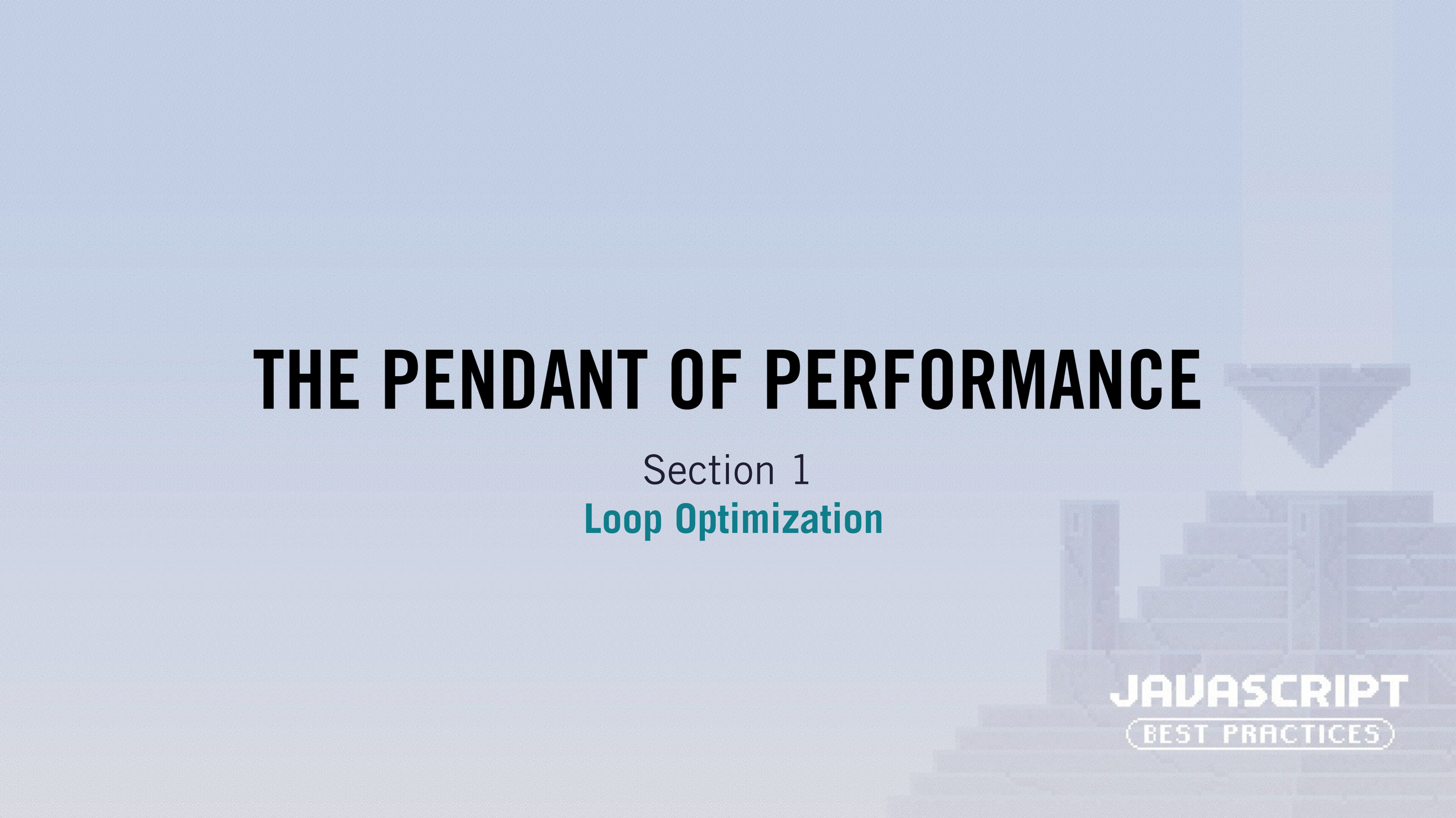
THE PENDANT OF PERFORMANCE

Section 1

Loop Optimization



A COMMON FOR-LOOP SCENARIO

Memory access is not necessarily ideal in our normal for-loop structure.

```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

→ You've found the following necklaces:
→ ruby
→ pearl
→ sapphire
→ diamond

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

A COMMON FOR-LOOP SCENARIO

Memory access is not necessarily ideal in our normal for-loop structure.

```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
                        length: 4
openLid: function () {
            alert("Creeeeak!");
         }
```

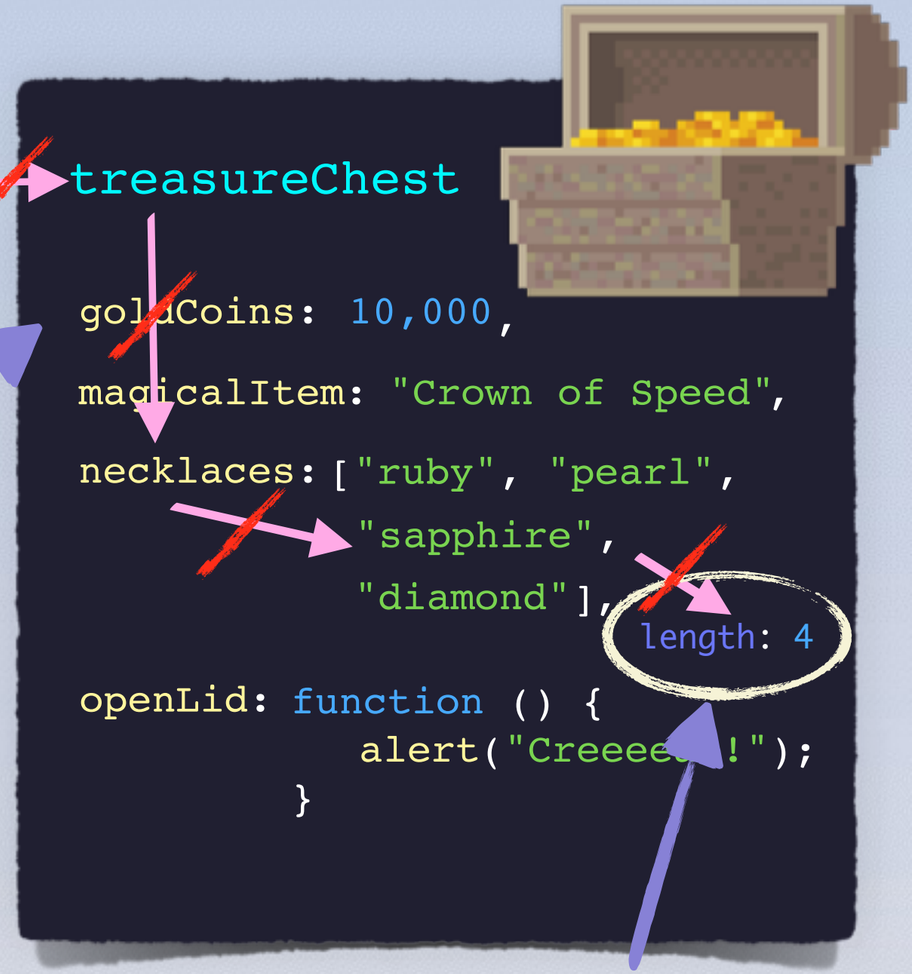
At the start of each potential loop cycle, the program will need to find and retrieve:

1. the value of `i`
2. the `treasureChest` object
3. the `necklaces` property
4. the array pointed to by the property
5. the `length` property of the array

WHAT STEPS CAN WE ELIMINATE?

Any intermediary step to the bit of data we need is a candidate for death.

```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
   console.log(treasureChest.necklaces[i]);
}
```



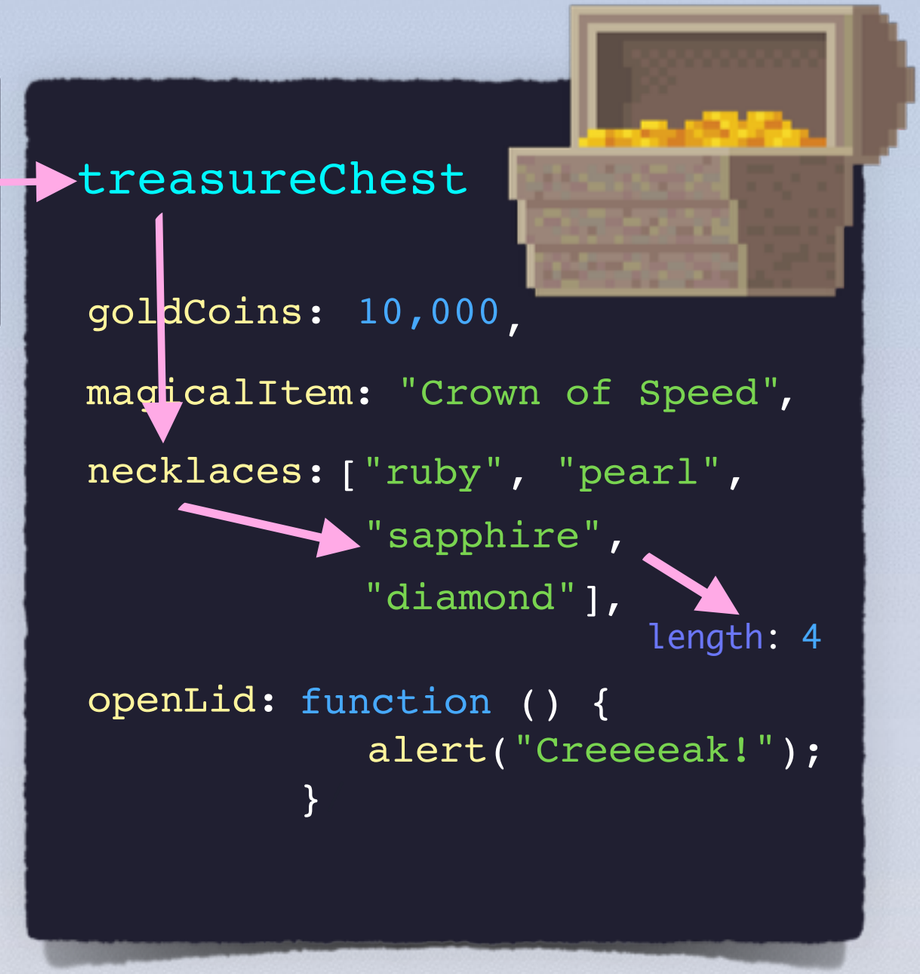
...so we don't really need all these other steps.

All we are interested in during loop control is this single value and the loop counter.

LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

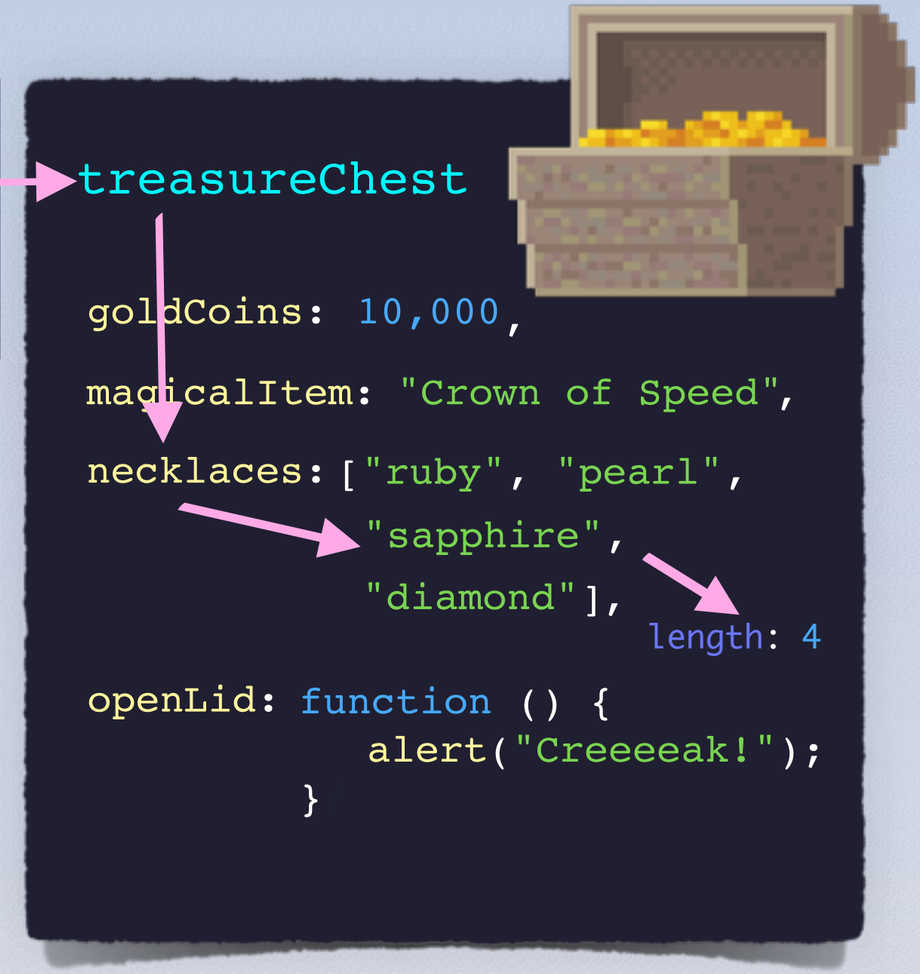
```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
   console.log(treasureChest.necklaces[i]);
}
```



LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x =
for(var i = 0; i < treasureChest.necklaces.length; i++) {
   console.log(treasureChest.necklaces[i]);
}
```



LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i <                                    i++) {
   console.log(treasureChest.necklaces[i]);
}
```

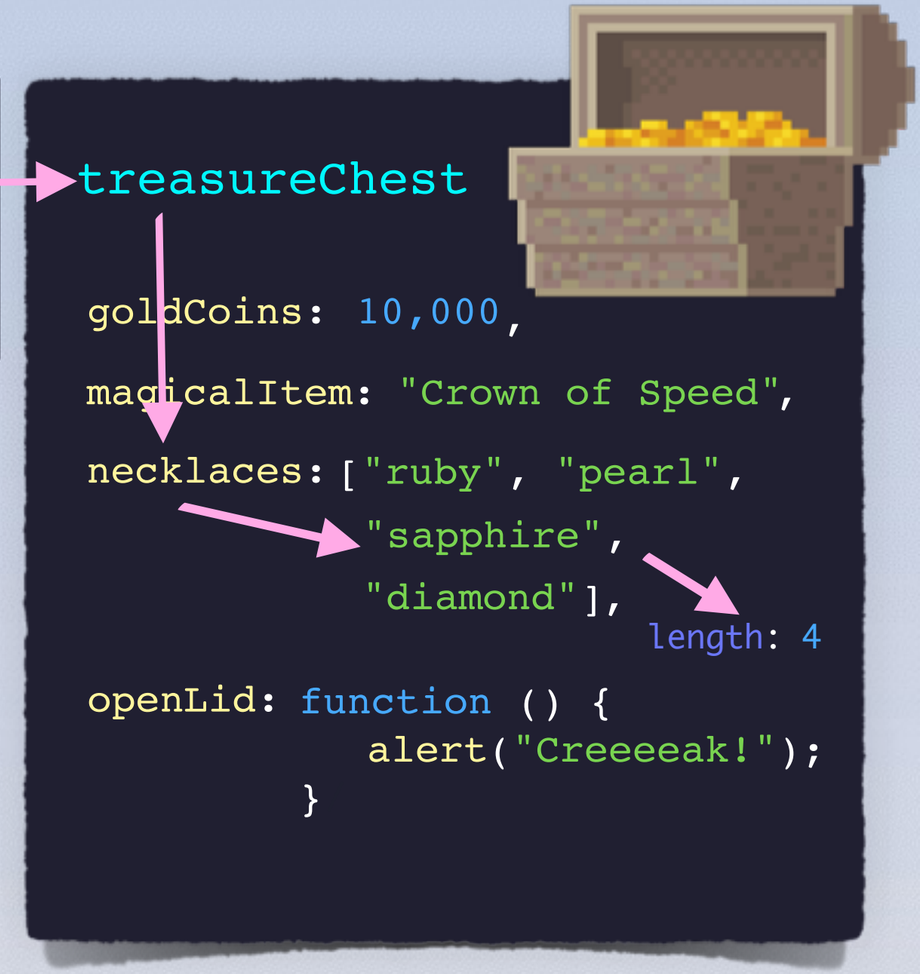
```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
                        length: 4
openLid: function () {
            alert("Creeeeak!");
         }
```

First, we assign the length's value to a new variable. This will use all of the accesses as before, but we'll only do it once.

LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

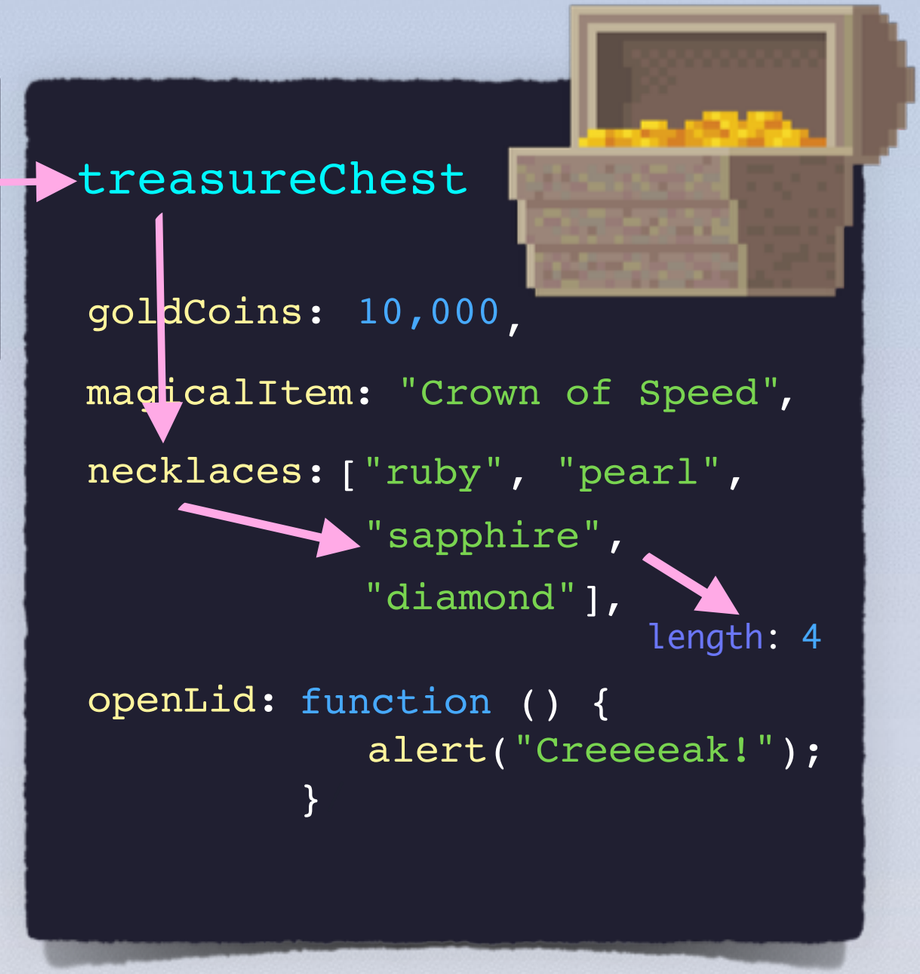
```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i <                                    i++) {
   console.log(treasureChest.necklaces[i]);
}
```



LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x;                                    i++) {
   console.log(treasureChest.necklaces[i]);
}
```

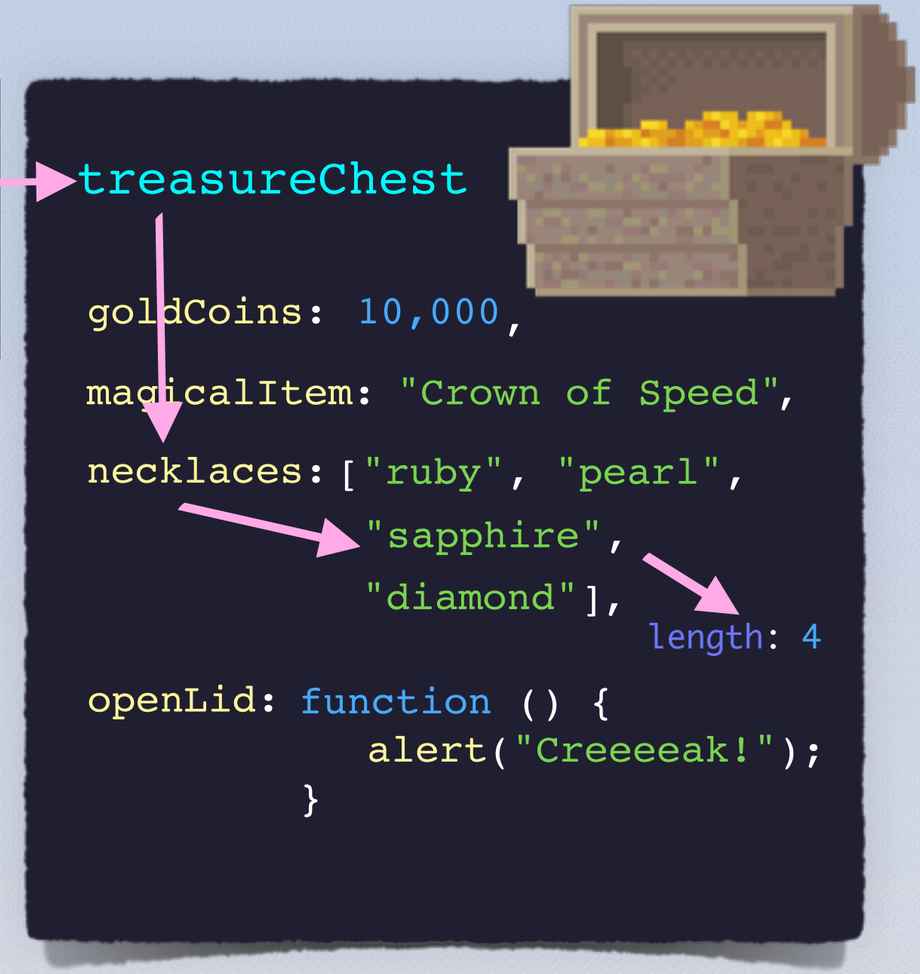


LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

Now we'll let our loop use the local `x` during control. The majority of extraneous steps have now been eliminated.



PROCESSOR SAVINGS MEANS MORE SPEED

Let's check out what we've saved on our relatively simple example.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

Memory access during loop control now only needs to:

1. retrieve the value of `i`
2. retrieve the value of `x`

Then, add in our one-time cost in creating `x`:

1. creating the variable `x` in memory.

2-5. the 4 steps finding the value of `length`

PROCESSOR SAVINGS MEANS MORE SPEED

Let's check out what we've saved on our relatively simple example.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,

magicalItem: "Crown of Speed",

necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

2 steps x (4 executed loops + 1 check to stop)

= 10 steps for the processor, just in memory access.

\+ 5 steps in the creation of the extra variable.

= 15 steps of strictly memory access to execute the loop

PROCESSOR SAVINGS MEANS MORE SPEED

Let’s check out what we’ve saved on our relatively simple example.

```
console.log("You’ve found the following necklaces:”);
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

15 steps of strictly memory access to execute the loop

PROCESSOR SAVINGS MEANS MORE SPEED

Let's check out what we've saved on our relatively simple example.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,

magicalItem: "Crown of Speed",

necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

25 steps in our previous version

\- 15 steps of strictly memory access to execute the loop

10 steps of savings

WHAT IF WE HAD 10,000 NECKLACES?

With large data amounts, this improvement produces really noticeable results.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

5 steps x (10,000 executed loops + 1 check to stop)

= 50,005 memory access steps

---

2 steps x (10,000 executed loops + 1 check to stop)

= 20,002 memory access steps

\+ 5 extra steps for x

20,007 memory access steps

WHAT IF WE HAD 10,000 NECKLACES?

With large data amounts, this improvement produces really noticeable results.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

50,005 memory access steps
- 20,002 memory access steps

~30,000 steps of savings!

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],

openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0,                                    i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

Surprise! A comma will allow us to execute multiple statements within the first parameter of the loop.

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0, var x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");

for(var i = 0, var x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0, var x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

This `var` is now unnecessary, because the comma tells the compiler that more variables will be declared ... and maybe even assigned.

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0,      x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

By creating both variables in the loop parameters, we signal they're only intended for use inside the loop.

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
console.log(x);
```

4

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

Beware! JavaScript does not scope to blocks, so any variables declared in loops will be available after the loop, and may overwrite pre-existing globals!

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

Here's another access of a property within an object.

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(treasureChest.necklaces[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(                    list[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(list[i]);
}
```

Now we've avoided the extra step of accessing the `treasureChest` in each cycle.

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(list[i]);
}
```

This function would now be available to all Arrays in our program.

```
Array.prototype.countType = function (type) {
    var count = 0;
    for(var i = 0, x = this.length; i<x; i++){
       if(this[i] === type){
         count++;
       }
    }
    return count;
};
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(list[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

```
Array.prototype.countType = function (type) {




};
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(list[i]);
}
```

```
Array.prototype.removeAll = function (item) {
   var removed = [];
   for(var i = 0, x = this.length; i<x; i++){
      if(this[i] === item){
        removed.push(item);
        this.splice(i, 1);
      }
   }
   return removed;
};
```

```
Array.prototype.countType = function (type) {
   ...
};
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(list[i]);
}
```

```
Array.prototype.removeAll = function (item) {




};
```

```
Array.prototype.countType = function (type) {
   ...
};
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(list[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

```
Array.prototype.removeAll = function (item) {
   ...
};
```

```
Array.prototype.countType = function (type) {
   ...
};
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
   console.log(list[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

```
Array.prototype.removeAll = function (item) {
   ...
};
```

```
Array.prototype.countType = function (type) {
   ...
};
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(p in list                                          ) {
   console.log(list[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

```
Array.prototype.removeAll = function (item) {
   ...
};
```

```
Array.prototype.countType = function (type) {
   ...
};
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(p in list) {
   console.log(list[i]);
}
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

```
Array.prototype.removeAll = function (item) {
   ...
};
```

```
Array.prototype.countType = function (type) {
   ...
};
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(p in list) {
   console.log(list[i]);
}
```

```
You've found the following necklaces:
ruby
pearl
sapphire
diamond
removeAll
countType
```

Using a property approach to access indices will also add in ALL methods that have been added to the Array prototype.

Methods we add to the prototype become "enumerable" just like indices.

```
Array.prototype.removeAll = function (item) {
   ...
};
```

```
Array.prototype.countType = function (type) {
   ...
};
```

```
treasureChest

goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
            alert("Creeeeak!");
         }
```

PERFORMANCE

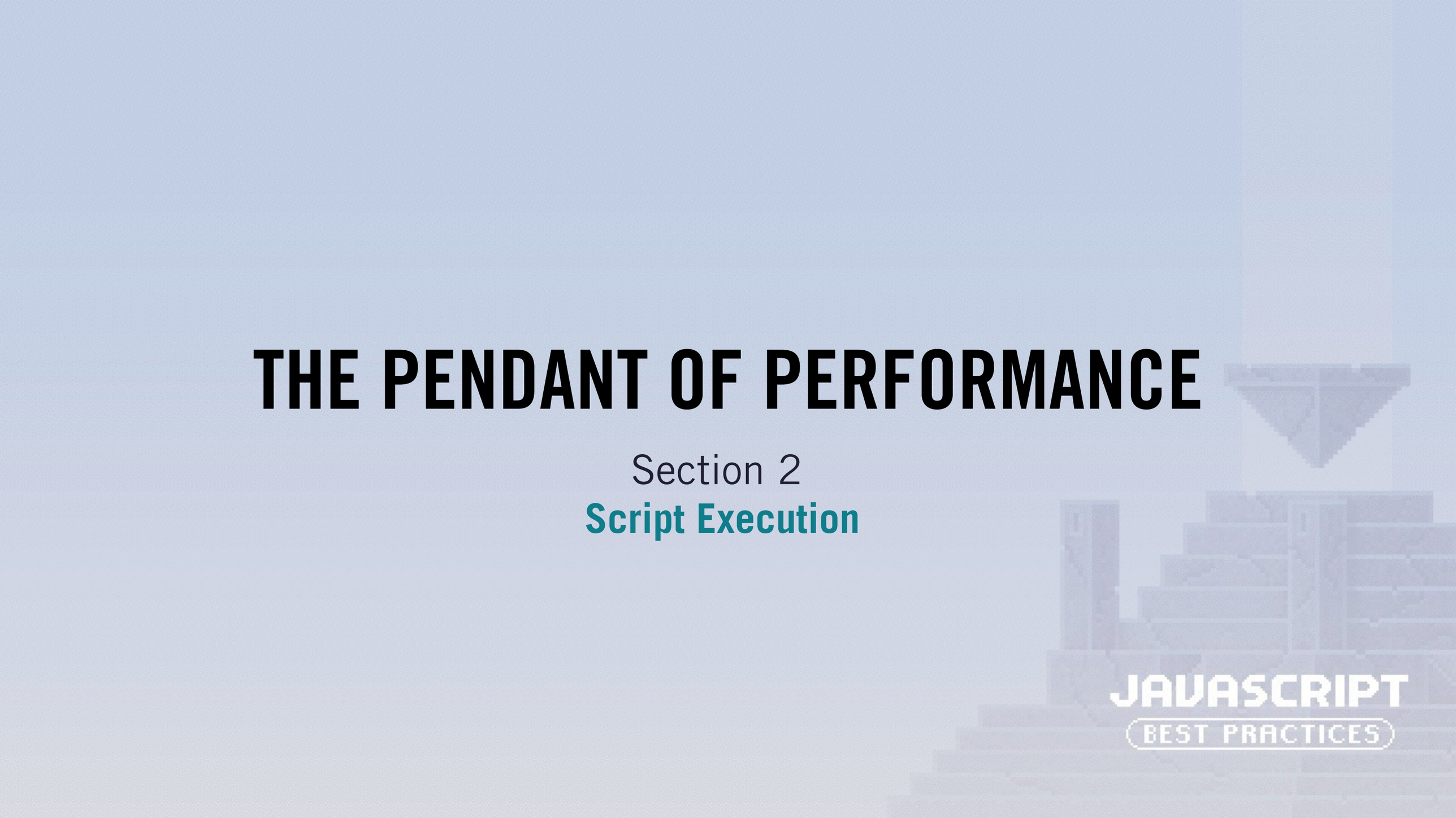
Section 1

Loop Optimization

THE PENDANT OF PERFORMANCE

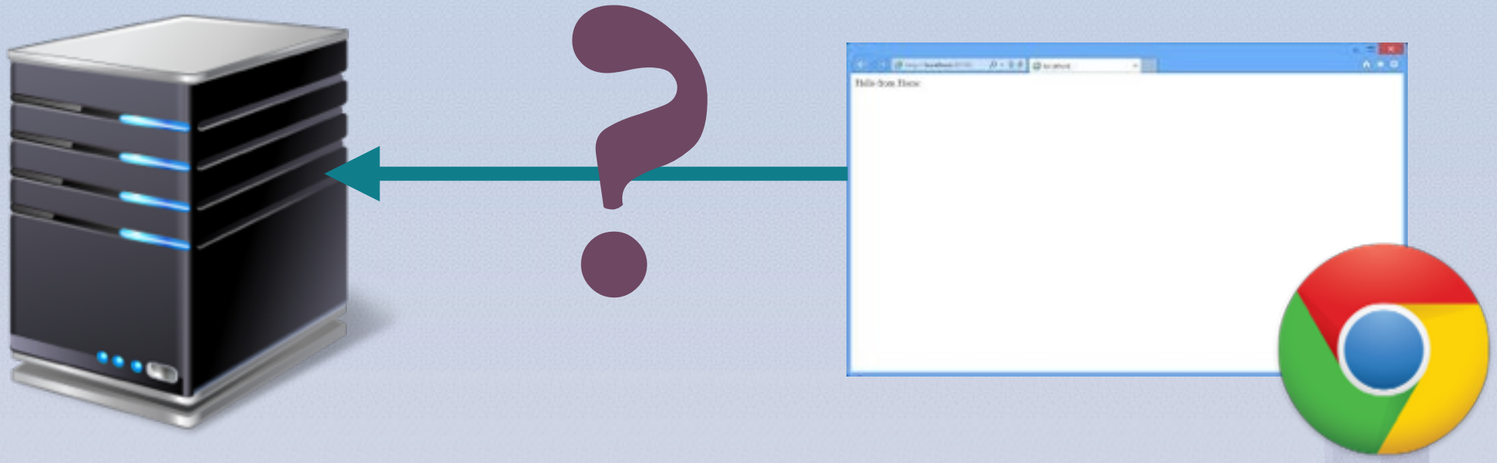
Section 2

Script Execution



SCRIPT PLACEMENT IMPACTS PERFORMANCE

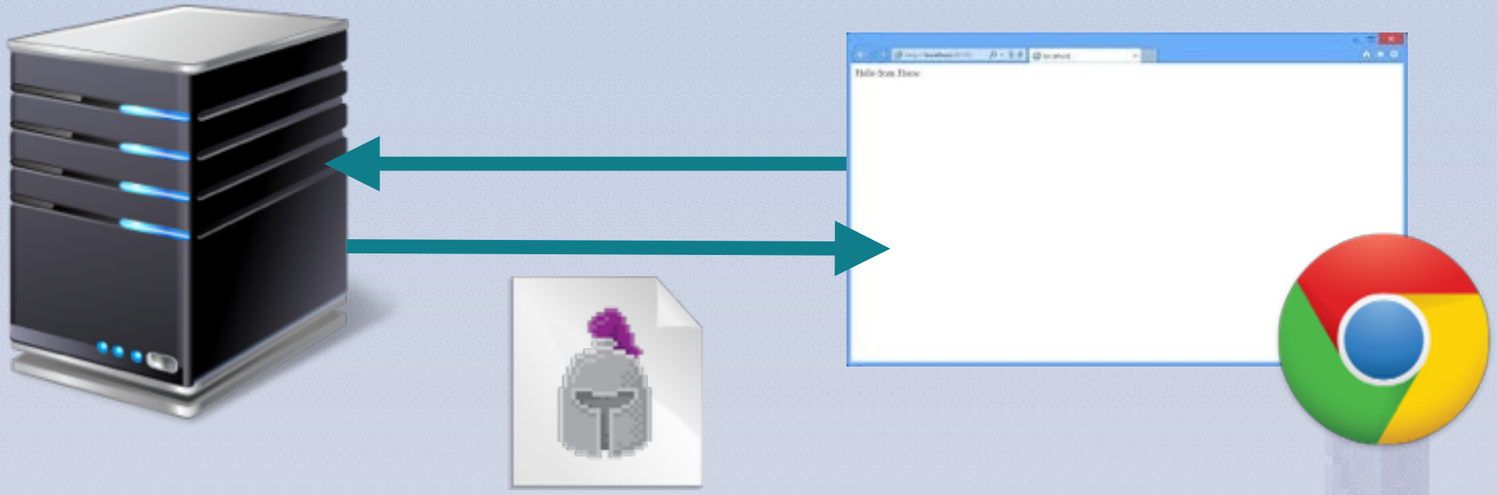
Let’s take a closer look at how a browser retrieves and acts on scripts.



JAVASCRIPT
BEST PRACTICES

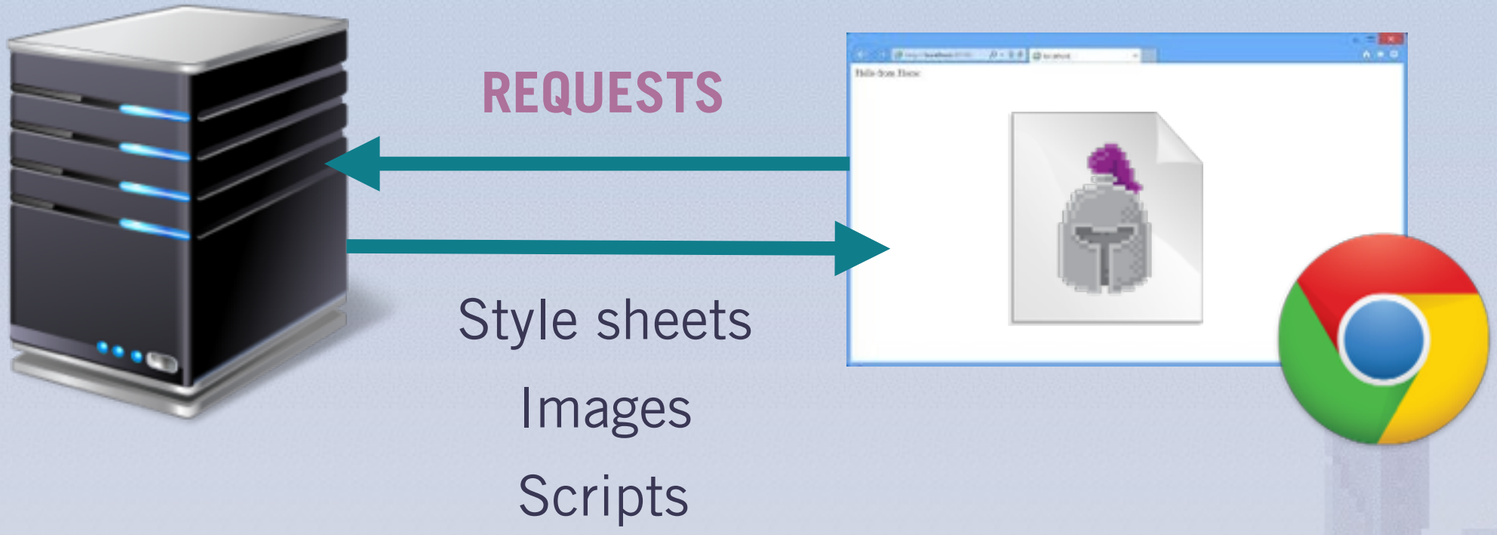
SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let's take a closer look at how a browser retrieves and acts on scripts.



SCRIPT PLACEMENT IMPACTS PERFORMANCE

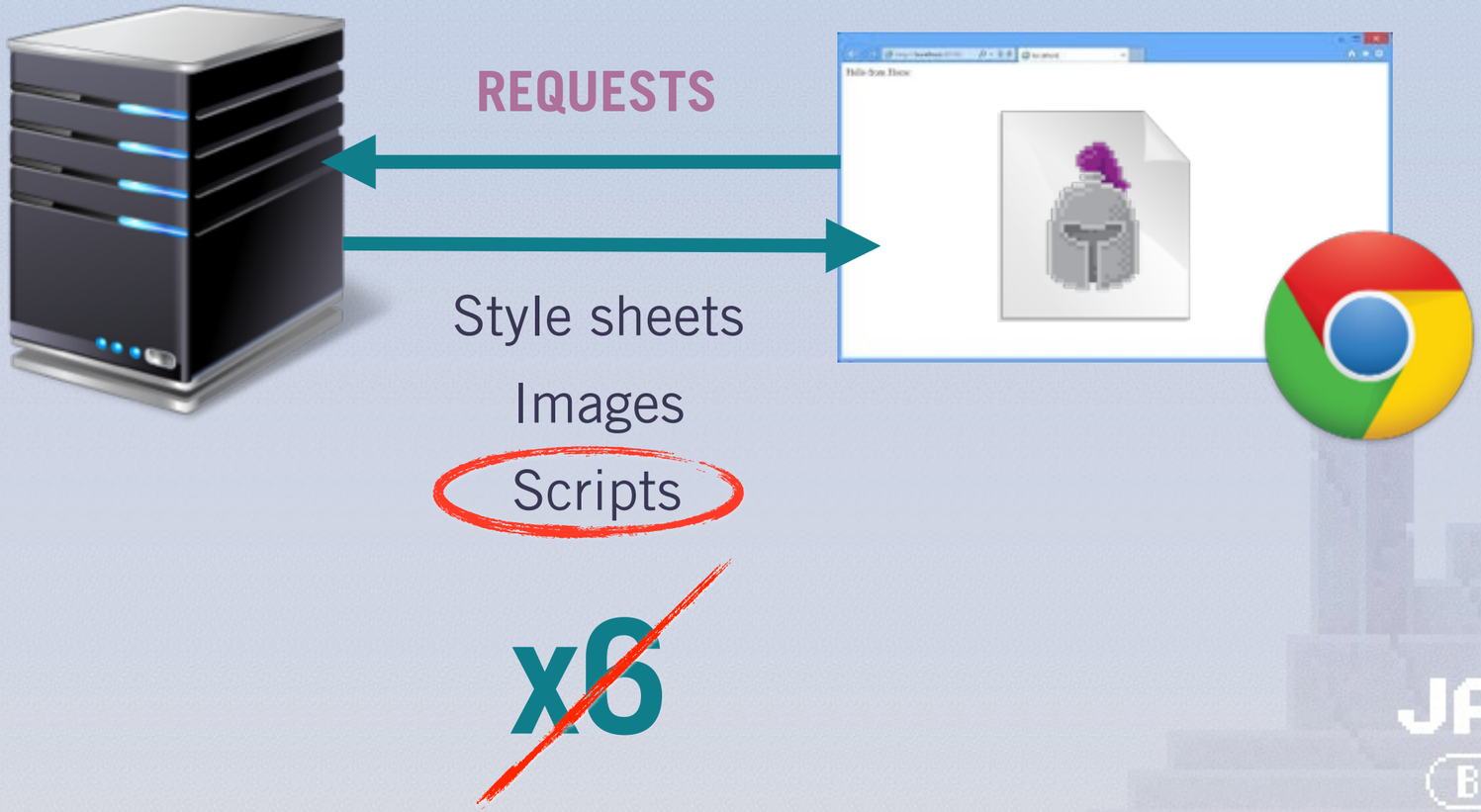
Let's take a closer look at how a browser retrieves and acts on scripts.



JAVASCRIPT
BEST PRACTICES

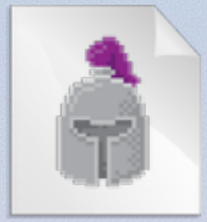
SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let's take a closer look at how a browser retrieves and acts on scripts.



SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let’s take a closer look at how a browser retrieves and acts on scripts.



JAVASCRIPT
BEST PRACTICES

WHERE WOULD WE FIND TROUBLING SCRIPTS?

Scripts encountered high in the <head> or <body> tags of an HTML page can have adverse effects.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>

  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"></script>

  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>

  (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```

This script tag seems harmless enough, right? Just a quick download of a file, and then we'll get on with our lives ...

JAVASCRIPT
BEST PRACTICES

WHERE WOULD WE FIND TROUBLING SCRIPTS?

Scripts encountered high in the <head> or <body> tags of an HTML page can have adverse effects.

```
<!DOCTYPE html>
<html>
   <body>
   <h1>The Third Regiment</h1>

   <script type="text/javascript"
           src="http://www.ThirdReg.com/sparring.js"></script>

   <p>
   Sparring event at 3:30 Tuesday, homies!
   <img src="sparring.gif"
    alt="Are you ready?"
    width="42" height="42">
   </p>

   (...and perhaps a bunch of other important HTML...)
   </body>
</html>
```

SPARRING.JS

-builds a huge array of soldiers from a separate file.

-calculates all the possible unique groups for:

- -sparring groups of 2
- -sparring groups of 3
- -sparring groups of 4

-randomizes order of those groups

-collects groups in a pairs for larger matches

...more processes...
...more processes...
...more processes...

…tick-tick-tick-tick-tick-tick…

ONE SOLUTION: RELOCATE WORK-INTENSIVE SCRIPTS

Scripts that are not essential to immediate loading of the page should be moved as low as possible.

```
<!DOCTYPE html>
<html>
   <body>
   <h1>The Third Regiment</h1>

   <script type="text/javascript"
           src="http://www.ThirdReg.com/sparring.js"></script>

   <p>
   Sparring event at 3:30 Tuesday, homies!
   <img src="sparring.gif"
    alt="Are you ready?"
    width="42" height="42">
   </p>

   (...and perhaps a bunch of other important HTML...)
   </body>
</html>
```

SPARRING.JS

```
-builds a huge array of
soldiers from a separate file.

-calculates all the possible
unique groups for:
     -sparring groups of 2
     -sparring groups of 3
     -sparring groups of 4

-randomizes order of those groups

-collects groups in a pairs
for larger matches

...more processes...
...more processes...
...more processes...
```

…tick-tick-tick-tick-tick-tick…

ONE SOLUTION: RELOCATE WORK-INTENSIVE SCRIPTS

Scripts that are not essential to immediate loading of the page should be moved as low as possible.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>

  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"></script>

  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>

  (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```

JAVASCRIPT
BEST PRACTICES

ONE SOLUTION: RELOCATE WORK-INTENSIVE SCRIPTS

Scripts that are not essential to immediate loading of the page should be moved as low as possible.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>
  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>

  (...and perhaps a bunch of other important HTML...)

  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"></script>

  </body>
</html>
```

Now, most of the visual and informational components of the site will become available to the user before the script is loaded and processed.

JAVASCRIPT
BEST PRACTICES

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 async attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>
  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>

  (...and perhaps a bunch of other important HTML...)

  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"></script>

  </body>
</html>
```

JAVASCRIPT
BEST PRACTICES

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 async attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>

  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"></script>

  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>

  (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```

If we add an async attribute inside our script tag, we'll prevent the script from blocking the page's load.

JAVASCRIPT
BEST PRACTICES

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 async attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>

  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"
                                              ></script>

  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>
  (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```

JAVASCRIPT
BEST PRACTICES

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 async attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>

  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"
          async></script>

  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>
  (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```

Though we've got it in the back here, the async attribute could be added anywhere inside our tag.

JAVASCRIPT
BEST PRACTICES

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 async attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
  <h1>The Third Regiment</h1>
  <script type="text/javascript"
          src="http://www.ThirdReg.com/sparring.js"
          async></script>
  <p>
  Sparring event at 3:30 Tuesday, homies!
  <img src="sparring.gif"
   alt="Are you ready?"
   width="42" height="42">
  </p>
  (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```

The browser will now fetch the script resource, but will continue parsing and rendering HTML, without waiting for the script to complete!

JAVASCRIPT
BEST PRACTICES

PERFORMANCE

Section 2

Script Execution

THE PENDANT OF PERFORMANCE

Section 3

Short Performance Tips

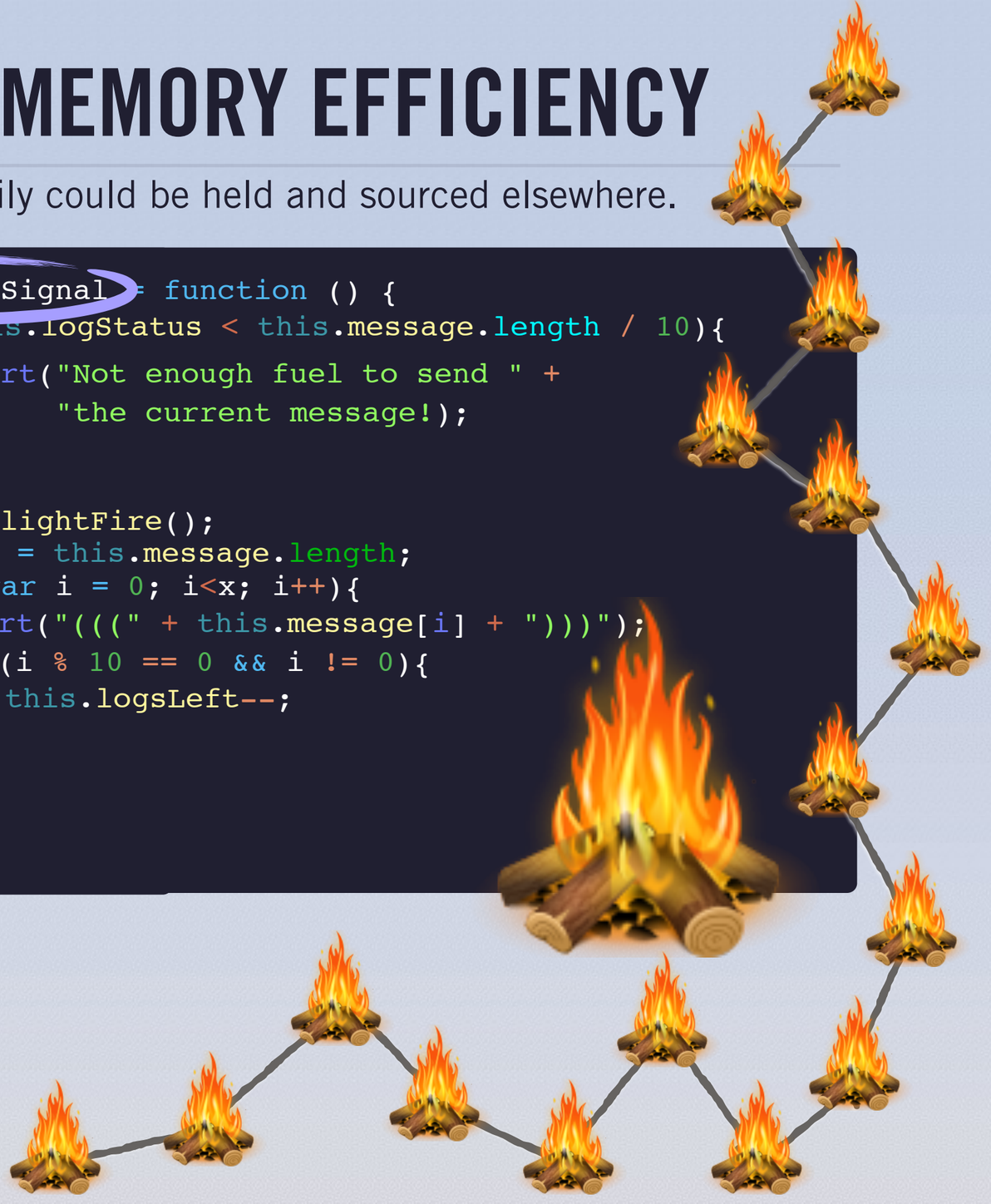
LET INHERITANCE HELP WITH MEMORY EFFICIENCY

Beware of loading up individual objects with code that easily could be held and sourced elsewhere.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;

 this.addLogs = function ( numLogs ){
     this.logsLeft += numLogs;
 }
 this.lightFire = function () {
     alert( whoooosh!");
 }
```

```
this.smokeSignal = function () {
    if (this.logStatus < this.message.length / 10){
        alert("Not enough fuel to send " +
              "the current message!);
    }
    else {
      this.lightFire();
      var x = this.message.length;
      for(var i = 0; i<x; i++){
        alert("(((" + this.message[i] + ")))");
        if (i % 10 == 0 && i != 0){
            this.logsLeft--;
        }
      }
    }
  }
}
```

We don't need to build all of these methods within every single `SignalFire` object, which would use extra memory AND take longer to create.



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;

 this.addLogs = function ( numLogs ){
     this.logsLeft += numLogs;
 }
 this.lightFire = function () {
     alert("Whoooosh!");
 }
```

```
this.smokeSignal = function () {
    if (this.logStatus < this.message.length / 10){
        alert("Not enough fuel to send " +
              "the current message!");
    }
    else {
      this.lightFire();
      var x = this.message.length;
      for(var i = 0; i<x; i++){
        alert("(((" + this.message[i] + ")))");
        if (i % 10 == 0 && i != 0){
            this.logsLeft--;
        }
      }
    }
  }
}
```

USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;

 this.addLogs = function ( numLogs ){
     this.logsLeft += numLogs;
 }
 this.lightFire = function () {
     alert("Whoooosh!");
 }
```

```
this.smokeSignal = function () {
    if (this.logStatus < this.message.length / 10){
        alert("Not enough fuel to send " +
              "the current message!");
    }
    else {
      this.lightFire();
      var x = this.message.length;
      for(var i = 0; i<x; i++){
        alert("(((" + this.message[i] + ")))");
        if (i % 10 == 0 && i != 0){
            this.logsLeft--;
        }
      }
    }
  }
}
```

USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;
```

```
this.addLogs = function ( numLogs ){
    this.logsLeft += numLogs;
}
this.lightFire = function () {
    alert("Whoooosh!");
}
this.smokeSignal = function () {
   if (this.logStatus < this.message.length / 10){
       alert("Not enough fuel to send " +
             "the current message!);
   }
   else {
     this.lightFire();
     var x = this.message.length;
     for(var i = 0; i<x; i++){
       alert("(((" + this.message[i] + ")))");
       if (i % 10 == 0 && i != 0){
           this.logsLeft--;
       }
     }
   }
 }
}
```

USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;
}

SignalFire.prototype = {
    addLogs   function ( numLogs ){
   this.logsLeft += numLogs;
}
    lightFire   function () {
   alert("Whoooosh!");
}
    smokeSignal   function () {
  if (this.logStatus < this.message.length / 10){
     alert("Not enough fuel to send " +
           "the current message!");
  }
  else {
    this.lightFire();
    var x = this.message.length;
    for(var i = 0; i<x; i++){
      alert("(((" + this.message[i] + ")))");
      if (i % 10 == 0 && i != 0){
         this.logsLeft--;
      }
    }
  }
}
```

USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;
}

SignalFire.prototype = {
    addLogs:  function ( numLogs ){
        this.logsLeft += numLogs;
    },
    lightFire:  function () {
        alert("Whoooosh!");
    },
    smokeSignal:  function () {
        if (this.logStatus < this.message.length / 10){
            alert("Not enough fuel to send " +
                  "the current message!");
        }
        else {
            this.lightFire();
            var x = this.message.length;
            for(var i = 0; i<x; i++){
                alert("((((" + this.message[i] + "))))");
                if (i % 10 == 0 && i != 0){
                    this.logsLeft--;
                }
            }
        }
    }
}
```

USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;
```

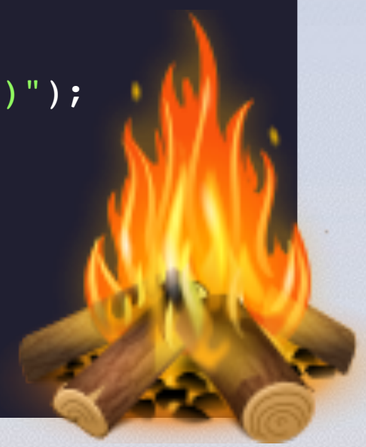
```
SignalFire.prototype = {
  addLogs:  function ( numLogs ){
      this.logsLeft += numLogs;
  }
  lightFire:  function () {
      alert("Whoooosh!");
  }
  smokeSignal:  function () {
     if (this.logStatus < this.message.length / 10){
         alert("Not enough fuel to send " +
               "the current message!);
     }
     else {
       this.lightFire();
       var x = this.message.length;
       for(var i = 0; i<x; i++){
         alert("(((" + this.message[i] + ")))");
         if (i % 10 == 0 && i != 0){
             this.logsLeft--;
         }
       }
     }
   }
}
```

USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;









}
```

```
SignalFire.prototype = {
  addLogs:  function ( numLogs ){
      this.logsLeft += numLogs;
  }
  lightFire:  function () {
      alert("Whoooosh!");
  }
  smokeSignal:  function () {
     if (this.logStatus < this.message.length / 10){
         alert("Not enough fuel to send " +
               "the current message!);

     }
     else {
       this.lightFire();
       var x = this.message.length;
       for(var i = 0; i<x; i++){
         alert("(((" + this.message[i] + ")))");
         if (i % 10 == 0 && i != 0){
            this.logsLeft--;
         }
       }
     }
  }
}
```

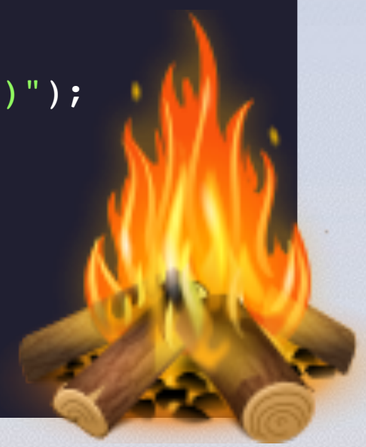


USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;
}
```

```
SignalFire.prototype = {
  addLogs:  function ( numLogs ){
      this.logsLeft += numLogs;
  }
  lightFire:  function () {
      alert("Whoooosh!");
  }
  smokeSignal:  function () {
     if (this.logStatus < this.message.length / 10){
         alert("Not enough fuel to send " +
               "the current message!);
     }
     else {
       this.lightFire();
       var x = this.message.length;
       for(var i = 0; i<x; i++){
         alert("(((" + this.message[i] + ")))");
         if (i % 10 == 0 && i != 0){
            this.logsLeft--;
         }
       }
     }
  }
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;
}
```

```
var fireOne = new SignalFire(1, 20);
var fireTwo = new SignalFire(2, 16);
var fireThree = new SignalFire(3, 24);
```

```
fireOne.addLogs(8);
fireTwo.addLogs(10);
fireThree.addLogs(4);
```

Now, the `addLogs` method is found in one useful place, instead of being replicated across all `signalFires`.

```
SignalFire.prototype = {
   addLogs:  function ( numLogs ){
       this.logsLeft += numLogs;
   }
   lightFire:  function () {
       alert("Whoooosh!");
   }
   smokeSignal:  function () {
      if (this.logStatus < this.message.length / 10){
          alert("Not enough fuel to send " +
                "the current message!);

      }
      else {
        this.lightFire();
        var x = this.message.length;
        for(var i = 0; i<x; i++){
          alert("(((" + this.message[i] + ")))");
          if (i % 10 == 0 && i != 0){
             this.logsLeft--;
          }
        }
      }
   }
}
```

USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

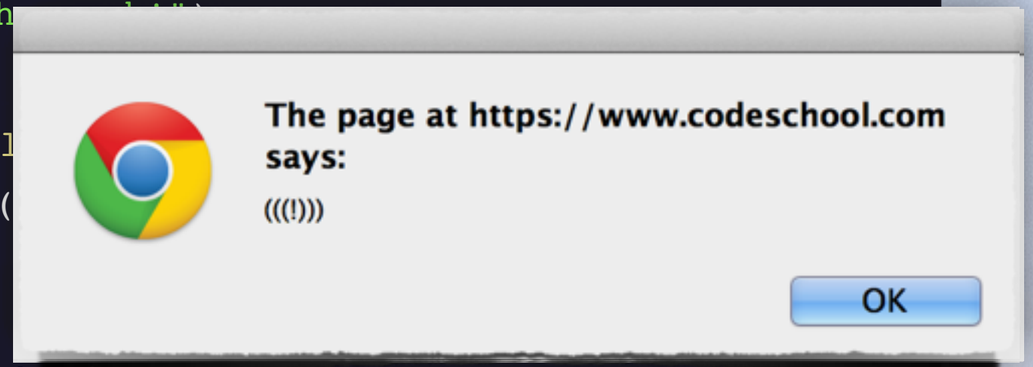
```
function SignalFire( ID, startingLogs ){
 this.fireID = ID;
 this.logsLeft = startingLogs;
}
```

```
var fireOne = new SignalFire(1, 20);
var fireTwo = new SignalFire(2, 18);
var fireThree = new SignalFire(3, 24);
```

```
fireOne.addLogs(8);
fireTwo.addLogs(10);
fireThree.addLogs(4);
```

```
fireThree.smokeSignal("Goblins!");
```

```
SignalFire.prototype = {
  addLogs:  function ( numLogs ){
      this.logsLeft += numLogs;
  }
  lightFire:  function () {
      alert("Wh
  }
  smokeSignal:
     if (this.l
         alert(


     }
     else {
       this.lightFire();
       var x = this.message.length;
       for(var i = 0; i<x; i++){
         alert("(((" + this.message[i] + ")))");
         if (i % 10 == 0 && i != 0){
            this.logsLeft--;
         }
       }
     }
  }
}
```



ADDING INDIVIDUAL DOM ELEMENTS AIN'T ALWAYS SPEEDY

Each new addition to the DOM causes document "reflow", which can really hinder user experience.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];

for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  list.appendChild(element);
}
```

Each time the list is appended, we access the DOM and cause an entire document reflow. Not as speedy as we'd like, especially if our list was huge...

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];

for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  list.appendChild(element);
}
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  list.appendChild(element);
}
```

First we create a fragment, which will function as a staging area to hold all of our new `li` elements.

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
      .appendChild(element);
}
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
```

Now we add each new `li` element to the staging fragment, instead of to the document itself.

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

Finally, we append all of our new text nodes in one fell swoop, using only one DOM touch!

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

BONUS BP: DECLARE VARIABLES AS FEW TIMES AS POSSIBLE

Every var keyword adds a look-up for the JavaScript parser that can be avoided with comma extensions.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

BONUS BP: DECLARE VARIABLES AS FEW TIMES AS POSSIBLE

Every var keyword adds a look-up for the JavaScript parser that can be avoided with comma extensions.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

Commas used after an initial declaration can signal that you'll be declaring further variables. It's also arguably more legible for those reading your code.

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

This variable will be declared every time the loop executes!

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment()
for (var i = 0, x = kotw.length; i<x; i++) {
      element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment()
for (var i = 0, x = kotw.length; i<x; i++) {
  element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment(),
    element;
for (var i = 0, x = kotw.length; i<x; i++) {
  element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

In the tradeoff, we've opted to avoid processor burden instead of having `element` declared where it is used.

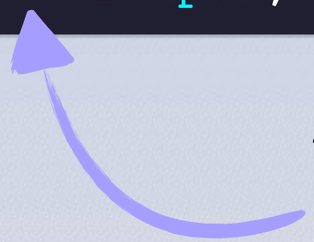
```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type=text/javascript
        src="http://www.ThirdReg.com/kotw.js">
</script>


</body>
</html>
```

EFFICIENT CHOICES FOR STRING CONCATENATION

In building strings, different methods will yield different results in terms of execution speed.

```
var knight = "Jenna Rangespike”;
var action = " strikes the dragon with a ";
var weapon = "Halberd";
```

```
var turn = "";
turn += knight;
turn += action;
turn += weapon;
```



The standard concatenation operator has been optimized in most modern browser versions, and is an ideal choice for a small number of string concatenations.

EFFICIENT CHOICES FOR STRING CONCATENATION

In building strings, different methods will yield different results in terms of execution speed.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
                     ***a hundred or more other html elements***,
                     "</script>", “</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```

While concatenation does get the job done, another method enjoys a performance boost when your strings are present in a array.

DECIDE WHAT'S BETTER BEFORE REACHING FOR +=

For concatenations over an array's contents, use the join() method inherited from the Array prototype.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
                     ***a hundred or more other html elements***,
                     "</script>", “</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```

```
page = newPageBuild.join("\n");
```

The `join` method concatenates each index of the array, "joined" by any string parameter you pass in. In addition to being faster in tests than many String methods, it is also easier to read.

DECIDE WHAT'S BETTER BEFORE REACHING FOR +=

For concatenations over an array's contents, use the join() method inherited from the Array prototype.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
                     ***a hundred or more other html elements***,
                     "</script>", “</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```

```
page = newPageBuild.join("\n");
```

DECIDE WHAT'S BETTER BEFORE REACHING FOR +=

For concatenations over an array's contents, use the join() method inherited from the Array prototype.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
                     ***a hundred or more other html elements***,
                     "</script>", “</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```

```
page = newPageBuild.join("\n");
console.log(page);
```

```
<!DOCTYPE html>
<html>
<body>
<h1>
...
</script>
</body>
</html>
```

PERFORMANCE

Section 3

Short Performance Tips

THE PENDANT OF PERFORMANCE

Section 4

Measuring Performance I:

`Console.time`

JAVASCRIPT
BEST PRACTICES

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
      case 1:
        this.weapon = "Broadsword";
        break;
      case 2:
        this.weapon = "Claymore";
        break;
      case 3:
        this.weapon = "Longsword";
        break;
      case 5:
        this.weapon = "War Hammer";
        break;
      case 6:
        this.weapon = "Battle Axe";
        break;
      case 4:
      case 7:
      case 8:
        this.weapon = "Morning Star";
        break;
      case "King":
        this.weapon = "Excalibur";
        break;
      default:
        alert(name + " has an incorrect " +
              "regiment, Master Armourer!" +
              "\n\nNo weapon assigned!");
    }
}
```

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

We'll steadily add each newbie Knight to the regiment's list of members.

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

To unite timer boundaries into one timer, their parameter labels must match.

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
Time to add 3 Knights: 0.036ms
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

`console.time` automatically prefaces the time measurement with the label we passed in as a parameter, plus a colon.

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

→ `Time to add 4 Knights: 0.040ms`

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
                new Knight( firstRegimentNewbs[i], 1)
  firstRegimentKnights.push(       );
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
                            new Knight( firstRegimentNewbs[i], 1)
  firstRegimentKnights.push(                                    );
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){

  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
Time to add 4 Knights: 0.029ms
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
   switch (regiment) {
     case 1:
       this.weapon = "Broadsword";
       break;
     case 2:
       this.weapon = "Claymore";
       break;
     case 3:
       this.weapon = "Longsword";
       break;
     case 5:
       this.weapon = "War Hammer";
       break;
     case 6:
       this.weapon = "Battle Axe";
       break;
     case 4:
     case 7:
     case 8:
       this.weapon = "Morning Star";
       break;
     case "King":
       this.weapon = "Excalibur";
       break;
     default:
       alert(name + " has an incorrect " +
             "regiment, Master Armourer!" +
             "\n\nNo weapon assigned!");
   }
}
```

Times will vary by browser, but ours was slightly better when not using the extra variable declaration and assignment inside the loop (a best practice!)

MULTIPLE TIMERS CAN RUN AT ONE TIME

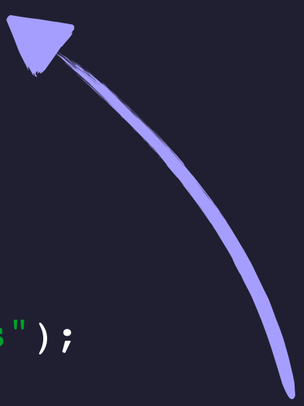
Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip”,
                           "Vorn Sharpeye”, "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];


console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```



First, we'll add a set of new guys that need to be added to the Second Regiment of Knights.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip”,
                           "Vorn Sharpeye”, "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];

console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

Next we'll start a timer for the total time of operation.

Both parts of our existing timer for the First Regiment additions stay in place.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];

console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");

console.time("Time to add " + secondRegimentNewbs.length + " Knights");
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + secondRegimentNewbs.length + " Knights");
```

Once these new additions are done, we'll stop the second regiment timer.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip”,
                           "Vorn Sharpeye”, "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];

console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");

console.time("Time to add " + secondRegimentNewbs.length + " Knights");
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + secondRegimentNewbs.length + " Knights");

console.timeEnd("Total completion time");
```

Finally, we'll wrap up the timing of the entire operation.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip”,
                           "Vorn Sharpeye”, "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];

console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");

console.time("Time to add " + secondRegimentNewbs.length + " Knights");
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + secondRegimentNewbs.length + " Knights");

console.timeEnd("Total completion time");
```

```
Time to add 4 Knights: 0.031ms
Time to add 8 Knights: 0.063ms
Total completion time: 0.324ms
```

Remember, setting up the timers takes time, too!

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip”,
                           "Vorn Sharpeye”, "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];

console.time("Total completion time");

for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}



for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}


console.timeEnd("Total completion time");
```

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                          "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip”,
                           "Vorn Sharpeye”, "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];

console.time("Total completion time");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Total completion time");
```

```
Total completion time: 0.069ms
Total completion time: 0.059ms
Total completion time: 0.055ms
Total completion time: 0.074ms
Total completion time: 0.050ms
```

If we want more accuracy in our estimate, we need to use the average of multiple tests.

PERFORMANCE

Section 4

Measuring Performance I:

`Console.time`

THE PENDANT OF PERFORMANCE

Section 5

Measuring Performance II:

Speed Averaging

JAVASCRIPT
BEST PRACTICES

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = new Date();
console.log(rightNow);
```

→ `Mon Apr 10 2014 17:50:38 GMT-0500 (EST)`

A new Date object immediately captures the current date and time, measured in milliseconds since 12:00 am, January 1st, 1970!

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = new Date();
console.log(rightNow);
```

RETRIEVING AND USING NUMERICAL TIME DATA

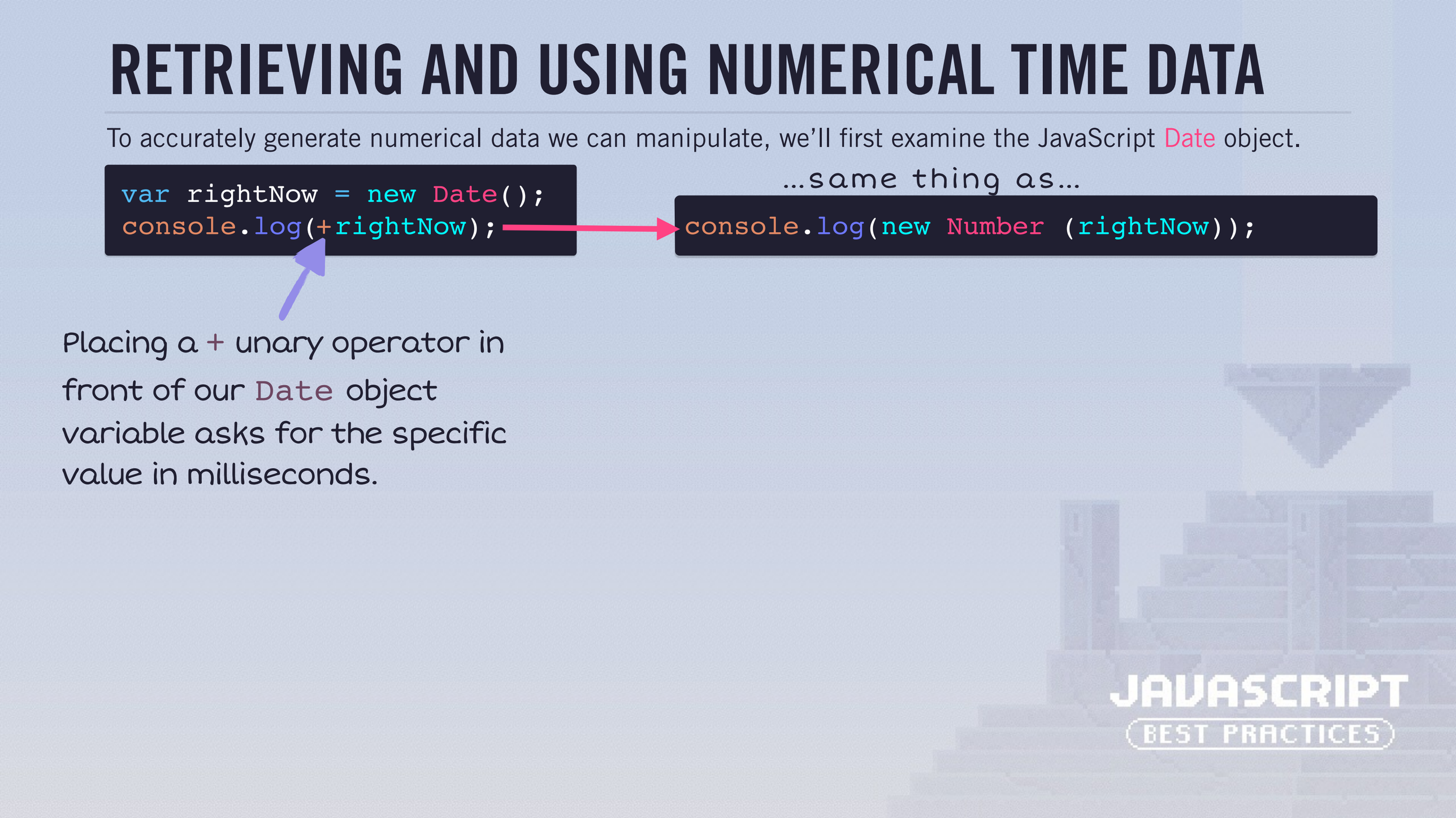
To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = new Date();
console.log(+rightNow);
```

...same thing as...

```
console.log(new Number (rightNow));
```

Placing a + unary operator in front of our Date object variable asks for the specific value in milliseconds.



RETRIEVING AND USING NUMERICAL TIME DATA

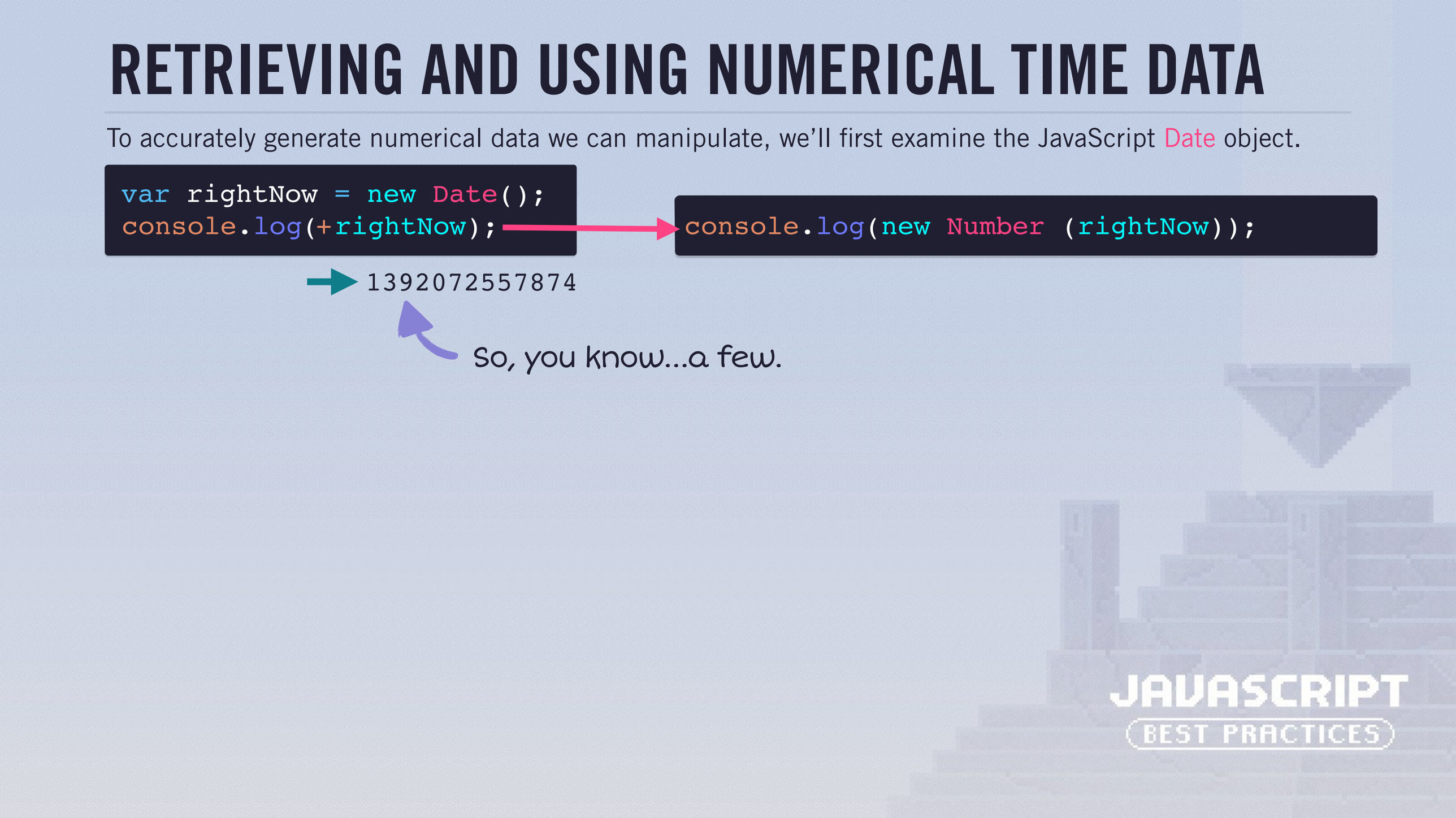
To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = new Date();
console.log(+rightNow);
```

```
console.log(new Number (rightNow));
```

```
1392072557874
```

So, you know...a few.



RETRIEVING AND USING NUMERICAL TIME DATA

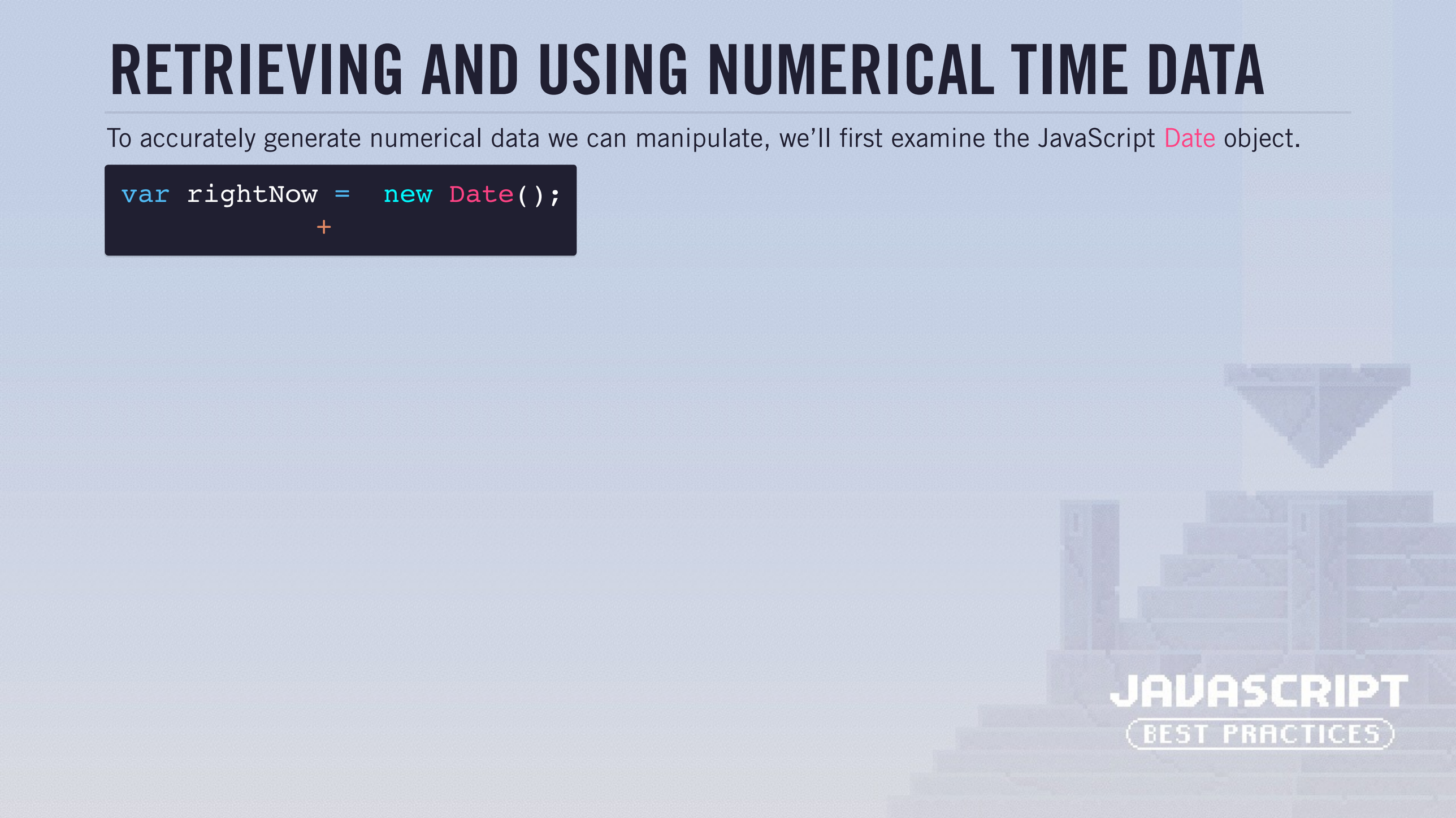
To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = new Date();
console.log(+rightNow);
```

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow =   new Date();
               +
```



RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = +new Date();
```

Since we know we want an actual number, we can go ahead and assign the variable to be the numerical version of our new `Date` Object.

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = +new Date();
```

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = +new Date();
var endTime = +new Date();
```

We can do this for a later moment
and also get a millisecond value.
You probably see what's coming...

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = +new Date();
var endTime = +new Date();
```

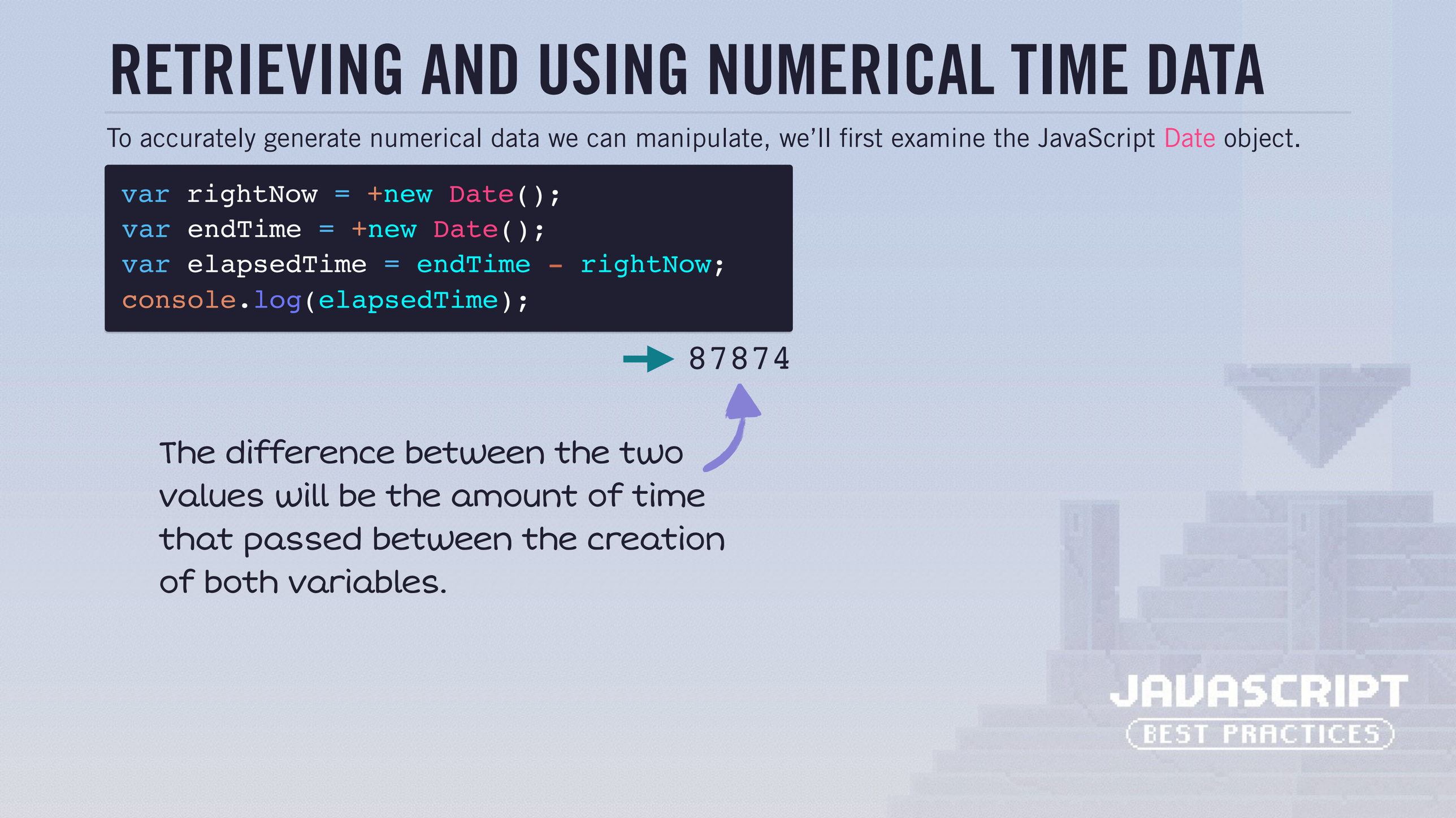
RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript Date object.

```
var rightNow = +new Date();
var endTime = +new Date();
var elapsedTime = endTime - rightNow;
console.log(elapsedTime);
```

→ 87874

The difference between the two values will be the amount of time that passed between the creation of both variables.



ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){



}
```

This will be the specific code we want to test for performance speed. We'll encapsulate it all within its own function later.

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){

    This represents whatever parameters
    our test code needs in order to work
    correctly. Might be an array of values,
    or it might be a single value.
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){


                    The higher the repetitions, the
                    more reliable our average speed.

}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){
    this.testImplement = testImplement;
    this.testParams = testParams;
    this.repetitions = repetitions || 10000;
    this.average = 0;
}
```

We make our `repetitions` parameter optional by defaulting to 10,000 executions, using our Logical Assignment best practice. We'll look at this optional nature in a bit.

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(                                                    ){
    this.testImplement = testImplement;
    this.testParams = testParams;
    this.repetitions = repetitions || 10000;
    this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(...){
    this.testImplement = testImplement;
    this.testParams = testParams;
    this.repetitions = repetitions || 10000;
    this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {



}
```

To use our best practice of adding commonly used methods to prototypes, we'll need to build the prototype itself.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){



  }
}
```

We'll call this inherited method to begin calculating an average speed for some test implementation.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
    var beginTime, endTime, sumTimes = 0;
    These two variables will get our
    numerical Date objects. Notice the
    legibility best practice of using a
    comma with no extra typed var's.
  }
}
```

This variable will sum up all the times for all the repetitions.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){



   }
  }
}
```

We'll loop over the full amount of requested repetitions. Another best practice: no repetitive property access!

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();



   }

  }
}
```

Start the clock for this individual repetition!

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );


   }

  }
}
```

Here's where we run the code for this repetition!

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();

   }

            Stop the clock!

  }
}
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }


 }
}
```

Here we add in the individual time spent on this particular test.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;

  }
}
```

Average is sum of all times, divided by repetitions.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

We return a message with the average time.

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
SpeedTest.prototype = {
  startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
```

```
for(var i = 0; i < firstRegimentNewbs.length; i++){
    var newGuy = new Knight(firstRegimentNewbs[i], 1);
    firstRegimentKnights.push(newGuy);
}
```

To test this block, we need to be able to pass all of it into the SpeedTest constructor as its own function.

```
var noBPtest = new SpeedTest(                    );
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
```

```
for(var i = 0; i < firstRegimentNewbs.length; i++){
    var newGuy = new Knight(firstRegimentNewbs[i], 1);
    firstRegimentKnights.push(newGuy);
}
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

```
var noBPtest = new SpeedTest(                );
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
```

```
var noBP = function (                ){
   for(var i =  ; i < firstRegimentNewbs.length; i++){
       var newGuy = new Knight(firstRegimentNewbs[i], 1);
       firstRegimentKnights.push(newGuy);
   }
};
```

If we wrap the code in its own function expression and assign it to a variable, we can easily pass it around as a "set" of code to be tested.

```
var noBPtest = new SpeedTest(noBP,          );
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < firstRegimentNewbs.length; i++){
       var newGuy = new Knight(firstRegimentNewbs[i], 1);
       firstRegimentKnights.push(newGuy);
   }
};
```

Since our `SpeedTest` lumps all the test code's important parameters into one `testParams` property, we'll need to modify our function expression and important data to use just one ARRAY of important stuff.

```
var noBPtest = new SpeedTest(noBP,          );
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < firstRegimentNewbs.length; i++){
       var newGuy = new Knight(firstRegimentNewbs[i], 1);
       firstRegimentKnights.push(newGuy);
   }
};
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

```
var noBPtest = new SpeedTest(noBP,          );
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < firstRegimentNewbs.length; i++){
       var newGuy = new Knight(firstRegimentNewbs[i], 1);
       firstRegimentKnights.push(newGuy);
   }
};
```

Now that we have an array of the parameters that noBP will need, we have to replace the existing names in the function with references to our single parameter array.

```
var noBPtest = new SpeedTest(noBP, listsForTests);
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i <    listOfParams[0].length; i++){
       var newGuy = new Knight(   listOfParams[0][i], 1);
                          .push(newGuy);
   }
};
```

Since `firstRegimentNewbs` is in the zeroth index of our new parameter array, we'll use that index to access its contents.

```
var noBPtest = new SpeedTest(noBP,listsForTests);
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i <    listOfParams[0].length; i++){
       var newGuy = new Knight(   listOfParams[0][i], 1);
                          .push(newGuy);
   }
};
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

```
var noBPtest = new SpeedTest(noBP,listsForTests);
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
            listOfParams[1].push(newGuy);
   }
};
```

Similarly, we'll use the next index for `firstRegimentKnights`.

```
var noBPtest = new SpeedTest(noBP,listsForTests);
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
            listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests);
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests);
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests);
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

In creating our new `SpeedTest` object, we're first opting to NOT pass in a `repetitions` amount, and thus receive our logical assignment default of 10,000.

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests);
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests);
noBPtest.startTest();
```

```
Average execution across 10000: 0.0041
```

Notice that we have about 10 times a speed improvement over the estimations from `console.time`.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests,100000);
noBPtest.startTest();
```

```
Average execution across 100000: 0.00478
```

Our newer estimate brings us ever closer to the truth.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0; i < listOfParams[0].length; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests,100000);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0                                    i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests,100000);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0, x = listOfParams[0].length; i < x; i++){
       var newGuy = new Knight(listOfParams[0][i], 1);
       listOfParams[1].push(newGuy);
   }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests,100000);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

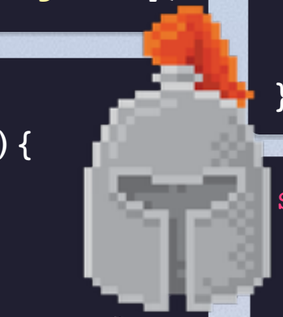
```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
   for(var i = 0, x = listOfParams[0].length; i < x; i++){
                    new Knight(listOfParams[0][i], 1)
       listOfParams[1].push(       );
   }
};
```



```
var noBPtest = new SpeedTest(noBP,listsForTests,100000);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
 for(var i = 0, x = listOfParams[0].length; i < x; i++){
                        new Knight(listOfParams[0][i], 1)
   listOfParams[1].push(                                );
 }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests,100000);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ){
 for(var i = 0, x = listOfParams[0].length; i < x; i++){
   listOfParams[1].push(new Knight(listOfParams[0][i], 1));
 }
};
```

```
var noBPtest = new SpeedTest(noBP,listsForTests,100000);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var    BP = function ( listOfParams ){
 for(var i = 0, x = listOfParams[0].length; i < x; i++){
   listOfParams[1].push(new Knight(listOfParams[0][i], 1));
 }
};
```

```
var    BPtest = new SpeedTest(noBP,listsForTests,100000);
  BPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var BP = function ( listOfParams ){
 for(var i = 0, x = listOfParams[0].length; i < x; i++){
   listOfParams[1].push(new Knight(listOfParams[0][i], 1));
 }
};
```

```
var BPtest = new SpeedTest(noBP,listsForTests,100000);
BPtest.startTest();
```

```
Average execution across 100000: 0.00274
```

Significantly better execution, especially if this process were replicated a few thousand times in a program!

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
 startTest: function(){
   var beginTime, endTime, sumTimes = 0;
   for (var i = 0, x = this.repetitions; i < x; i++){
     beginTime = +new Date();
     this.testImplement( this.testParams );
     endTime = +new Date();
     sumTimes += endTime - beginTime;
   }
   this.average = sumTimes / this.repetitions;
   return console.log("Average execution across " +
                      this.repetitions + ": " +
                      this.average);
  }
}
```

PERFORMANCE

Section 5

Measuring Performance II:

Speed Averaging