You Don't Know JS: Async & Performance

Chapter 6: Benchmarking & Tuning

As the first four chapters of this book were all about performance as a coding pattern (asynchrony and concurrency), and Chapter 5 was about performance at the macro program architecture level, this chapter goes after the topic of performance at the micro level, focusing on single expressions/statements.

One of the most common areas of curiosity -- indeed, some developers can get quite obsessed about it -- is in analyzing and testing various options for how to write a line or chunk of code, and which one is faster.

We're going to look at some of these issues, but it's important to understand from the outset that this chapter is **not** about feeding the obsession of micro-performance tuning, like whether some given JS engine can run `++a` faster than `a++`. The more important goal of this chapter is to figure out what kinds of JS performance matter and which ones don't, *and how to tell the difference*.

But even before we get there, we need to explore how to most accurately and reliably test JS performance, because there's tons of misconceptions and myths that have flooded our collective cult knowledge base. We've got to sift through all that junk to find some clarity.

Benchmarking

OK, time to start dispelling some misconceptions. I'd wager the vast majority of JS developers, if asked to benchmark the speed (execution time) of a certain operation, would initially go about it something like this:

```
var start = (new Date()).getTime();     // or `Date.now()`

// do some operation

var end = (new Date()).getTime();

console.log( "Duration:", (end - start) );
```

Raise your hand if that's roughly what came to your mind. Yep, I thought so. There's a lot wrong with this approach, but don't feel bad; **we've all been there.**

What did that measurement tell you, exactly? Understanding what it does and doesn't say about the execution time of the operation in question is key to learning how to appropriately benchmark performance in JavaScript.

If the duration reported is `0`, you may be tempted to believe that it took less than a millisecond. But that's not very accurate. Some platforms don't have single millisecond precision, but instead only update the timer in larger increments. For example, older versions of windows (and thus IE) had only 15ms precision, which means the operation has to take at least that long for anything other than `0` to be reported!

Moreover, whatever duration is reported, the only thing you really know is that the operation took approximately that long on that exact single run. You have near-zero confidence that it will always run at that speed. You have no idea if the engine or system had some sort of interference at that exact moment, and that at other times the operation could run faster.

What if the duration reported is `4`? Are you more sure it took about four milliseconds? Nope. It might have taken less time, and there may have been some other delay in getting either `start` or `end` timestamps.

More troublingly, you also don't know that the circumstances of this operation test aren't overly optimistic. It's possible that the JS engine figured out a way to optimize your isolated test case, but in a more real program such optimization would be diluted or impossible, such that the operation would run slower than your test.

So... what do we know? Unfortunately, with those realizations stated, **we know very little.** Something of such low confidence isn't even remotely good enough to build your determinations on. Your "benchmark" is basically useless. And worse, it's dangerous in that it implies false confidence, not just to you but also to others who don't think critically about the conditions that led to those results.

Repetition

"OK," you now say, "Just put a loop around it so the whole test takes longer." If you repeat an operation 100 times, and that whole loop reportedly takes a total of 137ms, then you can just divide by 100 and get an average duration of 1.37ms for each operation, right?

Well, not exactly.

A straight mathematical average by itself is definitely not sufficient for making judgments about performance which you plan to extrapolate to the breadth of your entire application. With a hundred iterations, even a couple of outliers (high or low) can skew the average, and then when you apply that conclusion repeatedly, you even further inflate the skew beyond credulity.

Instead of just running for a fixed number of iterations, you can instead choose to run the loop of tests until a certain amount of time has passed. That might be more reliable, but how do you decide how long to run? You might guess that it should be some multiple of how long your operation should take to run once. Wrong.

Actually, the length of time to repeat across should be based on the accuracy of the timer you're using, specifically to minimize the chances of inaccuracy. The less precise your timer, the longer you need to run to make sure you've minimized the error percentage. A 15ms timer is pretty bad for accurate benchmarking; to minimize its uncertainty (aka "error rate") to less than 1%, you need to run your each cycle of test iterations for 750ms. A 1ms timer only needs a cycle to run for 50ms to get the same confidence.

But then, that's just a single sample. To be sure you're factoring out the skew, you'll want lots of samples to average across. You'll also want to understand something about just how slow the worst sample is, how fast the best sample is, how far apart those best and worse cases were, and so on. You'll want to know not just a number that tells you how fast something ran, but also to have some quantifiable measure of how trustable that number is.

Also, you probably want to combine these different techniques (as well as others), so that you get the best balance of all the possible approaches.

That's all bare minimum just to get started. If you've been approaching performance benchmarking with anything less serious than what I just glossed over, well... "you don't know: proper benchmarking."

Benchmark.js

Any relevant and reliable benchmark should be based on statistically sound practices. I am not going to write a chapter on statistics here, so I'll hand wave around some terms: standard deviation, variance, margin of error. If you don't know what those terms really mean -- I took a stats class back in college and I'm still a little fuzzy on them -- you are not actually qualified to write your own benchmarking logic.

Luckily, smart folks like John-David Dalton and Mathias Bynens do understand these concepts, and wrote a statistically sound benchmarking tool called Benchmark.js (http://benchmarkjs.com/). So I can end the suspense by simply saying: "just use that tool."

I won't repeat their whole documentation for how Benchmark.js works; they have fantastic API Docs (http://benchmarkjs.com/docs) you should read. Also there are some great (http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/) writeups (http://monsur.hossa.in/2012/12/11/benchmarkjs.html) on more of the details and methodology.

But just for quick illustration purposes, here's how you could use Benchmark.js to run a quick performance test:

```
function foo() {
        // operation(s) to test
}
```

```
var bench = new Benchmark(
        "foo test",                             // test name
        foo,                                    // function to test (just contents)
        {
                // ..                           // optional extra options (see docs)
        }
);

bench.hz;                                       // number of operations per second
bench.stats.moe;                        // margin of error
bench.stats.variance;           // variance across samples
// ..
```

There's *lots* more to learn about using Benchmark.js besides this glance I'm including here. But the point is that it's handling all of the complexities of setting up a fair, reliable, and valid performance benchmark for a given piece of JavaScript code. If you're going to try to test and benchmark your code, this library is the first place you should turn.

We're showing here the usage to test a single operation like X, but it's fairly common that you want to compare X to Y. This is easy to do by simply setting up two different tests in a "Suite" (a Benchmark.js organizational feature). Then, you run them head-to-head, and compare the statistics to conclude whether X or Y was faster.

Benchmark.js can of course be used to test JavaScript in a browser (see the "jsPerf.com" section later in this chapter), but it can also run in non-browser environments (Node.js, etc.).

One largely untapped potential use-case for Benchmark.js is to use it in your Dev or QA environments to run automated performance regression tests against critical path parts of your application's JavaScript. Similar to how you might run unit test suites before deployment, you can also compare the performance against previous benchmarks to monitor if you are improving or degrading application performance.

Setup/Teardown

In the previous code snippet, we glossed over the "extra options" `{ .. }` object. But there are two options we should discuss: `setup` and `teardown`.

These two options let you define functions to be called before and after your test case runs.

It's incredibly important to understand that your `setup` and `teardown` code **does not run for each test iteration**. The best way to think about it is that there's an outer loop (repeating cycles), and an inner loop (repeating test iterations). `setup` and `teardown` are run at the beginning and end of each *outer* loop (aka cycle) iteration, but not inside the inner loop.

Why does this matter? Let's imagine you have a test case that looks like this:

```
a = a + "w";
b = a.charAt( 1 );
```

Then, you set up your test `setup` as follows:

```
var a = "x";
```

Your temptation is probably to believe that `a` is starting out as `"x"` for each test iteration.

But it's not! It's starting `a` at `"x"` for each test cycle, and then your repeated `+ "w"` concatenations will be making a larger and larger `a` value, even though you're only ever accessing the character `"w"` at the `1` position.

Where this most commonly bites you is when you make side effect changes to something like the DOM, like appending a child element. You may think your parent element is set as empty each time, but it's actually getting lots of elements added, and that can significantly sway the results of your tests.

Context Is King

Don't forget to check the context of a particular performance benchmark, especially a comparison between X and Y tasks. Just because your test reveals that X is faster than Y doesn't mean that the conclusion "X is faster than Y" is actually relevant.

For example, let's say a performance test reveals that X runs 10,000,000 operations per second, and Y runs at 8,000,000 operations per second. You could claim that Y is 20% slower than X, and you'd be mathematically correct, but your assertion doesn't hold as much water as you'd think.

Let's think about the results more critically: 10,000,000 operations per second is 10,000 operations per millisecond, and 10 operations per microsecond. In other words, a single operation takes 0.1 microseconds, or 100 nanoseconds. It's hard to fathom just how small 100ns is, but for comparison, it's often cited that the human eye isn't generally capable of distinguishing anything less than 100ms, which is one million times slower than the 100ns speed of the X operation.

Even recent scientific studies showing that maybe the brain can process as quick as 13ms (about 8x faster than previously asserted) would mean that X is still running 125,000 times faster than the human brain can perceive a distinct thing happening. **X is going really, really fast.**

But more importantly, let's talk about the difference between X and Y, the 2,000,000 operations per second difference. If X takes 100ns, and Y takes 80ns, the difference is 20ns, which in the best case is still one 650-thousandth of the interval the human brain can perceive.

What's my point? **None of this performance difference matters, at all!**

But wait, what if this operation is going to happen a whole bunch of times in a row? Then the difference could add up, right?

OK, so what we're asking then is, how likely is it that operation X is going to be run over and over again, one right after the other, and that this has to happen 650,000 times just to get a sliver of a hope the human brain could perceive it. More likely, it'd have to happen 5,000,000 to 10,000,000 times together in a tight loop to even approach relevance.

While the computer scientist in you might protest that this is possible, the louder voice of realism in you should sanity check just how likely or unlikely that really is. Even if it is relevant in rare occasions, it's irrelevant in most situations.

The vast majority of your benchmark results on tiny operations -- like the `++x` vs `x++` myth -- **are just totally bogus** for supporting the conclusion that X should be favored over Y on a performance basis.

Engine Optimizations

You simply cannot reliably extrapolate that if X was 10 microseconds faster than Y in your isolated test, that means X is always faster than Y and should always be used. That's not how performance works. It's vastly more complicated.

For example, let's imagine (purely hypothetical) that you test some microperformance behavior such as comparing:

```
var twelve = "12";
var foo = "foo";

// test 1
var X1 = parseInt( twelve );
var X2 = parseInt( foo );

// test 2
var Y1 = Number( twelve );
var Y2 = Number( foo );
```

If you understand what `parseInt(..)` does compared to `Number(..)`, you might intuit that `parseInt(..)` potentially has "more work" to do, especially in the `foo` case. Or you might intuit that they should have the same amount of work to do in the `foo` case, as both should be able to stop at the first character `"f"`.

Which intuition is correct? I honestly don't know. But I'll make the case it doesn't matter what your intuition is. What might the results be when you test it? Again, I'm making up a pure hypothetical here, I haven't actually tried, nor do I care.

Let's pretend the test comes back that `X` and `Y` are statistically identical. Have you then confirmed your intuition about the `"f"` character thing? Nope.

It's possible in our hypothetical that the engine might recognize that the variables `twelve` and `foo` are only being used in one place in each test, and so it might decide to inline those values. Then it may realize that `Number( "12" )` can just be replaced by `12`. And maybe it comes to the same conclusion with `parseInt(..)`, or maybe not.

Or an engine's dead-code removal heuristic could kick in, and it could realize that variables `X` and `Y` aren't being used, so declaring them is irrelevant, so it doesn't end up doing anything at all in either test.

And all that's just made with the mindset of assumptions about a single test run. Modern engines are fantastically more complicated than what we're intuiting here. They do all sorts of tricks, like tracing and tracking how a piece of code behaves over a short period of time, or with a particularly constrained set of inputs.

What if the engine optimizes a certain way because of the fixed input, but in your real program you give more varied input and the optimization decisions shake out differently (or not at all!)? Or what if the engine kicks in optimizations because it sees the code being run tens of thousands of times by the benchmarking utility, but in your real program it will only run a hundred times in near proximity, and under those conditions the engine determines the optimizations are not worth it?

And all those optimizations we just hypothesized about might happen in our constrained test but maybe the engine wouldn't do them in a more complex program (for various reasons). Or it could be reversed -- the engine might not optimize such trivial code but may be more inclined to optimize it more aggressively when the system is already more taxed by a more sophisticated program.

The point I'm trying to make is that you really don't know for sure exactly what's going on under the covers. All the guesses and hypothesis you can muster don't amount to hardly anything concrete for really making such decisions.

Does that mean you can't really do any useful testing? **Definitely not!**

What this boils down to is that testing *not real* code gives you *not real* results. In so much as is possible and practical, you should test actual real, non-trivial snippets of your code, and under as best of real conditions as you can actually hope to. Only then will the results you get have a chance to approximate reality.

Microbenchmarks like `++x` vs `x++` are so incredibly likely to be bogus, we might as well just flatly assume them as such.

jsPerf.com

While Benchmark.js is useful for testing the performance of your code in whatever JS environment you're running, it cannot be stressed enough that you need to compile test results from lots of different environments (desktop browsers, mobile devices, etc.) if you want to have any hope of reliable test conclusions.

For example, Chrome on a high-end desktop machine is not likely to perform anywhere near the same as Chrome mobile on a smartphone. And a smartphone with a full battery charge is not likely to perform anywhere near the same as a smartphone with 2% battery life left, when the device is starting to power down the radio and processor.

If you want to make assertions like "X is faster than Y" in any reasonable sense across more than just a single environment, you're going to need to actually test as many of those real world environments as possible. Just because Chrome executes some X operation faster than Y doesn't mean that all browsers do. And of course you also probably will want to cross-reference the results of multiple browser test runs with the demographics of your users.

There's an awesome website for this purpose called jsPerf (http://jsperf.com). It uses the Benchmark.js library we talked about earlier to run statistically accurate and reliable tests, and makes the test on an openly available URL that you can pass around to others.

Each time a test is run, the results are collected and persisted with the test, and the cumulative test results are graphed on the page for anyone to see.

When creating a test on the site, you start out with two test cases to fill in, but you can add as many as you need. You also have the ability to set up `setup` code that is run at the beginning of each test cycle and `teardown` code run at the end of each cycle.

Note: A trick for doing just one test case (if you're benchmarking a single approach instead of a head-to-head) is to fill in the second test input boxes with placeholder text on first creation, then edit the test and leave the second test blank, which will delete it. You can always add more test cases later.

You can define the initial page setup (importing libraries, defining utility helper functions, declaring variables, etc.). There are also options for defining setup and teardown behavior if needed -- consult the "Setup/Teardown" section in the Benchmark.js discussion earlier.

Sanity Check

jsPerf is a fantastic resource, but there's an awful lot of tests published that when you analyze them are quite flawed or bogus, for any of a variety of reasons as outlined so far in this chapter.

Consider:

```
// Case 1
var x = [];
for (var i=0; i<10; i++) {
	x[i] = "x";
}

// Case 2
var x = [];
for (var i=0; i<10; i++) {
	x[x.length] = "x";
}

// Case 3
var x = [];
for (var i=0; i<10; i++) {
	x.push( "x" );
}
```

Some observations to ponder about this test scenario:

- It's extremely common for devs to put their own loops into test cases, and they forget that Benchmark.js already does all the repetition you need. There's a really strong chance that the `for` loops in these cases are totally unnecessary noise.
- The declaring and initializing of `x` is included in each test case, possibly unnecessarily. Recall from earlier that if `x = []` were in the `setup` code, it wouldn't actually be run before each test iteration, but instead once at the beginning of each cycle. That means `x` would continue growing quite large, not just the size `10` implied by the `for` loops.

  So is the intent to make sure the tests are constrained only to how the JS engine behaves with very small arrays (size `10`)? That *could* be the intent, but if it is, you have to consider if that's not focusing far too much on nuanced internal implementation details.

  On the other hand, does the intent of the test embrace the context that the arrays will actually be growing quite large? Is the JS engines' behavior with larger arrays relevant and accurate when compared with the intended real world usage?
- Is the intent to find out how much `x.length` or `x.push(..)` add to the performance of the operation to append to the `x` array? OK, that might be a valid thing to test. But then again, `push(..)` is a function call, so of course it's going to be slower than `[..]` access. Arguably, cases 1 and 2 are fairer than case 3.

Here's another example that illustrates a common apples-to-oranges flaw:

```
// Case 1
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort();

// Case 2
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort( function mySort(a,b){
	if (a < b) return -1;
	if (a > b) return 1;
	return 0;
} );
```

Here, the obvious intent is to find out how much slower the custom `mySort(..)` comparator is than the built-in default comparator. But by specifying the function `mySort(..)` as inline function expression, you've created an unfair/bogus test. Here, the second case is not only testing a custom user JS function, **but it's also testing creating a new function expression for each iteration.**

Would it surprise you to find out that if you run a similar test but update it to isolate only for creating an inline function expression versus using a pre-declared function, the inline function expression creation can be from 2% to 20% slower!?

Unless your intent with this test *is* to consider the inline function expression creation "cost," a better/fairer test would put `mySort(..)`'s declaration in the page setup -- don't put it in the test `setup` as that's unnecessary redeclaration for each cycle -- and simply reference it by name in the test case: `x.sort(mySort)`.

Building on the previous example, another pitfall is in opaquely avoiding or adding "extra work" to one test case that creates an apples-to-oranges scenario:

```
// Case 1
var x = [12,-14,0,3,18,0,2.9];
x.sort();

// Case 2
var x = [12,-14,0,3,18,0,2.9];
x.sort( function mySort(a,b){
        return a - b;
} );
```

Setting aside the previously mentioned inline function expression pitfall, the second case's `mySort(..)` works in this case because you have provided it numbers, but would have of course failed with strings. The first case doesn't throw an error, but it actually behaves differently and has a different outcome! It should be obvious, but: **a different outcome between two test cases almost certainly invalidates the entire test!**

But beyond the different outcomes, in this case, the built in `sort(..)`'s comparator is actually doing "extra work" that `mySort()` does not, in that the built-in one coerces the compared values to strings and does lexicographic comparison. The first snippet results in `[-14, 0, 0, 12, 18, 2.9, 3]` while the second snippet results (likely more accurately based on intent) in `[-14, 0, 0, 2.9, 3, 12, 18]`.

So that test is unfair because it's not actually doing the same task between the cases. Any results you get are bogus.

These same pitfalls can even be much more subtle:

```
// Case 1
var x = false;
var y = x ? 1 : 2;

// Case 2
var x;
var y = x ? 1 : 2;
```

Here, the intent might be to test the performance impact of the coercion to a Boolean that the `? :` operator will do if the `x` expression is not already a Boolean (see the *Types & Grammar* title of this book series). So, you're apparently OK with the fact that there is extra work to do the coercion in the second case.

The subtle problem? You're setting `x`'s value in the first case and not setting it in the other, so you're actually doing work in the first case that you're not doing in the second. To eliminate any potential (albeit minor) skew, try:

```
// Case 1
var x = false;
var y = x ? 1 : 2;

// Case 2
```

```js
var x = undefined;
var y = x ? 1 : 2;
```

Now there's an assignment in both cases, so the thing you want to test -- the coercion of `x` or not -- has likely been more accurately isolated and tested.

Writing Good Tests

Let me see if I can articulate the bigger point I'm trying to make here.

Good test authoring requires careful analytical thinking about what differences exist between two test cases and whether the differences between them are *intentional* or *unintentional*.

Intentional differences are of course normal and OK, but it's too easy to create unintentional differences that skew your results. You have to be really, really careful to avoid that skew. Moreover, you may intend a difference but it may not be obvious to other readers of your test what your intent was, so they may doubt (or trust!) your test incorrectly. How do you fix that?

Write better, clearer tests. But also, take the time to document (using the jsPerf.com "Description" field and/or code comments) exactly what the intent of your test is, even to the nuanced detail. Call out the intentional differences, which will help others and your future self to better identify unintentional differences that could be skewing the test results.

Isolate things which aren't relevant to your test by pre-declaring them in the page or test setup settings so they're outside the timed parts of the test.

Instead of trying to narrow in on a tiny snippet of your real code and benchmarking just that piece out of context, tests and benchmarks are better when they include a larger (while still relevant) context. Those tests also tend to run slower, which means any differences you spot are more relevant in context.

Microperformance

OK, until now we've been dancing around various microperformance issues and generally looking disfavorably upon obsessing about them. I want to take just a moment to address them directly.

The first thing you need to get more comfortable with when thinking about performance benchmarking your code is that the code you write is not always the code the engine actually runs. We briefly looked at that topic back in Chapter 1 when we discussed statement reordering by the compiler, but here we're going to suggest the compiler can sometimes decide to run different code than you wrote, not just in different orders but different in substance.

Let's consider this piece of code:

```js
var foo = 41;

(function(){
	(function(){
		(function(baz){
			var bar = foo + baz;
			// ..
		})(1);
	})();
})();
```

You may think about the `foo` reference in the innermost function as needing to do a three-level scope lookup. We covered in the *Scope & Closures* title of this book series how lexical scope works, and the fact that the compiler generally caches such lookups so that referencing `foo` from different scopes doesn't really practically "cost" anything extra.

But there's something deeper to consider. What if the compiler realizes that `foo` isn't referenced anywhere else but that one location, and it further notices that the value never is anything except the `41` as shown?

Isn't it quite possible and acceptable that the JS compiler could decide to just remove the `foo` variable entirely, and *inline* the value, such as this:

```
(function(){
	(function(){
		(function(baz){
			var bar = 41 + baz;
			// ..
		})(1);
	})();
})();
```

Note: Of course, the compiler could probably also do a similar analysis and rewrite with the `baz` variable here, too.

When you begin to think about your JS code as being a hint or suggestion to the engine of what to do, rather than a literal requirement, you realize that a lot of the obsession over discrete syntactic minutia is most likely unfounded.

Another example:

```
function factorial(n) {
	if (n < 2) return 1;
	return n * factorial( n - 1 );
}

factorial( 5 );		// 120
```

Ah, the good ol' fashioned "factorial" algorithm! You might assume that the JS engine will run that code mostly as is. And to be honest, it might -- I'm not really sure.

But as an anecdote, the same code expressed in C and compiled with advanced optimizations would result in the compiler realizing that the call `factorial(5)` can just be replaced with the constant value `120`, eliminating the function and call entirely!

Moreover, some engines have a practice called "unrolling recursion," where it can realize that the recursion you've expressed can actually be done "easier" (i.e., more optimally) with a loop. It's possible the preceding code could be *rewritten* by a JS engine to run as:

```
function factorial(n) {
	if (n < 2) return 1;

	var res = 1;
	for (var i=n; i>1; i--) {
		res *= i;
	}
	return res;
}

factorial( 5 );		// 120
```

Now, let's imagine that in the earlier snippet you had been worried about whether `n * factorial(n-1)` or `n *= factorial(--n)` runs faster. Maybe you even did a performance benchmark to try to figure out which was better. But you miss the fact that in the bigger context, the engine may not run either line of code because it may unroll the recursion!

Speaking of `--`, `--n` versus `n--` is often cited as one of those places where you can optimize by choosing the `--n` version, because theoretically it requires less effort down at the assembly level of processing.

That sort of obsession is basically nonsense in modern JavaScript. That's the kind of thing you should be letting the engine take care of. You should write the code that makes the most sense. Compare these three `for` loops:

```
// Option 1
for (var i=0; i<10; i++) {
        console.log( i );
}

// Option 2
for (var i=0; i<10; ++i) {
        console.log( i );
}

// Option 3
for (var i=-1; ++i<10; ) {
        console.log( i );
}
```

Even if you have some theory where the second or third option is more performant than the first option by a tiny bit, which is dubious at best, the third loop is more confusing because you have to start with `-1` for `i` to account for the fact that `++i` pre-increment is used. And the difference between the first and second options is really quite irrelevant.

It's entirely possible that a JS engine may see a place where `i++` is used and realize that it can safely replace it with the `++i` equivalent, which means your time spent deciding which one to pick was completely wasted and the outcome moot.

Here's another common example of silly microperformance obsession:

```
var x = [ .. ];

// Option 1
for (var i=0; i < x.length; i++) {
        // ..
}

// Option 2
for (var i=0, len = x.length; i < len; i++) {
        // ..
}
```

The theory here goes that you should cache the length of the `x` array in the variable `len`, because ostensibly it doesn't change, to avoid paying the price of `x.length` being consulted for each iteration of the loop.

If you run performance benchmarks around `x.length` usage compared to caching it in a `len` variable, you'll find that while the theory sounds nice, in practice any measured differences are statistically completely irrelevant.

In fact, in some engines like v8, it can be shown (http://mrale.ph/blog/2014/12/24/array-length-caching.html) that you could make things slightly worse by pre-caching the length instead of letting the engine figure it out for you. Don't try to outsmart your JavaScript engine, you'll probably lose when it comes to performance optimizations.

Not All Engines Are Alike

The different JS engines in various browsers can all be "spec compliant" while having radically different ways of handling code. The JS specification doesn't require anything performance related -- well, except ES6's "Tail Call Optimization" covered later in this chapter.

The engines are free to decide that one operation will receive its attention to optimize, perhaps trading off for lesser performance on another operation. It can be very tenuous to find an approach for an operation that always runs faster in all browsers.

There's a movement among some in the JS dev community, especially those who work with Node.js, to analyze the specific internal implementation details of the v8 JavaScript engine and make decisions about writing JS code that is tailored to take best advantage of how v8 works. You can actually achieve a surprisingly high degree of performance optimization with such endeavors, so the payoff for the effort can be quite high.

Some commonly cited examples (https://github.com/petkaantonov/bluebird/wiki/Optimization-killers) for v8:

- Don't pass the `arguments` variable from one function to any other function, as such "leakage" slows down the function implementation.

- Isolate a `try..catch` in its own function. Browsers struggle with optimizing any function with a `try..catch` in it, so moving that construct to its own function means you contain the de-optimization harm while letting the surrounding code be optimizable.

But rather than focus on those tips specifically, let's sanity check the v8-only optimization approach in a general sense.

Are you genuinely writing code that only needs to run in one JS engine? Even if your code is entirely intended for Node.js *right now*, is the assumption that v8 will *always* be the used JS engine reliable? Is it possible that someday a few years from now, there's another server-side JS platform besides Node.js that you choose to run your code on? What if what you optimized for before is now a much slower way of doing that operation on the new engine?

Or what if your code always stays running on v8 from here on out, but v8 decides at some point to change the way some set of operations works such that what used to be fast is now slow, and vice versa?

These scenarios aren't just theoretical, either. It used to be that it was faster to put multiple string values into an array and then call `join("")` on the array to concatenate the values than to just use `+` concatenation directly with the values. The historical reason for this is nuanced, but it has to do with internal implementation details about how string values were stored and managed in memory.

As a result, "best practice" advice at the time disseminated across the industry suggesting developers always use the array `join(..)` approach. And many followed.

Except, somewhere along the way, the JS engines changed approaches for internally managing strings, and specifically put in optimizations for `+` concatenation. They didn't slow down `join(..)` per se, but they put more effort into helping `+` usage, as it was still quite a bit more widespread.

Note: The practice of standardizing or optimizing some particular approach based mostly on its existing widespread usage is often called (metaphorically) "paving the cowpath."

Once that new approach to handling strings and concatenation took hold, unfortunately all the code out in the wild that was using array `join(..)` to concatenate strings was then sub-optimal.

Another example: at one time, the Opera browser differed from other browsers in how it handled the boxing/unboxing of primitive wrapper objects (see the *Types & Grammar* title of this book series). As such, their advice to developers was to use a `String` object instead of the primitive `string` value if properties like `length` or methods like `charAt(..)` needed to be accessed. This advice may have been correct for Opera at the time, but it was literally completely opposite for other major contemporary browsers, as they had optimizations specifically for the `string` primitives and not their object wrapper counterparts.

I think these various gotchas are at least possible, if not likely, for code even today. So I'm very cautious about making wide ranging performance optimizations in my JS code based purely on engine implementation details, **especially if those details are only true of a single engine**.

The reverse is also something to be wary of: you shouldn't necessarily change a piece of code to work around one engine's difficulty with running a piece of code in an acceptably performant way.

Historically, IE has been the brunt of many such frustrations, given that there have been plenty of scenarios in older IE versions where it struggled with some performance aspect that other major browsers of the time seemed not to have much trouble with. The string concatenation discussion we just had was actually a real concern back in the IE6 and IE7 days, where it was possible to get better performance out of `join(..)` than `+`.

But it's troublesome to suggest that just one browser's trouble with performance is justification for using a code approach that quite possibly could be sub-optimal in all other browsers. Even if the browser in question has a large market share for your site's audience, it may be more practical to write the proper code and rely on the browser to update itself with better optimizations eventually.

"There is nothing more permanent than a temporary hack." Chances are, the code you write now to work around some performance bug will probably outlive the performance bug in the browser itself.

In the days when a browser only updated once every five years, that was a tougher call to make. But as it stands now, browsers across the board are updating at a much more rapid interval (though obviously the mobile world still lags), and they're all competing to optimize web features better and better.

If you run across a case where a browser *does* have a performance wart that others don't suffer from, make sure to report it to them through whatever means you have available. Most browsers have open public bug trackers suitable for this purpose.

Tip: I'd only suggest working around a performance issue in a browser if it was a really drastic show-stopper, not just an annoyance or frustration. And I'd be very careful to check that the performance hack didn't have noticeable negative side effects in another browser.

Big Picture

Instead of worrying about all these microperformance nuances, we should instead be looking at big-picture types of optimizations.

How do you know what's big picture or not? You have to first understand if your code is running on a critical path or not. If it's not on the critical path, chances are your optimizations are not worth much.

Ever heard the admonition, "that's premature optimization!"? It comes from a famous quote from Donald Knuth: "premature optimization is the root of all evil.". Many developers cite this quote to suggest that most optimizations are "premature" and are thus a waste of effort. The truth is, as usual, more nuanced.

Here is Knuth's quote, in context:

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of **noncritical** parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that **critical** 3%. [emphasis added]

(http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf, Computing Surveys, Vol 6, No 4, December 1974)

I believe it's a fair paraphrasing to say that Knuth *meant*: "non-critical path optimization is the root of all evil." So the key is to figure out if your code is on the critical path -- you should optimize it! -- or not.

I'd even go so far as to say this: no amount of time spent optimizing critical paths is wasted, no matter how little is saved; but no amount of optimization on noncritical paths is justified, no matter how much is saved.

If your code is on the critical path, such as a "hot" piece of code that's going to be run over and over again, or in UX critical places where users will notice, like an animation loop or CSS style updates, then you should spare no effort in trying to employ relevant, measurably significant optimizations.

For example, consider a critical path animation loop that needs to coerce a string value to a number. There are of course multiple ways to do that (see the *Types & Grammar* title of this book series), but which one if any is the fastest?

```
var x = "42";   // need number `42`

// Option 1: let implicit coercion automatically happen
var y = x / 2;

// Option 2: use `parseInt(..)`
var y = parseInt( x, 0 ) / 2;

// Option 3: use `Number(..)`
var y = Number( x ) / 2;

// Option 4: use `+` unary operator
var y = +x / 2;

// Option 5: use `|` unary operator
var y = (x | 0) / 2;
```

Note: I will leave it as an exercise to the reader to set up a test if you're interested in examining the minute differences in performance among these options.

When considering these different options, as they say, "One of these things is not like the others." `parseInt(..)` does the job, but it also does a lot more -- it parses the string rather than just coercing. You can probably guess, correctly, that `parseInt(..)` is a slower option, and you should probably avoid it.

Of course, if `x` can ever be a value that **needs parsing**, such as `"42px"` (like from a CSS style lookup), then `parseInt(..)` really is the only suitable option!

`Number(..)` is also a function call. From a behavioral perspective, it's identical to the `+` unary operator option, but it may in fact be a little slower, requiring more machinery to execute the function. Of course, it's also possible that the JS engine recognizes this behavioral symmetry and just handles the inlining of `Number(..)` 's behavior (aka `+x`) for you!

But remember, obsessing about `+x` versus `x | 0` is in most cases likely a waste of effort. This is a microperformance issue, and one that you shouldn't let dictate/degrade the readability of your program.

While performance is very important in critical paths of your program, it's not the only factor. Among several options that are roughly similar in performance, readability should be another important concern.

Tail Call Optimization (TCO)

As we briefly mentioned earlier, ES6 includes a specific requirement that ventures into the world of performance. It's related to a specific form of optimization that can occur with function calls: *tail call optimization*.

Briefly, a "tail call" is a function call that appears at the "tail" of another function, such that after the call finishes, there's nothing left to do (except perhaps return its result value).

For example, here's a non-recursive setup with tail calls:

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

function bar(y) {
	return foo( y + 1 );	// tail call
}

function baz() {
	return 1 + bar( 40 );	// not tail call
}

baz();						// 42
```

`foo(y+1)` is a tail call in `bar(..)` because after `foo(..)` finishes, `bar(..)` is also finished except in this case returning the result of the `foo(..)` call. However, `bar(40)` is *not* a tail call because after it completes, its result value must be added to `1` before `baz()` can return it.

Without getting into too much nitty-gritty detail, calling a new function requires an extra amount of reserved memory to manage the call stack, called a "stack frame." So the preceding snippet would generally require a stack frame for each of `baz()` , `bar(..)` , and `foo(..)` all at the same time.

However, if a TCO-capable engine can realize that the `foo(y+1)` call is in *tail position* meaning `bar(..)` is basically complete, then when calling `foo(..)` , it doesn't need to create a new stack frame, but can instead reuse the existing stack frame from `bar(..)` . That's not only faster, but it also uses less memory.

That sort of optimization isn't a big deal in a simple snippet, but it becomes a *much bigger deal* when dealing with recursion, especially if the recursion could have resulted in hundreds or thousands of stack frames. With TCO the engine can perform all those calls with a single stack frame!

Recursion is a hairy topic in JS because without TCO, engines have had to implement arbitrary (and different!) limits to how deep they will let the recursion stack get before they stop it, to prevent running out of memory. With TCO, recursive functions with *tail position* calls can essentially run unbounded, because there's never any extra usage of memory!

Consider that recursive `factorial(..)` from before, but rewritten to make it TCO friendly:

```
function factorial(n) {
	function fact(n,res) {
		if (n < 2) return res;

		return fact( n - 1, n * res );
	}

	return fact( n, 1 );
}

factorial( 5 );		// 120
```

This version of `factorial(..)` is still recursive, but it's also optimizable with TCO, because both inner `fact(..)` calls are in *tail position*.

Note: It's important to note that TCO only applies if there's actually a tail call. If you write recursive functions without tail calls, the performance will still fall back to normal stack frame allocation, and the engines' limits on such recursive call stacks will still apply. Many recursive functions can be rewritten as we just showed with `factorial(..)`, but it takes careful attention to detail.

One reason that ES6 requires engines to implement TCO rather than leaving it up to their discretion is because the *lack of TCO* actually tends to reduce the chances that certain algorithms will be implemented in JS using recursion, for fear of the call stack limits.

If the lack of TCO in the engine would just gracefully degrade to slower performance in all cases, it wouldn't probably have been something that ES6 needed to *require*. But because the lack of TCO can actually make certain programs impractical, it's more an important feature of the language than just a hidden implementation detail.

ES6 guarantees that from now on, JS developers will be able to rely on this optimization across all ES6+ compliant browsers. That's a win for JS performance!

Review

Effectively benchmarking performance of a piece of code, especially to compare it to another option for that same code to see which approach is faster, requires careful attention to detail.

Rather than rolling your own statistically valid benchmarking logic, just use the Benchmark.js library, which does that for you. But be careful about how you author tests, because it's far too easy to construct a test that seems valid but that's actually flawed -- even tiny differences can skew the results to be completely unreliable.

It's important to get as many test results from as many different environments as possible to eliminate hardware/device bias. jsPerf.com is a fantastic website for crowdsourcing performance benchmark test runs.

Many common performance tests unfortunately obsess about irrelevant microperformance details like `x++` versus `++x`. Writing good tests means understanding how to focus on big picture concerns, like optimizing on the critical path, and avoiding falling into traps like different JS engines' implementation details.

Tail call optimization (TCO) is a required optimization as of ES6 that will make some recursive patterns practical in JS where they would have been impossible otherwise. TCO allows a function call in the *tail position* of another function to execute without needing any extra resources, which means the engine no longer needs to place arbitrary restrictions on call stack depth for recursive algorithms.