CS193J: Programming in Java
Winter Quarter 2003

Lecture 13
SAX XML Parsing, HW4, Advanced Java

Manu Kumar

sneaker@stanford.edu

Handouts

- 3 Handouts for today!
  - #29: Advanced Java
  - #30: HW4: XEdit
  - #31: Java Implementation and Performance

Guest Lecture Reminder

- When
  - August 7th in class (4:15 PM in Gates B01)
- Speakers
  - George Grigoryev (J2EE Senior Product Manager, Sun)
  - Pierre Delisle (Senior Staff Engineer, Sun)
- Topics
  - Structure of Java Platforms: J2SE, J2EE, J2ME, J2EE 1.4/1.5 Platform Overview and Roadmap, Introduction to the JSP and Servlets, Hands-on JSTL 1.1/JSP 2.0 , Code Samples; Demo, Where to get free Runtime, Compilers and Tools, Good books, good links - Q & A

Recap

- Last Time
  - Files and Streams
  - XML
    - Introduction
    - Java XML
    - DOM
    - DotPanel example
- Assigned Work Reminder
  - HW 3a: ThreadBank
  - HW 3b: LinkTester
    - Both due before midnight on Wednesday, August 6th, 2003

Today

- Today:
  - SAX XML Parsing
    - XMLDotReader example
  - Advanced Java
    - Regular Expressions
    - Assert
  - HW4 – XEdit
  - Java Implementation and Performance
    - Bytecode
    - Optimization Techniques

SAX XML Parsing (Handout #28)

- SAX parsing is cheaper than DOM parsing
  - SAX tells you of each element as it is found in a single pass of the XML document
  - We must maintain state ourselves

- XMLDotReader Examples
  - Code walkthrough in emacs…

Advanced Java (Handout #29)

- Features that are new in Java 1.4
  - Regular Expressions
  - Assertions

Regular Expressions

- Regular Expressions
  – Regular expressions ("regex's" for short) are sets of symbols and syntactic elements used to match patterns of text
  – Example: cp *.html ../
    - Here *.html is a regular expression!
  – Bottomline: used for matching patterns in text
    - Can very often state some very complex patterns in a simple regular expression
  – Resources
- http://developer.java.sun.com/developer/technicalArticles/releases/1.4regex/

Pattern

- java.util.regex.Pattern
  - Represents a regular expression pattern
  - Supports Perl-style regular expressions
    - \w+ \s [^a-z0-9]* etc.
  - Need to use double backslash (\\) in strings to get a single backslash (\)
    - \\s translates to \s set to the regular expression engine

Matcher

- java.util.regex.Matcher
  - Create a matcher out of a pattern and some text
  - The matcher can search for a pattern in the text
  - find() searches for an occurrence of the pattern
    - Searches from a point after the previous find()
  - group() returns the matched text from the previous find
  - Matcher support lots of different ways to look and iterate with the pattern on the text
    - Refer to the API documentation for details

Regular Expression Example

```
import java.util.regex.*;
...
// Extract email addrs from text
// email addr is @ surrounded by \w.-_
// We need to use double \\ in the " string, to put a single \ in the pattern
// (\w represents a 'word' character: a-zA-Z and _
String text = "blah blah, nick@cs, binky binky foo@bar.com; spam_me@foo.edu ";

String re = "[\\w\\.\\-]+\\@[\\w\\.\\-]+";
Pattern pattern = Pattern.compile(re);

// Create a matcher on the string
Matcher matcher = pattern.matcher(text);

// find() will iterate through matches in the text
while (matcher.find()) {
    // group() returns the most recently matched text
    System.out.println("email:" + matcher.group() );
}
```

Regular Expression Example

```
/*
        Output
         email:nick@cs
         email:foo@bar.com
         email:spam_me@foo.edu
*/
```

pattern.split()

- Used to extract the strings separated by a given pattern

```
// Use split() to extract parts of a string
String text2 = "Hello, what's with all the punctuation, and stuff here;   I want just
    the words.";

// The pattern matches one or more adjacent whitespace or punctuation chars
Pattern splitter = Pattern.compile("[\\s,.;]+");

// Split() uses the pattern as a separator, returns all the other strings:
// "Hello" "what's" "with" ...
String[] words = splitter.split(text2);
for (int i=0; i<words.length; i++) {
    System.out.println(words[i]);
}
```


Static Pattern.match()

- Static convenience method
  - Builds a pattern and matcher, runs the matches() method against the given text
  - Returns true is the *entire text* matches the pattern
  - Less efficient
    - Pattern and matcher are instantiated, used once and discarded
  - Useful for simple cases

```
boolean found = Pattern.matches("\\w+\\s\\w+", "hello there");
System.out.println(found);          // true
```

Assertions

- Assert
  - Added in Java 1.4
  - Use assertions to sprinkle code with tests of what should be true.
    - The assertion throws an AssertionError exception if the test is false.
  - Helps document what you think is going on, say, at the top of each loop iteration
    - help find bugs more quickly in this code, and in client code that calls this code.
  - The assert code may be deleted by the JVM at runtime, so do not put code that must execute in the assert. Assert should do read-only tests.

Assertions continued

- Assert
  - To compile with asserts, use the '-source 1.4' to javac.
    - If you compile this way, the code will only work on a 1.4 or later JVM.
    - By default at runtime, assert is not enabled -- they are NOPs
  - Turn asserts on with the -enableassertions switch to the 'java' command
    - (-ea is the shorthand)
    - java -ea // turns on asserts for the whole program
    - java -ea MyClass // turns on asserts for just that class
    - java -ea -da MyClass // turn on asserts, but turn them off for MyClass

Assert Example

```
public static void main(String[] args) {
    int len=1;

    // assert a condition that should be true
    assert len<100;
    // Can include a : <string> after the assert that goes in the error printout
    assert len<1 : "len=" + len;

    /*
    Output:
    Exception in thread "main" java.lang.AssertionError: len=1
      at Assert.main(Advanced.java:80)
     */

      // Suppose your code calls foo(), and it returns 0 on success
      // Never do this:
      //   assert foo()==0;
      // Do it this way, so it still works if the assert is disabled
      //   int result = foo();
      //   assert result==0;
    }
}
```

HW 4: XEdit (Handout #30)

- XEdit
  - Tool to do search and replace on XML files
  - Your job:
    - Traverse the DOM to do search and replace
      - Can use regexs if you want or just use String methods such as indexOf()
  - Simple assignment – 2-5 hours
- Due
  - Before midnight, Wednesday, August 13th, 2003

- Java Compiler Structure
  - .java files contain source code
  - Compiled into .class files which contain *bytecode*
- Bytecode
  - A compiled class stored in a .class file or a .jar file
  - Represent a computation in a portable way
    - As a PDF is to an image
- Java Virtual machine
  - Abstract stack machine
    - Bytecode is written to run on the JVM
  - Program that loads and runs the bytecode
    - Interprets the bytecode to "run" the program
  - Runs code with various robustness and safety checks

Verifier and Bytecode

- Verifier
  - Part of the VM that checks that bytecode is well formed
    - Makes Java virus proof
  - A malicious person can write invalid bytecode, but it will be detected by the Verifier
    - Usually no verifier errors since the compiler produces “correct” bytecode
    - Still possible to write bytecode by hand
- Bytecode example
  - javap –c will print the actual bytecode for a class

Bytecode Primer

- The byte code executes against a stack machine -- adding 1 + 2 like this

```
iload 1;        // push a 1 onto the stack
iload 2;        // push a 2 onto the stack
add;            // add the two numbers on the stack
                // leaving the answer on the stack
```

- "load" means push a value onto the stack
- aload_0 = push address of slot 0 -- slot 0 is the "this" pointer
- iload_1 = push an int from slot 1 (a parameter)
- getfield -- using the pointer on the stack, load an ivar
- putfield -- as above, but store to the ivar

Student Bytecode example

```
nick% javap -c Student
Compiled from Student.java
public class Student extends java.lang.Object {
    protected int units;
    public static final int MAX_UNITS;
    public static final int DEFAULT_UNITS;
    public Student(int);
    public Student();
    public int getUnits();
    public void setUnits(int);
    public int getStress();
    public boolean dropClass(int);
    public static void main(java.lang.String[]);
}
<snip>
```

Student Bytecode Example

```
Method int getUnits()
   0 aload_0
   1 getfield #20 <Field int units>
   4 ireturn
Method void setUnits(int)
   0 iload_1
   1 iflt 10
   4 iload_1
   5 bipush 20
   7 if_icmple 11
  10 return
  11 aload_0
  12 iload_1
  13 putfield #20 <Field int units>
  16 return
Method int getStress()
   0 aload_0
   1 getfield #20 <Field int units>
   4 bipush 10
   6 imul
   7 ireturn
```

JITs and Hotspot

- Just-In-Time Compiler
  - JVM may compile the bytecode into native code at runtime
    - Maintains robustness/safety checks (slow startup)
- HotSpot
  - Does a sophisticated runtime optimization for which part to compile
  - Sometimes does a better job than native C++ code since it has more information about the running program
- Future
  - May cache the compiled version to speed up class loading
  - Bytecode is a distribution format
    - Similar to PDF

Optimization Quotes

- Rules of Optimization
  - Rule 1: Don’t do it.
  - Rule 2 (for experts only): Don’t do it yet.
    - M.A. Jackson

- “More computing sins are commited in the name of efficiency (without necessariy achieving it) that for any other reason – including blind stupidity.” – W.A. Wulf
  - Y2K bug! – saving 2 bytes!

- We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” – Donald Knuth

Optimization 101

- Reality
  - Hard to predict where the bottlenecks are
  - Easier to use tools to measure the bottlenecks once the code is written
    - Write the code you want to be correct and finished first, then worry about optimization
- "Premature Optimization" = evil
  - Classic advice from Don Knuth
  - Write the code to be straightforward and correct first
  - May already be fast enough!
    - If not, measure the bottleneck
    - Focus optimization on bottleneck using Algorithms and Language optimizations

Optimization 101

- Data Structures
  - Have a profound influence on performance
    - Early design helps once
  - Choice of datastructure can constrain what algorithms you can use
- Proportionality to Caller
  - Foo() takes 1 ms. Bar() calls foo.
    - If Bar() takes 20 ms, it's not worth looking at Foo()
    - If Bar() takes 2ms, then we should look at Foo()

Optimization 101

- 1-1 User Event Rule
  - If something happened a fixed number of times (1-3) for each user event, then it's not worth looking at
  - If something happens 100s of times for each user even then it is worth looking at
  - User events are very slow from the CPU's perspective

Java Optimization Tip #1

- 1-10-100 Rule
  - Assignment – 1 unit of time
  - Method call – 10 units of time
  - New Object or Array – 100 units of time
    - Rule of thumb only. Not scientific.
    - Hard to determine the actual cost
- Also sometimes known as the 1-10-1000 rule, but modern GC is much more efficient
  - Bad idea to try and maintain your own free list. The GC knows best.

Java Optimization Tip #2

- int getWidth() vs. Dimension getSize()
  - getSize() requires a heap allocated object
  - getWidth() and getHeight() may just be inlined to move the two ints right into the local variables of the caller code
- With Hotspot, shortlived object (like Dimension) are less of a concern…

Java Optimization Tip #3

- Locals are faster than Instance variables
  - Local (stack) variables faster than any member variables of objects
    - Easier for the optimizer to work with
- Inside loops, pull needed values into local variables
  - 1. Slow: message send
    - … i < piece.getWidth()
  - 2. Medium: instance variable
    - … i < piece.width
  - 3. Fast: local variable
    - … final int width = piece.getWidth
    - … i < width
      - This is faster since the JIT can put the value in a native register

Java Optimization Tip #4

- Avoid Synchronized (Vector)
  - Synchronized methods have a cost associated with them
    - This is significantly improved in Java 1.3
  - Can have synchronized and unsynchronized methods and switch based on some flag
  - Use “immutable” objects to finesse synchronization problems
  - Vector class is sychronized for everything
    - Use ArrayList instead!
    - If you can use a regular array, even better

Java Optimization Tip #5

- StringBuffer
  - Use StringBuffer for multiple append operations
  - Convert to String only when done
- Automatic case
  - Compiler optimizes the case of several string +'d together on one line
  - String s = "a string" + foo.toString() + "more"
- No:

```
String record;                  // ivar
void transaction(String id) {
    record = record + " " + id;        // NO, chews through memory
}
```

- Yes:

```
StringBuffer record;
void transaction(String id) {
    record.append(" ");
    record.append(id); + id;
}
```

Java Optimization Tip #6

- Don’t Parse
  - Obvious but slow strategy – read in XML, ASCII, etc.
  - Build a big data structure
- Faster approach
  - Read into memory, but keep as characters
  - Search/Parse when needed
  - Or Parse only subparts

Java Optimization Tip #7

- Avoid weird code
  - JVM will optimize most stadard coding styles
    - So write code in the most obvious, common way
  - Weird code is often the result of an attempt at optimization!
- Let the JIT/Hotspot do it’s thing!

Java Optimization Tip #8

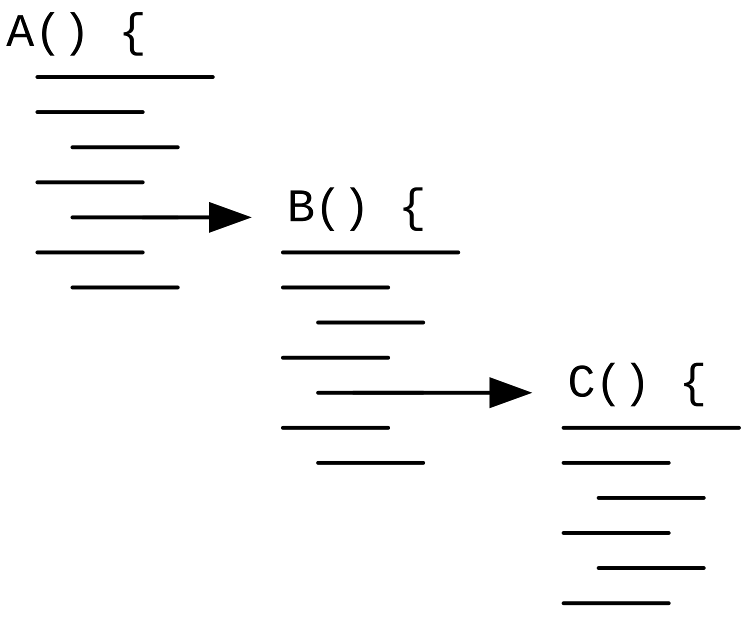
- Threading / GUI Threading
  - Use separate thread to ensure the GUI is *snappy*
- Pros
  - Makes best use of parallel hardware
- Cons
  - Software is harder to write
  - Bugs can be subtle
  - Locking/Unlocking costs

Java Optimization Tip #9

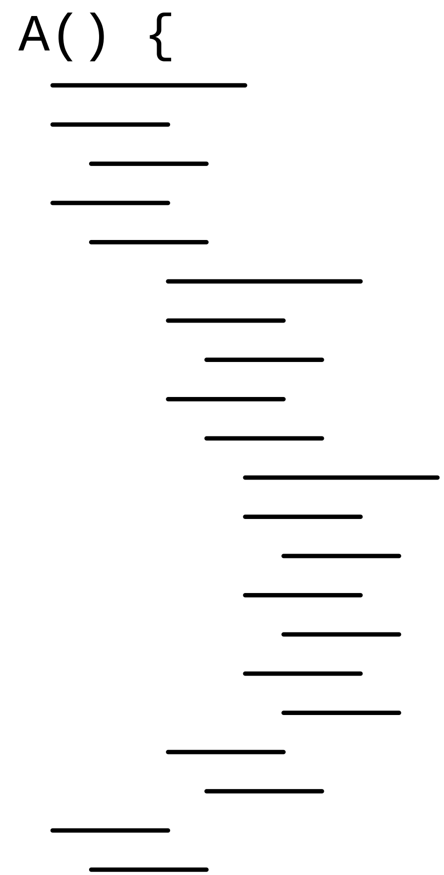
- Inlining Methods/Classes
  - JVM optimizers and HotSpot use aggressive inlining
    - Pasting called code into the caller code
  - *final* keyword
    - for a class means it will not be subclassed
    - for a method means it will not be overridden
  - Use final keyword to help the optimizer do more inlining

Java Optimization Tip #9 cont.

Non-Inlined



Inlined



Java Optimization Tip #9 cont.

- Advantages of inlining
  - Values are passed from A() to B() to C()
    - After inlining, the values can just stay in registers
    - Reduced number of load/saves
  - Propogation of analysis
    - Having code inlined can often lead to better optimizations since the compiler can see values from start to finish

Java Optimization Tip #10

- Think about memory traffic
  - Old: CPU bound
  - New: Memory bound
    - CPU operations are cheaper and faster
    - Once data is in the cache it is cheaper to work with
      - Reduce the roundtrips to memory, disk, network
  - Linked List vs. Chunked List
    - Linked List: Read a node, then fetch the next node
    - Chunked List: Each element contains a small array of elements
      - Makes better use of cache lines/memory pages
      - … some of this is very low level! ☺

Summary!

- Today
  - SAX XML Parsing
    - XMLDotReader example
  - Advanced Java
    - Regular Expressions
    - Assert
  - HW4 – XEdit
  - Java Implementation and Performance
    - Bytecode
    - Optimization Techniques
- Assigned Work Reminder
  - HW 3a: ThreadBank
  - HW 3b: LinkTester
    - Both due before midnight on Wednesday, August 6th, 2003
  - HW 4: XEdit
    - Due before midnight on Wednesday, August 13th, 2003

