Solutions to Quiz 2 (September 22)

Problem 1 (Prototypes) (1 point).
Here’s a short JavaScript program that creates two objects with explicit prototypes and then calls methods on those objects.

```javascript
var proto = {
    speak: function() {
        console.log("oof.");
    }
};

var obj1 = Object.create(proto);
obj1.speak = function() {
    console.log("WOOF!");
};

var obj2 = Object.create(proto);
proto.speak = function() {
    console.log("MEOW!");
}

obj1.speak();
obj2.speak();
```

What are the two lines of text, in order, that this program outputs to the console?

A. MEOW!, oof.   B. WOOF!, oof.   C. WOOF!, MEOW!   D. MEOW!, MEOW!

Solution. The answer is C. JavaScript objects are mutable, and the prototype object P of one object O may be mutated after O is created, with the consequences still applying to O. As a result, obj1’s “overriding” method prevails (we don’t need to walk further down the prototype chain when a field is found immediately), and obj2 picks up the behavior from its prototype proto.

Problem 2 (Concurrency) (1 point).
Consider an HTML document with two different <button> tags, one with id button1 and the other with id button2. Upon loading the document, we run the following JavaScript/jQuery code to install a click event handler on each button.

```javascript
$('#button1')
    .click(function() {
        console.log("button1 clicked.");
        console.log("button1 really clicked.");
    });
```
Imagine the user is going to click each button exactly once. However, the buttons may be clicked in either order, and the delay between presses may be arbitrarily long. How many different sequences of 4 output lines could be generated?

A. 2   B. 3   C. 4   D. 5

Solution. The answer is B, corresponding to these three log sequences:

button1 clicked.
button1 really clicked.
button2 clicked.
Happy 1-second anniversary, button2!

button2 clicked.
Happy 1-second anniversary, button2!
button1 clicked.
button1 really clicked.

button2 clicked.
button1 clicked.
button1 really clicked.
Happy 1-second anniversary, button2!

A few potentially tricky elements: Remember that, in browser JavaScript, each event handler gets to run in full, with no interruption by other threads, so the two entries for button1 must always occur together. Also, the setTimeout callback operation for button2 must always wait to run until the main button2 event handler finishes. Thus, the choices come from picking an ordering of the two clicks, plus deciding if button1 could be clicked long enough after button2 to give the setTimeout callback a chance to run.

Problem 3 (Encapsulation) (3 points).
Your startup company sells cutting-edge JavaScript counters to embed in web pages. You provide a constructor function Counter, which is used like so:

```javascript
var counter = Counter();

function clientCode() {
    counter.increment();
    counter.increment();
    console.log(counter.get());
```
Your business model is to charge $1 per call to increment, so each counter reports the number of increments on the console, one day after being created. Each of the 3 code snippets below gives a candidate definition of Counter, using setTimeout to queue up the code to log the current counter value, one day later. Which of the snippets are secure, in the sense that the number printed in the setTimeout callback must always be nonnegative, regardless of what the client code does? (Fair warning: one of these implementations contains an especially silly coding mistake that endangers security!)

**Suggestion 1.**
Secure or Insecure?

```javascript
function Counter() {
  var that = {
    value: 0,
    increment: function() {
      this.value += 1;
    },
    get: function() {
      return this.value;
    }
  };

  setTimeout(function() {
    console.log("Counter value: " + that.value);
  }, MILLISECONDS_PER_DAY);

  return that;
}
```

**Suggestion 2.**
Secure or Insecure?

```javascript
function Counter() {
  value = 0;

  var that = {
    increment: function() {
      value += 1;
    },
    get: function() {
      return value;
    }
  };

  setTimeout(function() {
    console.log("Counter value: " + value);
  }, MILLISECONDS_PER_DAY);

  return that;
}
```
Suggestion 3.
Secure or Insecure?

```javascript
function Cell(value) {
    return {
        set: function(v) {
            value = v;
        },
        get: function() {
            return value;
        }
    };
}

function Counter() {
    var cell = Cell(0);
    var that = {
        increment: function() {
            cell.set(cell.get() + 1);
        },
        get: cell.get
    };
    setTimeout(function() {
        console.log("Counter value: " + cell.get());
    }, MILLISECONDS_PER_DAY);
    return that;
}
```

Solution. The answers are I, I, and S.
The first suggestion allows the client code to write the value field directly, via counter.value.
The second suggestion is almost correctly using the good JavaScript pattern of object encapsulation via closures. However, we forgot the var keyword to make value local to the constructor function! By default, it becomes a new global variable, which client code may certainly set directly.
The last suggestion is finally secure. Both Cell() and Counter() confine their private state to local variables.

■