NPM Commands

Creates a `package-lock.json` file.

- [x] `npm install`
- [ ] `npm`
- [ ] `npm init`

EXPLANATION

Because the `package-lock.json` file requires exact source information, we can't generate it without going through the installation process.

Shows additional commands that are available.

- [ ] `npm install`
- [x] `npm`
- [ ] `npm init`

EXPLANATION

Without running any extra commands, `npm` just shows a summary of commands that are available. This is the same as running `npm help`.

Asks a series of questions about your project.

- [ ] `npm`
- [ ] `npm install`

- [x] `npm init`

EXPLANATION

The `npm init` command uses your responses to generate a `package.json` file in your app's root directory.

Creates a `node_modules` directory.

- [ ] `npm`
- [ ] `npm init`
- [x] `npm install`

EXPLANATION

Creating the project with `npm init` prepares our metadata, but it's `npm install` that brings in dependencies. These get installed into the `node_modules` directory, which will be created if it doesn't already exist.

To add the dependency 'lodash' to the package.json file, which command do you use?

- [ ] `npm init lodash`
- [ ] `npm install`
- [x] `npm install lodash`
- [ ] `npm`

EXPLANATION

If you've already initialized your app's root directory, `npm install` will notice the `package.json` file and add the requested dependency to it. This is a helpful way of keeping all your npm-related files in sync.

Object-Oriented Principles Quiz

Which of the following keywords does JavaScript support to call methods on a parent class (prototype) when defined with ES6 syntax?

- ◯ `parent`
- ◯ `super`
- ◯ `base`
- ◯ `prototype`

EXPLANATION

The ES6 syntax for declaring classes provides the `super` keyword with which to access methods on a parent class (prototype)

Which of following is the definition of the "inheritance" object-oriented principle?

- ◯ Inheritance is the mechanism by which an object acquires the behavior (and data) from the definition of its parent class (prototype).
- ◯ Inheritance is the ability to provide multiple methods with the same name.
- ◯ Inheritance is the mechanism by which an object passes information from its constructor to an instance method defined for that object.
- ◯ Inheritance is the mechanism by which a programmer can copy and paste code from one file to another.

EXPLANATION

Inheritance is a way to write code in methods and share those between other classes to reduce the maintenance cost and promote easy-to-understand code.

Which one of the following types of inheritance does JavaScript support?

- **Interface inheritance**where a class must implement any undefined methods of its parent class (prototype).
- **Iterative inheritance**where each iteration of the parent class (prototype) is used to increment the version of the object created from it.
- **Implementation inheritance**where, given an object, code can invoke methods on the object defined on "parent" classes (prototypes).
- **Identity inheritance**where each object gets an identity which is used to identify that object from its parent class (prototype).

EXPLANATION

JavaScript has *implementation inheritance*which means that an object will inherit the implementation of the methods and data from its parent classes (prototypes) unless otherwise overridden.

With respect to object-oriented programming, what is a "constructor"?

- A constructor is a method used to allocate memory on the computer and is called when the program starts.
- A constructor is a method used to initialize the state of an object at the time of object creation.
- A constructor is a class that creates other classes.
- A constructor

EXPLANATION

The special method known as a "constructor" provides the object-oriented system a way to initialize the state of an object and ensure all of its information dependencies are met.

Choose two or more of the following that describe what a "class" is in JavaScript.

- [x] A class is a template by which objects are created using the `new` keyword.
- [ ] A class is the mechanism by which the primitive values of numbers, Booleans, `null`, and `undefined` are represented in JavaScript.
- [x] A class is a representation of the implementation of an object that you want to create.
- [x] A class is the collection of behavior and data that an object should have.

EXPLANATION

The representation of the object is like a blueprint for the behavior (methods) and data (fields) that will be associated with and available on the object constructed from the class.

Which of following is the definition of the "encapsulation" object-oriented principle?

- ( ) A method by which a programmer can *increase* the coupling between components.
- ( ) Putting your data into a "module" (or "capsule") so that it is easy to import and export.
- ( ) Providing more than one way to create an object from a class.
- (x) Providing a simple way to interact with a potentially complex implementation of a concept.

EXPLANATION

The object-oriented principle of "encapsulation" is the method by which a programmer can *provide a simple way to interact with the potentially complex implementation of a concept*. Code that uses

the simple way needs not know the data types that make up the state of the object nor the implementation of each of the behavioral methods on the class.

Which of following is the definition of the "polymorphism" object-oriented principle?

- ◯ You can treat an object as an instance of its own class and only its direct parent class (prototype).
- ◯ You cannot call any methods on the object.
- ◯ You can treat an object as an instance of its own class and none of its parent classes (prototypes).
- ◯ You can treat an object as an instance of its own class and all of its parent classes (prototypes).

EXPLANATION

Polymorphism allows you to treat the object as its own class or any of its parent classes (prototypes). This allows code that uses the object to take advantage of the behavior that it inherits from its parent classes (prototypes).

In JavaScript, for a class declared using the ES6 syntax and named `Animal`, what is the name of its constructor method?

- ◯ `animal`
- ◯ `ctor`
- ◯ `constructor`
- ◯ `Animal`

EXPLANATION

In JavaScript, the special method named `constructor` is the method run at the time the object is created.

The Recursion Quiz Again

```
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 recursive case
- [ ] A parameter.
- [x] A recursive step
- [x] A base 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.

- [ ] Base: `volume - 1`
- [x] Base: `volume === 0`
- [ ] Recursive: `volume === 10`
- [x] 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, what is the recursive step?

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

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 errors will result from running the above function?

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

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, 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`.

What do you remember about callbacks?

```
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"?

- ◯ `foo`
- ◯ `bar`
- ◯ `console.log`

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, fizz, zap, buzz, zoop
- ◯ fizz, buzz, zip, zap, zoop
- ◯ zip, buzz, zap, fizz, zoop

EXPLANATION

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

```
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'.

```
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
- ◯ vroom, swoosh, swoosh, swoosh, skrrt
- ◯ vroom, swoosh, swoosh, skrrt
- ◯ swoosh, vroom, skrrt

EXPLANATION

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

```
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?

- ◯ Ramen, Gazpacho, Egusi, Bisque, Pho
- ◯ Bisque, Pho, Gazpacho, Egusi, Ramen
- ◯ Bisque, Gazpacho, Ramen, Egusi, Pho
- ◯ Bisque, Gazpacho, Egusi, Ramen, 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?

- ◯ Zion, Everglades, Arches, Sequoia, Acadia
- ◯ Zion, Everglades, Sequoia, Arches, Acadia
- ◯ Arches, Everglades, Sequoia, Zion, 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(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?

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

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

- ◯ ability to be an argument to a function
- ◯ ability to be a return value of a function
- ◯ ability to be assigned to a variable
- ◯ ability to be mutated

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.

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;
};

console.log(foo);
```

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

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

EXPLANATION

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

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

foo;
```

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

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

EXPLANATION

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

Can you still think async?

```
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 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, boop!, buzz
- ◯ boop!, buzz, fizz
- ◯ fizz, buzz, boop!
- ◯ boop!, fizz, buzz

EXPLANATION

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

```
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?

- ◯ hello!, async
- ◯ async, hello!

EXPLANATION

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

```
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!
- fizz, buzz, farm!, boop!
- farm!, boop!, fizz, buzz

EXPLANATION

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

Can you control the flow?

```
let groceries = ["apples", "potatoes", "milk"];
```

What is the above `groceries` variable?

- [x] Array
- [ ] Function
- [ ] String

EXPLANATION

A list of comma separated values surrounded by square brackets `[]` is an array.

```
let puppies = ["Laser", "Katy", "Jet", "Layla"];
```

What is the value we'd receive if we read the value at `puppies[1]`?

- [ ] "Layla"
- [ ] "Laser"
- [ ] "Jet"
- [x] "Katy"

EXPLANATION

Array indices *always* start at 0. So if we access the `puppies` array at the index of `1` we'd get "Katy"!

```
function potatoSpeak() {
  console.log("I am potato!");
}
function sadSpeak() {
  console.log("I am NOT potato :(");
}

function isThisPotato(word) {
  if (word === "potato") {
    potatoSpeak();
  } else {
    sadSpeak();
  }
}
```

In the code snippet above we have written a function that accepts one word and if that word is "potato" the `potatoSpeak` function is called - otherwise the `sadSpeak` function is called. The condition inside the `isThisPotato` function above is an example of what kind of conditional?

- ◯ mutually exclusive
- ◯ mutually acceptable
- ◯ None of the above
- ◯ doubly doable

EXPLANATION

Since the word is either "potato" or not "potato" it means that the scenario we are faced with is **mutually exclusive.** Meaning that the condition can only be true or false but not both (the word is either "potato" or not).

Fill in the blank for the following: A(n) _ is an ordered list of values defined by using square brackets.

- ( ) String
- ( ) Function
- (•) Array

EXPLANATION

An *array* is a list of comma separated values surrounded by square brackets `[]`.

```
let groceries = ["apples", "potatoes", "milk"];
```

Which of the following are the correct ways to access the value of `"milk"` in the above `groceries` array?

- [ ] `groceries[groceries.length - 2]`
- [x] `groceries[2]`
- [ ] `groceries[0]`
- [x] `groceries[groceries.length - 1]`
- [ ] `groceries[groceries.length]`
- [ ] `groceries[1]`

EXPLANATION

Since `"milk"` is the last value in the `groceries` array we can access in through it's index directly - if we count up from 0 we get 2. Or we can look at the length of the above array minus 1 to get the same answer!

```
let age = 30;

if (age > 30) {
  console.log("older than 30");
} else {
  console.log("younger than 30");
}
```

Predict what will happen in the above example. Which `console.log`s will actually print?

- ◯ prints `"younger than 30"`
- ◯ prints `'older than 30'`

EXPLANATION

This is an example of a *mutually exclusive* condition! Since `age` is set to 30 and the first condition will only be met if age is *greater* than 30 then the `else` statement will be run - printing `"younger than 30"`