CHICAGO

INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2015

goto;
conference

Making This Rhinoceros Thunder

Attila Szegedi

@asz

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE®

What is Nashorn?

- Nashorn is an ECMAScript 5.1 runtime on top of JVM.

- Open source: all development happens in OpenJDK.

- Ships as standard part of Oracle's Java SE starting with version 8.

- Accessible through standard javax.script.* API, or directly through jdk.nashorn.api.scripting package.

- Command line: $JAVA_HOME/bin/jjs

- Has no interpreter currently; compiles to Java bytecode on-the-fly.

ORACLE®

Why Nashorn?

- Full ECMAScript 5.1 compliance.

- Modern codebase.

- Security minded.

- Some internally interesting goals:

  - Proving ground for invokedynamic.

  - Laying groundwork for general dynamic languages platform support.

ORACLE®

Performance, performance, performance

- It's easy to write a slow language runtime.

- You can spend a lifetime writing optimizations in your runtime.

- Not all optimizations apply equally well.

- So I want to show you some good practices.

ORACLE®

Why Don't I Just Make Nashorn So This Talk Isn't Necessary?

ORACLE
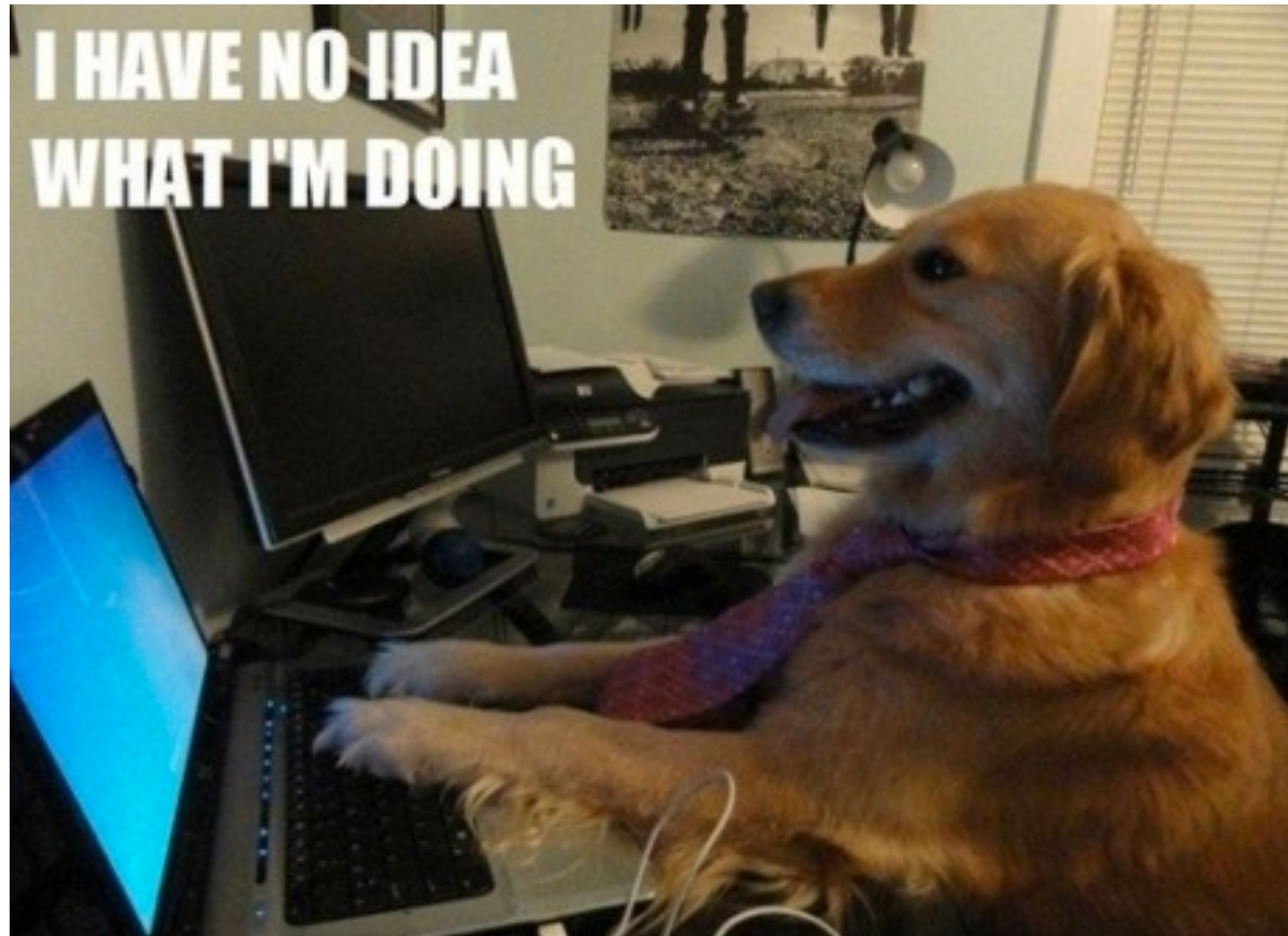
There Are Things Nashorn Doesn't Control

Why Does This Talk Exist?

- We made Nashorn internals pretty smart by now:
  - Parameter-type specialized compilation of functions.
  - Static type inference for local variables.
  - Optimistic typing with gradual deoptimizing recompilation where static typing can't take us.
- So what's left then?

ORACLE

Where Nashorn Can't Help You

- … so you'll need to help yourself when you:
  - integrate it into your Java-based system.
  - make things too hard for the runtime to reason about.

ORACLE®

Let's Look At Some Existing Features

- I'll quickly show you how Nashorn does:
  - parameter type specialized compilation,
  - local type inference, and
  - optimistic typing.

|

ORACLE®

Parameter type specialized compilation

- Here's code versions for square generated when invoked with int and double:

```
function square(x) {
    return x*x;
}
print(square(500));
print(square(500.1));
```

```
public static square(Object;I)I
    0       iload 1
    1       iload 1
    2       invokedynamic imul(II)I
    7       ireturn


public static square(Object;D)D
    0       dload 1
    1       dload 1
    2       dmul
    3       dreturn
```

ORACLE®

Static Type Inference

- Here's a little number cruncher from crypto.js Octane benchmark:

```
function am3(i,x,w,j,c,n) {
  var this_array = this.array;
  var w_array    = w.array;

  var xl = x&0x3fff, xh = x>>14;
  while(--n >= 0) {
    var l = this_array[i]&0x3fff;
    var h = this_array[i++]>>14;
    var m = xh*l+h*xl;
    l = xl*l+((m&0x3fff)<<14)+w_array[j]+c;
    c = (l>>28)+(m>>14)+xh*h;
    w_array[j++] = l&0xfffffff;
  }
  return c;
}
```

ORACLE®

Static Type Inference

- Here's Nashorn's inferred types:

```
function 〚D〛am3(〚D〛i,〚O〛x,〚O〛w,〚D〛j,〚I〛c,〚I〛n) {
   var 〚O〛this_array = this.array;
   var 〚O〛w_array    = w.array;

   var 〚I〛xl = x&0x3fff, 〚I〛xh = x>>14;
   while(--〚D〛n >= 0) {
      var 〚I〛l = this_array[i]&0x3fff;
      var 〚I〛h = this_array[i++]>>14;
      var 〚D〛m = xh*l+h*xl;
      〚O〛l = xl*l+((m&0x3fff)<<14)+w_array[j]+c;
      〚D〛c = (l>>28)+(m>>14)+xh*h;
      w_array[j++] = l&0xfffffff;
   }
   return c;
}
```

- Trouble spots:

  - "n" becomes double
  - "l" becomes object
  - "c" becomes double

ORACLE®

Optimistic Typing

- Here's Nashorn's inferred types:

```
function 〚I〛am3(〚I〛i,〚I〛x,〚O〛w,〚I〛j,〚I〛c,〚I〛n) {
  var 〚O〛this_array = this.array;
  var 〚O〛w_array    = w.array;

  var 〚I〛xl = x&0x3fff, 〚I〛xh = x>>14;
  while(--〚I〛n >= 0) {
    var 〚I〛l = this_array[i]&0x3fff;
    var 〚I〛h = this_array[i++]>>14;
    var 〚I〛m = xh*l+h*xl;
    〚I〛l = xl*l+((m&0x3fff)<<14)+w_array[j]+c;
    〚I〛c = (l>>28)+(m>>14)+xh*h;
    w_array[j++] = l&0xfffffff;
  }
  return c;
}
```

- Trouble spots are gone!
- Optimistically assume that:
  - decrement won't overflow
  - adds and multiplies won't overflow
  - w_array has int elements

ORACLE®

Synergy!

- These features working together ensure that the generated code evolves to the tightest version that can handle the data.

```
function twice(f, x) {
    return f(f(x));
}


function inc(x) {
    return x++;
}

print(twice(inc, 5));
print(twice(inc, 5.1));
```

- We'll end up with `int twice(f, int)` and `int inc(int)`.

- Then `double inc(double)`, and gradually `int twice(f, double)` will morph into a `double twice(f, double)`.

ORACLE

Synergy!

```
print(⟦I⟧twice(inc, 5.1));

function ⟦I⟧twice(f, ⟦D⟧x) {
    return ⟦I⟧f(⟦I⟧f(⟦D⟧x));
}


function ⟦D⟧inc(⟦D⟧x) {
    return ⟦D⟧x++;
}
```

Synergy!

```
print(⟦I⟧twice(inc, 5.1));

function ⟦I⟧twice(f, ⟦D⟧x) {
    return ⟦I⟧f(⟦D⟧f(⟦D⟧x));
}

function ⟦D⟧inc(⟦D⟧x) {
    return ⟦D⟧x++;
}
```

ORACLE

Synergy!

```
print(⟦D⟧twice(inc, 5.1));

function ⟦D⟧twice(f, ⟦D⟧x) {
    return ⟦D⟧f(⟦D⟧f(⟦D⟧x));
}

function ⟦D⟧inc(⟦D⟧x) {
    return ⟦D⟧x++;
}
```

ORACLE®

Synergy!

```
print(⟦D⟧twice(inc, 5.1));              print(⟦I⟧twice(inc, 5));

function ⟦D⟧twice(f, ⟦D⟧x) {            function ⟦I⟧twice(f, ⟦I⟧x) {
    return ⟦D⟧f(⟦D⟧f(⟦D⟧x));                return ⟦I⟧f(⟦I⟧f(⟦I⟧x));
}                                       }


function ⟦D⟧inc(⟦D⟧x) {                 function ⟦I⟧inc(⟦I⟧x) {
    return ⟦D⟧x++;                          return ⟦I⟧x++;
}                                       }
```

- The int versions of functions continue to exist, and will be used when invoked with int parameters.

ORACLE®

Okay, But How Do We Deoptimize Running Code?

- To deoptimize running code, we must be able to:
  - recompile it on the fly, and
  - replace running code on top of the stack.
- We achieve this with a pure bytecode solution (runs on any JVM) that
  - throws an exception where type assumptions are too narrow,
  - links call site in caller with exception handler that derails into compiler,
  - recompiles a new version of the code with wider type,
  - compiles a separate one-shot continuation version of the code too,
  - jumps into the continuation variant to resume execution.

ORACLE®
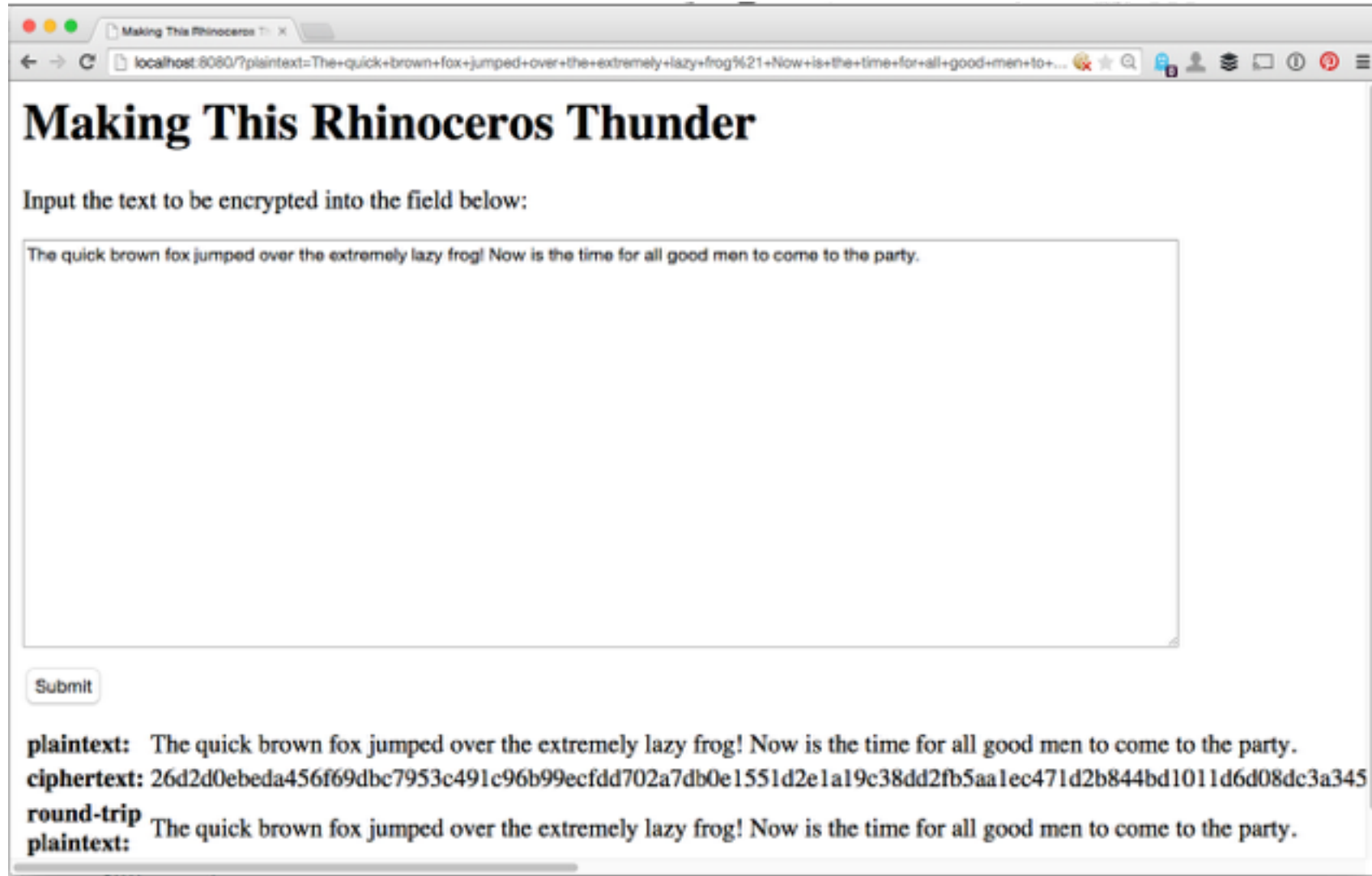
Let's Write a Small Web Application

- We'll use the Servlet API.
- To minimize overhead, I'll use Jetty as an embedded server.

ORACLE

Let's Write a Small Web Application

- The webapp will run RSA encryption/decryption of plaintext.
- Crypto code taken from Google's Octane benchmark suite.
- The logic is entirely written in JavaScript.
- Reasonably complex computation, with no I/O.
  - JavaScript doesn't have its own I/O libraries, so you'd just use Java's I/O facilities
  - There's no JavaScript performance story in that.

ORACLE®

Let's Write a Small Web Application

Most Naive Approach

- Instantiate a new ScriptEngine, on each request.
- Evaluate the JavaScript code in it, on each request.

ORACLE®

Most Naive Approach

```java
protected EvaluationResult evaluate(final String plainText) throws ScriptException {
    final ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
    engine.eval(new URLReader(scriptURL));

    engine.put("plainText", plainText);
    final String cipherText = (String) engine.eval("encrypt(plainText)");
    engine.put("cipherText", cipherText);
    final String roundTripPlainText = (String) engine.eval("decrypt(cipherText)");
    return new EvaluationResult(cipherText, roundTripPlainText);
}
```

ORACLE®

How well does it perform?

- Not really well.
- Let's look at percentage of the requests served within certain time (ms):

50%	**2020**
66%	**2304**
75%	**2580**
80%	**2745**
90%	**2986**
95%	**3770**
98%	**6241**
99%	**7041**
100%	**7041**

ORACLE®

Where does the time go?

- Most time spent in one-time code setup (dynamic linking).

CPU samples	Thread CPU Time			

Snapshot

Hot Spots – Method	Self Time [%] ▼	Self Time		Se
org.eclipse.jetty.util.BlockingArrayQueue.**poll** ()		169,340 ms	(39.4%)	
jdk.internal.dynalink.ChainedCallSite.**relinkInternal** ()		68,866 ms	(16%)	
jdk.internal.dynalink.support.AbstractRelinkableCallSite.**initialize** ()		55,835 ms	(13%)	
jdk.nashorn.internal.runtime.Context$ContextCodeInstaller$1.**run** ()		10,876 ms	(2.5%)	
jdk.nashorn.internal.runtime.ScriptLoader.**installClass** ()		8,474 ms	(2%)	
jdk.internal.dynalink.DynamicLinker.**createRelinkAndInvokeMethod** ()		5,613 ms	(1.3%)	
jdk.internal.dynalink.ChainedCallSite.**makePruneAndInvokeMethod** ()		3,366 ms	(0.8%)	
jdk.nashorn.internal.runtime.JSType.**toInt32** ()		2,096 ms	(0.5%)	
jdk.internal.dynalink.support.CallSiteDescriptorFactory.**tokenizeOperators** ()		1,906 ms	(0.4%)	
jdk.internal.dynalink.support.CallSiteDescriptorFactory.**tokenizeName** ()		1,685 ms	(0.4%)	
jdk.nashorn.internal.runtime.ScriptLoader.**loadClass** ()		1,603 ms	(0.4%)	
jdk.nashorn.internal.lookup.MethodHandleFactory$StandardMethodHandleFunctionality.**findStatic** ()		1,495 ms	(0.3%)	
jdk.nashorn.internal.runtime.GlobalConstants$Access.**invalidate** ()		1,453 ms	(0.3%)	
jdk.nashorn.internal.runtime.ScriptObject.**get** ()		1,444 ms	(0.3%)	
jdk.internal.dynalink.linker.GuardedInvocation.**compose** ()		1,201 ms	(0.3%)	
jdk.nashorn.internal.lookup.MethodHandleFactory$StandardMethodHandleFunctionality.**insertArgumer**		813 ms	(0.2%)	
jdk.nashorn.internal.scripts.Script$Recompilation$9728$3269IIADIA$\^eval_.am3 ()		789 ms	(0.2%)	

ORACLE

Let's Use a Single Engine Instance!

```java
private static final ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");

protected EvaluationResult evaluate(final String plainText) throws ScriptException {
    final Bindings b = engine.createBindings();
    ScriptContext context = new SimpleScriptContext();
    context.setBindings(b, ScriptContext.ENGINE_SCOPE);
    engine.eval(new URLReader(scriptUrl), context);
    final String cipherText = (String) ((JSObject) b.get("encrypt")).call(null, plainText);
    final String roundTripPlainText = (String) ((JSObject) b.get("decrypt")).call(null, cipherText);
    return new EvaluationResult(cipherText, roundTripPlainText);
}
```

ORACLE®

How well does it perform?

- Much better!
- Let's look at percentage of the requests served within certain time (ms):

50%	195
66%	229
75%	281
80%	323
90%	424
95%	523
98%	681
99%	828
100%	2968

ORACLE

Code caching

- There's code caching within a single engine instance.

- If you pass URLReader to eval, Nashorn will retrieve already generated code for that URL on subsequent attempts.

- Code is only compiled once.

- `--class-cache-size=nnnn` can be used to govern the cache size. Defaults to 50 scripts.

- Still, why open a Reader on every request if we don't even read the script?

ORACLE®

Let's Use Compilable/CompiledScript!

```java
private static final ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
private static final URL scriptUrl = CryptoServletBase.class.getResource(SCRIPT_NAME);
private static final CompiledScript compiledScript =
    ((Compilable)engine).compile(new URLReader(scriptUrl)));

protected EvaluationResult evaluate(final String plainText) throws ScriptException {
    final Bindings b = engine.createBindings();
    ScriptContext context = new SimpleScriptContext();
    context.setBindings(b, ScriptContext.ENGINE_SCOPE);
    compiledScript.eval(context);
    final String cipherText = (String) ((JSObject) b.get("encrypt")).call(null, plainText);
    final String roundTripPlainText = (String) ((JSObject) b.get("decrypt")).call(null, cipherText);
    return new EvaluationResult(cipherText, roundTripPlainText);
}
```

ORACLE

How well does it perform?

- Somewhat better!
- Let's look at percentage of the requests served within certain time (ms):

	rdr	csr
50%	195	159
66%	229	194
75%	281	261
80%	323	289
90%	424	380
95%	523	455
98%	681	533
99%	828	710
100%	2968	3076

ORACLE

Are We Re-Evaluating Everything?

- There's an eval call on every request. Isn't that slow?
- Not necessarily, as the script defines functions.
- Code is compiled once, what happens on every evaluation is that Bindings is populated with the function objects.
- Function object is effectively a pair of (lexical scope, code). Cheap to construct.

ORACLE®

Flexible Separation of Compile and Run Time

- JavaScript (and most dynamic languages) don't mention "compile time" and "run time" in their specifications.

- In their world, a program is just run. Everything else is implementation detail.

- That's why when you integrate, you have a discrete set of choices of how to split these tasks.

- Of course, there's the further tiny detail of program's global variable namespace.

ORACLE®

What if We Used a Single Bindings Object?

```java
private static final ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
static {
    engine.eval(new URLReader(scriptUrl));
}
private static final JSObject encryptFunction = (JSObject)engine.get("encrypt");
private static final JSObject decryptFunction = (JSObject)engine.get("decrypt");

protected EvaluationResult evaluate(final String plainText) throws ScriptException {
    final String cipherText = (String) encryptFunction.call(null, plainText);
    final String roundTripPlainText = (String) decryptFunction.call(null, cipherText);
    return new EvaluationResult(cipherText, roundTripPlainText);
}
```

How well does it perform?

- Whoa!
- Let's look at percentage of the requests served within certain time (ms):

50%	35
66%	36
75%	38
80%	39
90%	42
95%	45
98%	52
99%	60
100%	2968

ORACLE®

Yes, But Is It Threadsafe?

ORACLE®

Let's Turn Optimism On

- Optimistic typing is off by default.
- If we turn it on, Nashorn's compiler will emit type-speculative code, and adaptively recompile code on-the-fly when needed.

ORACLE®

Let's Turn Optimism On

- We need to use Nashorn-specific API to instantiate a type-optimistic engine.

- Technique can be used in general to pass command-line flags to a Nashorn engine.

```java
import jdk.nashorn.api.scripting.NashornScriptEngineFactory;

private static final ScriptEngine engine;

static {
    final NashornScriptEngineFactory factory = new NashornScriptEngineFactory();
    engine = factory.getScriptEngine("--optimistic-types=true");
    …
}
```

ORACLE®

Non-Optimistic vs. Optimistic Performance

Percentage of the requests served within a certain time (ms)

	nonopt	opt
50%	35	12
66%	36	13
75%	38	14
80%	39	14
90%	42	17
95%	45	20
98%	52	26
99%	60	30
100%	2968	4703

2x-3x faster, …

… but slower to start up

ORACLE®

Why is Optimistic Slower to Start?

- When a type can't be proven statically, it'll be presumed to be int.
- When the assumption fails, code is recompiled.
- 35 functions are recompiled 67 times total for this application.

ORACLE

Why is Optimistic Slower to Start?

```
function bnpMultiplyTo(a,r) {
  var this_array = this.array;
  var r_array = r.array;
  var x = this.abs(), y = a.abs();
  var y_array = y.array;

  var i = x.t;
  r.t = i+y.t;
  while(--i >= 0) r_array[i] = 0;
  for(i = 0; i < y.t; ++i) r_array[i+x.t] = x.am(0,y_array[i],r,i,0,x.t);
  r.s = 0;
  r.clamp();
  if(this.s != a.s) BigInteger.ZERO.subTo(r,r);
}
```

ORACLE

Benefiting From Compile/Run Time Interleaving

```
var this_array = this.array;
 var r_array = r.array;
 var x = this.abs(), y = a.abs();
 var y_array = y.array;
```

- Compiler kicks in while code is running. It can peek into runtime objects.
- It evaluates side-effect free expressions and looks at their types.
- Above, when it recompiled because of "this.array" failed to be int, it peeked into "r.array" and saw it's an object.
- "y.array" couldn't be peeked into before "y" was evaluated, though.

ORACLE

Optimistic vs. Non-Optimistic Performance

- Time for a car analogy!
- Gears!
  - Lower gear: easier to start, but lower maximum speed
  - Higher gear: harder to start, but higher maximum speed

Image from http://libreshot.com/vehicles/gear-stick/

ORACLE

Optimistic vs. Non-Optimistic Performance

- Nashorn has no interpreter currently.

- Starts with a compiler; so already the lowest gear is 2nd.

- Optimistic types can be considered a 3rd gear.

- Also, there's no shifting mechanism at present… Whichever you start up with is the one you get for the whole duration of the engine.

Image from http://libreshot.com/vehicles/gear-stick/

ORACLE®

Using the Type Info Cache

- Nashorn can remember the type information between JVM runs.

- Disabled by default, but can be enabled with
`-Dnashorn.typeInfo.maxFiles=`*nnnn* system property.

  - Number specifies the number of cache files. There's one file per JavaScript function across any number of scripts, so plan accordingly.

  - If cache is outgrown, oldest entries get evicted.

  - Also possible to specify
`-Dnashorn.typeInfo.maxFiles=`*unlimited* for unlimited cache (can help you with initial sizing).

ORACLE®

Type Caching Performance Improvement

Longest request time (ms)

```
non-optimistic              2968
optimistic                  4703
optimistic w/type cache   3570
```

- It got better, but still somewhat slower startup than non-optimistic.
- It's a tradeoff we need to live with for now.

ORACLE®

Yes, But Is It Threadsafe?

- Try to stress it with two different texts to encrypt concurrently.
- Our servlet checks if it ends up with the same cleartext and if not, sends back a 500 Internal Server Error.

```
$ ab -n 1000 -c 2 http://…
…
Non-2xx responses: 0
…
```

```
$ ab -n 1000 -c 2 http://…
…
Non-2xx responses: 0
…
```

- Seems okay.
- BUT…

ORACLE®

Let's Make It Stateful

- Let's pass data through engine bindings. Don't do this at home.

```java
private static final ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
static {
    engine.eval(new URLReader(scriptURL));
}

protected EvaluationResult evaluate(final String plainText) throws Exception {
    engine.put("plainText", plainText);
    final String cipherText = (String) engine.eval("encrypt(plainText)");
    engine.put("cipherText", cipherText);
    final String roundTripPlainText = (String) engine.eval("decrypt(cipherText)");
    return new EvaluationResult(cipherText, roundTripPlainText);
}
```

ORACLE

Yes, But Is It Threadsafe?

```
$ ab -n 1000 -c 2 http://…
…
Non-2xx responses: 12
…
```

```
$ ab -n 1000 -c 2 http://…
…
Non-2xx responses: 12
…
```

- Oopsie.

ORACLE®

Shared Mutable State Is the Enemy of Thread Safety

```java
protected EvaluationResult evaluate(final String plainText) throws ScriptException {
    engine.put("plainText", plainText);
    final String cipherText = (String) engine.eval("encrypt(plainText)");
    engine.put("cipherText", cipherText);
    final String roundTripPlainText = (String) engine.eval("decrypt(cipherText)");
    return new EvaluationResult(cipherText, roundTripPlainText);
}
```
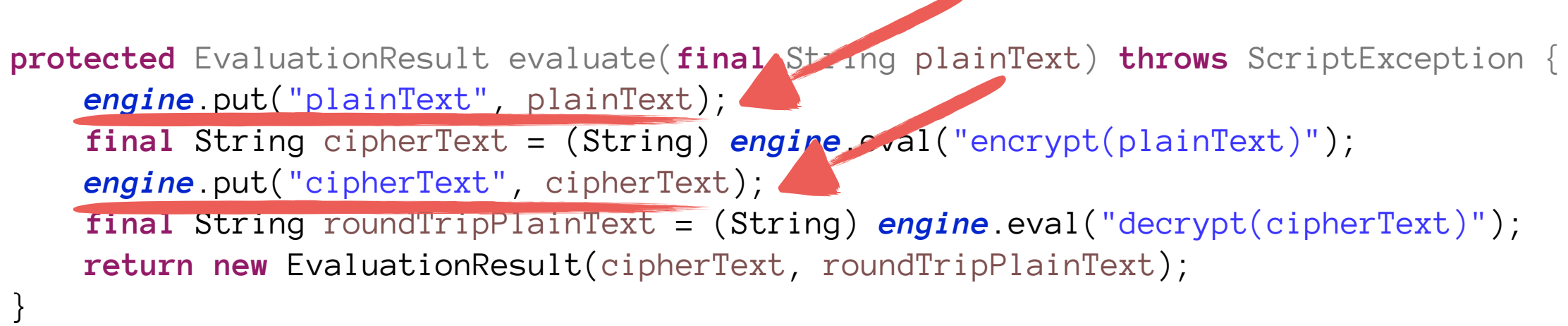
ORACLE

So, Let's Go Back To Separate Bindings

- We already saw the separate bindings performance for Reader vs. CompiledScript.

- Let's see it with optimistic too.

	rdr	csr	opt
50%	195	159	135
66%	229	194	145
75%	281	261	154
80%	323	289	160
90%	424	380	234
95%	523	455	288
98%	681	533	369
99%	828	710	459
100%	2968	3076	3682

ORACLE

Recommendations

- Keep it stateless.
- If you can't:
  - Use a single engine instance, but…
  - … use separate bindings.
  - … or make it Java specific with synchronization:

```
var syncedFn = Java.synchronized(fn, lockObj);
```

ORACLE®

Why Use a Single Engine?

- Code is cached on engine level.

- Hidden classes are maintained on engine level.

- These are interleaved:

  - code contains call sites, linking is hidden-class specific.

ORACLE®

Why Use a Single Engine?

```
┌──────────┐        ┌──────────┐        ┌──────────┐
│  Engine  │───────▶│   Code   │───────▶│  Script  │
│          │        │  cache   │        │  class   │
└──────────┘        └──────────┘        └──────────┘
```

```
┌──────────┐   ┌──────────┐   ┌──────────┐
│ Bindings │   │ Bindings │   │ Bindings │
└──────────┘   └──────────┘   └──────────┘
```
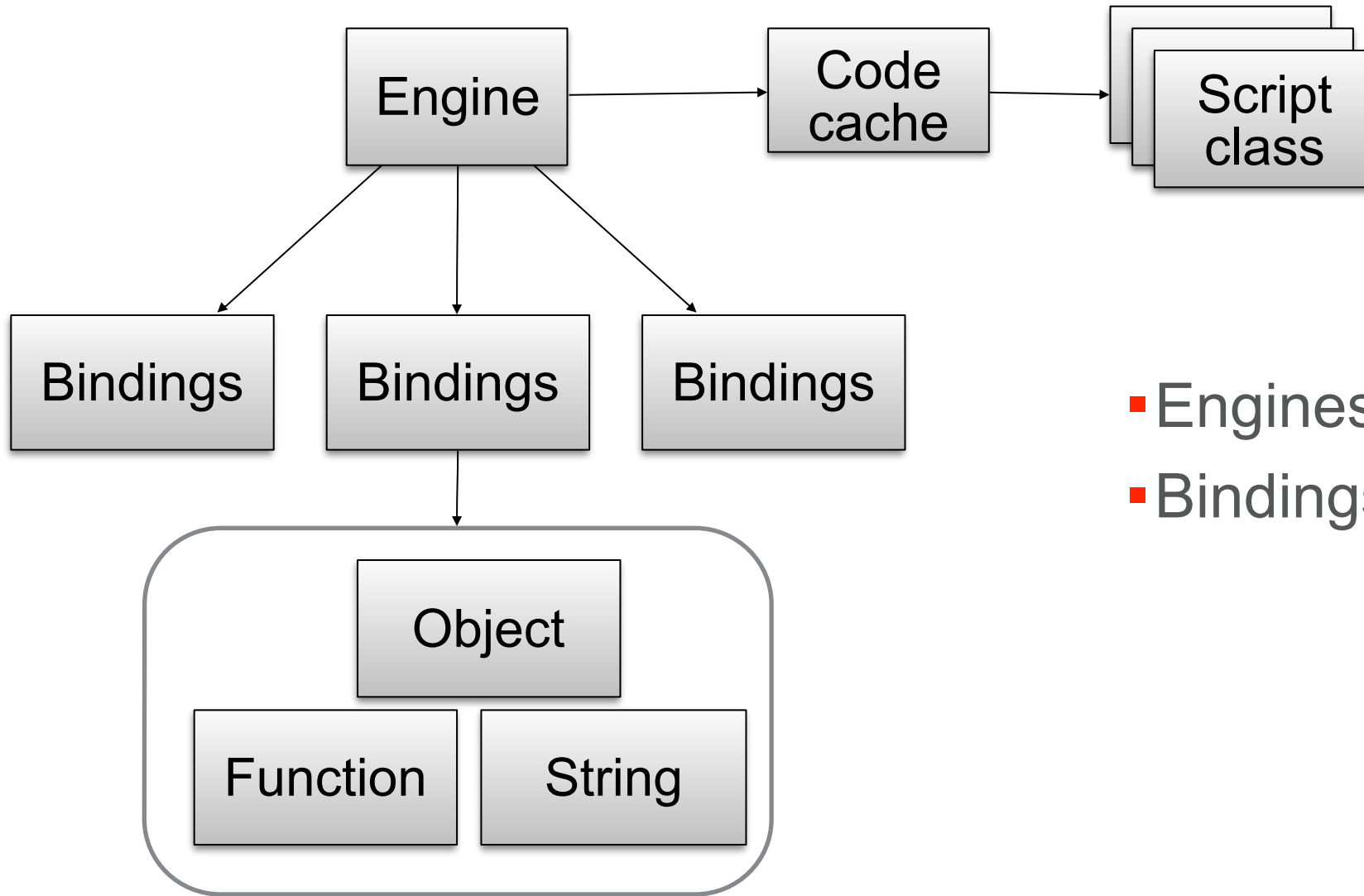
- Engines are thread-safe.
- Bindings are not.

```
╭─────────────────────────────╮
│        ┌──────────┐         │
│        │  Object  │         │
│        └──────────┘         │
│  ┌──────────┐ ┌──────────┐ │
│  │ Function │ │  String  │ │
│  └──────────┘ └──────────┘ │
╰─────────────────────────────╯
```

ORACLE

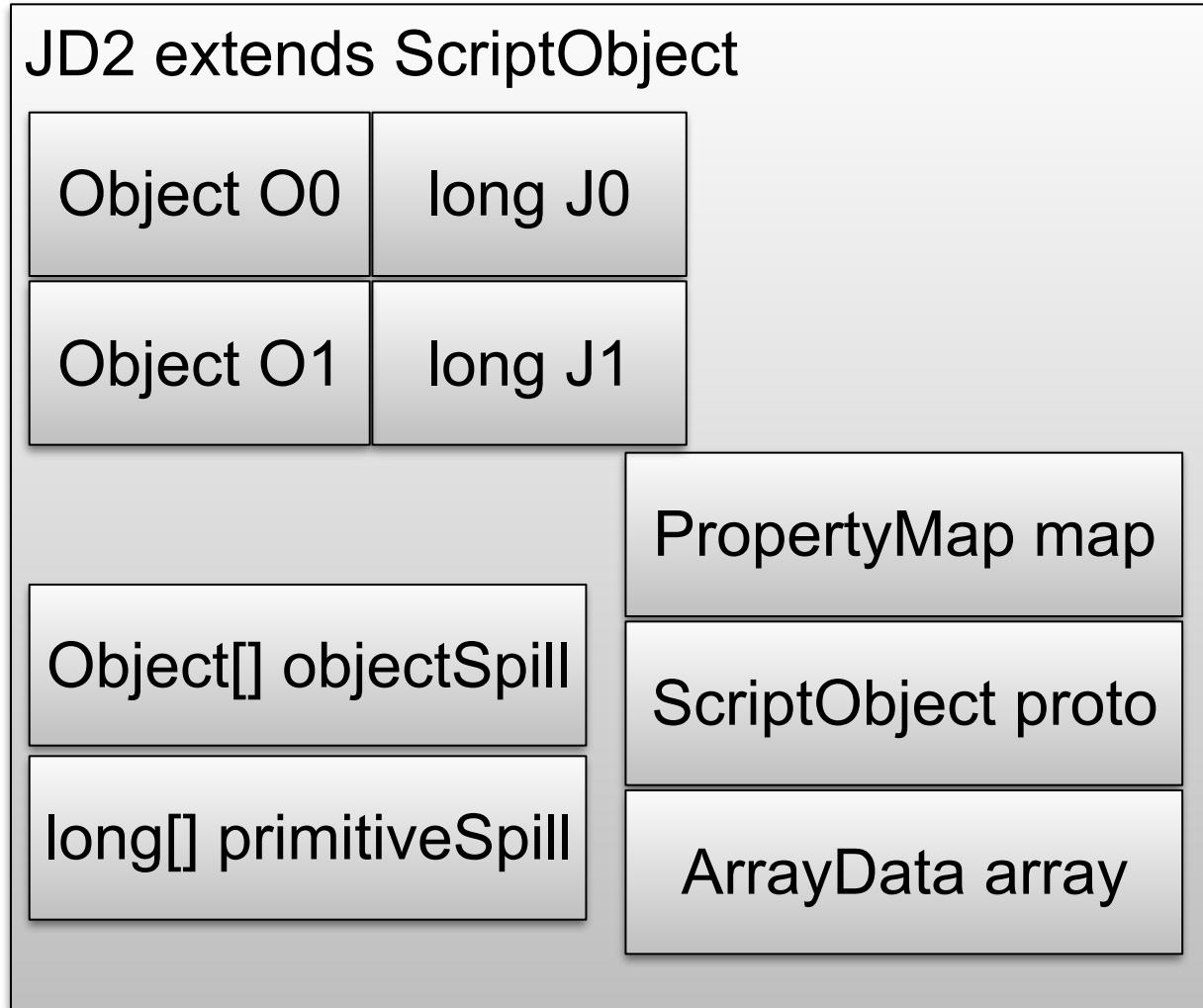# If you move data across bindings, though…

- Linking is less effective.
- Nashorn has two representations of internal objects:
  - ScriptObject (never seen outside of a Bindings it belongs to)
  - ScriptObjectMirror (implements JSObject)
- ScriptObject outside of a Bindings is always mirrored.
- Even when used in a different engine instance.
- Even when used in a same engine, but different Bindings.

ORACLE®

Structure of a Nashorn Object

JO2 extends ScriptObject

Object O0

Object O1

Object[] objectSpill

PropertyMap map

ScriptObject proto

ArrayData array
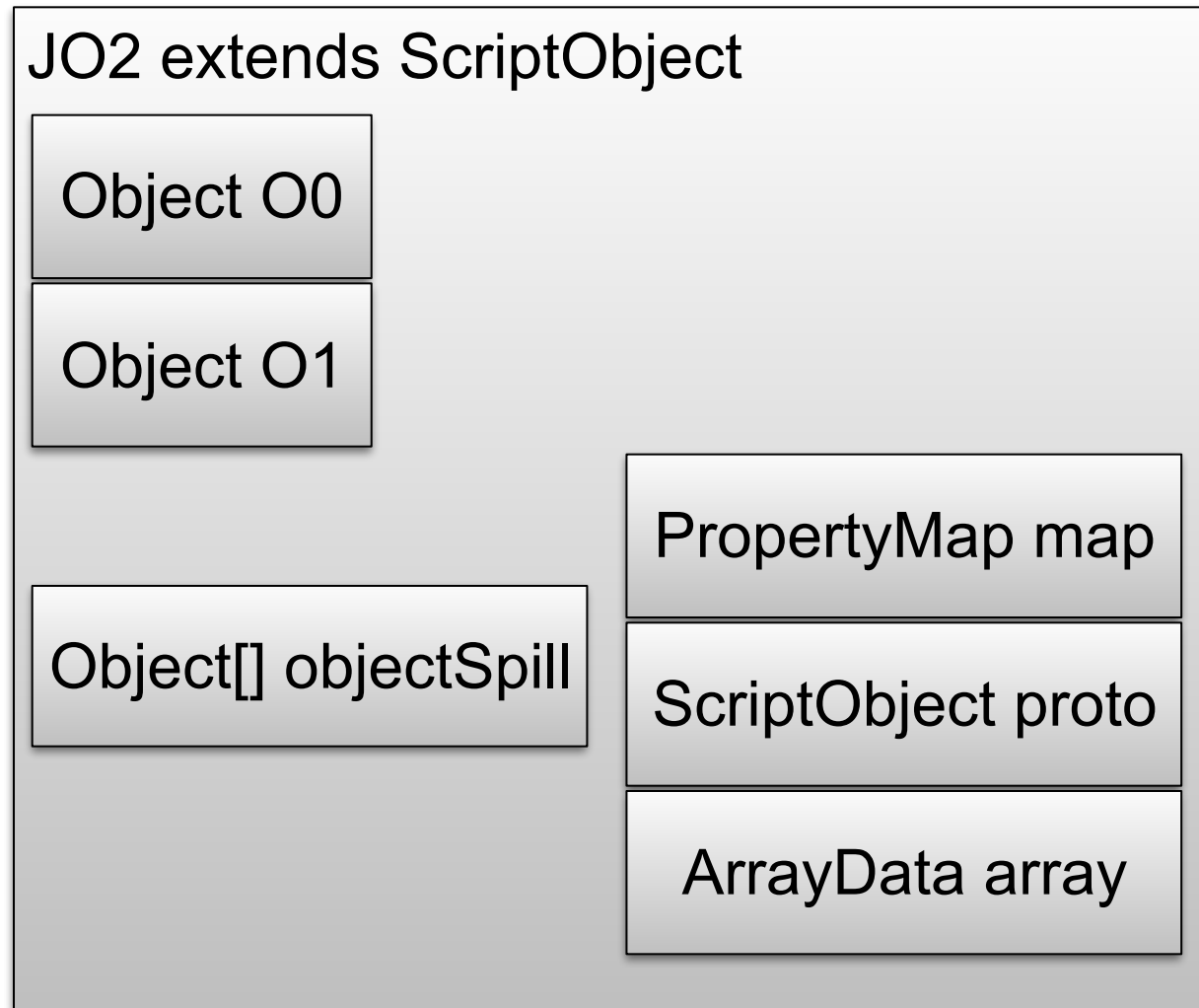
- Several general purpose Object fields, and an array to handle spillover.
- "Map" is what other engines call "Hidden class", mapping names to general purpose fields.
- In basic case, everything is boxed.

ORACLE

Structure of a Nashorn Object

JD2 extends ScriptObject

Object O0	long J0
Object O1	long J1

Object[] objectSpill

long[] primitiveSpill

PropertyMap map

ScriptObject proto

ArrayData array

- When optimistic typing is used, we add 64-bit fields that can hold a primitive int/long/double.

ORACLE

How Are Properties Allocated

- Number of fields is determined at compile time for object literals, scope objects, and constructors.

- In the examples below, Nashorn always figures out the objects have two properties.
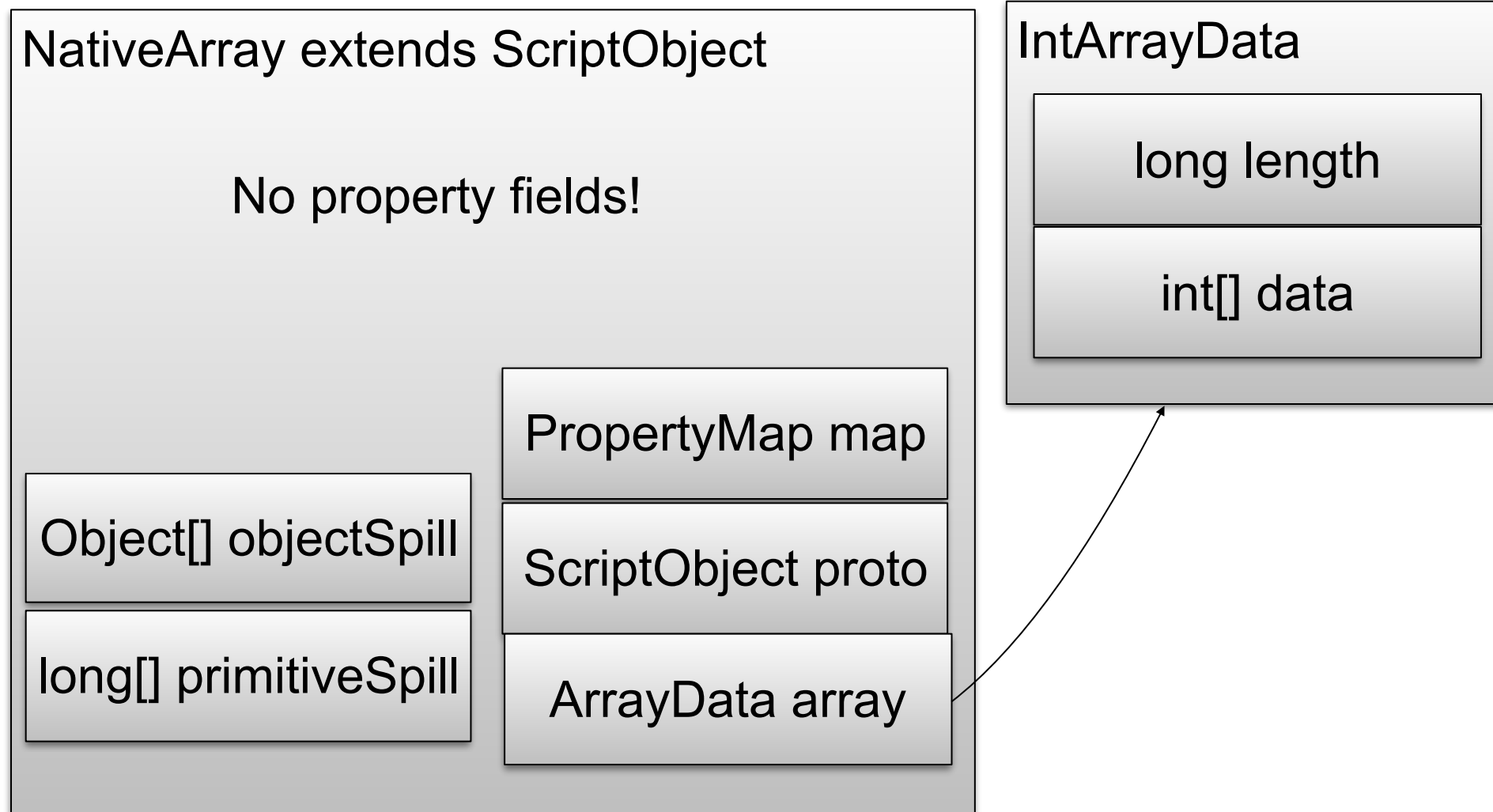
```
{
  a: 1,
  b: 2
}
```

```
function F(a, b) {
    this.a = a;
    this.b = b;
}
```

```
function F(a, b) {
    return function() {
        return [a, b];
    }
}
```

- {} will have 0 fields, everything goes into spills.

- Numeric properties usually go into ArrayData.

ORACLE®

Structure of a Nashorn Array object

NativeArray extends ScriptObject

No property fields!

Object[] objectSpill

long[] primitiveSpill

PropertyMap map

ScriptObject proto

ArrayData array

IntArrayData

long length

int[] data

ORACLE

Use Homogeneous Arrays

- Nashorn internally specializes arrays.
- An empty array starts out as an (empty) array of 32-bit signed ints.
- Can evolve to 64-bit ints, 64-bit floating point, or object.
- When a large int array gets an object element, all ints get boxed.
  - Undefined elements and deleted indices are tracked with a bitmap.
  - They don't cause change of underlying storage type.
- Traversal on object arrays is obviously slower.
- Try to avoid differently typed sentinels.

ORACLE

Avoid eval

- If you use eval(), compiler loses most of its reasoning abilities.
- Function with eval becomes variable arity; we don't type-specialize it.
- No function local variables are stored in JVM local variables.
  - True even if a nested function contains eval().
- Optimistic typing still works though, but all variable access goes through property getters/setters on a lexical scope object.

ORACLE®

Avoid with

- with statement creates a dynamic scope fork within the lexical scope.
- It's as scary as it sounds. Not as bad as eval, though.
- Variables potentially accessed within eval are promoted into scope.

```
var x; // goes into scope object
var y; // can remain JVM local variable

with(z) {
    do_something(x);
}
```

ORACLE®

Property getter

```
var array = [
    { a: 1}, // "map1"
    { b: 1, a: 2}, // "map2"
    { c: 1, b: 2, a: 3}, // "map3"
    { d: 1, c: 2, b: 3, a: 3}, // "map4"
];

var x = 0;
for(var j = 0; j < array.length; ++j) {
    x += array[j].a;                              relink();
}
```

ORACLE®

Property getter

```
var array = [
    { a: 1}, // "map1"
    { b: 1, a: 2}, // "map2"
    { c: 1, b: 2, a: 3}, // "map3"
    { d: 1, c: 2, b: 3, a: 3}, // "map4"
];

var x = 0;
for(var j = 0; j < array.length; ++j) {
    x += array[j].a;
}
```

```
if(obj.map == obj.map1) {
    obj.J0
} else {
    relink();
}
```

ORACLE®

Property getter

```
var array = [
    { a: 1}, // "map1"
    { b: 1, a: 2}, // "map2"
    { c: 1, b: 2, a: 3}, // "map3"
    { d: 1, c: 2, b: 3, a: 3}, // "map4"
];

var x = 0;
for(var j = 0; j < array.length; ++j) {
    x += array[j].a;
}
```

```
if(obj.map == obj.map2) {
    obj.J1
} else if(obj.map == obj.map1) {
  obj.J0
} else {
    relink();
}
```

ORACLE®

Property getter

```
var array = [
    { a: 1}, // "map1"
    { b: 1, a: 2}, // "map2"
    { c: 1, b: 2, a: 3}, // "map3"
    { d: 1, c: 2, b: 3, a: 3}, // "map4"
];

var x = 0;
for(var j = 0; j < array.length; ++j) {
    x += array[j].a;
}
```

```
if(obj.map == obj.map3) {
    obj.J2
} else if(obj.map == obj.map2) {
    obj.J1
} else if(obj.map == obj.map1) {
  obj.J0
} else {
    relink();
}
```

ORACLE®

Avoid polymorphic call sites

- Every object carries a pointer to a hidden class ("map").

- Call sites (e.g. "get property color" in `obj.color`) are linked with guards that check map referential identity.

- All objects at a call site have the same map: fast. Only a Java field getter/setter or array element getter/setter.

- As the number of different maps at a call site increases: slows down.

    - As cascading `if(map == map1)/else if(map == map2)/…`

- At 8 cascades, site switches to a new, megamorphic linkage where it does a lookup through map on every invocation.

ORACLE

Avoid polymorphic call sites

```
var array = [
    { a: 1, b: 1},
    { a: 1, c: 1},
    { a: 1, d: 1},
    { a: 1, e: 1},
    { a: 1, f: 1},
    { a: 1, g: 1},
    { a: 1, h: 1},
    { a: 1, i: 1},
    { a: 1, j: 1}, // mega
];

var x = 0;
for(var i = 1; i < 10000000; ++i) {
    for(var j = 0; j < array.length; ++j) {
        x += array[j].a;
    }
}
```

Megamorphic call site is ".a"

- The line marked "`//mega`" will take the execution time from 1.7 to 8.5 seconds on my machine.
- You can use `--log=fields` to have Nashorn warn you when a call site goes megamorphic.

ORACLE

Summary

- Nashorn does a lot under the hood to make sure your JavaScript code runs fast: type specialized compilation, static type inference, optimistic typing, typed arrays. The result is more than the sum of its parts.

- When you integrate with javax.script API, you must take care how you compose it into your system (single engine, either separate bindings for separate threads, or single bindings with explicit synchronization or other way to take care of shared mutable state.)

- Initializing objects as literals, scopes, or in constructor functions is most efficient storage-wise.

- Avoid polymorphism if you can.

- With statements and eval calls defeat lots of compiler optimizations.

ORACLE®