Monitoring Java enviroment / applications

Uroš Majcen
uros@quest-slo.com

Java is Everywhere

Java in Mars Rover

With the help of Java Technology, and the Jet Propulsion Laboratory (JPL), scientists can control the Mars Rover all the way from planet Earth.

Smart Phones

The Blackberry, Android and many other PDAs and cell phones use Java as an operating system.

BMW

Many car companies including BMW have on-board computer systems that are run by Java.

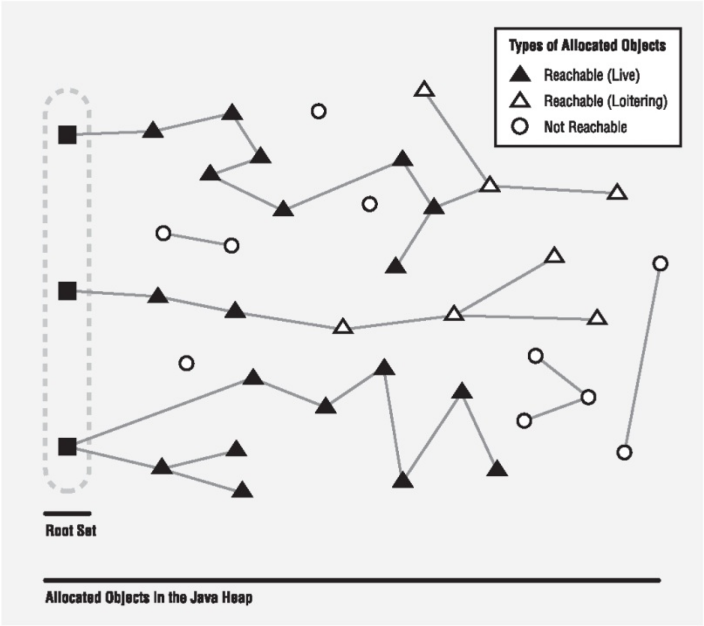
hroug
hrvatska udruga oracle korisnika

Application Design and Performance

QUEST SOFTWARE®
Simplicity At Work™

Origins of Poor Performance

- Majority of Java performance problems arise from:
  - Memory-Related Problems
  - Algorithm-Related Problems
- The most dramatic improvements in performance are often made at the application level
  - Fast hardware and highly tuned JVMs cannot overcome the inherent limitations of a poorly designed application



Loitering Objects in the Java Heap

Framework-Based Development

- Java development is undertaken in the context of application frameworks:
  - JFC/Swing
  - J2EE (Servlet/JSP/EJB)
- Several advantages:
  - Programmer Productivity
  - Reduced Development Time
  - Software Correctness

Framework-Based Development

- Unfortunately, little consideration is given to the *performance characteristics* of these frameworks
  - Space and time costs
  - Scalability
- Abstraction versus Implementation
  - To achieve good performance, you have to have some insight into the underlying implementation
- You need a toolset that provides a deep level of insight into your Java software

Memory and Performance Issues in Java

Memory Safety in Java

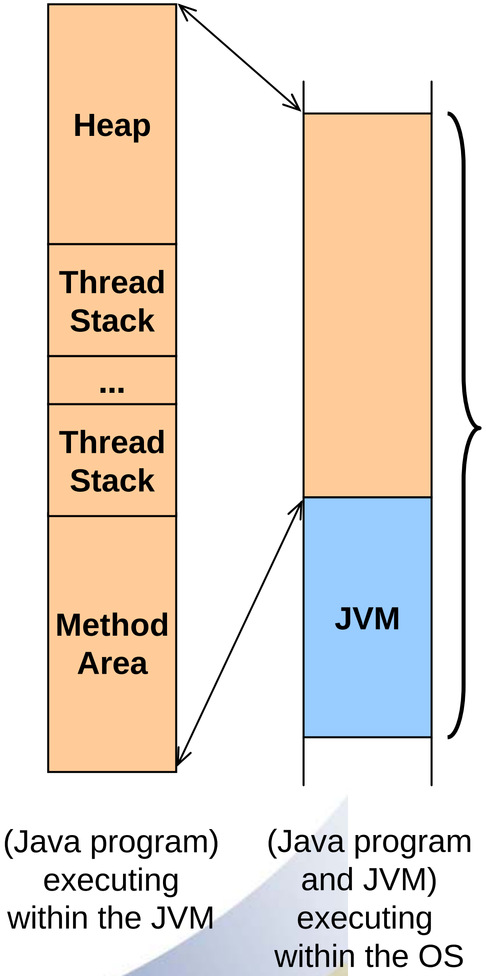
- Memory safety was a key aspect in the design of...
  - **The Java Language**
    - Absence of any form of pointer arithmetic
    - Can not directly reclaim object memory
  - **And the Java Virtual Machine (JVM)**
    - Bytecode instruction set
    - Runtime checks (array bounds, reference casts)
    - Garbage collection

Memory Safety in Java

- Entire classes of memory-related problems were eliminated
  - Buffer overruns
  - De-referencing stale pointers
  - Memory leaks
- However memory management issues remain
  - Loitering Objects
  - Object Cycling
- Either of these issues can easily undermine the performance of your application

JVM Runtime Data Areas

- **Heap**
  - The common memory pool where *all* objects and arrays are stored
- **Thread Stack(s)**
  - One stack *per thread of execution*
  - Each stack consists of a series of *method frames* (one per called method) which contain the method arguments and return value, the local variables within the method and a *bytecode operand stack* for intermediate results
- **Method Area**
  - Maintains the data structures for *each* loaded class in the JVM

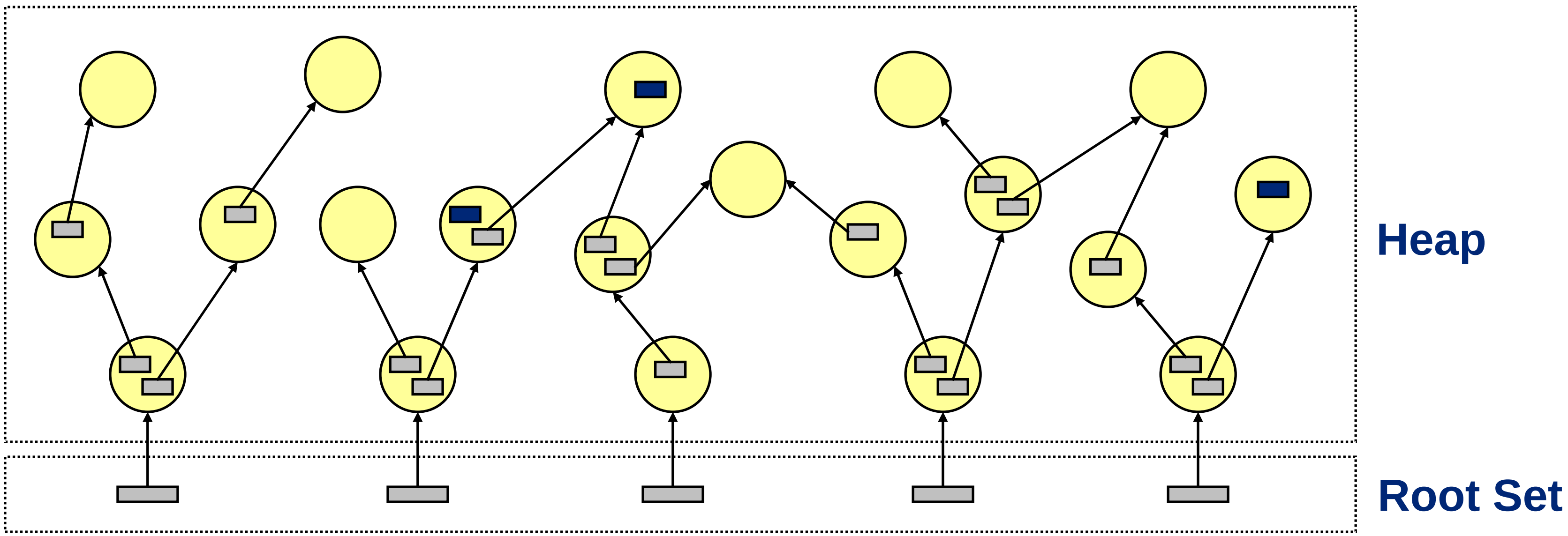


Java Memory Management

- Central to Java's memory management subsystem is the notion of **garbage collection (gc)**
  - Removes objects that are no longer needed
  - Undecidable in general, so Java uses an approximation...
    - Removes objects that are no longer ***reachable*** (accessible to the program at the beginning of a garbage collection cycle)
  - The ***reachability test*** starts at the heap's ***root set***

Reachable Objects

- Elements within the root set directly refer to objects within the heap of the JVM

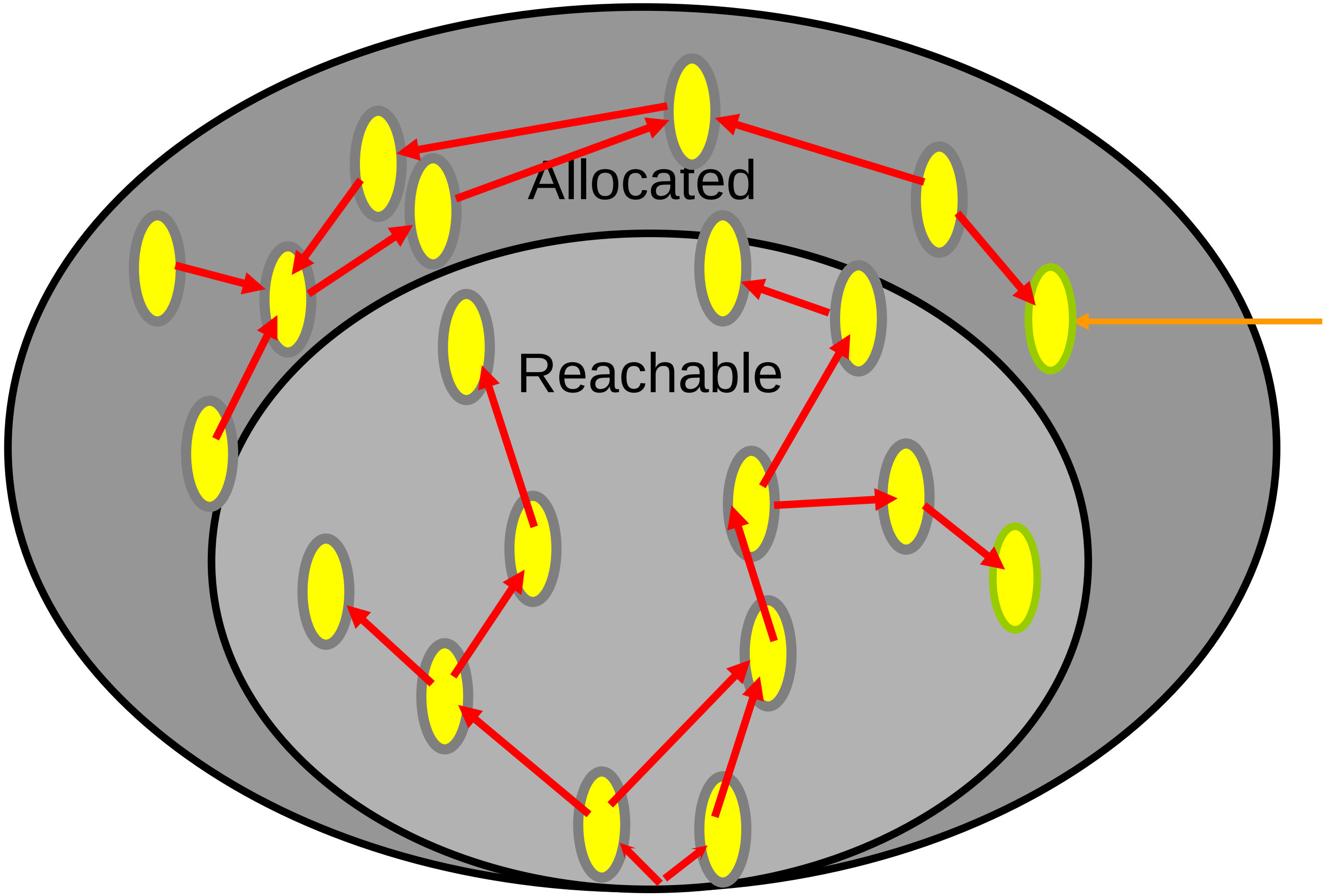


- Reference variables within those objects refer to further objects within the Heap (indirectly reachable from the Root Set)

Reachable Objects & GC

- At the beginning of a GC cycle, objects within the heap can be considered to be in one of two progressive “states”:
  - ***Allocated***
    - Exists within the JVM’s heap
  - ***Reachable***
    - A path exists (directly or indirectly) from a member of the root set, through a sequence of references, to that object

Reachable Objects & GC



At the beginning of a GC cycle, objects that are allocated *but no longer reachable* are reclaimed by the Garbage Collector

What is a “Memory Leak” in Java ?

- Memory leaks (as traditionally defined in C/C++) cannot occur in Java
  - That memory is reclaimed by the Garbage Collector
- However, Java programs can still exhibit the macro-level symptoms of traditional memory leaks
  - Heap size seemingly grows without bounds
- Occurs when objects that have outlived their usefulness to the application remain within the heap through successive garbage collections

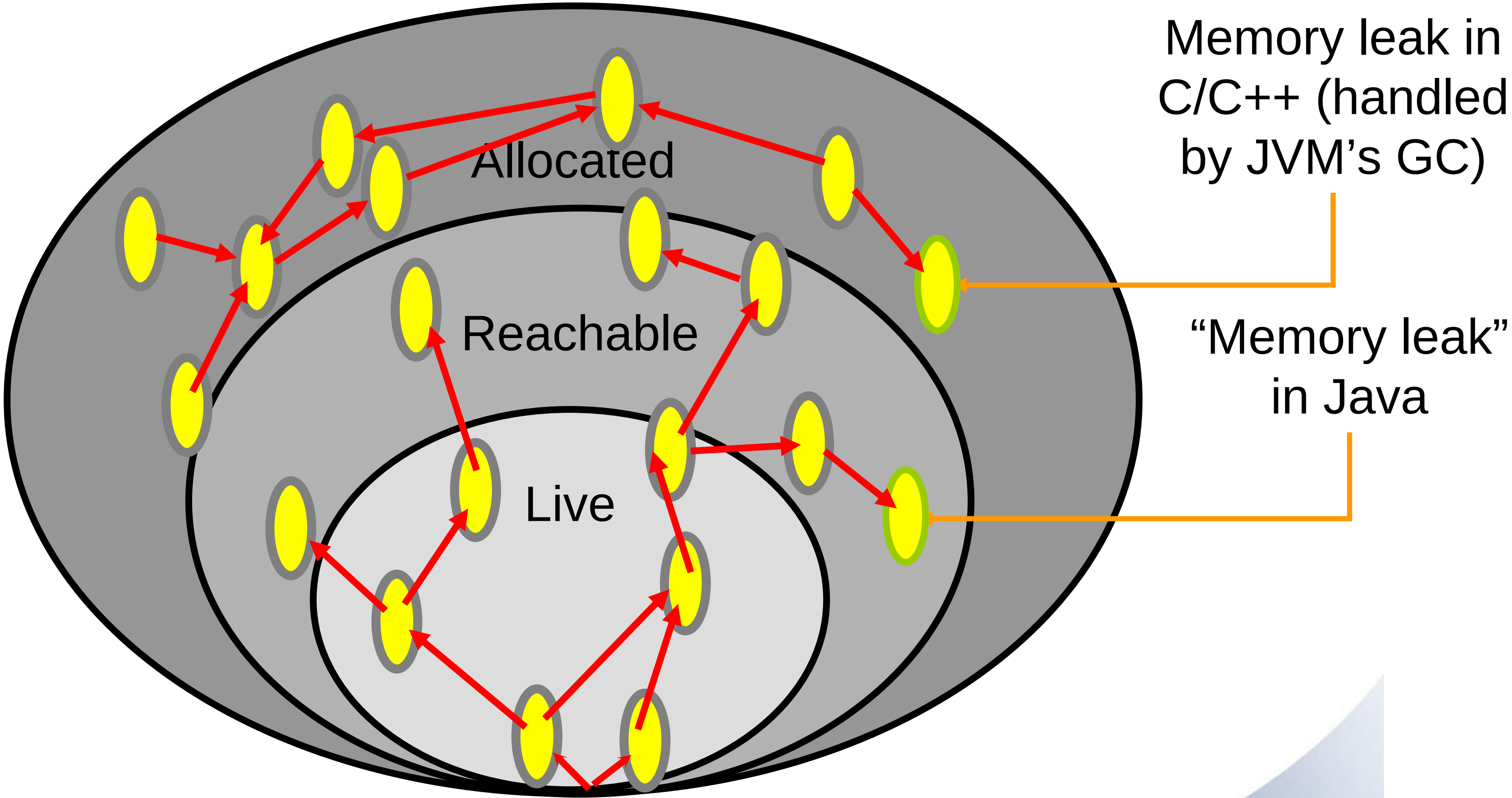
What is a “Memory Leak” in Java?

- We can extend the set of object states to three:
  - **Allocated**
    - Exists within the JVM’s heap
  - **Reachable**
    - A path exists (directly or indirectly) from a member of the root set, through a sequence of references, to that object
  - ***Live***
    - From the *intent* of the application’s design, the program will *use* the object (meaning at least one of its public fields will be accessed and/or one of its public methods will be invoked) along some *future* path of execution

Memory Leaks: C/C++ vs. Java

- Memory leak in C/C++
  - The object has been allocated, but it’s not reachable
    - **`malloc()/new`**, but forgot to **`free()/delete`** before overwriting the pointer to the object
- “Memory leak” in Java
  - The object is reachable, but it’s not *live*
    - The object has reached the end of its designed lifecycle and should be reclaimed, but an erroneous reference to it prevents the object from being reclaimed by the GC
  - Object is reachable to the GC, but the code to fix the leak may not be available to us
    - e.g. **`private`** reference field within an obfuscated class that you don’t have the source code to

What is a “Memory Leak” in Java?



Memory leak in C/C++ (handled by JVM’s GC)

“Memory leak” in Java

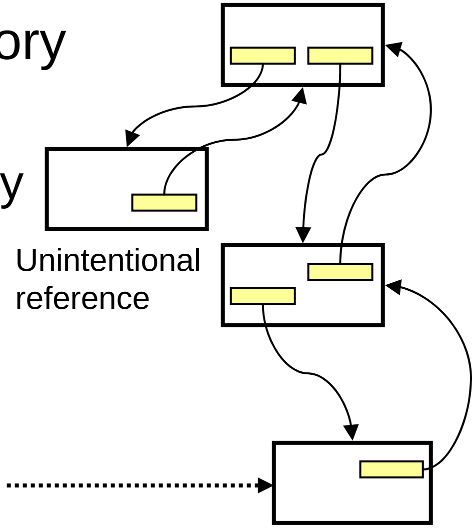
Loitering Objects

- The term “Memory Leak” has a lot of historical context from C/C++ and it doesn’t accurately describe the problem as it pertains to Java

- New term: **Loitering Object** or **Loiterer**
  - An object that remains within the Heap past its useful life to the application
  - Arise from an invalid reference that makes the object reachable to the GC

Loitering Objects

- Impact can be ***very*** severe
  - Rarely a single object, but an entire sub-graph of objects
- A single lingering reference can have massive memory impact (and a significant performance impact)
  - Overall process requires more memory than necessary
  - JVM's memory subsystem works harder
  - In the worst case, your Java application will throw an **`OutOfMemoryError`** and terminate



Failure to Remove Stale Object References from Data Structures

```
Object obj = new BigObject( );
set.add( obj );
...
...                 // Object is at the end of its life, but
obj = null;         // we forgot to remove it from the set !
```

```
Object obj = new BigObject( );
set.add( obj );
...
...
set.remove( obj )      // Remove the object first !
obj = null;
```

Reference Management

- The key to effective memory management in Java is ***effective reference management***
- What undermines effective reference management ?
  - Lack of awareness of the issue
  - Bad habits from C/C++ development
  - Class Libraries and Application Frameworks
    - Ill-defined reference management policies
    - Encapsulate flawed reference assignments
    - Tool (IDEs and others) generated software

Object Cycling

- One of the principal causes of performance loss in Java is the excessive creation of short life cycle objects
  - Objects typically exist only within the scope of a method
- Performance loss is due to...
  - Memory allocation within the JVM heap
  - Object initialization via chain of constructor calls
  - Enhanced garbage collection activity

Object Cycling

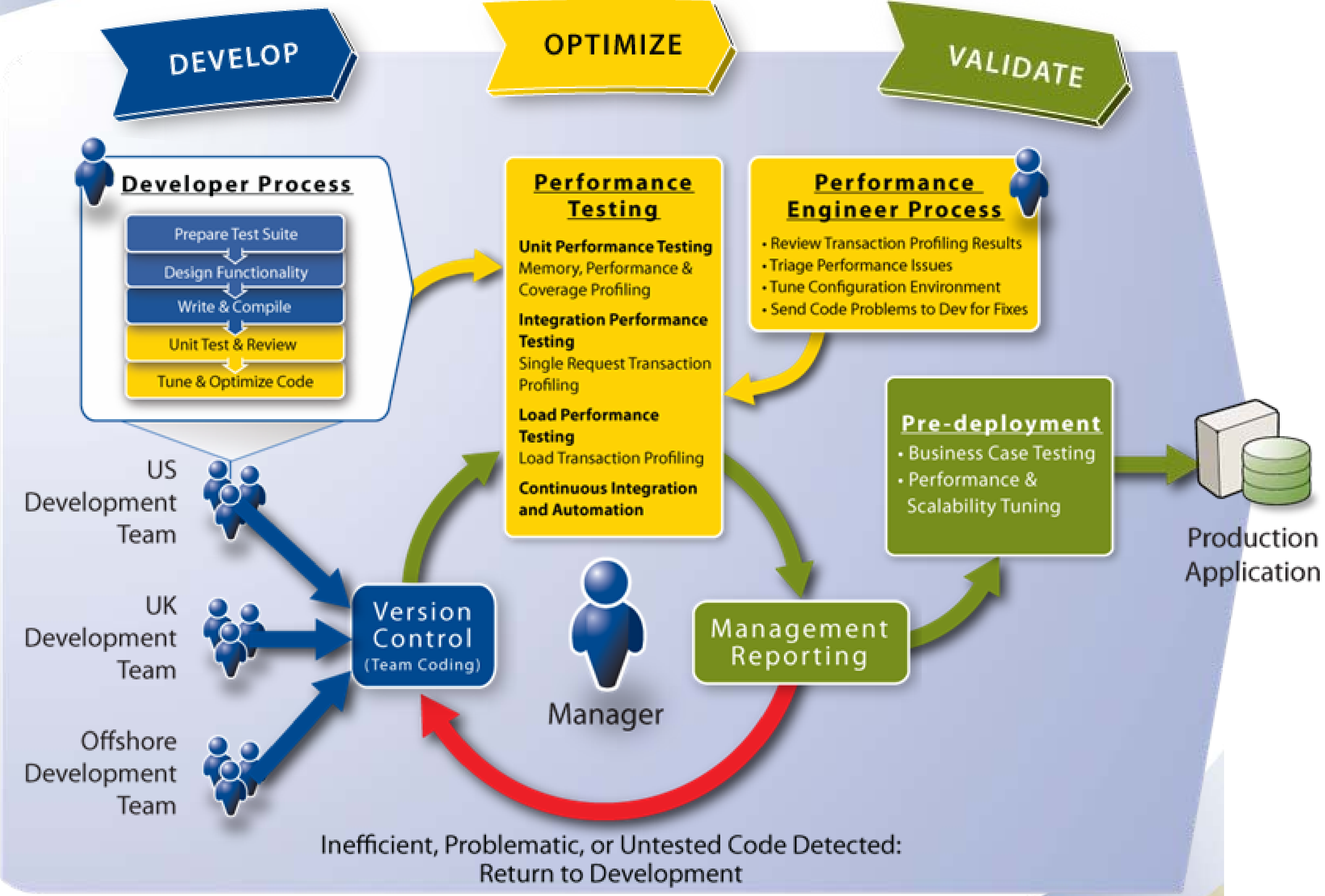
- As a performance investigator, you want to identify those methods in your application-level use case that are creating objects that are soon reclaimed by the garbage collector
- They are your ***first*** candidates for refactoring to improve performance

JProbe

The JProbe Java Profiling Solution

- Primary Capabilities
  - Identify and resolve **memory allocation issues** to ensure program efficiency and stability
  - Identify and resolve **code bottlenecks** to maximize program performance and scalability
  - Identify **unexecuted lines of code** during performance testing to ensure full test coverage
  - Enable **performance test automation** and **metrics reporting** to boost productivity and save valuable time

Development Best Practice Workflow



DEVELOP

OPTIMIZE

VALIDATE

Developer Process

- Prepare Test Suite
- Design Functionality
- Write & Compile
- Unit Test & Review
- Tune & Optimize Code

Performance Testing

Unit Performance Testing
Memory, Performance & Coverage Profiling

Integration Performance Testing
Single Request Transaction Profiling

Load Performance Testing
Load Transaction Profiling

Continuous Integration and Automation

Performance Engineer Process

- Review Transaction Profiling Results
- Triage Performance Issues
- Tune Configuration Environment
- Send Code Problems to Dev for Fixes

Pre-deployment

- Business Case Testing
- Performance & Scalability Tuning

Production Application

US Development Team

UK Development Team

Offshore Development Team

Version Control (Team Coding)

Manager

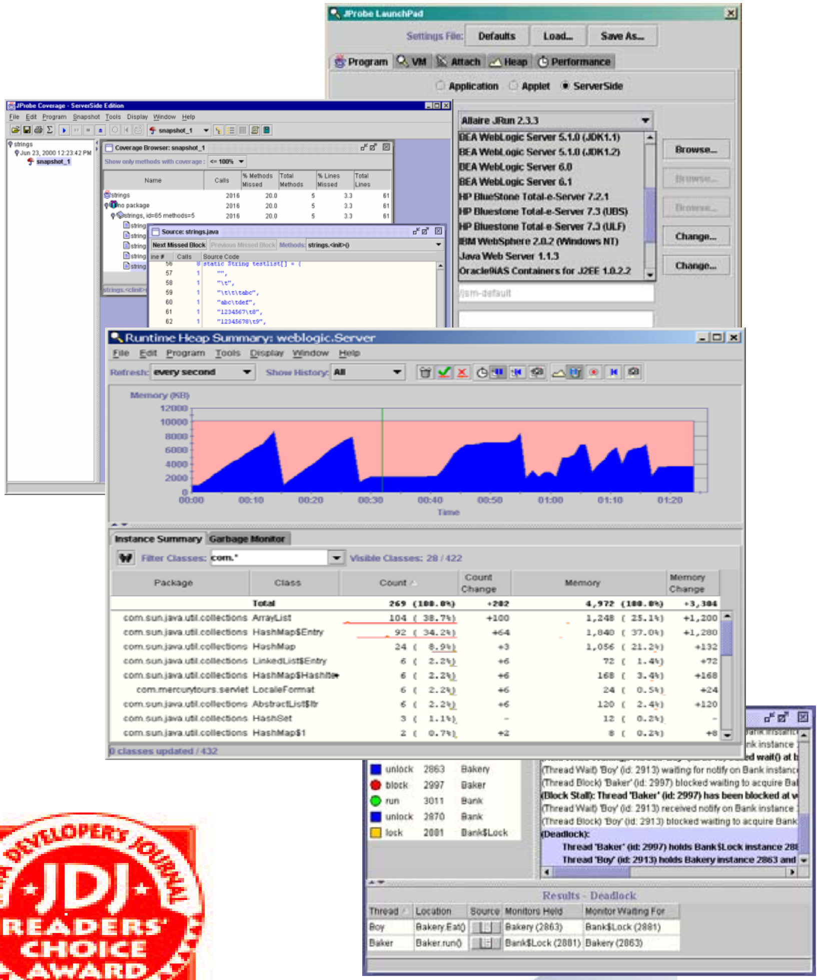
Management Reporting

Inefficient, Problematic, or Untested Code Detected:
Return to Development

JPr

Performance tuning tools

- **JProbe** Memory Debugger
- **JProbe** Profiler
- **JProbe** Coverage



JProbe LaunchPad

Settings File: Defaults | Load... | Save As...

Program | VM | Attach | Heap | Performance

○ Application ○ Applet ● ServerSide

Allaire JRun 2.3.3
BEA WebLogic Server 5.1.0 (JDK1.1)
BEA WebLogic Server 5.1.0 (JDK1.2)
BEA WebLogic Server 6.0
BEA WebLogic Server 6.1
HP BlueStone Total-e-Server 7.2.1
HP Bluestone Total-e-Server 7.3 (UBS)
HP Bluestone Total-e-Server 7.3 (ULF)
IBM WebSphere 2.0.2 (Windows NT)
Java Web Server 1.1.3
Oracle9iAS Containers for J2EE 1.0.2.2

JProbe Coverage - ServerSide Edition

Coverage Browser: snapshot_1

Show only methods with coverage: <= 100%

Name	Calls	% Methods Missed	Total Methods	% Lines Missed	Total Lines
strings	2016	20.0	5	3.3	61
no package	2016	20.0	5	3.3	61
strings, id=65 methods=5	2016	20.0	5	3.3	61

Source: strings.java

Next Missed Block | Previous Missed Block | Methods: strings.<init>()

```
56   8  static String testlist[] = {
57   1      "",
58   1      "\t",
59   1      "\t\t\tabc",
60   1      "abc\tdef",
61   1      "1234567\t8",
62   1      "12345678\t9",
```

Runtime Heap Summary: weblogic.Server

Refresh: every second   Show History: All

Instance Summary | Garbage Monitor

Filter Classes: com.*   Visible Classes: 28 / 422

Package	Class	Count	Count Change	Memory	Memory Change
	Total	269 (100.0%)	+202	4,972 (100.0%)	+3,304
com.sun.java.util.collections	ArrayList	104 (38.7%)	+100	1,248 (25.1%)	+1,200
com.sun.java.util.collections	HashMap$Entry	92 (34.2%)	+64	1,840 (37.0%)	+1,280
com.sun.java.util.collections	HashMap	24 (8.9%)	+3	1,056 (21.2%)	+132
com.sun.java.util.collections	LinkedList$Entry	6 (2.2%)	+6	72 (1.4%)	+72
com.sun.java.util.collections	HashMap$HashIte	6 (2.2%)	+6	168 (3.4%)	+168
com.mercurytours.servlet	LocaleFormat	6 (2.2%)	+6	24 (0.5%)	+24
com.sun.java.util.collections	AbstractList$Itr	6 (2.2%)	+6	120 (2.4%)	+120
com.sun.java.util.collections	HashSet	3 (1.1%)	-	12 (0.2%)	-
com.sun.java.util.collections	HashMap$1	2 (0.7%)	+2	8 (0.2%)	+8

0 classes updated / 432

unlock	2863	Bakery
block	2997	Baker
run	3011	Bank
unlock	2870	Bank
lock	2881	Bank$Lock

```
(Thread Wait) 'Boy' (id: 2913) waiting for notify on Bank instanc
(Thread Block) 'Baker' (id: 2997) blocked waiting to acquire Ba
(Block Stall): Thread 'Baker' (id: 2997) has been blocked at w
(Thread Wait) 'Boy' (id: 2913) received notify on Bank instance
(Thread Block) 'Boy' (id: 2913) blocked waiting to acquire Bank
(Deadlock):
    Thread 'Baker' (id: 2997) holds Bank$Lock instance 28
    Thread 'Boy' (id: 2913) holds Bakery instance 2863 and
```

Results - Deadlock

Thread	Location	Source	Monitors Held	Monitor Waiting For
Boy	Bakery.Eat()		Bakery (2863)	Bank$Lock (2881)
Baker	Baker.run()		Bank$Lock (2881)	Bakery (2863)

EDITORS' CHOICE
JavaWorld
FINALIST 2003

software development 2002 product excellence award JOLT

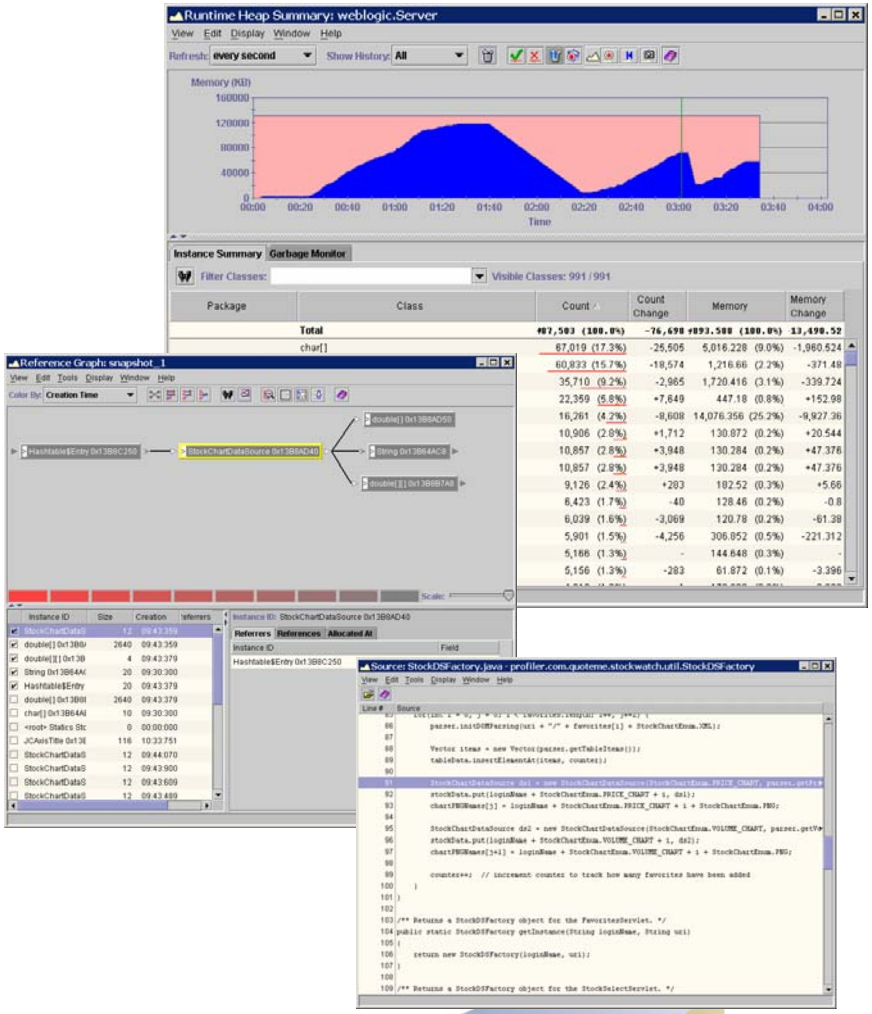
software development 13th annual productivity award

JAVA DEVELOPER'S JOURNAL JDJ READERS' CHOICE AWARD

QUEST SOFTWARE
Simplicity At Work™

JProbe Memory Debugger

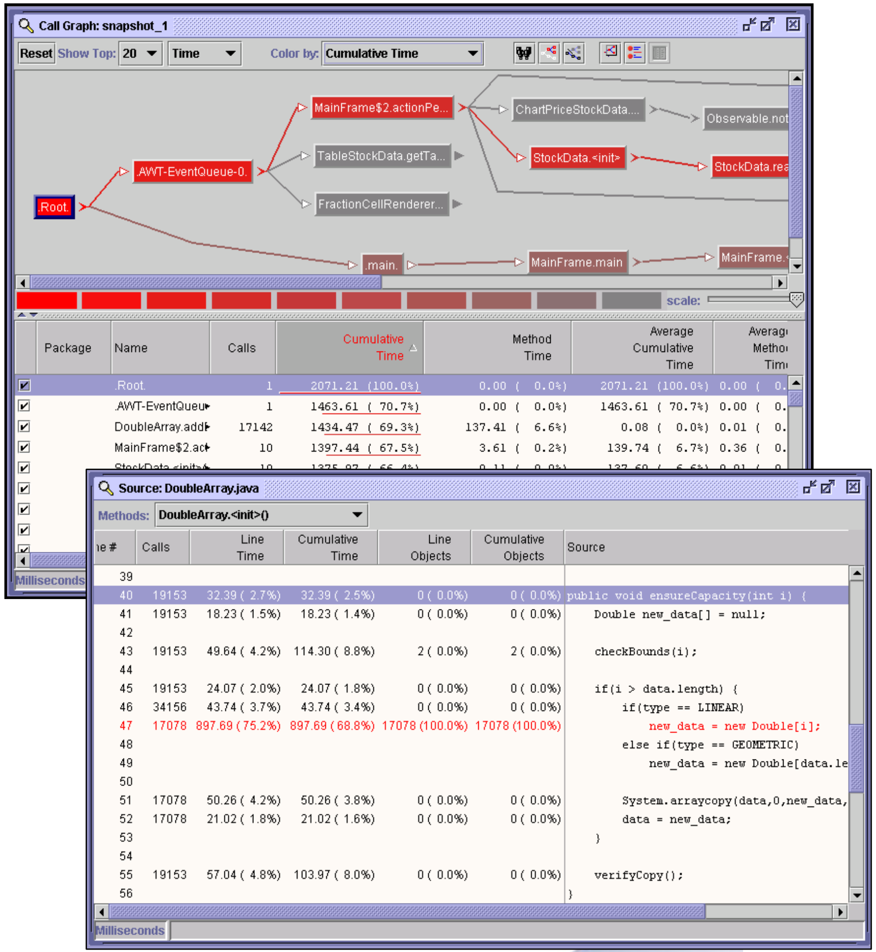
- Pinpoint underlying causes of memory leaks
- Dramatically reduce memory consumption
- Quickly identify methods that create excessive numbers of short-lived objects



Package	Class	Count	Count Change	Memory	Memory Change
	Total	387,503 (100.0%)	-76,698	[illegible] (100.0%)	13,490.52
	char[]	67,019 (17.3%)	-25,505	5,016.228 (9.0%)	-1,960.524
		60,833 (15.7%)	-18,574	1,216.66 (2.2%)	-371.48
		35,710 (9.2%)	-2,965	1,720.416 (3.1%)	-339.724
		22,359 (5.8%)	+7,649	447.18 (0.8%)	+152.98
		16,261 (4.2%)	-8,608	14,076.356 (25.2%)	-9,927.36
		10,906 (2.8%)	+1,712	130.872 (0.2%)	+20.544
		10,857 (2.8%)	+3,948	130.284 (0.2%)	+47.376
		10,857 (2.8%)	+3,948	130.284 (0.2%)	+47.376
		9,126 (2.4%)	+283	182.52 (0.3%)	+5.66
		6,423 (1.7%)	-40	128.46 (0.2%)	-0.8
		6,039 (1.6%)	-3,069	120.78 (0.2%)	-61.38
		5,901 (1.5%)	-4,256	306.052 (0.5%)	-221.312
		5,166 (1.3%)	-	144.648 (0.3%)	-
		5,156 (1.3%)	-283	61.872 (0.1%)	-3.396

JProbe Profiler

- Uncover performance bottlenecks
- Gather line-level metrics on your running program
- Deadlock Detection



Show Top: 20 Time

Color by: Cumulative Time

Package	Name	Calls	Cumulative Time	Method Time	Average Cumulative Time	Averag Metho Tim
☑	.Root.	1	2071.21 (100.0%)	0.00 (0.0%)	2071.21 (100.0%)	0.00 (0.
☑	.AWT-EventQueu	1	1463.61 (70.7%)	0.00 (0.0%)	1463.61 (70.7%)	0.00 (0.
☑	DoubleArray.add	17142	1434.47 (69.3%)	137.41 (6.6%)	0.08 (0.0%)	0.01 (0.
☑	MainFrame$2.ac	10	1397.44 (67.5%)	3.61 (0.2%)	139.74 (6.7%)	0.36 (0.

Milliseconds

Source: DoubleArray.java

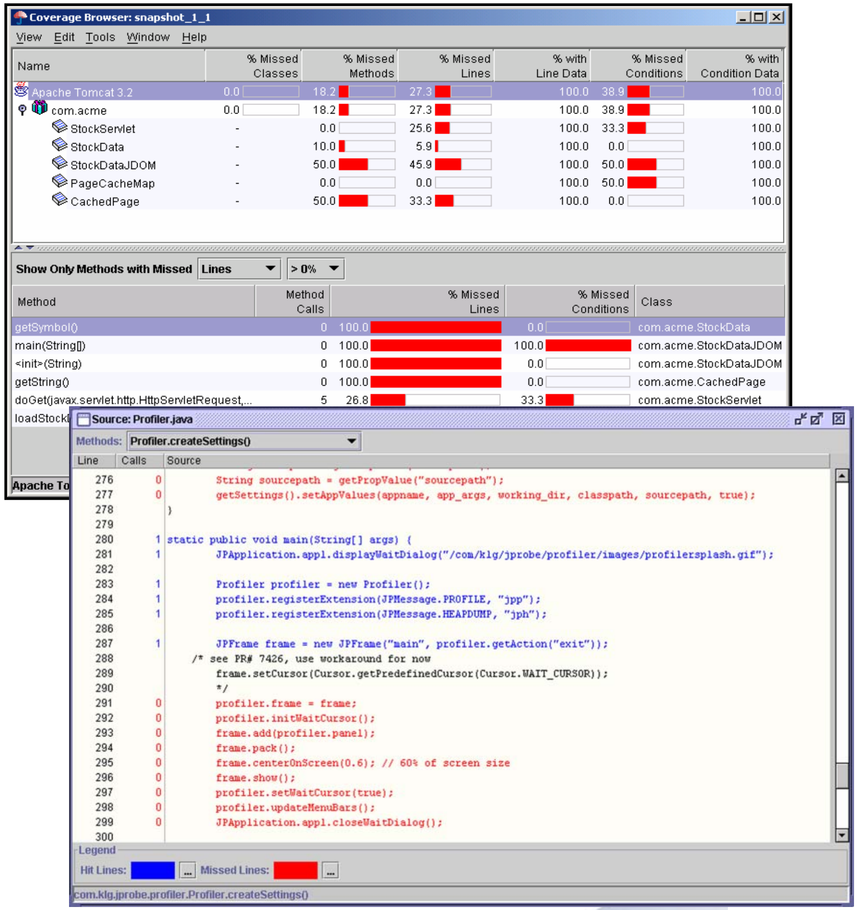
Methods: DoubleArray.<init>()

```
ne #  Calls      Line       Cumulative        Line     Cumulative   Source
                 Time             Time      Objects       Objects
 39
 40   19153   32.39 ( 2.7%)   32.39 ( 2.5%)        0 ( 0.0%)        0 ( 0.0%) public void ensureCapacity(int i) {
 41   19153   18.23 ( 1.5%)   18.23 ( 1.4%)        0 ( 0.0%)        0 ( 0.0%)     Double new_data[] = null;
 42
 43   19153   49.64 ( 4.2%)  114.30 ( 8.8%)        2 ( 0.0%)        2 ( 0.0%)     checkBounds(i);
 44
 45   19153   24.07 ( 2.0%)   24.07 ( 1.8%)        0 ( 0.0%)        0 ( 0.0%)     if(i > data.length) {
 46   34156   43.74 ( 3.7%)   43.74 ( 3.4%)        0 ( 0.0%)        0 ( 0.0%)         if(type == LINEAR)
 47   17078  897.69 (75.2%)  897.69 (68.8%)  17078 (100.0%)  17078 (100.0%)             new_data = new Double[i];
 48                                                                                  else if(type == GEOMETRIC)
 49                                                                                      new_data = new Double[data.le
 50
 51   17078   50.26 ( 4.2%)   50.26 ( 3.8%)        0 ( 0.0%)        0 ( 0.0%)         System.arraycopy(data,0,new_data,
 52   17078   21.02 ( 1.8%)   21.02 ( 1.6%)        0 ( 0.0%)        0 ( 0.0%)         data = new_data;
 53                                                                              }
 54
 55   19153   57.04 ( 4.8%)  103.97 ( 8.0%)        0 ( 0.0%)        0 ( 0.0%)     verifyCopy();
 56                                                                          }
```

Milliseconds

JProbe Coverage

- Identify and quantify untested code
- Merge line-level data from different test runs
- Generate reports in HTML, XML, Text or PDF
- Conditional Code Analysis



Name	% Missed Classes	% Missed Methods	% Missed Lines	% with Line Data	% Missed Conditions	% with Condition Data
Apache Tomcat 3.2	0.0	18.2	27.3	100.0	38.9	100.0
com.acme	0.0	18.2	27.3	100.0	38.9	100.0
StockServlet	-	0.0	25.6	100.0	33.3	100.0
StockData	-	10.0	5.9	100.0	0.0	100.0
StockDataJDOM	-	50.0	45.9	100.0	50.0	100.0
PageCacheMap	-	0.0	0.0	100.0	50.0	100.0
CachedPage	-	50.0	33.3	100.0	0.0	100.0

Show Only Methods with Missed: Lines, > 0%

Method	Method Calls	% Missed Lines	% Missed Conditions	Class
getSymbol()	0	100.0	0.0	com.acme.StockData
main(String[])	0	100.0	100.0	com.acme.StockDataJDOM
<init>(String)	0	100.0	0.0	com.acme.StockDataJDOM
getString()	0	100.0	0.0	com.acme.CachedPage
doGet(javax.servlet.http.HttpServletRequest,...	5	26.8	33.3	com.acme.StockServlet
loadStock				

Source: Profiler.java

Methods: Profiler.createSettings()

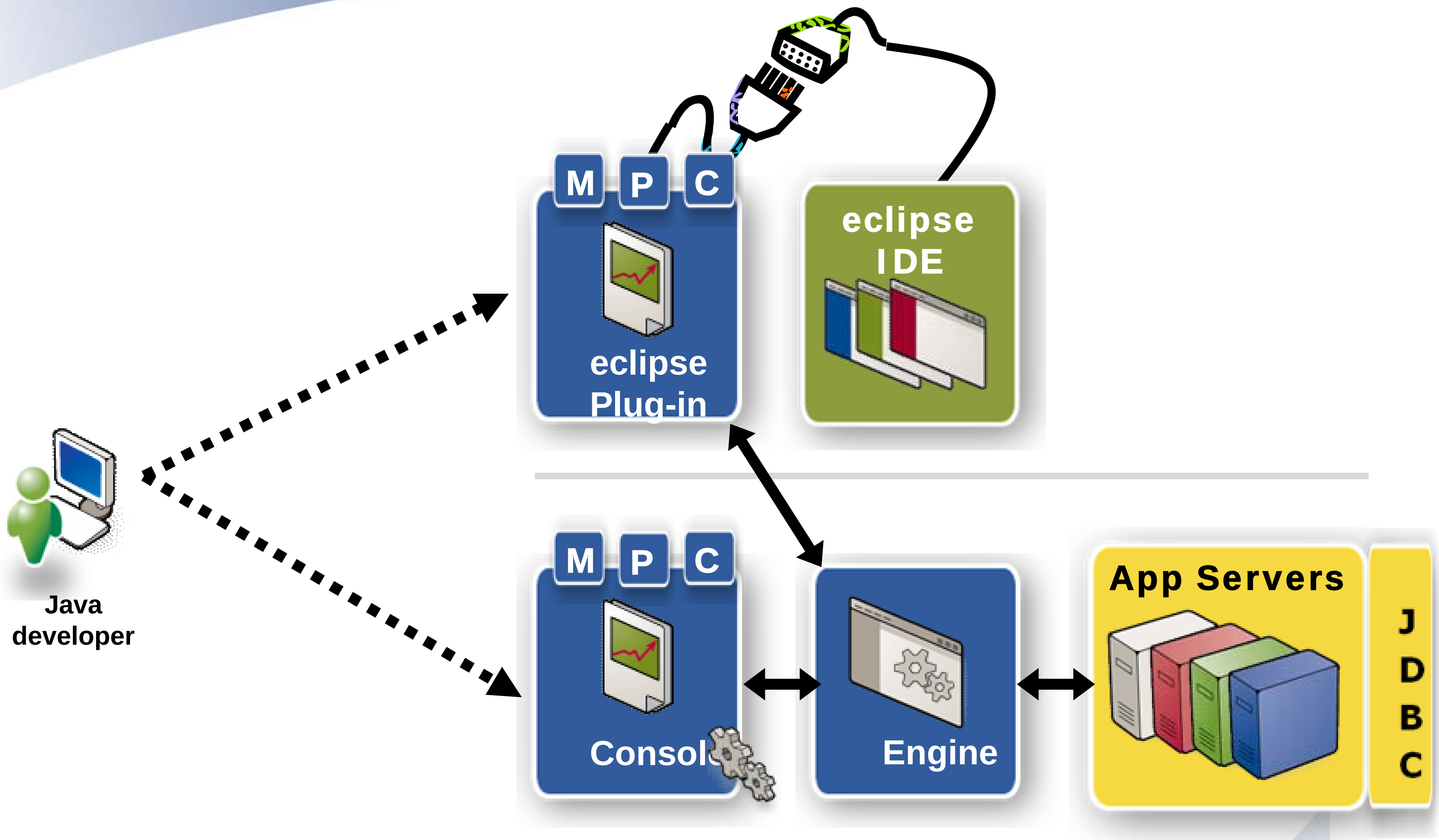
```
Calls  Source
  0          String sourcepath = getPropValue("sourcepath");
  0          getSettings().setAppValues(appname, app_args, working_dir, classpath, sourcepath, true);
       }

  1    static public void main(String[] args) {
  1          JPApplication.appl.displayWaitDialog("/com/klg/jprobe/profiler/images/profilersplash.gif");

  1          Profiler profiler = new Profiler();
  1          profiler.registerExtension(JPMessage.PROFILE, "jpp");
  1          profiler.registerExtension(JPMessage.HEAPDUMP, "jph");

  1          JFrame frame = new JPFrame("main", profiler.getAction("exit"));
          /* see PR# 7426, use workaround for now
             frame.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
             */
  0          profiler.frame = frame;
  0          profiler.initWaitCursor();
  0          frame.add(profiler.panel);
  0          frame.pack();
  0          frame.centerOnScreen(0.6); // 60% of screen size
  0          frame.show();
  0          profiler.setWaitCursor(true);
  0          profiler.updateMenuBars();
  0          JPApplication.appl.closeWaitDialog();

```

Legend

Hit Lines: 🟦 Missed Lines: 🟥

com.klg.jprobe.profiler.Profiler.createSettings()

JProbe Architecture



Garbage Monitor

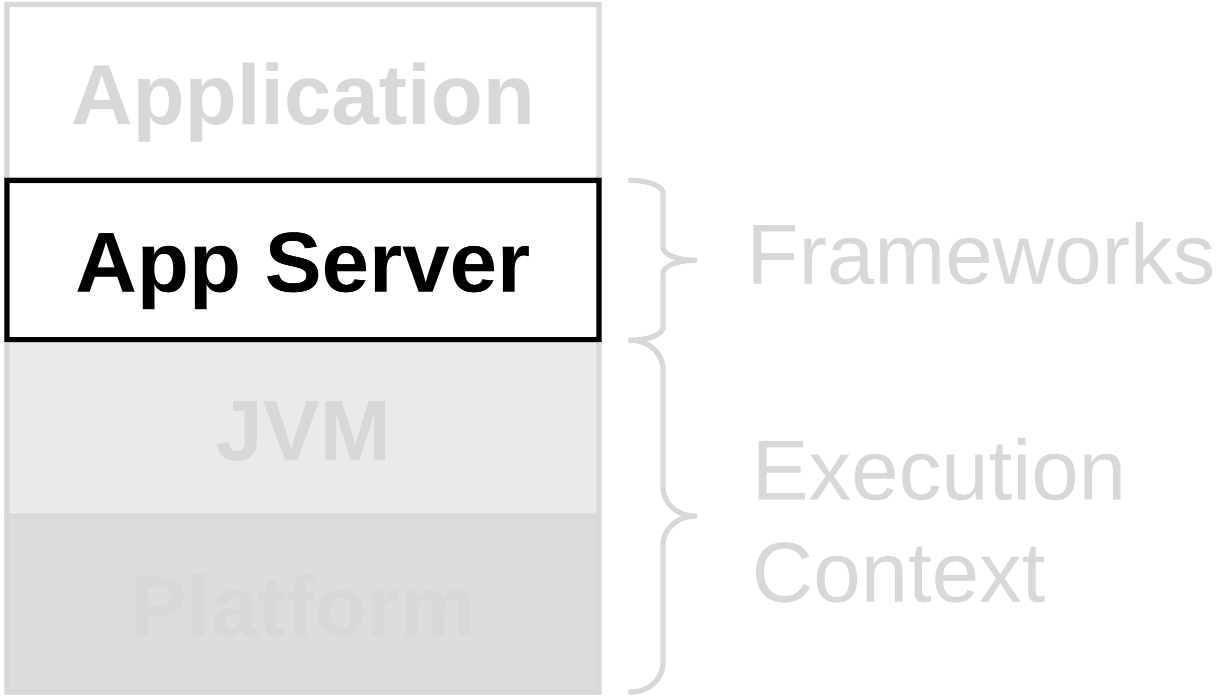
- Identifies the type and number of objects reclaimed after each garbage collection ***and their allocation points within your code***
- Those methods are the candidates for refactoring to lessen the excessive creation of short-lived objects

Achieving Faster Performance

- Most frequent criticism of Java
- Nature of the Java Execution Model
  - Platform Independence
  - Runtime Checks (array bounds, ref casts)
  - Garbage Collection
- Given that context, what can be done to increase performance?

Areas to Examine

- Examine performance at 3 or 4 levels



Platform: Physical Memory

- By far the most important resource
  - Address the needs of your running application *plus* that of the underlying JVM
- Examine your JVM’s paging characteristics
  - On a memory-constrained system with virtual memory (where much of the process resides in the paging file/partition), the reachability test of garbage collection can cause excessive paging activity

Platform: Physical Memory

- While your application is running, observe the page fault activity of the JVM running your application
  - Especially during GC events which you can monitor via **`–verbose:gc`**
- Excessive paging activity indicates that the OS doesn't have enough physical memory available to run your JVM efficiently
- Solution
  - Reduce # of applications running concurrently
  - Increase physical memory

Platform: CPU

- The faster, the better
  - Your overall CPU utilization rate should be below 75%
- Multiprocessor (MP)
  - To take advantage of a multiprocessor environment
    - the JVM must support native threads
      - all modern Java 2 VMs use native threads
    - your Java application must be multithreaded

Java Virtual Machines

- JVM performance has been steadily increasing
  - Bytecode execution strategies
  - Memory management & garbage collection
  - Native thread support and sync overhead
- Competition among JVM vendors
  - The specification permits a great deal of freedom for implementation choices

Application Server Tuning

- There are a number of things you can do to improve the performance of your application server:
  - install native performance packs (if available)
  - modify shared resource settings
    - JDBC Connection Pools
    - EJB Pool Size
    - Thread Pool Size

Framework-based Development

- Java development is undertaken in the context of application frameworks:
  - Persistence Frameworks
    - Hibernate
    - JPA
    - IBatis SQLMaps
  - Presentation Layer Frameworks
    - JSF
    - AJAX – Dojo, HTML, JavaScript
    - JSP
  - Middle Tier Frameworks
    - EJB 3.0
    - The Spring Framework
- Advantages
  - Productivity
  - Reduced Development time
  - Software Correctness
- Pit-falls
  - Incomplete knowledge on use of framework
  - Incorrect Assumptions about framework
  - Unforeseen consequences of framework use

Localized Performance Optimizations

- ***Pareto Principle***
  - You get 80 percent of the result from 20 percent of the effort
- The question is: Which 20 percent?
  - Programmers are notoriously bad at *subjectively* identifying performance bottlenecks in their application
  - Don’t waste time optimizing code that is rarely used
- You need objective information to identify the critical performance path within your application
  - Refining the methods along this path will give you the most benefit for the time you invest

Performance Investigation Features in JProbe

- Performance snapshots
  - Captures the performance of your application between two points in time

Integrating Performance Analysis into Your Development Cycle

- Before undertaking performance analysis
  - Use JProbe Memory Debugger’s heap analysis to ensure that you’ve resolved any loitering object problems
- Undertake your performance analysis under realistic conditions

Integrating Performance Analysis into Your Development Cycle

Make it Work Right

then

Make it Work Fast

The Best Time to Tune

- When is the best time to tune code?
  - Development
    - Good for small modules, may be too soon for system-wide performance analysis
  - Integration
    - But now it may be difficult to get down deep to fix problems
  - QA
    - QA people often don’t have the application knowledge to tune performance effectively
  - Pre-production/Staging
    - Usually where final capacity plan is determined - often too late to tune
  - Production
    - Technically possible, but not recommended

Threads

- A thread is an independent sequence of program code execution within a running process
- *Single threaded* model of computation
  - Single thread of execution within a single process
  - The model most programmers are familiar with
- *Multithreaded* model of computation
  - Several threads of execution within a single process
  - Many benefits, but also new pitfalls

Java Threads

The notion of threading is so ingrained in Java that it's almost impossible to write all but the simplest programs without creating and using threads. And many of the classes within the Java API are already threaded, so that you often are using multiple threads without realizing it.

Scott Oaks and Henry Wong
Java Threads (2nd Edition)
O'Reilly and Associates, 1999

Issues in Multithreaded Design

- Creating and starting threads within an application is not a problem
- It’s in the *coordination* and *synchronization* of their work that problems typically arise
  - **Race Conditions**
  - **Deadlocks**

Race Conditions

- A data race occurs when two concurrent threads access a shared variable, and
  - At least one access is a write operation, and
  - The threads use no explicit mechanism to prevent their accesses from being simultaneous
- Two forms:
  1. Read/Write race conditions
  2. Write/Write race conditions
- Race conditions arise from a failure to coordinate, or *synchronize,* thread access to a shared variable

Synchronizing Shared Resources

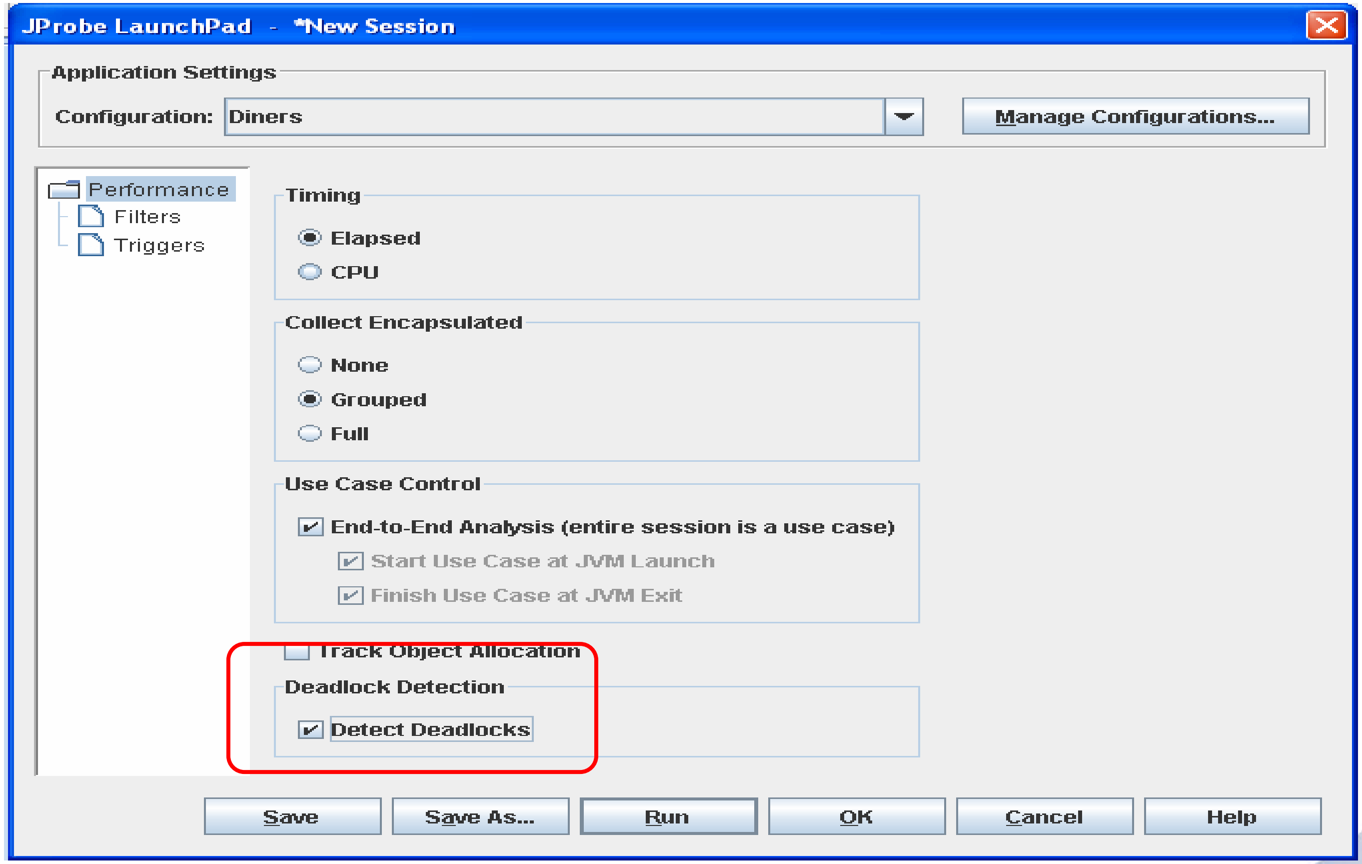
- Historically, multithreaded applications have used a variety of objects, such as *semaphores* and *monitors,* to lock the critical sections of code that access shared variables
- Within Java
  - Each object within the heap has a lock associated with it
  - The **`synchronized`** keyword is used to ensure that critical sections of code (that access shared variables) are executed by *only* one thread at any given time

Deadlocks

- A thread suffers from deadlock if it blocks waiting for a condition that will never occur
- Typically arises from the overuse of synchronization
  - There is a constant tension in multithreaded programs between *safety* and *liveness*
- In the classic deadlock case, a thread requires access to a resource that is already locked by a second thread, and that thread is trying to access a resource that has already been locked by the first

Thread Investigation Part of JProbe Performance

- Detects thread deadlocks within your application



Application Settings

Configuration: Diners

Timing

● Elapsed
○ CPU

Collect Encapsulated

○ None
● Grouped
○ Full

Use Case Control

☑ End-to-End Analysis (entire session is a use case)
  ☑ Start Use Case at JVM Launch
  ☑ Finish Use Case at JVM Exit

☐ Track Object Allocation

Deadlock Detection

☑ Detect Deadlocks