Chapter 104: Performance Tips

JavaScript, like any language, requires us to be judicious in the use of certain language features. Overuse of some features can decrease performance, while some techniques can be used to increase performance.

Section 104.1: Avoid try/catch in performance-critical functions

Some JavaScript engines (for example, the current version of Node.js and older versions of Chrome before Ignition+turbofan) don't run the optimizer on functions that contain a try/catch block.

If you need to handle exceptions in performance-critical code, it can be faster in some cases to keep the try/catch in a separate function. For example, this function will not be optimized by some implementations:

```
function myPerformanceCriticalFunction() {
    try {
        // do complex calculations here
    } catch (e) {
        console.log(e);
    }
}
```

However, you can refactor to move the slow code into a separate function (that *can* be optimized) and call it from inside the `try` block.

```
// This function can be optimized
function doCalculations() {
    // do complex calculations here
}

// Still not always optimized, but it's not doing much so the performance doesn't matter
function myPerformanceCriticalFunction() {
    try {
        doCalculations();
    } catch (e) {
        console.log(e);
    }
}
```

Here's a jsPerf benchmark showing the difference: https://jsperf.com/try-catch-deoptimization. In the current version of most browsers, there shouldn't be much difference if any, but in less recent versions of Chrome and Firefox, or IE, the version that calls a helper function inside the try/catch is likely to be faster.

Note that optimizations like this should be made carefully and with actual evidence based on profiling your code. As JavaScript engines get better, it could end up hurting performance instead of helping, or making no difference at all (but complicating the code for no reason). Whether it helps, hurts, or makes no difference can depend on a lot of factors, so always measure the effects on your code. That's true of all optimizations, but especially micro-optimizations like this that depend on low-level details of the compiler/runtime.

Section 104.2: Limit DOM Updates

A common mistake seen in JavaScript when run in a browser environment is updating the DOM more often than necessary.

The issue here is that every update in the DOM interface causes the browser to re-render the screen. If an update

changes the layout of an element in the page, the entire page layout needs to be re-computed, and this is very performance-heavy even in the simplest of cases. The process of re-drawing a page is known as *reflow* and can cause a browser to run slowly or even become unresponsive.

The consequence of updating the document too frequently is illustrated with the following example of adding items to a list.

Consider the following document containing a `<ul>` element:

```
<!DOCTYPE html>
<html>
    <body>
        <ul id="list"></ul>
    </body>
</html>
```

We add `5000` items to the list looping 5000 times (you can try this with a larger number on a powerful computer to increase the effect).

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
    list.innerHTML += `<li>item ${i}</li>`;  // update 5000 times
}
```

In this case, the performance can be improved by batching all 5000 changes in one single DOM update.

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
    html += `<li>item ${i}</li>`;
}
list.innerHTML = html;     // update once
```

The function `document.createDocumentFragment()` can be used as a lightweight container for the HTML created by the loop. This method is slightly faster than modifying the container element's `innerHTML` property (as shown below).

```
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();
for(var i = 1; i <= 5000; i++) {
    li = document.createElement("li");
    li.innerHTML = "item " + i;
    fragment.appendChild(li);
    i++;
}
list.appendChild(fragment);
```

Section 104.3: Benchmarking your code - measuring execution time

Most performance tips are very dependent of the current state of JS engines and are expected to be only relevant at a given time. The fundamental law of performance optimization is that you must first measure before trying to optimize, and measure again after a presumed optimization.

To measure code execution time, you can use different time measurement tools like:

Performance interface that represents timing-related performance information for the given page (only available in browsers).

process.hrtime on Node.js gives you timing information as [seconds, nanoseconds] tuples. Called without argument it returns an arbitrary time but called with a previously returned value as argument it returns the difference between the two executions.

Console timers `console.time("labelName")` starts a timer you can use to track how long an operation takes. You give each timer a unique label name, and may have up to 10,000 timers running on a given page. When you call `console.timeEnd("labelName")` with the same name, the browser will finish the timer for given name and output the time in milliseconds, that elapsed since the timer was started. The strings passed to time() and timeEnd() must match otherwise the timer will not finish.

Date.now function `Date.now()` returns current Timestamp in milliseconds, which is a Number representation of time since 1 January 1970 00:00:00 UTC until now. The method now() is a static method of Date, therefore you always use it as Date.now().

Example 1 using: `performance.now()`

In this example we are going to calculate the elapsed time for the execution of our function, and we are going to use the Performance.now() method that returns a DOMHighResTimeStamp, measured in milliseconds, accurate to one thousandth of a millisecond.

```
let startTime, endTime;

function myFunction() {
    //Slow code you want to measure
}

//Get the start time
startTime = performance.now();

//Call the time-consuming function
myFunction();

//Get the end time
endTime = performance.now();

//The difference is how many milliseconds it took to call myFunction()
console.debug('Elapsed time:', (endTime - startTime));
```

The result in console will look something like this:

```
Elapsed time: 0.10000000009313226
```

Usage of `performance.now()` has the highest precision in browsers with accuracy to one thousandth of a millisecond, but the lowest compatibility.

Example 2 using: `Date.now()`

In this example we are going to calculate the elapsed time for the initialization of a big array (1 million values), and we are going to use the `Date.now()` method

```
let t0 = Date.now(); //stores current Timestamp in milliseconds since 1 January 1970 00:00:00 UTC
let arr = []; //store empty array
for (let i = 0; i < 1000000; i++) { //1 million iterations
   arr.push(i); //push current i value
```

```
}
console.log(Date.now() - t0); //print elapsed time between stored t0 and now
```

Example 3 using: `console.time("label")` & `console.timeEnd("label")`

In this example we are doing the same task as in Example 2, but we are going to use the `console.time("label")` & `console.timeEnd("label")` methods

```
console.time("t"); //start new timer for label name: "t"
let arr = []; //store empty array
for(let i = 0; i < 1000000; i++) { //1 million iterations
   arr.push(i); //push current i value
}
console.timeEnd("t"); //stop the timer for label name: "t" and print elapsed time
```

Exemple 4 using `process.hrtime()`

In Node.js programs this is the most precise way to measure spent time.

```
let start = process.hrtime();

// long execution here, maybe asynchronous

let diff = process.hrtime(start);
// returns for example [ 1, 2325 ]
console.log(`Operation took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
// logs: Operation took 1000002325 nanoseconds
```

Section 104.4: Use a memoizer for heavy-computing functions

If you are building a function that may be heavy on the processor (either clientside or serverside) you may want to consider a **memoizer** which is a *cache of previous function executions and their returned values*. This allows you to check if the parameters of a function were passed before. Remember, pure functions are those that given an input, return a corresponding unique output and don't cause side-effects outside their scope so, you should not add memoizers to functions that are unpredictable or depend on external resources (like AJAX calls or randomly returned values).

Let's say I have a recursive factorial function:

```
function fact(num) {
  return (num === 0)? 1 : num * fact(num - 1);
}
```

If I pass small values from 1 to 100 for example, there would be no problem, but once we start going deeper, we might blow up the call stack or make the process a bit painful for the JavaScript engine we're doing this in, especially if the engine doesn't count with tail-call optimization (although Douglas Crockford says that native ES6 has tail-call optimization included).

We could hard code our own dictionary from 1 to god-knows-what number with their corresponding factorials but, I'm not sure if I advise that! Let's create a memoizer, shall we?

```
var fact = (function() {
  var cache = {}; // Initialise a memory cache object

  // Use and return this function to check if val is cached
```

```
  function checkCache(val) {
    if (val in cache) {
      console.log('It was in the cache :D');
      return cache[val]; // return cached
    } else {
      cache[val] = factorial(val); // we cache it
      return cache[val]; // and then return it
    }

    /* Other alternatives for checking are:
    || cache.hasOwnProperty(val) or !!cache[val]
    || but wouldn't work if the results of those
    || executions were falsy values.
    */
  }

  // We create and name the actual function to be used
  function factorial(num) {
    return (num === 0)? 1 : num * factorial(num - 1);
  } // End of factorial function

  /* We return the function that checks, not the one
  || that computes because  it happens to be recursive,
  || if it weren't you could avoid creating an extra
  || function in this self-invoking closure function.
  */
  return checkCache;
}());
```

Now we can start using it:

```
> fact(100)
< 9.33262154439441e+157
> fact(100)
  It was in the cache :D
< 9.33262154439441e+157
```

Now that I start to reflect on what I did, if I were to increment from 1 instead of decrement from *num*, I could have cached all of the factorials from 1 to *num* in the cache recursively, but I will leave that for you.

This is great but what if we have **multiple parameters**? This is a problem? Not quite, we can do some nice tricks like using JSON.stringify() on the arguments array or even a list of values that the function will depend on (for object-oriented approaches). This is done to generate a unique key with all the arguments and dependencies included.

We can also create a function that "memoizes" other functions, using the same scope concept as before (returning a new function that uses the original and has access to the cache object):

WARNING: ES6 syntax, if you don't like it, replace ... with nothing and use the `var args = Array.prototype.slice.call(null, arguments);` trick; replace const and let with var, and the other things you already know.

```
function memoize(func) {
  let cache = {};

  // You can opt for not naming the function
  function memoized(...args) {
    const argsKey = JSON.stringify(args);
```

```
    // The same alternatives apply for this example
    if (argsKey in cache) {
      console.log(argsKey + ' was/were in cache :D');
      return cache[argsKey];
    } else {
      cache[argsKey] = func.apply(null, args); // Cache it
      return cache[argsKey]; // And then return it
    }
  }

  return memoized; // Return the memoized function
}
```

Now notice that this will work for multiple arguments but won't be of much use in object-oriented methods I think, you may need an extra object for dependencies. Also, `func.apply(null, args)` can be replaced with `func(...args)` since array destructuring will send them separately instead of as an array form. Also, just for reference, passing an array as an argument to func won't work unless you use `Function.prototype.apply` as I did.

To use the above method you just:

```
const newFunction = memoize(oldFunction);

// Assuming new oldFunction just sums/concatenates:
newFunction('meaning of life', 42);
// -> "meaning of life42"

newFunction('meaning of life', 42); // again
// => ["meaning of life",42] was/were in cache :D
// -> "meaning of life42"
```

Section 104.5: Initializing object properties with null

All modern JavaScript JIT compilers trying to optimize code based on expected object structures. Some tip from mdn.

> Fortunately, the objects and properties are often "predictable", and in such cases their underlying structure can also be predictable. JITs can rely on this to make predictable accesses faster.

The best way to make object predictable is to define a whole structure in a constructor. So if you're going to add some extra properties after object creation, define them in a constructor with **`null`**. This will help the optimizer to predict object behavior for its whole life cycle. However all compilers have different optimizers, and the performance increase can be different, but overall it's good practice to define all properties in a constructor, even when their value is not yet known.

Time for some testing. In my test, I'm creating a big array of some class instances with a for loop. Within the loop, I'm assigning the same string to all object's "x" property before array initialization. If constructor initializes "x" property with null, array always processes better even if it's doing extra statement.

This is code:

```
function f1() {
    var P = function () {
        this.value = 1
    };
    var big_array = new Array(10000000).fill(1).map((x, index)=> {
```

```
        p = new P();
        if (index > 5000000) {
            p.x = "some_string";
        }

        return p;
    });
    big_array.reduce((sum, p)=> sum + p.value, 0);
}

function f2() {
    var P = function () {
        this.value = 1;
        this.x = null;
    };
    var big_array = new Array(10000000).fill(1).map((x, index)=> {
        p = new P();
        if (index > 5000000) {
            p.x = "some_string";
        }

        return p;
    });
    big_array.reduce((sum, p)=> sum + p.value, 0);
}


(function perform(){
    var start = performance.now();
    f1();
    var duration = performance.now() - start;

    console.log('duration of f1  ' + duration);


    start = performance.now();
    f2();
    duration = performance.now() - start;

    console.log('duration of f2 ' + duration);
})()
```

This is the result for Chrome and Firefox.

```
       FireFox     Chrome
-------------------------
f1      6,400      11,400
f2      1,700       9,600
```

As we can see, the performance improvements are very different between the two.

Section 104.6: Reuse objects rather than recreate

Example A

```
var i,a,b,len;
a = {x:0,y:0}
function test(){ // return object created each call
    return {x:0,y:0};
}
```

```
function test1(a){ // return object supplied
    a.x=0;
    a.y=0;
    return a;
}

for(i = 0; i < 100; i ++){ // Loop A
   b = test();
}

for(i = 0; i < 100; i ++){ // Loop B
   b = test1(a);
}
```

Loop B is 4 (400%) times faster than Loop A

It is very inefficient to create a new object in performance code. Loop A calls function `test()` which returns a new object every call. The created object is discarded every iteration, Loop B calls `test1()` that requires the object returns to be supplied. It thus uses the same object and avoids allocation of a new object, and excessive GC hits. (GC were not included in the performance test)

Example B

```
var i,a,b,len;
a = {x:0,y:0}
function test2(a){
    return {x : a.x * 10,y : a.x * 10};
}
function test3(a){
    a.x= a.x * 10;
    a.y= a.y * 10;
    return a;
}
for(i = 0; i < 100; i++){  // Loop A
    b = test2({x : 10, y : 10});
}
for(i = 0; i < 100; i++){ // Loop B
    a.x = 10;
    a.y = 10;
    b = test3(a);
}
```

Loop B is 5 (500%) times faster than loop A

Section 104.7: Prefer local variables to globals, attributes, and indexed values

JavaScript engines first look for variables within the local scope before extending their search to larger scopes. If the variable is an indexed value in an array, or an attribute in an associative array, it will first look for the parent array before it finds the contents.

This has implications when working with performance-critical code. Take for instance a common `for` loop:

```
var global_variable = 0;
function foo(){
    global_variable = 0;
    for (var i=0; i<items.length; i++) {
        global_variable += items[i];
    }
```

```
}
```

For every iteration in `for` loop, the engine will lookup `items`, lookup the `length` attribute within items, lookup `items` again, lookup the value at index `i` of `items`, and then finally lookup `global_variable`, first trying the local scope before checking the global scope.

A performant rewrite of the above function is:

```
function foo(){
    var local_variable = 0;
    for (var i=0, li=items.length; i<li; i++) {
        local_variable += items[i];
    }
    return local_variable;
}
```

For every iteration in the rewritten `for` loop, the engine will lookup `li`, lookup `items`, lookup the value at index `i`, and lookup `local_variable`, this time only needing to check the local scope.

Section 104.8: Be consistent in use of Numbers

If the engine is able to correctly predict you're using a specific small type for your values, it will be able to optimize the executed code.

In this example, we'll use this trivial function summing the elements of an array and outputting the time it took:

```
// summing properties
var sum = (function(arr){
        var start = process.hrtime();
        var sum = 0;
        for (var i=0; i<arr.length; i++) {
                sum += arr[i];
        }
        var diffSum = process.hrtime(start);
        console.log(`Summing took ${diffSum[0] * 1e9 + diffSum[1]} nanoseconds`);
        return sum;
})(arr);
```

Let's make an array and sum the elements:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Result:

```
Summing took 384416 nanoseconds
```

Now, let's do the same but with only integers:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Result:

```
Summing took 180520 nanoseconds
```

Summing integers took half the time here.

Engines don't use the same types you have in JavaScript. As you probably know, all numbers in JavaScript are IEEE754 double precision floating point numbers, there's no specific available representation for integers. But engines, when they can predict you only use integers, can use a more compact and faster to use representation, for example, short integers.

This kind of optimization is especially important for computation or data intensive applications.