

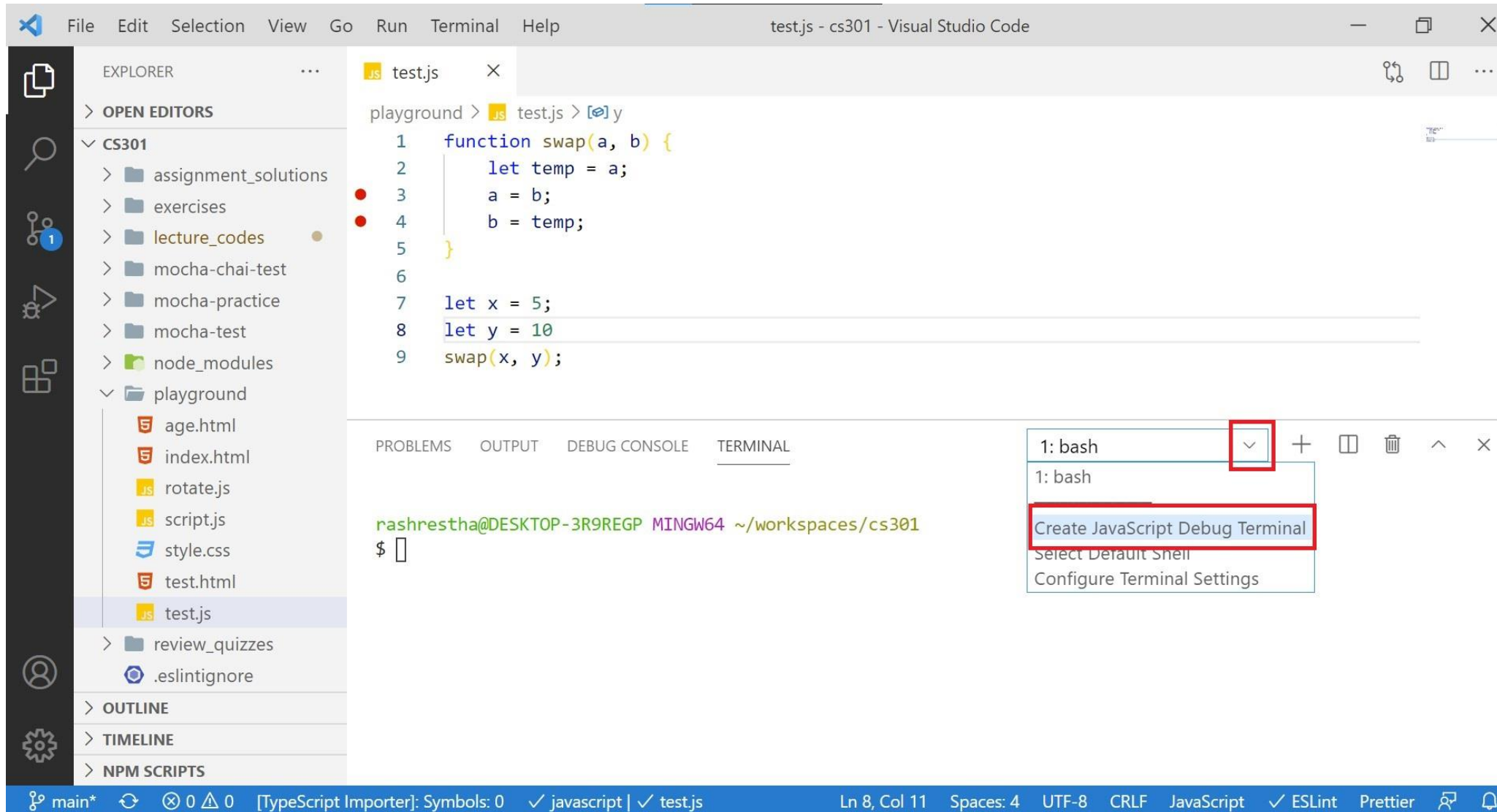
LESSON CODE QUALITY

Source of Orderliness and Coherence

Slides based on material from <https://javascript.info> licensed as [CC BY-NC-SA](#).
As per the CC BY-NC-SA licensing terms, these slides are under the same license.

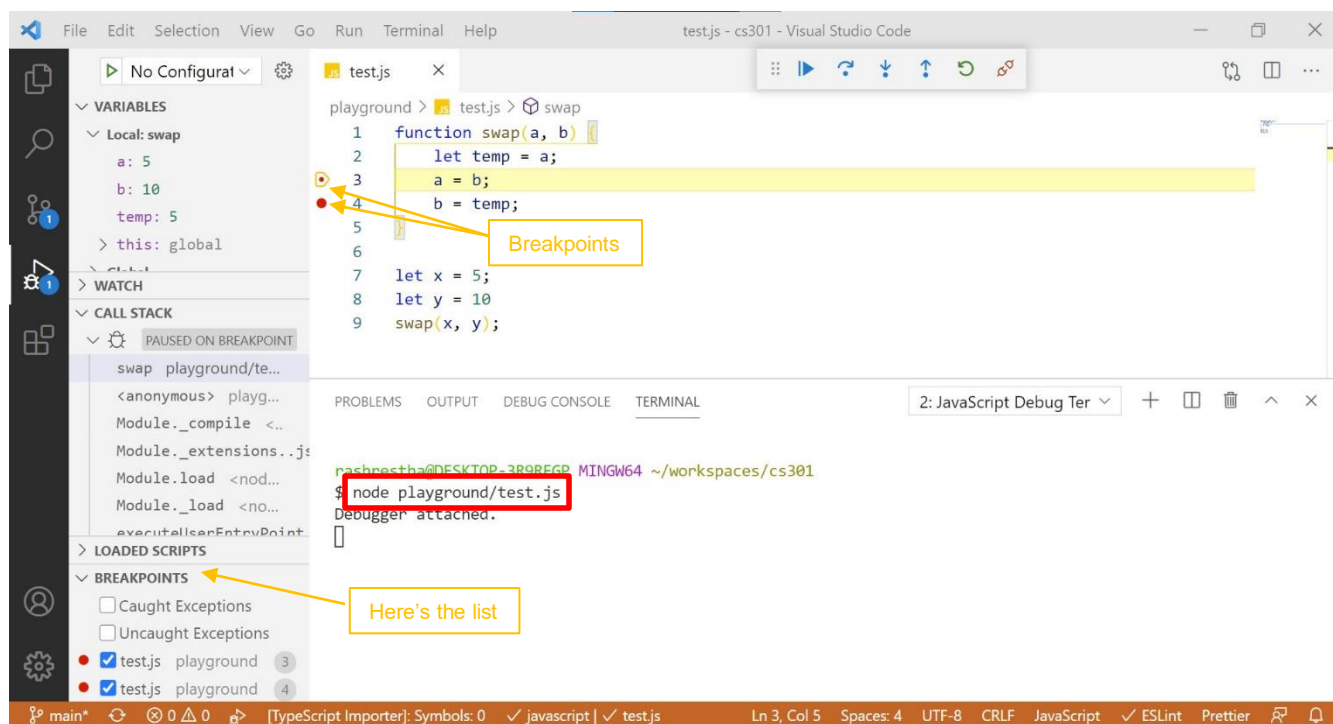
Wholeness: The best quality code is easy to understand and maintain. Science of Consciousness: Pure consciousness is a state of perfect orderliness and simplicity. Having this experience promotes the ability to find the simplest and most orderly solutions to problems.

Debugging in Node



Breakpoints

- list of breakpoints in the left panel. It allows us to:
 - Quickly jump to the breakpoint in the code by clicking in the breakpoints list.
 - Temporarily disable the breakpoint by unchecking it.
 - Remove the breakpoint by right-clicking and selecting Remove.



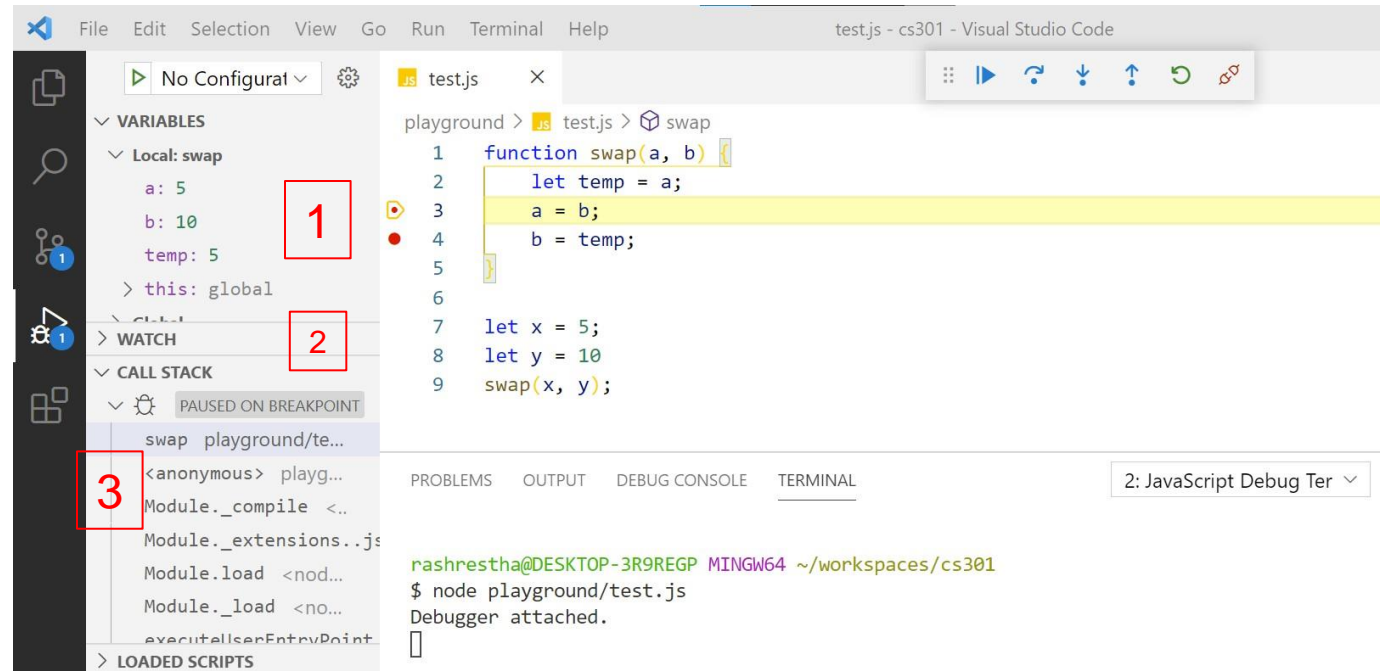
Debugger command

- pause the code by using the debugger command in it
 - Helpful in sandbox sites like jsfiddle or plunker
 - and when have infinite loop in code
 - in browser code runs automatically upon loading
 - like a breakpoint, but not dependent on the visual interface

```
function hello(name) {  
  let phrase = `Hello, ${name}!`;   
  
  debugger; // <-- the debugger stops here  
  
  say(phrase);  
}
```

Pause and look around

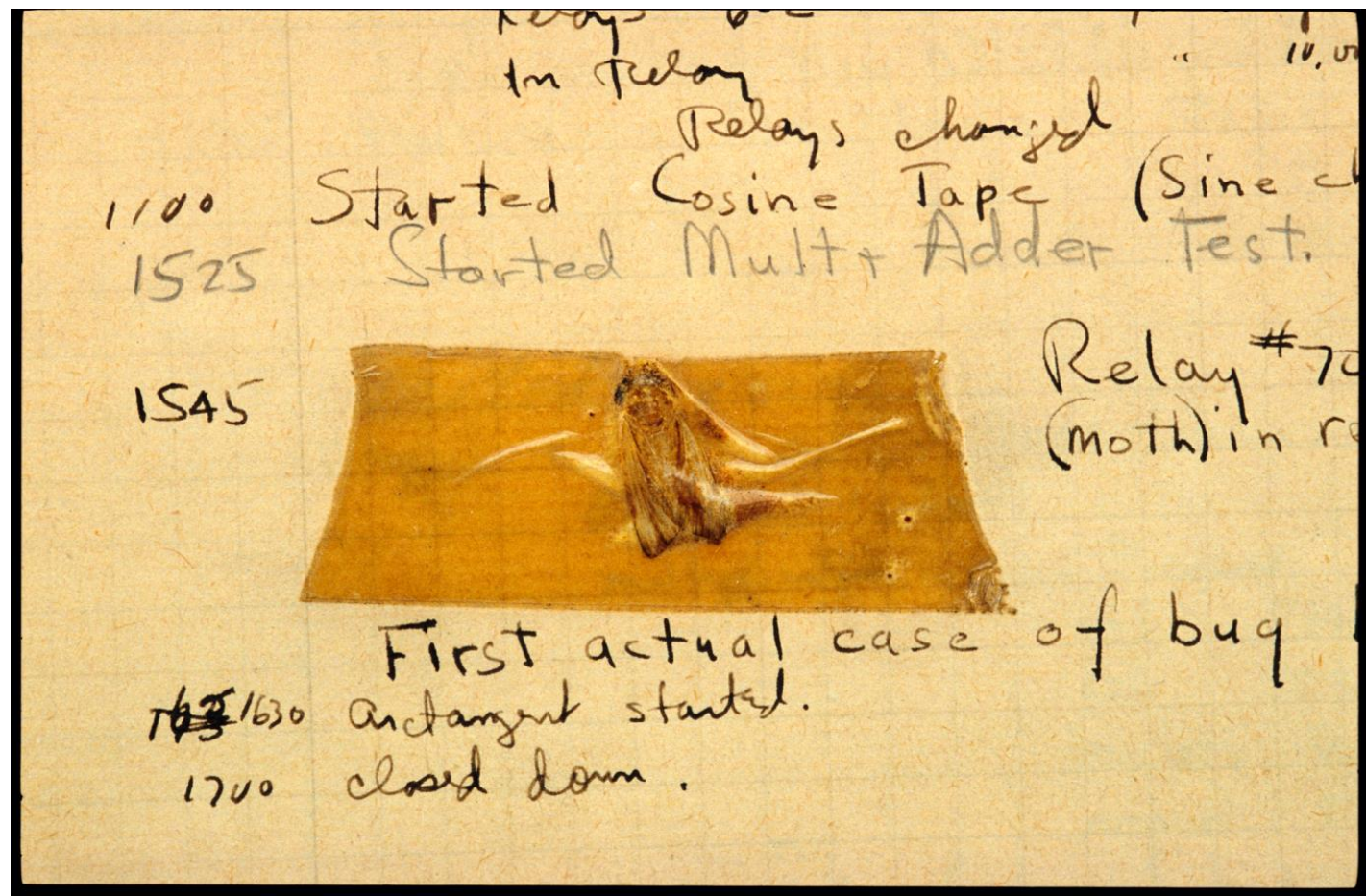
- In our example, as the breakpoints are set at line 3 and 4, the execution pauses at the first breakpoint on 3rd line:
 - VARIABLES – current variables in different scopes.
 - WATCH – shows current values for any expressions (if any).
 - CALL STACK – shows the nested calls chain



Tracing the execution

- Continue the execution, F5.
- Step Over (run the next command), but don't go into the function, F10.
- Step Into, F11.
- Step Out, Shift + F11
- Restart, Ctrl + shift + F5
- Disconnect

the First Computer "Bug"



- first computer bug' in the Harvard Mark II computer's log book. The problem was traced to a moth stuck between relay contacts in the computer

Main Point: Using the Node or Chrome debugger

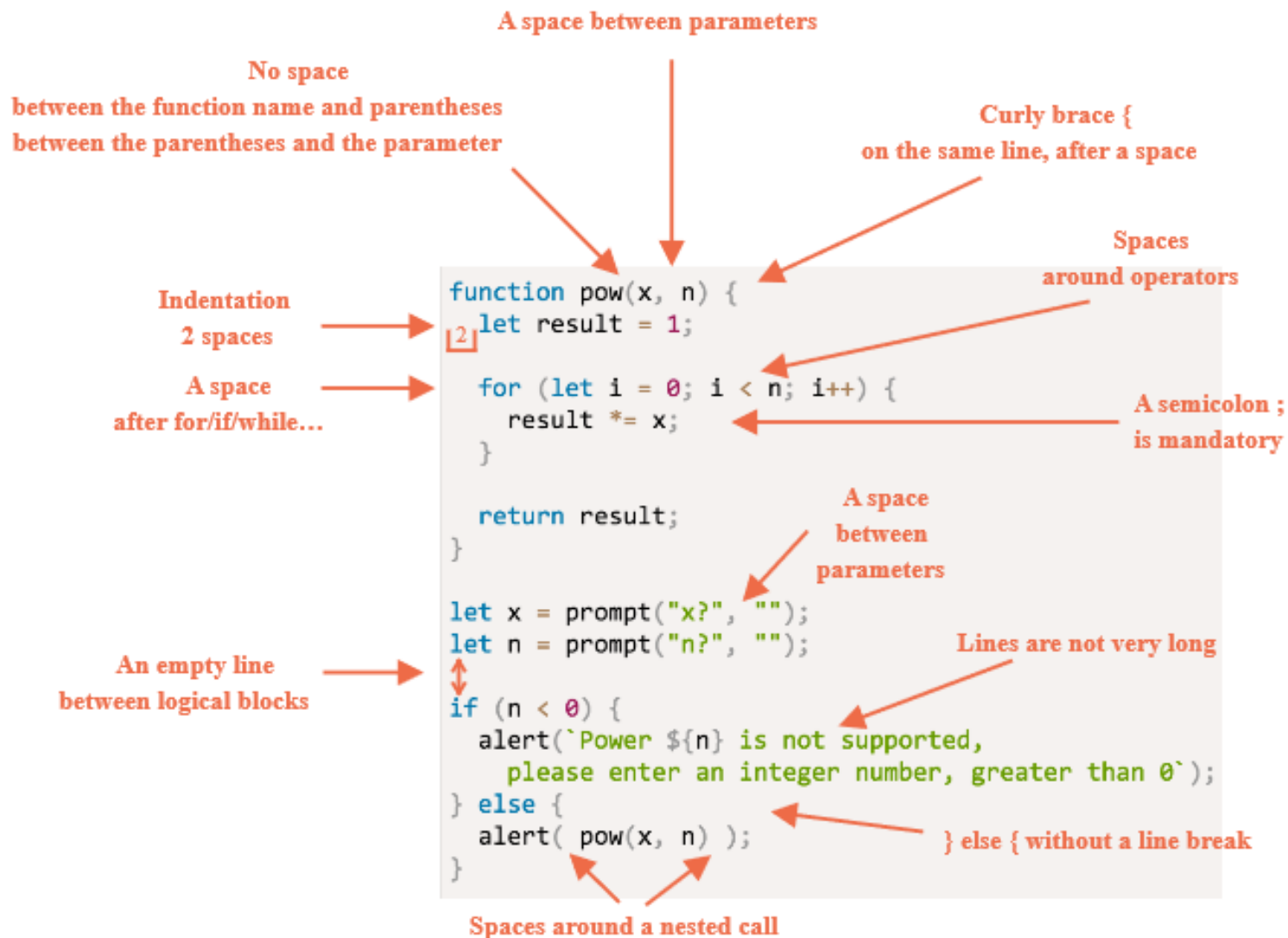
One should carefully develop code to avoid bugs, but in the real world some mistakes are inevitable even for the best developers. Proper use of debugging tools greatly reduces the time required to remove errors. Science of Consciousness: Ideally one can avoid stress and strain in daily life, but in the real world some daily stress is inevitable. The TM Technique is an effective and efficient manner to remove stress.

Coding conventions

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible you are, by definition, not smart enough to debug it.”

Kernighan and Plauger, *The Elements of Programming Style*

Good code looks good.



Coding conventions

➤ Curly Braces

- In most JavaScript projects curly braces are written with the opening brace on the same line as the corresponding keyword – not on new line.

```
if (n < 0) {  
    alert(`Power ${n} is not supported`);  
}
```

➤ Line Length

- No one likes to read a long horizontal line of code. It's best practice to split them.
- Horizontal indents: 2 or 4 spaces.
- Vertical indents: empty lines for splitting code into logical blocks
- A semicolon should be present after each statement, even if it could possibly be skipped
 - not needed at the end of a code block `{ ... } //no semicolon needed here`
- VS Code "Format Document": SHIFT+ALT+F in windows (or right click context menu in file)

Function Placement

- Code first, then functions
 - when reading code, we first want to know what it does.
 - If the code goes first, then it becomes clear from the start.
 - Then, maybe we won't need to read the functions, especially if their names are descriptive

```
// the code which uses the functions
let elem = createElement();
setHandler(elem);
walkAround();
```

```
// --- helper functions ---
function createElement() {
  ...
}
```

```
function setHandler(elem) {
  ...
}
```

```
function walkAround() {
  ...
}
```

Comments

```
function showPrimes(n) {  
  nextPrime:  
  for (let i = 2; i < n; i++) {  
  
    // check if i is a prime number  
    for (let j = 2; j < i; j++) {  
      if (i % j == 0) continue nextPrime;  
    }  
  
    alert(i);  
  }  
}
```



refactor

```
function showPrimes(n) {  
  
  for (let i = 2; i < n; i++) {  
    if (isPrime(i)) {  
      alert(i);  
    }  
  }  
}  
  
function isPrime(n) {  
  for (let i = 2; i < n; i++) {  
    if (n % i == 0) return false;  
  }  
  
  return true;  
}
```

Good comments

- An important sign of a good developer is comments:
- Good comments help maintain code,
 - come back after a delay and use effectively
- Comment this:
 - Overall architecture, high-level view.
 - Function usage.
 - Parameters, return values, exceptions, side effects
 - process (if nontrivial, effectively a defining table)
- Important solutions, especially when not immediately obvious
 - If you worked out some complex problem and implemented solution
 - Highlight the solution with a succinct comment so others quickly understand

“Think like a programmer”

- Computers have no inherent intelligence
- Do exactly and only what they are told to do
- Intelligence comes from the programmer
- Use comments to design your solution to the problem
 - This is the important step
 - Developer .. Engineer ... architect
- Then write code to explain your comments to the computer
 - Relatively simple step
 - coder

Automated Linters

- Linters are tools that automatically check the style of your code and make improving suggestions.
- can also find some bugs, like typos in variable or function names.
- using a linter is recommended
- some well-known linting tools:

JSLint – one of the first linters.

JSHint – more settings than JSLint.

ESLint – newest and most widely used (see instructions for installing eslint in Sakai Resources)

- Instructions to install ESLint in VSCode are in Sakai > Resources > lab helpers > esLintSetupInstructions.pdf

JS doc

- common development problem:
 - you have written JavaScript code that is to be used by others and need a nice-looking HTML documentation of its API.
 - standard tool in the JavaScript world is JSDoc.
 - It is modeled after its Java analog, JavaDoc.
- JSDoc takes JavaScript code with `/** */` comments (normal block comments that start with an asterisk) and produces HTML documentation for it
- For functions and methods, you should document parameters and return values,
 - and exceptions they may throw:

```
/**
 *
 * @param {number} x The number to raise.
 * @param {number} n The power, must be a natural number.
 * @return {number} x raised to the n-th power.
 */
function pow(x, n) {
  ...
}
```

Install JS doc

- At terminal prompt:
 - npm install -g jsdoc
 - jsdoc pow.js
- Will then find pow.js.html under an out directory

```
/**
 * Returns x raised to the n-th power.
 *
 * @param {number} x The number to raise.
 * @param {number} n The power, must be a natural number.
 * @return {number} x raised to the n-th power.
 */
function pow(x, n) {
  ...
}
```

127.0.0.1:5500/out/global.html#pow

Methods

`pow(x, n) → {number}`

Returns x raised to the n-th power.

Parameters:

Name	Type	Description
x	number	The number to raise.
n	number	The power, must be a natural number.

Source: [pow.js, line 8](#)

Returns:

x raised to the n-th power.

Type
number

Documentation generated by [JSDoc 3.6.3](#) on Fri Sep 06 2019 22:36:21 GMT-0500 (Central Daylight Time)

Main Point: Coding conventions

Coding conventions are standardized best practices that ensure that code is easy to read and maintain. Science of Consciousness: The TM Technique is taught in a standard manner to ensure that the knowledge is understood and maintained over time and across individuals.

Main Point Preview: Behavior Driven Development (BDD)

In Behavior Driven Development we write tests that define what a function is supposed to do before we implement the function. Science of Consciousness: This provides a goal to guide development and a means to test success. This is like the SCI principle of Capture the Fort or Seek the Highest First. Once you have defined the required outcome the rest of the implementation flows from that.

Automated testing and Behavior Driven Development (BDD)

- Automated testing means tests are written separately, in addition to the code.
 - run functions and compare actual results with expected.
 - BDD is automated testing where tests are written before the code is implemented
- BDD is three things in one:
 - tests
 - documentation
 - examples

BDD Specification

➤ imagine what the function should do and describe it.

➤ **describe**("title", function() { ... })

- functionality we're describing.
- Used to group "workers" – the it blocks.

➤ **it**("use case description", function() { ... })

- In the title of it describe the particular use case
- second argument is a function that tests it.

➤ **assert.equal**(actual, expected)

- first argument is call to function under test
- second argument the result you expect

```
describe("pow", function() {  
  it("returns 2 raised to power 3", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
});
```

Development flow

1. An **initial spec is written, with tests** for the most basic functionality.
2. An initial implementation (stub) is created.
3. To check whether it works, we run the testing framework Mocha (more details soon) that runs the spec. While the functionality is not complete, errors are displayed. We make corrections until everything works.
4. Now we have a working initial implementation with tests.
5. **We add more use cases to the spec, probably not yet supported by the implementations. Tests start to fail.**
6. Go to 3, update the implementation till tests give no errors.
7. Repeat steps 3-6 till the functionality is ready.

JavaScript libraries for tests

- Mocha – the core framework: it provides common testing functions including `describe` and `it` and the main function that runs tests.

Additional libraries commonly used with Mocha but beyond scope of 305

- Chai – the library with many assertions.
 - for now, we need only assert from Mocha
- Sinon – library for stubs, mocks, spies

```
sinon.stub(window, "prompt"); //insert Sinons static version of prompt function  
prompt.onCall(0).returns("2");//first call to prompt will return "2"  
prompt.onCall(1).returns("3");
```

Principle: one test tests one thing

The first variant – add one more assert into the same it:

```
describe("pow", function() {  
  
  it("raises to n-th power", function() {  
    assert.equal(pow(2, 3), 8);  
    assert.equal(pow(3, 4), 81);  
  });  
  
});
```



refactor

The second – make two tests:

```
describe("pow", function() {  
  
  it("2 raised to power 3 is 8", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
  
  it("3 raised to power 3 is 27", function() {  
    assert.equal(pow(3, 3), 27);  
  });  
  
});
```

In BDD, the spec comes first

The spec can be used in three ways:

1. As Tests – guarantee code works
2. As Docs – titles of describe and it tell what the function does.
3. As Examples – tests are working examples how a function can be used.

- With the spec, we can safely improve, change, even rewrite the function from scratch and make sure it still works right.
- Without tests, developers have two options:
 - perform the change, no matter what.
 - then users meet bugs, as we probably fail to check something manually.
 - developers become afraid to modify such functions,
 - code becomes outdated, no one wants to get into it
- [What's wrong in the test? \(javascript.info\)](http://javascript.info)

how to run Mocha in VSCode

- Install Mocha Globally
 - `$npm install -g mocha`
- Create a project
 - Create a project directory `mocha-test` (can be any name)
 - Write your first test in file `myCodeTests.js` (can be any name)
 - Write your code in separate JavaScript file `myCode.js`.
 - Connect your test.js file to the functions you write in app.js using Node's CommonJS modules
 - See following slides for demo
 - run `mocha myCodeTests.js`
- If you encounter an error message when installing or running Mocha "... ps1 cannot be loaded because running scripts is disabled ... ", then look at the file, [Set-ExecutionPolicy Bug fix.txt](#), in Sakai > Resources > lab helpers

```
//basic example of using mocha (test code in same file)

"use strict";
const assert = require("assert");

function checkPrime(num) {
  for (let i = 2; i < num; i++)
    if (num % i === 0) return false;
  return num > 1;
}

describe("checkPrime returns true if argument is prime", function () {

  it("37 is prime", function () {
    assert.strictEqual(checkPrime(37), true);
  });
  it(" 77 is not prime", function () {
    assert.strictEqual(checkPrime(77), false);
  });
});
```

```
//suppose have many functions to test and want tests in separate file
"use strict";
const assert = require("assert");

/* import from functions.js module */
const functionsModule = require("../functions.js");
const checkPrime = functionsModule.checkPrime;
const ssReverse = functionsModule.ssReverse;

describe("checkPrime returns true if argument is prime ", function () {
  it("37 is prime", function () {
    assert.strictEqual(checkPrime(37), true);
  });
  it(" 77 is not prime", function () {
    assert.strictEqual(checkPrime(77), false);
  });
});

describe("ssReverse", function () {
  it("[1, 2, 3, 4]", function () { //test array equality
    assert.deepStrictEqual(ssReverse([1, 2, 3, 4 ]), [4,3,2,1] );
  });
});
```

```
//functions.js
"use strict";
module.exports = {checkPrime, ssReverse };

/**
 * @param {number} num is an integer
 * @returns {boolean} true if number is prime, else false
 * Prime numbers have only 2 factors: 1 and themselves.
 */
function checkPrime(num) {
    for (let i = 2; i < num; i++)
        if (num % i === 0) return false;
    return num > 1;
}

function ssReverse(){ .. . }
```

Main Point: Behavior Driven Development (BDD)

In Behavior Driven Development we write tests that define what a function is supposed to do before we implement the function. Science of Consciousness: This provides a goal to guide development and a means to test success. This is like the SCI principle of Capture the Fort or Seek the Highest First. Once you have defined the required outcome the rest of the implementation flows from that.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Source of Orderliness and Coherence

1. Coding conventions provide guidelines to write code without bugs. Debuggers are critical for efficiently finding bugs that do occur.
2. Automated testing and Behavior Driven Development help guide development to prevent bugs during development and provide a means to ensure everything continues to work when there are extensions and modifications.

-
3. **Transcendental consciousness.** State of perfect order and coherence
 4. **Impulses within the transcendental field:** Thoughts arising from this level will naturally be orderly and coherent.
 5. **Wholeness moving within itself:** In unity consciousness everything is appreciated in terms of this experience of order and coherence.

