LESSON 2
CODE QUALITY

Source of Orderliness and Coherence

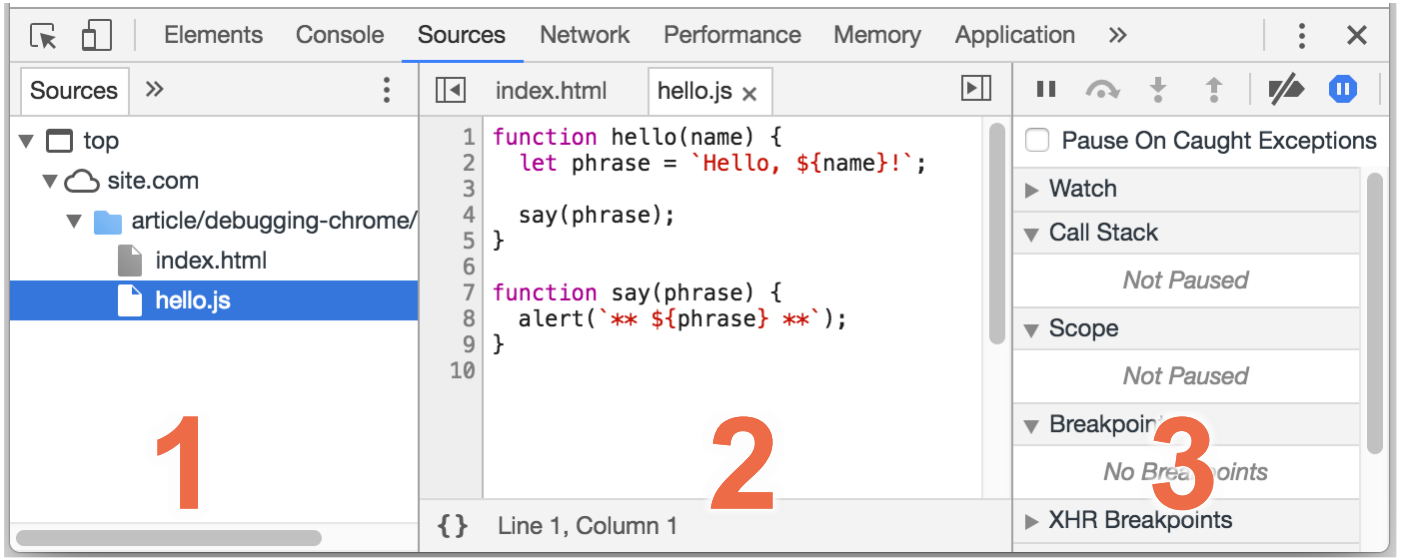
Slides based on material from https://javascript.info licensed as CC BY-NC-SA.
As per the CC BY-NC-SA licensing terms, these slides are under the same license.

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.

Main Point Preview: Using the 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.

Debugging in Chrome



- Open this example page

1. The **Resources zone** lists HTML, JavaScript, CSS and other files
2. The **Source zone** shows the source code
3. The **Information and control zone** is for debugging

Breakpoints

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

Debugger command

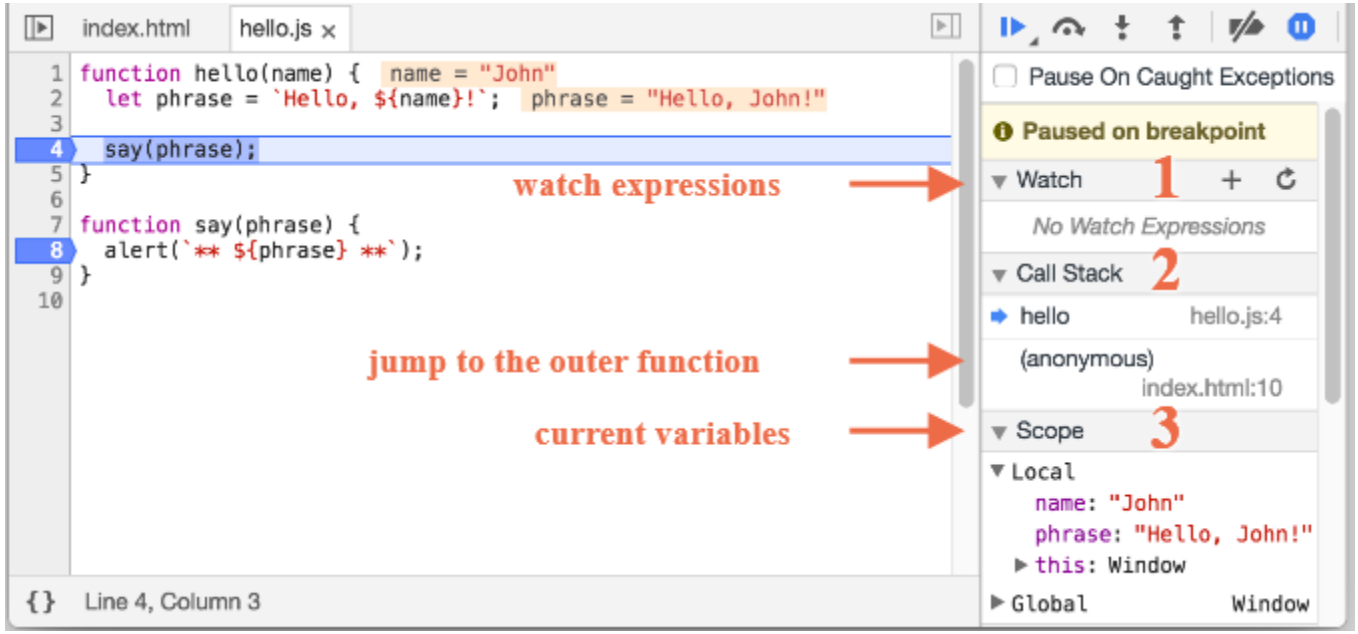
pause the code by using the debugger command in it

```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  debugger;  // <-- the debugger stops here

  say(phrase);
}
```

convenient when we are in a code editor and don’t want to switch to the browser and look up the script in developer tools to set the breakpoint

Pause and look around

- In our example, hello() is called during the page load,
  - easiest way to activate the debugger (after we've set the breakpoints) is to reload the page.
  - F5 (Windows, Linux) or Cmd+R (Mac).
  - As the breakpoint is set, the execution pauses at the 4th line:
  - Watch – shows current values for any expressions.
  - Call Stack – shows the nested calls chain
  - Scope – current variables.



Tracing the execution

- continue the execution, F8.
- make a step (run the next command), but don’t go into the function, F10.
- step in, F11.
- step out, Shift F11
- enable/disable all breakpoints.
  - That button does not move the execution. Just a mass on/off for breakpoints.

Main Point: Using the Chrome debugger

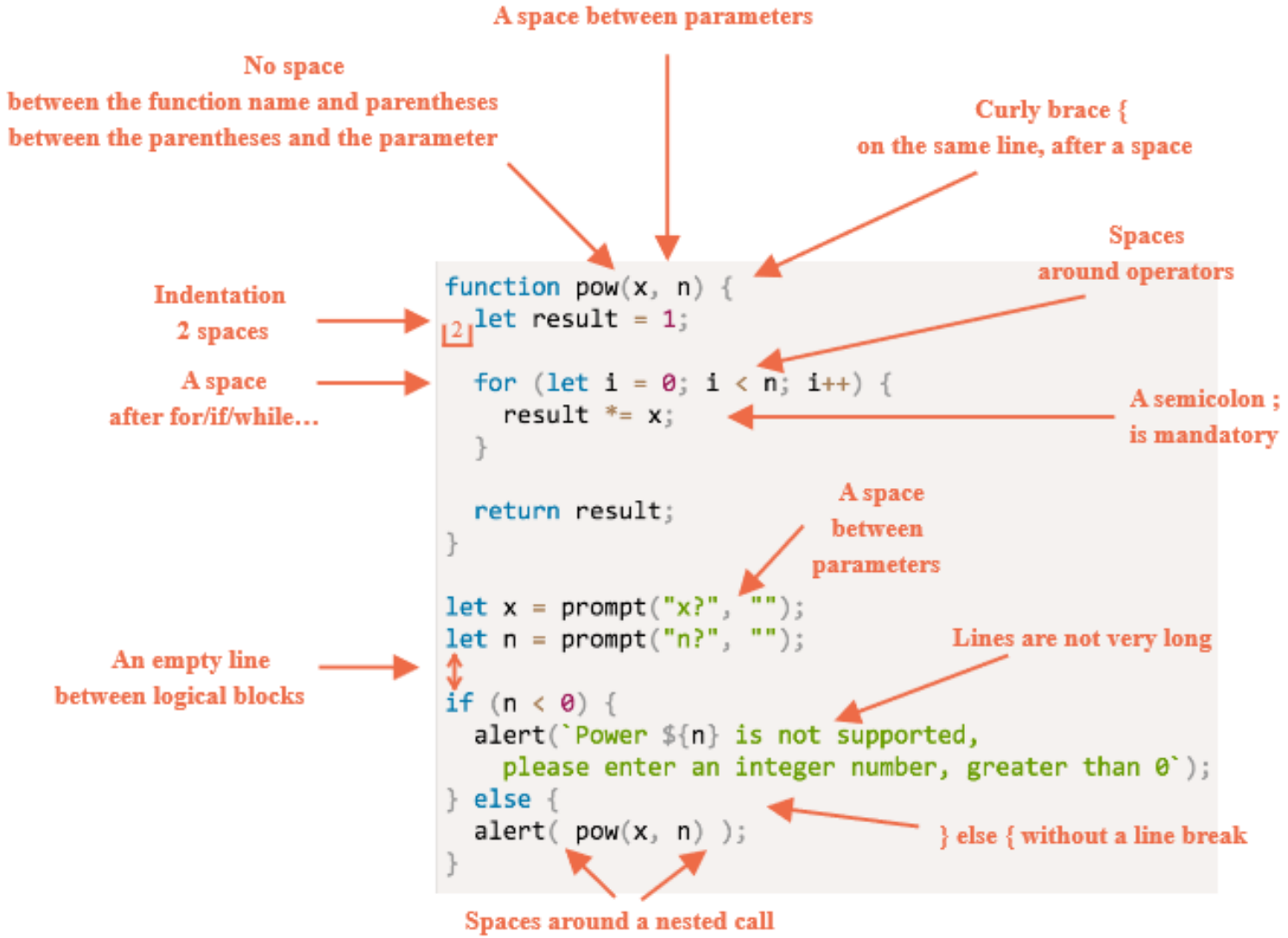
One should carefully develop code to avoid bugs, but in the real world some mistakes are inevitable even for the best developers. Proper use of debugging tools greatly reduces the time required to remove errors. Science of Consciousness: Ideally one can avoid stress and strain in daily life, but in the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

Main Point Preview: 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.

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*



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

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() {
  ...
}
```

Automated Linters

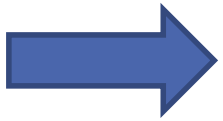
- Linters are tools that automatically check the style of your code and make improving suggestions.
- can also find some bugs, like typos in variable or function names.
- using a linter is recommended
- some well-known linting tools:

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)

- Code style exercise: **Bad style**

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)) continue;

    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

  - 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

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

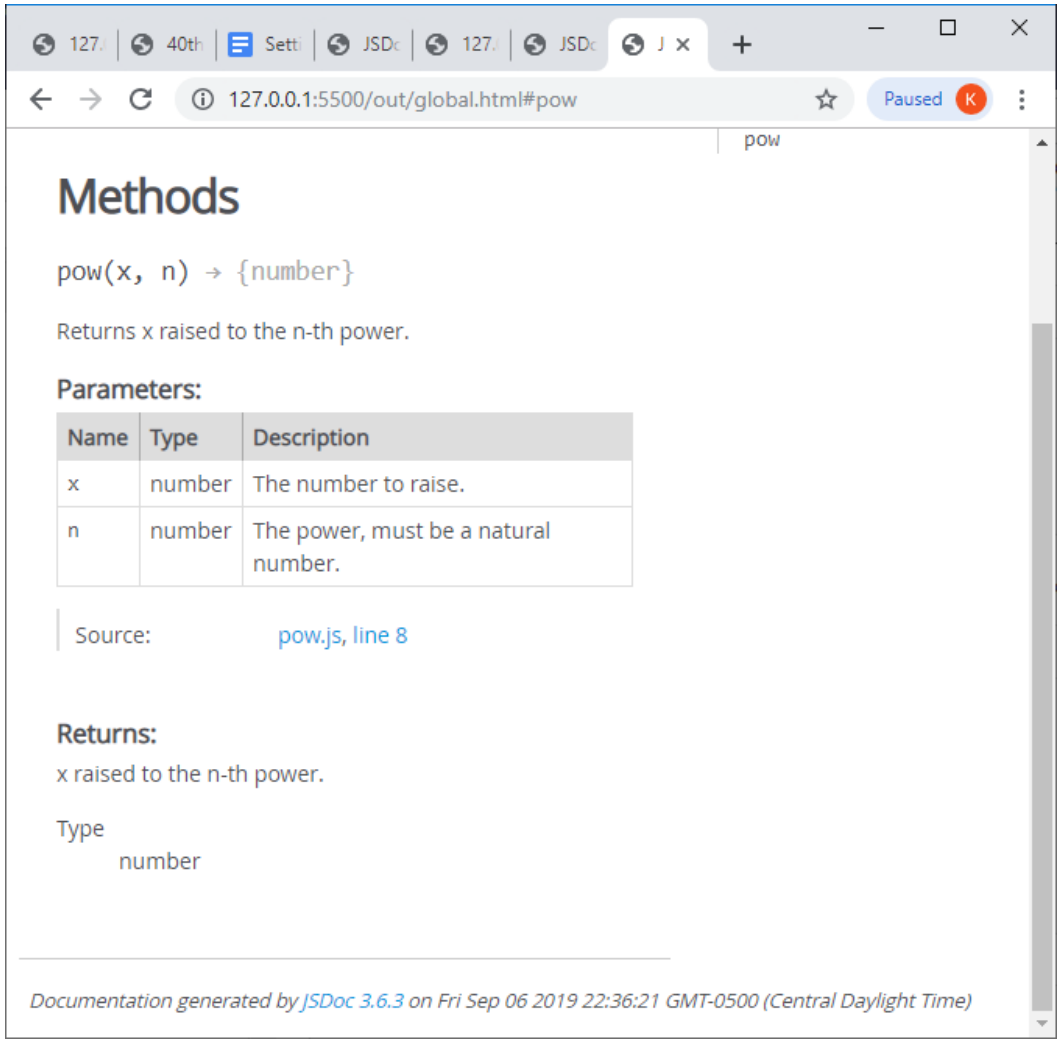
- For functions and methods, you can document parameters, return values, and exceptions they may throw:

```
/**
 * 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) {
  ...
}
```

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) {
  ...
}
```



Exercise: Ninja code

- Read this section on your own
- Write the real rules implied by the irony examples
- Submit this as part of today's homework

For example:

- Ninja irony: Make the code as short as possible. Show how smart you are
  - Meaning: do not sacrifice code clarity for brevity.

- use single-letter variable names everywhere.
  - Meaning: *your answer here*

- If the team rules forbid the use of one-letter and vague names – shorten them, make abbreviations
  - Meaning: *your answer here*

- While choosing a name try to use the most abstract word
  - Meaning: *your answer here*

- Etc etc

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.

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

- **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(value1, value2)
  - if the implementation is correct, should execute code inside it block without errors.

Development flow

1. An initial spec is written, with tests for the most basic functionality.
2. An initial implementation 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.

- Chai – the library with many assertions.
  - for now we need only assert.equal.

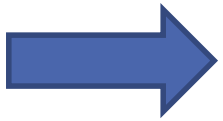
- Sinon – library for stubs, mocks, spies

```
sinon.stub(window, "prompt");
prompt.onCall(0).returns("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);
  });

});
```

Extending the spec

- The basic functionality of pow is complete.
- function pow(x, n) is meant to work with positive integer values n.
  - To indicate a mathematical error, JavaScript functions usually return NaN. Let's do the same for invalid values of n.
- Let's first add the behavior to the spec(!):
- The newly added tests fail, because our implementation does not support them.
  - first we write failing tests,
  - then make an implementation for them

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

  // ...

  it("for negative n the result is NaN",
function() {
    assert.isNaN(pow(2, -1));
  });

  it("for non-integer n the result is NaN",
function() {
    assert.isNaN(pow(2, 1.5));
  });

});
```

In BDD, the spec comes first

The spec can be used in three ways:

1. As Tests – they guarantee that the code works correctly.
2. As Docs – the titles of describe and it tell what the function does.
3. As Examples – the tests are working examples showing 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

- Exercise: What's wrong in the test?

how to run Mocha in VSCode

- Copy the mocha.html page from Sakai Resources to your folder
  - Use this as a template for unit testing in your 303 projects
  - Rename it to something meaningful for the application or exercise you are working on
    - E.g., lab2Tests.html
- Write your own test file and replace the test2.js file with it
  - BDD – write this first!
- Write your own JavaScript file of your code instead of pow.js
- Open your html page
  - E.g., using Live Server

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.