Java™ magazine

By and for the Java community

//PREMIERE ISSUE 2011/

7 is Here

The Java platform and ecosystem move forward. Learn how Java SE 7 addresses new trends in programming and hardware architectures.

7

DON'T TOUCH THAT DIAL

Java and Globo TV make interactive television a reality in Brazil

38

RESOURCE INJECTION WITH JAVA EE 6

Master annotations, master configured resources

45

POLYGLOT PROGRAMMER

Dick Wall: What Scala teaches us about the strengths and limits of the JVM

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

COMMUNITY

JAVA IN ACTION

JAVA TECH

COVER ART BY I-HUA CHEN

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java.net
blog

//from the editor /

Welcome to the Premiere issue of *Java Magazine*, which we hope will become an important part of the immense and still growing Java ecosystem.

The *Java Magazine* tagline, “By and for the Java community,” is reflective of its DNA. On the “for” side, the publication is designed to serve the ecosystem in all its diversity: from the hands-on technical craftspeople who make the language dance, to the decision-makers who place very expensive bets on strategic technology platforms, to the learners and newcomers who are just getting a handle on why This Java Thing is so great. People in all those categories will find something to like here.

Just as important, on the “by” side, experts from across the globe will be pitching in. In this issue, Java Champions Adam Bien, Michael Kölling, Kevin Nilson, and Dick Wall have made contributions, and we’ll see participation by other community figures in future issues. Maybe you’ll be one of them. (If you’re interested, drop us a line.)

The whole thing is delivered in a highly interactive package, designed from the ground up to take full advantage of its digital format. (If you were wondering if this project was a fun one for us, you’d be right.)

If there’s anything I can leave you with, it’s this: *Java Magazine* is a work-in-progress, and we need your help to make it better—whether from an editorial or a design standpoint. So, explore this issue. Take your time with it. And when you’re ready, send us a message with your thoughts.

We hope you enjoy reading *Java Magazine* just as much as we enjoyed making it.

Justin Kestelyn, Editor in Chief BIO

//send us your feedback /

We’ll review all suggestions for future improvements. Depending on volume, some messages may not get a direct reply.

PHOTOGRAPH BY RICHARD MERCHÁN

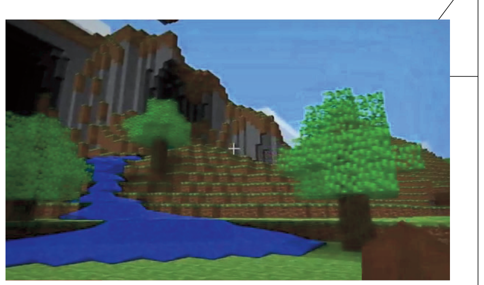
Java 7 Goes Global

On July 7, Oracle hosted a global celebration of the imminent availability of Java Platform, Standard Edition 7 (Java SE 7 or Java 7). Community members were present on three stages around the globe simultaneously for this event: at Oracle headquarters in **Redwood Shores**, California; in **London**, England; and in **São Paulo**, Brazil. Participants included representatives from Accenture, HP, IBM, the London Java Community, Royal Bank of Scotland, Riot Games, SouJava, and Travelex—all of whom shared their reasons why Java is critical to their success.

The Redwood Shores event included a general session with Adam Messinger, vice president of development, Oracle Fusion Middleware. This was followed by technical breakouts about key Java 7 improvements including language enhancements (JSR-334: Project Coin), the new lightweight fork/join framework, and support for dynamically typed languages (JSR-292: InvokeDynamic).

More than 200 Java 7 "tech session" kits were also distributed to Java user groups in 59 countries for their meetings. The world is now ready for Java 7!

PHOTOGRAPH BY ENRIQUE AGUIRRE



Get a look inside Minecraft, an award-winning game based on Java.

JAVA-BASED GAME WINS BIG

Congratulations to Swedish game developer Mojang for winning five awards for its Java-based game Minecraft. The Game Developers Choice Awards recognized Minecraft with the Innovation Award, Best Downloadable Game Award, and Best Debut Game Award. The Independent Games Festival awarded Minecraft the Audience Award and the Seumas McNally Grand Prize. Watch the fan-made trailer to see what the game is all about.

EVENTS

JAVAONE *OCTOBER 2–6, SAN FRANCISCO, CALIFORNIA*

The dates have been set, the conference tracks have been identified, the papers have been submitted, the final sessions list has been selected, and the entertainment has been announced (Sting, and Tom Petty and the Heartbreakers). Only one thing is missing: your JavaOne 2011 registration.

Head to San Francisco in October to focus on the latest Java technologies. Visit the JavaOne conference site for all the details. You can follow the events leading up to JavaOne 2011 at the conference site, and via the JavaOne Conference blog, the JavaOne Twitter feed, JavaOne on Facebook, JavaOne on LinkedIn, the JavaOne Oracle Mix group, and Java.net.

AUGUST

PPPJ 2011
AUGUST 24–26, KONGENS LYNGBY, DENMARK
The 9th International Conference on the Principles and Practice of Programming in Java brings together researchers, teachers, practitioners, and programmers who study or work with the Java language or its virtual machine.

Research Triangle Software Symposium
AUGUST 27–29, RALEIGH, NORTH CAROLINA
Hear about the latest technologies and best practices emerging in the enterprise software development space.

PHOTOGRAPH BY HARTMANN STUDIOS

SEPTEMBER

JavaZone
SEPTEMBER 7–8, OSLO, NORWAY
JavaZone is the biggest meeting place for software developers in Scandinavia and a forum for knowledge exchange among IT professionals.

QCon
SEPTEMBER 10–11, SÃO PAULO, BRAZIL
This international software development conference includes a track about the various ways the Java platform is being used independent of the Java language.

OCTOBER

Silicon Valley Code Camp
OCTOBER 8–9, LOS ALTOS HILLS, CALIFORNIA
At this community event, developers learn from each other. All are welcome.

GOTO
OCTOBER 10–12, AARHUS, DENMARK
GOTO (formerly JAOO) is an educational and networking forum for software developers, IT architects, and project managers.

BlackBerry DevCon Americas
OCTOBER 18–20, SAN FRANCISCO, CALIFORNIA
This developer conference showcases the latest innovations with the BlackBerry development platform.

Bruno Souza

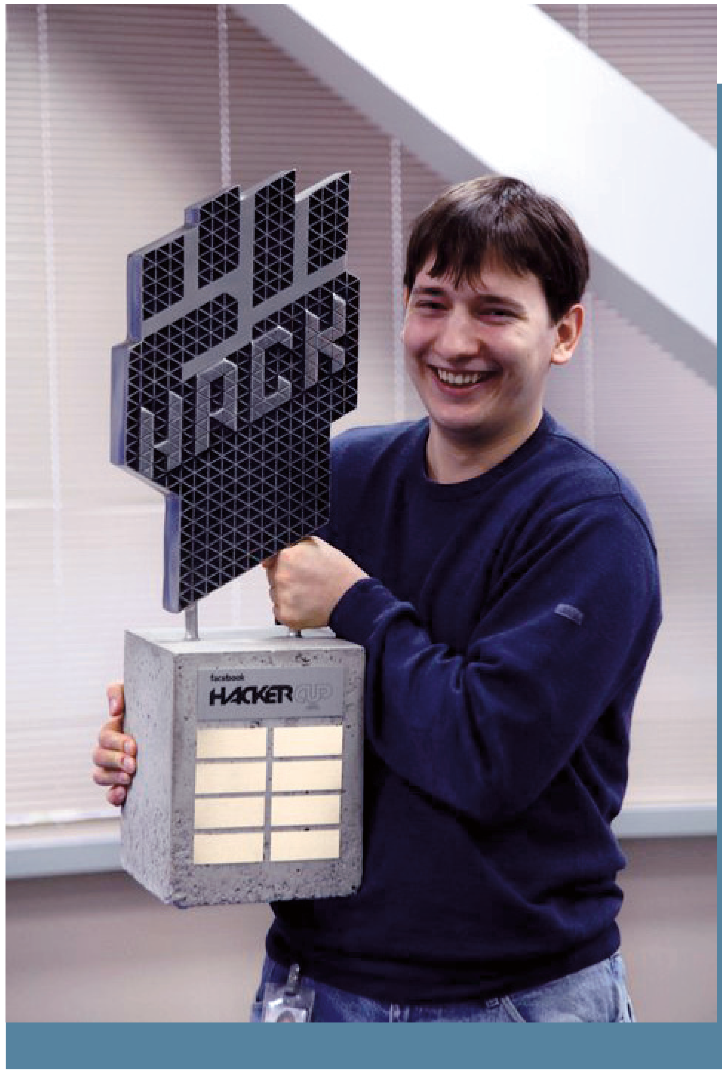
Ben Evans

Martijn Verburg

JAVA USER GROUPS ON THE JCP EXECUTIVE COMMITTEE

A promising new development for the Java Community Process (JCP) Executive Committee is the recent election of SouJava and the London Java Community as new members of the Standard/Enterprise Edition Executive Committee. The 2011 Executive Committee **Special Elections** placed **SouJava**, the Brazilian Java Users Society, into a ratified Standard Edition/Enterprise Edition seat, with representation by SouJava President **BRUNO SOUZA**. Meanwhile, the **London Java Community (LJC)** won an open seat on the same committee, with representation by **BEN EVANS**. LJC co-leader **MARTIJN VERBURG** noted, "We are humbled by the trust that the JCP members have given us and alongside the Brazilian JUG [Java user group] . . . we look forward to representing the millions of Java developers and users around the world." The presence of JUGs on the JCP Executive Committee has been widely applauded and represents an important step toward greater openness and transparency in the JCP. In other election news, Goldman Sachs won the other ratified seat on the Standard/Enterprise Edition Executive Committee and will be represented by John Weir. The open seat on the Micro Edition Executive Committee went to Alex Terrazas.

BEN EVANS' PHOTOGRAPH BY CATHERINE CURRIE



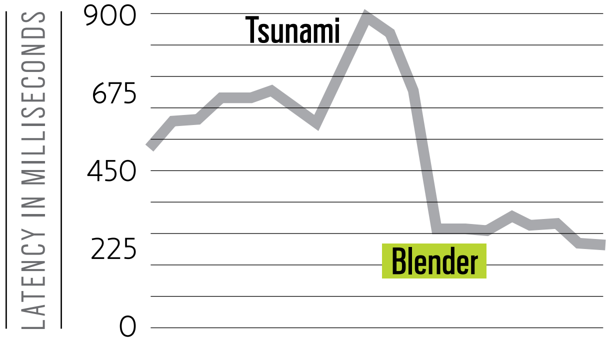
PHOTOGRAPH BY MATT HARNACK

First-Ever Facebook Hacker Cup World Champion Uses Java

Russian developer **Petr Mitrichev**, who works in Java, was named the first Facebook Hacker Cup World Champion. The 25 finalists had to solve three algorithmic problems—Party Time, Safest Place, and Alien Game—in the fastest possible time (under two hours). Only three finalists submitted answers to all three problems, and Mitrichev was the only one to get three correct answers. Read more about Mitrichev and see the three problems here.

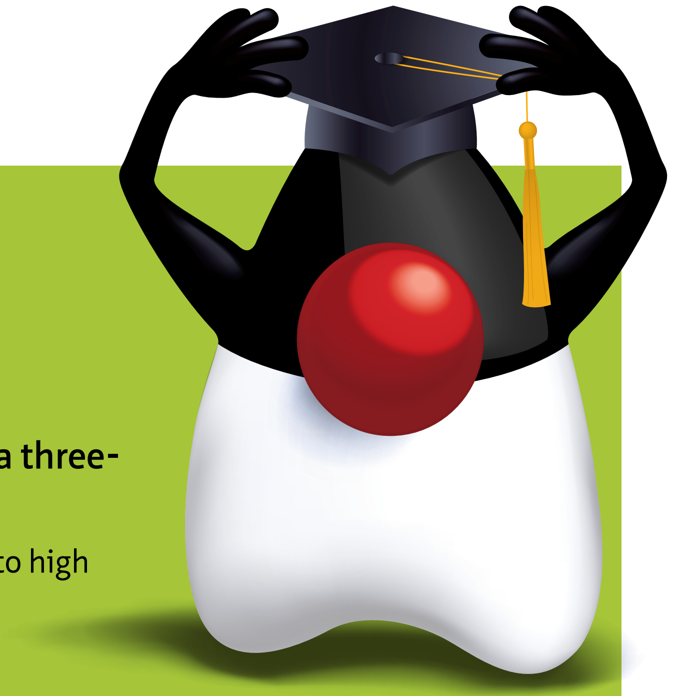
TWITTER SEARCH IS NOW THREE TIMES FASTER THANKS TO JAVA!

In an attempt to reduce the number of times Twitter users see the "fail whale," Twitter has moved from a Ruby on Rails front end to a Java server they call Blender. Get the details straight from Twitter.



95th Percentile Search API Latencies Before and After Blender Launch

LEARN JAVA BASICS AND TRICKS



With the Java 7 release, academic events are front and center, including a three-day workshop in August and weeklong activities at JavaOne 2011.

Java Summer Workshop, August 10–12. Oracle is offering a free three-day workshop to high school students and teachers in California's San Francisco Bay Area on programming with Alice and Greenfoot. Attendees will learn how to create animations and games and get an introduction to the Java programming language using 3-D Alice software and 2-D Greenfoot software. Tutorials will be available online after the event. The event takes place at the Oracle Conference Center in Redwood Shores, California.

JavaOne, October 2–6. With the affordable Discover Pass, teachers and qualifying students can benefit from the industry-leading Java conference. Students and teachers can attend programming sessions designed for people with some or no programming experience. Attendees with no programming experience will discover tools and projects related to programming. Those with programming experience will learn tips and tricks for programming complex logic and expand their knowledge of programming projects.

Other learning opportunities include JavaOne keynotes, which offer the latest news on Java technology, and the JavaOne exhibit hall, with live, hands-on demonstrations and discussions of the latest Java software technologies. Students and teachers can also network with top industry programmers at the Oracle User Groups Pavilion and the OTN Lounge. Register today.

ART BY I-HUA CHEN

JCP Chairman Patrick Curran chats with *Java Magazine* Editor in Chief Justin Kestelyn.

Defining the Future of the Java Community Process

PATRICK CURRAN, chairman of the Java Community Process (JCP), announced the first of two new Java Specification Requests (JSRs) that will define the future of the JCP. The new "JCP.next" JSRs will specify changes to the JCP Process Document, which defines the formal procedures for using the Java Specification development process. The first JCP.next JSR, JSR-348, proposes a variety of changes and adjustments to the Java Community Process, with the objective of improving transparency, participation, agility, and governance in the JCP. The second JSR will address more-complex issues. Follow progress at the JCP.next Resources page, or participate in the JCP.next project on Java.net.

Java.net Expanding Project Hosting Capability

2011 has been a year of rapid expansion in the toolset Java.net offers to open source projects. First, in late February Java.net's project infrastructure was migrated to Kenai. Since then, the Java.net/Kenai infrastructure has undergone successive enhancements, providing greater choice and power to open source project leaders and developers.

Java.net's offerings now include the Subversion and Git version control systems, JIRA issue and project tracking, and Sonatype's Nexus Maven Repository Manager service. More than 2,000 active projects are currently utilizing the upgraded Java.net project infrastructure. Visit the Java.net Create a Project page if you'd like to join them.

African Developers Get Organized

Something is brewing in Africa: developers are organizing themselves into groups that span the continent. In 2010, Jean-François (Max) Bonbhel founded JUG-AFRICA as an umbrella group for African Java user groups, with the objective of facilitating collaboration. With 5,000 members in 14 countries, JUG-AFRICA sponsors the biggest Java community event in Central Africa, JCertif (September 3–4).

A new group, Coders4Africa, held its first major event, Coders4Africa 2011 Ghana, June 18–19 in Accra, Ghana, with Oracle's sponsorship. Cofounder Amadou Daffe says the group's objectives include "developing a community of African developers across Africa under one structured roof."

Both groups provide organizational structures that address the unique challenges that African developers face and exemplify the collaborative spirit that's spreading across Africa.

JCertif conference participants

PHOTOGRAPH BY JCERTIF 2010

OpenJDK Becomes Official Java SE 7 Reference Implementation

In a recent blog post, Java SE Product Manager Henrik Ståhl announced that Oracle will create Java SE 7 Reference Implementation binaries based only on the OpenJDK, will make those binaries available under the Binary Code License for commercial implementers and General Public License v2 (with the Classpath exception) for open source implementers, and will update the OpenJDK TCK License Agreement (OCTLA) so that it covers Java SE 7. With these changes, open source implementers will have access to Java SE RI source code that allows more-direct comparisons to their implementations for verifying compatibility, as well as to the TCK on a free-as-in-beer basis.

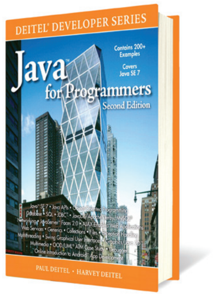
JAVA BOOKS

THE WELL-GROUNDED JAVA DEVELOPER—JAVA 7 AND POLYGLOT PROGRAMMING ON THE JVM

By Benjamin J. Evans and Martijn Verburg
Manning Publications

The Well-Grounded Java Developer is a unique guide written for developers with a solid grasp of Java fundamentals. It provides a fresh, practical look at new Java 7 features, along with the array of ancillary technologies that a working developer will use in building the next generation of business software.

Following its thorough coverage of new Java 7 features, the book explores a cross section of emerging Java virtual machine (JVM)–based languages, including Groovy, Scala, and Clojure. You'll find dozens of valuable development techniques showcasing modern approaches to concurrency and performance.



JAVA FOR PROGRAMMERS, 2ND EDITION

By Paul Deitel and Harvey Deitel
Pearson/Prentice Hall Professional

Written for programmers with a background in high-level language programming, this book applies the Deitel signature live-code approach to teaching programming and explores the Java language and Java APIs in depth. The book presents concepts in the context of fully tested programs, complete with syntax shading, code highlighting, line-by-line code walkthroughs, and program outputs. The book features more than 200 complete Java programs with more than 18,000 lines of proven Java code, as well as hundreds of tips that will help readers build robust applications.

Read a sample chapter, "Object-Oriented Programming: Polymorphism"

Globo TV brings new content and capabilities to Brazilian viewers.

Interactive TV Takes Off with Java

BY DAVID BAUM

It's just another day at home as you tune in to your favorite interactive TV show. Helena (Taís Araújo), the main character on the popular Brazilian telenovela *Viver a Vida*, is on location in Buzios, a beach village in the state of Rio de Janeiro. As Helena strolls down the waterfront, the camera zooms in and a small, nonintrusive alert appears at the bottom of your TV screen—an invitation to access further information about Helena's personal history and how she fits into the plot. By the time the show airs again tomorrow, you will have become a veritable expert on the many intrica-

PHOTOGRAPHY BY PAULO FRIDMAN

Carlos Fini, Engineering Manager, Globo TV

Carlos Fini, an engineering manager at Globo TV, oversees operations in the control room. With Java, Globo TV is taking viewers from a passive to an active television experience.

SNAPSHOT

GLOBO TV
globotvinternational.com

Location:
Rio de Janeiro, Brazil

Industry:
Media and entertainment

Employees:
8,500

cies of *Viver a Vida*'s dramatic story line.

Thanks to Java, Brazilian television is no longer a passive medium. The interactive digital TV services offered by Globo TV, Brazil's largest broadcaster, let viewers control the audio/video experience, including participating in polls; responding to ads; monitoring sports statistics; downloading clips; participating in quizzes; playing games; and customizing how they receive weather reports, traffic information, and local news broadcasts. Java apps also enable TV viewers to send e-mail, review player profiles, verify bank balances, and purchase products and services during TV episodes.

"We are providing viewers with better mobile coverage, new interactive capabilities, and a rich media experience," says Carlos Fini, an engineering manager at Globo TV. "Java is open source, royalty free, and supported by a large and active developer community. It was the perfect choice for our needs."

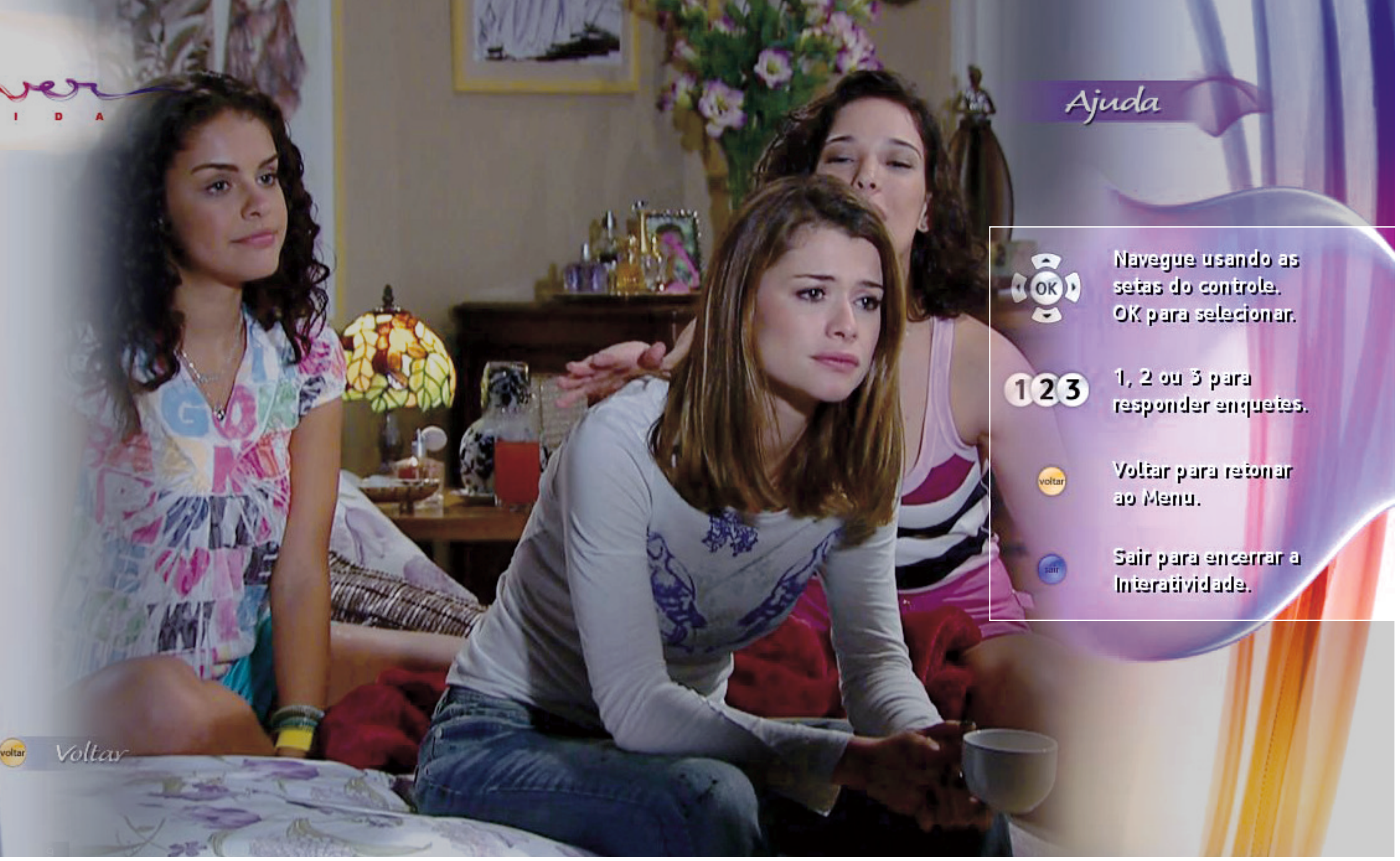
Globo TV is the largest and most influential broadcaster in Brazil and also exports content to 171 countries. The majority of the Brazilian population enjoys free-to-air television, and Globo TV has a 70 percent share of this over-the-air (OTA) market. Fini has spent 15 years leading a technical team at Globo TV. His Java programming staff is currently engaged in creating software applications for delivering new types of interactive TV content. Viewers simply push the *i* button on their remotes to access these interactive features, launching Java applets that have been transmitted to their set-top boxes along with the digital media broadcast.

"Java is supported by many entertainment devices, which means our interactive TV programs can be delivered using the same technology received by gaming systems, mobile phones, set-top boxes, and Blu-ray players," Fini adds. "In conjunction with the Ginga middleware standard, Java provides a new way to send content to TV viewers and receive feedback from those viewers."

JAVA CONSUMPTION
Consumers have purchased 3 billion Java-enabled devices, from set-top boxes to embedded high-definition tuners, for at-home use.

THE PROMISE OF DIGITAL TELEVISION

Many countries are replacing OTA analog television with digital television to maximize the uses of the radio spectrum.



Navegue usando as setas do controle. OK para selecionar.

1, 2 ou 3 para responder enquetes.

Voltar para retonar ao Menu.

Sair para encerrar a Interatividade.

Viewers of the Brazilian soap opera *Viver a Vida* can push the *i* button on their remote controls to interact with the program—in this case, to call up a synopsis of the previous episode (at right).

In the U.S., digital TV has become popular mainly due to its ability to deliver superior picture and sound quality. In Brazil, digital TV is also interactive, enabling viewers to customize and control the viewing experience—whether by voting for their favorite player during a World Cup Soccer match or changing the viewing angle during a reality TV show.

Indeed, it was the pending 2010 FIFA World Cup championship that motivated Globo TV to finalize this interactive environment. The company's content developers and programming experts knew that Brazil's millions of passionate soccer fans would enjoy being able to choose how and when to view tournament stats, player profiles, and synopses of previous matches. In addition, Globo TV management knew that it could offer a very interesting business model.

A digital signal not only requires less bandwidth but also permits rich content and interactive capabilities for shopping, voting, gaming, and other new-media functions. Broadcasters can transmit Ginga applications within their production signals to augment the audio/video stream, control media playback, control the display hardware and tuning functions, launch other applications, and overlay supplemental media content on the standard broadcast signal.

"We needed a very flexible development environment to compete with other new-media services," says Fini. "Additionally, we wanted to make the user experience as compelling as possible to captivate viewers and build the audience. Java is open and works on many different platforms."

Best of all, Java is ubiquitous. Like most countries, Brazil has a large community of Java developers, giving Globo TV a large and reliable pool of expertise for its development projects. Java is pervasive not only in the computer industry—with more than 840 million Java desktops—but in the home entertainment industry, where consumers have already purchased 3 billion Java-enabled devices, from set-top boxes to embedded high-definition tuners. Java is also quickly gaining ground in the mobile computing world, with 2.6 billion Java-enabled phones in circulation—approximately 85 percent of all mobile phones worldwide. In the broadcasting industry, about 180 operators are deploying Java content.

MIDDLEWARE FOR BROADCASTERS AND MANUFACTURERS

Sun created the Java DTV specification in partnership with the SBTVD (Sistema Brasileiro de Televisão Digital) Forum, a Brazilian digital TV forum founded in 2006 to guide standards and technical specifications for the transmission and reception of digital TV in Brazil. The objective of the group was to develop and implement a digital TV standard that not only addresses technical and economic issues but also helps promote an "information society" that brings government closer to the population. This is important in a country where 96 percent of households have a TV set and less than 20 percent have a computer.

"We believed that Java was the more appropriate tool available to create new solutions for interactive TV services," notes Fini, who was one of the founding members of the SBTVD Forum.

One of the significant decisions of the Forum was to adopt Ginga as the country's middleware standard. Ginga simplifies the creation of digital TV applications by providing a high-level open development environment and libraries of standard functions. As an open specification, Ginga is easy to learn and free of royalties, encouraging widespread use by content producers. More and more manufacturers are including it in their TVs, set-top boxes, and peripheral devices. By enabling a variety of e-commerce

BIG BUSINESS
Globo TV exports content to 171 countries.

applications, Ginga applications ultimately lower the cost of television receivers and set-top boxes for consumers.

Ginga includes two primary programming paradigms: Ginga-NCL and Ginga-J. Ginga-NCL is a multimedia presentation environment for declarative applications. Developers use this XML-based language to synchronize media objects, control media content, and present interactive programs to consumer devices. Ginga-J provides an execution infrastructure for Java applications.

"Ginga-J extends Java DTV for the Brazilian system with specific APIs for interacting with consumer devices and managing an asynchronous messaging environment," explains Fini.

JAVA FACT

In the mobile computing world, there are 2.6 billion Java-enabled phones in circulation—approximately 85 percent of all mobile phones worldwide.

Ginga-J was approved by the Forum in April 2009 along with the Java DTV API. These specifications (known respectively as *transmission*, *reception*, and *code-back*) enable Java developers to create apps that can receive content from broadcasters, read this content, and send content to subscribers. Oracle's royalty-free specification has since become the digital terrestrial TV standard in Brazil and most other South American countries.

The Forum also adopted the ISDB-T standard, which was originally popularized in Japan and is now pervasive across most of South America. ISDB-Tb (the *b* indicates the Brazilian version of the standard) had an edge especially because of its *one-seg* feature, which allows broadcasters to embed a free-for-all low-resolution signal for mobile phones and tablets, as well as standard-definition and high-definition TV broadcasts. One-seg has proven to be a very enticing feature for its ability to democratize access to news and information in developing countries.

THE NUTS AND BOLTS OF INTERACTIVE TV

Today 10 developers on Fini's team are bolstering Globo TV's lineup with a wide array of interactive TV apps and content. "Everything is brand new, so the team is pioneering new models," he says. The developers use Oracle's Lightweight UI Toolkit (LWUIT), a UI library targeted for mass-market mobile devices that shields developers from the need to write device-specific code for different screen sizes. This is important because Globo TV simultaneously broadcasts content for large-format televisions and tiny mobile devices.

Globo TV's digital TV applications take the form of *PBP Xlets*—self-contained Java apps that are packaged into signed JAR files and transmitted as part of the broadcast stream. Embedded Xlets allow the broadcaster to synchronize the application functions with the TV program, permitting viewers to start, stop, and pause the signal as well as to control interactive TV functions such as shopping, voting, and downloading content.

"The use of Java technology allows for rich media and interactive applications combined with the regular TV broadcast such as program-related information, games, voting, targeted advertising, and e-government," says Fini. "Platform architects, application developers, and media content authors are actively developing these applications."

The Java DTV specification permits multiple applications to be executed simultaneously yet run autonomously, shielded from each other in a persistent file system. Applications, sounds, images, and other information can be embedded into the transport stream, which originates on the broadcast carousel and is transmitted to the set-top box. A Java Media Framework player handles time-based media streams, typically associated with a media decoder.

Media pundits acknowledge the bright potential for interactive TV in Brazil. By starting with a complete, royalty-free Ginga specification, supplemented by Java, Brazil is avoiding the gridlock that has plagued interactive TV in the U.S. and elsewhere.

"In 2011 the Brazilian market will reach between 17 million and 20 million DTV consumers, and part of them are using our interactive TV service," concludes Fini. "Using a standard language ensures compatibility, and Java is rich and robust." ●

Resources for Java Developers

Downloads

Java DTV API 1.0 Specification

Forums

LWUIT Open Source Community

Java TV Forums Home

Java Embedded Forums Home

Data Sheets and White Papers

Java Technology for Digital Media

Java Technologies for Interactive Television

JavaTV API Technical Overview

Based in Santa Barbara, California, **David Baum** writes about innovative businesses, emerging technologies, and compelling lifestyles.

Colin Renouf, Enterprise Solutions Architect, Travelex

Sold on Java

Java provides a global IT platform for foreign exchange leader **Travelex Group.** **BY PHILIP J. GILL**

PHOTOGRAPHY BY JOHN BLYTHE
AND GETTY IMAGES

The dollar, the euro, the pound, and the yen may dominate international currency transactions, trade, and travel, but they are far from the only currencies that businesses and tourists need to concern themselves with today. In this era of globalization, little-known currencies from far-off countries are becoming increasingly important.

Consider the national currency of Angola, the *kwanza*. Just a few years back, few thought they would need to exchange their dollars or euros for the kwanza (named for the country's largest

SNAPSHOT

TRAVELEX GROUP
www.travelex.com

Headquarters:
London, England

Industry:
Foreign exchange

Revenue:
More than US$1.1 billion in FY 2010

Employees:
More than 7,000

Java version used:
Java Platform, Enterprise Edition 6 (Java EE 6)

Portability, scalability, and internationalization are key reasons why Travelex is banking on Java, says Travelex Enterprise Solutions Architect Colin Renouf.

river). But today many do, because the southern African republic is an emerging business and tourist destination.

Following the daily fluctuations of the kwanza—and more than 80 other world currencies—and navigating the many national finance laws that guide international currency transactions is the business of London, England–based Travelex Group, the world's leading specialist provider of foreign exchange and international payments.

"We are in the foreign exchange business," explains Travelex Enterprise Solutions Architect Colin Renouf, "but I also like to think of us as being in the data exchange business because of our expertise in currencies."

In the past, the IT systems that supported and enabled Travelex' many regional operations around the world were built on Windows and .NET technologies. However, as operations have expanded around the globe, the Travelex senior management team has come to realize that those systems do not provide the kind of standards-based, portable, and scalable platform the group needs to support its growth, especially in emerging economies. "The difficulty with Windows is that it's proprietary and it's difficult to integrate," says Renouf.

To support current and future growth, senior management realized that the company needed to trade in those old systems for a new platform. Today Travelex is sold on Java. "All current global enterprise systems under development are in Java," says Renouf. "Going forward, the majority of global enterprise systems in development will be in Java."

FROM 1 TO 1,000

Travelex was founded in 1976 as a single exchange shop in London. Today its consumer-focused operations provide cash and prepaid cards to more than 30 million retail customers each year, through a network of close to 1,000 stores and more than 500 ATMs spread across 24 countries. Travelex Global Business Payments (TGBP), the group's business-focused operation, provides international business payments to more than 35,000 businesses. On July 5, 2011, Travelex announced that it had agreed to the sale of TGBP to Western Union for £606 million, providing significant capital for further investment as the group looks to accelerate its growth in both existing and new markets, as well as provide customers with access to innovative foreign exchange products and services across the globe. This transaction is likely to close toward the end of 2011. Until then, TGBP will remain an important business for Travelex.

Partly due to the company's growth through mergers and acquisitions, Travelex' regional IT systems operated independently of each other. Often whatever systems were in place at the time of an acquisition tended to stay in place.

JAVA FACT
Travelex is the world's largest nonbank provider of international currency payments.

That began to change when Travelex' current CIO, Steve Grigg, came on board in spring 2008. A financial industry veteran, Grigg knew firsthand the difficulties finance organizations face in building and integrating enterprise systems, as well as the benefits of getting all operating units on the same IT page. Grigg's mission, explains Renouf, was to bring the regions together by building new systems on a single platform that Travelex could use to develop, integrate, and deploy in many regions around the globe.

That platform is Java, which provides a rich, standards-based, portable, scalable, reliable, and fully internationalized platform for application development. "Java is a very rich development environment, and you can use it for any application," says Renouf, whom Grigg recruited. "But the richness of Java comes more into its own in enterprise and B2B [business-to-business] systems," Renouf continues, "because the facilities for integration that different open source and EJB [Enterprise JavaBeans] technologies offer is coupled with the richness of the Web and internationalization functionality. This allows us to take our organically grown local systems and federate them to our regional operations around the world.

At the same time, Renouf adds, the interfaces and mechanisms Travelex uses allow the company to gradually align the architecture across its many different regions, "while still changing the overall estate to give richer functionality and more-agile delivery."

One of Travelex' first new Java systems has been its Global Payments Gateway (GPG), which was developed, tested, and deployed in less than nine months. GPG comprises a "complete Java and Oracle stack," says Renouf, including Oracle WebLogic Server 11g, Oracle Database 11*g*, and other Oracle products. Besides processing international currency transactions for Travelex' many business and financial institution clients, GPG also connects those clients to the many Society for Worldwide Interbank Financial Telecommunications (SWIFT) financial messaging networks. SWIFT links more than 9,000 financial institutions in 200 countries and territories.

JAVA FACT

Travelex' Java-based Global Payments Gateway can scale from a few transactions per minute to between 400 and 500 transactions per second.

GPG allows Travelex' clients to send payments in any currency to anywhere, from anywhere. "If a customer wants to send money to somebody in Kuala Lumpur, they can just select the country, the currency, and how much they want to send—then enter their bank account and it's done," says Renouf. In addition, he says, a new Web-based feature will allow individuals to send money from their home and business computers.

Portability, scalability, and internationalization are key reasons why GPG is written in Java. In some locations or regions, GPG can scale up to between 400 and 500 transactions per second, while in other locations it will only need to process a few transactions per minute, says Renouf.

But the real point of Java, he explains, is that it provides a common architecture that is the same everywhere, which Travelex can easily integrate from development to deployment. "That way we can do the same thing in different regions," he says.

Java's internationalization features enable Travelex to support dozens of local languages in the front end of their systems, as well as the business logic that runs behind them. In this way, it's now easier to deploy features developed for the U.S. market, for example, in places like China and Japan.

One area in which Travelex has done a great deal of work on its own is reliability, says Renouf. Because of government regulations around the world, Travelex needs to understand how, why, and where its systems might fail and how to recover. Java makes this considerably easier than other programming environments, says Renouf. "Java provides an awful lot of tools out of the box to enable you to understand exactly what's going on, though most people don't exploit them," he says. "We build on what we learn from those tools to diagram and document behavior under exceptional conditions and to work out how to develop more-reliable and secure systems." ●

Philip J. Gill is a freelance writer and editor based in San Diego, California.

More than a Technology

For Colin Renouf, enterprise solutions architect at Travelex Group, Java is more than a technology platform for developing enterprise systems whose virtues of portability, scalability, programmability, and reliability are unmatched. To Renouf, one of the most important assets of Java is the Java community.

As with any vibrant community, individual participation is vital for the Java community's health and growth. Travelex, for instance, was involved in the security around internationalization of the Java platform—looking at some of the conversions in different layers as a result of double-byte character sets, for example, and how to ensure that, as Renouf puts it, "what hits the Java layer is what it should be." Some of this work has been provided back to Oracle and other partners, and some has gone back to the community as a whole.

Travelex is also involved in educational efforts. "A number of people on our team are very active in the Java community," says Renouf. "Three of us have written books and for magazines, worked on standards, run community events, and on occasion have even done work for some of our customers or have externally supported big industry events." Renouf adds that he and his colleagues encourage others to participate as well. "It's a great way to learn and get collaboration on meeting some of your requirements, and it brings a level of satisfaction in knowing that you have 'made a difference,'" he says, concluding, "Power belongs to those who participate."

MICHAEL KÖLLING

BIO

PHOTOGRAPH BY JOHN BLYTHE

Getting Your Feet Wet

Use Greenfoot to create Java classes, objects, and methods.

In previous articles I've written, for example, "Wombat Object Basics" and "Wombat Classes Basics," I covered important basic concepts of object orientation. The concepts were explored mostly by investigating existing source code and by discussing theory. Programming, however, is about doing. So in this article, we'll jump right in and start to make our own class. In the process, you will learn about objects, methods, and parameters.

Prerequisites

To follow along, you will need the following software installed on your computer:

- Java Platform, Standard Edition (Java SE)
- Greenfoot version 2.1.0 or newer

Ensure that you have a new enough version of Greenfoot. You can check your version using the **About Greenfoot** menu item.

Creating Your Own Project

In my previous articles, I worked with an existing project (wombats). This time, let's create our own project.

Greenfoot calls its projects *scenarios*. So the first thing we'll do is to create a new scenario.

1. Start Greenfoot.
2. Greenfoot usually opens the last scenario you worked on. Close this by selecting the **Close** function from the **Scenario** menu.
3. Select **New** from the **Scenario** menu.
4. In the New Scenario dialog box, name the new scenario turtle and click **OK**.

You should now have a new scenario window that is mainly gray, as shown in **Figure 1**.

We now have a shell for a new scenario, a place where we can start to work. We do not have a world yet to make things happen in, and we do not have any actors that can do things.

The two classes you see on the right side of the window, World and Actor, are *superclasses*. They are not a specific world or a specific actor, but rather descriptions of all possible worlds and actors. We will create a specific world and actors by creating *subclasses* of these. Subclasses represent specializations of the superclasses.

Let's start by making a world.

Creating a New World

1. Right-click the **World** class (the beige box) and select the **New Subclass** function.

A dialog box asks you for a name and an image for your new class (see **Figure 2**).

1. Name your class TurtleWorld, and select an image from the **Backgrounds** category of the image library.

For the screenshots in this article, I selected a background image called weave.jpg.

2. Click **OK** and then click the

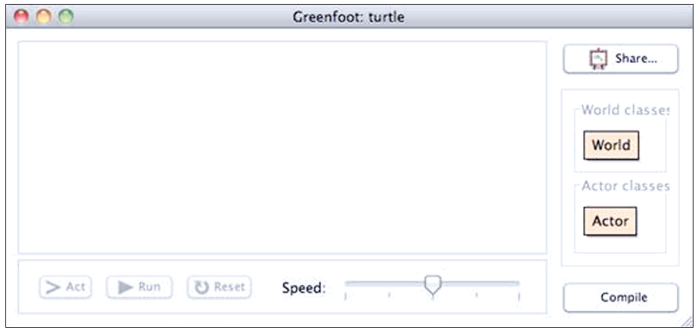
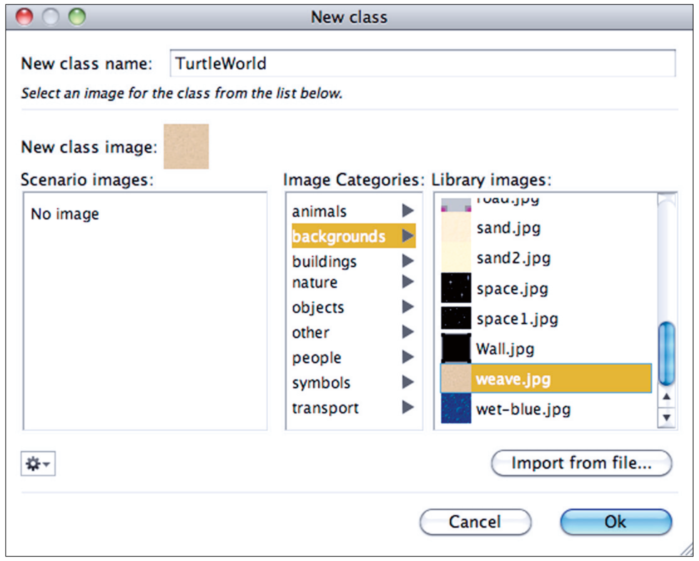


Figure 1


New class

New class name: TurtleWorld

Select an image for the class from the list below.

New class image:

Scenario images: No image

Image Categories: animals, backgrounds, buildings, nature, objects, other, people, symbols, transport

Library images: road.jpg, sand.jpg, sand2.jpg, space.jpg, space1.jpg, Wall.jpg, weave.jpg, wet-blue.jpg

Figure 2

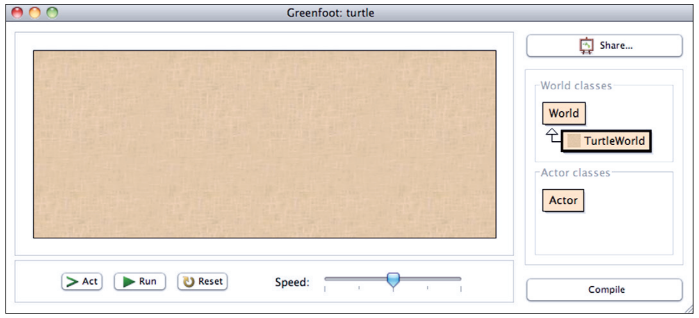


Figure 3

Compile button in the main Greenfoot window.

You should see a world appear in your Greenfoot window with your selected background (see **Figure 3**).

Now that we have created a world, we are ready to create actors to put into it.

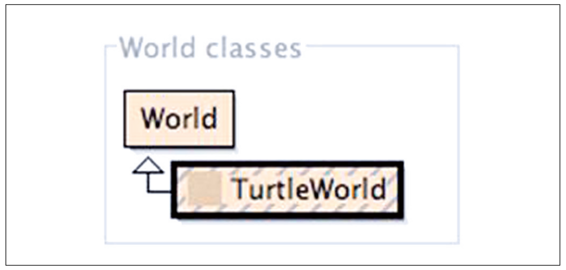


Figure 4

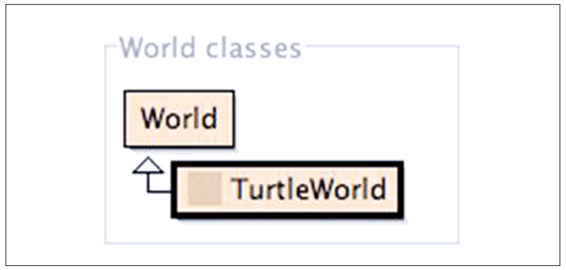
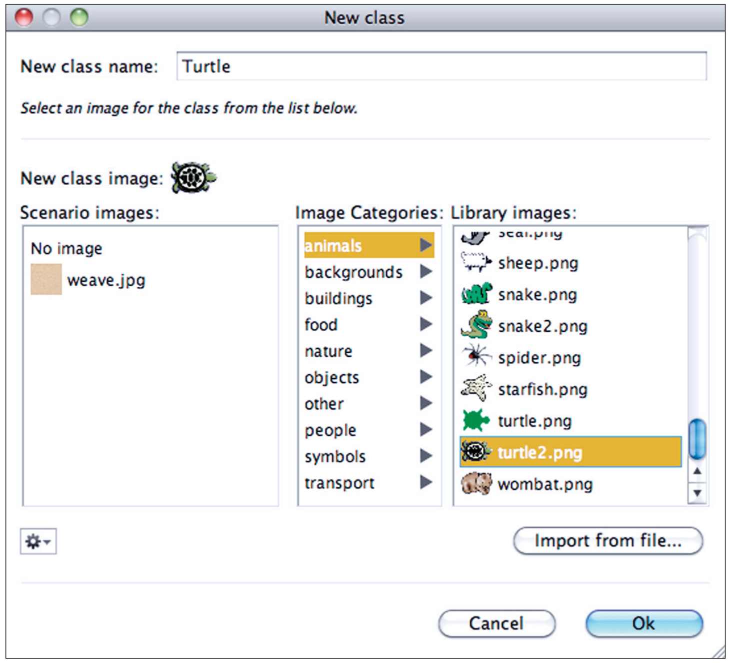


Figure 5



New class name: Turtle

Select an image for the class from the list below.

New class image:

Figure 6

A Side Note About Compiling

When we program, we write Java code. Unfortunately, your computer cannot execute Java code directly. It needs its instructions in a specialized machine language.

Fortunately, there is an easy solution: the compiler. A compiler is software that can translate Java code into machine language. Every time you create a new class or you make a change to the source code of an existing class, the class needs to be translated to machine language again. You can do this easily just by clicking the **Compile** button.

In Greenfoot, a class appears striped if it has not been compiled since the last change (see **Figure 4**).

Once you click the **Compile** button and the class is translated, the stripes disappear (see **Figure 5**), and the class is ready to be used.

Creating an Actor

We now have a world that our actors can live in. However, we do not have an actor to do anything yet. So let's make an actor.

1. Right-click the **Actor** class and select the **New subclass** function.
2. In the resulting dialog box, name the new class Turtle, and select the **turtle2.png** image from the **Animals** category (see **Figure 6**).
3. Click **OK**.

You now have a brand-new Turtle class in your scenario.

LISTING 1

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class Turtle here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Turtle extends Actor
{
  /**
   * Act - do whatever the Turtle wants to do. This method is
   * called whenever
   * the 'Act' or 'Run' button gets pressed in the environment.
   */
  public void act()
  {
    // Add your action code here.
  }
}
```

See listing as text

Note that this class is initially striped (that is, uncompiled).

4. Click the **Compile** button to compile the class.

Note that by convention, Java class names always start with a capital letter.

Creating Objects

We now have an actor class (Turtle), but we do not have actor *objects* yet.

Right-click the **Turtle** class and select **new Turtle()**.This creates a turtle object—that is, an actual actor—which you can then place into the world.

That's it. You just created your first turtle. Note that you can create as many turtles as you like. Try it out.

Programming Your Objects

1. Click the **Run** button.

In my previous articles, clicking **Run** made the wombats run around the world. Now, nothing happens. Our turtle just sits there, not doing anything.

That's because we haven't programmed our turtle to do anything.

So here is where the fun starts. We will now start programming in Java to make the turtle act.

2. Open the editor for the Turtle by double-clicking the **Turtle** class box.

You will see some code for the Turtle class, as shown in **Listing 1**.

We will concentrate on the act method. That is the bit of code that looks like this:

```
public void act()
{
  // Add your action code here.
}
```

This bit of code specifies what the turtle does when it acts. The code between the curly braces ({ and }) is executed every time the turtle acts. In this case, all that is written here is the following:

```
// Add your action code here.
```

The double slash at the beginning of the line marks this line as a comment. It is ignored by the Java system and is just a reminder for the human programmer. In other words, this act method contains no code at all. That's why our turtle does not do anything.

Let's change that.

3. Replace the comment with an instruction so that your act method looks like this:

```
public void act()
{
  move(2);
}
```

4. Compile, create a new turtle, and try clicking the **Act** button and the **Run** button.

So, what did we just do?

The instruction move(2); tells the turtle to move two pixels forward. The turtle does this every time it acts. When you click the **Act** button, the act method is executed once, so the turtle moves a little bit (2 pixels) to the right. Clicking the **Run** button calls the act method over and over again (until you pause again), so the turtle keeps moving.

5. Experiment with values other than 2. What happens when you use 20 instead of 2?

6. Try another instruction by replacing the move(2); instruction with this instruction:

```
turn(2);
```

Remember: Change the code, compile, create a new turtle, and then run. Experiment with different values here as well.

Methods and Parameters

So, what have we just done?

We have called (or *invoked*) a method called **move** and a method called **turn**, and we have passed a parameter, 2, to each of them.

A method is a bit of behavior that an object knows to execute, and all actors know the **move** and **turn** methods (meaning they know how to move and turn).

Both of these methods expect a *parameter*, which is a bit of additional information that tells them exactly how far to move or how much to turn. Both methods expect a number as a parameter, and we supply that number by writing it in parentheses after the name of the method. And each instruction in Java is ended with a semicolon.

Errors

You might have noticed that you need to write your code very precisely. Getting even one character wrong makes the whole program not work. The compiler then reports an error message, and you need to fix your code.

If you have not yet seen an error message, try it now. For example, remove the semicolon after your instruction and try to compile. You'll see what I mean.

Sequences of Instructions

You can write multiple instructions, as many as you like, into your act method. In fact, you can write one after the other:

```
public void act()
{
  move(4);
  turn(2);
}
```

Try this out. Also, place multiple turtles into the world, and experiment with different parameter values for both the **move** and **turn** methods (see **Figure 7**).

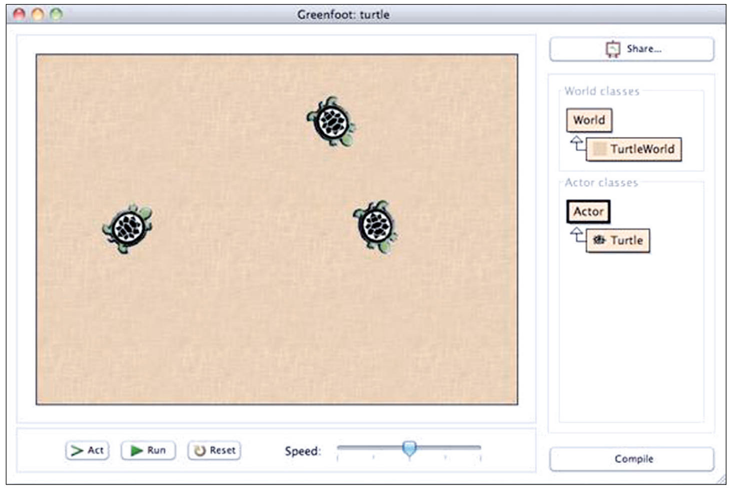


Figure 7

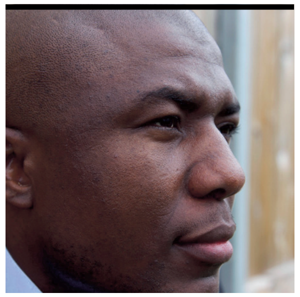
Summary

In this article, you learned the first few steps of writing your own Java code, which demonstrated the following principles:

- The behavior of an object is specified in the object's class.
- More precisely, the behavior of an object is specified in a method definition in the object's class.
- Actor classes in Greenfoot have a method called act that specifies their main behavior.
- The body of the method (the bit between the curly brackets) contains the code that determines what the method does when it is called.
- Instructions can be specified by calling existing methods (such as **move** and **turn** in our example).
- Method calls consist of the name of the method you are calling followed by parameters in parentheses.
- All instructions are terminated with a semicolon.
- We can also write sequences of instructions. ●

LEARN MORE

- Greenfoot
- Java SE API
- Young Developer Resources
- Young Developers Series
  "Wombat Object Basics (Part 1)"
  "Wombat Classes Basics (Part 2)"



PHOTOGRAPH BY
ALLEN MCINNIS/GETTY IMAGES

Introduction to RESTful Web Services

Extend or add functionality into your applications with RESTful Web services.

MAX BONBHEL

BIO

This article, the first of a three-part series, demonstrates how to create a Java Platform, Enterprise Edition (Java EE) application integrating Representational State Transfer (REST) Web services. I will use the NetBeans 6.9.1 IDE and Java EE 6 to create the application and Oracle GlassFish Server 3.1 to deploy it.

Part 2 (coming in the next issue) covers managing the answers sent by a Web service using the JavaScript Object Notation (JSON) framework, while Part 3 focuses on the integration of Java API for XML Web Services (JAX-WS). These articles highlight methods for increasing application agility.

What Are Web Services?

Web services provide a standard means of interoperating among software applications that run on a variety of platforms or frameworks. In other words, Web services are the best and standard way to extend your application in order to make it interoperable with other systems via HyperText Transfer Protocol (HTTP) using Extensible Markup Language (XML). Web services allow you to mask the complexity of each application included in the exchange and achieve complex operation.

Web services are more than the sum of the technologies used to deploy an application. They are a method for carving out services from business functionalities that are then exposed to be consumed by client applications.

Why RESTful Web Services?

Java EE 6 provides natively all mechanisms for REST via the "Java API for RESTful Web Services" (JAX-RS), JSR-311. There are many good reasons to choose REST to add the Web services into your system. Here are some examples:

- The most important reason is that RESTful Web services are easy to learn, easy to build, and easy to deploy.
- REST provides a uniform interface between the producers and the consumers of the services. And RESTful Web services provide support for a variety of message formats (XML, JSON, HTML, and more).
- RESTful Web services are easy to integrate into existing applications in order to extend or add new functionality.

REST is just an architectural style, not a technology. That is why there is a specification, JSR-311, to describe how REST should be implemented in Java. There have been several implementations of this standard. Jersey is the official reference implementation and the one that is most widely used in development and production. Jersey is open source and backed by Oracle.

However, REST might have some limitations when it comes to "big" Web services. This will be discussed in Part 3 of the series.

Prerequisites

The following software was used to develop the application described in this article:

- NetBeans IDE. Download here.
- Jersey (included in NetBeans): It is the open source, production-quality reference implementation for a Java EE specification (JAX-RS JSR-311) for building RESTful Web services.

Note: This article was tested using NetBeans IDE 6.9.1; as of this writing, the latest version is NetBeans IDE 7.0.

Real-Life Application

In this practical section, we will build a real-life application step by step so that you can learn quickly the basics of the RESTful Web services.

The application is an online auction place (like eBay). Sellers post their items in listings, and buyers bid on the items. A seller can post one or many items, and a buyer can bid on one or many items. To simplify, we will consider the following entities: Seller, Item, and Bid.

You can also download all the

source code here.

Now let's code this application in five minutes using NetBeans and JavaServer Faces.

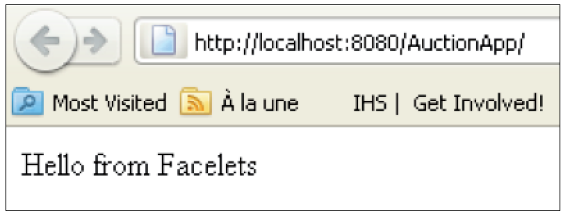
1. Generate the initial NetBeans project:
   a. Launch NetBeans and create a new project.
   b. From the File menu, choose **New Project**.
   c. From Categories, select **Java Web**.
   d. From Projects, select **Web Application**.
   e. Click **Next**.
   f. Type a project name, AuctionApp, and click **Next**.
   g. Make sure the Server is GlassFish Server (or similar wording).
   h. Click **Finish**.

The AuctionApp is created with a simple index.xhtml.

2. Right-click the project and select **Run**.

The default page, as seen in **Figure 1**, will be displayed with the simple message, "Hello from Facelets."

3. Create the entities:
   a. Right-click the AuctionApp project and select **New**; then select **Entity class**.
   b. Type Seller in the **Class Name** field, type com.bonbhel.oracle.auctionApp in the **Package** field, and click **Next**.
   c. From the Provider and Database, select **EclipseLink (JPA 2.0)(default)**.
   d. Choose one of the datasources provided by NetBeans. Click **Finish**.
   e. Repeat Step 3 for each entity.



Hello from Facelets

Figure 1

NetBeans generates the Seller.java, Item.java, and Bid.java files.

Now we are going to add properties in the entities using the NetBeans wizard.

1. Open the Seller.java file, right-click anywhere in the code, and select **Insert code**.
2. Select **Add property** and add the seller properties (String firstName, String lastName, and String email).
3. Open the Item.java file and add the item properties (String title, String description, Double initialPrice, and Seller seller).
4. Click the NetBeans warning to define the entity relationship (bidirectional ManyToOne).

This action creates a list of items in the Seller entity.

5. Open the Bid.java file and add the item properties (String bidderName, Double amount, and item).
6. Click the NetBeans warning to define the entity relationship (bidirectional ManyToOne).

This action creates a list of bids in the Item entity.

7. Generate the Getters and Setter, respectively, for the list of items and bids created in the Seller and Item entities.

At this point, your Seller.java file will look like **Listing 1**.

LISTING 1 LISTING 2

```
@Entity
public class Seller implements Serializable {
   @OneToMany(mappedBy = "seller")
   private List<Item> items;
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   protected String firstName;
   protected String lastName;
   protected String email;

   public List<Item> getItems() {
      return items;
   }

   public void setItems(List<Item> items) {
      this.items = items;
   }
```

See all listings as text

8. Add the RESTful capacities in the initial NetBeans project:
   a. Right-click the AuctionApp project and select **New**; then select **RESTful Web Services** from Entity Classes.
   b. From Entity Classes, click **Add all**; then click **Next**.
   c. Two names need to be specified: a Resource Package name such as com.bonbhel.oracle.auctionApp.resource and a Converter Package name, com.bonbhel.oracle.auctionApp.converter.
   d. Click **Finish**.

New resources classes (that use JAX-RS annotations to define the representation of the entities) and converters classes (that use JAXB annotations such as @XmlElement and @XmlAttribute to define the way to marshal and unmarshal the data) are added to the project. Now the AuctionApp has RESTful capacities.

Take a look at the BidConverter.java code in **Listing 2**.

NetBeans has generated methods that use GET and PUT annotation for retrieving or updating an instance of Bid identified by ID in XML format.

Notice:

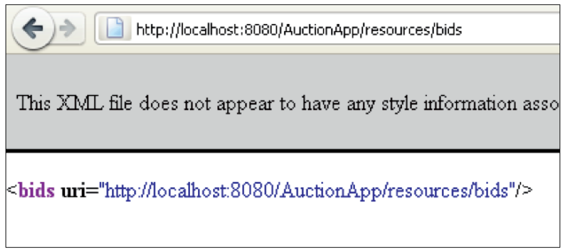
- The @Produces({"application/xml", "application/json"}) annotation allows JAX-RS to specify XML format and JSON as the types of representations a resource can produce.
- The @Consumes({"application/xml", "application/json"}) annotation allows JAX-RS to specify XML format and JSON as the types of representations a

resource can consume.

Take a look at the BidConverter.java code in **Listing 3**.

NetBeans has generated Getter methods that use XmlElement annotation to map the field of Bid entity to XML format.

- The @XmlElement annotation tells JAXB to map the id to a <id> element of the XML document.
- The @XmlAttribute annotation tells JAXB to map the uri to an uri attribute on the the XML document.

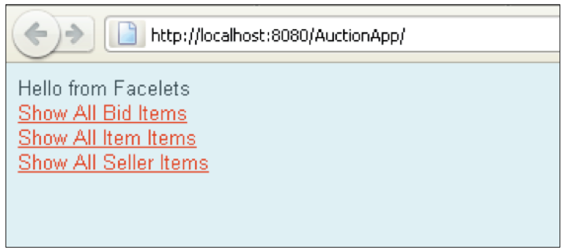


http://localhost:8080/AuctionApp/resources/bids

This XML file does not appear to have any style information asso

<bids uri="http://localhost:8080/AuctionApp/resources/bids"/>

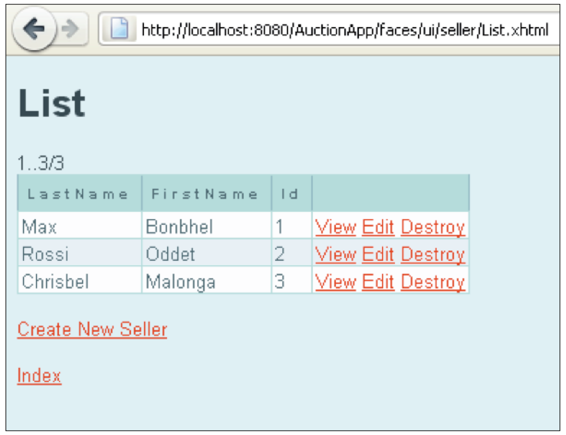
Figure 2



http://localhost:8080/AuctionApp/

Hello from Facelets
Show All Bid Items
Show All Item Items
Show All Seller Items

Figure 3



http://localhost:8080/AuctionApp/faces/ui/seller/List.xhtml

List

1..3/3

LastName	FirstName	Id	
Max	Bonbhel	1	View Edit Destroy
Rossi	Oddet	2	View Edit Destroy
Chrisbel	Malonga	3	View Edit Destroy

Create New Seller

Index

Figure 4

In **Figure 2**, you can see the XML representation of your resources in the browser by typing this URL in your browser: http://localhost:8080/AuctionApp/resources/bids.

If you see this result, it means your RESTful services are working correctly.

But to test it correctly, we need to use Oracle's JavaServer Faces. So we are going to quickly build a JavaServer Faces front end to allow us to create data.

1. Create the JavaServer Faces pages:
   - **a.** Right-click the Web Pages node of the AuctionApp project and select **New**; then select **Folder**.
   - **b.** Type a Folder Name, such as ui. Click **Finish**.
   - **c.** Right-click the AuctionApp project and select **New**; then select **JSF Pages** from Entity Classes.
   - **d.** From Entity Classes, click **Add all**; then click **Next**.
   - **e.** Type a Session Bean Package name, such as com.bonbhel.oracle.auctionApp.facade, and a JSF Classes Package name, com.bonbhel.oracle.auctionApp.presentation.
   - **f.** Click **Browse** to select the ui folder as the JSF pages folder.
   - **g.** Click **Finish**.
2. Run the AuctionApp and add the data:
   - **a.** Right-click the AuctionApp project and select **Run**.

   As seen in **Figure 3**, the home page is displayed and shows the links to create, update, or delete the entities.
   - **b.** From the home page, click on **Show All Seller Items**; then click **Create New Seller** and add a new seller, as seen in **Figure 4**.

LISTING 3 LISTING 4

```
@XmlElement
public Long getId() {
   return (expandLevel > 0) ? entity.getId() : null;
}
@XmlAttribute
public URI getUri() {
   return uri;
}
```

See all listings as text

3. Test the RESTful Web services availability:
   - **a.** Open your browser and type the resource URL http://localhost:8080/AuctionApp/resources/sellers.

   As seen in **Listing 4**, the XML representation of the resource (Seller) is displayed. You can see all Sellers you already created in XML format.

Conclusion

We have seen how NetBeans can help to quickly develop a Java EE application that features RESTful Web services as well as an interface based on JavaServer Faces. The integration of REST with Jersey has shown how Web services can make functionality reusable and the system as a whole more flexible.

In Part 2, I will discuss Web services and the JSON framework. In Part 3, I will cover JAX-WS Web services. ●

LEARN MORE

- The Jersey project home page
- Roy T. Fielding's dissertation defining the REST architectural style
- JavaServer Faces
- Download the source code for this article

Showtime! Java 7 Is Here

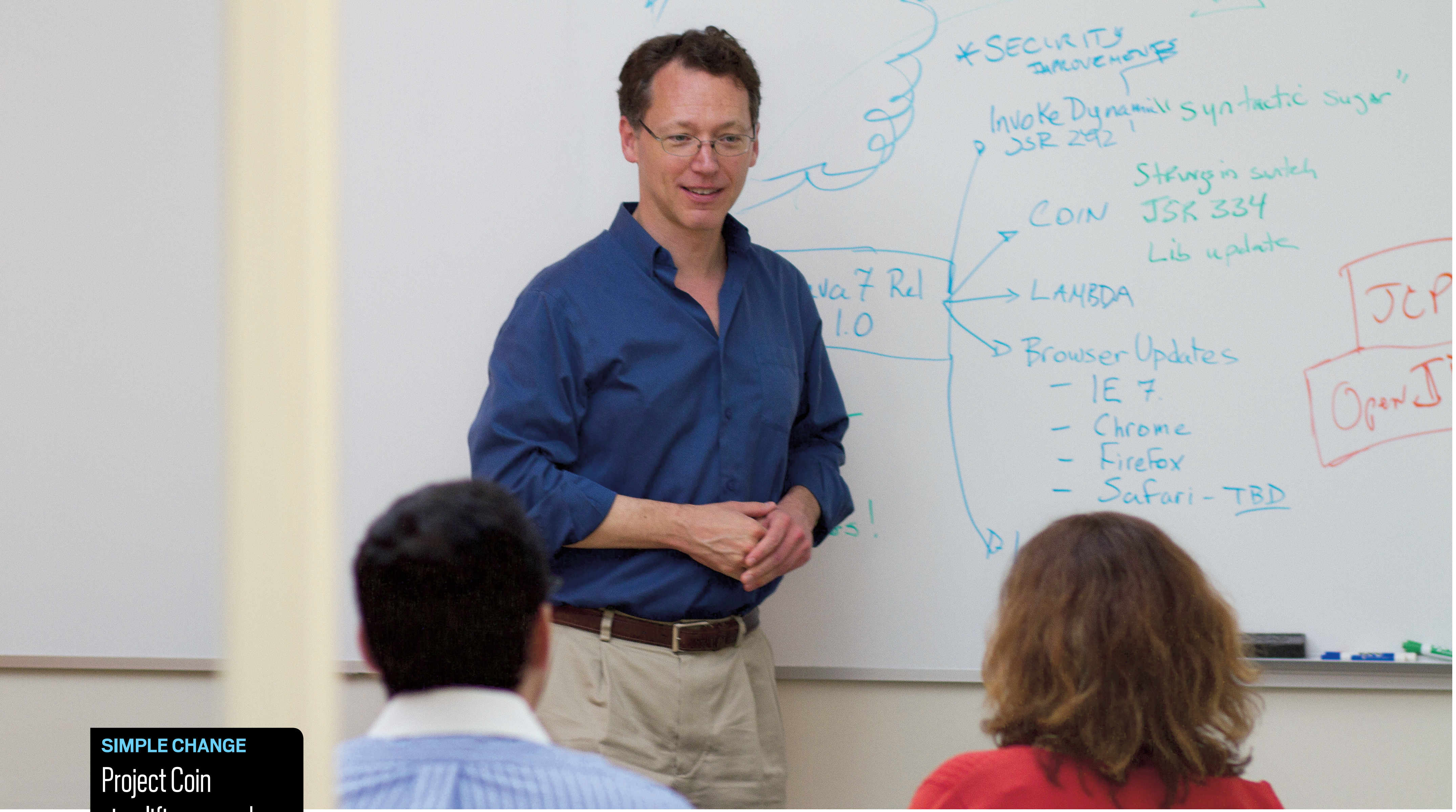
Oracle's **Mark Reinhold** talks with *Java Magazine* about the most important features in Java SE 7. **BY MICHAEL MELOAN**

Java Platform, Standard Edition (Java SE) is the core Java platform for general-purpose computing. The Java SE 7 release addresses a number of important areas, reflecting trends in the programming community, developments in hardware architectures, and a continuing commitment to the success of Java technologies. Java SE 7 (Java 7) will support the creation of maintainable, scalable, high-performance Java applications across a broad range of computing environments. Mark Reinhold, chief architect of the Java Platform Group at Oracle, discusses key features and advantages for developers in the Java SE 7 release.

PHOTOGRAPHY BY BOB ADLER
ART BY I-HUA CHEN

Java Magazine: What are the most important aspects of Java SE 7 for developers, system architects, and the entire Java enterprise?

Reinhold: There are four tent poles in this release: Project Coin (JSR-334, a set of small language enhancements), the new invokedynamic bytecode instruction (JSR-292), New I/O Part 2 (JSR-203), and the fork/join framework. Each of these offers new and valuable options for developers.



Mark Reinhold, chief architect of the Java Platform Group at Oracle, and his team discuss new features in Java SE 7.

SIMPLE CHANGE
Project Coin simplifies everyday programming tasks with features such as the **diamond** and the **try-with-resources** construct.

Project Coin's mission was to make everyday programming tasks easier. The initiative was led by Oracle's Joe Darcy, who began it as an OpenJDK project about two years ago. He prioritized a large number of requests, many from outside the company, and came up with a good set of improvements. One is the *diamond*, <>, which is used when constructing an object whose type is an instance of a generic type. Before Project Coin, it was necessary to write the full generic type in angle brackets on both sides of the assignment statement. Now you can write the full type just on the left side, and on the right side you can use an empty type parameter, <>, and the compiler infers the type on the right side.

For example:

```
Map<String, List<String>> myMap =
  new HashMap<String, List<String>>();
```

can be more compactly rewritten as

```
Map<String, List<String>> myMap =
  new HashMap<>();
```

Another valuable feature of Project Coin is the try-with-resources construct, which addresses a correctness problem that has been implicit in Java APIs from the beginning. When an API allocates external resources, such as sockets, frame buffers, or file descriptors, the application needs to ensure that they are properly released or closed so that those limited resources can be reused. Prior to Project Coin, that was accomplished through the very careful application of try-catch blocks. To handle more than one resource correctly, however, you need multiple levels of nested try-catch blocks. Getting that right is somewhat tricky and often done incorrectly.

The try-with-resources feature extends the syntax of the existing try construct. In the top of a try block, for example, you can create a resource and then, within the body, use it as you

Moving Java Forward: A Video Discussion About Java 7

normally would. There's no need for any catch clauses because the compiler generates all the logic necessary to make sure the resource is properly closed upon exiting the block.

Listing 1 uses a try-with-resources construct to automatically close a java.sql.Statement object.

Project Coin also offers developers the option of using string constants in switch statements, which is a definite ease-of-coding improvement. Before Project Coin, only integral values and instances of enum types could be used in switch statements.

Java Magazine: Is support for asynchronous I/O operations the key feature of New I/O Part 2 (a.k.a. "NIO.2")?

Reinhold: Yes. The asynchronous I/O API will be very useful for certain kinds of high-end server apps and other software that requires massive I/O throughput. Java SE 7 also offers a true file system API. The platform has always been somewhat limited in support for interaction with file systems. For example, simple operations such as creating symbolic links, checking file permissions, and being able to request callbacks when a file is updated are very common facilities in operating systems today. All of that is now available in an API that is part of the standard.

In the early days of Java, platform independence was a prime directive. Often that was appropriate, but it was sometimes painful. That's why the original file system API was intentionally somewhat rudimentary. In this new file system API, some platform-specific features are exposed. It's like an onion. The first layer defines platform-independent operations that work the same way everywhere. Beneath that layer, however, things become more platform-specific. For instance, if you need to look at a POSIX access control list, you can peel a layer back and use support for that feature when in a POSIX environment, and you can do likewise for features that are specific to a Windows environment.

Java Magazine: Today, multicore processors are becoming the norm. How can developers utilize the fork/join framework to their advantage?

Reinhold: The fork/join API makes it very straightforward to take a problem that can be decomposed in a recursive manner and spread the work required to solve it across an arbitrary number of processor cores. If the divide-and-conquer strategy is applicable to a given problem, then fork/join is often a very good fit. It takes care of all the concurrency details for you.

Java Magazine: Can you describe an application environment where fork/join would be particularly useful?

Reinhold: Image processing is a good example. Many of the classic image processing algorithms decompose recursively in a very natural way. The image is broken up, and fork/join tasks are assigned to process those segments. Large array computations are another good example. It's fairly straightforward to recursively decompose some problems so that subcomputations focus on just parts of the array simultaneously, after which the results are aggregated back into a single value or a vector.

Java Magazine: What does the invokedynamic bytecode instruction deliver for developers?

Reinhold: The invokedynamic instruction is not useful for the Java language as it is defined today. The main goal of invokedynamic is to facilitate the compilation of dynamic languages down to Java bytecodes. This allows

LISTING 1

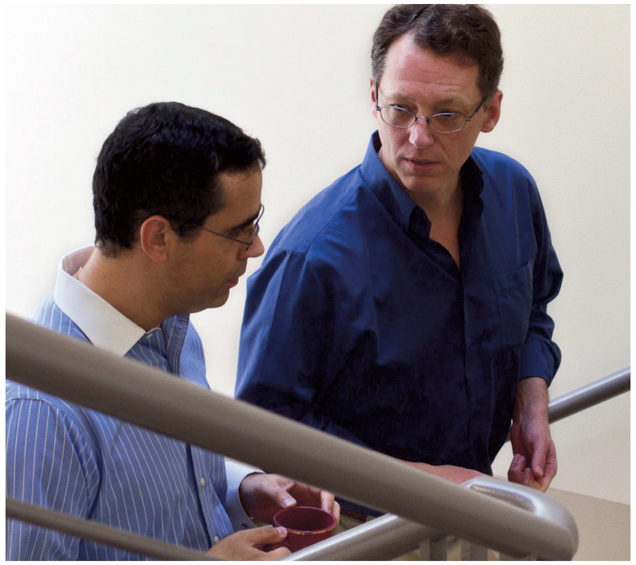
```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, PRICE from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");

      float price = rs.getFloat("PRICE");

      System.out.println(coffeeName + ": " + price);
    }

  }
}
```

See listing as text



Mark Reinhold and Java SE product managers discuss final testing and launch plans for Java SE 7.

for the implementation of languages such as Ruby on top of the Java virtual machine (JVM). The problem it solves is not dynamic typing per se, but rather dynamic method dispatch. Languages such as Ruby, Smalltalk, and JavaScript have patterns of method dispatch that depend on runtime information in a way that the Java method dispatch does not. The invokedynamic instruction enables that dispatch to be expressed at the bytecode level in a way that the JVM can optimize it as efficiently as it optimizes typical Java method dispatch patterns. In terms of performance, the potential improvements in dynamic language execution are dramatic. We've been working closely with Charles Nutter, the lead developer on JRuby. He's had a great deal of influence on the invokedynamic design and implementation. He's using it in JRuby today and getting good results.

Java Magazine: Given that invokedynamic will facilitate interoperability between languages, what would be a typical scenario where a developer might build a hybridized Java/Ruby application?

Reinhold: Dynamic languages such as Ruby can offer higher levels of developer productivity for some kinds of applications. Many developers find that they're more productive writing Web front ends in Ruby. So a Ruby front end coupled with Java on the back end for handling complex business logic, for example, could be a powerful combination.

Another example is developers whose IT organizations only allow them to deliver Java Platform, Enterprise Edition (Java EE) components into an application server. Using the regular Ruby interpreter in an environment like that is just not an option. JRuby, however, can be bundled inside a WAR file together with a Ruby Web app. It looks just like another Java application, even though on the inside it contains a Ruby runtime and Ruby code. So, additional flexibility is provided for developers facing those kinds of constraints.

Java Magazine: Will invokedynamic be leveraged in future Java releases?

Reinhold: Quite possibly. Project Lambda is slated to add closures to Java in the Java SE 8 release. The current prototype implementation uses invokedynamic as an efficient way to implement Lambda expressions. Even though invokedynamic was originally aimed at the problem of making other languages compile and perform well, it turns out that it will also be beneficial for Java itself in the longer term.

Java Magazine: How important have external contributors been to this release?

Reinhold: Very. Sun open-sourced the JDK in 2007, and Oracle continues to place a high value on the participation of external contributors. The fork/join framework was developed by Professor Doug Lea at SUNY Oswego; the new sound synthesizer was contributed by open source developer Karl Helgason; and the new graphics pipeline for Java 2D was written by Clemens Eisserer, the winner of the OpenJDK Innovators' Challenge a few years ago. It's great to have that level of external participation, which makes yet more innovation available to the entire Java ecosystem.

Java Magazine: Any closing remarks about the Java SE 7 release?

Reinhold: Yes: Go forth, download, and have fun! ●

Michael Meloan began his professional career writing IBM mainframe and DEC PDP-11 assembly languages. He went on to code in PL/I, APL, C, and Java. In addition, his fiction has appeared in *WIRED*, *BUZZ*, *Chic*, *L.A. Weekly*, and on National Public Radio. He is also a Huffington Post blogger.

LEARN MORE

- JDK 7 Features
- JDK 7 Preview Release Downloads
- Project Coin
- JSR-334: "Small Enhancements to the Java Programming Language"
- JSR-292: "Supporting Dynamically Typed Languages on the Java Platform"
- JSR-203: "More New I/O APIs for the Java Platform ("NIO.2")"
- Fork/join framework (PDF)
- Mark Reinhold's blog
- Joe Darcy's Oracle blog

n software

We didn't invent the Internet...

...but our components help *you* power the apps that bring it to business.

PURE JAVA COMPONENTS & ENTERPRISE ADAPTERS FOR

- **E-Business**
  AS2, EDI/X12, NAESB, OFTP …
- **Credit Card Processing**
  Authorize.Net, TSYS, FDMS …
- **Shipping & Tracking**
  FedEx, UPS, USPS …
- **Accounting & Banking**
  QuickBooks, OFX …
- **Internet Business**
  Amazon, eBay, PayPal …
- **Internet Protocols**
  FTP, SMTP, IMAP, POP, WebDav …
- **Secure Connectivity**
  SSH, SFTP, SSL , Certificates …
- **Secure Email**
  S/MIME, OpenPGP …
- **Network Management**
  SNMP, MIB, LDAP, Monitoring …
- **Compression & Encryption**
  Zip, Gzip, Jar, AES …

The Market Leader in Internet Communications, Security, & E-Business Components

Each day, as you click around the Web or use any connected application, chances are that directly or indirectly some bits are flowing through applications that use our components, on a server, on a device, or right on your desktop. It's your code and our code working together to move data, information, and business. We give you the most robust suite of components for adding Internet Communications, Security, and E-Business Connectivity to any application, on any platform, anywhere, and you do the rest. Since 1994, we have had one goal: to provide the very best connectivity solutions for our professional developer customers. With more than 100,000 developers worldwide using our software and millions of installations in almost every Fortune 500 and Global 2000 company, our business is to connect business, one application at a time.

To learn more please visit our website → **www.nsoftware.com**

JDK 7 Will Change the Way You Write Code—Today!

New features in JDK 7 reduce errors and make several difficult or tedious tasks easier.

HERB SCHILDT

BIO

Having written extensively about the Java language since its original 1.0 version, I have watched it evolve, mature, and grow with every major release. Each new version added features that responded to the needs and desires of the programmers who used the language. This process of ongoing refinement and adaptation helped secure the dominance of Java in the world of programming. It also kept the language fresh, vibrant, and alive.

One thing I have always found interesting about the evolution of Java is that sometimes it took quite a while for a new feature to really catch on—to fully mainstream. The best example of this is generics, which were added by JDK 5. Generics fundamentally expanded the power of the language and the reliability of programs. They also added a completely new syntax element and a new way to think about writing Java code. Because generics were such a large change, it took some time before the use of generics became commonplace.

With the release of JDK 7, Java is once again evolving, responding to the needs of programmers. And once again, new features have been added that expand the power and scope of the language. However, unlike some of the additions in the past, which have been slow to catch on, the new language features in JDK 7 will change the way you write code today.

SMALL CHANGE

For many programmers, the Project Coin changes will be **the most important** new features in JDK 7.

As you might know, the new language features in JDK 7 were developed by *Project Coin*. The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7.

But here's the interesting thing. Although these new features are collectively referred to as *small*, the effects of these features will be quite large in terms of the *code they affect*. Simply put: for many programmers, the Project Coin changes will be the most important new features in JDK 7.

To understand why, let's consider the following list of Project Coin additions.

- An expanded `try` statement, called `try`-*with-resources*, supports the automatic closing of a resource, such as a file stream.
- Type inference (via *diamond*) is provided when constructing a generic instance.
- Enhanced exception handling, in which two or more unrelated exception types can be caught by a single type, was added, plus better type checking for exceptions that are rethrown.
- A `string` can now control a `switch` statement.
- There is support for binary integer literals with the new prefix `0b` or `0B`—for example, `0b1010`.
- Support for underscores in numeric literals, such as `59_234_412`, was added. The underscores are ignored by the compiler, but they add clarity to long numeric values.
- Compiler warnings associated with varargs methods that have nonreifiable parameters have been improved, and you have more control.

All are the types of things that programmers have been wanting—and waiting for. All streamline or simplify some previously difficult or tedious tasks. All help you write better, more error-free code. While there isn't space to examine each of these features here, it's useful to look at examples from both ends of the "change spectrum."

Binary Literals and Underscores in Numeric Literals

Let's begin with the two features that would seem, justifiably, to be called small changes: the ability to specify a binary literal and to use underscores in a numeric literal. At first glance, these seem like nearly inconsequential additions, hardly worth mentioning, but the opposite is true. Not only do they add convenience, they help prevent errors.

For example, consider a situation in which some specific bit pattern is required, perhaps for use as a bit mask. Obviously, one normally thinks about a bit pattern in terms of binary. Thus, it would be helpful to specify a bit pattern using a binary literal. The trouble was that, in the past, there were no binary literals. This meant that a different approach was required, of which there were several.

For instance, if the bit pattern 0110 1101 was needed, you might have used Byte.parseByte("01101101", 2), but this involves a method call. If what you wanted was an actual *literal*, it was not uncommon to press a hexadecimal literal into service. For example, who hasn't seen something like this?

```
byte myBits = 0x6D; // 0110 1101
```

Here, the value is encoded as the hexadecimal literal 0x6D, and the comment depicts the bit pattern. This approach is not, however, without problems, one being that it is possible to make a mistake when converting from binary into hexadecimal, resulting in the wrong bit pattern. (Maybe your finger presses the wrong key on the calculator, and you don't catch it.) Unfortunately, such a mistake could result in a bug that is very difficult to find. Alternatively, the value might be right, but the bit pattern in the comment might be wrong, thus misleading anyone reading the code.

With JDK 7, you can eliminate the possibility of such errors because you can now use a binary literal to specify a bit pattern. For example:

```
byte myBits = 0b01101101;
```

Here, the value is encoded directly by a binary literal. This means that there is no chance for conversion errors, and the bit pattern is self-documented. Thus, you have a direct, visual representation of precisely the bit pattern you wanted—a much more reliable, transparent approach.

You can further enhance the readability of a binary value by inserting underscores, like this:

```
byte myBits = 0b0110_1101;
```

Although useful here, underscores in large binary values are even more valuable. For example, which of the following values is easier to read?

```
0b0110110111000111
0b0110_1101_1100_0111
```

Even though binary literals and underscores in numeric values are two of the smallest of the "small" language enhancements, they both offer significant improvements that let you write code with greater clarity and less chance for error.

LISTING 1

```
FileInputStream fIn = null;
try {
  fIn = new FileInputStream("somefilename");
  // Access the file ...
} catch(IOException e) {
  // ...
} finally {
  // Close file.
  try {
    if(fIn != null) fIn.close();
  } catch(IOException e) {
    // ...
  }
}
```

See all listings as text

Try-with-Resources Statement

If binary literals and underscores in numeric values are on one end of the change spectrum, at the other end is try-with-resources. I consider try-with-resources to be the *single most important* new language feature added by JDK 7. It not only addresses a long-standing issue, it also prevents an entire class of errors. One of the thorniest things about handling resources, such as file streams, is ensuring that they are closed when they are no longer needed. Forgetting to close a resource can lead to memory leaks and other problems. The try-with-resources statement automates the closing process, and it does so in an elegant way.

To understand the importance of try-with-resources, let's begin by reviewing an example of the type of situation it is designed to improve. As you know, working with a file has traditionally involved three separate actions. You need to open the file, use the file, and then close the file. Prior to JDK 7, you might have used some variation of the code shown in **Listing 1**.

Notice that the file stream is closed in the finally block, which is automatically executed in all cases when the try block is left. In this example, fIn is initially assigned null. Then, the try block is entered. If fIn is successfully opened, fIn is given a non-null value. If an error opening the file occurs, fIn remains null. When the try block ends (either normally or because of an exception), the finally block is executed. Then, if fIn is not null, it means that the file was successfully opened and must be closed. Otherwise, an error has occurred, in which case the call to close() is not executed. Because close() can also cause an exception, it,

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
java.net
blog

too, is wrapped in its own try block. Of course, there are many variations of this sequence, including those that throw exceptions to a calling routine, but no matter how it was implemented, the file still needs to be explicitly closed in the finally block.

When used correctly (and consistently), the preceding sequence *does* ensure that the file associated with fIn is properly closed. There are, of course, also troubles with this approach. First, it is still possible to forget to close a file. For example, in the case in which several files are being accessed within the same try block, a programmer might inadvertently forget to close one in the finally block. It is also possible to commit a coding error that prevents the file from being closed. Let me give you a simple example from my own experience.

A while back, I was working on some code for use as an example in one of my books. I was using a sequence similar to that shown above. However, when it came time to close the file, I typed the following line in the finally block:

```
if(fIn == null) fIn.close();
```

Can you see the problem? Instead of using != in the if statement, I accidentally typed ==. As a result, an attempt to close the file would be made only if the file wasn't open!

Fortunately, I caught the mistake. But if I hadn't, it would have resulted in a bug that could have been difficult to discover. The only way the error would have been apparent is when an attempt to open the file failed. (In that case, fIn would still be null, and the call to close() would generate a null-pointer exception.) However, in the example I was developing, that was a very unlikely event. In general, the file would have opened successfully and no symptoms would have been displayed, except that the resources associated with the file would have never been released. Therefore, inadvertently typing == instead of != created a resource leak, but the overall code sequence still "looked right." Fortunately, with JDK 7, such sources of error are a thing of the past.

TRY IT, YOU'LL LIKE IT

try-with-resources is a major, powerful addition to the language. **Yes, it really is** that important. Start using it now.

The try-with-resources statement performs two functions. First, it declares and initializes a resource, such as a file stream. Second, when the try block ends, the resource is automatically closed. In the case of a file stream, this means that the file is automatically closed. You no longer need to call close() explicitly.

Listing 2 shows how by using try-with-resources, the preceding sequence can be rewritten. Notice how fIn is now declared and initialized within the try statement, instead of requiring a separate step.

LISTING 2

```
try(FileInputStream fIn = new FileInputStream("somefilename")) {
  // Access the file ...
} catch(IOException e) {
  // ...
}
```

See all listings as text

Not only is the code in **Listing 2** much shorter code, it also ensures that fIn will be closed in all cases. When the try block is left (whether normally or because of an exception), fIn is automatically closed. You can't forget to close it, and a programming mistake can't prevent it from being closed.

At this point, one thought might have occurred to you. If an I/O exception occurs inside the try block, and if another I/O exception occurs when the file stream is automatically closed, what happens to those two exceptions? In such a case, the first one is thrown and the other is added to the suppressed exception list. You can obtain this list by calling the getSuppressed() method defined by Throwable. This is another benefit of try-with-resources.

One other thing: try-with-resources can manage any resource that implements the new AutoCloseable interface. So, it's not just for file streams.

Because try-with-resources streamlines your code, prevents resource leaks, and in the process makes your code more resilient, it is hard to argue against it. It is a major, powerful addition to the language. Yes, it really is that important.

In my view, try-with-resources is something that every Java programmer will want to start using right away.

Conclusion

Of course, I feel strongly about all the other new language features in JDK 7, too. For example, type inference via diamond simplifies the syntax for creating generic instances; the ability to catch multiple exceptions with a single catch statement reduces code bloat; and the ability to use a string with a switch answers a long-standing need. (Who hasn't, at one time or another, wanted the ability to control a switch with a string?) Combined, the new JDK 7 language features add real benefits to the language, and they make our lives as programmers a little easier. Simply put, these features are just too useful to ignore. ●

LEARN MORE

- Project Coin
- Project Coin mailing-list archives
- Blogs about Project Coin
- Project Coin: JSR-334 in Public Review
- JSR-334 documents and public reviews

RAYMOND GALLARDO

BIO

PHOTOGRAPH BY
GENEVIÈVE ARBOIT

Dynamically Typed Languages and the invokedynamic Instruction

The new invokedynamic instruction enables a runtime system to customize the linkage between a call site and a method implementation, instead of using hardwired linkage behavior.

When Sun was developing the Java platform, it did not encourage developers to run other programming languages on the Java virtual machine (JVM). As a matter of fact, Sun wanted developers to use Java as the foremost programming language for enterprise development. Ten years later, close to the release of Java Platform, Standard Edition 7 (Java SE 7), this goal has been accomplished.

Developers recognize the extraordinary value in the Java platform and in the JVM as a deployment vehicle. However, they also recognize that the Java platform isn't always the best tool. New and old languages, such as Ruby and Python, have been reimplemented, for example, as JRuby and JPython, to run on the JVM.

Why have developers bothered to reimplement these languages rather than use their original C counterparts? Developers could have instead converted their old code to Java, but implementations such as JRuby and JPython can take advantage of the following unique JVM features:

- High optimization
- Cross-platform compatibility and portability
- Open source
- Access to standard Java libraries
- Multithreading
- Garbage collection
- Better performance in large enterprise systems with multiple processors
- Widespread popularity and large installed base

The difficulty in implementing a language for the JVM is that the JVM was made for Java. This means that if the language you want to reimplement has different object or method resolution mechanics than Java, the JVM might not work very well. In particular, the one feature that makes it difficult to implement Ruby and Python on the JVM is that they are dynamically typed languages, while Java is a statically typed language. This is where the new bytecode instruction, invokedynamic, comes in. It can simplify and improve the implementation of compilers and runtime systems for dynamically typed languages on the JVM.

The Difference Between Statically and Dynamically Typed Languages

Java is a statically typed language. This means it performs type checking at compile time. Type checking is the process of verifying that a program is type-safe. A program is type-safe if the arguments for all its operations are the correct type.

Ruby and Python are dynamically typed languages. This means they perform type checking at runtime. These languages typically do not have any type information available at compile time, so the type of an object can be determined only at runtime.

Statically Typed Languages Are Not Necessarily Strongly Typed Languages

A programming language that features strong typing specifies restrictions on the types of values supplied to its operations. If a computer language such as Java implements strong typing, it prevents the execution of an operation if its arguments have the wrong type. Conversely, a language that features weak typing would implicitly convert (or cast) the arguments of an operation if those arguments have wrong or incompatible types.

Statically typed programming languages can employ strong typing or weak typing. Similarly, dynamically typed languages can also apply strong typing or weak typing. For example, Ruby is dynamically typed and strongly typed. Once a variable has been initialized with a value of some type, Ruby will not implicitly convert the variable into another datatype. For example, Ruby would not allow the following:

```
a = "40"
b = a + 2
```

In this example, Ruby will not implicitly cast the number 2, which has a Fixnum type, to a string.

The Challenge of Compiling Dynamically Typed Languages

Consider the following dynamically typed method (of a hypothetical programming language), addtwo, which adds any two numbers (that can be of any numeric type) and returns the sum:

```
def addtwo(a, b)
    a + b;
end
```

Suppose your organization is implementing a compiler and runtime system for the programming language in which the method addtwo is written. In a strongly typed language, whether typed statically or dynamically, the behavior of + (the addition operator) depends on the types of the operands.

A compiler for a statically typed language chooses which implementation of + is appropriate based on the static types of a and b. For example, a Java compiler implements + with the iadd JVM instruction if a and b are of type int. The addition operator will be compiled to a method call because the JVM's iadd instruction requires the operand types to be statically known.

In contrast, a compiler for a dynamically typed language must defer the choice until runtime. The statement a + b would be compiled as the method call +(a, b), where + is the method name. (Note that a method named + is permitted in the JVM but not in the Java programming language.) Suppose then that the runtime system for the dynamically typed language is able to identify that a and b are variables of type int. The runtime system would prefer to call an implementation of + that is specialized for integer types rather than arbitrary object types.

The challenge of compiling dynamically typed languages is how to implement a runtime system that can choose the most appropriate implementation of a method or function—after the program has been compiled. Treating all variables as objects of type Object would not work efficiently; the Object class does not contain a method named +.

NEW BYTECODE

invokedynamic can simplify and improve the implementation of compilers and runtime systems.

The invokedynamic Instruction

Java SE 7 introduces the invokedynamic instruction, which enables the runtime system to customize the linkage between a call site and a method implementation. This contrasts with other JVM instructions, such as invokevirtual, in which linkage behavior specific to Java classes and interfaces is hardwired by the JVM.

In the previous addtwo example, the invokedynamic call site is +. An invokedynamic call site is linked to a method by means of a *bootstrap method*, which is a method specified by the compiler for the dynamically typed language that is called once by the JVM to link the call site. The object returned from the bootstrap method permanently determines the call site's behavior.

LISTING 1

```
invokedynamic  InvokeDynamic
  REF_invokeStatic:
    Example.mybsm:
      "(Ljava/lang/invoke/MethodHandles/Lookup;
        Ljava/lang/String;
        Ljava/lang/invoke/MethodType;)
      Ljava/lang/invoke/CallSite;":
    +:
      "(Ljava/lang/Integer;
        Ljava/lang/Integer;)
      Ljava/lang/Integer;";
```

See listing as text

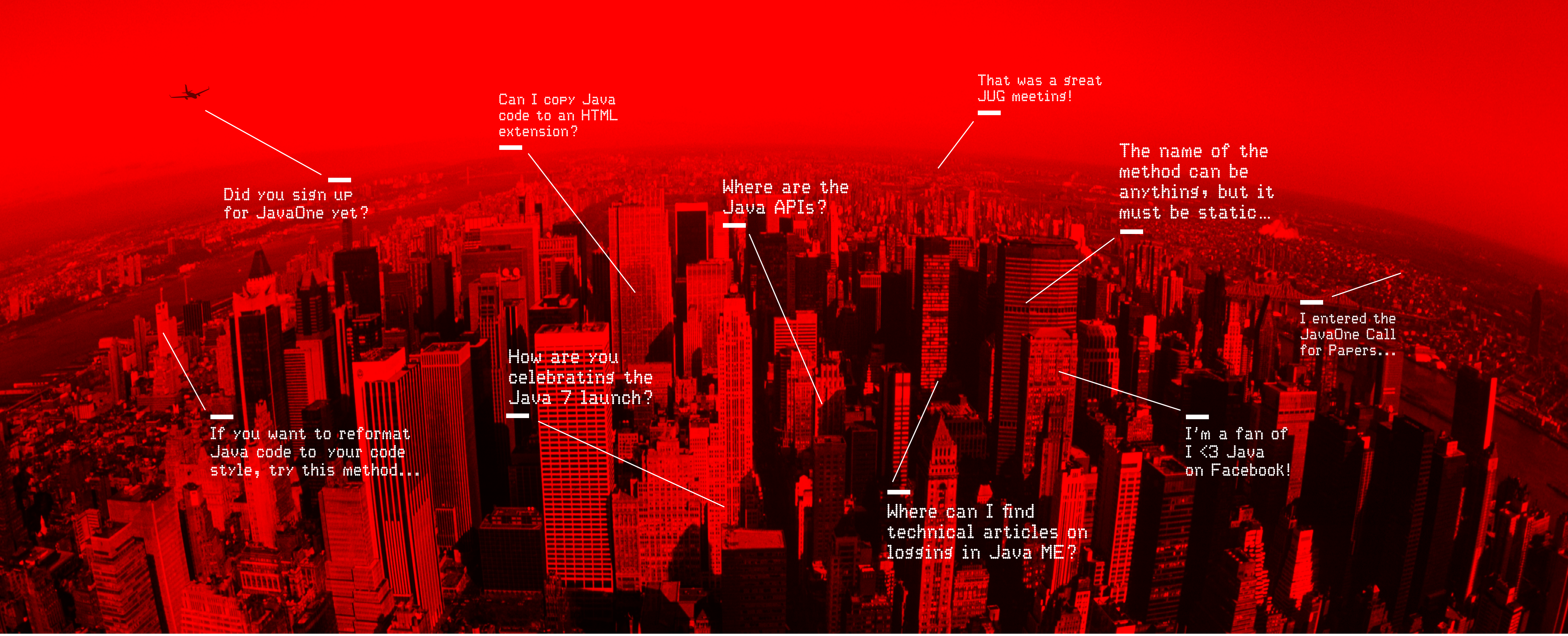
Listing 1 shows an example of an invokedynamic instruction. Note that this example uses the syntax of the ASM Java bytecode manipulation and analysis framework, and line breaks have been added for clarity.

In this example, the runtime system links the dynamic call site specified by the invokedynamic instruction (which is +, the addition operator) to the method IntegerOps.adder. (The IntegerOps class belongs to the library that accompanies the dynamic language's runtime system your organization is implementing.) It does this by using the bootstrap method Example.mybsm, which your organization is responsible for writing.

Java SE 7 introduces the package java.lang.invoke, which contains an API that is essential for writing bootstrap methods, including the new datatype MethodHandle. A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation. The java.lang.invoke package includes other methods that create and manipulate method handles. ●

LEARN MORE

- Java Virtual Machine Support for Non-Java Languages
- java.lang.invoke Package
- The Da Vinci Machine Project
- John Rose's blog at oracle.com (John Rose is the project lead for the Da Vinci Machine Project and the specification lead for the invokedynamic instruction.)


Did you sign up for JavaOne yet?
Can I copy Java code to an HTML extension?
That was a great JUG meeting!
Where are the Java APIs?
The name of the method can be anything, but it must be static...
I entered the JavaOne Call for Papers...
If you want to reformat Java code to your code style, try this method...
How are you celebrating the Java 7 launch?
Where can I find technical articles on logging in Java ME?
I'm a fan of I <3 Java on Facebook!

Oracle Technology Network: Your Java Nation.

Come to the best place to collaborate with other professionals on everything Java.

Oracle Technology Network is the world's largest community of developers, administrators, and architects using Java and other industry-standard technologies with Oracle products. Sign up for a free membership and you'll have access to:

- Discussion forums and hands-on labs
- Free downloadable software and sample code
- Product documentation
- Member-contributed content

Take advantage of our global network of knowledge.

JOIN TODAY ▶ Go to: oracle.com/technetwork/java

ORACLE®

Using Adobe Flex and JavaFX with JavaServer Faces 2.0

Take advantage of new features in JSF 2.0 and integrate Adobe Flex and JavaFX into your JSF applications.

RE LAI

BIO

The JavaServer Faces (JSF) 2.0 specification builds on the success and lessons from the last six years of usage of the JSF 1.0 specification. It takes inspiration from Seam and other Web frameworks and incorporates popular agile practices, such as convention over configuration and annotation over XML. This results in a more streamlined framework. Highlights include standardized Ajax support; Facelets as the default view technology; and custom composite components, which finally make component authoring straightforward and even enjoyable.

This article explores how these new features can be utilized to facilitate embedding rich client applications. Adobe Flex has been a popular rich internet application framework. JavaFX, while relatively new, builds on top of the Java platforms and has attracted much attention. There has been constant interest in integrating rich clients into Java Web applications. With JSF 2.0 and its focus on simplified development, integration has become easier than ever.

We start with a sample Flex pie chart application that displays the results of a survey about the popularity of ice cream flavors. A JSF composite component is used to encapsulate the embedding. Next, instead of hard-coding, the survey result is passed to the Flex application from a JSF managed bean. Then, we further enhance the sample by adding server round-trips that submit a user's choice of the favorite flavor. Finally, we reimplement the chart in JavaFX and show how to embed it into the JSF application.

Running the Sample

The source code for the sample application is available here.

The application is developed using Flex SDK 4.1, JSF Mojarra Implementation 2.0.2, and JavaFX SDK 1.3.1. NetBeans 6.9.1 is used as the IDE, which already bundles the latter two libraries. The three attached projects are SampleChartFlex, SampleChartFX, and SampleWeb.

To run the Web application inside NetBeans, open these projects using NetBeans, right-click project SampleWeb, and run.

To modify and compile the Flex project, you need to install Flex SDK and modify SampleChartFlex/build.xml to point to the Flex SDK installation location (see **Listing 1**).

Afterward, you can invoke the Build command from NetBeans to build these projects. The ant build files of both the SampleChartFlex and SampleChartFX projects are customized so that the packaged swt or jar files are copied into project SampleWeb automatically during the build.

Creating the Application

First, we create a simple pie chart application in Flex to display the popularity of ice cream flavors (see **Figure 1**). You click a chart item, and then the message label displays your choice.

The Flex application consists of a pie chart and a message label. The pie chart data is provided by function getChartData(), as seen in **Listing 2**. When a user clicks a chart item, onItemClick processes the event and updates the message label. The source file is compiled into SampleChartFlex.swf using mxmlc. The provided sample project SampleChartFlex has customized its ant build.xml file, which invokes mxmlc when you build the project.

Embedding the Flex Application

To embed the Flash object into our JSF Web application, we first add SampleChartFlex.swf into folder

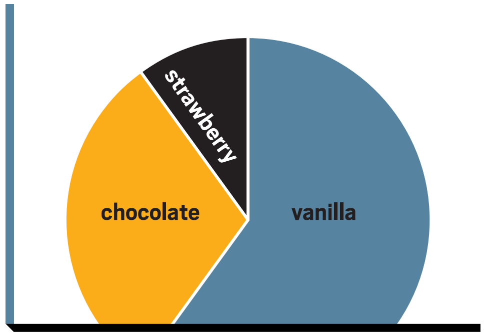


Figure 1

resources/demochart of the Web content of our JSF application project SampleWeb. We create a composite component to encapsulate the embedding.

Composite components are a new facility in JSF 2.0 that tremendously eases the development of custom components. You no longer need to worry much about encoding, decoding, tag library descriptors (TLDs), and renderers. You simply declare a Facelet composite component file and use it, similar to the acclaimed custom tag support in Grails. **Listing 3** shows our custom component demo:chart.

The open source SWFObject is used to embed the Flash content. The JavaScript file, swfobject.js, can be found under folder templates\swfobject of the Flex 4 SDK installation. Copy it into folder resources\demochart of our Web content.

ADOBE TALK

LiveCycle Data Services **would be particularly appealing** if you could devote both the client and server sides to a complete Adobe solution, but open sourcing of BlazeDS and AMF makes it possible to work with other technologies.

To mitigate name conflicts, our local variables are defined in an anonymous function and the div HTML element ID is prefixed with the composite component client ID.

Now that we've created the custom component, we can use tag demo:chart just like any other JSF tags. It is transparent that Flex is used in the implementation. **Listing 4** shows an example.

Passing Variables to Flex Applications

More often than not, embedded Flex applications rely on dynamic data. It turns out to be easy to pass variables into Flex applications with the help of flashVars.

This section extends our sample chart by passing the ice cream flavor survey result from a JSF managed bean, as shown in **Listing 5**. We use the JSF 2.0 annotation to designate a managed bean.

To feed the survey result from the managed bean to Flex, we first modify our JSF composite component chart.xhtml by adding an attribute named data to the interface section to accept the survey result and passing the survey result as flashVars into Flex (see **Listing 6**).

Now in the consuming JSF page, we just need to pass the ice cream flavor survey result to the demo:chart tag. We use the following JSF page source code snippet (index.xhtml):

```
<demo:chart
 data="#{iceCreamSurvey.result}"/>
```

On the Flex application side, we need to modify function getChartData to fetch the parameter. We use the Flex source code snippet shown in **Listing 7**.

In this example, the data format is simple. Therefore, we just parse it using regular expressions. In more-complicated cases, consider formal encoding such as JavaScript Object Notation (JSON).

LISTING 1 / LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6

```
<!-- Change me to your Flex SDK installation location -->
<property name="FLEX_HOME" value="C:/Programs/Adobe/flex_sdk/4.1"/>
```

See all listings as text

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java.net
blog

Using JSF Ajax

In this section, we move on to a more complicated scenario: round-trip communications between Flex and JSF server sessions. We'll use a novel, yet practical, approach to integrating the best of Flex and JSF using the JSF 2.0 Ajax feature.

There are several ways Flex applications can communicate with servers. LiveCycle Data Services is Adobe's data solution that umbrellas several technologies, including the Java server-based BlazeDS and Action Message Format. Flex also provides generic data access components to communicate with servers, including HTTP and Web services. In addition, Flex has good integration with JavaScript, enabling us to integrate at the browser side, relaying to the Ajax application to communicate with servers. These approaches can all be used with JSF, each with pros and cons.

With the arrival of JSF 2.0, the Ajax API has been standardized. You can exploit the feature to integrate Flex applications with JSF. It is in essence integration at the browser side. We'll rely on the JSF Ajax framework to handle session and view state tracking. Because the JSF Ajax API is part of the 2.0 specification, it is guaranteed to be supported by all implementations. On the server side, it is fairly transparent that a Flex client is used. Therefore, this approach is easy to plug in to an existing JSF application.

The additional JSF Ajax layer conceivably would add performance overhead. This should not be an issue for the majority of Ajax cases, when the data exchange is small.

We'll modify our sample by submitting the selection when a user clicks on a flavor in the pie chart. A JSF managed bean would process the selection and reply with a message, which is in turn displayed in the Flex application. On the Flex application side, we'll modify function onItemClick to use ExternalInterface to invoke JavaScript function demo.ajax.submit inside the embedding Web page, which we will define shortly. **Listing 8** shows Flex source code snippets.

Add a callback function named refresh to update the message label. The function is exposed to JavaScript via ExternalInterface.addCallback during the initialization of the Flex application, as seen in **Listing 9**.

For our JSF composite component, we'll add one more attribute, response, in the interface section, which is mapped to the server response to our asynchronous submit.

JSF composite component source code snippets (resources/demo/chart.xhtml):

```
<cc:interface>
  <cc:attribute name="data" />
   <cc:attribute name="response" />
</cc:interface>
```

Next, inside the implementation section, define a hidden form to submit to and receive response from the server:

```
<h:form id="form"
 style="display:none">
 <h:outputText id="out"
  value="#{cc.attrs.response}"/>
</h:form>
```

LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10 / LISTING 11

```
private function getChartData() : ArrayCollection {
    // Retrieve "data" from flashVars,
    // Formatted as Map.toString(), e.g.,
    // {Strawberry=10, Chocolate=30, Vanilla=60}
    var input : String = Application.application.parameters.data;
    var data : Array = input ? input.split(/\W+/) : [];
    var source = [];
    for (var index : int = 1; index < data.length - 1; index += 2) {
      source.push( (flavor: data[index], rank: parseInt(data[index+1])} );
    }
    return new ArrayCollection(source);
}
```

See all listings as text

Add the JavaScript from **Listing 10** to handle the asynchronous submission and reply.

The function demo.ajax.submit is invoked by Flex function onItemClick to submit the request to the server. It uses the JSF 2.0 JavaScript function jsf.ajax.request to submit an asynchronous request using the hidden form with the following options:

- The payload is sent as the pass-through request parameter named input.
- It instructs the server to render the child outputText named in the form.
- The server response would be processed by event handler demo.ajax.onevent.

The demo.ajax.onevent handles the Ajax submit events. Upon success, it fetches the response from the *outputText* node, and calls the refresh method exposed by Flash. It works around browser differences by trying to fetch the node text in different ways.

On the JSF server side, add the JSF managed bean source code snippet to process the submission (see **Listing 11**).

In the consuming JSF page, we first add jsf.js to the page head to enable JSF

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java.net
blog

JavaScript inside the page. Here is a JSF page source code snippet (index.xhtml):

```
<h:outputScript
  library="javax.faces"
  name="jsf.js" target="head"/>
```

We need to further map the request parameter input as well as the response attribute exposed by our custom chart tag. There are several options to do this. One way is to utilize a JSF 2.0 enhancement that allows EL action binding to take variables (see **Listing 12**).

Another way to do it is to leverage view parameters. You can map a request parameter to an EL expression via view parameters, as seen in **Listing 13**.

Each approach is interesting in its own right. The first one involves fewer configurations. The second one relies on the view parameter, which is an editable value holder and can take converters and validators. When complicated encoding is needed, the second approach is best.

Integrating with JavaFX

We can similarly implement the chart application in JavaFX (see Figure 2).

Listing 14 shows the JavaFX source code (demo.piechart.Main.FX). The pie chart data is provided by function getChartData(), as seen in **Listing 15**.

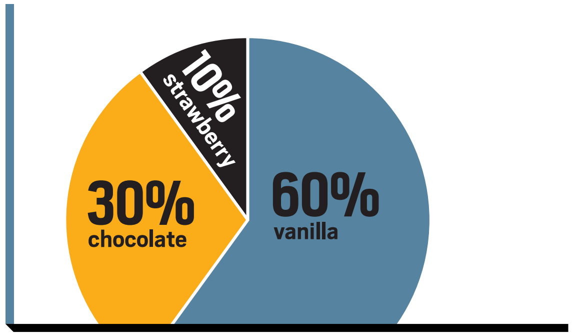


Figure 2

The code is intentionally similar to our Flex application. The chart is populated by a runtime argument named data. We use AppletStageExtension to invoke the container page's JavaScript function demo.ajax.submit. For JavaFX, it is easy to expose the callback function refresh. All script-level public functions are automatically visible to JavaScript in JavaFX.

To embed the JavaFX applet, copy SampleChartFX.jar and SampleChartFX_browser.jnlp into the resources/demochart folder of our Web content. Note the generated jnlp file by NetBeans points, by default, to a local codebase. Because we will specify the jar file location in our Web page anyway, simply remove the codebase attributes from the jnlp file.

Afterward, we just need to make minor changes to our JSF composite component to embed the JavaFX applet, as shown in **Listing 16**.

Most of the JavaScript code would continue to work for our JavaFX applet. The only change is how JavaScript calls back into JavaFX. Inside the demo.ajax.onevent function, instead of chart.refresh(response), it should be chart.script.refresh(response). To allow the code to work for both situations, use this:

```
chart.refresh?
   chart.refresh(response) :
   chart.script.refresh(response)
```

That's it. There is no need to change the consuming JSF page. Whether JSF or JavaFX is used to provide the chart is an implementation detail and is totally transparent to the consuming page.

LISTING 12 / LISTING 13 / LISTING 14 / LISTING 15 / LISTING 16

```
<demo:chart data="#{iceCreamSurvey.result}"
   response="#{iceCreamSurvey.reply(param.input)}" />
```

See all listings as text

Conclusion

In this article, we took advantage of new features in JSF 2.0 to integrate Adobe Flex and JavaFX into our JSF applications. These new capabilities free us from the need to take care of plumbing on encoding, decoding, and view state tracking. In particular, we created a custom component to encapsulate the embedding of Flex and JavaFX. ●

LEARN MORE

- JavaFX
- Adobe Flex
- JavaServer Faces 2.0 download

KEVIN NILSON

BIO

Why Automated Testing for Web Apps?

Java Champion Kevin Nilson talks about his language-agnostic testing toolset.

Kevin Nilson has been building complex Web applications for most of his career. He has realized that automated developer testing is the key to building quality Web applications. Over time, Nilson has pulled together the best open source tools and he has written custom tools to decrease the time needed to develop high-quality applications by making testing a part of the coding process. As he puts it, "It is easier to fix code while you are creating your applications." Here, he talks with Java Magazine *about his toolset, which is language agnostic and integrated with tools such as TestSwarm, QUnit, jQuery, Hudson, GlassFish Server Open Source Edition, and Sun SPOT Java Development Kit.*

Why are testing, in general, and automated testing run by developers, in particular, important?

Prioritizing testing in your developer environment is critical for creating and maintaining quality software. The cost of fixing bugs is directly proportional to how early they are found. If you find a bug five minutes after you write a line of code, you can very quickly fix the bug. On the other hand, if you find a bug after it is two weeks old, it will be much harder for you to fix the bug. After two weeks, you might even have forgotten why you added the code in the first place.

I always recommend adding full testing coverage for all new features. Without automated regression testing, bugs will haunt you. It is very common for bugs that have been fixed to come back again later. Every time a bug is found, you should start by writing a test that exposes the bug and fails. After a test that fails is in place, you can then fix the bug. Once the bug is fixed, the test should pass and you will know that you have a test that will fail if the bug comes back. Over time, you can easily create a large regression of tests without noticing the investment. In fact, over time, you will gain more from the tests than you have invested in writing the tests.

As your project's complexity increases, it takes more time to manually verify your product. During most projects, requirements change drastically for complex systems. Often, when you change or enhance a large system, you end up inadvertently introducing bugs.

Automated testing run by developers during the development process can greatly speed up the rate of development. Developers can run automated tests while they are coding to see what features are not working correctly and to see what bugs were introduced. The same tests can be run later as verification prior to a release.

I spend most of my time writing frameworks in Java and JavaScript that are used by several projects and many development teams. When writing a framework, you often don't know exactly how the framework will be used. Having strong automated tests allows me to use the framework to test scenarios that are not currently part of a product but might be someday.

What unique challenges do you face when testing Web applications?

Over the last few years, HTML5 and Web 2.0 have led a trend in Web applications moving logic from the server to the browser. The biggest challenge in writing Web applications is that your application must run on many platforms and many browsers. Each browser behaves nearly the same as the rest, but many browsers have bugs that cause incompatibilities.

When writing a Web application, it is very important to determine what browsers you want to target. Then you must thoroughly test your application on those browsers. One of the biggest

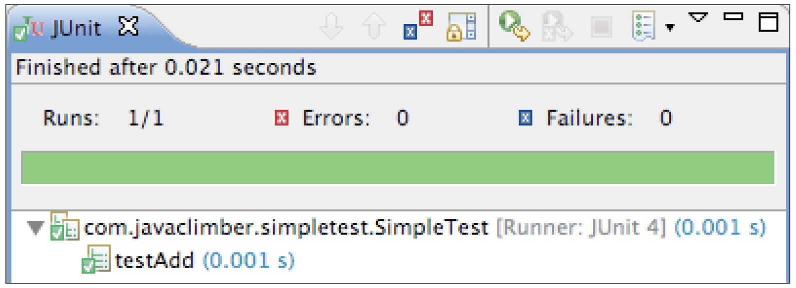
PHOTOGRAPH BY
MARGOT HARTFORD

advantages of Web applications is that you can write an application once and have it reach a wide range of users with no installation or configuration. Your users can be using new or old Microsoft Windows, Mac, Linux, or Oracle Solaris systems, and they can even be using mobile devices, such as iPhone, Android, or BlackBerry.

Another challenge for many Java developers is working with a dynamic language, such as JavaScript, which runs in a browser. Dynamic languages provide great flexibility and efficiency, but they introduce the possibility of several types of runtime bugs. Java developers are generally accustomed to working with static languages, such as Java, C, and C++. With JavaScript, many bugs are caught at runtime that would have been caught at compile time with Java. One simple example of such a bug would be adding a Boolean to an integer. Another simple example would be misspelling a variable. Both of these bugs would be caught by the Java compiler, which would prevent the application from being built, but they would not be caught with JavaScript.

What tools do you recommend using for writing tests for Web applications?

I recommend doing static code analysis and then writing unit and integration tests. JSLint has been a popular tool for years for doing static code analysis, and it helps find problems in JavaScript code.

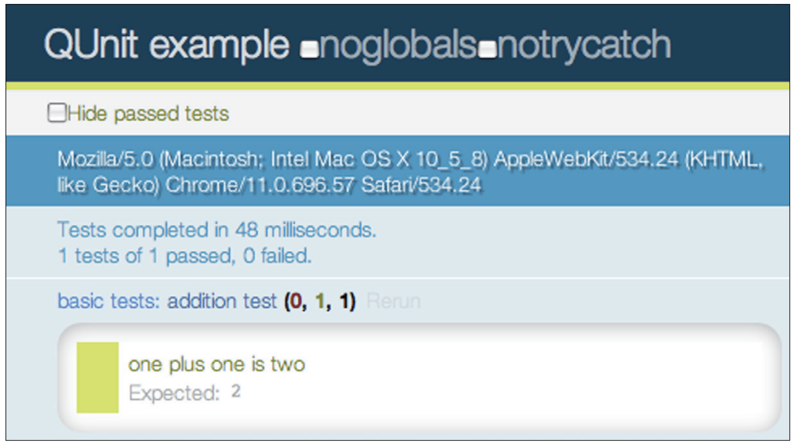


Finished after 0.021 seconds

Runs: 1/1 ☒ Errors: 0 ☒ Failures: 0

com.javaclimber.simpletest.SimpleTest [Runner: JUnit 4] (0.001 s)
testAdd (0.001 s)

Figure 1



QUnit example ■noglobals■notrycatch

☐Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.57 Safari/534.24

Tests completed in 48 milliseconds.
1 tests of 1 passed, 0 failed.

basic tests: addition test (0, 1, 1) Rerun

one plus one is two
Expected: 2

Figure 2

The Closure Compiler is a newer tool that can be used to compile your JavaScript into JavaScript that will download and run faster. The Closure Compiler parses JavaScript to remove dead code, rewrites code, checks syntax, checks variable references, checks types, and warns about common JavaScript pitfalls.

I recommend using QUnit and TestSwarm to unit test and functional test Web applications. QUnit is the test suite that is used by the jQuery project to test its code and plug-ins. TestSwarm allows you to run your QUnit tests in any browser that can connect to the code you want to test.

LISTING 1 LISTING 2

```
@Test
public void testAdd() {
  int a=1;
  int b=1;

  assertEquals("one plus one is two", 2, add(a,b));
}
```

See all listings as text

Are QUnit and JUnit similar?

QUnit and JUnit are similar tools that help you test code. JUnit has been a very popular framework for testing Java code. A basic JUnit test that tests the Java add method is shown in **Listing 1**.

The results look like what you see in **Figure 1**.

Tests in JUnit are marked by using the annotation @Test. The class org.junit.Assert has several static assertion methods to help tests, such as assertTrue, assertFalse, assertEquals, assertNull, assertNotNull, and so on. These assertion methods are used to compare expected results to actual results. JUnit tests are run by developers in their IDEs and can be run by build tools such as Maven, Gradle, and Ant.

Listing 2 shows an example of the same test written in QUnit to test the add function in JavaScript.

The results look like **Figure 2**.

You can group tests into modules to provide some logical separation of your tests. Each test can contain several assertions. QUnit provides assertion functions such as ok, equals, notEqual, and so on. QUnit tests are run in the browser, and the results are shown on a Web page. This allows you to run your QUnit test in any browser just by typing the URL of the test into your browser.

Writing QUnit tests can be challenging because there are no threads in JavaScript. Often you will want to test what happens after the user takes an action, such as clicking a button. When the user clicks a button, long-running tasks and asynchronous events, such as Ajax, might take place. This can cause problems, because the testing code executes before the code behind the action of the button executes. The testing code must release the thread and re-execute later.

One way to work around this problem is to use setTimeout to schedule your assertion after a certain amount of time. However, even this doesn't work well, because your code might not be completed when setTimeout executes. Polling can be used to check for completion of the code you are testing. This polling must also be configured to time out after a certain period.

A company called appendTo developed an open source jQuery plug-in, whenAvailable, that can be used if you are using jQuery. The whenAvailable plug-in continues polling for a Document Object Model (DOM) element before proceeding.

You can also use FuncUnit to help with the challenges of functional testing. FuncUnit is an add-on to QUnit that provides functional testing capabilities. FuncUnit exists to solve the problem of waiting for an element to appear before continuing your test.

How can I automate running tests in all browsers?

TestSwarm is a Mozilla Labs project that provides distributed continuous integration testing for JavaScript. Browsers can connect to the TestSwarm server to become part of the swarm of browsers that tests will be run against.

To run a test, you submit a simple form telling TestSwarm what tests to run and what browsers to run the tests in. The tests are run in an iFrame in each browser that is part of the swarm. Once each test is completed, the results of the tests are sent to the TestSwarm server. TestSwarm is a PHP application that manages the process of telling the connected browsers in the swarm to run tests. The TestSwarm server keeps track of the results of tests that have been run.

SCARY THOUGHT

Without automated regression testing, **bugs will haunt you.** It is very common for bugs that have been fixed to come back again later.

Most continuous integration tools for JavaScript try to launch browsers. TestSwarm is different because it lets any browsers on the network connect to it. This provides a great advantage when you are trying to test multiple browsers and platforms. To join the swarm with your iPhone or Android, all you need to do is go to the Web page of the TestSwarm server with your browser. TestSwarm can be used to run tests written in QUnit (jQuery), UnitTestJS (Prototype), JSSpec (MooTools), JSUnit, Selenium, and Dojo Objective Harness.

What tools are you using to provide continuous integration?

I have been using Hudson/Jenkins on GlassFish Server Open Source Edition to trigger running tests in TestSwarm. My team and I wrote a Hudson plug-in to integrate Hudson with TestSwarm. Hudson monitors my repository for code check-ins. When a code check-in occurs, Hudson uses the plug-in I developed to submit a new job to TestSwarm. The plug-in then polls TestSwarm for the results. If there are errors in the test, Hudson sends out an e-mail notification.

I am using Oracle VM VirtualBox to help manage the browsers that are connected to TestSwarm. Oracle VM VirtualBox is used to run four operating systems that run 10 different browsers. Oracle VM VirtualBox has a Web Service API that can be used to stop and start virtual environments. My team and I wrote another Hudson plug-in to restart the virtual environments once a day.

Hudson provides a REST-style API that has an XML API that shows the status of builds. I am using Sun SPOT Java Development Kits to poll Hudson's XML API. Each Sun SPOT Java Development Kit has eight LED lights, so I can monitor eight builds at a given time.

How difficult would it be for someone to set up a testing environment similar to what you have built?

It should be fairly easy to set up a similar environment. My team and I were able to get a basic setup going in a few days. Over time, we have slowly added more "nice to have" features. All the tools I am using are free and open source.

What are some other tools that can be used for testing Web applications?

Before I started working with QUnit and TestSwarm, I mostly used Canoo WebTest. Canoo WebTest allows you to write your tests in XML or Groovy. Canoo WebTest runs from the command line using Rhino.

Selenium is another great tool that has been very popular for testing Web applications. Selenium has a click-and-record feature that allows you to write simple tests easily. I prefer using QUnit because of its simplicity and power. QUnit is a great tool for JavaScript developers. QA developers will probably be more comfortable with Selenium.

What Java and Oracle technologies have you been using to assist with testing?

I am using Hudson, GlassFish Server Open Source Edition, Oracle VM VirtualBox, and Sun SPOT Java Development Kits. Each of these tools is open source and very flexible. I have been able to integrate these tools with other open source testing tools to provide an end-to-end automated testing environment for Web applications. ●

LEARN MORE

- TestSwarm
- QUnit
- jQuery
- GlassFish Server Open Source Edition
- Sun SPOT Java Development Kit
- JSLint
- Closure Compiler
- JUnit
- Maven
- Gradle
- Ant
- appendTo
- FuncUnit
- Hudson
- Jenkins
- Oracle VM VirtualBox
- Canoo WebTest
- Rhino
- Selenium

ADAM BIEN

BIO

PHOTOGRAPH BY
THOMAS EINBERGER/
GETTY IMAGES

Resource Injection with Java EE 6

Learn about the various annotations that are available for resource injection with Java EE 6, why they are needed, and when they can be used.

Should you use @Resource, @Inject, @PersistenceContext, or plain old Java Naming and Directory Interface (JNDI) lookup? Java Platform, Enterprise Edition 6 (Java EE 6) offers multiple possibilities for injection of configured resources, such as a datasource, destination, Java Persistence API (JPA) EntityManager, Java Transaction API (JTA) UserTransaction, URL, J2EE Connector Architecture (JCA) connector, mail session, LDAP connection, and even a custom resource installed in JNDI. So why do we need several annotations for resource injection?

This article describes the various annotations that are available for resource injection with Java EE 6, why they are needed, and when they can be used.

@Resource—The Generic JNDI Resource Injector

The @Resource annotation is defined in JSR-250, "Common Annotations for the Java Platform." This specification also includes other well-known annotations such as @PreDestroy, @PostConstruct, and @RolesAllowed. While JSR-250 is required in Java EE 5 and Java EE 6, it is defined as an independent Java EE specification and, thus, it can be used by any other non-Java EE frameworks or libraries. Some of the JSR-250 annotations are even packaged with Java Platform, Standard Edition (Java SE). This is the case for @Resource.

The @Resource annotation is intended for injection of all resources installed into the JNDI namespace. The JNDI name is used as an alias for the configured resource. Usually, the specified API will be injected as an interface. JNDI decouples the user of the resource from its actual implementation and configuration.

```
public class
ControlWithDataSourceDI {
    @Resource(name="jdbc/sample")
    DataSource ds;
}
```

The name element specifies the actual JNDI name. The datasource in the above was configured and injected with the JNDI name jdbc/sample. Because of the ubiquitous "Convention over Configuration" principle in Java EE 6, the name element does not need to be specified. If it is not specified, the JNDI name is derived directly from the field name. This would be difficult in our case. The code would look like @Resource DataSource jdbc/sample, and it would not compile. The change of the application server configuration would break the code and require the field to be renamed. In this particular case, it is better to name the JNDI name explicitly. Renaming the JNDI resource would affect only the name element and not the field name.

The use of mappedName should be avoided. It is proprietary and depends on the application server implementation. Injected resources can be shared and can be configured with the shareable element, which is set to true by default. Most of the resources are either immutable (such as injected primitive types or injected URLs) or resource factories (such as DataSource), and so they are shareable.

The @Resource annotation supports field and setter injection. Field injection is leaner, because it does not require you to implement a superfluous setter. Admittedly, you will need to lose the field visibility to package visibility to allow a mock-out of the injected class during a unit test.

@DataSourceDefinition—A Touch of DevOps

Most of the resources are installed on applications in an unspecified way using admin

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java .net
blog

consoles, Java Management Extensions (JMX), command-line interfaces, or even Representational State Transfer (REST). A datasource, however, can be configured in a standardized way. The @DataSourceDefinition annotation introduced with version 1.1 of the JSR-250 specification allows the configuration, installation, and JNDI exposure of a datasource in a portable way, as seen in **Listing 1**.

One or more @DataSourceDefinition annotations declared on a class (potentially enclosed with @DataSourceDefinition annotations) and deployed with the application provide the necessary information for automatic installation. The datasource can be injected directly by using the JNDI name specified in the name element with the @Resource annotation. It is also accessible for a manual lookup. Direct access to a datasource in a typical application is necessary only for accessing stored procedure invocations or specific optimizations and is rather uncommon. The vast majority of all persistence use cases can be handled by JPA. The JPA EntityManager, however, requires a registered datasource with a well-known JNDI name, configured in the persistence.xml configuration file.

The majority of Java EE applications can be installed without any administrative tasks with @DataSourceDefinition. A properly installed JDBC driver on the server is the only prerequisite for a successful installation. The obvious drawback here is the loss of flexibility. You need to redeploy the application for configuration changes.

At first glance, the need to redeploy seems like a disadvantage, but the deployment of self-contained applications is the central idea of DevOps. Here, the operation and development of the application are considered as a single and consistent unit. Applications are built, configured, and installed continuously without any manual intervention. In this scenario, there is no difference between the application server, the operating system configuration, and the application code. All the information required to install or run the application is treated equally.

@PersistenceContext—A Special Case

@PersistenceContext was introduced in Java EE 5 with the JPA specification and not as part of JSR-250. Although an EntityManager could be considered an unshareable resource (and EntityManagerFactory could be considered a shareable resource), it cannot be injected with the @Resource annotation without nasty workarounds. To inject an EntityManager with the @Resource annotation, you need to register it in the JNDI namespace first. The registration of EntityManager in the JNDI namespace can be accomplished by applying the @PersistenceContext annotation on the class level.

JAVA FACT

The vast majority of all persistence use cases can be handled by JPA.

The element name binds the EntityManager to the JNDI name. The JNDI scope is assigned by adhering to a predefined naming convention. For instance, java:comp/env exposes the EntityManager to the component namespace. To fetch a resource from the local component namespace, the lookup element from the @Resource annotation needs to be used, as shown in **Listing 2**. Also, exposure to the global JNDI namespace (java:global) is possible. The JNDI name must be prefixed with java:global for this purpose. For the injection, the name element of the @Resource annotation can be used, as seen in **Listing 3**.

LISTING 1 LISTING 2 LISTING 3 LISTING 4

```
@DataSourceDefinition(
   className="org.apache.derby.jdbc.ClientDataSource",
   serverName="localhost",
   name="java:global/jdbc/InjectionSample",
   databaseName="InjectionSample;create=true",
   portNumber=1527,
   user="sample",
   password="sample"
)
@Stateless
public class JDBCDataSourceConfiguration {
  @Resource(lookup="java:global/jdbc/InjectionSample")
   private DataSource dataSource;
}
```

See all listings as text

After registration of the EntityManager in the JNDI namespace, a direct lookup with a SessionContext works as expected, as seen in **Listing 4**.

For a manual lookup, the SessionContext needs to be injected with @Resource first. The creation of the InitialContext with the default constructor is equally possible. The EntityManager can then be obtained with the registered JNDI name. The manual lookup should be performed during the initialization time in the @PostConstruct annotated method. A JNDI lookup of an EntityManager is rarely needed in a Java EE 6 application. An EntityManager can be directly injected into an Enterprise JavaBeans (EJB) 3.1 bean, as well as into a Contexts and Dependency Injection (CDI) managed bean. A manual lookup might still be interesting for nonmanaged components.

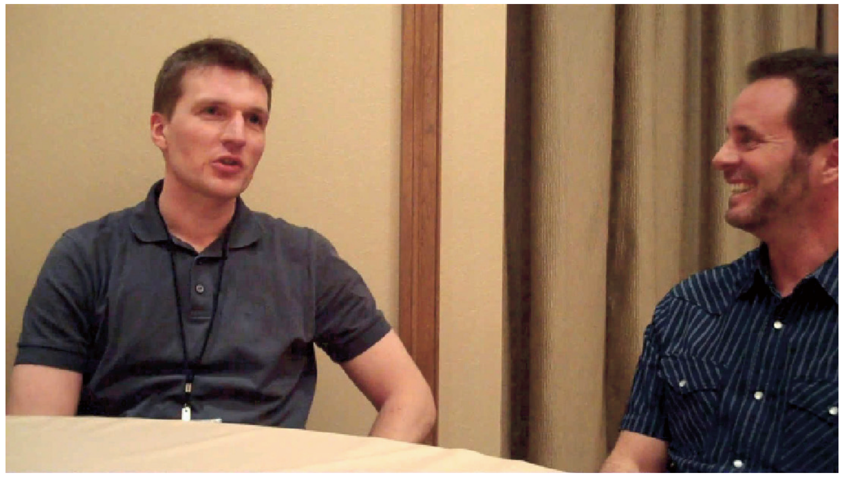
Although an EntityManager injection works also for CDI managed beans, the beans cannot be directly exposed to the UI layer. The EntityManager in a stateless environment can be configured only with the @PersistenceContext(type=

PersistenceContextType.TRANSACTION) annotation, which is also the default value. Every interaction with the EntityManager requires, therefore, an active transaction; otherwise, a javax.persistence.TransactionRequiredException is thrown. Transactions cannot be started in CDI managed beans out of the box. An EJB 3.1 Stateless Session Bean solves the problem in the simplest possible way because neither manual transaction management nor the use of CDI extensions is required in an EJB Stateless Session Bean. A single, no-interface view bean, such as a facade, manages the transactions without any further configuration, frameworks, or manual coding.

Why Not Just Use @Inject?

The @Inject annotation was introduced with JSR-330 ("Dependency Injection for Java") and is an integral part of Java EE 6 (in fact, it's even part of its Web Profile). "Contexts and Dependency Injection" (JSR-299) greatly enhances the dependency injection capabilities of the Java EE 6 platform and relies on the minimalistic JSR-330. Unfortunately, the plain @Inject annotation is not suitable for any direct resource injection. The reason is the lack of elements. The name of the field could be still leveraged as a JNDI name. Because of naming limitations in the Java language, particular patterns, such as jdbc/sample or queue/Orders, cannot be expressed with a field name. Furthermore, the injection of the EntityManager also needs additional parameters, such as the type (transactional or extended) and the name of the persistence unit.

Nevertheless, @Inject, together with producers and qualifiers, is an interesting option for clean and flexible resource handling. You can centralize the creation of resources in a plain class and inject them in a decoupled and clean way on demand. A single resource type could even be injected without any qualifier. In more-sophisticated projects, the existence of multiple DataSource instances or EntityManager from different persistence units is likely. A custom @Qualifier can be used to match the producer and the corresponding injection point. The @Qualifier annotation with a meaningful name increases the readability and decreases coupling at the same time. The consumer is no longer dependent on the JNDI name nor on the way the resource was actually obtained.



Adam Bien chats with *Java Magazine*'s Justin Kestelyn about Java 7, a typical day, and more.

LISTING 5 LISTING 6

```
public class LegacyDataSourceProducer {
  @Produces @Legacy @Resource(name="jdbc/sample")
   private DataSource ds;
}
```

See all listings as text

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, TYPE})
public @interface Legacy {}
```

A field or resource in an EJB bean or in a managed bean can be used for obtaining the resource and its "production" by using @Produces at the same time. The @Qualifier marks the producer and the injection point. If both match, the resource is injected. The injection would also work with the built-in @Named qualifier. The @Named annotation uses a simple String for matching what is not type-safe and could become a source of nasty errors. A misspelled String is hard to find in a larger code base.

The produced resource can be easily injected into the consumer using the @Inject and custom @Qualifier annotations. A resource can be directly exposed with a field or method. A method provides more flexibility—the returned resource could be logged, decorated, or even reconfigured. When using a method as a producer, you need to move the @Produces and @Legacy (see **Listing 5**) annotations from the field to a method of your choice.

To inject a resource, only the @Inject and @Legacy annotations are required, as seen in **Listing 6**. Further configuration is not needed. Also, the code becomes fluently readable: "Inject legacy data source." Because it is a matter of moving two annotations from a field to a "getter," you could start with a field and avoid unnecessary bloat. A method producer can be introduced on demand without affecting the consumers or injection points.

Conclusion

Preconfigured resources installed in JNDI are usually injected with the @Resource annotation. The type of resources injectable with @Resource ranges from a String to a CORBA service, for example, primitive types (String, long, and so on), javax.xml.rpc.Service, javax.xml.ws.Service, javax.jws.WebService, javax.sql.DataSource, javax.jms.ConnectionFactory, javax.jms.QueueConnectionFactory, javax.jms.TopicConnectionFactory, javax.mail.Session, java.net.URL, javax.resource

.cci.ConnectionFactory, org.omg.CORBA_2_3.ORB, javax.jms.Queue, javax.jms.Topic, javax.resource.cci.InteractionSpec, and javax.transaction.UserTransaction. These resources are usually administered on the application server in an application-agnostic way; more than one application could share the same resources.

The EntityManager is not registered in the JNDI namespace, and so it is not available for @Resource injection in a standard case. Furthermore, an EntityManager is configured inside an application and is not shared with other applications. For the injection of the EntityManager, additional information, such as its type and the unitName, is used. Although an EntityManager could be injected with @Resource or @Inject indirectly as well, in the vast majority of cases, a plain @PersistenceContext annotation is used for this purpose. The Java EE ubiquitous principle of "Convention over Configuration" provides suitable defaults. If there is only one persistence unit, it doesn't need to be specified. The EntityManager is injected with transactional configuration without any further ceremony.

The @Inject annotation is not suitable for direct injection of resources from the JNDI namespace without any extensions or workarounds. @Inject together with a custom @Qualifier becomes interesting for the injection of resources exposed by @Produces. The client (injection point) becomes entirely decoupled from JNDI, and this is also the case for any resource creation and lookup logic. This additional layer of indirection is rarely needed in typical projects, but it is very interesting for platform, product, or API development. In the latter case, the user of a particular service need only use the custom qualifier and the @Inject annotation to get the necessary resource injected.

The @EJB annotation cannot be used for injection of resources and is suitable only for the injection of dependent EJB beans. The old-fashioned InitialContext#lookup is needed only in exceptional cases where the JNDI name is unknown at compile time and needs to be provided at runtime. ●

LEARN MORE

- JSR-317: "Java Persistence 2.0"
- "Contexts and Dependency Injection in Java EE 6"
- "Enterprise JavaBeans 3.1 with Contexts and Dependency Injection: The Perfect Synergy"
- "Simplicity by Design"

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java.net
blog
41

FLEX on JAVA
Java The Good Parts
SQL
Java for Programmers
Clean Code
Java and Flex Integration Bible
Learn Java for Android Development
Effective Java Second Edition
Android Wireless Application Development
Java Programming
Java Fundamentals I and II
BlackBerry

VIKRAM GOYAL

BIO

Working with JSR-211: Content Handler API

Learn how to use CHAPI, which solves a very specific problem in a clean and elegant manner.

PHOTOGRAPH BY
JONATHAN WOOD/
GETTY IMAGES

The "Content Handler API (CHAPI)," also known as JSR-211, is one of those nifty little APIs that doesn't seem to do much at first glance, but the more you look at it, the more you realize how useful it really is. It solves a very specific problem, and it does that in a clean and elegant manner.

In this article, I will help you understand the problem domain this API addresses and what it does to solve the problem. I will give you a rundown of the API's structure and how best to use it in your own MIDlets. Finally, I will show an example of how to use this API with a simple use case.

Note: The source code that accompanies this article can be downloaded here.

The Problem Domain

Consider the case where you are creating an application that will allow budding photographers to browse the image library within their device, which contains images that they might have captured using their device's camera. Instead of using the device's own image browser, they can use your nifty little app, which provides extra tools (for example, a tool for sharing images online, perhaps). When users browse and want to see an image, you want to be able to put some copyright information on the image. Instead of relying on the built-in image viewing capabilities of the device or the Java virtual machine (JVM), you want to make sure that each image opens in a special image viewer that displays this copyright information.

In a nutshell, you want to create a new image content handler that can always display images with your copyright. You could write some tricky code. Or you could use the Content Handler API and register your special content handler for images with the device's Application Management Software (AMS).

The Content Handler API

CHAPI provides an execution model that allows your applications to invoke Java Platform, Micro Edition (Java ME) and non-Java applications. What this means is that you register your existing content handlers with this API and the API allows you to invoke them. You, of course, need to provide some sort of identification by which a content handler can be invoked.

For example, let's say you want all JPG images to be handled by a specific image content handler (as discussed earlier in "The Problem Domain"). You register the content handler class either through an entry in the manifest file or, programmatically, by notifying the AMS that JPG MIME types are now to be handled by this special content handler. (If there are multiple handlers registered, CHAPI will pick one of them randomly.) Identification can be done not only by MIME type (or content type) but also by the URL or the content handler ID construct. You can also register multiple content handlers for the same types, and then use the actions to specify which particular handler to invoke.

LOOK DEEPER

JSR-211 is **one of those nifty little APIs** that doesn't seem to do much at first glance.

The Registry and Invocation

The Registry class, as you might expect, is the central repository of all known content handlers within the Java ME environment. It provides the lifecycle methods of all content handlers (namely registration and un-registration), providing meta information and, of course, providing the actual invocation of content handlers. Each content handler is marked using zero or more content types (for example, image/jpeg), suffixes (for example, .jpg), and actions (for example, open). Of course, there is a unique ID associated with each handler, which is used to enforce access controls.

Content handlers can be chained through the registry. Thus, if one content handler cannot complete a request, it may invoke another one before passing back control to the calling (invoking) class. This chaining behavior is handled seamlessly by the registry.

FAST FACT

CHAPI and the execution model let an application invoke registered Java ME and non-Java applications by URL, content type, or content handler ID.

Accessing the central registry is done through the static method provided by this class: getRegistry(String classname). The class name is the name of the content handler or the application that will be accessing the registry methods, and it must have been registered previously by making entries in the Java Decompiler (JAD) file, or it can be registered programmatically using the register method before calling the getRegistry method. Instead of returning a null if the specified registry cannot be found, the getRegistry() method throws an IllegalArgumentException. You use the unregister(String classname) method to un-register a content handler or application from the registry.

An invocation is an encapsulation of the parameters that you might need to pass between your application and the content handler. This can include, but is not limited to, the URL, the type, the action, and the ID, plus whether a response is required from the handler. This encapsulation is identified in the API with the Invocation class. I would like to point out that the invocation is used not just for passing the parameters to the content handler but also for receiving responses from the handler. This class defines several handy status flags to indicate where a request to invoke a content handler is at a particular moment (for example, OK, ACTIVE, CANCELLED, and so on).

Once you have created the registry and the target parameters using an invocation instance, you need to call the registry method invoke(Invocation invoke) to actually initiate the content handler. This method returns a flag mustExit, which indicates whether the calling application should exit before the content handler itself can start.

Thus, a sequence of events for invoking a content handler might look like this:

1. Register the content handler either through JAD manifest entries or programmatically using the register method.
2. Access this handler's registry by using the getRegistry method.
3. Create an invocation request with your parameters using the Invocation class.
4. Create the invocation using the registry's invoke(Invocation invocation) method.

ContentHandler and ContentHandlerServer

The ContentHandler and ContentHandlerServer interfaces sit on the other side of the fence. That is, they provide the means to write your own content handlers. Note that you don't directly implement these interfaces. Implementation of these interfaces is provided by the device manufacturer's own API implementation. You use this implementation to work on your handler code.

The ContentHandler interface is a collection of details about the registration of each content handler (which you might have already registered). For example, methods such as getID() and getType() basically just provide the details that you would have provided at registration.

The ContentHandlerServer interface extends the ContentHandler interface and provides methods to receive new invocation requests, finish the processing of these requests, and provide other meta information. In a nutshell, its implementation provides the lifecycle methods for managing new invocation requests. Therefore, to create your own content handlers, you rely on its methods to manage this lifecycle (for example, queuing requests, responding to new requests, chaining requests, handling errors, and so on). You are thus left to create the code for your content handler, and you leave the overhead to the interface's implementation (provided by the device manufacturer's API implementation).

RequestListener and ResponseListener

Any implementation of a content handler should handle the listeners for when a new request is made and when a response is requested. These listeners are RequestListener and ResponseListener, respectively. By implementing the first interface, the content handler provides an implementation of the invocationRequestNotify(ContentHandlerServer handler) method, which is called automatically when a new request is made by a calling class (or any other application). By implementing the second interface, the content handler provides an implementation of the invocationResponseNotify(Registry registry) method, which is called each time the calling class requires a response from the content handler.

The RequestListener interface is set using the corresponding setListener() method of the ContentHandlerServer interface, while the ResponseListener interface is set using the corresponding setListener() method of the Registry.

AdvancedImageContentHandler: An Example Content Handler Implementation

Following on from the problem domain discussed earlier, in this section I describe an example content handler that will add a text string to the end of each image it displays. The constructor for this class is shown in **Listing 1**.

The AdvancedImageContentHandler constructor uses the registry to set itself as the listener for any responses that might be required, and, therefore, it implements the invocationResponseNotify(Registry registry) method, which is empty in this example. Then, it locates the ContentHandlerServer using the static registry method getServer(String classname). The handler is then notified that all new requests for content handling are to be done by this class using the invocationRequestNotify(ContentHandlerServer handler) method. Finally, it sets up the UI to display the image and the text below it.

When a new request is initiated, the invocationRequestNotify(ContentHandlerServer handler) method is called, as shown in **Listing 2**.

We first check whether there is an existing request, and we let the content handler server finish that up. Next, if there is not an existing request, we get the details of the new request, and we pass the new request on to the displayImage() method with the details.

Although fairly straightforward (in the example shown in **Listing 3**), the displayImage() method does all the magic of displaying the image with the added text.

We opened up a connection to the requested file (using the FileConnection API), and if the image is found, we display it on the form with the added text. We do some error checking to make sure we can load the image, and if not, we display a message accordingly.

On the caller's side, it takes only a three-step process to call this content handler, as shown in **Listing 4** in the excerpt from the calling class (CHAPIExample).

In Step 1, we create the registry using the getRegistry(String classname) method and pass it the class name of our content handler, AdvancedImageContentHandler. In Step 2, we create the data for the invocation. In the final step, we invoke the content handler using the invoke(Invocation inv) method of the registry with the invocation data we created in Step 2.

Before we can do all this, we need to register our content handler in the JAD file (or programmatically, as the case may be). Our JAD file entries look like **Listing 5**.

Notice the last three lines. These tell the AMS that for the content type of image/jpg or the .jpg extension, the AdvancedImageContentHandler class will be the handler. The AMS then registers this within the registry, and this class is available for handling open commands.

Note: When running the example code, make sure that the images are placed in the root folder. I used DefaultCLDCPhone1 as the emulator, and the root for that within a Windows Vista environment with SDK 3.0 is C:\users\username\javame-sdk\3.0\work\devicenumber\appdb\filesystem\root1. ●

LEARN MORE

- Read the final release of the JSR-211 API at the Java Community Process (JCP) Website
- Download Java ME

LISTING 1 / LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5

```
public AdvancedImageContentHandler() {

  // notify the registry that this class is a listener
  registry = Registry.getRegistry(this.getClass().getName());
  registry.setListener(this);

  // now, get the handler which was registered by making entries in the JAD

  // file
  try {
    handler = Registry.getServer(CH_CLASSNAME);
  } catch (ContentHandlerException che) {
    System.err.println("Registration not done! Check JAD file");
  }

  // this class is the handler for all new requests
  handler.setListener(this);

  // setup the ui
  display = Display.getDisplay(this);
  form = new Form("Advanced Image");
  backCommand = new Command("Back", Command.BACK, 1);
  form.setCommandListener(this);
  imageItem = new ImageItem(null, null, Item.LAYOUT_CENTER, "--");

}
```

See all listings as text

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java.net
blog

DICK WALL

BIO

PHOTOGRAPH BY AN NELISSEN

Scala on the Java Virtual Machine

What does the Scala programming language tell us about the strengths—and limits—of the JVM?

Scala is an object-oriented language that is interesting for a number of reasons. Like .NET's F#, it is a functional and object-oriented hybrid. It is more statically typed than Java, with a far more extensive type system. It has support for function literals and closures, mixins (called traits), properties, tail call optimization, structural typing (a sort of static duck typing), pattern matching, new idioms for dealing with concurrent execution, and many other features. It is also fair to say that Scala pushes the JVM hard, perhaps harder than any other mainstream language (particularly if you consider the type system), and this makes it an ideal spyglass through which to examine how the JVM copes with tasks that were often not even conceived when it was first created.

The features I consider in this article are

- Symbolic method names (frequently mistaken for operator overloading)
- Tail call optimization
- Function literals and closures
- Traits
- Implicit manifests
- Pattern matching

This is far from an exhaustive list of Scala features that exceed Java's own feature set, but this list is a set of the features that push the JVM in some manner. Some of them just work; others run afoul of limitations that cause the solution to be less than ideal—but all of them work. It is a testament to the design of the JVM that it has held up as well as it has against a barrage of difficult jobs it was never designed to do.

Symbolic Method Names

It is a common misconception that Scala has operator overloading. Certainly it offers a combination of two features that make it look that way, but the truth is simpler and more consistent.

Infix operators. Any Scala method that takes a single parameter may be called in two different ways. **Listing 1** shows a simple example.

Note: In the following example and in subsequent examples, bold indicates Scala keywords and italics indicates output from the Scala shell.

In this example, we create a `java.util.ArrayList` of integers. (Scala has its own collections, and very good ones, but more on that later.)

Next we add the integer value 1 into the `ArrayList` using the `add` method; this form looks just like Java.

But then we change things up a bit. The next line adds the integer 2 to the list, but this time, we are using the infix operator form. Because `add` takes only one parameter, we can do this in Scala. The effect is exactly the same as calling `a.add(2)`; only the syntax is different. This is the first part

The Java Virtual Machine

When Java and the Java virtual machine (JVM) made their first appearance in the early 1990s, the concept of a virtual machine was not new. A virtual machine provides an execution environment for compiled code that targets a hypothetical machine language and can span many "real" hardware platforms to provide a common runtime for the compiled programs.

the '90s

The virtual-machine idea dates back to at least 1966 with O-code (used for the language BCPL), but over time, it gained a reputation for slow code execution. The JVM, along with the just-in-time (JIT) compiler that came with Java 2, has mostly cleared that speed stigma, although it continues in some quarters. What Java and the JVM did was break virtual machines, and managed languages that run on top of them, into mainstream software development, particularly in the enterprise space. This was never clearer than when Microsoft adopted the Common Language Runtime (CLR) and .NET as its developer strategy around 2001.

1966

of our apparent operator overloading feature.

Symbolic names. Scala method and function definition names are not limited to alphanumeric characters. In fact, very few names are off-limits (although you can't use names that start with numbers, because they are parsed as numeric literals). A single underscore may not be used, and both = and == are reserved to avoid confusion, but most other symbols can be used, as can _ and = as long as they are not alone, as seen in **Listing 2**.

ArrayBuffer is a Scala collection very similar to ArrayList in Java; however, it has methods with names such as +=. There is nothing special about these methods other than they have symbolic rather than alphanumeric names. In fact, the append() method on ArrayBuffer works just as the += method does. You can see this in the call to add 7 to the ArrayBuffer near the end of the example. We can call the method just as we would call any other method. The form ab.+=(7) is just like ab.append(7).

When you couple infix operator notation and symbolic method names, you end up with something that looks a lot like operator overloading, but it really isn't.

Critics often say that operator overloading is evil, and that it has led to all sorts of horrors in languages such as C++. While that might be true, I would simply ask, is it any more wrong to have a method called add that deletes everything in your collection than to have one named + that does the same thing?

Tail Call Optimization in Scala

At present, Scala's abilities for tail call optimization are quite limited. It works only for a function that calls itself, not for mutually calling functions or more-complicated arrangements. Because of features such as inheritance, methods must be final or private to be optimized as well, and there are other restrictions.

Support from within the JVM for tail call optimization would be quite helpful. While some JVM engineers claim that the language can and should do it, and the JVM doesn't need to be involved, I believe the truth is a little more complicated. The JVM has some advantages over a compiler, not the least of which is that it has access to runtime profiling information. This is information that the JVM uses very effectively already for lots of other optimizations. Tail call support in the JVM would enable it to figure out whether optimization is even necessary, whether it is possible given the classes involved, and other useful contextual information. For example, it could find optimizations based on the classes that are actually loaded into the running VM, rather than optimizations based on what people might write in other code, which is what the compiler would have to guess.

In short, while Scala might be able to do more for itself with tail recursion, some kind of support within the JVM is likely to make the feature even more useful and complete. The truth is that some kind of joint effort between the compiler and the JVM is likely to yield the best results.

LISTING 1 LISTING 2

```
scala> val a = new java.util.ArrayList[Int]
a: java.util.ArrayList[Int] = []

scala> a.add(1)
res0: Boolean = true

scala> a add 2
res1: Boolean = true
scala> a
res2: java.util.ArrayList[Int] = [1, 2]
```

See all listings as text

One more note: Symbolic names are not limited to ASCII characters either. Mathematicians like to use Greek notation a lot. The following is perfectly valid in both Scala and in the underlying JVM:

```
scala> def Σ(numbers: Seq[Int]) =
  numbers.sum
Σ: (numbers: Seq[Int])Int
scala> Σ(1 to 10)
res16: Int = 55
```

We just created a function with the symbolic name Σ (sigma), which is used in math notation to denote a sum. We used it for exactly this purpose, to sum a sequence of numbers (integers in this case, but it could be made more general). Also note the infix operator used in the 1 to 10 expression. Before you decide that this is unnecessary for a programming language, why not ask a few mathematicians what they think of the idea?

Tail Call Optimization

Consider the following Scala code:

```
def factorial(n: Int, acc: Long = 1): Long =
  if (n <= 1) acc else factorial(n - 1, acc * n)
```

It is the venerable factorial function, implemented recursively.

There are a few things to note about the implementation:

- We pass an accumulator to hold on to the current factorial value down through the recursion.
- If the accumulator (acc) parameter is not supplied, the accumulator value defaults to 1 (the starting value).
- The accumulator (acc) is multiplied by n before the recursive call to factorial.
- If n is less than 2, we simply return the acc value.

We could do an implementation that appears simpler and looks like this:

```
def factorial(n: Int): Long =
  if (n <= 1) n else n * factorial(n – 1)
```

Indeed, this is pretty much the mathematical definition of factorial, but there is a reason we don't go with the simpler-looking one.

In the first version, by putting the multiplication in the parameter list for the recursive method call, the multiplication of the accumulator by n happens before the recursive call to factorial. In the second version, the recursive call must be evaluated first, and then the multiplication by n is carried out after that.

This puts the recursive call in the first example into something that functional programmers call the "tail" position, as in, it is the last thing that the function does.

Scala can then turn the recursive function into a looped one automatically. Because the last thing the function does is call itself, the whole thing can be looped, and this means a lot less work for the runtime. Recursion, while very clever, carries some fairly heavy overhead. Stack frames must be created for each call, and the stack can overflow if the recursion is deep enough. Beyond that, the JVM can do a lot to optimize looped code (tricks such as inlining, variable and register optimization, and so on), but the JVM doesn't always get a chance to do this with recursive code.

Function Literals and Closures

Much has been said about the inclusion of closures in the Java language, both for and against. It seems fairly certain that closures and function literals will be included in Java 8, but that is still a year or two away.

In the meantime, many (if not most) leading alternative languages for the JVM include function literals and closures.

You might have noticed that I keep referring to function literals as a concept distinct from a closure. It is. A closure is a kind of function literal that encloses some values or variables from the surrounding scope.

When either of these is needed in Java, inner or anonymous inner classes provide the same features. These work, but there is a lot of boilerplate code involved, and it often ends up being too much extra code to make the effort worthwhile. For example, this is a simple function literal in Scala:

```
scala> val primes = List(2,3,5,7,11,13)
primes: List[Int] = List(2, 3, 5, 7, 11, 13)
scala> primes.map(n => n * 3)
res0: List[Int] = List(6, 9, 15, 21, 33, 39)
```

This would expand to enough code in Java using an anonymous inner class that most developers would just fall back on a for loop to do the same operation. Of course, then you have to create another list to hold the results, and code to add the results (or you change the values in place in the list). All of these alternatives have their own costs, either more code or mutable state (which eventually might lead to problems in concurrent systems).

The use of Java anonymous inner classes to do what a function literal or closure would do is significant, because that's how they are implemented in Scala. This leads to quite an explosion of classes being generated when you compile a Scala file (each closure gets its own class generated). It is possible that the method handles being added into JDK 7 could help reduce the number of extra class files that are generated, and it might speed up compilation as well as reduce the size of the compiled binaries.

On the other hand, this is an area where the JVM delivers fairly well right now. The biggest liability with closures in alternative languages is that there is currently no standard approach, so Scala closures are not likely to be compatible with Groovy closures or JRuby closures. Perhaps when closures are available in Java 8, that will be the standard to which all other implementations will adhere.

Traits

Java bucked the trend toward multiple inheritance when it came out. Multiple inheritance is very powerful, but it brings some issues along, such as which method in which superclass is actually being referred to (the diamond inheritance problem).

Comparing CLR to the JVM

It is interesting to compare Microsoft's CLR to the JVM. From the start, CLR was intended to be source-language agnostic, and it was launched with a variety of supported source code languages, including C#, J# (a Java-like language), and VB.NET. Since then, a number of third-party source language options, such as Iron Python and Iron Ruby, as well as new Microsoft-supported languages, such as F#, have been added. The JVM, by contrast, has always concentrated on supporting the Java programming language first and foremost, but this has not stopped other languages from targeting the JVM.

In fact, the JVM has many different source languages that target its bytecode execution, drawn by the promise of easy cross-platform support and a runtime that is installed on a large number of machines and devices. Robert Tolksdorf and his group, is-research, have long provided an exhaustive list of hundreds of languages targeting the JVM.

Although there are many languages running on the JVM, not all of them do well with the limitations of the JVM, which was not designed to support the wide diversity of language features that the union of all these languages represents.

The most successful languages tend to work well with the features that the JVM provides, and they find clever ways to work around some of the limitations. Many of these languages have started to move to the forefront of the new generation of languages for the JVM.

These languages include Ruby (JRuby), Python (Jython), Mirah (like Ruby with static typing), Clojure, Groovy, Fantom, and others.

However, the language this article focuses on is called Scala. It is the brainchild of Martin Odersky, who has an intimate knowledge of the JVM and the Java language. In fact, the javac compiler used for Java 1.3 was based on Odersky's GJ compiler. GJ was an experiment to extend the type system in Java with generics (although Odersky is always quick to point out that wildcards were not his idea).

Java made a restriction that there could be only one superclass for any class, but there could be many interfaces implemented by each new class defined, and that would give many of the benefits of polymorphism without the associated issues.

In the time since Java was created (and even in some cases before that), another strategy known as *mixins* provided a safer alternative to full multiple inheritance. Mixins are like Java interfaces that can still have behavior defined on them. Any given class still has only one actual superclass, but it can also mix in other, richer aspects, providing more value than just interface definitions (which then need to be satisfied with code implementations).

Scala calls these *traits*, and it has good support for them. Surprisingly, they work just fine in the JVM, even though the JVM was never designed to support them. Behind the scenes, both an interface and a class with the behavior and state necessary are created when you define a trait, and a clever hookup by the compiler makes the whole thing work pretty well. **Listing 3** shows an example of traits in Scala.

And here's an example of use:

```
scala> val kermit = new Frog
kermit: Frog = Frog@1dfela
scala> kermit.move
I move using 4 legs
scala> kermit.color
res2: java.lang.String = Green
scala> kermit.swims
res3: Boolean = true
```

The class Frog has only one actual superclass, but it mixes in the Green and HasLegs traits. The HasLegs trait brings with it the need to supply a number of legs before the class can be instantiated, so this is filled in when we define the Frog class.

Traits are a fantastic feature in Scala, and they lead to a lot more reuse than is generally possible in Java. They can also be used for a lot of things that you rely on annotations and an annotation processor to do in Java.

They already work pretty well with the JVM. Perhaps some improvements might be possible through the method handle features coming in Java 7 and Java 8, and perhaps support for interface injection would simplify the job for the Scala compiler, but even without these, the JVM handles traits and mixins just fine.

LISTING 3 LISTING 4

```
abstract class Amphibian {
  def color: String
  def swims = true
  def breathes = true
}
trait Green {
  def color = "Green"
}
trait HasLegs {
  def legs: Int
  def move = println("I move using %d legs".format(legs))
}
class Frog extends Amphibian with Green with HasLegs {
  val legs = 4
}
```

See all listings as text

Implicit Manifests

The final features we are going to look at in this article are, on the surface, not apparently linked. In fact, they are linked by a JVM shortcoming: type-safe erasure and the lack of reified types.

When Java was first created, it had a rich set of collections (a novelty for a language at the time, when it was considered normal to write your own collections or buy a library such as Rogue Wave in C++). The problem was that while Java was strongly typed, collections ignored the type, so when you asked for an ArrayList you got an ArrayList that could hold anything. Until generics were added in Java 5, you had to test and/or cast the type of objects you got out of a collection before you could use them as that type.

When generics were added, it was suddenly possible for the compiler to check and enforce type safety on collections, and to handle the checking and casting of types for you. This was a big step forward for code reliability (fewer class cast exceptions) and readability (in most cases). However, to maintain backward compatibility with code running against older collections without generics defined in the code, it was decided that generics would just be a compiler "fiction," and that the type information would be erased in the collection, rather than stored there.

Generics in Java were a big leap forward, but developers still run into all sorts of irritations because of this erasure of type information. You cannot tell at runtime what the generic type of an entire collection is; instead, you have to check the individual objects after you have retrieved them. If the compiler has the information, everything is great, but if not, you are forced back to the bad old days of check and cast.

An often-seen workaround in Java is for library or collection writers to have a parameter be passed in that contains a class reference to the class that will be stored, or is stored, or is requested in a method.

Scala has a feature to ease this pain. It is the concept of an implicit manifest. **Listing 4** shows an example.

When the compiler sees an implicit manifest such as this on a generic function or class definition, it automatically adds the class of the generic in question for that second parameter list. So, as a caller, you no longer have to supply the class explicitly in cases where you might need it in the function or class.

This partially works around the erasure of types, and it is good enough for maybe 80 percent of times you run into problems, but it is still not a complete solution. The issue is really highlighted by another feature of Scala.

Pattern Matching

Scala's pattern matching feature is extremely powerful and flexible. It looks a bit like a switch statement in Java, but in fact, it can do almost anything you can think of. This is not a surprise, because pattern matching is a cornerstone feature of many functional languages, such as Haskell and Erlang.

In Scala, patterns can be matched for literals, just as in Java, and also for objects. Strings work fine, but so do classes (particularly case classes that are intended to be used with pattern matching, but any class can be disassembled to its components and used in a pattern match with a little effort). Collections also may be matched, and this is an area where type-safe erasure once again rears its head.

For example, take a look at **Listing 5**.

It's a bit of a contrived example, but in fact, this limitation really does spring up quite a lot in practice. And in fact, it's not just lists or collections, but any kind of genericized class that you see the issue with.

PAINKILLER

Generics in Java were **a big leap forward**, but developers still run into all sorts of irritations because of this erasure of type information. You cannot tell at runtime what the generic type of an entire collection is. Scala has a feature to **ease this pain**.

Listing 5 defined a function called sumItUp, which takes a List of Any. (Any means any type can be in the list; it's a bit like Object in Java except it also includes all scalar types, such as int and double.)

Our implementation is simple enough. We are only worried about two types of lists (for now). If the list contains integers, we want to sum them up. If it contains strings, we want to convert them to integer values and then sum those. For anything else, we just return 0.

When we define the function, we get a warning about unchecked types. This is our first indication that something is not right.

When we try to use the function, it seems to work at first. If we supply a list of int values, we get back a sum of those. Everything seems great until we try a list of strings. Then, we get a class cast exception, but why?

The problem is that because of type-safe erasure, the JVM can't tell us what kind of type is stored in the collection. The only way to tell is to examine each element individually. Scala believes that we know what we are doing, so it lets us tell it that List is a list of integers in the first case, and it narrows the list to integers for us as a convenience. The problem is that when a list of strings comes in, Scala can't tell that apart from the list of integers, so it just narrows the list to a list of integers again. As soon as we try to use any of the contents, we get a class cast exception.

LISTING 5 LISTING 6

```
scala> def sumItUp(list: List[Any]): Int = list match {
     | case listOfInts: List[Int] => listOfInts.sum
     | case listOfStrings: List[String] =>
     |           listOfStrings.map(_.toInt).sum
     | case _ => 0
     | }
warning: there were unchecked warnings; re-run with -unchecked for details
scala> sumItUp(List(1,2,3))
res8: Int = 6
scala> sumItUp(List("1","2","3"))
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
```

See all listings as text

The "unchecked" warning from Scala is actually telling us exactly that, but because it is convenient to be able to narrow the types if you know the list can only be a list of integers, Scala doesn't enforce a compile error; it just issues the warning.

In fact, there is no way to be able to tell a list of int from a list of String at the collection level, and there will never be unless reified types are added to the JVM (so that collections can actually report back what they are storing).

There are workarounds that can be used, but they usually involve nesting matches inside one another, as in **Listing 6**.

Now, after we match a list of anything (the underscore matches any contained type, and this is the correct way to match a generic container if there is any doubt as to its contents), we then can match each element in turn, and convert the value to an integer. The values are then summed. Our new implementation has the advantage of being able to sum a mix of integers and strings, but it is still a workaround, and we might want to enforce the homogeneity of the list. In that case, we have to do even more work to make it safe.

There are other areas where type-safe erasure hurts Scala developers, but this is one of the most easily demonstrated.

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US
Java.net
blog

Conclusion

The JVM is already flexible beyond the requirements of the Java language. Support for rich symbolic method names beyond what is allowed by the Java language, and the addition of InvokeDynamic and method handles, already provides a welcome to other languages beyond what they might have a right to expect from a virtual machine that was written to serve one language above all others. It is clear, though, that there are still desirable features that would make the JVM even better for alternative languages.

In an ideal world, perhaps we will see a JVM with reified types, tail call optimization, interface injection, and other features not covered here that might help Scala or other JVM languages flourish; work faster, more consistently, and more concisely; and push the boundaries of what is possible even further.

In reality, if these features are added, it is likely to take a while for them to make it to the general releases of the JVM, and it is likely to take even longer for languages to switch to using those features exclusively if they want to support earlier versions of the JVM for a while. Just because method handles are available in Java 7 doesn't mean inner classes can immediately be replaced, because to do so would make Scala or other languages immediately incompatible with the earlier JVM, and perhaps not everyone is willing to switch to JDK 7 right away.

However, the growing momentum of JVM improvements speaks well for future support of alternative languages on the JVM. In the meantime, we can at least look forward to Java closures and function literals in JDK 8, and with luck, that will provide a standard for closures that all languages can agree on and allow better interoperability among all languages on the platform.

And in the meantime, until some of the more-advanced features are added, I have no doubt that the creators of alternative languages for the JVM will refuse to be held back by the limitations and will find clever ways around the issues, even if the workarounds are less than perfect. ●

LEARN MORE

- Scala Website
- The Da Vinci Machine Project
- Project Lambda
- More on Scala types and erasure
- Listen to Java Posse podcasts
- Download Scala
- Get Scala training
- *Programming in Scala, Second Edition* (Artima, 2010)

Oracle Press™ Your Destination for Java Expertise

Written by leading technology professionals, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Oracle products and technologies—including the latest Java release.

Acclaimed programming author Herb Schildt's books have sold more than 3.5 million copies worldwide

Java: The Complete Reference, Eighth Edition

Herb Schildt

A fully updated edition of the definitive guide for Java programmers

Java: A Beginner's Guide, Fifth Edition

Herb Schildt

Essential Java programming skills made easy

Java Programming

Poornachandra Sarang

Learn advanced skills from an internationally renowned Java expert

Available in print and e-book formats

Join the Oracle Press Community at
www.OraclePressBooks.com

twitter @OraclePress

Welcome to the first edition of Fix This. The idea of this section is to challenge your coding skills. In each issue, we will publish a code brainteaser. In the following issue, we will let you know what the right answer was and why. We'll also share what percentage of submitters gave what answer so you can see how you fared against other submitters. Our first submission is from Arun Gupta, Java evangelist at Oracle.

1 THE PROBLEM

Contexts and Dependency Injection (CDI) is a new specification in the Java EE 6 platform. It provides standards-based type-safe dependency injection for your Web applications. The CDI unifies JSF and EJB programming models and bridges the gap between the Web and the transactional tier by allowing an EJB to be used as a JSF backing bean.

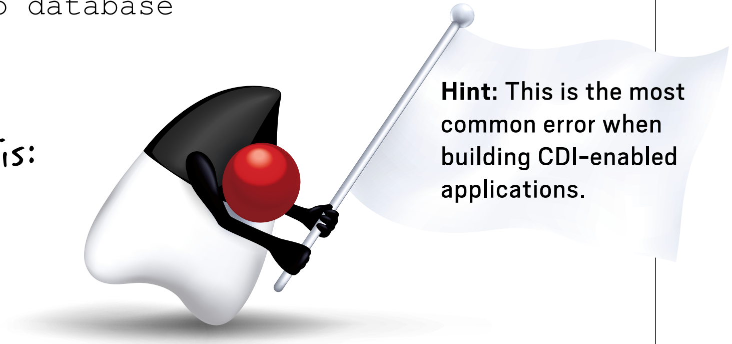
2 THE CODE

Consider the following code fragment for a JSF backing bean:

```
@Named @Stateless
public class MyBean {
 public void save() {
   // business logic to persist to database
 }
}
```

The WAR structure looks like this:

```
WEB-INF/classes
                /MyBean
index.xhtml
```



Hint: This is the most common error when building CDI-enabled applications.

3 WHAT'S THE FIX?

@Named allows the bean to be accessible in the .xhtml file for a JSF page as an Expression Language #{myBean.save}. Why can't the EJB be injected in the JSF page?

1) EJBs must be packaged in a JAR or EAR file to enable injection.
2) "beans.xml" is required to enable injection.
3) The business methods of an EJB must have ActionEvent as the parameter in order to be invoked.
4) CDI injection is not available from spec-defined classes.

GOT THE ANSWER?
E-mail it to us here.
Answers will be posted next issue.

PHOTOGRAPH BY MARGOT HARTFORD
ART BY I-HUA CHEN

COMMUNITY
JAVA IN ACTION
JAVA TECH
ABOUT US

EDITORIAL

Editor in Chief
Justin Kestelyn

Senior Managing Editor
Caroline Kvitka

Community Editors
Cassandra Clark, Sonya Barry,
Yolande Poirier

Java in Action Editor
Michelle Kovac

Technology Editors
Janice Heiss, Tori Wieldt

Contributing Writer
Kevin Farnham

Contributing Editors
Blair Campbell, Claire Breen, Karen Perkins

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Contributing Designers
Chris Strach, Jaime Ferrand

Production Designer
Sheila Brennan

PUBLISHING

Publisher
Jeff Spicer

Production Director and Associate Publisher
Jennifer Hamilton +1.650.506.3794

Senior Manager, Audience Development and Operations
Karin Kinnear +1.650.506.1985

ADVERTISING SALES

Associate Publisher
Kyle Walkenhorst +1.323.340.8585

Northwest and Central U.S.
Tom Cometa +1.510.339.2403

Southwest U.S. and LAD
Shaun Mehr +1.949.923.1660

Northeast U.S. and EMEA/APAC
Mark Makinney +1.805.709.4745

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (U.S./Canada)

Oracle Services
+1.888.283.0591 (U.S.)

Oracle Press Books
oraclepressbooks.com

ARTICLE SUBMISSION

If you are interested in submitting an article, please e-mail the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or e-mail address not be included in this program, contact Customer Service.

JAVA MAGAZINE IS PROVIDED ON AN "AS IS" BASIS. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DAMAGES OF ANY KIND ARISING FROM YOUR USE OF OR RELIANCE ON ANY INFORMATION PROVIDED HEREIN. The information is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle. Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Java Magazine is published bimonthly with a free subscription price by
Oracle, 500 Oracle Parkway, MS OPL-3C, Redwood City, CA 94065-1600.

Digital Publishing by Texterity