Bringing The Performance of Structs To Java (Sort Of)

© Copyright Azul Systems 2015

Simon Ritter
Deputy CTO, Azul Systems
@speakjava | azul.com

© Copyright Azul Systems 2016

ObjectLayout Project Focus

- Match the speed benefits that C-based languages get from commonly used forms of memory layout
  - Expose these benefits to normal, idiomatic POJO usage
- *Speed. For regular Java objects. On the heap*

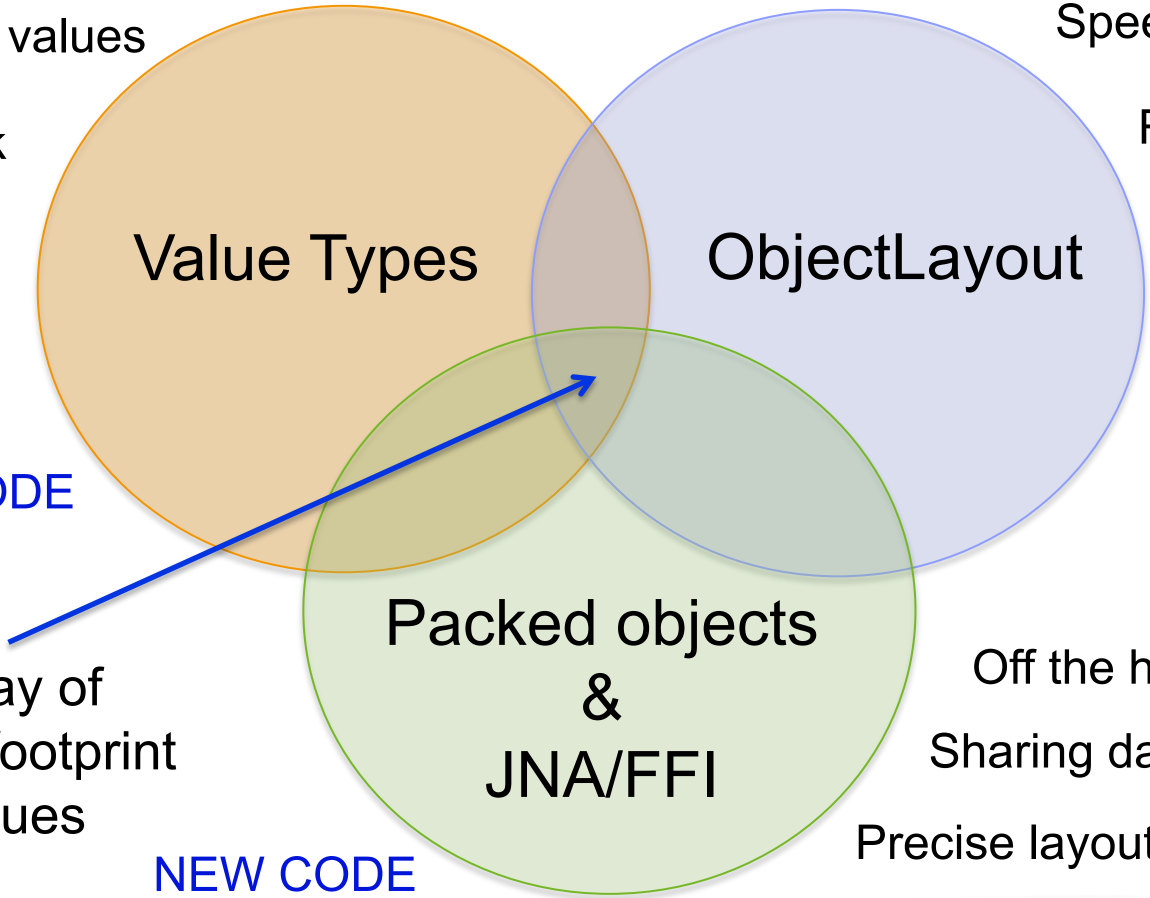
- What this is not looking at:
  - Improved footprint
  - Off-heap solutions
  - Immutability

© Copyright Azul Systems 2016

Goal Overlap For ObjectLayout

- Relationship to value types: None
- Relationship to packed objects (JNR/FFI): None
- ObjectLayout is focused on a different problem
- Minimal overlap does exist
  - In the same way that `ArrayList` and `HashMap` overlap as containers for groups of objects

© Copyright Azul Systems 2016



Immutable

Return values

On stack

On heap

Value Types

ObjectLayout

Speed!

Regular Java objects

On the heap

`Objects` USABLE BY ALL EXISTING CODE

NEW CODE

Packed objects
&
JNA/FFI

Array of
small-footprint
values

NEW CODE

Off the heap

Sharing data

Precise layout control

© Copyright Azul Systems 2016

ObjectLayout Origin

- ObjectLayout started with a simple argument:
  - “We need structs in Java…”
    - People (mis-?)use sun.misc.Unsafe to try and replicate structs
    - C and C++ get this for free
  - “We already have structs. They are called Objects.”
    - We need competitive access speed to structs in C/C++
  - It’s all about capturing “enabling semantic limitations”

© Copyright Azul Systems 2016

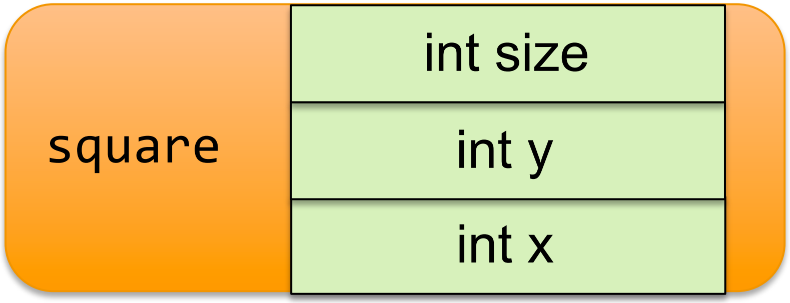
Where speed comes from

- C layout speed benefits are dominated by two factors:
  - Dead reckoning
  - Streaming for arrays of structs

© Copyright Azul Systems 2016

Data Grouping In C

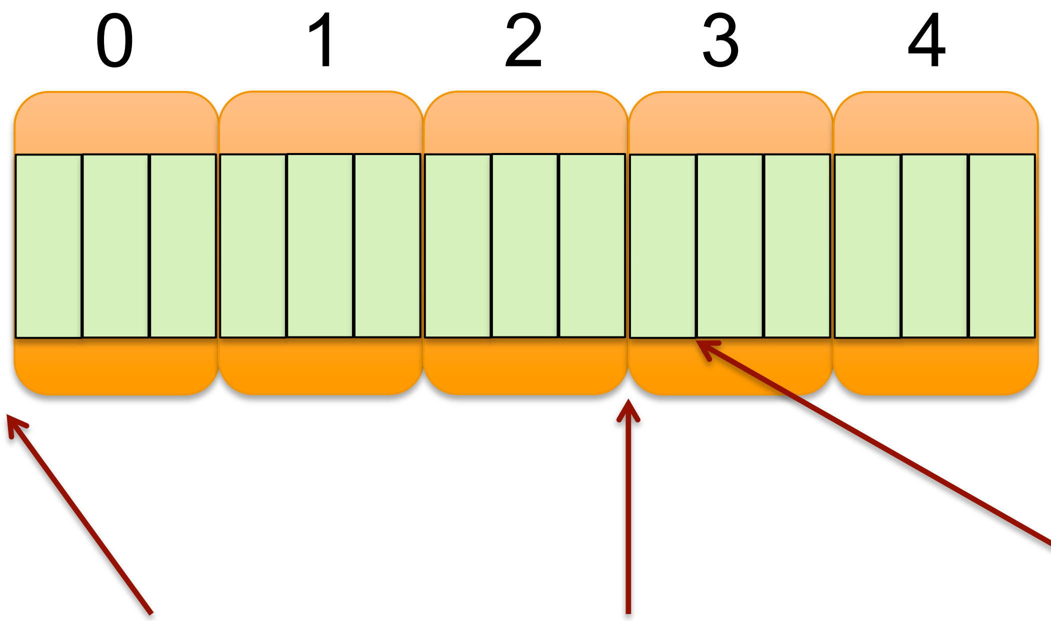
```
struct square {
  int x;
  int y;
  int size;
};

...

struct square s;
```



© Copyright Azul Systems 2016

Dead Reckoning In C

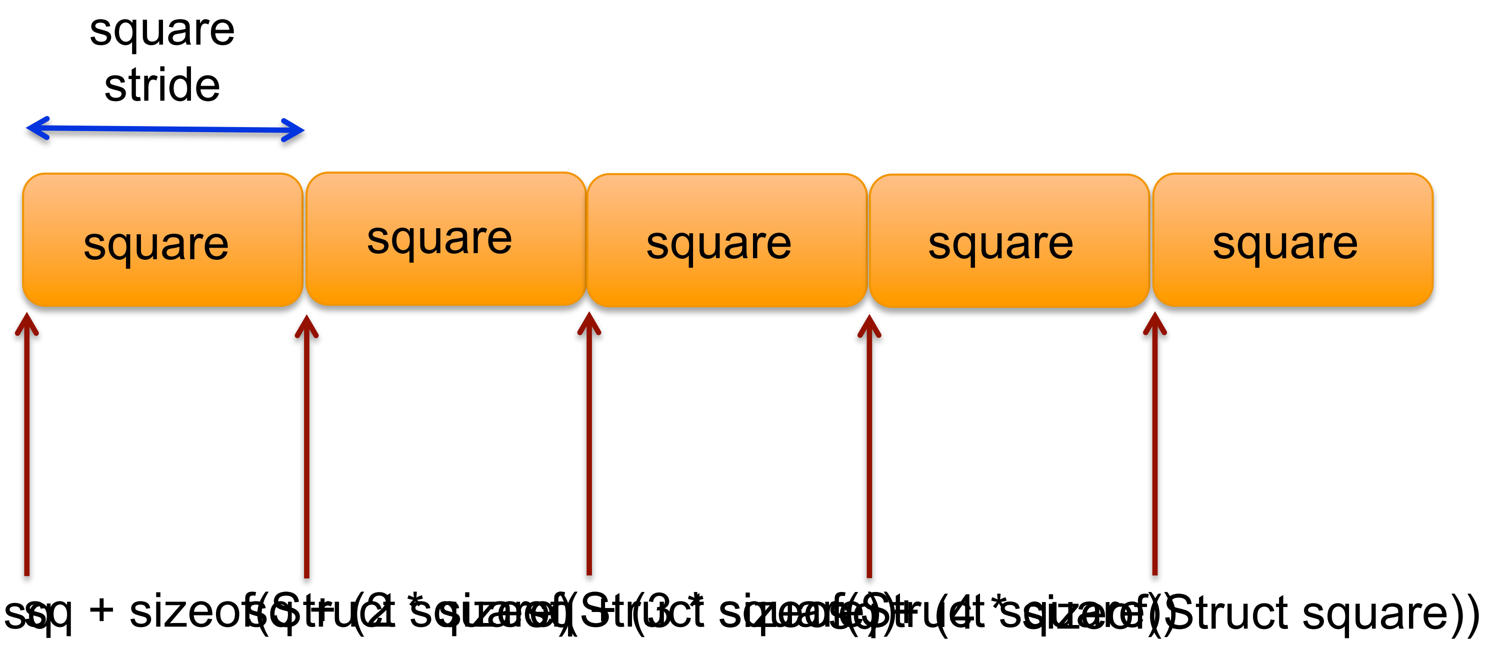
```
struct *sq = malloc(sizeof(Struct square)*5);
```



```
&sq[3].y = sq + (3*sizeof(Struct square)) + sizeof(int);
```

© Copyright Azul Systems 2016

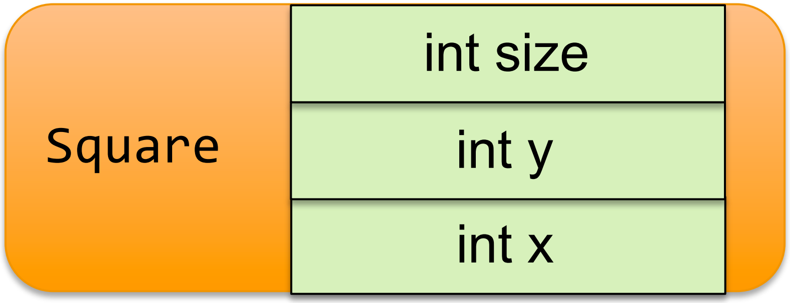
Streaming Arrays In C


square
stride
square
square
square
square
square
sq
sq + sizeof(struct square)
sq + (2 * sizeof(struct square))
sq + (3 * sizeof(struct square))
sq + (4 * sizeof(struct square))

© Copyright Azul Systems 2016

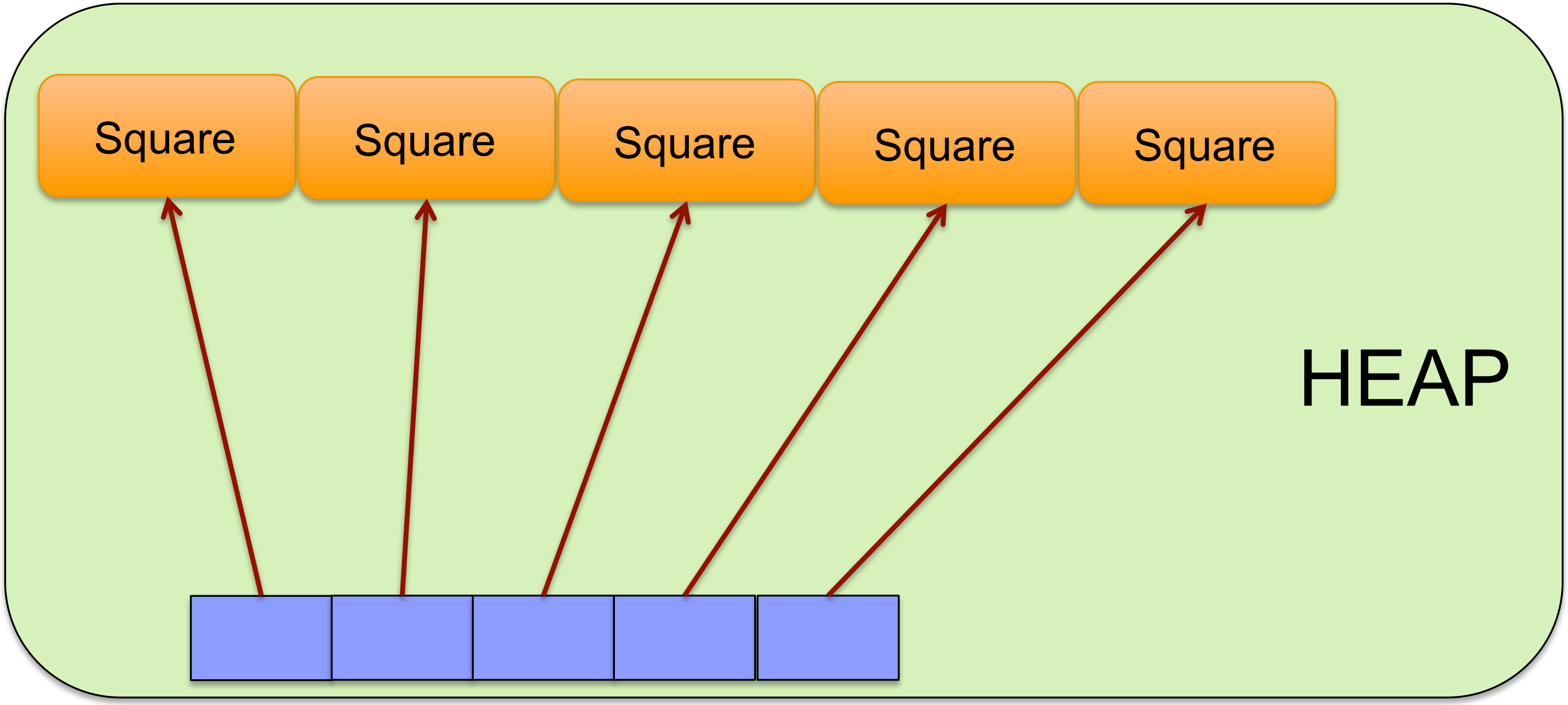
Data Grouping In Java

```
Class Square {
  int x;
  int y;
  int size;
};

...

Square square = new Square();
```



© Copyright Azul Systems 2016

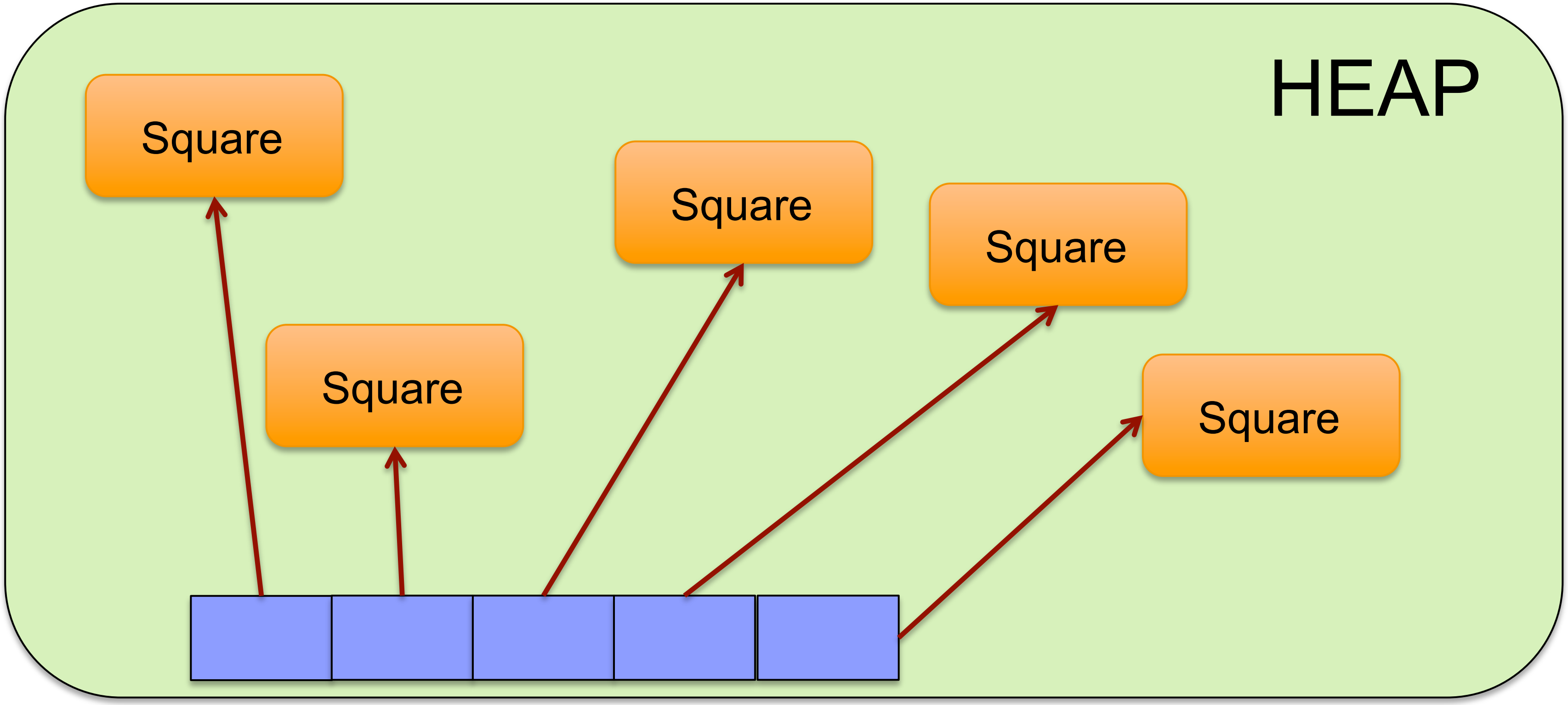
Arrays In Java



```
Square[] squares = new Square[5];
```

© Copyright Azul Systems 2016

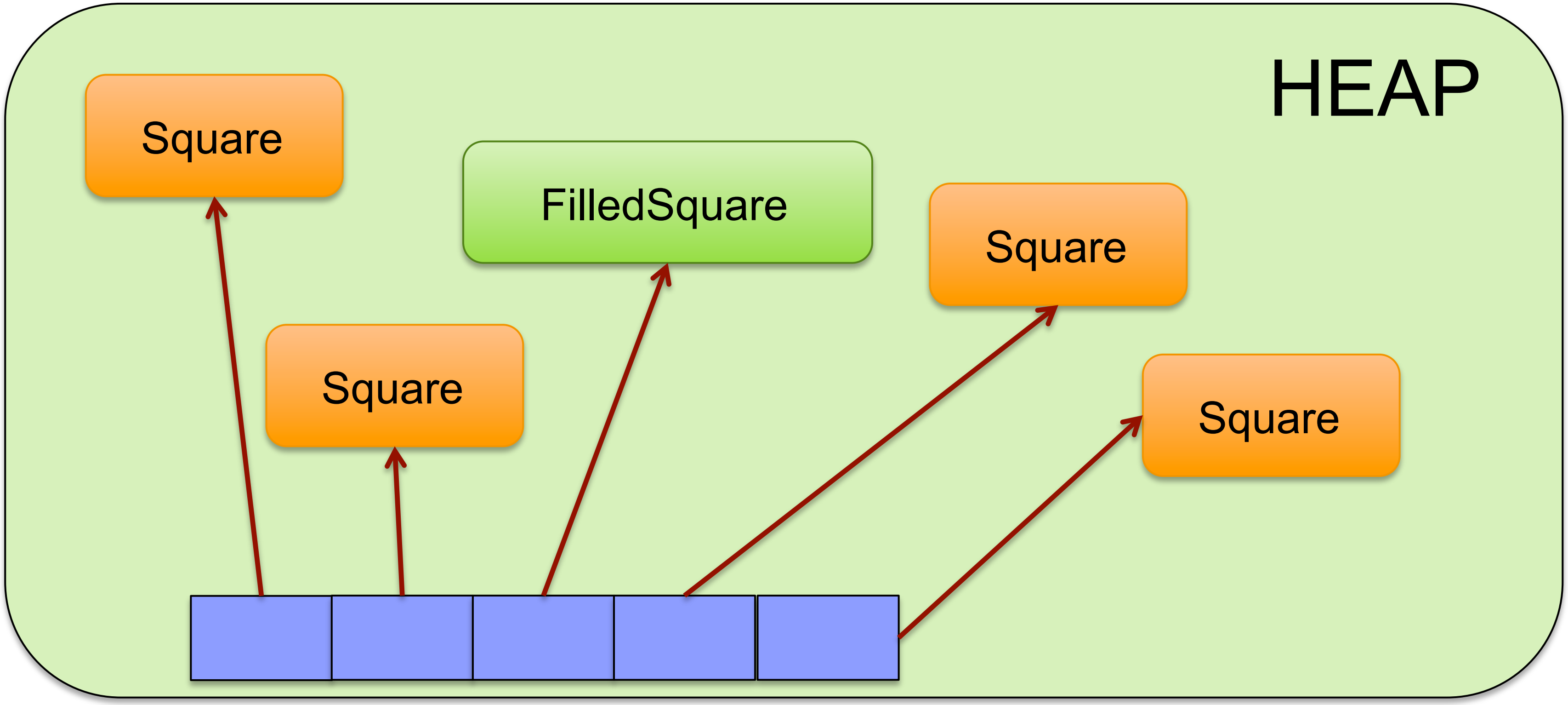
Arrays In Java



```
System.gc();
```

© Copyright Azul Systems 2016

Arrays In Java



```
square[2] = new FilledSquare();
```

© Copyright Azul Systems 2016

Array Semantics: Structs v. Objects

- C
  - An immutable array of exact same size structures
- Java
  - A mutable array of same base type objects
  - Can change the object reference of an array element
  - `squares[]` could hold `Square` or `FilledSquare` objects
    - No guarantee `Square` and `FilledSquare` are the same size

© Copyright Azul Systems 2016

org.ObjectLayout Target Forms

- The common C-style constructs we seek to match:
  - 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; }
    ```

- All of these can be expressed in Java
- None are currently (speed) matched in Java

© Copyright Azul Systems 2016

Modeled On 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

© Copyright Azul Systems 2016

ObjectLayout Starting Point

- Capture the semantics that enable speed in the various C-like data layout forms and behaviors
  - Theory: without any changes to the language
- Capture the needed semantics in “vanilla” Java classes (targeting e.g. Java SE 7)
- Have JDK/JVM recognize and intrinsify behavior, optimizing memory layout and access operations
  - “Vanilla” and “Intrinsified” implementation behavior should be indistinguishable (except for speed)

© Copyright Azul Systems 2016

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; }
```

© Copyright Azul Systems 2016

StructuredArray<T>

- A collection of object instances of arbitrary (exact) type T
  - Captures semantic limitations equivalent to C struct[]
- Arranged like an array: `T element = get(index);`
- Collection is immutable: cannot replace elements
  – Has get(), but no put()

© Copyright Azul Systems 2016

StructuredArray<T>

- 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`

© Copyright Azul Systems 2016

Context-Based Construction

```
public class Foo {
  private final long index;

  public Foo(long index) {
    this.index = index;
  }
  ...
}
```

© Copyright Azul Systems 2016

Context-Based Construction

```
final Constructor<Foo> constructor =
  Foo.class.getConstructor(Long.TYPE);

final StructuredArray<Foo> fooArray =
  StructuredArray.newInstance(Foo.class,
     context ->
       new CtorAndArgs<Foo>(constructor, context.getIndex()),
     8);
```

© Copyright Azul Systems 2016

StructuredArray Liveness

- Initial approach was:
  - Reference to element keeps the `StructuredArray` alive
  - This is what happens on other runtimes
- However, element Objects have real liveness
  - Already tracked by the JVM
- A `StructuredArray` is just a regular idiomatic collection
  - The collection keeps it's members alive
  - Collection members don't (implicitly) keep it alive

© Copyright Azul Systems 2016

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

© Copyright Azul Systems 2016

StructuredArray Features

- Indexes are longs
- Nested arrays are supported (multi-dimension, composable)
  - 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 the Catch-22?

© Copyright Azul Systems 2016

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 for the JVM to find 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 on github and in project discussion

© Copyright Azul Systems 2016

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 project discussion

© Copyright Azul Systems 2016

ObjectLayout

- array of structs

```
      struct foo[];
```

- struct with struct inside

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

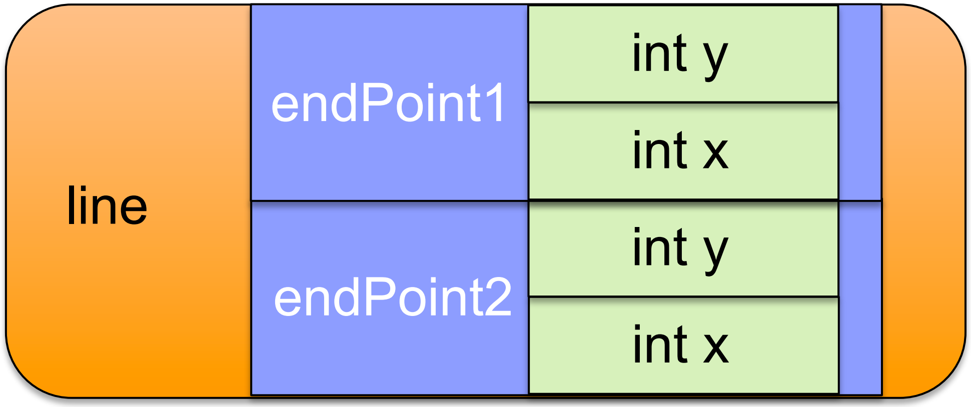
- struct with array at the end

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

© Copyright Azul Systems 2016

Encapsulated Struct In C

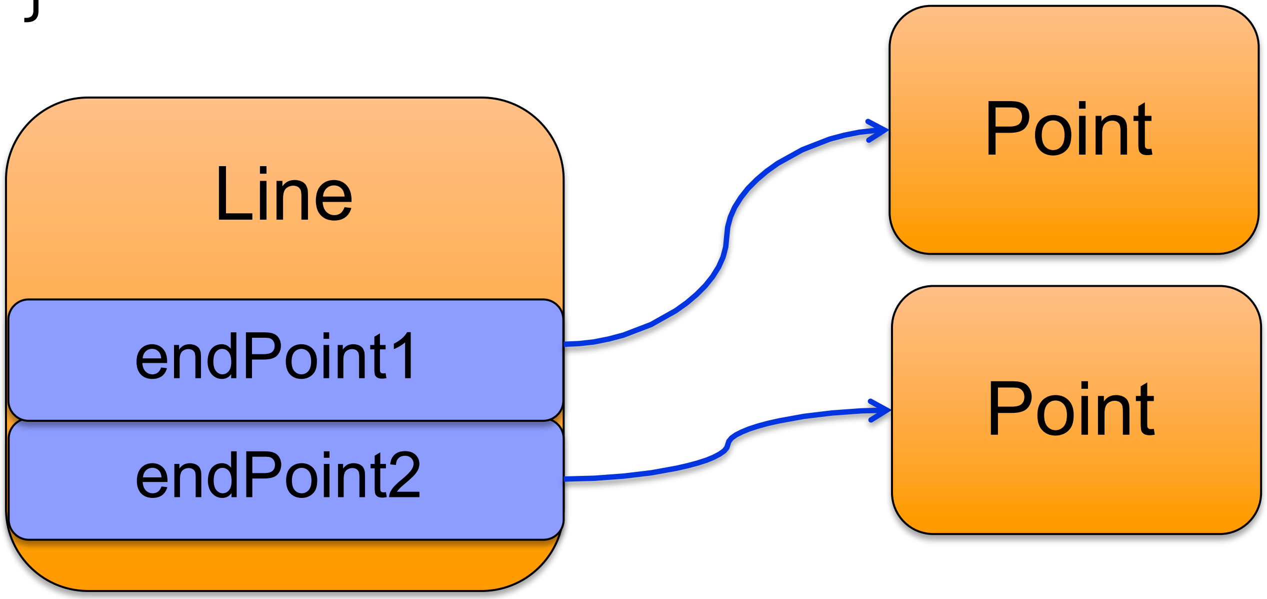
```
struct line {
  struct point endPoint1;
  struct point endPoint2;
};

...

struct line l;
```



© Copyright Azul Systems 2016

Struct-In-Struct Intrinsic Objects

```
Class Line {
  private final Point endPoint1= new Point();
  private final Point endPoint2 = new Point();
}
```



© Copyright Azul Systems 2016

Struct-In-Struct Intrinisic Objects

- Intrinsic objects can be laid out within containing object

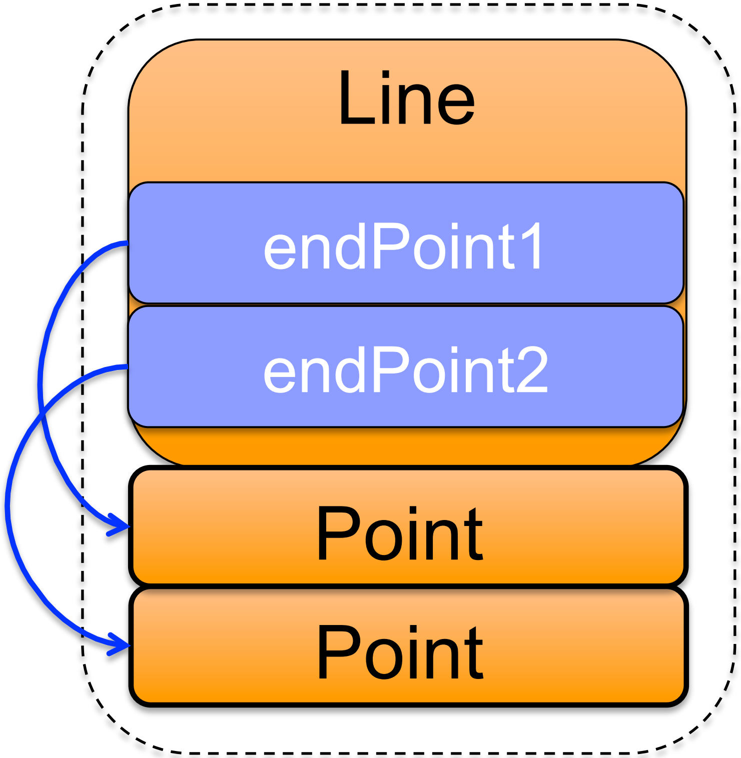
```
Class Line {
  private static final Lookup lookup =
      MethodHandles.lookup();

  @Intrinsic
  private final Point endPoint1 = IntrinsicObjects
      .constructWithin(lookup, “endPoint1”, this);
  ...
}
```

Struct-In-Struct Intrinsic Objects

- JVM makes the ‘Struct’ intrinsic to the enclosing object
  - Dead-reckoning can be used to determine address
- Java code sees no change (still an implicit pointer)
- Must deal with and survive reflection based overwrites

© Copyright Azul Systems 2016

Struct-in-Struct Virtual Object



- Three separate objects
- VM treats them as one from GC perspective
- Contiguous in memory
  - Moved as a unit

© Copyright Azul Systems 2016

Struct With Array At The End

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

© Copyright Azul Systems 2016

ObjectLayout Forms Are Composable

Heap

StructuredArray<StructuredArray<Foo>>

StructuredArray<Foo>

Foo

(@Intrisic)Bar (@Intrisic)Baz

(@Intrinsic length=4)StructuredArray<Moo>

© Copyright Azul Systems 2016

Status

- Vanilla Java code on github: www.objectlayout.org
- Fairly mature semantically
  - Working out “spelling”
- Intrinsified implementations coming for OpenJDK and Zing
- Early numbers look good
  - E.g. faster `HashMap.get()`
- Next steps: OpenJDK project with working code, JEP…
- Aim: Add ObjectLayout to Java SE (10?)
  - Vanilla implementation will work on all JDKs

© Copyright Azul Systems 2016

Summary

- New Java classes: org.ObjectLayout.*
  - Propose to move into java namespace in Java SE (10?)
- Works “out of the box” on Java 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 (10?)
  - Zing will support “go fast” for Java 7, 8, 9, 10…

© Copyright Azul Systems 2016

Q & A

© Copyright Azul Systems 2015

Simon Ritter
Deputy CTO, Azul Systems

@speakjava | azul.com

© Copyright Azul Systems 2016