
Java 2007: The year in preview

Open source Java programming means developers are driving -- but where to?

Skill Level: Introductory

[Elliote Harold \(elharo@metalab.unc.edu\)](mailto:elharo@metalab.unc.edu)

Adjunct Professor

Polytechnic University

06 Feb 2007

2007 will go down in history as the year Sun Microsystems gave up the reins of the Java™ platform, releasing it under an open source license to the Java developer community. In this article, Java developer Elliott Rusty Harold predicts new directions for the Java platform, in everything from scripting to bug fixing to new syntax.

2006 was another boom year for the Java platform. The Java language retained its title as the world's most used programming language, despite an onslaught of competition from both Microsoft (C#) and the scripting community (Ruby). And, while the release of Java 6 would have been cause enough for celebration, that paled in comparison to the announcement that Java was going to go fully open source under the GNU General Public License. Can the momentum continue in 2007? Let's consider the odds.

The Java platform goes open source

Before 2007 is half up, Sun will release the Java Development Kit (JDK) under an open source license. Freeing the JDK is a huge step for the Java developer community, and it will drive the evolution of the Java platform for the next decade.

Expect the quality of the JDK to improve dramatically as programmers stop merely reporting bugs and start fixing them. Bug reports at the Java Developer Connection will include detailed analysis of what's broken in the JDK and provide patches for

fixing it. As *Linus's Law* states, "Given enough eyeballs, all bugs are shallow." That is, debugging is parallelizable. The same is true of optimization. Open source makes both *massively* parallelizable.

Forks in the road

Unfortunately, design is not as parallelizable as debugging and optimization. A clean API occasionally requires a dictatorial hand. The downside of dictators, however, is that sometimes they know what they're doing and sometimes they don't. Competition among would-be dictators is often the only way to discover the best solution to a problem.

Few companies can afford to develop multiple independent implementations of a product with the goal of throwing all but one away, but the open source community thrives on that sort of thing. So look for forks at all levels of the Java platform: language, virtual machine, and libraries. Most of these will fail, but that's okay. The good ideas will rise to the top. Some will take on a life of their own, and some will be merged back into the standard JDK. It probably won't be obvious by this time next year which are which, but the process should be well underway.

Sun will get the ball rolling in a few months by releasing an early beta of Java 7, Dolphin. The company can't release earlier versions of the JDK because of build problems and license encumbrances that are only cured in Dolphin. However, look for third parties to start chopping pieces out of the Sun release to produce passable, open source implementations of Java 6, Java 5, Java 1.4, and maybe even earlier versions.

Some of these early forkers will probably run afoul of Sun's trademarks and get nasty letters from the company's lawyers. We'll need a generic, untrademarked name for the language that everyone can use. I propose "J" -- hopefully no one can trademark a single letter.

Open source projects never die, they just fade away. Like the Blackdown Project before them, GNU Classpath, Kaffe, and other open source JDK projects are going to see their developers move on to other things. If a project hasn't reached 1.0 yet, it is unlikely to do so in the future.

Looking forward to Java 7

Dolphin will not be released in 2007. Next year is a more realistic goal. That said, work is underway and some features may make their debut as standard extensions earlier, or at least as betas.

It's unfortunate that it's far, far easier to add features to a language than to take them away. Almost inevitably, languages get more complex and confusing over time, not less. Even features that look good in isolation become problematic when piled on top of each other.

Unfortunately, the Java community has not yet learned this lesson, despite the debacle that is generics. There's just something about new syntax that's too cool and exciting for language designers to resist, even when it doesn't solve any actual problems. Thus the huge clamor for new language features in Java 7, including closures, multiple inheritance, and operator overloading.

I suspect that before the year is out we will see closures in a Java 7 beta, and we may well see operator overloading (50/50 chance), but multiple inheritance simply will not happen. Too much of Java is based on a singly rooted inheritance hierarchy. There's no plausible way to retrofit multiple inheritance into the language.

Currently there's a lot of syntax sugar on the table, some of which makes sense, some of which doesn't. A lot of the proposals focus on replacing methods like `getFoo()` with operators like `->`.

Lists

The first possibility is using array syntax for collections access. For example, instead of writing this:

```
List content = new LinkedList(10);
content.add(0, "Fred");
content.add(1, "Barney");
String name = content.get(0);
```

you could instead write this:

```
List content = new LinkedList(10);
content[0] = "Fred";
content[1] = "Barney";
String name = content[0];
```

Another possibility is allowing array initializer syntax for lists. For example:

```
LinkedList content = {"Fred", "Barney", "Wilma", "Betty"}
```

Both of these proposals could be implemented with a little compiler magic without changing the virtual machine (VM), an important characteristic for any revised syntax. Neither proposal would invalidate or redefine any existing source code, an even more important issue for new syntax.

One language feature that might make a real difference in developers' productivity would be built-in primitives for managing tables, trees, and maps, such as you encounter when working with XML and SQL. Projects like E4X in JavaScript land and C# and Linq in the Microsoft world are pioneering this idea, but sadly, the Java platform seems to be missing the boat. If anyone feels like making a potential game-saving play by forking the compiler, here is a very good place to look.

Properties

We'll probably also see some syntax sugar for property access. One proposal is to use `->` as a shorthand for calling `getFoo` and `setFoo`. For example, instead of writing:

```
Point p = new Point();
p.setX(56);
p.setY(87);
int z = p.getX();
```

you could write"

```
Point p = new Point();
p->X = 56;
p->Y = 87;
int z = p->X;
```

Other symbols including `.` and `#` have also been proposed in lieu of `->`.

In the future, you may or may not have to explicitly identify the relevant field as a property in the `Point` class, like so:

```
public class Point {
    public int property x;
    public int property y;
}
```

Personally, I'm underwhelmed. I'd like to see the Java platform adopt a more Eiffel-like approach in which we could actually use public fields. However, if getters or setters are defined with the same names as the fields, then reads and writes to the fields are automatically dispatched to the methods instead. It uses less syntax and it's more flexible.

Arbitrary precision arithmetic

Not operator overloading

It's worth noting that this reuse of standard mathematical symbology

is **not** the same thing as operator overloading, at least not the kind that causes problems in C++. The plus sign and other operators will still have an unambiguous meaning in any program. They do not change their meaning from one program to the next. Reusing the same syntax for similar operations makes code easier to read. Redefining syntax so it can mean different things in different programs makes code harder to read.

Another proposal to replace methods with operators aims at `BigDecimal` and `BigInteger`. For example, currently you have to code unlimited precision arithmetic like so:

```
BigInteger low = BigInteger.ONE;
BigInteger high = BigInteger.ONE;
for (int i = 0; i < 500; i++) {
    System.out.print(low);
    BigInteger temp = high;
    high = high.add(low);
    low = temp;
};
```

This could more clearly be written as:

```
BigInteger low = 1;
BigInteger high = 1;
for (int i = 0; i < 500; i++) {
    System.out.print(low);
    BigInteger temp = high;
    high = high + low;
    low = temp;
};
```

The proposal seems inoffensive enough, though it could possibly lead to overuse of these classes and consequent performance degradation in naive code.

Taking the JAM out of the JAR

Java 7 could fix the most long-standing source of irritation to Java developers: the various class loaders and associated classpaths. Sun is taking another whack at this problem with the Java Module System. Instead of a `.jar` file, data will be stored in a `.jam` file. This is a sort of "superjar" that contains all the code and metadata. Most importantly, the Java Module System will support versioning for the first time, so you can say that a program needs Xerces 2.7.1 but not 2.6. It will also allow you to specify dependencies; for instance, to say that the program in the JAM requires JDOM. It should also enable you to load one module without loading them all. Finally, it will support a centralized repository that can provide many different versions of many different JAMs, from which applications can pick the ones they need. If the JMS works, `jr/lib/ext` will be a thing of the past.

Package access

I'm also hopeful that Java 7 will relax access restriction just a bit. It may become possible for subpackages to see the package-protected fields and methods of classes in their superpackages. Alternately, it may be possible for subpackages to see the package-protected members of superpackages that explicitly declare their friendliness. Either way, this would make dividing an application into multiple packages much simpler and dramatically improve testability. As long as unit tests were in a subpackage, you wouldn't have to make methods public to test them.

Filesystem access

Filesystem access has been a major problem for the Java platform since 1995. More than 10 years later, there's still no reliable cross-platform way to perform basic operations like copying or moving files. Fixing this has been an open issue for at least the past three JDKs (1.4, 5, and 6). Sadly, boring but necessary APIs for moving and copying files have been shunted aside in favor of less common but sexier operations like memory-mapped I/O. Possibly JSR 203 will finally fix this and give us a plausible, cross-platform file system API. Then again, the working group may once again spend so much time on the relatively unimportant problem of true asynchronous I/O that the filesystem gets left at the altar once again. We should know by this time next year.

Experimentation

Whatever changes are made, it would be nice if they could be implemented in open source forks first, so we can see just how much or how little difference they really make. Toward this end, Sun's Peter Ahè has started the Kitchen Sink Project on java.net. The goal is to branch and fork the javac compiler repeatedly to test many different ideas like these. Blogging about pet features is one thing; actually producing running code is something else entirely.

Client GUIs

Although many people haven't noticed, the Java platform has been a real presence on the desktop for four or five years now. More than a few quality desktop applications have been written in Java code, including RSSOwl, Limewire, Azureus, Eclipse, NetBeans, CyberDuck, and others. These applications are written in nearly every GUI toolkit available including Swing, AWT, SWT, and even platform-native toolkits such as Mac OS X's Cocoa. I don't see any one toolkit winning over the others in the next year, although Swing seems to be doing a better job than the others at producing applications that retain a native feel.

Swing is still relatively challenging to develop in, but the situation may improve in the next year with the advent of the Swing Application Framework. This framework is currently in development in the Java Community Process as JSR 296. Here's what the JSR has to say about it:

Well-written Swing applications tend to have the same core elements for startup and shutdown, and for managing resources, actions, and session state. New applications create all of these core elements from scratch. Java SE does not provide any support for structuring applications, and this often leaves new developers feeling a bit adrift, particularly when they're contemplating building an application whose scale goes well beyond the examples provided in the SE documentation.

This specification will (finally) fill that void by defining the basic structure of a Swing application. It will define a small set of extensible classes or "framework" that define infrastructure that's common to most desktop applications.

The Swing Application Framework should support most of a typical application, allowing developers to just plug in at a few customization points such as startup and shutdown. It will handle saving and restoring windows and other parts of an application between startups and shutdowns. Finally, it will allow developers to write asynchronous actions that run outside of the Swing event dispatch thread.

Work is also ongoing to improve JavaBeans and everything that depends on it, including Swing. JSR 295 is defining a standard way to bind beans together such that updates to one bean are automatically reflected in the others. For example, a GUI grid bean could update automatically when its associated database bean changes.

Finally, JSR 303 is working on an XML-based validation language to specify declaratively what values any given bean may take on. You'll be able to say that an int property must be between 1 and 10 or that a String property must contain a legal e-mail address. With a little luck, this may all be available in beta form by the end of the year and should be finished in time for Java 7 next year.

The Java platform as a desktop language

While a few programmers have chosen to write their desktop apps in Java code because they prefer the language, most are driven by the desire to ship across multiple platforms. Interest in the Java platform as a desktop language is thus closely tied to the number of non-Microsoft desktops out there. Let's consider the year ahead for Java programming on the three major desktops.

Windows

Swing will continue to make small improvements to its Windows look and feel over the next year, especially with the shift to open source development. As a result, pure-Java programs like LimeWire will look more native than ever on Windows. But the language of choice for developing native Windows apps will remain C# (with a few C and C++ holdouts), and the framework will be .NET. Java code will not make any significant inroads into the Windows ecosystem.

Macintosh

Like Microsoft, Apple Inc. has pretty much abandoned Java code. Apple favors Objective C and Cocoa, but the end result is the same: Mac-only developers will continue to be forced off of Java code to Apple's preferred language and environment.

On the positive side, while Apple is no longer supporting Java code for proprietary APIs like QuickTime and Cocoa, the Apple VM is in better shape than it's been in for years. Apple's Java 6 port should be released soon. It won't be open source (unlike Sun's JDK), but open source programmers will start fixing its bugs anyway.

Linux

The GPL license will make it possible to bundle Java code with even the purest of open source Linux distros, which will make the Java platform a much more attractive language for Linux development. If only this had happened five years ago: The Linux community wouldn't have had to continue struggling with C all this time and Mono wouldn't have been necessary.

There already are Java bindings for both Gnome and KDE, so look for these to attract a lot more attention in the coming year. Also expect at least one major project to be launched to develop a Linux GUI program using Java code as the language rather than C, C++, or C#.

Ruby wins the race

Bloatware

JavaScript is already bundled with JDK 6. Additional languages may be added to JDK 7. That smells a little bloated to me. For one thing, there's no way Sun is going to be able to stop with just one more. If it picks BeanShell, the Groovy folks will demand inclusion. If Groovy gets in, the Rubyists will insist on being added. If Ruby goes in, can one really omit Python? The standard JDK is already way too large. Supporting multiple scripting languages is one thing, but bundling them? The diplomatic move would be to support them all but bundle

none.

On the positive side, Sun is exploring ways to reduce initial download size and application startup time, especially for applets and Java Web Start applications. Possibly a huge amount of the class library can be left on the server and only downloaded as needed or in a slow background thread.

The world would be a boring place if we only spoke one language. While the Java platform is an excellent choice for full-blown application development, it's never really fit the need for small programs or macros. Java 6 recognized this by adding the `javax.script` package for interoperating with scripting languages like BeanShell, Python, Perl, Ruby, ECMAScript, and Groovy as well as an `invokedynamic` virtual machine instruction to allow direct compilation of dynamically typed languages to the Java VM.

For 2007, my money is on Ruby, although it's not actually my personal favorite. Python code seems to me a lot cleaner and easier to understand than Ruby code, and I think most Java programmers would agree. However, Python came out at the wrong time. Many developers had to make a choice between learning Python and learning Java code, and most chose Java code. Now that they've finally digested the Java syntax and are ready to add another language to their toolbox, they want tomorrow's language, not yesterday's, and that language looks to be Ruby. Most importantly, Ruby has an absolute killer app in Ruby on Rails. Its simplicity is incredibly attractive to the legion of disillusioned Java Enterprise Edition (JEE) developers.

Beyond Rails, the JRuby project offers as good or better integration with existing Java code and libraries than the other scripting languages. In fact, JRuby may well surpass the standard Ruby distribution and become the preferred platform for Ruby programmers, not just Java programmers doing a little Ruby on the side. It's that good. Python programmers will object that they've had the best aspects of JRuby for years with Jython, and they're right, but I'm talking about what *will* happen in 2007, not what *should* happen. It's sad but true: Ruby has the momentum and Python doesn't.

Other scripting languages will increasingly be relegated to the sidelines. Perl is too old-fashioned and doesn't fit modern applications very well. Groovy suffers from a lack of a clear vision and a penchant for choosing computer-science buzzwords over usability and familiarity. BeanShell, Jelly, and probably half a dozen others have never managed to attract more than a niche following. By this time next year, it will be all over but the shouting: Ruby will have become the Java programmer's scripting language of choice.

IDEs are getting better

What had been a rather moribund crop of IDEs really caught fire in 2006, proving once again that competition is good. Embarrassed by Eclipse, Sun poured some energy and resources into NetBeans, which finally started looking like a serious contender. By some measures, NetBeans had even surpassed Eclipse by the end of 2006. It has a far superior native look and feel and much better tools for designing GUIs. What it doesn't have is the Eclipse community. Far more plug-ins and third party products are based on Eclipse than on NetBeans -- at least an order of magnitude more -- and that trend only seems to be accelerating.

As for the year ahead, Eclipse is hard at work on version 3.3, which should be released in 2007. Sun will probably also succeed in getting NetBeans 6 out the door this year. Neither is likely to be a major release: they'll just focus on adding small features here and there, fixing bugs, and cleaning up their user interfaces (though probably not as much they need to).

NetBeans is likely to continue to gain market share at the expense of Eclipse. It's starting from further behind and has more room to grow. (Sun's relentless pushing of NetBeans along with the JDK downloads doesn't hurt either.) By the end of the year, the two IDEs will probably split the market, with each taking half.

Meanwhile, satisfied IntelliJ IDEA users will continue to wonder what all the fuss is about, confident in their belief that it's the best Java IDE available. However, most users won't be able to see beyond its \$500 price tag, and thus its market share will continue to hover at around 5%.

Java Enterprise Edition

No part of Java programming has been so successful and yet so reviled as JEE. It is the technology everyone loves to hate. It is complex, confusing, and heavyweight. No other part of Java programming has seen so many third-party efforts to replace it in whole or in part: Spring, Hibernate, Restlet, aspects, Struts ... and the list goes on. Nonetheless, almost every shop seeking Java programmers is looking for JEE experience, so Sun must be doing something right.

The overarching trend I see in the enterprise space is a desire for simplicity. Massive frameworks are out; small and simple is in. Increasingly, customers are rejecting large parts of the JEE stack, and that's likely to continue. Instead, customers are moving to simpler frameworks like Spring or off the Java platform entirely to Ruby on Rails. The desire for simpler, more understandable systems is also driving the interest in service oriented architectures (SOAs) and Representational State

Transfer (REST).

We can expect the drive to simplicity to continue in 2007. Many people impressed with Rails are trying to duplicate its success in other languages such as Python (Turbo Gears), Groovy (Grails), and Java (Sails). It's possible that one of these will succeed, but otherwise there are no new silver bullets on the horizon. Consequently, businesses will continue loading up the ones they already have: SOA, REST, and Rails.

Java Micro Edition

Moving from the largest to the smallest, what can we expect in the embedded space? The Java platform has been surprisingly successful on small devices over the years, and 2007 will likely build on this success. First up, look for version 3 of the Mobile Information Device Profile (MIDP) to leverage the capabilities of today's more powerful devices. In particular, we should soon be able to run several MIDlets in one VM, including running one or more in the background. Also look for encrypted record management system (RMS) stores and IPv6 support.

Currently in development, the Scalable 2D Vector Graphics API 2.0 for Java ME should expand on the animation capabilities available in many devices. Besides SVG animation, it will also enable audio and video streaming. If the mobile networks ever open up, this could be really important -- think YouTube on a cell phone. (Of course, if the networks don't open up, it's just two-inch corporate ads no one wants to see. I'm pessimistic about this in the United States, but it may be more interesting in Europe.)

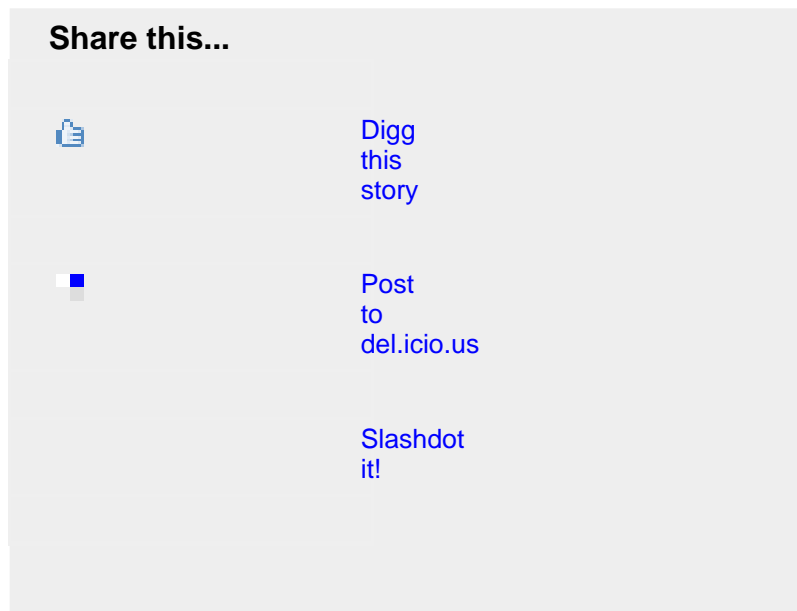
Mobile developers can also look forward this year to the first phones that support the XML API for Java ME. This API is a carefully selected subset of SAX, DOM, StAX, and JAXP designed to fit into the memory-constrained environment of a phone. A lot of people think real XML can't fit into a phone -- this year we'll find out if they're right or wrong.

Even with all that good news, Apple's iPhone still poses a major threat to the Java platform as a mobile phone development platform. The iPhone is already the hottest, sexiest phone on the planet, and it's six months away from release. The problem is it's going to be a relatively closed platform, even by cell phone network standards, and it's not going to run Java code. Needless to say, this is terrible news for anyone trying to sell third-party applications for mobile phones, PDAs, and personal communicators.

In summary

Thanks largely to the open sourcing of the JDK, 2007 promises to be the most exciting year in Java programming since the dot bomb. Up till now, the Java platform has always been limited by Sun's goals and investment capacity, but that is about to change. With the developer community in the driver's seat, expect to see Java programming propelled forward, backward, and sideways, probably all at once. Developers will do more with Java code (and to Java code) than we've ever been able to do before. Desktop, server, and embedded: everything will accelerate. Yes, there will be some spectacular flameouts along the way, but that's part of the fun. The good ideas will survive, the bad ones will fail. If there's anything you don't like about the Java platform or that has always annoyed you, launch your IDE and start hacking.

Ladies and gentlemen, start your compilers.



Resources

Learn

- "[Crossing borders: REST on Rails](#)" (Bruce Tate, developerWorks, August 2006): Explores the application of Ruby on Rails to Web services and introduces the principles of Representational State Transfer (REST).
- "[Managing the Java classpath \(Unix and Mac OS X\)](#)" (Elliote Rusty Harold, developerWorks, December 2006): Tips for mastering this most complex and infuriating part of the Java platform.
- "[Crossing borders: Closures](#)" (Bruce Tate, developerWorks, January 2007): Investigates the importance of closures in Java syntax.
- "[The Cathedral and the Bazaar](#)" (Eric Raymond, 2000): Explains why open source is good for the Java platform and other products.
- [Planet JDK](#): Aggregates the blogs of many people working on the next Java release.
- [Bean Binding](#): Explained by Swing architect Scott Violet.
- [Early ideas about Java 7](#): A video presentation by Sun Microsystems's Danny Coward.
- [The Java Kernel Project](#): Ethan Nicholas explains the inspiration between the newly renamed Java Browser Edition.
- [The java.net scripting project](#): Integrating dynamically typed languages with the Java Virtual Machine.
- [The Java Module System \(JSR-277\)](#): Explained by Specification Lead Stanley Ho.
- [The Swing Application Framework \(JSR-296\)](#): Join the effort to define the future of Swing development.
- [Beans Binding \(JSR-295\)](#): Will enable beans to synchronize with each other automatically.
- [Bean Validation \(JSR-303\)](#): Proposes "a meta-data model and API for JavaBean validation based on annotations, with overrides and extended meta-data through the use of XML validation descriptors."
- [NIO.2](#): An effort to finally define a decent cross-platform filesystem API.
- Throw your ideas into the [Kitchen Sink Project](#) and see what sticks.

Get products and technologies

- [Download Java 7](#): The open source paperwork isn't signed yet, but you can still get a head-start on working with the Java 7 sources.
- [Sails](#): A Rails knock-off written in Java code.

Discuss

- [Participate in developerWorks discussion forums](#).
- [developerWorks community](#): Get involved in the developerWorks community resources, including discussion forums and blogs.

About the author

Elliott Harold

Elliott Rusty Harold is originally from New Orleans, to which he returns periodically in search of a decent bowl of gumbo. However, he resides in the Prospect Heights neighborhood of Brooklyn with his wife Beth and cats Charm (named after the quark) and Marjorie (named after his mother-in-law). He's an adjunct professor of computer science at Polytechnic University, where he teaches Java and object-oriented programming. His [Cafe au Lait](#) Web site has become one of the most popular independent Java sites on the Internet, and his spin-off site, [Cafe con Leche](#), has become one of the most popular XML sites. His most recent book is [Java I/O, 2nd edition](#). He's currently working on the [XOM](#) API for processing XML, the [Jaxen](#) XPath engine, and the [Jester](#) test coverage tool.