# Intro to JavaScript tutorial

In this tutorial, you'll run unit tests on a collection of JavaScript functions. At first, all the tests are failing. You'll make changes in each step so that by the end, all the tests are passing.

## Step One: Start the tests

To get started, open this tutorial folder in Visual Studio Code.

In addition to this README, there are three other files:

- `tests.html` - an HTML file for displaying test results
- `tests.js` - a JavaScript file containing tests for `tutorial.js`
- `tutorial.js` - the JavaScript file you'll modify

Right-click `tests.html` and select "Open with Live Server." In your browser, you'll see a list of failing tests. Leave that browser window open as you work through the rest of the steps in the tutorial.

## Step Two: Reassign a variable

In Visual Studio Code, open `tutorial.js`, and look at the first function:

```
function stepTwo() {
  const result = false;
  result = true;
  return result;
}
```

This function declares a variable named `result` using the keyword `const`, and assigns it the value `false`. When you use `const` to declare a variable, that variable can't be reassigned. Since the next line attempts to assign the value `true` to `result`, an error occurs and the test fails. To make the test pass, replace `const` with `let`:

```
function stepTwo() {
  let result = false;
  result = true;
  return result;
}
```

When you declare a variable with `let`, that variable can be reassigned. After you change the `const` to `let` and save the file, you'll see the "Step two" test pass.

It's a good habit to declare variables with `const` by default. If you later find that the variable needs to change, you replace the `const` with a `let` like you did here.

## Step Three: Add values to an array

The next function creates an empty array and assigns it to a variable named `values`:

```
function stepThree() {
  const values = [];
  return values;
}
```

The test for step three is failing because it expects the array to contain a boolean, a number, and a string. JavaScript is a loosely typed language.

If you look at the `stepTwo()` function again, you'll see that the `result` variable wasn't declared a boolean. JavaScript variables can refer to data of any type. Similarly, JavaScript arrays can include a variety of data types.

To add items to an array, use the `push` method of the array. Add a boolean, number, and string to the array:

```
function stepThree() {
  const values = [];
  values.push(false);
  values.push(99.99);
  values.push('example');
  return values;
}
```

Now the test passes. Notice that `values` being declared with `const` didn't prevent you from adding items to the array. All that `const` prevents is reassignment, so you can still make changes to the array `values` refers to, but you can't assign a different array to `values`.

## Step Four: Round a number to two decimal places

The next function assigns the result of dividing 2 by 3 (0.6666666666666666) to a variable named `twoThirds`, and then assigns that value to a variable named `roundedTwoThirds`:

```
function stepFour() {
  const twoThirds = 2 / 3;
  const roundedTwoThirds = twoThirds;
  return roundedTwoThirds;
}
```

The test for step four is failing because it expects `roundedTwoThirds` to be rounded to two decimal places. You can round a number to any number of decimal places using its `toFixed` method:

```
function stepFour() {
  const twoThirds = 2 / 3;
  const roundedTwoThirds = twoThirds.toFixed(2);
```

```
    return roundedTwoThirds;
  }
```

Notice, though, that after you make this change the test still doesn't pass. The message from the failing test says, "expected '0.67' to equal 0.67" (the quotes around '0.67' indicate it's a string). That's because the `toFixed` method returns a string rather than a number. To convert that string to a number, use the `Number.parseFloat` method:

```
  function stepFour() {
    const twoThirds = 2 / 3;
    const roundedTwoThirds = Number.parseFloat(twoThirds.toFixed(2));
    return roundedTwoThirds;
  }
```

Now the test passes.

## Step Five: Check for strict equality

In the previous step, the string '0.67' didn't match the number 0.67 because they weren't *strictly* equal. Strict equality means the data type and value must be the same. The regular equality operator (`==`) in JavaScript doesn't require the data types to match, as the next function shows:

```
  function stepFive() {
    let answer;
    if (100 == '100') {
      answer = 'Yes';
    } else {
      answer = 'No';
    }
    return answer;
  }
```

The test is failing because the value of `answer` being returned is 'Yes'. That's because the number 100 and the string '100' can be converted to equal values, which is what `==` checks for. Since this is often a source of bugs, it's preferable to use the strict equality operator `===` unless you have a good reason not to. Change the `==` to `===`, and the test passes.

## Step Six: Iterate through an array

The next function includes an array of numbers named `amounts` and a variable named `sum`:

```
  function stepSix() {
    const amounts = [10, 20, 30, 40];
    let sum = 0;
    return sum;
  }
```

The test is failing because it expects `sum` to be the sum of the values in `amounts`. One way to add those up is with a `for` loop. You could add a `for` loop like this:

```
function stepSix() {
  const amounts = [10, 20, 30, 40];
  let sum = 0;
  for (let i = 0; i < amounts.length; i++) {
    sum += amounts[i];
  }
  return sum;
}
```

That gets the test to pass, but there's no need for the value `i` here, so it's preferable to use a `for..of` loop:

```
function stepSix() {
  const amounts = [10, 20, 30, 40];
  let sum = 0;
  for (const amount of amounts) {
    sum += amount;
  }
  return sum;
}
```

Notice that in the `for` loop, you declare `i` using `let` because after it's declared at the beginning of the loop, its value is reassigned for each iteration.

In the `for..of` loop, you use `const` because the `amount` variable is re-declared for each iteration, and there's no reassignment.

## Step Seven: Add a property to an object

The last function creates an object to represent an ice cream cone:

```
function stepSeven() {
  const iceCreamCone = {
    flavor: 'strawberry',
    coneType: 'waffle'
  }

  return iceCreamCone;
}
```

In JavaScript, you can create an object by enclosing a list of key-value pairs in curly brackets. The keys are the properties of the object. So in this example, the object `iceCreamCone` refers to has the properties `flavor` and `coneType`.

One reason the test is currently failing is because it's looking for a property named `numberOfScoops` with the value 2. Add that property to the object:

```javascript
function stepSeven() {
  const iceCreamCone = {
    flavor: 'strawberry',
    coneType: 'waffle',
    numberOfScoops: 2
  }

  return iceCreamCone;
}
```

To get the final test passing, you also need to add a `hasSprinkles` property with the value `true`. You can add properties to an object in JavaScript after it's created.

To do that, you assign a value to the property. Add the `hasSprinkles` property to `iceCreamCone`:

```javascript
function stepSeven() {
  const iceCreamCone = {
    flavor: 'strawberry',
    coneType: 'waffle',
    numberOfScoops: 2
  }

  iceCreamCone.hasSprinkles = true;
  return iceCreamCone;
}
```

And now, all the tests pass.

## Next steps

Experimenting with the basics of JavaScript can be complicated because of the configuration needed to get JavaScript running in your browser. A more direct method for trying out JavaScript is using the JavaScript console in your browser.

Open the developer tools (by pressing F12) and in the "Console" tab, type a JavaScript expression to have it immediately evaluated. Try declaring some variables, creating arrays and objects, and using the various methods described in the reading for manipulating strings and arrays.