Async Quiz

```
function boo() {
    console.log('boop!');
}

console.log('fizz');
setTimeout(boo, 1000);
console.log('buzz');
```

In the code above, what order will the messages be printed in?

- [✓ fizz, buzz, boop!]
- fizz, boop!, buzz
- boop!, buzz, fizz
- boop!, fizz, buzz

EXPLANATION

`setTimeout` does not block execution so 'buzz' will be printed before `boo` is called.

```
function boo() {
    console.log('boop!');
}

console.log('fizz');
setTimeout(boo, 0);
console.log('buzz');
```

In the code above, what order will the messages be printed in?

- [✓ fizz, buzz, boop!]
- fizz, boop!, buzz
- boop!, buzz, fizz
- boop!, fizz, buzz

EXPLANATION

`setTimeout` does not block execution even if a delay time of 0 is provided.

```
function far() {
    console.log('farm!')
}

function boo() {
    console.log('boop!');
    far();
}

console.log('fizz');
setTimeout(boo, 1000);
console.log('buzz');
```

In the code above, what order will the messages be printed in?

- [✓ fizz, buzz, boop!, farm!]
- boop!, buzz, fizz, farm!
- farm!, boop!, fizz, buzz
- fizz, buzz, farm!, boop!

EXPLANATION

`far`is called synchronously inside of `boo`, so 'farm!' will be printed right after 'boop!'

```
function far() {
    console.log('farm!')
}

function boo() {
    console.log('boop!');
    setTimeout(far, 1000);
    console.log('boop!');
}

setTimeout(boo, 1000);
console.log('buzz');
```

In the code above, what order will the messages be printed in?

- farm!, boop!, boop!, buzz
- [✓ buzz, boop!, boop!, farm!]
- boop!, farm!, boop!, buzz
- buzz, boop!, farm!, boop!

EXPLANATION

Since `far`is called asynchronously, it will not block execution of the second 'boop!'

```
function asyncy(cb) {
  setTimeout(cb, 1000);
  console.log("async");
}
```

```
function greet() {
  console.log("hello!");
}

asyncy(greet);
```

In the code above, what order will the messages be printed in?

[✓ async, hello!]

hello!, async

EXPLANATION

`setTimeout` will not block execution of lines that come after it

Identifying the Base &amp; Recursive Case Quiz

```
justDance(song) {
    justDance(song);
}

justDance("I Wanna Dance With Somebody (Who Loves Me)");
```

Which of the following errors will result from running the above function?

- ◯ `ReferenceError: song is not defined`
- ◯ `404: File not found`
- ◯ `` `RangeError: Maximum call stack size exceeded` ``
- ◯ `ENOENT: No such file or directory`

EXPLANATION

Because we're missing a base case, this function will recurse infinitely and cause a stack overflow. We expect a `RangeError` from this.

```
exercise(bottle) {
    console.log("Just a few more reps!");)
    drinkWater(bottle);
}

drinkWater(bottle) {
    if (bottle.water > 0) {
        exercise({ water: bottle.water - 1 });
```

```
      } else {
          console.log("Whew! Good workout.");
          return;
      }
  }

  exercise({ water: 5 });
```

For the recursive function above, what is the recursive step?

- ◯ `bottle.water - 1`
- ◯ `exercise(bottle)`
- ◯ `bottle.water === 0`
- ◯ `bottle.water > 0`

EXPLANATION

The *recursive step*should move us closer to the *base case*(here, `bottle.water === 0`). Decrementing the value of `bottle.water`does this. Careful not to confuse this with the *recursive case*, which is the input values that cause the function to recurse.

```
  justDance(song) {
      justDance(song);
  }

  justDance("I Wanna Dance With Somebody (Who Loves Me)");
```

Which of the following should we add to prevent an error from the above function? You should choose all answers that are appropriate.

- ☐ A parameter.
- ☐ A base case

[✓ A recursive step]

A recursive case

EXPLANATION

This function already has a recursive case, but it has no way of terminating nor anything helping it work towards that termination! While the function also has a parameter, it's not particularly helpful at the moment.

```
echo(message, volume) {
    if (volume === 0) {
        return;
    }

    console.log(message);
    echo(message, volume - 1);
}

echo("Hello there!", 10);
```

For the recursive function above, select the correct Base & Recursive Cases. There will be one of each type.

Recursive: `volume === 10`

[✓ Base: `volume === 0`]

Base: `volume - 1`

[✓ Recursive: `volume > 0`]

EXPLANATION

`echo()` will recurse as long as `volume > 0`, and will terminate as soon as `volume === 0`. Don't get the *recursive case* (here, when `volume` is greater than 0) confused with the *recursive step* (here, `volume - 1`)!

```
exercise(bottle) {
    console.log("Just a few more reps!");)
    drinkWater(bottle);
}

drinkWater(bottle) {
    if (bottle.water > 0) {
        exercise({ water: bottle.water - 1 });
    } else {
        console.log("Whew! Good workout.");
        return;
    }
}

exercise({ water: 5 });
```

For the recursive function above, select the correct Base & Recursive Cases. There will be one of each type.

- ☐ Base: `bottle.water > 0`
- ☐ Recursive: `drinkWater(bottle)`
- ☐ Recursive: `bottle.water > 0
- ☐ Base: `bottle.water === 0`

EXPLANATION

This indirectly recursive pair of functions will repeat until `bottle.water === 0`, at which point `drinkWater()` will `return`. Therefore, the recursive case is `bottle.water > 0`.

Callbacks Quiz Recall

```javascript
let foo = function(n, cb) {
  console.log("vroom");
  for (let i = 0; i < n; i++) {
    cb();
  }
  console.log("skrrt");
};

foo(2, function() {
  console.log("swoosh");
});
```

In what order will the code above print out?

- vroom, swoosh, skrrt, swoosh, skrrt
- swoosh, vroom, skrrt
- vroom, swoosh, swoosh, swoosh, skrrt
- [✓ vroom, swoosh, swoosh, skrrt]

EXPLANATION

Since the loop iterates twice, 'swoosh' will print twice between 'vroom' and 'skrrt'.

```javascript
let foo = function() {
  console.log("Everglades");
  console.log("Sequoia");
};

console.log("Zion");
```

```
  foo();
  console.log("Acadia");
```

In what order will the code above print out?

- [✓ Zion, Everglades, Sequoia, Acadia]
- Zion, Everglades, Acadia, Sequoia
- Everglades, Sequoia, Zion, Acadia
- Everglades, Zion, Acadia, Sequoia

EXPLANATION

The prints that belong to `foo` will be executed only when it is called after 'Zion', but before 'Acadia'.

Are functions considered first class objects in JavaScript?

- no
- [✓ yes]

EXPLANATION

Functions are first class objects in JavaScript, because they can be assigned, passed as an argument, and returned.

```
let foo = function() {
  console.log("hello");
  return 42;
};

foo;
```

When executed in node, what will the code snippet above print out?

- `[Function: foo]`
- `42`
- [✓ It will print nothing]
- `hello`

EXPLANATION

Nothing will be printed because the only `console.log` is within the `foo` function, but `foo()` is never called.

Which of the following is not required to be a first class object?

- ability to be assigned to a variable
- [✓ ability to be mutated]
- ability to be a return value of a function
- ability to be an argument to a function

EXPLANATION

A first class object does not need to mutable. For example, strings are immutable but still first class because they can be assigned, passed as an argument, and returned.

```
let foo = function() {
  console.log("hello");
  return 42;
};

console.log(foo);
```

When executed in node, what will the code snippet above print out?

- ◯ `hello`
- ◯ It will print nothing
- ◯ `42`
- [✓ `[Function: foo]`]

EXPLANATION

The `foo()` is not called, instead the `foo` function object itself is printed out.

```
let bar = function(s) {
  return s.toLowerCase() + "...";
};

let foo = function(message, cb1, cb2) {
  console.log(cb1(message));
  console.log(cb2(message));
};

foo("Hey Programmers", bar, function(s) {
  return s.toUpperCase() + "!";
});
```

When executed in node, what will the snippet above print out?

- ◯ `[Function]`, `[Function]`
- [✓ `hey programmers...`, `HEY PROGRAMMERS!`]

- ◯ `HEY PROGRAMMERS!`, `hey programmers...`

EXPLANATION

Since arguments are passed positionally, `cb1` is `bar` and `cb2` is the anonymous function. Both `cb1` and `cb2` are called and their return values are printed out.

```
let bar = function() {
  console.log("Ramen");
};

let foo = function(cb) {
  console.log("Gazpacho");
  cb();
  console.log("Egusi");
};

console.log("Bisque");
foo(bar);
console.log("Pho");
```

In what order will the code above print out?

- ◯ Bisque, Gazpacho, Egusi, Ramen, Pho
- ◯ Bisque, Pho, Gazpacho, Egusi, Ramen
- ◯ Ramen, Gazpacho, Egusi, Bisque, Pho
- [✓ Bisque, Gazpacho, Ramen, Egusi, Pho]

EXPLANATION

The `bar` function is passed as a callback to `foo`, so the name `cb` refers to `bar` inside of `foo`

```
let bar = function() {
  console.log("Arches");
};

let foo = function() {
  console.log("Everglades");
  bar();
  console.log("Sequoia");
};

console.log("Zion");
foo();
console.log("Acadia");
```

In what order will the code above print out?

○ Arches, Everglades, Sequoia, Zion, Acadia

[✓ Zion, Everglades, Arches, Sequoia, Acadia]

○ Zion, Everglades, Sequoia, Arches, Acadia

○ Zion, Arches, Everglades, Sequoia, Acadia

EXPLANATION

The code inside of functions only execute once the function is called. When a function returns, execution jumps back to the line after where it was called.

```
let bar = function(mystery) {
  mystery("sneaky");
};

let foo = function(secret) {
  console.log(secret);
};
```

```
bar(foo);
```

In the snippet above, which function is acting as a "callback"?

- ◯ `console.log`
- ◯ `bar`
- [✓ `foo`]

EXPLANATION

A callback is a function that is passed as an argument to another function. In this example, `foo`is passed as an argument to `bar`, making `foo`the callback.

```
function foo() {
  console.log("fizz");
}

function bar() {
  console.log("buzz");
}

function boom(cb1, cb2) {
  console.log("zip");
  cb1();
  console.log("zap");
  cb2();
  console.log("zoop");
}

boom(bar, foo);
```

In what order will the code above print out?

- zip, zap, zoop, buzz, fizz
- [✓ zip, buzz, zap, fizz, zoop]
- zip, fizz, zap, buzz, zoop
- fizz, buzz, zip, zap, zoop

EXPLANATION

`bar` and `foo` are passed in as arguments for `cb1` and `cb2` respectively.

Context Quiz Recall

```javascript
let cat = {
  name: "Jet",
  sayName: function() {
    return this.name;
  }
};

console.log(cat.sayName()); // ???
```

What is the returned value from the above invocation of `cat.sayName`?

- ○ `undefined`
- [✓ `Jet`]
- ○ the global object
- ○ `object`
- ○ `cat`

EXPLANATION

The `cat` object is the context of the `cat.sayName` method-style invocation so the returned value will be `cat.name` - which is `"Jet"`.

```javascript
let panther = {
  pounce: function() {
    console.log("woosh");
  },
  hunt: function() {
    this.pounce();
  }
};

let goHunt = panther.hunt;
goHunt(); // TypeError: this.pounce is not a function
```

What is the context of the above `goHunt` function?

- [✓ the global object]
- `panther`
- `hunt`
- `object`

EXPLANATION

When we extract the `goHunt` method to a separate variable and then try to invoke it - the `goHunt` will have lost the context of the `panther` object. So the `goHunt` will instead be called upon the global object.

```
function sayThis() {
  if (true) {
    console.log(this); // => ???
  }
}

sayThis();
```

What is the value printed when we invoke the `sayThis` function above in Node?

- [✓ the global object]
- `sayThis`

- `block`
- `undefined`
- None of the above

EXPLANATION

The global object is the context for every function call that does not have another defined context.

A ____ is a function that is a value within an object and belongs to that object.

- `this`
- [✓ `method`]
- `object`
- None of the above
- `context`

EXPLANATION

A *method*is a function that is a value within an object and belongs to an object.

```
let panther = {
  pounce: function() {
    console.log("woosh");
  },
  hunt: function() {
    this.pounce();
  }
};
```

```
let boundHunt = panther.hunt.bind(panther);
```

What is the context of invoking the `boundHunt` function above?

- `object`
- [✓ `panther`]
- `hunt`
- the global object

EXPLANATION

When we extract the `goHunt` method and *bind* it to the `panther` object then no matter where `boundHunt` is called it will have the bound context of the `panther` object.

```
let cat = {
  whoIsThis: function() {
    return this;
  }
};

console.log(cat.whoIsThis()); // ???
```

What is the returned context from the above invocation of `cat.whoIsThis`?

- `undefined`
- [✓ `cat`]
- the global object
- `object`

- `Jet`

EXPLANATION

The `cat` object is the context of the `cat.whoIsThis` method-style invocation.

The value of `this` in a function is the function's ___.

- `scope`
- `method`
- None of the above
- [✓ `context`]
- `this`

EXPLANATION

The value of `this` in a function is that function's context.

Dotfiles Quiz

Printing the current time, your user name, and the current `git` branch (if any).

- ○ `.bash_profile`
- ○ ✓ `.bashrc`

EXPLANATION

These details are friendly reminders, don't take a lot of time to process, and will likely be helpful each time you open a new terminal. `.bashrc` gets loaded with each terminal, so it's a good place to include small snippets like this.

Checking for & downloading software updates.

- ○ `.bashrc`
- ○ ✓ `.bash_profile`

EXPLANATION

We wouldn't want to slow down our computer every time we open a new terminal window! Since `.bash_profile` only gets loaded when logging in, we should keep any downloads or potentially long-running processes confined there.

Showing a long "Message of the Day" that welcomes new users to your server.

- ○ ✓ `.bash_profile`
- ○ `.bashrc`

EXPLANATION

A big message will get **very**annoying at the top of every window. We should only show this once when logging in for the first time, so `.bash_profile`is the place to go.

Modifying your `PATH`variable: `export PATH="/other/file:$PATH"`

- ○ `.bashrc`
- [✓ `.bash_profile`]

EXPLANATION

Consider what would happen if this code was run each time you opened a new terminal window. First you'd have `/other/file:$PATH`. With the second window, it would become `/other/file:/other/file:$PATH`. The third window: `/other/file:/other/file:/other/file:$PATH`. It won't take long for that to get out of control! Any modifications that aren't idempotent should be placed in `.bash_profile`to ensure they're not building up consequences.

Customizing your prompt and setting custom `alias`es.

- ○ `.bash_profile`
- [✓ `.bashrc`]

EXPLANATION

Minor customizations like these can make a big difference in your workflow! Don't lose them every time you open a new terminal. Placing these in `.bashrc`ensures that each terminal session has a consistent look & feel, and makes your transitions between sessions much easier to manage.

Falsey Values in JavaScript Quiz

```
if ("false") {
  console.log("Hello!");
} else if ([]) {
  console.log("Goodbye!");
} else if ("") {
  console.log("Have a nice day!");
} else {
  console.log("party time is over");
}
```

What will be printed when the above code is run?

- `"party time is over"`
- `"Have a nice day!"`
- [✓ `"Hello!"`]
- `"Goodbye!"`

EXPLANATION

The string `"false"` is still a non-empty string so when we hit our first conditional that condition will evaluate to true!

```
if (!"0") {
  console.log("Hello!");
} else if (!-42) {
  console.log("Goodbye!");
} else if (!-Infinity) {
  console.log("Have a nice day!");
} else {
    console.log("We meet again");
  }
```

What will be printed when the above code is run?

- ◯ `"Have a nice day!"`
- ◯ `"Hello!"`
- [✓ `"We meet again"`]
- ◯ `"Goodbye!"`

EXPLANATION

All of the statements within the `if..else` block will evaluate as truthy because none of them are one of the seven falsey values in JS (`NaN`, `false`, `0`, `""`, `On`, `undefined` and `null`).

Which of the following will evaluate as falsey in JavaScript?

- [✓ `""`]
- ◯ `{}`
- ◯ `[]`
- ◯ `17`

EXPLANATION

An empty string will evaluate as falsey in JavaScript - all the other answers are truthy!

Select the following that are falsey values in JavaScript:

- [ ] `false`
- [ ] `""`
- [ ] `NaN`
- [ ] `undefined`
- [ ] `null`
- [ ] `0`
- [ ] `0n`

EXPLANATION

These are all falsey values in JavaScript. These are actually the seven falsey values in JavaScript. `0n` is the BigInt primitive data type's falsey value.

Git Actions Quiz

Updates branch refs.

- [✓ Pushing to a remote]
- ☐ Adding to staging
- [✓ Committing]

EXPLANATION

Using `git add` doesn't affect branch refs, but any sort of commit will. Committing locally will move your local HEAD ref to your new commit, and `git push` will update the remote repository's branch ref to the new commit you've added.

Only affects your local repository.

- [✓ Adding to staging]
- ☐ Pushing to a remote
- [✓ Committing]

EXPLANATION

The key word here is "local". Your staging area and commit history are limited to the repository on your machine. Only after using `git push` does your commit history & code get shared with the remote.

Makes code available for a pull request.

- ○ Committing

- [✓ Pushing to a remote]
- Adding to staging

EXPLANATION

You must use `git push` to make code accessible to others. There's no way to open a pull request on your local repo!

Can be easily rolled back without affecting your repository's history.

- [✓ Adding to staging]
- Pushing to a remote
- Committing

EXPLANATION

Until you've used `git commit`, your commit history does not reflect your changes. `git add` can be easily rolled back with `git reset` or `git checkout`.

Creates a new commit in your local commit history.

- Pushing to a remote
- [✓ Committing]
- Adding to staging

EXPLANATION

Using `git commit` will add your changes as a new commit in your local repo. `git add` moves your changes to the staging area, but doesn't commit them, and pushing to a remote only adds a commit to a remote repository.

Git Rebase Quiz

Adds an additional commit in the event of a conflict.

○ Rebase

[✓ Merge]

EXPLANATION

A "merge commit" is created to preserve changes you've made while resolving a merge conflict.

"Rewrites history" and may create isolated, unreachable commits.

[✓ Rebase]

○ Merge

EXPLANATION

`git rebase` is a useful tool, but can be dangerous! Commit hashes will be regenerated when rebasing.

Is OK to use on code after it has been pushed to a remote.

○ Rebase

[✓ Merge]

EXPLANATION

Remember the "Golden Rule of Git": Never rebase or reset code that you've shared with others!

Generates new commit hashes for existing commits.

- ○ Rebase
- ○ Merge

EXPLANATION

Rebasing "rewrites history" - including the commit hashes!

Safely incorporates code from another branch into your current branch.

- ○ Merge
- ○ Rebase

EXPLANATION

`git merge` is a safe operation, as it won't change the history of your branch.

Function Hoisting in JavaScript Quiz

```
hello();

var hello = function() {
  console.log("hello!");
};
```

What type of error will be thrown when the above code snippet is run?

- ○ `ReferenceError: Cannot access 'hello' before initialization`
- ○ No error will be thrown from the above code snippet.
- [✓ ○ `TypeError: hello is not a function`]

EXPLANATION

In the above code snippet the named var declared variable is hoisted to the top of the scope with the value of `undefined`. The first line of the code snippet above will then attempt to invoke `undefined` resulting in a `TypeError` because the value of `hello` is not a function and therefore cannot be invoked.

```
let hello = "hello";

function hello(num) {
    console.log("hello!");
}

console.log(hello);
```

What will happen when the above code snippet is run?

- ○ `TypeError: hello is not a function`

○ `"hello"` will be printed to the console

✓ ○ `ReferenceError: Identifier 'hello' has already been declared`

EXPLANATION

Attempting to define a `let` declared variable and a function declaration with the same name in the same scope will throw a `ReferenceError` because the name cannot be declared twice.

```
console.log(goodNight());

var goodNight = function() {
  return "Good Night!";
};
```

What will happen when the above code snippet is run?

○ `"Good Night"` will be printed to the console

○ `ReferenceError: Cannot access 'goodNight' before initialization`

✓ ○ `TypeError: goodNight is not a function`

EXPLANATION

The `goodNight` function is a function expression defined using `var`. A `var` declared variable will have it's name hoisted to the top of it's scope and it's value set to `undefined`. So in the above code snippet we are attempting to invoke `undefined` so we receive a `TypeError`.

```
console.log(goodNight());

let goodNight = function goodNight() {
  return "Good Night!";
};
```

What will happen when the above code snippet is run?

- ◯ `TypeError: goodNight is not a function`
- [✓ `ReferenceError: Cannot access 'goodNight' before initialization`]
- ◯ `"Good Night"` will be printed to the console

EXPLANATION

The `goodNight` function is a function expression defined using `let`. Since any `let` variable declared variable won't be accessible until the value of the function is assigned we receive a `ReferenceError`.

```
console.log(shoutWord("apple"));

function shoutWord(word) {
  return word.toUpperCase();
}
```

What will happen when the above code snippet is run?

- ◯ `ReferenceError: Cannot access 'shoutWord' before initialization`
- [✓ `"APPLE"` will be printed to the console]
- ◯ `TypeError: shoutWord is not a function`

EXPLANATION

The `shoutWord` function is a named function declaration so it will be hoisted in memory and available in the above scope.

Retake Quiz

Retake Quiz

IIFE Quiz

IIFEs are one way to prevent the pollution of the global namespace by creating functions and variables that will disappear after the IIFE has been invoked.

- [✓ True]
- False

EXPLANATION

Variables and functions written within an IIFE cannot be accessed outside that function!

What does IIFE stand for?

- Invoked Immediately Function Enunciation
- [✓ Immediately-Invoked Function Expression]
- Involuntarily Invoked Function Expression
- Immediately-Invoked Function Embellishment

EXPLANATION

IIFE stands for Immediately-Invoked Function Expression.

A single IIFE can be invoked multiple times throughout an application.

[✓ False]

True

EXPLANATION

The exact opposite is true! An IIFE is invoked once then never again.

```
(function() {
  const test = "Hello world!";
})();

console.log(test); // ???
```

What will be printed when the above code snippet is run?

`[Function]`

`"Hello world!"`

[✓ An error is thrown.]

EXPLANATION

The variables defined within an IIFE are not available in an outer scope.

```
function() {
  console.log("hello world!");
}(); // => 'hello world!'
```

True or False: The above IIFE syntax is correct.

`true`

- [✓ `false`]

EXPLANATION

False! When we define and IIFE we need to wrap our anonymous function in the grouping operator before we invoke it. The above will give us a syntax error.

```
const result = (function() {
  return "food";
})();

console.log(result); // ???
```

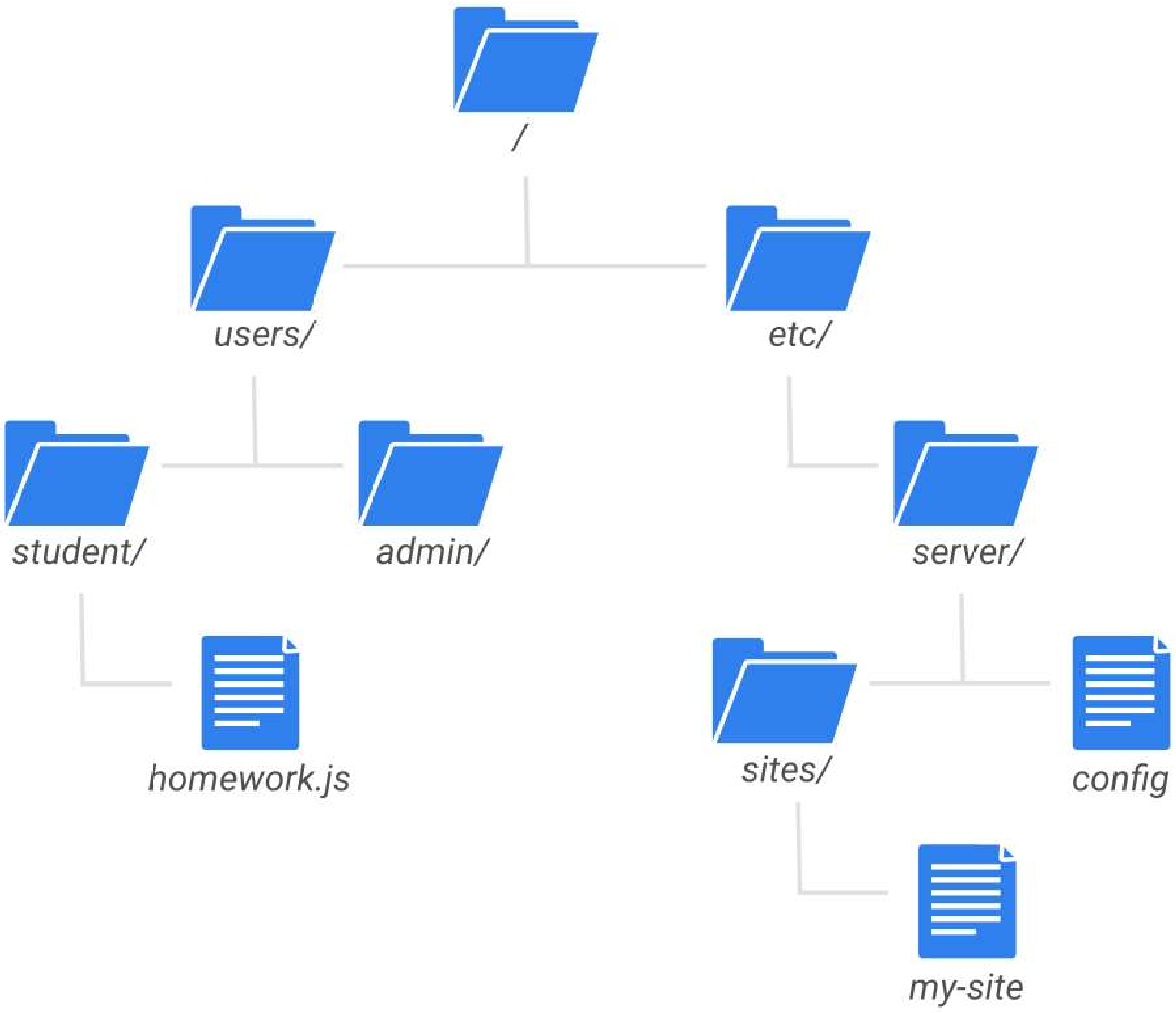
What will be printed when the above code snippet is run?

- `[Function]`
- [✓ `"food"`]
- An error is thrown.

EXPLANATION

When an IIFE is assigned to a variable the function will be invoked and then the `return` value of that function will be assigned to the variable name. So in the above example the `result` variable would have the value returned by the IIFE (which in this case is `food`).

Navigation with cd Quiz

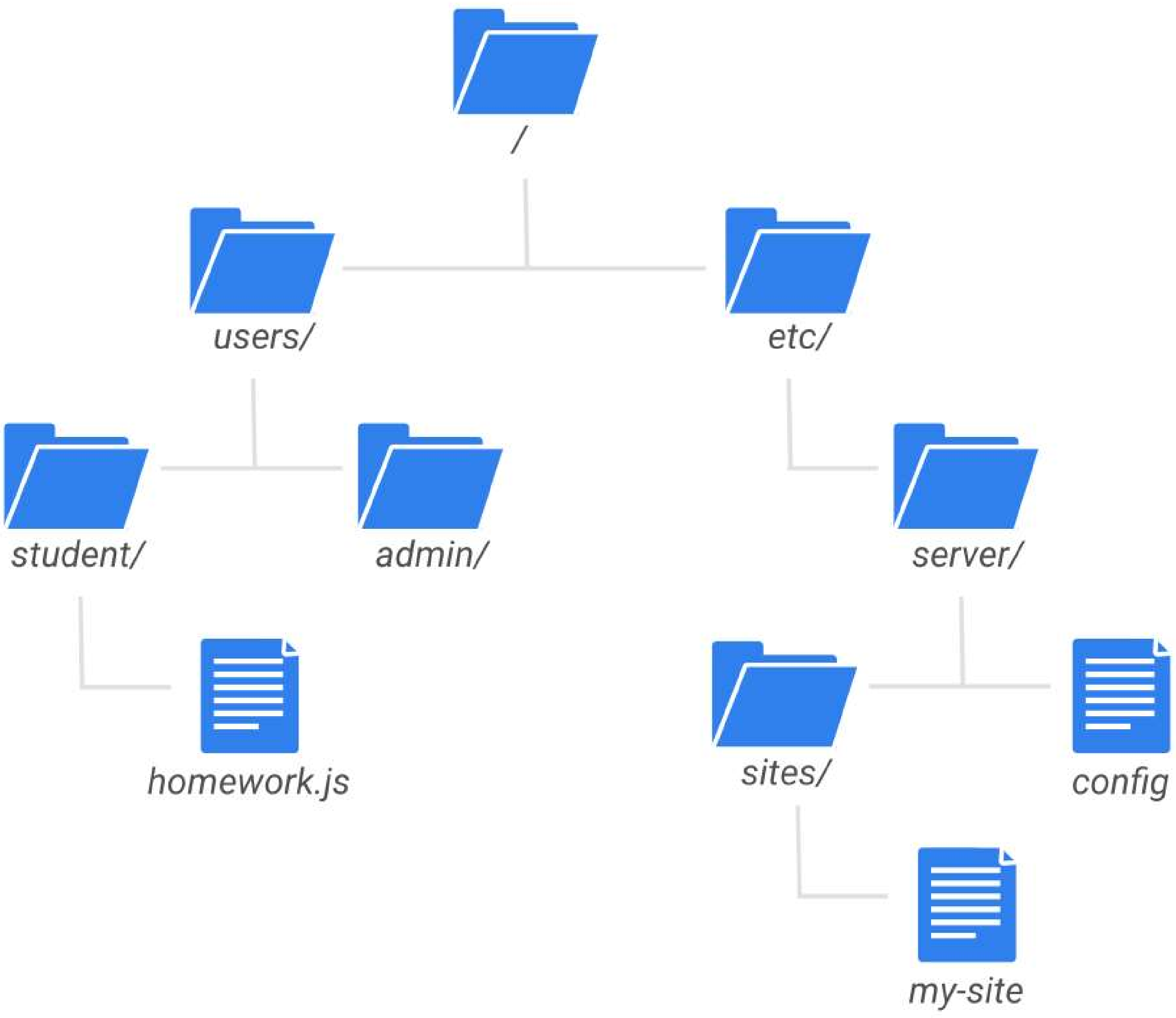


We're way down in `/etc/server/sites/`. What's the best way to get to `/users/`?

- ○ `cd ../../../users/`
- ○ `cd server/etc/users/`
- ○ `cd /users/`

EXPLANATION

Remember that prefixing a path with `/` takes us back to the *root* directory with no intermediate steps. We can't navigate our directories backwards, and while the `..` method would get us to the right place, it's harder to move three levels up versus going directly back to `/` and down only one level.

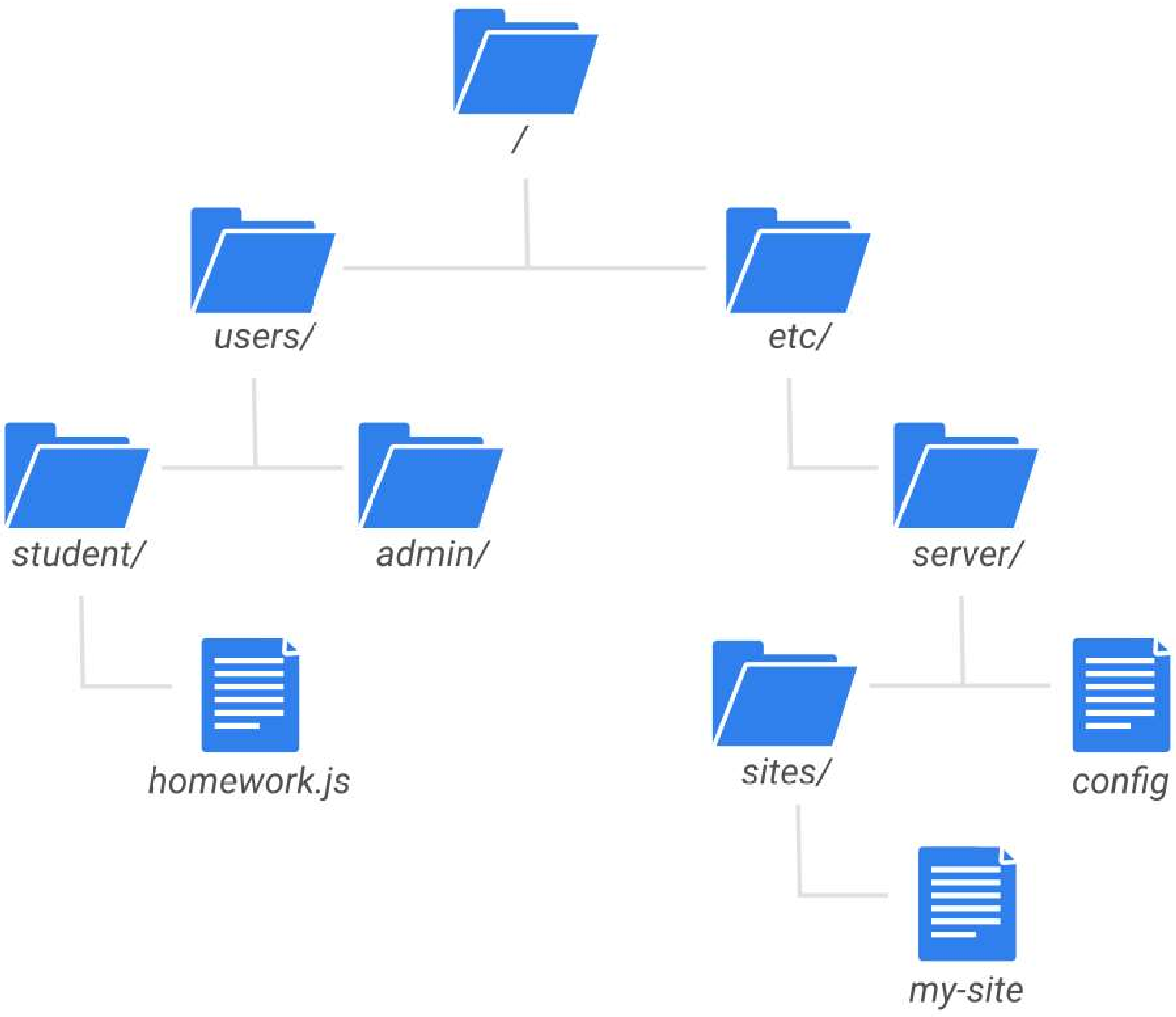


We would like to update our website. How can we get to `my-site`?

- `cd etc/server/sites/my-site/`
- `cd etc/sites/`
- `cd etc/server/sites/`

EXPLANATION

Notice that `my-site` is a file, not a directory. It's common for system & configuration file names to have no extension. `cd` is only good for changing directories, not opening files, so we don't want to include `my-site` in the path.

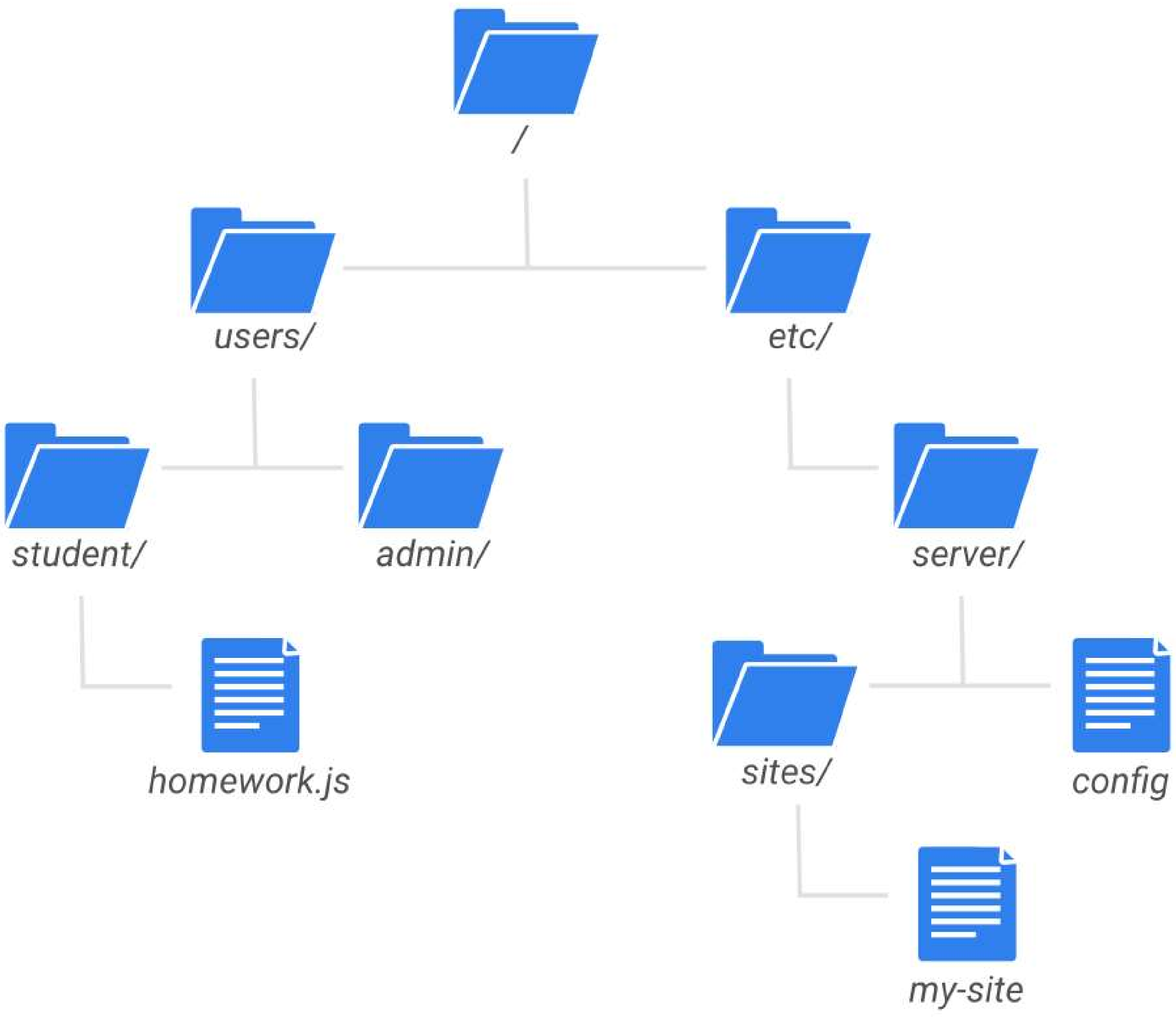


The server needs updating! Let's go to the directory containing `config`.

- ◯ `cd etc/sites/`
- ◯ `cd etc/server/`
- ◯ `cd users/`

EXPLANATION

Both the `sites` directory and the `config` file are in the `server` directory, so we want to be in `server`. Navigating to `etc/sites/` will fail since there's no such directory, and going to `users/` will succeed but isn't the correct directory.

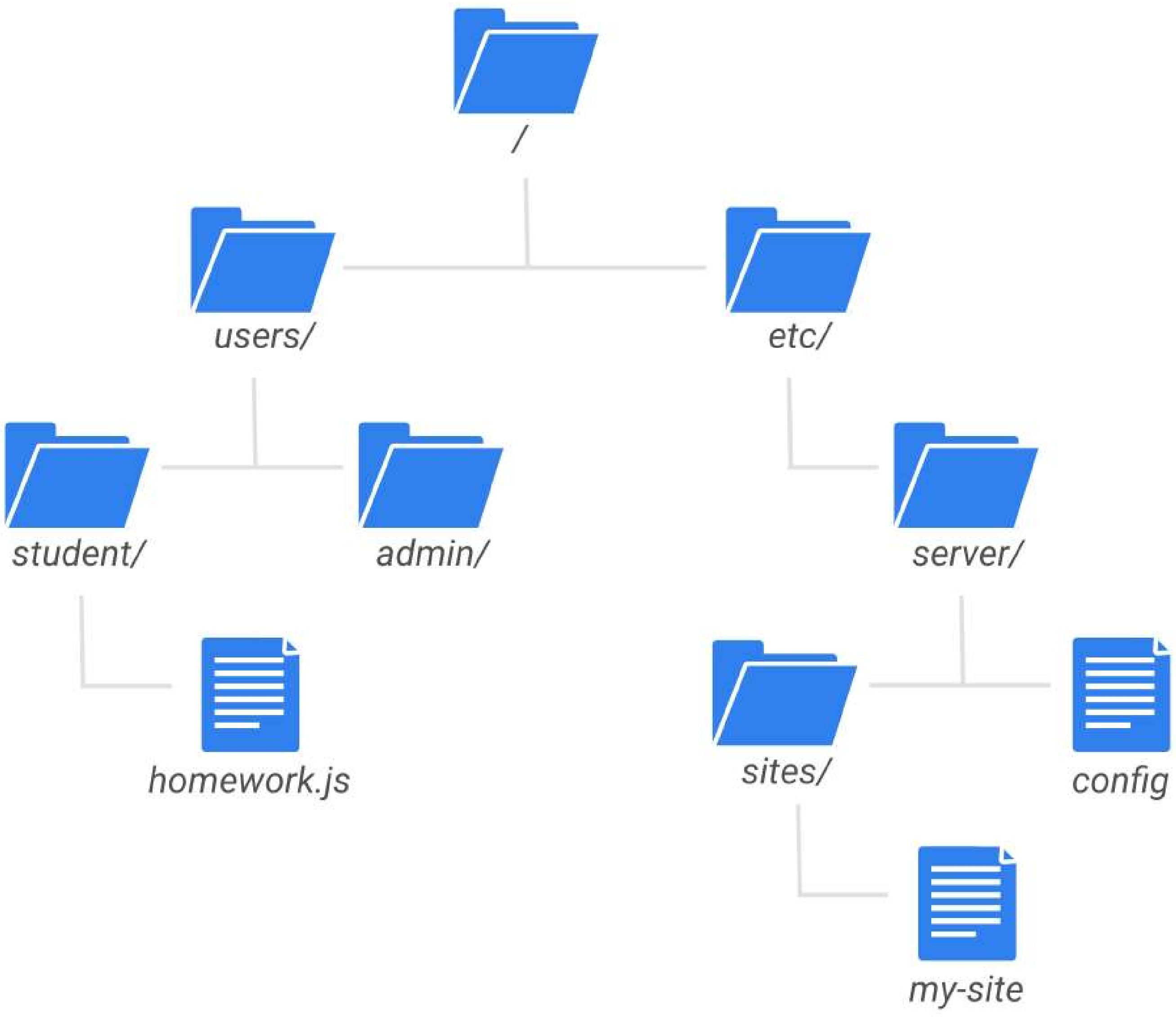


Let's move around a little more! Beginning in `/users/student`, how would we get to `/users/admin`?

- ○ `cd ../admin`
- ○ `cd admin/`
- ○ `cd users/admin`

EXPLANATION

The `..` notation in a path takes us to the parent directory (in this case, `/users/`) . Trying to go directly from the `student/` directory to `admin/` or `users/admin/` would fail as there are no `student/admin/` or `student/users/admin` directories.



It's time to turn in your homework! How do you get to the directory containing `homework.js`?

- ○ `cd users/student/`
- ○ `cd student/`
- ○ `cd users/`

EXPLANATION

Beginning from `/`, you must go through **all**intermediate directories to get to `homework.js`.

Predicting Variable Evaluations Quiz

```javascript
let func1 = () => {
  let hello;
  console.log(hello);
};

let func2 = () => {
  console.log(hello);
  let hello;
};

let func3 = () => {
  console.log(hello);
  var hello;
};
```

Which of the above functions will throw an error when invoked?

- [✓ func2]
- All three will throw errors
- func3
- func1

EXPLANATION

The `func1` function will run because a `let` declared variable with have a default value of `undefined` and will print that value. The `func3` function uses `var` to declare a variable which will hoist the name of the `hello` variable to the top of the function's scope - allowing it be logged with the default value of `undefined`. That leaves `func2` which will throw an error! This is because in `func2` we declare a variable using `let` which means that variable's name will be *hoisted* to the top of the function's scope but will be unavailable until it has been assigned a value because it is in the *temporal dead zone*.

```
const goodbye;
console.log(goodbye); // ???
```

What is printed when the above code snippet is run?

- `undefined`
- [✓ An Error is thrown.]
- `goodbye`

EXPLANATION

When declaring a new `const` variable we need to assign that variable a value because of the nature of `const` being unable to be reassigned after the variable's declaration.

```
let goodbye;
console.log(goodbye); // ???
```

What is printed when the above code snippet is run?

- `goodbye`
- [✓ `undefined`]
- An Error is thrown.

EXPLANATION

An declared but unassigned `let` variable will by default evaluate to `undefined`.

```
var hello;
console.log(hello); // ???
```

What is printed when the above code snippet is run?

- ○ `hello`
- ○ `undefined`
- ○ An Error is thrown.

EXPLANATION

A declared but unassigned `var` variable will by default evaluate to `undefined`.

Primitive Data Types Quiz

The String primitive data type has no methods.

- ○ False
- [✓ True]

EXPLANATION

The Object type is the only data type in JavaScript that has methods. The String Primitive data type is wrapped by a `String`object that has methods - but the String primitive itself has no methods.

The Object Data Type is immutable.

- ○ True
- [✓ False]

EXPLANATION

JS Primitive Data Types are immutable - and an Object is not a primitive data type.

Which of the following choices is a primitive data type in JavaScript?

- [✓ `Symbol`]
- □ `Array`
- [✓ `String`]
- [✓ `undefined`]
- [✓ `Boolean`]

☐ `Object`

EXPLANATION

Everything choice above is a primitive data type except for the `Object`type and an `Array`. An Object is **not**a primitive data type in JavaScript and an array is a type of Object.

The Object data type is the only JavaScript data type that has methods.

○ False

○ True ✓

EXPLANATION

The Object type is the only data type in JavaScript that has methods.

```
const cat = {
  name: "Jet",
  noise: function() {
    console.log("MEOW");
  }
};
```

The above `noise`function is a method of the `cat`object.

○ False

○ True ✓

EXPLANATION

A method is a function that belongs to an object. In the above example the `noise`function belongs to the `cat`object making it a method of that object.

Scope Quiz Recall

```
function letsJam() {
  // function1's scope
  let rand = 3;

  if (true) {
    const rand = 2;
  }

  if (true) {
    let rand = 1;
  }

  if (true) {
    const rand = "let's jam!";
  }

  return rand;
}

letsJam(); // ???
```

The value returned by the `letsJam`function is _.

- ◯ `3`
- ◯ `1`
- ◯ `let's jam!`
- ◯ An error is thrown
- ◯ `2`

EXPLANATION

The keywords `let` and `const` are block-scoped. Meaning that if a `let` or `const` are declared within a block `{}` that variable will stay within that block. In the above `letsJam` function the value returned will be the `rand` variable that was declared within the same outer scope - `3`.

```
function sayPuppy() {
  const puppy = "Wolfie";
  return puppy;
}

sayPuppy(); // "Wolfie"

console.log(puppy); // ????
```

What is the value logged in the last line of the snippet above (`console.log(puppy)`)?

- ○ `puppy`
- ○ `undefined`
- [✓ An Error is thrown]
- ○ `Wolfie`

EXPLANATION

Scope chaining allows an inner scope to reference an outer scope's variables but it will not allow an outer scope to access inner scope's variables.

```
function inner() {
  let str = "hello";
  return str;
}
```

```
function outer() {
  let test = inner();
  return test;
}

let result1 = outer();

result2 = inner();

result1 === result2; // ???
```

What is the value of the final line of the snippet above (`result1 === result2`) ?

- [✓ `true`]
- `false`
- An Error is thrown

EXPLANATION

No matter where `inner` is invoked it will always return the same result. This is because of *lexical scoping*.

```
let puppy = "Shasta";

function sayPuppy() {
  console.log(puppy);
}

sayPuppy(); // ???
```

What is the value logged inside the `sayPuppy` function?

◯ `puppy`

◯ `undefined`

◯ An Error is thrown

[✓ `Shasta`]

EXPLANATION

We declared a variable with `let` in the global scope. The `sayPuppy` function will have access to any variables within it's local scope as well as any variables declared in outer scopes because of scope chaining!

```
function catSound() {
  var sound = "meow";
  return sound;
}

function dogSound() {
  var sound = "bark";
  return sound;
}

let noise1 = catSound();
let noise2 = dogSound();

noise1 === noise2; // ???
```

The value of the last line in the code snippet above is: ____

[✓ `false`]

◯ `true`

EXPLANATION

Above we declared two different function-scoped variables using `var` in `catSound` and `dogSound`. Since the `var` declared variables will be function-scoped they will return different values from their separate functions.

```javascript
// 1. ???
let chicken = "bokbok";

function farmTime() {
  // 2. ???
  console.log(chicken);

  if (true) {
    // 3. ???
    let cow = "moo";
  }
}
```

In the above code snippet there are three scopes labelled with numbers. Below pick the correct answer for the name of each scope in order.

- **1.**Local/Function Scope **2.**Global Scope **3.** Block Scope
- [✓ **1.** Global Scope **2.**Local/Function Scope **3.**Block Scope]
- **1.** Block Scope **2.**Local/Function Scope **3.**Global Scope

EXPLANATION

The three scopes above are Global scope, Function scope, and Block scope in that order.

Safety with sudo Quiz

```
sudo rm -rf some-file.txt
```

○ Safe

[✓ Dangerous]

EXPLANATION

Yikes!This is **very**dangerous. If you don't have access to `rm`a file without `sudo`, there's likely a good reason. Never use `sudo`and `rm`together unless you are 100% certain you understand the potential consequences.

```
sudo ls /etc/init.d/
```

○ Dangerous

[✓ Safe]

EXPLANATION

The `ls`command is non-destructive, so it's safe to `sudo`! You might use this while browsing lower-level files in your operating system.

```
sudo chmod +x my-script.sh
```

○ Dangerous

[✓ Safe]

EXPLANATION

Using `chmod` to update the executable permission of a file is generally safe, but it's up to you to understand what that file will do when executed. You should never make scripts you've downloaded from the Internet executable unless you've read & understand the file's contents.

```
sudo chmod 777 ~/my-private-file
```

- Safe
- [✓ Dangerous]

EXPLANATION

`777` is the octal permissions notation for "everyone can do everything to this file". It's very unlikely that you want any file totally accessible to every person that uses your system! Whenever you see this command, think carefully: is there a less-permissive way to grant access to only the users that need this file, maybe by adding them to a group?

```
sudo cp /var/www/index.html /var/www/index.js
```

- [✓ Safe]
- Dangerous

EXPLANATION

The `cp` command changes the filesystem, but it doesn't remove any existing files - it just adds a new one! This is safe to `sudo` as you're very unlikely to cause negative side effects.

```
sudo mv /var/www/index.html /var/www/index.js
```

- Safe

Dangerous

EXPLANATION

Using `mv` moves a file and might cause problems with other applications that depend on the original file. This is destructive behavior; you should never `sudo` destructive behavior.

Object Key Quiz

7 minutes ✓ Completed

Score

You have submitted your quiz.
You got 5 questions correct, and 0 question(s) still being graded.
Your score so far is 100 out of 100.

Retake Quiz

Object Key Quiz

```
console.log(Symbol("foo") === Symbol("foo")); // ???
```

What happens when the above code snippet is run?

- ○ `true` is printed
- ○ An error is thrown.
- ✓ ○ `false` is printed

EXPLANATION

Each created symbol is unique! The optional description string is just for debugging purposes.

```
const species = Symbol("species");
const animal = {
  [species]: "whale",
    name: "Wally"
  };

  console.log(Object.getOwnPropertySymbols(animal)); // ???
```

What is printed when the above code snippet is run?

- ○ `[Symbol(species), "name"]`
- ✓ ○ `[Symbol(species)]`
- ○ `["name"]`
- ○ `[]`

EXPLANATION

The `Object.getOwnPropertySymbols` method will only return symbol keys - ignoring string keys.

Pick the following which are can be set as a `key` in an Object:

- ✓ ☐ `Symbol`
- ☐ `Object`
- ☐ `Number`
- ✓ ☐ `String`
- ☐ `Boolean`

EXPLANATION

An object's keys can be either a `String` *or* a `Symbol`.

```
const sym1 = Symbol();
const sym2 = Symbol();
console.log(sym1 === sym2); // ???
```

What happens when the above code snippet is run?

- ○ `true` is printed
- ✓ ○ `false` is printed
- ○ An error is thrown.

EXPLANATION

Each created symbol is unique!

```
const species = Symbol("species");
const animal = {
  [species]: "whale",
  name: "Wally"
};

console.log(Object.keys(animal)); // ???
```

What is printed when the above code snippet is run?

- ○ `[Symbol(species)]`
- ○ `[]`
- ○ `[Symbol(species), "name"]`
- ✓ ○ `["name"]`

EXPLANATION

The `Object.keys` method will only return string keys - ignoring Symbol keys.

Iteration vs. Recursion Free Response

Describe what makes a problem a good candidate for recursion. What should you look for when evaluating whether recursion or iteration is the better choice?

1431characters left

EXPLANATION

Problems with **complex**or **large**inputs may be good candidates for recursion.