Interview Cake

We're building a web game where everybody wins and we are all friends forever.

It's simple—you click on one of three boxes to see what nice thing you've won. You always win something nice. Because we love you.

Here's what we have so far. Something's going wrong though. Can you tell what it is?

```html
<button id="btn-0">Button 1</button>
<button id="btn-1">Button 2</button>
<button id="btn-2">Button 3</button>

<script type="text/javascript">
  const prizes = ['A Unicorn!', 'A Hug!', 'Fresh Laundry!'];
  for (var btnNum = 0; btnNum < prizes.length; btnNum++) {

    // For each of our buttons, when the user clicks it...
    document.getElementById(`btn-${btnNum}`).onclick = () => {

      // Tell her what she's won!
      alert(prizes[btnNum]);
    };
  }
</script>
```

The syntax is just fine—the problem is some unexpected *behavior*.

Gotchas

Coding style choices aside, what we found is a problem in *behavior*.

Solution

The user's prize is *always* undefined!

The Problem

The anonymous function we're assigning to the buttons' `onclicks` has access to variables in the scope outside of it (this is called a closure⤵

A closure is a function that accesses a variable "outside" itself. For example:

```javascript
const message = 'The British are coming.';
function sayMessage(){
  alert(message); // Here we have access to message,
  // even though it's declared outside this function!
}
```

We'd say that `message` is "closed over" by `sayMessage()`.

One useful thing to do with a closure is to create something like an "instance variable" that can change over time and can affect the behavior of a function.

```javascript
// Function for getting the id of a dom element,
// giving it a new, unique id if it doesn't have an id yet
const getUniqueId = (() => {
  let nextGeneratedId = 0;
  return element => {
    if (!element.id) {
      element.id = `generated-uid-${nextGeneratedId}`;
      nextGeneratedId++;
    }
    return element.id;
  };
})();
```

Why did we put `nextGeneratedId` in an immediately-executed anonymous function? It makes `nextGeneratedId` private, which prevents accidental changes from the outside world:

JavaScript

```
// Function for getting the id of a dom element,
// giving it a new, unique id if it doesn't have an id yet
let nextGeneratedId = 0;
const getUniqueId = element => {
  if (!element.id) {
    element.id = `generated-uid-${nextGeneratedId}`;
    nextGeneratedId++;
  }
  return element.id;
};

// ...
// Somewhere else in the codebase...
// ...

// WHOOPS--FORGOT I WAS ALREADY USING THIS FOR SOMETHING
nextGeneratedId = 0;
```

). In this case, it has access to `btnNum`.

When a function accesses a variable outside its scope, it accesses *that variable*, not a frozen copy. So when the value held by the variable changes, the function gets that new value. By the time the user starts pressing buttons, our loop will have already completed and `btnNum` will be 3, so this is what each of our anonymous functions will get for `btnNum`!

Why 3? The for loop will increment `btnNum` until the conditional in the middle is no longer met—that is, until it's not true that `btnNum` < `prizes.length`. So the code in the for loop won't run with `btnNum` = 3, but `btnNum` will be 3 when the loop is done.

Why undefined? `prizes` has 3 elements, but they are at indices 0,1,2. Array indices start at 0, remember? (Write this down—forgetting this is an easy way to create an off-by-one error in a whiteboard interview.) In JavaScript, accessing a nonexistent index in an array returns `undefined` (Python throws an `IndexError`, but Ruby returns `nil`).

The Solution

We can solve this by wrapping our anonymous function in *another anonymous function* that takes `btnNum` as an argument. Like so:

HTML

```
<button id="btn-0">Button 1!</button>
<button id="btn-1">Button 2!</button>
<button id="btn-2">Button 3!</button>

<script type="text/javascript">
  const prizes = ['A Unicorn!', 'A Hug!', 'Fresh Laundry!'];
  for (var btnNum = 0; btnNum < prizes.length; btnNum++) {

    // For each of our buttons, when the user clicks it...
    document.getElementById(`btn-${btnNum}`).onclick = (frozenBtnNum => {
      return () => {

        // Tell her what she's won!
        alert(prizes[frozenBtnNum]);
      };
    })(btnNum); // LOOK! We're passing btnNum to our anonymous function here!
  }
</script>
```

This "freezes" the value of `btnNum`. Why? Well...

Primitives vs. Objects

`btnNum` is a number, which is a **primitive** type in JavaScript.

Primitives are "simple" data types (string, number, boolean, null, and undefined in JavaScript). Everything else is an *object* in JavaScript (functions, arrays, Date() values, etc).

Arguments Passed by Value vs. Arguments Passed by Reference

One important property of primitives in JS is that when they are passed as arguments to a function, they are *copied* ("passed by value"). So for example:

> Heads up: This is *not* well-formed JavaScript. We're using it to prove a point.

JavaScript

```
let threatLevel = 1;

function inspireFear(threatLevel){
  threatLevel += 100;
}

inspireFear(threatLevel);
console.log(threatLevel); // Whoops! It's still 1!
```

The `threatLevel` inside `inspireFear()` is a *new* number, initialized to the same *value* as the `threatLevel` outside of `inspireFear()`. Giving these *different* variables the same name might cause confusion here. If we change the two variables to have different names we get the exact same behavior:

JavaScript

```
let threatLevel = 1;

function inspireFear(theThreatLevel){
  theThreatLevel += 100;
}

inspireFear(threatLevel);
console.log(threatLevel); // Whoops! It's still 1!
```

In contrast, **when a function takes an object, it actually takes a *reference* to *that very object***. So changes you make to the object in the function persist after the function is done running. This is sometimes called a **side effect**.

JavaScript

```
const scaryThings = ['spiders', 'Cruella de Vil'];

function inspireFear(scaryThings){
  scaryThings.push('nobody ever using Interview Cake');
  scaryThings.push('i should have gotten a real job');
  scaryThings.push('why am i doing this to myself');
}

inspireFear(scaryThings);
console.log(scaryThings);
// ['spiders', 'Cruella de Vil', 'nobody ever using Interview Cake', 'i should have gotten
```

Bringing it home

Back to our solution:

HTML

```
<button id="btn-0">Button 1!</button>
<button id="btn-1">Button 2!</button>
<button id="btn-2">Button 3!</button>

<script type="text/javascript">
  const prizes = ['A Unicorn!', 'A Hug!', 'Fresh Laundry!'];
  for (var btnNum = 0; btnNum < prizes.length; btnNum++) {

    // For each of our buttons, when the user clicks it...
    document.getElementById(`btn-${btnNum}`).onclick = (frozenBtnNum => {
      return () => {

        // Tell her what she's won!
        alert(prizes[frozenBtnNum]);
      };
    })(btnNum);
  }
</script>
```

So when we pass `btnNum` to the outer anonymous function as its one argument, we create a *new* number inside the outer anonymous function called `frozenBtnNum` that has the value that `btnNum` had *at that moment* (0, 1, or 2).

Our inner anonymous function is still a closure because it still reaches outside its scope, but now it closes over this *new* number called `frozenBtnNum`, whose value will not change as we iterate through our for loop.

What We Learned

Like several common Javascript interview questions, this question hinges on a solid understanding of closures and *pass by reference* vs *pass by value*. If you're shaky on either of those, look back at the examples in the solution.