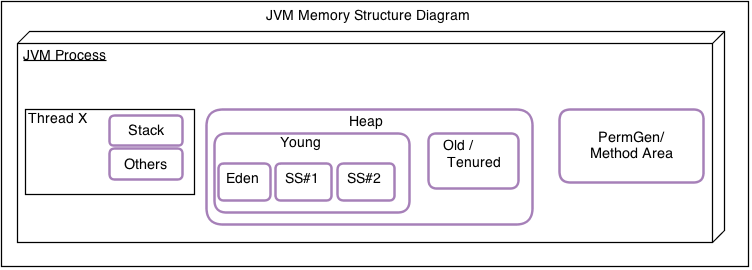
[**JVM Memory Structure**](http://www.ourdailycodes.com/2013/08/inside-java-jvm-memory-structure.html) **and more**

Whenever you execute a java program, a separate memory area is reserved for storing various parts of your application code which you roughly call **JVM memory**. Though no necessary, but having some knowledge about structuring of this memory area is quire beneficial. It becomes more important when you start working on deeper areas like performance tuning. Without having good understanding of how JVM actually consume the memory and how garbage collector uses different parts of this memory, you may miss some important considerations for better memory management; thus better performance.

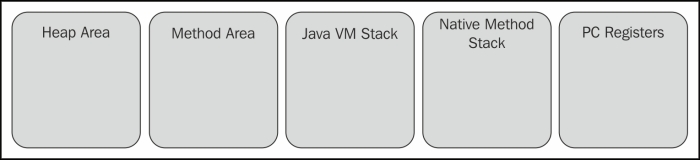
In this tutorial, I am discussing the various parts inside **JVM memory**, you should be aware of, and then in one of my future post we will discuss about how to use this information for performance tuning of your application.



## JVM Memory Model / Structure

The Java Virtual Machine defines various **run-time data areas** that are used during execution of a program. Some of these data areas are created on Java Virtual Machine start-up and are destroyed only when the Java Virtual Machine exits. Other data areas are per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

Let’s look at the most basic categorization of various parts inside runtime memory.



***JVM Memory Area Parts***

Let’s have a quick look at each of these components according to what is mentioned in the JVM specifications.

### Heap area

The heap area represents the runtime data area, from which the memory is allocated for all class instances and arrays, and is created during the virtual machine startup.

The heap storage for objects is reclaimed by an automatic storage management system. The heap may be of a fixed or dynamic size (based on system’s configuration), and the memory allocated for the heap area does not need to be contiguous.

*A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.*

If a computation requires more heap than can be made available by the automatic storage management system, the Java Virtual Machine throws an OutOfMemoryError.

### Method area and runtime constant pool

Method area stores per-class structures such as the runtime constant pool; field and method data; the code for methods and constructors, including the special methods used in class, instance, and interface initialization.

The method area is created on the virtual machine startup. Although it is logically a part of the heap but it can or cannot be garbage collected, whereas we already read that garbage collection in heap is not optional; it’s mandatory. The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. The memory for the method area does not need to be contiguous.

If memory in the method area cannot be made available to satisfy an allocation request, the Java Virtual Machine throws an OutOfMemoryError.

### JVM Stacks

Each of the JVM threads has a private stack created at the same time as that of the thread. The stack stores frames. A frame is used to store data and partial results and to perform dynamic linking, return values for methods, and dispatch exceptions.

It holds local variables and partial results and plays a part in the method invocation and return. Because this stack is never manipulated directly, except to push and pop frames, the frames may be heap allocated. Similar to the heap, the memory for this stack does not need to be contiguous.

This specification permits that stacks can be either of a fixed or dynamic size. If it is of a fixed size, the size of each stack may be chosen independently when that stack is created.

If the computation in a thread requires a larger Java Virtual Machine stack than is permitted, the Java Virtual Machine throws a StackOverflowError.

If Java Virtual Machine stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to effect the expansion, or if insufficient memory can be made available to create the initial Java Virtual Machine stack for a new thread, the Java Virtual Machine throws an OutOfMemoryError.

### Native method stacks

Native method stacks is called C stacks; it support native methods (methods written in a language other than the Java programming language), typically allocated per each thread when each thread is created. Java Virtual Machine implementations that cannot load native methods and that do not themselves rely on conventional stacks need not supply native method stacks.

The size of native method stacks can be either fixed or dynamic.

If the computation in a thread requires a larger native method stack than is permitted, the Java Virtual Machine throws a StackOverflowError.

If native method stacks can be dynamically expanded and native method stack expansion is attempted but insufficient memory can be made available, or if insufficient memory can be made available to create the initial native method stack for a new thread, the Java Virtual Machine throws an OutOfMemoryError.

### PC registers

Each of the JVM threads has its own program counter (pc) register. At any point, each of the JVM threads is executing the code of a single method, namely the current method for that thread.

As the Java applications can contain some native code (for example, using native libraries), we have two different ways for native and non-native methods. If the method is not native (that is, a Java code), the PC register contains the address of the JVM instruction currently being executed. If the method is native, the value of the JVM’s PC register is undefined.

The Java Virtual Machine’s pc register is wide enough to hold a return address or a native pointer on the specific platform.

# Java Heap Memory vs Stack Memory Difference

### Java Heap Memory

Heap memory is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

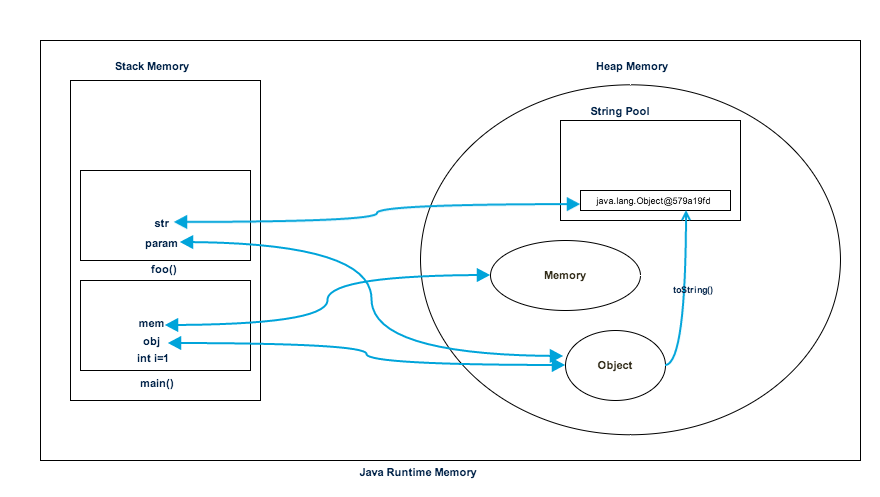
### Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.  
Stack memory size is very less compared to Heap memory.

Let’s understand the Heap and Stack memory usage with a simple program.

|  |  |
| --- | --- |
| Memory.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | package com.journaldev.test;    public class Memory {        public static void main(String[] args) { // Line 1          int i=1; // Line 2          Object obj = new Object(); // Line 3          Memory mem = new Memory(); // Line 4          mem.foo(obj); // Line 5      } // Line 9        private void foo(Object param) { // Line 6          String str = param.toString(); //// Line 7          System.out.println(str);      } // Line 8   } |

Below image shows the Stack and Heap memory with reference to above program and how they are being used to store primitive, Objects and reference variables.



Let’s go through the steps of execution of the program.

* As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.
* We are creating primitive local variable at line 2, so it’s created and stored in the stack memory of main() method.
* Since we are creating an Object in line 3, it’s created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
* Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
* A string is created in line 7, it goes in the [String Pool](http://www.journaldev.com/797/what-is-java-string-pool) in the heap space and a reference is created in the foo() stack space for it.
* foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.
* In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

### Difference between Heap and Stack Memory

Based on the above explanations, we can easily conclude following differences between Heap and Stack memory.

1. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
2. Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
3. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
4. Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, and will be affect by GC.
5. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
6. We can use **-Xms** and **-Xmx** JVM option to define the startup size and maximum size of heap memory. We can use **-Xss** to define the stack memory size.
7. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
8. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

That’s all for **Stack vs Heap Memory** in terms of java application

**Memory Allocation In Java**

What type of support Java provide for memory allocation :

**Heap Memory**I don't i have to explain this, all java application starts with this.  All object allocated using "new" keyword goes under Heap Memory

**ByteBuffer :**

ByteBuffer is one of the important class of Java NIO API. It was introduced in java.nio package on JDK 1.4, it not only allows you to operate on on heap byte arrays but also with direct memory, which resides outside the JVM. There are mainly three types f ByteBuffer, Direct, Non Direct and mapped byte buffers. You can create both direct and non direct buffers using java.nio.ByteBuffer class, whileMappedByteBuffer is a subclass of ByteBuffer, which is created by FileChannel.map() method, to operate on memory mapped file. Main difference between direct and non direct byte buffers are there memory location, non-direct byte buffers are just a wrapper around byte array and they reside in Java Heap memory, while direct byte buffer are outside of JVM and memory is not allocated from heap. You can remember this fact by there name, *Direct* indicates working with memory directly. Due to this reason, direct byte buffers are also not affected by Garbage Collection. *MappedByteBuffer* is also a type of direct byte buffer, which represent memory mapped region of a file. In this Java NIO tutorial, you will see couple of more differences between direct, non direct and mapped byte buffers, which will help you to understand the concept and there usage better. If you love books like me and wants to learn advanced concept e.g. high performance and low latency application development, performance tuning and JVM internals, I suggest to take a look at Definitive guide of Java Performance, one of the must read book for Java programmers.  
  
 **Direct vs Non direct vs MappedByteBuffer in Java**

As I said ByteBuffer is one of the very important class in high performance application. It is widely used in high frequency trading application, which strives for very very low latency, mostly in sub micro second level. When I first mentioned about memory mapped file in Java, I have outlined some benefits of using those files, and ByteBuffer class is key to operate them. Most of the differences between direct and non direct ByteBuffer derived from the fact that one is inside heap memory, while other is outside heap.  
1) First difference between non-direct and direct byte buffer comes from the fact, how you create them. You can create non-direct byte buffer either by allocating space for buffer's content or by wrapping an existing byte array into buffer. While a Direct byte buffer may be created by calling factory method allocateDirect() or by mapping a region of a file directly into memory , known  as MappedByteBuffer.  
2) In case of Direct byte buffer, JVM performs **native IO operation** directly into buffer, without copying them into any intermediate buffer, this makes it very attractive for performing high speed IO operation on them, but this facility comes with care. If a memory mapped file is shared between multiple process than you need to ensure that it won't get corrupted i.e. some regions of memory mapped file not becoming unavailable.  
3) One more difference between direct and non-direct byte buffers are that former's memory footprint may not be obvious because they are allocated outside of Java heap, while non-direct buffers consumes heap space and are subject to garbage collection.  
4) You can check whether a byte buffer is direct or non-direct by calling isDirect() method from java.nio.ByteBuffer class. It returns true, if byte buffer is direct.

[](http://1.bp.blogspot.com/-Qn6w5VIdKXM/VcIXD5LhTLI/AAAAAAAADjU/bT-9Rc2PtIc/s1600/Difference+between+Direct+and+Mapped+ByteBuffer+in+Java.jpg)

### These were some differences between direct, non-direct and mapped byte buffers in Java. If you are working in high volume low latency systems than  most of the  cases  you will work with either direct or mapped byte buffers. Since ByteBuffer indexes are integer based, which effectively limits there addressable space up-to 2GB, you may want to check BigByteBuffer class from Java 1.7 nio package, which provides long indexes, alternatively you can also use offsets to map different regions of memory mapped file. That's all on difference between direct, non-direct and mapped byte buffer in Java. Just remember that, Direct buffers are allocated outside heap and they are not in control of Garbage Collection, while non-direct buffers are simply a wrapper around byte arrays, located inside heap. Memory mapped files can be accessed by using MappedByteBuffer, which is also a direct buffer. One more thing to remember is that default order of bytes in ByteBuffer is BIG\_ENDIAN, which means the bytes of a multi-byte value are ordered from most significant to least significant. What is Memory Mapped File and IO in Java

[memory mapped file and io in java read write example](http://javarevisited.blogspot.com/2011/12/how-to-change-tomcat-default-port-8080.html)**Memory mapped files** are special files in Java which allows Java program to access contents  directly from memory, this is achieved by *mapping whole file or portion of file into memory* and operating system takes care of loading page requested and writing into file while application only deals with memory which results in very fast IO operations. Memory used to load Memory mapped file is outside of Java heap Space. Java programming language supports memory mapped file with **java.nio** package and has **MappedByteBuffer to read and write from memory**.

### Advantage and Disadvantage of Memory Mapped file

Possibly main advantage of Memory Mapped IO is*performance,* which is important to build high frequency electronic trading system. Memory Mapped Files are way faster than standard file access via normal IO. Another big advantage of memory mapped IO is that it allows you to load potentially larger file which is not otherwise accessible. Experiments shows that memory mapped IO performs better with large files. Though it has disadvantage in terms of increasing number of page faults. Since operating system only loads a portion of file into memory if a page requested is not present in memory than it would result in page fault Most of major operating system like Windows platform, UNIX, Solaris and other [UNIX](http://javarevisited.blogspot.com/2011/04/unix-commands-tutorial-and-tips-for.html) like operating system supports memory mapped IO and with 64 bit architecture you can map almost any file into memory and access it directly using Java programming language. Another advantages is that the file can be shared, giving you shared memory between processes and can be more than 10x lower latency than using a Socket over loopback.

### MappedByteBuffer Read Write Example in Java

Below example will show you *how to read and write from memory mapped file in Java*. We have used **RandomAccesFile** to open a File and than mapped it to memory using **FileChannel's map() method**, map method takes three parameter mode, start and length of region to be mapped. It returns MapppedByteBuffer which is a **ByteBuffer**for dealing with memory mapped file.

import java.io.RandomAccessFile;

import java.nio.MappedByteBuffer;

import java.nio.channels.FileChannel;

  public class MemoryMappedFileInJava {

    private static int count = 10485760; //10 MB

    public static void main(String[] args) throws Exception {

        RandomAccessFile memoryMappedFile = new RandomAccessFile("largeFile.txt", "rw");

        //Mapping a file into memory

        MappedByteBuffer out = memoryMappedFile.getChannel().map(FileChannel.MapMode.READ\_WRITE, 0, count);

        //Writing into Memory Mapped File

        for (int i = 0; i < count; i++) {

            out.put((byte) 'A');

        }

        System.out.println("Writing to Memory Mapped File is completed");

        //reading from memory file in Java

        for (int i = 0; i < 10 ; i++) {

            System.out.print((char) out.get(i));

        }

        System.out.println("Reading from Memory Mapped File is completed");

    }

}

### Summary

To summarize the post here is quick summary of memory mapped files and IO in Java:

1) Java supports Memory mapped IO with java.nio package.

2) Memory mapped files is used in performance sensitive application e.g. high frequency electronic trading platforms.

3) By using memory mapped IO you canload portion of large files in memory.

4) Memory mapped file can result in **page fault** if requested page is not in memory.

5) Ability to map a region of file in memory depends on addressable size of memory. In a 32 bit machineyou can not access beyond 4GB or 2^32.

6) Memory mapped IO is much faster than Stream IO in Java.

7) Memory used to load File is outside of Java heap and reside on shared memory which allow two different process to access File. By the way this depends upon, whether you are using direct or non-direct byte buffer.  
  
8) Reading and writing on memory mapped file is done by operating system, so even if your Java Program crash after putting content into memory it will make to disk, until OS is fine.

9) Prefer Direct Byte buffer over Non Direct Buffer for higher performance.  
10) Don't call MappedByteBuffer.force() method to often, this method is meant to force operating system to write content of memory into disk, So if you call force() method each time you write into memory mapped file, you will not see true benefit of using mapped byte buffer, instead it will be similar to disk IO.  
11) In case of power failure or host failure, there is slim chance that content of memory mapped file is not written into disk, which means you could lose critical data.  
12) MappedByteBuffer and file mapping remains valid until buffer is garbage collected. sun.misc.Cleaner is probably the only option available to clear memory mapped file.

That’s all on **memory mapped file and memory mapped IO in Java**. Its pretty useful concept and I encourage you to learn more about it. If you are working on high frequency trading space than memory mapped file is quite common there.

In Java, all user defined types have to exist on the heap.  The Java heap is managed by the garbage collector in the general case, however there is more to the wider heap in a Java process.  With the introduction of direct [ByteBuffer](http://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html), memory can be allocated which is not tracked by the garbage collector because it can be available to native code for tasks like avoiding the copying of data to and from the kernel for IO.  So one method of managing structures is to fake them within a ByteBuffer as a reasonable approach.  This can allow compact data representations, but has **performance and size limitations**.  For example, it is not possible to have a ByteBuffer greater than 2GB, and all access is bounds checked which impacts performance.  An alternative exists **using**[**Unsafe**](http://www.docjar.com/docs/api/sun/misc/Unsafe.html)**that is both faster and and not size constrained like**[**ByteBuffer**](http://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html).  
  
The approach I'm about to detail is not traditional Java.  If your problem space is dealing with big data, or extreme performance, then there are benefits to be had.  If your data sets are small, and performance is not an issue, then run away now to avoid getting sucked into the dark arts of native memory management.  
  
The benefits of the approach I'm about to detail are:

1. Significantly improved performance
2. More compact data representation
3. Ability to work with very large data sets while avoiding nasty GC pauses[1]

With all choices there are consequences.  By taking the approach detailed below you take responsibility for some of the memory managment yourself.  **Getting it wrong can lead to memory leaks, or worse, you can crash the JVM**!

**Suitable Example - *Trade Data***  
  
A common challenge faced in finance applications is capturing and working with very large volumes of order and trade data.  For the example I will create a large table of in-memory trade data that can have analysis queries run against it.  This table will be built using 2 contrasting approaches.  Firstly, I'll take the traditional Java approach of creating a large array and reference individual Trade objects.  Secondly, I keep the usage code identical but replace the large array and Trade objects with an off-heap array of structures that can be manipulated via a [Flyweight](http://www.oodesign.com/flyweight-pattern.html) pattern.   
  
If for the traditional Java approach I used some other data structure, such as a Map or Tree, then the memory footprint would be even greater and the performance lower.  
  
**Traditional Java Approach**

|  |
| --- |
| public class TestJavaMemoryLayout  {      private static final int NUM\_RECORDS = 50 \* 1000 \* 1000;        private static JavaMemoryTrade[] trades;        public static void main(final String[] args)      {          for (int i = 0; i < 5; i++)          {              System.gc();              perfRun(i);          }      }        private static void perfRun(final int runNum)      {          long start = System.currentTimeMillis();            init();            System.out.format("Memory %,d total, %,d free\n",                            Runtime.getRuntime().totalMemory(),                            Runtime.getRuntime().freeMemory());            long buyCost = 0;          long sellCost = 0;            for (int i = 0; i < NUM\_RECORDS; i++)          {              final JavaMemoryTrade trade = get(i);                if (trade.getSide() == 'B')              {                  buyCost += (trade.getPrice() \* trade.getQuantity());              }              else              {                  sellCost += (trade.getPrice() \* trade.getQuantity());              }          }            long duration = System.currentTimeMillis() - start;          System.out.println(runNum + " - duration " + duration + "ms");          System.out.println("buyCost = " + buyCost + " sellCost = " + sellCost);      }        private static JavaMemoryTrade get(final int index)      {          return trades[index];      }        public static void init()      {          trades = new JavaMemoryTrade[NUM\_RECORDS];            final byte[] londonStockExchange = {'X', 'L', 'O', 'N'};          final int venueCode = pack(londonStockExchange);            final byte[] billiton = {'B', 'H', 'P'};          final int instrumentCode = pack( billiton);            for (int i = 0; i < NUM\_RECORDS; i++)          {              JavaMemoryTrade trade = new JavaMemoryTrade();              trades[i] = trade;                trade.setTradeId(i);              trade.setClientId(1);              trade.setVenueCode(venueCode);              trade.setInstrumentCode(instrumentCode);                trade.setPrice(i);              trade.setQuantity(i);                trade.setSide((i & 1) == 0 ? 'B' : 'S');          }      }        private static int pack(final byte[] value)      {          int result = 0;          switch (value.length)          {              case 4:                  result = (value[3]);              case 3:                  result |= ((int)value[2] << 8);              case 2:                  result |= ((int)value[1] << 16);              case 1:                  result |= ((int)value[0] << 24);                  break;                default:                  throw new IllegalArgumentException("Invalid array size");          }            return result;      }        private static class JavaMemoryTrade      {          private long tradeId;          private long clientId;          private int venueCode;          private int instrumentCode;          private long price;          private long quantity;          private char side;            public long getTradeId()          {              return tradeId;          }            public void setTradeId(final long tradeId)          {              this.tradeId = tradeId;          }            public long getClientId()          {              return clientId;          }            public void setClientId(final long clientId)          {              this.clientId = clientId;          }            public int getVenueCode()          {              return venueCode;          }            public void setVenueCode(final int venueCode)          {              this.venueCode = venueCode;          }            public int getInstrumentCode()          {              return instrumentCode;          }            public void setInstrumentCode(final int instrumentCode)          {              this.instrumentCode = instrumentCode;          }            public long getPrice()          {              return price;          }            public void setPrice(final long price)          {              this.price = price;          }            public long getQuantity()          {              return quantity;          }            public void setQuantity(final long quantity)          {              this.quantity = quantity;          }            public char getSide()          {              return side;          }            public void setSide(final char side)          {              this.side = side;          }      }  } |

**Compact Off-Heap Structures**

|  |
| --- |
| import sun.misc.Unsafe;    import java.lang.reflect.Field;    public class TestDirectMemoryLayout  {      private static final Unsafe unsafe;      static      {          try          {              Field field = Unsafe.class.getDeclaredField("theUnsafe");              field.setAccessible(true);              unsafe = (Unsafe)field.get(null);          }          catch (Exception e)          {              throw new RuntimeException(e);          }      }        private static final int NUM\_RECORDS = 50 \* 1000 \* 1000;        private static long address;      private static final DirectMemoryTrade flyweight = new DirectMemoryTrade();        public static void main(final String[] args)      {          for (int i = 0; i < 5; i++)          {              System.gc();              perfRun(i);          }      }        private static void perfRun(final int runNum)      {          long start = System.currentTimeMillis();            init();            System.out.format("Memory %,d total, %,d free\n",                            Runtime.getRuntime().totalMemory(),                            Runtime.getRuntime().freeMemory());            long buyCost = 0;          long sellCost = 0;            for (int i = 0; i < NUM\_RECORDS; i++)          {              final DirectMemoryTrade trade = get(i);                if (trade.getSide() == 'B')              {                  buyCost += (trade.getPrice() \* trade.getQuantity());              }              else              {                  sellCost += (trade.getPrice() \* trade.getQuantity());              }          }            long duration = System.currentTimeMillis() - start;          System.out.println(runNum + " - duration " + duration + "ms");          System.out.println("buyCost = " + buyCost + " sellCost = " + sellCost);            destroy();      }        private static DirectMemoryTrade get(final int index)      {          final long offset = address + (index \* DirectMemoryTrade.getObjectSize());          flyweight.setObjectOffset(offset);          return flyweight;      }        public static void init()      {          final long requiredHeap = NUM\_RECORDS \* DirectMemoryTrade.getObjectSize();          address = unsafe.allocateMemory(requiredHeap);            final byte[] londonStockExchange = {'X', 'L', 'O', 'N'};          final int venueCode = pack(londonStockExchange);            final byte[] billiton = {'B', 'H', 'P'};          final int instrumentCode = pack( billiton);            for (int i = 0; i < NUM\_RECORDS; i++)          {              DirectMemoryTrade trade = get(i);                trade.setTradeId(i);              trade.setClientId(1);              trade.setVenueCode(venueCode);              trade.setInstrumentCode(instrumentCode);                trade.setPrice(i);              trade.setQuantity(i);                trade.setSide((i & 1) == 0 ? 'B' : 'S');          }      }        private static void destroy()      {          unsafe.freeMemory(address);      }        private static int pack(final byte[] value)      {          int result = 0;          switch (value.length)          {              case 4:                  result |= (value[3]);              case 3:                  result |= ((int)value[2] << 8);              case 2:                  result |= ((int)value[1] << 16);              case 1:                  result |= ((int)value[0] << 24);                  break;                default:                  throw new IllegalArgumentException("Invalid array size");          }            return result;      }        private static class DirectMemoryTrade      {          private static long offset = 0;            private static final long tradeIdOffset = offset += 0;          private static final long clientIdOffset = offset += 8;          private static final long venueCodeOffset = offset += 8;          private static final long instrumentCodeOffset = offset += 4;          private static final long priceOffset = offset += 4;          private static final long quantityOffset = offset += 8;          private static final long sideOffset = offset += 8;            private static final long objectSize = offset += 2;            private long objectOffset;            public static long getObjectSize()          {              return objectSize;          }            void setObjectOffset(final long objectOffset)          {              this.objectOffset = objectOffset;          }            public long getTradeId()          {              return unsafe.getLong(objectOffset + tradeIdOffset);          }            public void setTradeId(final long tradeId)          {              unsafe.putLong(objectOffset + tradeIdOffset, tradeId);          }            public long getClientId()          {              return unsafe.getLong(objectOffset + clientIdOffset);          }            public void setClientId(final long clientId)          {              unsafe.putLong(objectOffset + clientIdOffset, clientId);          }            public int getVenueCode()          {              return unsafe.getInt(objectOffset + venueCodeOffset);          }            public void setVenueCode(final int venueCode)          {              unsafe.putInt(objectOffset + venueCodeOffset, venueCode);          }            public int getInstrumentCode()          {              return unsafe.getInt(objectOffset + instrumentCodeOffset);          }            public void setInstrumentCode(final int instrumentCode)          {              unsafe.putInt(objectOffset + instrumentCodeOffset, instrumentCode);          }            public long getPrice()          {              return unsafe.getLong(objectOffset + priceOffset);          }            public void setPrice(final long price)          {              unsafe.putLong(objectOffset + priceOffset, price);          }            public long getQuantity()          {              return unsafe.getLong(objectOffset + quantityOffset);          }            public void setQuantity(final long quantity)          {              unsafe.putLong(objectOffset + quantityOffset, quantity);          }            public char getSide()          {              return unsafe.getChar(objectOffset + sideOffset);          }            public void setSide(final char side)          {              unsafe.putChar(objectOffset + sideOffset, side);          }      }  } |

**Results** 

Intel i7-860 @ 2.8GHz, 8GB RAM DDR3 1333MHz,

Windows 7 64-bit, Java 1.7.0\_07

=============================================

java -server -Xms4g -Xmx4g TestJavaMemoryLayout

Memory 4,116,054,016 total, 1,108,901,104 free

0 - duration 19334ms

Memory 4,116,054,016 total, 1,109,964,752 free

1 - duration 14295ms

Memory 4,116,054,016 total, 1,108,455,504 free

2 - duration 14272ms

Memory 3,817,799,680 total, 815,308,600 free

3 - duration 28358ms

Memory 3,817,799,680 total, 810,552,816 free

4 - duration 32487ms

java -server TestDirectMemoryLayout

Memory 128,647,168 total, 126,391,384 free

0 - duration 983ms

Memory 128,647,168 total, 126,992,160 free

1 - duration 958ms

Memory 128,647,168 total, 127,663,408 free

2 - duration 873ms

Memory 128,647,168 total, 127,663,408 free

3 - duration 886ms

Memory 128,647,168 total, 127,663,408 free

4 - duration 884ms

Intel i7-2760QM @ 2.40GHz, 8GB RAM DDR3 1600MHz,

Linux 3.4.11 kernel 64-bit, Java 1.7.0\_07

=================================================

java -server -Xms4g -Xmx4g TestJavaMemoryLayout

Memory 4,116,054,016 total, 1,108,912,960 free

0 - duration 12262ms

Memory 4,116,054,016 total, 1,109,962,832 free

1 - duration 9822ms

Memory 4,116,054,016 total, 1,108,458,720 free

2 - duration 10239ms

Memory 3,817,799,680 total, 815,307,640 free

3 - duration 21558ms

Memory 3,817,799,680 total, 810,551,856 free

4 - duration 23074ms

java -server TestDirectMemoryLayout

Memory 123,994,112 total, 121,818,528 free

0 - duration 634ms

Memory 123,994,112 total, 122,455,944 free

1 - duration 619ms

Memory 123,994,112 total, 123,103,320 free

2 - duration 546ms

Memory 123,994,112 total, 123,103,320 free

3 - duration 547ms

Memory 123,994,112 total, 123,103,320 free

4 - duration 534ms

**Analysis**  
  
Let's compare the results to the 3 benefits promised above.  
  
**1.  Significantly improved performance**  
  
The evidence here is pretty clear cut.  Using the off-heap structures approach is more than an order of magnitude faster.  At the most extreme, look at the 5th run on a Sandy Bridge processor, we have **43.2** **times** difference in duration to complete the task.  It is also a nice illustration of how well Sandy Bridge does with predictable access patterns to data.  Not only is the performance significantly better it is also more consistent.  As the heap becomes fragmented, and thus access patterns become more random, the performance degrades as can be seen in the later runs with standard Java approach.  
  
**2.  More compact data representation**  
  
For our off-heap representation each object requires 42-bytes.  To store 50 million of these, as in the example, we require 2,100,000,000 bytes.  The memory required by the JVM heap is:  
  
   memory required = total memory - free memory - base JVM needs   
  
     2,883,248,712 = 3,817,799,680 - 810,551,856 - 123,999,112  
  
This implies the JVM needs ~40% more memory to represent the same data.  The reason for this overhead is the array of references to the Java objects plus the object headers.  In a previous [post](http://mechanical-sympathy.blogspot.co.uk/2011/07/false-sharing.html) I discussed object layout in Java.  
  
When working with very large data sets this overhead can become a significant limiting factor.  
  
**3.  Ability to work with very large data sets while avoiding nasty GC pauses**  
  
The sample code above forces a GC cycle before each run and can improve the consistency of the results in some cases.  Feel free to remove the call to System.gc() and observe the implications for yourself.  If you run the tests adding the following command line arguments then the garbage collector will output in painful detail what happened.  
  
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintHeapAtGC -XX:+PrintGCApplicationConcurrentTime -XX:+PrintGCApplicationStoppedTime -XX:+PrintSafepointStatistics  
  
From analysing the output I can see the application underwent a total of 29 GC cycles.  Pause times are listed below by extracting the lines from the output indicating when the application threads are stopped. 

With System.gc() before each run

================================

Total time for which application threads were stopped: 0.0085280 seconds

Total time for which application threads were stopped: 0.7280530 seconds

Total time for which application threads were stopped: 8.1703460 seconds

Total time for which application threads were stopped: 5.6112210 seconds

Total time for which application threads were stopped: 1.2531370 seconds

Total time for which application threads were stopped: 7.6392250 seconds

Total time for which application threads were stopped: 5.7847050 seconds

Total time for which application threads were stopped: 1.3070470 seconds

Total time for which application threads were stopped: 8.2520880 seconds

Total time for which application threads were stopped: 6.0949910 seconds

Total time for which application threads were stopped: 1.3988480 seconds

Total time for which application threads were stopped: 8.1793240 seconds

Total time for which application threads were stopped: 6.4138720 seconds

Total time for which application threads were stopped: 4.4991670 seconds

Total time for which application threads were stopped: 4.5612290 seconds

Total time for which application threads were stopped: 0.3598490 seconds

Total time for which application threads were stopped: 0.7111000 seconds

Total time for which application threads were stopped: 1.4426750 seconds

Total time for which application threads were stopped: 1.5931500 seconds

Total time for which application threads were stopped: 10.9484920 seconds

Total time for which application threads were stopped: 7.0707230 seconds

Without System.gc() before each run

===================================

Test run times

0 - duration 12120ms

1 - duration 9439ms

2 - duration 9844ms

3 - duration 20933ms

4 - duration 23041ms

Total time for which application threads were stopped: 0.0170860 seconds

Total time for which application threads were stopped: 0.7915350 seconds

Total time for which application threads were stopped: 10.7153320 seconds

Total time for which application threads were stopped: 5.6234650 seconds

Total time for which application threads were stopped: 1.2689950 seconds

Total time for which application threads were stopped: 7.6238170 seconds

Total time for which application threads were stopped: 6.0114540 seconds

Total time for which application threads were stopped: 1.2990070 seconds

Total time for which application threads were stopped: 7.9918480 seconds

Total time for which application threads were stopped: 5.9997920 seconds

Total time for which application threads were stopped: 1.3430040 seconds

Total time for which application threads were stopped: 8.0759940 seconds

Total time for which application threads were stopped: 6.3980610 seconds

Total time for which application threads were stopped: 4.5572100 seconds

Total time for which application threads were stopped: 4.6193830 seconds

Total time for which application threads were stopped: 0.3877930 seconds

Total time for which application threads were stopped: 0.7429270 seconds

Total time for which application threads were stopped: 1.5248070 seconds

Total time for which application threads were stopped: 1.5312130 seconds

Total time for which application threads were stopped: 10.9120250 seconds

Total time for which application threads were stopped: 7.3528590 seconds

It can been seen from the output that a significant proportion of the time is spent in the garbage collector.  When your threads are stopped your application is not responsive.  These tests have been done with default GC settings.  It is possible to tune the GC for better results but this can be a highly skilled and significant effort.  The only JVM I know that copes well by not imposing long pause times, even under high-throughput conditions, is the Azul concurrent compacting collector.  
  
When profiling this application, I can see that the majority of the time is spent allocating the objects and promoting them to the old generation because they do not fit in the young generation.  The initialisation costs can be removed from the timing but that is not realistic.  If the traditional Java approach is taken the state needs to be built up before the query can take place.  The end user of an application has to wait for the state to be built up and the query executed.  
  
This test is really quite trivial.  Imagine working with similar data sets but at the 100 GB scale.  
  
**Note:** When the garbage collector compacts a region, then objects that were next to each other can be moved far apart.  This can result in TLB and other cache misses.  
  
**Side Note On Serialization**  
  
A huge benefit of using off-heap structures in this manner is how they can be very easily serialised to network, or storage, by a simple memory copy as I have shown in the previous [post](http://mechanical-sympathy.blogspot.co.uk/2012/07/native-cc-like-performance-for-java.html).  This way we can completely bypass intermediate buffer and object allocation.  
  
**Conclusion**  
  
If you are willing to do some C style programming for large datasets it is possible to control the memory layout in Java by going off-heap.  If you do, the benefits in performance, compactness, and avoiding GC issues are significant.  However this is an approach that should **not** be used for all applications.  Its benefits are only noticable for very large datasets, or the extremes of performance in throughput and/or latency.    
  
I hope the Java community can collectively realise the importance of supporting structures both on the heap and the stack.  John Rose has done some excellent [work](https://blogs.oracle.com/jrose/entry/tuples_in_the_vm) in this area defining how tuples could be added to the JVM.  His talk on [Arrays 2.0](http://medianetwork.oracle.com/video/player/1785452137001) from the JVM Language Summit this year is really worth a watch.  John discusses options for arrays of structures, and structures of arrays, in his talk.  If the tuples, as proposed by John, were available then the test described here could have comparable performance and be a more pleasant programming style.  The whole array of structures could be allocated in a single action thus bypassing the copy of individual objects across generations, and it would be stored in a compact contiguous fashion.  This would remove the significant GC issues for this class of problem.  
  
Lately, I was comparing standard data structures between Java and .Net.  In some cases I observed a 6-10X performance advantage to .Net for things like maps and dictionaries when .Net used native structure support.  Let's get this into Java as soon as possible!  
  
It is also pretty obvious from the results that if we are to use Java for real-time analysis on big data, then our standard garbage collectors need to significantly improve and support true concurrent operations.

# Java Direct ByteBuffer Performance Advantages and Considerations

JUN 5TH, 2015 9:38 PM

During execution, objects/variables created by Java programs gets their space allocated in the JVM heap memory. The total amount of heap memory available for a JVM is determined by the value set to -Xmx parameter when starting the Java process. When object allocated is released by the Java program, the corresponding memory is made available for later use by the JVM garbage collection (GC) process.

The GC process gets invoked typically when the amount of free memory in the JVM falls below a certain threshold. At a very high level, the GC process involves identification of objects which are not used any more i.e. not referenced anymore, releasing the memory and compacting the memory to reduce memory fragmentation. Readers who are interested in understanding the details of GC process can find it [here](http://blog.asquareb.com/blog/2014/12/13/jvm-gc-settings-and-hbase). As one can imagine, the time it takes to complete the GC process will increase with the increase in size of the Java heap memory since it takes more time to identify the objects which can be released and also to perform compaction.

If a Java application requires large memory (in GBs), the time it takes to complete the GC process will be detrimental to its performance. If the application is performance sensitive, then large heap memory size can adversely impact its performance. In order to mitigate this, one can try to use memory outside Java heap and hence reduce the Java heap memory use and its size. This can be done using the Java [ByteBuffer](http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html) class which provides the option to allocate ByteBuffers outside JVM heap using [allocateDirect](http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html#allocateDirect(int)) method.

The allocateDirect method allocates memory of requested size (in bytes) on memory outside the JVM heap (off-heap) and provides the object reference to the application with the starting offset of 0. The application can then use the reference to store and retrieve data into the off-heap memory. When the garbage collection runs, it doesn’t have to take into account the memory allocated off-heap to identify memory not being used or perform memory compaction which in turn reduce the time to complete GC.

While the time to complete GC can be reduced when large memory is used in a Java process by using off-heap memory, there are other overheads which need to be taken into consideration before using it. Allocation of off-heap memory will take more time than the on-heap memory since the JVM need to make native calls to get the memory allocated. Also when the off-heap memory is not used anymore by the application, during GC process, the JVM need to make native calls to free the off-heap memory in addition to releasing the memory used by the object reference in on-heap memory. Also as per the API documentation, the JVM will make the best effort to not to use any on-heap memory as a intermediate step to store and retrieve data to/from the off-heap memory. In order to compensate these additional overheads and at the same time take advantage of using large memory without the penalty of increased GC time, it is best to use off-heap memory for large objects which doesn’t get released often.

When a JVM is brought up to run a Java process, the total memory which can be used for off-heap memory can be specified using the JVM parameter -XX:MaxDirectMemorySize parameter. If the parameter is not set explicitly, the value is set to the free memory available in the system at the start of the process using [VM.maxDirectMemory()](https://github.com/openjdk-mirror/jdk/blob/icedtea/jdk7/master/src/share/classes/sun/misc/VM.java#L193) method call. When off-heap memory allocation is made, the JVM keeps track of the total memory used so far. When a new off-heap memory allocation request is made the JVM checks whether the sum of the requested memory size and the total memory allocated so far is greater than the available direct memory size set at the start of the Java process. If the sum exceeds the available memory, an explicit GC system call is made by the memory allocator and then the process thread sleeps for [100ms](https://github.com/openjdk-mirror/jdk/blob/icedtea/jdk7/master/src/share/classes/java/nio/Bits.java#L651) for the GC call to complete. After 100ms the allocator checks again to see whether there is enough space to satisfy the new memory allocation request before raising an out of memory exception.

Few things to note about this allocation process.

* For performance sensitive applications the explicit GC and the non-tunable sleep time in the allocation logic when there is not enough memory can be a large overhead.
* The second item to note is that a GC call means best effort will be made by the JVM to schedule one and doesn’t guarantee that one will be run immediately. So there can be situations when the Java process will fail with OOM error even when there is enough memory to be freed to accommodate new memory allocation request since a GC is not run immediately.
* Third, the thread sleep time of 100ms may not be sufficient in certain situations for the GC to complete and release unused memory to satisfy new memory allocation request. If any one is surprised that the 100ms is more than sufficient for a GC to complete, we came across the situation where trying to allocate 1 GB chunks of off heap space using a simple for loop failing on Ubuntu 12.04 LTS with OOM while the same runs fine on Redhat Linux machine which had a relatively less powerful hardware. With the current API this sleep time can’t be adjusted and hence the application may have to perform additional sleep to make sure that there is no memory to use.
* The last item of interest is the total memory available for use to allocate off-heap memory. This value is set at the start of the process either manually or by the VM. When set manually, the JVM doesn’t verify whether there is enough free memory available on the system. Even if the value is set automatically by the JVM, the available memory on the system can be lower during the process execution since memory usage of other processes in the system can change as time goes by which can result in the Java process failing due to unexpected exception in memory allocation. So it is important to make sure that the memory of size set in -XX:MaxDirectMemorySize is available for the Java process to use so that the failures doesn’t happen.

There are few options the JVM can do to prevent allocation related exceptions which would require changes to the JVM code.

* Verify the system free memory to make sure that it is greater than or equal to what is set by the users when the JVM is brought up and also during the process execution. This will require native system calls and may have a pronounced impact on the performance of the Java process. One way to mitigate is to provide an JVM option for the users to set if they need this strict condition checking.
* Instead of invoking a GC call when all the memory is used, it would be better to have a configurable parameter to set direct memory used threshold to make the GC call. This should be a fairly simple change to the JVM code.
* Calculate the sleep time after the GC process taking into consideration all the factors which impacts GC time. This will be complex and will not be of less importance if the previous suggestion is implemented in the JDK code.

# Interlude: A Short Tour of Virtual Memory

ByteBuffer isn't just used to retrieve structured data from a byte[]. It also allows you to create and work with memory outside of the Java heap, including memory-mapped files. This latter feature is a great way to work with large amounts of structured data, as it lets you leverage the operating system's memory manager to move data in and out of memory in a way that's transparent to your program.

A program running on a modern operating system thinks that it has a large, contiguous allotment of memory: 2 gigabytes in the case of 32-bit editions of Windows and Linux, 8 terabytes or more for x64 editions (limited both by the operating system and the hardware itself). Behind the scenes, the operating system maintains a “page table” that identifies where in physical memory (or disk) the data for a given virtual address resides.

I've written [elsewhere](http://www.kdgregory.com/index.php?page=java.outOfMemory) about how the JVM uses virtual memory: it assigns space for the Java heap, per-thread stacks, shared native libraries including the JVM itself, and memory-mapped files (primarily JAR files). On Linux, the program pmap will show you the virtual address space of a running process, divided into segments of different sizes, with different access permissions.

In thinking about virtual memory, there are two concepts that every programmer should understand: resident set size and commit charge. The second is easiest to explain: it's the total amount of memory that your program might be able to modify (ie, it excludes read-only memory-mapped files and program code). The potential commit charge for an entire system is the sum of RAM and swap space, and no program can exceed this. It doesn't matter how big your virtual address space is: if you have 2G of RAM, and 2G of swap, you can never work with more than 4G of in-memory data; there's no place to store it.

In practice, no one program can reach that maximum commit charge either, because there are always other programs running, and they have their own claims upon memory. If you try to allocate memory that would exceed the available commit charge, you will get an OutOfMemoryError.

The second concept, resident set size (RSS), refers to how many of your program's virtual pages are currently residing in RAM. If a page isn't in RAM, then it needs to be read from disk — faulted into RAM — before your program can access it. The important thing to know about RSS is that you have very little control over it. The operating system tries to minimize the number of system-wide page faults, typically by managing RSS on the basis of time and access frequency: pages that are infrequently accessed get swapped out, making room for pages that are actively accessed. RSS is one reason that “full” garbage collections can take a long time: when the GC compacts the heap it will touch nearly every page in the heap; pages that are filled with garbage may be faulted-in at this time (smart Swing programmers know this, and explicitly trigger GC before being minimized, in order to prevent page faults when they're maximized again).

One final concept: pages in the resident set can be “dirty,” meaning that the program has changed their content. A dirty page must be written to swap space before its physical memory can be used by another page. By comparison, a clean (unmodified) page may simply be discarded; it will be reloaded from disk when needed. If you can guarantee that a page will never be modified, it doesn't count against a program's commit charge — we'll return to this topic when discussing memory-mapped files.

# Direct ByteBuffers

There are three ways to create a ByteBuffer: wrap(), which you've already seen, allocate(), which will create the underlying byte array for you, and allocateDirect(). The API docs for this last method are somewhat vague on exactly where the buffer will be allocated, stating only that “the Java virtual machine will make a best effort to perform native I/O operations directly upon it,” and that they “may reside outside of the normal garbage-collected heap.” In practice, direct buffers always live outside of the garbage-collected heap.

Knowing this, you might think that a direct buffer is a great way to extend the memory that your program can use. It isn't. The JVM is very good about growing the heap to the limits of physical and virtual memory, so if you've already maxed out your heap, there won't be any place to put a direct buffer.

I'm leaving this paragraph in place, as a reminder that the truth can stare you in the face without being seen. The EHCache developers weren't quite so blind, and created [BigMemory](http://www.terracotta.org/bigmemory), with the tagline that it “defuses the GC timebomb.” Kudos to them.

In fact, the only reason that I can see for using direct buffers in a pure Java program is that they won't be moved during garbage collection. If you've read my article on [reference objects](http://www.kdgregory.com/index.php?page=java.refobj), you'll remember that the garbage collector compacts the heap after disposing of dead objects. If you have large blocks of heap memory allocated as buffers, they may get moved as part of compaction, and no matter how fast your CPU, that takes time; it's not something you want to do on every full collection. Since the direct buffer lives outside of the heap, it isn't affected by collections. On the other hand, every data access is a JNI call. Only benchmarking will tell you whether this helps or hurts your particular application.

Direct buffers are useful in a program that mixes Java and native libraries: JNI provides methods to access the physical memory behind a direct buffer, and to allocate new buffers at known locations. Since this technique has a limited audience, it's outside of the scope of this article. If you're interested, I link to an example program at the end.

# Mapped Files

While I don't see much reason to use direct buffers in a pure Java program, they're the foundation for mapping files into the virtual address space — a feature that is rarely used, but invaluable when you need it. Mapping a file gives you random access with — depending on your access patterns — a significant performance boost. To understand why, we'll need to take a short detour into the way that Java file I/O works.

The first thing to understand is that the Java file classes are simply wrappers around native file operations. When you call read() from a Java program, you invoke the POSIX system call with the same name (at least on Solaris/Linux; I'll assume Windows as well). When the OS is asked to read data, it first looks into its cache of disk buffers, to see if you've recently read data from the same disk block. If the data is there, the call can return immediately. If not, the OS will initiate a disk read, and suspend your program until the data is available.

The key point here is that “immediately” does not mean “quickly”: you're invoking the operating system kernel to do the read, which means that the computer has to perform a “context switch” from application mode to kernel mode. To make this switch, it will save the CPU registers and page table for your application, and load the registers and page table for the kernel; when the kernel call is done, the reverse happens. This is a matter of a few microseconds, but those add up if you're constantly accessing a file. At worst, the OS schedule will decide that your program has had the CPU for long enough, and suspend it while another program runs.

With a memory-mapped file, by comparison, there's no need to invoke the OS unless the data isn't already in memory. And since the amount of RAM devoted to programs is larger than that devoted to disk buffers, the data is far more likely to be in memory.

Of course, whether or not your data is in memory depends on many things. Foremost is whether you're accessing the data sequentially: there's no point to replacing a FileInputStream with a mapped buffer, even though the JDK allows it, because you'll be constantly waiting for pages to load from disk

The second important question is how big your file is, and how randomly you access it. If you have a multi-gigabyte file and bounce from one spot to another, then you'll constantly wait for pages to be read from disk. But most programs don't access their data in a truly random manner. Typically there's one group of blocks that are hit far more frequently than others, and these will remain in RAM. For example, a database server reads the root node of an index on almost every query, while individual data blocks are accessed far less frequently.

Even if you don't gain a speed benefit from memory-mapping your files, you may gain a maintenance benefit by accessing them via a bean-style wrapper class. This will also improve testability, as you can construct buffers around known test data, without any files involved.

## Creating the Mapping

Creating a mapped file is a multi-step process, starting with a RandomAccessFile (you can also start with a FileInputStream orFileOutptStream, but there's no point to doing so). From there, you create a FileChannel, and then you call map() on that channel. It's easier to code than to describe:

File file = new File("/tmp/example.dat");

FileChannel channel = new RandomAccessFile(file, "r").getChannel();

ByteBuffer buf = channel.map(MapMode.READ\_ONLY, 0L, file.length());

buf.order(ByteOrder.LITTLE\_ENDIAN);

System.console().printf("data = %x

", buf.getInt(0));

Although I assign the return value from map() to a ByteBuffer variable, it's actually a MappedByteBuffer. Most of the time there's no reason to differentiate, but the latter class has two methods that some programs may find useful: load() and force().

The load() method will attempt to load all of the file's data into RAM, trading an increase in startup time for a potential decrease in page faults later. I think this is a form of premature optimization. Unless your program constantly accesses those pages, the operating system may choose to use them for something else, meaning that you'll have to fault them in anyway. Let the OS do its job, and load pages as needed from disk.

The second method, force(), deserves its own section.

## Read-Only versus Read-Write Mappings

Few programmers think about what happens to their files when the power goes out. Those that do typically stop thinking once they've called flush(). However, even if you've flushed the writes out of the operating system and into the disk drive, they may not have found their way to the disk platter: disk drives generally have an on-drive RAM buffer, and blocks live in that buffer until the drive can write them to the physical disk — or the power goes out. You can typically tweak the drive's settings via the OS (not Java), so if you absolutely, positively must ensure writes, you should learn how your particular drives work.

You'll note that I created the RandomAccessFile in read-only mode (by passing the flag "r" to its constructor), and reiterated that when mapping the channel. This will prevent accidental writes, but more importantly, it means that the file won't count against the program's commit charge. On a 64-bit machine, you can map terabytes of read-only files. And in most cases, you don't need write access: you have a large dataset that you want to process, and don't want to keep reading chunks of it into heap memory.

Read-write files require some more thought. The first thing to consider is just how important your writes are. As I noted above, the memory manager doesn't want to constantly write dirty pages to disk. Which means that your changes may remain in memory, unwritten, for a very long time — which will become a problem if the power goes out. To flush dirty pages to disk, call the buffer'sforce() method.

buf.putInt(0, 0x87654321);

buf.force();

Those two lines of code are actually an anti-pattern: you don't want to flush dirty pages after every write, or you'll make your program IO-bound. Instead, take a lesson from database developers, and group your changes into atomic units (or better, if you're planning on a lot of updates, use a real database).

## Mapping Files Bigger than 2 GB

Depending on your filesystem, you can create files larger than 2GB. But if you look at the ByteBuffer documentation, you'll see that it uses an int for all indexes, which means that buffers are limited to 2GB. Which in turn means that you need to create multiple buffers to work with large files.

One solution is to create those buffers as needed. The same underlying FileChannel can support as many buffers as you can create, limited only by the OS and available virtual memory; simply pass a different starting offset each time. The problem with this approach is that creating a mapping is expensive, because it's a kernel call (and you're using mapped files to avoid kernel calls). In addition, a page table full of mappings will mean more expensive context switches. As a result, as-needed buffers aren't a good approach unless you can divide the file into large chunks that are processed as a unit.

A better approach, in my opinion, is to create a “super buffer” that maps the entire file and presents an API that uses long offsets. Internally, it maintains an array of mappings with a known size, so that you can easily translate the original index into a buffer and an offset within that buffer:

public int getInt(long index)

{

return buffer(index).getInt();

}

private ByteBuffer buffer(long index)

{

ByteBuffer buf = \_buffers[(int)(index / \_segmentSize)];

buf.position((int)(index % \_segmentSize));

return buf;

}

That's straightforward, but what's a good value for \_segmentSize? Your first thought might be Integer.MAX\_VALUE, since this is the maximum index value for a buffer. While that would result in the fewest number of buffers to cover the file, it has one big flaw: you won't be able to access multi-byte values at segment boundaries.

Instead, you should overlap buffers, with the size of the overlap being the maximum sub-buffer (or byte[]) that you need to access. In my implementation, the segment size is Integer.MAX\_VALUE / 2 and each buffer is twice that size; one sub-buffer starts halfway into its predecessor:

public MappedFileBuffer(File file, int segmentSize, boolean readWrite)

throws IOException

{

if (segmentSize > MAX\_SEGMENT\_SIZE)

throw new IllegalArgumentException(

"segment size too large (max " + MAX\_SEGMENT\_SIZE + "): " + segmentSize);

\_segmentSize = segmentSize;

\_fileSize = file.length();

RandomAccessFile mappedFile = null;

try

{

String mode = readWrite ? "rw" : "r";

MapMode mapMode = readWrite ? MapMode.READ\_WRITE : MapMode.READ\_ONLY;

mappedFile = new RandomAccessFile(file, mode);

FileChannel channel = mappedFile.getChannel();

\_buffers = new MappedByteBuffer[(int)(\_fileSize / segmentSize) + 1];

int bufIdx = 0;

for (long offset = 0 ; offset < \_fileSize ; offset += segmentSize)

{

long remainingFileSize = \_fileSize - offset;

long thisSegmentSize = Math.min(2L \* segmentSize, remainingFileSize);

\_buffers[bufIdx++] = channel.map(mapMode, offset, thisSegmentSize);

}

}

finally

{

// close quietly

if (mappedFile != null)

{

try

{

mappedFile.close();

}

catch (IOException ignored) { /\* \*/ }

}

}

}

There are two things to notice here. The first notice is my use of Math.min(). You can't create a mapped buffer that's larger than the actual file; map()will throw if you try. Since I specify segment size rather than number of segments, I need to ensure that they fit reality. At most two buffers will be shrunk by this call, but it's less code to check on every buffer.

The second — and perhaps more important — thing is that I I close the RandomAccessFile after creating the mappings. My original version of this class didn't; it had a close() method, along with a finalizer to catch programmer mistakes. Then one day I took a closer look at the FileChannel.map()docs, and discovered that the buffer will persist after the channel is closed — it's removed by the garbage collector (and this explains the reason thatMappedByteBuffer doesn't have its own close() method).

# Garbage Collection of Direct/Mapped Buffers

That brings up another topic: how does the non-heap memory for direct buffers and mapped files get released? After all, there's no method to explicitly close or release them. The answer is that they get garbage collected like any other object, but with one twist: if you don't have enough virtual memory space or commit charge to allocate a direct buffer, that will trigger a full collection even if there's plenty of heap memory available. Normally, this won't be an issue: you probably won't be allocating and releasing direct buffers more often than heap-resident objects. If, however, you see full GC's appearing when you don't think they should, take a look at your program's use of buffers.

Along the same lines, when you're using direct buffers and mapped files, you'll get to see some of the more esoteric variants of OutOfMemoryError. “Direct buffer memory” is one of the more common, and appears to to indicate an OS-imposed limit (based on my reading of Bits.java). And when I tried to allocate more direct buffers than available commit charge, I received an OOM that didn't even have a message.

# Enabling Large Direct Buffers

You may be surprised, the first time that you try to allocate direct buffers on a 64-bit machine, that you get OutOfMemoryError when there's plenty of RAM available. You can usually resolve this problem by passing the following options when starting the JVM:

**-d64**

This option instructs the JVM to run in 64-bit mode. At the time of writing, many 64-bit operating systems actually have 32-bit JVMs, because the 32-bit JVM may be more efficient for “small” programs. This option is [documented](http://java.sun.com/javase/6/docs/technotes/tools/solaris/java.html) only for Linux/Solaris JVMs, and the documentation has a lot of caveats regarding when and how a 64-bit JVM is invoked. But it won't hurt.

**-XX:MaxDirectMemorySize**

This option is a hack to trigger garbage collection (which will reclaim any unreachable buffers); the value takes the normal JVM memory specifies (eg, 1024 for 1024 bytes, 1024m for 1024 megabytes, and 10g for 10 gigabytes). The -XX prefix indicates that it's one of the “super-secret, undocumented, OpenJDK-specific options,” and may change or be removed at any time.

To summarize, if you're running a program that needs to allocate 12 GB of direct buffers, you'd use a command-line like this:

java -XX:MaxDirectMemorySize=12g com.example.MyApp

If you're working with large buffers (direct buffers or memory mapped files), you should also use the -XX:+UseLargePages option:

java -d64 -XX:MaxDirectMemorySize=12g -XX:+UseLargePages com.example.MyApp

By default, the memory manager maps physical memory to the virtual address space in small chunks (4k is typical). This means that page faults can be handled more efficiently, because there's less data to read or write. However, small pages mean that memory management hardware has to keep track of more information to translate virtual addresses to physical. At best, this means less efficient usage of the [TLB](http://en.wikipedia.org/wiki/Translation_lookaside_buffer), which makes every memory access slower. At worst, you'll run out of entries in the page table (which is reported as OutOfMemoryError).