
ORACLE®


Java™
ORACLE

Modules in Nashorn

Sundararajan Athijegannathan, Michael Haupt
Nashorn Team, Java Platform Group
JVMLS, August 2016

Safe Harbor Statement

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.

Agenda

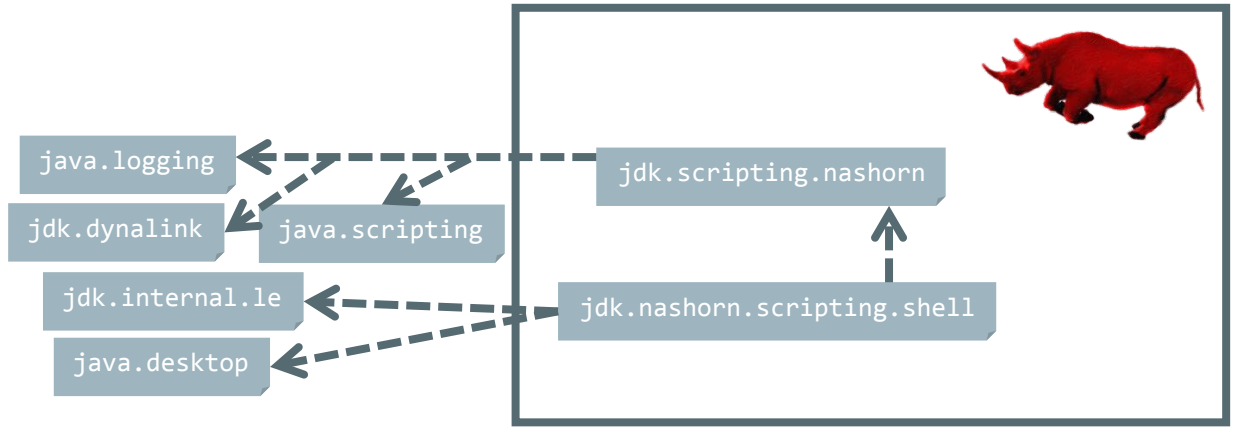
1. Nashorn Background
2. Class Loaders in Nashorn
3. Dynamic Modules in Nashorn
4. Cross-Layer Module (Qualified) Exports and Reads

What is Nashorn?

- ECMAScript 262 Edition 5.1 compliant script engine in JDK 8+
- Partial support for ECMAScript 6
  - See JEP 292 for details: http://openjdk.java.net/jeps/292
- Both programmatic access and command line `jjs` REPL
- Unix shell-like constructs as language extensions in `-scripting` mode
- Java API access from scripts
- Fine-grained security for scripts
- “compact1” compliant

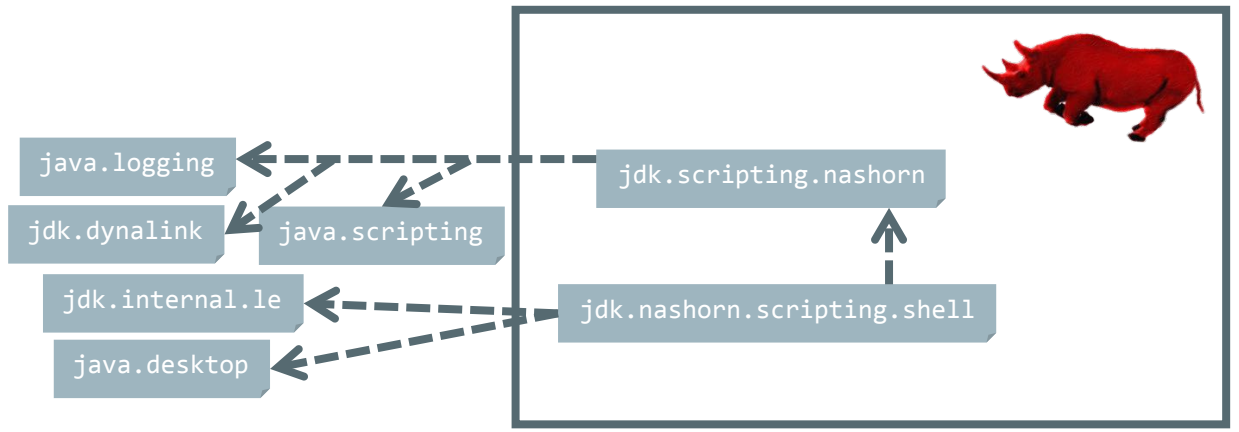
Nashorn Modules

- `jdk.scripting.nashorn`: “Nashorn module”
  - Depends only on `java.scripting`, `jdk.dynalink`, and `java.logging`
  - Provides `javax.script.ScriptEngineFactory`
  - Provides and uses Dynalink linker services
  - “compact1” compliant



Nashorn Modules

- `jdk.nashorn.scripting.shell`: "`jjs` module"
  - Nashorn `jjs` tool source code
- `jdk.dynalink`: "Dynalink module"
  - Independent module
  - Lives in Nashorn repository for historical reasons
  - Dynalink used to be an internal detail of Nashorn in JDK 8
  - Independent API in JDK 9 (JEP 276)



Nashorn Background

- No interpreter, compilation to bytecode only
- Uses JDK-internal ASM for ECMAScript compilation (class files)
- Almost 100 % `INVOKEDYNAMIC` for basic operations
  - Runtime calls are statically linked (e.g., == ➔ `ScriptRuntime.EQ`)
- Dynalink for `INVOKEDYNAMIC` call site bootstrap methods

Nashorn APIs

`jdk.nashorn.api.tree` API
- Parser API
- Similar to javac tree API for ECMAScript (JEP 236)
- Full ES6 support

`jdk.nashorn.api.scripting` API
- Additional API "missing" in `javax.script`
- Script object reflection: `ScriptObjectMirror`
- `JSObject`: user-defined "script-friendly" objects implemented in Java

`jdk.nashorn.api.linker` package
- Nashorn's own Dynalink linker exported as a service for other JVM languages
- JRuby or Groovy could use Dynalink and provide tight integration with Nashorn

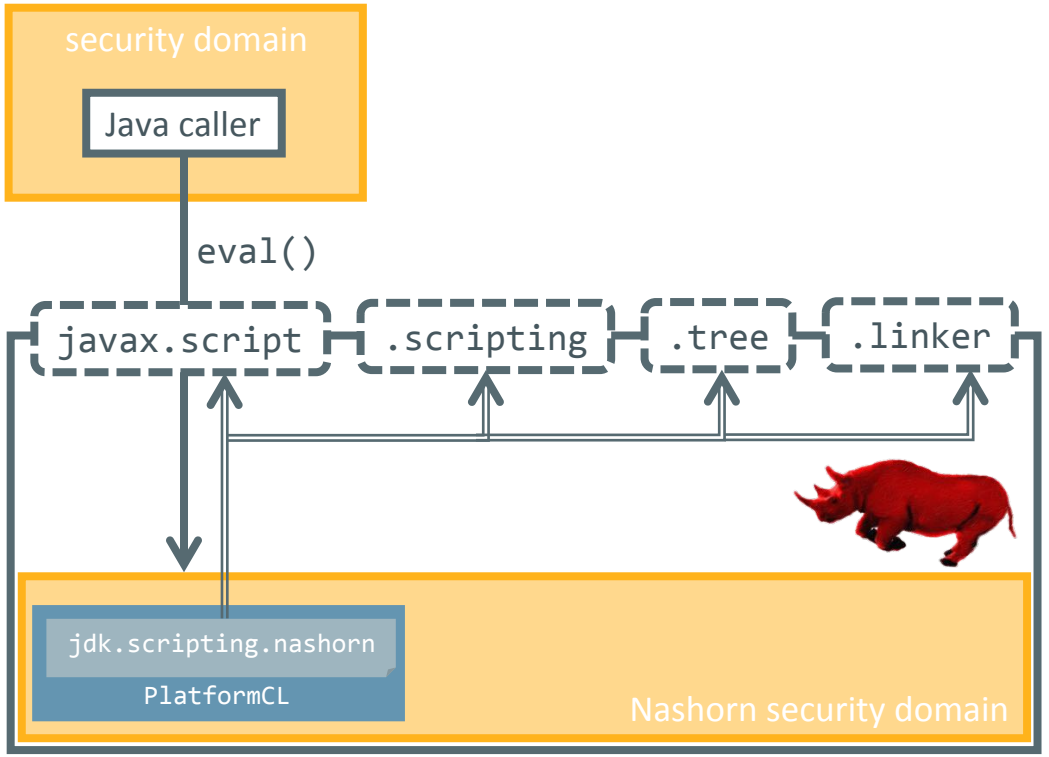
`javax.script` API (JSR 223)

Nashorn Java Integration

- Tight Java integration
  - Access to public members of public classes from JavaScript code
  - Access checks for restricted / exported packages
  - Lambdas as script functions and script functions as lambdas
- No Java reflection and `java.lang.invoke` access from scripts by default
  - Nashorn-specific reflection permission check
- `Java.extend` API to subclass Java classes and implement interfaces
  - Nashorn "adapter classes" generated for Java subclasses and interface implementations
  - Cannot use `java.lang.reflect.Proxy`, special support for `@CallerSensitive`
  - Subclassing support, `INVOKEDYNAMIC` generation for calls
  - Access to Nashorn runtime from adapters for internals

Nashorn Security

- Script sandbox
  - For scripts loaded via `ScriptEngine.eval()` methods
  - Java caller permissions not inherited by scripts
- Nashorn module
  - Assigned to platform class loader
  - Given `AllPermission`
- Fine-grained URL-based script permissions for scripts loaded via ...
  - `load`, `loadWithNewGlobal` builtins
  - `jdk.nashorn.api.scripting.URLReader`

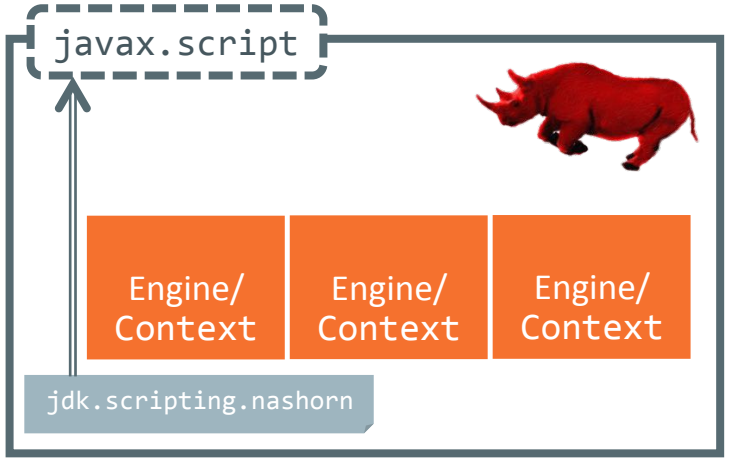


Agenda

1. Nashorn Background
2. Class Loaders in Nashorn
3. Dynamic (Run-Time) Modules in Nashorn
4. Cross-Layer Module (Qualified) Exports and Reads

Nashorn Background

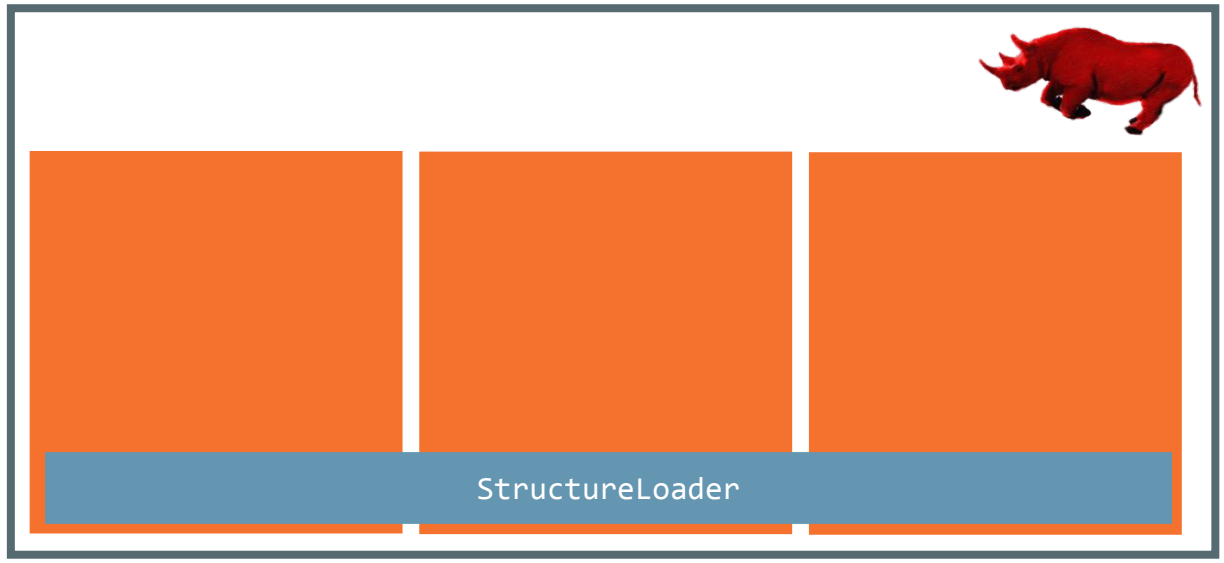
- `jdk.nashorn.internal.runtime.Context`
  - Isolation between multiple Nashorn “instances”
  - Manages configuration options, class loaders, script access to Java classes, compilation and loading of entry points
- `javax.script` API (JSR 223)
  - Supported for programmatic access
  - 1:1 correspondence between `NashornScriptEngine` and Nashorn `Context` instances



Class Loaders in Nashorn

`jdk.nashorn.internal.runtime.StructureLoader`

- Single instance per process
- Generates and loads “struct” classes
  - A bunch of `Object`/`double` fields
  - To store data for JS objects
- Shared across multiple Nashorn `Contexts`
  - Consequently, across multiple Nashorn `ScriptEngines`

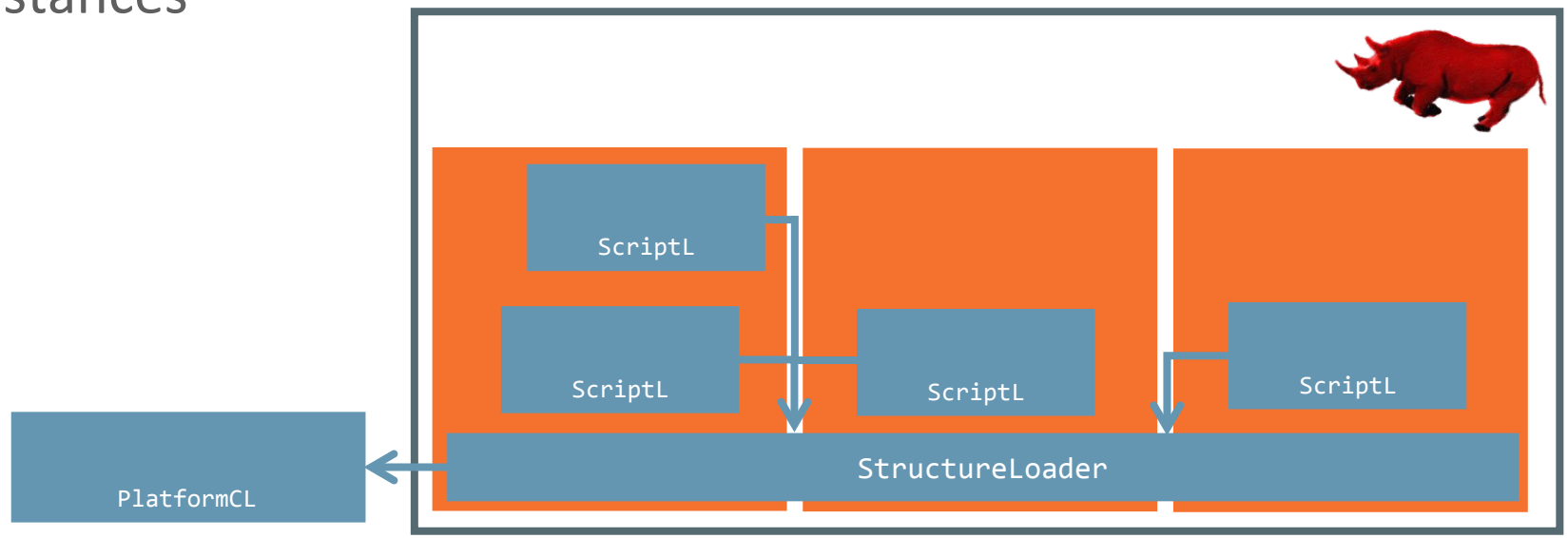


Class Loaders in Nashorn

`jdk.nashorn.runtime.internal.ScriptLoader`

- Loads one or more compiled script classes
- Fine-grained or one per Nashorn `Context`
  - Default is *fine-grained* (one per script)
  - Many `ScriptLoader` instances per `Context`
- Delegation chain
  - `ScriptLoader` → `StructureLoader` → platform loader

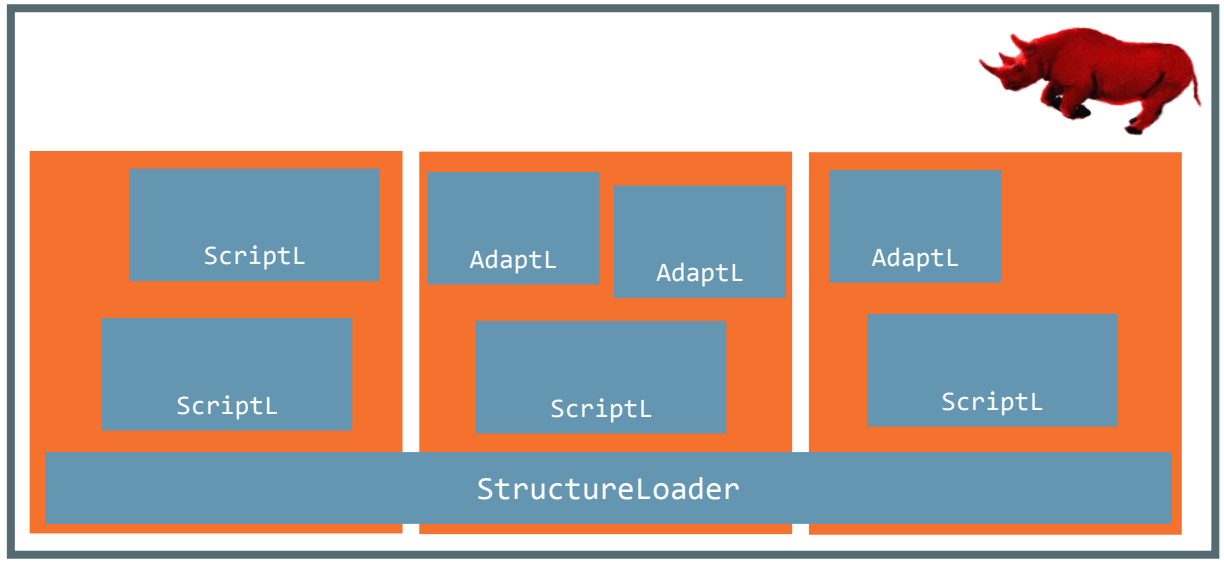
- Some script classes are loaded via `Unsafe.defineAnonymousClass()`
- A `ScriptLoader`-loaded class is used as host class
- This proves to be very effective in `eval` and/or recompilation-intensive scripts



Class Loaders in Nashorn

`jdk.nashorn.internal.runtime.linker.JavaAdapterClassLoader$...`

- Used to load Java interface implementations and Java subclasses implemented in scripts
- One class loader instance per super type combination
- Delegation chain
  - Parent loader must be able to see all types in implemented super type, and should see Nashorn runtime classes (e.g., `ScriptObject`)
  - Loader is chosen accordingly, e.g., thread context loader

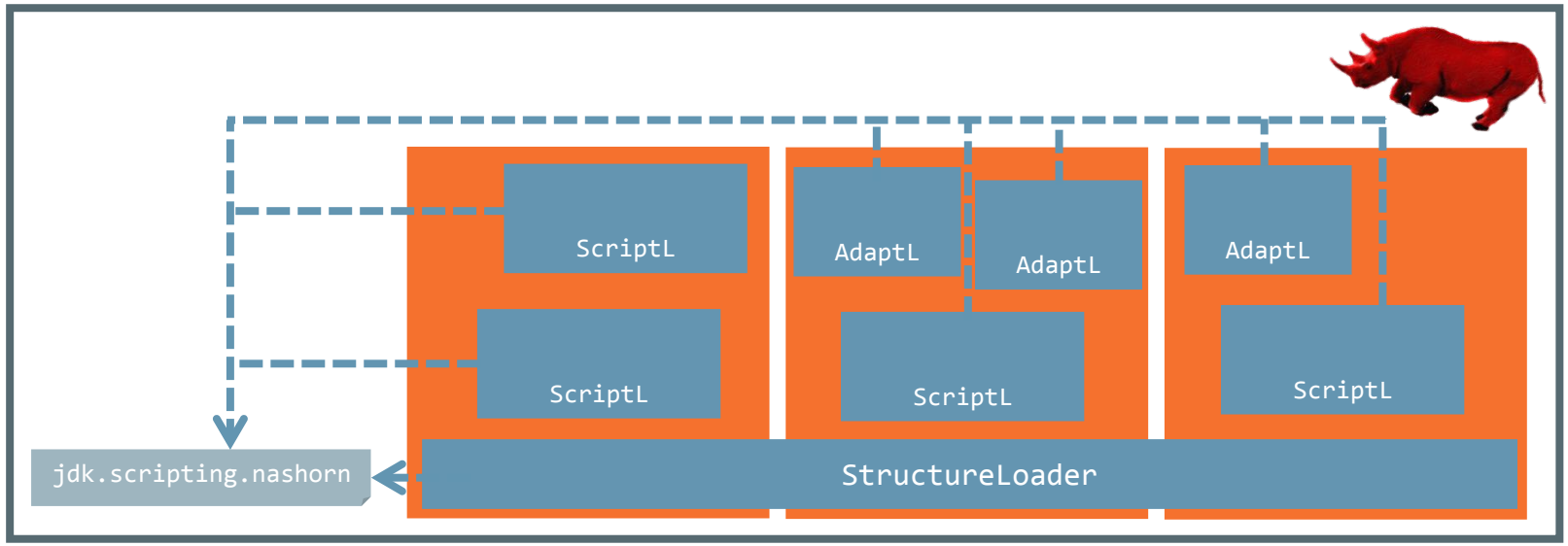


Agenda

1. Nashorn Background
2. Class Loaders in Nashorn
3. Dynamic Modules in Nashorn
4. Cross-Layer Module (Qualified) Exports and Reads

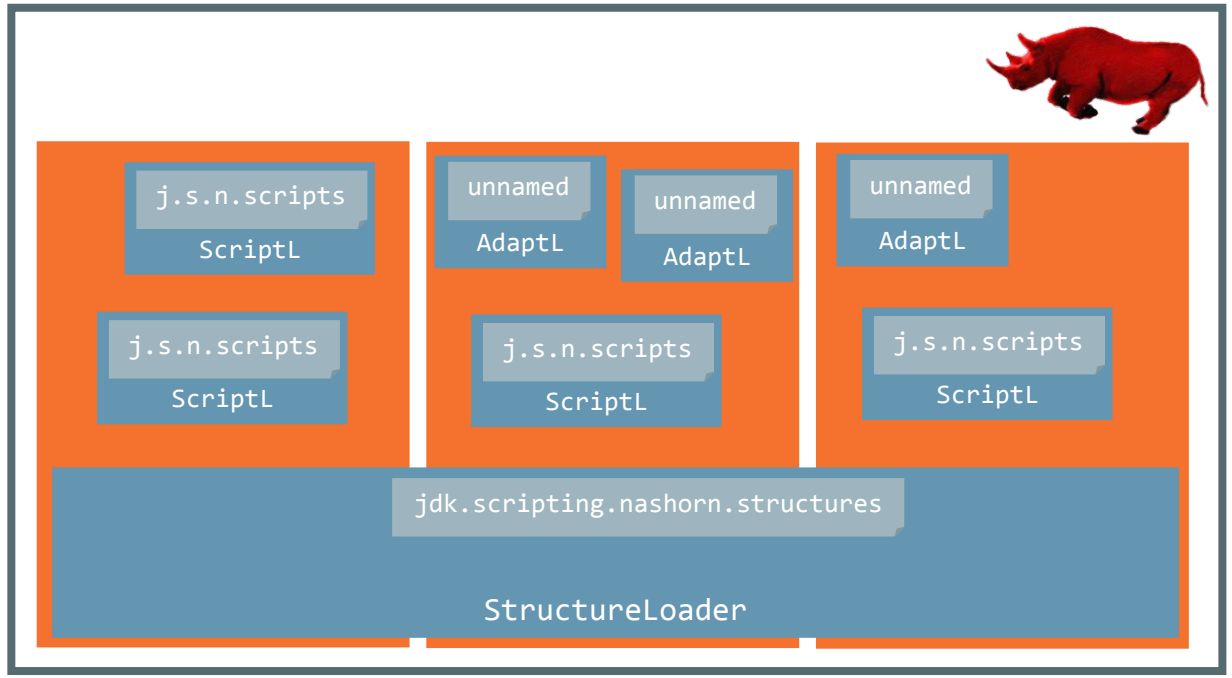
Why Dynamic Modules?

- Nashorn module only exports `jdk.nashorn.api.*` packages
- Access to Nashorn internals required by compiled script classes, generated "struct" classes, Java adapters
- Generated code should be accessible from Nashorn runtime (only) as well
- Default unnamed modules for generated classes are not enough



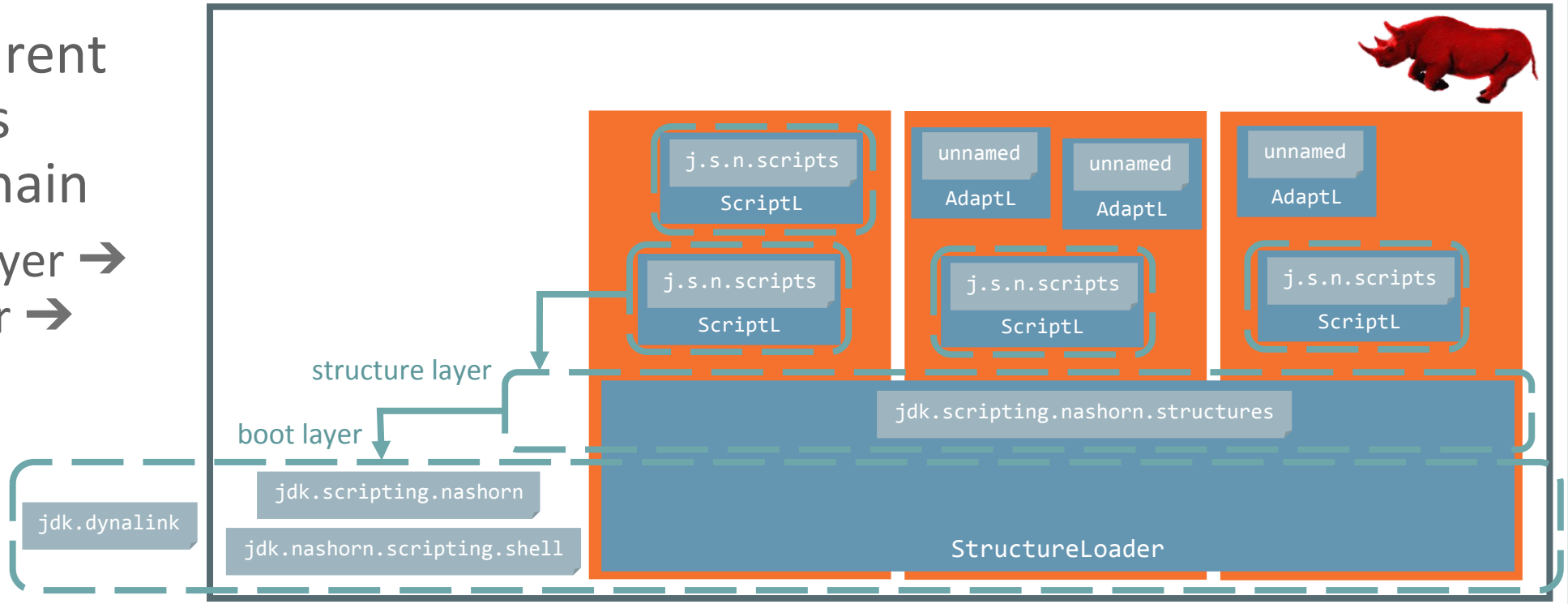
Dynamic Modules in Nashorn

- Dedicated dynamic *named* modules
  - For the `StructureLoader`: `jdk.scripting.nashorn.structures`
  - For `ScriptLoaders`: `jdk.scripting.nashorn.scripts` (per loader)
- Each adapter loader has its own dynamic *unnamed* module



Dynamic Modules in Nashorn

- Nashorn source modules live in boot layer
  - Boot layer configuration cannot be modified
  - `java.lang.reflect.Layer` API is used to create single-module layers
- Layer child/parent chain matches class loader chain
  - Some script layer ➔ structure layer ➔ boot layer



Agenda

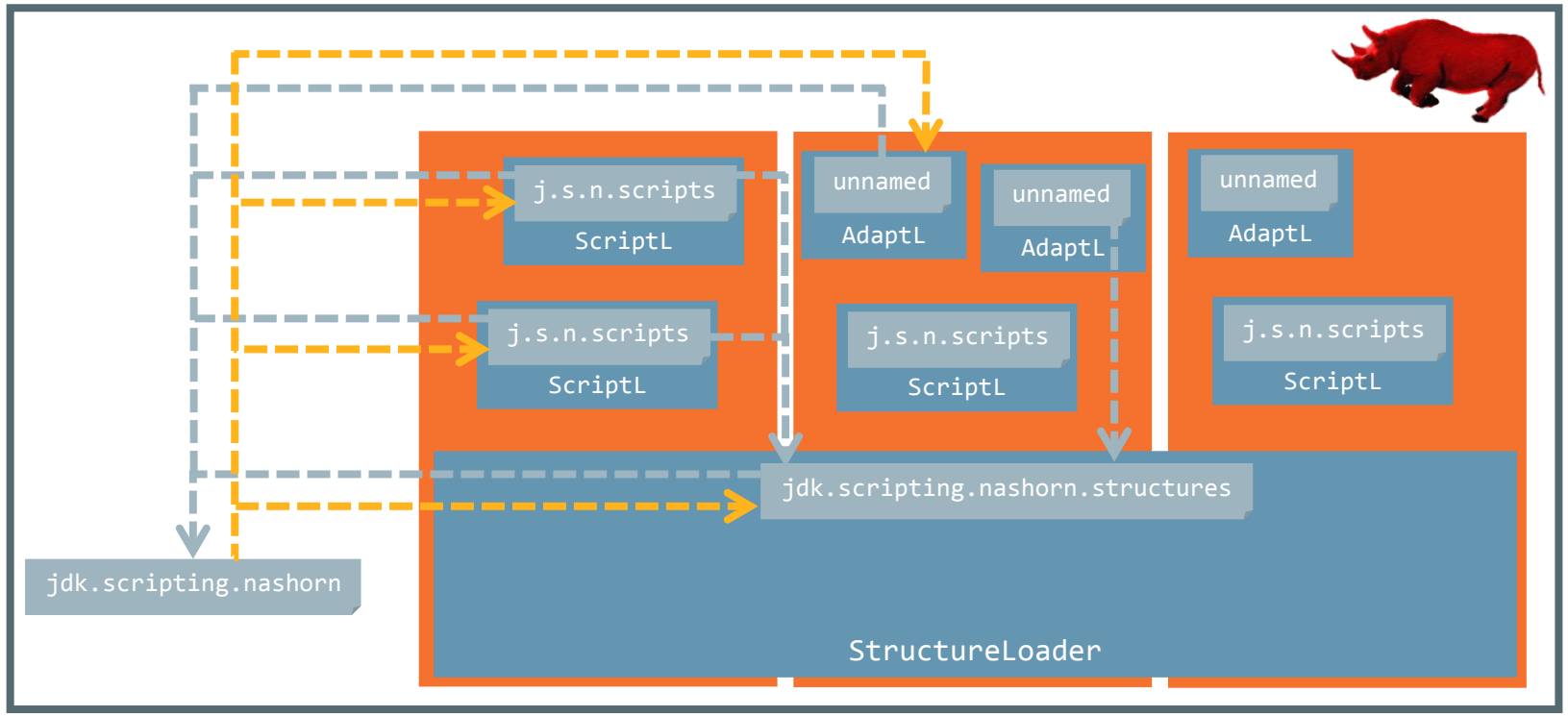
1. Nashorn Background
2. Class Loaders in Nashorn
3. Dynamic Modules in Nashorn
4. Cross-Layer Module (Qualified) Exports and Reads

Access Control in Nashorn

- JDK 8(u)
  - Nashorn uses package access runtime permission checks to implement access restrictions
  - All Nashorn packages except `jdk.nashorn.api.*` packages are `package.access` permission checked
- In JDK 9, dynamic named modules are used to enforce internal Nashorn access to generated/compiled code and vice versa

Access Control in Nashorn

- Characteristics
  - Each Java adapter loader has its own dynamic *unnamed* module
  - Script loaders and structure loader have their own dynamic *named* modules
- Rules
  - Nashorn runtime needs access to scripts/structs/ adapters and vice versa
  - Scripts and adapters need access to structs
  - No access between different script modules

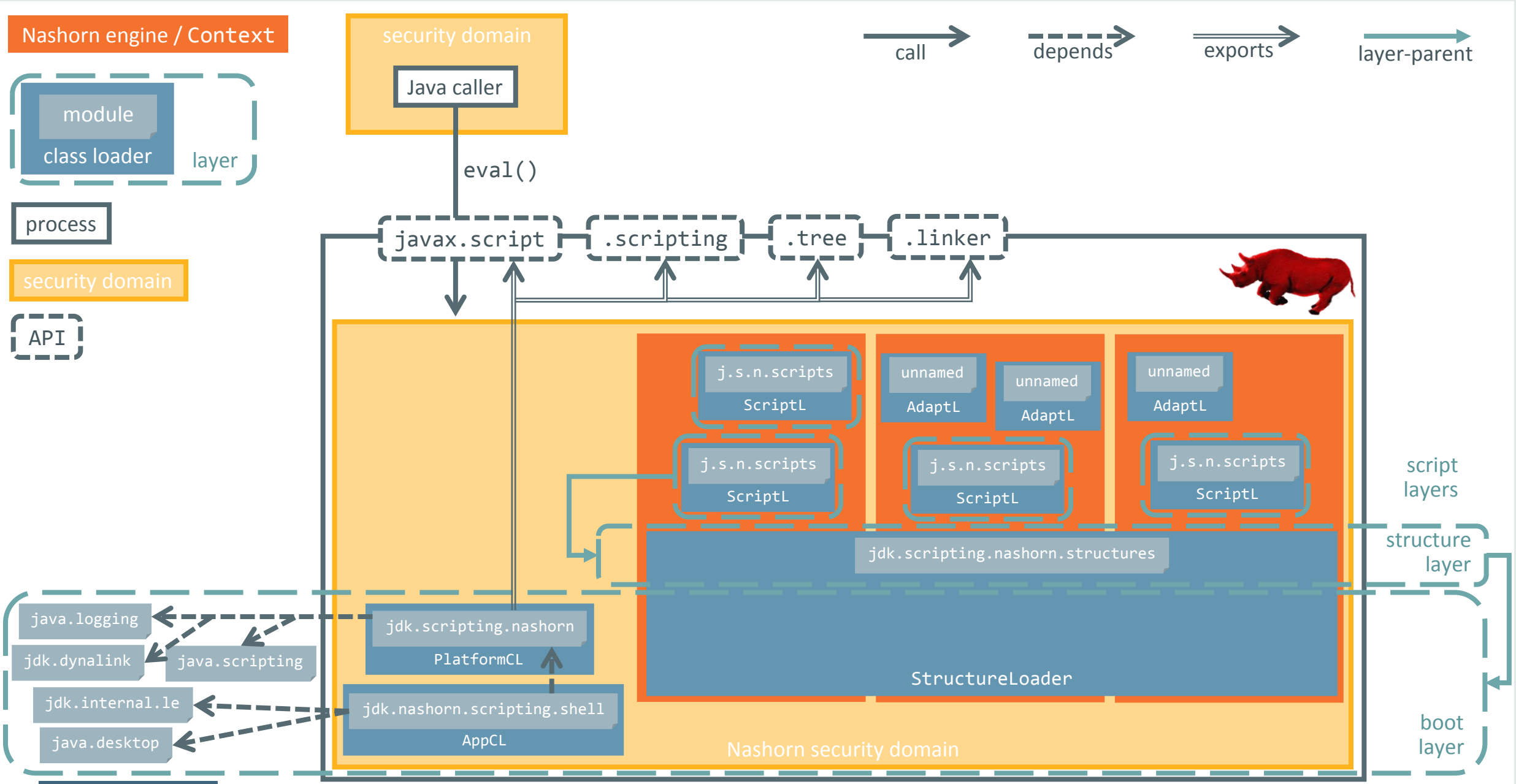


Managing Read/Export Edges

- Enabling access
  - While core reflection can ignore the module graph, method handles cannot
  - As Dynalink uses method handles, we have to enable the desired access patterns by adding the required read/exports edges in the module graph.
- `java.lang.reflect.Module.addReads/addExports`
  - To manage module graph edges between modules
  - Both `StructureLoader` and `ScriptLoader`(s) load `ModuleGraphManipulator` whose `<clinit>` code (qualified) exports package to Nashorn module
- Nashorn module
  - (Qualified) exports select `jdk.nashorn.internal.*` packages to generated code via `Module.addExports()`
  - Adds read links to generated modules to be able to create method handles

Managing Read/Export Edges

- Java adapter loaders need no explicit read/export edges
  - Unnamed modules implicitly read all other modules / export all public types
  - There is only one package with a public class in an adapter loader
- Qualified exports to unnamed modules
  - `namedModule.addExports("com.acme.internal", unnamedModule)`: qualified exports from Nashorn module to unnamed adapter module
  - Avoids having to add "read" links from adapter module to other referenced modules



Q&A

Integrated Cloud

Applications & Platform Services


Java™
ORACLE


ORACLE®