

A JVM Does What?

Eva Andreasson

Product Manager, Azul Systems

- Eva Andreasson – Innovator & Problem solver
 - Implemented the Deterministic GC of JRockit Real Time
 - Awarded patents on GC heuristics and self-learning algorithms
 - Most recent: Product Manager for Azul Systems' scalable Java Platform **Zing**
- I like new ideas and brave thinking!



- What is a JVM?
- What the JVM enables for Java
- Compilers and optimization
- Garbage collection and the pain of fragmentation
- What to (not) think about

What is a Java Virtual Machine (JVM)?

- A JVM described in a simple sentence

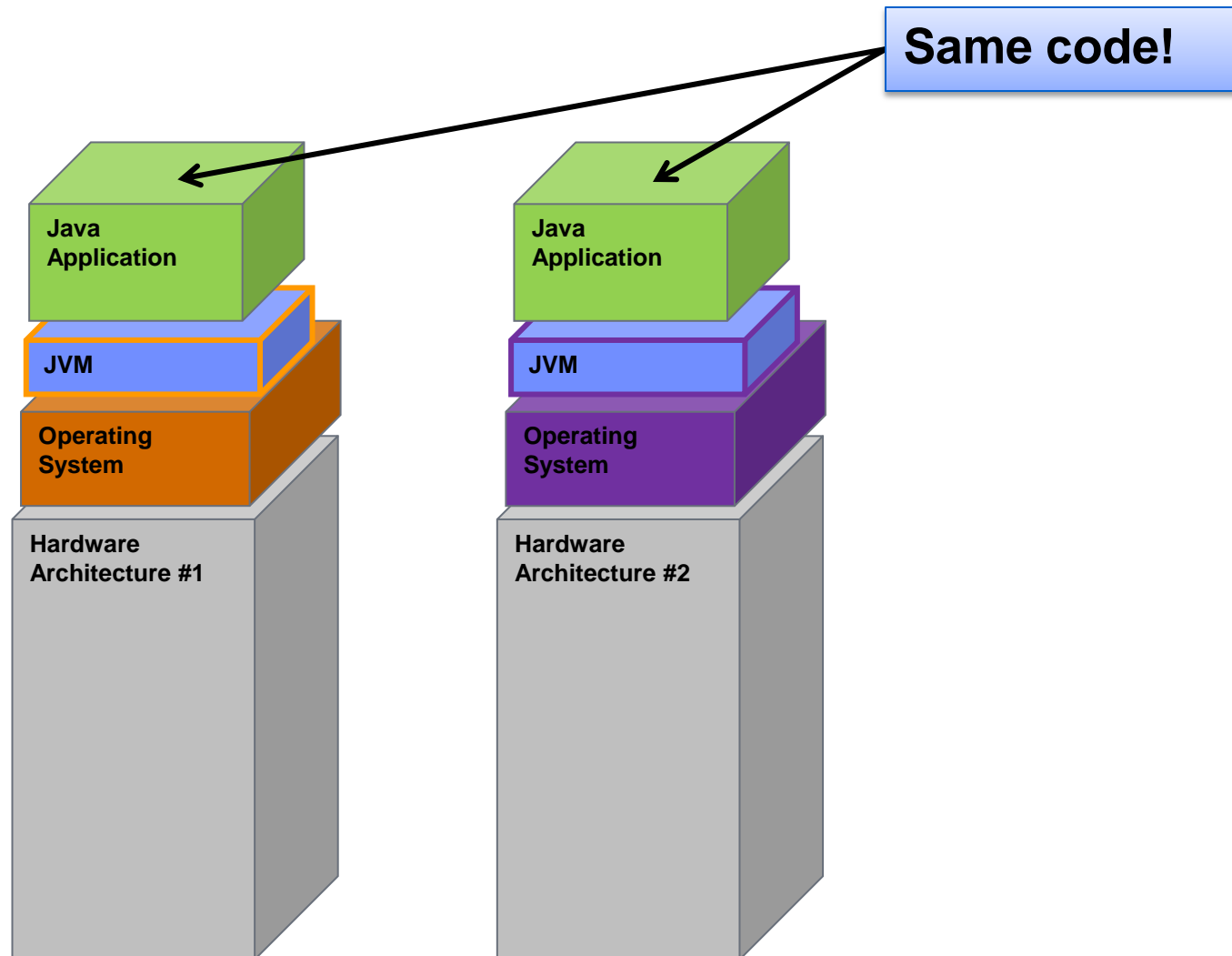
A software module that provides the same execution environment to all Java applications, and takes care of the “translation” to the underlying layers with regards to execution of instructions and resource management.

What is a Java Virtual Machine (JVM)?

- Key benefits of Java, enabled by the JVM
 - Portability
 - Dynamic Memory Allocation
- Other error-preventing and convenient services of the JVM
 - Consistent Thread Model
 - Optimizes and Manages Locking
 - Enforces Type Safety
 - Dynamic Code Loading
 - Quick high-quality Time Access (e.g. `currentTimeMillis`, `nanoTime`)
 - Internal introspection
 - Access to huge pre-built library
 - Access to OS and environmental information

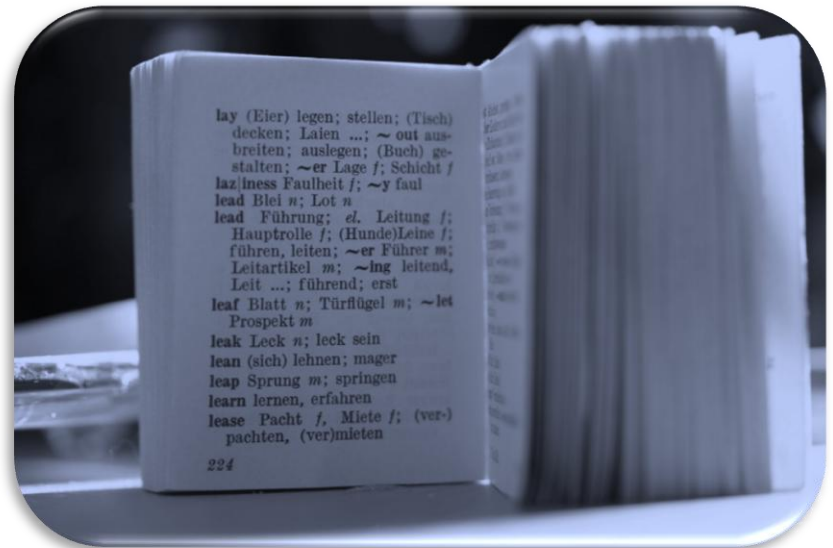
Portability

Compile once, run everywhere



What Makes Portability Possible?

- javac – takes Java source code compiles it into bytecode
 - MyApp.java → MyApp.class
- Bytecode can be “translated” by JVMs into HW instructions
 - Anywhere the JVM runs...
- Two paths of “translation”
 - Interpretation
 - The “dictionary” approach
 - Look up instruction, execute
 - (JIT) Compilation
 - The “intelligent translator”
 - Profile, analyze, execute faster



- Common concepts:
 - Hotspot detection, Re-compilation, and De-compilation
- Optimizations need resources
 - Temporary “JVM memory”
 - “Compiler threads”
 - Cost is covered by the faster execution time
- Different compilers for different needs
 - Client (“C1”), quicker compilation time, less optimized code
 - Server (“C2”), slower compilation time, more optimized code
 - Tiered, both C1 and C2
 - C1’s profiling used for C2’s compilation

Exploring Code Optimizations

Example 1: Dead Code Elimination

- Eliminates code that does not affect the program
- The compiler finds the “dead” code and eliminates the instruction set for it
 - Reduces the program size
 - Prevents irrelevant operations to occupy time in the CPU
- Example:

```
int timeToScaleMyApp(boolean endlessOfResources) {  
    int reArchitect = 24;  
    int patchByClustering = 15;  
    int useZing = 2;  
    if (endlessOfResources)  
        return reArchitect + useZing;  
    else  
        return useZing;  
}
```

```
int whenToEvaluateZing(int y) {  
    return daysLeft(y) + daysLeft(0) + daysLeft(y+1);  
}
```

```
int daysLeft(int x) {  
    if (x == 0)  
        return 0;  
    else  
        return x - 1;  
}
```

Each method call
takes time, and causes
extra jump instructions
to be executed

```
int whenToEvaluateZing(int y) {  
    int temp = 0;  
    if (y == 0) temp += 0; else temp += y - 1;  
    if (0 == 0) temp += 0; else temp += 0 - 1;  
    if (y+1 == 0) temp += 0; else temp += (y + 1) - 1;  
    return temp;  
}
```

Eliminating multiple method calls by inlining the method itself, speeds up the execution

```
int whenToEvaluateZing(int y) {  
    if (y == 0) return y;  
    else if (y == -1) return y - 1;  
    else return y + y - 1;  
}
```

Further optimizations can often be applied to speed up even more....

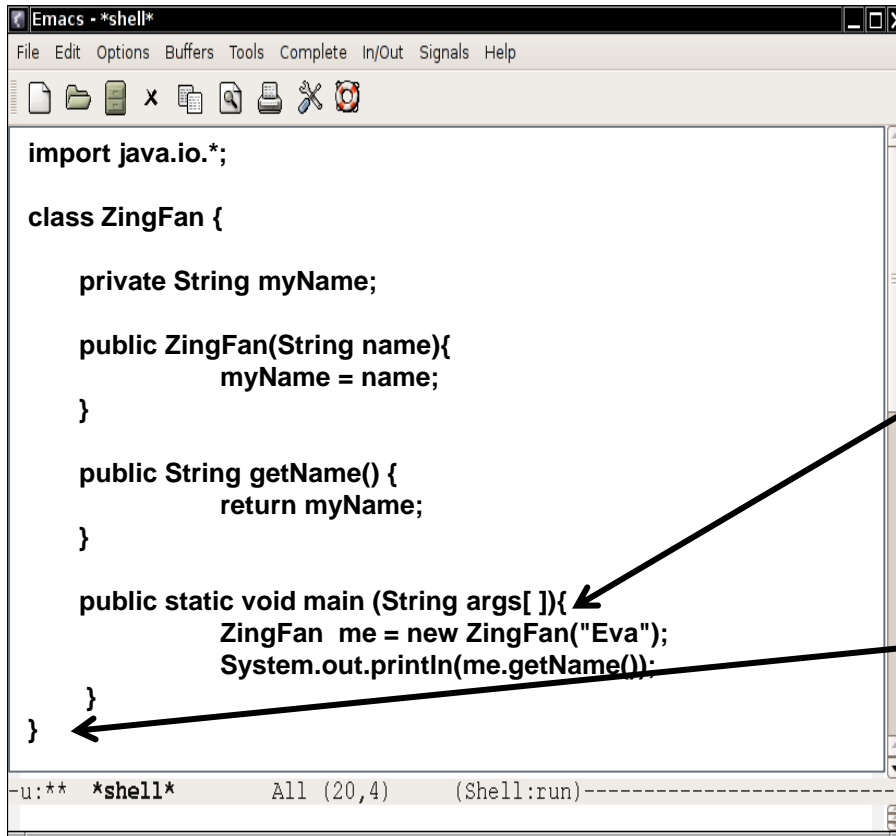
Summary on Portability, Compilers, and Optimizations

- Since the JVM's Compiler does the “translation” and optimization for you, you don't need to think about:
 - Different HW instruction sets
 - How to write your methods in a more instruction friendly way
 - How calls to other methods may impact execution performance
- Unless you want to? 😊

What is a Java Virtual Machine (JVM)?

- Key benefits of Java, enabled by the JVM
 - Portability
 - Dynamic Memory Allocation
- Other error-preventing and convenient services of the JVM
 - Consistent Thread Model
 - Optimizes and Manages Locking
 - Enforces Type Safety
 - Dynamic Code Loading
 - Quick high-quality Time Access (e.g. `currentTimeMillis`, `nanoTime`)
 - Internal introspection services
 - Access to huge pre-built library
 - Access to OS and environmental information

No Need to Be Explicit



```
Emacs - *shell*
File Edit Options Buffers Tools Complete In/Out Signals Help

import java.io.*;

class ZingFan {

    private String myName;

    public ZingFan(String name){
        myName = name;
    }

    public String getName() {
        return myName;
    }

    public static void main (String args[]){
        ZingFan me = new ZingFan("Eva");
        System.out.println(me.getName());
    }
}
```

-u:** *shell* All (20,4) (Shell:run)-----

**Just type “new” to
get the memory you
need**

**No need to track or
free used memory!**

Garbage Collection

- Dynamic Memory Management – The perceived pain that really comes with all the goodness
- Allocation and Garbage Collection
 - Parallel
 - Concurrent
 - Generational
 - Fragmentation and Compaction
 - The torture of tuning most JVMs

Allocation & Garbage Collection

- Java Heap (-Xmx)
- JVM Internal Memory
- Top: RES / RSS --- total memory footprint

Java Heap

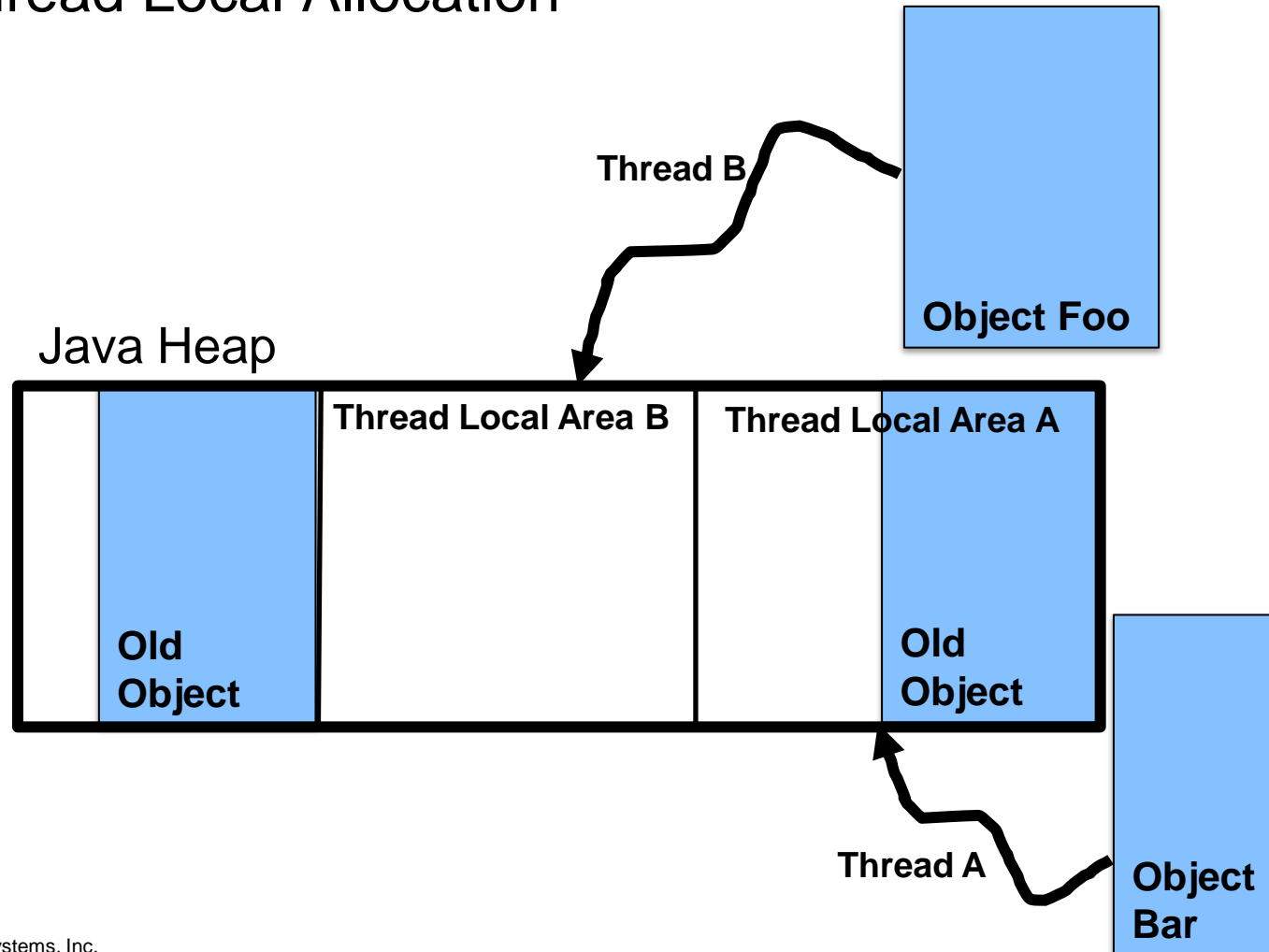
- Where all Java Objects are allocated

JVM Internal Memory

- Code Cache
- VM threads
- VM Structs
- GC Memory

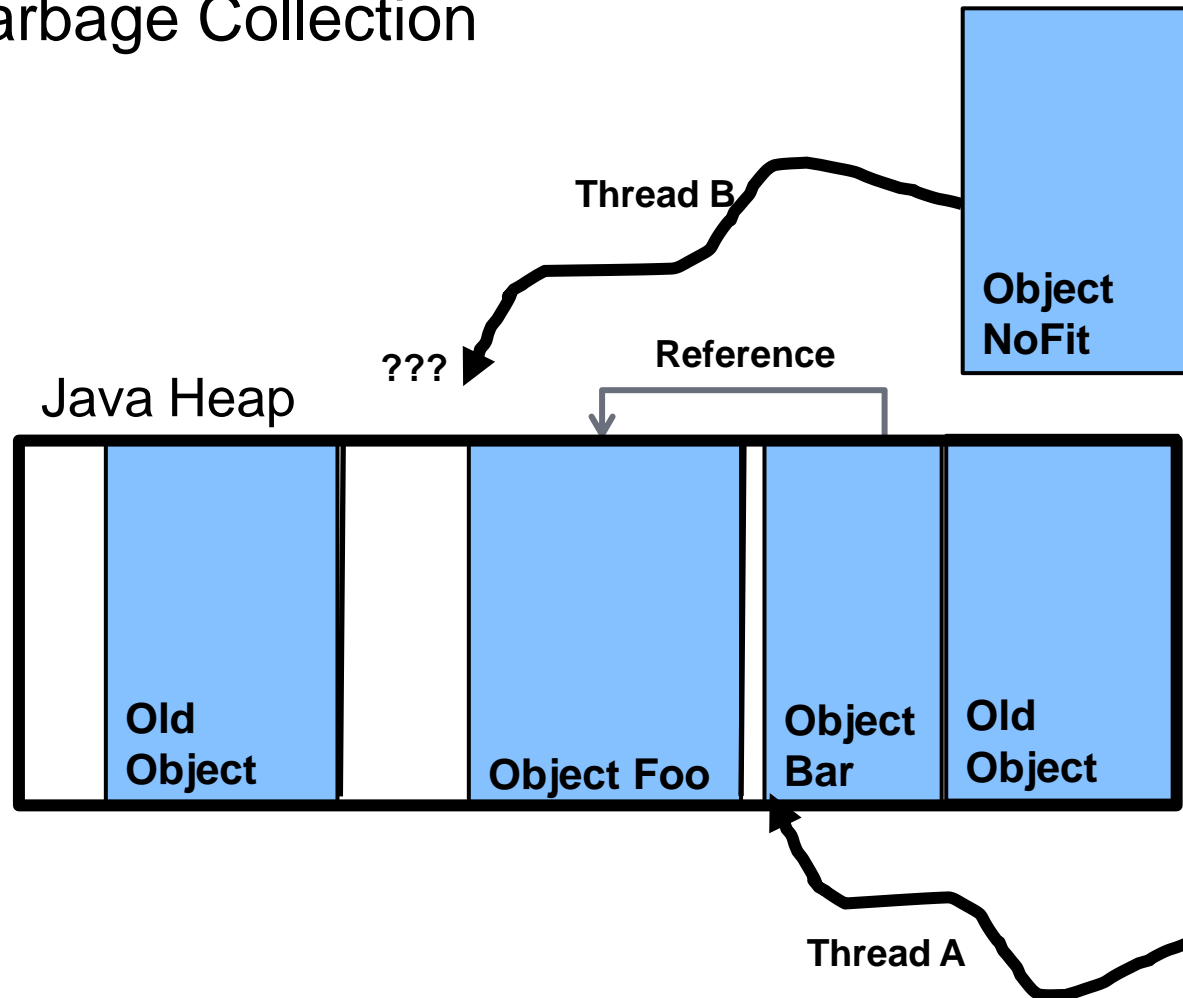
Allocation & Garbage Collection

- Thread Local Allocation



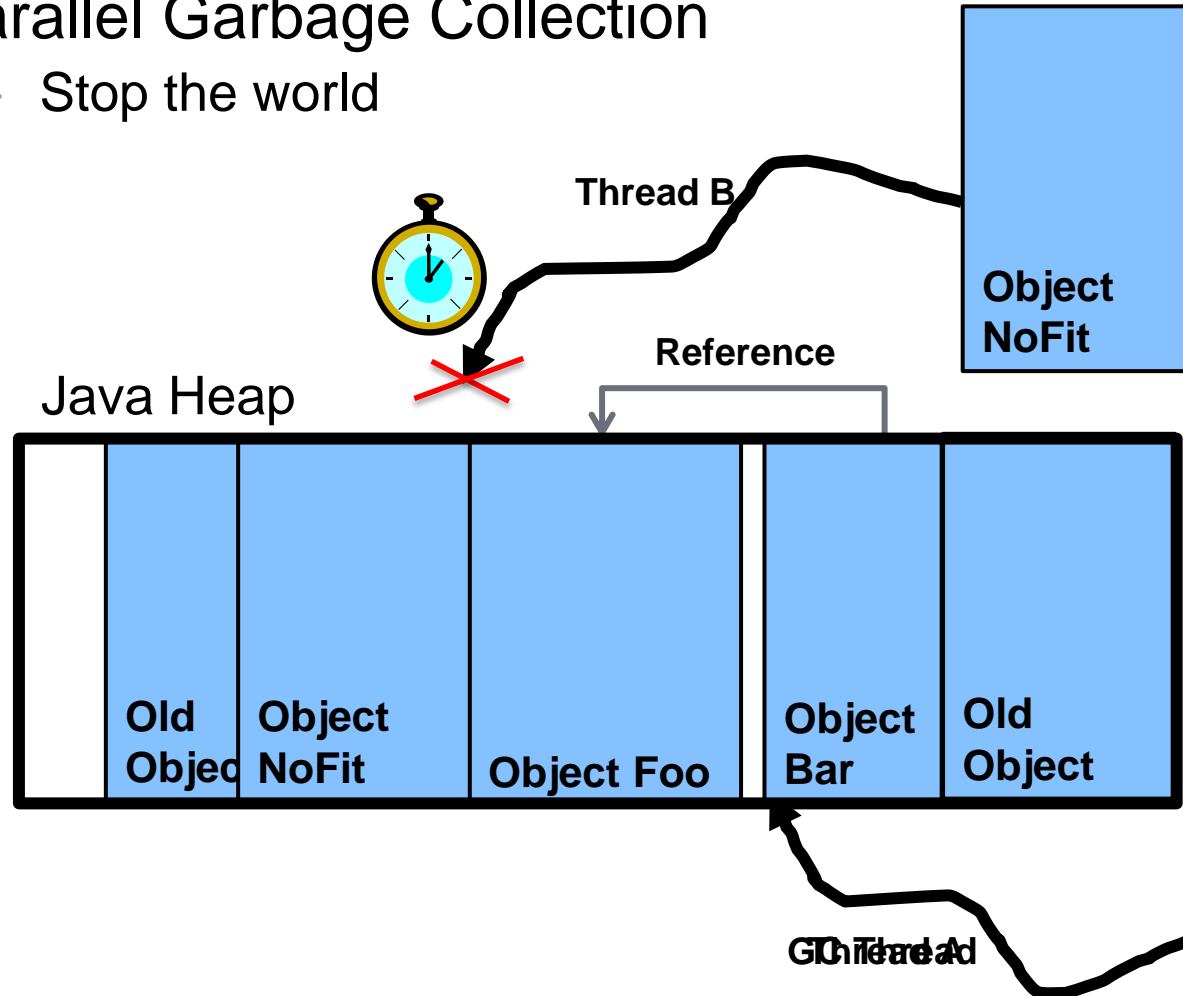
Allocation & Garbage Collection

- Garbage Collection



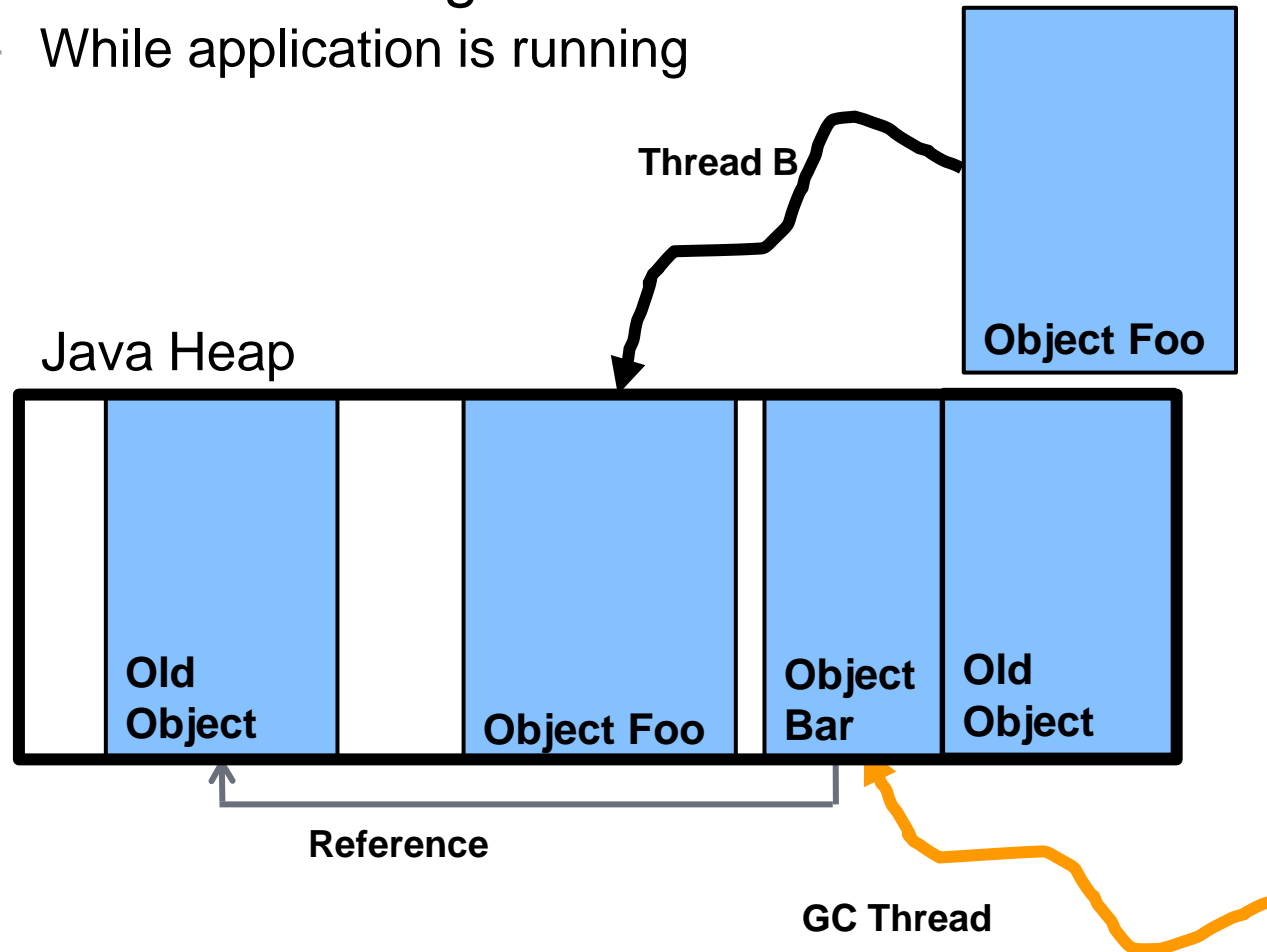
Garbage Collection

- Parallel Garbage Collection
 - Stop the world



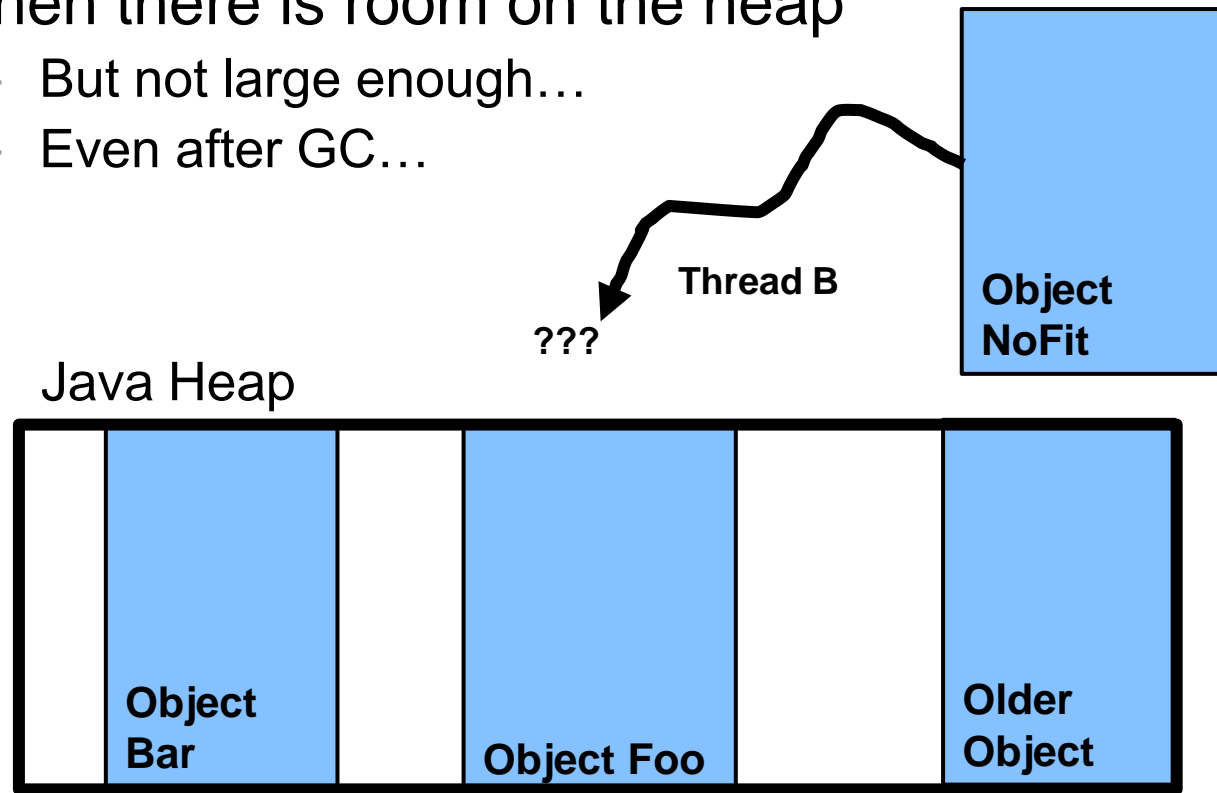
Garbage Collection

- Concurrent Garbage Collection
 - While application is running



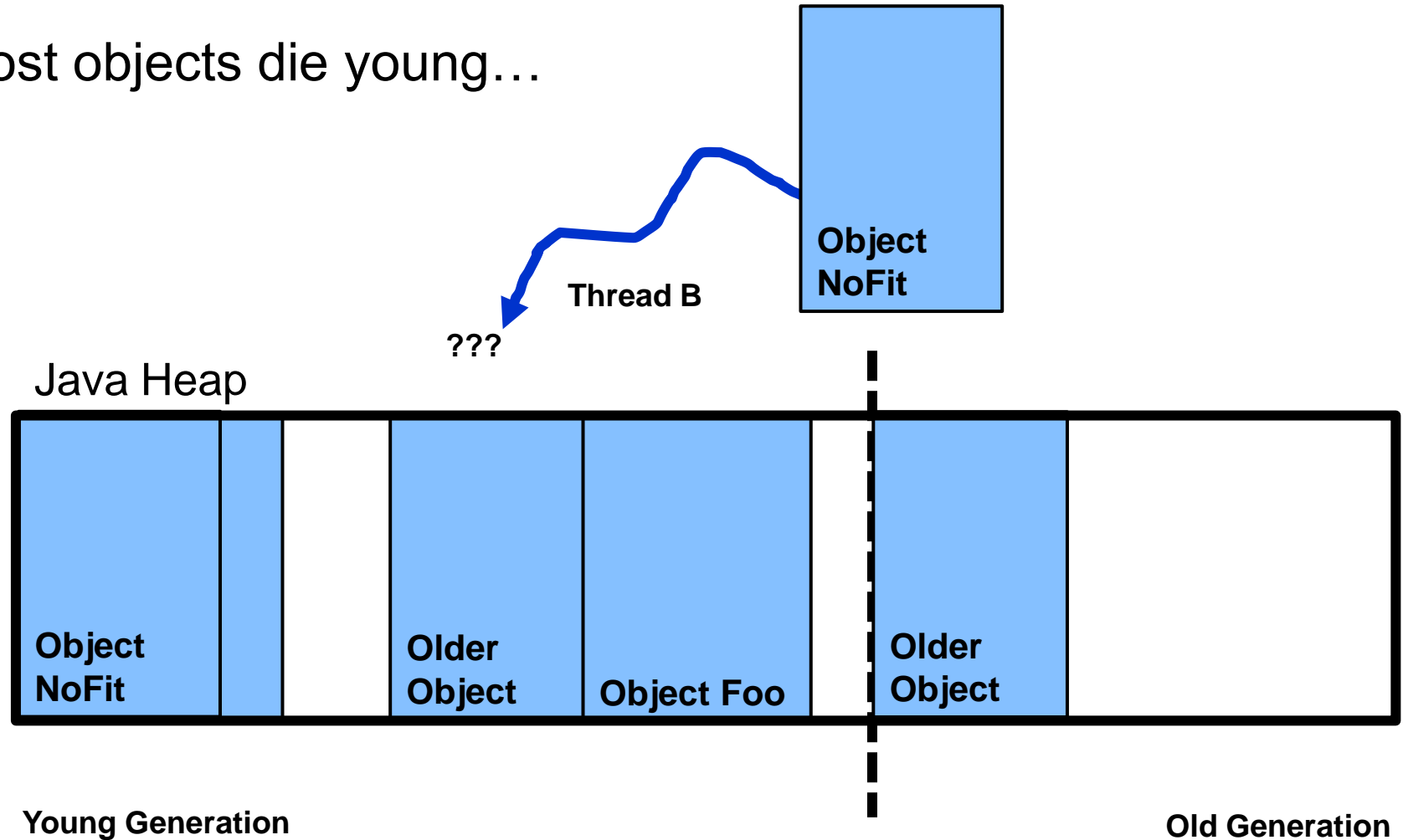
Fragmentation

- When there is room on the heap
 - But not large enough...
 - Even after GC...



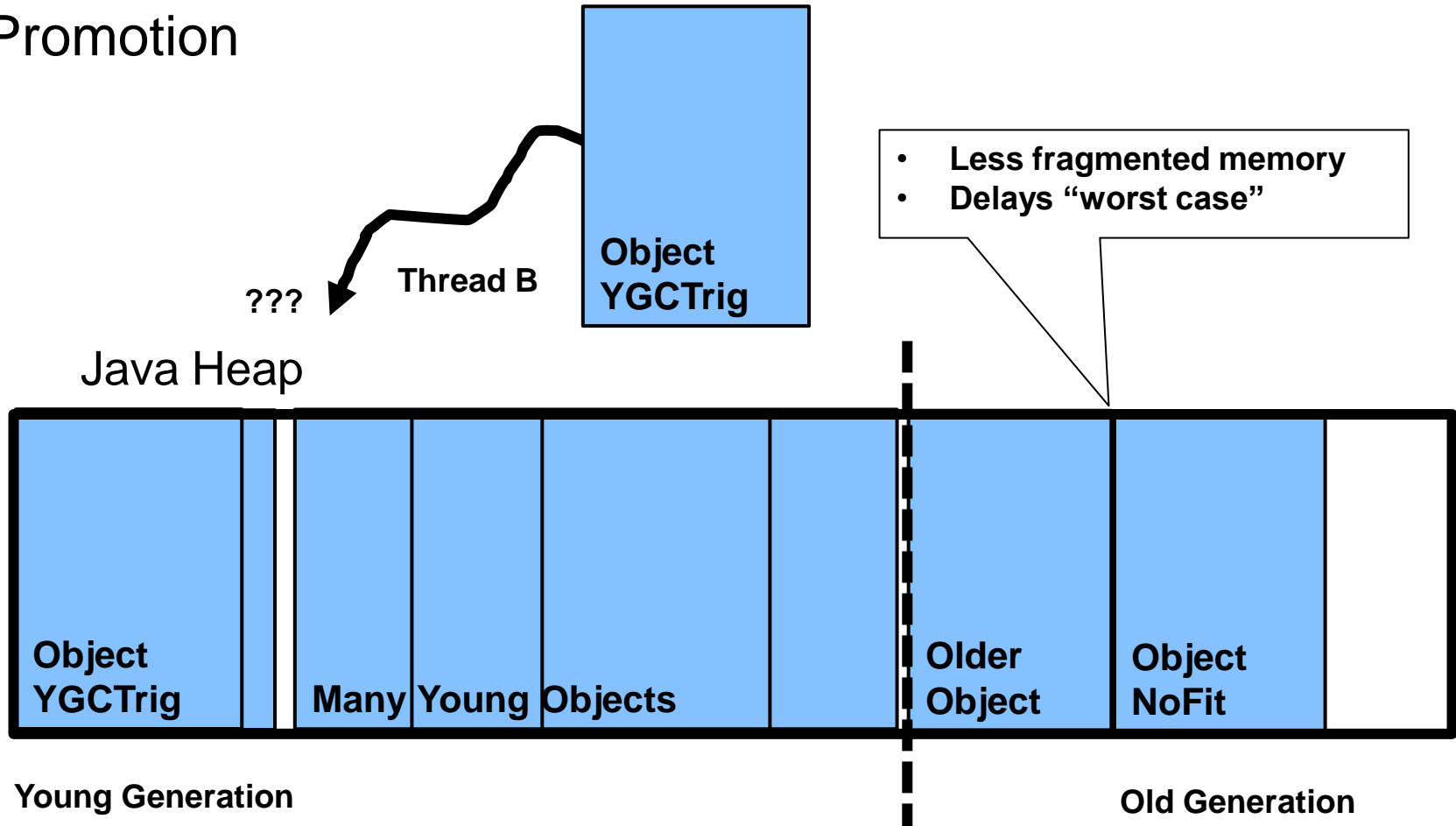
Generational Garbage Collection

- Most objects die young...

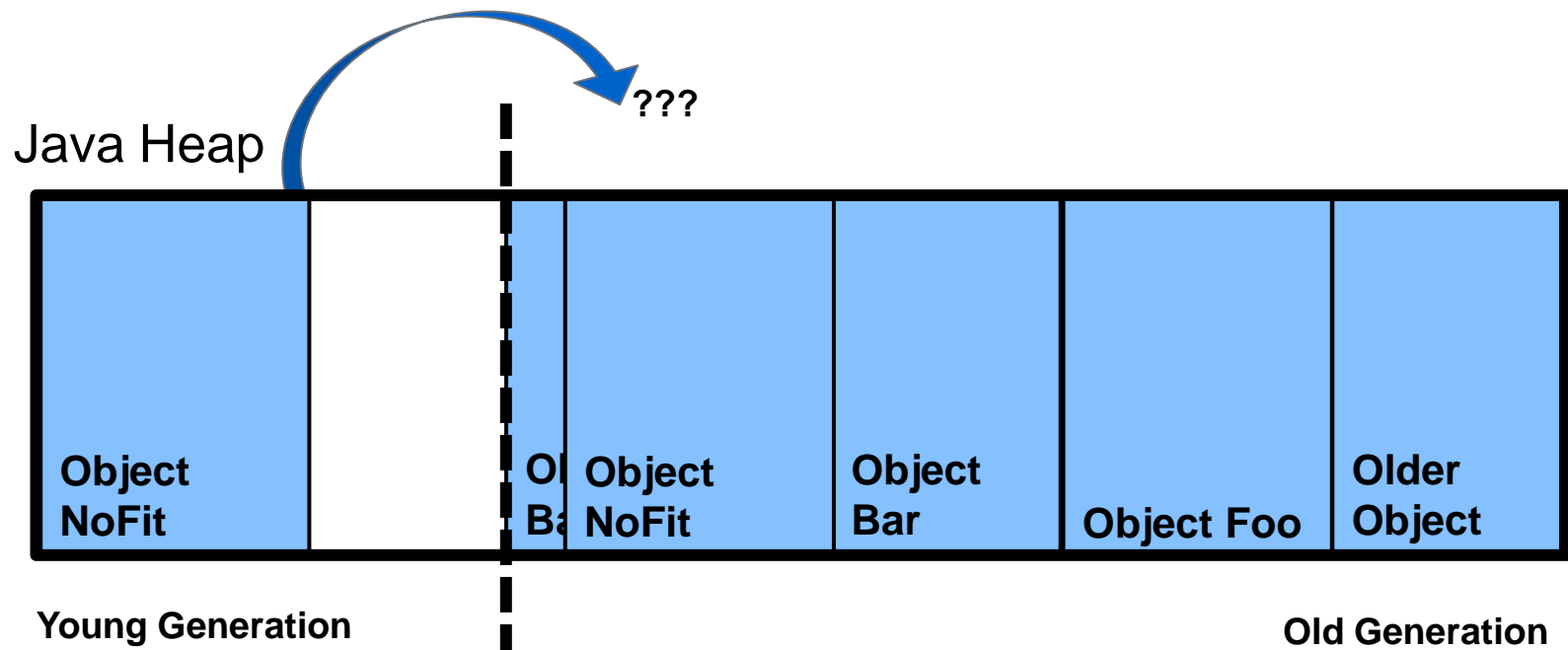


Generational Garbage Collection

- Promotion



- Move objects together
 - To fit larger objects
 - Eliminate small un-useful memory gaps



- Generational GC helps delay, but does not “solve” fragmentation
- Compaction is inevitable
 - When moving objects, you need to update references
 - Most JVMs have to stop the world as part of compaction
- Many approaches to shorten compaction impact
 - Partitioned compaction
 - Time controlled compaction
 - “Best result” calculations on what areas to compact
 - Reference counting

Pain of Tuning

- -Xmx – has to be set “right”
 - Exactly the amount you need for your worst case load
 - Too much – waste of resources
 - Too little – OutOfMemoryError
 - Unpredictable loads and missconfiguration cause a lot of down-time
- Example of production-tuned CMS:

```
java -server -Xmx4096m -Xms4096m -Xss512k -XX:MaxPermSize=256M -XX:NewSize=1800M -  
XX:MaxNewSize=1800M -XX:+UseNUMA -XX:+UseTLAB -XX:TLABSize=256k -XX:-ResizeTLAB -XX:+DoEscapeAnalysis -  
XX:+UseBiasedLocking -XX:+UseConcMarkSweepGC -XX:ParallelGCThreads=4 -XX:MaxGCPauseMillis=20 -  
XX:+UseParNewGC -XX:TargetSurvivorRatio=100 -XX:SurvivorRatio=4 -XX:-UseAdaptiveSizePolicy -XX:-  
UsePSAdaptiveSurvivorSizePolicy -XX:+BindGCTaskThreadsToCPUs -XX:+UseGCTaskAffinity -  
XX:+CMSParallelRemarkEnabled -XX:+CMSParallelSurvivorRemarkEnabled -XX:+CMSConcurrentMTEEnabled -  
XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:-UseAdaptiveGenerationSizePolicyAtMinorCollection -XX:-  
UseAdaptiveGenerationSizePolicyAtMajorCollection -XX:-UseAdaptiveGenerationSize=2M -XX:CMSMarkStackSizeMax=4M -  
XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=60 -XX:-TraceClassUnloading -  
XX:+PrintGCApplicationConcurrentTime -XX:+PrintGCApplicationStoppedTime -XX:+PrintTenuringDistribution -  
XX:+PrintHeapAtGC -XX:+PrintVMQWaitTime -XX:+PrintGCDetails -XX:+PrintTLAB
```

Biggest Java Scalability Limitation

- **For MOST JVMs, compaction pauses are the biggest current challenge and key limiting factor to Java scalability**
- The larger heap and live data / references to follow, the bigger challenge for compaction
- Today: most JVMs limited to 3-4GB
 - To keep “FullGC” pause times within SLAs
 - Design limitations to make applications survive in 4GB chunks
 - Horizontal scale out / clustering solutions
 - In spite of machine memory increasing over the years...
- *This is why I find Zing so interesting, as it has implemented concurrent compaction...*
 - *But that is not the topic of this presentation... ☺*

2c for the Road

What to (not) Think About

1. Why not use multiple threads, when you can?
 - Number of cores per server continues to grow...
2. Don't be afraid of garbage, it is good!
3. I personally don't like finalizers...error prone, not guaranteed to run (resource wasting)
4. Always be careful around locking
 - If it passes testing, hot locks can still block during production load
5. Benchmarks are often focused on throughput, but miss out on real GC impact – test your real application!
 - “Full GC” never occurs during the run, not running long enough to see impact of fragmentation
 - Response time std dev and outliers (99.9...%) are of importance for a real world app, not throughput alone!!

- JVM – a great abstraction, provides convenient services so the Java programmer doesn't have to deal with environment specific things
- Compiler – “intelligent and context-aware translator” who helps speed up your application
- Garbage Collector – simplifies memory management, different flavors for different needs
- Compaction – an inevitable task, which impact grows with live size and data complexity for most JVMs, and the current largest limiter of Java Scalability

For the Curious: What is Zing?

- Azul Systems has developed scalable Java platforms for 8+ years
 - **Vega** product line based on proprietary chip architecture, kernel enhancements, and JVM innovation
 - **Zing** product line based on x86 chip architecture, virtualization and kernel enhancements, and JVM innovation
- Most famous for our Generational Pauseless Garbage Collector, which performs fully *concurrent compaction*



eva.andreasson@azulsystems.com
<http://twitter.com/AzulSystemsPM>
www.azulsystems.com/zing

- For more information on...

- ...JDK internals: <http://openjdk.java.net/> (JVM source code)

- ...Memory management:

- http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf (a bit old, but very comprehensive)

- ...Tuning:

- http://download.oracle.com/docs/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/tune_stable_perf.html (watch out for increased rigidity and re-tuning pain)

- ...Generational Pauseless Garbage Collection:

- <http://www.azulsystems.com/webinar/pauseless-gc> (webinar by Gil Tene, 2011)

- ...Compiler internals and optimizations:

- <http://www.azulsystems.com/blogs/cliff> (Dr Cliff Click's blog)