Evolving the JVM: Principles and Directions

John Rose, Java VM Architect
Brian Goetz, Java Language Architect

JVM Language Summit, Santa Clara, July 2014

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

JVM Vision, circa 1995

Theory: JVM is not just for the Java language

- “The Java virtual machine knows nothing about the Java programming language, only of a particular binary format, the class file format.”
- “Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine.”
- “In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages.”
  - *JVM Specification* First Edition (1997), preface and §1.2
  - also reaffirmed in the preface to the J2SE 7 Edition (2013)

JVM Status, circa 1995

Practice: Some instructions have fixed, Java-like semantics

- invoke* instructions
  - Linkage is static; once done, cannot be re-done
  - Single-dispatch, receiver-based selection
  - Single inheritance of implementation
  - No argument or return type adaptation
  - Limited return types (multiple values, but only via heap)
- Serious pain point for language implementors
  - Workarounds exist, with pain (reflective dispatch and/or adapters)

ORACLE

JVM Vision, circa 2009

The down payment: JSR-292

- JSR-292 opened up method dispatch to arbitrary linkage semantics
  - invokedynamic: extensible invocation mode
  - MethodHandle: access to all previous invocation modes
  - guardWithTest, etc.: MethodHandle composition operators
  - (Mutable)CallSite, SwitchPoint: options to re-link calls

- Result: Less pain. Invocation sites can be shaped, not worked around.
  - *Caveat: Multiple return values are still a pain.*

JVM Status, circa 2013

- “In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages.”
  - *JVM Specification* First Edition (1997), preface
- “The Java SE 7 platform in 2011 made good on [this] promise.”
  - *JVM Specification*, J2SE 7 Edition (2013)

ORACLE

- Great start!
  - What's next?

JVM Vision, 2014

- Let's find some more pain points, where JVM semantics…

  … align too rigidly with Java language semantics,

  … fail to align closely with modern hardware,

  … or impose excessive costs in some other way.

- It appears we can relieve major pain points.
  - Improve simplicity and performance for new users
  - Retain compatibility and performance for present users

ORACLE

JVM Pain Points (for language implementors)

Pain Point	Tools & Workarounds	Upgrade Possibilities
Names (method, type)	mangling to Java identifiers	unicode IDs ✓1.5/JSR-202, structured names
Invocation (mode, linkage)	reflection, intf. adapters	indy/MH/CS ✓1.7/JSR-292, tail-calls, basic blocks
Type definition	static gen., class loaders	specialization, value types
Application loading	JARs & classes, JIT compiler	Jigsaw, AOT compilation
Concurrency	threads, synchronized	Streams ✓1.8/JSR-335, Sumatra (GPU), fibers
(Im-)Mutability	final fields, array encap.	VarHandles, JMM, frozen data
Data layout	objects, arrays	Arrays 2.0, value types, FFI
Native code libraries	JNI	Panama

+ sun.misc.Unsafe

ORACLE

JVM Pain Points (for language implementors)

Pain Point	Tools & Workarounds	Upgrade Possibilities
Names (method, type)	mangling to Java identifiers	unicode IDs ✓1.5/JSR-202, structured names
Invocation (mode, linkage)	reflection, intf. adapters	indy/MH/CS ✓1.7/JSR-292, tail-calls, basic blocks
Type definition	static gen, class loaders	specialization, value types
Application loading	JARs & classes, JIT compiler	Jigsaw, AOT compilation
Concurrency	threads, synchronized	Streams ✓1.8/JSR-335, Sumatra (GPU), fibers
(Im-)Mutability	final fields, array encap.	VarHandles, JMM, frozen data
Data layout	objects, arrays	Arrays 2.0, value types, FFI
Native code libraries	JNI	Panama

+ sun.misc.Unsafe

(got all that?)

ORACLE

Which language implementors?

…four letter word …starts with ‘J’

- Java is improving in each release
  - With great care and deliberation
  - Selective “sedimentation” of proven features (Reinhold)
  - The JVM evolves with the language
- The JVM also evolves with the underlying hardware
  - Sedimentation process is not just language-driven
- JVM design addresses a broad range of languages and hardware
- Wider applicability interests more people, applies more brainpower

When extending the JVM…

(some major principles)

- Never break old bytecodes
  - gate new behaviors on class file version number
  - new features must not interfere with old ones, even in new class files
- New mechanisms are supersets of old ones *(wherever applicable)*
  - e.g., invokedynamic provides complete access to old “bytecode behaviors”
  - overheads are collapsed by the compiler; no built-in “simulation overheads”
  - JVM users need not choose *clean* vs. *fast, expressive* vs. *compatible*
- Design to “categorical” language and machine capabilities

Some current JVM initiatives

- Project Valhalla http://openjdk.java.net/projects/valhalla/
  - Value Types – aggregates without identity
  - Specialization – templated types on demand
  - JMM Update – VarHandles
- Project Panama http://openjdk.java.net/projects/panama/
  - Arrays 2.0 – flexible array implementation and organization
  - Layouts – flexible object layout
  - FFI – better native code interop

What's in a value?

(word history first, as found in http://etymonline.com)

- **value (n.)** c.1300, "price equal to the intrinsic worth of a thing;" late 14c., "degree to which something is useful or estimable,"
- … from Latin *valere* "be strong, be well; … worth"
- … from Proto-Indo-European root **wal-* "be strong"
- cognates: Old English *wealdan* "to rule," Old High German *-walt*, *-wald* "power" (in personal names), Old Norse *valdr* "ruler," Old Church Slavonic *vlasti* "to rule over," Lithuanian *valdyti* "to have power," Celtic **walos-* "ruler," Old Irish *flaith* "dominion," Welsh *gallu* "to be able"

ORACLE

Blast from the past: Sumerian accounting

http://blogs.utexas.edu/dsb/tokens/the-evolution-of-writing/

- Mobile handheld
- Message based
- Token passing
- Polymorphic
- Secure envelope
- Linear (B) logic
- Silicate substrate
- Sub-cm process

…method and apparatus for symbolically transmitting objects of value…

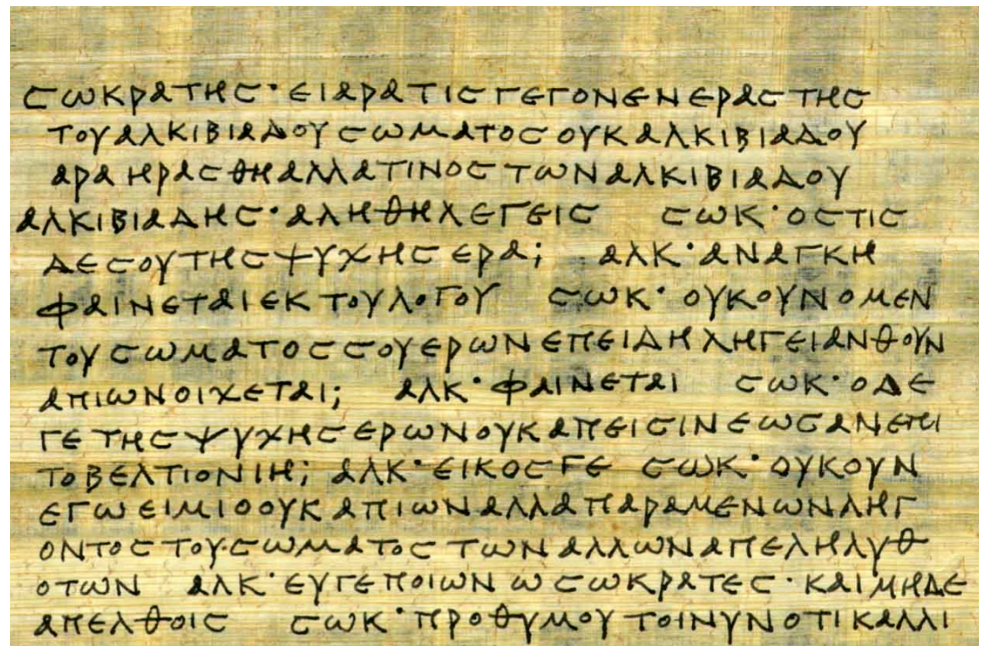
ORACLE

Subsequent refinements

http://magazine.uchicago.edu/1102/features/the_origins_of_writing.shtml

Subsequent refinements

http://magazine.uchicago.edu/1102/features/the_origins_of_writing.shtml



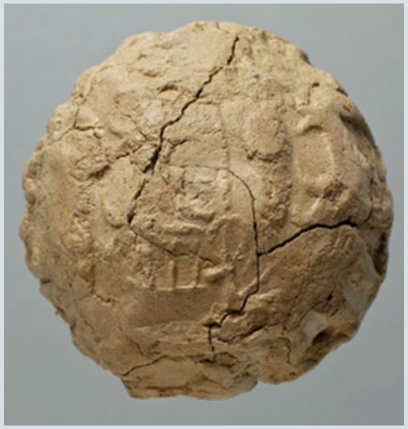
images from commons.wikimedia.org

ORACLE

What is a value, for computers?

- Any indication of quantity or quality, something like a symbol.
  - Chosen from a fixed set of alternatives (dynamic range, alphabet).
- Values can be recorded and copied at negligible cost.
  - Like written letters. Unlike clay tokens or coins.
- All such values (symbols) can be resolved to bits. (Shannon, 1948)
  - They also occupy channels: Clay, paper, media, ether, cache lines.
- In the setting of the JVM, a managed pointer, after “new”, is a value
  - **pre-existing managed pointer** = **special kind of bits**.

Values, objects, and immutability.

Object	Immutable Object	Value
		☺

(exaggerating to make a point here)

Values in the JVM

- So, we have value types already: Primitives and references
  - Yes, a reference is a value.
  - But (according to our working definitions) most **objects** are **not** values.
- The main problem with JVM values is composition (composite values)
  - Needed when the primitives are not quite right: BigInteger, Complex
  - JVM composites, **objects**, are expensive to construct
- Another problem is control of concurrent side effects (JMM)
- Simple answer: Make pointers optional

ORACLE

Pointer-free programming in the JVM

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)

ORACLE

Example: a *value-based* class

(N.B. this is not a value type, yet)

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

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


ORACLE

Value-based classes

- Let's just pretend the pointer isn't there!
  - And hope the optimizer gets the idea
  - (And the GC. And reflection. And…)
- A "value-based" class is defined as one whose pointer is negligible

  http://docs.oracle.com/javase/8/docs/api/java/lang/doc-files/ValueBased.html
- But we need a way to promise the JVM that it can always optimize
- This means new types, not new optimizations on old types

ORACLE

Example: a value type

(same as the previous class, with a little more markup)

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

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


Example: methods on a value type

```
final __ByValue class Point {
    ……
    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); }
}
```


ORACLE

Codes like a class, works like an int!

http://cr.openjdk.java.net/~jrose/values/values.html

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;
}
```


ORACLE

Pointer-free *values* in the JVM

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

Y U No Make Syntax Beauty?

- **__ByValue** and **__MakeValue** are calculated, blatant bad form
  - Lest anyone suppose we are proposing a source code notation
  - It is too early to define Java language syntax
- JVM folk need to define a bytecode syntax independently of JLS
  - Hey, JLS doesn’t define a syntax for the invokedynamic instruction either
- Lots of syntax choices involving ASCII and Unicode
  - JVM folks are deeply interested in syntax of UTF-8, u1, u2, u4, etc.

Selected details

(see values.html for many, many more)

- An array of values is not a subtype of an array of references
  - Works like an int, so like int[] or long[], flattening is expected.
- A value type is distinct from its (unique) box type
  - Auto(un)boxing rules apply to values as well as primitives
- Primitives are not exactly value types, but act like them
- A value field can be of almost any type, but must be final
  - Values should not have lots of fields, though limits are lax
  - No variable-length or recursive values. (Use a reference field.)

Some use cases

(see values.html for more)

- *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.

ORACLE

Arrays 2.0 – degrees of design freedom

A million ways to roll an array

- Rank and size (dimensions)
- Index type (int, long, *other?*)
- Element storage (type)
- Locality design (row- or column-major, chunked, nested)
- Managed vs. native
- Loops (elemental ops, linear algebra, streams, for-loop, etc.)
- Element variability (read-only, single write, concurrent update)
- Structure variability (append, insert, delete, etc.)

ORACLE

Sometimes freedom can slow you down!

For real freedom, define the ingredients, not the menu

- Remember that restaurant with the 20-page menu?
  - Better to select from a short list of fresh and healthy ingredients

- Use interfaces to create clean APIs (specialization will help also)
  - Use classes to cleanly layer implementations
- Supply a few low-level storage tactics, following hardware & GC design
- Let the library experts (JVM customers) invent the detailed recipes

ORACLE

Needed: a well-appointed kitchen

- Classic arrays: Flat, 1-d, mutable (except frozen), non-concurrent
- A couple of internal chunking styles, optimized for flatness
  - Blocked array (all element types, including multi-field value types)
  - Low-arity B*-tree nodes
- A few concurrency strategies (includes struct-tearing protection)
  - Final/persistent, fenced/concurrent, divide-and-process
- Beyond that, classes & interfaces can build & defend new APIs
  - Syntax alert: Might want abstract array notations a[i] = x

ORACLE

A note on scale

- N.B. Galaxy-sized contiguous arrays are an anti-pattern
  - 32 bits is almost large enough, for implementation blocks.
- Terabyte scale logical arrays should be segmented physically.
  - Segmentation should not appear to the user
  - Except perhaps via spliterators
- The most important locality is at the scale of a cache line (LSU)
  - Or perhaps a few hundred of them (HW/SW prefetch)

API notes

Standard operations (access)

- basic operations drawn from java.util.Arrays (binary search, sort, etc.)
  - sub-array, sub-matrix (aliased views, cf. List.subList)
  - element, row, column streams and/or collections
- array as matrix, matrix as array views (cf. APL reshape)
  - array and matrix gluing (aliased views, border creation)
- these can be defined as default methods
  - but can be specialized as needed

API Notes (2)

Standard operations (element processing)

- standard bulk (whole-array) arithmetic (cf. APL)
- transpose or reshape
  - as copy (not aliased)
  - as view (aliased)
  - in place (potentially overwrites representation)
- selected linear algebra operations on vectors, matrices

More JVM heap support for arrays

A few new tricks for managed memory

- Frozen arrays (safe immutability)
  - Requires a freeze() method to go with the clone() method
  - A few well-placed freeze() calls enable chains of copy-elimination
  - Must be a dynamic property of classic arrays, not a new kind of array
- Mixed arrays (It’s an object! No, it’s an array!)
  - Objects with inlined private arrays; **not** a subtype of classic arrays
  - Safe version of the C trick of a struct with trailing zero-length array
  - Requires special factory-style constructors (new;<init> consider harmful)
- Maybe, something with a more programmable mix of bits and refs.

ORACLE

Native interconnect

- 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
  - value and view transformation
- 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.

Synergy with other JVM initiatives

- *Arrays 2.0:* Native arrays can be presented using array APIs
- *Value Types:* Can efficiently carry native types: complex, int128
- *Specialization:* Native type management, but not built into the JVM
- *Invokedynamic:* Native linking rules, but not built into the JVM
- *Sumatra:* Effective binding to fast moving GPU-related APIs.

To GC or not to GC?

- Some high-end customers need “wired” buffers in native heap
  - Native interconnect can help place this data where it’s needed.
  - Arrays 2.0 APIs make it look clean.
- Low-level access methods (based on Unsafe) can reach both sides
  - Allows some temporary quasi-values to be buffered on the managed heap
  - Allows code to be written without a native/managed committment
- Thread-scoped temporaries will need new mechanisms
  - There are good options available using try-with-resources

There's lots more on the distant horizon

- More hooks to vary hardwired JVM behaviors (see specialization!)
- Bytecode design cleanups (many small ones)
  - But, we are trying to avoid a complete classfile format overhaul
- New tools for statically processing code and data (jlink)
- Lighter alternatives to threads (coro and/or fibers and/or GPU SIMD)
  - (J. Rose is still looking for a compelling design for the JVM.)
- Tail-call, plus reification of basic block states as MHs for tail-calling

 | | ORACLE

Java Language-VM co-evolution

Where to put new features?

- When we implement a language feature, we could…
  - 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
- The trick is finding the right balance
  - Minimize impedance mismatch between language and VM
  - …without exporting “language problems” onto the VM

ORACLE

Language-VM co-evolution

The balancing act

- Try to balance
  - Keeping Java language complexity from impinging on VM complexity
  - Avoiding impedance 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

ORACLE

Java Language Initiatives

- Specialized generics (generics over primitives and value types)
  - On-the-fly adaptation of classes and methods
  - Motivates some new general-purpose VM features
    - classdynamic – programmatic class generation
    - Possibly “method missing” support
- Value types (and tuples)
  - Builds on VM support
- Atomic and fenced data access
  - VarHandle

Generic Specialization

Motivation

- Refresher: why value types?
  - Smaller footprint (no object header)
  - Better locality (no dereference)
  - Simpler semantics (no identity, no aliasing worries)
  - Lower operational impact (no allocation and GC)
- *Don't make user choose between abstraction and performance*

Generic Specialization

Motivation

- The same reasoning applies to generics
- Consider ArrayList<Integer>
  - Lots of boxing, extra footprint, loss of locality, possible aliasing
  - Yuck!
- User really wants ArrayList<int>
  - And have it backed by a real int[]
- Its bad enough we have eight types that don’t play nicely with generics
  - When we have value types, more than half our types wouldn’t
  - Would undermine the usefulness of value types

Generic Specialization

Basic Idea

- Given a class Box<T>
  - Currently, erase T to Object
    - And insert casts at use site

```
class Box {
    Object val;

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

    public Object get() { return val; }
}
```

- To specialize Box<int>, we would need different signatures
- Conclusion: we can cheat by using one class for ref types, but this trick does not scale well to non-reference types
  - Two choices: try to unify, or admit reality

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

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

    public int get() { return val; }
}
```

 |

ORACLE

Generic Specialization

Basic Idea

- Converting Box.class for T=int is not just mangling signatures
  - Mangle the bytecodes too!
  - Also, Box<int> does not extend Box
- Classfile representation annotated to reflect which signatures and bytecodes need adjustment for specialization

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

public Box(Object);
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method Object."<init>":()V
   4:   aload_0
   5:   aload_1*T
   6:   putfield    #2; //Field t;
   9:   return

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


Generic Specialization

Classfile representation

- Specializations should be generated on the fly, as needed
- Classfile serves double-duty
  - Directly loadable as erased class
  - Can be used as a template for generating specializations
- Need a way to write “Box<int>” in a classfile
  - But don’t want to impart semantics to naming convention like Box${T=int}
- Box<int> really means “The result of applying the specialization transform, with parameters T=int, to class Box”
  - Can we somehow write that in the classfile?

ORACLE

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

ORACLE

Classdynamic

Invokedynamic for class generation

- Strawman: 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

ORACLE

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

*Even better if classdynamic can generate value types

Specialization of Generic Methods

- Java also supports generic methods
  – Need a mechanism for specialization of them too
  – Example: `<T> T identity(T t) { return t; }`
- 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 of Generic Methods

“Method Missing” handlers?

- Many systems have a mechanism for providing last-ditch nominal linkage when traditional resolution fails
  - Generic method specialization can be implemented with a traditional “method missing” handler
  - Method-missing handler would be associated with a class, would consume a signature and produce a MethodHandle
  - Search order would follow usual inheritance order
  - Interaction with reflection … TBD
- Alternately, specialized generic methods could be invoked with indy
  - Much simpler, but less flexible

ORACLE

Specialization Challenges

- Suppose I write a generic class ArrayList<any T>
  – Can I provide a hand-written replacement for ArrayList<boolean> ?
  – Can I do the same for a single method (e.g., hand-written version of ArrayList.contains() for T=int?)
  – Can I do the same for a specific instantiation of a generic method?
  – Can I add a sum() method to ArrayList<int>, that is not a member of other instantiations of ArrayList?
- Not being able to do these things would be a big limitation
  – Sometimes generic code is *too* generic

ORACLE

Specialization Challenges

- Extending generics to primitives/values brings new challenges
  - Can’t assume “T extends Object”
  - Can’t assume “T[] extends Object[]”
    - But ArrayList still need *some* means of expressing “new T[]”
  - Can’t assume `null` is a valid value for T
    - Bad news for Map.get()
  - Can no longer overload remove(T) with remove(int)

Conditional Methods

```
class ArrayList<any T> {
    ...
    <where T=int>  int sum() { ... }
    <where T=long> long sum() { ... }
}
```

- Most of these challenges can be met with *conditional methods*
  - These are methods that only appear in some instantiations
  - They can be new methods or override existing methods
- Methods marked by special attributes
  - Ignored during ordinary class loading
  - Acted on by specializer (include the method or not, depending on T)
- Can be conditional on T=primitive, “T is a reference”, “T is a value”
  - Maybe on “T extends bound”, maybe not

ORACLE

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
- Syntax still TBD!

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

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

    public T get() { return val; }
}
```


ORACLE

Value Types

- The sweet spot for value types are those aggregates that benefit from being passed … by value
  - Small tuples
  - Alternate numerics (complex, unsigned int)
  - Algebraic data types (Optional<T>, Choice<T,U>)
  - Cursors
- Motivation: don’t force users to choose between safety/abstraction and performance
  - Optional<T> provides a lot of type safety – but shouldn’t cost anything

Open question: value polymorphism

- In the current design, value types are not polymorphic
  - This may well be too limiting
- Considering a limited form of value polymorphism
  - Something like interfaces for values
  - But without boxing, identity, or heap allocation
- Example: Arrays 2.0
  - Many kinds of array representation
  - Would be nice if we could treat them all as Array<T>
    - While still allowing an array reference to be a value

ORACLE

Challenge: Migration Compatibility

- Some classes today, like Optional<T>, should have been values
  – Will we be able to migrate these to be real values in the future?
    - If the answer is “no”, that would be sad
  – These classes already disclaim use in an identity-sensitive fashion
  – Can we migrate `Ljava/util/Optional;` to `Qjava/util/Optional;` compatibly?

Tuples

Really just classdynamic + value types!

- Tuples are an obvious application for value types
  - Tuple of T,U can be represented by `classdyn { Tuple(T, U) }`
  - Tuple bootstrap spins a simple value class with fields of the given types
  - Two tuple classes are the same if their component types are the same
- To represent in the Java language, “all” we need is:
  - Means of denoting type “tuple of T,U” (e.g., `[T,U]`, `Tuple<T,U>`, etc)
  - Means of constructing a tuple value from components (e.g., `[e1, e2]`)
  - Means of destructuring a tuple into its components (e.g., `[ a, b ] = aTuple`)

Tuples

Not just for the Java language!

- Recall central challenges of language-VM co-evolution
  - Minimize impedance mismatch between Java language and VM
  - When we evolve the VM for Java, make sure others can play too
- These tuples build trivially on general-purpose VM building blocks
  - Value types and classdynamic
  - (Exact same trick can be used to generate function (arrow) types)
- We can put java.lang.Tuple bootstrap into the platform runtime
  - Others are free to use or ignore it
  - If they use it for their tuples, cross-language interop comes for free

ORACLE

VarHandles

Method handles for data

- Currently, support for atomicity and fencing is limited
  - Accesses to volatile fields automatically fenced, others not
  - Fenced operations and atomic operations (CAS) available through Unsafe
  - Is it time to bring these into the programming model “for reals”?
- VarHandle is like method handles for data
  - Abstracts over location – static fields, instance fields, array elts, off heap
  - Supports explicit fenced and atomic operations
- Safer than Unsafe, as fast as MethodHandle
  - Maybe will add language support, maybe VarHandle API is enough

Summary

- Common theme: more flexible access to data
  - Value types – aggregation without indirection
  - Arrays 2.0 – flexible data layout
  - FFI – access to off-heap data
  - Generic specialization – bring the benefits of value types to generics
  - VarHandle – more flexible, high-performance access to variables
- Time to stop making programmers choose between expressiveness/ abstraction/safety and performance

