

Java™ magazine

By and for the Java community



//NOVEMBER/DECEMBER 2011/

3

MOVING JAVA FORWARD AT JAVAONE 2011

A report from the
world's biggest
Java community
gathering

10

GREEN LIGHT

Startup dooApp
relies on JavaFX
to support green
construction

53

ONE VM TO RULE THEM ALL

Alex Buckley on
the evolution of
the JVM into a
multilanguage
platform

SHOCK THE SENSES

JavaFX 2.0 reinvigorates client-side Java development **32**

COMMUNITY

02

From the Editor

JAVA IN ACTION

15

Driven by Java

Austrian smartcard system connects patients, providers.

JAVA TECH

19

New to Java

Starting Your First Computer Game

Use Greenfoot to create your first interactive program.

22

New to Java

Introduction to RESTful Web Services (Part 2)

Max Bonbhel on the JSON response format.

24

Java Architect

Agile ALM for Java Developers

Michael Hüttermann on better-quality software.

27

Java Architect

Understanding the Hudson Plug-In Development Framework

Extend Hudson with plug-ins.

36

Rich Client

JavaFX and Swing Integration

Migrate Swing interfaces to JavaFX.

38

Rich Client

Using Transitions for Animation in JavaFX 2.0

Animate nodes in your scene.

41

Enterprise Java

Stress Testing Java EE 6 Applications

Adam Bien on bugs, bottlenecks, and memory leaks.

47

Mobile and Embedded Getting Started with GameCanvas

Vikram Goyal on the Mobile Sensor API.

50

Polyglot Programmer

Polyglot Programming on the JVM

How and why to choose a non-Java language.

56

Fix This

Arun Gupta challenges your coding skills.

3

Java Nation

JAVAONE 2011 BRINGS FOCUS, MOMENTUM

A report from the world's biggest Java community gathering. Plus community news and events.

10

Java in Action

GREEN LIGHT

Startup dooApp relies on JavaFX to support green construction.

32

Rich Client

SHOCK THE SENSES

JavaFX 2.0 reinvigorates client-side Java development. Oracle's Nandini Ramani talks with *Java Magazine* about the key features of the new release.

53

Polyglot Programmer

TOWARD A UNIVERSAL VM

Oracle's Alex Buckley on the JVM Language Summit 2011.

A man with dark, wavy hair, wearing a light blue and white checkered button-down shirt, is shown from the chest up. He is holding a black mobile phone to his left ear with his left hand, which also features a watch with a dark, textured strap. He is looking slightly to his right with a neutral expression. In the background, there is a large, bright sign that reads 'MOVING JAVA FORWARD' in blue and orange text. To the left of the man, another sign with a stylized orange flame logo is partially visible. The setting appears to be an indoor office or conference space with blurred background elements.

So what's next? Well, we need to get you ready for Java EE 7 (scheduled for release in 2012), which will provide a genuinely standard runtime for cloud-based apps. Look for that in the January/February 2012 issue.


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

PHOTOGRAPH BY BOB ADLER

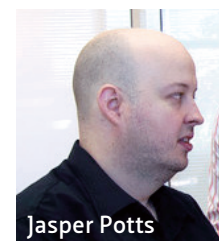
YOUR LOCAL JAVA USER GROUP NEEDS YOU

Find your JUG here



A man with dark hair, wearing a dark t-shirt, is speaking on a stage. He is gesturing with his hands. The background is a blue wall with a pattern of small white dots. In the bottom left corner, there is an orange square with a white play button icon.

JavaFX Interfaces with the Xbox Kinect



Ritter and Potts used the [OpenNI framework](#) to wrap the Kinect API so it could be accessed by JavaFX using Java Native Interface. It took some work, and some tweaks, but ultimately they were able to access 3-D depth data from the Kinect sensors using this type of JavaFX code:

Considering the demo and the JavaFX technology behind it, Potts said, "This is a cool 3-D application from JavaFX labs. We showed Duke in 3-D and I got to pretend to be Duke, with the Kinect controller controlling it. It was a lot of fun and all went well onstage, so I'm really pleased."

PHOTOGRAPH BY BOB ADLER

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.

connectivity
powered by 

To learn more please visit our website →

www.nsoftware.com

Duke's Best and Brightest

The Duke's Choice Awards were created in 2003 to honor extreme innovation in the world of Java technology. The 2011 Duke's Choice Awards, presented at JavaOne 2011, showcased the amazing results of projects supported by thousands of development hours. The winning projects and companies are as follows:

- **Arquillian.** This project simplifies integration testing for Java-based applications.
- **dooApp.** Infiltrate is dooApp's JavaFX and Java Platform, Standard Edition solution, designed for green building professionals whose job involves measuring the airtightness of buildings.
- **Inductive Automation.** Ignition is Inductive Automation's Java-based Web application that integrates different manufacturing machines using a central Web server.
- **jHome.** This complete home automation open source API, based on GlassFish Server Open Source Edition and Java Platform, Enterprise Edition (Java EE), enables developers to control anything in their homes.
- **JFrog.** JFrog's Artifactory is the world's first binary repository manager and is built with the Content Repository API for Java specification.
- **JRebel.** This Java Virtual Machine plug-in enables Java developers to instantly see any code change made to an application.
- **LMAX Disruptor.** This solution is a multithreaded, open source concurrent programming framework designed for high-performance and low-latency transaction processing.
- **Rockwell Automation.** This company's Java-enabled human-machine interface line of products will allow the automated communication and exchange of data to factory floor lines.
- **Sodbeans Project.** This open source, NetBeans-based module suite is designed to enhance accessibility for the blind in modern programming environments.
- **The Netty Project.** This Java-based, new I/O client/server framework enables quick and easy development of network applications.

"The Duke's Choice Awards showcase the incredible creativity that takes place every day in the Java community," says Amit Zavery, vice president of product management in Oracle Fusion Middleware. "This year's winners demonstrate the countless ways that Java technology can be used to create innovative applications and products. Put in the hands of creative and passionate individuals, Java can have a significant impact on our lives." —Marta Bright



Meet Vinny Senger, the technical lead of jHome, a complete home automation API based on GlassFish Server Open Source Edition and Java Platform, Enterprise Edition that lets you control anything in your home.

use of JavaFX, a leading user interface, graphics, and media framework for developing rich internet applications in the Java environment (see sidebar, page 11). The software is also the winner of this year's Duke's Choice Award for Innovative Green Technology, presented at this year's JavaOne conference in San Francisco, California, in October.

AIRTIGHT, ENERGY EFFICIENT

Airtightness is one of the key measurements of a building's energy efficiency, and it's a major goal of green building. According to the U.S. Energy Information Administration, buildings in the U.S. create 48 percent of all domestic greenhouse gas emissions and consume most of the electricity the nation's power plants generate. Making buildings more energy efficient reduces both the demand and pollution.

When buildings "leak," some of the energy their heating and cooling systems consume to create

a comfortable indoor environment is wasted; buildings that are airtight, on the other hand, consume less energy because they waste less through leakage. "Green buildings need to be airtight," explains Mischler.

In Europe, regulation EN 13829 mandates what is called a *blower door test* to determine if new construction, both residential and commercial, meets airtightness requirements. It also defines the testing method, criteria, and reporting requirements. Individual countries have their own regulations; in France, for example, RT 2012 and its stricter airtightness standard began taking effect in late 2011.

Infiltrate provides an end-to-end solution for the airtightness testing process, says Mischler. The application includes a blower door connection, standard tests and charts, and report-writing features. The charts, created with JavaFX, provide the main view of the test's measurements.



dooApp's Infiltrate application helps green building professionals to measure airtightness, a critical component in green building. The 2.0 release of Infiltrate is Web-enabled using the JavaFX WebView component, as seen here in the project outline view.



Antoine Mischler (center) and the dooApp team brainstorm about enhancements to the Infiltria application.

"The goal of the process is to compute some indicators saying whether the building is good or bad," says Mischler, noting that *good* means it meets regulatory requirements.

A blower door system consists of a calibrated fan, a door-panel system, and a device to measure the fan flow and building pressure. The blower door fan is temporarily sealed into an exterior doorway and then blows air into or out of the building, creating a pressure difference between inside and outside. The pressure difference forces air through all

the holes and penetrations in the building's exterior envelope, including the walls, doors, and windows. The tighter the building, the fewer holes or leaks there are in its envelope and the less air that is needed from the blower door fan to create a change in the building's pressure.

The Infiltria software runs on Windows, Mac OS, and Linux; it records the test and presents a visual representation of its progress in real time. "Our software is connected to the measuring device and to the blower door, and

acquires the measurements," says Mischler. "With Infiltria, the professional has a chart that shows the pressure in the building and the airflow in real time."

Once recorded, the measurements are published in two reports, directly from Infiltria. The first, a technical report, is a two-pager that includes the building's features, the testing criteria, and a chart of results. The second is a longer and more complex legal report that conforms to EN 13829. It can include photos, logos, plans, thermal surveys, and other relevant data and can be published in a number of formats, including PDF, .docx, HTML, RTF, and others. "At the end," Mischler says, "the software gives a full report to the professional that conforms to the regulations."

Infiltria doesn't show *where* a building is leaking, however, and that is the next step in the process of making a building airtight. If a building doesn't pass, then the diagnostic professional conducts smoke tests to identify leakage sites. "Once the leakage site is identified, the professional will seal it and run the airtightness tests again until all the leakage sites are identified," explains Mischler.

THE NATURAL CHOICE

Going with Java and JavaFX was a natural choice for dooApp's green building applications. Mischler says that when he and Dufour were working as Java developers, they real-

FAST FACT
Buildings in the U.S. create **48 percent** of all domestic greenhouse gas emissions.

ized they wanted to apply their expertise to an area in which they both had a strong personal interest: green building.

"What we wanted to do was to bring our technical skills to develop innovative software for professionals in the green building sector," says Mischler. "What we had seen is that most software developers in this sector were engineers or architects

who had moved into software but lacked specific skills in software. And we thought that, as a result, the tools they had developed were not so innovative or user friendly."

Making the software interface user friendly was a high priority, so they chose JavaFX as Infiltrate's graphics and user interface framework, says Mischler. "We already had worked a bit with Adobe Flash, but we knew this was not well suited to design a professional desktop application because of performance and maintenance issues," Mischler explains. "Since [Microsoft] Silverlight is pretty much the same, we didn't look at it any closer."

"We both had experience with Swing, and when we started

developing Infiltrate we had the choice between Swing and JavaFX," continues Mischler. "Swing is the 'old' Java graphics framework. So Swing was a mature and stable technology but had some major drawbacks: you have to write a lot of code for small results, it's hard to style components, and the default look and feel looks a bit outdated."

On the other hand, he continues, JavaFX "was a new technology that addressed these issues and introduced tons of cool new features, such as special effects and animations, as well as CSS skinning."

The choice of JavaFX wasn't made without trepidation, however. "Since it

was a new technology, making the choice of JavaFX was somewhat of a gamble," Mischler concedes. "After considering both options, we made the choice to work with JavaFX and we do not regret this today. The technology is fun and is evolving in the right way." `</article>`

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

DUKE'S GREEN CHOICE
French startup **dooApp** is the winner of this year's **Duke's Choice Award for Innovative Green Technology**.

COMMUNITY

JAVA IN ACTION

JAVA TECH

ABOUT US



**LEARN >
WITHOUT
LIMITS.**

Learn more about
Java technology
with Safari Books
Online



Use your QR code reader to access information about a group trial or visit safaribooksonline.com/javamagazine

SEARCH LESS. DEVELOP MORE.
With the intelligent library in the cloud



Rainer Schügerl, Director of Software Development and Security, SVC

Smartcard system connects patients, providers, hospitals, and pharmacies throughout Europe. **BY DAVID BAUM**

Austria is well known for its advanced and generous social services—marked by excellent healthcare; a strong social security system; and an extensive network of hospitals, physicians, and pharmacies. Austrian Social Security Law, a public insurance system that includes 22 institutions offering a variety of insurance coverage and social programs, insures most Austrian citizens, and in recent years this national health system has been bolstered by one of the world's most advanced programs for smartcards and electronic health records.

JAVA IN ACTION

JAVA TECH

ABOUT US

o

Java
.ne

blo

15



Today all insured citizens get a smartcard that verifies their insured status and facilitates the creation, transmission, and storage of electronic health records. The e-card system also authorizes citizens to avail themselves of a variety of e-health services, from preventive checkups to disease management programs. Thousands of healthcare providers have installed special equipment to scan the smartcards and transmit secure data across a secure health information network that spans the Austrian nation and extends to many other European countries within the scope of the NETC@RDS project of the European Union. Java is the *lingua franca* of the system.

"We chose Java because of the platform advantages, especially its tremendous portability among CPUs and hardware platforms," says Rainer Schügerl, director of software development and security at SVC, a public entity based in Vienna, Austria, that develops innovative solutions in the field of health telematics and e-government. SVC served as the systems integrator for the e-card project.

"Java provides a stable, high-quality programming language that suits all of our needs," Schügerl adds. "For enterprise-caliber development requiring high availability, reliability, and security, most Austrian organizations use Java."

insurance vouchers that had dominated Austria's medical world for 50 years. Manual processes between insured people, employers, doctors, hospitals, and the Main Association of Austrian Social Security Institutions have been systematically replaced by electronic solutions and e-business procedures governing the transmission, storage, processing, and virtualization of health and administrative data.

The Main Association of Austrian Social Security Institutions is a group of statutory insurance providers that is responsible for the country's health, pension, and accident insurance. The association enlisted SVC to design and deploy the e-card system and an associated virtual private network. As a 100 percent subsidiary company of the Main Association of Austrian Social Security Institutions, SVC has a proven track record of enforcing transparency and security in the processing of health and administrative data, whether for clinical use, administration, or science. Its mission is to improve the efficiency of information processes among patients, health service providers, and health administrators.

Schügerl has 40 software engineers on his team, including 20 Java developers. Due to the sensitive nature of health and medical information, the team needed a programming language with a robust security architecture. They decided to use Java Standard Edition for Embedded Devices, which supports the same platforms and functionality as Java Platform, Standard Edition (Java SE), along with additional capabilities for the embedded market such as support for small-footprint Java runtime environments (JREs), headless configurations, and memory optimizations. Java SE is ubiquitous on servers and desktop computers, thanks to its high-performance virtual machine and exceptional graphics support. To top it off, it has an extremely lightweight deployment architecture and a complete set of features and libraries for programmers.

Java applets are ideal for smartcards and similar small-memory devices because they carry all the essential security features such as cryptography, authentication, authorization, and public key infrastructure.

For SVC, these features ensure that new e-card applications can securely access centralized resources while protecting critical data from theft, loss, and corruption. The Java API supports a wide range of cryptographic services including digital signatures, message digests, ciphers, message authentication codes, key generators, and key factories, allowing

SVC developers to easily integrate security into their application code.

The smartcard does not contain software functions. Only the cardholder's administrative data is stored on the e-card—name,

PERFECT FIT
Java applets
carry essential
security features
and are **ideal for**
smartcards.

If the condition is false, the body is just ignored, and execution continues after the **if** statement.

With these two bits together, we can now do what we need to do. We need to write the following:

This is not real Java code (it's pseudocode). We can now fill in the pseudocode with real Java code:




Figure 2

Doing the same for the right arrow key to turn right is now simple. We just write a similar statement again, this time turning 3 degrees for the right arrow key. The complete new `act` method is shown in **Listing 1**.


You can now control a turtle on your screen with your arrow keys. Pretty good fun already!

Currently, the turtle will continuously walk forward. If you want to, you can change your code so that the turtle moves forward only when you press the up arrow. You do that by adding another `if` statement around the `move(4)` statement. The name of the up arrow key is `up`.

Next, we want to give the turtle something to do: collect some food and eat it. To do this, we need some new objects in our world: pizza slices. (Yes, our turtle eats pizza. This is not a biology lesson—it's a computer game.)

Right-click the **Actor** class (see **Figure 2**) and select the **New subclass** function. In the resulting dialog box, enter **Pizza** as the class name, and search

LISTING 2

 [See all listings as text](#)

As soon as the classes are compiled again, you can add a few pizza slices (as well as a turtle) to your world.

Try it out. Let your turtle run around to collect pizza. You will notice, though, that the turtle does not (yet) eat the pizza. It will just run straight across the pizza. We will fix that shortly.

If you worked along and experimented in Greenfoot for a bit, you will have noticed that you need to re-create all objects in the world after every compilation. This can get a little tedious after awhile, especially if we have many objects.

- You can shift-click into the world. This is a shortcut

- When all your objects are in the world as you want them to be at the start of your game, you can right-click the world background and select **Save the World** to save this world state. This state will then be re-created automatically after every compile or reset.

OK, we have pizza. We have a turtle. The turtle is hungry.

pizza and to make the turtle eat the pizza. For this we need what's called *collision detection* to detect when we've run into a pizza slice.

The full code to do this is in **Listing 2**. This code should be added to the end of the

isKeyDown lets us check whether a given key is being held down.

`act` method in the `Turtle` class. There are several new elements in this code segment. Let's look at them one by one.

The first line calls a Greenfoot method: `getOneIntersectingObject(Pizza.class)`.

This method checks whether the turtle intersects an object of class `pizza` (or, in other words, it asks, “Are we touching a pizza slice?”). If we are touching a pizza slice, this method returns the `pizza` object we found. If not, it returns a special value, `null`, which tells us that we are not touching anything.

The equal symbol in the first line is an *assignment* operator. It stores the result of our collision check (the `pizza` object or `null`) in a *variable*. A variable is a bit of storage where we can store objects.

In our case, the variable has been declared using the following statement

at the beginning of the first line:

■ Actor pizza

This declares a variable named `pizza` that can hold `Actor` objects (objects that live in the Greenfoot world).

After the first line, the `pizza` variable holds the `pizza` object we ran into, or it holds `null` if we didn't run into a pizza slice in this step.

Next follows another example of an **if** statement. In pseudocode, the next lines state the following:

```
if(we-have-run-into-pizza) {  
    remove-the-pizza-object-  
    from-our-world;  
}
```

The condition check is done with the following expression:

```

if(pizza != null) {

```

The `!=` symbol means *not equal*. So, here we are saying, "If our `pizza` variable is not equal to `null`..."

The effect is that the instruction in the body of the **if** statement is executed only if the `pizza` variable is not **null**, that is, if we have run into some pizza.

The next line then removes the **pizza**



Figure 3


LISTING 3

```
public void act()
{
    move(4);

    if(Greenfoot.isKeyDown("left")) {
        turn(-3);
    }

    if(Greenfoot.isKeyDown("right")) {
        turn(3);
    }

    Actor pizza = getOneIntersectingObject(Pizza.class);
    if(pizza != null) {
        getWorld().removeObject(pizza);
    }
}
```

 [See all listings as text](#)

object from the world. **Listing 3** shows the complete version of our `act` method at this stage.

Try it out. Once you have written this code, your turtle can walk around and eat pizza (see **Figure 3**).

Conclusion

In this article, you learned a bit more about writing Java code and about some useful Greenfoot methods.

- **If** statements can be used to execute an action only under a certain condition.
- **If** statements have a condition and a body. The body is executed only if the condition is **true**.
- The **Greenfoot.isKeyDown** method can be used to check for key presses.
- The **Greenfoot.isKeyDown** method can

be used as the condition in an `if` statement.

- Collision detection can be used to check whether one object touches another.
- Variables are used to store objects.
- The special value `null` is used to indicate that we do not have an object. `</article>`

LEARN MORE

- [The Greenfoot gallery](#)
- [Young developer resources](#)
- ["Wombat Object Basics \(Young Developers Series, Part 1\)"](#)
- ["Wombat Classes Basics \(Young Developers Series, Part 2\)"](#)

- A new resources HTML file is added to the project.
2. Edit the created page and modify the HTML `<body>` tag, as shown in **Listing 4**.
3. Add the JavaScript functionality to invoke the Web services and handle the data storage by modifying the HTML `<head>` tag, as shown in **Listing 5**.
4. Retrieve and display the stored information:
 - a. Modify the HTML `<head>` tag as shown in **Listing 6**.
 - b. Save the HTML file.
 - c. Build and deploy the project.

We need to use a Web client. So we are going to quickly build a sample HTML5 page front end to allow us to perform the following tasks using HTML, JavaScript, and Ajax functionalities:

1. Call the Web services for seller #2:
 - a. Open your browser and type the re-

```
@Path("/sellers/")
@Stateless
public class SellersResource {
    @javax.ejb.EJB
    private SellerResource sellerResource;
    @Context
    ....
    @GET
    @Produces({"application/json"})
    public SellersConverter get(@QueryParam("start")
                               @DefaultValue("0")
```

 [See all listings as text](#)

We have seen how easy it is to configure RESTful Web services to send a JSON response to the client. The integration of JavaScript and Ajax made it easy to

In Part 3 of this series, we will focus on the integration of Java API for XML Web Services (JAX-WS) technology for “big” Web services. </article>

- [Roy T. Fielding's dissertation defining the REST architectural style](#)
- [HTML5: Up and Running](#), by Mark Pilgrim (O'Reilly Media, 2010)

Agile ALM for Java Developers

Agile ALM makes for better-quality software and happier Java developers.

MICHAEL HÜTTERMANN



Agile application lifecycle management (ALM) is a way for Java developers, testers, release managers, technical project managers, and decision-makers to integrate flexible agile practices and lightweight tooling into software development phases.

Agile ALM synthesizes technical and functional elements to provide a comprehensive approach to common project activities and phases, addressing test, release, integration, build, requirements, change, and configuration management. (See **Figure 1**.)

With its interdisciplinary approach, agile ALM integrates project roles, project phases, and artifact types. Agile ALM enriches an ALM approach with agile values and strategies. An agile approach to ALM improves product quality, reduces time to market, and makes for happier developers.

In a nutshell, agile ALM

- Helps overcome process, technology, and functional barriers (such as roles and organizational units)
- Spans all artifact types as well as development phases and

project roles

- Uses and integrates light-weight tools, enabling the team to collaborate efficiently without any silos
- Makes the relationship of given or generated artifacts visible, providing traceability and reproducibility
- Defines task-based activities that are aligned with requirements, which means the activities are linked to requirements and all changes are traceable to their requirements

As a result, from a Java developer perspective, agile ALM helps developers focus on doing their jobs rather than wasting time with barriers or obstacles that reduce productivity. Streamlining Java development with agile ALM fosters collaboration among coworkers, leading to improved software quality and better results.

Agile ALM provides processes and tool chains that are flexible and open to change. This is one of the ways in which ALM provides structure for agile software development. Some underlying aspects of agile ALM are not completely

new, and developers should respect the struggles of previous decades, and put results together to get the best solution. In my view, ALM evolved from software configuration management (SCM), which in turn has its roots in basic version control.

Following are some major agile ALM building blocks that address and leverage widespread activities among Java developers.

Release Management

Release management involves producing software artifacts and releasing them according to a

defined process. Release management can be differentiated into a functional part and a technical part. To deliver software successfully, both parts are important and should be integrated with each other. Automation and continuous integration are crucial facets of the software release and delivery process.

Functional release management.

Functional release management involves identifying the customer's requirements, assigning them to releases, and delivering the functionality to the customer with high quality. Agile practices

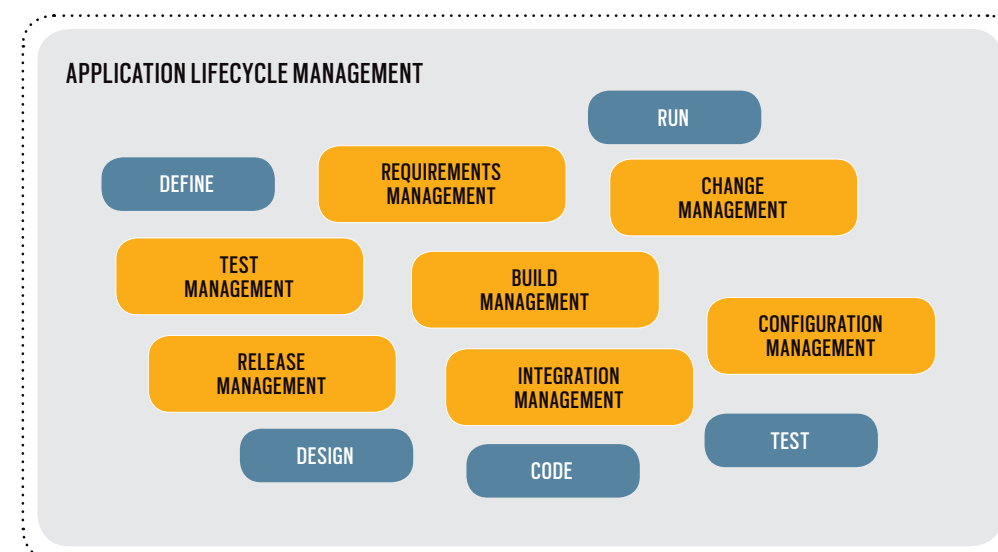


Figure 1

are often used to support this process, and many projects achieve good results from using the management template known as Scrum.

While defining a thin set of rules, Scrum fosters discipline and makes defects (in software as well as in the process) visible. But Scrum is too abstract to be applied “out of the book.” You must implement and adopt it for software engineering. Implementation practices, for instance, might include distinguishing between special development phases, on a micro level, inside a Scrum release. Here are some examples:

- Opening the release with an interval to write acceptance tests
- Opening the release with an interval to get more-detailed requirements
- Closing the release with a frozen zone that allows developers to work only on bug fixes, not on new features

- Closing the release with a code-freeze interval in order to complete and ship the final release

Technical release management.

Technical releasing consists of building the software and providing the final product to the customer. Build management (comprised of compiling scripts, packaging components, and distributing components) is essential for agile ALM.

Technical release management includes activities to identify configuration items, track and audit changes to requirements and configuration items, and integrate and deliver the implementation. In software engineering, change is more the rule than the exception. Because requirements change, it is important to keep the requirements in sync with their implementations. Possible gaps between functional and technical release management should

be bridged. Strategies such as version control system (VCS) hooks help marry both parts of release management. A leading tool that supports technical release management is [Maven](#). With Maven, the dependencies between your Java artifacts become visible.

Automation and Continuous Integration

Automating manual steps involves delivering results in an objective and reproducible way. Automating the most error-prone, repetitive, and

time-consuming activities is essential.

Continuous integration (CI) refers to the automation of the build, test, and release process with the goal of integrating the activities of colleagues and the work items others produce. This can result in a build ecosystem in which a new-code commit directly triggers a continuous build, including compilation, technical tests, audits, packaging, functional tests, and deployment.

All different artifact types, platforms, and languages (for example, Java, Groovy, Scala, PHP, COBOL, and others) should be integrated using a unified tool infrastructure. If no native build system exists for a respective language or platform, non-native build technologies can be used to include these artifact types in the CI farm.

One tool that can serve as your central tool hub is the build engine Hudson. Integrating Hudson with the tool Mylyn results in a tool chain that spans different project roles and activities. Then, the work on artifacts is traceable, starting with the source code in the developers' workspaces on up to tasks (work items) in your ticket system.

Using a component repository such as Nexus helps to differentiate between source code and Java binaries, and the repository operates on binaries separately—for example, through staging binaries from one test environment to another environment.

FLEXIBLE STRUCTURE

Agile ALM provides **processes and tool chains** that are flexible and open to change.

Collaborative Development

Writing software collaboratively means that all stakeholders, especially Java developers and testers, work on the solution constructively and with shared goals. The agile testing matrix (see **Figure 2**) defines test quadrants, which are integrated and aligned using the

outside-in and *barrier-free* approach to software development.

The main drivers of the outside-in approach are

- Understanding your stakeholders and the business context
- Mapping project expectations onto outcomes more effectively
- Building more-consumable software, making systems easier to deploy and use
- Enhancing alignment with stakeholder goals continuously

Acceptance tests determine whether a system satisfies its specified acceptance criteria, which helps customers decide whether to accept the software. This means that acceptance tests also tell the programmers what the customer doesn't want them to do.

Executable specifications allow you to use tools that read the specifications automatically (either after manually starting the process or continuously as part of a CI process); process them against the system under test; and output the results in an objectively measurable, efficient, and readable way. The domain expert specifies the tests in simple

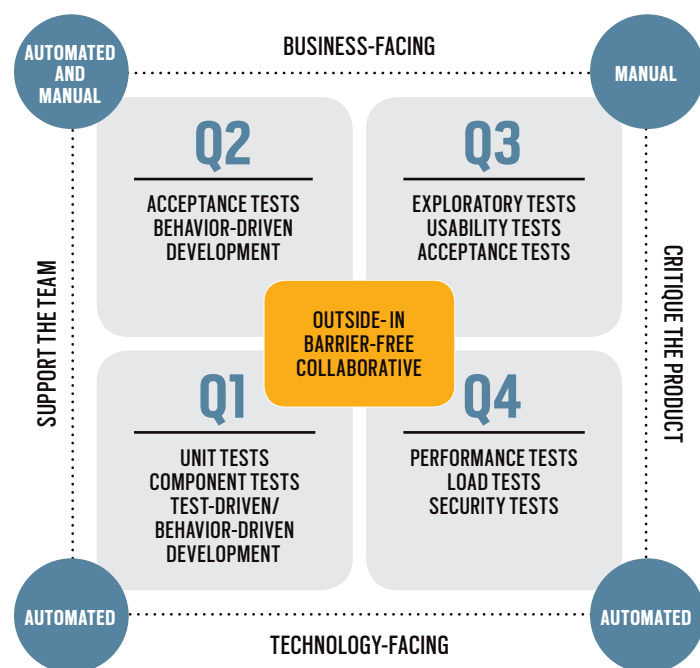


Figure 2



Use the Hudson Plug-in Interface tool to create, build, run, and debug custom plug-ins that extend Hudson to meet the specific needs of your projects.

Hudson provides a series of extension points that allow developers

- Writing a configuration for a Hudson extension
- Adding a global configuration for a Hudson extension

```
mvn hpi:create
```

Enter the groupId of your plug-in:
org.sample.hudson

Enter the artifactId of your
plug-in:
sample-plug-in

- **pom.xml**: The Maven Project Object Model (POM) file, which is used to build the plug-in
- **src/main/java**: The folder that contains the Java source files for the plug-in
- **src/main/resources**: The folder that contains the Jelly UI

POPULAR TOOL

Hudson's popularity is due to its **extensible nature.**

First, we'll look at how to create a Hudson plug-in using the HPI tool, create and run a project for the plug-in, and view the plug-in in action on the Hudson test server.

ORACLE.COM/JAVAMAGAZINE ////////////////////////////////// NOVEMBER/DECEMBER 2011



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®



RICH HISTORY
JavaFX 2.0 reflects
years of rigorous
development,
testing, and cross-
platform capability.

Nandini Ramani (center)
and members of the
JavaFX 2.0 team discuss
new capabilities.

include everything from the ability to play audio and video, to user interface controls and layout containers, to a scene graph and animation libraries, to observable collections and UI binding. At the bottom of the stack, we have a new windowing library and hardware accelerated graphics library.

The industry is moving toward multicore multithreading platforms with GPUs [graphics processing units], and JavaFX 2.0 leverages these attributes to improve execution efficiency and UI design flexibility. Our initial goal is to give architects and developers of enterprise applications a set of tools to help them build better business applications, dashboards, richly animated UIs, and mashups that integrate different Web-based datasources.

Java Magazine: Software developed using

JavaFX 2.0 can be run standalone, in the browser, or using Java Web Start. Will you describe application areas best suited to each of these options?

Ramani: The standalone mode is appropriate when higher control on the runtime environment is appropriate. The application is launched from a client-side JAR file, or perhaps bundled as a traditional OS-specific executable by using existing third-party tools. When executing in the browser, the JavaFX application is embedded in a Web page using the traditional plug-in model. With Java Web Start, the application is run on the desktop, but it's initially downloaded from the Web, and like a Web application, it can be updated every time the application is run. This simplifies deployment and update logistics by

downloading from a central Web location, and it handles updates automatically.

Java Magazine: What strengths does JavaFX 2.0 offer over Adobe Flash or Microsoft Silverlight?

Ramani: First and foremost, JavaFX 2.0 is built on top of the Java platform, so it reflects all the years of rigorous development, testing, and cross-platform capability that Java brings to the table. It's hosted on the efficient and widely available Java Virtual Machine. Many users of JavaFX 2.0 will already be using Java EE for back-end processing, so JavaFX on the front end allows them to utilize existing skill sets, tooling, and infrastructure. Everything they apply to back-end development becomes applicable on the front end. In addition, Java has the largest and strongest



NANDINI RAMANI
VP, DEVELOPMENT, JAVA CLIENT PLATFORM



Watch *Introducing JavaFX 2.0*

ecosystem in terms of third-party libraries around. Being able to leverage that kind of support for developing client applications is a big plus.

Java Magazine: Can languages like JRuby, Scala, and Groovy interoperate with JavaFX 2.0?

Ramani: Absolutely. Just as you can run any of these languages on top of the Java platform, they also all interoperate with JavaFX 2.0. In fact, there's a project in the Groovy community called GroovyFX, which uses the Builder design pattern for JavaFX applications. Most of this work is being done open source, and many of the language implementers are excited about using JavaFX 2.0 technology. For some Java developers, dynamic scripting languages are a vital part of their toolkit, and JavaFX can seamlessly work with them.

Because JavaFX uses familiar design patterns, such as POJOs following the JavaBeans naming conventions, many of these languages already have enhanced support for JavaFX right out of the gate. In addition, JavaFX was carefully designed with consistent API idioms, such as event handling, which make it easy for other languages to provide syntactic sugar for these APIs.

Java Magazine: Will you describe an application scenario where Prism, the hardware accelerated graphics pipeline, and the new Glass windowing toolkit might offer tangible advantages?

Ramani: Prism handles rasterization and rendering of JavaFX scenes. It can execute on both hardware and software renderers. The Glass windowing toolkit is the lowest-level framework for the JavaFX 2.0 graphics stack—it manages timers, surfaces, and windows.

When developers need gradients and

shading, or any sort of advanced graphics, hardware acceleration can offer a huge payoff in system responsiveness. And when applications need to build tables and display large data sets, Prism can also be valuable. The Prism/Glass combination eliminates the need to instantiate and load Swing and AWT classes, which gives JavaFX 2.0 a definite performance advantage.

Java Magazine: How will FXML, the JavaFX 2.0 XML-based markup language, add value for developers? And will you describe how Scene Builder will be useful for generating FXML?

Ramani: The fact that FXML is XML-based is a huge advantage from a tools perspective, because Java IDEs such as NetBeans and Eclipse interoperate easily with XML. Layout design for enterprise applications is one of the trickiest and most-labor-intensive endeavors. The visual JavaFX Scene Builder tool mitigates the need for hand coding, which is a definite productivity bonus. Also, FXML ties into JavaFX Scene Builder in two ways: developers can either programmatically make changes and then generate FXML, or they can use JavaFX Scene Builder to design the JavaFX UI components and then save them as FXML markup. JavaScript can also be seamlessly integrated into this scenario. So there's a great deal of flexibility. These tools also allow the use of CSS. Once an application is created, a completely new visual look can be generated without recompilation by changing either the style sheet or the FXML code. This is an industry trend, and JavaFX 2.0 provides the toolset.

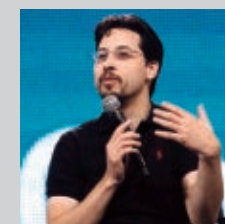
Java Magazine: Is HTML5 part of the JavaFX 2.0 landscape?

Ramani: HTML5 technologies are still under development by the W3C. But support for HTML5, as defined today, is built into JavaFX 2.0. Our Web component is based on WebKit, which supports HTML5. Many developers are already using WebKit-based browsers. Possible future developments, like offline storage, Web SQL databases, native drag-and-drop, and geolocation, will be of interest as they evolve. We plan to stay in sync with HTML5.

Java Magazine: How do the JavaFX UI control libraries fit into the JavaFX 2.0 toolset?

Ramani: I was a longtime Swing developer myself, and I love its capabilities—a great way to program. But every component in Swing tends toward rectangular forms. Today, we need more flexibility. JavaFX 2.0 UI controls offer the ability to create rounded shapes, a much wider range of components, and

COMMUNITY COMMENT



“Making JavaFX open source is a huge step in the right direction for this technology. It will enable businesses to use JavaFX with no fear about vendor lockin, and let the community participate and grow the platform in new and innovative ways. I have been advocating for this move since the 1.0 release, and am glad that Oracle listened to the community and made this possible.”

—Stephen Chin, a technical expert in RIA technologies and chief agile methodologist at GXS

PHOTOGRAPH BY HARTMANN STUDIOS





Nandini Ramani and members of the JavaFX 2.0 team discuss the product's roadmap.



LEAPS AND BOUNDS
JavaFX 2.0 is a leap forward. Developers can seamlessly **mix and match** Web and media content within their Java applications.

interface that the lambda proposal for Java SE 8 requires.

But we don't have to look to the future for something exciting. JavaFX 2.0 is a leap forward. It offers developers the ability to seamlessly mix and match Web and media content within their Java applications using a variety of powerful coding and layout options, which can be running inside a browser or as standalone applications. We believe that a broad range of enterprise applications will benefit from this multifaceted new technology.

Java Magazine: What role will open source play in JavaFX's future?

Ramani: At JavaOne, we announced our intention to submit a proposal to open source the JavaFX platform as a new OpenJDK project. We plan to initially contribute the JavaFX UI controls and related libraries; other JavaFX components will follow in multiple phases. By the time you read this article, we are confident that the OpenJDK community will have approved this proposal. You can check the status of the JavaFX project on the [OpenJDK site](#). **</article>**

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*, *LA Weekly*, and on National Public Radio. He is also a Huffington Post blogger.

LEARN MORE

- [JavaFX home page](#)
- [JavaFX 2.0 downloads](#)
- [FX Experience](#)
- [Overview of JavaFX](#)

skinning via CSS. This new set of powerful components will allow any developer to build advanced enterprise application interfaces. Also, the JavaFX UI controls will be the first component of JavaFX to be open sourced as an OpenJDK project. This will enable developers to build their own custom versions of components, which the Swing community has been doing for many years.

Java Magazine: Can JavaFX 2.0 code be easily integrated into existing Swing applications?

Ramani: Yes, Swing and JavaFX cooperate well. Within existing Swing applications, you can either add on or embed JavaFX. All Swing capabilities are retained, but the portions rendered with JavaFX leverage all the new capabilities. For instance, if you want to embed a Web browser into your Swing application, you can do so by embedding the JavaFX WebView. Another option is to rewrite some of the Swing code for JavaFX. It's a great migration path for existing Swing applications.

Java Magazine: What operating systems will be supported by JavaFX 2.0?

Ramani: The first release will support 32-bit and 64-bit versions of Microsoft Windows XP, Windows Vista, and Windows 7. In addition, we'll be releasing a developer preview of the JavaFX 2.0 SDK for Mac OS X. A Linux version will be made available at a later date.

Java Magazine: How will JavaFX evolve going forward?

Ramani: On the client side, we're trying to stay in lockstep with the evolution of the Java platform. We will integrate new features as they emerge, like [invokedynamic](#) and [NIO.2](#). Two of the biggest enhancements to the Java platform planned for Java SE 8 are modularization and lambda expressions. JavaFX APIs were specifically designed with both of these features in mind. For example, the event handling APIs are all designed around SAM [single abstract method] interfaces, which happen to be the exact type of



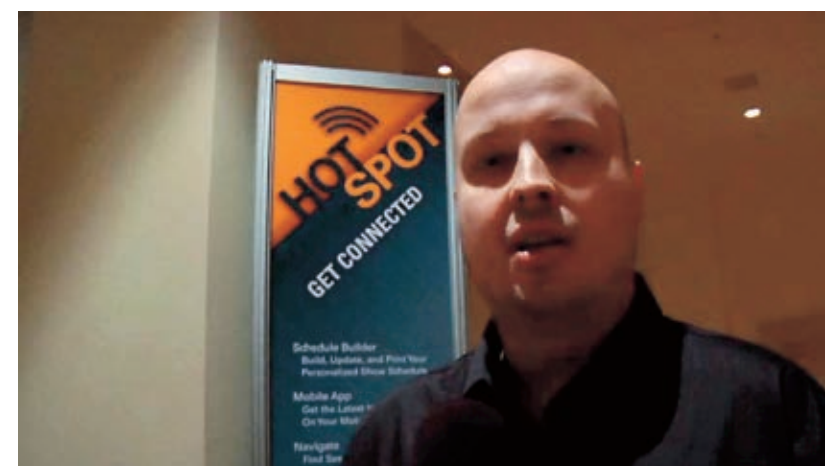
Use the power of JavaFX 2.0 to migrate Swing interfaces to JavaFX.

This is the first in a series of articles that will show how to modify an existing Swing-based application so that the interface can be migrated from Swing to JavaFX. This series will show how parts of the application can be replaced gradually with JavaFX to improve the appearance and usability of

To demonstrate the power of JavaFX, we'll use a relatively simple Swing application that has enough complexity in its components but can be easily understood and modified. The first ap-

Thankfully, most of the hard work is taken care of by the JavaFX platform. Included in the JavaFX API is the `JFXPanel` class. The name is a little misleading, because this class extends the Swing `JComponent` class rather than the Abstract Windows Toolkit (AWT) `Panel` class. As such, we can use this easily in a Swing application, but we can't use it in a pure AWT application.

WHAT'S NEXT
JavaFX 2.0
represents the
next step in the
evolution of Java
as a premier rich
client platform.



 Jasper Potts, the developer experience architect for Java client, talks about the JavaFX 2.0 release.

PHOTOGRAPH BY BOB ADLER

Listing 1 is a simple example of using this class. Let's work through the details, because some things aren't obvious.

In line 8, we instantiate a `JFXPanel` object, but initially we tell it nothing about what it will contain. That information will be provided later. This component can be added to a Swing Container just as any other Swing component.

To understand how we encapsulate the JavaFX scene, we need some details about threads in Swing and JavaFX. Swing uses a single event dispatch thread (EDT), and JavaFX uses an application thread. Both effectively do the same thing, which is to process asynchronous events that affect the UI (for example, button presses, updates to displayed values, window resizing, and so on).

As a Swing component, the `JFXPanel` object must be accessed only from the Swing EDT. The only exception to this is the `setScene()` method, which must be accessed from the JavaFX application thread.

In order to set the scene for the `JFXPanel` object, we must create another object that implements the `Runnable` interface, which can then be posted to the event queue and executed at some (unspecified) time in the future. We use the static utility method, `runLater()`, provided in the `Platform` utility class, and then an anonymous inner class to instantiate the `Runnable` object. Due to the inner class restrictions, the reference to the `JFXPanel` must be made final.

The `initFXComponent()` method is where the real JavaFX work happens. Because JavaFX uses a scene graph, we

need to create a **Scene** object to hold the parent node reference. When we instantiate this object, we can also set the size of the scene in pixels, which will then be used by the Swing layout manager to position the JavaFX component. We also set the root node of the scene to be the graph of objects that will be displayed.

If we wish to include more than one JavaFX scene in our Swing application, we simply instantiate more `JFXPanel` objects. To optimize the use of multiple JavaFX scenes, use a single call to `Platform.runLater()` and have the `run()` method of the inner class perform initialization for all scenes.

Swing and JavaFX Events

There is one more basic part of Swing and JavaFX integration that needs to be addressed, which is how the components interact when a Swing component triggers a change in a JavaFX scene or vice versa. If the JavaFX scene wants to interact with Swing components, the code must be wrapped in a **Runnable** object and placed on the EDT using the **SwingUtilities.invokeLater()** method, as shown in **Listing 2**.

For the opposite case, where a Swing component needs to interact with a JavaFX scene, the code must again be wrapped in a `Runnable` object, but this time it must be placed on the JavaFX application thread using the `Platform.runLater()` method, as shown in **Listing 3**.

Conclusion

So far, we've seen that adding a JavaFX scene to a Swing container requires

LISTING 1 / LISTING 2 / LISTING 3

```

1 public void init() {
2     setSize(300, 200);
3     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
4     setLayout(new BorderLayout());
5
6     /* Put the JavaFX scene in the middle */
7     fxComponent = new SimpleFXComponent(this);
8     final JFXPanel fxPanel = new JFXPanel();
9     add("Center", fxPanel);
10
11    /* Construct the JavaFX scene on its own application thread */
12    Platform.runLater(new Runnable() {
13        @Override
14        public void run() {
15            Scene scene = new Scene(fxComponent, 200, 100);
16            scene.setFill(Color.BLUEVIOLET);
17            fxPanel.setScene(scene);
18        }
19    });
20
21    /* Put the Swing button at the bottom */
22    changeButton = new JButton("Change JavaFX Button");
23
24    ...
25
26    add("South", changeButton);
27    setVisible(true);
28 }

```



only a few more lines of code than using a normal Swing component. Having the JavaFX scene and Swing components interact is also straightforward. You just have to remember to wrap the necessary code in a **Runnable** object and place it on the correct queue for execution.

In the next article, we'll take this knowledge and start to modify our Swing

application to add exciting new components that are created using `javaFX`. `</article>`

LEARN MORE

- [Documentation and tutorials](#)
- [API documentation \(Javadoc\)](#)
- [JavaFX Showcase](#)



JAMES L. WEAVER



Using Transitions for Animation in JavaFX 2.0

Animate the nodes in your scene the easy way.

JavaFX 2.0 is an API and runtime for creating rich internet applications (RIAs). JavaFX was introduced in 2007, and version 2.0 was released in October 2011. One of the advantages of JavaFX 2.0 is that the code may be written in the Java language, using mature and familiar tools. This article focuses on using JavaFX 2.0 *transitions* to animate visual nodes in the UI.

JavaFX comes with its own transition classes, shown in **Figure 1**, whose purpose is to provide convenient ways to do

commonly used animation tasks. These classes are located in the [javafx.animation](#) package. This article contains an example of using the [TranslateTransition](#) class to animate a node, moving it back and forth between two positions in the UI.

Overview

To help you learn how to use the [TranslateTransition](#) class, an example named [TransitionExampleX](#) will be employed. As shown in **Figure 2**, this example contains a couple of buttons, a couple of

rectangles, and a circle that you'll animate a little later.

The [TransitionExample](#) project that you'll download in the next section contains starter code for this example, and it has an appearance at runtime similar to **Figure 2**. During the course of this article, you'll modify the code to implement the animated behavior of the [TransitionExampleX](#) project, which is also available in the download.

When you click the Play button, the text on the button changes to "Pause," as shown in **Figure 3**, and the ball moves back and forth between the paddles indefinitely.

Clicking the Pause button causes the animation to pause and the text on the button to change to "Play."

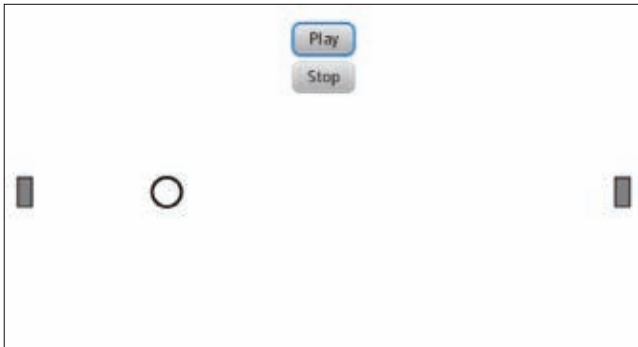


Figure 2



Figure 3

Class	Description
TranslateTransition	Translates (moves) a node from one location to another
RotateTransition	Rotates a node
ScaleTransition	Scales (increases or decreases the size of) a node
FadeTransition	Fades (increases or decreases the opacity of) a node
PathTransition	Moves a node along a geometric path
SequentialTransition	Allows you to define a sequential series of transitions
PauseTransition	Used in a SequentialTransition to wait for a period of time
ParallelTransition	Allows you to define a parallel series of transitions

Figure 1

Obtaining and Running the TransitionExample Example

1. Download the [NetBeans project](#) that includes the [TransitionExample](#) program.
2. Expand the project into a directory of your choice.
3. Start NetBeans, and select **File -> Open Project**.

This is an update to an [article](#) that was originally published on Oracle Technology Network (July 2011).

PHOTOGRAPH BY
STEVE GRUBMAN

4. From the Open Project dialog box, navigate to your chosen directory and open the **TransitionExample** project, as shown in **Figure 4**. If you receive a message dialog stating that the **jfxrt.jar** file can't be found, click the **Resolve** button and navigate to the **rt/lib** folder subordinate to where you installed the JavaFX 2.0 SDK.

Note: You can obtain the NetBeans IDE from the [NetBeans site](#).

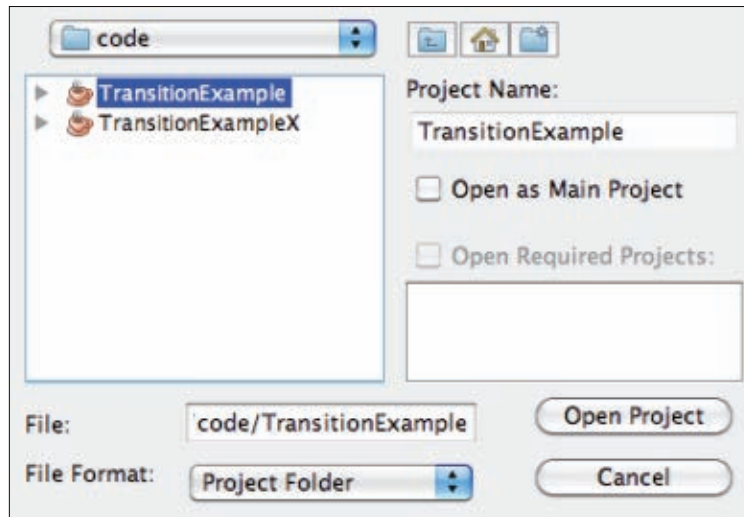


Figure 4



Figure 5



Figure 6

5. To run the application, click the **Run Project** icon on the toolbar, or press the F6 key. The **Run Project** icon has the appearance of the Play button on a DVD player, as shown in **Figure 5**.

The **TransitionExample** application should appear in a window, as shown in **Figure 6**.

You'll notice that clicking the Play and Stop buttons has no effect. Your mission will be to add code that implements the behavior described previously. Here are steps you can follow to implement this behavior:

Step 1: Create a TranslateTransition Instance to Animate the Node

To cause the ball to move (also known as *translate*) between two different positions in the scene, we'll create an instance of the **TranslateTransition** class.

Take a look at the code in the **TransitionExampleMain.java** file in the **TransitionExample** project, which shows the starter code for this example.

Using the TranslateTransitionBuilder class to build TranslateTransition.

The starter code in **TransitionExampleMain.java** makes use of *builder* classes in the JavaFX 2.0 API, in-

LISTING 1 LISTING 2 LISTING 3 LISTING 4

```
transition = TranslateTransitionBuilder.create()
// TO DO: Insert code to set the duration to 1500 milliseconds

.node(ball)
// TO DO: Insert code to set the fromX property to 0

// TO DO: Insert code to set the toX property to 440

.interpolator(Interpolator.LINEAR)
// TO DO: Insert code to set the autoReverse property to true

.cycleCount(Animation.INDEFINITE)
.build();
```

[See all listings as text](#)

cluding the **TranslateTransitionBuilder** class shown in **Listing 1**.

Go ahead and fill in the lines indicated by the "TO DO" comments, so the code in **Listing 1** turns into the code shown in **Listing 2**.

There are many builder classes in the JavaFX API, and their purpose is to enable a declarative style of programming to create and set the properties of objects. For example, the code you completed in **Listing 2** creates an instance of the **TranslateTransition** class, and it populates that instance with properties such as the duration of the animation and the node in the scene graph being animated. As you can see in **TransitionExampleMain.java**, other builder classes used in this application include **ButtonBuilder**, **CircleBuilder**,

RectangleBuilder, **SceneBuilder**, and **VBoxBuilder**.

Step 2: Define an Event Handler in the Play Button

To make the **TranslateTransition** start, pause, and stop, you'll define event handlers in the buttons that call the appropriate methods on the **TranslateTransition** object. For example, the starter code in **Listing 3** from **TransitionExampleMain.java** contains the builder that creates a **Button** instance for the Play button.

As you did before, fill in the lines indicated by the "TO DO" comments, turning the code shown in **Listing 3** into the code shown in **Listing 4**.

Using methods of the Animation class to control the animation. All the transition classes in **Figure 1** are subclasses of the

JAVA POWER

One of the **advantages of JavaFX 2.0** is that the code may be written in the Java language, using mature and familiar tools.

Using binding to keep properties updated.

Property binding is a very convenient and powerful feature of the JavaFX API, because it enables you to keep the values of properties automatically updated. The excerpts from **Listing 1** that are shown in

As the value of `playButtonText` is updated by the event handler that you coded, the text on the button displays the updated value.

To keep the `buttonsContainer` node horizontally centered in the application window, the `layoutXProperty` of one of the nodes in the scene graph is bound

Note: The excerpt in **Listing 6** is for demonstration purposes only, because the use of layout containers (in the `javafx.scene.layout` package) is the preferred approach for dynamically positioning nodes within a scene.


To finish the `TransitionExample`, you'll need to enter some code into the starter code from `TransitionExampleMain.java` shown in **Listing 7**.

Conclusion

JavaFX 2.0 comes with several transition classes that extend the [Transition](#) class, whose purpose is to animate visual nodes in your application. JavaFX also contains many builder classes that provide the ability to express a UI in a declarative style. Plus, JavaFX has a powerful property binding capability in which properties may be bound to expressions to automatically keep them updated. </article>

- [JavaFX Technology at a Glance](#)
- [JavaFX Documentation](#)

```
String playText = "Play";
...
StringProperty playButtonText =
    new SimpleStringProperty(playText);
...
playButton.textProperty()
    .bind(playButtonText);
```

 [See all listings as text](#)



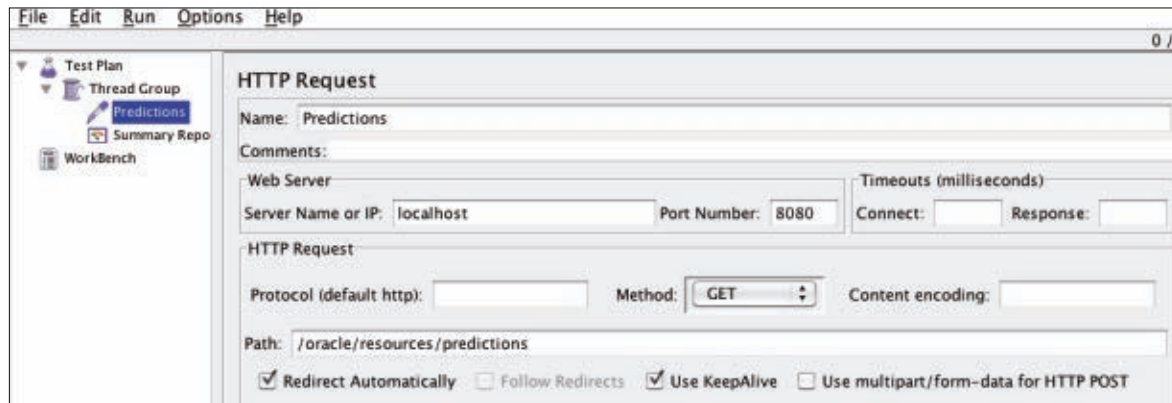


Figure 1

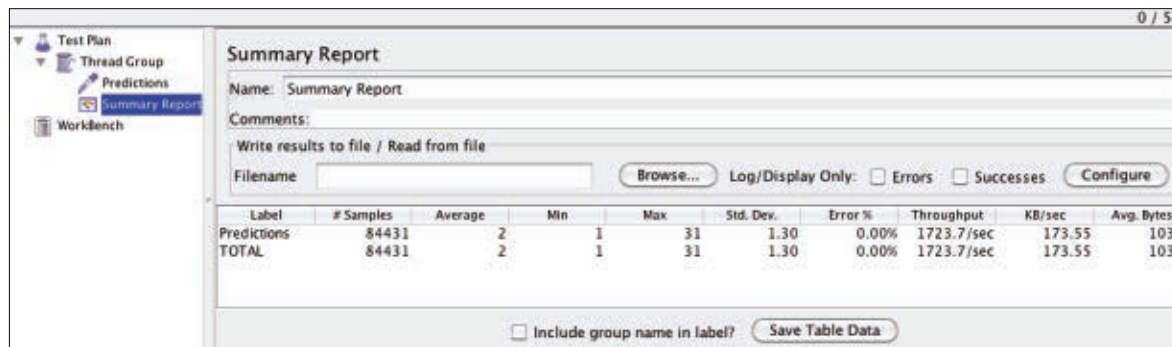


Figure 2

The open source load testing tool Apache JMeter comes with built-in HTTP support. After creating the **ThreadGroup** and setting the number of threads (and, thus, concurrent users), an HTTP request has to be configured to execute the GET requests (right-click, select **Sampler**, and then select **HTTP Request**). See **Figure 1**.

While the results can be visualized in various ways, the JMeter Summary Report is a good start (see **Figure 2**). It turns out that the sample application is able to handle 1,700 transactions per second for five concurrent users out of the box.

Every request is a true transaction
and is processed by an Enterprise

JavaBeans 3.1 (EJB 3.1) JAX-RS
[PredictionArchiveResource](#), delegated to
the [PredictionAudit](#) EJB 3.1 bean, which
in turn accesses the database through
[EntityManager](#) (with exactly one record).

At this point, we have learned only that with EJB 3.1, JPA 2, and JAX-RS, we can achieve 1,700 transactions per second without any optimization. But we still have no idea what is happening under the hood.

VisualVM Turns Night to Day

GlassFish Server Open Source Edition 3.1.x and Java DB (the open source version of Apache Derby) are Java processes that can be easily monitored with VisualVM. Although VisualVM is shipped

with the current JDK, you should check the [VisualVM Website](#) for updates.

VisualVM is able to connect locally or remotely to a Java process and monitor it. VisualVM provides an overview showing CPU load, memory consumption, number of loaded classes, and number of threads, as shown in **Figure 3**.

The overview is great for estimating resource consumption and monitoring the overall stability of the system. We learn from **Figure 3** that for 1,700 transactions per second, GlassFish Server Open Source Edition 3.1 needs 58 MB for the heap, 67 threads, and about 50 percent of the CPU. The other 50 percent was consumed by the load generator (JMeter).

Although this colocation is adequate for the purposes of this article, it blurs the results. The load generator should run on a dedicated machine or at least in an isolated (virtual) environment. Sometimes, you even have to run [distributed JMeter load tests](#) to generate enough load to stress the server. For internet applications, it might be necessary to deploy the load generators into the cloud.

In a stress test scenario, the plain numbers are interesting but unimportant. Stress tests do not generate a realistic load, but rather they try to break the system. To ensure stability, you should monitor the VisualVM Overview

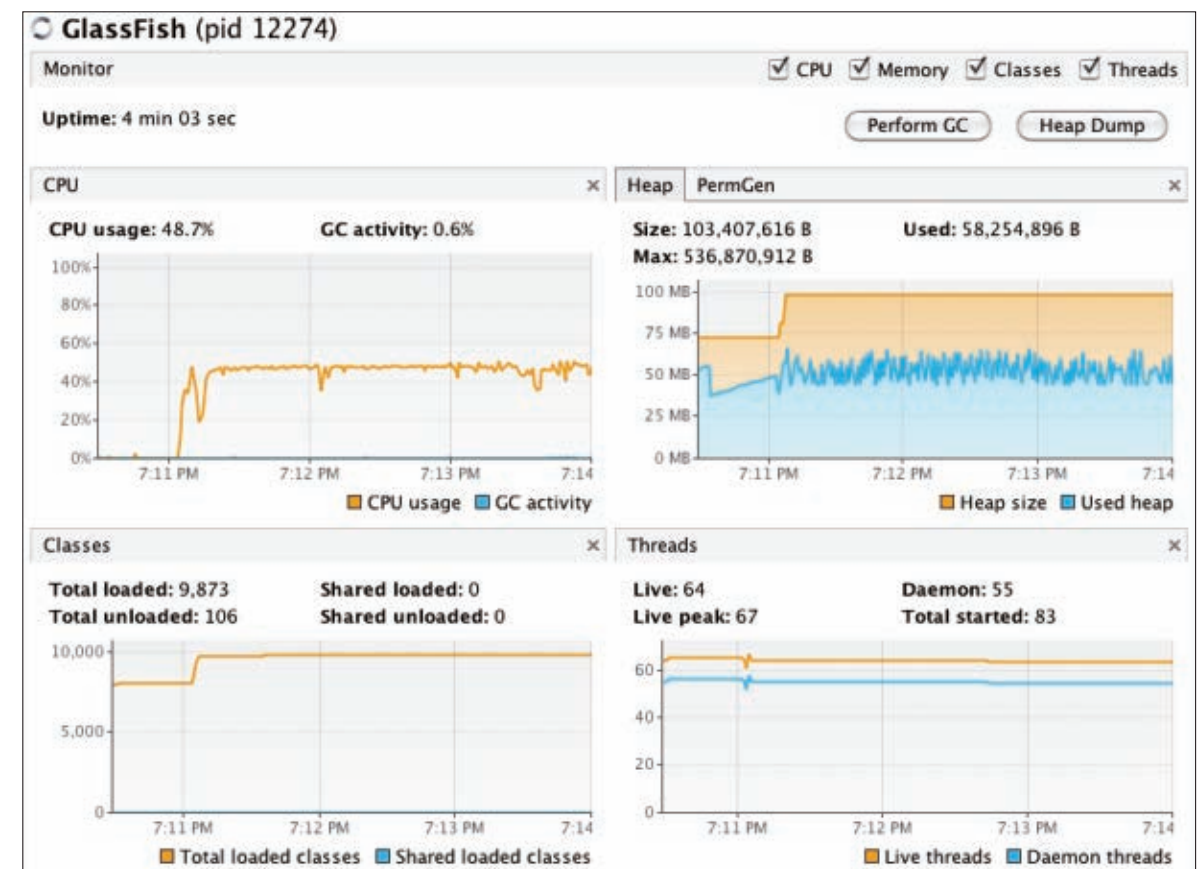


Figure 3

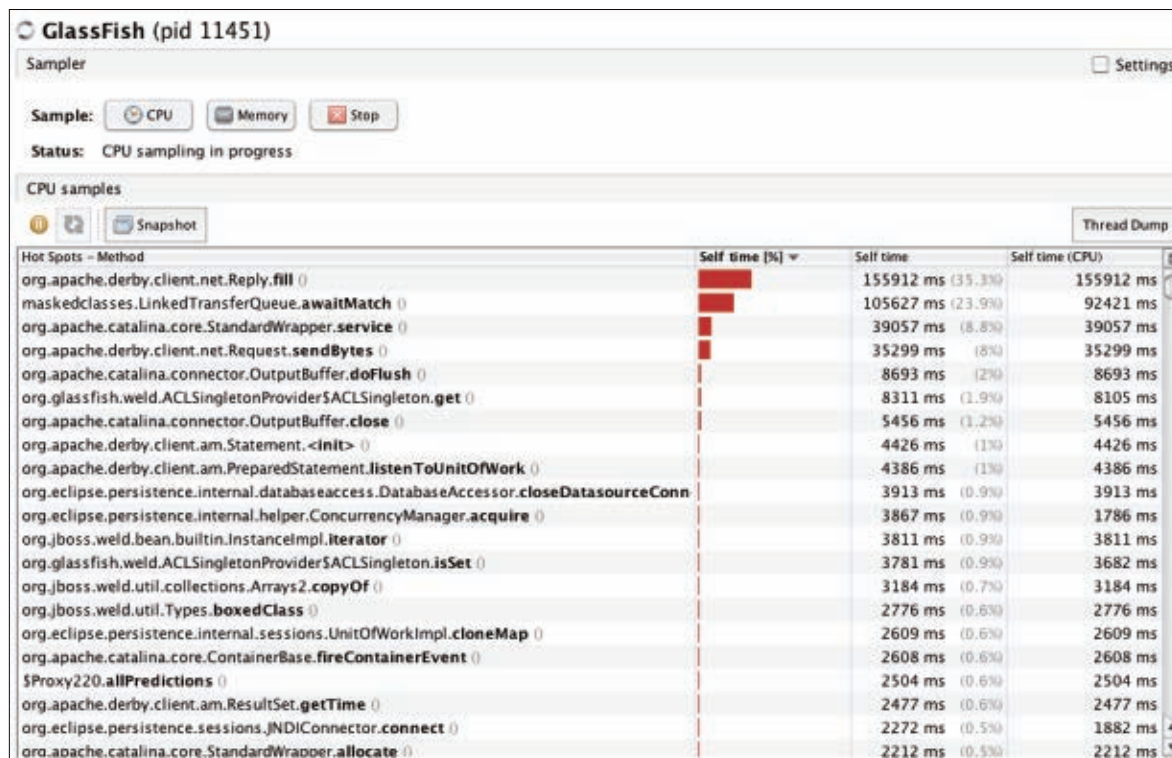


Figure 4

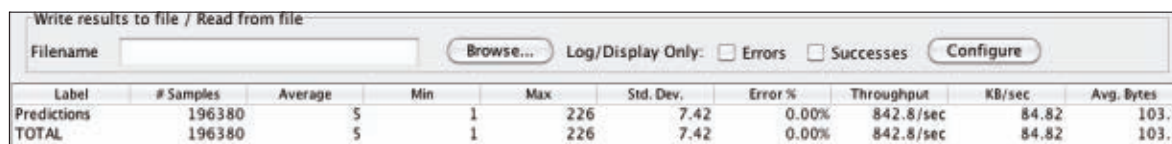


Figure 5

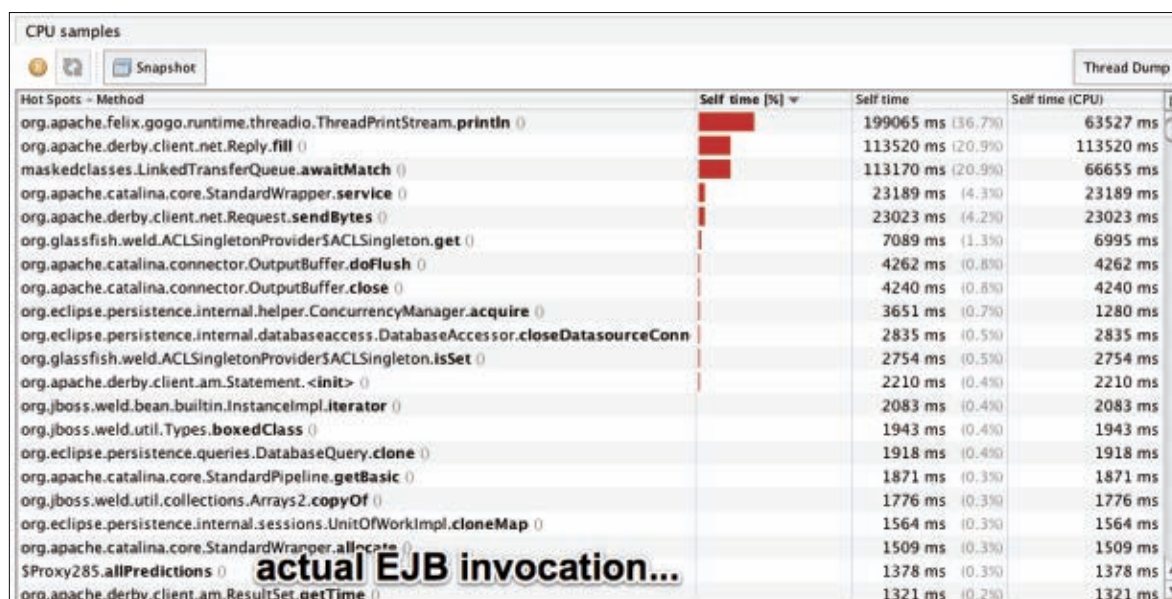



Figure 6

LISTING 1

```
public List<Prediction> allPredictions(){
    System.out.println("-- returning predictions");
    return this.em.createNamedQuery(Prediction.findAll).getResultList();
}
```

 [See all listings as text](#)

average values. All the lines should be, on average, flat.

An increasing number of loaded classes might indicate problems with class loading and can lead to an **OutOfMemoryError** due to a shortage of **PermGen** space. An increasing number of threads indicates slow, asynchronous methods. A **ThreadPool** configured with an unbounded number of threads will also lead to an **OutOfMemoryError**. And a steady increase in memory consumption can eventually lead to an **OutOfMemoryError** caused by memory leaks.

VisualVM comes with an interesting profiling tool called Sampler. You can attach and detach to a running Java process with a little overhead and measure the most-expensive invocations or the size of objects (see **Figure 4**).

The sampling overhead is about 20 percent, so with an active sampler, you can still achieve 1,400 transactions per second. As expected, the application spends the largest amount of time communicating with the database.

How Expensive Is System.out.println?

A single `System.out.println` can lead to significant performance degradation. To

measure the overhead, every invocation of the method `allPredictions` is logged with a `System.out.println` invocation, as shown in **Listing 1**.

Instead of 1,700 transactions per second, we are able to perform only about 800 transactions per second, as shown in **Figure 5**.

Let's take a look at the VisualVM Sampler output shown in **Figure 6**. More time is spent in `ThreadPrintStream.println()` than in the most expensive database operation.

The actual EJB 3.1 overhead is negligible. The `$Proxy285.allPredictions()` invocation is in the very last position and orders of magnitude faster than a single `System.out.println`.

Having a reference measurement makes identification of potential bottlenecks easy. You should perform stress tests as often as possible and compare the results. Performing nightly stress tests from the very first iteration is desirable. You will get fresh results each morning so you can start fixing potential bottlenecks.

Causing More Trouble

Misconfigured application servers are a common cause of bottlenecks. GlassFish

Server Open Source Edition 3.1 comes with reasonable settings, so we can reduce the maximum number of connections from the Derby pool to two connections to simulate a bottleneck. With five concurrent threads (users) and only two database connections, there should be some contention (see **Figure 7**).

The performance is still surprisingly good. We get 1,400 transactions per second with two connections. The max response time went up to 60 seconds, which correlates surprisingly well with the “Max Wait Time: 60000 ms” connection pool setting in GlassFish Server Open Source Edition 3.1. A hint in the log files also points to the problem, as shown in **Listing 2**.

Also interesting is the Sampler view in VisualVM (see **Figure 8**).

The method `JNDIConnector.connect()` became the most expensive method. It even displaced the `Reply.fill()` method from its first rank.

The package `org.eclipse.persistence` is the JPA provider for GlassFish Server Open Source Edition 3.1, so it should give us a hint about the bottleneck’s location. There is nothing wrong with the persistence layer; it only has to wait for a free connection. This contention is caused by the artificial limitation of having only two connections available for five virtual users.

A look at the `JNDIConnector.connect` method confirms our suspicion (see **Listing 3**). In the method `JNDIConnector.connect`, a connection is acquired from a `DataSource`. In the case of an empty pool, the method will block until either an

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Predictions	290190	3	0	60028	258.55	0.00%	1398.2/sec	146.14	107.0
TOTAL	290190	3	0	60028	258.55	0.00%	1398.2/sec	146.14	107.0

Figure 7

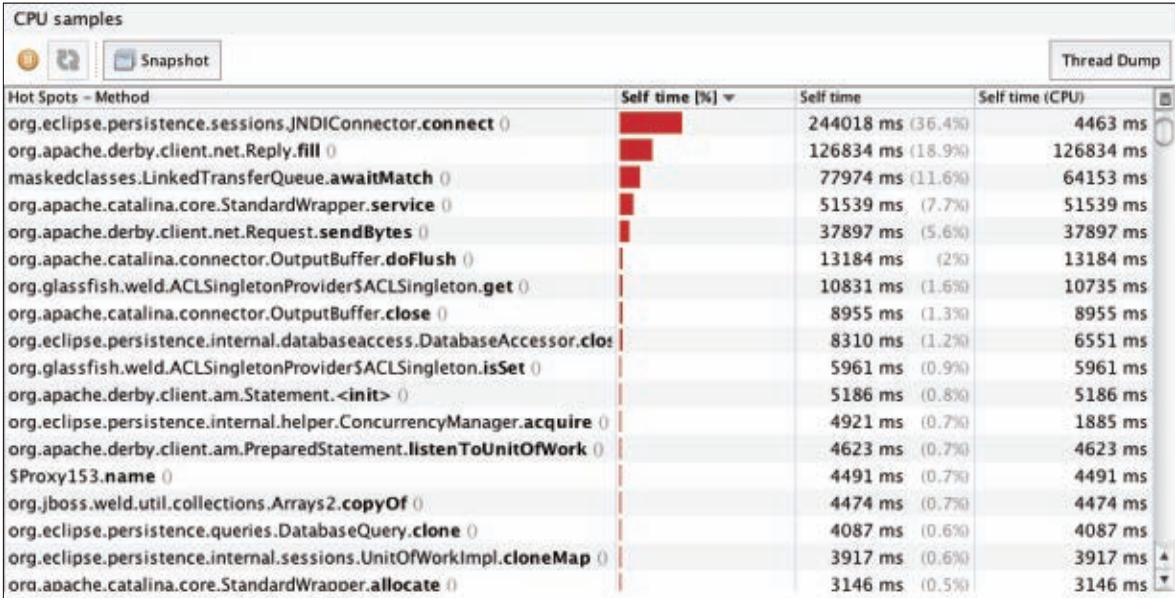


Figure 8

LISTING 2 LISTING 3

```
WARNING: RAR5117 : Failed to obtain/create connection from connection pool
[ SamplePool ]. Reason : com.sun.appserv.connectors.internal.api.PoolingException:
In-use connections equal max-pool-size and expired max-wait-time. Cannot allocate
more connections.
WARNING: RAR5114 : Error allocating connection : [Error in allocating a connection.
Cause: In-use connections equal max-pool-size and expired max-wait-time. Cannot
allocate more connections.]
```

See all listings as text

in-use connection becomes free or the Max Wait Time is reached. The method can block up to 60 seconds with the GlassFish Server Open Source Edition default settings. This rarely happens with the default settings, because the server ships with a Maximum Pool Size of 32 database connections.

How to Get the Interesting Stuff

The combination of JMeter and VisualVM is useful for ad hoc measurements. In real-world projects, stress tests should be not only repeatable but also comparable. A history of results with visualization makes the resultant comparison and identification of hotspots easier.

VisualVM provides a good overview, but the really interesting monitoring information can be obtained only from an application server in a proprietary way. All major application servers provide extensive monitoring information via Java Management Extensions (JMX).

GlassFish Server Open Source Edition 3.1 exposes its monitoring and management data through an easily accessible Representational State Transfer (REST) interface.

To activate the monitoring, open the GlassFish Admin Console by specifying the Admin Console URI (`http://localhost:4848`). Then select **Server**, select **Monitor**, and then select **Configure Monitoring**. Then select the **HIGH** level for all components. Alternatively, you can activate monitoring by using the `asadmin` command from the command line or by using the REST management interface.

Now, all the monitoring information is accessible from the following root URI: `http://localhost:4848/monitoring/domain/server`. The interface is self-explanatory. You can navigate through the components from a browser or from the command line.

The command `curl -H "Accept: application/json" http://localhost:4848/monitoring/domain/server/jvm/memory/`

`usedheapsize-count` returns the current heap size formatted as a JSON object (see **Listing 4**).

The most interesting key is `usedheapsize-count`. It contains the amount of used memory in bytes, as described by the description tag. The good news is that the entire monitoring API relies on the same structure and can be accessed in a generic way.

Monitoring Java EE 6 with Java EE 6

Executing HTTP GET requests from the command line still does not solve the challenge. To be comparable, the data has to be persistently stored. A periodic snapshot between 1 and 30 seconds is good enough for smoke tests and stress tests.

It turns out that you can easily persist

monitoring data with a simple Java EE 6 application. JPA 2, EJB 3.1, Contexts and Dependency Injection (CDI), and JAX-RS reduce the task to only three classes. The JPA 2 entity **Snapshot** holds the relevant monitoring data (see **Listing 5**).

The entity **Snapshot** represents the interesting data, such as the number of

busy threads, `queuedConnections`, errors, committed and rolled-back transactions, heap size, and the time stamp. You can extract such information from any other application server through different channels and APIs.

Listing 6 shows how to access REST services with Jersey. The managed bean `DataProvider` uses the Jersey client to access the GlassFish Server Open Source Edition's REST interface and convert the JSON result into Java primitives. The `fetchData` method is the core functionality of `DataProvider`, and it returns the populated `Snapshot` entity.

Every five seconds, the `MonitoringController @Singleton` EJB 3.1 bean shown in **Listing 7** asks the `DataProvider` for a `Snapshot` and persists it. In addition, the persisted data is exposed through REST. You can access all snapshots using `http://localhost:8080/stm/resources/snapshots`, and you will get `Snapshot` instances as a JSON object (see **Listing 8**).

Interestingly, a Java EE 6 solution is significantly leaner than a comparable Plain Old Java Object (POJO) implementation. Periodic timer execution, transactions, and **EntityManager** bookkeeping are provided out of the box in Java EE 6 but must be implemented in Java Platform, Standard Edition (Java SE).

Automating the Stress Test

Using the StressTestMonitor (STM) application, we can collect application server monitoring data systematically and persistently, and we can analyze and compare the stress test results after

LISTING 4

LISTING 5

LISTING 6


LISTING 7

LISTING 8

```
{
  "message": "",
  "command": "Monitoring Data",
  "exit_code": "SUCCESS",
  "extraProperties": {
    "entity": {
      "usedheapsize-count": {
        "count": 217666480,
        "lastsampletime": 1308569037982,
        "description": "Amount of used memory in bytes",
        "unit": "bytes",
        "name": "UsedHeapSize",
        "starttime": 1308504654922
      }
    }
  },
  "childResources": {}
}
```

GO FOR AVERAGE

In a **stress test scenario**, the plain numbers are interesting but unimportant. Stress tests do not generate a realistic load, but rather they try to break the system. To ensure **stability**, you should monitor the average values.

 [See all listings as text](#)



VIKRAM GOYAL



PHOTOGRAPH BY
JONATHAN WOOD/
GETTY IMAGES

Getting Started with GameCanvas and the Mobile Sensor API

Learn the concepts behind developing a location-aware game using the Mobile Sensor API.

As location-aware applications have become de rigueur, there has been an increasing demand to include location-based features in gaming as well. Most smartphones these days support the concepts behind location-aware games and applications, and Java is not behind in this respect. The Location API, JSR-179, provides all the nuts and bolts you need, so it is quite easy to incorporate location-based features in your games. For a quick overview of this API, see my article on [gaming basics](#).

This article is the first article in a two-part series. In this article, I introduce the concepts behind developing a location-aware game, and I discuss how to use the Mobile Sensor API, JSR-256.

In Part 2, I will cover the Location API and explain the concepts required for combining location data with information from sensors to make a complete game.

This two-part series is not about developing a full-fledged game;

rather, it introduces the concepts for how to combine location data with mobile sensors to make a game that can be the launching pad for a complete game. It introduces these concepts from a single player's perspective.

Note: The source code for the game I develop in this two-part series can be downloaded [here](#).

The Game

The idea behind the game that I develop in this two-part series is simple. There is a red ball on the screen (see **Figure 1**), and you, as the first player, have to roll the ball left or right. Your friend, who has a compatible phone and game, has to “catch” the ball as it falls off your screen.

Two things have to be done to enable this. First, your gaming MIDlet needs to know if there are other players near you who have the capability of being a catcher, while you announce your availability. Second, you need to roll the ball using the accelerometer

sensor on your phone and broadcast that to the other player so that her phone can catch it.

Understanding Gaming

From the simplest coding perspective, gaming is the art of repeatedly executing the same piece of code and updating screen variables depending on the user's (and game engine) input. A loop for repeatedly executing code is required in order to do the same thing over and over again until something in the gaming environment changes, due to either an external event (a player quits, for example) or an internal event (a time-out interval, for example). The loop checks an instance variable to make sure that none of those events has happened, and if none has, the repeated execution continues.

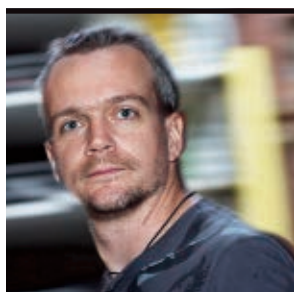
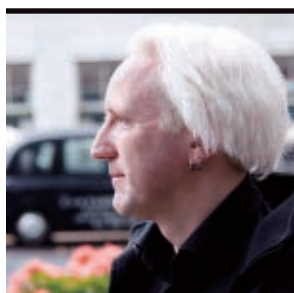
The loop checks for game inputs from the player and the environment and responds to them by updating its own internal variables and the game state.

The game state is reflected on the screen, and the screen is thus updated. The loop pauses for a predetermined interval to let the game refresh itself.

The Java Platform, Micro Edition (Java ME) API provides five classes that help with the development of games: [GameCanvas](#), [Layer](#), [Sprite](#), [TiledLayer](#), and [LayerManager](#). All these classes are in the [javax.microedition.lcdui.game](#) package and provide attributes and properties over and above the normal classes.



Figure 1



**BENJAMIN J. EVANS AND
MARTIJN VERBURG**

BIO

Polyglot Programming on the JVM

Tips about how and why to choose a non-Java language for your project

The following is an excerpt from *The Well-Grounded Java Developer*, which Manning Publications is due to release in January 2012.

The phrase *polyglot programming on the JVM* is relatively new. It was coined to describe projects that utilize one or more non-Java Java Virtual Machine (JVM) languages alongside a core of Java code. One common way to think about polyglot programming is as a form of separation of concerns. As you can see in **Figure 1**, there are potentially three layers where non-Java technologies can play a useful role. This diagram is sometimes called the polyglot programming pyramid and comes from the work of Ola Bini.

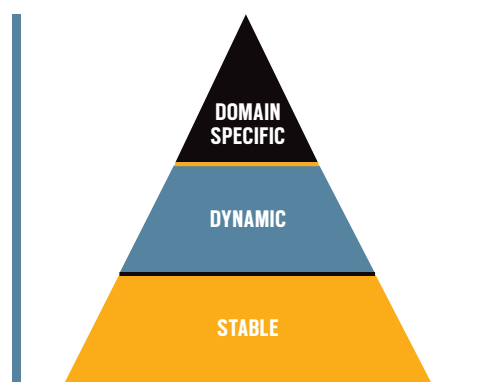


Figure 1

Within the pyramid, you can see three well-defined layers: domain-specific, dynamic, and stable. **Table 1** shows these three layers in more detail.

As you can see, there are patterns in the layers—the statically typed languages tend to gravitate toward tasks in the stable layer. Conversely, the less powerful and more specific-purpose technologies tend to be suited to domain roles at the top of the pyramid.

In the middle of the pyramid, we see that there is a rich role for languages in the dynamic tier. These languages are also the most flexible; in many cases, overlap might exist between the dynamic tier and either of the neighboring tiers.

Let's dig a little deeper into this diagram and look at why Java isn't the best choice for everything on the pyramid.

Why Use a Non-Java Language?

Java's nature as a general-purpose, statically typed, compiled language provides many advantages. These qualities make the language a great choice for implementing functionality in the stable layer.

LAYER	DESCRIPTION	EXAMPLES
DOMAIN-SPECIFIC	DOMAIN-SPECIFIC LANGUAGE (DSL); TIGHTLY COUPLED TO A SPECIFIC PART OF THE APPLICATION DOMAIN	APACHE CAMEL DSL, DROOLS, WEB TEMPLATING
DYNAMIC	RAPID, PRODUCTIVE, FLEXIBLE DEVELOPMENT OF FUNCTIONALITY	GROOVY, JYTHON, CLOJURE
STABLE	CORE FUNCTIONALITY, STABLE, WELL-TESTED, PERFORMANT	JAVA, SCALA

Table 1

However, these same attributes become a burden in the middle and upper tiers of the pyramid; for example:

- Recompilation is laborious.
- Static typing can be inflexible and lead to long refactoring times.
- Deployment is a heavyweight process.
- Java's syntax is not a natural fit for producing DSLs.

A pragmatic solution is to play to Java's strengths and take advantage of its rich API and library support to do the heavy lifting for the application—down in the stable layer.

If you're starting a new project from scratch, you might also find that another stable layer language (for example, Scala) has a

particular feature (for example, superior concurrency support) that is important to your project. In most cases, however, you should not throw out working stable layer code and rewrite it in a different stable language.

At this point, you may be asking yourself, "What type of programming challenges fit inside these layers? Which languages should I choose?" A well-grounded Java developer knows that there is no silver bullet, but there are some criteria you can consider when evaluating your choices.

How to Choose a Language for Your Project

Once you've decided to experiment with non-Java approaches in your project, you need to iden-

PHOTOGRAPHY BY
JOHN BLYTHE AND BOB ADLER



tify which parts of your project best suit a dynamic layer or domain-specific layer approach. **Table 2** highlights some tasks that might be suitable for each layer.

There is a wide range of natural use cases for alternative languages. However, identifying a task that could be accomplished with an alternative language is just the beginning. You now need to evaluate whether using an alternative language is appropriate. Here are some useful criteria that we take into account when considering technology stacks:

- Is the project area low risk?
- How easily does the language inter-operate with Java?
- What tooling and testing support is available for the language?
- How easy is it to compile, test, build, and deploy software in this language?
- How difficult is the learning curve?
- How easy or difficult is it to hire developers with experience in this language?

Let's dive into each of these areas so you get an idea of the sorts of questions you need to be asking yourself.

Is the Project Area Low Risk?

Let's say you have a core, payment-processing rules engine that handles more than 1 million transactions a day. It is a stable piece of Java software that has been around for more than seven years, but there are not many tests for it and there are plenty of dark corners in the code. The core of this engine is clearly a high-risk area for a new language, especially because it is running successfully and there is a lack of test

coverage and a shortage of developers who fully understand it.

However, there is more to a system than just its core processing. For example, this is a situation where better tests would clearly help. [Scala](#) has a great testing framework called `ScalaTest`. This enables developers to produce JUnit-like tests for Java code but without a lot of the boilerplate that JUnit seems to generate. So, once they are over the initial learning curve, developers can be much more productive at improving the test coverage. `ScalaTest` also provides a great way to gradually introduce concepts such as Behavior-Driven Development to the codebase. These concepts can really help when the time comes to refactor or replace parts of the core—regardless of whether the new processing engine ends up being written in Java or Scala.

Or, suppose that the operations users would like to have a Web console built so they can administer some of the noncritical, static data behind the payment-processing system. The developers already know Struts and JavaServer Faces, but they don't feel any enthusiasm for either technology. This is another low-risk area to try out a new language and technology stack. One obvious choice would be Grails. Developer buzz, backed up by some studies (including one by [Matt Raible](#)), is that Grails is the best available Web framework for productivity.

By focusing on a limited pilot in a low-risk area, the manager always has the option of terminating the project

and porting to a different delivery technology without too much disruption, if it turns out that the attempted technology stack was not a good fit for the team or system.

Does It Interoperate with Java?

You don't want to lose the value of all that great Java code you've already written. This is one of the main reasons organizations are hesitant to introduce a new programming language into their technology stack. However, with alternative languages that run on the JVM, you can turn this concern on its head—so it becomes about maximizing the existing value in your codebase, not about throwing away working code.

Alternative languages on the JVM are able to cleanly interoperate with Java and can, of course, be deployed in a pre-existing environment. This is especially important when discussing this step with the production management folks. By using a non-Java JVM language as part of your system, you'll be able to make use of their expertise in supporting the existing

environment. This can help reduce risk and alleviate any worries they might have about the new solution.

Note: DSLs are typically built using a dynamic (or, in some cases, stable) layer language, so many of them run on the JVM via the languages they were built in.

Some languages interoperate with Java more easily than others. We've found that most popular JVM alternatives have good interoperability with Java. Run a few experiments first to really make sure you understand how the integration can work for you.

Take [Groovy](#), for example. You can simply import Java packages directly into its code using the familiar **import** statement. You can build a quick Website using the Groovy-based Grails framework and yet still reference your Java model objects. Conversely, it's very easy for Java to call Groovy code in a variety of ways and to receive familiar Java objects. One example use case could be calling out to Groovy from Java to process some JSON and having a Java object be returned.

Layer	Example Problem Domains
Domain-Specific	Build, Continuous Integration, and Continuous Deployment Dev-Ops Enterprise Integration Pattern (EIP) Modeling Business Rules Modeling
Dynamic	Rapid Web Development Prototyping Interactive Administrative/User Consoles Scripting Test-Driven Development (TDD) and Behavior-Driven Development (BDD)
Stable	Concurrent Code Application Containers Core Business Functionality

Table 2

means that the team needs to think carefully about how to divide up a project, especially for deployment of separate but related components.

Is It Difficult to Learn?

Some languages (for example, Groovy) have had long-standing IDE support for compiling, testing, and deploying the end result. Other languages might have tooling that is not as fully matured yet. For example, Scala's IDEs are not as polished as those of Java, but Scala

One alternative is to look at the JVM languages that are reimplementations of existing languages. Ruby and Python are well-established languages with plenty of available material with which developers can educate themselves. The JVM incarnations of these languages could provide a sweet spot for your teams to begin working with an easy-to-learn, non-Java language.

Do Developers Use the Language?

Organizations have to be pragmatic; they can't always hire the top two per-

Again, the reimplemented languages can potentially help here. Few developers might have JRuby on their résumés, but because it is just Ruby on the JVM there is actually a large pool of developers to hire from.

Note: One word of warning about the reimplemented languages: Many existing packages and applications written, for example, in Ruby, are tested only against the original, C-based implementation. This means that there might be problems when trying to use them on top of the JVM. When making platform decisions, you should factor in extra testing time if you are planning to leverage an entire stack written in a reimplemented language.

Summary

For the polyglot programmer, languages fall roughly into three programming layers: stable, dynamic, and domain-specific. Languages such as Java and Scala are best used for the stable layer of software development, while others, such as Groovy and Clojure, are more suited to tasks in the dynamic or domain-specific realms.

Certain programming challenges fit well into particular layers, for example,

It's worth emphasizing again that the core business functionality of an existing production application is almost never the correct place to start when introducing a new language. The core is where high-grade support, excellent test coverage, and a proven track record of stability are paramount. Rather than start here, choose a low-risk area for the first deployment of an alternative language.

Finally, always remember that every team and project has its own unique characteristics that will affect the language choice. When choosing to implement a new language, managers and senior techs must consider the nature of their projects and team.

A small team composed exclusively of experienced propellerheads might choose Clojure for its clean design, sophistication, and power. Meanwhile, a Web shop that is looking to grow the team quickly and attract young developers might choose Groovy and Grails for the productivity gains and relatively deep talent pool. **</article>**

LEARN MORE

- *The Well-Grounded Java Developer*
- *Scala in Action* (Manning, 2010)
- *AspectJ in Action, Second Edition* (Manning, 2009)
- *DSLs in Action* (Manning, 2010)

Note: Use code **evans0735** to save 35 percent on your next manning.com purchase .

WHEN TO EVALUATE

There is a wide range of natural use cases for **alternative languages**. However, identifying a task that could be accomplished with an alternative language is just the beginning. You **now need to evaluate** whether using an alternative language is appropriate.

a method, but without all the surrounding class structure.

Java Magazine: Will you give us an overview of some of the most important languages running on the JVM? And how might they be used with Java to improve developer productivity and create more-flexible and robust applications?

Buckley: Large systems today succeed based on their technical architecture. That architecture typically has many loosely coupled components, each fulfilling a very specific purpose. Typically, these components collaborate by sharing data in a relational database, or a message bus, or by exposing XML services to each other. The more loosely coupled this architecture, the easier it is to choose the best language for the different components.

User interfaces are generally Web-based and written in HTML or JavaScript, with middleware often written in Java, and back-end logic written in Java, Ruby, or Scala. Groovy is fairly common at build time, and server-side JavaScript is growing in popularity.

A well-designed architecture is clear about the functional components and how they interact. Then the appropriate language can be chosen for those components. Organizations are searching for the right mix of productivity and maintainability in the languages they use. Especially on the JVM, components can be swapped in and out over time. Sometimes a Java component is rewritten in Scala, and sometimes a JRuby component is rewritten in Java. In many cases, the same tool suite can be used on the

JVM regardless of the source language.

Java Magazine: Oracle's upcoming Nashorn JavaScript interpreter was covered at the JVM Language Summit. Can you give us some details?

Buckley: Nashorn is an Oracle-led project currently scheduled to be available to developers in JDK 8. It's an implementation of JavaScript running on the JVM, targeted specifically for the benefit of Java developers. We want to expose JavaScript to Java collections and Enterprise JavaBeans. And we want to ensure that the boundary between Java and JavaScript is as thin and interoperable as possible.

The development of a JavaScript interpreter is partially motivated by the fact that much of the world is moving toward HTML5 for cross-platform GUIs. HTML5 offers many JavaScript APIs for accomplishing some very useful tasks, such as location sensing or database access in the browser. The scope of HTML5 has expanded far beyond "markup" for layout, and the natural way for these additional front-end services to be exposed is through JavaScript. Nashorn would allow a non-browser JavaScript program running on the client or server direct access to standard JavaScript APIs and the vast collection of Java APIs.

The use of `invokedynamic` and method handles makes implementing JavaScript, which is an extremely dynamic and weakly typed object-based language, on the JVM surprisingly straightforward.

Java Magazine: How will the `invokedynamic` instruction be directly useful to the Java language in Java SE 8?

Buckley: The implementation of lambda expressions ([Project Lambda](#)) is a good example. It's something of a challenge, because the JVM is conceptually streamlined and very focused. The JVM has no knowledge of a lambda expression, a closure, or a continuation. It has no knowledge of type inference or any other language feature that might be designed in support of lambda expressions.

For many years, we could add features to the Java language that mapped directly to JVM features, or in the case of generics, did not show up in the JVM at all. That era of direct mapping is over. With Project Lambda, we needed a way to represent lambda expressions on top of the JVM in a way that is efficient and extensible. In terms of disk space, memory usage, and other overhead, anonymous inner classes were not a reasonable solution.

The `invokedynamic` instruction provided the solution. It essentially facilitates quick access to method handles. Method handles are really function pointers, and function pointers are a way of manipulating individual blocks of code, which is what lambda expressions are all about. The goal is to give the Java compiler a great deal of flexibility in how to represent these lambda expressions at runtime. That is accomplished by using `invokedynamic` to set up their creation.

Java Magazine: What are some examples of how lambda expressions could benefit Java developers in a tangible way?

Buckley: Lambda expressions are relevant with APIs. The bulk of what developers do involves utilizing someone else's APIs. A very common task in Java programming is iterating through a collection of business objects, finding ones that match some requirement, and passing that result on to the next step of the program.


Rather than extracting items and writing code to perform those operations (and hoping that nobody else is mutating the collection at the same time), using lambda expressions simplifies things dramatically. If you can pass code, in the form of a lambda expression, to a library, you are basically passing it to the collection itself. Then, concerns such as thread safety become the responsibility of the library implementer, rather than the responsibility of the calling programmer.

High-level operations improve software readability and performance.

From Lambdas to Bytecode

Translation options

- Could translate directly to method handles
 - Desugar lambda body to a static method
 - Capture == take method reference + curry captured args
 - Invocation == MethodHandle.invoke
- Whatever translation we choose becomes not only implementation, but a binary specification
 - Want to choose something that will be good forever
 - Is the MH API ready to be a permanent binary specification?
 - Are raw MHs yet performance-competitive with inner classes?




Brian Goetz, From Lambdas to Bytecode



In the last issue, Arun Gupta, a Java evangelist at Oracle, posed a challenge about the CDI specification and asked readers for a fix.

The correct answer is #2: "beans.xml" is required to enable injection. The CDI specification requires "beans.xml" in the WEB-INF directory in order for bean injection to work. So even though the code is fine, a missing file did not allow the EJB injection to go through.

And now, for Gupta's latest challenge.

1 THE PROBLEM

Java EE 6 allows you to create Web applications very easily using annotations on POJOs. A POJO can be easily converted to a Servlet by adding @WebServlet and extending HttpServlet. A POJO becomes a JPA bean by adding @Entity. Each method of a Java class becomes implicitly transactional by adding @Stateless. Following the convention-over-configuration paradigm, the deployment descriptors are optional for most of the common cases.

PHOTOGRAPH BY MARGOT HARTFORD
ART BY I-HUA CHEN

2 THE CODE

Consider the following piece of code to have database access within a Servlet:

```
@WebServlet(name = "TestServlet", urlPatterns = {"/TestServlet"})
public class TestServlet extends HttpServlet {

    @PersistenceContext
    EntityManager em;

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        // Invoke methods on Entity Manager
    }
}
```

3 WHAT'S THE FIX?

Is this code guaranteed to work in a multithreaded environment?

- 1) Of course. All the samples are written this way.
- 2) No. Servlets are re-entrant, and EntityManager is not thread-safe. Instead inject as

```
@PersistenceUnit EntityManagerFactory em;
```

and then get EntityManager within the doGet() method.

- 3) Database access in Servlets must be done through EJBs only. The correct way is to move the database code in an EJB, inject the EJB in this Servlet, and invoke methods on the EJB.
- 4) EntityManager injection will not work in Servlets. Instead use Persistence.createEntityManager to obtain EntityManager in the doGet() method and then invoke operations on it.



GOT THE ANSWER?

Look for the answer in the next issue. Or submit your own code challenge!

