

# ObjectLayout (and StructuredArray)

Closing the [last?] inherent C  
vs. Java speed gap

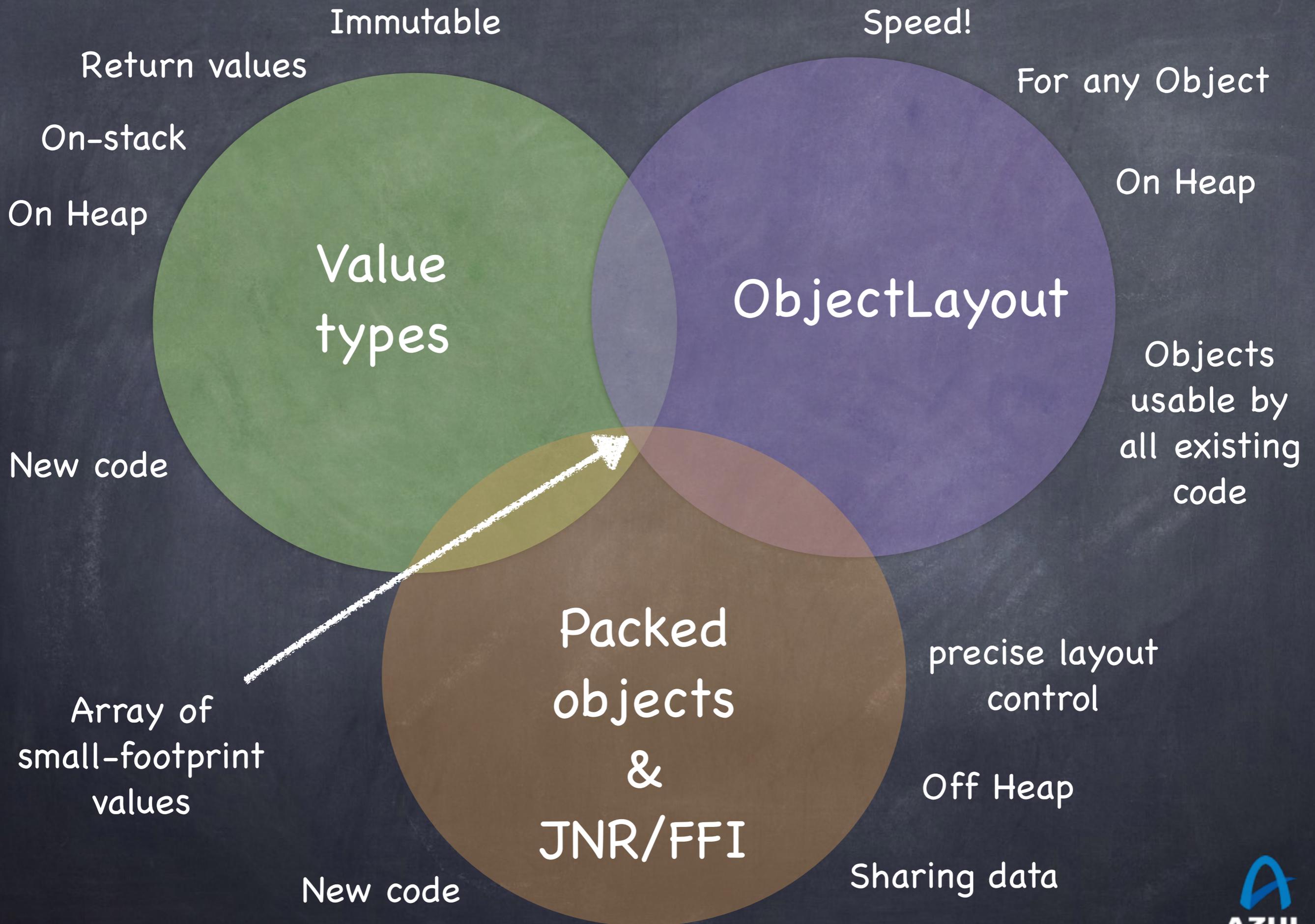
<http://objectlayout.org>

# ObjectLayout

- Focus: Match the raw speed benefits C based languages get from commonly used forms of memory layout
  - Expose these benefits to normal idiomatic POJO use
  - Focus: Speed. For regular Java Objects. On the heap.
  - Not looking for:
    - Improved footprint
    - off-heap solutions
    - immutability
- } These are all orthogonal concerns

# ObjectLayout: goal overlap?

- Relationship to Value types: none
- Relationship to Packet Objects (or JNR/FFI): none
- Laser-focused on a different problem
- Does not conflict or contradict concerns that drive these other efforts
- Minimal overlap does exist
- ... The kind of overlap that ArrayList and HashMap have as good alternatives of a bag of objects



# ObjectLayout Origin

- ObjectLayout/StructuredArray started with a simple argument. The common sides of the argument are:
- “We need structs in Java...”: Look at all the unsafe direct access stuff we do using flyweights over buffers or byte arrays, just to squeeze out speed that C gets trivially...
- “We already have structs. They are called Objects.”: What we need is competitively speedy access for the data collection semantics that are currently faster in C
- It’s all about capturing “enabling semantic limitations”

# speed comes from ???

- C's layout speed benefits are dominated by two factors:
- Dead reckoning:
  - Data address derived from containing object address
  - no data-dependent load operation
- Streaming (In the case of an array of structs):
  - sequential access through multiple members
  - predictable striding access in memory
  - Prefetch logic compensates for miss latency

# example of speed-enabling limitations

- Why is `Object[]` inherently slower than `struct foo[]`?
- Java: a mutable array of same-base-type objects
- C: An immutable array of exact-same-type structures
- Mutability (of the array itself) & non-uniform member size both force de-reference, and both break streaming
- `StructuredArray<T>`: An immutable array of [potentially] mutable exact-same-type (T) objects
  - Supports Instantiation, `get()`, but not `put()`...

# ObjectLayout target forms

- Easily identified C-based layout forms:
  - array of structs  
`struct foo[];`
  - struct with struct inside  
`struct foo { int a; struct bar b; int c; };`
  - struct with array at the end  
`struct packet { int length; char[] body; }`
- None are currently (speed) matched in Java

# ObjectLayout: starting point

- Capture the semantics that enable speed in the various C-like data layout forms behaviors
- Theory: we can do this all with no language change...
- Capture the needed semantics in “vanilla” Java classes (targeting e.g. Java SE 6)
- Have JDKs recognize and intrinsify behavior, optimizing memory layout and access operations
- “Vanilla” and “Intrinsified” implementation behavior should be indistinguishable (except for speed)

# Modeled after `java.util.concurrent`

- Captured semantics enabled fast concurrent operations
- No language changes. No required JVM changes.
- Implementable in “vanilla” Java classes outside of JDK
  - e.g. AtomicLong CAS could be done with synchronized
- JDKs improved to recognize and intrinsify behavior
  - e.g. AtomicLong CAS is a single x86 instruction
- Moved into JDK and Java name space in order to secure intrinsification and gain legitimate access to unsafe

# ObjectLayout: StructuredArray

- array of structs  
struct foo[];

- struct with struct inside

```
struct foo { int a; struct bar b; int c; };
```

- struct with array at the end

```
struct packet { int len; char[] body; }
```

# StructuredArray<T>

- A collection of object instances of arbitrary type T
- Arranged as array: T element = get(index);
- Collection is immutable: cannot replace elements
- Instantiated via factory method:  

```
a = StructuredArray.newInstance(SomeClass.class, 100);
```
- All elements constructed at instantiation time
- Supports arbitrary constructor and args for members
  - Including support for index-specific CtorAndArgs

# StructuredArray<T> liveness

- We considered an “inner pointer keeps container alive” approach, because that’s what other runtimes seem to do with arrays of structs and field references
  - But then we realized: real objects have real liveness
  - A StructuredArray is just a regular idiomatic collection
    - The collection keeps its members alive
    - Collection members don’t (implicitly) keep it alive
- \*\*\* Under the hood, optimized implementations will want to keep the collection “together” as long as it is alive

# Benefits of liveness approach

- StructuredArray is just a collection of objects
  - No special behavior: acts like any other collection
  - Happens to be fast on JDKs that optimize it
- Elements of a StructuredArray are regular objects
  - Can participate in other collections and object graphs
  - Can be locked
  - Can have an identity hashCode
  - Can be passed along to any existing java code
- It's "natural", and it's easier to support in the JVM

# StructuredArray<T> continued...

- Indexes are longs (it's 2014...)
- Multi dimensional support
  - Non-leaf Elements are themselves StructuredArrays
- StructuredArray is subclassable
  - Supports some useful coding styles and optimizations
- StructuredArray is not constructable
  - must be created with factory methods

(\*\*\* Did you spot that small contradiction?)

# Optimized JDK implementation

- A new heap concept: “contained” and “container” objects
  - Contained and container objects are regular objects
  - Given a contained object, there is a means of finding the immediately containing object
  - If GC needs to move an object that is contained in a live container object, it will move the entire container
- Very simple to implement in all current OpenJDK GC mechanisms (and in Zing’s C4, and in others, we think)
  - More details in workshop...

# Optimized JDK implementation

- Streaming benefits come directly from layout
  - No compiler optimizations needed
- Dead-reckoning benefits require some compiler support
  - no dereferencing, but....
  - $e = (T) ( a + a.bodySize + (index * a.elementSize) );$
  - elementSize and bodySize are not constant
  - But optimizations similar to CHA & inline-cache apply
  - More details in workshop...

# ObjectLayout forms 2 & 3

- array of structs

```
struct foo[];
```

- struct with struct inside

```
struct foo { int a; struct bar b; int c; };
```

- struct with array at the end

```
struct packet { int len; char[] body; }
```

# “struct in struct”: intrinsic objects

- Object instance  $x$  is intrinsic to object instance  $y$
- Proposal: factory based instantiation of final member objects

```
Class SomeClass {  
    int a;  
    final SomeOtherClass o =  
        IntrinsifiedObjects.createIntrinsicObject(SomeOtherClass.class);  
}
```

- Intrinsic object can be laid out within containing object
- Must deal with & survive reflection based overwrites

# “struct with array at the end”: subclassable arrays

- Semantics well captured by subclassable arrays classes
- ObjectLayout describes one for each primitive type. E.g. PrimitiveLongArray, PrimitiveDoubleArray, etc...
- Also ReferenceArray<T>
- StructuredArray<T> is also subclassable, and captures “struct with array of structs at the end”

# Status

- Vanilla Java code on github. Public domain under CC0. See <http://objectlayout.org>
- StructuredArray is fairly mature semantically. Other two forms (intrinsic objects and subclassable arrays) are getting there...
- Intrinsic implementations coming over the next few months for both Zing and OpenJDK
- Next steps: OpenJDK project with working code, JEP...
- Aim: Add ObjectLayout to Java SE (9?)
  - Vanilla implementation will work on all JDKs

# ObjectLayout Summary

- New Java classes: `org.ObjectLayout.*`
  - Propose to move into `java` namespace in Java SE (9?)
- Work “out of the box” on Java 6, 7, 8, 9, ...
  - No syntax changes, No new bytecodes
  - No new required JVM behavior
- Can “go fast” on JDKs that optimize for them
  - Relatively simple, isolated JVM changes needed
  - Proposing to include “go fast” in OpenJDK (9?)
  - Zing will support “go fast” for Java 6, 7, 8, 9, ...

# Q & A

@giltene

<http://www.azulsystems.com>

<http://objectlayout.org>

<https://github.com/ObjectLayout/ObjectLayout>