Demystifying v8 and JavaScript Performance

twitter @thlorenz | github @thlorenz | irc thlorenz

Data Types

- ECMA defines data types
- v8 maps them to optimized data types

Primitive Types

- Boolean
- Number
- String
- Null
- Undefined

Reference Types

- Object
- Array
- Typed Array

Number

- ECMA double-precision 64-bit binary format IEEE 754 value
- v8 32-bit numbers to represent all values

- ECMAScript standard:
- number between -(2^53 -1) and 2^53 -1
- no specific type for integers
- can represent floating-point numbers
- three symbolic values: +Infinity, -Infinity, and NaN

Tagging

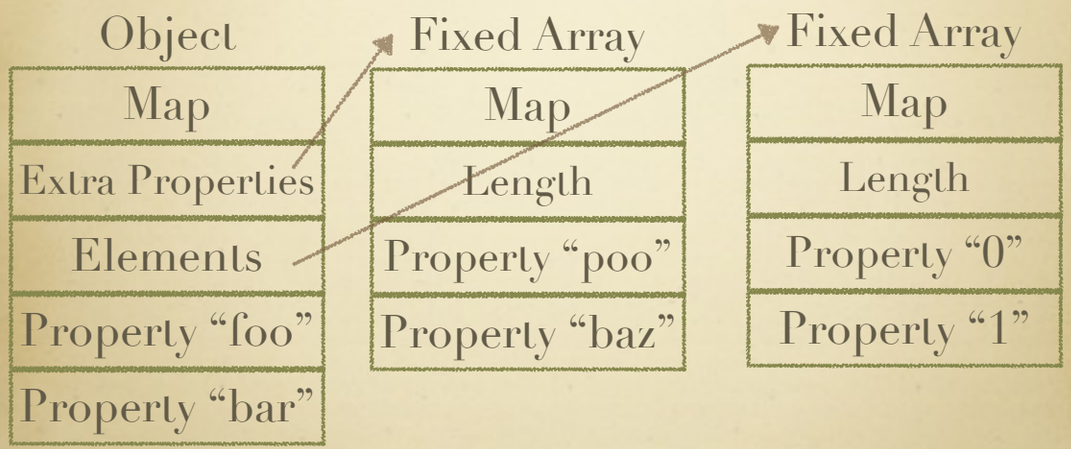
32 bit signed integer (SMI)

object pointer

Tagging

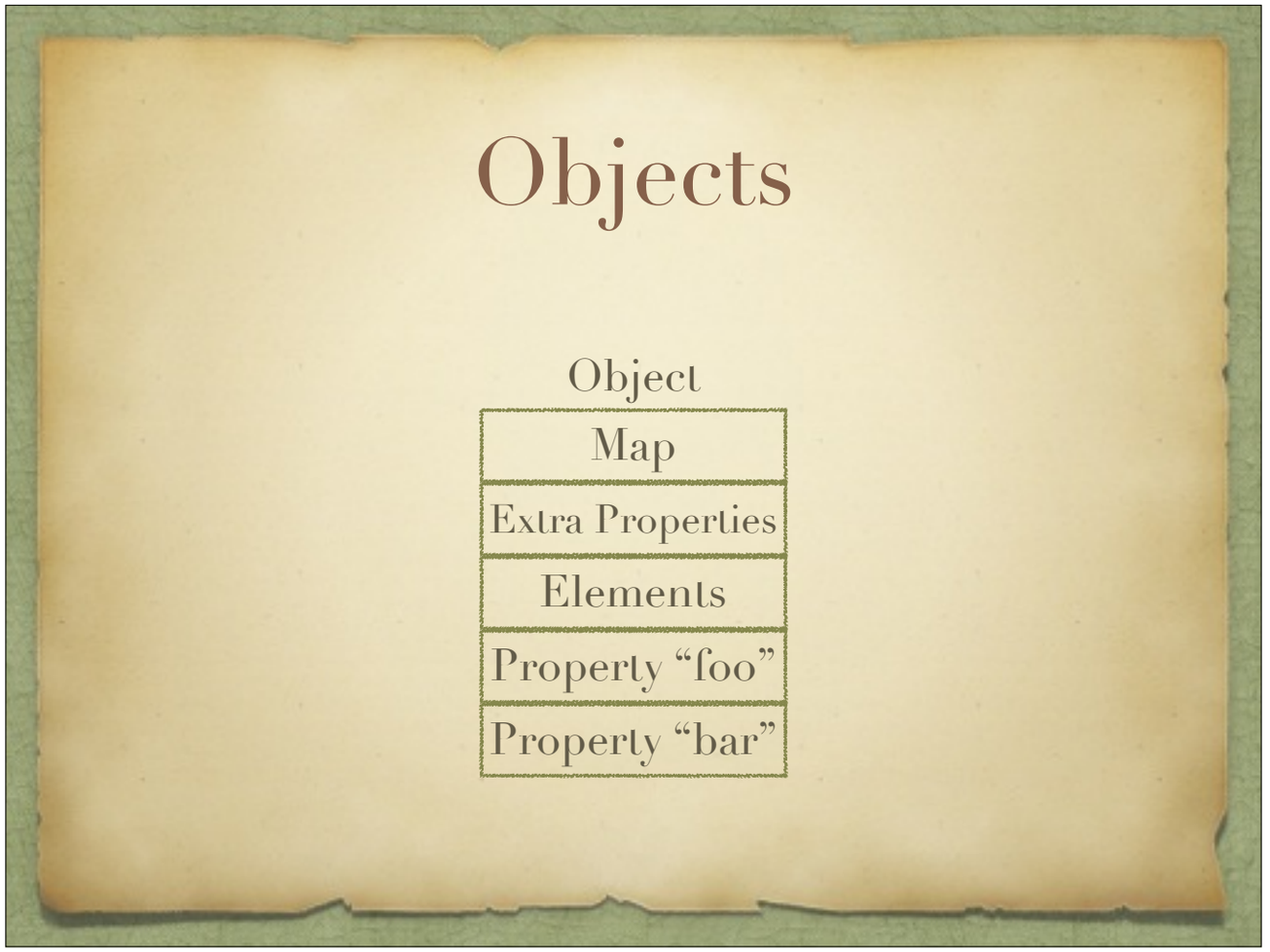
- numbers bigger than 31 bits are boxed
- stored inside an object referenced via a pointer
- adds extra overhead (at a minimum an extra lookup)
- prefer SMIs for numeric values whenever possible
-

Objects



- above shows most common optimized representation
- all blocks have a Map property describing their structure
- most objects contain all their properties in single block of memory "foo", “bar"
- object is a collection of properties aka key-value pairs
- named properties that don't fit are stored in overflow array "poo", "baz"
- numbered properties are stored in a separate contiguous array "1", “2"

Objects



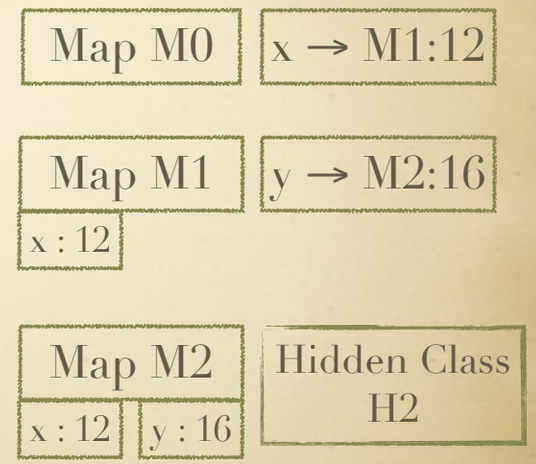
- property names are always strings
- any name used as property name that is not a string is stringified via .toString(), even numbers, so 1 becomes "1"
- Arrays in JavaScript are just objects with magic length property

Objects

- v8 describes the structure of objects using maps that are used to create hidden classes and match data types
  - resembles a table of descriptors with one entry for each property
  - map contains info about size of the object
  - map contains info about pointers to constructors and prototypes
  - objects with same structure share same map
- objects created by the same constructor and have the same set of properties assigned in the same order
  - have regular logical structure and therefore regular structure in memory
  - share same map
- adding new property is handled via transition descriptor
  - use existing map
  - transition descriptor points at other map

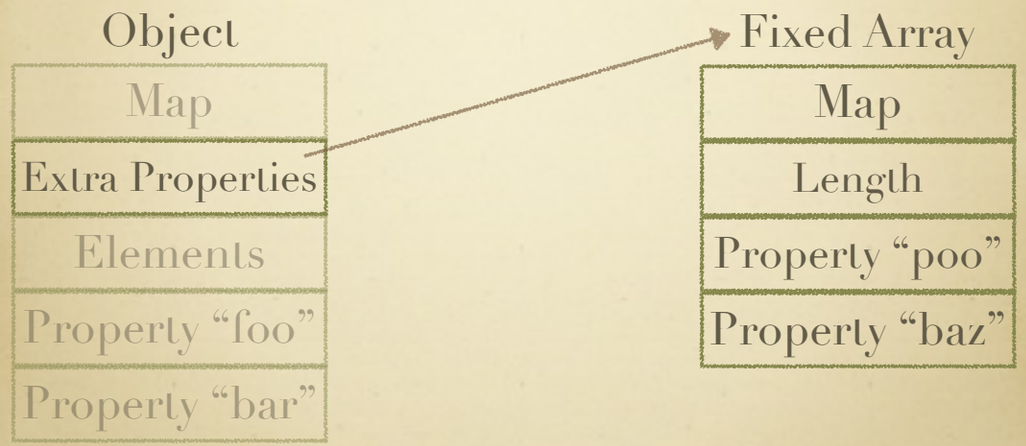
Objects

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



- Point starts out without any fields with M0
- this.x =x -> map pointer set to M1 and value x is stored at offset 12 and "x" Transition descriptor added to M0
- this.y =y -> map pointer set to M2 and value y is stored at offset 16 and "y" Transition descriptor added to M1

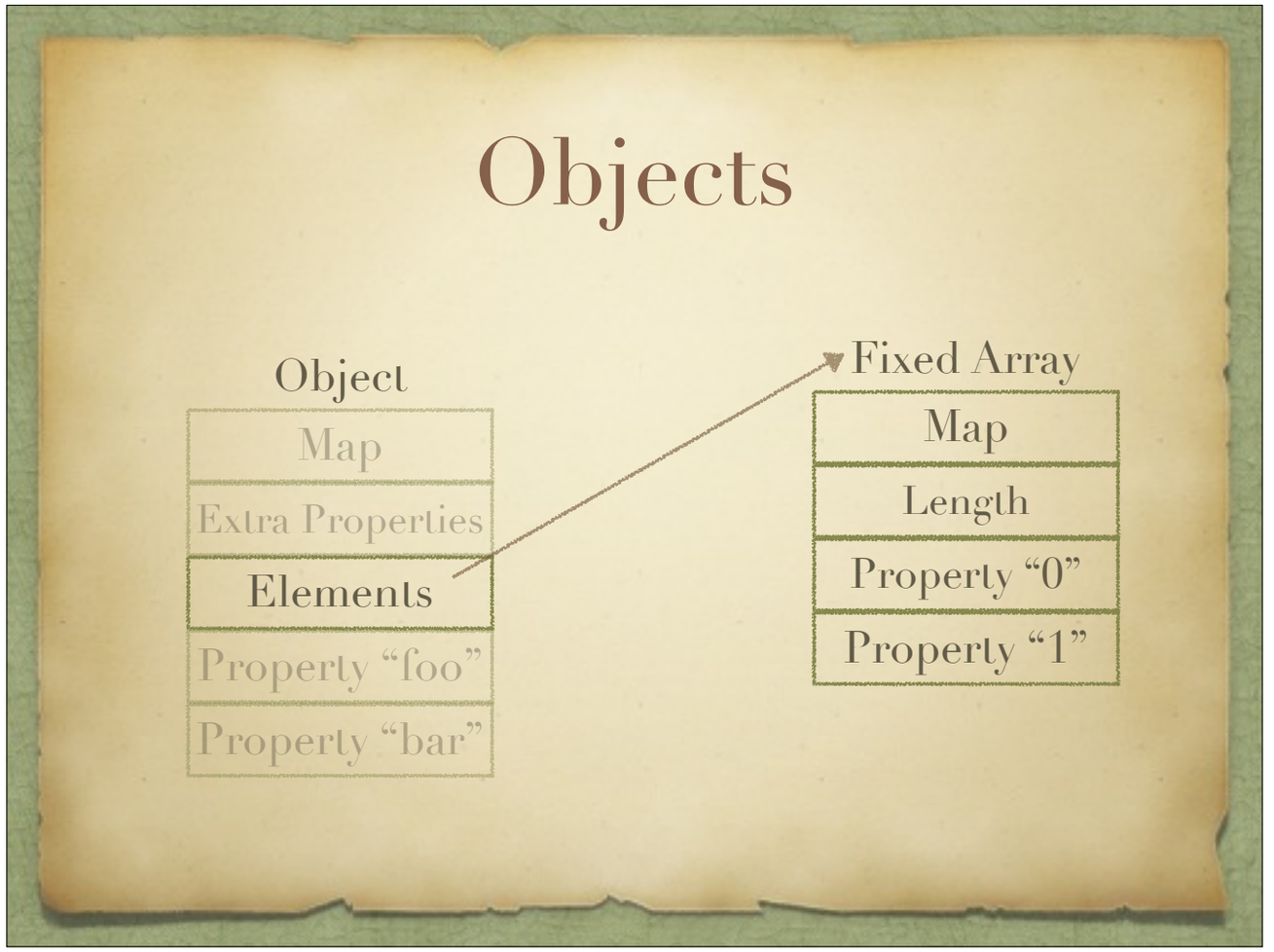
Objects



In-object Slack Tracking

- objects allocated by a constructor are given enough memory for 32 fast properties to be stored (foo, bar)
- after certain number of objects (8) were allocated from same constructor
  - v8 traverses transition tree from initial map to determine size of largest of these initial objects
  - new objects of same type are allocated with exact amount of memory to store max number of properties
  - initial objects are resized (down)
- if more properties are added to object afterwards they are strored in extra properties

Objects



- numbered properties are treated and ordered differently than others since any object can behave like an array
- v8 stores elements separate from named properties in an elements kind field
- most elements are fast elements which are stored in a contiguous array
- maps don't need transitions to maps that are identical except for element kinds

Hash Tables

- v8 tries to create object maps for whatever you are doing, but if amount of maps would get ridiculous it just gives up and drops object into dictionary mode
- hash table used for difficult objects
- aka objects in dictionary mode
- accessing hash table property is much slower than accessing a field at a known offset
- if non-symbol string is used to access a property it is uniquified first
- v8 hash tables are large arrays containing keys and values

Typed Arrays

- v8 uses unboxed backing stores
- Float64 gets 64-bit allocated for each element

Double Array Unboxing

- Array's hidden class tracks element types
- if all doubles, array is unboxed aka upgraded to fast doubles
    - wrapped objects layed out in linear buffer of doubles
    - each element slot is 64-bit to hold a double
    - SMIs that are currently in Array are converted to doubles
    - very efficient access
    - storing requires no allocation as is the case for boxed doubles
    - causes hidden class change
    - requires expensive copy-and-convert operation
- careless array manipulation may cause overhead due to boxing/unboxing

Arrays

- fast elements
- dictionary elements

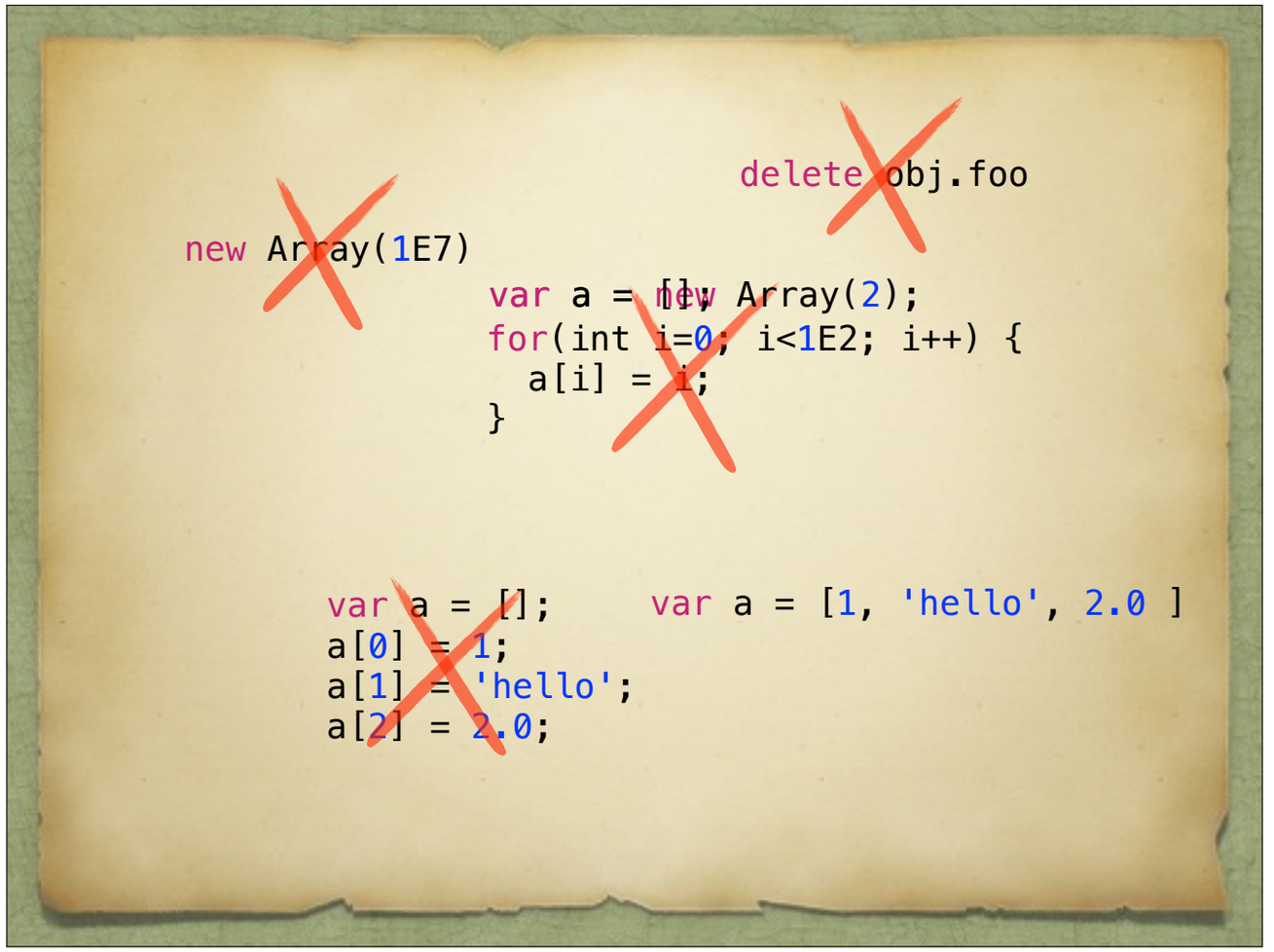
Fast Elements

- compact keysets
- linear storage buffer

- fast elements kinds in order of increasing generality:
    - fast SMIs (small integers)
    - fast doubles (Doubles stored in unboxed representation)
    - fast values (strings or other objects)
- allows access elements via offset

Fast Elements
Requirements

- contiguous (non-sparse)
- 0 based
- allocated with <100K elements



- don't pre-allocate large arrays (>=100K elements), instead grow as needed, to avoid them being considered sparse
- do pre-allocate small arrays to correct size to avoid allocations due to resizing
- don't delete elements
- use literal initializer for Arrays with mixed values
- use typed arrays whenever possible
- copying an array, you should avoid copying from the back (higher indices to lower indices) because this will almost certainly trigger dictionary mode

Compilers

- full compiler

- generates code for any JavaScript
- all code starts unoptimized
- initial (quick) JIT
- is not great and knows (almost) nothing about types
- needed to start executing code ASAP
- uses Inline Caches (ICs) to refine knowledge about types at runtime

Compilers

- full compiler
- optimizing compiler

- recompiles and optimizes hot code identified by the runtime profiler
- optimization decisions are based on type information collected while running the code produced by the full compiler

Optimization

- if function executes a lot it becomes hot
- hot function is re-compiled with optimizing compiler

- optimistically
- lots of assumptions made from the calls made to that function so far

- type information takend from ICs

- operations get inlined speculatively using historic information
- monomorphic functions/constructors can be inlined entirely
- inlining allows even further optimizations

Inline Caches

- gather knowledge about types while program runs
- type dependent code for operations is given specific hidden classes as inputs

1. validate type assumptions (are hidden classes as expected)
2. do work
- Inline Caches alone without optimizing compiler step make huge performance difference (20x speedup)

- change at runtime via backpatching as more types are discovered to generate new ICs watch

Deoptimization

- optimizations are speculative
- assumptions are made and maybe violated

- if assumption is violated
  - function deoptimized
  - execution resumes in full compiler code
  - in short term execution slows down
  - normal to occur
  - more info about about function collected
  - better optimization attempted
  - if assumptions are violated again, deoptimized again and start over

Deoptimization

- too many deoptimizations cause function to be sent to deoptimization hell
    - considered not optimizable and no optimization is ever attempted again (especially bad on server)
- certain constructs like try/catch (not on FireFox) are considered not optimizable and functions containing it go straight to deoptimization hell due to bailout watch

Deoptimization

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

var p = new Point(1, 2);
// => hidden Point class created

// ...

p.z = 3;
// => another hidden class (Point') created
```

- Point class created, code still deoptimized
- functions that have Point argument are optimized
- z property added which causes Point' class to be created
- functions that get passed Point' but were optimized for Point get deoptimized
- later functions get optimized again, this time supporting Point and Point' as argument

Considerations

- initialize all members in constructor function in the same order
- avoid polymorphic functions
- don't do work inside unoptimizable functions

Resources

https://thlorenz.github.io/v8-perf/

Thanks

Thorsten Lorenz
twitter @thlorenz | github @thlorenz | irc thlorenz