

A photograph of a city skyline at night, with several tall buildings illuminated and their lights reflecting on the water in the foreground. The image is partially covered by a green semi-transparent rectangle on the right side.

Making the JVM fly

Experiences with Enterprise JVM Performance tuning

John Davies - CTO Incept5

Aarhus/Århus - 2nd October 2012

@jtdavies

Landing on the beach (New Zealand)



White Island (Volcano in New Zealand)



Sydney



The Olgas (Australia)



Parking at the pub (Australia)



- William Creek in the Outback

With Ross Mason over San Francisco



A great way to see the bridge



Agenda

- Memory management - yesterday and today
- Simple code tuning
- Obvious mistakes - hang on, not so obvious!
- Simple is better
- JVM Memory

The Problem

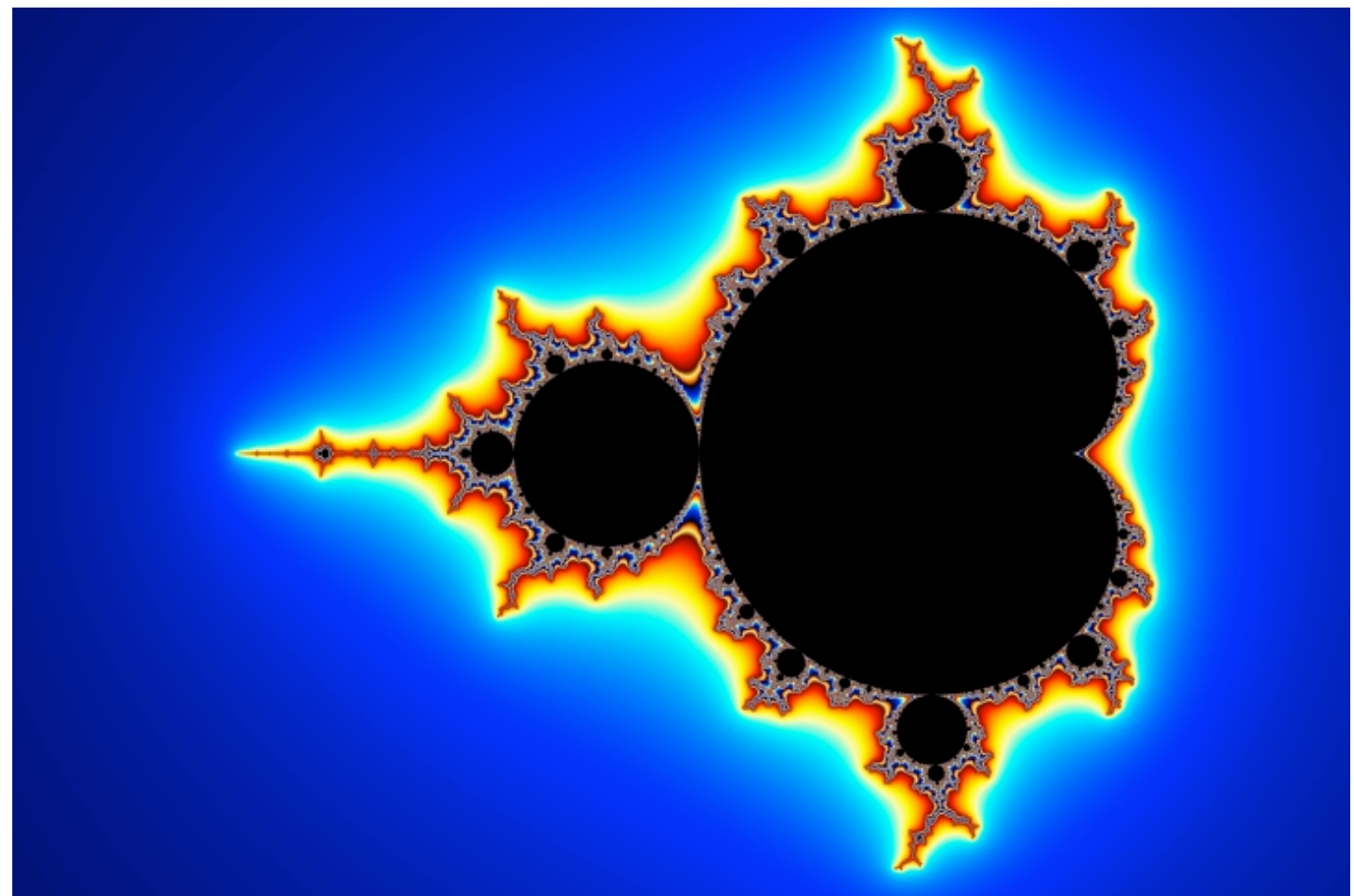
- **Seriously large volumes of data to be parsed**
 - Tens of thousands of Fix messages per second
 - Thousands of payments messages per second
- **Twenty dual quad-core machines, each processing 2,000 per second each might work**
 - But with a few weeks work and just 3 of those machines running 20,000 each is far better, cheaper and more resilient
- **Performance isn't just about doing things faster**
 - It's also better CPU usage, servers cost money and they're even more expensive to support in production
 - Latency costs money (“1ms = \$100m” in trading, “100ms = 1% of sales” for Amazon)

Today's talk

- No magic bullets :-)
- Hopefully some ideas for you
 - Decades of experience in some of the world's largest systems
- No go-faster JVM options
 - Hopefully a few that will help though
 - ... and I hope a better understanding of the options
- Most performance tuners are fine tuners, I like to tackle the architecture and design
 - Most of my challenges need an order of magnitude more “oomph”
 - Transactions are the biggest problem - that's another talk though!

Java vs. C/C++

- 15 years ago I used to teach Java for Learning Tree
- One part was demonstrating how to hook Java up to C to run things faster
- Naturally we used JNI
- The demo was impressive
 - It ran like a dog in Java
 - It flew in C



Mandelbrot - Early Java

- The Mandelbrot engine...

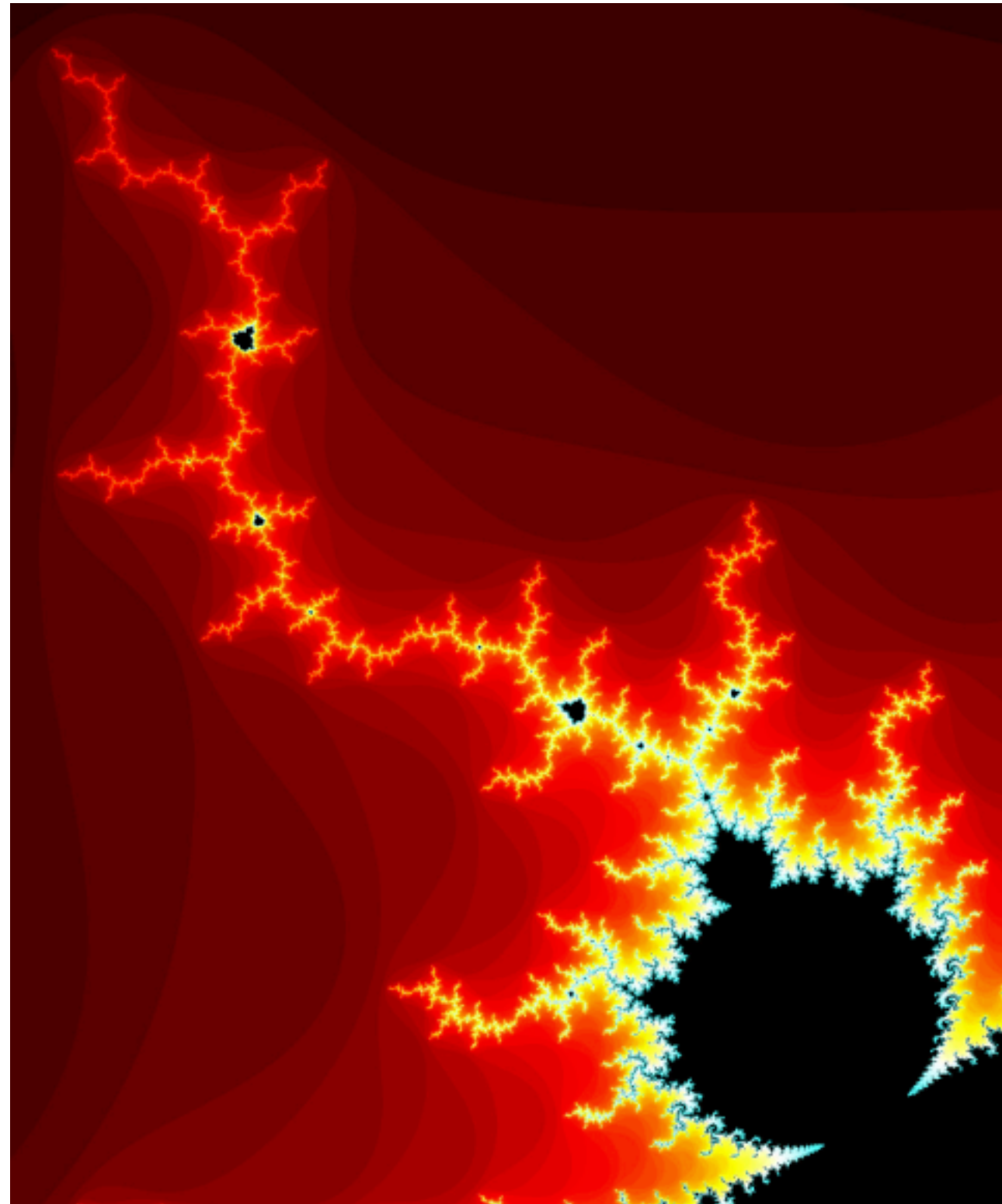
$$z_{n+1} = z_n^2 + c$$

new z previous z point on the plain to be calculated

```
private int iterate( double complex_real, double complex_imag ) {  
    double real = 0;  
    double imag = 0;  
    double new_real, new_imag;  
    int iteration_count = 0;  
  
    while (( real * real + imag * imag < 4 ) && ( iteration_count != MAX_ITER )) {  
        new_real = real * real - imag * imag + complex_real;  
        new_imag = real * imag + real * imag + complex_imag;  
  
        real = new_real;  
        imag = new_imag;  
  
        iteration_count++;  
    }  
    return( iteration_count );  
}
```

Java vs. C/C++

- By 2000 things had changed
 - The same demo showed a very different effect
- It became a demonstration of why NOT to use JNI
- Since then things have changed on the server-side
- Java became a serious player in the enterprise



Millions of calculations

- The Mandelbrot was a good benchmark but it tested mainly the CPU, at least the floating point part of it
 - A 1024x768 screen has 786,432 pixels, the Mandelbrot usually runs for about 100 to 1,000 iterations to get a smooth textures
 - That's up to 786 million floating point calculations per VGA screen
- Today we aim for 3D rendering in 1080p (1080x1920)
 - We also have enough memory to pre-calculate and store in-memory
 - Once the fractal is understood mathematically then interesting optimisations can be applied
- The Mandelbrot remains a good test of floating point performance

On to data processing

- Floating-Point processing is vital for financial calculations but a lot of the information comes in complex messages
- Let's take a quick look at Strings
 - We'll start a few years ago, about 20...

Strings in Pascal

- The first byte is the length of the String
 - '0C' or 13 in decimal is the length of "Hello Aarhus!"
 - It's interesting that we couldn't have used "Århus"
- Testing the length of the string was very quick
 - It was simply `string[0]`
- The maximum length was limited to 255 though
- Copying and substrings meant physical memory manipulation
 - Concatenation was relatively easy

0A457A	0C
0A457B	H
0A457C	e
0A457D	l
0A457E	l
0A457F	0
0A4580	
0A4581	A
0A4582	a
0A4583	r
0A4584	h
0A4585	u
0A4586	s
0A4587	!

Strings in C

- The last byte of the string is a '\0' (zero)
 - Strings are now only limited in size by available memory
 - Again we can't write "Århus" in classic C
- Testing the length of the string meant searching for the zero byte at the end
- Not being able to use the '\0' character limits C strings to 255 characters, unicode was out
 - But it was easy to pass a length too if needed
- Copying, substrings and concatenation also relatively easy

0A457A	H
0A457B	e
0A457C	l
0A457D	l
0A457E	0
0A457F	
0A4580	A
0A4581	a
0A4582	r
0A4583	h
0A4584	u
0A4585	s
0A4586	!
0A4587	\0

Working with strings in C

- So take a string in C (we use `char[]` which is like `byte[]`)...

```
char[] hello_string = "Hello Aarhus!";
```

- Copy it to another string...

```
char[] hello_string2 = hello; // cool and super fast!
```

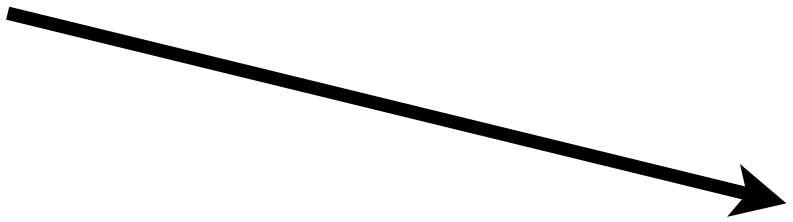
- In fact an explicit string was hard-coded at compile time

- Now modify the second string and print the first...

```
hello_string2[5] = '\0';  
printf("%s", hello_string);
```

- > **Hello**

- Oops!



0A457A	H
0A457B	e
0A457C	l
0A457D	l
0A457E	o
0A457F	
0A4580	A
0A4581	a
0A4582	r
0A4583	h
0A4584	u
0A4585	s
0A4586	!
0A4587	\0

Passing by reference...

- Back to the previous slide...

```
char[] hello_string = "Hello Aarhus!";
```

- Now we call a function (like a method)...

```
doSomething( hello_string );
```

- In the method we modify the string for some reason...

```
void doSomething( char[] string ) {  
    hello_string2[5] = '\0';  
}
```

- This changes the original hello_string

- But boy is it fast!!!

Strings and Collections

- In the days of C things weren't as simple as they were today
- Write a string to a collection (list, set, map etc.) and there was the question of who “owned” the string
 - If the writer deleted (freed) the string then the collection had a corrupt string (as did the reader)
 - If the reader freed it then the collection and writer had a corrupt string
 - Ownership of the string had to be part of the API created for the collections
- Eventually standard libraries came along and then OO languages like C++ came to the rescue, well almost!

Memory management

- In the “old days” we managed our own memory
- A good programmer or team with good disciplines could write some pretty slick code than ran like a rocket
- Sadly there was only a few of us :-)
- Virtual machines with memory management became the best solution for the masses
 - People could create string, copy them, duplicate them, write them to collections and create arrays of them and never worry about who owned them or who was going to delete them

Strings in Java

- We all know the classic error with Java Strings...
 - Running something like `firstString += otherString` in a loop
 - Obviously `StringBuffer` is a better option here
- But take something only slightly more complex and it's not so obvious...

```
String alphabet = "abcdefghijklmnopqrstuvwxyz";  
for( ... ) {  
    myObject.setLetter(alphabet.substring(9,10));  
}
```
- Here we're creating a new String with the `substring`
 - It's not a mistake but it creates a lot of extra work

The JVM will take care of it!

- Java's Garbage Collector is superb, the JIT compiler is superb
 - But neither will cope with bad design or crap code
- When you start to scale up in size or performance then you need to start thinking like a C programmer
- Ideally we should pass by reference not by value
 - But it's not possible in Java (C# yes but not Java)
 - In this case Java's String can be your friend
- We need to start thinking intelligently about passing large amounts of memory/data around

Working with byte[] instead of String

- The previous example

```
String alphabet = "abcdefghijklmnopqrstuvwxyz";  
for( ... ) {  
    myObject.setLetter(alphabet.substring(9,10));  
}
```

- When optimised runs over 60 times faster...

```
byte[] alphabetBytes = alphabet.getBytes();  
for( ... ) {  
    myObject.setLetter(alphabetBytes[9]);  
}
```

- If course you may need char[] rather than byte[] and be careful not to pass byte[] in the method call, that's passed by value!

A side line...

- Last week I visited a client
 - A large one who shall remain nameless
- They will go live in a few weeks and somewhat short of their performance target
 - Although they have time so they're not panicking yet
- I started at the top, the architecture
 - Diagrammatically is was relatively simple, decoupled and distributed
 - A few strange technology choices in some areas but nothing new there
- One of their applications was running a 4GB JVM

Too much memory

- We did a “ps -ef” and also found Apache Tomcat was also running on the same box
- The total memory allocated to the two JVMs running on the same machine was 6GB
 - Fine you might think, my laptop has 16GB, 6 is nothing these days
- I asked how big the machine was...
- Ah it's a virtual machine with 4GB RAM and ...
- Stop, 6GB of Java on a 4GB machine, check the swap space!

Too much memory

- It's very easy to give the JVM a lot of memory
 - But don't allocate more than you have and watch what else you have running on the same machine
- As soon as the JVM allocates memory past the VM (Linux VM) limits it hits swap space which is disk
- If things start getting busy on those machines and the memory gets used then they start hitting disk
- We start to see 2 orders of magnitude (over 100 times) drop in performance
 - It will be difficult to debug as everything will appear to be working

Over-engineering

- Another example of architectural issues
- A client needs a “safe store”, say 20,000 messages a second for 24 hours - 8 hours of live trading
 - Each message is just under 1k in size
 - 20 meg per second, 72 GB per hour
 - 72 million message per hour, 576 million (576 GB) in 8 hours
- So they used an in-memory storage product
 - Not only is it very expensive, it doesn't work very well for them (yet)
 - It's probably going to get refactored out

What's wrong with disk?

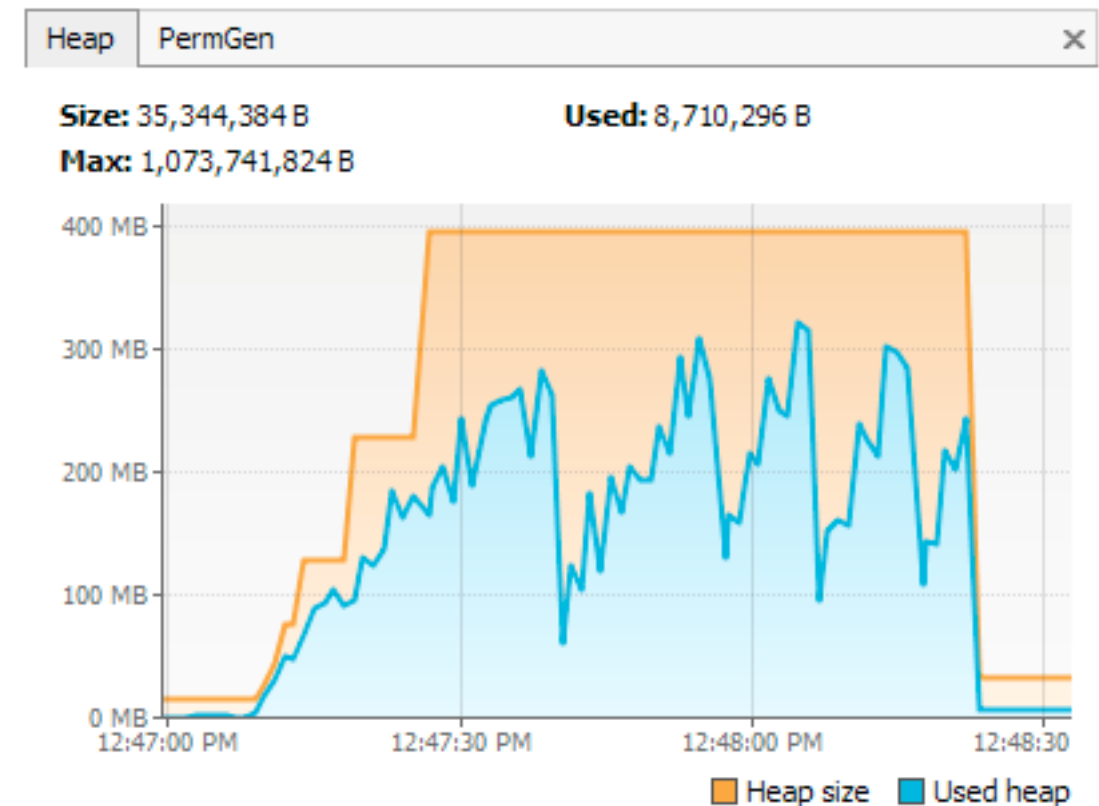
- Today's SSDs will write over 500m per second
 - 1.8TB per hour
 - Total cost of 1TB of SSD disk - under \$1,000
- We have clients writing up to 350,000 Fix messages a second from the CME onto disk, hard-disk!
 - Each hour we rotate the log, tar/gz it and upload it to EC2
- People tend to forget the simple solutions

Understanding the JVM memory

- The JVM has two main sections of memory
- Perm (“perm gen” or permanent generation) - used for classes etc.
- Heap - used for the application - more on this shortly
 - Both sections are garbage collected
- Perm tends to be fairly static, it usually runs out if you load a lot of classes
 - You can set the max size with
`-XX:MaxPermSize=250m`
 - Changing this is unlikely to effect performance in any way other than making your application work or not

The JVM heap

- The heap is the part of memory used by your application
 - You can set the maximum size with **-Xmx256m**
 - You can also set the initial size with **-Xms32m**
 - In the above example the heap will grow, as required, from 32 meg up to 256 meg
 - Each time it grows there is a minor pause as the new memory is requested from the OS
 - If you want consistent performance then set both values to the same
-Xms256m -Xmx256m

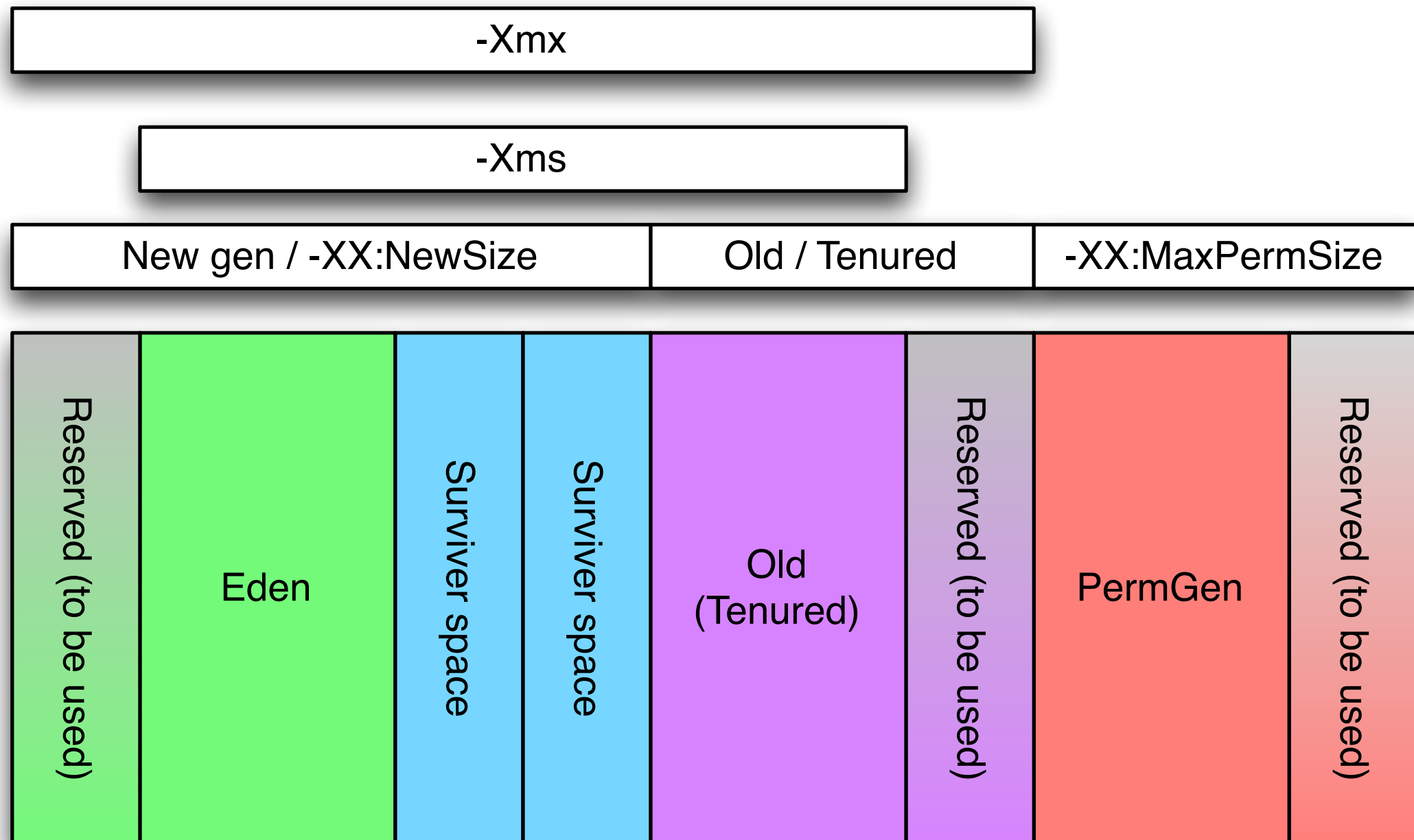


There's more...

- The heap is actually divided into several sections...
 - Old generation (tenured)
 - New generation - Subdivided into Eden and Survivor spaces, there are two survivor spaces
 - To set the ratio between new and old you can use `-XX:NewRatio=3`
- New Objects are created in “Eden”
 - When this is full a “young generation GC” takes place and surviving objects are put into survivor space
 - When survivor space gets full objects are moved to old generation
 - When old generation gets full a full GC takes place
 - As the heap get full the GC battles with the application for memory
- Eventually you get an `OutOfMemoryException`

JVM memory

- In a nutshell...



New vs. Old

- Understanding how your application uses memory is key to optimising the JVM
- If you're creating lots of temporary objects then you're going to get a lot of young GCs
 - Try setting the NewRatio to something like 2 or even 1
 - You can set the ration of the Eden to survivor space with `-XX:SurvivorRatio=8`
 - If it's too small too much is copied into old-gen, too large and it remains empty
- If you get this far then you'd better know what you're doing!!!

Some useful tools

- Assuming you've now got the perfect architecture
 - And you've optimised your code
 - You've killed transactions and taken out the ORM layer
 - Your application is now perfect in every way
- Finally you can move on to the JVM tuning...

jstat - for monitoring you application

jvisualvm - jstat in pretty graphics

**-verbosegc, -Xloggc:gc.log, -XX:+PrintGCDetails, -XX:
+PrintTenuringDistribution, -XX:+PrintGCTimeStamps, -XX:
+PrintHeapAtGC, -XX:+PrintGCApplicationStoppedTime**

- If you need more “oomph” from your application...
- Look first at the architecture and environment
 - Include the users (actors) and how they interact with the application
 - If you use transactions, you probably don't need them
 - Get rid of ORM - look into NoSQL
- Then the code
 - Look for excessive use of Strings and objects being passed around
- Finally the JVM
 - Other than simple tuning this should be your last stop