Turn Your XML into Binary
Make It Smaller and Faster
JavaOne 2014

by

John Davies | CTO

@JTDavies

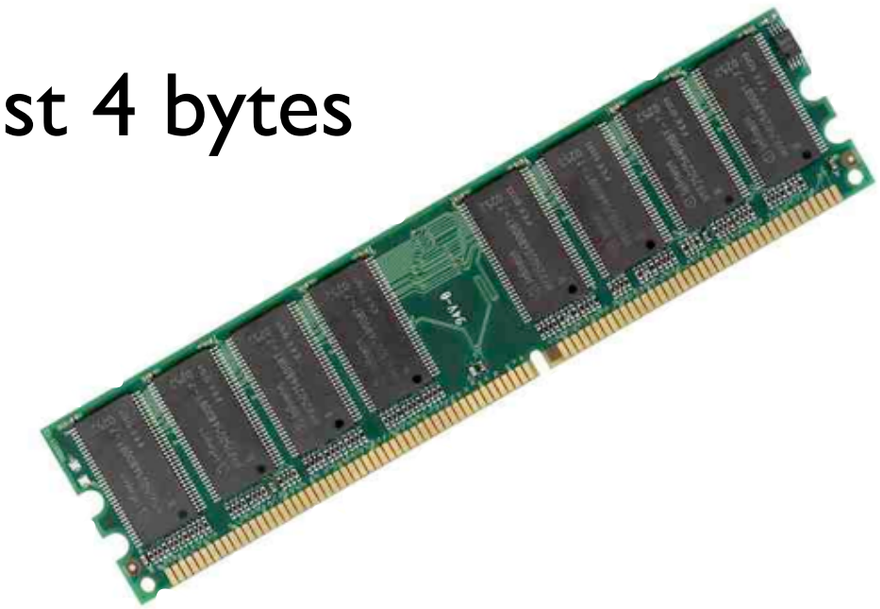
Please ask questions

- Feel free to ask questions during the talk
- We'll also have time during the demos
- Please tweet questions or comments to me @jtdavies

- My question for you…
- How much memory do you need to store this String?
- "John"

Java IS the problem!

- Java is very inefficient at storing data in memory
- It was designed specifically to abstract the hardware
  - Why should you need to know, write once - run anywhere!
- Take the string "ABC", typically it needs just 4 bytes
  - Three if you know the length won't change
  - Even less if "ABC" is an enumeration



- Java takes 48 bytes to store "ABC" as a String
  - You could argue that we don't need to run down the entire length of the String to execute length() but it's a big price to pay

It's not just String

- If it was just String then we could use byte[] or char[] but Java bloating is endemic
  - Double
  - BigDecimal
  - Date
  - ArrayList

- Use just one or two and we're OK but write a class with a few of these and we really start to see the problem

- A class with 10 minimum sized Objects can be over 500 bytes in size - for each instance
  - What's worse is that each object requires 11 separate memory allocations
  - All of which need managing
  - Which is why we have the garbage collector(s)

Start Simple…

- Simple data we're going to be referring to for the next few slides…

ID	TradeDate	BuySell	Currency1	Amount1	Exchange Rate	Currency2	Amount2	Settlement Date
1	21/07/2014	Buy	EUR	50,000,000	1.344	USD	67,200,000.00	28/07/2014
2	21/07/2014	Sell	USD	35,000,000	0.7441	EUR	26,043,500.00	20/08/2014
3	22/07/2014	Buy	GBP	7,000,000	172.99	JPY	1,210,930,000.00	05/08/2014
4	23/07/2014	Sell	AUD	13,500,000	0.9408	USD	12,700,800.00	22/08/2014
5	24/07/2014	Buy	EUR	11,000,000	1.2148	CHF	13,362,800.00	31/07/2014
6	24/07/2014	Buy	CHF	6,000,000	0.6513	GBP	3,907,800.00	31/07/2014
7	25/07/2014	Sell	JPY	150,000,000	0.6513	EUR	97,695,000.00	08/08/2014
8	25/07/2014	Sell	CAD	17,500,000	0.9025	USD	15,793,750.00	01/08/2014
9	28/07/2014	Buy	GBP	7,000,000	1.8366	CAD	12,856,200.00	27/08/2014
10	28/07/2014	Buy	EUR	13,500,000	0.7911	GBP	10,679,850.00	11/08/2014

Start with the CSV

- Each line is relatively efficient

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

- But it's in human-readable format not CPU readable
  - At least not efficient CPU readable

- We could store the lines as they are but in order to work with the data we need it in something Java can work with
  - The same goes for any other language, C, C++, PHP, Scala etc.

- So typically we parse it into a Java class and give it a self-documenting name - Row

CSV to Java

- This seems like a reasonably good implementation
- From this…

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

- We get this…

```
public class ObjectTrade {
    private long id;
    private Date tradeDate;
    private String buySell;
    private String currency1;
    private BigDecimal amount1;
    private double exchangeRate;
    private String currency2;
    private BigDecimal amount2;
    private Date settlementDate;
}
```

Everything's fine

- With very simple getters and setters, something to parse the CSV and a custom toString() we're good

```
public BasicTrade parse( String line ) throws ParseException {
        String[] fields = line.split(",");
        setId(Long.parseLong(fields[0]));
        setTradeDate(DATE_FORMAT.get().parse(fields[1]));
        setBuySell(fields[2]);
        setCurrency1(fields[3]);
        setAmount1(new BigDecimal(fields[4]));
        setExchangeRate(Double.parseDouble(fields[5]));
        setCurrency2(fields[6]);
        setAmount2(new BigDecimal(fields[7]));
        setSettlementDate(DATE_FORMAT.get().parse(fields[8]));

        return this;
    }
```

- What could possibly go wrong?

This is Java

- In fact everything works really well, this is how Java was designed to work
  - There are a few "fixes" to add for SimpleDateFormat due to it not being thread safe but otherwise we're good

- Performance is good, well it seems good and everything is well behaved

- As the years go on and the volumes increase, we now have 100 million of them

- Now we start to see some problems
  - To start with we don't have enough memory - GC is killing performance
  - When we distribute the size of the objects are killing performance too

Why is Java one of the problems?

- A simple CSV can grow by over 4 times…

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

- From roughly 70 bytes per line as CSV to around 328 in Java
- That means you get over 4 times less data when stored in Java
- Or need over 4 times more RAM, network capacity and disk
  - Serialized objects are even bigger!

Java bloats your data

- You probably thought XML was bad imagine what happens when you take XML and bind it to Java!
- Anything you put into Java objects get horribly bloated in memory
- Effectively you are paying the price of memory and hardware abstraction

Java Objects - Good for vendors

- These fat Java Objects are a hardware vendor's wet dream
  - Think about it, Java came from Sun, it was free but they made money selling hardware, well they tried at least

- Fat objects need more memory, more CPU, more network capacity, more machines
  - More money for the hardware vendors

- And everything just runs slower because you're busy collecting all the memory you're not using

- Java for programmers was like free shots for AA members

This isn't just Java

- Think this is just a Java problem?

- It's all the same, every time you create objects you're blasting huge holes all over your machine's RAM

- And someone's got to clean all the garbage up too!

- Great for performance-tuning consultants :-)

In-memory caches

- If you use an in-memory cache then you're most likely suffering from the same problem…

Neo4j

GEMFIRE ENTERPRISE

ORACLE Coherence

hazelcast

GIGASPACES

MEMCACHED

EHCACHE

TERRACOTTA

- Many of them provide and use compression or "clever" memory optimisation
  - But this usually slows things down, introduces restrictions and only goes so far to resolve the issue

Classic Java Binding…

- This is how we'd typically code this simple CSV example…

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

```
public class ObjectTrade {
    private long id;
    private Date tradeDate;
    private String buySell;
    private String currency1;
    private BigDecimal amount1;
    private double exchangeRate;
    private String currency2;
    private BigDecimal amount2;
    private Date settlementDate;
}
```

- It's easy to write the code and fast to execute, retrieve, search and query data BUT it needs a lot of RAM and it slow to manage

Just store the original data?

- We could just store each row

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

```
public class StringTrade {
    private String row;
}
```

- But every time we wanted a date or amount we'd need to parse it and that would slow down analysis

- If the data was XML it would be even worse
  - We'd need a SAX (or other) parser every time

Just store the original data?

```
public class StringTrade {
    private String row;
}
```

- Allocation of new StringTrades are faster as we allocate just 2 Objects

- Serialization and De-Serialization are improved for the same reason

- BUT over all we lose out when we're accessing the data
  - We need to find what we're looking each time
  - This is sort of OK with a CSV but VERY expensive for XML

```
public class XmlTrade {
    private String xml;
}
```

Compression or Compaction?

- OK, a lot of asks, why don't we just use compression?

- Well there are many reasons, mainly that it's slow, slow to compress and slow to de-compress, the better it is the slower it is

- Compression is the lazy person's tool, a good protocol or codec doesn't compress well, try compressing your MP3s or videos

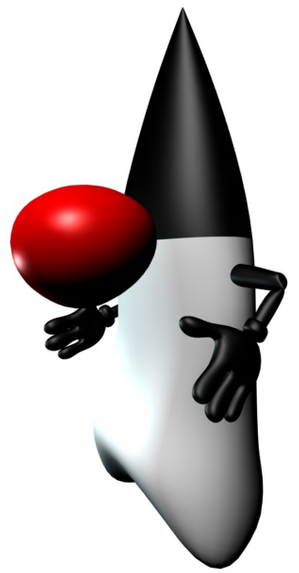
- It has it's place but we're looking for more, we want compaction not compression, then we get performance too

Now in binary…

- This is what our binary version looks like…

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

```
public class ObjectTrade extends SDO {
    private byte[] data;
}
```



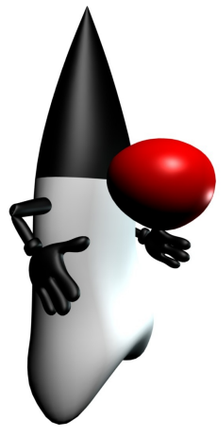
- Just one object again so fast to allocate
- If we can encode the data in the binary then it's fast to query too
- And serialisation is just writing out the byte[]

Same API, just binary

- Classic getter and setter vs. binary implementation
- Identical API

```
@Override
public Date getTradeDate() {
    return tradeDate;
}

@Override
public void setTradeDate(Date tradeDate) {
    this.tradeDate = tradeDate;
}
```



```
@Override
public Date getTradeDate() {
    long date = wordFromBytesFromOffset(8);
    date *= 86_400_000L; // milliseconds in a day
    return new Date(date);
}

@Override
public void setTradeDate(Date tradeDate) {
    long date = tradeDate.getTime();
    date /= 86_400_000L; // milliseconds in a day

    data[8] = (byte)(date >>> 8);
    data[9] = (byte)(date);
}
```

Just an example…

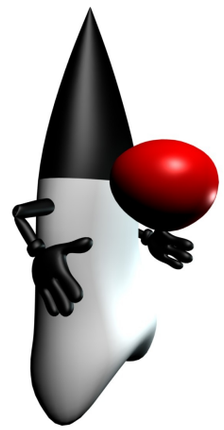
```
@Override
public Date getTradeDate() {
    long date = wordFromBytesFromOffset(8);
    date *= 86_400_000L; // milliseconds in a day
    return new Date(date);
}
```

```
@Override
public void setTradeDate(Date tradeDate) {
    long date = tradeDate.getTime();
    date /= 86_400_000L; // milliseconds in a day

    data[8] = (byte)(date >>> 8);
    data[9] = (byte)(date);
}
```

Did I mention … The Same API

- This is a key point, we're changing the implementation not the API
- This means that Spring, in-memory caches and other tools work exactly as they did before
- Let's look at some code and a little demo of this…



```
@Override
public Date getTradeDate() {
    long date = wordFromBytesFromOffset(8);
    date *= 86_400_000L; // milliseconds in a day
    return new Date(date);
}

@Override
public void setTradeDate(Date tradeDate) {
    long date = tradeDate.getTime();
    date /= 86_400_000L; // milliseconds in a day

    data[8] = (byte)(date >>> 8);
    data[9] = (byte)(date);
}
```

Time to see some code

- A quick demo, I've created a Trade interface and two implementations, one "classic" and the other binary
  - We'll create a List of a few million trades (randomly but quite cleverly generated)
  - We'll run a quick Java 8 filter and sort on them
  - We'll serialize and de-serialize them to create a new List
- Finally for the binary version we'll write out the entire list via NIO and read it in again to a new List

How does it perform?

- Compare classic Java binding to binary…
  - These are just indicative, the more complex the data the better the improvement, this is about the worse case (i.e. least impressive)

	Classic Java version	*Binary Java version*	*Improvement*
Bytes used	***328***	***39***	***840%***
Serialization size	***668***	***85***	***786%***
Custom Serialization	***668***	***40***	***1,670%***
Time to Serialize/ Deserialize	***41.1µS***	***4.17µS***	***10x***
Batched Serialize/ Deserialize	***12.3µS***	***44nS***	***280x***

Batching & Mechanical sympathy

- You probably noticed that the actual byte[] size was 39 but Java used 48 bytes per instance

- By batching and creating batch classes that handle the large numbers of instances not as a List or Array but more tightly we can get further improvements in memory and performance

- 1 million messages or 39 bytes should be exactly 39,000,000 bytes
  - As things get more complex the message size varies and we often have to compromise with a larger batch quanta
  - We usually have to stick to 8 byte chunks too

- To do this safely we'd probably have to use something like 48 bytes per instance in this example

Batching & Mechanical sympathy

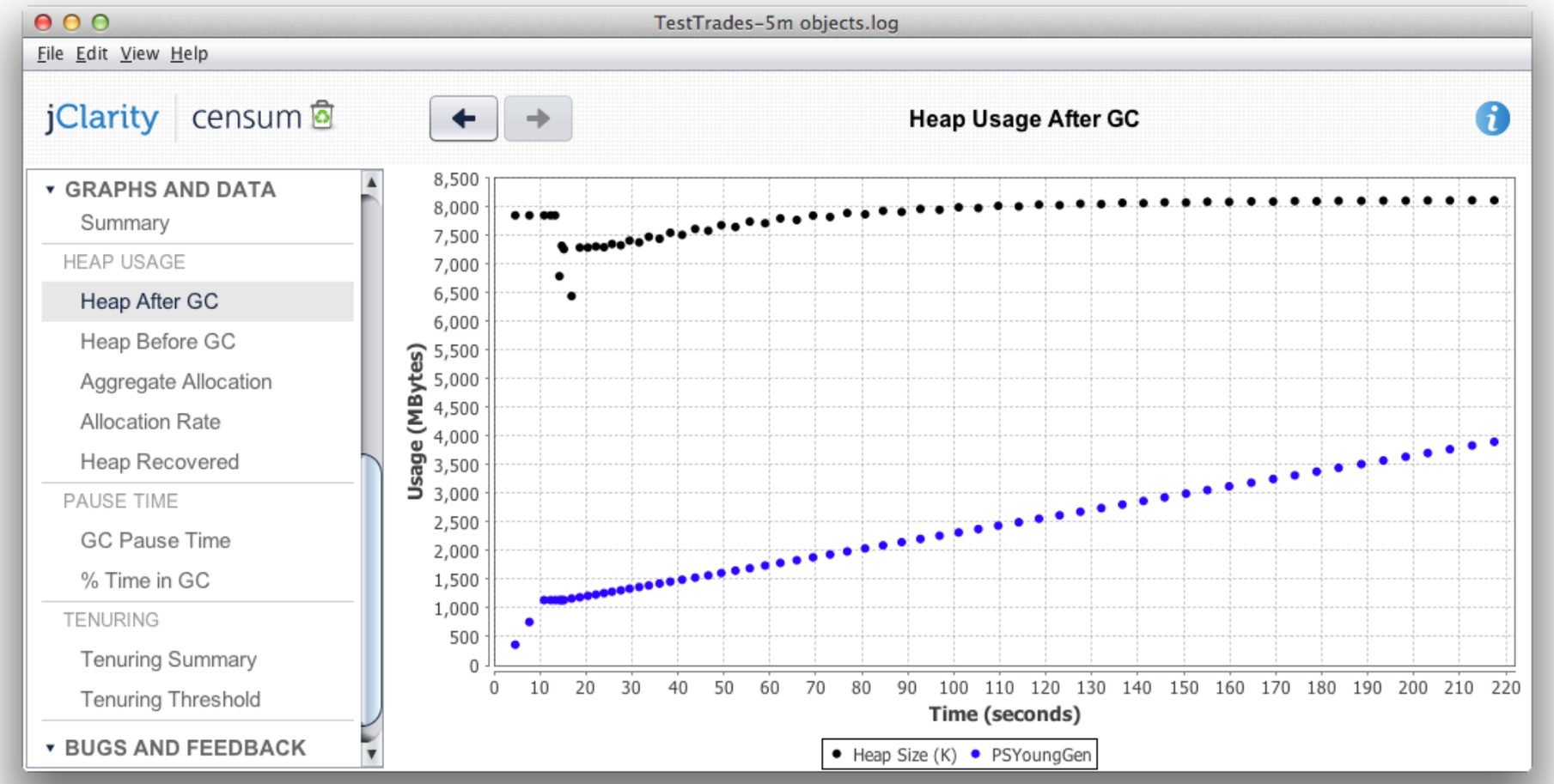
- Knowing how your disk (HDD or SSD) works, knowing how you network works means we can make further optimisations

- A typical network packet it about 1.5k in size, if we can avoid going over that size we see considerable network performance improvements

- What Java set out to do was to abstract the programmer from the hardware, the memory, CPU architecture and network, that abstraction has cost us dearly
  - With binary encoding we can keep Java but take advantage of the lower-level memory usage and batching

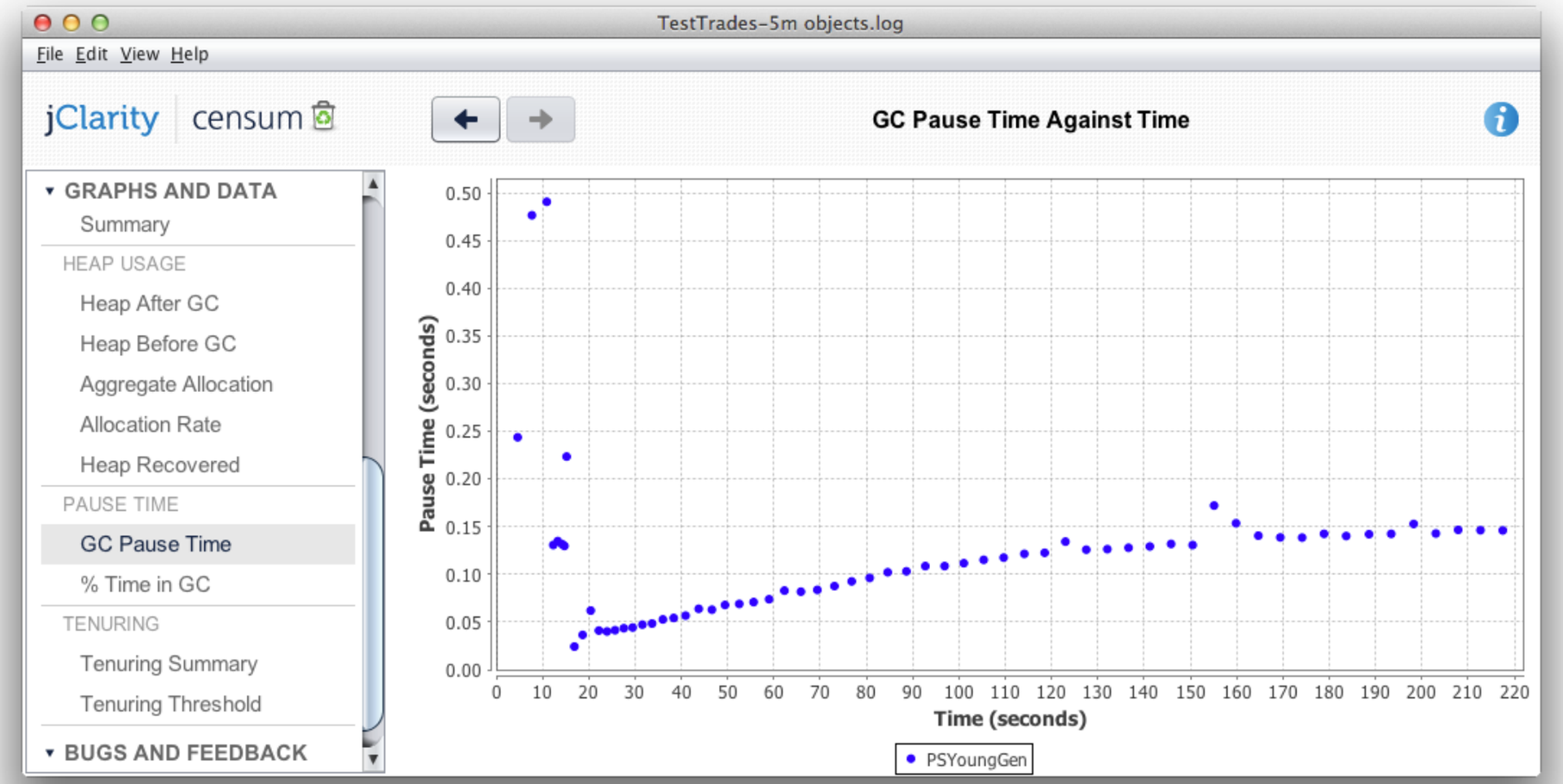
Memory heap usage (Object version)

- 200 Seconds for serialization and 4GB of heap used



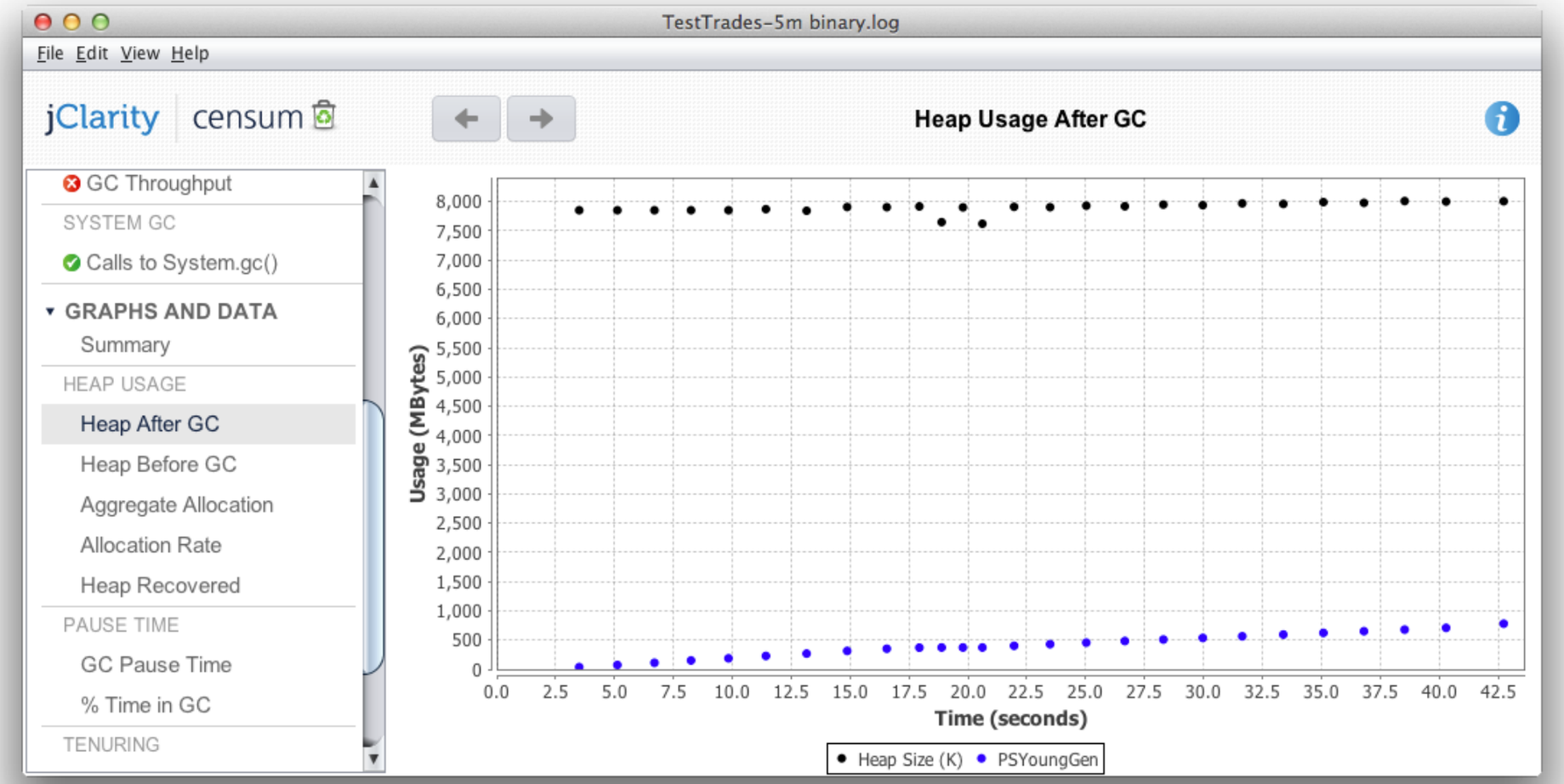
Memory heap usage (Object version)

- GC pause up to 500mS, averaging around 150mS



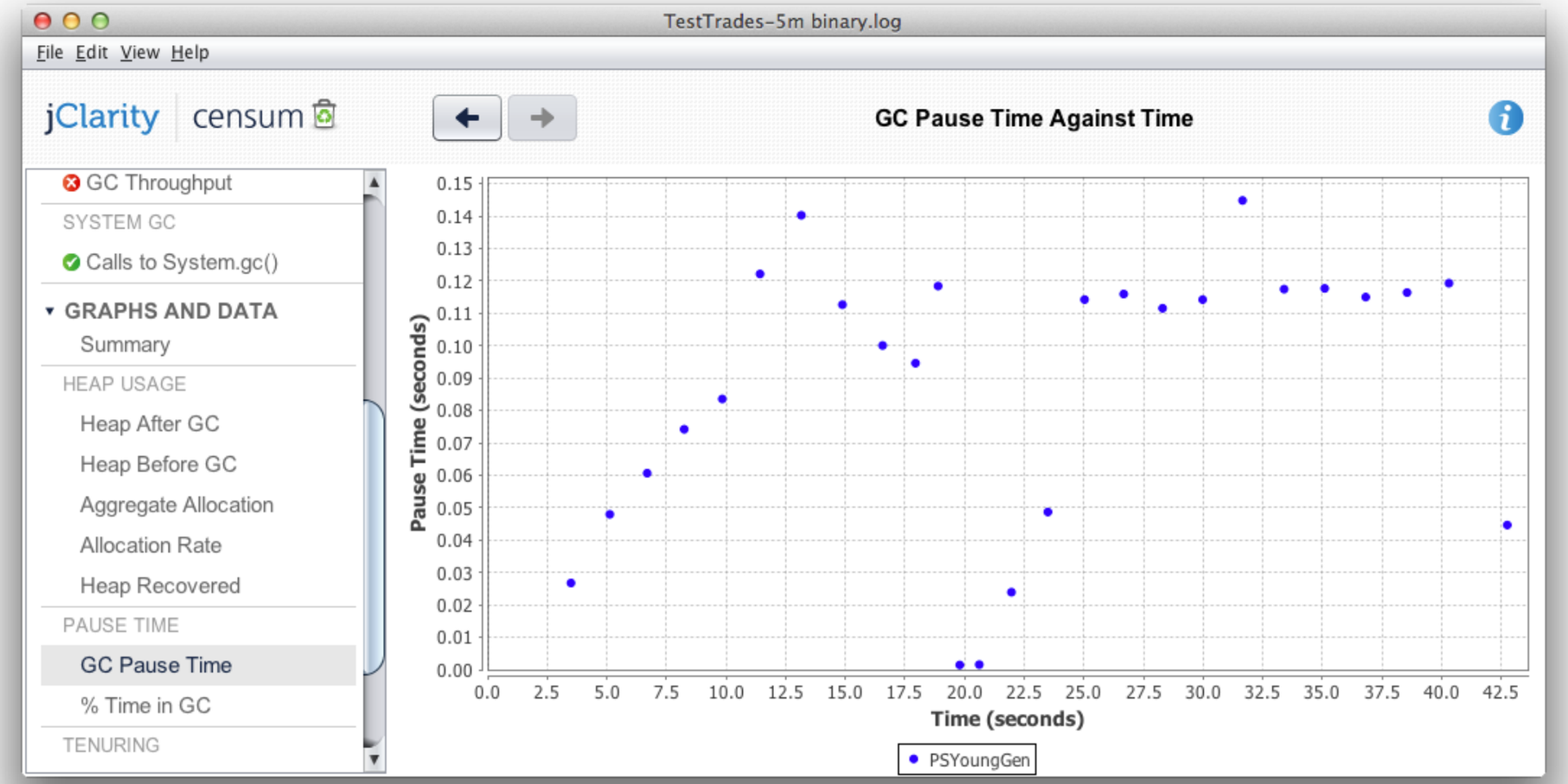
Memory heap usage (Binary version)

- 40 Seconds for serialization and 700MB of heap



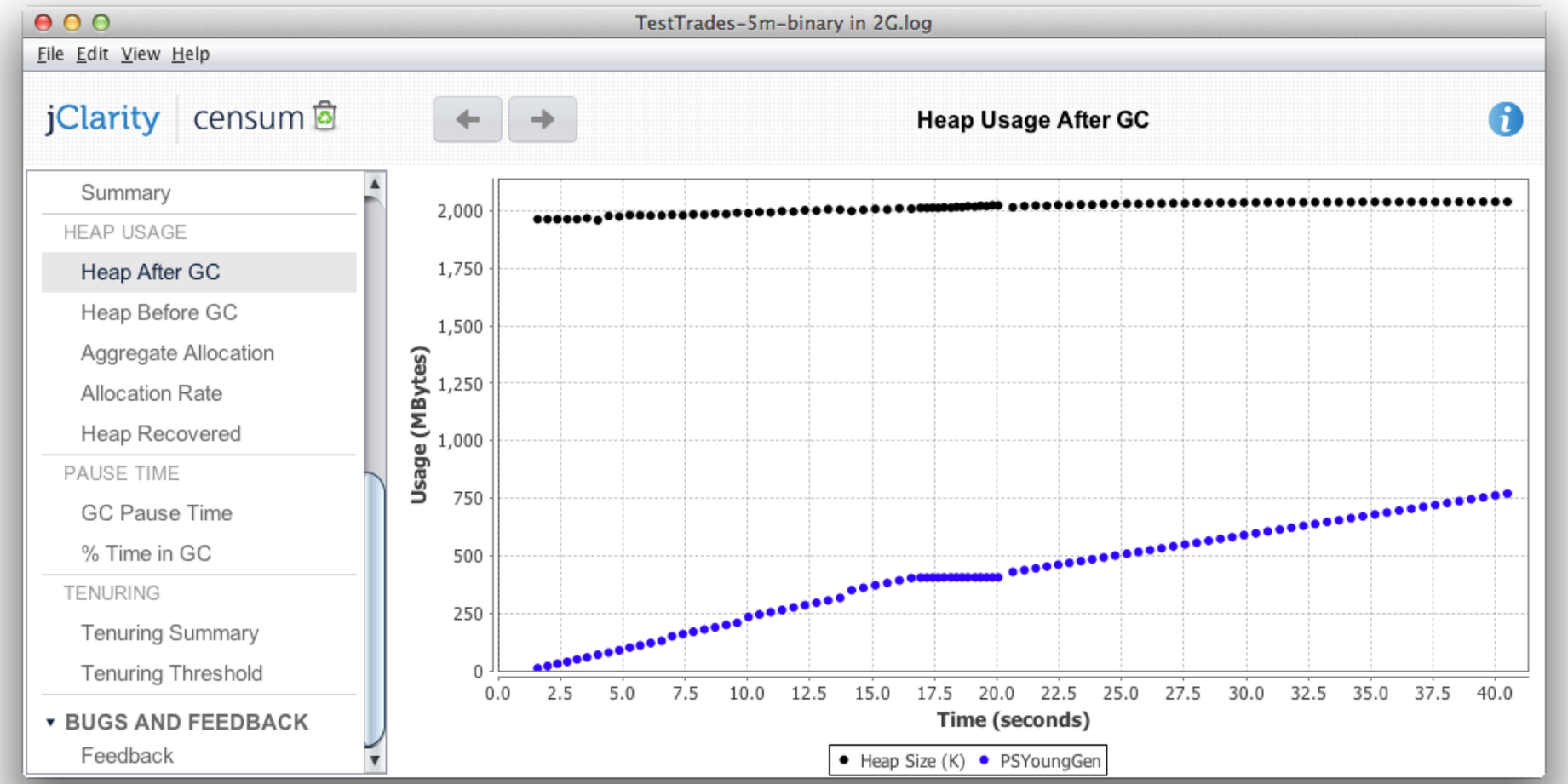
Memory heap usage (Binary version)

- GC pause up to 150mS, averaging around 100mS but a lot less



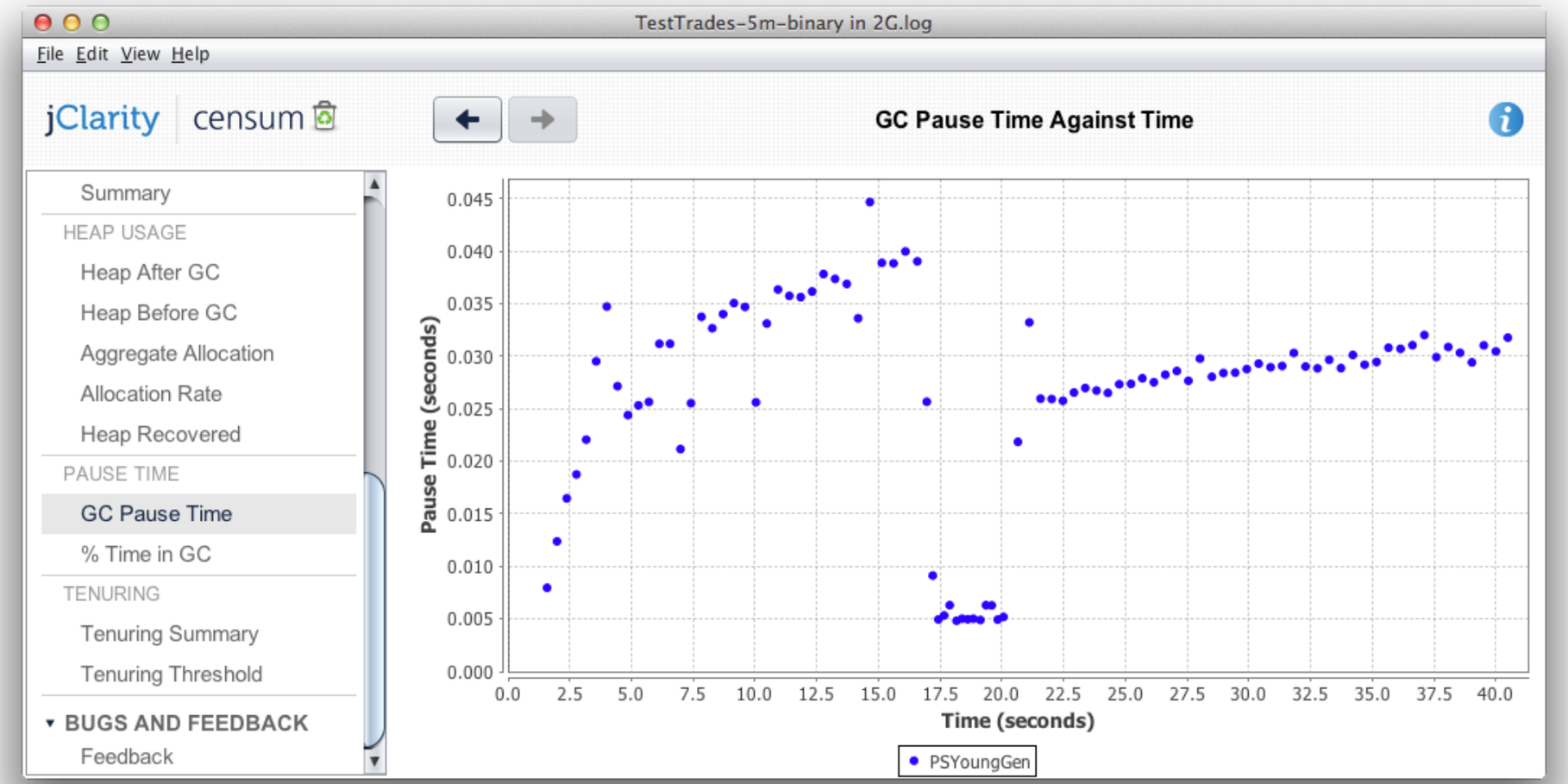
Memory heap usage (Binary version)

- The same but in a 2GB heap



Memory heap usage (Binary version)

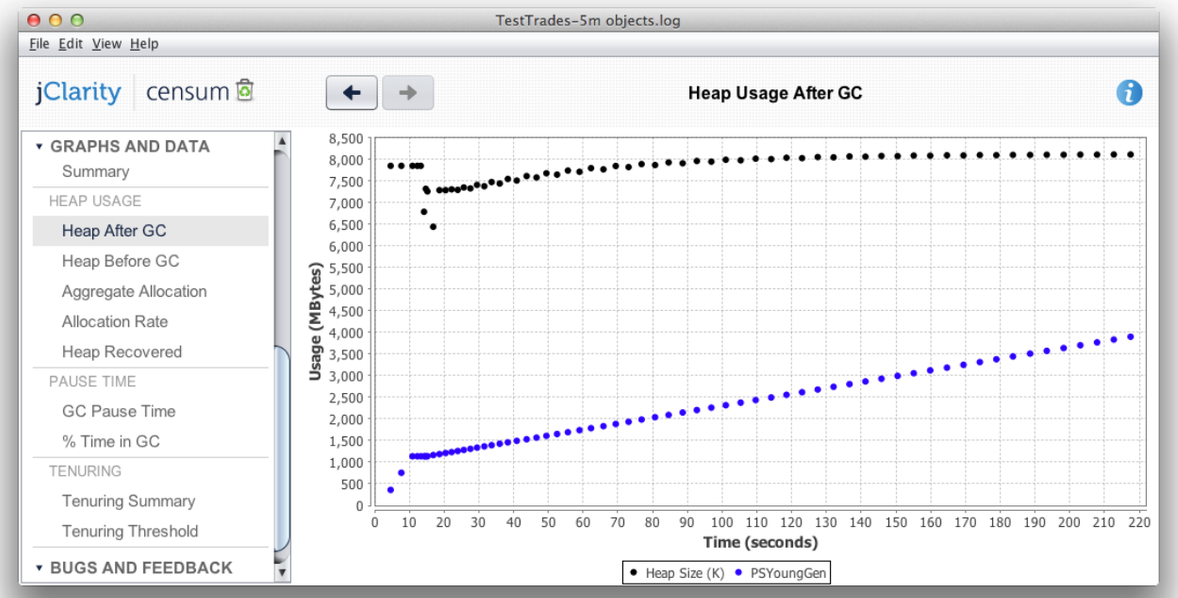
- GC pause up to 45mS, averaging around 30mS

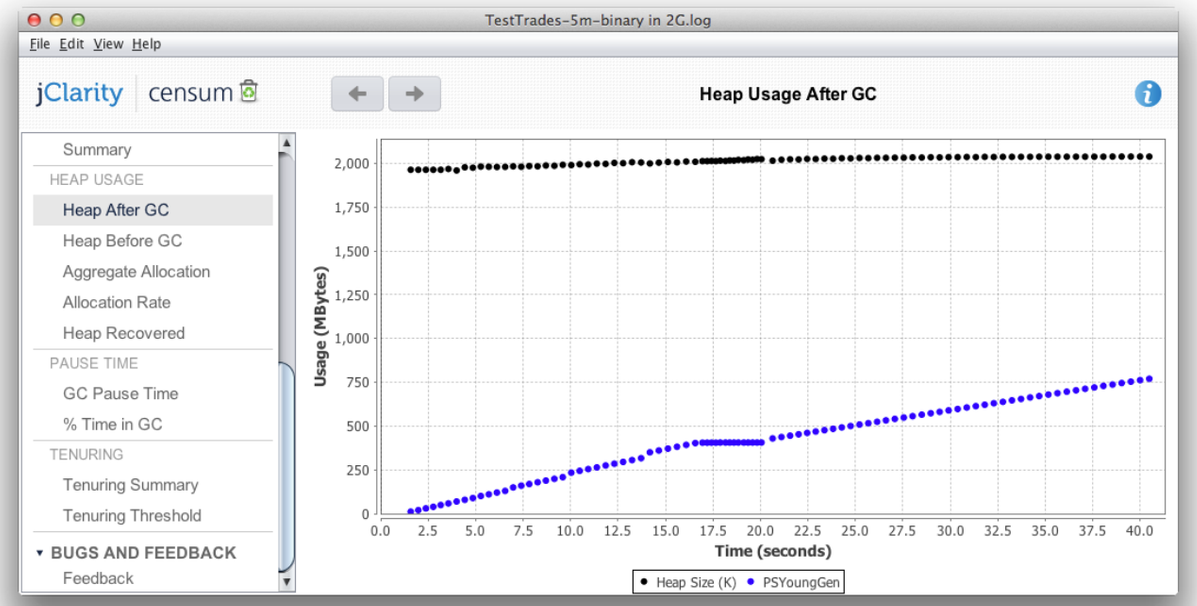


Comparing...

- While the shapes look the same you can clearly see the differences on both axis
  - The Object version is a lot slower
  - And the Object version uses significantly more memory

- Note that the left side (objects) has a heap size of 8GB, the right (binary) has a heap size of just 2GB

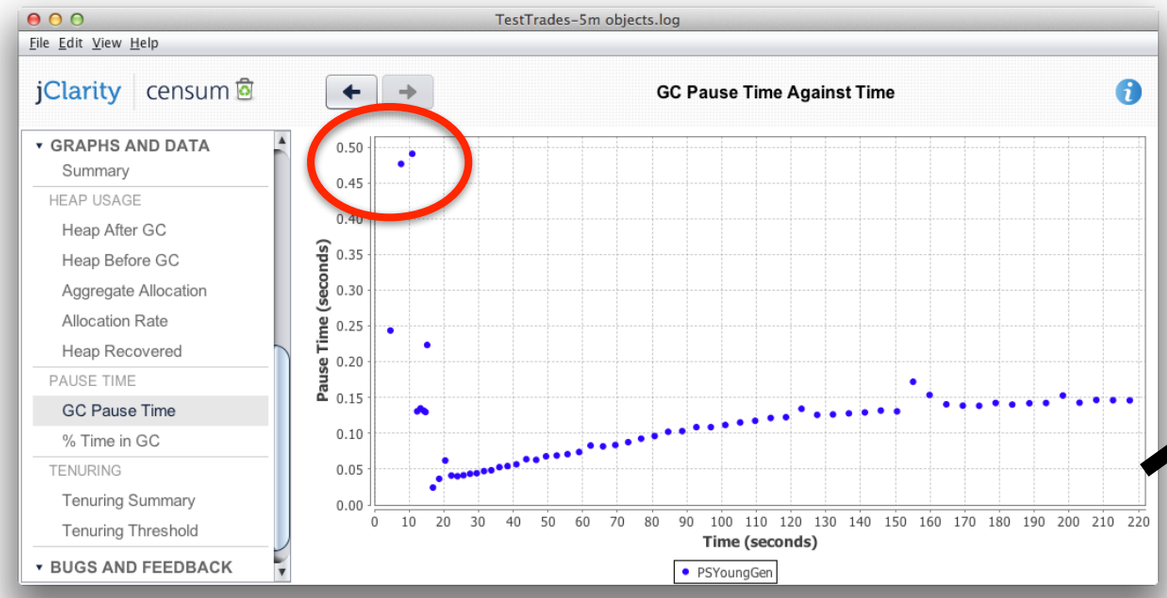


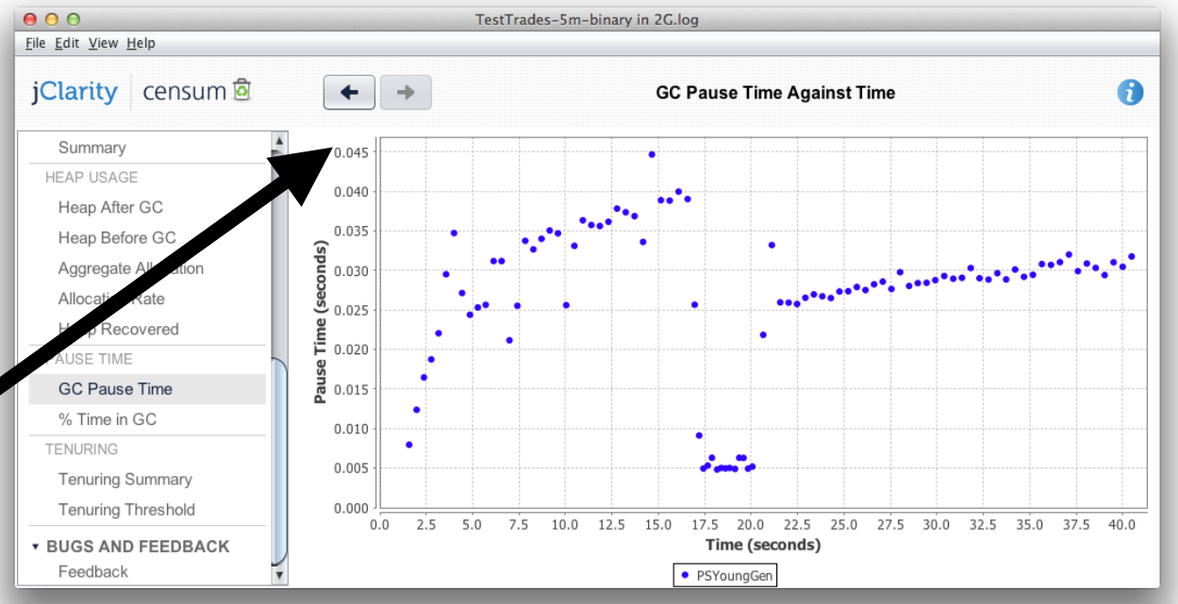


Comparing...

- These two graphs show the GC pause time during message creation and serialisation
  - Left is "classic" Java
  - Right is the binary version

- The top of the right hand graph is lower than the first rung of the left (50ms)





jClarity

- The memory usage graphs were created using jClarity's Censum
  - Many thanks to Martijn and especially Kirk for their help

- Next talk at 4pm in the Continental Ballroom 5

Generated code

- The C24 generated code for the Trade example is actually smaller, it averages around 33 bytes
- It uses run-length encoding so we can represent the 64 bit long by as little as a single byte for small numbers
- This means the size of the binary message varies slightly but for more complex models/message
- This is probably not a huge advantage for this CSV example but makes a huge difference with more complex XML

Beyond the CSV file...

- It worked for a simple CSV how about more complex models like XML or JSON?

- Once we have a mapping of types to binary and code we can extend this to any type of model

- But it gets a lot more complex as we have optional elements, repeating elements and a vast array of different types

Standard Java Binding

```
<resetFrequency>
    <periodMultiplier>6</periodMultiplier>
    <period>M</period>
</resetFrequency>
```

- JAXB, JIBX, Castor etc. generate something like …

```
public class ResetFrequency {
    private BigInteger periodMultiplier; // Positive Integer
    private Object period;               // Enum of D, W, M, Q, Y

    public BigInteger getPeriodMultiplier() {
        return this.periodMultiplier;
    }
    // constructors & other getters and setters
```

- In memory - 3 objects - at least 144 bytes
  - The parent, a positive integer and an enumeration for Period
  - 3 Java objects at 48 bytes is 144 bytes and it becomes fragmented in memory

Our Java Binding

```
<resetFrequency>
    <periodMultiplier>6</periodMultiplier>
    <period>M</period>
</resetFrequency>
```

- Using C24's SDO binary codec we generate …

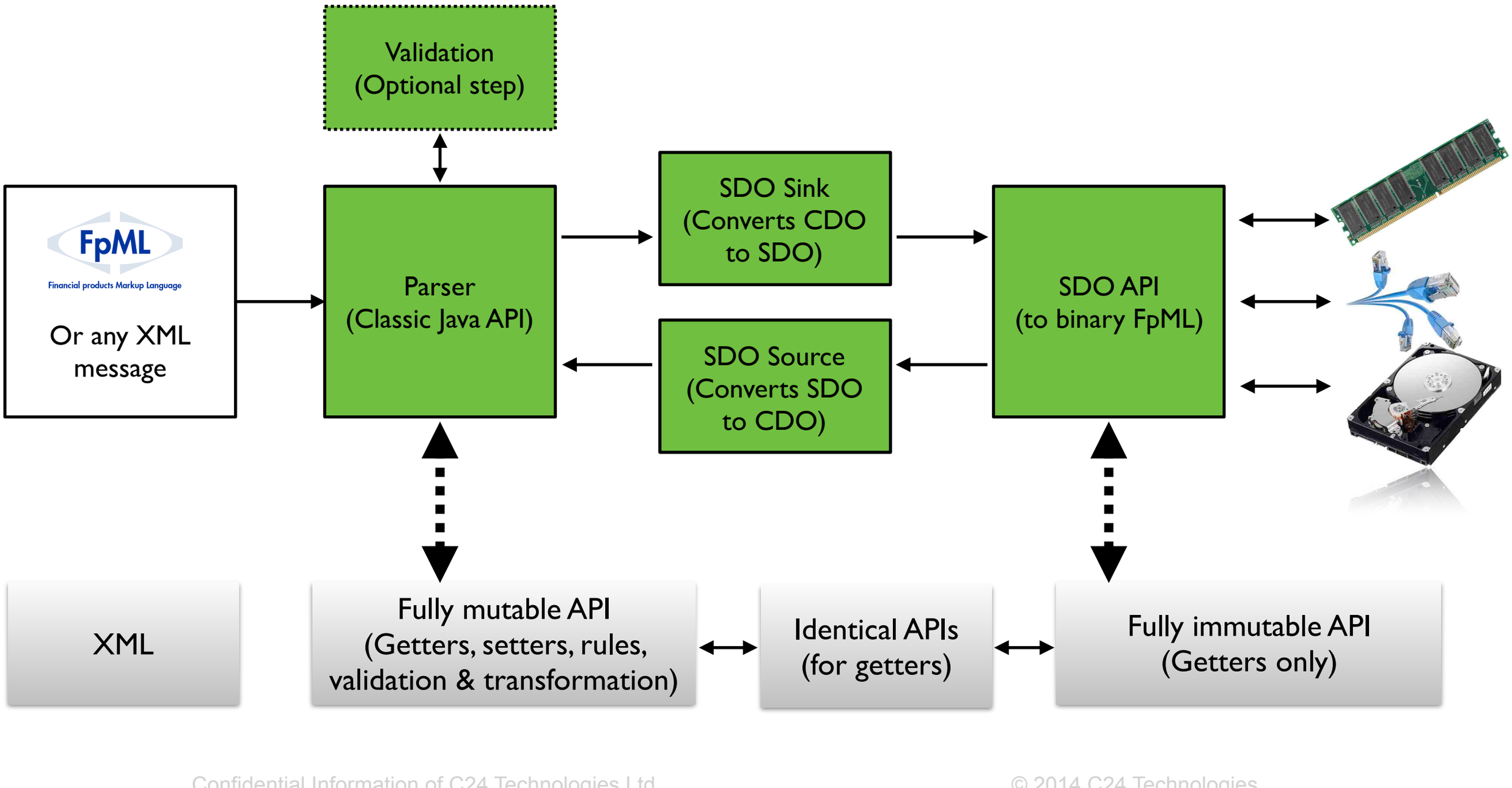
```
ByteBuffer data;    // From the root object

public BigInteger getPeriodMultiplier() {
        int byteOffset = 123;  // Actually a lot more complex
        return BigInteger.valueOf( data.get(byteOffset) & 0x1F );
    }
    // constructors & other getters
```

- In memory - 1 byte for all three fields
  - The root contains one ByteBuffer which is a wrapper for byte[]
  - The getters use bit-fields, Period is just 3 bits for values D, W, M, Q or Y

How it works

~5-8k	10-25k	**Size**	< 500 bytes
10k/sec	~1m/sec	**Performance**	~1m/sec



Another demo...

- Take some raw XML (an FpML derivative, about 7.4k of XML)
- Parse it, mutate a few variables and then compact each one to its binary version - We do this 1 million times
  - This takes about 100 seconds

- Now the test begins
  - We take a look at the binary version (just 370 bytes)
  - We search through the 1 million trades for data and aggregate the results
  - Then we try it multi-threaded (using `parallelStream()`)
  - A few more similar operations (I will discuss)

- Finally we'll write all 1 million to disk and then read them back into another array (comparing to make sure it worked)
  - This will be a test of serialisation

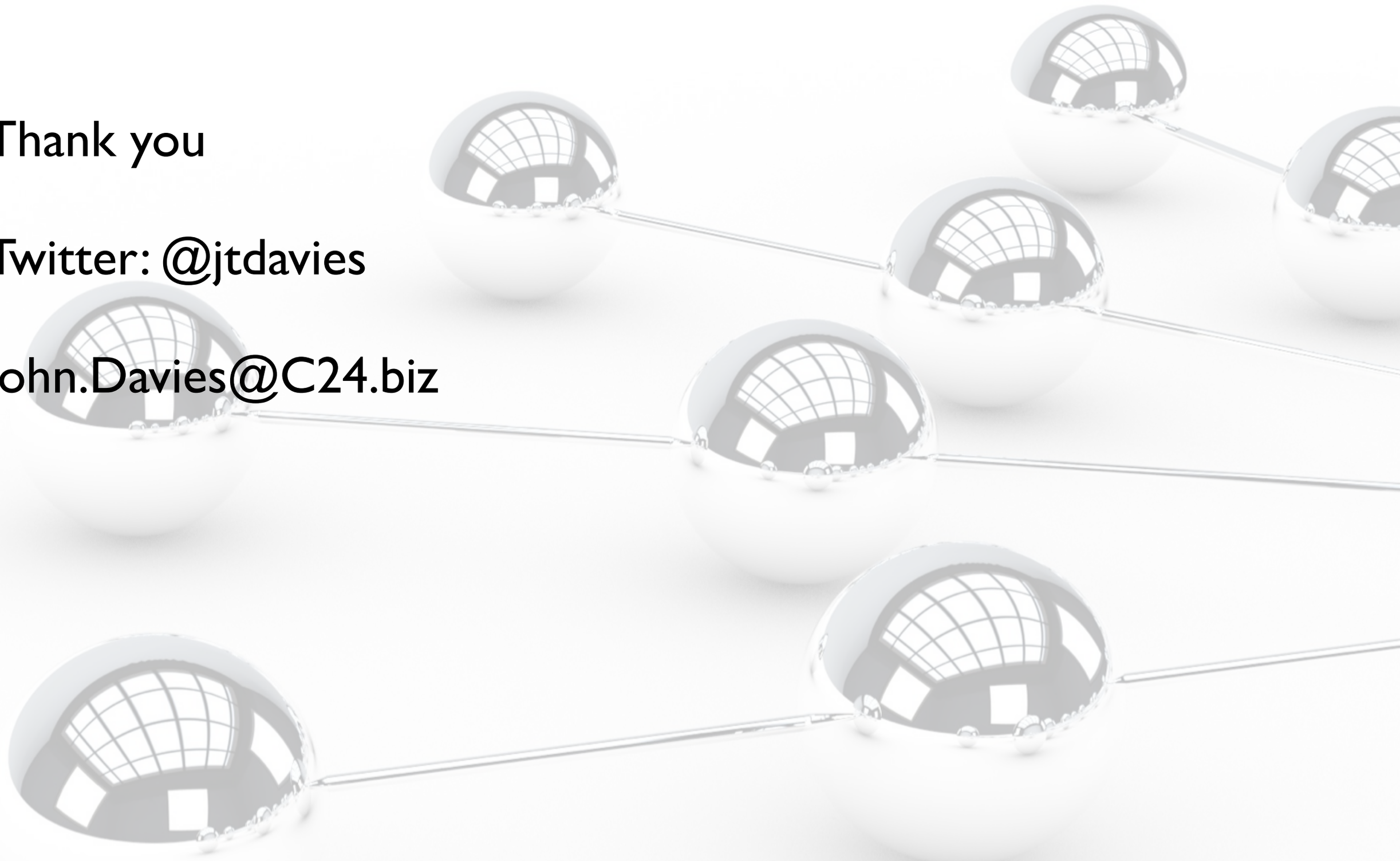
So it works

- Key points from the slides…

- If you want performance, scalability and ease of maintenance then you need…
  - Reduce the amount of memory you're using
  - Reduce the way you use the memory
  - Try using binary data instead of objects
  - Start coding like we used to in the 80s and 90s

- Or just buy much bigger, much faster machines, more RAM, bigger networks and more DnD programmers
  - And you'll probably get free lunches from your hardware vendors

- Thank you
- Twitter: @jtdavies
- John.Davies@C24.biz



- Thank you to Kirk Pepperdine, Andrew Elmore and the C24 team