Project Valhalla Update

Brian Goetz, Java Language Architect

JVM Language Summit, Santa Clara, August 2016

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.

Project Valhalla

- Project Valhalla starts with a simple-seeming feature: *value types*
  - Pure data aggregates that (ideally) should have no ancillary overhead
  - This is the same reason Java has primitive types in the first place!
- But, features interact with other features
  - Adding one feature means adjusting many others
  - It's a long string…
- This will be a whirlwind tour of some of the areas we're investigating
  - The map is not fully drawn
  - Some parts are better filled in than others!

Project Valhalla

Why value types?

- The motivation for value types is simple

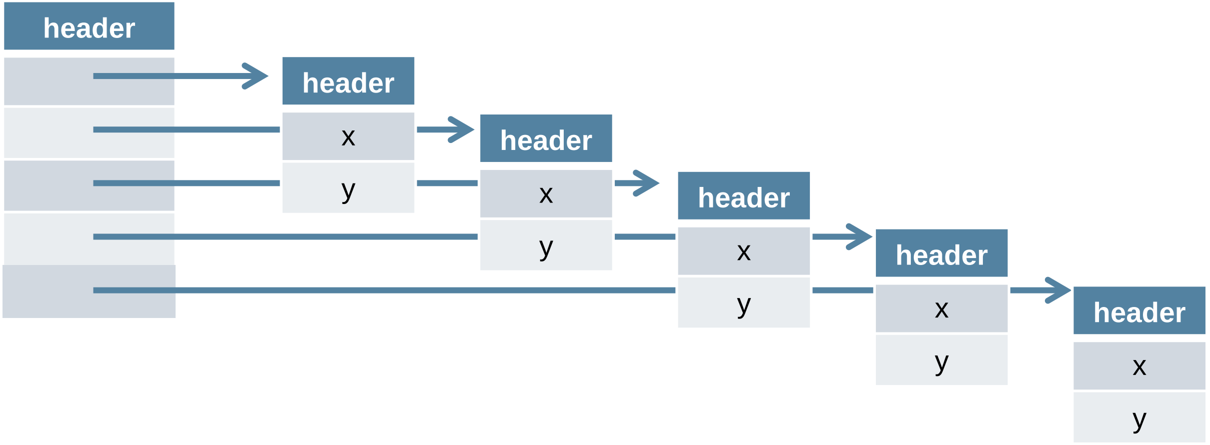
- Identity leads to pointers
- Pointers lead to indirection
- Indirection leads to suffering

Project Valhalla

The data layout we have

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

```
Point[] pts =
```



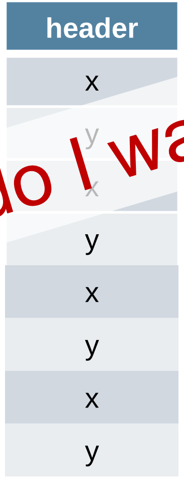
ORACLE

Project Valhalla

The data layout we want

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

Point[] pts =
```



What code do I want to write to get this layout?

ORACLE

Valhalla: Goals

Performance goals

- Density and Flatness!
  - Get rid of extraneous headers and pointers (and heap allocation) when they don't add value
- *Stop making users choose between performance and abstraction*
  - Eliminate temptations to hand-unroll object abstractions into primitives
  - Eliminate need to hand-roll primitive specializations (like IntStream)
  - Generics should be the tool of choice for abstracting over types
- Value types, and specialized generics over values, eliminate these frictions

Valhalla: Goals

Expressiveness goals

- Caulk the seam between primitives and references
- Let generics abstract over references, primitives, values (and void!)
  - Write it once, not N+1 times
- Java 8 libraries illustrated the limitations here
  - Hand-written specializations like IntStream
  - Explosion of functional interfaces (Consumer<T>, ToIntFunction<T>, etc)
  - Streams of tuples are painful, inefficient
- Plus: *existing libraries* need help taking take advantage of new features
  - Just as Collections acquired lambda-friendly behavior via default methods

Value Types

Our starting point

- Value types are “pure data” aggregates
  - Just data, no identity
  - No representational polymorphism (no superclasses or subclasses)
  - Not mutable
  - Not nullable*
  - Equality comparison based on state (since there is no identity)
- By giving up on identity, mutability, polymorphism, we get…
  - Values routinely flattened into arrays, other values, objects
  - No object header needed
  - Aggregates (with behavior) that have runtime behavior of primitives

*Some possible relaxation may be needed here for migration compatibility

Value Types

- But, unlike primitives
  - Can have methods, fields
  - Can implement interfaces
  - Can use encapsulation to hide representation
  - Can be generic
- General rubric for answering “how would it work” questions
  - “What Would Int Do”

Codes like a class, works like an int

ORACLE

Value Types

Who wants value types?

- Application writers
  – Can reason about locality and footprint of data-intensive code
- Library writers
  – Efficient and expressive implementations of smart pointers, alternate numerics, cursors, abstract data types
  – More efficient collections
- Compiler writers
  – Efficient substrate for language features like tuples, multiple return, built-in numeric types, wrapped native resources
- Everyone wants value types!

ORACLE

Generics

- We can easily *express* everything we want with boxed generics
  - ArrayList<Integer> expresses what the user needs
- But each Integer in that list has the same problems as our Point class
  - Object header, indirection, allocation, GC overhead
- For all the same reasons we wanted values, we want ArrayList<int> instead of ArrayList<Integer>
  - Where the List is backed by a real int[]
  - (And same with generics over user-defined value types)
- Having value types, but no generics over values, would be terrible!
  - This is the string…

Value Types: Alternatives

Why not “just” do structs?

- The question suggests that structs are simpler than values
- But they aren’t simpler, they’re just more familiar!
  - A struct needs an identity
    - Sometimes the identity of the enclosing object
    - Sometimes an ad-hoc identity, if the struct is held in a local
  - Structs need both pass-by-value and pass-by-reference
    - Java only has pass-by-value – so this is new complexity surface
- Less optimizable
- Structs is really *more* work, and complexity, than values
  - Not “A OR B”, it’s “A OR (A AND B)”
- Conclusion: more cost, less safety

ORACLE

Value Types: Alternatives

Why not “just” do tuples?

- We could easily denote “tuple of int and long” with the descriptor “(IJ)”
  - And provide opcodes for pushing, popping, and decomposing tuples
  - Semantics are very straightforward
  - Verification is easy
- But … almost as much new classfile surface area as values
- Would work for some value use cases
- What we’d lose is: encapsulation and nominality
  - Not suitable for secure representations (e.g., native pointers)
  - Point and IntRange would be the same type
- Conclusion: slightly less cost, measurably less benefit

Value Types

- What does it mean for the JVM to support value types?
  - How do we construct values?
  - How do we transfer values between stack and local variables?
  - How do we access fields of values?
  - How do we invoke methods of values?
  - How do we embed values as fields of objects?
  - How are values denoted in member descriptors?
  - How many stack slots does a value take?
  - How do we convert values to objects?
- Need new bytecodes, new type descriptors
  - Unlike refs or primitives, value types have **variable size**

Stack slots

- If a Point has two longs, how many slots should it take?
- Obvious (but wrong) answer is “4”
  - Would burn representation into client’s bytecode
  - Adding/removing fields would not be binary compatible
  - Might undermine encapsulation
- Reasonable answers include “1” and “2” (and other fixed numbers)
  - But both mean extra work for interpreter
- As we’ll see later, would like to get to all `xload` ops taking 1 slot – even dload/lload

Value Boxes

- Need a way to convert values to/from Object (and interfaces)
- Don't want ad-hoc, hand-written boxes like java.lang.Integer
  - Want to derive boxed projection from value classfile
- Do we need a way to separately denote boxed and unboxed values?
- Working theory:
  - **`LPoint;`** describes the boxed projection
  - **`QPoint;`** describes the unboxed projection

Value Bytecodes

- Bytecode design has many constituents
  - Verifier – can we guarantee type safety (especially pointer safety)?
  - GC – can we find all the pointers?
  - Compiler writers / code generators
  - Tools – can we easily extract and optimize data and control flow?
- Various bytecode schemes possible
  - Tradeoff between number of new bytecodes, footprint, and complexity
  - Some cases (e.g., getfield) could be retrofitted onto existing bytecodes
  - Could model most v* bytecodes with a prefix (**`typed QFoo; aload`**)
- We'll err on the side of simplicity now, and optimize later

Value Bytecodes

```
Point point = __make Point(3, 4);
int x = point.x;
int y = point.y;
```

```
 0: iconst_3
 1: iconst_4
 2: vnew          #19 // <vinit>:(II)QPoint;
 5: vstore        1 QPoint;
 7: vload         1 QPoint;
 9: vgetfield     #12 // Field x:I
12: istore_2
13: vload         1
15: vgetfield     #15 // Field y:I
18: istore_3
```

How many slots?

 |

Object Model

Lots of new questions

- Should Object be the top type?
  - Seems like “has identity” should be reflected in the type system
- Should there be a new “Any” type?
  - What would its in-memory representation be?
- Should there be a top type for values?
- Where should common methods (equals, hashCode) be defined?
- Should primitives become more like values?
  - Have methods, implement interfaces?

ORACLE

Generics

- Generics embed an uneasy compromise: cannot generify over primitives
- Why?
  - No common top type between Object and int
  - No bytecode that can move both a ref and an int
- Assuming away primitives solved a lot of problems
  - But leaves us with lousy performance with boxed primitives
  - More allocation, less locality
- Library writers compensate with tricks like IntStream
  - Usually by cut and paste duplication
  - More footprint, more bugs
  - Less abstraction!

Generics

Erasure

- Erasure gets a bad rap, but was a *pragmatic compromise*
  - Enabled Java to acquire generics with no VM changes
  - Significant additional type safety, ZERO additional runtime cost
  - No additional runtime code footprint
- Permitted *gradual migration compatibility*
  - Libraries could be generified independently from clients
  - Clients could generify immediately, later, or never
  - No “flag days”
- Libraries are often in different maintenance domains than their clients (e.g., java.util.ArrayList)
  - Dynamic linkage is the norm, not the exception

Specialized Generics

- What does the bytecode look like for this class?
  - What are the method and field descriptors?
  - What bytecode pushes a T? aload? Something else?

```
class Box<any T> {
    T t;                                         // LBox;

    T get() { return t; }                        // ()LObject;
    void reset() { this.t = T.default; }         // ()V
    void set(T t) { this.t = t; }                // (LObject;)V
    Box<T> dup() { return new Box<T>(t); }       // ()LBox;
}
```


Specialized Generics

Model 1

- Our first attempt ("Model 1") annotated the classfile with type-variable metadata and specialized at runtime
- An aload bytecode that moved a \`T\` would get annotated as such
  - Specializer would rewrite appropriately to iload, dload, etc
- Amazingly, this worked!
  - All done with compiler and classloader trickery – no VM involvement
  - But … messy, intrusive and complex
- No commonality between List<int> and List<long>
  - No wildcards!

Specialized Generics

The prime directive: compatibility

- Just as with the first time around, we need gradual migration compatibility for enhanced generics
  - Anyfying an existing type variable must be source- and binary-compatible (for clients and subclasses)
  - Generifying an enclosing scope must be source/binary compatible
  - Alpha-renaming a type variable must be source/binary compatible
  - Adding a new type var at the end should be binary compatible
- Hierarchies anyfied from the top down
  - Clients/subclasses should have choice of anyfying immediately, later, or never
  - No flag days!

Specialized Generics

Bytecode set is hostile to parametric polymorphism

- The bytecode set has various annoying non-orthogonalities
- Some data types take one slot, some take two
  - How many LVT slots should we allow for a T in List<T>?
- Some instructions are not symmetric across types
  - Compare and branch: if_acmpeq for refs, dcmp + if for doubles
  - Array creation: anewarray for refs; newarray for primitives
  - Default values; aconst_null for refs; iconst_0 for int, etc
- These make it hard to represent a generic class in a classfile
  - Also made the specialization transform in Model 1 highly intrusive

ORACLE

Specialized Generics

Model 3

- Our 3rd attempt lightly refactors the classfile format to move all specializable metadata to the constant pool
- Declarations and uses of generic types captured in the classfile
  - Attributes to capture generic class declaration
  - Constants to describe uses of type variables, parameterized types
  - Bytecodes / bytecode modifiers to describe moving tvar-valued quantities, boxing conversions
- End result – specializing a class becomes specializing the constant pool!
- Still needs some help with long/double taking two slots

 | |

Specialized Generics

GenericClass attribute

- An any-generic class has a GenericClass attribute
- "Table of contents" for type variables for this class *and enclosing classes*
  - Type variables from enclosing classes are implicitly part of a class declaration!
- Then refer to type variables by *number*

```
class Outer<any T> {
    class Inner<any U> { ... }
}
```

```
GenericClass {
    u2 name_index;
    u4 length;
    u1 classCount;
    struct {
        u2 clazz;
        u1 tvarCount;
        struct {
            u2 name;
            u1 isAny;
            u2 bound;
        } tvars[tvarCount];
    } classes[classCount];
}
```

Specialized Generics

ParameterizedType constant

- We need a way to denote List<int> in a descriptor
  - Where List is a generic class
- We describe a class with a Constant_Class_info
  - So how about describing a parameterized class with a similar constant?
- Needs to capture
  - Parameterization of enclosing class, if any
  - Name of the generic class to be parameterized
  - The type parameters

Specialized Generics

ParameterizedType constant

- Can denote **`List<int>`** as ***`ParamType[List, I]`***
- Type params can refer to “type entries” in constant pool
  - Including other parameterized types
  - Preserves structure of type description
- Can denote **`List<Optional<int>>`** as ***`ParamType[List, ParamType[Optional, I]]`***

```
CONSTANT_ParameterizedType_info {
        u1 tag;
        u2 enclosing;            // ParamType of enclosing class
        u2 templateClassName;    // Generic class
        u1 count;                // # of tvars
        u2 params[count];        // type parameters
}
```


Specialized Generics

Incorporating erasure

- We also need a way to describe an *erased* parameterization
  - At the very least, need this for compatibility with existing generics
  - Existing classfiles only know about erased parameterizations
- Use a special type parameter token (we use _) to denote “erased”
  - List of int : ***`ParamType[List, I]`***
  - List of reified String : ***`ParamType[List, Class[String]]`***
  - List of erased String : ***`ParamType[List, _]`***

Specialized Generics

MethodDescriptor constant

- How do we put a parameterized type in a method descriptor?
  - Currently, method descriptors just concatenate nominal descriptors of parameter types
  - But parameterized types don't have a nominalization…
- Need a structural descriptor for method descriptors!
  - Return type, arg types can refer to other types in CP

```
CONSTANT_MethodDescriptor_info {
    u1 tag;
    u1 argCount;
    u2 returnType;
    u2[argCount] argTypes;
}
```

Specialized Generics

ArrayType constant

- How do we refer to an array of a parameterized type?
  - Same trick – make a structural descriptor for array types
- Can denote **`List<int>[]`** as
  `ArrayType[1, ParamType[List, I]]`
- Type entries in ParamType, MethodDescriptor can also refer to arrays

```
CONSTANT_ArrayType_info {
   u1 tag;
   u1 arrayDepth;
   u2 componentType;
}
```

Specialized Generics

TypeVar constant

- How do we refer to a type variable, or a parameterization that includes a type variable?
  - Same trick – a TypeVar constant
  - Refers to a type var (by number), and carries (contextual) erasure with it
    - “In case of erasure, break glass”
  - Can denote **`List<T>`** as
    `ParamType[List, TypeVar[0, “LObject;” ]]`

```
CONSTANT_TypeVar_info {
   u1 tag;
   u1 tvarNumber;  // index into tvar table
   u2 ifErased;    // type to use for erasure
}
```


ORACLE

Specialized Generics

Structural descriptions of types

- Parameterized types (and array types) are fundamentally structural
  - Method descriptors are structural too
- New CP forms retain this structure, rather than flattening it
  - Leaves of tree are ground types (classes, primitives)
- “Type entry” fields can refer to a ground type, or to a ArrayType, ParamType, or TypeVar constant

Specialized Generics

Constant pool reduction

- Strategy: consolidate all type information in the constant pool
  - Much of the type information is already there (e.g., method sigs)
  - There should be *one* place where the binding T=int is recorded
  - Turn specialization of *classes* into specialization of the *constant pool*
- There is a simple, mechanical transformation on the CP for a generic class to produce a CP for any given specialization
  - Storing the erasure with each type variable use means that erasure computation is owned entirely by the language compiler
- VM is free to share rest of the class

ORACLE

Specialized Generics

Specialization example

```
class Example<any T, any U> {
    Example<T,U> example;
    Example<int, int> ii;
    Example<int, String> is;

    void m(Example<T, U> e) { }
}
```

```
 #2 = Utf8                 _              // erased
 #3 = TypeVar              0/#2           // T
 #7 = Utf8                 V
#11 = Utf8                 Example
#12 = TypeVar              1/#2           // U
#13 = ParameterizedType    #11<#3,#12>    // Example<T,U>
#23 = Utf8                 I
#24 = ParameterizedType    #11<#23,#23>   // Example<I,I>
#27 = ParameterizedType    #11<#23,#2>    // Example<I,_>
#32 = MethodDescriptor     (#13)#7
```

```
               T=_, U=_
 #2 = Utf8     _
 #3 = Utf8     Object
 #7 = Utf8     V
#11 = Utf8     Example
#12 = Utf8     Object
#13 = Utf8     Example
#23 = Utf8     I
#24 = Utf8     Example${II}
#27 = Utf8     Example${I_}
#32 = Utf8     (LExample;)V
```

```
               T=int, U=_
 #2 = Utf8     _
 #3 = Utf8     I
 #7 = Utf8     V
#11 = Utf8     Example
#12 = Utf8     Object
#13 = Utf8     Example${I_}
#23 = Utf8     I
#24 = Utf8     Example${II}
#27 = Utf8     Example${I_}
#32 = Utf8     (LExample${I_};)V
```

```
               T=int, U=int
 #2 = Utf8     _
 #3 = Utf8     I
 #7 = Utf8     V
#11 = Utf8     Example
#12 = Utf8     I
#13 = Utf8     Example${II}
#23 = Utf8     I
#24 = Utf8     Example${II}
#27 = Utf8     Example${I_}
#32 = Utf8     (LExample${II};)V
```

Specialized Generics

New bytecodes

- Wait, what about bytecodes?
  - Bytecodes operands point into the CP too!
  - So if we modify a bytecode with a “typed” prefix…

```
typed operand  aload_0
```

  - Operand points to a type entry in the CP (like a TypeVar constant)
- Theoretically only need a “typed” prefix, and a conversion bytecode (for boxing and unboxing)

```
a2b from-operand to-operand
```

  - Alternately could define family of uload/etc which take a type operand

ORACLE

Specialized Generics

Runtime representation

- Historically, there was a (mostly) 1:1 relationship between source files, classfiles, and runtime types
  - Not for values: classfile describes at least two types, the value and the box
  - Not for generics: classfile describes a *parametric family* of runtime types
- We've been using the term *species* to describe deriving multiple related runtime types from a single classfile
  - The *class* of **`List<int>`** is still List, but the *species* is **`List<int>`**
  - **`Object.getClass()`** will still return the class
  - Something else (TBD) will have to return the species

Specialized Generics

Generic methods

- Generic methods pose a new challenge
  - VM has a notion of class, but no first-class notion of method
  - But code can refer to type variables defined in enclosing generic methods
- Need to include enclosing generic methods in GenericClass “table of contents”

```
class Outer<any T> {
    class Inner<any U> {
        <any V> void m() {
            class Local<any W> {
                void m(T t, U u, V v, W w) { ... }
            }
        }
    }
}
```


ORACLE

Specialized Generics

Generic methods

- Current strategy is to desugar generic methods into nested classes
  - Reduces method specialization to class specialization
  - Invoke specialized via invokedynamic – bootstrap takes specialization params (statically known at compile time)

```
class Foo<any T> {
    <any U> void m(T t, U v) { ... }
}
```

```
class Foo<any T> {
    bridge void m(Object t, Object u) { ... } // erased bridge

    synthetic class Foo$m<any U> {
        species-static void m(T t, U v) { ... }
    }
}
```

Specialized Generics

These are not the reified generics you're looking for…

- A specialized class List<int> *is* reified
  - But List<String> (probably) won't be
  - M3 classfiles can express both List<String> and List<_>
  - Choice of when, and how, to erase becomes language's prerogative
- Reified generics are harder to program with
  - Real-world code resorts to tricks that implicitly assume erasure
  - Casting through raw, unchecked ops
- Many potential compatibility, performance pitfalls
  - We still want gradual migration compatibility for anyfying a library
- So erasure is *still* (probably) a pragmatic compromise here

Specialized Generics

Species statics

- There are currently two “places” to put state / behavior
  - Static members – associated with a class
  - Instance members – associated with an instance
- Is it useful to associate members with a species too?
- Yes! This is the natural placement for
  - Cached instances (e.g., empty list)
  - Instantiation tracking (e.g., counters, interning)
  - Reified type variables (e.g., List<T> has a species-specific field T)
  - Static factories
  - Cached associations (e.g., preferred box type of X in Y)

Specialized Generics

Species statics

```
interface List<any T> {
    // new way
    private species-static List<T> empty = new EmptyList<>();

    public species-static <T> List<T> emptyList() {
        return empty; // no cast needed
    }

}
```

Specialized Generics

Species statics

```
class Foo<any T> {
    public synthetic final species-static ReflectiveThingie T;
}

Foo<int> fi = ...
... fi.T ...  // reflective mirror for int
Foo<String> fs = ...
... fs.T ...  // reflective mirror for “erased”
```

Specialized Generics

Nestmates

- There's a mismatch between the language-level rules for accessibility and the VM rules
  - In Java language, private means "accessible from anywhere in my top-most enclosing class"
  - In VM, it means "only from within this classfile"
  - Compiler emits access bridges (access$000) and downgrades private to package to make up difference
- With species, the set of runtime types that derive from a single top-level source class gets bigger (and more complicated)
  - We introduce the concept of *nest-mate,* which eliminates the need for bridges / encapsulation downgrades

Specialized Generics

Nestmates

- Nests form a partition over classes
  - Each class belongs to exactly one nest
  - What constitutes a nest is defined by language compiler
  - All specializations of a class C belong to C’s nest
  - All inner classes of C belong to C’s nest
  - Private becomes “accessible from within my nest”
- This is *not* “friends”
  - Simply a generalization of “common compilation unit”
  - Fixing some age-old technical debt

Specialized Generics

Partial methods

- As the domain of generics broadens to include things like numerics…
  - We start to want to condition behavior on receiver type parameters

```
interface Stream<any T> {
    <where T implements Arithmetic>
    T sum();
}
```

  - Here, sum() would be a member of Stream<int> but not Stream<Shoe>
  - Represented in classfile as a ConditionalMethod attribute on the method
- Both an expressivity feature *and* a migration compatibility feature

Specialized Generics

Wildcards

- Model 1 had no wildcards
  - This was “wildly” unpopular
  - Very difficult to port existing generic libraries without them

```
class Foo<any T> extends Bar<T> { ... }
```

  - Foo<int> is a subtype of Bar<int>
  - And also a subtype of Foo<?>
- Foo<?> can be neither a class nor an interface
  - Needs to be something new
  - Need VM help here

Reflection

- How do we reflect over specialized classes?
  - Can we get a Class for List<int>?
  - Can we reflect over an abstract template List<T>?
  - How do we reflect over generic methods?
  - How do we model erasure in reflection?
- See “One Mirror To Rule Them All”, later today…

Arrays

- The lack of a common useful supertype between int[] and Object[] becomes a bigger problem
  – What happens when a method returns T[]?
- Some subset of Arrays 2.0 is needed here
  – Common supertype Array<any T> for all array types?
- For migration compatibility, some way to migrate an Object[]-bearing method to a T[]-bearing method is needed

Migration

- Phew, that was a lot
  - … and we're not done
  - … actually we're just getting started!
- What we've outlined so far might be OK if this was a NEW language
- What about migrating existing APIs, clients, and implementations?
  - Some APIs won't anyfy cleanly
  - Anyfying core libraries *must* be compatible – for clients *and* subclasses
  - Can we migrate existing reference types to value (like Optional?)
  - Can we consolidate IntStream into Stream<int>?
  - Can we consolidate the nine version of Arrays.fill() into one method?

Migration

Anyfying existing libraries

- Just as generifying libraries not designed for generics posed challenges…
- … Anyfying APIs not designed for values also poses challenges
  – Some signatures use Object instead of T, or Foo<?> instead of Foo<? {extends,super} T>
  – Some overloads become questionable (e.g., remove(T) vs remove(int))
  – Some methods use null to signal “no answer” (e.g., Map.get())
- Anyfying implementations also requires code adjustments
  – Assignment to / comparison with null
  – Array creation – new T[n]
  – Instanceof / cast

Migration

Migrating classes to values

- Some classes, like Optional, LocalDateTime, and BigInteger are semantically values already
  - We'd like to migrate their implementation to be values too!
- L/Q descriptor split was motivated by enabling binary compatibility via bridge methods
  - Good story, but doesn't get us 100% of the way there
  - LOptional is nullable and QOptional is not, means there are source compatibility issues to be worked out

Migration

Signature migration

- There are many kinds of signature migrations we'd like to make compatible
  - Migrate off of deprecated types: m(Date) -> m(LocalDateTime)
  - Widen returns: migrate Collection.size() to return long
  - Convert value types to reference types
  - Squeeze IntStream to implement Stream<int>
- The older our libraries get, the more important it is that we be able to flexibly and compatibly evolve them
- As we add language features, we'd like for existing libraries to have a migration path, rather than making them "instant legacy"

ORACLE

Migration

Signature migration

- Investigating features to enable library authors to provide metadata for "this method migrated from …"
  - Old signature, conversion functions for argument/return
  - Box/Unbox, Date <-> LocalDateTime, etc
- Binary compatibility for clients can be handled by bridges
- Binary compatibility for subclasses is harder … working on this
- Source compatibility is another story…

Summary

- We started out with a simple performance goal – dense/flat aggregates
  - This led to value types
- In order for values to be useful, must address interaction with arrays, generics, reflection, core libraries, …
  - It's a long string!
- In order for new language features to be useful, must be possible to bring old libraries up to date
  - So migration tools are a big part of the story too
- Some areas well understood … some areas still research

ORACLE

Summary

Current status

- 3rd-generation prototype of anyfied generics (works over primitives) in Valhalla repo
  - More limited prototype of value types
- For further reference:
  - Model 3: http://cr.openjdk.java.net/~briangoetz/valhalla/eg-attachments/model3-01.html
  - State of the Values: http://cr.openjdk.java.net/~jrose/values/values-0.html

ORACLE

Project Valhalla Update

Brian Goetz, Java Language Architect

JVM Language Summit, Santa Clara, August 2016

ORACLE