The future is native
How Java is learning to interoperate with native data

Dan Heidinga, J9 VM Interpreter Lead
Daniel_Heidinga@ca.ibm.com
@DanHeidinga
10 August 2015
Important disclaimers

- THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.
- WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.
- ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.
- ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.
- IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.
- IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.
- NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:
  - CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS
Who am I?

- I've been involved with virtual machine development at IBM since 2007 and am now the J9 Virtual Machine Team Lead. J9 is IBM's independent implementation of the JVM.

- I've represented IBM on both the JSR 292 (‘invokedynamic’) and JSR 335 (‘lambda’) expert groups and lead J9's implementation of both JSRs.

- I've also maintain the bytecode verifier and deal with various other parts of the runtime.
Get involved now
Overview of PackedObjects

From Packed to Panama & Valhalla

Layouts

Open Issues
PackedObjects

- PackedObjects is an experimental feature in IBM J9 Virtual Machine.
  - Introduced in Java 7.1

Goals of Feature

- Improve serialization and I/O of Java objects
- Reduce overhead of interoperating with non-Java data
- Allow direct access to off-heap data
- Improve memory utilization
- Improve data locality
- Provide natural programming idioms for interoperating with non-Java data
What are we trying to solve?

Simple enough…
What are we trying to solve?

Simple enough…

- Header overhead
What are we trying to solve?

Simple enough…

- Header overhead
- Pointer chasing
What are we trying to solve?

Simple enough…

- Header overhead
- Pointer chasing
- Locality
What are we trying to solve?

- Locality problems come in other flavours
- Consider a class with a dozen fields, but only two get used regularly
  - Declare them together, and they’ll be stored together, right?
  - Well, maybe…
- Within a class the JVM may reorder fields
  - Ensure 64-bit alignment of \texttt{long} / \texttt{double} fields
  - Move references together for GC efficiency
- This can even move fields into "holes" left by a superclass

```java
class A {
    int i1;
    double d1;
}
class B extends A {
    double d2;
    int i2;
}
```
What are we trying to solve?

Fighting the Java/Native interface
Ok so we have some criteria…

- Ability to do away with headers
- Ability to bring multiple objects close together
- On heap / off heap seamless referencing of data

- This actually sounds a lot like C structure types

```c
struct Address {
    char[4] addr;
    short port;
}

struct Header {
    struct Address src;
    struct Address dst;
}
```

- Packed Objects!
Packed Objects: Under the covers

```
int x
int y
```
Packed Objects: Under the covers

- aPoint
  - int x
  - int y

- aPackedPoint
  - target
  - offset
  - int x
  - int y
Packed Objects: Under the covers

```
int x
int y
```

```
Object header
Object field / data
```

```
target
offset
int x
int y
```
Packed Objects: In Practice

int y
int x

Point s
Point e

aLine

aPoint

int x
int y

Object header

Object field / data
Packed Objects: In Practice

- **Object header**
- **Object field / data**
Packed Objects: In Practice

```
int y
int x
aPoint
Object header
Object field / data

int y
int x
Point s
Point e
aLine

int x
int y
aPoint

int x
int y
aPoint

int x
int y
aPackedPoint s

int x
int y
int x
int y
aPackedLine

target
offset
aPackedPoint e
```
Packed Objects: In Practice

```java
@Packed
final class PackedPoint extends PackedObject {
    int x;
    int y;
}

@Packed
final class PackedLine extends PackedObject {
    PackedPoint s;
    PackedPoint e;
}
```
Packed Objects: In Practice

```
int y
int x
```

```
aPoint
Object header
Object field / data
```

```
aLine
Point s
Point e
```

```
aPoint
int x
int y
```

```
aPackedLine
target
offset
int x
int y
int x
int y
```

- Object header
- Object field / data
Packed Objects: In Practice

```
int y
int x
aPoint
Point s
Point e
aLine
int x
int y
aPoint
int x
int y
aPackedLine
offset
int x
int y
int x
int y
aPackedLine.e
```
Packed Objects: In Practice

```
int y
int x
```

```
aPoint
```

```
Point s
Point e
```

```
aLine
```

```
Object header
```

```
Object field / data
```

```
aPackedLine
```

```
offset
```

```
target
```

```
aPackedPoint
```

```
offset
```

```
target
```

```
aPackedLine.e
```
Packed Objects: In Practice with Native Access

Java

Native

struct Point {
    int x;
    int y;
}

int x
int y

Object header

Struct field / data
Packed Objects: In Practice with Native Access

```java
@Packed
final class PackedPoint extends PackedObject {
    int x;
    int y;
}
```
@Packed
final class PackedPoint extends PackedObject {
    int x;
    int y;
}

struct Point {
    int x;
    int y;
}
struct Address {
    char[4] addr;
    short port;
}
struct Header {
    struct Address src;
    struct Address dst;
}

- Nested substructures
- Compact representation
- Alignment aspects
Let’s Build the Same “Something” in Java!

```java
class Address {
    byte[] addr;
    short port;
}

class Header {
    Address src;
    Address dst;
}
```

- Headers
- No locality
- Alignment
What if we did this with Packed Objects?

@Packed
final class Address extends PackedObject {
    PackedByte[[4]] addr;
    short port;
}

@Packed
final class PacketHeader extends PackedObject {
    Address src;
    Address dest;
}

- The Java code is pretty clean… and a pretty good result!
What about native access?

How do we implement this normally?
JNI implementation

```java
public class PackedHeader {
    private long pointer;

    public byte[] getSourceAddress() { return getSourceAddressImpl(pointer); }
    public short getSourcePort() { return getSourcePortImpl(pointer); }
}

JNICALL jshort Java_pkg_PackedHeader_getSourcePort(JNIEnv* env, jobject recv, jlong pointer) {
    struct PacketHeader* header = (struct PacketHeader*)pointer;
    return (jshort)header->src.port;
}

JNICALL jbyteArray Java_pkg_PackedHeader_getSourceAddress(JNIEnv* env, jobject recv, jlong pointer) {
    struct PacketHeader* header = (struct PacketHeader*)pointer;
    jbyteArray result = (*env)->NewByteArray(env, 4);
    (*env)->SetByteArrayRegion(env, result, 0, 4, &header->src.addr);
    return result;
}
```

- Usual “stash pointers in long types” tricks
- JNI costs tend to be high
DirectByteBuffer implementation

class PackedHeader {
    private ByteBuffer buffer;
    private static final int SRC_ADDR_OFFSET = 0;
    private static final int SRC_PORT_OFFSET = 4;
    private static final int DEST_ADDR_OFFSET = 8;
    private static final int DEST_PORT_OFFSET = 12;

    public short getSourcePort() { return buffer.getShort(SRC_PORT_OFFSET); }
    public byte[] getSourceAddress() {
        byte[] result = new byte[4];
        buffer.get(result, SRC_ADDR_OFFSET, 4);
        return result;
    }
}

- No extra JNI to write (this is good)
- Keeping your indices straight is never fun
Unsafe implementation

```java
class PackedHeader {
    private Unsafe unsafe;
    private long pointer;
    private static final int SRC_ADDR_OFFSET = 0;
    private static final int SRC_PORT_OFFSET = 4;
    private static final int DEST_ADDR_OFFSET = 8;
    private static final int DEST_PORT_OFFSET = 12;

    public short getSourcePort() { return unsafe.getShort(pointer + SRC_PORT_OFFSET); }
    public byte[] getSourceAddress() {
        byte[] result = new byte[4];
        unsafe.copyMemory(null, pointer + SRC_ADDR_OFFSET, result, 0, 4);
        return result;
    }
}
```

- You shouldn’t be here
- Still playing the indices game
**PackedObject answer**

```java
final class PacketHeader extends PackedObject {
    Address src;
    Address dest;

    public short getSourcePort() { return src.port; }
    public PackedByte[] getSourceAddress() { return src.addr; }
}
```

- Looks like natural Java code
- Foregoes JNI
- Same type capable of on-heap representation
Design Challenges

- Annotations (@ImportPacked), packed array classes are cumbersome
- All packed types must descend from PackedObject class
- Primitive arrays, e.g. byte[], are incompatible with packed arrays
- Incompatibility with conventional Java
  - Nested field semantics
  - Identity semantics – implications for synchronization, finalization
Split Hierarchy

Object

PackedObject

abstract or packed subclass of PackedObject

P-Array of any packed class

Object[]

PackedObject[]

Array of any packed class
array[1] = new Point3D();
Point3D point1 = array[1];

if (point1 == array[1]) {
    /* dead code */
}

Packed Arrays
@Packed
final class Point extends PackedObject {
    int x;
    int y;

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

@Packed
final class Line extends PackedObject {
    Point start;
    Point end;

    Line(int sx, int sy, int ex, int ey) {
        start = new Point(sx, sy);
        end = new Point(ex, ey);
    }
}

Line line = new Line(0, 0, 100, 200);

"=" doesn't work because start and end aren't reference fields.

Implicitly instantiates start & end, but cannot invoke Point(x, y) constructor.
Nested Field Initialization

```java
@Packed
final class Point extends PackedObject {
    int x;
    int y;

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

@Packed
final class Line extends PackedObject {
    Point start;
    Point end;

    Line(int sx, int sy, int ex, int ey) {
        start.x = sx; start.y = sy;
        end.x = ex; end.y = ey;
    }
}

Line line = new Line(0, 0, 100, 200);
```

start & end's contents must be explicitly initialized.
Final Nested Fields

@Packed
final class Line extends PackedObject {
    final Point start;
    final Point end;

    Line(int sx, int sy, int ex, int ey) {
        start.x = sx; start.y = sy;
        end.x = ex; end.y = ey;
    }
}

Does not compile because start & end are not initialized using "+="
Volatile Fields

```java
@Packed
class Point extends PackedObject {
    int x;
    volatile int y;
}

@Packed
class Line extends PackedObject {
    Point start;
    Point end;

    Line(int sx, int sy, int ex, int ey) {
        start.x = sx; start.y = sy;
        end.x = ex; end.y = ey;
    }
}

Line line1 = new Line(0, 0, 10, 0);
Line line2 = new Line(0, 0, -10, 0);

line1.copyFrom(line2);

Point e = line1.end;
e.copyFrom(new Point(1, 1));
```

Multiple ways to change the value of $y$ without triggering volatility checks
(aPackedLine != anotherPackedLine), but they refer to the same data.
Synchronization

Thread T1:
synchronized (aPackedLine) {
    aPackedLine.end.x = 10;
}

Thread T2:
synchronized (anotherPackedLine) {
    anotherPackedLine.end.x = 13;
}

The data location is not protected against simultaneous update because there are two monitors for the same location.
Finalization

- Can pt0 be finalized when its data is still referenced by pt1 and polyline?
Weak References

- Can WeakReference(pt0) be enqueued if pt1 is still strongly reachable?
Overview of PackedObjects

From Packed to Panama & Valhalla

Layouts

Open Issues
PackedObjects

- PackedObject is an experimental feature in IBM J9 Virtual Machine.

Goals of Feature

- Improve serialization and I/O of Java objects
- Reduce overhead of interoperating with non-Java data
- Allow direct access to off-heap data
- Improve memory utilization
- Improve data locality
- Provide natural programming idioms for interoperating with non-Java data
PackedObjects

- PackedObject is an experimental feature in IBM J9 Virtual Machine.

Goals of Feature

- Improve serialization and I/O of Java objects
- Reduce overhead of interoperating with non-Java data
- Allow direct access to off-heap data
- **Improve memory utilization**
- **Improve data locality**
- Provide natural programming idioms for interoperating with non-Java data
PackedObjects

- PackedObject is an experimental feature in IBM J9 Virtual Machine.

Goals of Feature

- Improve serialization and I/O of Java objects
- Reduce overhead of interoperating with non-Java data
- Allow direct access to off-heap data
- Improve memory utilization
- Improve data locality
- Provide natural programming idioms for interoperating with non-Java data
Others

PackedObjects

Others

Valhalla

Panama
Overview of PackedObjects
From Packed to Panama & Valhalla

Layouts

Open Issues
State of the Layout Descriptor Language

http://danheidinga.github.io/J9-Panama/StateOfTheLDL.html
What are we trying to solve?

- So, what’s the problem we’re trying to solve?
  - You’re implementing a language runtime and want to use that great C library
  - You need to talk to the OS doing something that isn’t standard Java
  - You own some legacy system and want to interop with its data (COBOL)
  - You’ve got access to some awesome h/w and want to drive it from Java
    - RDMA
    - GPU
Offsets game: that’s so yesterday

We need a better way.
Native data

- C / C++
- COBOL
- Network packets
- DWARF
- Key value stores

- POSIX structures
- ProtoBuf
- Database schema
- etc
Native data

- 32 vs 64 bit?
- `#pragma packed`
- Big vs little endian
- Defined endian
- Bit fields

- Compiler
- Padding
- OS
- Library versions
Contract to program against

Native Data → Stub Interface

Layout Descriptor Language

interface Foo extends Layout
{
    ...
}

Interfaces for the win

interface Foo
extends Layout
{
    ....
}

class UserFoo
implements Foo
{
    ....
}

class Foo$Impl
implements Foo
{
    ....
}
Runtime system

```
Runtime system

Native Data  Stub Interface  Runtime behaviour

LayoutFactory f = LayoutFactory.getFactory();
Unsafe unsafe = Unsafe.getUnsafe();

long addr = unsafe.allocateMemory(Point.sizeof());
Point aPoint = f.mapLayout(Point.class, addr);
aPoint.x(11);
aPoint.y(32);
```
<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ByteArray1D</code></td>
<td><code>fromByteBuffer(java.nio.ByteBuffer buffer)</code> Get a layout to access a direct byte buffer.</td>
</tr>
<tr>
<td><code>&lt;T extends Layout&gt; Array1D&lt;T&gt;</code></td>
<td><code>getArray1D(java.lang.Class&lt;T&gt; cls, long length)</code> Allocate Java heap memory, and overlay a 1D array layout on it.</td>
</tr>
<tr>
<td><code>&lt;T extends Layout&gt; Array2D&lt;T&gt;</code></td>
<td><code>getArray2D(java.lang.Class&lt;T&gt; cls, long dim1, long dim2)</code> Allocate Java heap memory, and overlay a 2D array layout on it.</td>
</tr>
<tr>
<td><code>static LayoutFactory</code></td>
<td><code>getFactory()</code> Access the layout factory.</td>
</tr>
<tr>
<td><code>&lt;T extends Layout&gt; T</code></td>
<td><code>getLayout(java.lang.Class&lt;T&gt; cls)</code> Allocate Java heap memory, and overlay a singleton layout on it.</td>
</tr>
<tr>
<td><code>&lt;T extends LayoutType&gt; T</code></td>
<td><code>getPrimArray1D(java.lang.Class&lt;? super T&gt; primCls, long length)</code> Allocate Java heap memory, and overlay a 1D array layout on it.</td>
</tr>
<tr>
<td><code>&lt;T extends LayoutType&gt; T</code></td>
<td><code>getPrimArray2D(java.lang.Class&lt;? super T&gt; primCls, long dim1, long dim2)</code> Allocate Java heap memory, and overlay a 2D array layout on it.</td>
</tr>
<tr>
<td><code>&lt;T extends Layout&gt; Array1D&lt;T&gt;</code></td>
<td><code>mapArray1D(java.lang.Class&lt;T&gt; cls, byte[] data, long length)</code> Overlay a 1D array layout on a byte[].</td>
</tr>
<tr>
<td><code>&lt;T extends Layout&gt; Array1D&lt;T&gt;</code></td>
<td><code>mapArray1D(java.lang.Class&lt;T&gt; cls, long addr, long length)</code> Overlay a 1D array layout on a given memory address.</td>
</tr>
<tr>
<td><code>&lt;T extends Layout&gt; Array2D&lt;T&gt;</code></td>
<td><code>mapArray2D(java.lang.Class&lt;T&gt; cls, byte[] data, long dim1, long dim2)</code> Overlay a 2D array layout on a byte[].</td>
</tr>
<tr>
<td><code>&lt;T extends Layout&gt; Array2D&lt;T&gt;</code></td>
<td><code>mapArray2D(java.lang.Class&lt;T&gt; cls, long addr, long dim1, long dim2)</code> Overlay a 2D array layout on a given memory address.</td>
</tr>
<tr>
<td><code>T</code></td>
<td><code>mapLayout(java.lang.Class&lt;T&gt; cls, byte[] data)</code> Overlay a singleton layout on a byte[].</td>
</tr>
<tr>
<td><code>T</code></td>
<td><code>mapLayout(java.lang.Class&lt;T&gt; cls, long addr)</code> Overlay a singleton layout on memory at a given address.</td>
</tr>
<tr>
<td><code>&lt;T extends LayoutType&gt; T</code></td>
<td><code>mapPrimArray1D(java.lang.Class&lt;? super T&gt; primCls, byte[] data, long length)</code> Overlay a primitive 1D array on a byte buffer.</td>
</tr>
<tr>
<td><code>&lt;T extends LayoutType&gt; T</code></td>
<td><code>mapPrimArray1D(java.lang.Class&lt;? super T&gt; primCls, long addr, long length)</code> Overlay a primitive 1D array at a memory address.</td>
</tr>
<tr>
<td><code>&lt;T extends LayoutType&gt; T</code></td>
<td><code>mapPrimArray2D(java.lang.Class&lt;? super T&gt; primCls, byte[] data, long dim1, long dim2)</code> Overlay a primitive 2D array on a byte buffer.</td>
</tr>
<tr>
<td><code>&lt;T extends LayoutType&gt; T</code></td>
<td><code>mapPrimArray2D(java.lang.Class&lt;? super T&gt; primCls, long addr, long dim1, long dim2)</code> Overlay a primitive 2D array at a memory address.</td>
</tr>
</tbody>
</table>
Memory model

- Something needs to describe the LDL and runtime behavior:
  The memory model

A memory model does the following:
- Describes the data layout
- Describes the data access disciplines
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>↑</td>
<td>Java</td>
<td>19.274%</td>
<td>+4.29%</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>↓</td>
<td>C</td>
<td>14.732%</td>
<td>-1.67%</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>↑</td>
<td>C++</td>
<td>7.735%</td>
<td>+3.04%</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>↑</td>
<td>C#</td>
<td>4.837%</td>
<td>+1.43%</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>↑</td>
<td>Python</td>
<td>4.066%</td>
<td>+0.95%</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>↓</td>
<td>Objective-C</td>
<td>3.195%</td>
<td>-6.36%</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>↑</td>
<td>PHP</td>
<td>2.729%</td>
<td>-0.14%</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>↑</td>
<td>Visual Basic .NET</td>
<td>2.708%</td>
<td>+1.40%</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>↑</td>
<td>JavaScript</td>
<td>2.162%</td>
<td>-0.01%</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>↓</td>
<td>Perl</td>
<td>2.118%</td>
<td>-0.10%</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td></td>
<td>Visual Basic</td>
<td>1.781%</td>
<td>-0.23%</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>↑</td>
<td>Assembly language</td>
<td>1.760%</td>
<td>+1.11%</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td></td>
<td>Ruby</td>
<td>1.416%</td>
<td>+0.17%</td>
</tr>
</tbody>
</table>

http://www.tiobe.com/ as of August 8th, 2015
1.7 The C++ memory model

The fundamental storage unit in the C++ memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set (2.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

[Note: The representation of types is described in 3.9. --- end note]

A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [Note: Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. --- end note] Two or more threads of execution (1.10) can update and access separate memory locations without interfering with each other.

[Note: Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of non-zero width. --- end note]

[Example: A structure declared as

---

63
Concurrent access

```c
typedef struct one_container {
    char a:2;
    char b:2;
    char c:2;
    char d:2;
} one_container;

typedef struct two_containers {
    char a:2;
    char b:2;
    char c:0;
    char c:2;
    char d:2;
} two_containers;
```

Thread 1          Thread 2          Thread 1          Thread 2
oc.b = 3;         oc.c = 0;         tc.b = 3;         tc.c = 0;
Concurrent access

```c
typedef struct one_container {
    char a:2;
    char b:2;
    char c:2;
    char d:2;
} one_container;
```

```c
typedef struct two_containers {
    char a:2;
    char b:2;
    char c:0;
    char d:2;
} two_containers;
```

Thread 1  Thread 2
oc.b = 3;  oc.c = 0;

Thread 1  Thread 2
tc.b = 3;  tc.c = 0;
A field is defined as a contiguous sequence of bits, where bit zero is the least significant bit (this is a LE-centric bit numbering scheme).
Layouts: containers

- A **field** is defined as a contiguous sequence of bits, where bit zero is the least significant bit (this is a LE-centric bit numbering scheme).

- A **container** is defined as a contiguous sequence of one or more fields. This implies that a field cannot have a greater size than its enclosing container. There are no other size restrictions on fields. The size of a container must be a multiple of 8 bits.
Layouts: the other bits
Layouts: the other bits
Layouts: the other bits
Layouts: the other bits
Layouts: the other bits

- A **field** is defined as a contiguous sequence of bits, where bit zero is the least significant bit (this is a LE-centric bit numbering scheme).

- A **container** is defined as a contiguous sequence of one or more fields. This implies that a field cannot have a greater size than its enclosing container. There are no other size restrictions on fields. The size of a container must be a multiple of 8 bits.

- A **member** of a layout can be a container, array, union, or nested layout. A container is the most primitive form of member.

- An **array** is defined as a contiguous sequence of identical member types. Multi-dimensional arrays are laid out in row-major order.

- A **union** is defined as a group of overlapping member types where the size of the union is the size of its largest member type.

- A **layout** is defined as a contiguous sequence of members.
Within the same container, updates may interfere.

Writes can’t interfere across containers.
Containers that are marked “atomic” must be updated as a unit.

Atomicity and tearing rules are only enforced for the individual member types in a union.
Layout data access: endian

Images by R. S. Shaw - Own work. Licensed under Public Domain via Wikimedia Commons - https://commons.wikimedia.org/wiki/File:Big-Endian.svg#/media/File:Big-Endian.svg
Layout alignment

- Layout Alignment
  \[
  = \text{Max}(\text{Align}(\text{member1}), \ldots, \text{Align}(\text{memberN}))
  
  = \text{Max}(\text{Max}(\text{Align}(\text{member1}_1), \text{Max}(\text{Align}(\text{member1}_2))), \ldots, \text{Align}(\text{memberN}))
  
  = \text{Max}(\text{Max}(\text{ROUND2N}(\text{sizeof}(\text{container1}_1)),
    \text{Max}(\text{ROUND2N}(\text{sizeof}(\text{container1}_2))),
    \ldots, \text{Align}(\text{memberN}))
  \]
Layout type information

- Primitive types
  - boolean
  - byte
  - char
  - short
  - int
  - long
  - float
  - double
Layout type information

- Primitive types
- Structured types

A layout member that has a structured type is accessed using a layout.

A getter for the member would return a layout instance.

I.e.: composites of containers (arrays, nested)
Layout type information

- **Primitive types**
- **Structured types**
- **Raw types**

Provide raw access to data

Use ByteBuffer to read / write
Layout type information

- Primitive types
- Structured types
- Raw types
- Pointer types

Need to be able to express and dereference pointers.
Layout type information

- Primitive types: Has an offset and size.
- Structured types: No accessor methods.
- Raw types
- Pointer types
- Opaque types
Layout: effective addresses

```java
public interface Packet {
    int source();
    int dest();
    ByteBuffer payload(); // Raw
}

interface EA {
    EffectiveAddress sourceEA();
    EffectiveAddress destEA();
    EffectiveAddress payloadEA();
}
```
Open issues: StackMapTable attribute

```c
full_frame {
    u1 frame_type = FULL_FRAME; /* 255 */
    u2 offset_delta;
    u2 number_of_locals;
    verification_type_info locals[number_of_locals];
    u2 number_of_stack_items;
    verification_type_info stack[number_of_stack_items];
}
```
Open issues: StackMapTable attribute

full_frame {
  u1 frame_type = FULL_FRAME; /* 255 */
  u2 offset_delta;
  u2 number_of_locals;
  verification_type_info locals[number_of_locals];
  u2 number_of_stack_items;
  verification_type_info stack[number_of_stack_items];
}

---

Fixed
Open issues: StackMapTable attribute

```c
full_frame {
    u1 frame_type = FULL_FRAME; /* 255 */
    u2 offset_delta;
    u2 number_of_locals;
    verification_type_info locals[number_of_locals];
    u2 number_of_stack_items;
    verification_type_info stack[number_of_stack_items];
}
```
Open issues: StackMapTable attribute

```c
full_frame {
    u1 frame_type = FULL_FRAME; /* 255 */
    u2 offset_delta;
    u2 number_oflocals;

    verification_type_info locals[number_of_locals];

    u2 number_of_stack_items;
    verification_type_info stack[number_of_stack_items];
}
```
Open issues: memory lifetime

```c
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```
Open issues: security

CC https://commons.wikimedia.org/wiki/File:Gate,_Minnowburn_Path_-_geograph.org.uk_-_1407363.jpg
Open issues: Existential questions

- Who am I?
- What’s the meaning of life?
Open issues: LDL grammar

To be honest, I am kind of mystified at the design choices for the grammar; it seems to be chosen to be both hard to mechanically parse *and* hard for humans to read!

--Brian Goetz
Get involved now

http://mail.openjdk.java.net/pipermail/panama-spec-experts/
http://danheidinga.github.io/J9-Panama/StateOfTheLDL.html

Ensure your off heap use cases are known!

Contact anyone on the panama spec experts list to discuss
IBM and the IBM logo are trademarks or registered trademarks of IBM Corporation, in the United States, other countries or both.

Java and all Java-based marks, among others, are trademarks or registered trademarks of Oracle in the United States, other countries or both.

Other company, product and service names may be trademarks or service marks of others.

THE INFORMATION DISCUSSED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, SUCH INFORMATION. ANY INFORMATION CONCERNING IBM'S PRODUCT PLANS OR STRATEGY IS SUBJECT TO CHANGE BY IBM WITHOUT NOTICE.