
Java™
ORACLE®

CREATE THE FUTURE

Future of Java

Java 9 and beyond

Vladimir Ivanov
HotSpot JVM Compiler
October 21, 2014

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.

Program Agenda

1. Overview
2. Project Jigsaw
3. Project Valhalla
4. Project Panama

Java Language-VM co-evolution

Where to put new features?

- Do it all in the front-end compiler
  - Generics, checked exceptions, autoboxing

Java Language-VM co-evolution

Where to put new features?

- Do it all in the front-end compiler
  - Generics, checked exceptions, autoboxing
- Do it mostly in the VM
  - New bytecodes, constant types, classfile attributes, privileged runtime, Unsafe
  - Front-end compiler is just syntax for VM features

Java Language-VM co-evolution

Where to put new features?

- Do it all in the front-end compiler
  - Generics, checked exceptions, autoboxing
- Do it mostly in the VM
  - New bytecodes, constant types, classfile attributes, privileged runtime, Unsafe
  - Front-end compiler is just syntax for VM features
- Mix and match
  - VM provides sensible low-level building blocks
  - Front-end compiler uses building blocks to implement feature

Language-VM co-evolution

The balancing act

- Try to balance
  - Keep Java language complexity isolated from VM
  - Avoiding mismatch between language and VM

Language-VM co-evolution

The balancing act

- Try to balance
  - Keep Java language complexity isolated from VM
  - Avoiding mismatch between language and VM
- How to win: find key language-agnostic VM/JDK improvements
  - Example: Lambda metafactory
  - Other compilers are free to use – or not

Language-VM co-evolution

The balancing act

- Try to balance
  - Keep Java language complexity isolated from VM
  - Avoiding mismatch between language and VM
- How to win: find key language-agnostic VM/JDK improvements
  - Example: Lambda metafactory
  - Other compilers are free to use – or not
- What not to do: push Java's wildcards into VM type system
  - A naïve version of reification would do this

Project Jigsaw

Java Modularization

Project Jigsaw

Java Modularization

Project Jigsaw

Motivation

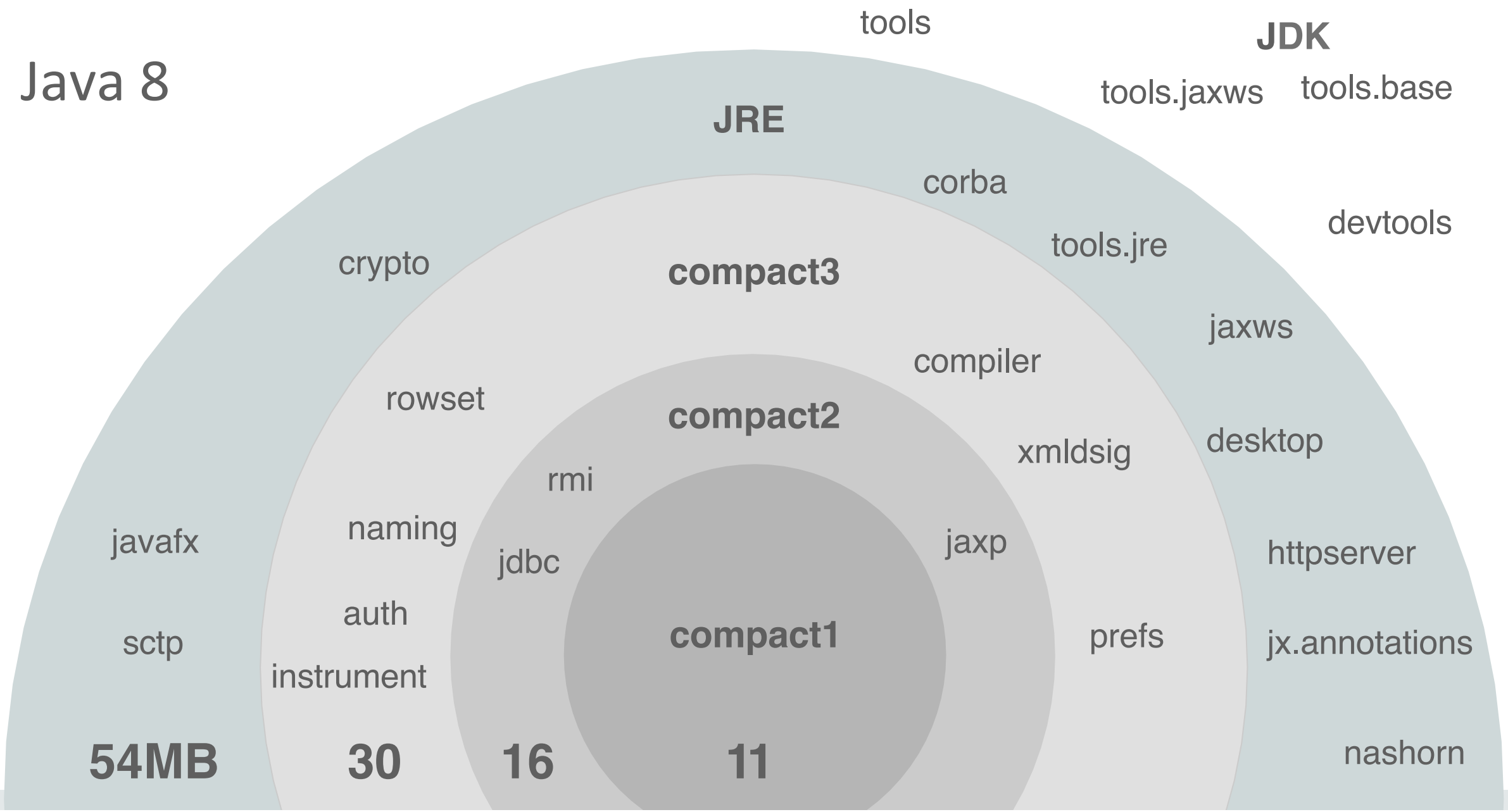
- Scalability
- Performance
- Security

Project Jigsaw

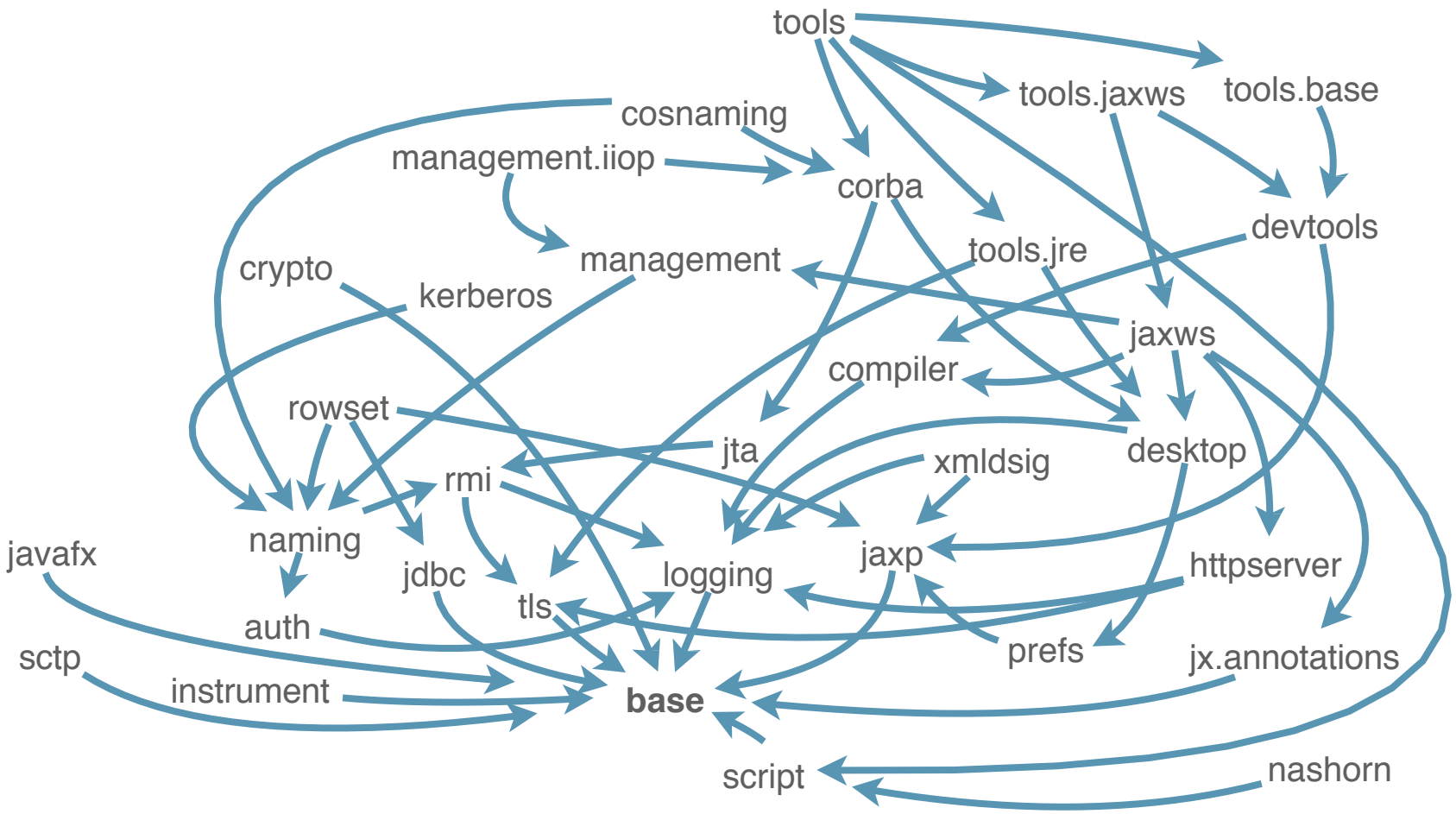
Scalability

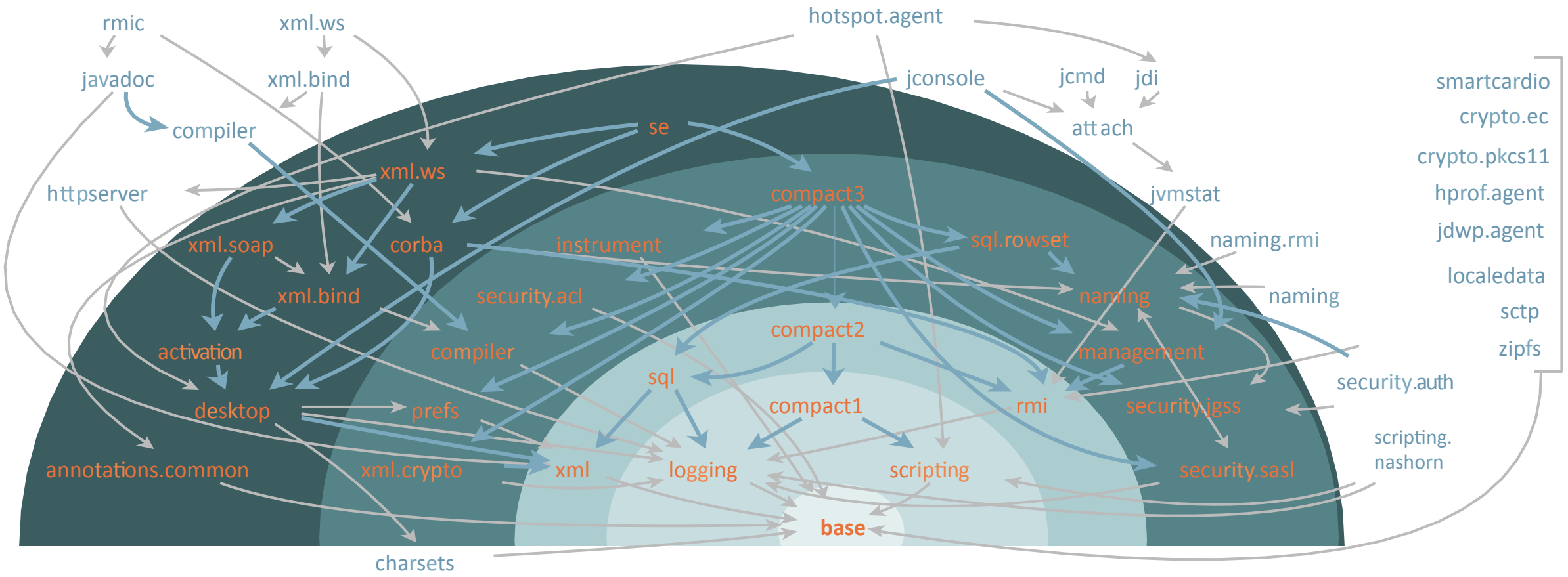
Project Jigsaw

Scalability

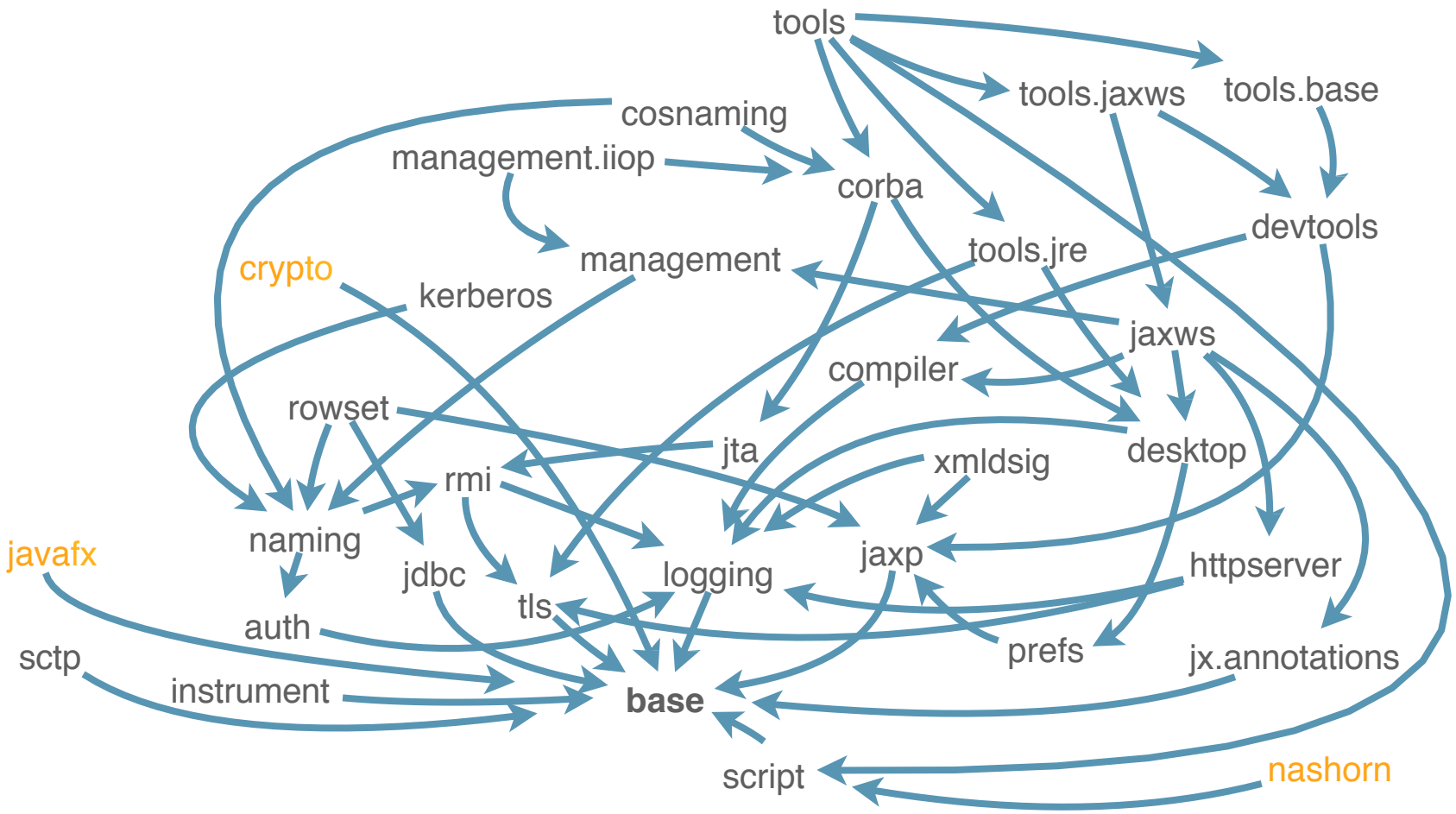

Java 8
tools
JDK
tools.jaxws
tools.base
JRE
corba
devtools
tools.jre
crypto
compact3
jaxws
compiler
rowset
compact2
desktop
xmldsig
rmi
naming
javafx
jaxp
httpserver
jdbc
auth
sctp
compact1
prefs
jx.annotations
instrument
54MB
30
16
11
nashorn

Project Jigsaw

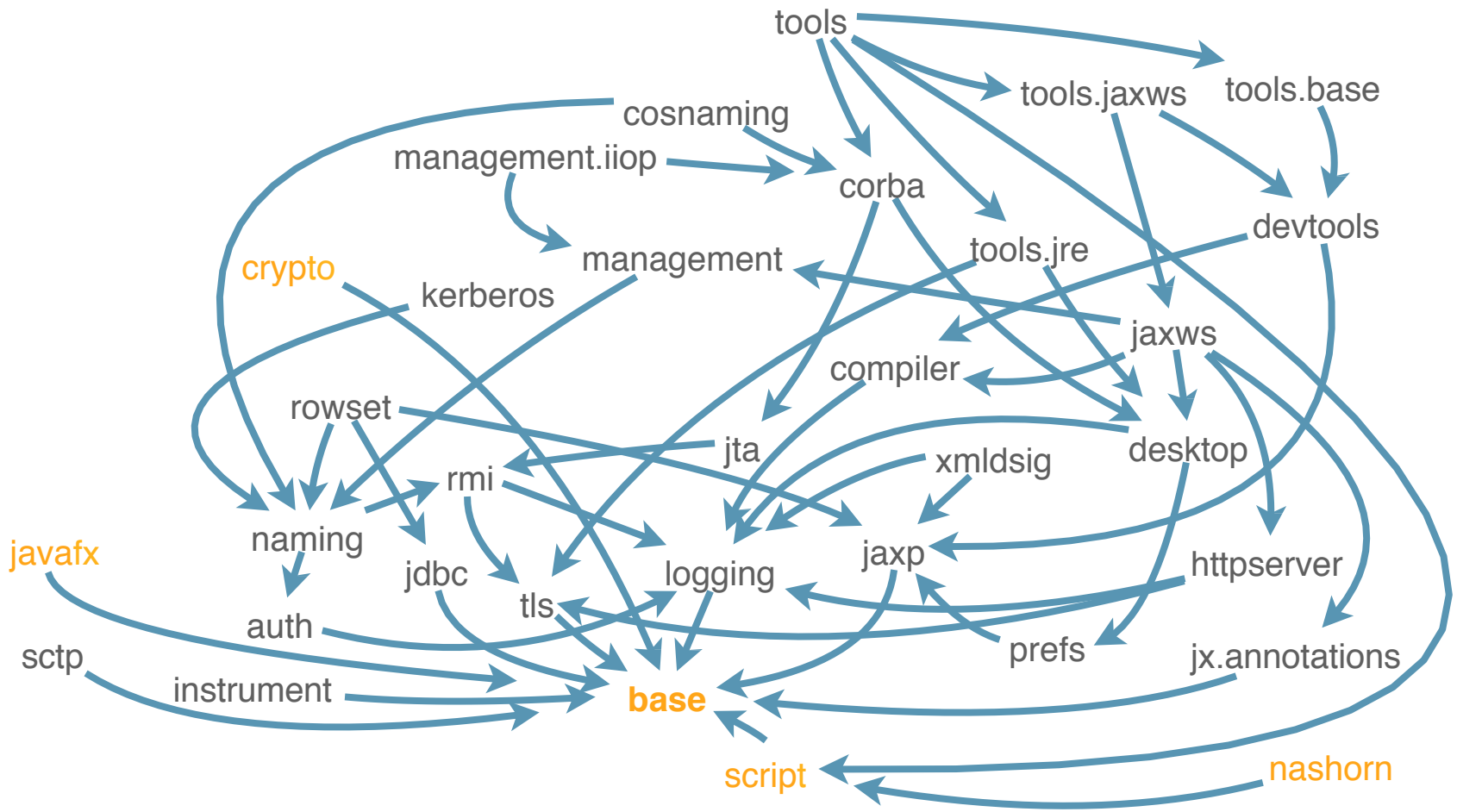




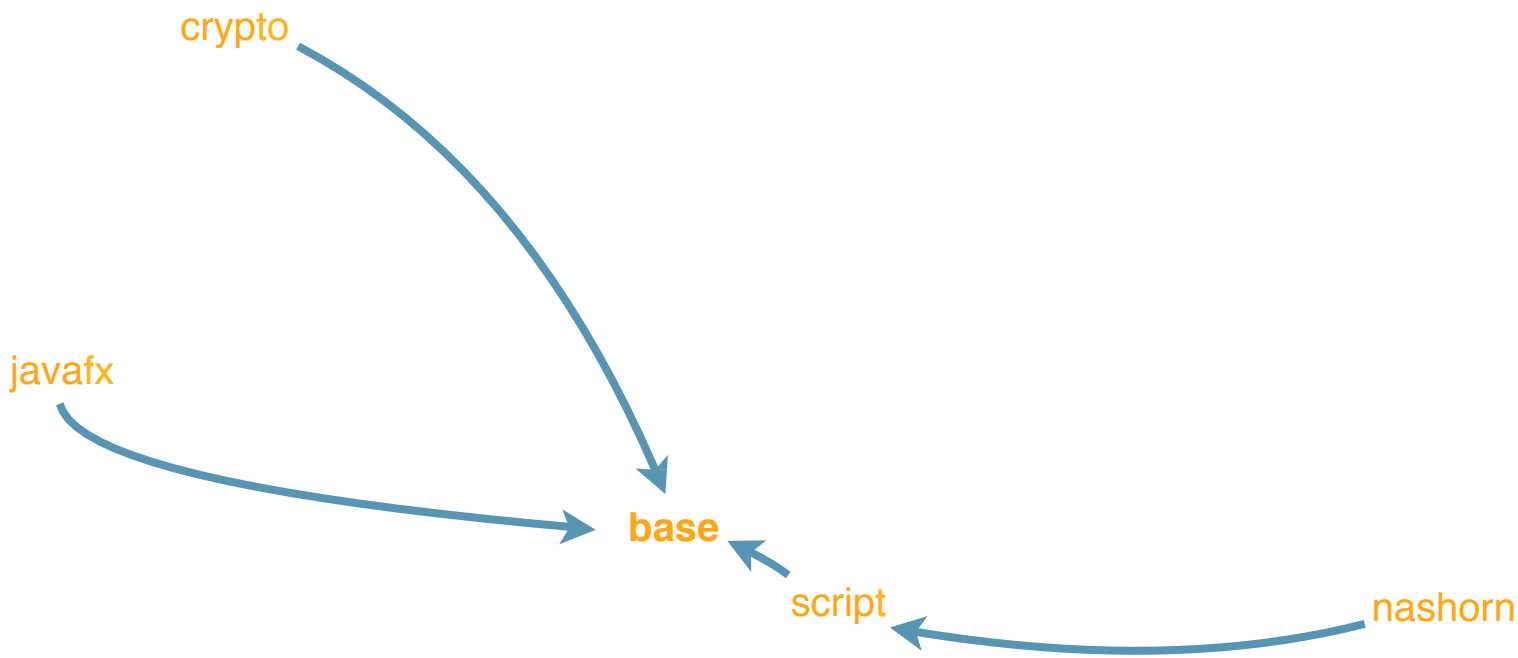
Project Jigsaw

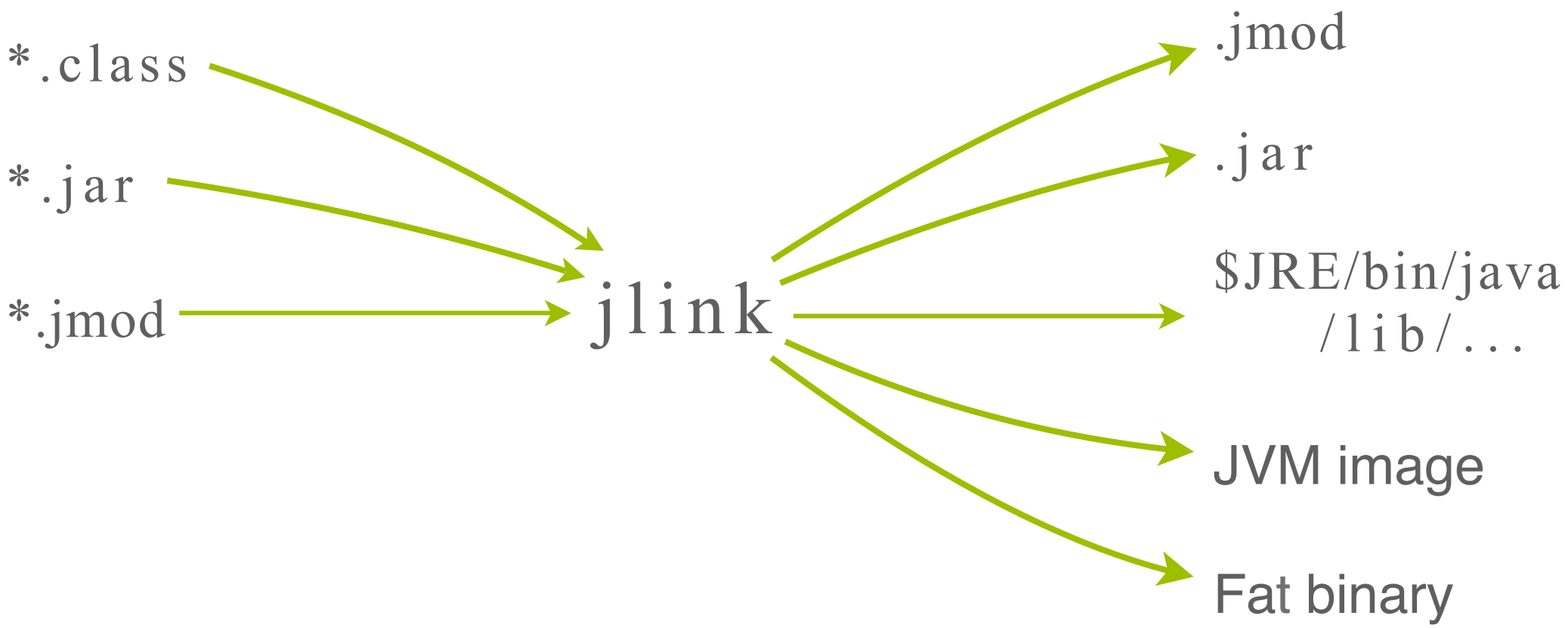


Project Jigsaw



Project Jigsaw

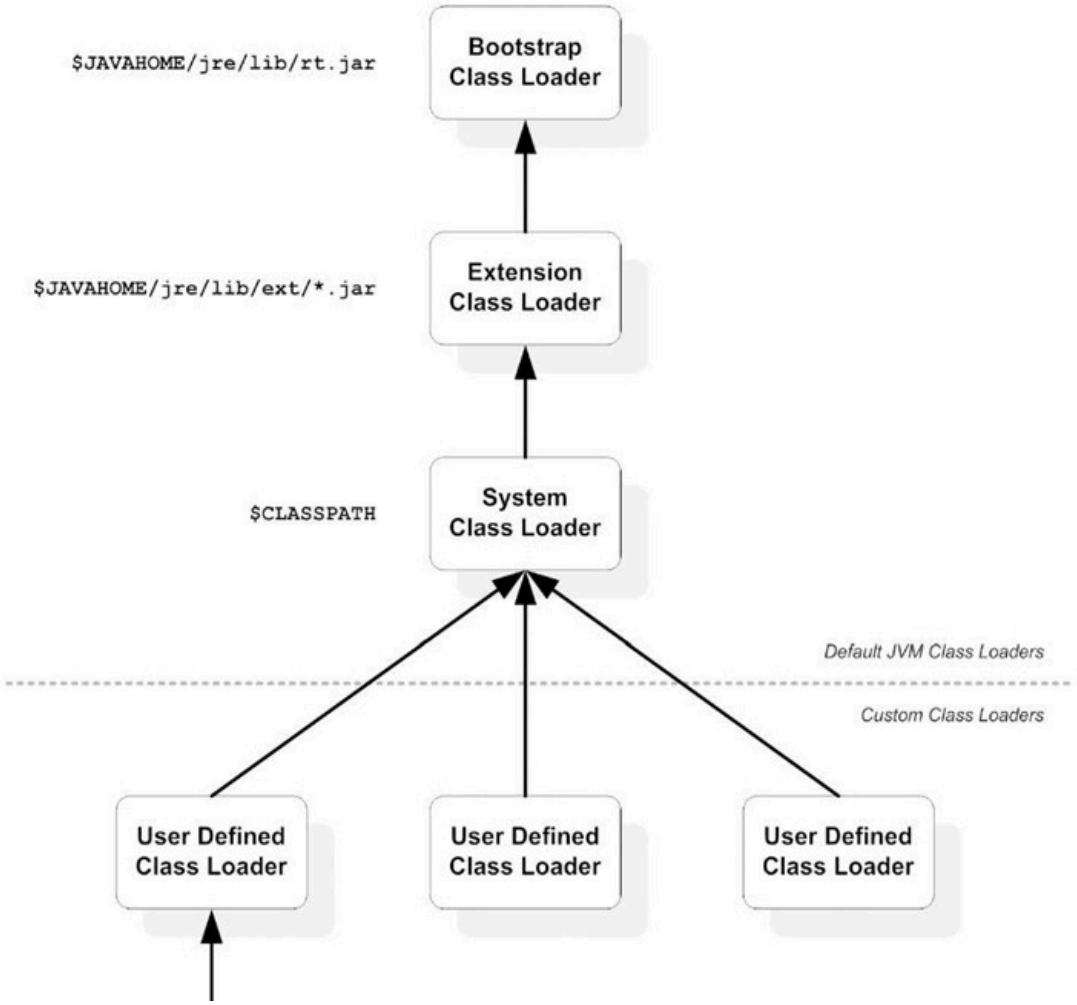




Project Jigsaw

Opportunities for performance improvements

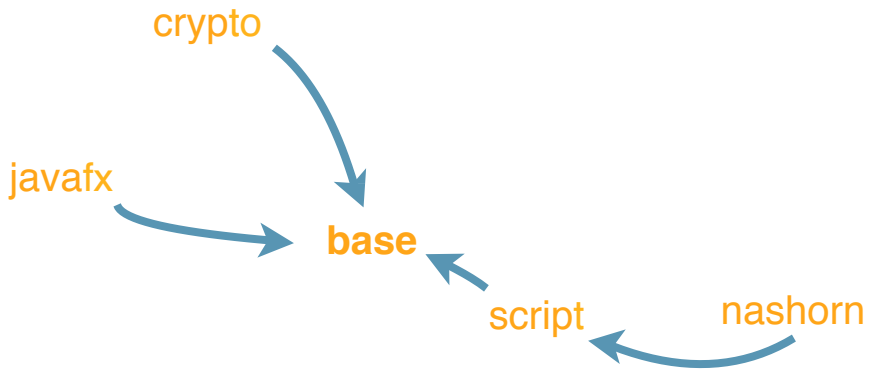
- Improved class loading architecture
  - fast class lookup



Project Jigsaw

Opportunities for performance improvements

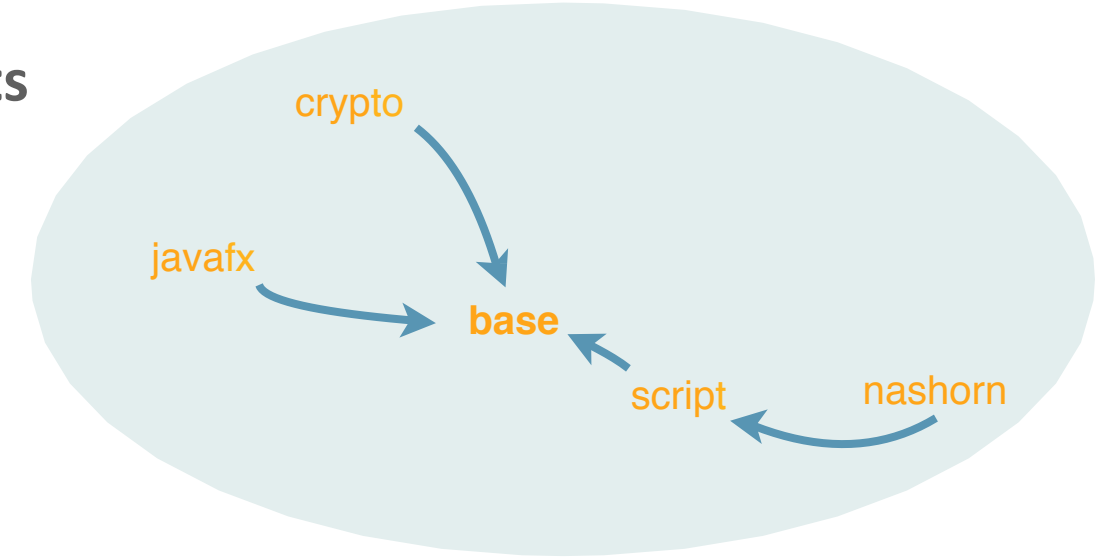
- Improved class loading architecture
  - fast class lookup



Project Jigsaw

Opportunities for performance improvements

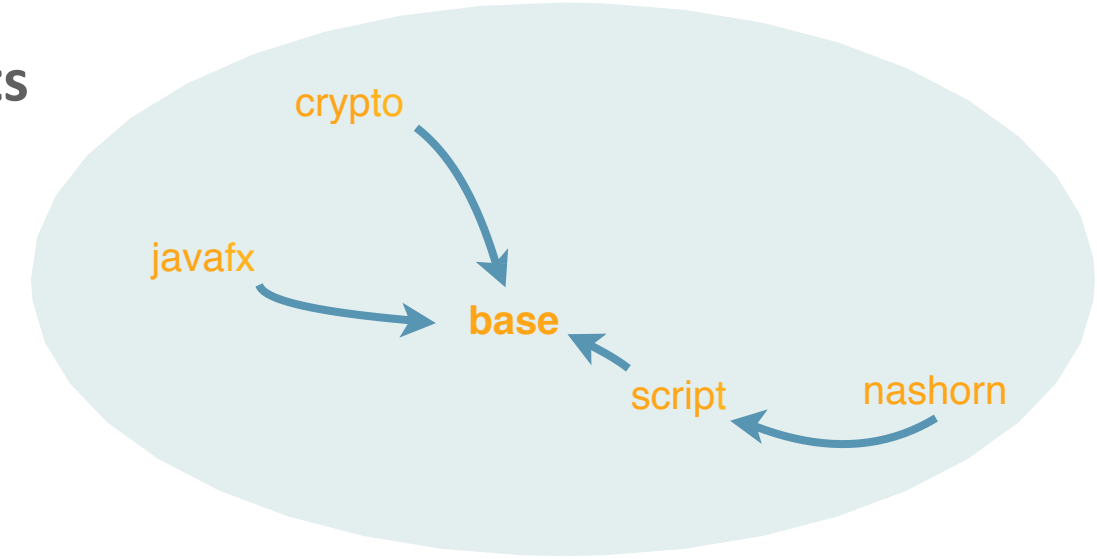
- Improved class loading architecture
  - fast class lookup



Project Jigsaw

Opportunities for performance improvements

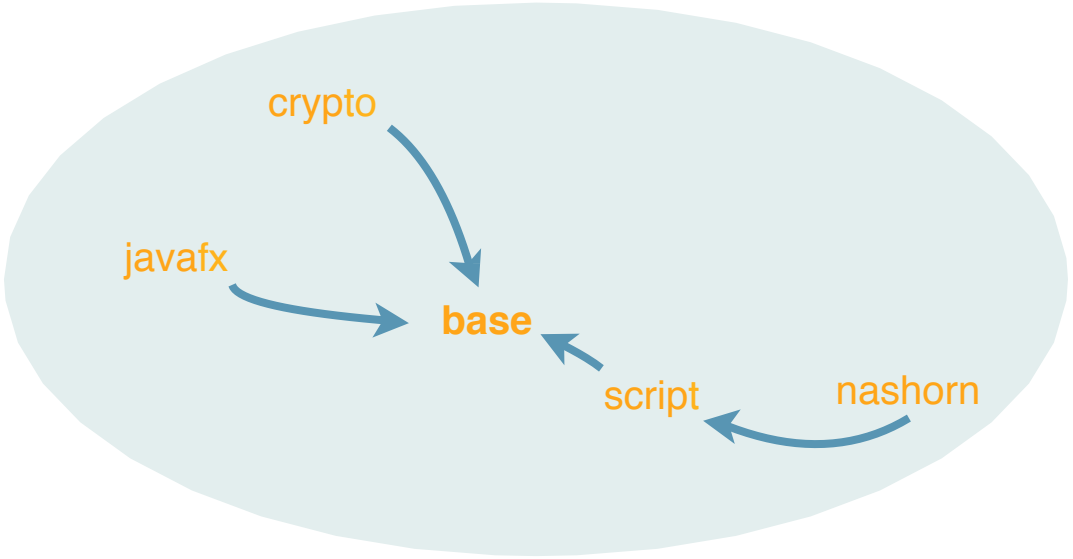
- Improved class loading architecture
  - fast class lookup



Project Jigsaw

Opportunities for performance improvements

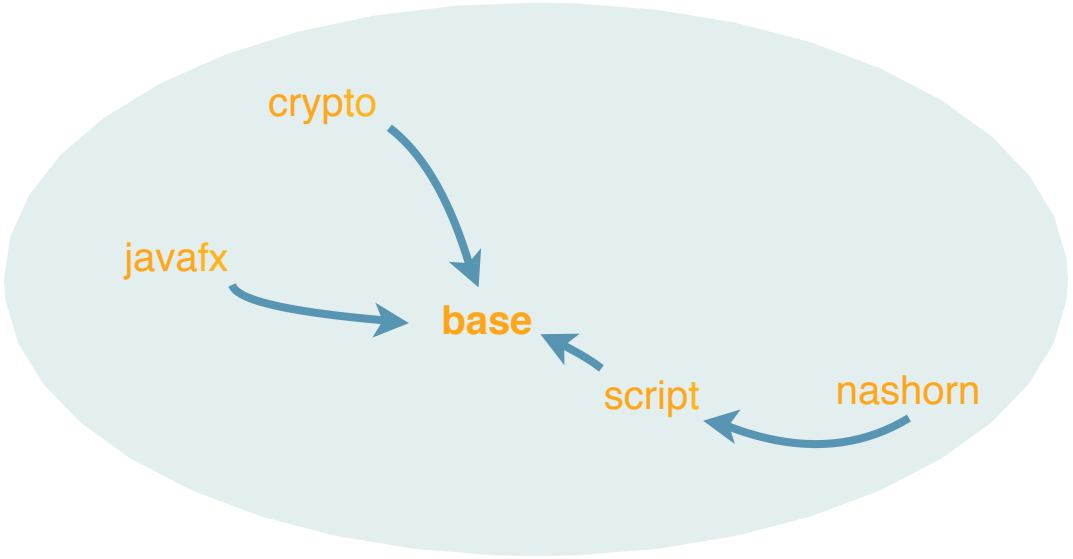
- Improved class loading architecture
  - fast class lookup
- Aggressive inlining



Project Jigsaw

Opportunities for performance improvements

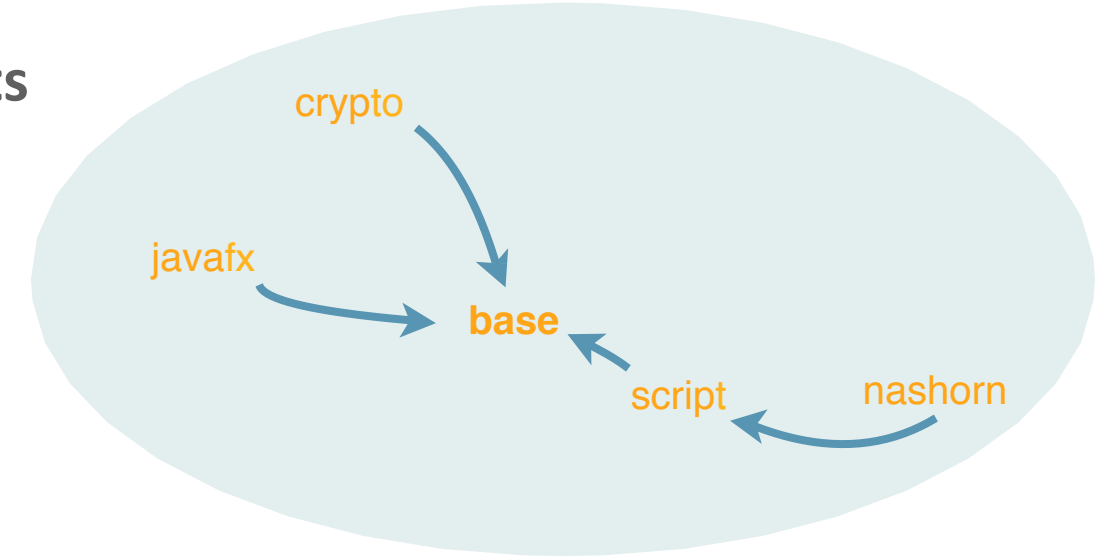
- Improved class loading architecture
  - fast class lookup
- Aggressive inlining
- Ahead-Of-Time compilation



Project Jigsaw

Opportunities for performance improvements

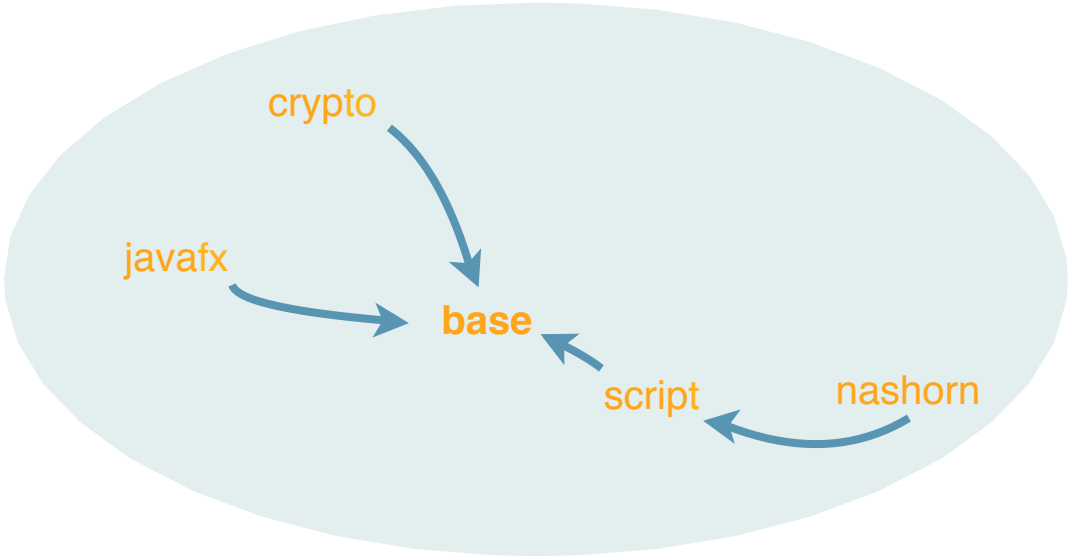
- Improved class loading architecture
  - fast class lookup
- Aggressive inlining
- Ahead-Of-Time compilation
- JVM-specific memory images
  - e.g. application Class Data Sharing (CDS)



Project Jigsaw

Opportunities for performance improvements

- Improved class loading architecture
  - fast class lookup
- Aggressive inlining
- Ahead-Of-Time compilation
- JVM-specific memory images
- Removal of unused fields/methods/classes



Project Jigsaw

Security

sun.*

.internal.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Project Jigsaw

Security

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Project Jigsaw

Non-goals

- not a replacement for Maven/Gradle
- multiple version management is out of scope
- interoperability with other package managers
  - Project Penrose for OSGi

Project Valhalla

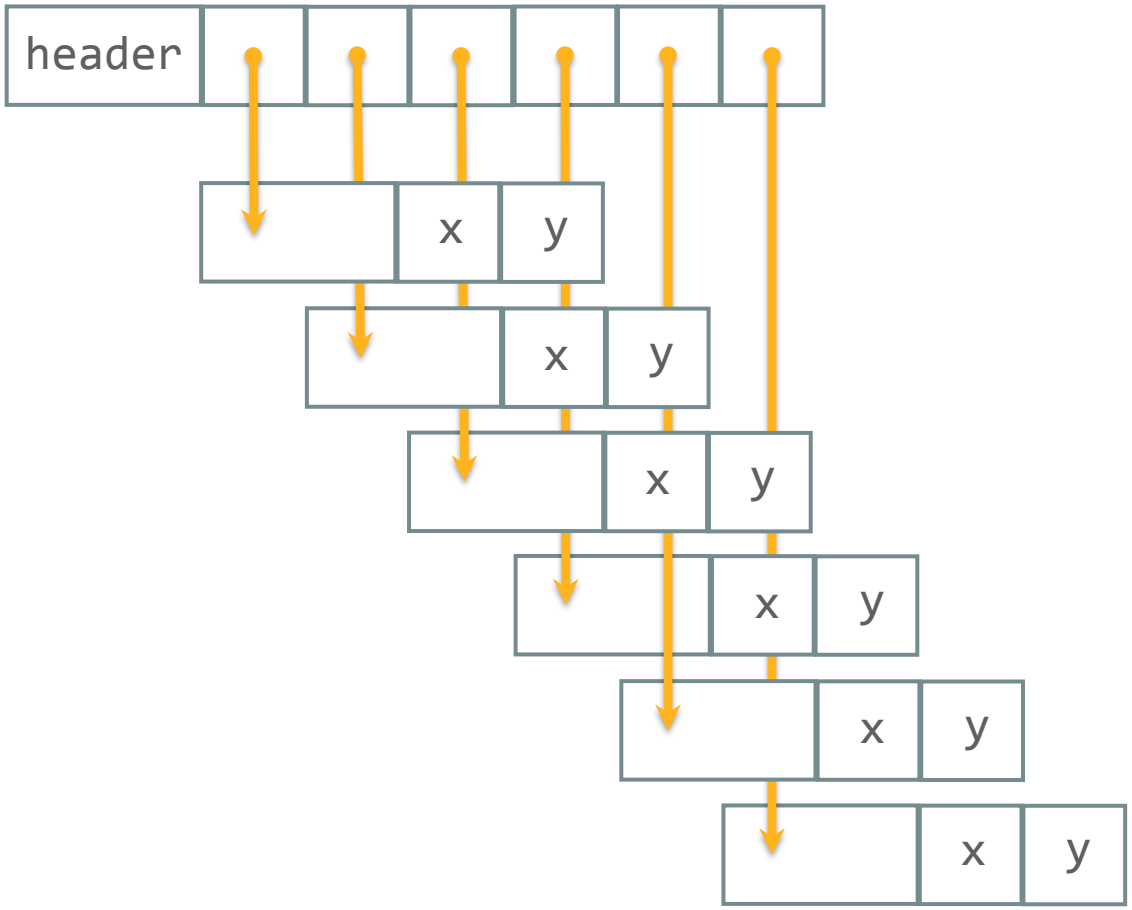
The hall of ~~valor~~ value

Value-based class

```
final class Point {
    public final int x;
    public final int y;
}
```

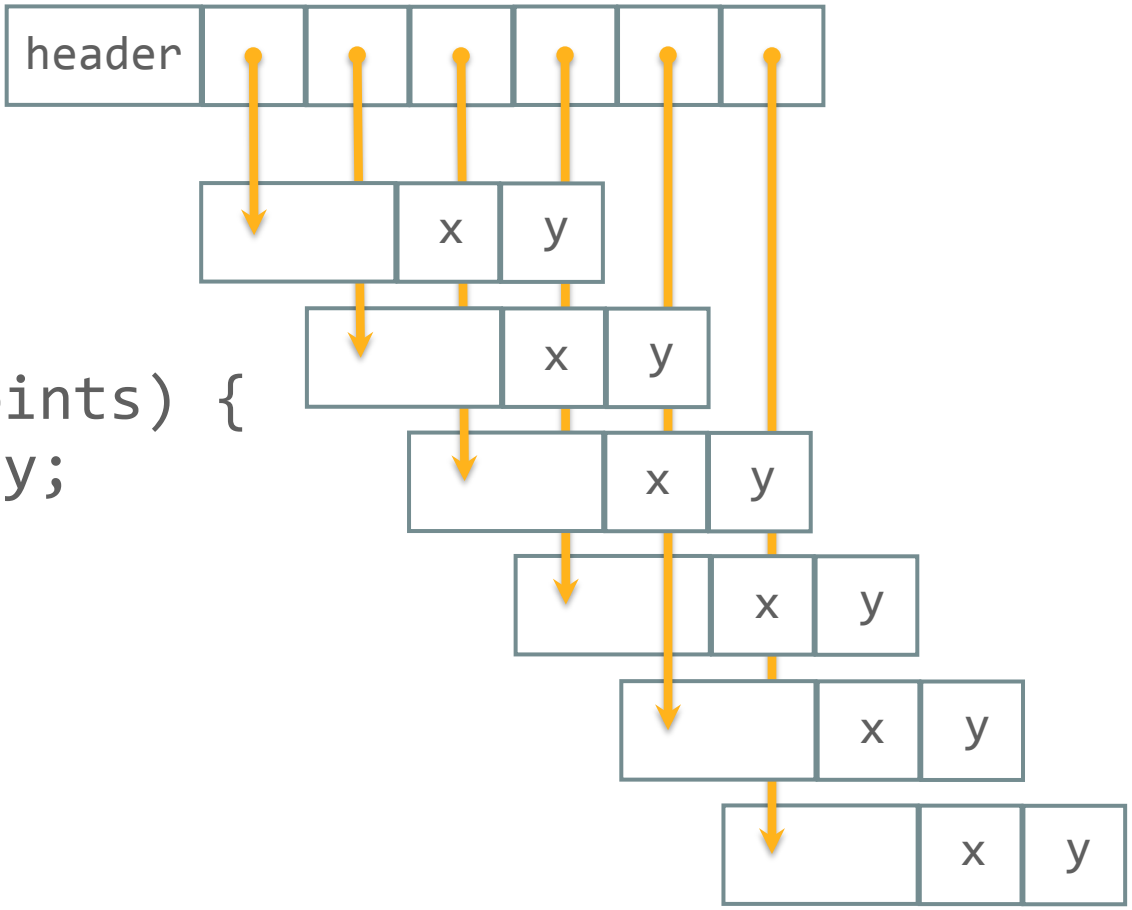
Value-based class

```
Point[] points =
```



Value-based class

```
Point[] points =
```

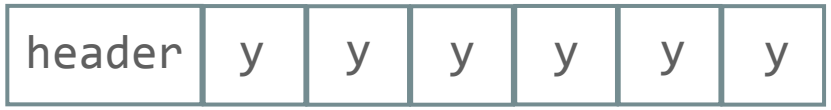



```
for (Point p : points) {
  sum += p.x + p.y;
}
```

Value-based class

```
int[] xPoints =
```

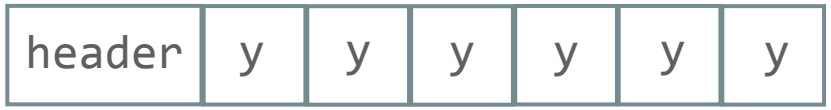

```
int[] yPoints =
```



Value-based class

```
int[] xPoints =
```

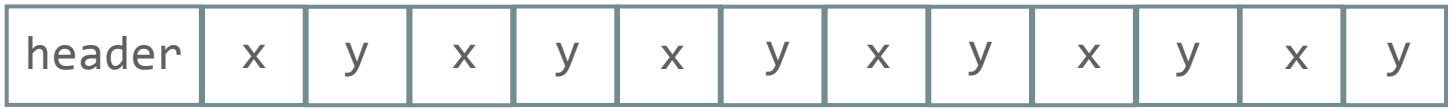

```
int[] yPoints =
```




```
assert(xPoints.length == yPoints.length);

for (int i = ; i < xPoints.length; i++) {
  sum += xPoint[i] + yPoint[i];
}
```

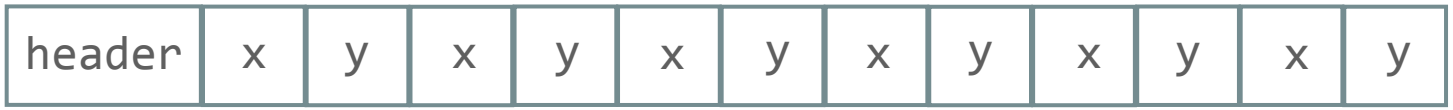
Value-based class

```
int[] points =
```



Value-based class

```
int[] points =
```

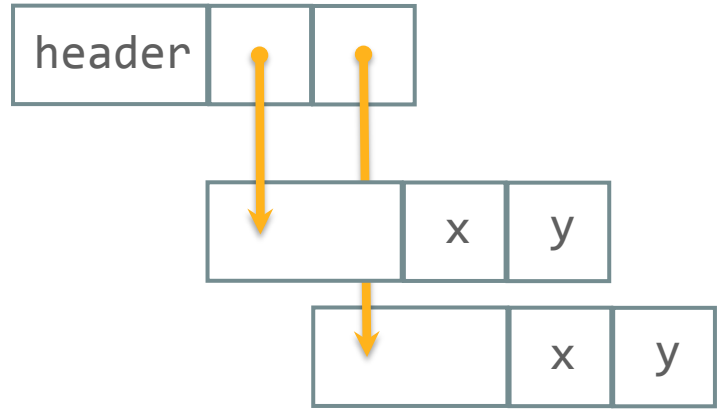



```
assert(points.length % 2 == 0);

for (int i = 0; i < points.length; i=i+2) {
   points[i] + points[i+1];
}
```

Value-based class

```
final class Point {
    public final int x;
    public final int y;
}

class Rectangle {
    public final Point corner1;
    public final Point corner2;
}
```

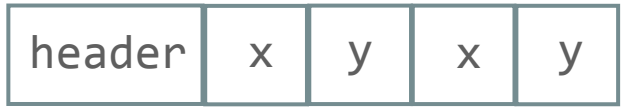
Rectangle r =



Value-based class

```
final class Point {
    public final int x;
    public final int y;
}

class Rectangle {
    public final Point corner1;
    public final Point corner2;
}
```

Rectangle r =



Make pointers optional!

"Use of identity-sensitive operations on instances of **value-based** classes may have unpredictable effects and should be avoided."

http://docs.oracle.com/javase/8/docs/api/java/lang/doc-files/ValueBased.html

"Codes like a **class**, works like an **int**!"

– John Rose, Brian Goetz, Guy Steele
"State of the Values"

Don’t make user choose between abstraction and performance!

Value class

```
value class Point {
    public final int x;
    public final int y;
}
```

Why value types?

Motivation

- Smaller footprint
  - no object header
- Better locality
  - no dereference
- Simpler semantics
  - no identity, no aliasing
- No allocation

Values in the JVM

- Primitives and references

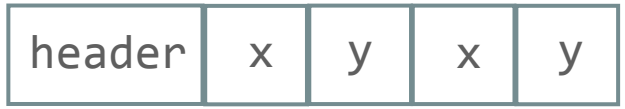
Values in the JVM

- Primitives and references
- Problems
  - Composite values
    - objects are expensive to construct
  - Control of concurrent side effects (JMM)

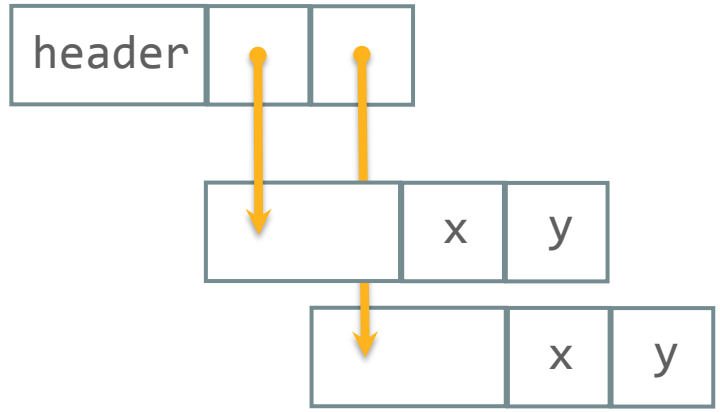
Value Types: concurrent side effects

```
value class Point {
    public final int x;
    public final int y;
}

class Rectangle {
    public final Point corner1;
    public final Point corner2;
}
```

Rectangle r =



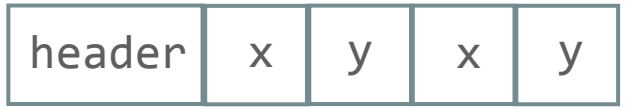
Rectangle r =



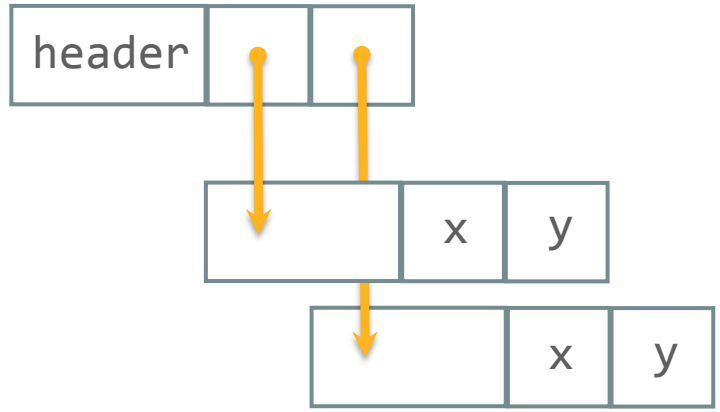
Value Types: concurrent side effects

```
value class Point {
    public final int x;
    public final int y;
}

class Rectangle {
    public          Point corner1;
    public volatile Point corner2;
}
```

Rectangle r =



Rectangle r =



Value types

- Value types are heterogeneous aggregates, like classes
  - Borrow many concepts from classes – methods, fields, etc
  - Declared like classes – with restrictions
- No inheritance
- No mutation
- No cyclic containment

Value types

The restrictions

- No locking
- No identity comparison (or, if forced, loose specification)
- No identity hash code
- No cloning
- No finalizer
- “null” is not a value
- No visible side effects
- No sub-class-ing (subtyping via extension)

Value types

The permissions

- Customized boxes are available (at nominal cost)
- Whole values can be assigned
  - Structure tearing is controlled: Nothing halfway between 'A' and 'B'
- Methods and fields can be defined (public/package/private)
- Via the boxes, all the comforts of objects:
  - Object.toString etc
  - Interfaces: Comparable, etc
  - Reflection

Example: methods on a value type

```
value class Point {
        public final int x;
        public final int y;

        public boolean equals(Point that) {
                return this.x == that.x && this.y == that.y;
        }

        private static String strValueOf(Point p) { … }

        public String toString() { return strValueOf(this); }
}
```

Example: coding with values

```
static final Point ORIGIN = __MakeValue(0, 0);

static Point displace(Point p, int dx, int dy) {
    if (dx == 0 && dy == 0)
        return p;
    Point p2 = __MakeValue(p.x + dx, p.y + dy);
    assert(!p.equals(p2));
    return p2;
}
```

Selected details

Point[] <:? Object[]

Selected details

Point[] <: Object[]

Value Types

JVM view

- New bytecodes (vaload, vnew, …)
- New types (interface,class,…value)
- New array layouts (array-of-values)
- Multi-word atomicity support (volatile vars)

Use cases

- Numerics: complex, decimal, rarely-big-num, etc.
- Native types: int128_t, vectors, unsigned, safe native pointers
- Algebraic data: optional (no box), choice-of, unit (no bits)
- Tuples: multiple-value return! (requires specialization machinery also)
- Cursors: unboxed iterators, STL-style bounds
- Flat data: values naturally represent pointer-poor data structures
- Caveat: values are not structs.

Generic Specialization

Motivation

ArrayList<Integer>

Generic Specialization

Motivation

ArrayList<int>

Objects vs Primitives

- Object, String, …, MyClass, …,

  Integer, Long, …

- 8 primitive types (+ reference)
  - boolean, byte, short, char, int, long, float, double

Objects vs Primitives

- Object, String, …, MyClass, …,

  Integer, Long, …

- 8 primitive types (+ reference)
  - boolean, byte, short, char, int, long, float, double
- Value types
  - User-defined

Generic Specialization

Motivation

ArrayList<Point>

Generic Specialization

```
class Box<T>  {
    T       val;

    public Box(T       val) { this.val = val; }
    public T       get()    { return val; }
}
```

Generic Specialization

```
class Box<Object> {
    Object val;

    public Box(Object val) { this.val = val; }
    public Object get()    { return val; }
}
```

Generic Specialization

```
class Box<int>    {
    int    val;

    public Box(int    val) { this.val = val; }
    public    int get()    { return val; }
}
```

Generic Specialization

```
class Box<any T>  {
    T       val;

    public Box(T       val) { this.val = val; }
    public T       get()    { return val; }
}
```

Generic Specialization

```
class Box extends Object {
    private final Object t;

    public Object get() {
      0: aload_0
      1: getfield #2 //Field t:LObject;
      4: areturn
    }
}
```

Generic Specialization

```
class Box extends Object {
    private final    int t;

    public    int get() {
      0: aload_0
      1: getfield #2 //Field t:I;
      4: ireturn
    }
}
```

Generic Specialization

```
class Box extends Object {
    private final Object*T t;

    public Object*T get() {
      0: aload_0
      1: getfield #2 //Field t:LObject;
      4: areturn*T
    }
}
```

Generic Specialization

Box<Integer> :> Box

Generic Specialization

Box<int> :>? Box

Generic Specialization

Box<int>  Box

Generic Specialization

```
Box<int> box = new Box<int>(1);
```

Generic Specialization

```
Box<int> box = new Box<int>(1);
```

```
Box${T=int}?
```

ClassDynamic

Invokedynamic for class generation

- The previous description sounds a lot like an indy callsite!
  - Bootstrap = specialization transform
  - Static args = class to specialize plus type substitutions
  - Together, these compose a structural description of a class
  - Type uses are compared structurally: the same if bootstrap and static args are the same
- Classdynamic = structural description of a dynamically generated class

ClassDynamic

Invokedynamic for class generation

- Strawman proposal: create a new constant pool type, dynamic class
  - Whose structure looks like a bootstrap + static args
  - Allow dynamic class wherever nominal type uses are allowed
    - (Actual classfile representation is TBD)
  - Expository notation: classdyn { bootstrap(args) }
  - So List<int> would be written as
    - classdyn { JavaSpecializer(List, int) }
- VM knows nothing about semantics of any given bootstrap
  - But there may be agreement between compiler and bootstrap

ClassDynamic

- Classdynamic can represent any mechanical class transform
  - Generic specialization (if the underlying class is suitably annotated)
  - Dynamic proxies
  - Synchronized wrappers
  - Forwarding proxies
  - Unreflectors
  - Tuples (*)
  - Function types (*)
- Moves code generation from compile time to runtime

Specialization Challenges

Generic Methods

```
<T> T identity(T t) { return t; }
```

Specialization Challenges

Generic Methods

- Same challenges as class specialization, and then some
  - Adding new methods to existing classes is painful
  - We could statically generate specializations for all primitives
    - But this would fall apart for value types
  - So need a mechanism for hooking into nominal method linkage

Specialization Challenges

When generic code is too generic

- Suppose I write a generic class ArrayList<any T>
  - Can I provide a hand-written replacement for ArrayList<boolean> ?

Specialization Challenges

When generic code is too generic

- Suppose I write a generic class ArrayList<any T>
  - Can I provide a hand-written replacement for ArrayList<boolean> ?
  - … hand-written replacement for a single method
    - ArrayList<int>.contains()

Specialization Challenges

When generic code is too generic

- Suppose I write a generic class ArrayList<any T>
  - Can I provide a hand-written replacement for ArrayList<boolean> ?
  - … hand-written replacement for a single method
    - ArrayList<int>.contains()
  - a specific instantiation of a generic method

Specialization Challenges

When generic code is too generic

- Suppose I write a generic class ArrayList<any T>
  - Can I provide a hand-written replacement for ArrayList<boolean> ?
  - … hand-written replacement for a single method
    - ArrayList<int>.contains()
  - a specific instantiation of a generic method
  - add new member for some specialization
    - e.g. ArrayList<int>.sum()

VarHandles

Current situation

- Support for atomicity and fencing is limited
  - Accesses to volatile fields automatically fenced, others not
  - Fenced operations and atomic operations (CAS) available through Unsafe

VarHandles

Current situation

- Support for atomicity and fencing is limited
  - Accesses to volatile fields automatically fenced, others not
  - Fenced operations and atomic operations (CAS) available through Unsafe
- sun.misc.Unsafe methods
  - Fenced loads/stores
  - Atomic updates (CAS)

VarHandles

Current situation

- Support for atomicity and fencing is limited
  - Accesses to volatile fields automatically fenced, others not
  - Fenced operations and atomic operations (CAS) available through Unsafe
- sun.misc.Unsafe methods
  - Fenced loads/stores
  - Atomic updates (CAS)
- j.u.c.atomic.Atomic* classes

VarHandles

The bad and the ugly

- Atomic* classes have overhead
  - Not used in j.u.concurrent classes
- sun.misc.Unsafe is... unsafe, not “portable”, and going away
  - CAS is too important to relegate
- No unified/safe model for accessing on and off-heap

VarHandles

JEP 193: Enhanced Volatiles

- Safe, performant, enhanced atomic
  - access to field and array elements
- Replace nearly all usages of Unsafe in java.util.concurrent classes

VarHandles

Method handles for data

- VarHandle is like method handles for data
  - Abstracts over location – static fields, instance fields, arrays, off heap
  - Supports explicit fenced and atomic operations
- Safer than Unsafe, as fast as MethodHandle

VarHandles: language support?

```
class Usage {
   volatile int count;

   int incrementCount() {
       return count.volatile.incrementAndGet();
   }
}
```

VarHandles API

```
VarHandle extends MethodHandle:

  MethodHandle COUNT_getVolatile = MethodHandle.lookup()…;
  COUNT_getVolatile.invokeExact(…);

  MethodHandle COUNT_compareAndSet.invokeExact();
  COUNT_getVolatile.invokeExact(…);
```

VarHandles API

```
public abstract class VarHandle extends … {
…
 public final native

 @PolymorphicSignature

 Object getVolatile(Object... args);
…
}
```

Project Panama

Bridging the gap

"If non-Java programmers find some library useful and easy to access, it should be similarly accessible to Java programmers."

– John Rose, JVM Architect, Oracle Corporation

JNI

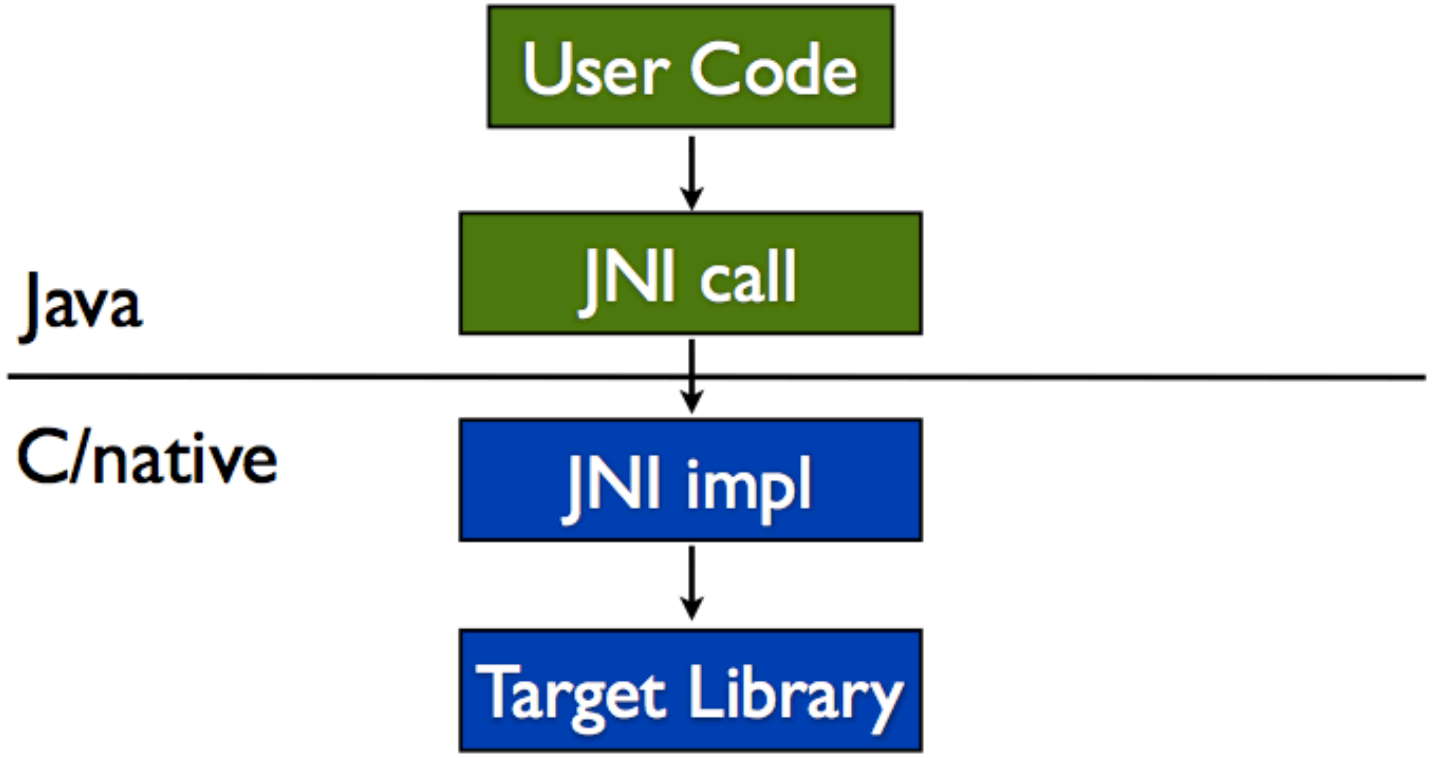
@since 1.0

JNI

A victim of its own success?

JNI

Usage scenario



JNI

Java:

```
public class GetPid {
    public static native long getpid();
}
```

JNI

Java:

```
public class GetPid {
    public static native long getpid();
}
```

C/C++:

get_pid.h

```
JNIEXPORT jlong JNICALL Java_GetPid_getpid (JNIEnv *, jclass);
```

get_pid.c

```
jlong JNICALL Java_GetPidJNI_getpid(JNIEnv *env, jclass c) {
 return getpid();
}
```

JNI: Method Invocation

Before call

1. if (GC_locker::needs_gc())

   SharedRuntime::block_for_jni_critical();

2. transition to thread_in_native

3. unpack array arguments

4. call native entry point

JNI: Invocation

After call

1. call native entry point
2. check for safepoint in progress
3. check if any thread suspend flags are set
   - call into JVM and possibly unlock the JNI critical if a GC was suppressed while in the critical native
4. transition back to thread_in_Java
5. return

sun.misc.Unsafe

Anti-JNI

Unsafe.getUnsafe().putInt(new Object(), 0, 0)

Unsafe.getUnsafe().putInt(null, 0, 0)

How many of you have used the Unsafe API?

…

– John Rose, JVM Architect, Oracle Corporation

@ JVM Language Summit 2014

How many of you have used the Unsafe API?

…

A lot of you. Gosh. I'm sorry.

– John Rose, JVM Architect, Oracle Corporation

@ JVM Language Summit 2014

Better JNI

Easier, safer, faster!

Better JNI

```
pid_t getpid();
```

Better JNI: Easier

```
public interface GetPid {
    long getpid();
}
```

Better JNI: Easier

```
public interface GetPid {
    long getpid();
}

LibraryLoader<GetPid> loader = FFIProvider
                .getSystemProvider()
                .createLibraryLoader(GetPid.class);
```

Better JNI: Easier

```
public interface GetPid {
    long getpid();
}

LibraryLoader<GetPid> loader = FFIProvider
                .getSystemProvider()
                .createLibraryLoader(GetPid.class);

GetPid getpid = loader.load("c");

getpid.getpid();
```

Better JNI: Faster

```
callq <getpid_address>
```

Optimize checks

```
void run(MyClass obj) {
    obj.f1(); // NPE
    obj.f2(); // NPE
    obj.f3(); // NPE
}
```

Optimize checks

```
void run(MyClass obj) {
    if (obj == null) jump throwNPE_stub;
    call MyClass::f(obj);
    call MyClass::f1(obj);
    call MyClass::f3(obj);
}
```

Optimize checks

```
void run(MyClass obj) {
    obj.nativeFunc1(); // checks & state trans.
    obj.nativeFunc2(); // checks & state trans.
    obj.nativeFunc3(); // checks & state trans.
}
```

Optimize checks

```
void run(MyClass obj) {
    if (!performChecks())  jump failed_stub;
    call transJavaToNative();
    MyClass::nativeFunc1(env, obj);
    MyClass::nativeFunc2(env, obj);
    MyClass::nativeFunc3(env, obj);
    call transNativeToJava();
}
```

Better JNI: Safety

- no crashes
- no leaks
- no hangs
- no privilege escalation
- rare outages
- no unguarded casts

Better JNI: Trust levels

Untrusted

Better JNI: Trust levels

Trusted

Better JNI: Trust levels

Privileged

Better JNI: Easier, Safer, Faster!

- Native access between the JVM and native APIs
  - Native code via FFIs (JNR is starting point)
  - Native data via safely-wrapped access functions
  - Tooling for header file API extraction and API metadata storage

Better JNI: Easier, Safer, Faster!

- Native access between the JVM and native APIs
  - Native code via FFIs (JNR is starting point)
  - Native data via safely-wrapped access functions
  - Tooling for header file API extraction and API metadata storage
- Wrapper interposition mechanisms, based on JVM interfaces
  - add (or delete) wrappers for specialized safety invariants

Better JNI: Easier, Safer, Faster!

- Native access between the JVM and native APIs
  - Native code via FFIs (JNR is starting point)
  - Native data via safely-wrapped access functions
  - Tooling for header file API extraction and API metadata storage
- Wrapper interposition mechanisms, based on JVM interfaces
  - add (or delete) wrappers for specialized safety invariants
- Basic bindings for selected native APIs

Foreign layouts

- Native data requires special address arithmetic
  - Native layouts **should not** be built into the JVM (sorry, no native classes)
  - Native types are unsafe (hello, C!), so trusted code must manage the bits
- Solution: A metadata-driven Layout API
- As a bonus, layouts other than C and Java are naturally supported
  - Network protocols, specialized in-memory data stores, mapped files, etc.

Project Jigsaw

Scalability
Performance
Security

http://openjdk.java.net

Project Valhalla

Specialized Generics
Value Types
Var Handles

Project Panama

Foreign Function Interface
Data Layout Control
Arrays 2.0

http://openjdk.java.net

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.

Thank you!


Java™
ORACLE®