Agenda

• Performance Analysis of Java Application
• Performance Bottlenecks
• Performance Tuning in Java
• Java Performance Tuning Tools
• Developing High Performance Java Application – Checklist
• Q&A
The goal of performance analysis in JAVA application is to determine which parts of a program require optimization to increase its execution speed or lower its memory usage.

• Primarily used for…
  – pro-active bottleneck identification.
  – "fixing" Scenarios that are not meeting requirements/SLAs
Performance Bottleneck

• A Bottleneck can be any factor that prevents the system in meeting the performance target

• It is a resistance to the flow of data

• Bottleneck can be within any layer of the technology stack or the infrastructure
Performance Bottleneck

• “In a given time instance only one bottleneck cause the system to under perform”

• “Strength of the chain is equal to the weakest link”
Bottleneck Classification

Software

Application Layer
- Objects
- Methods

Database Layer
- Query
- Locks

Hardware

<table>
<thead>
<tr>
<th>Processor</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Layer</td>
<td>Network Layer</td>
</tr>
</tbody>
</table>
Performance Problems In Java

**Two major performance problem in Java Application**

- Excess Creation of Objects
- Excess Garbage Collection

How to avoid these problems to eliminate performance bottlenecks?
Excess Object Creation

- Objects need to be created before they can be used, and garbage-collected when they are finished with.

- The more objects you use, the more garbage-cycling happens, the CPU cycle wasted.

- Each object creation is roughly as expensive as a `malloc` in C, or a `new` in C++, and there is no easy way of creating many objects together, so you cannot take advantage of efficiencies you get using bulk allocation.
Excess Object Creation

Example: A Person object that holds a Name object, consisting of first name and last name and an Address object, with street, number, etc. These three objects can be collapsed down to just the Person object, with all the fields moved up to the Person class.

continue....
public class Person
{
    private Name name;
    private Address address;
}

class Name
{
    private String firstName;
    private String lastName;
    private String[] otherNames;
}
class Address
{
    private int houseNumber;
    private String houseName;
    private String streetName;
    private String town;
    private String area;
    private String greaterArea;
    private String country;
    private String postcode;
}
These three classes collapse into one class:

```java
public class Person {
    private String firstName;
    private String lastName;
    private String[] otherNames;
    private int houseNumber;
    private String houseName;
    private String streetName;
    private String town;
    private String area;
    private String greaterArea;
    private String country;
    private String postCode;
}
```
Performance Tuning in Java

Reuse Object

• Objects are expensive to create so it is reasonable to reuse the same object.

• This requires awareness of when not to create new object.

• Look at the object and consider whether it is possible to reset the fields and then reuse the object, rather than throw it away and create another

• Important for objects that are constantly used and discarded: for example, in graphics processing, objects such as Rectangles, Points, Colors, and Fonts are used and discarded all the time. Recycling these types of objects can certainly improve performance.
Manage Pool Of Objects

• While not using the retained objects, we are holding on to more memory than if we simply discarded those objects, and this reduces the memory available to create other objects.

• Balance the need to have some free memory available against the need to improve performance by reusing objects.

• The space taken by retaining objects for later reuse is significant only for very large collections, and you should certainly know which ones these are in your application.
Canonicalizing Objects

This activity of replacing multiple copies of an object with just a few objects is often referred to as *canonicalizing* objects.

**Example:**

```java
Boolean t1 = new Boolean(true);
System.out.println(t1!=Boolean.TRUE);
System.out.println(t1.==(Boolean.TRUE));
```

produces the output:

false
true
Enumerating constants

Another canonicalization technique often used is replacing constant objects with integers this is called Enumeration. Enumeration can provide both speed and memory advantages. The enumeration requires less memory than the equivalent strings and makes network transfers faster.

Example: public interface GENDER

```java
{
    public static final int FEMALE=1;
    public static final int MALE=2;
}
```

Rather than use the strings "female" and "male", you should use a constant defined in an interface.
Comparison verses Identity

- Comparisons are faster compared to identity
- Comparison can be used instead of the equality comparison.

**Example:**

```java
you can use
this.gender == FEMALE;

instead of:
this.gender.equals("Female");
```
Avoid Excess Garbage Collection

• The canonicalization techniques are one way to avoid garbage collection fewer objects means less to garbage-collect.

• The pooling technique tends to reduce garbage-collection requirements, partly because you are creating fewer objects by reusing them, and partly because you deallocate memory less often by holding on to the objects you have allocated.

• Reducing garbage collection by using primitive data type. Hold an object in a primitive data-type format rather than another format as primitive data type use less memory.
Performance Tuning in Java

Lazy Initialization

• Technique of delaying object creation until the last possible moment. This technique is useful for avoiding unnecessary object creation when only a few objects are used although many possible objects can be created.

• Many objects that need to be created and initialized, and most of these objects will be used, but not immediately. In this case, it can be useful to spread out the load of object initialization so you don't get one large hit on the application. It may be better to let a background thread initialize all the objects slowly or to use lazy initialization to take many small or negligible hits, thus spreading the load over time.
There are many tools available in the market to analyze and monitor Java applications which help us to find which part of the code should be optimized. Some of them are listed below:

- **NetBean Profiler for analyzing java application**
- **JAMon** – Java application monitor.
- **GCViewer** - visualizes verbose garbage collection data generated by Sun and IBM Java Virtual Machines (JVM).
- **HPjmeter** – specific to Methods/Object.
- **HPjtune** - JVM
Java Performance Tuning Tools

NetBean Profiler for analyzing Java Applications
Java Performance Tuning Tools

Introduction

The NetBeans Profiler is a powerful tool that provides important information about the runtime behavior of an application. The NetBeans Profiler help in tracking.

- Thread state
- CPU performance
- Memory usage.
- Object Creation
- Method creation
Thread State

- Threads (Timeline) Shows current and historical thread state, updated as the application runs.
- Threads (Details) Shows a summary of thread state information for a single thread.

**Green**: the thread is either running or is ready to run.
**Purple**: the thread is sleeping; for example it called Thread.sleep().
**Yellow**: the thread is waiting in a call to Object.wait().
**Red**: the thread is blocked while trying to enter a synchronized block or method.
Java Performance Tuning Tools

CPU Performance

This tool have CPU snapshot contains the following tabs:

• **Call Tree** Displays the Calling Context Tree (CCT) showing the method call chain and the time/number of invocations for executing threads and methods in each context. (A context is a unique chain of method calls leading to the method's invocation.)

• **Hot Spots** Shows the total execution time and number of invocations for each method, irrespective of the context.
Java Performance Tuning Tools

CPU Performance

Combined view of Call tree and hotspot
Java Performance Tuning Tools

Memory Usage

This tool has snapshots containing the following tab:

- **Allocated Objects**: The number of objects that the NetBeans Profiler is actually monitoring.

- **Live Objects**: The number of the Allocated Objects that are still on the JVM's heap and are therefore taking up memory.

- **Live Bytes**: The two columns show the amount of heap memory being used by the Live Objects. One column displays a graph, the other displays text.

- **Avg. Age**: This value is calculated using the Live Objects. The age of each object is the number of garbage collections that it has survived. The sum of the ages divided by the number of Live Objects is the Avg. Age.
# Java Performance Tuning Tools

## Memory Usage

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Live Bytes</th>
<th>Live Objects</th>
<th>Allocated Objects</th>
<th>Avg. Age</th>
<th>Generations</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>23,110 B</td>
<td>338</td>
<td>1,476</td>
<td>8.0</td>
<td>15</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>12,320 B</td>
<td>514</td>
<td>2,578</td>
<td>8.8</td>
<td>14</td>
</tr>
<tr>
<td>sun.font.TrueTypeFont</td>
<td>10,120 B</td>
<td>422</td>
<td>413</td>
<td>5.0</td>
<td>5</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>9,528 B</td>
<td>397</td>
<td>413</td>
<td>8.7</td>
<td>16</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>4,144 B</td>
<td>39</td>
<td>46</td>
<td>6.0</td>
<td>11</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>3,912 B</td>
<td>163</td>
<td>179</td>
<td>7.1</td>
<td>12</td>
</tr>
<tr>
<td>sun.font.TrueTypeFont</td>
<td>2,704 B</td>
<td>26</td>
<td>26</td>
<td>5.0</td>
<td>5</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>2,056 B</td>
<td>25</td>
<td>25</td>
<td>5.0</td>
<td>5</td>
</tr>
<tr>
<td>sun.font.TrueTypeFont</td>
<td>1,904 B</td>
<td>11</td>
<td>14</td>
<td>7.5</td>
<td>7</td>
</tr>
<tr>
<td>int</td>
<td>1,504 B</td>
<td>54</td>
<td>679</td>
<td>9.1</td>
<td>8</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>1,400 B</td>
<td>35</td>
<td>35</td>
<td>6.1</td>
<td>11</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>1,248 B</td>
<td>39</td>
<td>161</td>
<td>5.8</td>
<td>10</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>1,040 B</td>
<td>1</td>
<td>1</td>
<td>8.0</td>
<td>1</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>1,040 B</td>
<td>1</td>
<td>1</td>
<td>8.0</td>
<td>1</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>848 B</td>
<td>53</td>
<td>55</td>
<td>8.1</td>
<td>7</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>696 B</td>
<td>1</td>
<td>1</td>
<td>14.0</td>
<td>1</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>672 B</td>
<td>28</td>
<td>32</td>
<td>5.4</td>
<td>8</td>
</tr>
<tr>
<td>java.util.LinkedList$Node</td>
<td>672 B</td>
<td>21</td>
<td>25</td>
<td>6.8</td>
<td>2</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>656 B</td>
<td>41</td>
<td>41</td>
<td>6.9</td>
<td>2</td>
</tr>
<tr>
<td>java.util.HashMap$Entry</td>
<td>568 B</td>
<td>8</td>
<td>14</td>
<td>5.6</td>
<td>5</td>
</tr>
</tbody>
</table>

Copyright © 2008 Accenture All Rights Reserved.
Java Performance Tuning Tools

Memory Usage
Developing High Performance Java Applications - Checklist

• Reduce the number of temporary objects being used, especially in loops.
• Avoid creating temporary objects within frequently called methods.
• Presize collection objects.
• Reuse objects where possible.
• Empty collection objects before reusing them. (Do not shrink them unless they are very large.)
• Use custom conversion methods for converting between data types (especially strings and streams) to reduce the number of temporary objects.
• Define methods that accept reusable objects to be filled in with data, rather than methods that return objects holding that data. (Or you can return immutable objects.)
Developing High Performance Java Applications - Checklist

- Canonicalize objects wherever possible. Compare canonicalized objects by identity.
- Use primitive data types instead of objects as instance variables.
- Preallocate storage for large collections of objects by mapping the instance variables into multiple arrays.
- Eliminate object-creation bottlenecks by moving object creation to an alternative time.
- Create objects early, when there is spare time in the application, and hold those objects until required.
- Use lazy initialization when there are objects or variables that may never be used, or when you need to distribute the load of creating objects.
"The biggest difference between Time and Space is that you cant re-use Time"

Megha Fulfagar
Performance Improvement In Java Application

Q & A