Test-Driven Development: A Case Study

- We've talked previously about ***test-driven development*** (i.e. ***TDD***). TDD is an important modern development strategy, but it doesn't always feel immediately intuitive.

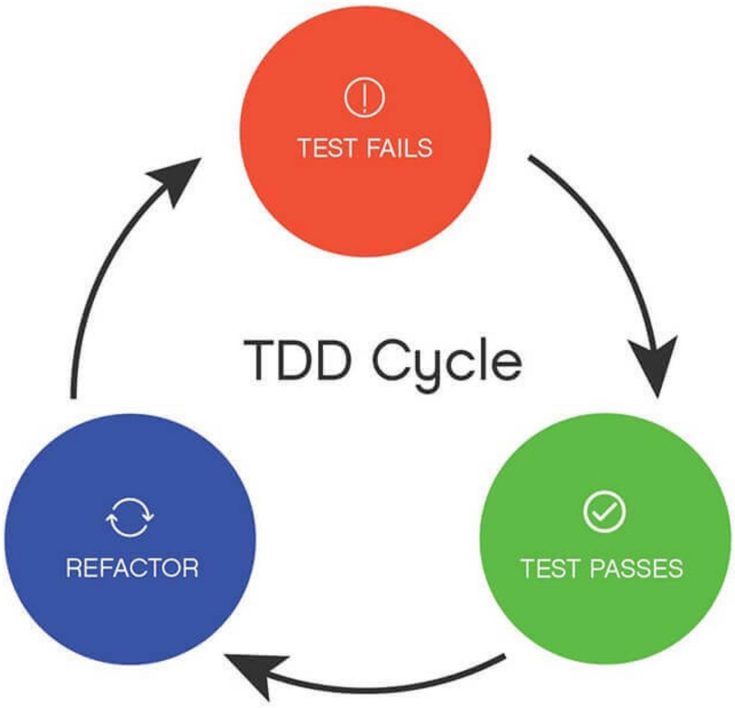
- For this reason, let's spend some time going through a real example of how TDD works. Hopefully, through this example, we can draw out some of the nuances of TDD so you feel comfortable using it.

- The example we'll walk through here will involve implementing a ROT13 cipher algorithm, inspired by James Shore's TDD example.

- Before we get into the example, let's do a brief review of what TDD is and how it works.

Test-driven development: a review

- Remember that TDD is a development strategy that places automated testing at the core of software development. It consists of a cycle of testing, coding, and refactoring that's often called the "red, green, refactor" cycle:



Courtesy of Kent C. Dodds

- The steps of this cycle, in more detail are:
    - ❌ **Red** – Start by writing a test that contains just enough code to test the behavior of the next feature you want to implement. Since the feature isn't implemented yet, this test will fail (giving you a red error message).
    - ✅ **Green** – Next, write just enough code to make your test pass (giving you a green success message) without worrying too much about design or elegance.
    - 🔄 **Refactor** – Once your test is passing, refactor your code to make sure it's clean and well-written. Your test should continue to pass as you refactor, and if it fails, that's a signal that your refactoring broke something.
- This cycle is repeated for each new feature/behavior you want to add to your code.
- **One of the major benefits of TDD is that writing the test first forces you to think first about your code's interface and behavior.** Implicit in the "red" phase of the red, green, refactor cycle is an exercise of design.

- In other words, in order to implement a test for code you haven't written yet, you need to make decisions about how you will invoke that code, what its behaviors will be, and how it will communicate its results back to you.

- Making these decisions within the context of a test forces you to think about your code from the perspective of its user (in this case, the test). Doing this up front can help yield code with an interface that's clear and simple, i.e. code that's easy to use and understand.

- **One of the keys to successfully employing TDD is working in very small increments.** This is something we'll focus on in the example below.

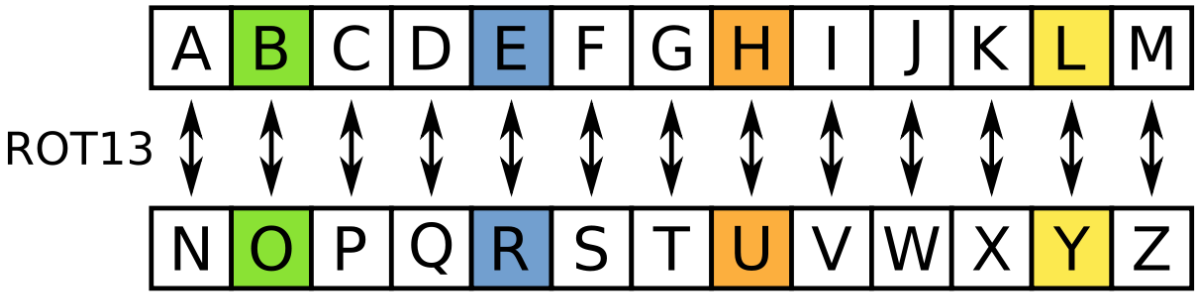
- In particular, with TDD, we start by implementing a single test that's very limited in scope, then we write just enough code to make that test pass. We repeat this process until an entire feature is implemented. Oftentimes, this will require breaking a problem that itself seems quite simple into even smaller steps.

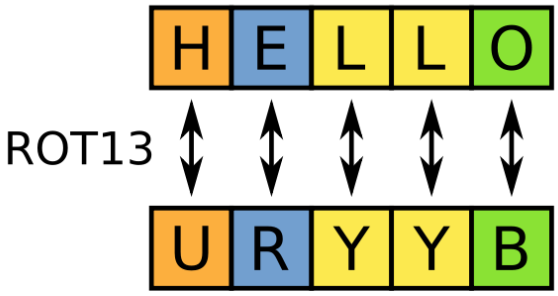
- **Importantly, TDD does *not* require that we implement *all* of our tests up front.** In fact, this is an anti-pattern in TDD. Instead, with TDD, we're always working on just one test at a time.

The ROT13 algorithm

- The ROT13 algorithm, which we'll implement here, is an extremely basic (and not very secure) cipher algorithm that "rotates" each letter in a plaintext string by 13 places to produce an encoded ciphertext string.

- The "rotation" that takes place in the ROT13 algorithm simply involves counting forward 13 places in the alphabet, looping back around to the beginning if we count forward past the end. The following picture depicts this rotation for every letter in the alphabet as well as the encoding for the string "HELLO" (which is "URYYB"):





From Wikipedia

- The version of ROT13 we'll implement here will always rotate an uppercase letter to another uppercase letter and a lowercase letter to another lowercase letter, and it won't rotate numbers or special characters at all.

- This version of the cipher has the special feature that it is its own inverse. In other words, for any uppercase or lowercase letter `a`, the following will hold:

  ```
  rot13(rot13(a)) = a
  ```

- This is why we see the bidirectional arrows in the image above.

Our first TDD cycle: Working in (very) small steps

- **Remember, the key to making TDD work is working in very small increments.** We'll practice that here with the ROT13 algorithm. At first, the increments in which we work may seem almost comically small, but in the context of TDD, taking very small steps helps make it easy to catch mistakes and, ultimately, we can move faster this way.

- Let's follow one trip around the "red, green, refactor" cycle to get a feel for how to work in very small increments.

Red: Design + test implementation

- Again, one of the primary benefits of writing a test first is that it forces us to think up front about our code's interface and its behavior. In particular, in order to implement a test, we need to know up front how we're going to call into our code as well as how we want it to behave. This is what we do in the "red" phase of the TDD cycle.

- When implementing our very first test, we want to focus on the very top level interface into our code. For example, here, we might want to decide whether we want to create a class or whether we'll just implement our ROT13 algorithm in a function.

- In this case, a full class might be overkill, since we really just have one function to implement, so let's implement a module from which we export a `rot13()` function.

- **The first test we implement for our function should test its *very* most basic behavior.** We might be tempted to write our first test to make sure our ROT13 implementation correctly transforms a single letter or something like that, but even that would be too big a step to start with.

- Instead, we'll start by writing a test verifying that our ROT13 function returns nothing (i.e. the empty string) when we pass it nothing (i.e. the empty string). This test will establish the basic interface we designed above. Here's what it will look like (in the file `__tests__/rot13.js`):

```
const rot13 = require("../rot13")

test("returns empty string for empty input", function () {
    expect(rot13("")).toBe("")
})
```

- Before we move onto the "green" phase of the TDD cycle, and **before even running the test, we typically want to make a hypothesis about what will happen when we run the test we just wrote**. In this case, we might

hypothesize something like “the test will fail because our `rot13` module doesn’t even exist yet.

- When we actually run the test, this is exactly what we’ll see. Let’s move onto the “green” phase of the TDD cycle.

Green: Make the test pass

- Our goal for the “green” phase of the TDD cycle is to **write just enough code to make the test we just implemented pass**. Here, we don’t have to worry about writing the most efficient code, since we’ll clean things up in the “refactor” phase.

- Here, all we need to do to make the test pass is create our basic interface and hardcode the return value (in the file `rot13.js`):

```
module.exports = function rot13() {
    return ""
}
```

- Note what a small step this is. We’ve barely even made any progress on our ROT13 function, only putting its most basic interface in place (and not even completely: notice the function we implemented doesn’t even take parameters yet). This is the way TDD works.

- You may be concerned that we won’t make much progress if we take such small steps, but (hopefully) getting to this point should have gone very quickly. It might have taken a minute or less to write the first test and to implement the code that made it pass, and as a result, we already have a meaningful test and the basic interface for the feature we’re implementing.

- This is the key to successfully employing TDD. By taking small steps, we move quickly, and the tests we implement verify that everything is working correctly.

Refactor: Clean things up

- The last phase of the TDD cycle, the “refactor” phase, gives us an opportunity to clean up the code we wrote in the “red” and “green” phases.

- We can do any kind of refactoring we need to here to make sure our code is well implemented, our test is clear and concise, etc. We could even decide to choose

a more appropriate name for our test if we wanted to.

- In this case, though, there's not much we can do in this phase.

Our second TDD cycle: Adding a (small) bit of logic

- Now we're ready for our second TDD cycle. In this cycle, we're ready to start working on the core logic of our ROT13 function.

- Again, we need to figure out how big of a step to take in this cycle. We know that we'll eventually need to implement some kind of loop through the characters of the input string, but this is way too big a step for a single TDD cycle.

- We might be tempted to take a step like "successfully transforms any single character", but even this would be too big a step. We want to make even more incremental progress on each TDD cycle.

- In particular, for this next cycle, let's decide to add functionality for successfully transforming a single, specific character. Specifically, let's just make sure we can transform a single lowercase letter forward by 13 letters. We won't worry about looping around, or uppercase letters, or numbers and symbols yet. We can implement that functionality later.

Red: Design + test implementation

- Again, the very small step we're taking here is to get our ROT13 function to successfully transform a single, specific lowercase letter forward by 13 letters without looping. Here's a test for this small step:

```
test("transforms one lowercase letter w/o looping",
function () {
    expect(rot13("a")).toBe("n")
})
```

- Before running the test and moving on to the "green" phase, let's make a hypothesis about what's going to happen when we run this test. In this case, we should expect that this test will fail because `rot13()` will still return `""` instead of `"n"`. This is indeed what happens.

Green: Make the test pass

- Let's write some code to make our test pass. To do this, we'll actually have to add a little transformation logic to our `rot13()` function. In addition, we'll have to set `rot13()` up to accept input, which we didn't need to do to make our first test pass. Let's start with this:

```
module.exports = function rot13(input) {...}
```

- As we continue to build onto our `rot13()` function, we want to make sure to retain the behaviors that we implemented to make previous tests pass. Here, to make sure our first test continues to pass, we need to make sure we still return the empty string if the empty string if it receives empty input:

```
if (input === "") {
    return ""
}
```

- Now, we have to start thinking about how to actually transform letters forward by 13 places. Note that because we're using TDD and taking such a small step forward in our implementation here, we can solve this problem in isolation, without worrying about other questions about our implementation.

- To do the actual transformation of letters, we can use functionality built into the JavaScript String class that allows us to represent individual characters as numerical codes. Specifically, the JS String class has the following methods built into it:
    - `charCodeAt()` – This method transforms a single character from a string into a numerical code (specifically a UTF-16 code). It is called on an existing string and takes the index of the character whose code should be returned. For example, `"abc".charCodeAt(0)` will return the UTF-16 code for the character `"a"`, which is 97.
    - `fromCharCode()` – This method generates a string from one or more UTF-16 character codes. It is a static method on the String class, so it is called like this: `String.fromCharCode()`. It returns the string generated from the character codes passed as arguments. For example, `String.fromCharCode(110)` will return the string `"n"`.

- Putting these pieces together, we can add the following code to our `rot13()` function to make our test pass:

```
const charCode = input.charCodeAt(0)
return String.fromCharCode(charCode + 13)
```

- As before, we write just enough code to make our test pass here and no more.

Refactor: Clean things up

- Things are written pretty cleanly for now, so there aren't many opportunities to refactor.

A few more TDD cycles to handle single letters

- Now that we've got a bit of a feel for how TDD works, let's take the next few steps more rapidly. Our goal for now is to make enough TDD cycles to be able to handle single letter inputs successfully.

Handling looping for lowercase letters

- As our next step, let's add functionality to handle transforming a single lowercase letter while looping around past "z" back to "a". Here's a test for this behavior:

```
test("transforms one lowercase letter with looping",
function () {
    expect(rot13("n")).toBe("a")
})
```

- To be able to handle this behavior correctly, we'll need to be able to ask whether the letter being transformed is in the first half of the alphabet (in which case we'll transform it forward by 13 letters) or in the second half of the alphabet (in which case, we'll transform it *backward* by 13 letters, which has the same effect as going forward 13 places with looping).

- To handle this, let's implement a couple utility functions in our ROT13 module. Specifically, we'll implement an `isBetween()` function that allows us to ask if a character's numeric code is between the codes corresponding to any two letters. To make this easier, we'll also write a function `charCodeFor()` that makes it

easy (and syntactically nicer) to access the character code for a letter:

```
function isBetween(charCode, firstLetter, lastLetter) {
    return charCode >= charCodeFor(firstLetter) &&
        charCode <= charCodeFor(lastLetter)
}

function charCodeFor(letter) {
    return letter.charCodeAt(0)
}
```

- With those utility functions in place, we can modify the core functionality of our `rot13()` function to correctly handle letters in the second half of the alphabet:

```
const charCode = input.charCodeAt(0)
if (isBetween(charCode, "a", "m")) {
    return String.fromCharCode(charCode + 13)
} else if (isBetween(charCode, "n", "z")) {
    return String.fromCharCode(charCode - 13)
}
```

- With this modification, our most recent test should pass, along with the ones we've previously written, and there don't appear to be good opportunities for refactoring right now.

Handling uppercase letters without looping

- Next, let's start to handle uppercase letters. Even though it's probably very clear how to handle uppercase letters with and without looping, let's still break this into two TDD cycles to stick with implementing things in very small increments.

- Here's a test to ensure our ROT13 function correctly transforms uppercase letters without looping:

```
test("transforms one uppercase letter w/o looping",
function () {
    expect(rot13("A")).toBe("N")
```

```
})
```

- To make this test pass, we just need to make a small modification to our `rot13()` function, adding an uppercase version of the check to see if the letter being transformed is in the first half of the alphabet:

```
if (
    isBetween(charCode, "a", "m")
    || isBetween(charCode, "A", "M")
) {
    return String.fromCharCode(charCode + 13)
}
```

Handling uppercase letters with looping

- To finish with uppercase letters, we have to handle uppercase letters in the second half of the alphabet. Here's a test to for this behavior:

```
test("transforms one uppercase letter with looping",
function () {
    expect(rot13("N")).toBe("A")
})
```

- To make this test pass, we just need to make a modification to our `rot13()` function similar to the one we just made, adding an uppercase version of the check to see if the letter being transformed is in the second half of the alphabet:

```
else f (
    isBetween(charCode, "n", "z")
    || isBetween(charCode, "N", "Z")
) {
    return String.fromCharCode(charCode - 13)
}
```

TDD cycles for handling boundary cases

- Now, let's go through some TDD cycles to finish up with single letter inputs by handling symbols and numbers, none of which should be transformed.

- There are a lot of symbols and numbers, and we won't be able to write tests to cover them all, so instead, let's just focus on tests for symbols and numbers that represent boundary cases.

- In particular, let's implement tests to cover the boundary cases corresponding to the first character before `a` (based on character code), the first character after `z`, the first character before `A`, and the first character after `Z`. We can discover these characters by executing the following lines of JS (e.g. by entering them into a JS terminal):

```
String.fromCharCode("a".charCodeAt(0) - 1)  // "`"
String.fromCharCode("z".charCodeAt(0) + 1)  // "{"
String.fromCharCode("A".charCodeAt(0) - 1)  // "@"
String.fromCharCode("Z".charCodeAt(0) + 1)  // "["
```

- So, the four boundary cases we'll test for are `` ` ``, `{`, `@`, and `[`.

- Let's implement a test for the first of these:

```
test("doesn't transform '`' (1st char before 'a')",
function () {
    expect(rot13("`")).toBe("`")
})
```

- In order to make this test pass, we simply need to add an else statement to the core logic of our `rot13()` function to make sure non-letter characters are not transformed:

```
else {
    return String.fromCharCode(charCode)
}
```

- Now, we can go ahead and add three tests for the three other boundary cases:

```
test("doesn't transform '{' (1st char after 'z')", function
() {
    expect(rot13("{")).toBe("{")
})

test("doesn't transform '@' (1st char before '@')",
function () {
    expect(rot13("@")).toBe("@")
})

test("doesn't transform '[' (1st char after '[')", function
() {
    expect(rot13("[")).toBe("[")
})
```

- In fact, these tests will all pass already without adding any additional code to our `rot13()` function.

One last refactoring step

- Once we've finished completely implementing single-letter transform behaviors, we can do one final refactoring step with an eye towards what we'll do next.

- In particular, at this point, we'll probably anticipate that the next thing we'll want to do is add functionality to transform multi-character strings.

- With this in mind, let's refactor our `rot13()` function so that the character transforming functionality is easily reusable, i.e. by moving that functionality into its own function:

```
function transformLetter(charCode) {
    if (
        isBetween(charCode, "a", "m")
        || isBetween(charCode, "A", "M")
    ) {
```

```
        return String.fromCharCode(charCode + 13)
    } else if (
        isBetween(charCode, "n", "z")
        || isBetween(charCode, "N", "Z")
    ) {
        return String.fromCharCode(charCode - 13)
    } else {
        return String.fromCharCode(charCode)
    }
}
```

- With that function implemented, we can simplify the `rot13()` function itself:

```
module.exports = function rot13(input) {
    if (input === "") {
        return ""
    }
    const charCode = input.charCodeAt(0)
    return transformLetter(charCode)
}
```

- After making these changes, we should run the tests to verify that we haven't broken anything.

A TDD cycle to handle multi-character strings

- Let's work on supporting multi-character strings. We'll track the complete TDD cycle for this.

Red: Design + test implementation

- Our `rot13()` function should handle multi-character strings in just the same way it handles single-character strings. Let's implement a test that passes a multi-character string to `rot13()`:

```
test("transforms a multi-character string", function () {
    expect(rot13("abc")).toBe("nop")
```

```
})
```

- As before, let's make a hypothesis after implementing this test. What do we expect will happen when we run this test?

- In this case, remember that we *are* correctly grabbing and transforming the first letter of the input string, so might expect that `rot13()` will transform the `a` into an `n` but miss the remaining characters `b` and `c`. This is indeed what happens when we run the test.

Green: Make the test pass

- To make our new test pass, we'll need to make `rot13()` process the entire input string. We can do this easily by adding a loop:

```
module.exports = function rot13(input) {
    let result = ""
    for (let i = 0; i < input.length; i++) {
        const charCode = input.charCodeAt(i)
        result += transformLetter(charCode)
    }
    return result
}
```

- If we run the tests after making that change, we'll see that they all pass.

Refactor: Clean things up

- We have another opportunity to do some refactoring now to clean up our tests a bit.

- In particular, now that we have the capability to transform multi-character strings, we can take advantage of this fact to easily implement tests that cover all lowercase letters, all uppercase letters, and all of our boundary conditions at once.

- Let's start by refactoring the test we just implemented to cover all lowercase letters:

```
~~test("transforms a multi-character string", function () {~~
test("transforms all lowercase letters", function () {
    expect(rot13("abcdefghijklmnopqrstuvwxyz"))
        .toBe("nopqrstuvwxyzabcdefghijklm")
})
```

- Then, let's add a test to cover all uppercase letters and one to cover our boundary-case symbols:

```
test("transforms all uppercase letters", function () {
    expect(rot13("ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
        .toBe("NOPQRSTUVWXYZABCDEFGHIJKLM")
})

test("doesn't transform multiple symbols", function () {
    expect(rot13("`{@[")).toBe("`{@[")
})
```

- At this point, since we have tests covering all lowercase letters, all uppercase letters, and all of our boundary cases, we might choose to get rid of some of our single-character tests, since in some ways they are now redundant. We'll choose to keep them, since it could be useful to have single-character tests, in case we change the implementation of `rot13()` at some point.

Some last TDD cycles for adding error/special case handling

- To wrap things up, we'll do a couple final TDD cycles to add error handling and to handle special input cases, such as emojis.

- Let's start with error handling. Remember, the "red" phase of the TDD cycle is our place to make design decisions about how we want to implement things. We figure out what we want our code's interface to look like so we can test against that interface. This includes the error interface.

- For error handling, let's make the design decision that when our `rot13()` function receives some kind of bad input, it will throw an `Error`.

- For our first TDD cycle here, let's follow this design decision and implement error handling for the case when our `rot13()` function isn't passed a parameter (i.e. when `input` is `undefined`). Here's a test that captures how we want our function to behave in this situation (remember, in order to use Jest's `toThrow()` matcher, `expect()` must be passed a *function* that we expect to throw an error when called):

```
test("throws an error when no parameter passed", function
() {
    expect(function () {
        rot13()
    }).toThrow("Expected string parameter")
})
```

- Now, in the "green" cycle, we can implement code to make this test pass. We can do this by adding a simple parameter check at the beginning of `rot13()`:

```
if (input === undefined) {
    throw new Error("Expected string parameter")
}
```

- We can go through a second TDD cycle to add error handling for non-string inputs, again ensuring that `rot13()` throws an error in this situation. Here's a test for this behavior:

```
test("throws an error when non-string passed", function ()
{
    expect(function () {
        rot13(123)
    }).toThrow("Expected string parameter")
})
```

- We can make this test pass by adding an extra clause to the parameter check we just added:

```
if (input === undefined || typeof input !== "string") {
    throw new Error("Expected string parameter")
}
```

- Finally, we'll add a few last tests to verify that `rot13()` behaves correctly for various special-case inputs:

```
test("doesn't transform numbers", function () {
    expect(rot13("0123456789")).toBe("0123456789")
})

test("doesn't transform non-English letters", function () {
    expect(rot13("ñåéîøüç")).toBe("ñåéîøüç")
})

test("handles emojis", function () {
    expect(rot13("🤓🤩")).toBe("🤓🤩")
})
```

- And that's it! We've now fully implemented our ROT13 function using TDD, and as a result, we have a whole suite of tests for it already.

- Hopefully this exercise gives you a better idea about how TDD works, so you can start to decide whether/when you want to incorporate it into your own development flow.

- In reality, TDD may not be useful in all situations, but it can be a very useful tool to pull out when it's appropriate. If you start to experiment with TDD, you should start to develop more of an intuition about which situations it's useful for you in.