Code Snippets: OOP

▼ Basic collaborator object

```
let cat = {
  name: 'Fluffy',

  makeNoise() {
    console.log('Meow! Meow!');
  },

  eat() {
    // implementation
  },
};

let pete = {
  name: 'Pete',
  pet: cat,

  printName() {
    console.log(`My name is ${this.name}!`);
    console.log(`My pet's name is ${this.pet.name}`);
  },
};
```

Material description:

The `pete` object has a collaborator object stored in its `pet` property. The `pete` object and the object referenced by its `pet` property work together. When we need to access the `pet` object or have it perform a behavior, we can use `pete.pet` to access and use the object's properties. For instance, on line 19, the `pete` object collaborates with the `cat` object (via `this.pet`) toaccess the `cat`'s name.

▼ Basic prototypal inheritance

```
let a = {
  foo: 1,
  bar: 2,
};

let b = Object.create(a);
b.foo; // => 1
```

```
> let a = { foo: 1, bar: 2 }
undefined

> let b = Object.create(a)
undefined

> b.foo
1

> b
{}
```

The function `Object.create` creates a new object that inherits properties from an existing object. It takes an object that is called the **prototype object** as an argument, and returns a new object that inherits properties from the prototype object. The newly created object has access to all the properties and methods that the prototype object provides.

An unusual aspect of this relationship is that the inheriting object (`b` above) doesn't receive any properties or methods of its own. Instead, it **delegates** property and method access to its prototype

▼ Prototype Chain

```
let a = {
  foo: 1,
};

let b = {
  bar: 2,
};

let c = {
  baz: 3,
};

Object.setPrototypeOf(c, b);
Object.setPrototypeOf(b, a);

console.log(c.bar); // => 2
console.log(c.foo); // => 1
```

In this code, object `c` inherits from object `b` which, in turn, inherits from `a`. Stated differently, `b` is the prototype of `c` and `a` is the prototype of `b`. All properties that

you can access on `a` or `b` are now available on `c`. We say that objects `b` and `a` are part of the **prototype chain** of object `c`. The complete prototype chain also includes the default prototype, which is the prototype of object `a` in this case. Since the prototype of `Object.prototype` is `null`, the complete prototype chain looks like this:

```
c --> b --> a --> Object.prototype --> null
```

▼ Practice Problems: Object Prototypes

1. What will the following code log to the console? Explain why it logs that value.

```
let qux = { foo: 1 };
let baz = Object.create(qux);
console.log(baz.foo + qux.foo);
```

▼ Answer:

On line 1, the global variable `qux` is declared and initialized to a reference to an object with one property, `foo`; the value of `foo` is `1`. On line 2, the global variable `baz` is declared and initialized to a reference to an object that is returned by `Object.create(qux)`. The `Object.create` method returns an object that inherits the properties from the existing object that is passed into the method as an argument. In this case, since `qux` is passed into an object, `baz` references an object that has none of its own properties but has inherited the `foo` property from the object referenced by `qux`. Since `baz.foo` and `qux.foo` reference the same property, which has a value of 1, the output of this is code is `2`.

2. What will the following code log to the console? Explain why it logs that value.

```
let qux = { foo: 1 };
let baz = Object.create(qux);
baz.foo = 2;

console.log(baz.foo + qux.foo);
```

▼ Answer:

On line 1, the global variable `qux` is declared and initialized to a pointer to an object with one property, `foo`; the value of `foo` is `1`. On line 2, the global variable `baz` is declared and initialized to a pointer to an object that is returned by `Object.create(qux)` . The `Object.create` method returns an object that inherits the properties from the existing object that is passed into the method as an argument. In this case, since `qux` is passed into the method, `baz` references an object that has none of its own properties but has inherited the `foo` property from the object pointed to by `qux`.

On line 3, `baz.foo` is reassigned to `2`; although `foo` was not an own property of the `baz`, property assignment does not use the prototype chain; instead, it creates a new property in the baz object named `foo`. As a result, this reassignment operation does not reflect in the prototype `qux`. On line 5, When `baz.foo` and `qux.foo` are added together, `baz.foo` returns the value of its "own" `foo` property, while `qux.foo` returns the value of its "own" `foo` property. Thus, the result is `3`.

3. What will the following code log to the console? Explain why it logs that value.

```
let qux = { foo: 1 };
let baz = Object.create(qux);
qux.foo = 2;

console.log(baz.foo + qux.foo);
```

▼ Answer:

On line 1, the global variable `qux` is declared and initialized to a pointer to an object with one property, `foo`; the value of `foo` is `1`. On line 2, the global variable `baz` is declared and initialized to a pointer to an object that is returned by `Object.create(qux)` . The `Object.create` method returns an object that inherits the properties from the existing object that is passed into the method as an argument. In this case, since `qux` is passed into the method, `baz` references an object that has none of its own properties but has inherited the `foo` property from the object pointed to by `qux`.

On line 3, the `foo` property of `qux` is reassigned to 2. Since `baz` doesn't have its "own" copy of the `foo` property, JavaScript uses the prototype chain to look up `baz.foo`, and it finds the `foo` property in `qux`. Thus, the output of line 5 is `4`.

4. Write a function that searches the prototype chain of an object for a given property and assigns it a new value. If the property does not exist in any of the prototype objects, the function should do nothing. The following code should work as shown:

```
let fooA = { bar: 1 };
let fooB = Object.create(fooA);
let fooC = Object.create(fooB);

assignProperty(fooC, "bar", 2);
console.log(fooA.bar); // 2
console.log(fooC.bar); // 2

assignProperty(fooC, "qux", 3);
console.log(fooA.qux); // undefined
console.log(fooC.qux); // undefined
console.log(fooA.hasOwnProperty("qux")); // false
console.log(fooC.hasOwnProperty("qux")); // false
```

▼ Answer:

```
function assignProperty(obj, property, value) {
  while (obj !== null) {
    if (obj.hasOwnProperty(property)) {
      obj[property] = value;
      break;
    }

    obj = Object.getPrototypeOf(obj);
  }
}
```

```
// recursive solution
function assignProperty(obj, property, value) {
  if (obj === null) { // property not found
```

```
    return;
  } else if (obj.hasOwnProperty(property)) {
    obj[property] = value;
  } else {
    assignProperty(Object.getPrototypeOf(obj), property, value);
  }
}
```

5. Consider the following two loops:

```
for (let property in foo) {
  console.log(`${property}: ${foo[property]}`);
}
```

```
Object.keys(foo).forEach(property => {
  console.log(`${property}: ${foo[property]}`);
});
```

If `foo` is an arbitrary object, will these loops always log the same results to the console? Explain why they do or do not. If they don't always log the same information, show an example of when the results differ.

▼ Answer:

The `Object.keys` method will only iterate through an object's own properties, while the `for/in` method of iteration will iterate through all enumerable properties of the object. As a result, if an arbitrary object has a prototype chain or inherited properties, the loops will not log the same results to console.

6. How do you create an object that doesn't have a prototype? How can you determine whether an object has a prototype?

▼ Answer:

Create an object with no prototype by using `Object.create(null)` or by using `Object.setPrototypeOf()` to set prototype to null. You can determine whether an object has a prototype using `Object.getPrototypeOf()`.

▼ Practice problems: Execution Context

▼ How call works

functions:

```
function logNum() {
  console.log(this.num);
}

let obj = {
  num: 42
};

logNum.call(obj); // logs 42

// above is functionally similar to:
function logNum() {
  console.log(this.num);
}

let obj = {
  num: 42
};

obj.logNum = logNum;
obj.logNum(); // logs 42
```

methods:

```
let obj1 = {
  logNum() {
    console.log(this.num);
  }
};

let obj2 = {
  num: 42
};

obj1.logNum.call(obj2); // logs 42
```

1. What will the following code output? Try to determine the results without running the code.

```
function func() {
  return this;
}

let context = func();

console.log(context);
```

▼ Answer:

On line 1, the function `func` is declared. The return value of `func` is `this`. On line 5, the global variable `context` is declared and initialized to the return value of `func`. Since there is no explicit context set in the invocation of `func`, the value of `this` is the global object; thus, `context` is initialized to a pointer to the global object. In Node, that is `global` and in browser that is `window`.

2. What will the following code output? Explain the difference, if any, between this output and that of problem 1.

```
let obj = {
  func: function() {
    return this;
  },
};

let context = obj.func();

console.log(context);
```

▼ Answer:

On line 1, a global variable `obj` is declared and initialized to a pointer to an object. This object has one property, `func`, which is a function that returns the value of `this`. On line 6, the global variable `context` is declared and

initialized to the return value of a method call: the `func` property of the object `obj`.

Since no explicit context is set for the method call, `this` is implicitly bound to the object where the function exists as a property. Thus, the `func` invocation returns `obj`.

```
{ func: [Function: func] }
```

3. What will the following code output?

```
message = 'Hello from the global scope!';

function deliverMessage() {
  console.log(this.message);
}

deliverMessage();

let foo = {
  message: 'Hello from the function scope!',
};

foo.deliverMessage = deliverMessage;

foo.deliverMessage();
```

▼ Answer:

On line 1, a variable `message` is initialized to a string without any variable declaration keywords. As a result, `message` becomes a property of the global object.

On line 3, the function `deliverMessage` is declared; the function logs the value of `this.message` to the console.

On line 9, the function `deliverMessage` is invoked; since no explicit execution context is set, `this` is implicitly bound to the global object. As a result, the value of `this.message` is the string `'Hello from the global scope!'`, which will be logged to console.

On line 11, the global variable `foo` is declared and initialized to a pointer to an object. This object has one property, `message`; the value of which is a

string. On line 15, a property `deliverMessage` is added to the object `foo`; the value of which is the function `deliverMessage`.

On line 17, the method `deliverMessage` of the object `foo` is invoked. Since no explicit execution context is set, `this` is implicitly bound to the object from which the method is called, `foo`. As a result, `this.message` refers to `foo.message`, resulting in the string `'Hello from the function scope!'` being logged to console.

4. Take a look at the following code snippet. Use `call` to invoke the `add` method but with `foo` as execution context. What will this return?

```
let foo = {
  a: 1,
  b: 2,
};

let bar = {
   a: 'abc',
   b: 'def',
   add: function() {
     return this.a + this.b;
   },
};
```

▼ Answer:

```
bar.add.call(foo);

// This will return 3
```

▼ Practice Problems: Hard Binding

1. What will the following code log to the console?

```
let obj = {
  message: 'JavaScript',
};
```

```
function foo() {
  console.log(this.message);
}

foo.bind(obj);
```

▼ Answer:

Nothing will be logged to the console.

On line 9, the bind method is invoked on the function `foo`; the bind method does not invoke the function, it simply returns a new function with a permanently bound execution context. As a result, nothing is logged to the console.

2. What will the following code output?

```
let obj = {
  a: 2,
  b: 3,
};

function foo() {
  return this.a + this.b;
}

let bar = foo.bind(obj);

console.log(foo());
console.log(bar());
```

▼ Answer:

When the `foo` function is invoked on line 12, no execution context is set; as a result, `this` is implicitly bound to the global object. Since properties `a` and `b` do not exist in the global object, `this.a` and `this.b` evaluate to undefined, resulting in a `NaN` output value.

On line 10, the variable `bar` is initialized to the return value of `foo.bind(obj)`, which is a new function: `foo` with the execution context set to `obj`. When `bar` is invoked, `5` is returned and logged to the console.

Note: if code was run in strict mode, `implicit` context `foo` would be `undefined` and as a result, a `TypeError` would be raised when attempting to find `this.a`.

3. What will the code below log to the console?

```
let positivity = {
  message: 'JavaScript makes sense!',
};

let negativity = {
  message: 'JavaScript makes no sense!',
};

function foo() {
  console.log(this.message);
}

let bar = foo.bind(positivity);

negativity.logMessage = bar;
negativity.logMessage();
```

▼ Answer:

On line 13, `bar` is initialized to the return value of `foo.bind(positivity)`, which is a new function `foo` with the execution context permanently bound to the object `positivity`. On line 15, a new property, `logMessage` is added to the object `negativity` and set to the function `bar`. On line 16, the function is invoked as a method of the object `negativity.` Since the function's execution context was permanently bound to the object `positivity` with the bind method, `this.message` refers to `positivity.message`, despite the function being invoked as a method of the negativity object.

Asa result, the output is `'JavaScript makes sense!'`.

4. What will the code below output?

```
let obj = {
  a: 'Amazebulous!',
```

```
};
let otherObj = {
  a: "That's not a real word!",
};

function foo() {
  console.log(this.a);
}

let bar = foo.bind(obj);

bar.call(otherObj);
```

▼ Answer:

Once a function's context gets bound using `bind`, its context can't be changed, even with `call` and `apply`. In keeping with this, the last line of our code outputs `"Amazebulous!"`, because the function `bar`'s context has been permanently bound to `obj`.

▼ Practice problems: Dealing with Context Loss

1. The code below should output `"Christopher Turk is a Surgeon"`. Without running the code, what will it output? If there is a difference between the actual and desired output, explain the difference.

```
let turk = {
  firstName: 'Christopher',
  lastName: 'Turk',
  occupation: 'Surgeon',
  getDescription() {
      return this.firstName + ' ' + this.lastName + ' is a '
                                  + this.occupation + '.';
  }
};

function logReturnVal(func) {
  let returnVal = func();
  console.log(returnVal);
}

logReturnVal(turk.getDescription);
```

▼ Answer:

Passing `turk.getDescription` into the function `logReturnVal` removes the `getDescription` method from its `turk` object context. As a result, when the function is executing within `logReturnVal`, the execution context is the global object, resulting in `this.firstName`, `this.LastName`, and `this.Occupation` evaluating to undefined.

2. Modify the program from the previous problem so that `logReturnVal` accepts an additional `context` argument. If you then run the program with `turk` as the context argument, it should produce the desired output.

   ▼ Answer:

```
let turk = {
  firstName: 'Christopher',
  lastName: 'Turk',
  occupation: 'Surgeon',
  getDescription() {
      return this.firstName + ' ' + this.lastName + ' is a '
                                  + this.occupation + '.';
  }
};

function logReturnVal(func, context) {
  let returnVal = func.call(context);  // use call to leverage context
  console.log(returnVal);
}

logReturnVal(turk.getDescription, turk);
```

3. Suppose that we want to extract `getDescription` from `turk`, but we always want it to execute with `turk` as its execution context. How would you modify your code to do that?

   ▼ Answer:

```
let extract = turk.getDescription.bing(turk);
logReturnVal(extract);
```

4. Consider the following code:

```
const TESgames = {
  titles: ['Arena', 'Daggerfall', 'Morrowind', 'Oblivion', 'Skyrim'],
  seriesTitle: 'The Elder Scrolls',
  listGames: function() {
    this.titles.forEach(function(title) {
      console.log(this.seriesTitle + ': ' + title);
    });
  }
};

TESgames.listGames();
```

   Will this code produce the following output? Why or why not?

```
The Elder Scrolls: Arena
The Elder Scrolls: Daggerfall
The Elder Scrolls: Morrowind
The Elder Scrolls: Oblivion
The Elder Scrolls: Skyrim
```

   ▼ Answer:

   Since functions lose their surrounding context when used as arguments to another function, the context of line 6 is not the `TESgames` object. Instead, it is the global object. Thus, `this.seriesTitle` resolves to `undefined` rather than `"The Elder Scrolls"`.

5. Use `let self = this;` to ensure that `TESgames.listGames` uses `TESGames` as its context and logs the proper output.

   ▼ Answer:

```
const TESgames = {
  titles: ['Arena', 'Daggerfall', 'Morrowind', 'Oblivion', 'Skyrim'],
  seriesTitle: 'The Elder Scrolls',
  listGames: function() {
    let self = this;
    this.titles.forEach(function(title) {
```

```
      console.log(self.seriesTitle + ': ' + title);
    });
  }
};

TESgames.listGames();
```

6. The `forEach` method provides an alternative way to supply the execution context for the callback function. Modify the program from the previous problem to use that technique to produce the proper output:

   ▼ Answer:

```
const TESgames = {
  titles: ['Arena', 'Daggerfall', 'Morrowind', 'Oblivion', 'Skyrim'],
  seriesTitle: 'The Elder Scrolls',
  listGames: function() {
    this.titles.forEach(function(title) {
      console.log(self.seriesTitle + ': ' + title);
    }, this);
  }
};

TESgames.listGames();
```

7. Use an arrow function to achieve the same result:

   ▼ Answer:

```
const TESgames = {
  titles: ['Arena', 'Daggerfall', 'Morrowind', 'Oblivion', 'Skyrim'],
  seriesTitle: 'The Elder Scrolls',
  listGames: function() {
    this.titles.forEach(title => {
      console.log(self.seriesTitle + ': ' + title);
    });
  }
};

TESgames.listGames();
```

8. What will the value of foo.a be after this code runs?

```
let foo = {
  a: 0,
  incrementA: function() {
    function increment() {
      this.a += 1;
    }

    increment();
  }
};

foo.incrementA();
foo.incrementA();
foo.incrementA();
```

▼ Answer:

The value of `foo.a` will be `0`. Since `increment` gets invoked as a function, `this.a` on line 5 references a property of the global object rather than a property of `foo`. Thus, the property `foo.a` isn't modified by the `increment`; its value remains 0.

9. Use one of the methods we learned in this lesson to invoke `increment` with an explicit context such that `foo.a` gets incremented with each invocation of `incrementA`.

▼ Answer:

```
let foo = {
  a: 0,
  incrementA: function() {
    function increment() {
      this.a += 1;
    }

    increment.call(this);
  }
};

foo.incrementA();
```

```
foo.incrementA();
foo.incrementA();
```