JavaCon 2001: Java University's east coast campus

Sun Microsystems expands educational offerings

Daniel H. Steinberg (DSteinberg@core.com)
Director of Java Offerings, Dim Sum Thinking, Inc.
March 2001

The Java University program from Sun Microsystems is expanding. As more Java conferences offer Java University sessions, more people have an opportunity to learn about Java technology in concentrated, code-centered, day-long sessions. At the International Conference for Java Development, the Java University classes were small, and attendees had plenty of opportunities to ask questions of the experts. In this article, I review two of the Java University sessions that were held at the International Conference for Java Development 2001.

Contents:

Related dW content:

Excerpts from Peter Haggar's *Practical Java*
Pass-by-value semantics in Java applications

Learning from guys with a lot of experience is a great way to spend a day. Tuesday, Sun Microsystems' Java University program assembled a group of experts, most of whom are not Sun employees.

- The day began with a keynote address about components from Sam Patterson, whose company, ComponentSource, distributes components. Patterson later co-taught the course on developing EJBs.
- Marty Hall, author of *Core Servlets and JavaServer Pages*, taught the course on JSP files and servlets using material from his popular book.
- IBM engineer Peter Haggar taught the advanced Java programming language techniques, including and expanding on material from his *Practical Java Programming Language Guide.*
- Finally, Java University favorite, Phil Heller taught his popular fast-path review for the Java2 Programmer Certification test. Phil works with Sun on grading their Developer Certification test and co-wrote the *Complete Java2 Certification Study Guide* with Simon Roberts and Michael Ernest.

Let's look at Sam Patterson's keynote address and Peter Haggar's discussion of advanced programming techniques.

Adapting to change with components

Sam Patterson, CEO of ComponentSource, delivered the morning keynote address on the advantages of components. He recommended that companies think in terms of reuse, advising that you "reuse before you buy, and buy before you build." With commercially available Java components, you can accelerate the development process and speed your time to market.

Patterson's business is distributing components, so it was not a surprise to hear him advocate their use. He did, however, discuss one of the key philosophical issues in component development and deployment: should components be distributed as white box or black box components. You may be tempted to think that because white box components are distributed with source code, they are preferred. After all, that's what open source is all about. Patterson argued that if you are developing with components and stressing reuse, white box components can actually break things. When developers can modify the code, they are starting a new version of the product. Without some sort of open-source project management in place, the result is several different versions of the original component. This means that when the original author of the component distributes an update to the component, you have to roll your changes into the new version. In addition to being difficult to upgrade, white box components propogate bugs and don't support an open market.

Patterson contrasted this with black box components. In this case, users wrap the components to fit into their system. If the author of the black box changes the implementation or adds functionality, it is easier to incorporate these changes into your system. He did point out that maintenance and upgrade requires the user to "unplug" the old version and "plug in" the new. Patterson said that incremental development and testing is easier with black box components. You can address single use cases and then extend the specification of the component. In his opinion, black box components improved reliability, maintenance, and stability. An audience member asked whether Patterson was arguing against open-source development. Patterson answered that black box components begin as white box components and that open source is a great way to initially develop the components. Once they are mature, however, he advocated that they become black box and that the source be shipped with them only as documentation -- not to encourage change from the community. In answer to a separate question, he pointed out that if you want to make money from selling components, you are less likely to ship the source.

Advanced techniques
I usually spend the day at a conference flitting from class to class, getting a taste of each one. This time I decided to get the student experience by spending the entire day in Peter Haggar's class on advanced techiques for Java programming. In the six and a half hours of class time, Haggar covered a lot of material. He explained that some of the material wouldn't feel like "advanced techniques," but that he knew from the responses he received from his book that many of the techniques he was covering continue to be misunderstood.

Parameter passing
Haggar summed up his first point with a quote from Arnold and Gosling's book *The Java Programming Language, Second Edition*. The authors stress that there "is exactly one parameter passing mode in Java -- pass by value -- and that helps keep things simple." Haggar noted that many Java programmers are under the idea that reference types are passed by reference. He demonstrated that this isn't true by taking a C++ program where parameters are passed by reference into a `swap()` method that swaps what the variables point at. The same program in Java code had no effect. The local variables within the swap method swapped what they were pointing at but the original variables were unchanged.

This example pretty much set the tone for the day. Take something we think is true or false and construct an example that tests our assumption. His examples were succint and clearly made their point.

Equality
Again, much of the material from this section of Haggar's presentation can be found in his book. He began by reviewing that `==` is used to test that two things are the same. For primitive types, this operation returns what is expected. For reference types, this operation returns true only when the two references point to the same object. What you often want to know, however, is if the two objects that are referenced are equivalent. For this you need to use the `equals()` method. Haggar had us consider the following example.

equals() example

```
class EqualTest{
  public static void main(String args[]){
    String s1 = "Programming Language";
    String s2 = "Programming Language";
    StringBuffer sb1 = new StringBuffer("Programming Language");
    StringBuffer sb2 = new StringBuffer("Programming Language");

    System.out.println(s1==s2);
    System.out.println(sb1==sb2);
    System.out.println(s1.equals(s2));
    System.out.println(sb1.equals(sb2));
  }
}
```

Perhaps surprisingly, the output is "true, false, true, false." If you've studied for the Java Programmer's exam, perhaps you know that the first true has to do with a single String being created in the constant pool. The final false is surprising. Haggar explained that it is the result of the equals() method not being implemented in the StringBuffer class. Now that he had our attention, he went on to explain what to consider when implementing equals() for yourself. Although much of what Haggar covered was also in his book, he brought the material to life.

Code optimization
Other topics included exception handling, multithreading, finalization, and immutable objects and cloning. Haggar also devoted one section to performance. His advice included using a profiler, looking at the cost of object creation, and using lazy evaluation. We also spent some time looking at bytecode. It wasn't the first time that I'd looked at bytecode, but it was a reminder to stop every once in a while to see what the compiler generates. For example, consider the following code.

Optimization example

```
class concat{
  public void foo(){
    String s = "The ";
    s+= "Java ";
    s+= "Programming Language";
  }
}
```

Create bytecode with the command `javap -c concat >concat.bc`. You'll see that first a String is created with the value "The ". Then a StringBuffer is created. The value of the String is appended to the empty StringBuffer. Then a String is created with the value "Java ". This is appended to the StringBuffer and this is converted to a String. The process is repeated to concatenate the remaining String. That's a lot of overhead from such simple code. Looking at the generated bytecode helped us see this.

Summary
If you are planning on attending JavaOne or another Java conference with Java University offerred, check out the course offerings. By now the organizers have a pretty clear idea of how to choose presenters and topics. Haggar, Patterson, and Heller will be on the Java University schedule at JavaOne in June.

Resources

- Find out more about Java University.
- Visit the main site for the International Conference for Java Development 2001.
- Get the scoop on the upcoming JavaOne conference in June.
- Read excerpts from Peter Haggar's *Practical Java* on developerWorks. Peter received so many comments on the excerpt Understanding that parameters are passed by value and not by reference, that he wrote a follow-up article on this specific topic.
- Discover an excellent reference for using the Java platform for Web-enabled applications in *Core Servlets and JavaServer Pages* by Marty Hall.

About the author

Daniel is the co-author of the *Java 2 Bible* and is a Java trainer and consultant for Dim Sum Thinking. He spends as much time as possible with Kimmy the wonderwife and their two daughters. Late at night when his family falls asleep, he sneaks back to his computer to write books and articles about Java programming and industry news. His hobbies include cooking with his daughters and Java development for Mac OS X. Contact Daniel at DSteinberg@core.com.

e-mail it!

What do you think of this article?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback