JS

CODE KATA

```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
alert(x);
```

What is x?

```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
alert(x);
```

What is x?

a. null
b. []
c. undefined
d. throw

```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
alert(x);
```

What is x?

a. null
b. []
c. undefined
d. throw

```
function swap(a, b) {
    var temp = a;
    a = b;
    b = temp
}
var x = 1, y = 2;
swap(x, y);
alert(x);
```

A.1
B.2
C.undefined
D.throw

What is x?

```
function swap(a, b) {
    var temp = a;
    a = b;
    b = temp
}
var x = 1, y = 2;
swap(x, y);
alert(x);
```

A.1
B.2
C.undefined
D.throw

A

What is x?

Write two binary functions,
`add` and `mul`, that take two
numbers and return their sum
and product.

```
add(3, 4)     //  7
mul(3, 4)     // 12
```

```
function add(x, y) {
    return x + y;
}
```

```
function mul(x, y) {
    return x * y;
}
```

Write a function that takes an argument and returns a function that returns that argument.

```
idf = identityf(3);
idf()     // 3
```

```
function identityf(x) {
    return function () {
        return x;
    };
}
```

Write a function that adds
from two invocations.

```
addf(3)(4)     // 7
```

```
function addf(x) {
    return function (y) {
        return x + y;
    };
}
```

Write a function that takes a binary function, and makes it callable with two invocations.

```
addf = applyf(add);
addf(3)(4)                  // 7
applyf(mul)(5)(6)           // 30
```

```
function applyf(binary) {
    return function (x) {
        return function (y) {
            return binary(x, y);
        };
    }
```

Write a function that takes a function and an argument, and returns a function that can supply a second argument.

```
add3 = curry(add, 3);
add3(4)                  // 7

curry(mul, 5)(6)         // 30
```

```
function curry(func, first) {
    return function (second) {
        return func(first, second);
    };
}
```

```
function curry(func, first) {
    return applyf(func)(first);
}
```

```
function curry(func) {
    var slice = Array.prototype.slice,
        args = slice.call(arguments, 1);
    return function () {
        return func.apply(
            null,
            args.concat(slice.call(arguments, 0))
        );
    };
}
```

Without writing any new functions, show three ways to create the `inc` function.

```
inc(5)                  // 6
inc(inc(5))             // 7
```

```
1. inc = addf(1);

2. inc = applyf(add)(1);

3. inc = curry(add, 1);
```

Write `methodize`, a function
that converts a binary function
to a method.

```
Number.prototype.add =
    methodize(add);
(3).add(4)     // 7
```

```
function methodize(func) {
    return function (y) {
        return func(this, y);
    };
}
```

```
function methodize(func) {
    return function (...y) {
        return func(this, ...y);
    };
}
```

Write `demethodize`, a function that converts a method to a binary function.

```
demethodize(Number.prototype.add)(5, 6)
    // 11
```

```
function demethodize(func) {
    return function (that, y) {
        return func.call(that, y);
    };
}
```

```
function demethodize(func) {
    return function (that, ...y) {
        return func.apply(that, y);
    };
}
```

Write a function `twice` that takes a binary function and returns a unary function that passes its argument to the binary function twice.

```
var double = twice(add);
double(11)    // 22
var square = twice(mul);
square(11)    // 121
```

```
function twice(binary) {
    return function (a) {
        return binary(a, a);
    };
}
```

Write a function composeu
that takes two unary functions
and returns a unary function
that calls them both.

```
composeu(double, square)(3)    // 36
```

```
function composeu(f, g) {
    return function (a) {
        return g(f(a));
    };
}
```

Write a function composeb
that takes two binary functions
and returns a function that
calls them both.

```
composeb(add, mul)(2, 3, 5)     // 25
```

```
function composeb(f, g) {
    return function (a, b, c) {
        return g(f(a, b), c);
    };
}
```

Write a function that allows another function to only be called once.

```
add_once = once(add);
add_once(3, 4)     // 7
add_once(3, 4)     // throw!
```

```
function once(func) {
    return function () {
        var f = func;
        func = null;
        return f.apply(
            this,
            arguments
        );
    };
}
```

Write a factory function that returns two functions that implement an up/down counter.

```
counter = counterf(10);
counter.inc()    // 11
counter.dec()    // 10
```

```
function counterf(value) {
    return {
        inc: function () {
            value += 1;
            return value;
        },
        dec: function () {
            value -= 1;
            return value;
        }
    };
}
```

Make a revocable function that takes a nice function, and returns a `revoke` function that denies access to the nice function, and an `invoke` function that can invoke the nice function until it is revoked.

```
temp = revocable(alert);
temp.invoke(7);     // alert: 7
temp.revoke();
temp.invoke(8);     // throw!
```

```
function revocable(nice) {
    return {
        invoke: function () {
            return nice.apply(
                this,
                arguments
            );
        },
        revoke: function () {
            nice = null;
        }
    };
}
```

```
function ident (arg) {
  return function () {
    return arg;
  };
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```

Create a function that takes an arg and returns a function that returns that arg.

Given:

```
function add(a, b) {
  return a + b;
}
```

```
function mul(a, b) {
  return a * b;
}
```