ORACLE®

ORACLE

Arrays [2.0^{64}] – opportunities and challenges

John R. Rose
Da Vinci Machine Project, JSR 292 Lead

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.

Why we can't live without arrays

- Need at least one type with variable size!
  - Bucket for puddles of data. (Linked lists are only bandoliers.)
- Strongly typed (consonant with the rest of Java)
  - → must be generic
- Smallest memory footprint (±ϵ)
  - → must have at least a few packed representations (`byte[]`)
- Efficiency: Minimum cache line accesses (±ϵ)
- Notation (yes, notation counts when programming)
  - Definition: `int a[] = {1,2,3};`
  - Element access: `a[1] += 5;`
- (Type safety and security are non-negotiable.)

Why arrays bother users

- Choose any flavor you want as long as it’s vanilla.
  - Length is never mutable. Body is always attached.
  - Elements are always homogeneous. (No C array+struct.)
  - Elements are always mutable (but never volatile).
  - Rank is always unity. Rows are always ragged.
  - Size is as big as you want, if you have modest expectations.
- T[] is covariant in T. (And there is no top Array type.)
  - Sometimes this seems to help. (Generic element types instead?)
  - Sometimes it’s just confusing.
  - Array store check is a hidden cost.
- Not a real object type. (`Arrays.copyOf(a)`!?)
  - Will the real `toString` method please expel the fake one?

Why arrays bother JVM implementors

- Lots of ad hoc special rules for arrays.
- Irregular appearance of fields and methods.
- Must provide generic instances Q[] on app. request.
- Suffer from megacephaly: Big headers.
  - Can you synch. on an array? Yes, but don’t.
- Big arrays provide bulky work units for GCs.

My so-called Meme

what Java coders say they want	what Java coders really want	what they would request if they knew to ask
what the JVM should provide to Java coders	what we manage to build into the JVM	what the coders finally receive

My so-called ~~Meme~~ stichomythia

what Java coders say they want	what Java coders really want	what they would request if they knew to ask
what the JVM should provide to Java coders	what we manage to build into the JVM	what the coders finally receive

What Java coders say they want...

- Rank > 1 (Fortran matrices, etc.)
- Size ≥ 2^{31} (“long” indexes)
- operator overloading
  - a[i], a[i]=x is really my method for my favorite array 1.01
  - maybe a[i,j...]; maybe a[i]+=x
  - *(hey, look at all those C++ and Scala folks having fun!)*
- layout control
  - array of structs; type with variable arrays
  - foreign (C) data access (w/ nio, JNI)
  - copy-free access to slices of data, scatter/gather

What Java coders say they want... (2)

- safe decomposition and sharing
  - final, volatile, etc.
- heterogeneous puddles of data
  - tagged data
  - buckets of less-structured data
  - serialization without tears (!?)
  - JSON/XML blobs (with pointers instead of bytes)
- flat-data performance
  - assume contiguous storage, close to the metal
  - dead-reckoned addresses → loop transforms in JIT

Speculative ravings follow!

- “View” of one Blind Man feeling the Elephant
- This is not a language design talk.
  - (Unless you believe invokedynamic is a language feature.)
- These thoughts are JVM-centric.
  - cowardly ducking away from controversy
  - most people care passionately about notation
  - JVM internals are mainly for us plumbers
- Let’s make progress...

What users "really" want (I think)

- Increase small scale collocation
  - Graceful use of cache lines
  - More dead reckoning of indexes; structs of arrays of structs
- Allow large-scale decomposition (tasks, etc.)
  - NOT large-scale contiguity (no terabyte memory blocks)
- Memory fencing/protection for safe sharing
- Compose complex sharing patterns
  - from a few independent and powerful primitives

rank: what the JVM should provide

- Which square matrix? Row-major? Col-major?
- About a dozen important sparse representations too.
- This is about cache-grace, not `A[i][j]` vs. `A[i,j]`
- Key operation: index computation
  - Must be a library-defined method, not a new part of the JVM.
- **Requirement:** Library definition of many array types.
- Key operation: loop decomposition
  - Today, this is a job for off-the-shelf BLAS/LAPACK type code.
  - Need to serialize chunk access between GC and BLAS
  - **Requirement:** pinning

bigness: what the JVM should provide

- Size ≥ 2^{31} `A[(long)x] = 5`
  - Hard requirement? More like a “red face test”.
- *Warning:* Big (contiguous) Data → Big Copies
- **Requirement:** Library definition of array types.
  - Hello, Scala & Fortress!
- Anti-pattern: planet-sized contiguous memory chunks
  - Modern GCs are regionalized.
  - Regions are continents, not galaxies.

resize: what the JVM should provide

- Supply a safe resize operation for arrays.
  - But only to *decrease* size.
- `Arrays.chop(T[] a, int newlength)`
- Needed to reduce copying in `StringBuilder`, etc.

notation: what the JVM should provide

- operator overloading?
  - The JVM’s ops are `arraylength`, `aaload`, `iastore`, etc.
  - Also `System.arrayCopy`, `Arrays.copyOf`, etc.
- **Requirement:** Re-interpret existing *bytecodes*
  - as shorthand for patterns like `x.length()`
  - verifier & JVM still hardwires privileged legacy types
  - verifier treats `arraylength` as `invokevirtual`
  - descriptor is `A.getArrayLength()int` (with A from verifier)

layout: what the JVM should provide

- Need a little more from the GC / heap manager
  - GC owns low-level memory layouts
  - That’s where the cache help has to be.
- **Requirement:** hybrid arrays
  - GC sees object with length (like legacy arrays)
- “Envelope + body” are fused into “head + tail”.
  - Dead-reckoned addressing: sizeof(hdr) + sizeof(elm) * N
  - GC knows how to find references in head and tail.
  - **Requirement:** Tail can be periodic repetition of small struct.
- Java sees a plain instance
  - Head is a first-class Java object with methods and *everything*.
  - Tail is accessed by new intrinsics. Private to enclosing class.

layout: what the JVM should provide

- Foreign data can be accessed via Unsafe.
- This should be encapsulated via header file import.
- Cf. CLR “delegate marshalling”, etc.
- LAPACK/BLAS needs to dictate layout details.
  - Copy-on-invoke is a lose.
  - ...Although the memory fabric is surely doing copies.
  - ...But we don’t want the JVM to interpose on memory ops!

hybrid arrays in the VM

- From our SCCS history, for `Klass::layout_helper`
  - src/share/vm/oops/SCCS/s.klass.hpp
  - D 1.136 07/01/29 21:20:30 jrose 281 280
  - c 6516018 Replace size_helper and is_objArray by more capable layout_helper.
- The layout-helper allows *arbitrary* (small) header size
  - `lh_header_size_mask = 0xFF    //` :-)
  - This is where the instance variables will go!

sharing: what the JVM should provide

- How to share?
- Java arrays are an uneasy fit in the JMM
- Key idea: JMM assumes ***serialized access***
  - Also multiple-readers of final values.
- **Requirement:** memory fences for arrays.
  - Explicit release and acquire for array slices.
  - Release-as-final for slices.
  - Release-as-volatile, acquire-as-volatile, probably.
  - (JMM experts, please correct this!)
- Key use case: Partition a work set w/o copying.
  - Fork, join, steal, repeat.

tags: what the JVM should provide

- heterogeneous containment?
- standard boxing is not cache-graceful
- fixnums? tagged unions? (working on this...)
- periodic-array-of-struct will reduce pressure for this
- but may still need ***non-periodic sequences***

flatness: what the JVM should provide

- flat data, *but please don't look behind the curtain*
  - assume flat storage, but there can be no proof
- trust JVM to provide as-if-flat performance (or better)
- low-pause technologies can help (“arraylets”)
- **Claim:** There is a *natural largest scale* for flat data.

More about library types

- in one step remove size & rank limits
- encapsulate complex layout algorithms
- **Requirement:** Value types for small structs / tuples
  - https://blogs.oracle.com/jrose/entry/value_types_in_the_vm
- We still need to build on nio, unsafe, etc.
- Hybrid arrays can be built from Java + MHs + unsafe
  - Existing `newInvokeSpecial` direct MH does this now.
  - Can build new ones that incorporate the hybrid “tail”.
- Non-periodic access can be done cheaply
  - Assuming encapsulated cursor values, with scalarization.
- Possible to build many patterns from few primitives.

What library types can define...

- Index width, number of indexes
- *Type* of indexes (associative arrays)
- Resize capability (how many indirections?)
- Periodicity (random vs. streaming access)
- Storage classes (final, volatile)
- Fencing / sharing / serialized access
- Compound or BLAS operations
- Super types
- Convenience methods
- Contiguity

More about operator overloading

- No, we won’t go and be C++!
- Java has to be “close to the metal”. No surprises.
- Even library-defined arrays have to avoid surprises
  - No excessive layers of indirection or cache accesses.
  - This is why we need hybrids.
- Probably just `getArrayElement`, `setArrayElement`
- slicing? (lvalues, index ranges): can do with libraries
  - work as much as possible within the existing language
  - (syntax, op spellings)
- for complex, application-specific notations: DSLs!
- residual language extension: allow multi-arg a[i,j,k]

Hard problem: Type templating

- Array appear to require a templating mechanism.
  - Type erasure won’t cut it, if you want primitives & structs.
- General-case reification not needed
  - Library classes can record “type dope” in ad hoc manner.
  - Cf. checked versions of list, set, etc.

Hard problem: Type templating (2)

- Need something like this:

```
template class BigSparseArray<template E>
    extends template ? super BigSparseArray<E>
{
    public E getArrayElement(long x) { ... }
}
```

- Type parameter(s) matches a “hole” in the body.
  - Holes are filled in by copying and pasting, in class loader.
  - Requires explicit value (struct) types to maintain sanity.

```
    value E getArrayElement(long x) { ... }
    ref   E getArrayElement(long x) { ... }
```

Power tools: JSR 292

- invokedynamic for truly new instructions (needed?)
- method handles for composing hybrid objects
  - new_hybrid_object + invokespecial (of regular constructor)
- maybe, method handles for privileged intrinsics
- New JVM pattern: Bootstrap method
  - Statically defined, lazily called method with static arguments.
- Possible BSM use cases
  - On-the-fly template instantiation (class loader calls BSM)
  - Use (with indy?) for defining the “holes” in a template.
  - Use for native method definition.
  - Useful for “intercessory” metaobject queries.

Killer Use Cases for Arrays 2.0

- fused (1-node) implementation of `java.lang.String`
  - | header | length | hash | body | → | header2 | char[] |
- cache-graceful impl. of `HashMap<Integer,String>`
  - Should be an array of struct { int, ref }.
  - 2-node representation. Envelope needed for resizing.
- cache-graceful B... trees (finally!)
- views on foreign data (N^2IO Direct{Data}Buffer)
- *(your contribution here...)*

Questions?

Questions?

(where’s the outrage?)

ORACLE®