Are you are interested in making JavaScript run blazingly fast in Chrome? This talk takes a look under the hood in V8 to help you identify how to optimize your JavaScript code.

We'll show you how to leverage V8's sampling profiler to eliminate performance bottlenecks and optimize JavaScript programs, and we'll expose how V8 uses hidden classes and runtime type feedback to generate efficient JIT code.

Attendees will leave the session with solid optimization guidelines for their JavaScript app and a good understanding on how to best use performance tools and JavaScript idioms to maximize the performance of their application with V8.

Performance optimization...

Today, we're going to talk about

Raw JavaScript Execution Performance

with


Krefeld
Düsseldorf-Nord
Büttgen
57
57
Heinsberg
Aachen
NE-Holzheim
46
NE-Zentrum


NE-Zentrum
source: autobahn image from iStockphoto

Who cares about performance?

You should.

JavaScript

Performance Matters


It's not just about making
your application run faster now

The Optimization Checklist

- Be prepared before you have a problem
- Identify and understand the crux of your problem
- Fix what matters


Performance
Problem

A JavaScript Performance Problem

The Problem

Compute the 25,000th prime

The Algorithm

For x = 1 to infinity: if x not divisible by any member of an initially empty list of primes, add x to the list until we have 25,000

The Contenders

C++

```
class Primes {
 public:
  int getPrimeCount() const { return prime_count; }
  int getPrime(int i) const { return primes[i]; }
  void addPrime(int i) { primes[prime_count++] = i; }

  bool isDivisibe(int i, int by) { return (i % by) == 0; }

  bool isPrimeDivisible(int candidate) {
    for (int i = 1; i < prime_count; ++i) {
      if (isDivisibe(candidate, primes[i])) return true;
    }
    return false;
  }

 private:
  volatile int prime_count;
  volatile int primes[25000];
};

int main() {
  Primes p;
  int c = 1;
  while (p.getPrimeCount() < 25000) {
    if (!p.isPrimeDivisible(c)) {
      p.addPrime(c);
    }
    c++;
  }
  printf("%d\n", p.getPrime(p.getPrimeCount()-1));
}
```

JAVASCRIPT

```
function Primes() {
  this.prime_count = 0;
  this.primes = new Array(25000);
  this.getPrimeCount = function() { return this.prime_count; }
  this.getPrime = function(i) { return this.primes[i]; }
  this.addPrime = function(i) {
    this.primes[this.prime_count++] = i;
  }

  this.isPrimeDivisible = function(candidate) {
    for (var i = 1; i <= this.prime_count; ++i) {
      if ((candidate % this.primes[i]) == 0) return true;
    }
    return false;
  }
};

function main() {
  p = new Primes();
  var c = 1;
  while (p.getPrimeCount() < 25000) {
    if (!p.isPrimeDivisible(c)) {
      p.addPrime(c);
    }
    c++;
  }
  print(p.getPrime(p.getPrimeCount()-1));
}

main();
```

The Results

C++

SHELL

```
% g++ primes.cc -o primes
% time ./primes
287107

real    0m2.955s
user    0m2.952s
sys     0m0.001s
```

JavaScript

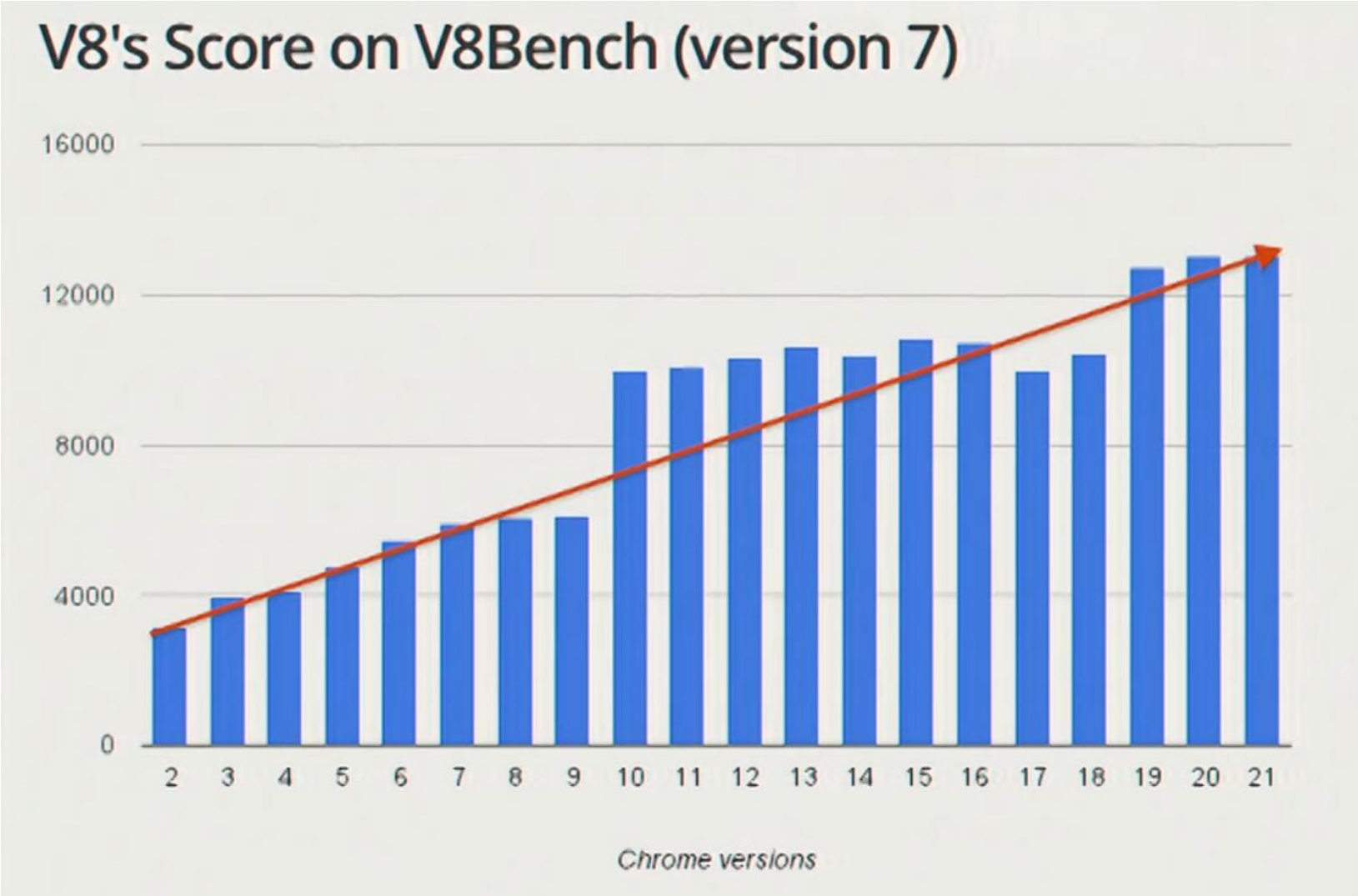
SHELL

```
% time d8 primes.js
287107

real    0m15.584s
user    0m15.612s
sys     0m0.073s
```

I guess that's not so bad, right?

Of course it is "so bad"!


V8's Score on V8Bench (version 7)
16000
12000
8000
4000
0
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
Chrome versions

Don't give up here!

How much faster?

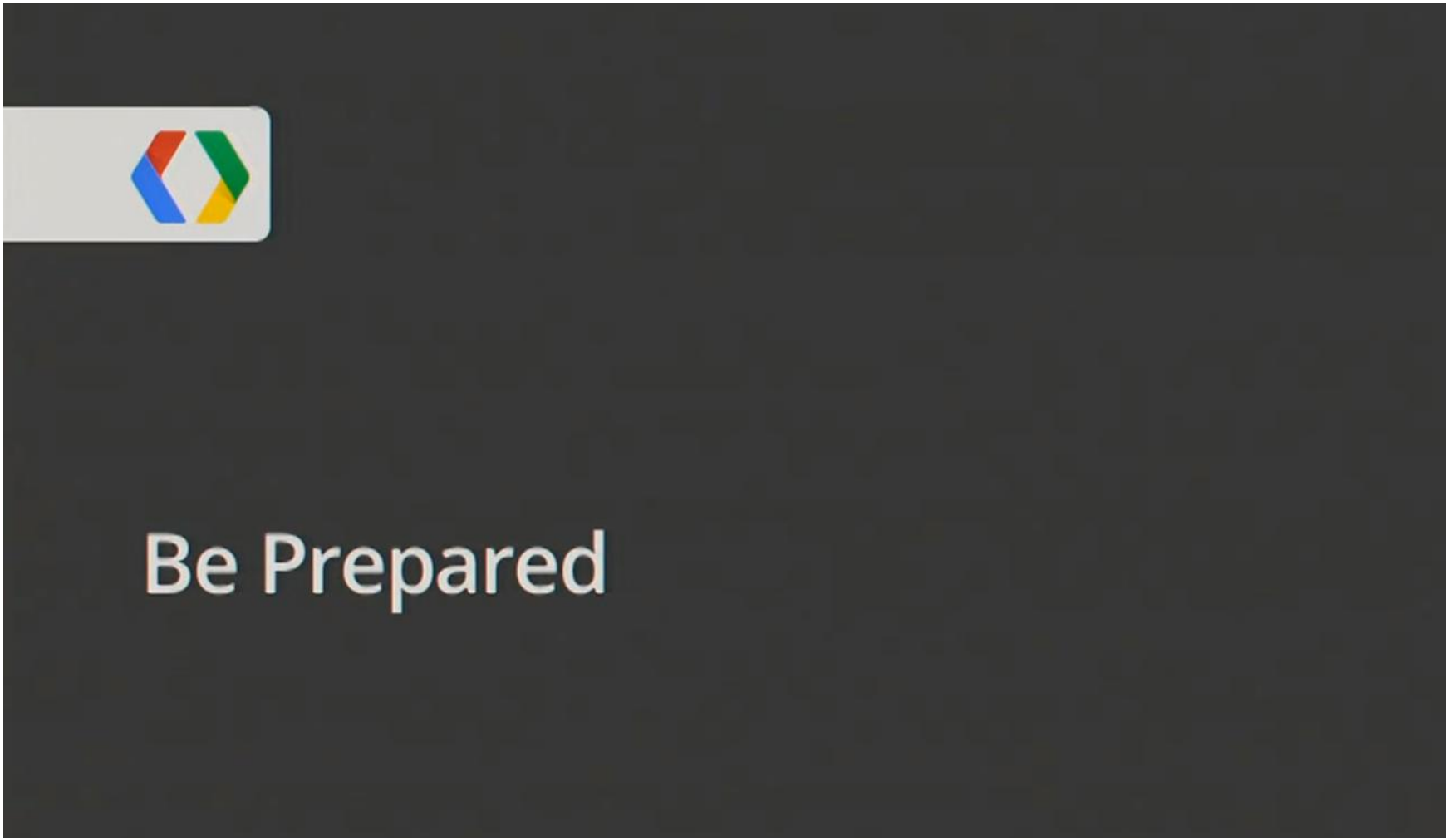
3.5x?

35x?

350x?

3500x?

Let's apply the checklist
to see how much faster we can get


Be Prepared

What does "Be Prepared" mean for V8?

- Understand how V8 optimizes JavaScript
- Write code mindfully
- Learn your tools and how they can help you

Be Prepared - Hidden Classes

Limited Compile-Time Type Information

It's expensive to reason about JavaScript types at compile time...

Limited Compile-Time Type Information

so how can JavaScript performance get anywhere close to C++?

Hidden Classes Make JavaScript Faster

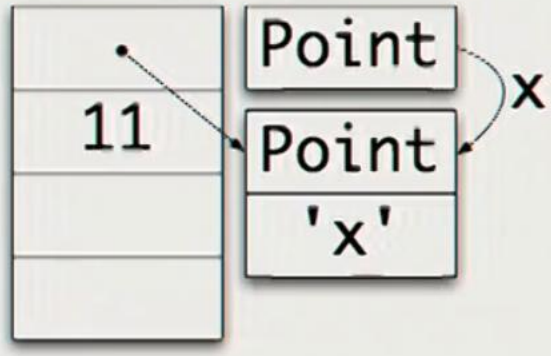
- V8 internally creates **hidden classes** for objects at runtime
- Objects with the same hidden class can use the same optimized generated code

Hidden Classes for Properties

JAVASCRIPT

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
```

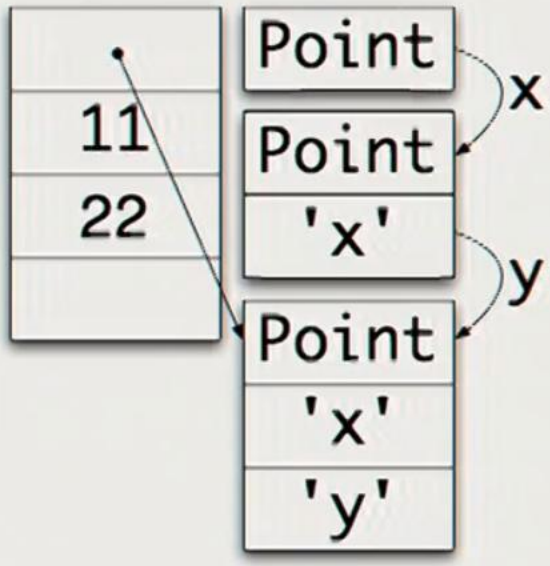


Hidden Classes for Properties

JAVASCRIPT

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
```

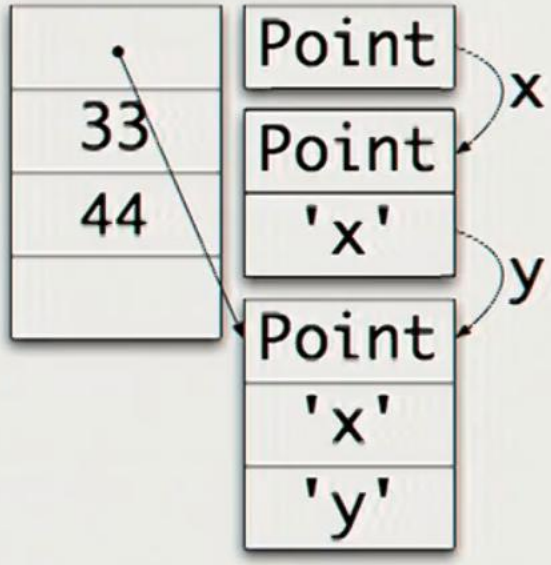


Hidden Classes for Properties

JAVASCRIPT

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
```

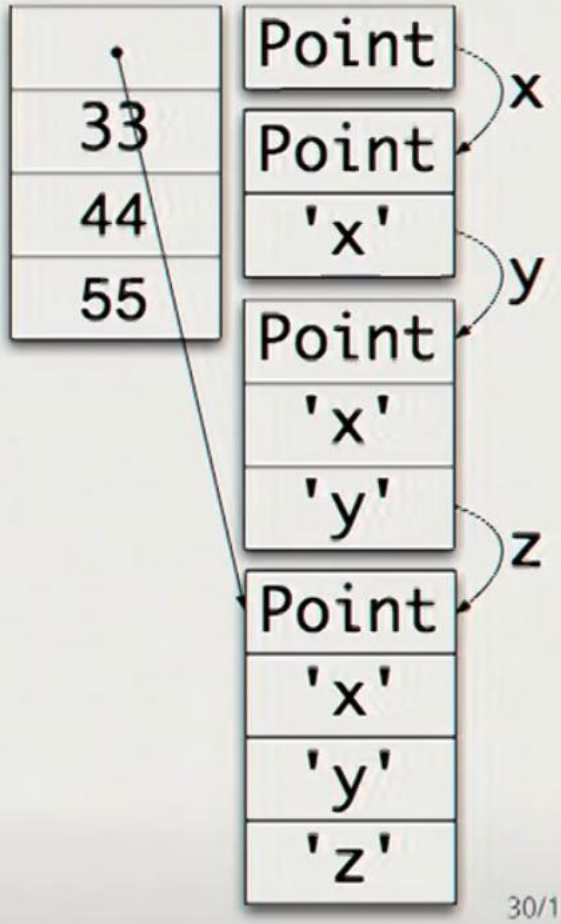


Hidden Classes for Properties

JAVASCRIPT

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
p2.z = 55;
```

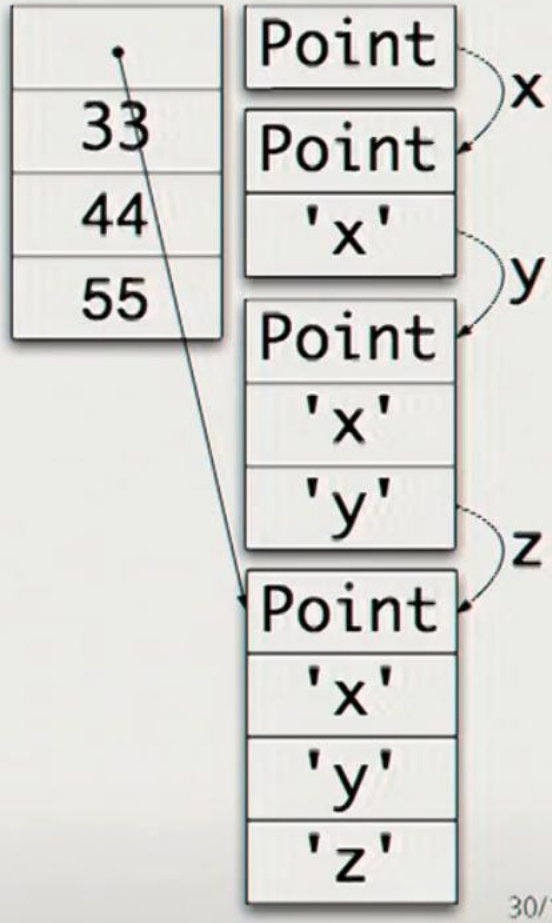


Hidden Classes for Properties

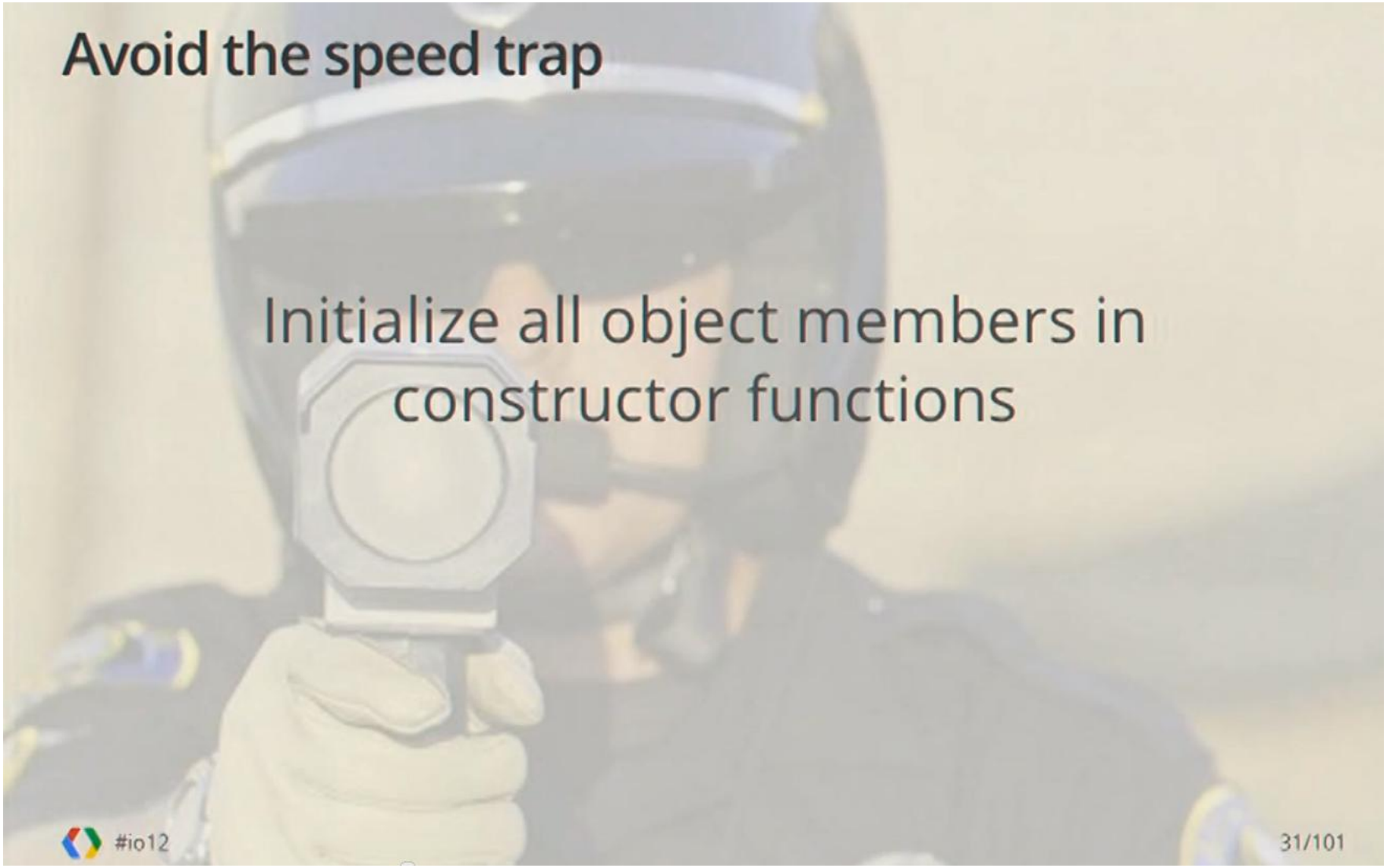
JAVASCRIPT

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
p2.z = 55
// warning! p1 and p2 now have
// different hidden classes
```

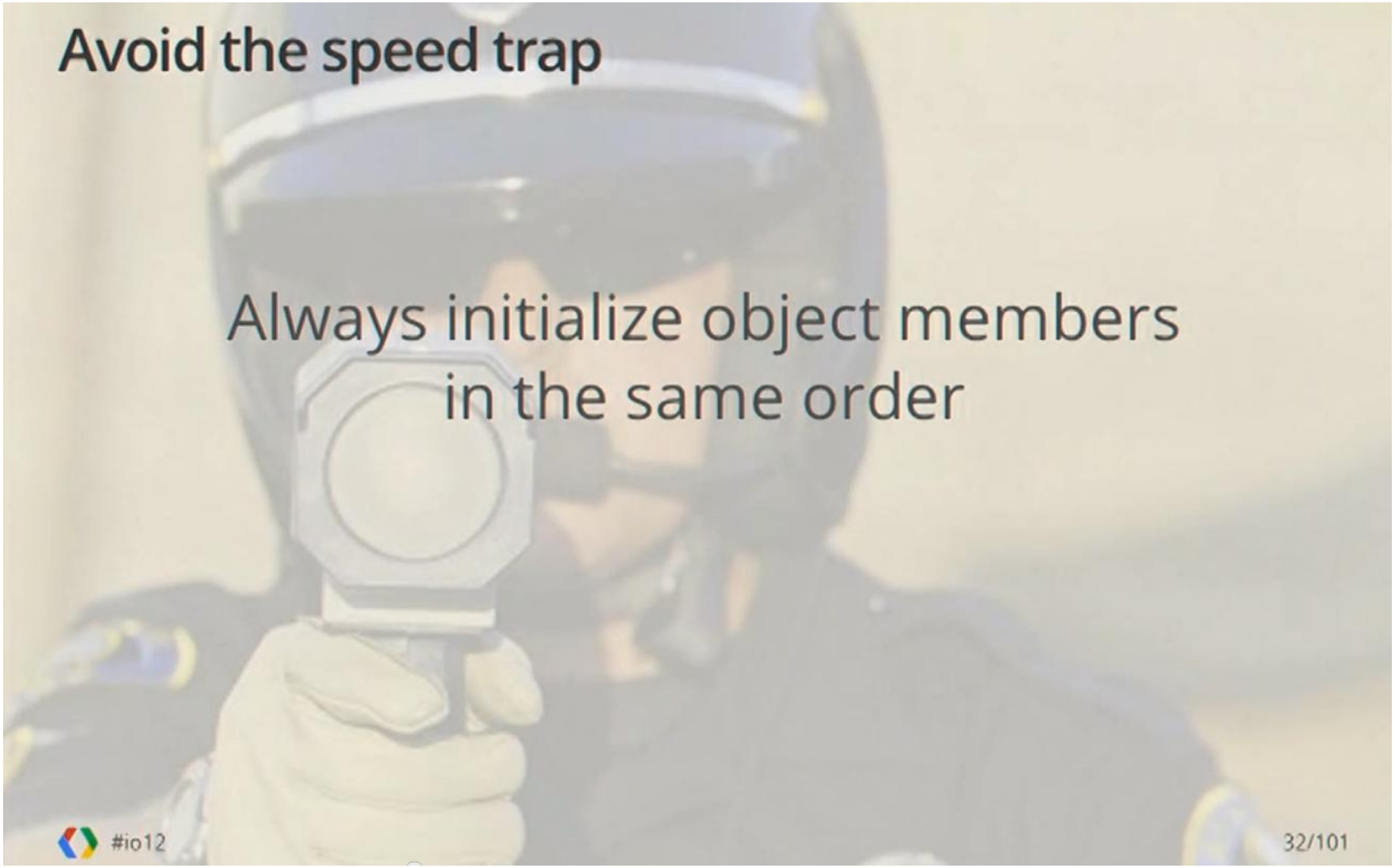


Avoid the speed trap

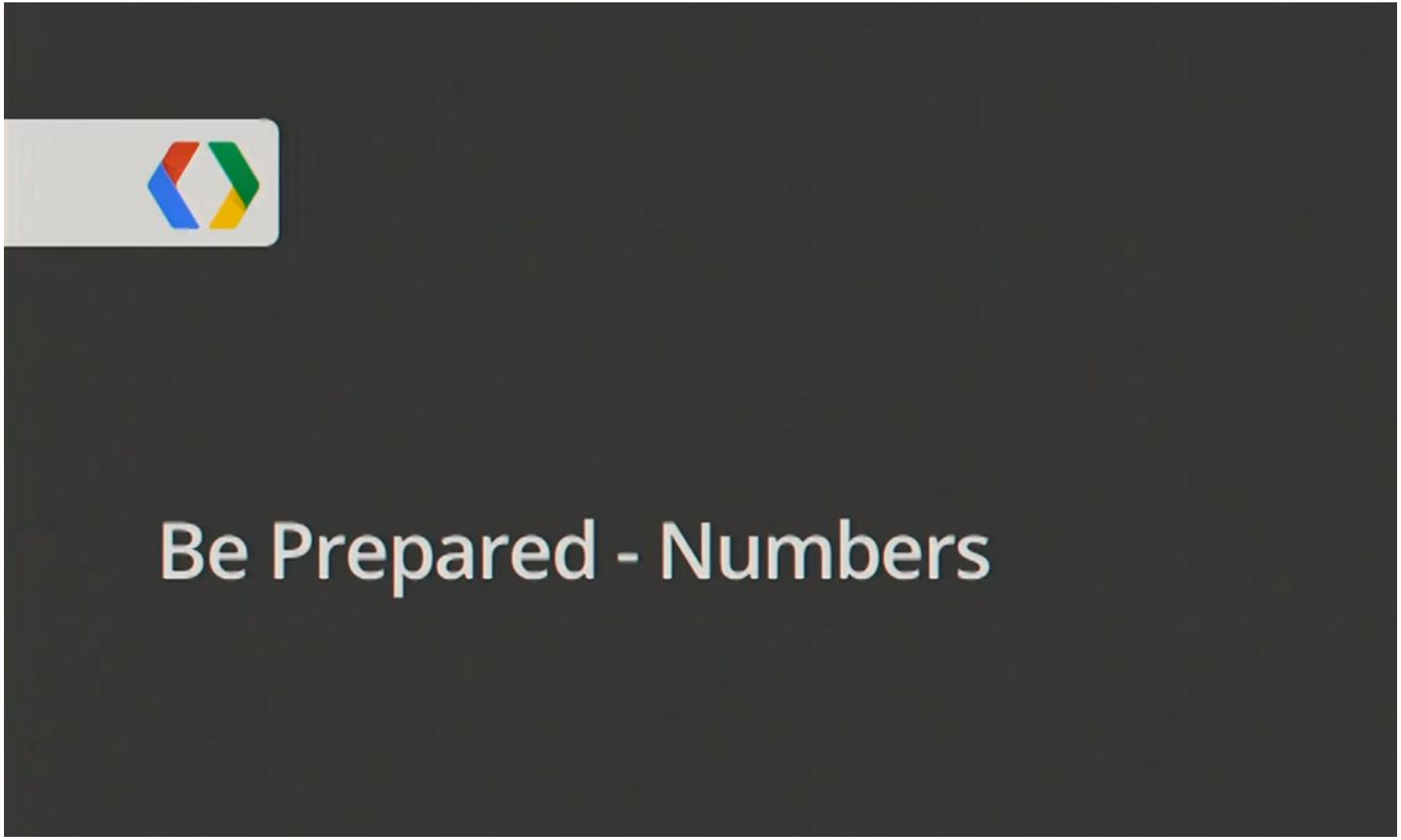


Initialize all object members in constructor functions

Avoid the speed trap



Always initialize object members
in the same order


Be Prepared - Numbers

Efficiently Representing Values

How does V8 represent values efficiently if types can change?

Tagged Values

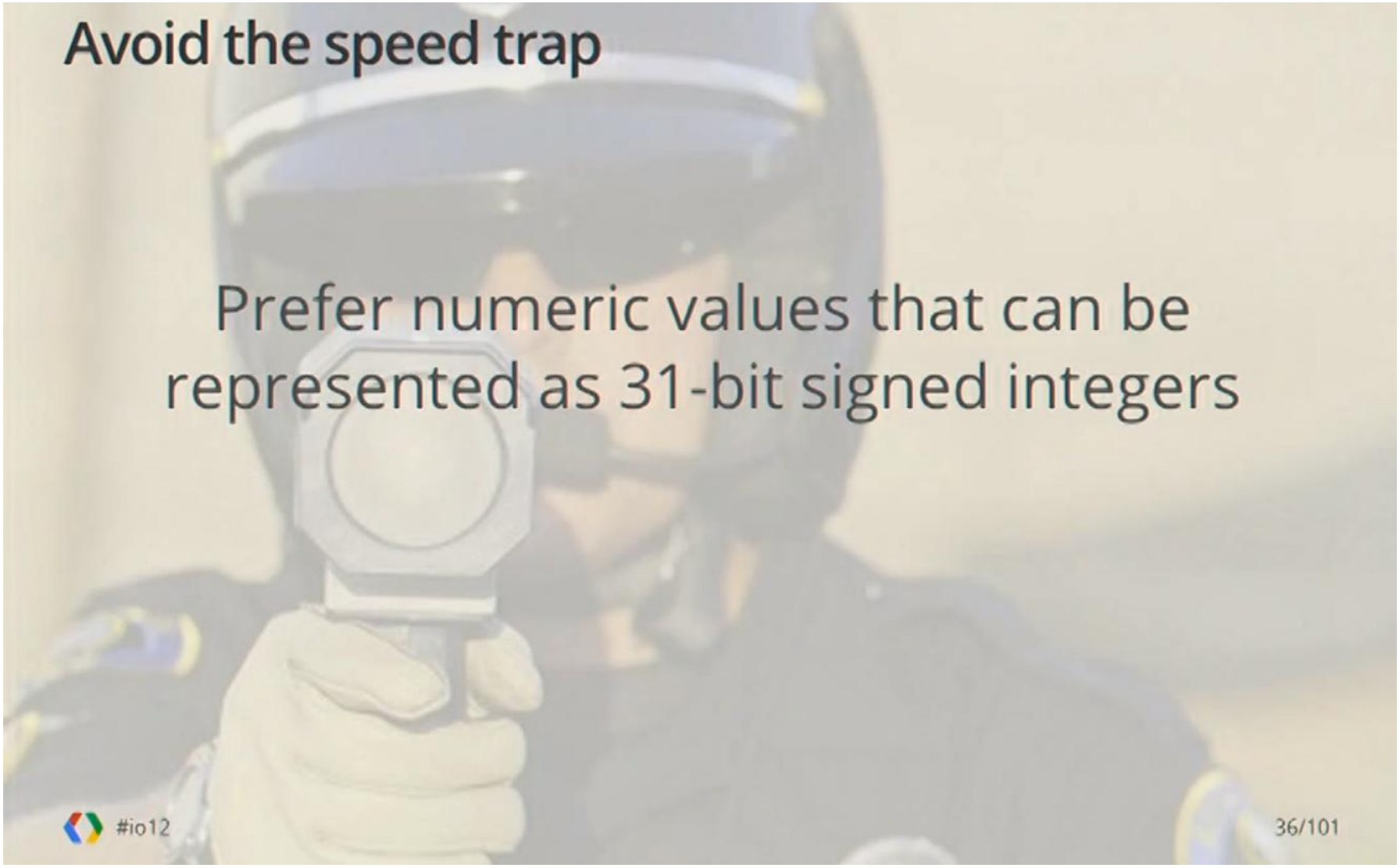
V8 uses tagging

- Objects

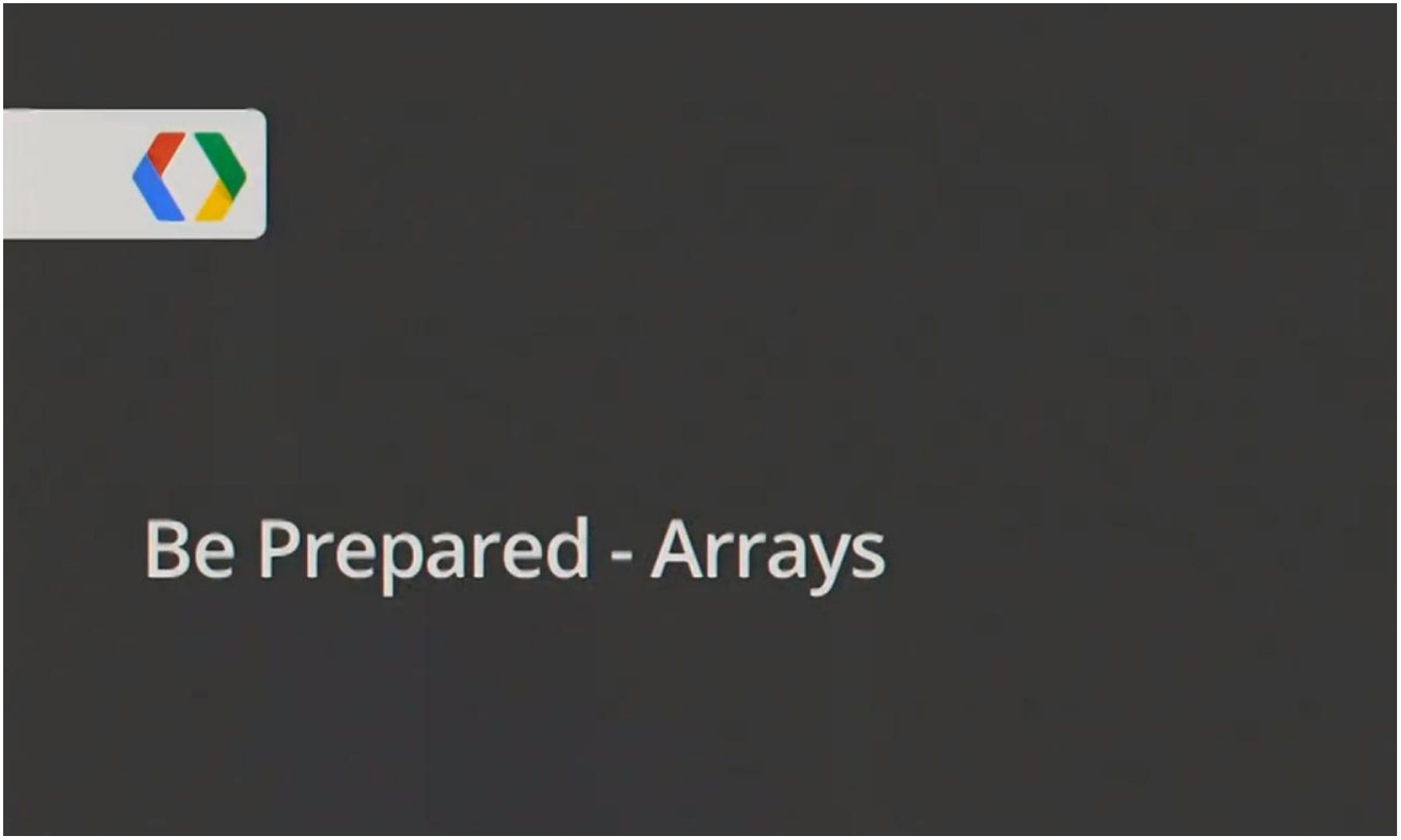
- SMall Integers

- Boxed double

Avoid the speed trap



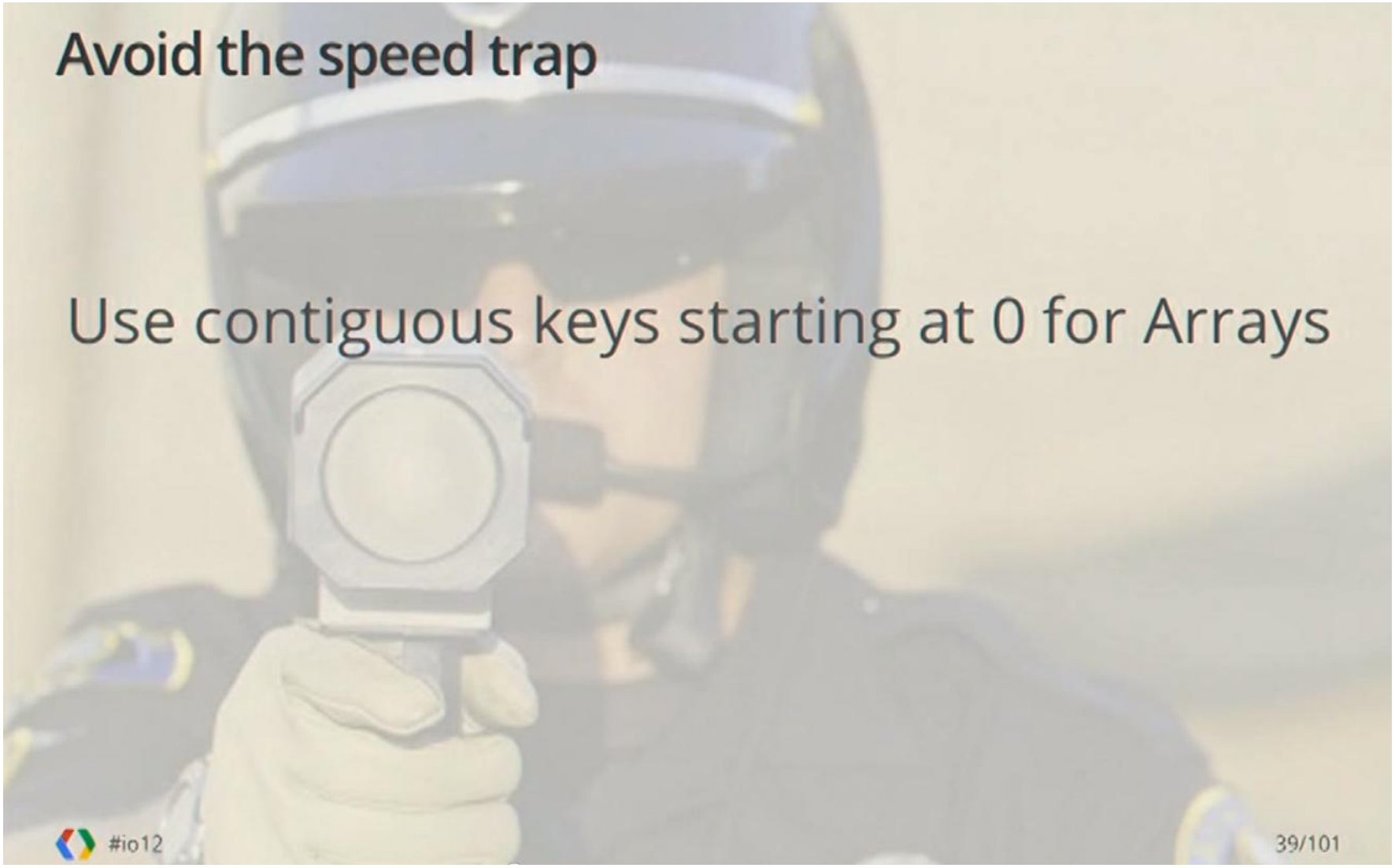
Prefer numeric values that can be represented as 31-bit signed integers


Be Prepared - Arrays

Handling Large and Sparse Arrays

- Fast Elements: linear storage for compact key sets
- Dictionary Elements: hash table storage otherwise

Avoid the speed trap

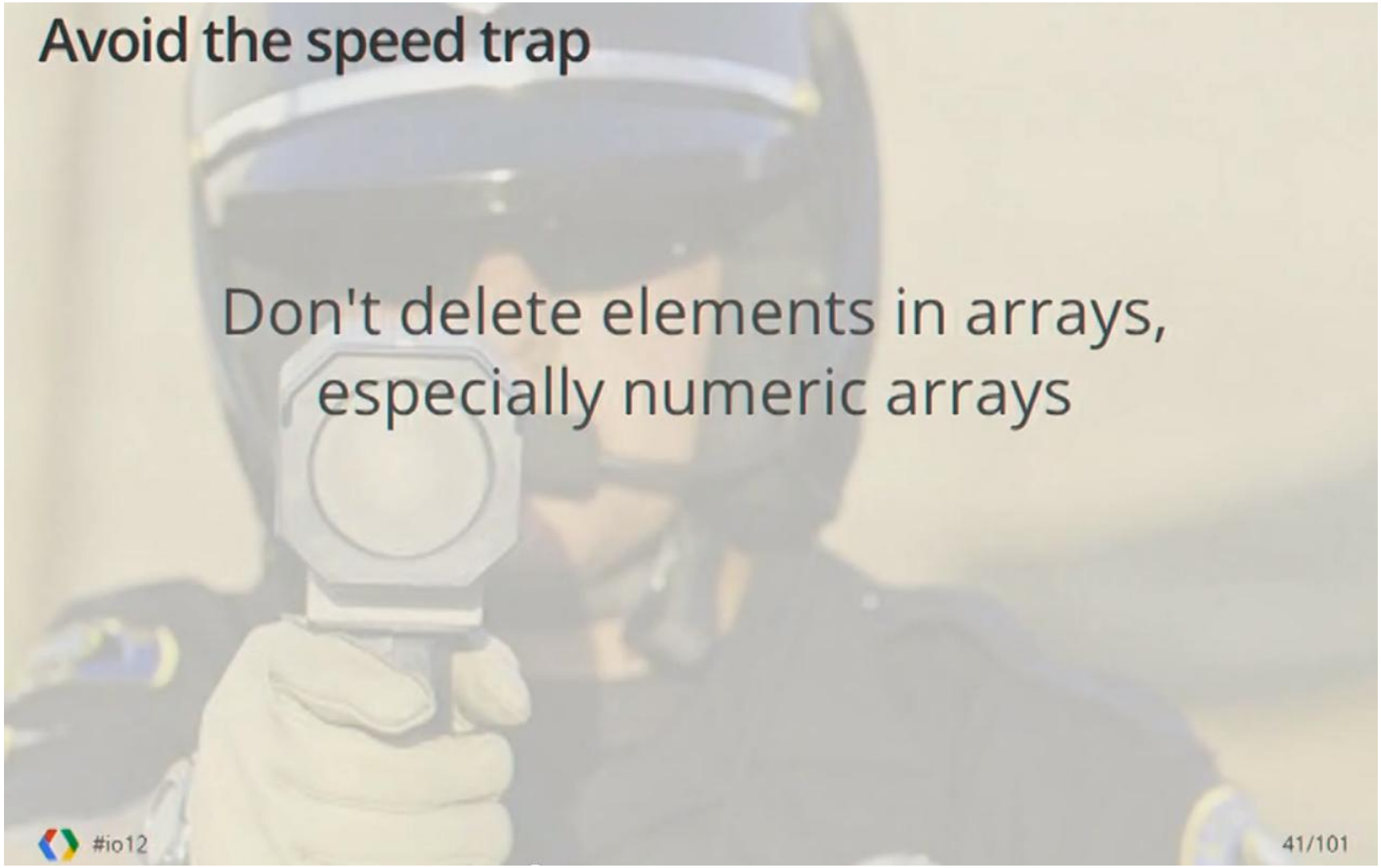


Use contiguous keys starting at 0 for Arrays

Avoid the speed trap

Don't pre-allocate large Arrays (e.g. > 64K elements) to their maximum size, instead grow as you go

Avoid the speed trap



JAVASCRIPT

```
a = new Array();
for (var b = 0; b < 10; b++) {
  a[0] |= b;  // Oh no!
}
```

vs.

JAVASCRIPT

```
a = new Array();
a[0] = 0;
for (var b = 0; b < 10; b++) {
  a[0] |= b;  // Much better! 2x faster.
}
```

But aren't Arrays of doubles slow because of tagging?

Double Array Unboxing

- Array's hidden class tracks element types
- Arrays containing only doubles are **unboxed**
- Unboxing causes hidden class change

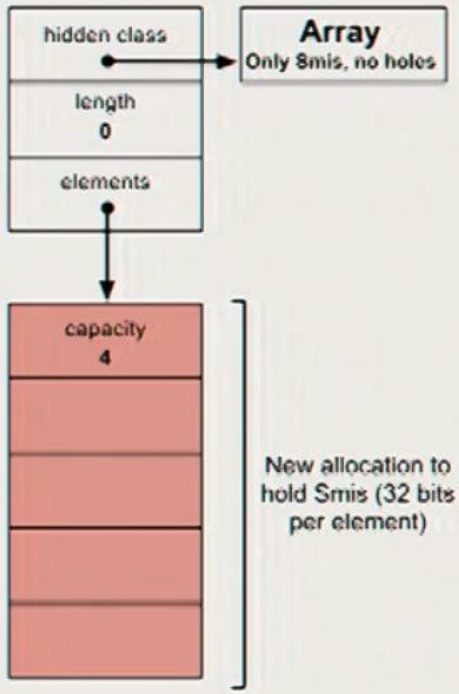
Careless manipulation of Arrays can cause extra work due to boxing and unboxing

Hidden Classes for Elements - Trouble Ahead

JAVASCRIPT

```
var a = new Array();

a[0] = 77;    // Allocates
a[1] = 88;
a[2] = 0.5;    // Allocates, converts
a[3] = true; // Allocates, converts
```

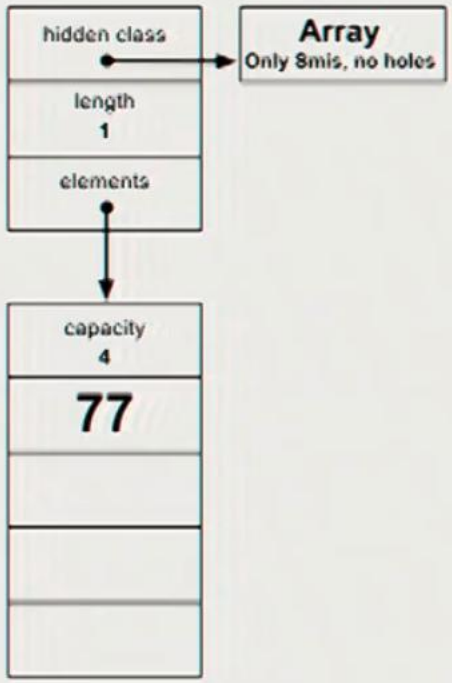


Hidden Classes for Elements - Trouble Ahead

JAVASCRIPT

```
var a = new Array();

a[0] = 77;    // Allocates
a[1] = 88;
a[2] = 0.5;   // Allocates, converts
a[3] = true; // Allocates, converts
```

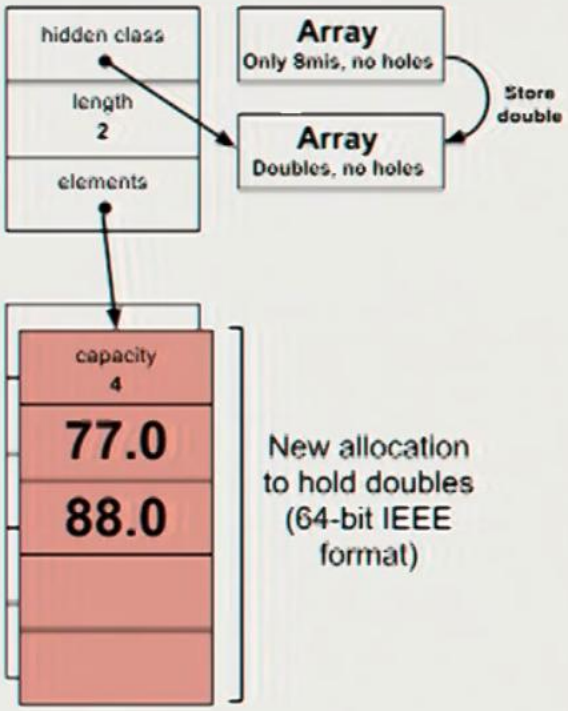


Hidden Classes for Elements - Trouble Ahead

JAVASCRIPT

```
var a = new Array();

a[0] = 77;    // Allocates
a[1] = 88;
a[2] = 0.5;    // Allocates, converts
a[3] = true; // Allocates, converts
```

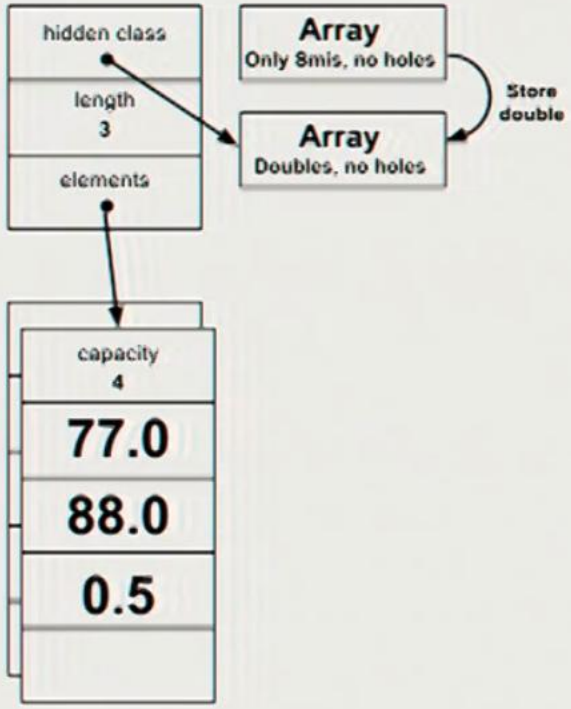


Hidden Classes for Elements - Trouble Ahead

JAVASCRIPT

```
var a = new Array();

a[0] = 77;    // Allocates
a[1] = 88;
a[2] = 0.5;    // Allocates, converts
a[3] = true; // Allocates, converts
```

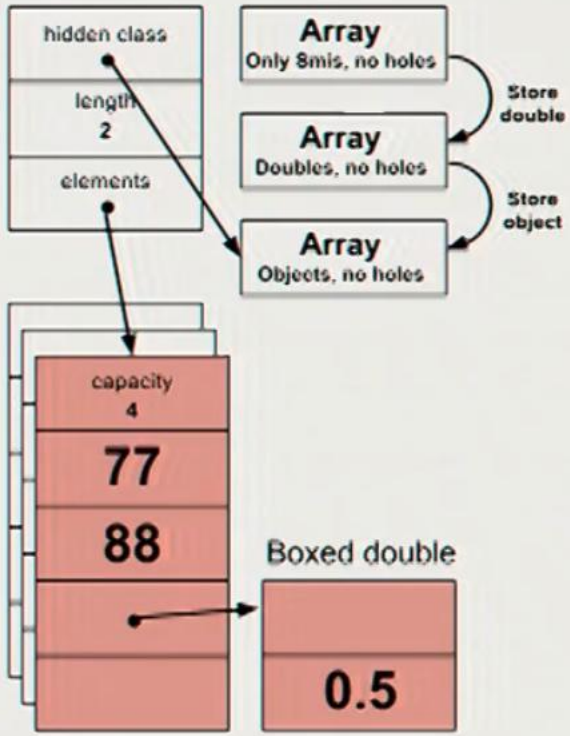


Hidden Classes for Elements - Trouble Ahead

JAVASCRIPT

```
var a = new Array();

a[0] = 77;    // Allocates
a[1] = 88;
a[2] = 0.5;    // Allocates, converts
a[3] = true; // Allocates, converts
```

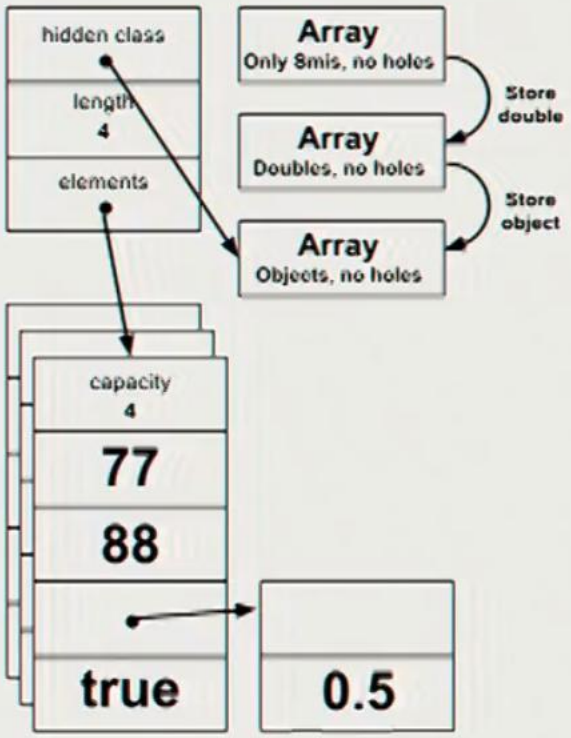


Hidden Classes for Elements - Trouble Ahead

JAVASCRIPT

```
var a = new Array();

a[0] = 77;    // Allocates
a[1] = 88;
a[2] = 0.5;    // Allocates, converts
a[3] = true; // Allocates, converts
```

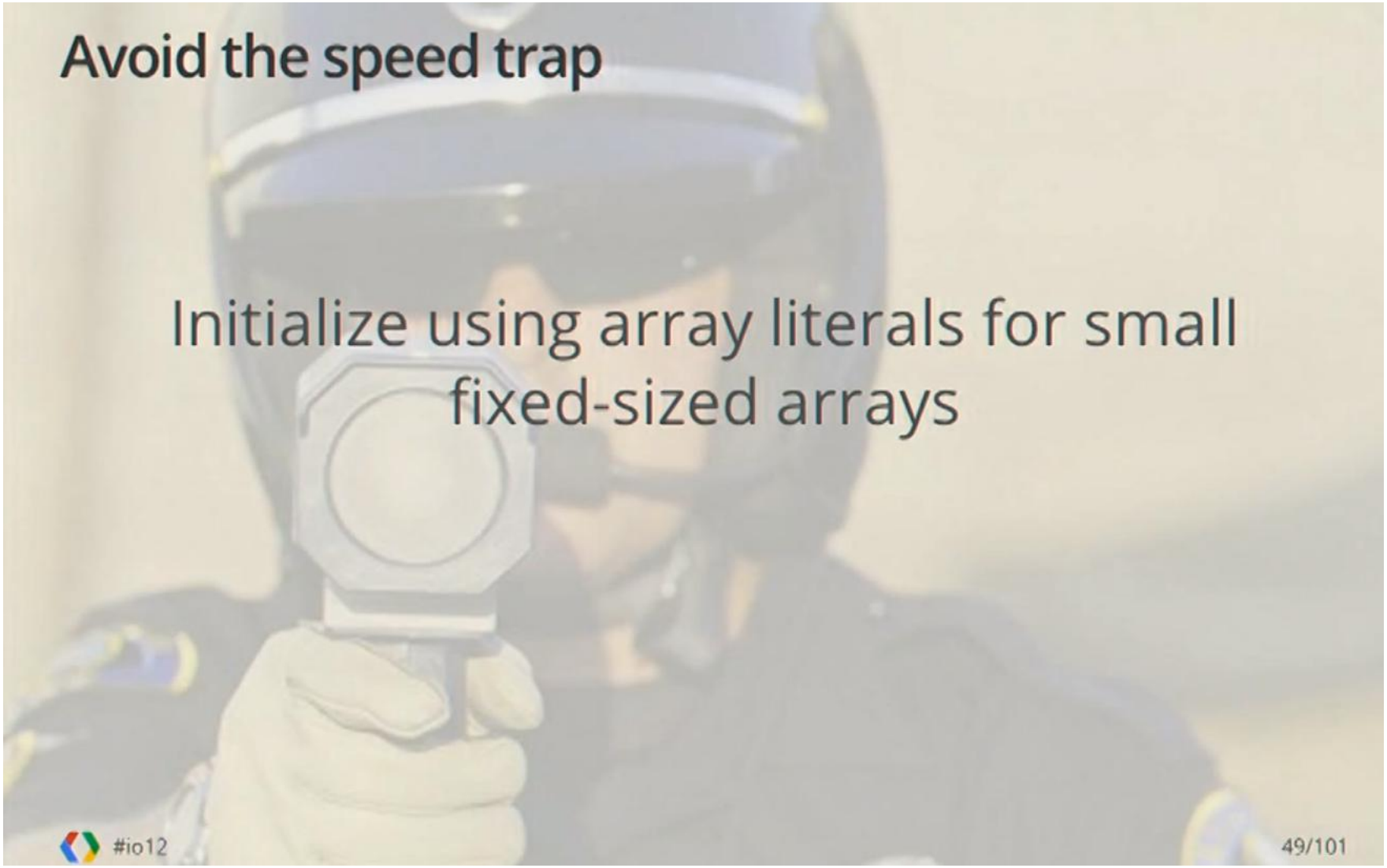


Hidden Classes for Elements - A Better Way

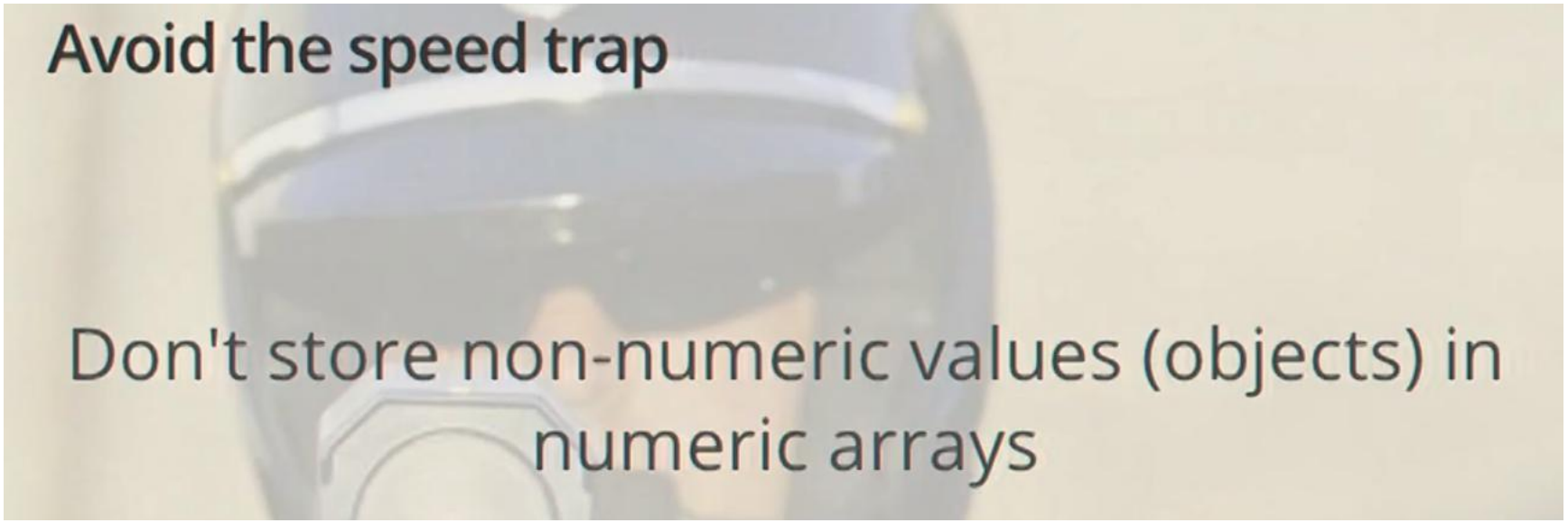
JAVASCRIPT

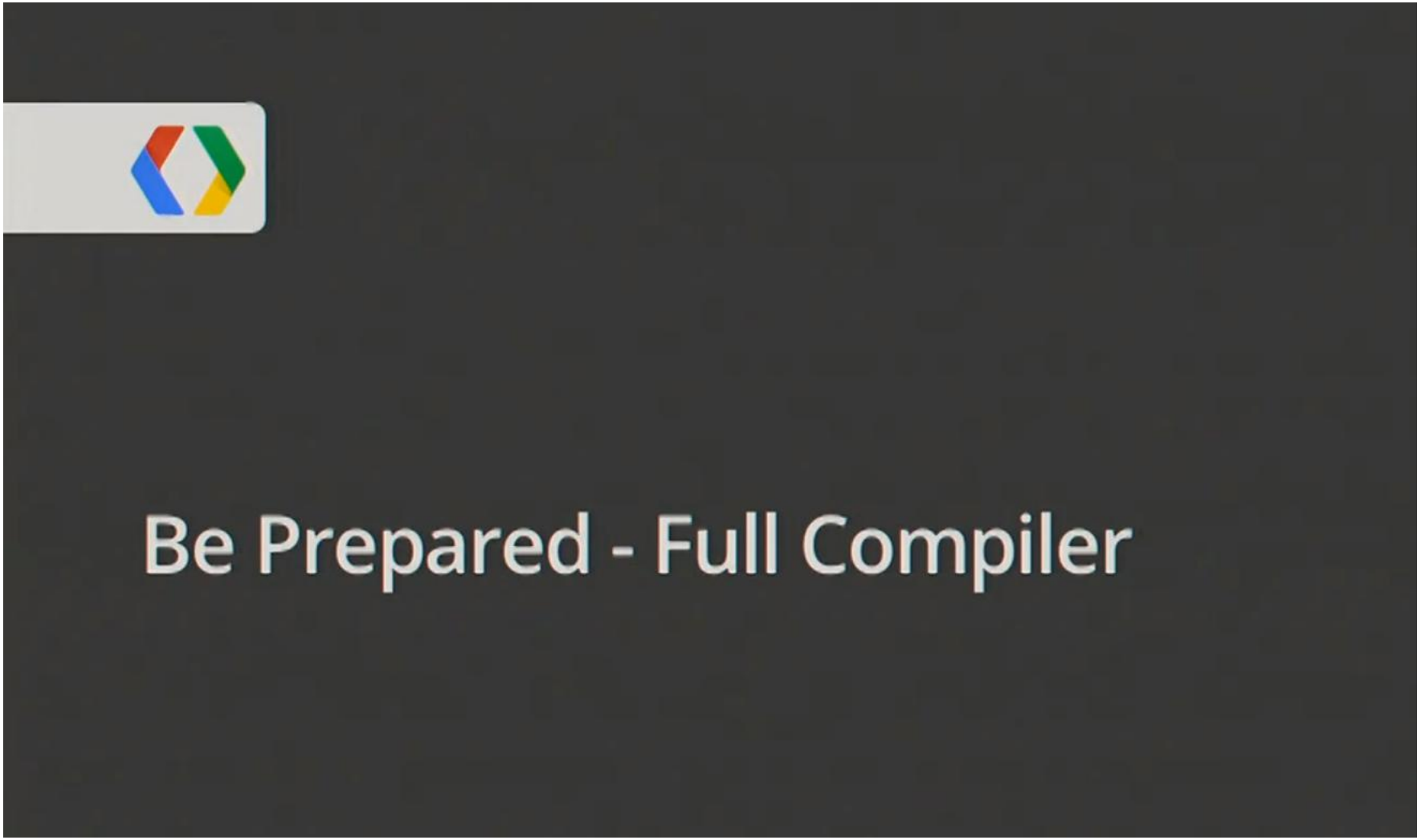
```
var a = [77, 88, 0.5, true];
```

Avoid the speed trap



Initialize using array literals for small fixed-sized arrays


Avoid the speed trap
Don't store non-numeric values (objects) in numeric arrays


Be Prepared - Full Compiler

V8 Has Two Compilers

- "Full" compiler can generate good code for any JavaScript
- Optimizing compiler produces great code for most JavaScript

"Full" Compiler Starts Executing Code ASAP

- Quickly generates good but not great JIT code
- Assumes (almost) nothing about types at compilation time
- Uses **Inline Caches** (or **ICs**) to refine knowledge about types while program runs

Inline Caches (ICs) Handle Types Efficiently

- Type-dependent code for operations
- Validate type assumptions first, then do work
- Change at runtime via backpatching as more types are discovered

Full Compiler Example

A fragment from the example:

JAVASCRIPT

```
this.isPrimeDivisible = function(candidate) {
  for (var i = 1; i <= this.prime_count; ++i) {
    if (candidate % this.primes[i] == 0) return true;
  }
  return false;
}
```

Full Compiler Example

A fragment from the example:

JAVASCRIPT

```
this.isPrimeDivisible = function(candidate) {
  for (var i = 1; i <= this.prime_count; ++i) {
    if (candidate % this.primes[i] == 0) return true;
  }
  return false;
}
```

Generating Code at Full Compile Time

```
candidate % this.primes[i]
```

IA32 ASSEMBLY

```
...
push [ebp+0x8]
mov eax,[ebp+0xc]
mov edx,eax
mov ecx,0x50b155dd
call LoadIC_Initialize          ;; this.primes
```

Generating Code at Full Compile Time

```
candidate % this.primes[i]
```

IA32 ASSEMBLY

```
...
push [ebp+0x8]
mov eax,[ebp+0xc]
mov edx,eax
mov ecx,0x50b155dd
call LoadIC_Initialize          ;; this.primes
push eax
mov eax,[ebp+0xf4]
pop edx
mov ecx,eax
call KeyedLoadIC_Initialize     ;; this.primes[i]
```

Generating Code at Full Compile Time

```
candidate % this.primes[i]
```

IA32 ASSEMBLY

```
...
push [ebp+0x8]
mov eax,[ebp+0xc]
mov edx,eax
mov ecx,0x50b155dd
call LoadIC_Initialize           ;; this.primes
push eax
mov eax,[ebp+0xf4]
pop edx
mov ecx,eax
call KeyedLoadIC_Initialize      ;; this.primes[i]
pop edx
call BinaryOpIC_Initialize Mod   ;; candidate % this.primes[i]
```

Backpatching Inline Caches at Runtime

IA32 ASSEMBLY

```
...
push [ebp+0x8]
mov eax,[ebp+0xc]
mov edx,eax
mov ecx,0x50b155dd
call 0x311286e0
push eax
mov eax,[ebp+0xf4]
pop edx
mov ecx,eax
➤call KeyedLoadIC_Initialize
pop edx
call BinaryOpIC_Initialize
...
```

LOAD IC

```
;; Code that knows how to
;; get fetch primes from a Prime object
...
ret
```

Backpatching Inline Caches at Runtime

IA32 ASSEMBLY

```
...
push [ebp+0x8]
mov eax,[ebp+0xc]
mov edx,eax
mov ecx,0x50b155dd
call 0x311286e0
push eax
mov eax,[ebp+0xf4]
pop edx
mov ecx,eax
call 0x31129ae0
pop edx
call 0x3112ade0
...
```

LOAD IC

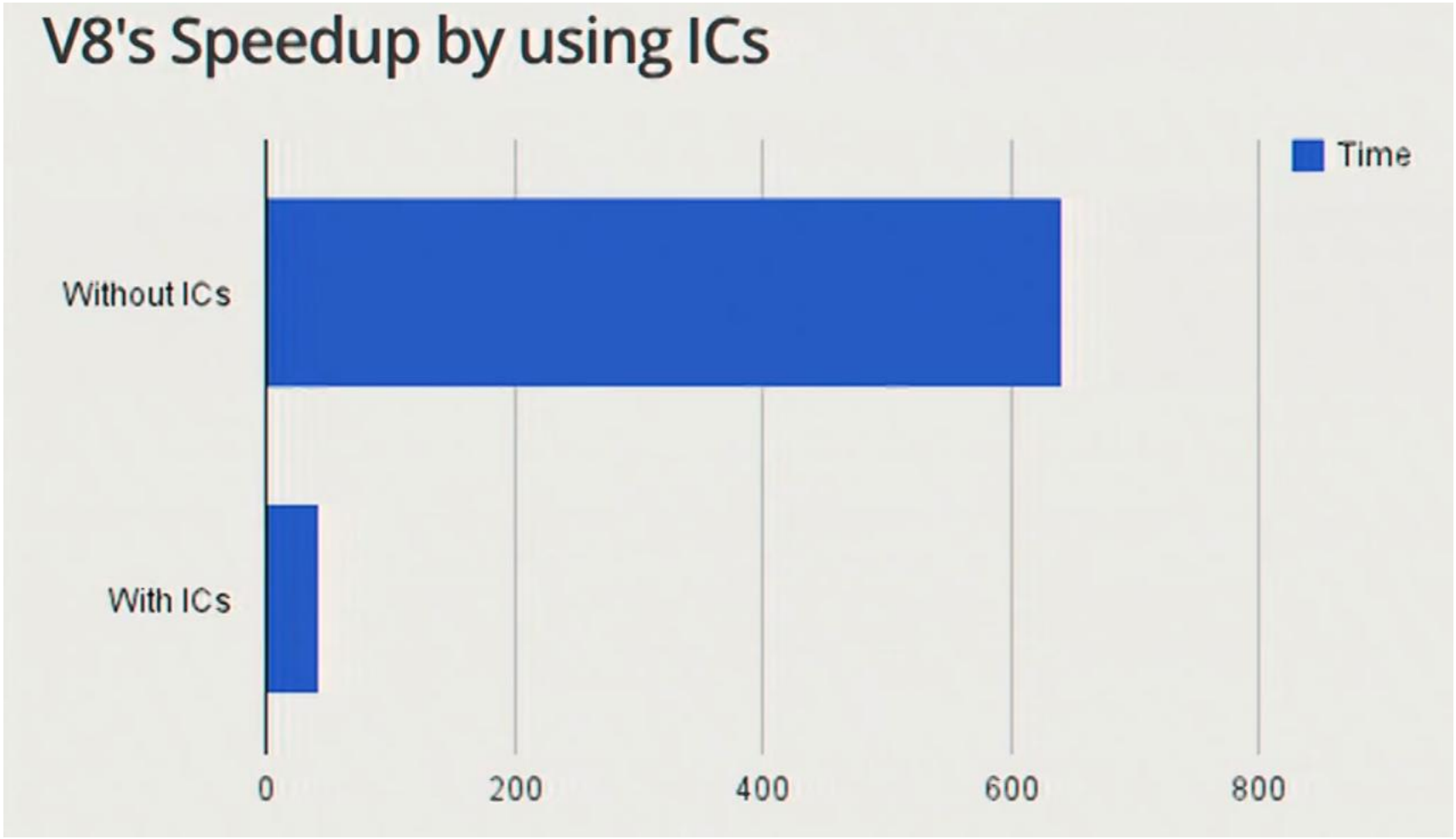
```
;; Code that knows how to
;; get fetch primes from a Prime object
...
ret
```

KEYED LOAD IC

```
;; Code that knows
;; how to get an element from SMI Array
...
ret
```

BINARY OP IC

```
;; Code that knows
;; how to calculate SMI % SMI
...
ret
```


V8's Speedup by using ICs
Time
Without ICs
With ICs
0
200
400
600
800

Monomorphic Better Than Polymorphic

- Operations are monomorphic if the hidden class is always the same

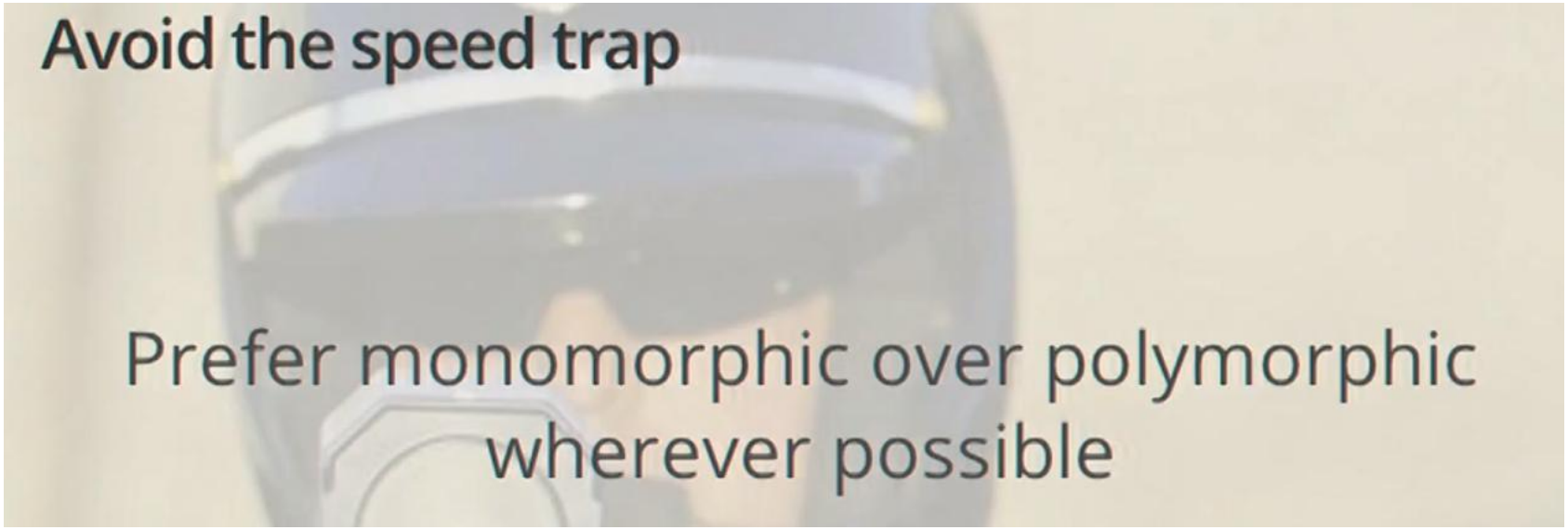
Monomorphic Better Than Polymorphic

- Operations are monomorphic if the hidden class is always the same
- Otherwise they are polymorphic

JAVASCRIPT

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

add(1, 2);      // + in add is monomorphic
add("a", "b");  // + in add becomes polymorphic
```


Avoid the speed trap
Prefer monomorphic over polymorphic wherever possible

The Optimizing Compiler

V8 re-compiles hot functions with an optimizing compiler

Type Feedback Makes Code Faster

- Types taken from ICs
- Operations speculatively get inlined
- Monomorphic functions and constructors can be inlined entirely
- Inlining enables other optimizations

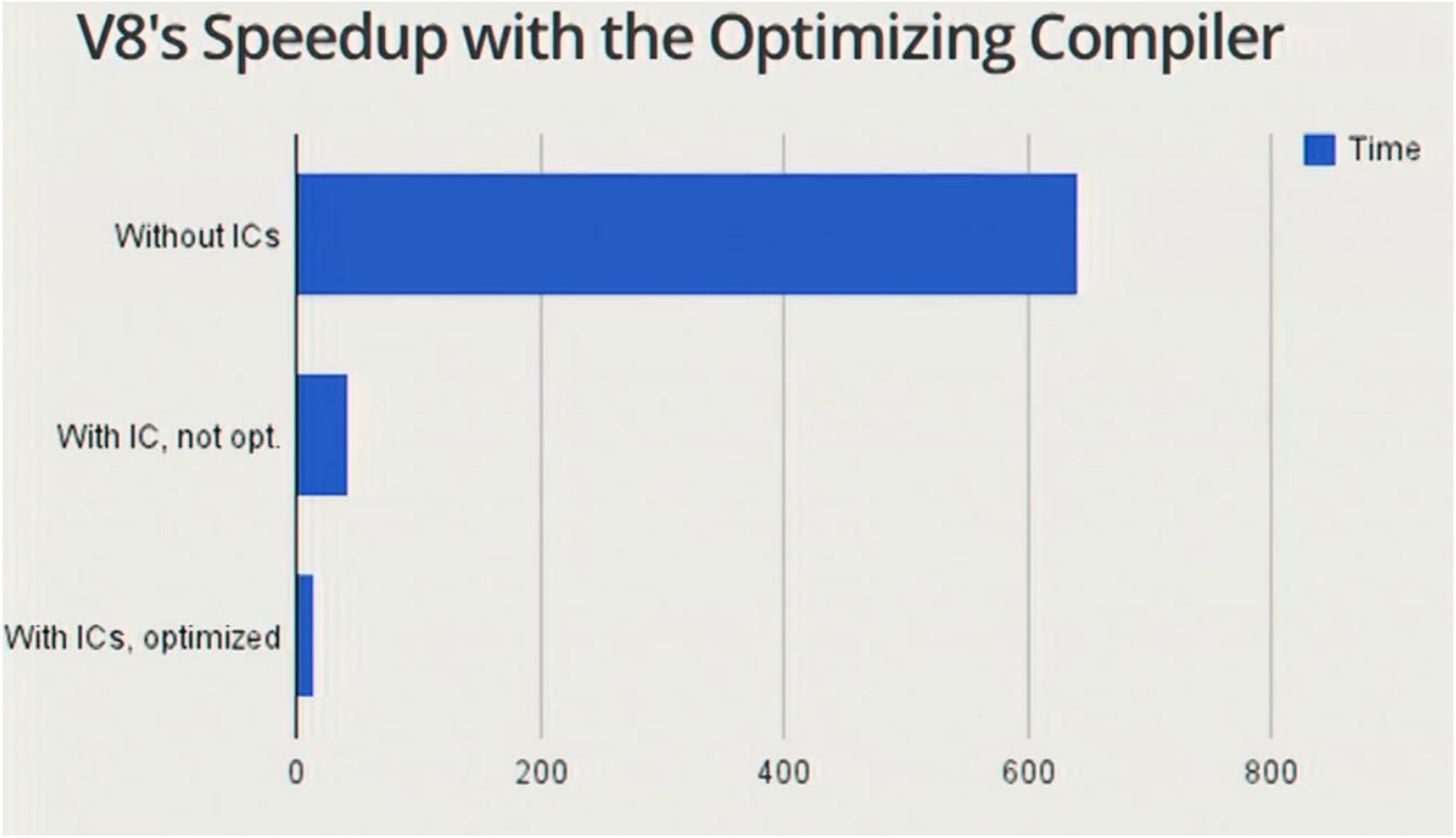
Generating Optimized Code

```
candidate % this.primes[i]
```

IA32 ASSEMBLY

```
cmp [edi+0xff],0x4920d181   ;; Is this a Primes object?
jnz 0x2a90a03c
mov eax,[edi+0xf]           ;; Fetch this.primes
test eax,0x1                ;; Is primes a SMI ?
jz 0x2a90a050
cmp [eax+0xff],0x4920b001   ;; Is primes hidden class a packed SMI array?
mov ebx,[eax+0x7]
mov esi,[eax+0xb]           ;; Load array length
sar esi,1                   ;; Convert SMI length to int32
cmp ecx,esi                 ;; Check array bounds
jnc 0x2a90a06e
mov esi,[ebx+ecx*4+0x7]     ;; Load element
sar esi,1                   ;; Convert SMI element to int32
test esi,esi                ;; mod (int32)
jz 0x2a90a078
...
cdq
idiv esi
```

Notice: No calls in this code!


V8's Speedup with the Optimizing Compiler
Time
Without ICs
With IC, not opt.
With ICs, optimized
0
200
400
600
800

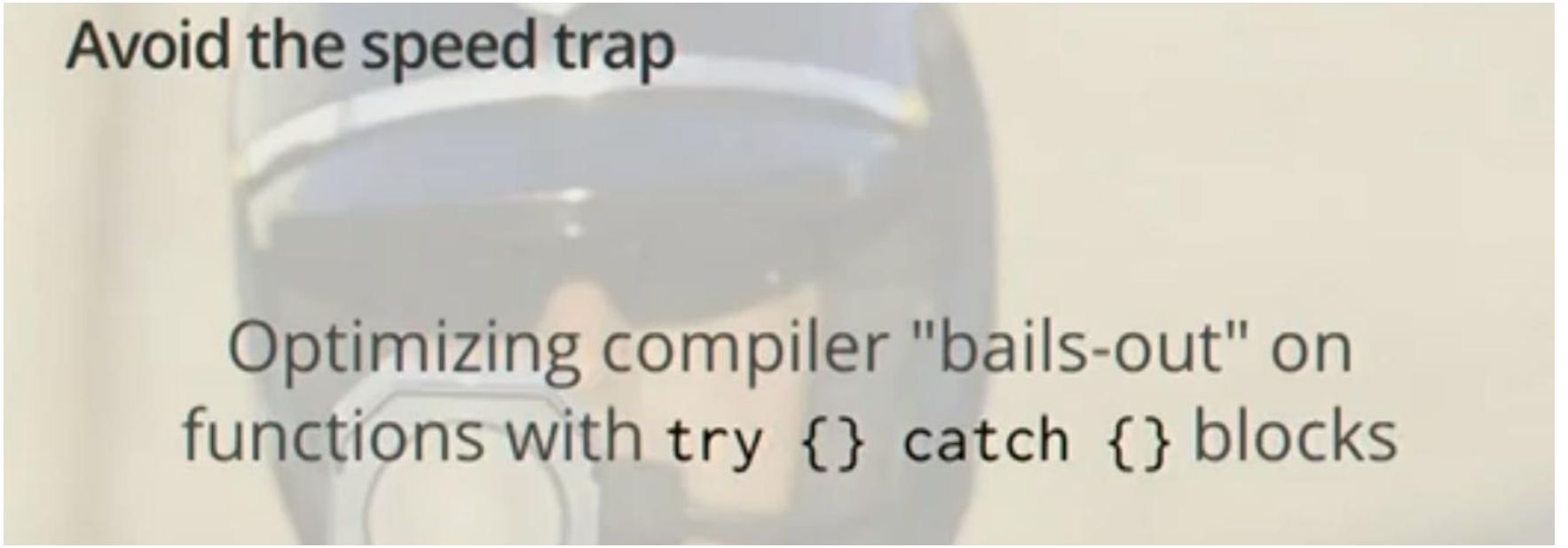
Logging What Gets Optimized

```
d8 --trace-opt primes.js
```

log names of optimized functions to stdout

Not Everything Can Be Optimized

Some features prevent the optimizing compiler from running (a **"bail-out"**)


Avoid the speed trap
Optimizing compiler "bails-out" on
functions with try {} catch {} blocks

Maximizing Performance With Exceptions

JAVASCRIPT

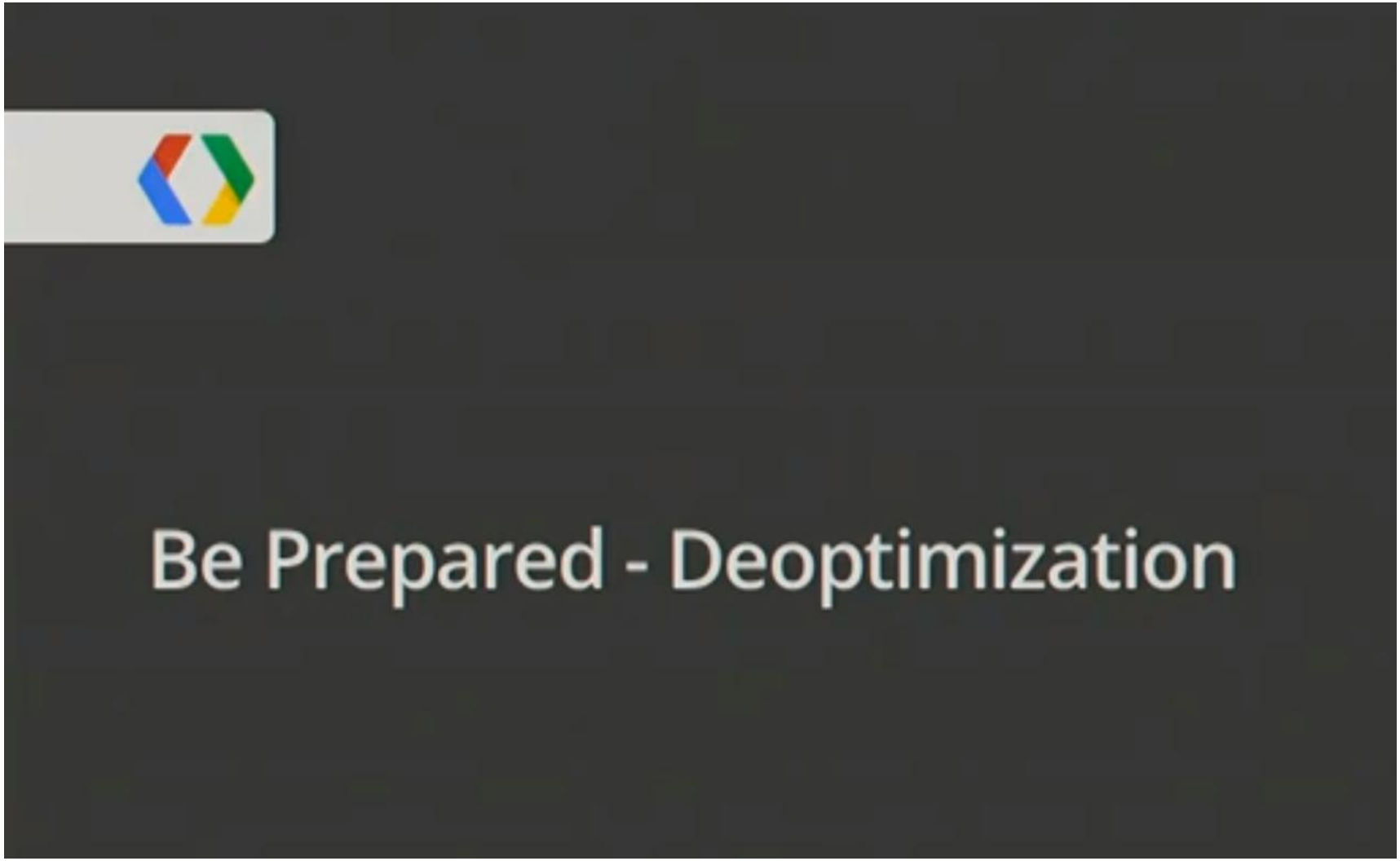
```
function perf_sensitive() {
  // Do performance-sensitive work here
}

try {
  perf_sensitive()
} catch (e) {
  // Handle exceptions here
}
```

How to Find Bailouts

```
d8 --trace-bailout primes.js
```

logs optimizing compiler bailouts


Be Prepared - Deoptimization

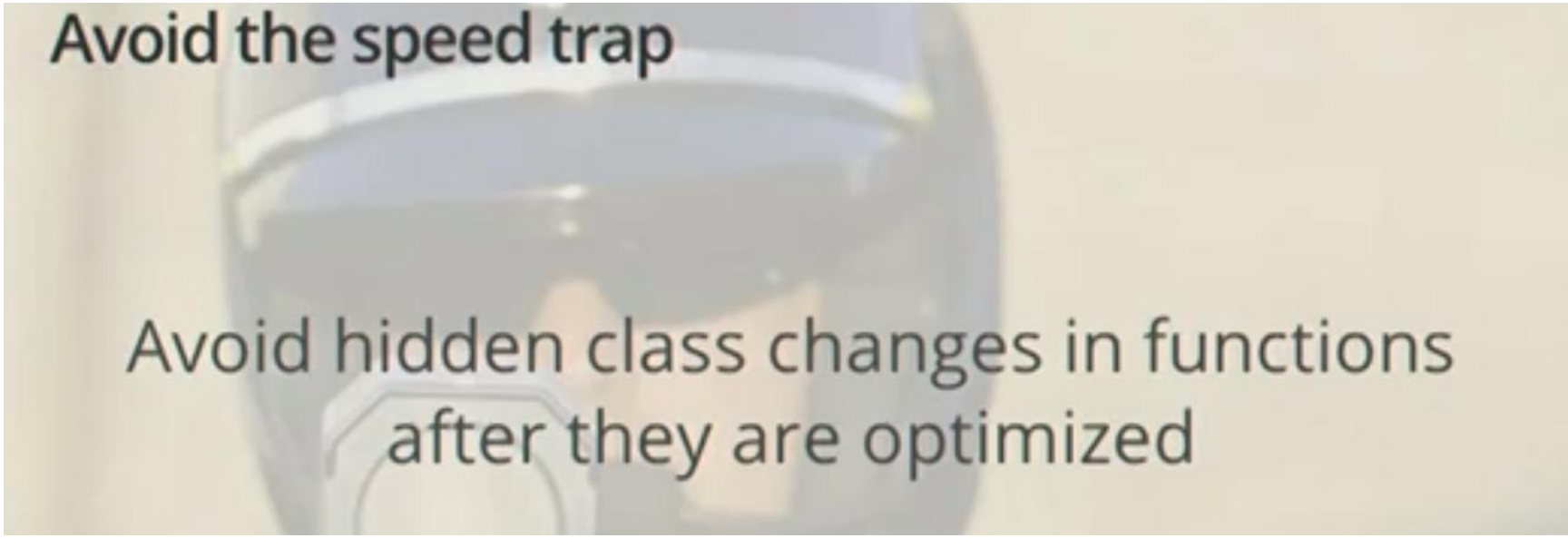
Optimizations are speculative

Usually, they pay off

Invalid assumptions lead to **deoptimization**

Deoptimization...

- ...throws away optimized code
- ...resumes execution at the right place in "full" compiler code
- Reoptimization might be triggered again later, but for the short term, execution slows down


Avoid the speed trap
Avoid hidden class changes in functions
after they are optimized

Finding Deoptimizations in Your Code

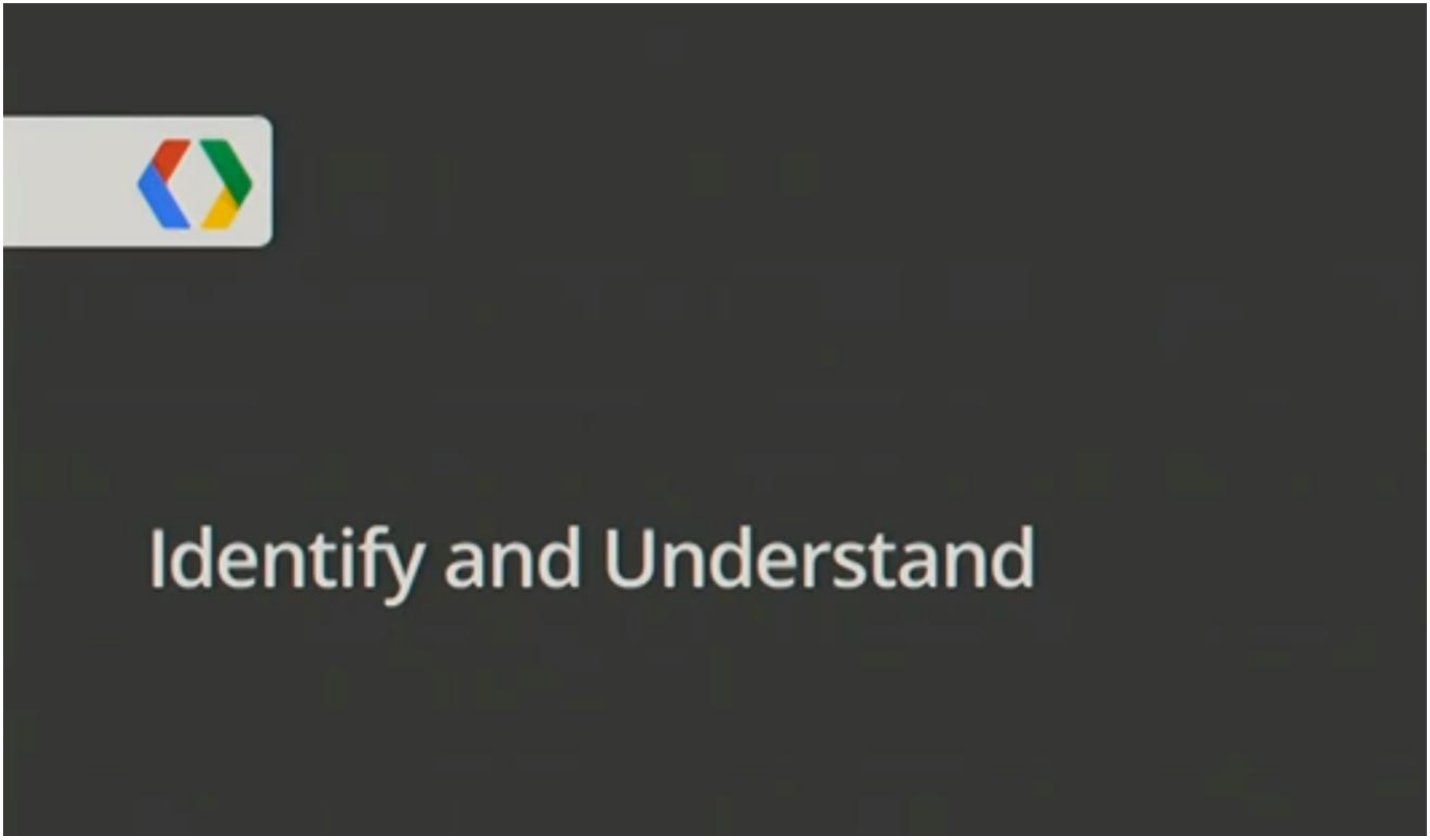
```
d8 --trace-deopt primes.js
```

log functions that v8 had to deoptimize

Passing V8 Options to Chrome

```
"/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" \

    --js-flags="--trace-opt --trace-deopt --trace-bailout"
```


Identify and Understand

"Identify and Understand" for V8

- Ensure problem is JavaScript
- Reduce to pure JavaScript (no DOM!)
- Collect metrics
- Locate bottleneck(s)

Prime Generator — Profile It

```
% out/ia32.release/d8 primes.js --prof
287107
```
SHELL

using the built-in sampling profiler:

- Takes sample every millisecond
- Writes v8.log

What to Expect from the Primes Code

JAVASCRIPT

```
function Primes() {
  ...
  this.addPrime = function(i) {
    this.primes[this.prime_count++] = i;
  }

  this.isPrimeDivisible = function(candidate) {
    for (var i = 1; i <= this.prime_count; ++i) {
      if ((candidate % this.primes[i]) == 0) {
        return true;
      }
    }
    return false;
  }
};
```

What to Expect from the Primes Code

JAVASCRIPT

```
function main() {
  p = new Primes();
  var c = 1;
  while (p.getPrimeCount() < 25000) {
    if (!p.isPrimeDivisible(c)) {
      p.addPrime(c);
    }
    c++;
  }
  print(p.getPrime(p.getPrimeCount()-1));
}
```

Prediction: Most Time Spent in `main`

- All properties and functions monomorphic
- All numeric operations are SMIs
- All functions can be inlined
- No deoptimizations or bailouts

Profile Results

SHELL

```
% out/ia32.release/d8 primes.js --prof
287107
```

SHELL

```
% tools/mac-tick-processor
 [JavaScript]:
   ticks  total  nonlib   name
   3958   25.1%   25.1%  LazyCompile: *main
    666    4.2%    4.2%  Stub: BinaryOpStub_MOD_Alloc_Oddball
     16    0.1%    0.1%  LazyCompile: *Primes.isPrimeDivisible

 [C++]:
   ticks  total  nonlib   name
   4917   31.1%   31.2%  v8::internal::modulo
   3467   22.0%   22.0%  v8::internal::Heap::NumberFromDouble
   1695   10.7%   10.7%  v8::internal::Runtime_NumberMod
```

Profile Results

SHELL

```
% out/ia32.release/d8 primes.js --prof
287107
```

SHELL

```
% tools/mac-tick-processor
 [JavaScript]:
   ticks  total  nonlib   name
   3958   25.1%   25.1%  LazyCompile: *main
    666    4.2%    4.2%  Stub: BinaryOpStub_MOD_Alloc_Oddball
     16    0.1%    0.1%  LazyCompile: *Primes.isPrimeDivisible

 [C++]:
   ticks  total  nonlib   name
   4917   31.1%   31.2%  v8::internal::modulo
   3467   22.0%   22.0%  v8::internal::Heap::NumberFromDouble
   1695   10.7%   10.7%  v8::internal::Runtime_NumberMod
```

Can you spot the bug?

JAVASCRIPT

```
this.isPrimeDivisible = function(candidate) {
  for (var i = 1; i <= this.prime_count; ++i) {
    if (candidate % this.primes[i] == 0) return true;
  }
  return false;
}
```

(Hint: `primes` is an array of length `prime_count`)

Profile Again with Out-of-Bounds Fix

SHELL

```
% out/ia32.release/d8 primes-2.js --prof
287107
```

SHELL

```
% tools/mac-tick-processor
 [JavaScript]:
   ticks  total  nonlib   name
   1789   99.2%   99.2%  LazyCompile: *main code/primes-2.js
      5    0.3%    0.3%  LazyCompile: *Primes.isPrimeDivisible
      1    0.1%    0.1%  LazyCompile: ~main code/primes-2.js

 [C++]:
   ticks  total  nonlib   name
      1    0.1%    0.1%  v8::internal::Label::pos() const
```

JavaScript is 60% faster than C++!

C++

```
% g++ primes.cc -o primes
% time ./primes
287107

real    0m2.955s
user    0m2.952s
sys     0m0.001s
```

SHELL

JavaScript

```
% time d8 primes-2.js
287107

real    0m1.829s
user    0m1.827s
sys     0m0.010s
```

SHELL

JavaScript is 17% slower than optimized C++

C++

```
% g++ primes.cc -o primes -O3
% time ./primes
287107

real    0m1.564s
user    0m1.560s
sys     0m0.002s
```

SHELL

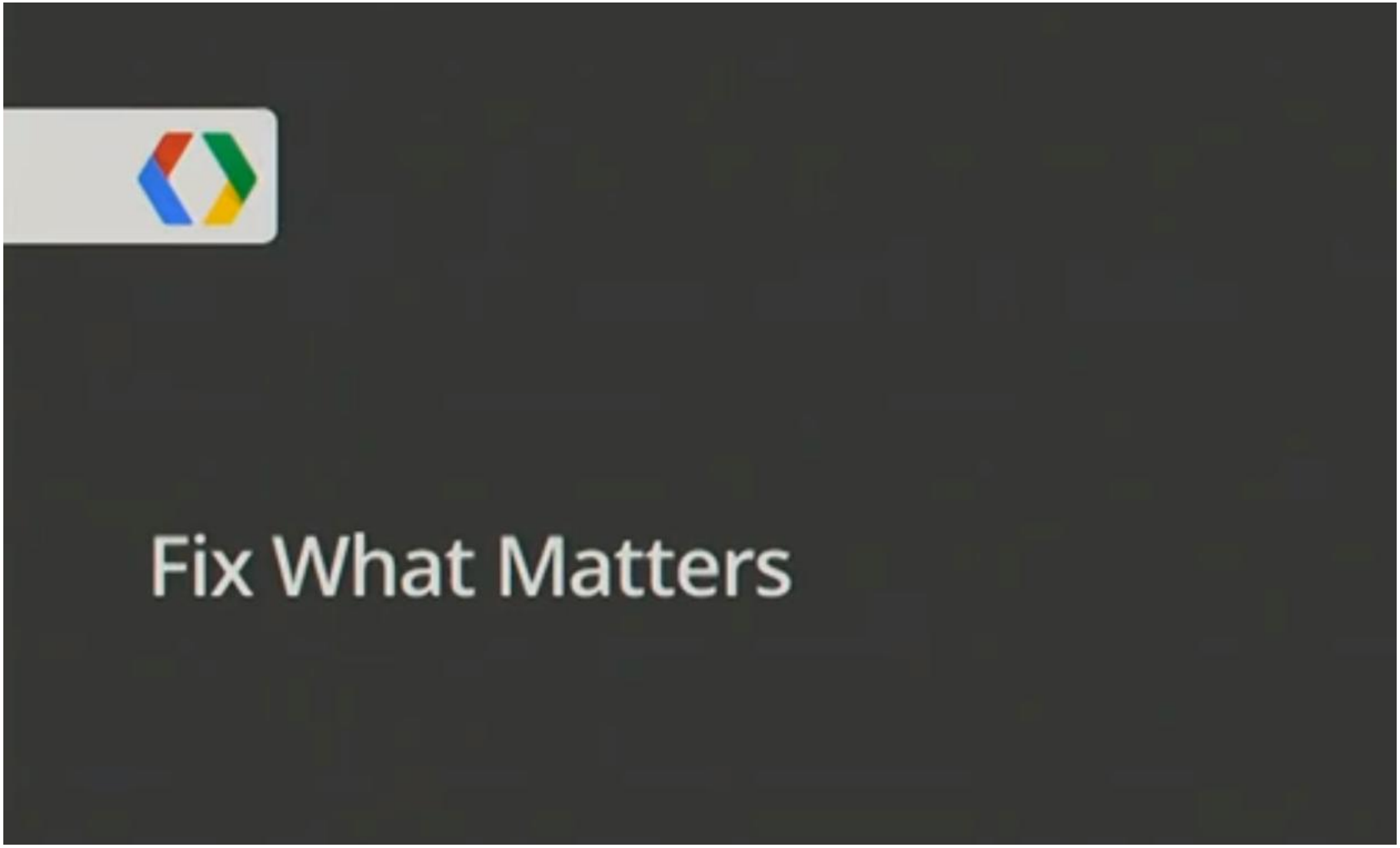
JavaScript

```
% time d8 primes-2.js
287107

real    0m1.829s
user    0m1.827s
sys     0m0.010s
```

SHELL

You must take all this with a grain of salt


Fix What Matters

Optimize Your Algorithm

JAVASCRIPT

```
this.isPrimeDivisible = function(candidate) {
  for (var i = 1; i < this.prime_count; ++i) {
    if (candidate % this.primes[i] == 0) return true;
  }
  return false;
}
```

Optimize Your Algorithm

JAVASCRIPT

```
this.isPrimeDivisible = function(candidate) {
  for (var i = 1; i < this.prime_count; ++i) {
    var current_prime = this.primes[i];
    if (current_prime * current_prime > candidate) {
      return false;
    }
    if ((candidate % current_prime) == 0) return true;
  }
  return false;
}
```

Final Results

SHELL

```
% time d8 primes-3.js
287107

real    0m0.044s
user    0m0.038s
sys     0m0.004s
```

vs.

SHELL

```
% time d8 primes.js
287107

real    0m15.584s
user    0m15.612s
sys     0m0.073s
```

Final Results

```
% time d8 primes-3.js
287107

real    0m0.044s
user    0m0.038s
sys     0m0.004s
```

That's more than a 350x speed-up!

```
% time d8 primes.js
287107

real    0m15.584s
user    0m15.612s
sys     0m0.073s
```

Keep Your Eyes on the Road

- Be prepared
- Identify and Understand the Crux
- Fix what matters

<Thank You!>

email danno@chromium.org
g+ plus.ly/danno
www goo.gl/U2vuT