

High Performance Clients 1

Minimizing Startup Time

java.sun.com/javaone/sf

Radim Kubacki
Petr Nejedly
Kenneth Russell
Sun Microsystems, Inc.



Presentation Goal

Provide techniques for reducing the startup time of Java™ technology-based applications

Startup time of Java applications

- Fast startup of Java technology-based applications is one of the critical quality features for desktop Java technology application customers
- Goal of this presentation is to give developers techniques for reducing startup time of both small and large applications
- Distilling experience from startup time optimizations done in the J2SE™ 1.4.2 and 1.5 platforms and NetBeans™ 3.3 software and above

Startup time of Java applications

- Giving concrete examples of optimizations, both to show techniques and illustrate new features of the Java platform
- Providing general techniques that can be applied to your applications

Agenda

Profiling

Lazier Initialization

Caching and Memoization

Large-Scale Changes

Performance Results

Conclusion

Agenda

Profiling

Lazier Initialization

Caching and Memoization

Large-Scale Changes

Performance Results

Conclusion

Profiling

- “Premature optimization is the root of all evil”
 - C.A.R. Hoare
- Find out where the time is going before optimizing
- Many commercial and free Java profilers are available
- Spend time thinking about profiling data
 - Often not obvious why given areas of code are expensive
 - Sometimes due to poor algorithms at higher levels
 - Often hardest part of performance work

Profiling

- Add logging to your code to be able to keep track of large units of initialization work
 - Can help see the forest for the trees
 - Profilers tend to show too much detail, making it hard to see where time is going sometimes
 - Example of making logging calls less expensive by avoiding evaluation of logging messages:
<http://www.netbeans.org/download/dev/javadoc/OpenAPIs/org/openide/doc-files/upgrade.html#api-errman>

Profiling

- If necessary, write new tools to focus in on problems
 - Custom JVM™ machine PI-based tool written for analyzing J2SE platform startup in 1.4.2
- Measure, optimize, measure again
- Optimization is a permanent, ongoing effort needing clear, well-defined goals

Agenda

Profiling

Lazier Initialization

- Split up and defer work
- Eliminate unnecessary work
- Break dependencies
- Eliminate statics

Caching and Memoization

Large-Scale Changes

Performance Results

Conclusion

Lazier Initialization

Overview

- Dynamic nature of the Java platform allows incremental loading and initialization of applications
 - Dynamic class loading
 - Lazy linking
- However, programming models carried over from other languages tend to do too much initialization work up front

Lazier Initialization: Split up and defer work

Case Study: Logging in the J2SE1.4.2 platform

- Initialization of logger inserted into `System.initializeSystemClass()` in 1.4
 - Expensive; parses large property file
- Logging initialization made lazier
 - Removed from `java.lang.System`
 - Done on first call to `LogManager.getLogManager()`
 - Decreased Hello, World startup by 6.9%
 - `WindowsPreferences` pulling in logging eagerly for GUI apps
 - Made lazier
 - Decreased Notepad startup by ~1%

Lazier Initialization: Split up and defer work

Case Study: Character in J2SE 1.4.2 platform

- `java.lang.Character` initialization made lazier
 - Largest component of `System.initializeSystemClass`
 - Sets up large attribute tables for identifiers
 - `isDigit()`, `isJavaIdentifierStart()`
 - Sets up large case-mapping tables for Unicode
 - `toUpperCase()`, `toLowerCase()`
 - Most accessors rewritten to have fast paths for Latin-1 characters
 - Tables only initialized upon receiving input ≥ 256
 - Handles JDK™ software bootstrapping at least for Latin-1 locales
 - Decreased Hello, World startup by 18.9%

Lazier Initialization: Split up and defer work

Case Study: DataLoaders in the NetBeans 3.3 IDE

- Used by the NetBeans IDE to install various functionality
- Deferred some initialization by moving it to first use rather than overall initialization time

Lazier Initialization: Split up and defer work

Case Study: DataLoaders in the NetBeans 3.3 IDE

```
// Called once the first time a given DataLoader's
// class is used
protected void initialize() {
    setExtensions(...);
    setDisplayName(NbBundle.getMessage
        (TXTDataLoader.class,
         "PROP_TXTLoader_Name"));
    setActions(new SystemAction[] {
        SystemAction.get(OpenAction.class),
        SystemAction.get(FileSystemAction.class),
        ...});
}
```

Lazier Initialization: Split up and defer work

Case Study: DataLoaders in the NetBeans 3.3 IDE

// Changed to:

```
protected void initialize() {  
    setExtensions(...);  
}
```

```
protected String defaultDisplayName() {  
    return NbBundle.getMessage  
        (TXTDataLoader.class,  
         "PROP_TXTLoader_Name");  
}
```

```
protected SystemAction[] defaultActions() {  
    return new SystemAction[] {  
        SystemAction.get(OpenAction.class),  
        SystemAction.get(FileSystemAction.class),  
        ...};  
}
```

Lazier Initialization: Eliminate unnecessary work

Case Study: Manifest parsing in the J2SE 1.4.2 platform

- JarFile.getManifest() very expensive
- Large component of many initialization phases
 - AWT's Toolkit.<clinit>
- Why are manifests read?
- Do not contain index of files; that's in the zip file format

Lazier Initialization: Eliminate unnecessary work

Case Study: Manifest parsing in the J2SE 1.4.2 platform

- Extension class loader reads manifest to determine class-path attribute
 - Finding dependent jars of this one
 - None of the jars in jre/lib/ext have class-path attributes
- JarVerifier uses manifest to verify signed jar files
 - JarFiles verified by default
 - JarVerifier always created
 - None of core jars read during startup are signed

Lazier Initialization: Eliminate unnecessary work

Case Study: Manifest parsing in the J2SE 1.4.2 platform

- `Fast JarFile.hasClassPathAttribute()` implemented
 - Hand-coded Boyer-Moore string search
 - Called by `URLClassPath`
 - Only parse manifest if absolutely necessary
 - Additional optimization for core extension jars known to not have class-path attribute
- Fast scan of `META-INF` directory looking for signature files
 - If none, assume jar is unsigned
 - No `JarVerifier` needed in this case

Lazier Initialization: Break dependencies with reflection

Case Study: DataLoaders in the NetBeans 3.3 IDE

- DataLoaders can be used to provide a DataObject to the IDE
 - Need to identify class of object
 - Refer to class by name instead of class constant

```
public PDFDataLoader() {  
    super(PDFDataObject.class);  
}
```

// Changed to:

```
public PDFDataLoader() {  
    super(  
        "org.netbeans.modules.pdf.PDFDataObject");  
}
```

Lazier Initialization: Eliminate statics

Case Study: SystemOption in the NetBeans 3.3 IDE

- Used by NetBeans software modules to store persistent state
- Mistake was to place static reference to SystemOption in module class

```
static MyOptions OPTIONS =  
    (MyOptions) SystemOption.findObject  
        (MyOptions.class, true);
```

- findObject() is expensive because it parses XML files, among other things

Lazier Initialization: Eliminate statics

Case Study: SystemOption in the NetBeans 3.3 IDE

- Should be done with lazy getter instead

```
private static MyOptions OPTIONS;  
static synchronized MyOptions getMyOptions() {  
    if (OPTIONS == null) {  
        OPTIONS = (MyOptions)  
            SystemOption.findObject  
                (MyOptions.class, true);  
    }  
    return OPTIONS;  
}
```

Lazier Initialization

- Performing initialization more lazily may simply push work later into the startup cycle
- Try to find ways to eliminate the work completely rather than defer it
 - Pay for what you use

Agenda

Profiling

Lazier Initialization

Caching and Memoization

- Cache expensive computations

- Avoid redundant caching

Large-Scale Changes

Performance Results

Conclusion

Caching and Memoization: Cache Expensive Computations

Case Study: File name canonicalization in 1.4.2

- Significant amount of time spent in file name canonicalization
 - Resolves file names through symlinks (UNIX[®]) and case-insensitive specification (Windows)
 - Used by security subsystem to check access to file system
 - Applications use for other purposes, including permissions checks in web servers
 - Implemented time-expiring cache for canonicalization results
 - Prefix cache used where possible
 - Saves time canonicalizing multiple file names in same directory
 - Decreased Notepad startup by ~5%

Caching and Memoization: Cache Expensive Computations

Case Study: Binary layers in the NetBeans 3.5 IDE

- Module extension: XML fragment (layer)
- All modules: over 300 KB of XML
- Final database: tree structure
- Need to be ready for queries during startup
- But only part of data really used
- Parse once, store snapshot, verify integrity
- Snapshot structured for tree traversal without fetching it all at once

Caching and Memoization: Cache Expensive Computations

Case Study: Binary layers in the NetBeans 3.5 IDE

Benefits:

- Fewer files opened
 - Only one file instead of one per module
- No parsing on startup
 - Just reads the header; the rest is read on demand
- Lower memory usage
 - No unused data in memory; more compact
- Doesn't touch XML parser, but it will be needed elsewhere anyway

Caching and Memoization: Cache Expensive Computations

Case Study: Class Data Sharing in the J2SE 1.5 platform

- Primary startup time and footprint optimization done in Java HotSpot™ virtual machine in 1.5
- Enables sharing of loaded class data for core classes among multiple running JVM software
- During JRE installation, the Java HotSpot VM preloads a set of core classes into memory
- Resulting JVM software data is written to a file on disk called a **shared archive**

Caching and Memoization: Cache Expensive Computations

Case Study: Class Data Sharing in the J2SE 1.5 platform

- On subsequent launches, shared archive is memory-mapped
 - Roughly half read-only, half copy-on-write
 - Saves parse time for these core classes
 - Allows data to be shared between running JVM software
- Provides largest startup time reductions for smaller applications
 - Where loading of core classes dominates startup time
- Also provides good footprint reduction
 - Currently, up to 5-6 megabytes of loaded class data shared between JVM software instances

Caching and Memoization

Use Buffered Streams and Readers

- Seemingly obvious, but common source of performance problems
- System-specific tools can help track down use of unbuffered streams
 - truss
 - strace
 - dtrace (Solaris 10)

Caching and Memoization: Avoid redundant caching

Case Study: BeanInfo in the NetBeans 3.3 IDE

- Provide metadata to IDE about various subcomponents
 - Icons, configuration
- Data were being stored in static fields:

```
public class MyBeanInfo extends SimpleBeanInfo {
    private static PropertyDescriptor[] desc;
    static {
        desc = new PropertyDescriptor[] {
            new PropertyDescriptor("field",
                My.class, null, "setField"),
            ...
        };
    }
    ...
}
```

Caching and Memoization: Avoid redundant caching

Case Study: BeanInfo in the NetBeans 3.3 IDE

- Standard JavaBeans architecture Introspector performs caching of these values
- Designed to be able to unload cached values when all instances of BeanInfos are unloaded
- Therefore, previous initialization is not only too eager, but also causes a memory leak
- Need to understand semantics when integrating into a framework

Caching and Memoization: Avoid redundant caching

Case Study: BeanInfo in the NetBeans 3.3 IDE

- New, correct code:

```
public class MyBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[]
        getPropertyDescriptors () {
        return new PropertyDescriptor[] {
            new PropertyDescriptor("field",
                My.class, null, "setField"),
            ...
        };
    }
}
```

Caching and Memoization: Avoid redundant caching

Case Study: Resource Bundles in the NetBeans 3.3 IDE

- Another example of too much caching via use of statics
- NetBeans software manages a central bundle cache
- No need for users to cache themselves

Caching and Memoization: Avoid redundant caching

Case Study: Resource Bundles in the NetBeans 3.3 IDE

- Undesirable:

```
private static ResourceBundle bundle;  
private synchronized static String getText  
    (String key) {  
    if (bundle == null) {  
        bundle = NbBundle.getBundle(ThisClass.class);  
    }  
    return bundle.getString(key);  
}
```

```
public String getSomeText(String something) {  
    return MessageFormat.format(getText  
        ("FMT_foo"), new Object[] {something});  
}
```

Caching and Memoization: Avoid redundant caching

Case Study: Resource Bundles in the NetBeans 3.3 IDE

- Better and easier:

```
public String getSomeText(String something) {  
    return NbBundle.getMessage(ThisClass.class,  
        "FMT_foo", something);  
}
```

Caching and Memoization:

Avoid redundant caching

Case Study: Image Caching in the NetBeans IDE

- Problems with modules loading icons and other images in static initializers
 - Too early
 - Images never garbage collected
- Caching mechanisms exist both in the JRE and NetBeans software
 - NetBeans IDE's cache converts to BufferedImage which saves space compared to AWT-provided image

Caching and Memoization: Avoid redundant caching

Case Study: Image Caching in the NetBeans IDE

- This is bad:

```
static Image myicon = ...;
public Image method() {
    return myicon;
}
```

- This is better but never garbage collects the image:

```
static Image myicon = null;
public Image method() {
    if (myicon == null) {
        myicon = ...;
    }
    return myicon;
}
```

Caching and Memoization:

Avoid redundant caching

Case Study: Image Caching in the NetBeans IDE

- Better yet:

```
public Image method() {  
    return Toolkit.getDefault().getImage(...);  
    // similar for SimpleBeanInfo.loadImage(...)  
}
```

- Best:

```
public Image method() {  
    return Utilities.loadImage  
        ("my/pkg/resources/something.gif");  
}
```

Caching and Memoization

- Take-home message: refactor
- Cache if necessary, but centralize the caches
- Don't spread them throughout the code
 - Will get out of sync (hard to reason about)
 - Consume too much space

Agenda

Profiling

Lazier Initialization

Caching and Memoization

Large-Scale Changes

- UI Tuning

- Declarative specification

- Algorithmic changes

Performance Results

Conclusion

Large-Scale Changes: UI Tuning

- Sometimes user interface changes can smooth out initialization and other expensive computations
- Splash screens
 - Make total time worse
 - Significantly improve perception
- Menu reorganization
 - Move items that have expensive enable/disable checks deeper into the menu hierarchy
 - Avoid affecting common menu invocations
 - Example: “go to source”, “go to definition” moved into new submenu in NetBeans IDE editor

Large-Scale Changes: UI Tuning

Case Study: Lazier Initialization of Components

- Have discussed deferring initialization in several contexts
- May be tradeoff between reducing startup time and impacting GUI responsiveness
 - If initialization occurs when a given menu option is first selected, pause may occur during interaction
- Case of complicated component with many sub-components:
 - Some items may take a long time to initialize
 - Would like panel to appear and be painted, and perhaps have these items be filled in a second or two later

Large-Scale Changes: UI Tuning

Case Study: Lazier Initialization of Components

- NetBeans framework supports this via set of interfaces and APIs
 - `org.openide.util.AsyncGUIJob`
 - `org.openide.util.Cancellable`
 - `org.openide.Utilities.attachInitJob(Component comp4Init, AsyncGUIJob initJob)`
- Can be useful technique for improving GUI responsiveness

Large-Scale Changes: Declarative Specification

- Consider specifying system configuration declaratively rather than programmatically
 - Via XML configuration file or something similar
 - META-INF/services
- Makes it easier to automate lazy instantiation and deferral of work

Large-Scale Changes: Declarative Specification

Case Study: WindowSystem API in the NetBeans IDE

- Formerly, some NetBeans software modules would manually instantiate top-level and sub-components
 - For example, to be able to specify alignments
- Caused creation of many widgets that were just going to be hidden
- Now specified declaratively using XML

Large-Scale Changes: Declarative Specification

Case Study: WindowSystem API in the NetBeans IDE

```
<mode version="2.0">
  <module name="org.netbeans.core.ui/1" spec="1.2"
/>
  <name unique="explorer" />
  <kind type="view" />
  <state type="joined" />
  <constraints>
    <path orientation="vertical" number="0"
weight="0.7" />
    <path orientation="horizontal" number="0"
weight="0.25" />
  </constraints>
  <active-tc id="filesystems" />
  <empty-behavior permanent="true" />
</mode>
```

Large-Scale Changes: Declarative Specification

Case Study: ModuleInstall in the NetBeans IDE

- Used by NetBeans IDE to install modules upon startup and serialize their state upon shutdown
- Problem was that modules were doing too much work in these phases
 - Unnecessary work during construction of ModuleInstall
 - Static or instance fields being initialized
- Style guide was published indicating how to write startup-friendly ModuleInstalls
 - Best is not to have one at all

Large-Scale Changes: Declarative Specification

Case Study: ModuleInstall in the NetBeans IDE

- Switching to declarative style gave IDE better control over loading and unloading modules
- Examples of things that can be done declaratively:
 - Install actions into the IDE's global menus, toolbars, or keymap
 - Register templates in the Templates/ folder
 - Most generally, to install arbitrary services into the system

Large-Scale Changes: Declarative Specification

Case Study: ModuleInstall in the NetBeans IDE

- Example of installing user's action into the Main
-> Tools menu:

```
<filesystem>
  <folder name="Menu">
    <folder name="Tools">
      <file name="com-me-MyAction.instance"/>
    </folder>
  </folder>
</filesystem>
```

Large-Scale Changes: Algorithmic Changes

- Rethink expensive areas of code
- Choose different algorithms

Large-Scale Changes: Algorithmic Changes

Case Study: JFileChooser Multiple Selection

- Used $O(n^2)$ algorithm to fire selection changed events
 - Two lists: “before” and “after” current selection change
 - Walked down “before” list, iterating entire “after” list for each element
- Now sorts before-and-after selection lists
 - Walks down both lists simultaneously
 - New algorithm is $O(n * \lg n)$
- For one particular test case of 10,000 files, sped up by a factor of 300

Large-Scale Changes: Algorithmic Changes

Case Study: XML Parsing

- Use proper parser for the purpose
- SAX vs. DOM
 - SAX fires events for each lexical element
 - DOM builds an object graph for the whole document
- Custom parsing for critical parts
- Special parsers (XMPP, JADE)

Large-Scale Changes

- Is your application large?
 - If not, simply attach a profiler
- If so, is it modular?
 - If yes, initialize modules on demand
 - If not, first modularize the application
 - Once the application is modular enough, it is easier to attribute startup costs to components

Agenda

Profiling

Lazier Initialization

Caching and Memoization

Large-Scale Changes

Performance Results

Conclusion

Performance Results

NetBeans 3.3 software

- Pentium III, 800 MHz, 512 MB RAM, Linux kernel 2.4.10, JDK 1.3.1_07
- Comparing NetBeans 3.2.1 to 3.3 software
- Warm start decreased from ~24 seconds to 16 seconds
 - Decrease of 33%
- Cold start decreased from 90 seconds to between 44-49 seconds
 - Decrease of 45-51%
 - Impact of disk I/O significant for startup, especially cold start

Performance Results

J2SE platform Warm Start Results – Solaris™ Operating System

	<u>1.4.1_07</u>	<u>1.4.2_04</u>	<u>vs. 1.4.1_07</u>	<u>1.5 b49</u>	<u>vs. 1.4.1_07</u>	<u>vs. 1.4.2_04</u>
				<u>Sharing On</u>		
Noop	0.26	0.22	-15%	0.18	-31%	-18%
Framer	1.53	1.20	-22%	0.95	-38%	-21%
XFramer	1.71	1.34	-22%	1.08	-37%	-19%
JEdit	3.97	3.58	-10%	3.65	-8%	+2%
LimeWire	6.54	6.06	-7%	6.63	+1%	+9%
NetBeans	14.57	14.03	-4%	14.22	-2%	+1%

Alacrity Startup3; Beta 2 Classlist

Solaris/SPARC 1 X 900 Mhz

5037149: 4% regression in "Tiger" b44 for startup3 on solaris client

5037116: 7% Regression in Startup3 for Solaris SPARC®

Performance Results

J2SE Platform Warm Start Results – Windows

	<u>1.4.1_07</u>	<u>1.4.2_04</u>	<u>vs. 1.4.1_07</u>	<u>1.5 b49</u>	<u>vs. 1.4.1_07</u>	<u>vs. 1.4.2_04</u>
				<u>Sharing On</u>		
Noop	0.13	0.14	+8%	0.10	-23%	-29%
Framer	0.94	0.92	-2%	0.73	-22%	-21%
XFramer	0.92	0.85	-8%	0.81	-12%	-5%
JEdit	3.15	3.02	-4%	2.82	-10%	-7%
LimeWire	3.80	3.84	+1%	3.71	-2%	-3%
NetBeans	6.94	7.19	+4%	6.38	-8%	-11%

Alacrity Startup3; Beta 2 Classlist
Windows/x86 1 x 2.8 Ghz

Performance Results

J2SE Platform Warm Start Results - Linux

	<u>1.4.1_07</u>	<u>1.4.2_04</u>	<u>vs. 1.4.1_07</u>	<u>1.5 b49</u> <u>Sharing On</u>	<u>vs. 1.4.1_07</u>	<u>vs. 1.4.2_04</u>
Noop	0.12	0.12	+0%	0.09	-25%	-25%
Framer	0.82	0.62	-24%	0.26	-68%	-58%
XFramer	0.86	0.67	-22%	0.29	-66%	-57%
JEdit	2.58	2.38	-8%	1.88	-27%	-21%
LimeWire	4.12	3.95	-4%	3.57	-13%	-10%
NetBeans	7.77	7.56	-3%	6.25	-20%	-17%

Alacrity Startup3; Beta 2 Classlist
Linux/x86 1 x 2.8 Ghz

Agenda

Profiling

Lazier Initialization

Caching and Memoization

Large-Scale Changes

Performance Results

Conclusion

Conclusion

- Optimizations to applications can yield significant startup time improvements
 - Lazier initialization
 - Caching and memoization
 - UI tuning
 - Declarative specification of system
 - Algorithmic changes
- Profile first, then optimize
- Continuing to make optimizations to JVM software and JDK software to reduce startup time and improve run-time performance

For More Information

- TS-1335 High Performance Clients II
- TS-1216 Choices and Trade-Offs in Garbage Collection in the Java Hotspot™ Virtual Machine
- TS-1218 Java™ Platform Performance

For More Information

- BOF-1600 INSANE (INternal Storage Analysis Now Easy – detecting memory leaks)
- Hands-On Lab 7212: Client Performance
- Books
 - Wilson, Kesselman: Java Platform Performance
 - Shirazi: Java Performance Tuning
- URLs
 - <http://performance.netbeans.org/>
- Tools
 - jvmstat, JFluid

Q&A



High Performance Clients 1

Minimizing Startup Time

java.sun.com/javaone/sf

Radim Kubacki
Petr Nejedly
Kenneth Russell
Sun Microsystems, Inc.

