Java Under The Hood and Future

Java 7 Ideas
• Some sort of closure system -- syntactic show way to make little bits of code to pass around
  SwingUtilities.invokeLater( Runnable() { ..code.. } );
• Maybe get rid of "erasure" generic system -- generic types work more fully (at the cost of compatibility
  with old JVMs, while old source code will compile fine).

Java Compiler Structure
• Compile classes in .java files -- produce bytecode in .class files

Bytecode
• A compiled class is stored in a .class files or a .jar file
• Represent a computation in a portable way -- as PDF is to an image.

Java Virtual Machine
• The Java Virtual Machine (JVM) is an abstract "stack machine" -- the bytecode is written to run on the
  JVM, manipulating values by pushing and popping them on a virtual stack (examples below).
• The JVM is also the name of the (written in C/C++) program that loads and runs the bytecode. The JVM
  interprets the bytecode to "run" the program.
• The JVM runs the code with the various robustness/safety checks in place -- robustness vs. performance
  tradeoff. Java aims to have portability to different hardware and safety.

Portability and Competition
• Some sort of JVM/bytecode structure seems like a good architecture moving forward to achieve
  portability -- Microsoft is using it with .net, but .net is targeted only at Windows while Java targets all
  OS's. Java is an especially safe domain for code, now that it is open source.
• Portability can mean the ability to use a different CPU in the future. Think about x86 architecture --
  carrying lots of baggage to support code compiled for an old chip.
• Intel thinks that having software portable across OS's is great, so long as it only runs on Intel chips
• Microsoft thinks that portable software is great if that means it runs on different chips, just so long as it
  only works with Windows.
• Customers sincerely love portability -- the opportunity to switch among software and hardware vendors
  creates competition --> avoiding lock-in, yielding better prices and features. There is a linkage between
  portability and competition. That's why consumers benefit and vendors are, at times, wary. There is a
  theory that the Internet has been such a huge success because standards, like HTTP and HTML, have
  led to rapid competition, yielding great price/performance.

Verifier
• The JVM also has a "verifier" that checks that the bytecode is well-formed (e.g. an int is never directly
  used as a pointer, a variable is never used without being initialized). This is a step in making Java
  robust/virus-proof.
• A bad guy can try to feed a "bad" program to the JVM, but the verifier will catch in places where the
  program tries to get around the runtime type, array, etc. checks.
• Usually, you don't see verifier errors, since the compiler will only generate "correct" bytecode, but the verifier is still needed, in case a bad guy hand crafts some bad bytecode.

• In Java 6, a new verifier architecture is being put in which does more of the verification at compile time - making the load/run process a little quicker.

ByteCode Example
• You can use the "javap" command to print out the bytecode for a class. Normally it prints a summary. The -c switch causes it to print the actual bytecode. The "javap" command looks for .class files in the current directory, just like the "java" command.

Bytecode Strategy
• The bytecode is just a description of the computation -- moving values around, computing things.
• JIT/HotSpot can translate the bytecode into real register machine code for the particular architecture. Point: the bytecode describes the computation, but it does not actually need to run exactly that way. The bytecode can be translated into anything, so long as it produces exactly the same output.
• The Bytecode format is deliberately structured to enable the verifier to check the necessary aspects of the code at load-time.

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, later slots are the parameters
• 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

Why a Stack?
• Why does bytecode use a stack to represent the computation?
• It this way, the bytecode description does not depend on a particular number of registers
• It describes the computation in a clean way -- the stack bytecode describes the computation, but it does not need to be implemented that way.
• The JIT/Hotspot layer (below) can translate from the stack storage paradigm to use the actual number of registers on a particular machine. At runtime, the JVM does not need to actually use exactly this stack to run the code -- the bytecode stack is just a description.

Student Bytecode
• The Student class has a "units" instance variable, and getUnits(), setUnits(), getStress() methods

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[]);
```java
Method int getUnits()
0  aload_0
1  getfield #20 <Field int units>
4  ireturn

Method void setUnits(int)
0  iload_1
1  iflt 10  // if negative, go to step 10
4  iload_1  // if <= 20, goto 20
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 -- modern JVMs compile the bytecode to "native" code for the actual CPU at hand at runtime (with the robustness checks still in). This is one reason why java programs have slow startup times and use more memory than you might guess.

- This happens to your code under the hood, without your code knowing. The native version just gets "swapped in" as your code runs. Did you ever notice your code seeming so speed up dramatically after a few seconds?

- The "Hotspot" project (now open source) within the Sun JVM tries to do a sophisticated job of which parts of the program to compile and when -- don't just do the whole thing at once. In some cases, hotspot can do a better job of optimization than a C++ compiler, since Hotspot is playing with the code at runtime and so has more information.

- IMHO, the most interesting optimizations now happen at runtime with a Hotspot type system vs. the old view that optimization is done by the compiler.

- What I love about Hotspot is that every year it gets 15% smarter. Code I wrote years ago automatically runs a little faster each time.

**Bytecode Future**

- Maybe cache the compiled version, to speed class loading

- Think of bytecode as a distribution format, while at runtime something more native is happening. The native code is created from the bytecode when the app is installed, or perhaps created when the program is first run and then cached.

- This creates great potential freedom for the code to work on unheard of hardware and operating systems in the future, making bytecode a very safe format if you just want the computation to work in the future.

- Security implication of keeping the compiled native version around -- needs to be stored in a secure way, since it gets past the verifier. This is true of the JVM binary too.

**Java Performance Trick: Locals Faster Than iVars**

- Local (stack) variables are faster than member/instance variables of any object (the receiver or some other object). Locals are also easier for the optimizer to work with for a variety of optimizations.
• Access to an ivar in an object, eg foo.width or this.width, is typically slower than access to a local stack variable.

• Inside loops, pull needed values into local variables (int i, temp;).

• Suppose we are in a for loop:
  - for (int i=0; i<piece.getWidth(); i++) {
  
1. Slow -- message send
    - ...i < piece.getWidth()

2. Medium -- instance variable -- with a good JIT, this case and (1) above are essentially the same.
    - ...i < piece.width
    - -or-
    - ...i < this.width (suppose the code is executing against the receiver)

3. Fast -- pull the state into a local (stack) variable, and then use it. This makes it easier for the JIT to pull the value into a native register. If the value is in an ivar, the runtime may need to retrieve it from memory every time it is used. It's hard for the runtime to deduce that .width is not being changed, so it reloads it from memory every iteration. In contrast, it's easy for it to deduce that localWidth is not being changed in the loop, so it can just put it in a register and use that value the whole time.
    - int localWidth = piece.getWidth();
    - ...i < localWidth...
    - -or-
    - // Use "final" to make it even more clear for the JIT
    - // (Supposedly modern VMs are smart enough to figure
    - // this out without needing the "final")
    - final int localWidth = piece.getWidth();

• The code does not actually need to read and write values to some "localWidth" memory location -- the JIT can re-write the code to use a register to hold that value, and it's fine so long it updates all the code to use the register. That's easy with local variables, since the code that touches the variable is well constrained.

• Theme: the JIT can do aggressive re-writes of the code.

• Theme: good to avoid creating memory traffic

Hotspot: Inlining Methods/Classes

• JVM optimizers, and Hotspot in particular, make aggressive use of inlining -- pasting called code into the caller code.

• Inlining enables many other optimizations.

• The "final" keyword for a class means it will not be subclassed, and a "final" method will not be overridden.

• These assumptions can help HotSpot figure out message-method mappings before they happen.

• As Hotspot gets smarter, it can figure things out even without "final".

Not Inlined

A() {
  
  -
  - -+-> B() {
    
    -
    -    - -+-> C() {
      
      -
      -
    }
  }

  -
Inlined

\[
A() \{ \\
\}
\]

Inline Advantages

- **Data Flow**
  - Values in \( A() \) are passed to parameters in \( B() \), passed to \( C() \), where they are used.
  - Now, the flow of that value through the whole \( A/B/C \) sequence can be analyzed -- the value can just live in one variable/register for the whole computation
  - This saves on memory traffic, which is just what we need

- **Propagation of analysis**
  - Suppose \( A() \) is running a for(\( i=0; i<\text{array}.\text{len}; i++ \) ) loop, and calls \( B() \) and \( C() \), passing in the \( i \) value.
  - Down in the \( C() \) code, a statement like \( \text{array}[i] \) would need to be checked for \( i<0 \) and \( i>=\text{len} \) normally.
  - But now Hotspot can see the value of \( i \) from start to finish, show that it's always in range, and so remove the cost of the array-bounds check.
  - Similar optimizations work for example: checking if pointers are null, checking instanceof on a pointer.

Java Bytecode Ecosystem

Java Ecosystem

- JVM/bytecode is a high quality, stable runtime environment -- especially now that Java is open source
- Other languages can be compiled to bytecode -- run in the JVM
- e.g. Jython project compiles Python to bytecode, JRuby runs ruby in the JVM

BeanShell Example

- BeanShell is a very simple scripting language
- BeanShell code is compiled on the fly to bytecode which interacts with full Java class and data structures
- Beanshell code understands Java objects, methods etc., integrating nicely with all that infrastructure
- Can have a Beanshell text area, the end user types some code in and hits run ... it compiles to bytecode and runs on the fly within the JVM. (e.g. the text area at the lower left below)
In this example the "dots" ArrayList has been exported to the Beanshell context at the lower left. The user can write Beanshell code there, hit the Run button, and the code is compiled and run against the JVM and all its objects.

The Future of Programming Languages

1. Java -- Structured, "high road"
   - "Programmer efficient"
   - GC for memory
   - Great libs of off-the-shelf code
   - Robust type system, anti-virus, anti-bug safety, checking of things at runtime (makes it a bit slower)
   - Typed -- has a real type system to structure the code (makes it a bit verbose)
     - Most attractive for large and team projects
     - Attractive for APIs to be used by others (i.e. Eclipse auto complete)
   - Speed is very good in some ways, but weak in some ways and startup time is not great. HotSpot paradigm of speedup via JVM optimization looks very good going forward.
   - Memory use is high
   - Now that Java is open source, Java is an extremely safe, open infrastructure technology if you just want things just to work.
   - An excellent default choice unless there is a mismatch between the problem and java
   - Open source projects tend to attract good community iteration and improvement (e.g. Hotspot research) - we'll see how that plays out for Java.
   - C# is like Java but locked in to Microsoft. Use it like Visual Basic -- to implement things that only need to work on Microsoft OS.

Java Niche / Future
   - Large, complex project -- play to Java's strengths in safety, structured type system
   - Use very often for the server-side of Web applications
   - Benefits from programmer efficiency (as Moore's law runs along, programmer efficiency looks better and better. Hardware gets cheaper, programmer time to create correct software does not!)
Java Weaknesses

- Typing can be annoying
- Runs a little slower, users more memory
- Portable, but hard to access OS-specific features
- GUI can look ok, but hard to make it look really great

2. C/C++ -- "old road" / "fast road"

- Can run fast, use the minimum memory
- Not portable in the way that Java is, but can access system specific features
- The standard lib is tiny, but on the other hand, there are totally free, open source versions
- IMHO C/C++ is best for things that are not that complicated, and possibly where performance really matters.
  - e.g. disk driver, JPEG decompression -- having a nice, low-level C version to build on is great.
  - I wonder if the optimal system of the future will have some low-level parts in C, but with the upper, "system" part with a lot of custom coding in Java. From the Java side, you may not be aware what "native" C parts there are under the hood.
  - As a system, gets large, complex, and has a lot of people... it's tempting to build it in a safe system such as Java
- Legacy -- many things nowadays are in C/C++ because they were started 10 years ago and still work today (e.g. Acrobat, MS Word), or things where performance really matters. If Microsoft could snap its fingers and have MS Word in Python or C#, they would do it in a second. It's in C++ now because it was that way 10 years ago.

3. Python / Ruby / Javascript -- Scripting, "low road"

- Short little programs -- many projects have some part where this is a good fit (e.g. unit tests)
- Lacks type system in the code -- just declare variables/parameters and go -- very neat in its way
  - Makes the code short/dense, but hard to scale up to large projects
  - A language without declared types is great for 2 page programs. IMHO, typing is very useful for a 100,000 line project worked on by many people.
  - Contrast to Java large/team project
- Neat projects: "bean shell" and "Groovy"
  - Integrate scripting languages closely with Java -- run in JVM, access same classes objects
  - scripting languages integrated to work closely within Java -- use it for parts -- unit tests, allowing end-user scripting.