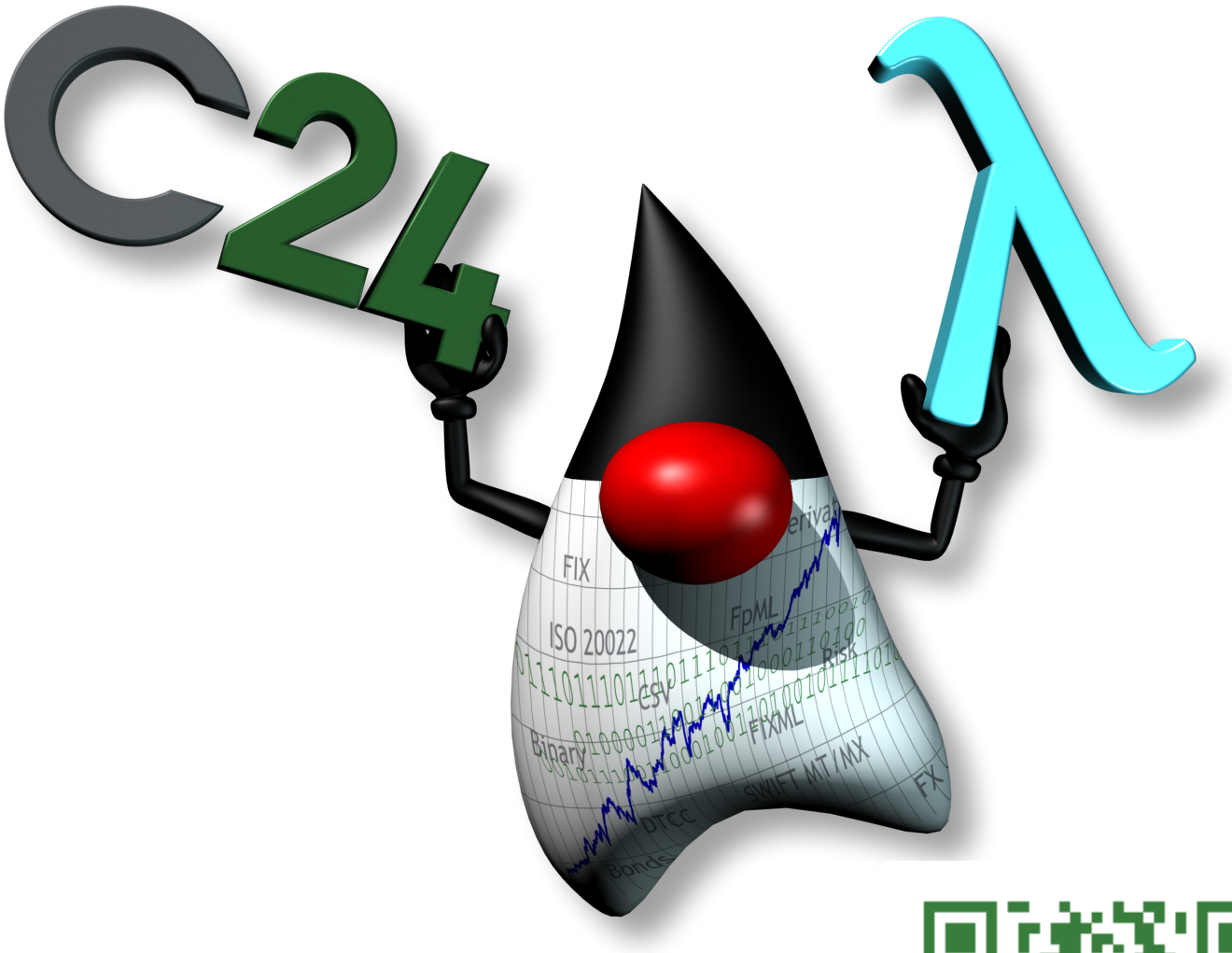
Java 8 for Financial Services

Java 8 isn't just that latest software gadget from Oracle, it can vastly simplify your code and even make it run faster



John Davies
CTO
C24 Technologies, 2014

New Features of Java 8 - Relevant to FS

I'm a great fan of the latest gadgets but Java 8 brings more than just new gadgets to Java. With functional programming in the form of lambdas making its debut in Java 8, this is the biggest change to the language since generics. I have worked in financial services for well over 25 years now and the move from Java 7 to Java 8 is almost as exciting as it was moving from C++ to Java itself way back in '95.

I'm not going to go through the entire list of new features, you can find that here:

http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Let me briefly cover some of the main points from a financial service perspective.

Java Programming Language

As I mentioned there has been a huge change to the language in the form of lambdas, and this is going to be the main topic of this paper.

Collections

There's no obvious difference to Collections until you start to work with the java.util.stream package. New methods have been cleverly added to Collections using new "default" methods enabling them to be used with streams. "Default" allows new methods to be added to Collections without changing backward compatibility in a really quite clever way.

Security

It would be difficult to pass "security" without a mention, especially for work with financial systems. However most financial services applications run behind physical security and rarely have any connection to the outside world. Retail banking is a little different and I'm sure a lot of the new cryptography and encryption support will be invaluable for those trying to keep out undesirables. It is certainly true that for desktop/browser apps the security upgrade will be vital.

JavaFX

I personally really like JavaFX. It's been around since Java 6 technically, but it's come on a lot. I'm not a good screen designer so I stick to server-side code. JavaFX is a huge change from Swing allowing you to design the layout (using the Screen Builder tool) separately from the functionality. I've now created a nice library of simple frameworks that I use to wrap demos that would otherwise be command-line printf() statements. I can display GC or performance in graphs with just a few lines of code. I will include one example at the end of this paper just to inspire anyone who's not seen what it can do yet. The biggest change in Java 8 is the addition of the Webkit engine which could make a huge difference to internally delivered apps.

Date-Time Package

This is another one of those packages that should have been changed a long time ago. Dates and time are critical in finance and so much has had to be added just to cope with the most basic of features like timezones and ISO-8601 used in XML. If you're familiar with JodaTime then you'll be at home with this package. Similarly, classes are immutable too. I'll try to add a few examples below but again my goal is not to cover the entire Java 8 feature list.

JDBC

Seriously, does anyone still use RDBMSs? OK, sorry, you're in a bank, of course you do. You'll be delighted to know the JDBC-ODBC bridge has been removed and we have some new features in JDBC 4.2 like SQLType and JDBCType but unless you're a die-hard DBA you should really be limiting your exposure and dependence on JDBC these days. If you really need the features of JDBC 4.2 then you're very likely going to be locking yourselves into many more years of RDBMS when most modern banks have been exploring "NoSQL" solutions for years now.

Concurrency

Concurrency is probably the most widely used and least understood library of Java. Since the re-write in Java 1.5 it's now up with almost any of the other languages for multithreaded work. The biggest changes in Java 8 are probably the ForkJoinTask used under the hood with parallelStreams and the new Atomic Adders and Accumulators. I would be surprised if anyone uses these low-level APIs in their raw forms as they're usually best used through abstraction layers such as Vert.x or Akka. The latter is getting a lot of coverage in the banking world so Java 8 is going to add some great performance and reliability to existing AKKA based applications. I was temped to include AKKA in this but it deserves its own paper. I will however demonstrate the great ease with which you can add concurrency through "parallel" or "parallelStream" in steams to get a very appreciable performance boost.

HotSpot

There are 3 biggies in Java 8: tiered compilation, Metaspace and G1GC. Garbage 1st Garbage Collector was formally released in the 1.7.0 but in 1.8.0 it becomes a viable alternative to the mostly concurrent collector and parallel collectors we've become familiar with. G1GC has been designed to scale beyond the 4GB heap size ceiling that the other collectors tend to run into. It's not something that your average programmer is going to notice but for architects, at least the hands-on ones (real ones) and ops guys this will be a big win.

For those who have been plagued with OutOfMemoryError PermGen exhausted you may be pleased to hear that you will suffer no more as Permspace has been removed. The usual occupants of Permspace have been dispersed into normal Java heap or into one of two new data structures known as Metaspace or Classspace. However as is often the case, the devil is in the details as this change does not remove the underlying cause of OOME in perm, it just moves it about in a way that should lead to fewer instances of OOME.

Finally the Hotspot compilers now offer by default tiered compilation. Tiered compilation is a mode that combines the quick simple optimisations you see with the C1 or client Hotspot engine with the deeper more complex optimisations that can be calculated by the C2 or

optimising Hotspot engine. Though this has been in the works for quite some time it is relatively untested in a wide range of production environments so you may want to test it against the more familiar -server option to make sure it delivers the desired effects.

Java Mission Control

Mission Control is Oracle's [reasonably good] attempt at providing additional value add for corporate licenses. You can use it out of the box in Java 8 but if you rely on it for production then you're going to need to pay Oracle some (more) money. You can still find jVisualVM as its still bundled with the JDK.

What are Lambdas?

Lambdas are very simply anonymous functions. Until Java 8 "functions" were methods which had to be associated with a class, sometimes the class serves no useful purpose. Lambdas allow us to create little classless methods with some "magical" typing, like generics but better. There is nothing you can do with lambdas that you can't do without but you can almost always make the code look simpler and neater and in many cases the compiler/JVM will make clever optimisations based on your lambdas.
Think of a background logger where we need a little Runnable…

```
Runtime runtime = Runtime.getRuntime();
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

Runnable oldTask = new Runnable() {
        @Override
        public void run() {
                System.out.printf( "Memory used: %,.3fGb\r",
                        (runtime.totalMemory() - runtime.freeMemory())/1073741824.);
        }
};

scheduler.scheduleAtFixedRate(oldTask, 0, 1, TimeUnit.SECONDS);
```

The class structure of the actual Runnable we need serves no useful purpose in the code so with a lambda we can shorten this to the following (I've changed the red bit to the green bit).

```
Runtime runtime = Runtime.getRuntime();
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
Runnable task = () -> System.out.printf( "Memory used: %,.3fGb\r",
        (runtime.totalMemory() - runtime.freeMemory())/1073741824.);
scheduler.scheduleAtFixedRate(task, 0, 1, TimeUnit.SECONDS);
```

Listeners for asynchronous actions are also a perfect place for lambdas, no more anonymous inner classes for your actionListeners.

What we're doing in this paper

To move forward with this paper I'm going to do roughly the following…

- Start with some simple CSV data from an Excel spreadsheet,import it into a Java model and run some lambdas on the Java model containing the data
- Generate more of the same data by randomising it; this way we can generate a few million instead of just the 10 lines from above - better for sort demonstrations
- Finally we will dispense with the simple data and use more complex XML data from FpML, randomise that and run similar lambdas
- Lastly as a close we'll take a quick look at some of the tools and a very last page on how to show off the results in JavaFX.

Setup - Read in some test data…

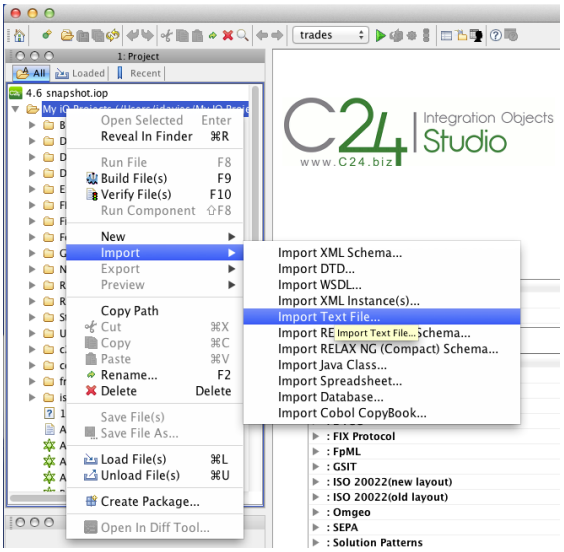
Let's start with a data set we can play with, something we can refer to for the rest of this paper. It's simple so that we can keep the example simple but later on I'll show you how everything we do with the simple version we can also do with more complex data sources such as FpML, ISO-20022, SWIFT or FIX.

First the data…

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

It's purely fictitious data, made up in 5 minutes on Excel but we're going to use this for the examples. Now we need this in Java. We have a few choices; I could hand code the Trade class and hard-wire in the data, OK for a demo but not great for anything else. I could write a quick parser to read the CSV in but now we're talking about quite a bit of code before we can start playing. My preferred way and what I recommend is to use C24's Integration Objects (download free from ***http://www.c24.biz/downloads***) and simply Java Bind the CSV or XLS to generate the Java in seconds.

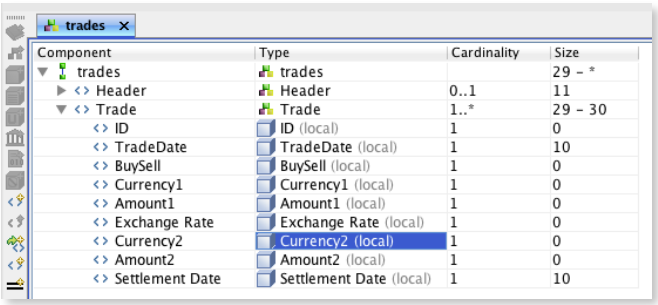
Diversion - Importing the CSV into C24 iO

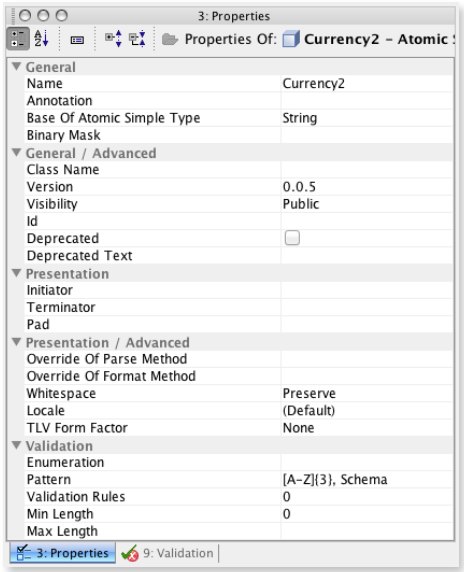


Starting by importing the CSV file Import (via “Import Text File…”), this will create a model based on the file.

You will get a wizard asking you about the file. This is very similar to importing something into Excel - usually the defaults work but you can change everything once it’s imported. During the import wizard there are far less options for data types for example, but once it’s imported you get literally hundreds for data-type options, even types from previously imported standards (such as FpML or ISO-20022).

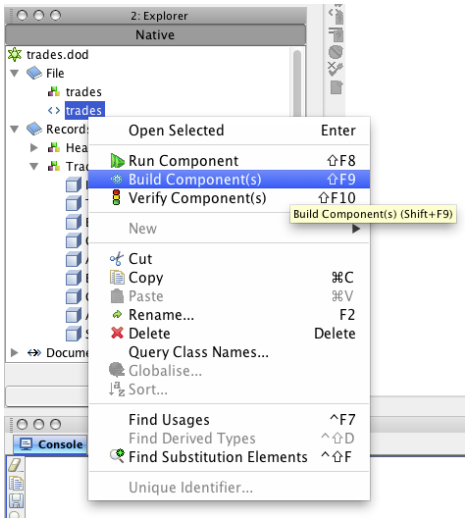
Once you have a model (right), you now have the opportunity to clean it up. You may find for example that the amounts are “doubles” but you may prefer BigDecimal so this is where to make those changes. You could also further restrict the currencies to 3 uppercase letters so you would add a “[A-Z]{3}” to the validation pattern (example below).





Once you’ve “tuned” your model, you’re ready to deploy the Java binding for the CSV file. This is simply a case of right clicking on the root of the model or the part you want to deploy and selecting “Build Component(s)”. This will create you the Java binding code, an ANT file to build it, a Spring config file (for use in Spring Integration) and finally compile it all into a Jar.

Any changes and you can go back to the model, make the change and re-deploy the code.



From here we can start to play with the code.

Creating and using a Java 8 Stream

Streams are a new way to work with data, as the name would suggest, as streams. Again like lambdas here is nothing you can do with Streams that you can't do without them. It does however make your code a lot simpler and, as long as you understand it, easer to read and maintain. Lastly using these new constructs gives a lot more information to the compiler and JVM so further improvements can be made at runtime giving you better performance. There are a few gotchas like exception handling, debugging and infinite streams but we'll cover those as we go along.

Take a list of currencies, we want to perhaps filter or print them out, I'm sure you can think of a dozen ways of doing it with arrays, Lists, Collections, iterators, for-loops etc. Here's the Streams version or should I say a Stream version…

```
Stream<String> currencies = Stream.of("GBP", "EUR", "USD", "CAD", "AUD", "JPY", "HKD" );
currencies.forEach( ccy -> System.out.println( ccy ) );
```

If you've immediately spotted the forEach and said there's a simpler way of doing that you're right, I wanted to show something that's a little easier to understand. So we create a variable "ccy" which we can call anything, typically we just use a single letter, then define what applies to that variable.

These are all equally valid…

```
currencies.forEach( currency -> System.out.println( currency ) );
```

```
currencies.forEach( c -> System.out.println( c ) );
```

```
currencies.forEach( x -> System.out.println( x ) );
```

I'm a great fan of self-documenting code so on those grounds I should really prefer the first version with "currency" but to be honest I'm starting to get used to the single letter version. My advice would be to just choose a letter than makes sense, "c" makes more sense to me here than "x". What's interesting and worth pointing out is that in a stream the variable is magically typed to the items in the stream, in this case Stream elements (in the <>) so a String.

We can actually go one stage further with this lambda and use a method reference, the "::" syntax refers to the method (which can be a constructor, static or and instance method) to be applied to each of the elements.

```
currencies.forEach( System.out::println );
```

Now let's try a filter…

```
Stream<String> currencies = Stream.of("GBP", "EUR", "USD", "CAD", "AUD", "JPY", "HKD" );
       currencies
       .filter( c -> c.matches( "GBP|EUR"))
       .forEach( System.out::println );
```

The output of this is basically GBP and EUR, we could try this too…

```
currencies
        .filter( c -> c.contains("A"))
        .forEach( System.out::println );
```

And we get CAD and AUD.

Let's quickly move on to our trade data and start really using the streams and lambdas…

Reading in Trades as Streams

Reading our CSV test data in code is just a single line, it's read in and populated by the bound code that was generated by the C24 process above. We now have a Trades object that actually has an optional Header and an array of Trade objects.

```
Trades tradeData = C24.parse(Trades.class).from(new File("tradedata.csv"));
ArrayList<Trade> tradesList = new ArrayList<>(Arrays.asList(tradeData.getTrade()));
Stream<Trade> tradeStream = tradesList.stream();
tradeStream.forEach(System.out::print);
```

You may remember (assuming you read the intro) that we had some clients that were still on Java 1.4, well for that reason we still deploy arrays[] rather than Collection classes, this is changing as we move to a minimum of Java 1.6 and even introduce an optional Java 1.8 deployment with native Streams and new Date/Time.

What I've done here is to create a List from the array[] using the static Arrays.asList method and as a result we get a List of Trade objects, these are the lines of Trade data (minus the header of course). Finally we get the stream and then do what we did above and print them out.

```
1,21/07/2014,Buy,EUR,50000000,1.344,USD,67200000,28/07/2014
2,21/07/2014,Sell,USD,35000000,0.744,EUR,26043500,20/08/2014
3,22/07/2014,Buy,GBP,7000000,172.99,JPY,1210930000,05/08/2014
4,23/07/2014,Sell,AUD,13500000,0.941,USD,12700800,22/08/2014
5,24/07/2014,Buy,EUR,11000000,1.215,CHF,13362800,31/07/2014
6,24/07/2014,Buy,CHF,6000000,0.651,GBP,3907800,31/07/2014
7,25/07/2014,Sell,JPY,150000000,0.651,EUR,97695000,08/08/2014
8,25/07/2014,Sell,CAD,17500000,0.902,USD,15793750,01/08/2014
9,28/07/2014,Buy,GBP,7000000,1.837,CAD,12856200,27/08/2014
10,28/07/2014,Buy,EUR,13500000,0.791,GBP,10679850,11/08/2014
```

Let's apply a filter…

```
tradeStream
        .filter( t -> t.getID() == 9 )
        .forEach(System.out::print);
```

and we get just the row starting with number 9.

```
9,28/07/2014,Buy,GBP,7000000,1.837,CAD,12856200,27/08/2014
```

Let's try the currencies…

```
tradeStream
        .filter( t -> t.getCurrency1().matches("GBP|EUR") )
        .forEach(System.out::print);
```

```
1,21/07/2014,Buy,EUR,50000000,1.344,USD,67200000,28/07/2014
3,22/07/2014,Buy,GBP,7000000,172.99,JPY,1210930000,05/08/2014
5,24/07/2014,Buy,EUR,11000000,1.215,CHF,13362800,31/07/2014
9,28/07/2014,Buy,GBP,7000000,1.837,CAD,12856200,27/08/2014
10,28/07/2014,Buy,EUR,13500000,0.791,GBP,10679850,11/08/2014
```

NB: I'm using "t ->" in my lambdas rather than "trade ->" simply because in most cases it fits more easily on to one line, no other reason.

OK now we just want to count the number of "Buy" trades…

```
long count = tradeStream
        .filter(t -> t.getBuySell().matches("Buy"))
        .count();

System.out.printf("count = %d%n", count);
```

And we get "`count = 6`", now let's get the sum of all the GBP trades…

```
BigDecimal total = tradeStream
        .filter(t -> t.getCurrency1().matches("GBP"))
        .map(t -> t.getAmount1())
        .reduce(BigDecimal.ZERO, BigDecimal::add);

System.out.printf("total = %,d", total.intValue());
```

Firstly a filter to apply just to the GBP values (yes we've ignored the other side of the trade for simplicity), then the "map" basically turns the stream from a stream of Trade objects to a stream of BigDecimal objects, the output of getAmount1(). Finally reduce() initialises itself with the first value BigDecimal.ZERO and then performs the BigDecimal::add method on each member of the stream. And the result, as, hopefully expected…

```
total = 14,000,000
```

We could have done this a number of different ways. The following gets the double value from the amount1() and then uses a slightly different stream which adds a sum() method. The warning I would give over this is that we're casting to double which is not good for financial operations. The result, for this small demo is luckily the same but with larger volumes we could see errors accumulating with the rounding and precision of double.

```
double total = tradeStream
        .filter(t -> t.getCurrency1().matches("GBP"))
        .mapToDouble(t -> t.getAmount1().doubleValue())
        .sum();

System.out.printf("total = %,d", total);
```

Our trades are sorted but let's shuffle them up a bit so that we can demo the sort…

```
Trades tradeData = C24.parse(Trades.class).from(new File(fileName));
ArrayList<Trade> tradesList = new ArrayList<>(Arrays.asList(tradeData.getTrade()));
Collections.shuffle(tradesList);
Stream<Trade> tradeStream = tradesList.stream();

tradeStream.sorted( Comparator.comparing(Trade::getID) )
        .forEach(System.out::print);
```

I've highlighted the two lines, first the standard (old) Java shuffle and then the new sort passing in the attribute or column I want to sort on.

Let's look at a few other features before we get into higher volumes and more complex messages. We have predicates where we can see if any or all of the trades match the predicate, let's check that all the amount calculations are correct…

```
boolean match = tradeStream
        .allMatch(t -> t.getAmount1().multiply(BigDecimal.valueOf(t.getExchangeRate()))
                .compareTo(t.getAmount2()) == 0);
System.out.println("allMatch = " + match);
```

Stream.allMatch() simply runs the predicate agains all of the items in the stream and returns a boolean, true if they all match. What we're doing here is checking that the first amount multiplied by the exchange rate is equal to the second amount. We use BigDecimal here because that's just what you do in financial services, we have to have precise control over every penny or cent and IEEE double can give us a few errors after a while.

We could also put the predicate into a method to re-use it another time…

```
private static Predicate<Trade> rateCheck() {
        return t -> t.getAmount1().multiply(BigDecimal.valueOf(t.getExchangeRate()))
                .compareTo(t.getAmount2()) == 0;
}
```

and then just call it…

```
        boolean match = tradeStream.allMatch(rateCheck());
        System.out.println("allMatch = " + match);
```

Note that I use static here purely because I'm writing the code in main() for this paper, no other reason.

The predicate could also be part of the Trade object but we can also reuse a validation method on the Trade object and the predicate would simply be that the result of the isValidate() method is valid or true. C24 provides very powerful validation built into the models, particularly useful for FpML, FIX, ISO-20022, SWIFT and other standards requiring complex semantic validation in addition to syntactic validation.

We have noneMatch()…

```
boolean match = tradeStream
        .noneMatch( t -> t.getTradeDate().getTime() > t.getSettlementDate().getTime());
System.out.println("allMatch = " + match);
```

This just checks that none of the trades have a trade date greater than their settlement date, not using the new Java Date classes but the good ol' java.util.Date. Finally there's anyMatch() but hopefully by now you're getting the picture.

Higher volumes of Trades

Let's drop the example file of just 10 trades and use another useful method called generate() to create a larger stream. The reason we're doing this is to demonstrate the performance enhancements we can get from using parallel operations.

First we need something that creates new Trade objects, I've basically randomised the content of the trade in a new method createTrade(). To fit it on one page I've taken the comments out but I think you'll find it largely understandable with just pure code. It goes through each of the fields in the trade and creates a new random one. For the currencies we need to make sure the second isn't the same as the first so we loop until its different and do the same with the tradeDate making sure it's not a weekend. Finally I've used fixed values for the exchange rate but randomised them slightly using a Gaussian distribution with a standard deviation of 0.5% and then limited it to 5 significant figures.

Running this we get something like this, obviously each time we run it it's different, it's random of course…

```
1,25/08/2014,Buy,CAD,3000000,0.82775,CHF,2483250,15/09/2014
2,27/08/2014,Sell,GBP,14000000,1.84579,CAD,25841060,03/09/2014
3,01/08/2014,Buy,CHF,17000000,0.65699,GBP,11168830,22/08/2014
4,14/08/2014,Buy,CHF,24000000,1.18559,AUD,28454160,04/09/2014
5,19/08/2014,Sell,AUD,7000000,0.68886,EUR,4822020,02/09/2014
```

The code is below…

```
private static Map<String, Double> currencies = null;
private static LongAdder adder = new LongAdder();

private static Trade createTrade() {
        if( currencies == null ) {
                currencies = new HashMap<>(7);
                currencies.put("GBP", 1.0);
                currencies.put("EUR", 1.2521);
                currencies.put("USD", 1.6818);
                currencies.put("AUD", 1.8061);
                currencies.put("CHF", 1.5240);
                currencies.put("JPY", 172.54);
                currencies.put("CAD", 1.8362);
        }
        Trade trade = new Trade();
        Random randomGen = new Random();

        adder.add(1);
        trade.setID( adder.intValue() );

        LocalDate startingDate = LocalDate.of(2014, Month.AUGUST, 1);
        LocalDate tradeDate;
        do {
                tradeDate = startingDate.plusDays(randomGen.nextInt(startingDate.lengthOfMonth()-1));
        } while ( tradeDate.getDayOfWeek() == DayOfWeek.SUNDAY ||
                tradeDate.getDayOfWeek() == DayOfWeek.SATURDAY );

        trade.setTradeDate( Date.from(tradeDate.atStartOfDay()      // Using a little Java 8 Date
                .atZone(ZoneId.systemDefault()).toInstant()));      // and Instant
```

```
        trade.setBuySell(randomGen.nextBoolean() ? "Buy" : "Sell");

        String[] currencyArray = currencies.keySet().toArray(new String[0]);
        trade.setCurrency1(currencyArray[randomGen.nextInt(currencyArray.length)]);

        do {
                trade.setCurrency2(currencyArray[randomGen.nextInt(currencyArray.length)]);
        } while( trade.getCurrency2().equals( trade.getCurrency1() ));

        trade.setAmount1(BigDecimal.valueOf((randomGen.nextInt(50) + 1) * 1_000_000));

        double rate = currencies.get(trade.getCurrency2()) / currencies.get(trade.getCurrency1());

        rate *= (1.0 + randomGen.nextGaussian()/200.0);

        rate = BigDecimal.valueOf(rate).setScale(5, BigDecimal.ROUND_UP).doubleValue();
        trade.setExchangeRate(rate);
        trade.setAmount2(trade.getAmount1().multiply(BigDecimal.valueOf(trade.getExchangeRate())));

       LocalDate settmentDate = tradeDate.plusDays(7 * (randomGen.nextInt(3) + 1));

        trade.setSettlementDate( Date.from(settmentDate.atStartOfDay()
                .atZone(ZoneId.systemDefault()).toInstant()) );

        return trade;
}
```

Creating a stream from this is very simple, we can use Stream.generate()

```
Stream<Trade> tradeStream = Stream.generate(() -> {
        return createTrade();
});
```

Using this stream of randomly generated trades we can do everything we did above on the small sample. Note however that the results that I print here will not necessarily be the same as yours. There is one catch though, if you were to run this…

```
tradeStream.forEach(System.out::print);
```

you would have a lot of output and it simply wouldn't end. Similarly if we were to calculate the sum or count the number of items we'd never return a result so we need to limit the stream's output; limit(n) does the job nicely.

```
tradeStream
        .limit(100)
        .forEach(System.out::print);
```

Now that we can generate a larger number let's get a list of 1,000 trades of just Buy/Sell GBP to USD. I am using a Collector this time to collect all the results into a List using toList(). One reason, apart from demonstrating it here, is that we can use the result more than once as we print out the results. The down side is that each Trade is now stored in memory and we've lost one of the advantages of streams.

```
List<Trade> gbp2usdTradeList = tradeStream
        .filter(t -> t.getCurrency1().matches("GBP") && t.getCurrency2().matches("USD"))
        .limit(1_000)
        .collect(Collectors.toList());
```

And to print out the first 3 and last three…

```
gbp2usdTradeList.stream().limit(3).forEach(System.out::print);
System.out.println("...");
gbp2usdTradeList.stream().skip(997).forEach(System.out::print);
```

We get, or at least I get (as yours will have different numbers)…

```
20,28/08/2014,Sell,GBP,34000000,1.68473,USD,57280820,11/09/2014
29,11/08/2014,Sell,GBP,18000000,1.69772,USD,30558960,18/08/2014
39,07/08/2014,Buy,GBP,13000000,1.68216,USD,21868080,21/08/2014
...
40594,26/08/2014,Buy,GBP,33000000,1.67706,USD,55342980,02/09/2014
40631,29/08/2014,Buy,GBP,40000000,1.69239,USD,67695600,12/09/2014
40672,07/08/2014,Buy,GBP,40000000,1.68191,USD,67276400,14/08/2014
```

Just as a sideline, as I like to check my results, given there are 7 currencies the odds on the first one being GBP is 1/7 and then the odds on the second being USD is 1/6 so the probability of GBP/USD is 1/42 and as we can see we generated roughly 42,000 trades to get 1,000 GBP/USD examples. Just for fun, running this with limit(1_000_000) gave the last tradeId of 41,913,060, just 0.2% out.

Let's test the parallel sorting now, the data is already sorted by tradeId and the amounts are not terribly unique so let's sort on the exchange rate, first serially (not in parallel)…

```
long start = System.nanoTime();
tradeStream
        .filter(t -> t.getCurrency1().matches("GBP") && t.getCurrency2().matches("USD"))
        .limit(1_000_000)
        .sorted(Comparator.comparing(Trade::getExchangeRate))
        .limit(3)
        .forEach(System.out::print);

System.out.printf("time = %.3f%n", (System.nanoTime() - start) / 1e9);
```

And I get…

```
37387422,29/08/2014,Sell,GBP,18000000,1.64245,USD,29564100,05/09/2014
16612950,21/08/2014,Buy,GBP,11000000,1.6431,USD,18074100,28/08/2014
24092486,11/08/2014,Sell,GBP,18000000,1.64346,USD,29582280,18/08/2014
time = 91.153
```

And now adding the parallel()…

```
long start = System.nanoTime();
tradeStream
        .filter(t -> t.getCurrency1().matches("GBP") && t.getCurrency2().matches("USD"))
        .limit(1_000_000)
        .parallel()
        .sorted(Comparator.comparing(Trade::getExchangeRate))
        .limit(3)
        .forEach(System.out::print);

System.out.printf("time = %.3f%n", (System.nanoTime() - start) / 1e9);
```

I get…

```
23330640,25/08/2014,Buy,GBP,16000000,1.64217,USD,26274720,15/09/2014
31114616,29/08/2014,Buy,GBP,35000000,1.64179,USD,57462650,05/09/2014
7073144,22/08/2014,Sell,GBP,33000000,1.64487,USD,54280710,12/09/2014
time = 29.270
```

Again remember that each time I run this the Trades are generated so the results will not be the same, at the sort of volumes we're working with though, 1 million trades from 42 million (roughly) generated everything time-wise certainly is going to be averaged out.

My machine is a 4 core (hyper-threaded) MacBookPro so this 3 fold performance increase is about what I'd expect and impressive going for adding just one method call. What's happening behind the scenes is the new fork/join is being used. It's worth pointing out that I wouldn't see this sort of gain if I hadn't first filtered the data simply because the bottleneck would have been the stream generate.

Before we jump into a total distraction of JavaFX let's take a look at some more complex stream and lambda operations…

Let's count the number of each currency pair using a groupBy operation, this is similar to what you'd do in SQL…

```
select CCY1,CCY2,count(*) from Trades group by CCY1,CCY2
```

Now in Java using Streams and lambdas…

```
Map<String, Long> map = tradeStream
        .limit(1_000_000)
        .collect(Collectors.groupingBy(t -> t.getCurrency1() + "/" + t.getCurrency2(),
                Collectors.counting()));

System.out.println("map = " + map);
```

And we get (or at least I got)…

```
map = {AUD/JPY=23816, USD/JPY=23706, AUD/GBP=23949, USD/GBP=23745, CHF/GBP=23666, JPY/CHF=23864, EUR/
CAD=23934, CHF/JPY=23844, CHF/AUD=24077, EUR/USD=23934, USD/AUD=23982, GBP/EUR=23564, EUR/AUD=23568,
USD/EUR=23606, GBP/CAD=23735, GBP/USD=23676, JPY/GBP=23551, EUR/JPY=24097, USD/CAD=23791, CHF/
USD=23738, AUD/CHF=23869, CHF/CAD=23903, CAD/CHF=23875, JPY/AUD=23759, CHF/EUR=23780, EUR/GBP=23975,
GBP/AUD=23831, GBP/JPY=23606, CAD/AUD=23752, JPY/USD=23773, JPY/CAD=24081, EUR/CHF=23860, CAD/
JPY=24001, JPY/EUR=23783, CAD/GBP=23835, USD/CHF=23770, AUD/USD=23900, AUD/CAD=23799, AUD/EUR=23969,
CAD/EUR=23566, CAD/USD=23543, GBP/CHF=23927}
```

The groupingBy() creates a map, in this case Map<String, Long>, the String comes from the groupingBy() and the Long from the Collectors.counting().

A little further now, we'll groupBy currency (just Currency1) and then groupBy Buy/Sell and finally aggregate the amounts (Amount1).

```
Map<String, Map<Object, BigDecimal>> map = tradeStream
        .limit(1_000_000)
        .collect(
                Collectors.groupingBy(t -> t.getCurrency1(),
                        Collectors.groupingBy(t -> t.getBuySell(),
                                Collectors.reducing(
                                        BigDecimal.ZERO, Trade::getAmount1, BigDecimal::add))));

System.out.println("map = " + map);
```

And the output…

```
map = {AUD={Sell=1826959000000, Buy=1822442000000}, CHF={Sell=1818975000000, Buy=1823776000000},
JPY={Sell=1826692000000, Buy=1812326000000}, EUR={Sell=1828203000000, Buy=1824140000000},
GBP={Sell=1807283000000, Buy=1818057000000}, CAD={Sell=1818615000000, Buy=1826496000000},
USD={Sell=1817626000000, Buy=1820617000000}}
```

If you were wondering how you might debug this, here's a tip. Use peek() but remember that you can't put conditionals in so you can have `if(t.something() < 5) print(t)`. The best plan is to have a method to do that like so…

```
Map<String, Map<Object, BigDecimal>> map = tradeStream
        .limit(1_000_000)
        .peek( t -> occasionallyDebug(t) )
        .collect( Collectors.groupingBy(t -> t.getCurrency1(),
```

And the method/function…

```
private static void occasionallyDebug( Trade trade ) {
        if( trade.getID() % 100_000 == 0 ) {
                System.out.print("DEBUG: " + trade);
        }
}
```

In real life we could use this calculation for position keeping. We could do it by counter-party, by currency and of course by date.

High volumes and complex XML messages

So far we played with simple trade models. It's usually the easiest way to understand, but we're now going to step things up and move to real trades, defined in FpML. Well when I say "real" I mean real-looking, trades, naturally we're going to have to randomise them again.

This is the example I'm going to use, it's several pages so I won't waste space here printing it…

http://www.fpml.org/spec/fpml-5-6-3-tr-1/html/confirmation/xml/products/interest-rate-derivatives/ird-ex01-vanilla-swap.xml

In the TradeHeader I'm going to change the two TradeId values from TW9235 and SW2000 to "Party1-1234" and "Party2-1234" where the "1234" is the index of the generated message and I'm going to add a random date (again a weekday) from 2013 into the TradeDate. Then I'm going to randomise the InitialValue with a value from 0 to 10 million (with 2 decimal places). This occurs in two areas (two of the SwapStreams) so both will be changed.

```
- <trade>
  - <tradeHeader>
    - <partyTradeIdentifier>
        <partyReference href="party1"/>
        <tradeId tradeIdScheme="http://www.partyA.com/swaps/trade-id">TW9235</tradeId>
      </partyTradeIdentifier>
    - <partyTradeIdentifier>
        <partyReference href="party2"/>
        <tradeId tradeIdScheme="http://www.barclays.com/swaps/trade-id">SW2000</tradeId>
      </partyTradeIdentifier>
      <tradeDate>1994-12-12</tradeDate>
    </tradeHeader>
  - <swap>
```

```
- <calculationPeriodAmount>
  - <calculation>
    - <notionalSchedule>
      - <notionalStepSchedule>
          <initialValue>50000000.00</initialValue>
          <currency currencyScheme="http://www.fpml.org/coding-scheme/external/iso4217">EUR</currency>
        </notionalStepSchedule>
      </notionalSchedule>
    - <fixedRateSchedule>
        <initialValue>0.06</initialValue>
      </fixedRateSchedule>
      <dayCountFraction>30E/360</dayCountFraction>
    </calculation>
  </calculationPeriodAmount>
```

That is all I will randomise for this though, as any other values just make it pointless for what we're going to look at.

FpML is pretty complex; messages can have 13 levels of hierarchy which is why I didn't start with it. Using Java is far easier than a relational database for this sort of thing. We can use a hierarchical XML binding to work with the XML. We could also do this with XQuery and XPath but they are both very XML centric languages and not why we're here. Inside FpML are several substitution groups, these are a little like references to an interface where the implementation is defined at run-time so we have to also navigate these as well as sometimes cast interfaces to concrete classes in order to use the right getters. We have a lot more about this on our web site so we'll skip any more detail at this point.

Reading the FpML template (the one in the link) is very simple, we use the same API as with the CSV file…

```
File XML_INPUT_FILE = new File("valid-ird-ex01-vanilla-swap.xml");

Fpmlmain54DocumentRoot message = C24.parse(Fpmlmain54DocumentRoot.class).from(XML_INPUT_FILE);
```

Setting the trade date…

```
Trade trade = cdo.getDataDocument().getDataDocumentSG1().getTrade()[0];
trade.getTradeHeader().getTradeDate().setValue(new ISO8601Date(tradeDate.toString()));
```

Setting the two initial values…

```
Swap swap = (Swap) trade.getProduct();

BigDecimal value = BigDecimal.valueOf(Math.random() * 10_000_000).setScale(2, BigDecimal.ROUND_UP);

swap.getSwapStream()[0].getCalculationPeriodAmount().getCalculation().getCalculationSG1()
        .getNotionalSchedule().getNotionalStepSchedule().setInitialValue(value);

swap.getSwapStream()[1].getCalculationPeriodAmount().getCalculation().getCalculationSG1()
        .getNotionalSchedule().getNotionalStepSchedule().setInitialValue(value);
```

Naturally we could write a little method to hide some of this complexity which is exactly what I did for the lambdas we're going to use in a few paragraphs.

```
private void setInitialValue( Fpmlmain54DocumentRoot message, int index, BigDecimal value ) {
        Swap swap = (Swap) message.getDataDocument().getDataDocumentSG1().getTrade()[0].getProduct();
        swap.getSwapStream()[index].getCalculationPeriodAmount().getCalculation().getCalculationSG1()
                .getNotionalSchedule().getNotionalStepSchedule().setInitialValue(value);
}
```

I should also point out that this helper method can actually be added the the FpML model in C24's studio, meaning that we can add a "virtual" InitialValue to the root of the message with getters and setters, similar to below. This vastly simplifies the code, both for traditional Java and our new lambdas. We can now do the following…

```
message.setInitialValue( 0, value );
BigDecimal value = message.getInitialValue( 0 );
```

So we've got the message with randomised data we just need a few thousand of them now.To do that we duplicate them and add them to a List.

```
private static final int ARRAY_SIZE = 10_000;
private static List<Fpmlmain54DocumentRoot> messageList = new ArrayList<>(ARRAY_SIZE);

messageList.add((Fpmlmain54DocumentRoot) message.cloneDeep());
```

Let's start working with the messageList.

We're going to loop through the messages and count the number of trades with a value over 9.9 million, remembering that they're a lot more complex now.

```
long start = System.nanoTime();

long count = messageList.stream()
        .map(t -> t.getInitialValue(0))
        .filter(v -> v.compareTo(BigDecimal.valueOf(9_900_000)) > 0)
        .count();

System.out.println("count = " + count);

double seconds = (System.nanoTime() - start) / 1e9;
System.out.printf("Time to process: %,d messages: %,.3f seconds (%,.0f per second)%n%n",
        ARRAY_SIZE, seconds, ARRAY_SIZE / seconds);
```

I get the following with 10,000 FpML messages. I should point out that I didn't do any JIT warmup so it's just indicative.

```
count = 123
Time to run: 10,000 messages: 0.028 seconds (362,371 per second)
```

We'll come back to the performance in a second. Let's now try to sum all the trades from the month of July…

```
BigDecimal result = messageList.stream()
        .filter(t -> getTradeDate(t).getMonth() == 7)
        .map(t -> t.getInitialValue(0))
        .reduce(BigDecimal.ZERO, BigDecimal::add);
```

Again the performance is similar. What I'd like to do now is demonstrate the parallel performance. All we need to do is use a parallelStream()…

```
for( int loop = 0; loop < 10; loop++ ) {
        start = System.nanoTime();

        result = cdoList.parallelStream()
                .filter(t -> getTradeDate(t).getMonth() == 7)
                .map(t -> t.getInitialValue(0))
                .reduce(BigDecimal.ZERO, BigDecimal::add);
        seconds = (System.nanoTime() - start) / 1e9;
}
System.out.println("result = " + result);
System.out.printf("Time to process (parallel): %,d messages: %,.3f seconds (%,.0f per second)%n%n",
        ARRAY_SIZE, seconds, ARRAY_SIZE / seconds);
```

What I've done here to get a better timing result is to loop the test 10 times and just take the last result. Remember that if you're using -server in your JVM settings that the default is 10,000 iterations before the code is compiled by JIT.

I ran this serially and parallel with 100,000 messages…

```
result = 44656591648.06
Time to process (serial): 100,000 messages: 0.028 seconds (3,561,634 per second)

result = 44656591648.06
Time to process (parallel): 100,000 messages: 0.007 seconds (13,713,659 per second)
```

Reducing memory usage - SDOs

As you can see we have quite an impressive performance with the parallel stream. If you tried running this you may have noticed that you'd need quite a bit of RAM and some large -Xms/-Xmx settings. The reason for this is that the messageList requires all of the messages to be in memory. Binding FpML to Java results in message objects that are a good 15-25k in size, create 100,000 of these and we need a good 2 GB of RAM.

We believe we've solved this problem with a new Java binding technology that binds complex models directly to binary. With the code above but using this new binding (no change to the code, just the libraries) we can get over 40 times more messages into RAM. I was able to run the test above with up to 20 million FpML trades in memory on my laptop. If you're suffering with memory sizes and network bandwidth you may like to read more here…

http://ref.c24.biz/whitepapers/C24-SDOs-Big-Data-In-Memory.pdf

You can read more detail about SDOs with some of the more widely used caches here. This is not an exhaustive list; notably missing, mainly because we already have other papers in different areas are Redis, MongoDB and HazelCast. We have good working version if you need examples, please just contact us.simply because we haven't written the papers yet

http://ref.c24.biz/whitepapers/C24-SDOs-and-Coherence.pdf

http://ref.c24.biz/whitepapers/C24-SDOs-and-Ehcache.pdf

http://ref.c24.biz/whitepapers/C24-SDOs-and-GemFire.pdf

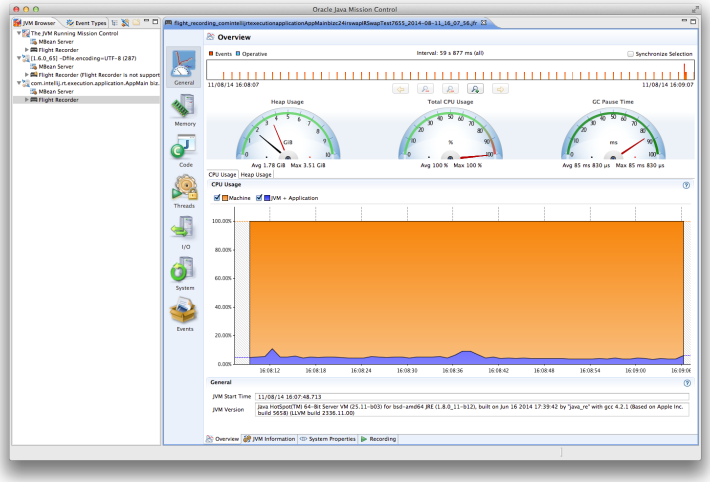
http://ref.c24.biz/whitepapers/C24-SDOs-and-GigaSpaces.pdf

Improvements in memory range from 20 to over 50 fold.

Analysing memory and GC performance

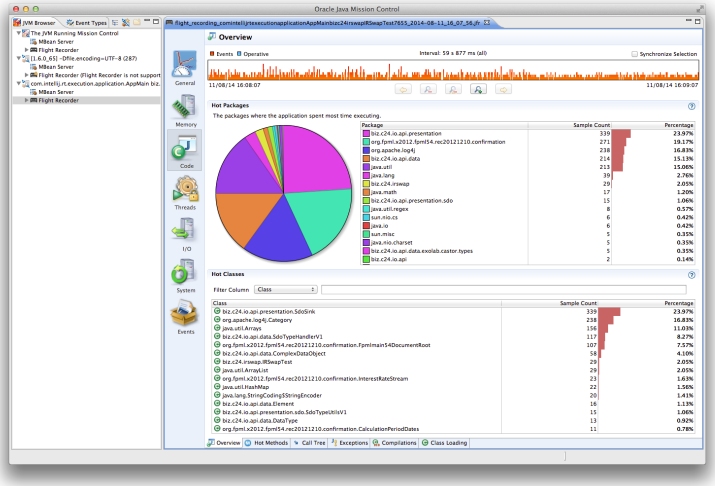
I said I'd briefly touch on GC, memory and performance measurement. I've used a number of tools, but what I tend to use now is the jVisualVM and Oracle's new Java Mission Control ("jmc" for short and on the command line) for code profiling and jClarity's Censum for GC profiling, read more here:

http://www.jclarity.com/censum



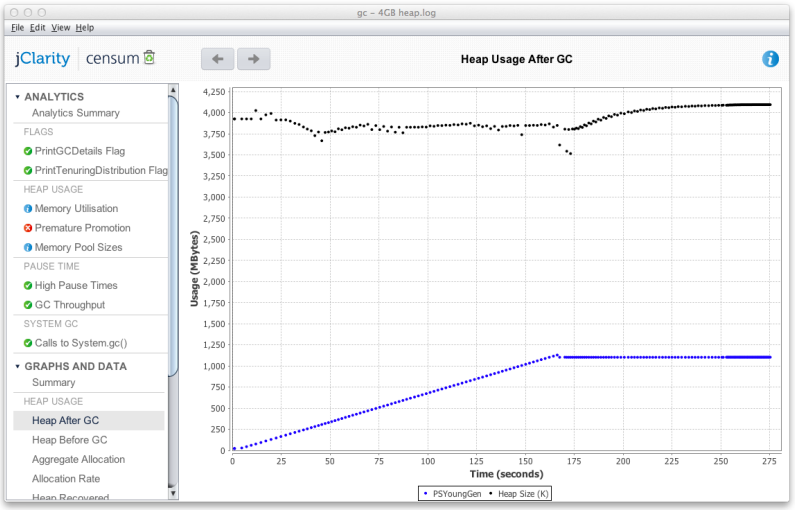
These are the only tools I found that give accurate results especially with Java 8 and the G1 GC.

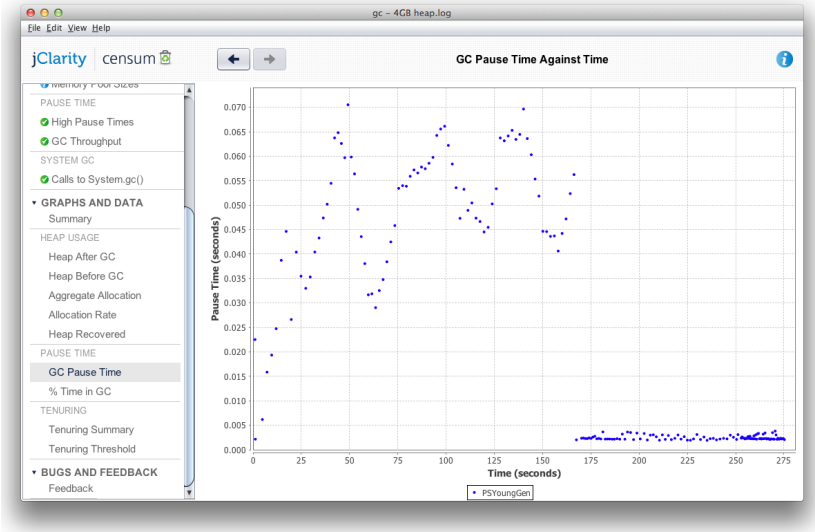
JMC is basically like jVisualVM on steroids. Oracle are obviously looking to make money from this and it looks like it may well become a useful tool. The licensing is a little strange though and I'm not even sure I should be putting screen shots in here. If they complain I'll simply not promote it.



You can use it without a license (with restrictions) but you do need to add a few -XX and -D parameters on startup. It is, as you can see, quite sexy though. I have to point out the gotcha with these sorts of tools though and that's basically that they seriously effect your runtime performance. For this reason I prefer to use them for code profiling and not performance baselining. Of course one leads to the other so you get there in the end. You can however get a very good idea of what's going on in your code. The level of detail is perfect for seeing where large amounts of memory are being allocated or parts of your code that are spending too long polling or waiting on IO.

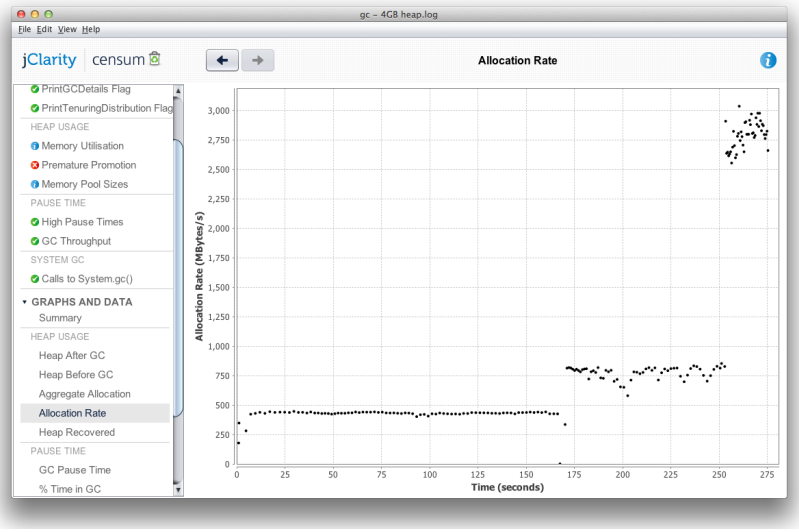
jClarity's Censum: this is a GC log file analyser - a few parameters in the JVM start up again and you can do a very neat post-mortem analysis of the log files. This is far less intrusive than run-time analytics and by far the best and most accurate mechanism I've found for looking at memory and GC behaviour. On the right you can see the post GC heap usage slowly climbing as we create the 2 million messages. Finally a satisfyingly flat plateau at about 1.08GB. This was using a 4GB heap, obviously the total usage doesn't change with the heap size but the performance does, especially the parallel performance.





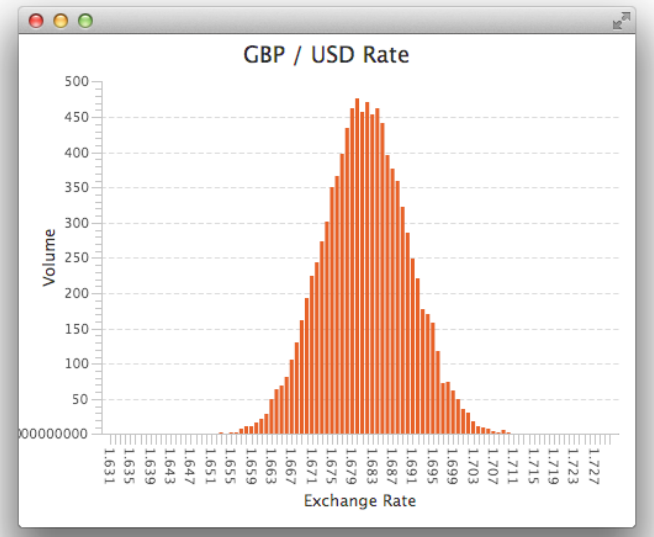
On the left you can see the GC pause time during 2 million tests. The first part "dancing around" up to about the 170 seconds mark is the data generation before the test. The pause times hit a maximum of 70ms during the data generation but remain in single milliseconds for the queries / searches which is good. Censum, combined with expertise from the jClarity guys is the perfect way to test out your code for performance.

Finally the allocation rate, creating the trades was more complex so the allocation rate is relatively low. Running the tests increased it to about 800MB/sec and then finally significantly higher during the parallel test. Normally I wouldn't want to see such a high allocation rate but in this case we're using an API that returns a BigDecimal which in Java terms is a monster. If the API returned a double or float then we'd see virtually zero allocation with the exception of anything created by the stream or lambdas.



Show it off in JavaFx

Lastly, just because I like it, I thought I'd show a little graph in JavaFx. This is so simple to do I really feel like it could be classified as a test as part of your TDD. After all much of your data is going to be consumed by humans at some point and this is just a quick way to check that it looks right. Looking at a stream of numbers will not tell you whether the distribution of your data is correct for example and I would be very surprised if you could write a test routine to tell me whether a data set is normally distributed without getting into fairly complex statistics and probability.



I used the code from above with a little groupingBy to put the exchange rates into groups of 0.001 resolution, 1.681 being the mean (and centre) and the values ranging from 1.631 in 100 steps of 0.001 to 1.731.

The Stream/Lambda code was simple…

```
Map<String, Long> map = tradeStream
        .filter(t -> t.getCurrency1().matches("GBP") && t.getCurrency2().matches("USD"))
        .limit(10_000)
        .collect(Collectors.groupingBy(t -> String.format("%.3f", t.getExchangeRate()),
                Collectors.counting()));
```

Just so that you can understand what we get from this it is the following…

```
map = {1.706=7, 1.707=6, 1.708=5, 1.709=3, 1.676=371, 1.677=385, 1.711=1, 1.678=422, 1.712=1,
1.679=483, 1.713=1, 1.714=1, 1.715=1, 1.670=208, 1.671=217, 1.672=252, 1.673=297, 1.674=287, 1.675=342,
1.718=1, 1.687=352, 1.688=372, 1.645=1, 1.689=299, 1.680=492, 1.681=437, 1.682=419, 1.683=479,
1.684=449, 1.685=434, 1.686=408, 1.654=2, 1.698=68, 1.699=74, 1.656=6, 1.657=4, 1.658=15, 1.659=9,
1.690=300, 1.691=275, 1.692=253, 1.693=206, 1.650=1, 1.694=165, 1.651=1, 1.695=142, 1.696=115, 1.653=3,
1.697=86, 1.665=79, 1.666=78, 1.700=52, 1.667=110, 1.701=39, 1.668=116, 1.702=24, 1.669=152, 1.703=13,
1.704=13, 1.705=8, 1.660=17, 1.661=22, 1.662=31, 1.663=40, 1.664=48}
```

Putting this into a little JavaFX was simple. I used Oracle's "Ensemble" demo as it's perfect for looking for ideas, very much like the old Java Swing demo but sexier. I cut/pasted one of the examples, modified the code a little and got the listing on the next page. It was probably 10 minutes work.

To walk through the code briefly, we call init() which sets up the Stage (the main window) and adds just one child, our graph. The graph details (x-axis and y-axis etc.) were changed a little from the Ensemble demo and I added a new initialisation for the x-axis. Finally we launch a background timer every second to re-run the stream and display a new set of values before finally calling show() which puts it on the screen.

```
public class JavaFxDemo extends Application {
    private XYChart.Data<String, Number>[] rateData;

    public static void main(String[] args) { launch(args); }

    private void init(Stage primaryStage) {
        Group root = new Group();
        primaryStage.setScene(new Scene(root));
        root.getChildren().add(createChart());

        Timer timer = new Timer();
        timer.scheduleAtFixedRate( createUpdateTask(), 0, 1000);
    }

    private TimerTask createUpdateTask() {
        return new TimerTask() {
            public void run() {
                Stream<Trade> tradeStream = Stream.generate(() -> {
                    return Java8TradeDemo.createTrade();
                });

                Map<String, Long> map = tradeStream
                        .filter(t -> t.getCurrency1()
                              .matches("GBP") && t.getCurrency2().matches("USD"))
                        .limit(10_000)
                        .collect(Collectors.groupingBy(t -> String.format("%.3f",
                              t.getExchangeRate()), Collectors.counting()));

               for (int i = 0; i < rateData.length; i++) {
                    Long value = map.get(rateData[i].getXValue());
                    rateData[i].setYValue(value == null ? 0 : value);
                }
            }
        };
    }

    protected BarChart<String, Number> createChart() {
        final CategoryAxis xAxis = new CategoryAxis();
        final NumberAxis yAxis = new NumberAxis();
        final BarChart<String, Number> bc = new BarChart<>(xAxis, yAxis);
        bc.setLegendVisible(false);
        bc.setVerticalGridLinesVisible(false);

        bc.setTitle("GBP / USD Rate");
        yAxis.setAutoRanging(true);
        xAxis.setLabel("Exchange Rate");
        yAxis.setLabel("Volume");
        XYChart.Series<String, Number> series1 = new XYChart.Series<>();

        final int NUM_POINTS = 100;
        rateData = new XYChart.Data[NUM_POINTS];
        String[] categories = new String[NUM_POINTS];
        for (int i = 0; i < rateData.length; i++) {
            categories[i] = Double.toString(1.6318 + (i * 0.001)).substring(0, 5);
            rateData[i] = new XYChart.Data<>(categories[i], 0);
            series1.getData().add(rateData[i]);
        }
        bc.getData().add(series1);
        return bc;
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        init(primaryStage);
        primaryStage.show();
    }
}
```

We've just touched on the streams API and lambdas. My goal was to give you a quick introduction and to show you how it might be applicable to financial services. I've personally found hundreds of resources and examples on the internet although getting used to the syntax can be a little daunting at first. I found the best method was to start with something simple like this and try out what you want first.

We started with a few lines of fictitious trades, we moved up to a million or so and touched on parallel streams. Finally we vastly increased the complexity to FpML trades (Interest Rate Derivatives) and mentioned some clever memory compaction (C24 SDOs) to extend the ability for a single JVM to parallel search well over 10 million FpML trades in just the memory of a laptop. I hope you found something of interest, and we'd love to get some feedback and ideas for this or other papers so we look forward to hearing from you. I must admit I was tempted to write an AKKA comparison but this is about Java 8 so I'll leave it for another paper perhaps.

Please note: All the tests for this paper were run on a laptop, benchmarks are deceptive at the best of times so I can't see much point in doing timings on a production box. The goal was to show you the difference in performance not the actual figures. Naturally for testing and production use you get to run the code on your own boxes and get real figures, the best comment we got so far from a client was "Holy shit, it actually works!" :-)

For More Information

To learn more about C24 Technologies, C24 Integration Objects and C24's SDOs including data-sheets, code reference implementations, and more technical information, please visit ***http://www.c24.biz***.

Or contact us directly at:

C24 - London T: +44 20 7117 0024
C24 - New York T: +1 212 572 6493
C24 - Japan T: +81 3 5212 7077

E: info@c24.biz