Doing Things with Java that Should Not Be Possible

Once Upon an Oak ...

by Heinz Kabutz

July 15, 2003

Summary

Let us walk down memory lane together as we look at the pre-cursor to Java, a language called Oak, developed by Sun Microsystems for use on their handheld device. The will answer many old questions: Why are there no enums in Java? Why does the protected modifier allow package access? What is an asynchronous exception?

Once Upon an Oak ...

During Java courses, we often end up speaking about the origins of Java, and it occurred to me that I had big gaps in that part of world history. History had never been my forte, at school this was my weakest subject at 35% (at my worst level). Perhaps I should add that I was very weak with *South African history*, which at the time of me going to high school supposedly started with the white settlers arriving on the shores, and consisted almost entirely of remembering dates. I found such parrotting frivolous, and so my marks were not too hot.

Trying to fill my gaps of Java's history, I started digging around on Sun's website, and eventually stumbled across the Oak Language Specification for Oak version 0.2. Oak was the original name of what is now commonly known as Java, and this manual is the oldest manual available for Oak (i.e. Java). For more history on the origins of Java, please have a look at The Green Project and Java(TM) Technology: An Early History. I printed the manual and kept it next to my bed in case of insomnia (so I thought). When I started reading it, I discovered answers to ancient questions: Why can we have only one public class per .java file? Why are protected members also accessible from the same package? Why do `short` fields use as much memory as `int` fields (at least pre-JDK 1.4)?

In this blog I want to highlight the differences between Java 0.2 (i.e. Oak 0.2) and what we have now. For your reference, I will include the section numbers in the Oak 0.2 specification.

Why is each public class in a separate file? (Section 1)

This is a question that I have frequently been asked during my courses. Up to now I have not had a good answer to this question. In section 1, we read: "Although each Oak compilation unit can contain multiple classes or interfaces, at most one class or interface per compilation unit can be public".

In the sidebar it explains why: "This restriction is not yet enforced by the compiler, although it's necessary for efficient package importation"

It's pretty obvious - like most things are once you know the design reasons - the compiler would have to make an additional pass through all the compilation units (.java files) to figure out what classes were where, and that would make the compilation even slower.

One-line Javadoc comments (Section 2.1)

Oak had the ability to write a one-line JavaDoc comment.

```
public class OneLineJavaDoc {
  //* The number of lines allowed in a document.
  public int numberOfLines;
}
```

This is *fortunately* not supported in Java as it is quite confusing.

Additional Keywords (Section 2.3)

Oak had some additional keywords that are not currently used in Java:

- `ushort`, `string`, `Cstring` and `unsynchronized` were already obsolete in version 0.2 of Oak. Isn't it amazing how quickly things become obsolete in our industry?
- `clone`, `const` and `goto` were keywords
- `protect` and `unprotect` were nasty keywords for writing terrible exception code. More about that later...
- `enum` was a keyword, but in the sidebar it said: `enum` isn't implemented yet. So, in answer to the question: Why does Java not have enum? The answer is not some heavy object-oriented philosophical answer about polymorphism and strategy patterns. It is simply that James Gosling didn't implement it in time and his management forced him to push the language out of the door before he could finish the feature :-)

Unsigned integer values (Section 3.1)

The specification says: "The four integer types of widths of 8, 16, 32 and 64 bits, and are signed unless prefixed by the `unsigned` modifier.

In the sidebar it says: "`unsigned` isn't implemented yet; it might never be." How right you were.

Storage of Integer values (Section 3.1)

In my newsletter on Determining Memory Usage in Java, I noted that when you have a data member of byte or short, that they still use up at least 4 bytes of memory. This was changed in JDK 1.4, but the historical reason is found in the Oak spec:

"A variable's type does not directly affect its storage allocation. Type only determines the variable's properties and legal range of values. If a value is assigned to a variable that is outside the legal range of the variable, the value is reduced modulo the range."

I wish I had known that when I wrote my first Java program. I spent a lot of time deciding which data types I wanted to use in order to save memory, but I was actually wasting my

time. Please note that this has changed as of JDK 1.4, so now it does help to use `bytes` and `shorts` to reduce the memory footprint.

Arrays (Section 3.5)

In Oak, you were able to declare arrays as follows:

```
int a[10];
a[5] = 1;
```

However, we were also able to declare an array in the current Java fashion, with `new`. I think having two ways of doing the same thing causes confusion, so I am very pleased that this way of making new arrays has been removed.

Const vs. Final (Sections 4.6 and 4.9)

It appears that `final` was initially only meant to be used for classes and methods, and that `const` was used for making fields constant.

Private did not exist (Sections 4.10)

This is the most surprising part of Oak 0.2. There were only three access levels as opposed to our current four. There was **private** which did not require a keyword, and which equated to our current "package private" or "friendly" or "package access" type of access level. All classes in a particular package could use all variables and methods declared in the classes in that package, regardless of **public**, **protected** and **private** declarations. I am very glad that they introduced a more private version of **private**, one that was only accessible within the class.

The lack of private as we know it today explains why when a member is protected, it is also accessible from within the same package. When Oak was written, protected was obviously accessible within the package because their private (our "package access") was the most restrictive access level. I seem to remember that in JDK 1.0 we had a fifth access level called "private protected", which meant that *only* subclasses could access the member.

This piece of surprising history also explains why fields are not private by default - they actually were "private" originally when private meant different things.

I don't need to emphasize how pleased I am that we now have the more restrictive "private" modifier. Without that, our industry would be in even more trouble.

Abstract Methods (Section 5)

Abstract methods were defined as in C++:

```
public interface Storing {
  void freezeDry(Stream s) = 0;
}
```

Assertions and Pre/Postconditions (Sections 7, 7.1 and 7.2)

Unfortunately the assertions in Oak 0.2 were not implemented in time, so they were thrown out to satisfy the release deadline. In case you think that assertions are back in JDK 1.4, have a look at the power that "those" assertions gave you:

```
class Calender {
  static int lastDay[12]=
    {31,29,31,30,31,30,31,31,30,31,30,31};
  int month assert(month>=1 && month<=12);
  int date assert(date>=1 && date<=lastDay[month]);
}
```

While objects are not required to obey the legality constraints within methods, the constraints are enforced at the entry and exit of every public and protected method.

I wish that James Gosling had worked a few extra weekends (if that were possible) to finish the implementation of `assert` as it appeared in the original Oak spec. Preconditions and Postconditions were also loosely defined, but were also kicked out due to time pressure. Pity.

Post-incrementing Strings (Section 8.1.4)

In Oak, you were able to post-increment a String. You could literally say `s1++;`, which was equivalent to `s1 = s1 + 1;`. The post-increment statement is often (if not always) implemented as +=1 so it seems that this also was true for Strings. Fortunately this is not allowed in Java anymore.

Goto ... (Section 9.3)

Of course, being based on C, Oak included the infamous `goto` statement. Fortunately this is not in Java.

Exceptions (Section 9.4)

Where does the name `RuntimeException` come from? Aren't all exceptions thrown at runtime? These are exceptions which are thrown *by the runtime system*.

In Oak 0.2, all exceptions were unchecked, meaning that there was no `throws` clause. My guess is that checked exceptions were only added once the whole exception hierarchy had already been set in wet concrete. I would have had a separate branch for checked exceptions, something like:

```
public class Throwable { }

/** Serious errors in the virtual machine */
public class Error extends Throwable { }

public class CheckedException extends Throwable { }
public class IOException extends CheckedException { }

public class UncheckedException extends Throwable { }
public class NullPointerException extends UncheckedException { }
```

This way you would avoid catching unchecked exceptions when you catch `CheckedException`. However, as it appears, the exception class hierarchy was developed before the idea of checked vs. unchecked exceptions, so we are stuck with an exception mechanism that will cause headaches for generations to come.

Asynchronous Exceptions (Section 9.4.2)

If your program gets an asynchronous exception, you are dead. A few weeks ago I was looking at a program that was throwing OutOfMemoryError. This can happen at any time really, which is why it is called an *asynchronous* exception. Another place where this can happen is with `Thread.stop()`. As you can imagine, this is inherently dangerous, that is why it is deprecated. In Oak, life was even more dangerous. You could cause an asynchronous exception in another thread using Thread's `postException()` method. Now *that* was dangerous! Imagine if other threads could cause asynchronous exceptions at any place in your code!

In order to safeguard your code, you could "protect" it. If you wanted to indicate that you had some critical code which could not handle asynchronous exceptions, you did the following:

```
protect {
  /* critical section goes here */
}
```

And code that was quite happy with asynchronous exceptions did the following:

```
unprotect {
  /* code that can afford asynchronous exceptions */
}
```

There was a note in the sidebar saying that the default would probably be changed to *not* allow asynchronous exceptions except in explicitely *unprotected* sections of code.

I'm very glad that this feature was binned. I cannot imagine how complex our Java programs would have become with it in place.

Summary

That's it. The rest of the manual was filled with a Glossary and an Index, to push the manual to 38 pages.

Even though this manual was written way back in 1994, it provided for fascinating reading, even late at night ;-)

Heinz

P.S. This article first appeared in **The Java(tm) Specialists' Newsletter**.

Talk Back!

Have an opinion? Readers have already posted 1 comment about this weblog entry. Why not add yours?

RSS Feed

If you'd like to be notified whenever Heinz Kabutz adds a new entry to his weblog, subscribe to his RSS feed.

About the Blogger

Heinz Kabutz enjoys driving Java to the limits, and then a bit beyond. He has been programming in Java since 1997 on several very unimportant projects. During that time, he has picked up some horrifying tips on how you can get the most out of Java. These are published on his bi-monthly "The Java(tm) Specialists' Newsletter" (http://www.javaspecialists.co.za). It is not for the uninitiated :-) Heinz received a PhD in Computer Science from the University of CapeTown. He loves living in South Africa as it is both beautiful and interesting. Professionally, Heinz survives by writing Java code, insulting, ahem, consulting, and presenting courses on Java and Design Patterns.