There are three main components of this program:

1. Lines 1-2 import the Jasmine module and create a new Jasmine object, **`jasmine`**. This object is responsible for finding and executing our tests.
2. Lines 4-9 configure Jasmine to look for tests in the **`spec/`** directory of our project. Any file in this directory of the form **`fileName.spec.js`** will be assumed to contain tests, and will be executed by Jasmine.
3. Line 11 triggers Jasmine to find and execute the tests.

Try It!

Hit *run* on the project. Two things happen:

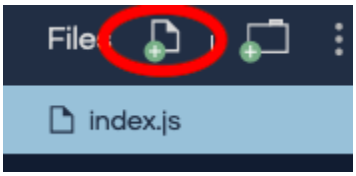
- repl.it installs Jasmine.
- Jasmine searches for tests, finding none.

Let's add some code to test.

14.2.2.2. `hello.js`

If you have not already done so, click *Fork* on the repl.it menu bar so you can edit the starter code.

Create a new file in your project by clicking the icon in the menu bar.



Name the new file `hello.js`, then add this code:

```
function hello(name) {
  if (name === undefined)
     name = "World";

  return "Hello, " + name + "!";
}

```

The `hello` function takes a single argument representing a person's name and returns a string greeting that person. If the function is called without an argument, the function returns `"Hello, World!"`.

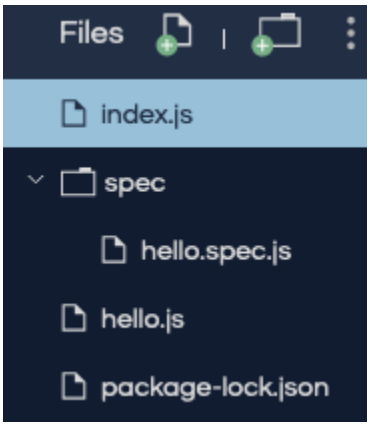
To use this function outside `hello.js` we must export it. Add this statement at the bottom of the file.

```
module.exports = hello;
```

14.2.2.3. spec/hello.spec.js

Now that we have a function to test, let's write some test code. Add a folder named `spec` to the project. Within the folder, create the file `hello.spec.js`. It is conventional to put tests for `fileName.js` in `spec/fileName.spec.js`. This makes it easy to find the tests associated with a given file.

Your file tree should look something like this:



At the top of the `hello.spec.js` file, import your function from `hello.js`, along with the `assert` module:

```
const hello = require('../hello.js');
const assert = require('assert');

```

Below that, call the function `describe`, passing in the name of the function we want to test along with an empty anonymous function. `describe` is a Jasmine function that is used to group related tests. Related tests are placed *within* the anonymous function that it receives.

```
describe("hello", function(){

});
```

14.2.2.4. Specifications and Assertions

There are two cases we want to test:

1. The function is called with a string argument. In this case, a customized greeting should be returned.
2. The function is called with no argument. In this case, the general greeting should be returned.

Within `describe`'s function argument, place a test for case 1:

```
it("should return custom message when name is specified", function(){
   assert.strictEqual(hello("Jasmine"), "Hello, Jasmine!");
});
```

The `it` function is part of the Jasmine framework as well. Calling `it` creates a **specification**, or **spec**, which is a description of expected behavior. The first argument to `it` is a string describing the expected behavior. This string serves to document the test and is also used in reporting test results. Your expectation strings will usually begin with "should", followed by an expected action.

The second argument to `it` is yet another anonymous function. This function contains the test code itself, which takes the form of an **assertion**. An assertion is a declaration of expected behavior *in code*. Let's examine the contents of the anonymous function:

```
assert.strictEqual(hello("Jasmine"), "Hello, Jasmine!");
```

Calling `assert.strictEqual` with two arguments declares that we expect the two arguments to be (strictly) equal. As you get started with unit testing, nearly *all* of your tests will take this form. The first argument to `assert.strictEqual` is a call to the function `hello`. The second argument is the expected output from that function call.

If the two arguments are indeed equal, the test will pass. Otherwise, the test will fail. In this case, we are declaring that `hello("Jasmine")` should return the value `"Hello, Jasmine!"`.

Note

Jasmine also has a `.equal` comparison, which tests for *loose* equality. The difference between loose and strict equality with Jasmine is the same as that of JavaScript in general. For this reason, we prefer `.strictEqual` over `.equal`.
Your test file should now look like this:

```
const hello = require('../hello.js');
const assert = require('assert');

describe("hello world test", function(){

   it("should return a custom message when name is specified", function(){
      assert.strictEqual(hello("Jasmine"), "Hello, Jasmine!");
   });

});

```

14.2.2.5. Test Reporting

This is a fully-functioning test file. Hit *run* to see for yourself. If all goes well, the output will look like this:

```
Randomized with seed 00798
Started
.


1 spec, 0 failures
Finished in 0.016 seconds
Randomized with seed 00798 (jasmine --random=true --seed=00798)

```

The most important line in the output is this one:

```
1 spec, 0 failures
```

It tells us that Jasmine found 1 test specification, and that 0 of the specs failed. In other words, *our test passed!* The third line also contains useful information. It will contain one dot (`.`) for each successful test, and an `F` for each failed test. As our test suite grows, this becomes a nice visual indicator of the status of our tests.

Let's see what a test failure looks like. Go back to `hello.js` and remove the `"!"` from the return statement:

```
return "Hello, " + name;
```

Run the tests again. This time, the output looks quite different:

```
Randomized with seed 98738
Started
F


Failures:
1) hello world test should return a custom message when name is specified
Message:
   AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
   + expected - actual

   - 'Hello, Jasmine'
   + 'Hello, Jasmine!'
Stack:
   error properties: Object({ generatedMessage: true, code: 'ERR_ASSERTION', actual: 'He
llo, Jasmine', expected: 'Hello, Jasmine!', operator: 'strictEqual' })
      at <Jasmine>
      at UserContext.<anonymous> (/home/runner/spec/reverse.spec.js:23:14)
      at <Jasmine>
      at runCallback (timers.js:705:18)
      at tryOnImmediate (timers.js:676:5)
      at processImmediate (timers.js:658:5)

1 specs, 1 failure
Finished in 0.021 seconds

```

We intentionally made a test fail. The failing test appears in the `Failures:` section on line 5. This describes exactly what went wrong. The test expected the value `'Hello, Jasmine!'` but received `'Hello, Jasmine'`. Notice that the failure description is the result of joining the two string arguments from `describe` and `it`. This is why we intentionally defined those strings the way we did.

The `Stack:` section on line 13 can be mostly ignored for now. Line 22 has a key statistic showing how many tests, called specs, were run and how many failed `1 specs, 1 failure`.

Put `hello.js` back as it was and run the tests again to make sure it works.

Let's add a final spec to test our other case.

```
it("should return a general greeting when name is not specified", function(){
     assert.strictEqual(hello(), "Hello, World!");
 });
```

This spec declares that calling `hello()` should return `"Hello, World!"`. Run the tests again and you'll see this output:

```
Randomized with seed 81081
Started
..


2 specs, 0 failures
Finished in 0.025 seconds
Randomized with seed 81081 (jasmine --random=true --seed=81081)
```

Nice work! You just created your first program with a full test suite. You can view our full Hello, Jasmine! project for reference.

There are a lot of details in the setup of these tests, so take a few minutes to look over the code and describe to yourself what each component is doing.

Note

There are many ways to structure test specifications. If you look at the official Jasmine documentation, you'll see specs with different code in place of `assert.strictEqual`:

We have chosen to use `assert.strictEqual` because its syntax is more similar to common testing frameworks in other languages like Java and C#. Learning to use `assert.strictEqual` will make it easier for you to transition to one of those frameworks later in the class.

14.2.3. Check Your Understanding

Question

Examine the function below, which checks if two strings match:

```
function doStringsMatch(string1, string2){
  if (string1 === string2) {
     return 'Strings match!';
  } else {
     return 'No match!';
  }
}
```

Which of the following tests checks if the function properly handles case-*sensitive* answers.

a. `assert.strictEqual(doStringsMatch('Flower', 'Flower'), 'Strings match!');`
b. `assert.strictEqual(doStringsMatch('Flower', 'flower'), 'No match!');`
c. `assert.strictEqual(doStringsMatch('Flower', 'plant'), 'No match!');`
d. `assert.strictEqual(doStringsMatch('Flower', ''), 'No match!');`

14.3. Unit Testing in Action

Testing is a bit of an art; there are no hard and fast rules about how to go about writing good tests. That said, there are some general principles that you should follow. In this section, we explore some of these.

In particular, we focus on identifying good **test cases** by working through a specific example. A test case is a single situation that is being tested.

14.3.1. What to Test

When writing tests for your code, what should you test? You can't test *every* possible situation or input. But you also don't want to leave out important cases. A function or program that isn't well-tested might have bugs lurking beneath the surface.

Note

Since we are focused on *unit* testing, in this chapter we will generally use the term "unit" to refer to the function or program under consideration.
Regardless of the situation, there are three types of test cases that you should consider: positive, negative, and edge cases.

1. A **positive test** verifies expected behavior with valid data.
2. A **negative test** verifies expected behavior with *invalid* data.
3. An **edge case** is a subset of positive tests, which checks the extreme edges of valid values.

Example

Imagine a function named `setTemperature` that accepts a number between `50` and `100`.

1. Positive test values: `56`, `75`, `80`
2. Negative test values: `-1`, `101`, `"70"`
3. Edge case values: `50`, `100`

Considering positive, negative, and edge tests will go a long way toward helping you create well-tested code.

Let's see these in action, by writing tests for our isPalindrome function.

14.3.2. Setting Up

Here's the function we want to test:

```
function reverse(str) {
  return str.split('').reverse().join('');
}

function isPalindrome(str) {
  return reverse(str) === str;
}

```

repl.it

Code along with us by forking our repl.it starter code project, which includes the above code in `palindrome.js` and the Jasmine test runner code in `index.js`. Note that we have removed the `console.log` statements from the original code and exported the `isPalindrome` function:

```
module.exports = isPalindrome;
```

Tip

When creating a unit-tested project, *always* start by copying the Jasmine test runner code into `index.js` and putting the code you want to test in an appropriately named `.js` file.
You have become used to testing your code by running it and printing output with `console.log`. When writing unit-tested code, we no longer need to take this approach.

Tip

If you find yourself tempted to add a `console.log` statement to your code, write a unit test instead! You would mostly likely remove that `console.log` after getting your code to work, while the test will remain for you and other developers to use in the future.
Finally, create `spec/` folder and add a spec file, `palindrome.spec.js`. This file should include imports and a describe block:

```
const isPalindrome = require('../palindrome.js');
const assert = require('assert');

describe("isPalindrome", function(){

  // TODO - write some tests!

});

```

Okay, let's write some tests!

14.3.3. Positive and Negative Test Cases

14.3.3.1. Positive Test Cases

We'll start with positive and negative tests. For `isPalindrome`, some positive tests have inputs:

- `"a"`
- `"aaaa"`
- `"aba"`
- `"racecar"`

Calling `isPalindrome` with these inputs should return `true` in each case. Notice that these tests are a simple as possible. Keeping test inputs simple, while still covering your desired test cases, will make it easier to fix a bug in the event that a unit test fails.

Let's add tests for these inputs to `spec/palindrome.spec.js`:

```
const isPalindrome = require('../palindrome.js');
const assert = require('assert');

describe("isPalindrome", function(){

  it("should return true for a single letter", function(){
     assert.strictEqual(isPalindrome("a"), true);
  });

  it("should return true for a single letter repeated", function(){
     assert.strictEqual(isPalindrome("aaa"), true);
  });

  it("should return true for a simple palindrome", function(){
     assert.strictEqual(isPalindrome("aba"), true);
  });

  it("should return true for a longer palindrome", function(){
     assert.strictEqual(isPalindrome("racecar"), true);
  });

});

```

Note the clear test case descriptions (for example, "should return true for a single letter repeated"), which will help us easily identify the expected behavior of our code later.

After adding the positive tests to your file, run them to make sure they all pass.

14.3.3.2. Negative Test Cases

For `isPalindrome`, some negative tests have inputs:

- `"ab"`
- `"launchcode"`
- `"abA"`
- `"so many dynamos"`

Calling `isPalindrome` with these inputs should return `false` in each case. The last two of these negative tests deserve a bit more discussion.

When writing our `isPalindrome` function initially, we made two important decisions:

- Case should be considered, and
- whitespace should be considered.

The definition of a palindrome differs sometimes on these two matters, so it's important to test them.

Testing with input `"abA"` ensures that case is considered, since the lowercase version of this string, `"aba"`, *is* a palindrome. Testing with `"so many dynamos"` ensures that whitespace is considered, since the version of this string with whitespace removed, `"somanydynamos"`, *is* a palindrome.

Note

It's important to isolate your test cases. For example, `"So Many Dynamos"` is a poor choice of input for a negative test, since it contains *two* characteristics that are being tested for - case and whitespace. If a test with this input failed, it would NOT be clear why it failed.
Including specific tests that demonstrate how *our* `isPalindrome` function behaves in these situations helps make our code *self-documenting*. Someone can read our tests and easily see that we *do* consider case and whitespace.

Let's add some test for these negative cases. Add these within the `describe` call.

```
it("should return false for a longer non-palindrome", function(){
   assert.strictEqual(isPalindrome("launchcode"), false);
});

it("should return false for a simple non-palindrome", function(){
   assert.strictEqual(isPalindrome("ab"), false);
});

it("should be case-sensitive", function(){
   assert.strictEqual(isPalindrome("abA"), false);
});

it("should consider whitespace", function(){
   assert.strictEqual(isPalindrome("so many dynamos"), false);
});
```

Now run the tests to make sure they pass. Your code now includes a set of tests that considers a wide variety of positive and negative cases.

14.3.4. Edge Cases

Recall our definition of **edge case**:

An edge case is a test case that provides input at the extreme edge of what the unit should be able to handle.

Edge cases can look very different for different units of code. Most of the examples we provided above dealt with numerical edge cases. However, edge cases can also be non-numeric.

In the case of `isPalindrome`, the most obvious edge case would be that of the empty string, `""`. This is the smallest possible string that we can use when calling `isPalindrome`. Not only is it the smallest, but it is essentially *different* from the next longest string, `"a"`—one has characters and one doesn't.

Should the empty string be considered a palindrome? That decision is up to us, the programmer, and there is no right or wrong answer. In our case, we decided to take a very literal definition of the term "palindrome" by considering case and whitespace. In other words, our definition says that a string is a palindrome exactly when it equals its reverse. Since the reverse of `""` is also `""`, it makes sense to consider the emtpy string a palindrome.

Let's add this test case to our spec:

```
it("should consider the empty string a palindrome", function(){
   assert.strictEqual(isPalindrome(""), true);
});
```

Now run the tests, which should all pass.

You might think that another edge case is that of the longest possible palindrome. Such a palindrome would be as long as the longest possible string in JavaScript. This case is not worth considering for a couple of reasons:

- The length of the longest string can vary across different JavaScript implementations.
- The most recent JavaScript specification, ES2016, states that the maximum allowed length of a string should be $2^{53} - 1$ characters. This is a LOT of characters, and it is unrealistic to expect that our function will ever be given such a string.

14.3.5. Toward a Better Testing Workflow

In this case, we had a well-written function to write tests for, so it was straightforward to create tests that pass. Most situations will not be this simple. Your tests will often uncover bugs, forcing you to go back and update your code. That's okay! This is precisely what tests are for.

The workflow for this situation is:

1. Write code
2. Write tests
3. Fix any bugs found while testing

The rest of the chapter focuses on a programming technique that allows you to completely *eliminate* the third step, by reversing the order of the first two:

1. Write tests
2. Write code

As you will soon learn, writing your tests *before* the code is a great way to enhance your programming efficiency and quality.

14.3.6. Check Your Understanding

Let's assume we updated `isPalindrome` to be case-insensitive
(e.g. `isPalindrome('Radar')` returns `true`).

Question

Which of the following is an example of *positive* test case for checking if `isPalindrome` is case-insensitive?

a. `aa`
b. `aBa`
c. `Mom`
d. `Taco Cat`
e. `AbAb`

Question

Which of the *negative* test cases listed above are no longer valid for our case-insensitive `isPalindrome`?

a. `ab`
b. `launchcode`
c. `abA`
d. `so many dynamos`

14.4. Test-Driven Development

Now that we know more about unit testing, we are going to learn a new way of using them. So far we have written tests to verify functionality of *existing* code. Next we are going to use tests to verify functionality of code that does NOT already exist. This may sound odd, but this process has many benefits as we will learn.

As the name sounds **Test-driven development (TDD)** is a software development process where the unit tests are written first. However that doesn't tell the entire story. Writing the tests first and intentionally thinking more about the code design leads to better code. The name comes from the idea of the tests *driving* the development process.

Before we can start using TDD, we need a list of discrete features that can be turned into unit tests. This will help keep our tests focused on specific functionality which should lead to code that is easy to read. Along the way we will build confidence as we add features.

Note

TDD is a process that some organizations choose to use. Using the TDD process is not required when using unit tests.

14.4.1. The Test/Code Cycle

With TDD you start with the unit test first. As with any unit test, the test should describe a clearly described behavior that can be tested.

Example

Example test case for a data parsing project.

- Take in string of numbers delimited by a character and return an array.

Because the test is for a feature that does NOT exist *yet*, we need to think about how the feature will be implemented. This is the time to ask questions like. Should we add a new parameter? What about an entirely new function? What will the function return?

Example

How could we implement our test case? Remember we aren't writing the code yet, only thinking about the design.

- A function named `parseData`
- `parseData` will
  - have a `data` parameter will a string of data
  - have a `delimiter` parameter will be used to split the array
  - return an array
- `parseData` will be defined in a module

Next, write the unit test as if the parameter or function you imagined already exists. This may seem a bit odd, but considering how the new code will be used helps find bugs and flaws earlier. We also have to use test utilities such `assert.strictEqual` to will clearly demonstrate that the proposed new code functions properly.

Example

This is were the ideas are typed out into a test. In this example the test references a module and a function that have not been created yet. The code follows the plan we came up with earlier. Very importantly there is an `assert.strictEqual` that verifies an array is returned.

```
const assert = require('assert');
const parse = require('../parse-numbers');

describe("parse numbers", function(){

   it("returns array when passed comma separated list of numbers", function(){
      let items = parse("5,8,0,17,6,4,9,3", ",");
      assert.strictEqual(Array.isArray(items), true);
   });

});
```

Now run the test! The test should fail or possibly your code will not compile because you have referenced code that does not exist yet.

Finally, write code to pass the new test. In the earlier chapters, this is where you started, but with TDD writing new code is the *last* step.

Example

To make the above test pass a file would be created that exports a `parseData` function with logic that satisfies the expected result.

```
function parseData(text, delimiter) {
   return text.split(delimiter);
}

module.exports = parseData;
```

Coding this way builds confidence in your work. No matter how large your code base may get, you know that each part has a test to validate it's functionality.

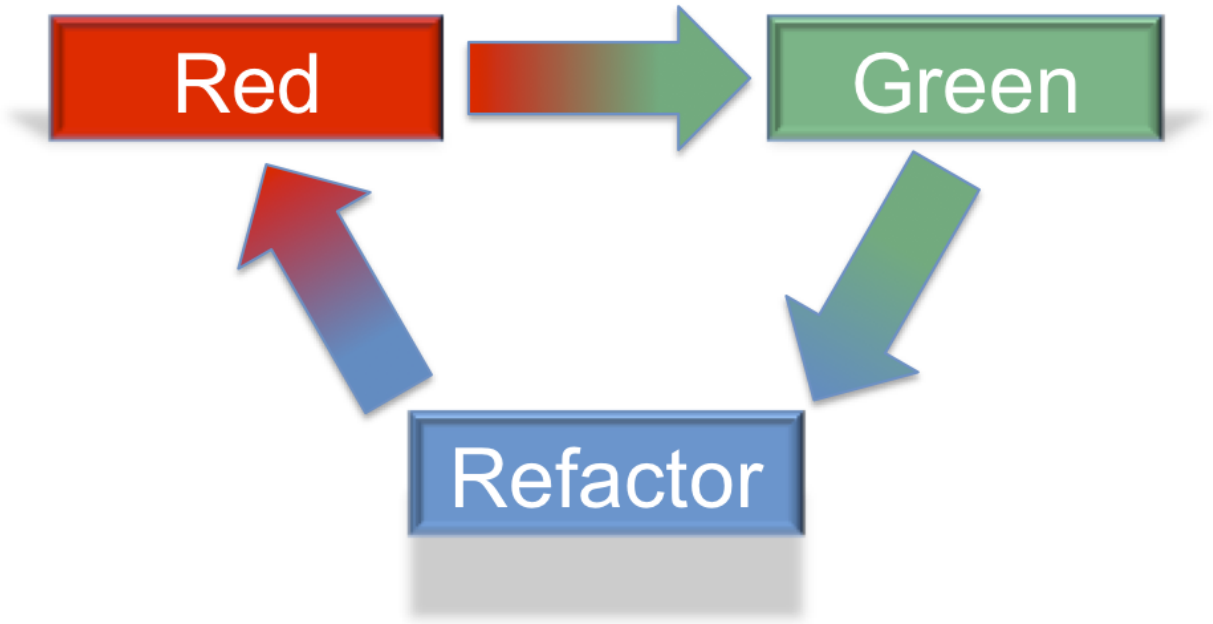
Example

Now that we have one passing test for our data parser project, we could confidently move on to writing tests and code for the remaining features.

14.4.2. Red, Green, Refactor

While adding new features and making our code work is the main goal, we also want to write readable, efficient code that makes us proud. The red, green, refactor mantra describes the process of writing tests, seeing them pass, and then making the code better. As the name suggests, the cycle consists of three steps. Red refers to test results that fail, while green represents tests that pass. The colors refer to test results which are often styled with red for failing tests and green for passing tests.

1. Red -> Write a failing test.
2. Green -> Make it pass by implementing the code.
3. Refactor -> Make the code better.



Red, green, refactor cycle.

Refactoring code means to keep the same overall feature, but change how that feature is implemented. Since we have a test to verify our code, we can change the code with confidence, knowing that any regression will be immediately identified by the test. Here are a few examples of refactoring using different data structures, reducing the number of times needed to loop through an array, or even moving duplicate logic into a function so it can be reused.

The refactor is also done in a TDD process. Decide on what is the improved way of implementing the feature and then change the unit test to use this new idea. See the test fail, then implement the refactor idea. Finally see the tests pass with the refactored design.

14.5. TDD in Action

Fork our starter code repl.it and follow along as we implement a project using TDD.

We need to write a Node module to process transmissions from the Voyager1 probe.

Example

Transmission

```
"1410::<932829840830053761>"
```

Expected Result

```
{
   id: 1410,
   rawData: 932829840830053761
}
```

14.5.1. Requirements

The features for this project have already been broken down into small testable units. Let's review them and then we will take it slow, one step at a time.

1. Take in a transmission string and return an object.
2. Return `-1` if the transmission does NOT contain `"::"`.
3. Returned object should contain an `id` property
    - The value of `id` is the part of the transmission *before* the `"::"`
4. The `id` property should be of type `Number`
5. Returned object should contain a `rawData` property
    - The value of `rawData` is the part of the transmission *after* the `"::"`
6. Return -1 for the value `rawData` if the `rawData` part of the transmission does NOT start with `<` and end with `>`

14.5.2. Requirement #1

Requirement: Take in a transmission string and return an object.

To get started on this we need to:

a. Create a blank test function.
b. Give the test a name that is a clear, testable statement.

Creating a blank test is easy, go to `processor.spec.js` and add an empty test method. Tests in Jasmine are declared with an `it` function. Remember that tests go inside of the `describe` function, which along with the string parameter describe the group of tests inside.

```
const assert = require('assert');

describe("transmission processor", function() {

  it("", function(){

  });

});

```

Give the test the name `"takes a string returns an object"`.

```
const assert = require('assert');

describe("transmission processor", function() {

   it("takes a string returns an object", function() {

   });

});

```

Now that we identified a clear goal for the test, let's add logic and `assert` calls in the test to verify the desired behavior. *But wait…* we haven't added anything except an empty at this point. There isn't any actual code to verify. That's okay, this is part of the TDD process.

We are going to think about and visualize how this feature should be implemented in code. Then we will write out in the test how this new code will be used.

We need to think of something that will satisfy the statement `it("takes a string returns an object"`. The `it` will be a function that is imported from a module. Below on line 2, a `processor` function is imported from the `processor.js` module.

```
const assert = require('assert');
const processor = require('../processor.js');

describe("transmission processor", function() {

   it("takes a string returns an object", function(){

   });

});

```

We have an idea for a function named `processor` and we have imported it. Keep in mind this function only exists as a concept and we are writing a test to see if this concept makes sense.

Now for the real heart of the test. We are going to use `assert.strictEqual` to verify that if we pass a string to `processor` that an object is returned. Carefully review lines 7 and 8 shown below.

```
const assert = require('assert');
const processor = require('../processor.js');

describe("transmission processor", function() {

   it("takes a string returns an object", function(){
      let result = processor("9701::<489584872710>");
      assert.strictEqual(typeof result, "object");
   });

});

```

On line 7 the `processor` function is called, with the value being stored in a `result` variable. On line 8 the result of the expression `typeof result` is compared to the value `"object"`. Reminder that the typeof operator returns a string representation of a type. If `typeof result` evaluates to the string `"object"`, then we know that `processor` returned an object.

14.5.2.1. Code Red

Let's run the test! Click the `run >` button in your repl.it. You should see an error about `processor.js` not existing. This makes sense, because we have not created the file yet. We are officially in the Red phase of Red, Green, Refactor!

```
Error: Cannot find module '../processor.js'
```

14.5.2.2. Go Green!

Now that we have a failing test, we have only one choice. Make it pass.

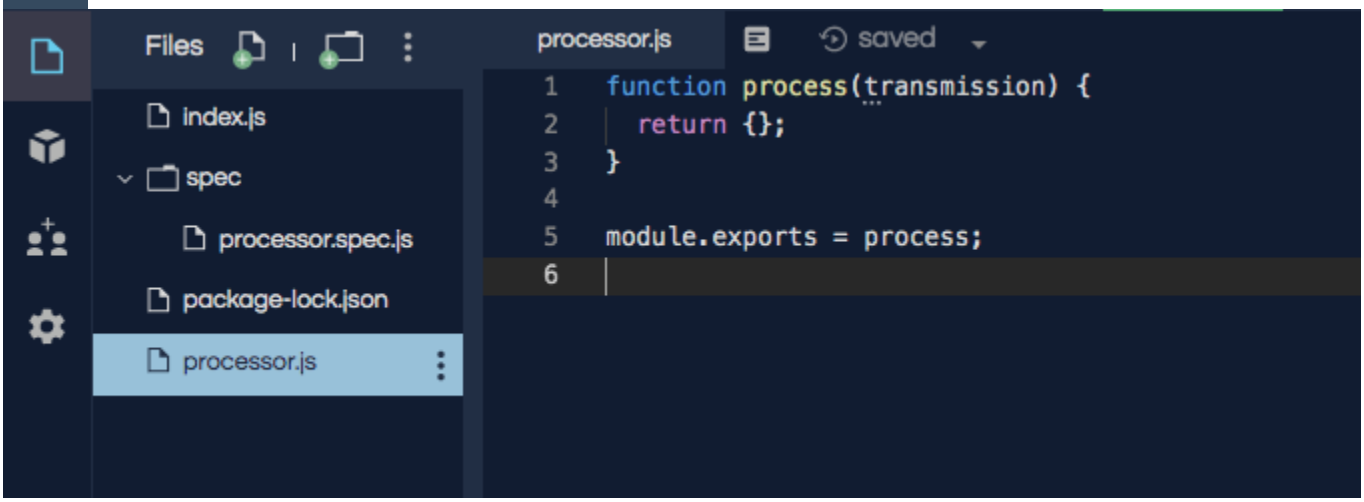
a. Add a `processor.js` file to your repl.it.
b. Inside of the module declare a `processor` function that takes a parameter and returns an object.

Contents of the new `processor.js` file.

```
function process(transmission) {
  return {};
}

module.exports = process;

```



processor.js file

Run the test again.

We did it! `1 spec, 0 failures` means 1 passing test. In repl.it you have to imagine the satisfying green color of a passing test.

```
1 spec, 0 failures
Finished in 0.011 seconds
```

14.5.2.3. Refactor if Needed

This solution is very simple and does not need to be improved. The refactor step does not always lead to an actual changing of your code. The most important part is to review your code to make sure that it's efficient and meets your team's standards.

14.5.3. Requirement #2

Requirement: Return `-1` if the transmission does NOT contain `"::"`.

Next we have a negative test requirement that tells us what should happen if the data is invalid. Before jumping into the code, let's review the steps we took to implement requirement #1.

Review of TDD process:

1. Create a blank test function.
2. Give the test a name that is a clear, testable statement.
3. Come up with test data that will trigger the described behavior.
4. Think about what is needed, then write code that fulfills the stated behavior.
5. Run the test and see the it fail.
6. Implement the new code or feature used in the test.
7. Run the test and see it pass.
8. Review to see if refactor needed.

For requirement #2, the solution for *steps 1 - 4* can be seen on lines *11 - 14* below.

```
 const assert = require('assert');
1const processor = require('../processor.js');
2
3describe("transmission processor", function() {
4
5   it("takes a string returns an object", function(){
6      let result = processor("9701::<489584872710>");
7      assert.strictEqual(typeof result, "object");
8   });
9
10  it("returns -1 if '::' not found", function(){
11     let result = processor("9701<489584872710>");
12     assert.strictEqual(result, -1);
13  });
14
15});
16
```

Now for *step 5*, run the test and see it fail. When you run the tests, you should see the below error message. Notice that `-1` was the expected value, but the actual value was `'object'`.

```
Failures:
1) transmission processor returns -1 if '::' not found
Message:
 AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
 + expected - actual

 - 'object'
 + -1
```

Next is *step 6*, write code that will make the test pass. Go to `processor.js` and update the `processor` function to check the `transmission` argument for the presence of `'::'`.

```
1function process(transmission) {
2   if (transmission.indexOf("::") < 0) {
3      // Data is invalid
4      return -1;
5   }
6
```

```
    return {};
}

module.exports = process;
```

Lucky *step 7* is to run the tests again. They should both pass.

```
2 specs, 0 failures
Finished in 0.035 seconds
```

Finally *step 8* is to review the code to see if it needs to be refactored. As with the first requirement our code is quite simple and can not be improved at this time.

14.5.4. Requirement #3

Requirement: Returned object should contain an `id` property. The `id` is the part of the transmission *before* the `"::"`

The same steps will be followed, even though they are not explicitly listed.

See lines *16 - 19* to see the test added for this requirement. To test this case `notStrictEqual` was used, which is checking if the two values are NOT equal. `notStrictEqual` is used to make sure that `result.id` is NOT equal to `undefined`. Remember that if you reference a property on an object that does NOT exist, `undefined` is returned.

```
const assert = require('assert');
const processor = require('../processor.js');

describe("transmission processor", function() {

  it("takes a string returns an object", function(){
     let result = processor("9701::<489584872710>");
     assert.strictEqual(typeof result, "object");
  });

  it("returns -1 if '::' not found", function(){
     let result = processor("9701<489584872710>");
     assert.strictEqual(result, -1);
  });

  it("returns id in object", function() {
    let result = processor("9701::<489584872710>");
    assert.notStrictEqual(result.id, undefined);
  });

});
```

The fail message looks a little different than what we have seen. The phrase “Identical input passed to notStrictEqual” lets us know that the two values were equal when we didn’t expect them to be.

```
Failures:
1) transmission processor returns id in object
Message:
   AssertionError [ERR_ASSERTION]: Identical input passed to notStrictEqual: undefined
```

The object returned from `processor` doesn’t have an id property. We need to split the transmission on `'::'` and then add that value to the object with the key `id`. See solution in `processor.js` below.

```
function process(transmission) {
   if (transmission.indexOf("::") < 0) {
      // Data is invalid
      return -1;
   }
   let parts = transmission.split("::");
   return {
      id: parts[0]
   };
}

module.exports = process;

```

Run the tests again. That did it. The tests pass! :-)

Line 6 splits `transmission` into the `parts` array, and line 8 assigns the first entry in the array to the key `id`.

```
3 specs, 0 failures
Finished in 0.011 seconds
```

14.5.5. Requirement #4

Requirement: The `id` property should be of type `Number`

Again the same steps are followed, though not listed.

New test to be added to `specs/processor.spec.js`

```
it("converts id to a number", function() {
   let result = processor("9701::<489584872710>");
   assert.strictEqual(result.id, 9701);
});

```

Fail Message

```
Failures:
1) transmission processor converts id to a number
Message:
   AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
   + expected - actual

   - '9701'
   + 9701
```

Convert the id part of the string to be of type `number`.

```
function process(transmission) {
   if (transmission.indexOf("::") < 0) {
      // Data is invalid
      return -1;
   }
   let parts = transmission.split("::");
   return {
      id: Number(parts[0])
   };
}

module.exports = process;
```

Now for the great feeling of a passing tests!

```
4 specs, 0 failures
Finished in 0.061 seconds
```

Note

You may be wondering what happens if that data is bad and the id can't be turned into a number. That is a negative test case related to this feature and is left for you to address in the final section.

14.5.6. Requirement #5

Requirement: Returned object should contain a `rawData` property. The `rawData` is the part of the transmission *after* the `"::"`

New test to be added to `specs/processor.spec.js`

```
    it("returns rawData in object", function() {
1     let result = processor("9701::<487297403495720912>");
2     assert.notStrictEqual(result.rawData, undefined);
3   });
4
```

Fail Message

```
Failures:
1) transmission processor returns rawData in object
Message:
   AssertionError [ERR_ASSERTION]: Identical input passed to notStrictEqual: undefined
```

We need to extract the rawData from the second half of the transmission string after it's been split. Then return that in the object.

```
    function process(transmission) {
1     if (transmission.indexOf("::") < 0) {
2       // Data is invalid
3       return -1;
4     }
5     let parts = transmission.split("::");
6     let rawData = parts[1];
7     return {
8       id: Number(parts[0]),
9       rawData: rawData
10    };
11  }
12
13  module.exports = process;
14
```

It's that time again, our tests pass!

```
5 specs, 0 failures
Finished in 0.041 seconds
```

14.5.7. Requirement #6

Requirement: Return -1 for the value `rawData` if the `rawData` part of the transmission does NOT start with `<` and end with `>`

Let's think about what test data to use for this requirement. What ways could the transmission data be invalid?

1. It could be missing `<` at the beginning
2. It could be missing `>` at the end
3. It could be missing both `<` and `>`
4. Has `<` but is in the wrong place
5. Has `>` but is in the wrong place

All these cases need to be covered by a test. Let's start with #1, which is missing `<` at the beginning.

New test to be added to `specs/processor.spec.js`

```
    it("returns -1 for rawData if missing < at position 0", function() {
1     let result = processor("9701::487297403495720912>");
2     assert.strictEqual(result.rawData, -1);
3   });
4
```

Fail Message

```
Failures:
1) transmission processor returns -1 for rawData if missing < at position 0
Message:
   AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
   + expected - actual

   - '487297403495720912>'
   + -1
```

New code added to `processor.js` to make tests pass. Note that we don't simply return `-1`, the requirement is to return the object and set the value of `rawData` to `-1`.

```
    function process(transmission) {
1     if (transmission.indexOf("::") < 0) {
2       // Data is invalid
3       return -1;
4     }
5     let parts = transmission.split("::");
6     let rawData = parts[1];
7     if (rawData[0] !== "<") {
8       rawData = -1;
9     }
10    return {
11      id: Number(parts[0]),
12      rawData: rawData
13    };
14  }
15
16  module.exports = process;
17
```

You know what's next, our tests pass!

```
6 specs, 0 failures
```

```
Finished in 0.056 seconds
```

Try It!

The test data we used was missing `<` at the beginning. Add tests to cover these cases. `-1` should be returned as the value for `rawData` for all of these.

- `"9701::8729740349572>0912"`
- `9701::4872<97403495720912"`
- `9701::487297403495720912"`
- `9701::<487297403495<720912>"`

14.5.8. Use TDD to Add These Features

Use the steps demonstrated above to implement all or some of the below features. Take your time, you can do it!

1. Trim leading and trailing whitespace from `transmission`.
2. Return -1 if the `id` part of the `transmission` can not be converted to a number.
3. Return -1 if more than one `"::"` found in `transmission`
4. Return -1 for value of `rawData` if anything besides numbers are present
5. Allow for multiple `rawData` values
    - `rawData` would be returned as an array of numbers
    - Get the new test working and then fix any broken existing tests
    - Example Transmission: `"9701::<21212.232323.242424>"`
    - Result: `{ id: 9701, rawData: [21212,232323,242424] }`

14.6. Exercises: Unit Testing

In many of your previous coding tasks, you had to verify that your code worked before moving to the next step. This often required you to add `console.log` statements to your code to check the value stored in a variable or returned from a function. This approach finds and fixes syntax, reference, or logic errors AFTER you write your code.

In this chapter, you learned how to use unit testing to solve coding errors. Even better, you learned how to PREVENT mistakes by writing test cases before completing the code. The exercises below offer practice with using tests to find bugs, and the studio asks you to implement TDD.

14.6.1. Automatic Testing to Find Errors

Let's begin with the following, simple code:

```
function checkFive(num){
   let result = '';
   if (num < 5){
      result = num + " is less than 5.";
   } else if (num === 5){
      result = num + " is equal to 5.";
   } else {
      result = num + "is greater than 5.";
   }

   return result;
}
```

repl.it

The function checks to see if a number is greater than, less than, or equal to 5. We do not really need a function to do this, but it provides good practice for writing test cases.

Note that the repl.it contains three files:

a. `checkFive.js`, which holds the code for the function,
b. `checkFive.spec.js`, which will hold the testing code,
c. `index.js` which holds special code to make Jasmine work.

Warning

Do NOT change the code in `index.js`. Messing with this file will disrupt the automatic testing.

1. We need to add a few lines to `checkFive.js` and `checkFive.spec.js` to get them to talk to each other.
    a. `checkFive.spec.js` needs to access `checkFive.js`, and we also need to import the `assert` testing function. Add two `require` statements to accomplish this (review Unit Testing in Action if needed).
    b. Make the `checkFive` function available to the spec file, by using `module.exports` (review Unit Testing in Action if needed).
2. Set up your first test for the `checkFive` function. In the `checkFive.spec.js` file, add a `describe` function with one `it` clause:

```
3. const checkFive = require('../checkFive.js');
```

```
4. const assert = require('assert');
5.
6. describe("checkFive", function(){
7.
8.    it("Descriptive feedback...", function(){
9.       //code here...
10.   });
11.
12.});
```

13. Now write a test to see if `checkFive` produces the correct output when passed a number *less than 5*.

    a. First, replace `Descriptive feedback...` with a DETAILED message. This is the text that the user will see if the test *fails*. Do NOT skimp on this. Refer back to the Specifications and Assertions section to review best practices.

    b. Define the variable `output`, and initialize it by passing a value of `2` to `checkFive`.

```
c. const checkFive = require('../checkFive.js');
d. const assert = require('assert');
e.
f. describe("checkFive", function(){
g.
h.    it("Descriptive feedback...", function(){
i.       let output = checkFive(2);
j.    });
k.
l. });
```

    m. Now use the `assert` function to check the result:

```
n. const checkFive = require('../checkFive.js');
o. const assert = require('assert');
p.
q. describe("checkFive", function(){
r.
s.    it("Descriptive feedback...", function(){
t.       let output = checkFive(2);
u.       assert.strictEqual(output, "2 is less than 5.");
v.    });
w.
x. });
```

    y. Run the test script and examine the results. The test should pass and produce output similar to:

```
z. Started
aa..
bb.
cc.1 spec, 0 failures
dd.Finished in 0.006 seconds
```

    ee. Now change line 3 in `checkFive.js` to `if (num > 5)` and rerun the test. The output should look similar to :

```
ff.Started
gg.F
hh.
ii.Failures:
jj.1) checkFive should return 'num' is less than 5 when passed a number smaller than
   5.
```

```
kk.Message:
ll.   AssertionError [ERR_ASSERTION]: Input A expected to strictly equal input B:
mm.   + expected - actual
nn.
oo.   - '2 is greater than 5.'
pp.   + '2 is less than 5.'
```

qq. Change line 3 back.

Note

We do NOT need to check every possible value that is less than 5. Testing a single example is sufficient to check that part of the function.

14. Add two more `it` clauses inside `describe`—one to test what happens when `checkFive` is passed a value greater than 5, and the other to test when the value equals 5.

14.6.2. Try One on Your Own

Time for Rock, Paper, Scissors! The function below takes the choices (`'rock'`, `'paper'`, or `'scissors'`) of two players as its parameters. It then decides which player won the match and returns a string.

```
function whoWon(player1,player2){

   if (player1 === player2){
      return 'TIE!';
   }

   if (player1 === 'rock' && player2 === 'paper'){
      return 'Player 2 wins!';
   }

   if (player1 === 'paper' && player2 === 'scissors'){
      return 'Player 2 wins!';
   }

   if (player1 === 'scissors' && player2 === 'rock '){
      return 'Player 2 wins!';
   }

   return 'Player 1 wins!';
}
```

repl.it

1. Set up the `RPS.js` and `RPS.spec.js` files to talk to each other. If you need to review how to do this, re-read the previous exercise, or check Hello Jasmine.
2. Write a test in `RPS.spec.js` to check if `whoWon` behaves correctly when the players tie (both choose the same option). Click “Run” and examine the output. SPOILER ALERT: The code for checking ties is correct in `whoWon`, so the test should pass. If it does not, modify your `it` statement.
3. Write tests (one at a time) for each of the remaining cases. Run the tests after each addition, and modify the code as needed. There is one mistake in `whoWon`. You might spot it on your own, but try to use automated testing to identify and fix it.

14.6.3. Bonus Mission

What if something OTHER than `'rock'`, `'paper'`, or `'scissors'` is passed into the `whoWon` function? Modify the code to deal with the possibility.

Don't forget to add another `it` clause in `RPS.spec.js` to test for this case.

14.7. Studio: Unit Testing

Let's use Test Driven Development (TDD) to help us design a function that meets the following conditions:

1. When passed a number that is ONLY divisible by 2, return `'Launch!'`
2. When passed a number that is ONLY divisible by 3, return `'Code!'`
3. When passed a number that is ONLY divisible by 5, return `'Rocks!'`
4. When passed a number that is divisible by 2 AND 3, return `'LaunchCode!'`
5. When passed a number that is divisible by 3 AND 5, return `'Code Rocks!'`
6. When passed a number that is divisible by 2 AND 5, return `'Launch Rocks!'`
7. When passed a number that is divisible by 2, 3, AND 5, return `'LaunchCode Rocks!'`
8. When passed a number that is NOT divisible by 2, 3, or 5, return `'Rutabagas! That doesn't work.'`

Rather than complete the code and *then* test it, TDD flips the process:

1. Write a test first - one that checks the program for a specific outcome.
2. Run the test to make sure it fails.
3. Write a code snippet that passes the test.
4. Repeat steps 1 - 3 for the remaining features of the program.
5. Examine the code and test scripts, and refactor them to increase efficiency. Remember the DRY idea (Don't Repeat Yourself).

14.7.1. Source Code

Open this repl.it and note the expected files:

1. `index.js` holds the Jasmine script. DO NOT CHANGE THIS FILE.
2. `launchCodeRocks.js` holds the function we want to design, which we will call `launchOutput`.
3. `launchCodeRocks.spec.js` holds the testing script.

Besides `index.js` the files are mostly empty. Only a framework has been provided for you.

14.7.2. Write the First Test

In `launchCodeRocks.spec.js`, complete the `describe` function by adding a test for condition #1:

When passed a number that is ONLY divisible by 2, `launchOutput` returns `'Launch!'`

Run the test. It should fail because there is no code inside `launchOutput` yet!

14.7.3. Write Code to Pass the First Test

In `launchCodeRocks.js`, use an `if` statement inside the `launchOutput` function to check if the parameter is evenly divisible by 2, and then return an output. (*Hint: modulus*).

Run the test script again to see if your code passes. If not, modify `launchOutput` until it does.

14.7.4. Write the Next Two Tests

In `launchCodeRocks.spec.js`, add tests for the conditions:

2. When passed a number that is ONLY divisible by 3, `launchOutput` returns `'Code!'`
3. When passed a number that is ONLY divisible by 5, `launchOutput` returns `'Rocks!'`

Run the tests (you have three now). The two new ones should fail, but the first should still pass. Modify the `it` statements as needed if you see a different result.

14.7.5. Write Code to Pass the New Tests

Add more code inside `launchOutput` to check if the parameter is evenly divisible by 2, 3, or 5, and then return an output based on the result.

Run the test script again to see if your code passes all three tests. If not, modify `launchOutput` until it does.

14.7.6. Hmmm, Tricky

In `launchCodeRocks.spec.js`, add a test for the condition:

4. When passed a number that is divisible by 2 AND 3, `launchOutput` returns `'LaunchCode!'` (not `'Launch!Code!'`).

Run the tests. Only the new one should fail.

Modify `launchOutput` until the function passes all four of the tests.

14.7.7. More Tests and Code Snippets

Continue adding ONE test at a time for the remaining conditions. After you add EACH new test, run the script to make sure it FAILS, while the previous tests still pass.

Modify `launchOutput` until the function passes the new test and all of the old ones.

Presto! By starting with the *testing* script, you constructed `launchOutput` one segment at a time. The result is complete, valid code that has already been checked for accuracy.

14.7.8. New Condition

Now that your function passes all 8 tests, let's change one of the conditions. For the case where a number is divisible by both 2 and 5, instead of returning `'Launch Rocks!'`, we want the function to return `'Launch Rocks! (CRASH!!!!)'`.

Modify the testing and function code to deal with this new condition.

14.7.9. Bonus Missions

14.7.9.1. DRYing the Code

Examine `launchOutput` and the `describe` functions. Notice that there is quite a bit of repetition in the code.

Try adding arrays, objects and/or loops to refactor the code into a more efficient structure.

14.7.9.2. What if We Already Have Code?

A teammate tried to help you out by writing the `launchOutput` code before class. Unfortunately, the code contains some errors.

Open the flawed code here, and cut and paste your testing script into the `launchCodeRocks.spec.js` file.

Run the tests and see how many fail. Use the *descriptive feedback* from your `it` statements to find and fix the errors in `launchOutput`.

Note: You could just *cheat* and compare your correct code to the flawed sample, but the point of this mission is to give you more practice interpreting and using test results. To gain the most benefit, honor the spirit of this task.