JAVASCRIPT QUIRKS
QUIZ #2

#1

```
function foo1()
{
  return {
     bar: "hello"
  };
}

function foo2()
{
  return
  {
     bar: "hello"
  };
}
````

console.log(foo1());
console.log(foo2());
```

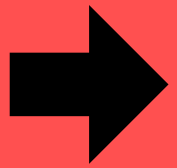
1) What will be the output of the above code snippet? *

- Object {bar: "hello"} / Object {bar: "hello"}
- undefined / Object {bar: "hello"}
- Object {bar: "hello"} / undefined
- undefined / undefined
```

```
function foo1()
{
  return {
     bar: "hello"
  };
}

function foo2()
{
  return
  {
     bar: "hello"
  };
}
````

console.log(foo1());
console.log(foo2());
```



returns Object {bar: "hello"}

returns undefined

- Semicolons are technically optional in JavaScript.
- As a result, when the line containing the return statement (with nothing else on the line) is encountered in foo2(), a semicolon is automatically inserted immediately after the return statement.
```

#2

```
typeof NaN
```

2) What will be the output of the above code snippet? *

- "undefined"
- "object"
- undefined
- "number"

typeof NaN === "number"

- NaN is a property of the global object
- It is the returned value when Math functions fail
  - (Math.sqrt(-1) or x/0)
  - or when a function trying to parse a number fails (parseInt("blabla")).
  - or when one of the operands is non-numeric ("abc"/4)
- NaN compared to anything – even itself! – is false
- Testing if value is equal to NaN
  - isNaN() NOT reliable
  - Number.isNaN() as of ES6 - Reliable
  - value !== value Reliable

#3

```
console.log(0.1 + 0.2);
console.log(0.1 + 0.2 == 0.3);
```

3) What will be the output of the above code snippet? *

- 0.3 / true
- 0.30000000000000004 / true
- 0.3 / false
- 0.30000000000000004 / false

0.30000000000000004 / FALSE

- A Number is a floating point number with a limited precision of 64 bits, about 16 digits.
- Because of the limited precision of floating point numbers round-off errors can occur during calculations.
- 9999999999999999 === 10000000000000000 Is true also

#4

```
(function() {
    console.log(1);
    setTimeout(function(){console.log(2)}, 1000);
    setTimeout(function(){console.log(3)}, 0);
    console.log(4);
})();
```

4) What will be the output of the above code snippet? *

- 1, 2, 3, 4
- 1, 3, 4, 2
- 1, 4, 2, 3
- 1, 4, 3, 2

1 4 3 2

```
(function() {
   console.log(1);
   setTimeout(function(){console.log(2)}, 1000);
   setTimeout(function(){console.log(3)}, 0);
   console.log(4);
})();
```

- When a value of zero is passed as the second argument to setTimeout(), it attempts to execute the specified function “as soon as possible”. Specifically, execution of the function is placed on the event queue to occur on the next timer tick. Note, though, that this is not immediate; the function is not executed until the next tick.
- The delay is the *minimum* time required for the runtime to process the request, but not a guaranteed time.

#5

```
for (var i = 0; i < 5; i++) {
  setTimeout(function() { console.log(i); }, i * 1000 );
}
```

5) What will be the output of the above code snippet? *

- ○ 0, 1, 2, 3, 4
- ○ 0, 0, 0, 0, 0
- ○ 4, 4, 4, 4, 4
- ○ 5, 5, 5, 5

5 5 5 5 5

```
for (var i = 0; i < 5; i++) {
  setTimeout(function() { console.log(i); }, i * 1000 );
}
```

- 'var' doesn't have block-level scope, so it gets declared and updates the global scope
- We're simply passing the reference to the variable i, and not the actual value at the moment inside each loop.
- Therefore, by the time the setTimeout() function is executed (after 0, 1, 2, 3, and 4 seconds in this case), the for statement has already been executed and incremented i to the final value

CLOSING OVER THE INDEX VARIABLE

```
for (let i = 0; i < 5; i++) {
    (function() { setTimeout(() => console.log(i), i * 1000)}());
}

// or

for (let i = 0; i < 5; i++) {
    void function() { setTimeout(() => console.log(i), i * 1000)}();
}

// or

for (let i = 0; i < 5; i++) {
    ~~ function() { setTimeout(() => console.log(i), i * 1000)}();
}
```

- We are creating an iife (immediatley invoked function expression) that closes over the index value at each iteration to make this work correctly.
- Parens are the normal way to run an iife, but as long as there is something that will instantiate the call, it will run.

#6

```
var hero = {
    _name: "John Doe",
    getSecretIdentity: function() {
        return this._name;
    }
};

var stoleSecretIdentity = hero.getSecretIdentity;

console.log(stoleSecretIdentity());
console.log(hero.getSecretIdentity());
```

6) What will be the output of the above code snippet? *

- ○ "John Doe" / "John Doe"
- ○ undefined / undefined
- ○ "John Doe" / undefined
- ○ undefined / "John Doe"

UNDEFINED / 'JOHN DOE'

```
var hero = {
    _name: "John Doe",
    getSecretIdentity: function() {
        return this._name;
    }
};

var stoleSecretIdentity = hero.getSecretIdentity;

console.log(stoleSecretIdentity());
console.log(hero.getSecretIdentity());
```

- first console.log: 'this' is the window object
- second console.log: 'this' is the hero object

#7

```
(function(x) {
    return (function(y) {
        console.log(x);
    })(2)
})(1);
```

7) What will be the output of the above code snippet? *

- 1
- 2
- undefined
- Error

ANSWER IS 1

```
(function(x) {
    return (function(y) {
        console.log(x);
    })(2)
})(1);
```

- x is not defined in the inner function, the scope of the outer function is searched for a defined variable x, which is found to have a value of 1

#8

```
console.log((function f(n){return ((n > 1) ? n * f(n-1) : n)})(10));
```

8) What will be the output of the above code snippet? *

- undefined
- 3628800
- Error
- 0

3628800

- 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2

```
f(1): returns n, which is 1
f(2): returns 2 * f(1), which is 2
f(3): returns 3 * f(2), which is 6
f(4): returns 4 * f(3), which is 24
f(5): returns 5 * f(4), which is 120
f(6): returns 6 * f(5), which is 720
f(7): returns 7 * f(6), which is 5040
f(8): returns 8 * f(7), which is 40320
f(9): returns 9 * f(8), which is 362880
f(10): returns 10 * f(9), which is 3628800
```

#9

```
console.log(false == '0');
console.log(false === '0');
```

9) What will be the output of the above code snippet? *

- ○ true / true
- ○ true / false
- ○ true / false
- ○ false / false

TRUE / FALSE

```
console.log(false == '0');
console.log(false === '0');
```

- false & 0 have the same value
- but are differents types of values

#10

```
function test() {
  console.log(a);
  console.log(foo());

  var a = 1;
  function foo() {
    return 2;
  }
}

test();
```

10) What will be the output of the above code snippet? *

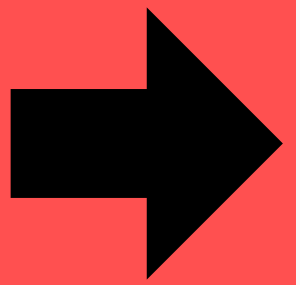
- ○ undefined / undefined
- ○ 1 / 2
- ○ 1 / undefined
- ○ undefined / 2

UNDEFINED / 2

```
function test() {
    console.log(a);
    console.log(foo());

    var a = 1;
    function foo() {
        return 2;
    }
}

test();
```



```
function test() {
    var a;
    function foo() {
        return 2;
    }

    console.log(a);
    console.log(foo());

    a = 1;
}

test();
```