JDJ'S EXCLUSIVE JAVAONE COVERAGE

JAVA™ DEVELOPER'S JOURNAL

The World's Leading Java Resource

Volume: 4 Issue: 7, July 1999

JAVADEVELOPERSJOURNAL.COM

INSIDE THIS ISSUE:

Exclusive SYS-CON Radio Interviews with

Scott McNealy &
James Gosling

pg.18

"We grew faster than Intel, IBM, Compaq, SCI and everybody else"

"My fantasy for how Java should evolve is that nobody should be aware that it exists"

SYS-CON PUBLICATIONS

"JDJ is the No. 1 Java Publication!"
—George Paolini

JAVA DEVELOPER'S JOURNAL

Editor-in-Chief:	Sean Rhody
Art Director:	Jim Morgan
Executive Editor:	M'lou Pinkham
Production Editor:	Cheryl Van Sise
Assistant Editor:	Nancy Valentine
Editorial Consultant:	Scott Davison
Technical Editor:	Bahadir Karuv
Product Review Editor:	Ed Zebrowski
Industry News Editor.	Alan Williamson
E-commerce Editor.	Ajit Sagar

WRITERS IN THIS ISSUE
Jim Crafton, Steven Feuerstein, Ron Harris, Rick Hightower, Scott Howard, Christopher Lambert, Vasu Ramachandriah, Sean Rhody, Rick Ross, Ajit Sagar, Prasad Thammineni, Jason Westra, Alan Williamson

SUBSCRIPTIONS
For subscriptions and requests for bulk orders, please send your letters to Subscription Department

Subscription Hotline: 800 513-7111

Cover Price: $4.99/issue
Domestic: $49/yr. (12 issues) *Canada/Mexico:* $69/yr.
Overseas: Basic subscription price plus airmail postage (U.S. Banks or Money Orders). *Back Issues:* $12 each

Publisher, President and CEO:	Fuat A. Kircaali
Vice President, Production:	Jim Morgan
Vice President, Marketing:	Carmen Gonzalez
Chief Financial Officer:	Ignacio Arellano
Accounting Manager:	Eli Horowitz
Circulation Manager.	Mary Ann McBride
Advertising Account Managers:	Robyn Forma Megan Ring
Project Coordinator:	Jaclyn Redmond
Advertising Assistant:	Christine Russell
Graphic Designers:	Robin Groves Alex Botero
Graphic Design Intern:	Aarathi Venkataraman
SYS-CON Radio Editor:	Chad Sitler
Webmaster:	Robert Diamond
Web Services Intern:	Digant B. Dave
Customer Service:	Sian O'Gorman Ann Marie Milillo
Online Customer Service:	Amanda Moskowitz

EDITORIAL OFFICES
SYS-CON Publications, Inc.
39 E. Central Ave., Pearl River, NY 10965
Telephone: 914 735-7300 Fax: 914 735-6547
Subscribe@SYS-CON.com

JAVA DEVELOPER'S JOURNAL (ISSN#1087-6944) is published monthly (12 times a year) for $49.00 by SYS-CON Publications, Inc., 39 E. Central Ave., Pearl River, NY 10965-2306. Application to mail at Periodicals Postage rates is pending at Pearl River, NY 10965 *and additional mailing offices.*
POSTMASTER: Send address changes to:
JAVA DEVELOPER'S JOURNAL, SYS-CON Publications, Inc., 39 E. Central Ave., Pearl River, NY 10965-2306.

Worldwide Distribution by
Curtis Circulation Company
739 River Road, New Milford NJ 07646-3048 Phone: 201 634-7400

Sean Rhody, Editor-in-Chief

Testing My Patience

I recently attended a technical conference and sat in on an interesting discussion concerning moving from traditional testing to testing of object-oriented systems. As many of you know, testing and quality assurance is one of my pet peeves. All too often, groups of otherwise intelligent, experienced software developers become spineless jellyfish when it comes to putting together and sticking with a realistic estimate for the amount of testing needed to ensure that the system under development is completed and debugged. We've all paid the price for this neglect. How many service packs, patches and point updates have you had to install for your commercial software? And that's just the tip of the iceberg. The press rarely gets a glimpse inside the halls of industry to report on the situation within internal development groups. Gross underestimation of the testing effort involved in software development is a significant contributor to our software problems.

One of the root causes of this difficulty is the lack of understanding around the testing process. Many software development organizations have abandoned the traditional waterfall methodology of software development in favor of iterative or rapid-application development approaches. These approaches often have significant business advantages – such as shorter time to market – that justify the departure from the more rigorous waterfall method, but they have a significant impact on how, when and how much testing must occur. It's common knowledge that a defect discovered in design is several orders of magnitude less expensive to fix than a defect caught after implementation. Likewise, the cost to fix a bug that is discovered after software is shipped or deployed is even greater. The effort required to test in an iterative environment increases because the pace and rapid change involved in RAD approaches often lead to the introduction of new defects, and the reemergence of old ones. Additionally, it's difficult to ensure that sufficient testing occurs before the product is released, so a defect must often be corrected in the next release.

Web and distributed computing add significant complication to this already busy picture. At the simplest level, a distributed, component-based approach requires the creation of testing harnesses because components need to be tested in isolation, as components, in addition to the testing they receive when the entire system is tested. This isn't as easy as exercising a screen, as there is typically no GUI interface for the individual component. In the Java world we create components for either CORBA or EJB. Then we build screens, or pages, to use these components. To unit-test the component, we have to have some program that will exercise the component. So, at a minimum, extra coding is required.

In reality, I'm afraid the situation is somewhat worse. I've known people who believe that object-oriented development decreases the amount of testing required because only the changed objects need to be retested. I can see where they're coming from with this, but I don't think their points are truly valid. Encapsulation of behavior doesn't lead to encapsulation of defects. If this were so, a bug in one of the Windows system DLLs wouldn't be capable of bringing the entire system to a halt.

The only solution I'm aware of for these problems is adequate testing. One expert I spoke with recently suggested that every iteration of a system developed using object-oriented techniques needs to be fully tested, not just unit-tested. Obviously this is a departure from many testing methodologies, where integration and acceptance testing occur only at the end of the process, even when using a RAD approach.

I'm sure you hate discussing testing with project planners and management as much as I do. Many of the proponents of object-oriented programming misunderstand the cost savings involved when they claim it will increase development agility. They neglect to emphasize that these improvements will be measured over time, and are unlikely to be realized in just a single project. It takes several projects, leveraging previous work, to gain this advantage. Nowhere is this more apparent than when we come to the testing area. So next time you're on the spot for a testing estimate, try to get a little education concerning the amount of testing needed. And stick to your guns.

About the Author
Sean Rhody is the editor-in-chief of Java Developer's Journal. *He is also a senior consultant with Computer Sciences Corporation, where he specializes in application architecture – particularly distributed systems. He can be reached by e-mail at sean@sys-con.com.*

RON HARRIS, PRESIDENT AND CEO, PERVASIVE SOFTWARE

From the Wallet to the Web, Java Cards Get Pervasive

Based on published research by industry analysts, over 400 million special-purpose electronic, non-PC embedded devices will be sold this year – and over 1.2 billion units by the year 2001. By 2002, analysts predict shipments of smart handheld devices such as PDAs, Palm Pilots and smart phones will reach 25 million units. By year 2003, analysts project that smart cards equipped with miniature databases could make their way into virtually every purse and wallet in America. More than a billion chip cards were shipped worldwide in 1998 alone.

Imagine carrying your new multi-application smart card, which may have credit, debit and reloadable electronic cash functions. Add to that your personal profile information and the ability that lets you store loyalty programs and loyalty points on the card. With the security of a smart card and the power of Pervasive's database, an online Internet transaction can be placed securely without your having to provide your credit card number. Electronic copies of the receipts can be recorded on the card, as can new loyalty programs you may choose to participate in. With a standard method for storing and accessing data, such as Pervasive's Java Card SQL database, the electronic receipts can later be queried and exported directly into desktop applications such as Quicken. Imagine never worrying about lost receipts or filling out corporate expense reports! Never before has this level of application and database interoperability been available in smart cards.

In partnership with Schlumberger, the leading supplier of smart card technology, Pervasive is participating in the Java Card Forum Database Subcommittee to lead the effort to promote an industry-standard database interface for smart cards. The objectives of the database subcommittee are to define an open, on-card database access API for Java Card, basing the on-card database API on JDBC, and to define a SQL dialect for use with all on-card implementations. Pervasive is committed to providing an industry-standard open card specification promoting open database access across all cards.

Pervasive's small-footprint database, Pervasive.SQL 2000 for Smart Cards, is a sub-8 K engine designed to integrate with the Java Card environment. It provides both on-card and off-card interfaces; the former is referred to as Java Card-JDBC or JC-JDBC, while off-card the engine can be accessed using standard desktop interfaces such as ODBC and/or JDBC.

Pervasive.SQL 2000 for Smart Cards was announced in early June as part of a family of database information management solutions that also includes Pervasive.SQL for Embedded Devices as a sub-100 K data engine for real-time systems such as VxWorks, WinCE, QNX, Neutrino, Phar Lap and PalmOS. This highly portable, full-featured, small-footprint engine is well suited for use in Internet-enabled information appliances such as set-top boxes, Internet screen phones, Web-enabled PDAs and even smart refrigerators. A sibling, Pervasive.SQL for Mobile Devices, is a sub-350 K data engine for WinCE that includes integrated replication, remote access and event management – designed to fill the needs of nomadic applications.

The big brother of the family, Pervasive.SQL 2000 for PC platforms, is built on decades of experience in delivering database software for use in low-maintenance applications. The new release features industry-standard SQL support and a type III JDBC driver in the SDK.

To summarize, Pervasive's goal is to support applications from the wallet to the Web. Realizing that no one size fits all, Pervasive has developed appropriate solutions for all applications. These solutions integrate with one another, expose compatible interfaces and support data replication. For the wallet (Java cards) Pervasive has developed a sub-8 K SQL engine that supports a significant subset of the JDBC API. For the Web there's Pervasive.SQL 2000. Pervasive.SQL for Embedded Devices and Pervasive.SQL for Mobile Devices round out the product line and provide support for everything in between.

You can learn more about these exciting technologies through Pervasive's Beta and partner programs. Beta release of the complete Pervasive.SQL 2000 family is available for download at www.pervasive.com. Developer resources for Pervasive's products are available from www.pervasive.com/developerzone.

ABOUT THE AUTHOR

Ron Harris, president and CEO of Pervasive Software, has been involved in successful entrepreneurial ventures for most of the last decade. He spun Pervasive Software out of Novell, Inc., in 1994 and engineered Pervasive's successful IPO in 1997 and a secondary IPO in 1999. Harris is also a cofounder of Citrix Systems.

Developing with DCOM and Java

Part One

DCOM Architecture and the JVM compared to CORBA and RMI

by Rick Hightower

Developing distributed components with Java and DCOM (distributed component object model) simplifies developing distributed applications. If you know CORBA or RMI, DCOM is easy to learn. Microsoft's Java Virtual Machine makes developing COM and DCOM components painless.

Overview of COM and DCOM

The Component Object Model provides a means to create extensible services called components. As *components* mature (evolve) and add new features and functions, the services they provide remain backward-compatible with older incarnations of the components they replace. This enables older applications (COM clients) to treat new components like older components. Thus, when you upgrade the component, older client applications continue working.

COM uses polymorphism to accomplish the extensible component architecture. COM compares to LPC roughly the same way C++ compares to C, one being procedural and the other object-oriented. A remote procedure call (RPC) is to DCOM as C is to C++. DCOM groups data and methods into objects that you can use through various interfaces. Think of DCOM as COM with a longer wire. The terms DCOM and COM are thus used interchangeably throughout this text.

COM is designed from the ground up to support distributed computing. Just by changing a few Windows NT Registry settings, you can use a legacy COM client with a DCOM server, or the client can request a specific server.

The major difference between DCOM and COM is that DCOM uses RPC. You can transparently use DCOM with COM clients that predate the release of DCOM. You can also use most existing COM servers that predate the release of DCOM as DCOM servers – again, just by changing a few registry settings.

At some point you'll probably have to deal with COM/DCOM. Knowing COM is a good skill if you have to interface with commercial off-the-shelf components and applications or existing in-house applications that use DCOM. Because Windows NT is prevalent in the client/server market, and DCOM is heavily integrated with the NT operating system, it's important to understand DCOM because of the proliferation of Windows NT and the size of the COM component market.

DCOM currently ships with the Microsoft Windows NT 4.0 and Windows 98 operating systems. It's also available for download for the Windows 95 operating system. In addition, there are efforts to make DCOM available on a number of UNIX platforms. Many of the Java application server providers, such as BEA's WebLogic Java server and Bluestone's Sapphire, provide DCOM support.

Microsoft has made sure that their JVM integrates well with DCOM. Thus Java classes are treated like COM objects. Also, Sun provides an ActiveX bridge to expose JavaBeans as ActiveX controls. (An ActiveX control is a type of COM component.) In addition, Halcyon provides a Java DCOM server.

With their JVM you can use the Java classes you create as scriptable COM components. Java classes can be scripted using Visual Basic, VBScript, JScript, Perl, Python and other scripting languages. These classes can be used inside a Web browser or an Excel spreadsheet, or as part of an Active Server Page. Essentially, you can use your Java classes anywhere you can use Automation (late binding). Automation enables users to take control of components and applications through easy-to-use scripting languages such as Visual Basic.

With Microsoft's JVM you can use COM components and ActiveX control (which are written in other languages) as Java classes and JavaBeans. You can use your COM components virtually anywhere you can use Java.

You may wonder why all this matters to Java. It's simple, really. There are a lot of COM components out there, and chances are you'll need to integrate them in one of your projects. For that matter, there are a lot of COM component developers in the market, and you may need to integrate their skills in your next project.

Saying there are a lot just doesn't cut it – I wanted numbers. So I did a little research on the demand for COM/DCOM and related technology skills versus CORBA and RMI skills. I looked up want ads under "Information Systems" in the employment sections for five major cities. I searched for keywords regarding DCOM, CORBA and RMI. What I found is shown in Table 1 and Figures 1 and 2.

Whatever our backgrounds, we all have have one thing in common – at one time in our lives we were looking for work. I was looking when I found my current job. The demand for COM/DCOM skills is anywhere from 20 to 400% greater than for CORBA skills. With this in mind let's review what COM is.

The Component Revolution

Declaring one technology the winner and any of the others the loser is impossible. There's something else of greater importance that all these technologies supply: the plumbing for the component revolution.

	COM/DCOM	CORBA	RMI
Phoenix	58	29	0
San Francisco	1718	1428	31
Los Angeles	418	100	2
Tampa Bay	70	24	0
New York	763	486	16
Chicago	223	132	6
Total	3250	2199	55

Table 1: Comparison of demand for component architecture skills

These technologies enable the component revolution, which allows companies to assemble frameworks of components into working solutions. Most information technology shops have the option to buy commercial off-the-shelf components on the basis of what functionality they provide, not on the basis of what distributed object technology they were built with. They have this option because there are enough tools to form a bridge between any two technologies at least half a dozen ways.

The component revolution is based on the following precepts:

- Whenever possible and feasible, buy before you build.
- Don't reinvent the wheel. Be distributed object/component-architecture agnostic and buy the components that best fit your organization's objectives.

Following these precepts accomplishes the following:

- It allows IT shops to embrace and extend frameworks. Instead of focusing on the mundane, they can focus on the IT tools that will give their organization a competitive edge.
- It saves support and development costs.
- It invests money in the component industry, which will grow and prosper, thus pushing more application features into the mundane space and allowing more innovation and creation of cutting-edge IT tools.

DCOM Architecture

All distributed object architecture must provide the following basic features:

- *Interface definition/negotiation* is important to distributed systems. It allows distributed objects the opportunity to communicate and evolve separately without breaking the existing contract.
- *Directory services* provide a means of finding, activating and connecting to remote objects.
- *Marshaling* is a way to make the object appear to be in a local process, yet communicate the invocation of methods along with their parameters over process and machine boundaries. It allows access to interfaces from remote sites and moves data to and from the client and server process. It's just a means of formatting data between clients and components so they can communicate clearly at the bit and byte level.
- *Object persistence* is saving an object state to a persistent storage, such as a flat file or database. It's also how to connect to a unique instance of an object, e.g., when the object is already running in another process.
- *Security* is needed to protect access to components at various levels.

Interfaces

Defining an interface between a client and a component is like defining a contract between two people. The interface exposes a collection of methods that define what behavior and functionality the component will provide.

In DCOM you don't deal with objects directly. Instead, you deal with interfaces to the objects. A DCOM interface is a collection of methods that define a service contract. Actually, what you get is an interface pointer that points to a vtable (a vtable is a collection of pointers to methods). Java doesn't support interface pointers – or any pointers, for that matter. However, Microsoft allows Java developers to access COM objects in a natural way. The interface defines the behavior of an object independent of any one implementation of an object. DCOM is a binary interoperability agreement for how clients interact with interfaces via pointers and local and remote proxies; proxies act as surrogate objects that are involved in marshaling the parameters to and from the components.

The Microsoft JVM is a precursor to COM+. How DCOM is handled in Java is a precursor to the way DCOM will be handled in other languages with the introduction of COM+. COM+ will make DCOM programming a lot easier. Let's compare getting a pointer to an interface in Java to doing the same thing in C++:

```
IHelloDCOM pHelloDCOM;
CoCreateInstance (CLSID_HelloDCOM, NULL,
CLSCTX_INPROC_SERVER,
IID_HelloDCOM,
(void **) &pHelloDCOM);
```

Here's the equivalent Java code:

```
IHelloDCOM helloDCOM = (IHelloDCOM) new
HelloDCOM();
```

As you know, there are no pointers in Java. So instead of dealing with pointers, the JVM handles all the low-level complexity. It also allows you to cast an interface to an object instead of using the IUnknown interface negotiation, which in many cases makes programming COM in Java much easier than doing it in C++. Actually, Java's multiple-interfaces inheritance model maps nicely to working with IUnknown.

DCOM provides standard interfaces for dealing with objects. One such interface is IUnknown. Every DCOM object must support IUnknown. Also, Java classes, via the JVM, support a lot of other standard interfaces. So what's a COM object? A COM object is a component that supports one or more interfaces; a COM interface refers to a collection of related methods.

There are standard interfaces and there are user-defined interfaces. COM objects are accessed only through interfaces. A COM class implements one or more interfaces, and COM objects are runtime instantiations of COM classes.

IDispatch is a standard interface that all COM objects that support automation must have. Java classes in the JVM, by default, support automation via the IDispatch interface.

COM works with many computer programming languages. However, there's a special language for describing interfaces called the Interface Definition Language (IDL).

A DCOM stub equates to a CORBA or RMI skeleton. A DCOM proxy equates to an RMI or CORBA stub. A stub in CORBA-speak is the client; a stub in DCOM-speak is the server.

DCOM's IDL, unlike CORBA's, doesn't support inheritance, which is a key ingredient to object-oriented design. Instead, DCOM supports containment, delegation and aggregation. It also uses interface negotiation (IUnknown), which provides the key feature of inheritance, that is, polymorphism. Thus DCOM can support many interfaces.

The good news is that you don't have to know COM IDL to do Java DCOM programming. One of the keys to COM's success is ease of use. Java DCOM isn't tied to IDL the way CORBA is. In fact, there's nothing special about IDL. It's just a C-like language for creating proxies and stubs. Even if you use the Microsoft Java SDK with no fancy IDE, you don't have to write any IDL. And with the release of COM+, COM IDL, like Latin, may be a dead language in a few years.

COM uses the registry and the COM library to perform an object lookup. When a COM client tries to create a COM object, the COM libraries look up the associated COM class implementation in the registry. (This is somewhat analogous to the way RMI uses the RMI Registry or CORBA uses COSNaming.) The COM class implementation is executable code called the server. The executable code

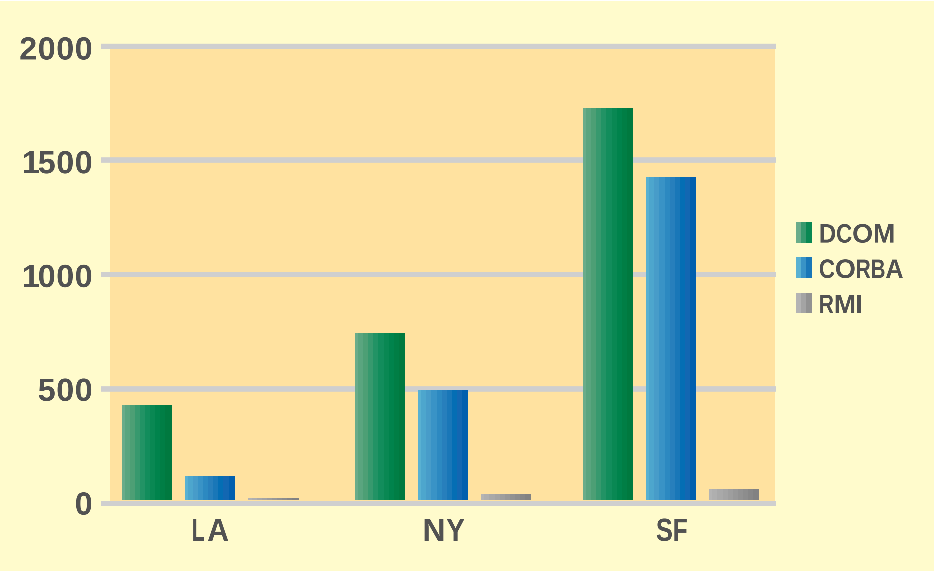


Figure 1: The demand for DCOM skills is highest in every job market.

that the COM class is associated with could be a dynamic link library, an executable file or a Java class. The COM libraries load the COM server and work with the server to create the object (the instance of the COM class) and then return an interface pointer to the COM client. With DCOM, the COM libraries are updated to create COM objects on remote machines.

To create remote objects, the COM libraries read the network name of the remote server machine from the registry to create remote COM objects. Alternatively, the name can be passed to the COM libraries' CoCreateInstanceEx function call. We'll cover a code example that uses this call with the name of the server passed as a parameter.

For remote components (i.e., DCOM components) the COM libraries use the service control manager (SCM, pronounced "scum") to perform object activation. In this scenario, when a COM client attempts to create a COM component, the COM library looks up the COM object in the Windows NT Registry as usual. What it finds in the registry is information on how to instantiate the COM object just as before. However, if the COM class configuration in the registry specifies a remote server, the COM library will collaborate with SCM. SCM's job is to contact the SCM on the remote server. The remote SCM then works with the COM library on the remote machine to instantiate the object and return an instance to the client application. Unlike CORBA, DCOM has no object ID. Instead, if you want to connect to the same unique instance of an object, you use a moniker.

With the release of Windows NT 5.0, COM adds a central store for COM classes. All activation-related information about a component can be stored in the Active Directory of the domain controller. The COM libraries will get activation information – such as the remote server name – transparently from the Active Directory. Reconfiguring the component will be a simple matter of changing the setting for the component in the Active Directory. The Active Directory then proliferates these changes to all the clients connected to the portion of the Active Directory that contains the component's information. This further closes the gaps between CORBA's activation model and DCOM's.

Interface negotiation is the ability to ask a COM object at runtime which other interfaces it supports. Because all COM objects must implement the IUnknown interface, all COM objects support interface negotiation. Thus COM clients can access any COM object and use QueryInterface to determine which interfaces the COM object supports. The ability to query the interface supported allows COM clients to decide at runtime which interface to use.

QueryInterface allows the COM object to pass an interface pointer to other COM objects that don't even have to be on the same machine. COM uses QueryInterface to aggregate many COM objects. It allows components to evolve over time and yet still be backward-compatible with older clients, while new clients are allowed to access new features through new interfaces.

This interface negotiation feature gives COM architectural appeal. COM objects describe their features at a high level of abstraction. This permits COM clients the ability to query the COM object to see whether it supports a particular interface (a feature set). Compare this to a CORBA object's single interface model. The ability of a COM client to request the feature set of a COM object allows for the flexibility you'd expect from a component object model. In other words, COM objects should be allowed to mature and develop new features without breaking old clients, yet allow new clients access to those features.

Directory Services

RMI is currently lacking a solid default directory service. However, third-party tools that implement Java naming and directory interface (JNDI) give RMI a robust directory service. CORBA has an advanced directory service, COSNaming, that provides a transparent location of objects depending on your CORBA vendor's COSNaming implementation. DCOM's current directory service lacks a truly distributed transparent nature like CORBA's COSNaming. This lack of support seems to be the result of different approaches to solving similar problems rather than to a missing feature or an architectural advantage.

In Windows NT 5.0, however, DCOM can be used in connection with the Active Directory. Activation-related information about a component is stored in the Active Directory of the domain controller. The COM libraries then get activation information – such as the remote server name – transparently from the Active Directory. The Active Directory will proliferate configuration changes to all the clients that are registered to receive a component's information.

Marshaling

When a client makes a method call on a COM interface, the COM objects in the other process can be down the hall or on the other side of the globe. The differences between local and remote access are abstracted from the COM clients. Marshaling involves taking an interface pointer in a server's process, making that interface pointer available to the client process and setting up interprocess communication (either RPC or LPC). Next, marshaling must take the arguments to an interface method call as passed from the client and serialize those arguments to the remote object's process.

Custom marshaling is fundamental for certain applications. COM offers standard marshaling for the built-in standard COM interfaces. With standard marshaling COM furnishes a generic proxy and stub that communicate through standard RPC for each standard COM interface. Custom marshaling is not a trivial matter with Java and DCOM.

By default, all objects are passed by reference, which means that when the client calls a method of a remote interface, the call is marshaled over the wire. If you want your objects to be passed by value, you need to do custom marshaling.

You probably won't ever need to write your own custom marshaler because DCOM/Java integration centers around IDispatch. IDispatch is a built-in interface, and COM provides a marshaler for it. In addition, Microsoft provides a special optimized marshaler for Java COM objects.

By comparison, RMI provides good support for marshaling in both ease of use and the overall feature set. With RMI, if an object defines a remote interface it's passed by reference. However, RMI can pass objects by value.

Imagine defining a remote hashtable type of class that contains results to a query. Every time your client accesses the remote

hashtable object the call goes over the wire, which can really slow things down because of the latency of the network. RMI gives you another option. If you pass a parameter to a remote method and the parameter (1) doesn't implement a remote interface and (2) is an instantiation of a class that implements Serializable, then the parameter will be marshaled over the network. If the code for the parameter isn't available on the client machine, RMI will load the class from the remote machine. Not only are the values moved across the network, but the code that accesses those values is moved across the network as well. In essence, you've moved code and data so that the object has been relocated to the client's process.

RMI has an architectural advantage with reference to marshaling. Neither CORBA nor DCOM approaches this technique of moving the code from one JVM to another, but both allow you to pass by value. By default, DCOM, like CORBA, uses pass by reference, whereas RMI allows both pass by reference and pass by value. In addition, RMI allows you to pass code.

Future versions of CORBA will have support for pass by value. It's possible to create your own pass-by-value support with DCOM, but it isn't as straightforward as the RMI approach. To perform pass by value in DCOM, you need to define your own custom vtable interface and write your own custom marshaler for the custom vtable, which involves using C programming and Raw Native Interface (RNI). There are ways around the DCOM marshaling issue. For example, you could pack all class data in a string and then write your own unpacker, but it isn't an elegant solution.

Persistence

CORBA has a fairly straightforward persistence mechanism – which Java and DCOM don't seem to have – for reconnecting to unique instances of an object. DCOM does provide a flexible way to manage persistence, yet it's not as implicit as the CORBA technique, so it's more complex to implement.

As mentioned earlier, CORBA provides an objectId (called an object reference) to connect to specific instances of an object. Conversely, COM objects, by default, are stateless objects. COM objects don't have object identifiers. Instead they use monikers to connect to a particular instance.

COM's instance-naming mechanism is extremely flexible but at the price of complexity. An IMoniker specifies an instance for COM. Monikers – also referred to as instance names – for COM objects are themselves COM objects. This explains their flexibility – and their complexity (compared to the CORBA approach). The standard COM interface for these naming objects is IMoniker.

If the COM object the moniker is referring to isn't already running in a server process, IMoniker can create and initialize a COM object instance to the state it had before. On the other hand, if the COM object that IMoniker is referring to is running in an existing COM server process, IMoniker can connect to the running instance of the COM object via the COM server.

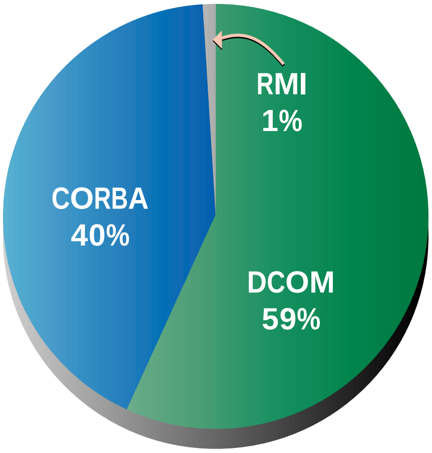


Figure 2: The demand for DCOM skills is more than CORBA and RMI combined.

Security and Administration

As far as security goes, DCOM has some clear architectural advantages with its tight integration with the NT security model. This gives DCOM an edge in administration and ease of development. The same or similar tools that are included with the OS can manage DCOM security. In other words, if you know how to administer Windows NT, you can easily learn to administer DCOM.

Interoperability and Bridging

It seems RMI is moving closer to interoperating with CORBA – a big plus for RMI and CORBA. Of course, RMI interoperating with CORBA will degrade some of its functionality (you'd have to give up its most innovative feature: its ability to transfer code and data in a pass-by-value call).

There's already a lot of bridging technologies from one distributed object architecture to another. For example, IONA has a CORBA/COM bridge that takes a CORBA object and makes it appear as an ActiveX control, which can then be embedded easily in a Visual Basic program (or a Visual J++ or Delphi program, for that matter). Here's another example: the forthcoming CORBBeans implementation will allow CORBA distributed objects to look like JavaBeans on the client. In effect, this gives CORBA a local component model and will make CORBA "toolable" on the client. Making CORBA toolable makes it easier to use in applications like Visual Basic by using Sun's ActiveX bridge to bridge the CORBA bean to look like an ActiveX control.

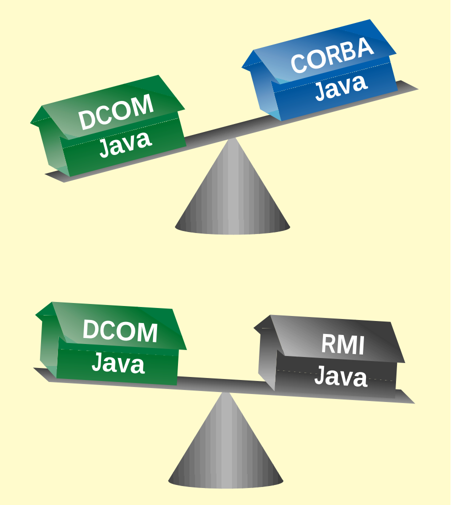


Figure 3: How DCOM compares in ease of use

Comparing DCOM to RMI and CORBA

I don't think it's fair to advocate any one distributed object framework (DCOM, RMI or CORBA) over another; each one has advantages that give it an edge for certain types of applications. Also, using one distributed object framework doesn't preclude using another.

Ease of Development and IDL

Java's transparent DCOM support (in JVM) clearly gives it an architectural advantage: namely, you don't have to learn another language to create DCOM/Java components. Conversely, when you develop a CORBA component you typically start by creating an IDL file and then deriving your client and server from another class. It should be noted, however, that there are tools – such as Inprise's Caffeine – that help reduce CORBA complexity by allowing you to define your interfaces in Java.

Typically, you don't need IDL to create Java DCOM components. But there are times when you do need to create IDL files; for example, when you want to provide custom marshaling or create vtable components. Concerning comparisons of the IDL languages (Microsoft's IDL to CORBA's IDL), it's been stated that CORBA's IDL seems more thought out and easier to use. CORBA may have a cleaner IDL syntax because it doesn't extend an existing IDL as Microsoft extends RPC IDL for DCOM (see Figure 3).

Conversely, Java's RMI has no IDL; it doesn't need one because it provides only Java-to-Java communication. You define your remote interfaces in Java, then create an implementation class in Java that implements the remote interface you defined. Although it doesn't have an IDL to deal with, as CORBA does, the inheritance model of defining a remote object is a bit more complicated than the DCOM approach. RMI is a bit less complicated than the CORBA approach (unless you use something like Inprise's Caffeine). Again, I've seen demonstrations of IDEs that make RMI development fairly trivial.

Various companies seem to be working hard to make CORBA and RMI development easier, so any advantage DCOM has in ease of development may be short-lived.

Using a Model That Works

Splitting hairs over architecture issues

BEST NEW MAGAZINE OF THE YEAR
JBuilder Developer's Journal
JBuilder by Borland
Features Both Ease of Use and Power

may be the wrong way to pick a distributed object framework. Instead, the component model you use may depend heavily on the talent pool at your company. If you have a department full of Visual Basic programmers, you should consider using mostly DCOM, and RMI and CORBA if you have to connect to third-party components and frameworks. Conversely, if you use Java a lot on both the middle tier and the client, you might consider RMI, and use COM only when you want to capitalize on a huge install base of applications that have ActiveX Automation support. CORBA is the obvious choice if you need to connect to a lot of legacy applications that support it. Since COM custom marshaling is nontrivial and it's easy to pass objects by value with RMI, use RMI if you want to move a lot of objects around the network.

Conclusion

DCOM is an excellent tool for creating distributed applications as well as for enabling the next revolution in history: the component revolution. Using the Microsoft Java SDK, you can easily write both DCOM clients and servers, and you can integrate with existing applications and in-house components developed by Visual Basic, Delphi and Visual C++ developers. You can still use CORBA, DCOM and RMI from the JVM, so you don't have to select just one distributed object technology.

In this article we covered:

- How DCOM compares to RMI and CORBA
- Why DCOM may be important to you
- What DCOM architecture looks like
- How to use the Microsoft Java SDK to create DCOM objects

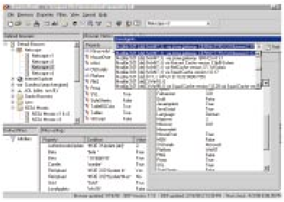
In Part 2 we'll cover using DCOM from the Microsoft JVM with hands-on examples, and details on just how easy it is to create COM/DCOM servers in Java.

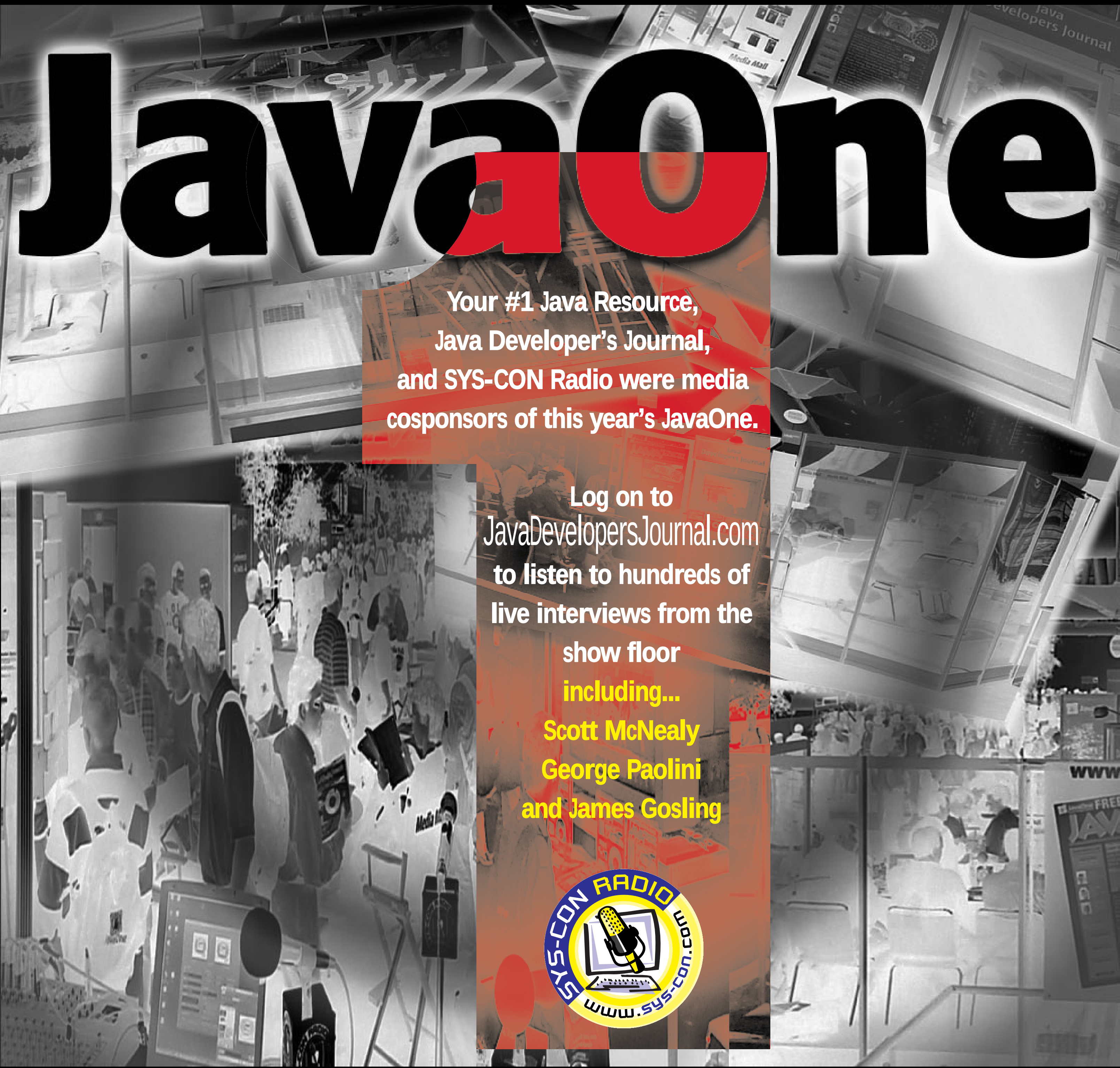
In the book *Java Distributed Objects* by Bill McCarty and Luke Cassady-Dorion (Sam Publishing), the subject of DCOM is covered in more depth. The book also covers RMI and CORBA in detail (with an emphasis on CORBA). I wrote Chapter 20 on DCOM, which covers Java and DCOM in more detail and relates how to create callbacks in DCOM and how to use JActiveX to create Java wrappers around existing COM components. I also have an example that uses late bound calls using IDispatch.

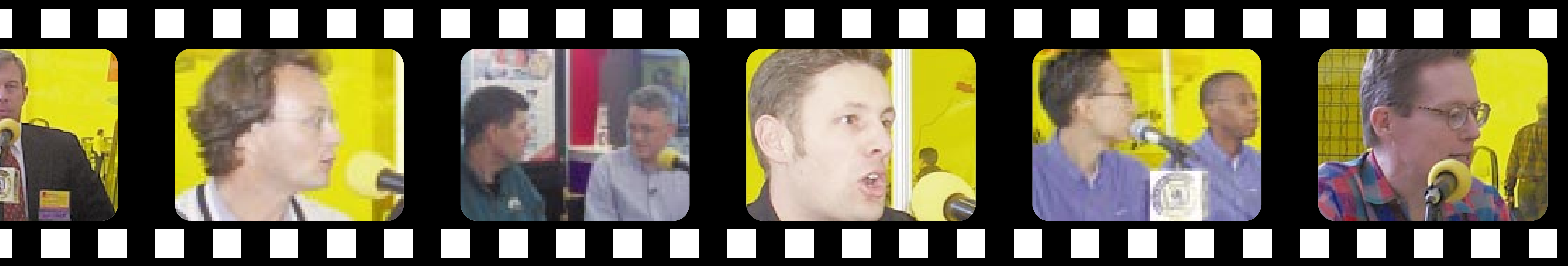
About the Author

Rick Hightower, a senior software engineer at LookSmart, a category based Web directory, has been writing software for a decade, from embedded systems to factory automation solutions. Rick recently worked at Intel's Enterprise Architecture Lab, where he researched emerging middleware and component technologies. Rick can be reached at Rick_M_Hightower@hotmail.com.

Rick_M_Hightower@hotmail.com.







EXPOSED

JavaOne '99, opened by keynote speaker John Gage, was the largest developer conference yet, with 20,000 in attendance and more than 800 speakers. It proved to be a great four days. Dr. Alan Baratz drew prolonged applause when he reported that the Java community now exceeds 1.7 million. Baratz continued with a feel-good, stat-laden speech, announcing three new editions of the Java platform that collates all relevant APIs in one package. This move will "simplify the Java deployment" of applications. Netscape will now ship with Java 2 SE. The following pages excerpt *JDJ*'s exclusive SYS-CON Radio coverage from JavaOne.

SYS-CON Radio Interview
SCOTT McNEALY

***JDJ:* Thanks for taking some time. Now, this conference has been going on for a couple of days...**

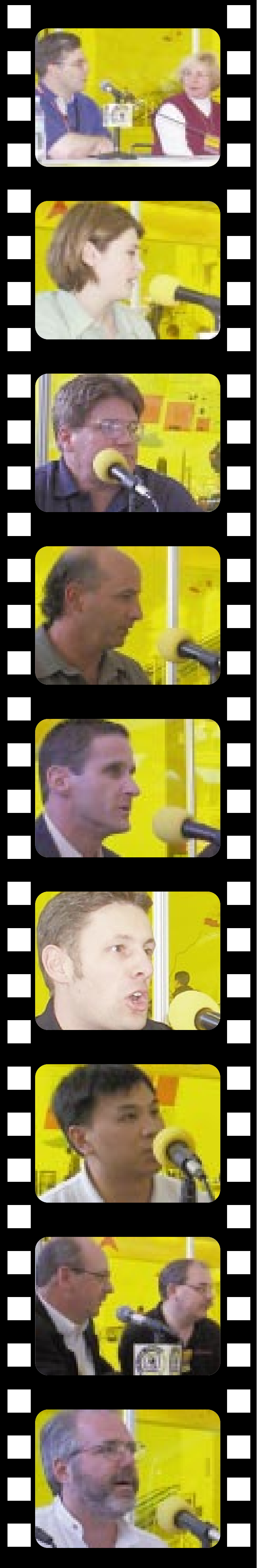
McNealy: The real opportunity here is to keep the momentum going by having regular Java-oriented events. Get small working groups where you can sit down and work through the issues, understand the opportunity and get the customers next time. It would be just great to get every customer here. We basically rented San Francisco, and it's too small. I think it's a good opportunity, though, to get other customers involved, get your customers to meet in smaller half-day or day events in your area. Get some people from the JavaSoft group to come out and support it or whatever, but get the Java evangelists in your customer base in touch with those that aren't there and let them help you do the cross selling. It's fantastic how our customers are selling our customers here.

***JDJ:* Java – in the short term – sometimes people tend to overestimate the impact. For example – the Web – four or five years ago everybody thought it was there, it was going to be hot. It has taken four or five years until we've seen the e-Bays and the Amazons really come into fruition. What do you think about Java and JINI long term?**

McNealy: You have to get your customers to help aim their technologies to where Java will be, to where market acceptance, the Web, network performance, the tools and the functionality will be. And that requires taking a little leap of faith. I think a lot of companies out there are finding out that they didn't adopt the Web, or they wrote it in Windows and are waiting for Windows 2000, and now they've got viruses; the product isn't there; it's going to be buggy; it's going to be slow and they can't deliver their apps out over the network; and they're thinking why didn't I do it in Java, why didn't I start two years ago in Java. I think the biggest problem we have is underhyping, not overhyping, this technology.

***JDJ:* But Sun is changing the rules of the game with Java. Is there a message you have from the field on how are we actually going to make money on it?**

McNealy: Are we making money? We grew at 24% last quarter. That's not bad. In fact, that was faster than Intel, IBM, Compaq, SCI and everybody else out there. That's pretty phenomenal growth. And by the way, we're operating at unprecedented levels of contribution. Our pretax margins are up significantly over 15%. We're making good money. Something is working right, and I don't want to mess with it. The way to make money is not by making money on Java. You don't make money on English. You don't make money owning English. You make money doing things in English. And the way we will make money is doing things in Java, not necessarily on Java itself. The chips, the operating systems, the computers, the tools that run Java and training for Java, all of those kinds of things: that's how we're going to make money.



JDJ: There are a lot of neat gizmos and gadgets on the floor of the show here. How many Java-enabled devices have you got in your life? How many do you own?

McNealy: Basically every desktop I go to is a Java browser. As far as I'm concerned, the most exciting Java-enabled device is a full-blown Java browser that lets me get at Sun.net because that's my desktop. In fact, now I go in and I use CD. I would rather use Sun.net. CD is total overkill for me. To me, the browser is the greatest Java-enabled device. Now, I don't usually need the other nomadic devices. All I want is the phone. I'm experimenting right now with the Motorola Page Writer. There are a lot of people running around here actually writing software down in this new Palm V with Java. I think that's going to be an interesting platform. It might be my next Java-enabling device.

JDJ: One of the SEs asked me to ask you a question from this morning. Apparently IBM released VisualAge on Linux; they tested but haven't yet released it on Solaris. Are there any comments about that? Are we going to see VisualAge on Solaris?

McNealy: I think they should get it on Solaris, and I think that's going to be something that will be rectified quickly.

JDJ: Okay. So if we're outgrowing San Francisco, where is JavaOne going to be held next year? Any ideas?

McNealy: I don't know where we could go. It's too hot down in Las Vegas. Hmm, what would be a good city? I don't know, let's go do it in Palm Springs.

To listen to the unedited version of the Scott McNealy interview, log on to JavaDevelopersJournal.com

SYS-CON Radio Interview

GEORGE PAOLINI

JDJ: We're back on SYS-CON Radio. Joining us right now is the vice president of marketing for JavaSoft, George Paolini. George, would you tell us a little bit about your editorial column in this month's Java Developer's Journal?

Paolini: Sure. I was invited to participate in the latest issue of *JDJ* as a sort of celebration of the upcoming JavaOne. The piece that I wrote was really just to step back for a minute and take a look at where we are with the technology and what's happening. I think this event here today is back in real life. As you can see out here, we've got some amazing momentum, 20,000+ folks coming through trying to learn the latest and greatest on the technology. I think it says a lot about this technology and its place in what's happening in network computing.

JDJ: George, you are marketing for Sun, which means that you're in charge of communicating Sun's message to the developer. What is that message you want the developer to go home with after JavaOne?

Paolini: I think the message that I would like to leave in the heads of the developers is the same message I would leave in the heads of any consumer or customer of any product or technology. When you're purchasing that technology or that product or investing in it (purchasing is one way you invest because you invest with your money, but you also invest with your time and your efforts -- I guess it would be a bit of advice), make sure that investment has a return, which is in the form of allowing you some choice, choice in how you use it and freedom to continue to be able to purchase products that innovate and are innovative as opposed to products that lock you into a particular platform. I won't mention names, but you can guess what I'm talking about here. But the point here is I think the computer industry has a lot to learn about this from the consumer electronics industry. We learned some hard lessons. When you invest in technology, you want to be sure that you get the best return on that investment. And the computer industry hasn't exactly been architected to make that happen, and I think it's changing now. I think with what's happening with the Web and giving people the ability to make choices, it's actually driving more innovation, and innovation is what gives customers choice.

JDJ [Alan Williamson]: One of the things that I as a columnist always come across is people still ask me, "Where can I buy Java?" Java is still perceived as a product, which I think maybe was symptomatic of where Sun possibly marketed at the beginning. So how do we address that now?

Paolini: Let me try to explain for the less technical members of the audience exactly how this works. And what we have done is a model that we didn't invent but I think we maybe improved upon it, certainly expanded upon it, and it's the model of allowing some freedom in the use of the technology. In this case, it's a binary version of what we call a runtime or a piece of software that will execute a program or an application, and that's free. It's free for developers, it's free for end users, and in this case usually for corporate customers to use at runtime without charge to Sun. Where we do charge is for the use of the source code. We allow you to actually download the source code and use it, but at the point when you take it, innovate upon it and put it into a product, then we start charging for it. This is a model that has evolved. We now call it our community-source licensing program, and we think it's pretty innovative. So the folks that actually are paying licensee fees to Sun are companies like IBM, Novell and Oracle, many of the companies that you see here today. But for the developer who is taking that runtime and putting it in their application, there is no charge. So there's a technology and it's a product in the sense that we do license it, but we don't license or charge for the binary version.

JDJ: You still talk of Java as a product and not as a language. We had James Gosling here previously, and his dream was – I don't want to know it's Java, I just want the solution.

Paolini: Right. That's absolutely true. And I guess another cue that you could take from the consumer electronics industry is exactly that. When this technology is really truly successful is when you don't even know it's there, and that is really the dream. But I think we're a ways off from that happening, but it's certainly what you see on the show floor here this year, and certainly what we've been talking about a lot. I have one of these devices in front of me here; it's the new Palm V that has the most recent version - the Java platform micro-addition on it. What that says is this is making the technology usable in many ways that weren't even thought popular just a couple years ago by most people. Certainly James had the vision many years ago, and it's now coming, I think, to fruition.

JDJ: It's a shame that the Palm Pilot wasn't like the Java ring last year, which you were giving away, but now you're making us pay for it.

Paolini: We're making you pay because it's a considerable amount of investment. We are charging the exact cost for this device plus the tax. But the cost was considerable, and so we're only charging what the device itself costs.

JDJ: The Palm Pilot is the first example of a consumer device for the masses as it were. Where are we likely to see other examples popping up in everyday life with Java?

Paolini: I think you're going to see examples in lots of devices that include things like Web phones, screen phones and other PDA-type devices. I think what makes the Palm the real hit this year is this is the device that many of us have come to know and understand. And if you really look at the device and the people that use it, what you'll recognize is that Palm and 3Com really broke the mold on how to build a device that is really user-friendly. Everyone else tried to figure out a way to shrink the computer desktop, and I mean the computer desktop in functionality, and the user interface down to a palm device, and what we have all learned is that is not how people use devices.

JDJ: You are talking of Windows CE.

Paolini: Well, I'm not only talking about Windowa CE, quite honestly. I think many other companies tried a lot of different models, but the point is that what Palm did was they actually looked at many operating systems and tried to figure out which of those might be most appropriate for this device when they went out and built the Palm OS. And what they did, if you look at the history, is they scrapped it all and went back and did some hard studies on how people actually interact with devices. And they looked at

Paolini on JDJ...

‘Hats off to you. You are obviously doing something right by supplying the information in appropriate content that these people really want’

what people really wanted. Well, what they wanted was a calendar, a very simple e-mail tool, a very simple organizer and they wanted it in a way that was acceptable through basically point and click; they built it based on that. I think the message with Java on this device is it's the marriage of, I think, two very user-friendly technologies. I think you'll see Java technology in many other types of consumer devices, some of which are hard to even really try to envision today. You know, the average automobile today has 40 microprocessors. Most of those microprocessors don't talk to one another. The average home has somewhere around 40 microprocessors. Any microprocessor with network capability is a good candidate to have some version of a Java platform actually driving the instructions.

***JDJ:* What do you see from a marketing standpoint as the reason for the success of Java and also what do you see as the difficulties in marketing something like Java?**

Paolini: The easy part is that for the software programmers what we hear back time and time again is that, number one, it improves productivity. And what we hear is that it makes it easier to write applications, and obviously the ability to run those applications on multiple platforms, or what we sometimes refer to as the switching cost of writing an application and running it on various platforms. That's the easy part. The hard part is that for the general consumer it's hard to really understand and appreciate where and what this technology does. Kind of an amusing anecdote, but I think one that illustrates what is really still a very technical piece of work, is that a friend's mother went off and bought a PC (she's 73 years old), and he had been doing some research for me on what is going on with Java. So he thought he would just do a little test sample here, and he said, “So, Mom, you are going out to buy a PC. What do you think of Java?“ She said, “Well, I don't really know much about it, but I have to have it on my PC.“ And he said, “Well, why?“ She said, “I don't know. I just know I have to have it.“ That is kind of a good problem to have. But obviously there is some work to do in articulating what the value is of this technology to the average consumer. So I would say that would be the challenge.

***JDJ:* Our last question for SYS-CON Radio is, I know you write for us, how do you like Java Developer's Journal? What do you think JDJ is doing for the Java industry?**

Paolini: Well, rather than ask my opinion I guess let's just look at the numbers. I think what I've heard is that you're the number one Java-specific publication in terms of your subscriber base. I obviously feel good about that because that says to me that there is really a demand out there for more information about the technology. Hats off to you. You are obviously doing something right by supplying the information in appropriate content that these people really want. So good luck with that.

SYS-CON Radio Interview JAMES GOSLING

***JDJ:* We're very proud to have James Gosling with us here today. James, how do you think this show is going?**

Gosling: A lot of people who have been working on it are now going around in an advanced state of sleep deprivation, really looking forward to crashing for about the next week or two. But it's pretty exciting. A lot of this stuff is getting very real. I saw the Palm Page Writer download demo, and that just about knocked my socks off.

***JDJ:* I listened to your keynote speech. You mentioned that you spent most of last year behind lawyers. Has this frustrated you?**

Gosling: Yes, it has been pretty frustrating. It definitely does feel like sticking your head into a black hole. It has been educational; it has been amusing; it has been sickening. I mean, having to read all that e-mail and getting some of that truth out. I don't think I wanted to know that.

***JDJ:* Have you not been developing anymore? Do you mean pure administration?**

Gosling: Well, no. I do as much development as I can. I have this other problem of having really severe carpal tunnel problems. So, even if I weren't stuck in a room with the lawyers, I kind of have a quota of about a couple hours a day.

***JDJ:* What areas of Java are you actually working on yourself? Are you part of the core of Java? What is your baby at the moment?**

Gosling: Nothing is really my baby at the moment. Rather, I sort of get involved in various bits and pieces of things. People come and ask me for advice. There is kind of a list of language change proposals, and I'm one of the people that sort of argues about those and in great depth. I'm on the real-time working group, so I pull a fair amount of time on that.

***JDJ:* We have our first question out here from the floor. Your name, sir?**

***Ed Roberts:* Ed Roberts. Basically my question is: What are you looking at in terms of templates, what were your design tradeoffs and are you specifically involved in that?**

Gosling: I've personally tried to stay out of that particular food fight. I had been looking into this stuff for quite a long time, and there are many different flavors of template-like things and many people who consider themselves experts and no two experts agreed five or six years ago. We started this debate that has largely been run on this mailing list run by Gilad Brocka, and there has been lots of interesting and gory debates over the last four years about what's right, what's wrong and what are the correct things to do. It feels like things are converging, and there are really three proposals right now that are sort of the frontrunners. One thing that has been hard to get over is to come up with something that gives you polymorphic behavior but is also fairly simple and comprehensible. One problem with the whole area is most of the solutions tend to create all kinds of complexity that nobody can actually understand. Also, there have been problems with things like the template mechanism in C++. It's sort of an invitation to have a huge amount of code blows; struggling with ways to deal with that and come up with a proposal has taken a fair while. The frontrunner proposal is something called GJ, Generic Java, that was done largely by a guy at the University of Southern Australia (his name escapes me right now), in conjunction with Phil Wadler who was originally at the University of Glasgow and is now at Lucent and a guy named Dave Stoutimier and Gilad Brocka. One of the interesting things about that particular proposal is that it involves no changes at all to the VM specification, and it has a really nice migration story from existing container classes into sort of parameterized container classes, because the sort of parameterized versions of classes and the nonparameterized versions are compatible in an elegant kind of way. Then there is sort of an extension to that. The number two proposal is basically the number one proposal with some facilities for doing runtime introspection. The number one proposal, one of the things it gives out for not having any VM changes is there are a few things you can't do, and really the only important one is that there are some sort of introspection things

you can't do. Then there is this other one that came from MIT. Its implementation is yet more complicated, but it allows parameterization over primitive types as well. And so the debate that is going on right now is sort of at what level, what sort of point in the spectrum is most appropriate to the community. That one is actually being run through the Java community process. I don't know what the state is exactly with it, but if you go to the Java.Sun.com Web site, you'll find it. You'll find the descriptions of all of these things in incredibly gory detail. It is all out there.

SYS-CON Radio Listener: *I'm from Oracle. And my question is: When you designed Java, Java was a programming language. It had a virtual machine specification and a language specification. Now anything and everything that's written in Java, whether you write libraries or something, has become part of the Java phenomenon. When someone writes, say, compile it in Java, it's just another programming language, but somehow people have taken the concept of Java and anything that has to do with the Web or the programming language. Any program that I write in Java, I put J in front of the name, and I say it's a J-Compiler for C++. Now, no one ever said that C stands in the way because Java compilers are done in C for instance, but everybody seems to just take Java, and if you touch your code with Java, it just becomes a Java phenomenon. What do you think about taking the concept of Java just a bit too far, especially the media types? Do you think Java is the solution to everything?*

Gosling on JDJ...

‘Just the advertising is fun, you know, what it is that people are doing out there in the world. It's always a good thing to look at’

Gosling: It's really, really bizarre. In some sense my sort of fantasy for how Java should evolve is that nobody should be aware that it exists, and that people build devices...I mean, it's sort of part of the JINI story that things should just work. You shouldn't worry about, oh, is it a Java this? What kind of a boot disk do I need to make this thing work? You don't do that; you just do it, you just get on with your life.

SYS-CON Radio Listener: *Hi, I'm Thomas. I'm curious about the thought process behind the fact that an interface isn't allowed to have static methods. Because inherently there isn't anything wrong with doing that, and occasionally it's really nice to put a factory method in an interface that returns instances of that interface rather than to put it off in another real class for that sort of purpose. I was wondering what thoughts were behind that.*

Gosling: I think it's now seven years after the fact on that particular decision. I'm not sure I can exactly justify interfaces very pure, that they are just a description of an interface, that they sort of exist almost independent of behavior in space and time and the rest of that, they just describe the shape of the socket that you plug something into. And it was just trying to keep things conceptually clean. And so if you wanted to do something that had behavior, you go to a class. As time passed, it's not clear if that distinction was actually as important as I felt it was at the time. I mean, I would be half tempted to actually get rid of interfaces completely and just do something based on class. But it turns out that there are some other interesting reasons for doing interfaces that have to do with the way that method calls are implemented, in that sort of under the sheets there are two forms of dynamic dispatch, one that is very fast and one that is almost as fast. And when you use an interface, you get the not quite so fast but much more dynamic dispatch mechanism. And when you just call a class, you get the three instructions, bang! You are there. It's fast as the procedure call version. So I sort of kept the schism alive, but boy oh boy, it was a delicate one. It still feels right, but it was one of those ones that was kind of on the edge and it was done more for purity than anything else.

JDJ: *Why is there only one Java Network Station in the whole building this time, where last year it was everywhere? Sun was hailing the Java net station as the next-generation solution...*

Gosling: You know, you ought to ask Scott McNealy that one.

JDJ: *How do you feel about that?*
Gosling: Ask Scott that one.

JDJ: *Because we're not on the Palm talk. We're going to this beautiful-looking machine down to this palm-sized machine now.*
Gosling: Well, in some sense that is what the Network Station ought to be. I mean, the big shark fin thing weighed several pounds, and in some sense it was more a conceptual demo than a real device. I have always thought that things like the Palm were a much more real device for that kind of thing, much more compelling. Because if you're in a foreign factor that is like the net station, that's big enough, bulky enough and expensive enough; it might as well be a big box.

JDJ: *So you're more than happy to see Java being used where it was really intended to be, which was consumer hand devices as opposed to running back-end servers, etc..*
Gosling: Oh, running back-end servers works out really well too. Some of the place is sort of in the middle. Whether they make sense or whether they don't make sense is sort of a matter of design trade-offs that are dependent on the particular moment in time. You know, how much the CPUs cost, how much does D-RAM cost. You get one answer one day, the next answer the next day, and it changes your design.

JDJ: *Next year, JavaOne 2000, what will we be talking about?*
Gosling: You know if I knew the answer to that one...

JDJ: *You would be even richer now.*
Gosling: Well, I would actually be rich. I would not be a guy with a mortgage. Let's see. What is most likely to be real?

JDJ: *Are you seeing XML as a big part in the new wave?*
Gosling: Yeah. XML has been a real part of it. I think a lot of people get confused about what XML is. XML as a data interchange format, which is what it is, works great. There is a lot of stuff in Java to deal with that. You will probably see a lot of XML-based things next year. I would like to see a bunch of real-time-based things. I would like to see some serious supercomputer stuff next year. There will probably be a lot more than just the Palm that runs KJava. The thing I'm really intrigued about is so there were 10, 15-odd thousand Palm VIIs delivered at this conference, the SDK for them is available, what are you guys going to do? I mean, it has taken the Palm Pilot from being something that you can write notes on to something that you can go party with. For me, the exciting thing about what goes on in this whole business is that it's always a surprise, it's sort of not us, it's you. And the things that you guys do always surprise me. If you asked me that question last year, I would never have predicted that the guy with the Lego Mind Storm Robot playing laser tag -- I mean, that was just incredibly goofy and a lot of fun.

JDJ: *My last question would be what do you think JDJ itself is doing for the Java industry? What do you think of the magazine?*
Gosling: It's getting information out to people, getting people talking, all of these mechanisms that sort of get people together, that get the technology out and get the interesting things in. Just the advertising is fun, you know, what it is that people are doing out there in the world. It's always a good thing to look at.

Benchmarking with

How to build a utility that can be used with ease and flexibility

by Steven Feuerstein

I've spent over a decade working with Oracle technology to develop and deploy applications. In the process I've developed an area of expertise: the Oracle PL/SQL language.

PL/SQL is a flexible, powerful procedural database programming language. There is no doubt that if you want to interact with the Oracle database, PL/SQL is the way to go. It has even adopted some object-oriented capabilities in Oracle8. With Oracle's decision to integrate a Java Virtual Machine directly into its database, however, it's incumbent upon all Oracle developers to learn the Java language. Most important, we need to figure out how to use both Java and PL/SQL within the Oracle server technology to get the best of both worlds.

I'm busy learning Java and am both delighted and dismayed by the features of this most modern object-oriented language. I recently spent some time wrestling with and figuring out how to use abstract classes. This article shares the lessons I learned; my hope is that developers new to Java will be able to understand more easily this important but somewhat obscure facet of Java.

Analyzing Program Performance

I built an enormous body of generic, reusable code for developers in the PL/SQL environment; I even bundled it all together into a library called PL/Vision (available from RevealNet, www.revealnet.com). As I move into the Java world, I find myself trying to figure out which techniques and functionalities that are useful in PL/SQL carry over into Java.

One of the handiest utilities I constructed in PL/SQL was the PLVtmr package (packages in PL/SQL are similar to static classes in Java). Developers use this package's programs (PLVtmr.capture and PLVtmr.showelapsed) to calculate the elapsed time of a particular program or code segment. This timer utility proved handy because it offered a granularity of timing (down to the nearest hundredth of a second) and focus. Rather than run an elaborate analysis/trace session, I could quickly determine the performance of a single program. I could also compare different implementations of a given requirement to find the version with optimal performance.

I decided to build a mechanism in Java similar to the PLVtmr package. After a small amount of research, I discovered that the System.currentTimeMillis method returns the current time as the number of milliseconds since January 1, 1970 00:00:00. And just in case you were concerned about Y2K problems in Java, rest assured that System.currentTimeMillis will not overflow till the year 292280995.

(I joke with PL/SQL students that the authors of Java experienced a bout of "language envy." PL/SQL calculates elapsed time to the hundredth of a second, so naturally Java must go down to the thousandth of a second. In reality, of course, I imagine that Gosling and others didn't pay a whole lot of attention to Oracle PL/SQL as they created their own "Write once, run everywhere" language.)

Great! So I have a mechanism that provides the time to the nearest millisecond. How do I use that method to calculate the elapsed time of a program? By comparing consecutive calls to the method. Listing 1 shows a very simple class that demonstrates the technique by timing how long it takes to count the number of bytes in the specified file.

The first time I executed HowFast.main, 70 milliseconds elapsed. Subsequent executions revealed a steady-state elapsed time of 40 milliseconds (see Figure 1).

Well, if you hadn't previously known about System.currentTimeMillis before picking up this article, you've now learned something new about Java! You've also seen how to put this method to use in your code to calculate elapsed time. Sadly, this is as far as most developers go; they learn about a new method, then write a script each time they need to apply the technology. They might even construct a "template" class in a file as follows (also stripping the code down to its bare minimum):

```
class HowFast {
   public static void main (String[] args) {
long timeBefore = System.currentTimeMillis();

// Run the code you want to time here.

      System.out.println (
          System.currentTimeMillis()-
           timeBefore);
   }
}
```

I suggest, however, that a much more sensible, productive and elegant solution would be to construct a class that encapsulates the details of the elapsed-time computation and exposes a set of methods to get the job done. I will build such a class in this article, and then extend it to an abstract class so you can easily perform benchmarks on code that you construct.

Performance Comparison Example

Before diving into the build process, let's take a look at the code to demonstrate the usefulness of a timer mechanism. The most common application of a timer is to compare the performance of two or more different implementations of a requirement. As I mentioned before, I'm just learning Java, and started playing around with streams by manipulating files. I decided to write a program to count the number of bytes in a file and came up with two different implementations. I put both of these methods into my InFile class (it returns information about the contents "in a file"), which can be seen in Listing 2.

The InFile.numBytes method counts the number of bytes explicitly, while InFile.numBytes2 simply takes advantage of the available() method to return the total. By using available() I'm taking a bit of a risk because if it's a very large file, not all bytes will be available. For the purposes of my test, however, this will serve us just fine. I also catch any IO exceptions and return -1 to indicate a problem; that way developers can use numBytes without having to worry about exceptions popping out of the program.

Now I'd like to determine which of these implementations is the fastest – my hunch is

that numBytes2 is faster since it takes advantage of the available() method. I should also see if their behavior changes in response to differently sized files. So let's build a Timer class to help us answer these questions.

The Timer Class

The objective of the Timer class is to provide a set of methods that allows us to calculate the performance of an individual portion of code. The full text of the Timer class is displayed in Listing 2 and may be found in Timer.java; the following sections examine separate parts of the class. First, however, let's apply the class's methods to the HowFast class to see how Timer would be used:

```
class HowFast1 {
   public static void main (String[] args) {
      Timer t = new Timer();

      t.start();

      int result = InFile.numBytes
("c:\\temp\\te_employee.tps");

      t.showElapsed();
   }
}
```

The results of running HowFast1 are shown in Figure 2.

Notice how much is missing in this new implementation of a timing script?

- There's no mention of the System.currentTimeMillis method; the code used to perform the timing is hidden. This makes it easier to deploy new and better implementations of a timing mechanism as they become available.
- There are no local elements such as timeBefore and timeAfter. These longs were only needed by the timing mechanism and are now subsumed within the Timer class.
- I no longer expose how I am showing the elapsed time. I may want to rely on System.out.println today, but change it to something else later. My reliance on a distinct class, Timer, to handle these details means I can change implementations at any time without affecting the users of the class.

Of course, I can also perform multiple timings by instantiating more than one Timer object, which is useful when I want to compare different implementations. This approach is shown in Listing 3.

The output from executing this script is shown in Figure 3. As you can see, and as I suspected, the version based on available() is consistently faster. Actually, now that I mention it, how can you tell which result goes with which test? Not simply by eyeballing the output. Wouldn't it be nice to be able to include a message or context with the results so you can interpret them more easily? Let's modify the HowFast script once more and see, in Listing 4, how intelligible we can make the output.

Now, I have added a word describing each test in the call to showElapsed(). This information is then added to the output message, as shown in Figure 4.

You should now have a solid idea of how we want the Timer class to work. Let's explore the implementation.

Class Elements

Each object instantiated in the Timer class has the following data elements:

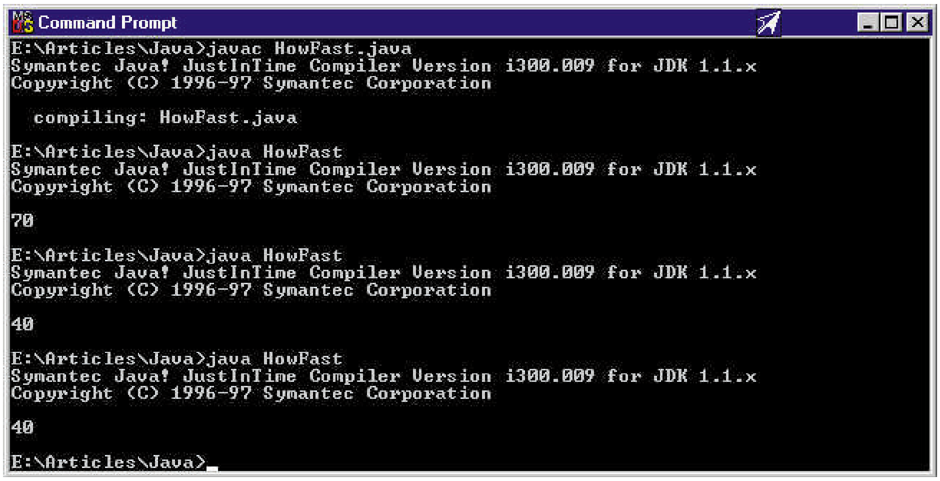
```
// Start time elements
   private long mstart = 0;
   private boolean mstart_set = false;

// Stop time elements
   private long mstop = 0;
   private boolean mstop_set = false;

// Context, as in: description of the
// timing run
   private String mcontext;
```

For both start and stop points, Timer keeps track of the value returned by System.currentTimeMillis. (mstart and mstop therefore correlate to timeBefore and timeAfter in the original script example.) It also uses boolean elements to remember whether a timing session for the object has started and/or ended. These flags will ensure that valid actions are taken (you can't, for example, end a timing session before you start it).

Finally, the mcontext element is a string that will be used to display information about the timing session if the programmer provides context information in the calls to start() and/or stop().

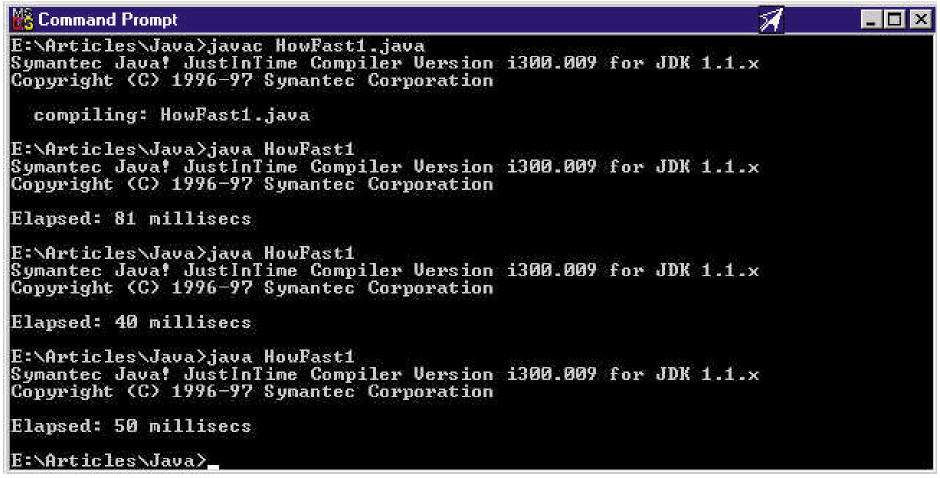
Starting the Timer

To start the timer, you must first instantiate a Timer object and then call the start() method. At the time of start you can also pass a string that will be used in the context or elapsed time display. This is the implementation of start():



```
E:\Articles\Java>javac HowFast.java
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

  compiling: HowFast.java

E:\Articles\Java>java HowFast
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

70

E:\Articles\Java>java HowFast
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

40

E:\Articles\Java>java HowFast
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

40

E:\Articles\Java>_
```

Figure 1: Multiple passes using HowFast for timing



```
E:\Articles\Java>javac HowFast1.java
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

  compiling: HowFast1.java

E:\Articles\Java>java HowFast1
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed: 81 millisecs

E:\Articles\Java>java HowFast1
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed: 40 millisecs

E:\Articles\Java>java HowFast1
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed: 50 millisecs

E:\Articles\Java>_
```

Figure 2: Enhancing HowFast to provide better content

```
// Start the "clock ticking". Of course,
// the clock is always ticking. This just
// captures the starting point.

   public void start (String context) {
      mstart = System.currentTimeMillis();
      mstart_set = true;
      mstop_set = false;
      mcontext = "Elapsed";

      if (context.length() > 0)
      mcontext = "Elapsed from " + context;
      else
         mcontext = "Elapsed from start";
 }
```

The start() method sets the various elements ("I have started but I have not yet ended.") and then constructs the context string based on the value passed in.

Stopping the Timer

When you've finished running the code you wish to time, you call the stop() method to stop the timer. As you'll see below, you can simply call elapsedMessage() or show Elapsed(). They will call stop() if you haven't already done so. Here is the implementation of stop():

```
public void stop (String context) {
   if (mstart_set)
   {
      mstop = System.currentTimeMillis();
      mstop_set = true;

    if (context.length() > 0)
       mcontext = mcontext + " to " + context;
    else
       mcontext = mcontext + " to stop";
   }
   else
      System.out.println (
         "You must start the Timer before you
           can stop it.");
   }
```

There isn't much to stop(); it makes sure you started the timer. Then it gets a snapshot of the current time and adds information to the context string.

Retrieving Elapsed Time

Once the timer has stopped, you can retrieve the elapsed time information. You can do this with one of several different methods seen in Table 1.

As you can see from Table 1, I'm very careful to use previously defined methods to implement other methods. By doing so I avoid code redundancy and get more consistent behavior from the various methods.

Here is the implementation of the elapsed() method:

```
public long elapsed (String context) {
   if (mstop_set == false) stop (context);
      return (mstop - mstart);
   }
```

If the developer hasn't already stopped the timer, elapsed() calls stop() and returns the difference between the stop and start values. It's important to provide the "raw" elapsed() method, because a programmer might want to take that value and manipulate it further by performing calculations, comparing it to values stored in a hashtable or formatting a different message.

The elapsedMessage() method is implemented as follows:

```
public String elapsedMessage (String
 context) {

// Shut down the timing ASAP to make
// results more accurate.
      long elapsedVal = elapsed(context);
      return (mcontext + ": " + elapsedVal
+ " millisecs");
   }
```

The first thing it does is obtain the elapsed time and, as a reminder, the elapsed() method calls end() (if it hasn't already happened). Then it constructs the standard message by incorporating the context information.

The toString method is provided so you can reference a Timer object as part of a string concatenation and see something sensible. Here is its implementation:

```
public String toString () { return
elapsedMessage(""); }
```

It does nothing more than call elapsedMessage(), passing null for the context (you can't provide arguments to the toString() method if you want it to be used automatically by the Java runtime engine).

Finally, there is the showElapsed() method. As you can see below, all it does is display the value returned by elapsedMessage.

```
public void showElapsed (String context) {
   System.out.println (elapsedMessage
    (context));
   }
```

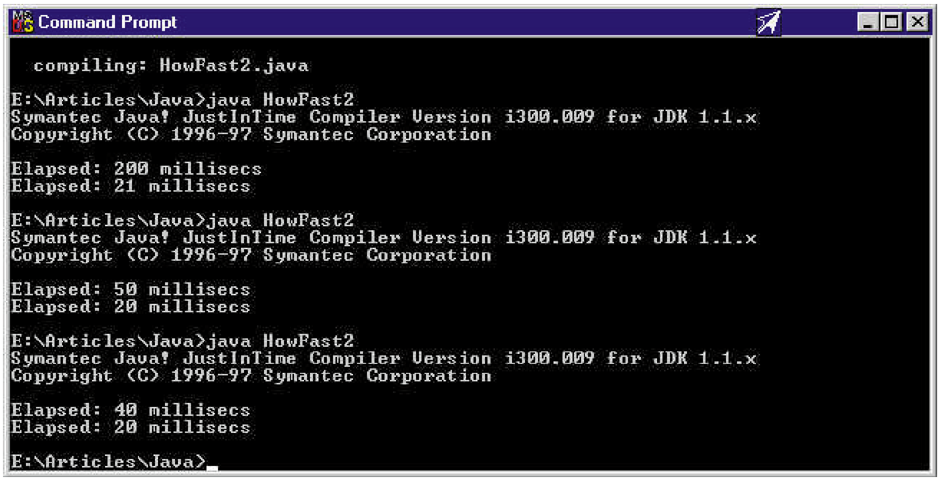
Overloadings for Null Contexts

The Timer class also provides a set of overloadings of all but the toString() method so you can invoke the methods without providing any context information (sometimes you just don't care, so why should you have to pass a dummy "" value?):

```
public void start () { start (""); }
public void stop () { stop (""); }
```

Method Name	Description
elapsed	Returns the "raw" value: the number of milliseconds between start and stop
elapsedMessage	Returns a standard, formatted message containing the elapsed time
toString	Returns the string returned by elapsedMessage, but it passes a null context. In other words, context information must be set in start() and end()
showElapsed	Displays, with a call to System.out.println, the string returned by elapsedMessage

Table 1: Methods of the Timer class



```
 compiling: HowFast2.java

E:\Articles\Java>java HowFast2
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed: 200 millisecs
Elapsed: 21 millisecs

E:\Articles\Java>java HowFast2
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed: 50 millisecs
Elapsed: 20 millisecs

E:\Articles\Java>java HowFast2
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed: 40 millisecs
Elapsed: 20 millisecs

E:\Articles\Java>_
```

Figure 3: Using the Timer class to compare performance of different implementations

```
public long elapsed () { return elapsed (""); }
public long elapsedMessage () { return
         elapsedMessage (""); }
public void showElapsed () { showE
         lapsed(""); }
```

These overloadings might seem like an unnecessary step, but users of the Timer class will appreciate this kind of effort. Well, to be honest, they probably won't appreciate it because they won't even notice it. They'll take it totally for granted and that's just fine – it means you've designed your code very well.

Improving Ease of Use of Timer

So are we done? I hope not. The true objective of this article is, after all, to introduce the concept of abstract classes and show you how to take advantage of them. Why would such a thing be needed to perform timings? Let's investigate the needs in performance analysis a bit more closely.

First, I haven't created any constructor methods for the Timer. Yet it seems that whenever I instantiate a new Timer, I almost always want to call the start() method for that object. Why not create a constructor or two that will automatically set the start time?

Here are the definitions of those constructors:

```
public Timer (String context) { this.start
   (context); }
public Timer () { this.start (""); }
```

I'll make use of these in the subsequent examples.

What else might we want to do? Well, most of the time when I'm testing the performance of a program, I don't want to run that code just once. I might want to run it many times, perhaps even thousands, for a number of reasons, including:

- To vary the inputs and study the response
- To ensure that the program achieves a steady state of performance and returns an accurate average
- To test for aberrant behavior in the program after many executions

If I wanted to run InFile.numBytes multiple times using the current Timer design, I'd need to build a script like this:

```
class HowFast4 {
   public static void main (String[] args) {
      int count = Integer.parseInt
         (args[0]);

      Timer bruteForce = new Timer();

      for (int execnum = 1; execnum <=
         count; execnum++) {
         int result = InFile.numBytes
         ("c:\\temp\\te_event.tps");
      }

      bruteForce.showElapsed("Countem");
   }
}
```

When I invoke the class, I pass the number of times to run the code as the first and only argument, as in:

```
java HowFast4 100
```

and I get a line of output that looks like this:

```
Elapsed from start to Countem: 1723 millisecs
```

The main method calls the Integer.parseInt method to return the numeric value of the integer represented by the contents of the given string object, which is the first element in the string array, args. Then I use this value (the count element) to limit the execution of a numeric FOR loop. Notice that I'm assuming the presence of the Timer constructor to automatically call start().

If I want to compare the performance of my two different implementations, I end up with code similar to Listing 5, which results in an output like this:

```
E:\Articles\Java>java HowFast5 250
Elapsed from start to Countem: 18247 millisecs
Elapsed from start to Available: 150 millisecs
```

As I write this code, it strikes me that I'm engaged in a fairly awkward, repetitive process. What if I want to compare three of four implementations? And wouldn't it be nice to see not only the total elapsed time, but also the average amount of time it took for each execution?

Notice that I execute the same body of code again and again. Here, for example, is all of the "generic" code that performs the timing:

```
Timer myTimer = new Timer();

for (int execnum = 1; execnum <= count;
execnum++) {
   codeToTest; }

myTimer.showElapsed("Context");
```

Doesn't it seem that I should be able to create a template of the repetitive stuff and then just "fill in the blanks" in the template when I need it? Well, I can do precisely that with an abstract class.

Driving Timer to Abstraction

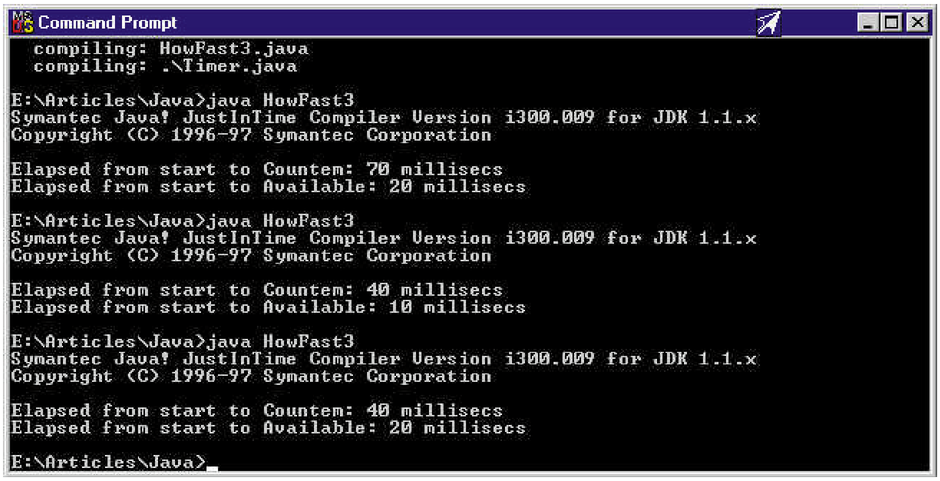
Before I go any further, I must pay proper homage to the second edition of Arnold and Gosling's *The Java Programming Language* published by Addison Wesley, where I got started on this idea by reading page 76. To explain abstract classes, they provided the following example:

```
abstract class Benchmark {
   abstract void benchmark ();

   public long repeat (int count) {
      long start = System.currentTimeMillis();
      for (int i = 0; i < count; i++)
         benchmark ();
      return (System.currentTimeMillis() -
start);
   }
}
```

They then extended that abstract class with a "do nothing" benchmark that I must admit I found hard to understand. After playing around with it for a while and reading through other books, I finally felt comfortable enough with abstract classes to design my own benchmarking class -- built on top of the Timer package.

Back to "abstract class." What is it? How does it help me get my job done? A class is considered "abstract" if it has one or more abstract methods. An abstract method is a method that has a signature or header but no implementation or body. Here's a very simple abstract class:

```
abstract class Beliefs
{
   abstract String aboutGod ();
   abstract String dayOfRest ();
}
```



```
  compiling: HowFast3.java
  compiling: .\Timer.java

E:\Articles\Java>java HowFast3
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed from start to Countem: 70 millisecs
Elapsed from start to Available: 20 millisecs

E:\Articles\Java>java HowFast3
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed from start to Countem: 40 millisecs
Elapsed from start to Available: 10 millisecs

E:\Articles\Java>java HowFast3
Symantec Java! JustInTime Compiler Version i300.009 for JDK 1.1.x
Copyright (C) 1996-97 Symantec Corporation

Elapsed from start to Countem: 40 millisecs
Elapsed from start to Available: 20 millisecs

E:\Articles\Java>_
```

Figure 4: Enhanced output from the Timer class to improve usability

What am I expressing in this class? Human beings have belief systems, but their beliefs differ according to many factors, in this case religious affiliation. If I'm a Jew, my belief about God differs from that held by a Christian. The day of the week that is considered my Sabbath or day of rest is also different. On the one hand, humans have common characteristics: views on God and the day of rest. On the other hand, the actual content behind those characteristics (i.e., implementation of the method) is different for each religion or belief system.

Once I have defined these general characteristics and created "placeholders" for their implementation, I can extend my abstract Beliefs class for particular faiths (or lack thereof) as shown in the four classes in Listing 6, which generates this output:

```
Muslims are sometimes found reading the
Quran on Friday
Jews are sometimes found reading the Torah
on Saturday
Christians are sometimes found reading the
Bible on Sunday
Atheists are sometimes found reading the
Non-Existent on Not Applicable
```

Notice that even though I create an array of Belief objects (called believers), I can assign subclasses of Beliefs (Jews, Muslims, etc.) to that array without casting to the Shape class. In addition, I can invoke the holyBook() and dayOfPrayer() methods for these Shape objects (the elements of the believer array) -- and the methods defined in each of the various subclasses will be invoked.

Thus I can write programs that don't explicitly reference subclasses (Jew, Muslim, etc.) and are therefore more general, but still take advantage of the more specific functionality. This also means that as I extend the abstract class, Beliefs, to other subclasses, existing programs that reference only the Beliefs class will run for the new subclasses as well.

That is a brief explanation of abstract classes. Let's see how we can apply this Java feature to my Timer class.

The RunTimer1 Class

Earlier I identified a typical series of steps taken when using the Timer class: instantiate a Timer, start the clock, run your code, stop the clock and display elapsed time. I then discovered a more complex requirement for use with Timer: run the code *n* times to obtain a more accurate picture of the code's performance. Now my objective is to allow a developer to take advantage of this more sophisticated testing model without having to write the FOR loop and other logic each time.

This is a somewhat tricky prospect. The code to be tested must be placed inside the FOR loop, but I don't want to have to make them write the FOR loop (and other functionality, as we'll see). Conceptually, I want to build a method that implements the FOR loop but contains a kind of "placeholder" for the developer's code.

This is exactly where the abstract class comes in handy. I'm going to extend Timer to an abstract class (remember: Timer is not abstract) called RunTimer1, as follows:

```
abstract class RunTimer1 extends Timer {
...
}
```

As you can see, I extended a nonabstract class to an abstract class. For a class to be abstract it must contain at least one abstract method. In fact, it contains two methods, only one of which is abstract:

```
timeIt()
```

This is the abstract method of the class and therefore contains no implementation. It's the "placeholder" program run by repeat(). It accepts a single object in its parameter list, which is why the class is called RunTimer1 (1 argument version). Here is the complete definition of timeIt() in RunTimer1:

```
abstract void timeIt (Object arg1);
```

Since it's abstract, it presents only the header information – everything you need to know to be able to invoke the method. But since it's abstract, you can't invoke it as an instantiation of RunTimer1; what would the Java runtime engine execute?

```
repeat()
```

The method starts the timer, executes the FOR loop calling timeIt() and then shows elapsed time, as well as some new information: total number of iterations executed and the average elapsed time per iteration. Here is the implementation of repeat():

```
public void repeat (String context, int count,
Object arg1)
{
   super.start();

   for (int execnum = 0; execnum < count;
         execnum++)
         timeIt (arg1);

   super.showElapsed( context);
   System.out.println ("Number of iterations:
     " + count);
   System.out.println (
     "Per iteration elapsed: " +
     (float)super.elapsed() / (float)count);
}
```

As you can see, it calls various methods of the superclass, Timer, as it moves through the steps previously described. (I fully qualify the method invocations to show the reliance on the superclass method, but it's not necessary.) The object argument, arg1, is simply passed in to the abstract timeIt() method. After it calls showElapsed() to display the elapsed time, it offers some "added value" appropriate to this looped execution by displaying additional information available only in RunTimer, where it knows about the number of iterations.

Notice that the body of the loop consists of a call to the timeIt() method. But what good does that do? The timeIt() method doesn't have a body. In fact, it's impossible to even instantiate the RunTimer1 class much less invoke its repeat() method. You can't instantiate an abstract class. Why? Consider the following class:

```
class ZoomZoom
{
   public void main (String args[])
   {
      RunTimer1 myTimer = new RunTimer1();
      myTimer.repeat ("My test", 100, "Who
                Knows?");
   }
}
```

This class will fail to compile with this error:

```
ZoomZoom.java:5: class RunTimer1 is an
```

```
abstract class. It can't be instantiated.
```

What sense could it possibly make anyway? myTimer.repeat calls timeIt() and…there's nothing there to run. The only thing you can do with an abstract class is extend it to another nonabstract class. Then we'll be able to obtain the precise desired behavior: my subclass of RunTimer1 will implement its own version of timeIt(), and when it invokes repeat(), the subclass's code will execute and be timed. And that is pretty darn cool.

Extending RunTimer1

I'll extend RunTimer1 so I can see how long it takes to read the number of bytes in a file. Here is my TestInFile class (separated by explanations of the code; see TestInFile.java for the class as a whole):

```
class TestInFile extends RunTimer1 {
```

TestInFile is a subclass of RunTimer1; an instantiation of TestInFile (as is seen in the main() method) will thus inherit two methods, timeIt() and repeat(). One of them, repeat(), is "ready to go," but the timeIt() method was never implemented in RunTimer1. That's okay – we'll just implement it in TestInFile.

```
void timeIt (Object arg1) {
int result = InFile.numBytes
  (arg1.toString());
   }
```

This very first method in the class has the same signature as timeIt in RunTimer1. This overloading, however, adds an implementation that defines timeIt to be a "pass-through" to call the InFile.numBytes() method. It converts the single object argument to a string that contains the file name.

When I extend an abstract class, I must provide an implementation for each and every abstract method in my abstract superclass (in this case, just one). If I don't, then my subclass is automatically an abstract class, even if it's not declared explicitly as such. Now my subclass will compile and I'll be able to run the repeat() method. So let's now take a look at the main() method and explore how it does its testing:

```
public static void main (String[] args) {
   TestInFile testIt = new TestInFile();
   testIt.repeat (
   "Countem", Integer.parseInt (args[0]),
     args[1]);
   }
} // End of TestInFile package
```

First, main() instantiates a TestInFile object. I can do this since TestInFile extends an abstract class and provides an implementation of its abstract methods. I then call the inherited repeat() method, passing in the context or name of the test, the number of iterations and the name of the file. When repeat() executes, it calls timeIt(). Then the wonders of polymorphism come to the fore, so TestInFile's timeIt() is executed rather than RunTimer1's timeIt() abstraction.

Here is the output from an execution of TestInFile:

```
E:\Articles\Java>java TestInFile 250
c:\temp\te_event.pks
Elapsed from start to Countem:
   18857 millisecs
   Number of iterations: 250
   Per iteration elapsed: 75.428
```

Comparing Performances with RunTimer

As noted earlier, one way I find this sort of timing mechanism to be most useful is in comparisons of the performance of different implementations. You saw this earlier with my numBytes and NumBytes2 programs. Can I use RunTimer to execute each of these and compare the results?

Not as far as I can tell. I'd need to replace the abstract timeIt with two different implementations, one calling numBytes and another calling numBytes2. That doesn't seem possible. I could, however, create two different classes and run them separately, generating output like the following:

> "The objective of the Timer class is to provide a set of methods that allows us to calculate the performance of an individual portion of code"

```
E:\Articles\Java>java TimeNumBytes 250
c:\temp\te_event.pks
Elapsed from start to Countem:
   18297 millisecs
   Number of iterations: 250
   Per iteration elapsed: 73.188

E:\Articles\Java>java TimeNumBytes2 250
c:\temp\te_event.pks
Elapsed from start to Countem: 161 millisecs
   Number of iterations: 250
   Per iteration elapsed: 0.644
```

This works, but it's not as clean as I'd like. Create a new class just to test a different implementation? Yuck! An alternative is to create another version of the RunTimer1 class that has two abstract methods (or, to extend the concept, *n* methods to test *n* implementations) and two repeat methods, as in RunTime1x2 (timeIt1 and timeIt2 each accept 1 argument) in Listing 7.

Then I can create a class to time both of my implementations using the abstract class, as seen in Listing 8.

Here is the output from running this class's main() method:

```
E:\Articles\Java>java TimeBoth 250
c:\temp\te_event.pks
Elapsed from start to Countem: 19228 millisecs
   Number of iterations: 250
   Per iteration elapsed: 76.912
Elapsed from start to Available: 160 millisecs
   Number of iterations: 250
   Per iteration elapsed: 0.64
```

This isn't the ideal solution as far as I'm concerned. You'd have to create a new version of RunTimer for each combination of the number of arguments to timeIt() and the number of implementations you want to compare. There may be a more flexible way to support this variability, but I haven't found it yet.

Summary

This article has explored:

1. How to move beyond learning a specific tip to building a utility that can be used with ease and flexibility. In particular, after discovering System.currentTimeMillis() and figuring out how to use it to calculate elapsed time, I encapsulated that function within the Timer class. The simple methods of this class make it much easier to do performance analysis.
2. The advantages of abstract classes: if you have identified method signatures that apply to multiple subclasses but have different implementations in each subclass, use abstract classes to hide those distinctions. In the case of the timing mechanism, the RunTimer abstract class allows developers to avoid writing complicated benchmarking code again and again.

If you have comments on the techniques discussed in this article, or an idea about how to build a class that would have more flexibility when comparing alternative implementations, please let me know.

▼▼▼▼▼▼ CODE LISTING ▼▼▼▼▼▼

Additional code listings for
this article can be located at
www.JavaDevelopersJournal.com

About the Author

Steve Feuerstein is the author of several books on the Oracle PL/SQL language, including the best-selling Oracle PL/SQL Programming *from O'Reilly and Associates. Chief technology officer at RevealNet, Inc., he is a presenter and trainer of PL/SQL and can be reached at feuerstein@revealnet.com.*

✉ feuerstein@revealnet.com

STRAIGHT TALKING

by **Alan Williamson**

Can Someone Please Take Out the Trash?

The garbage is always coming and going in the Java business

Thirteen – or as I prefer it – 26 over 2. Yes, this is article number (26 over 2) in the series, and the more superstitious of you will know this isn't the luckiest of numbers. So with fingers crossed, let's delve into this month's rants and raves and see what pops out.

This article has been written all over the world. I have composed columns on the move in Sydney, San Francisco, London and even on a napkin in a Tokyo hotel. This month the column comes to you from both sides of the Atlantic. It was started in New York and finished back here in Scotland. I was up in Toronto giving a talk on Java Servlets at this year's WWW8 conference. This is where all the Internet boffins get together and generally discuss the emerging technologies that we are likely to see over the next 12 months.

As you can probably guess, XML featured quite heavily. I'm not too sure about this new technology at the moment; I'm sitting on the fence before making a move. If you're unaware of XML, it's basically a means of transferring data in a format that anyone can make use of, since the information on how the data is organized is contained within the same file. However, it suffers from a lack of global standards on how data is going to be moved about. Sure, the actual XML standard is rubber-stamped, but two Web sites offering catalog information would have to agree on the same data format to make it of real use. On a one-on-one basis, XML seems to be finding applications within large corporate environments where they only need to transfer data between locally controlled boxes. How XML will fare in the open network is anyone's guess.

At the moment I'm camping out in SYS-CON's Web Services Department for the week until I go home. This is much appreciated and very welcome as it allows me to be a fly on the wall in somebody else's office. It's quite strange sitting here watching Americans at work. You're definitely a funny breed, aren't you? I have never met a friendlier race of people in my life. It's quite spooky...and only goes to confirm what we Europeans believe, which is that everyone exhibits at least some of the behavior seen on "The Waltons." (As I pen this, it's just dawned on me that we even have a Mary Ann floating around. My God, this is Walton Mountain!)

JavaOne is only a matter of a couple of weeks away, and by the time you read this it will all be over. I'm sitting here arranging sessions for the SYS-CON radio broadcast, which we'll do from the floor. So, to those of you that have come up and said Hello, and really told me what you think of this column, I thank you – I think – in advance. I'll have a full update in the next column.

Thread.stop();

A couple of days before stepping on the plane to come here, I was wrestling with an absolutely evil problem. Fortunately, I got it fixed in a matter of hours before leaving; otherwise it would have bugged me the whole trip. Let me explain my wee problem and hopefully preempt your having a similar experience.

> *"...either the JVMs are not handling threads... or the documentation is incorrect, leading developers to believe threads are just classes"*

Speaking to you Java developers out there, we've all heard about the wonders of garbage collection and how we don't need to worry about such things as freeing objects and memory anymore. In fact, it's one of Java's main selling points. A wee word of caution though – don't believe everything you read.

One of our clients, Focus Digital, has just launched a new, exciting e-business–based Web site here in the UK, www.bargainfinder.co.uk/. It seeks out the cheapest price for a particular item from a host of merchants. We developed the technology, using Java Servlets that essentially perform a number of concurrent searches, collating the results in a real-time environment to minimize the amount of waiting experienced from the user. The final system works like a dream and is very efficient under heavy load.

It wasn't always efficient, however. Let me explain why.

Since we had a number of concurrent problems to satisfy, we developed a whole system based on the java.lang.Thread class that would spawn off the necessary amount of threads and coordinate the results between them.

During development we had no problems. In fact, the system was running extremely fast and we were very pleased with our initial design. However, a queer problem raised its head after a period of time. Around the three thousandth database query the system would simply hang and not respond to any more client requests. Since it was consistent at the three thousand mark, we automatically looked toward the database as being the problem. It was a place to start, and at that moment it was all we had.

A number of belts and braces went round our database manager class. But no difference. We even took the bold step of rewriting the underlying class to incorporate a number of enhancements. We have a very stable database pool manager that has evolved over a number of years and forms the heart of most of our projects. So I was reluctant to accept that something was wrong with this piece of software. It was performing admirably well in all other clients' products so why should it start going pear-shape now?

Redeveloping it did speed things up a little, and with the new version all tested we inserted it into the main problem. No difference. Bugger! Oh well, at least we got a new improved version of the database manager, so we can't look at it as a complete waste of time.

Scratching our heads for a while didn't yield anything useful. Nothing left but to develop a watchdog thread that would periodically print out the status of various core parts of the system. We've used such devices in the past to great success, but in this instance nothing new was highlighted. One thing we did notice, however, was that even our watchdog thread stopped printing out information after the three thousandth query. The plot thickens.

Eliminating the database from the problem, Ceri started to look at the process usage under Linux to see if anything obvious was going on. Ceri noticed that memory was being used up at a tremendous rate, but was never released. This was not a major concern to me, knowing that most JVM implementations rarely give memory back to the system when allocated. The memory internally may not be all used, but as far as the operating system is concerned the JVM has it all. Time to move on.

We did some more testing, but this time,

instead of hitting the server with just one process, we flooded it with over 20 requests per second to see when it would break. It still broke at the three thousand mark, but it got there a hell of a lot quicker. Oh well, if anything, it would speed up the debugging procedure.

Still thinking about the information Ceri had thrown up, I decided to modify our watchdog timer to print out the virtual memory usage from the point of view of the JVM. In addition to this – on a whim, it has to be said – I decided to print out the number of active threads in the JVM. We ran the tests again, not expecting a great deal. But an interesting picture started to emerge.

The memory was indeed being used up, as was the number of threads. However, the memory did go down, but the number of active threads? Never. They never once moved down the way. Aha! Caught the bugger! I mustn't be catching an exception in a thread and it's obviously crashing. How remiss of me! Not like me to miss such things. But after inspection of all our thread-based classes, all the proper try–catches were being dealt with. So what was going on? I looked carefully at each of my thread classes, but nothing untoward was shining through. I then took the drastic step of placing println(...) at the end of each run() method just to confirm that the threads were running to their logical end and weren't caught in an infinite loop somewhere.

Ran the tests again, and all threads were completing, but the active thread count still continued to grow. I looked back at the documentation on threads and, as I expected, the garbage collector cleared up threads after they finished and went out of scope. No need to call the stop() method, and since Java 2.0 has deprecated this method I looked elsewhere.

Calling the garbage collector manually didn't help one single bit. What the hell, I thought, let's call the stop() method after each one. With a little reworking of some logic, all created threads were having their stop() method called. Ran the tests again and lo and behold, the thread count went up and down. Hooray. Problem solved.

Couldn't believe it. We thought, "Maybe it's the JVM on Linux," but after setting up with the official Sun JVM under NT, we found the exact same behavior. Now we can conclude one of two things: either the JVMs are not handling threads properly – they seem to escape the attention of the garbage collector – or the documentation is incorrect, leading developers to believe threads are just classes. I'm not sure which side of the fence I'm going to come down on, but I'd be interested if any of you have found the same thing or seen similar behavior.

Mailing List

Which leads us nicely into my monthly plug of our mailing list. It's good to see the list growing, and some of the topics that are debated are very interesting. I'd be very keen on hearing your views on this threading issue, so please join and let's discuss it. Send an e-mail to listserv@listserv.n-ary.com with subscribe straight_talking-l in the body of the e-mail. From there you'll get instructions on how to participate on the list.

Salute of the Month

Last month I started a new feature that takes a person and gives them the "Salute of the Month" for work above and beyond the call of duty. This month I have to give a group salute to the whole team at the SYS-CON offices for making my time there extremely enjoyable. I'd like to single out the main man, Fuat Kircaali, for taking care of me the whole week and for a "Fuyacht" boat trip I'll never forget. When I look at postcards of New York, I still can't believe I had the honor. This more than makes up for leaving a kilted Scotsman stranded late at night in Newark Airport, and having a brush with the law later that night in Pearl River. 'Nuff said!

Book Review

This month I'm in the middle of the Oracle book that profiles the ups and downs of Oracle and Mr. Ellison. Not being a great fan of the way Oracle works, I'll leave the review till I finish the book. But I have to admit to not being able to put it down. What a read! I've read many of these types of books, and like many of you entrepreneurs I get a lot of inspiration from seeing how the big companies became big companies. But the one thing I've never read is how they broke into their first million. We hear the stories only after they have multimillion sales, and as interesting as this is, I'm sure there are equally as many tales and advice to be told at the start of the journey as opposed to halfway through. So, to any big CEOs out there that read this column, we want to know the details of the basic steps.

Maybe Ellison, Grove, Gates and Kaplin can do a George Lucas and release a prequel to their books so we can see how the story started. The only book I've read that really goes through the early steps is from Richard Branson regarding the Virgin empire, but, sadly, this isn't computing-related. So come on, you big CEOs: tell it to us from the start.

I'd better finish up as I'm running out of paper, and now that I'm back in Scotland it has a wee bit longer distance to travel to get back to New York. So in the time-honored tradition, as I look back at Walton Mountain, I shall bid you farewell....Goodnight, Mary Ann....Goodnight, Jim-Babb....Goodnight, M'lou!

About the Author

Alan Williamson is CEO of n-ary (consulting) Ltd. A Java consultancy company with offices in Scotland, England and Australia, they specialize solely in Java at the server side. Alan is the author of two Java Servlet books and contributed to the 2.1 Servlet API. He can be reached at alan@n-ary.com (www.n-ary.com) and welcomes all suggestions and comments.

alan@n-ary.com

Securing Java Commerce

How Java leverages the technologies used to provide security in distributed systems

by **Ajit Sagar**

Last week a friend of mine who lives in Hong Kong was telling me how advanced the business environment is there. Folks that have Internet access actually use the business facilities the Internet offers. For example, people use the Internet for their regular grocery shopping. They place orders via the Internet, use their credit cards for the transactions and have the goods delivered to their home. Everyday business is conducted on the Web. For some reason commerce in the United States hasn't advanced to that level. The technology is here and has been for a while; however, people don't completely trust the Web for business transactions.

Security is the primary concern of people and businesses that commit to using the Internet for conducting business transactions. An Internet transaction is conducted between two parties, and each party needs to have a certain level of trust in the other. For electronic transactions to be the primary mechanism for conducting business, the level of trust needs to be as high as – if not higher than – that of person-to-person transactions. The parties involved in electronic trade also need to have a high degree of trust in the medium used for the information transfer, namely the Internet.

This month we'll take a look at the security issues involved in electronic trade, some of the technologies used for providing security in distributed systems and how Java leverages these technologies. Although my focus is on Java and doesn't cover Internet security in detail, I've provided a brief introduction to some of the related concepts where required.

Security Concerns for Business Applications

Internet commerce typically involves two or more machines connected via a public medium. Some of the security issues in this environment are:

- Private and sensitive information exchanged between the two machines may by viewed by some third party that's monitoring the transmission channel.
- The originator of the transmission may not be who he or she claims to be.
- The recipient of the information may not be who he or she claims to be.
- The data may be corrupted for malicious purposes during transmission.

Business applications in general have the following categories of security concerns:

- *Security of content:* This addresses the protection of data at its point of origin, before it's transmitted or received. Parties conducting a business transaction need to be sure that the data isn't available to unsolicited parties for viewing or manipulation.
- *Security of communications:* The data needs to be protected against tampering and visibility during the process of transmission.
- *Security of the corporate computing infrastructure*

The above concerns address the integrity of the data during the electronic transaction across the Internet. The data itself could have malicious programs embedded in it. Thus, even after it passes the security checks for correctness, it could cause harm to the local environment. This usually happens in the case of newly installed or downloaded software.

Ensuring an acceptable level of security for each of these categories requires the following:

- *Proof of identity (authentication):* Confirm that the source and destination parties are who they say they are.
- *Data integrity:* Ensure that the data hasn't been tampered with since the originator created it.
- *Privacy:* Ensure that the data isn't available to a third party (one not involved in the trade) for reading or manipulating during the transaction.
- *Nonrepudiation:* Retain proof that a business transaction did take place. Nonrepudiation prevents a party from denying its participation in a transaction.

AAA

When the acronym *AAA* is used in the context of security, it doesn't stand for American Automobile Association (although different security concerns are addressed by that organization). AAA stands for *authentication*, *authorization* and *accounting* – three mechanisms to ensure the security of a distributed computer system during remote access.

- *Authentication:* Authentication is required on secure systems to ensure that persons logging in are who they say they are. Authentication is also required for a message exchange to verify that a particular message has not been fabricated or altered in transit.
- *Authorization:* Authorization determines what users can do once they are authenticated. Once the person is logged in, authorization controls are used to restrict their access to various resources based on the rights and privileges assigned to their user accounts or the objects they access.
- *Accounting/logging:* Auditing is the collecting and monitoring of events on servers and networks for the purpose of tracking security violations and keeping track of how systems are used. A network auditing system logs details of what users are doing on the network so that malicious and unintended activities can be tracked.

Security Concerns in the Java Platform

The Java platform supports a new paradigm for distributed computing. While this enables the creation of very powerful and extensible applications, it also introduces new concerns about Internet security for applications written in Java. These concerns are addressed by Java's Security APIs and by the Java Virtual Machine.

Java provides rich support for distributed commerce by being the application building platform for full-scale client/server systems. Java components that constitute a complete commerce application may exist as a combination of applets on the client, servlets on the server and client or server Java applications. Commerce services may be exposed in a distributed topology via CORBA or RMI. Data may be transmitted between tiers of an *n*-tier application via IIOP (Internet Inter-ORB Protocol), HTTP or raw sockets. Figure 1 illustrates a distributed topology in the Java environment.

These different facets of a Java application give rise to several potential security loopholes. A security solution for a distributed application needs to address all these loopholes since the system is only as strong as its weakest link. In other words, a security solution needs to be holistic. The Java Security APIs, the Java Cryptography Architecture (JCA) and the Java sandbox, ClassLoader and SecurityManager all work together to ensure security in a Java application.

Protecting the Java Client Environment

The ability to dynamically download and run Java applets over the Internet is one of Java's most powerful features supporting distributed computing. However, downloaded code is untrusted code from the client's perspective, and this opens up the following kind of security risks:

- *System modification:* The applet can make changes to the browser or the client system in a harmful way.
- *Privacy invasion:* The privacy of restricted information on the client system could be compromised.
- *Denial of service:* The downloaded code could use up system resources and thus interfere with the normal operation of the system by denying service to other, more critical system processes.

Java's sandbox model addresses these security risks. The ClassLoader, SecurityManager and bytecode Verifier ensure that untrusted code executes only within a restricted environment. Within this environment the code can't access resources in a system that's sensitive from the security viewpoint. The code executing within the sandbox has no local disk access, linkage to local code or printing facilities.

However, placing these restrictions also prevents applets from performing more useful functions and severely limits their capabilities. The Java security model is built around the concept of a protection domain. The applet sandbox is a protection domain with very tight controls. Java applications, on the other hand, execute in an environment with no control other than those imposed by the underlying operating system. Java 1.1 relaxes the security model by introducing the concept of digitally signed applets. Signed applets are treated as trusted code since the signature ensures that the applet was downloaded from a source trusted by the client.

Data Encryption in Java

Encryption is the process of converting data from a readable to an unreadable format. Decryption is the opposite process. Data is encrypted when transmitted from the sender to the receiver, the premise being that only the intended recipient knows how to decrypt the message. Thus data encryption ensures data integrity and privacy. A discussion of the protocols involved in encrypting and decrypting data is beyond the scope of this article. However, we'll briefly discuss Java's support for data encryption.

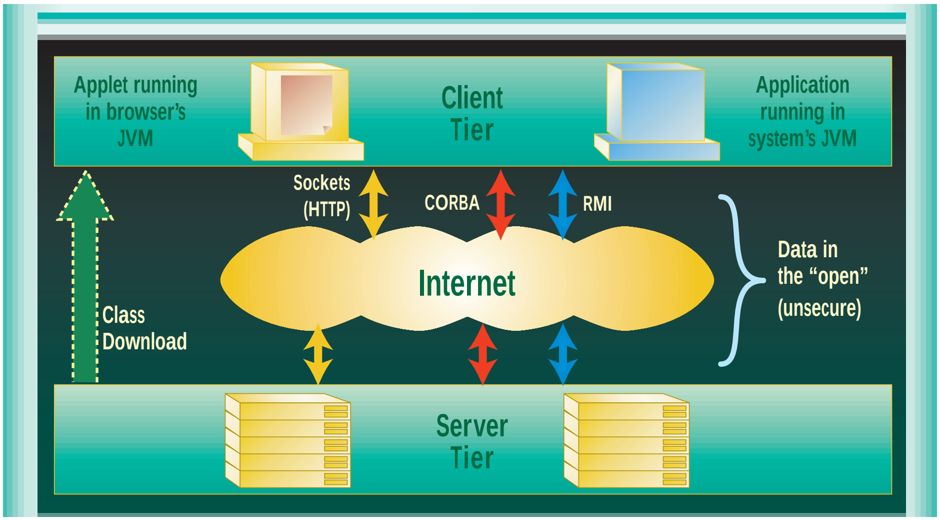

Applet running in browser's JVM
Client Tier
Application running in system's JVM
Sockets (HTTP)
CORBA
RMI
Internet
Data in the "open" (unsecure)
Class Download
Server Tier

Figure 1: Distributed topology

The Java security architecture allows security algorithms for encrypting data to be plugged in using a java.security.Provider interface. A security provider is a package that implements a set of security algorithms, including message digest, signing and encryption algorithms -- three mechanisms for encrypting/decrypting data.

JDK 1.1 provides general-purpose APIs that support encryption – Java Cryptography Extensions (JCE) and Java Cryptography Architecture (JCA). JCE comprises a set of additions for the java.security package that implements cryptographic streams.

JDK1.1 also introduced the JAR (Java Archive) format for distributing code. JAR files are single files with the .jar extension that package multiple Java files. The files in the JAR can be digitally signed prior to distribution. This allows the end user to authenticate the code running on his or her client machine. The package javax.security.cert is a Java extension API that provides support classes for X.509 certificates.

Java Support for SSL

Several communications protocols used in network communications today add a security layer to existing communications protocols. The most popular of these is the Secure Socket Layer (SSL) protocol used to create secure connections between the client and the server in a networked environment. SSL was developed by Netscape to allow their browser and server products to conduct electronic commerce securely, and with the confidence that the customer's financial data, such as credit card information, could be transmitted over the Internet without being compromised. SSL attempts to solve end-to-end transmission security by providing authentication for both the client and the server.

SSL support for Java-based applications is included in many popular Web browsers and servers. These programs depend on SSL to provide the necessary encryption when transmitting data across the Internet. In order to use SSL, "https://" is used in the URL instead of the traditional "http://". The "s" in "https" implies SSL communications.

Similar to Netscape's SSL, several vendors also provide SSLSocket and SSLServerSocket classes that extend the java.net package's Socket and ServerSocket classes, respectively. The javax.net.ssl is an SSL API, which is a standard extension to JDK 1.1 core API. The classes in this API resemble the java.net socket classes; the dif-

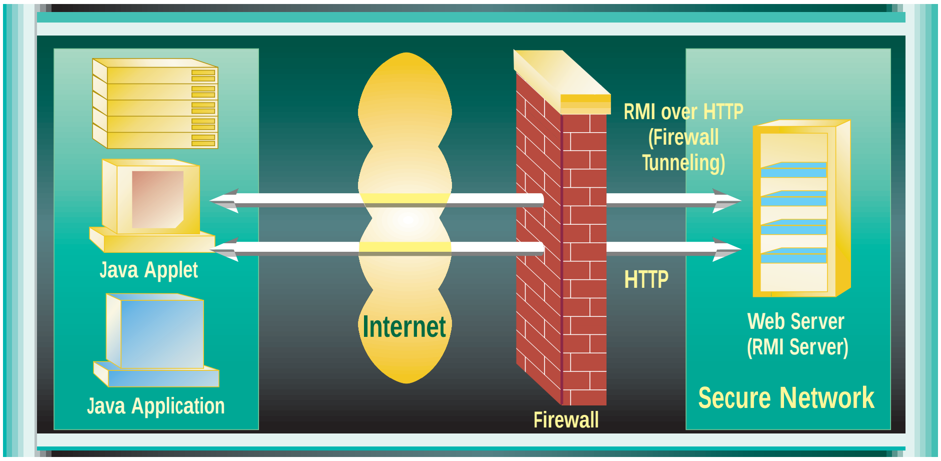


Figure 2: Firewall tunneling

ference is that when sockets from this package are used, they establish secure connections.

Java Servlets

Java servlets are Java classes that are loaded and run by Web servers. While they aren't subject to the same restrictions as applets, they're only capable of serving HTML to the client. Servlets are a popular mechanism for accessing system resources and services on the server. Since they run within the Web server, the built-in safety features of the server ensure secure execution. Since they're not downloaded classes, they don't have the "trust" issue that applets do.

Java Access Through a Firewall

A firewall is a computer service that links two or more networks and enforces some access control policy between them. It can be implemented in hardware, software or a combination of both. Companies implement a firewall as a front end to their corporate intranet to control access to their computer resources. These resources are said to sit "behind" the firewall.

When working with a Java-based architecture, firewalls control the loading of Java applets to a client and network accesses by Java applets to a server. Java applets are accessed using HTTP as the transport protocol. If the security policy of the firewall is to allow HTTP traffic to pass through, the applet or the JAR housing the applet will be like any other component of a Web page. If HTTP is disallowed, then downloading class files is going to be very difficult. Most firewalls allow protocol access via well-known ports (e.g., HTTP via port 80, FTP via port 21). If an application needs to communicate using other protocols, they'll have to "tunnel" that protocol within the HTTP connection.

In firewall tunneling the communication protocol used by the application is wrapped in another protocol (like HTTP) so it can pass through the firewall. Behind the firewall a special program, called a *gateway*, is used to extract the contents of the actual protocol. For example, in the case of RMI, JavaSoft provides the RMI-CGI Gateway, a program that allows the RMI connection to pass through firewalls. Firewall tunneling in the context of a Java environment is illustrated in Figure 2.

RMI Security

Java's RMI allows distribution of Java objects across different machines. RMI's RMISecurityManager and RMIClassLoader can be used to impose strict security policies on Java objects exchanged across the network. Since encryption and authentication are not part of RMI, new security issues are opened up when RMI is used across the Internet.

Additional issues need to be considered when the RMI client and server are connected through one or more firewalls. As mentioned earlier and illustrated in Figure 2, JavaSoft's RMI-CGI Gateway may be used for firewall tunneling. Typically, RMI is tunneled over HTTP. However, if a CORBA environment is used, RMI may be tunneled over IIOP. A detailed discussion on RMI tunneling warrants a separate discussion and is beyond the scope of this article.

Trading Places

This has been a quick whirlwind tour of some of the security issues involved in using Java for distributed architectures. Security has many other aspects I haven't even begun to cover. One of the main things to remember is that the designers of the Java platform treated security as one of their primary design criteria. Hence Java tends to address these issues in a much better and more holistic way than other computing platforms. That's one of the many reasons Java is being used to design various tiers of distributed commerce systems.

About the Author

Ajit Sagar, a member of the technical staff at i2 Technologies in Dallas, Texas, holds an MS in computer science and a BS in electrical engineering. He focuses on Web-based e-commerce applications and architectures. Ajit is a Sun-certified Java programmer with nine years of programming experience, including two and a half in Java. You can e-mail him at Ajit_Sagar@i2.com.

Ajit_Sagar@i2.com

JDJ FEATURE

AN ONLINE AIRLINE TICKET STORE USING JAVA AND COLDFUSION

Part 2

Working Together:

Competing, yet complementary, technologies

by **Ajit Sagar**

This is the second in a series of articles focused on using some of the prominent Internet and Java technologies to develop a Ticket Store application. In the last issue of JDJ we defined the APIs and technologies and the network topology that would be used to develop the Ticket Store. We also walked through skeletal definitions of the classes used to implement this application. Now we'll add some meat to these classes.

We'll also define some workflows and scenarios for our application, with emphasis on the design of the objects for the middleware tier of the store. Our focus will still be on the purchase of tickets via an online travel agent that gets quotes from different airlines and presents the Internet user with the best quote. The online store portion of our application will be discussed in the next article in the series.

The UI design for this application isn't a major part of our Java-based design. A basic browser UI will be developed for the end user. In corresponding issues of ColdFusion Developer's Journal (Vol. 1, issues 4–6) we'll develop the more sophisticated and personalized UI in parallel using ColdFusion. I'd like to reiterate that as this isn't a real-world application, several decisions made for the design will be oversimplified, their primary purpose being to illustrate how the components defined here can be integrated into a distributed application.

Of the four tiers of the Ticket Store application described in the last article, the Service Access tier, and specifically the Ticket Reservation and Sales Broker, will occupy most of our attention. As defined in last month's article, the Service Access tier is a middleware tier that accepts service requests from the Merchant Server tier, routes them to the Application Services tier and serves back the response to the Merchant Server tier. The Merchant Server tier adds in user-specific data and sends back the response to the user interface.

Ticket Reservation and Sales Broker Requirements

The Ticket Reservation and Sales Broker (we'll call it "Broker" from this point) is actually our simulation (albeit oversimplified) of a real-life ticket agent. The software modules that make up this tier attempt to duplicate the base functionality of a human ticket agent who would typically gather information about the end user's (user's) flight requirements, search in his or her reservation system for flight availability and prices, and get back to the end user with a quote for the flight. Thus the main functions of the Broker are:

1. Accept a request for a flight from the user.
2. Define search criteria for getting flight quotes.
3. Search for available flights based on the criteria.
4. Obtain quote(s) from the reservation system.
5. Add promotions or discounts on the price based on the user's purchase history.
6. Return the quote(s) to the user.
7. Accept a reservation request from the user. This includes getting his or her credit card information.
8. Reserve the seat(s) for the user
9. Return a confirmation to the user.

This is a simple workflow that runs through the system in sequential fashion. We'll implement this workflow in our system, ignoring other functionality such as canceling a reservation, etc. The idea is to demonstrate how data flows through our Ticket Store from the end customer to the back office and back. Figure 1 illustrates the use cases for the Broker. The workflow is illustrated in Figure 2.

Class Design

Now let's define the classes involved in this set of transactions from the Broker's point of view. For now, we'll forgo discussion on the UI and assume that the user's input somehow arrives at the Broker. Allaire's ColdFusion will be used to develop a more sophisticated UI. (This will be discussed in issue 5 of ColdFusion Developer's Journal.)

Step 3 of the workflow defined above is encapsulates the main functionality of the Broker. Let us break this step down and identify the classes that we will need to provide this functionality. The classes needed for the design of the Broker and their function are summarized in Table 1. The relationship between the classes is illustrated in a class diagram in Figure 3. The classes and code listings are described below.

TicketBrokerServlet

This class was defined in last month's article as the TicketServlet. I've renamed it here to better indicate its functions, and I'll also expand on its functionality. The TicketBrokerServlet is the crux of the Broker. It receives a ticket request from the Merchant Server, packages it into a query, submits it to the Service Access tier, receives ticket quotes and

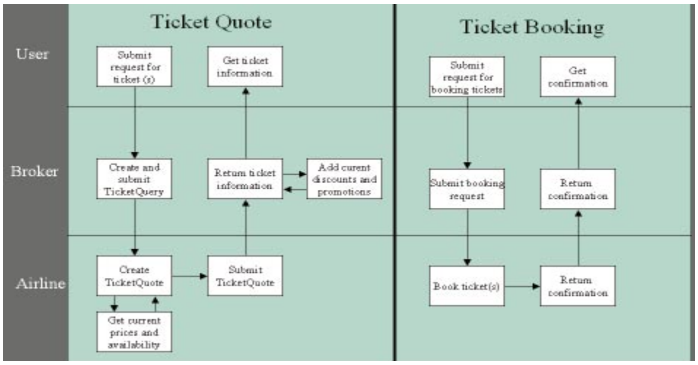


Figure 2: Workflows for the Broker

Source	Class Name	Purpose
Listing 1	*TicketBrokerServlet*	The Broker's interface into the Merchant Server tier
Listing 2	*TicketQuery*	Encapsulates a request for a ticket quote
Listing 3	*TicketQuote*	Encapsulates a ticket quote from the airline
Listing 4	*TicketService*	An interface implemented by the client stubs and server skeletons of the middleware protocols used to interact with the Application Services tier
Listing 5	*TicketServiceManager*	Provides an implementation of the TicketService interface that manages the four types of protocols (RMI, CORBA, Servlet an socket) for the ticket service
Listing 6	*SocketTicketClient*	Client for socket-based quote service offered by SeemaAir
Listing 7	*RMITicketClient*	Client for RMI-based quote service offered by KarunaAir
Listing 8	*CORBATicketClient*	Client for CORBA-based quote service offered by NitiAir
Listing 9	*ServletTicketClient*	Client for Servlet-based quote service offered by ApuAir
Listing 10	*TicketPricer*	Calculates the final price for the ticket based on current discounts offered by the Broker

Table 1:Ticket Reservation and Sales Broker classes

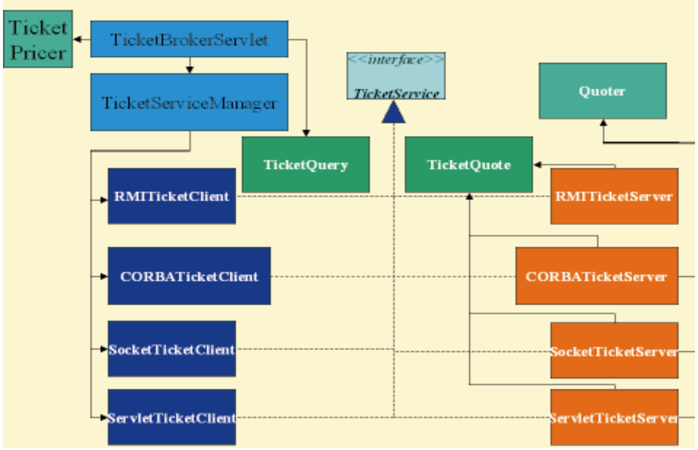


Figure 3: Ticket agent classes

tious airline carriers in the Service Access tier – SeemaAir, KarunaAir, NitiAir and ApuAir. SeemaAir makes its prices available via a simple Socket interface. KarunaAir exposes its services via an RMI service. NitiAir uses a CORBA server. ApuAir offers a service via a Servlet, i.e., it supports a URL invocation. TicketBrokerServlet (which represents the Broker's connection layer) sends the TicketRequest to each of these carriers and waits for responses on each of them. When it receives all the responses, it uses the TicketPricer (described later) to send back the best quotation. The network connections between the Broker and the airline components in the Application Services layer are illustrated in Figure 4.

Since the Broker supports four kinds of services, it creates four clients, one for each service. These clients are instantiated by the TicketClientManager class, which establishes and manages connections to the different airline carriers:

```
 if (clientManager_ == null)
          clientManager_ = new
TicketClientManager();
```

The service() method of the TicketBrokerServlet instantiates the TicketClientManager. It then calls a getQuotes() method on this instance. The method getQuotes() returns a vector of TicketQuote objects. The four client classes (SocketTicketClient, RMITicketClient, CORBATicketClient and ServletTicketClient) and the TicketQuery and TicketQuote classes are described later in this article.

The next method call in the service() method is to the method getBestQuote(). A vector of TicketQuote objects is passed in as a parameter. The method getBestQuote() calls static methods on the TicketPricer object to get the final price on each of the quotes obtained from the airlines. It selects the best price and sends a new TicketQuote object to the service() method of the servlet. The service() method calls the getQuoteString() method and passes it the TicketQuote object that it just got back. The resultant string is returned to the invoker of the URL. The string is in the form of name-value pairs, similar to the input received by the TicketBrokerServlet.

One of the parameters passed into the servlet is a "TYPE=VALUE" parameter. The "VALUE" can be "QUERY" or "BOOK". When the string "Query" is passed in as an argument to the servlet, the getQuotes() method described above is invoked. When the string "BOOK" is passed in, the bookSeats() method is called to actually book the flight. The bookSeats() method is described later in this article under the TicketService class.

TicketQuery

The user will use certain criteria for his or her flight query/request. Since this query will have the same parameters for all the users (e.g., name, address), it's encapsulated in a separate class called TicketQuery. This class was defined in last issue's article. The TicketQuote class is a simple class with only getter and setter methods for the data fields. The fields in the TicketQuery class are:

- query ID
- departure city
- arrival city
- begin date
- end date
- no. of seats
- seat class (coach, business, first class)
- seating preference (window, aisle)
- smoking preference (smoking/nonsmoking)

Notice that the class has a "begin date" and an "end date." This is because our reservation system allows a user to specify a range of days on which he or she can fly. For customers who have flexibility on the actual day of the flight and want to base their reservations on the cheapest fare, the system will search for the cheapest fare available in the time date range. If the user has no flexibility concerning the date, he or she will pass in the same value for the begin and end dates.

TicketQuote

The TicketQuote class encapsulates the response from an airline in the Application Services tier to the Broker. Similar to the TicketQuery class, the TicketQuote class is a simple class with only getter and setter methods for the data fields. Some of the fields are just copied from the TicketQuery object to the TicketQuote object. The queryId is a unique ID that indicates the request.

The fields in the TicketQuote class are:

- query ID
- airline (SeemaAir, KarunaAir, NitiAir, ApuAir)
- departure city
- departure date
- departure time
- arrival city
- arrival date
- arrival time
- no. of seats
- seat class (coach, business, first class)
- smoking preference (smoking/nonsmoking)
- total price

In case there is no availability based on the search criteria, the TicketQuote returns with a "0" in the noOfSeats and the other fields are invalid. This is not the most efficient way of sending a response, but it will serve our purpose.

TicketService

All four <protocol>TicketClient classes inherit from the TicketService interface and hence provide implementations of the getQuote() method. The functionality for all the clients is the

same. They basically take in a TicketQuery object and return a TicketQuote object. They also implement a bookSeats() method that takes in a queryId parameter for booking the flights. In our implementations the server-side counterparts for these classes don't perform any real-time function (actually checking for availability, etc.). The prices are looked up in an MS Access database and the seats are assumed to be booked as soon as the request is made.

My assumption is that the reader is familiar with the workings of the different protocols used in this application. Thus I won't go into details of the classes used here. Each of the service protocols (Socket/RMI/CORBA/Servlet) has the following set of classes as illustrated earlier in Figure 3:

- ***<protocol>TicketClient, e.g., RMITicketClient***: This implements the TicketService interface. The methods getQuote() and booksSeats() in this class throw a more specific exception (RemoteException) as compared to the superclass methods in the TicketService interface.
- ***<protocol> TicketServer, e.g., RMITicketServer***: This class resides in the Application Service Layer and also extends the TicketService class. The RMI TicketServer is described later in this article.

In the interest of space and general sanity, I won't go into details of the classes for all four types of services. I will cover the RMI example here. The understanding of the rest is left as an exercise for readers.

In a real-world application the responses from each "airline" could take varying amounts of time.

There would also be a timeout for waiting for the responses. This kind of a transaction is asynchronous and would be best handled by spawning each ticket quote service as a separate thread. However, in our application we'll assume that the responses are instantaneous (indeed they will be, because our Service Application Tier is pretty much hard-coded). Therefore, each of the clients that request a quote from the corresponding airline server is started sequentially as shown in the TicketBrokerServlet class. The clients for the different protocols are described in the next four sections.

TicketClientManager

As the name suggests, this class is responsible for managing the TicketClients. The TicketClientManager constructor creates all four clients. Each client does the actual connect (Sockets) or bind (RMI/CORBA), or establishes the URL connection (servlets) and returns a reference to the TicketClientManager.

The TicketClientManager contains a getQuotes() method that calls a getQuote() on each of the clients. Each client in turn calls the getQuotes() method on its corresponding remote server. The resultant value is a TicketQuote object that is passed back to the TicketBrokerServlet. The TicketClientManager also contains a bookSeats() method that is forwarded in a similar fashion to the corresponding protocol server. The bookSeats() method returns a boolean value that indicates the result of the operation. The bookSeats() method takes in a string parameter that indicates the protocol (and corresponding airline) that was selected for the best fare.

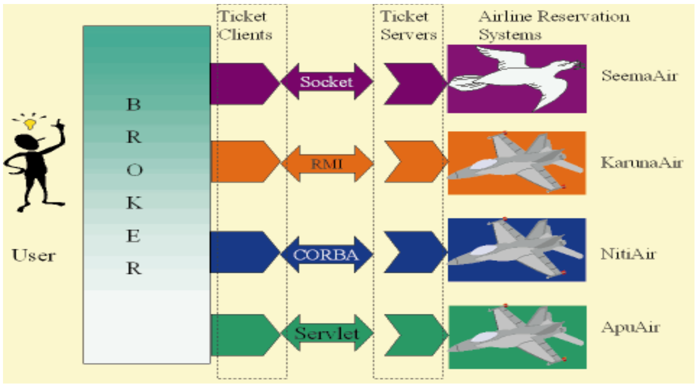


Figure 4: Broker connections with the airlines

Source	Class Name	Purpose
Listing 11	*SocketTicketServer*	This provides the quotation service for SeemaAir that uses raw sockets as the mechanism to transfer the quote back to the Service Access Tier.
Listing 12	*RMITicketServer*	This provides the quotation service for KarunaAir that uses RMI as the protocol to transfer the quote back to the Service Access Tier.
Listing 13	*CORBATicketServer*	This provides the quotation service for NitiAir that uses CORBA as the protocol to transfer the quote back to the Service Access Tier.
Listing 14	*ServletTicketServer*	This provides the quotation service for ApuAir that uses a servlet as the mechanism to transfer the quote back to the Service Access Tier.
Listing 15	*Quoter*	This provides a quote for the ticket request.

Table 2: Application Service Tier classes

RMITicketClient

This is the RMI client that gets the ticket quote from an RMI-based service. It implements the TicketService interface. This service is the one used by KarunaAir. The constructor establishes a connection with the RMITicketServer. The getQuote() method calls the corresponding getQuote() method on the RMITicketServer handle. Similarly, the bookSeats() method calls the corresponding method on the server.

TicketPricer

The purpose of the TicketPricer is to get the final price on a quote. The implementation of this class uses a random selection to give a discount on the price. In a real-world application the broker would add promotions, discounts, and so forth based on the ticket agent's pricing policies. The TicketPricer has a single method, getDiscountedPrice(), that returns a new price for the tickets.

Application Service Tier Classes

The Service Access Tier consists of five classes. Four of them represent the different type of connection protocols – RMITicketServer, CORBATicketServer, SocketTicketServer and ServletTicketServer. Each of these classes looks up the ticket prices from a static database implemented in Microsoft Access. This is achieved via the Quoter class. These classes and their purpose are listed in Table 2.

Each class inherits from the TicketService interface. The classes provide implementations for the client-side stubs. Hence, they provide implementations of the getQuote() and bookSeats() methods. The code in the ticket server classes is trivial and is not described here.

Quoter

The Quoter class establishes a database connection with the Microsoft Access table, Quotes.mdb. It looks up the price of each ticket based on the airline. The ticket prices are hard-coded.

Running the Programs

The code for this article is available at www.JavaDeveloperJournal.com. It was compiled and tested on a Windows NT 4.0 workstation. To run the programs you will need the following: *JDK 1.1.x, JSDK 2.0 (Java Servlet Development Kit), your servlet engine and Web server, MS Access database and Visigenic Visibroker 2.5+.* Instructions are also available at the Web site.

Ajit_Sagar@i2.com

Interfacing with Legacy Libraries Using Remote Method Invocation

Getting there from here

by **Scott Howard**

The assignment was enough to make any neophyte Java developer bolt for the door: to provide a remote method for use by an applet that invokes a native method that wraps a function in an existing legacy library. Mentally calculating the odds of making it to the parking lot, I discarded that option and indicated my willingness to assume responsibility for the task with an air of cautious confidence. The purpose of the remote method is to return an instance of a class object whose contents reflect the data structure returned by the legacy function. Little did I know what I was getting myself into.

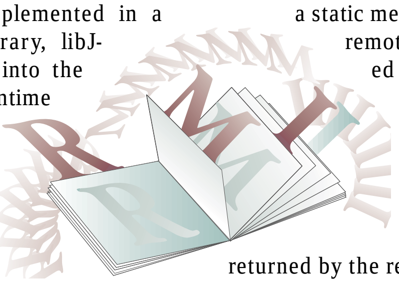
Perhaps the most significant hurdle I had to overcome was the lack of useful documentation to help direct my efforts. While embroiled in implementation, I spent an entire day poring through the RMI usergroup archive on Sun's Web site searching for guidance – to no avail. I would've spent a lot of time wading through their JNI usergroup archive as well, but I couldn't seem to locate one. Subsequently I made the decision to document my findings so as to assist others.

Before we start on the class design, let's look at what the existing legacy code (Get_Legacy_Data) does. An ASCII file is read from the local disk, then its contents are parsed into a Legacy_Type structure whose address is passed as an argument by the caller (see Listing 1). Not much to it, really. The legacy code was compiled into a shared object library, legacy.so, using the IRIX 6.2 compiler and then loaded onto the Web server, a Silicon Graphics Indy station loaded with the IRIX 6.4 operating system.

The first requirement for class design is a class that acts as a template for the data structure that's returned by the legacy function. This class, JLegacy, declares a series of public instance variables that correspond to the members of Legacy_Type and provides a constructor that has no parameters (see Listing 2). This constructor is never called, not even by the native method that allocates the object for return to the remote method.

Next, the remote interface declaration for the remote object must be defined. The remote interface is a Java interface that extends java.rmi.Remote, used exclusively to identify remote objects. The remote method getJLegacy, which is defined by JLegacyIF, returns a JLegacy instance and throws java.rmi.RemoteException, which provides a mechanism to handle any failures (see Listing 3).

Now that the remote interface has been defined, let's look at the design of the remote object, JLegacyRO (see Listing 4). For JLegacyRO to implement getJLegacy, it must interface with the existing legacy code through a native method, getN. This method is declared in the JLegacyRO class but implemented in C, just like the legacy code. It returns a JLegacy instance and is declared static since its implementation is the same for all instances of the JLegacyRO class. It's implemented in a native shared-object library, libJLEG.so, which is loaded into the Java Virtual Machine at runtime using a static initializer in the JLegacyRO class. Static initializers are executed once by the JVM when the class is first loaded. If JLegacyRO doesn't load the native library, an UnsatisfiedLinkError exception is thrown when getN is called. Failure to load libJLEG.so is established only by catching one of the exceptions thrown by System.loadLibrary. The JVM qualifies the library name, assigns the prefix lib and appends the library extension .so for UNIX and .dll for Microsoft Windows.



JLegacyRO calls getN and returns the JLegacy object returned by it to implement the method defined by JLegacyIF. Nothing to it, right? Well, let's finish the JLegacyRO class before we call this one complete.

The JLegacyRO class exports itself by extending UnicastRemoteObject and calling the constructor of its superclass in its own constructor. In addition, UnicastRemoteObject redefines the equals, hashCode and toString methods inherited from java.lang.Object for remote objects.

The first thing the main method provided by the JLegacyRO class does is install RMISecurityManager to protect its resources from remote client stubs during transactions. The RMISecurityManager is the equivalent of the applet security manager for remote object applications. Next, the main method creates an instance of the JLegacyRO class and a remote object registry listening on a port number, which is declared static final. The JLegacyRO class is the only application that will use this registry. Finally, the main method binds the instance of the JLegacyRO class to a unique name in the remote object registry, making the object available to clients on other virtual machines. The name bound to the object is formed using the port number and the name of the remote object's host, which is passed to the application as a command line argument and the String "JLegacyRO".

Before delving into the details of the native method, let's look at the last class – the client-side class that invokes the method on the remote object, JLegacyC (see Listing 5). JLegacyC provides a constructor without parameters, which is never intended to be called, and a static method, get, which looks up the remote object in the registry created by JLegacyRO. This static method also retrieves a reference to JLegacyIF through which the remote method, getJLegacy, is invoked. The get method returns the JLegacy object that was returned by the remote method invocation.

These three classes and the interface are all compiled into the same package. All classes, including the stub and skeleton created from the JLegacyRO class using the rmic compiler, are served from the Indy Web server. The environment settings are explained at the conclusion of this article. The native method is also relatively straightforward.

Before we can discuss the details, however, we must establish its C prototype. The C header file, which defines the prototype for the native method, is generated using the javah tool with the -jni option on the compiled JLegacyRO class (see Listing 6). Since the JLegacyRO class has been compiled into a package, the package name must be appended to the class name when javah is executed (e.g., javah -jni my.jlegacy.classes.JLegacyRO). The resulting header file will be prefixed with the package name (e.g., my_jlegacy_classes_JLegacyRO.h).

If you've read the Java Native Interface specification, you're already familiar with the method used by javah in composing native method names. If you haven't, I must warn you it's not pretty. A native method name has the following signature: Java_<mangled fully qualified class name>_<mangled method name>. The term *mangled* is actually used in the JNI specification. If the native method is an overloaded method, the name is further appended with __<mangled argument signature>. There's that word again. For further information on the JVM's type signatures, I recommend reading the JNI specification.

The JNI interface (or JNIEnv) pointer is always the first argument to a native method. The interface pointer points to a table of function pointers, each a JNI function. In standard C, all JNI functions are called via this pointer (e.g., (env)->FindClass(env,"java/lang/String")). The JNIEnv structure is defined in C++ with inline functions that ultimately resolve to the same references as the standard C functions. Since the sole purpose of the JNIEnv pointer is to invoke the JNI functions, and because it has a well-defined syntax, I wrapped all the JNI functions so as to promote greater readability and easy maintenance.

The second argument to a native method varies depending on whether or not the method is declared static. If the method is nonstatic, the argument is type jobject and is a pointer to the Java object that invoked the method. If the method is declared static, the argument is type jclass and is a pointer to the Java class that declared the method (i.e., the remote object class JLegacyRO). Any arguments passed to the native method in its Java declaration follow the second argument in the function prototype. In this case the method is declared with no arguments.

Remember about getN being declared as returning an instance of the JLegacy class? This is the jobject returned by the function in the C prototype. Briefly, the native method will retrieve the required data using the existing legacy function, instantiate the jobject to be returned and populate it with the retrieved data (see Listing 7).

> ***"The purpose of the remote method is to return an instance of a class object whose contents reflect the data structure returned by the legacy function"***

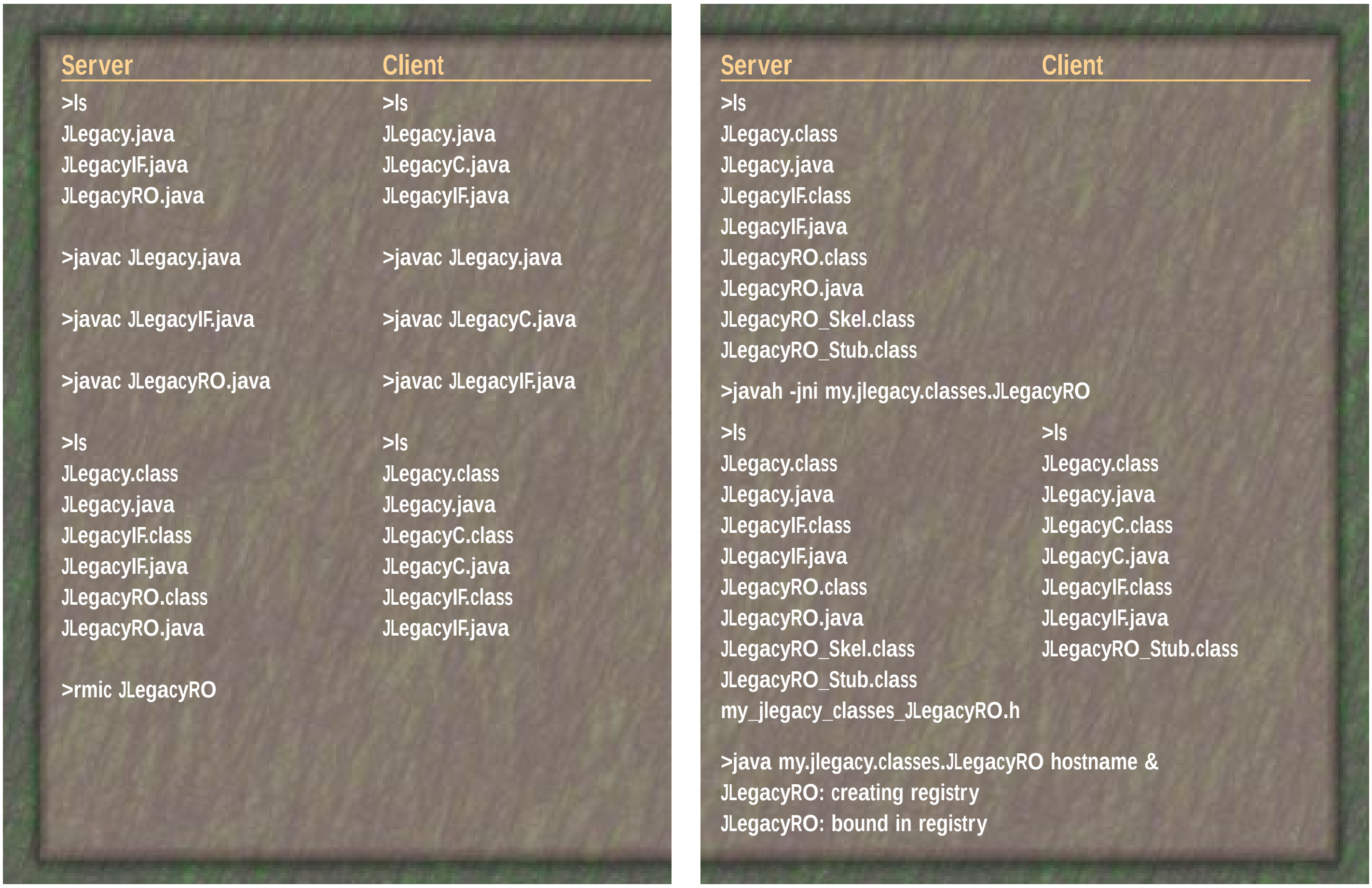
First the native method calls Get_Legacy_Data, passing it a pointer to the Legacy_Type structure to be populated. Then the fun begins. Using the JNI AllocObject function, the native method allocates an object of the JLegacy class. The jclass must be established first, using the JNI FindClass function, because the native method is declared static in the JLegacyRO class. This means that the jclass argument passed to it isn't the class to which an object is to be allocated. The FindClass function requires a fully qualified class name (i.e., my/jlegacy/classes/JLegacy).

The JLegacy object is an example of a local reference, meaning its scope is for the lifetime of the native method and it's automatically freed by the JVM upon return. All objects passed into or returned from native methods are local references. Global references remain visible until they're freed.

Once the JLegacy object is returned, the native method must establish the field IDs for the public instance (nonstatic) variables within the Java object in order to access the variables or fields. Fields are identified by the JNI, using their symbolic names and type signatures.

Finally, the instance fields are set to the contents of the Legacy_Type structure returned by Get_Legacy_Data using the JNI Set<type>Field family of accessor routines, and the populated JLegacy object is returned to the interface implemented by JLegacyRO. Former C programmers should note that the Set<type>Field routines are provided only for the following primitives: boolean, byte, char, short, int, long, float and double; everything else is an object of some sort.

In this case a series of the members in the



```
Server                          Client
>ls                             >ls
JLegacy.java                    JLegacy.java
JLegacyIF.java                  JLegacyC.java
JLegacyRO.java                  JLegacyIF.java

>javac JLegacy.java             >javac JLegacy.java

>javac JLegacyIF.java           >javac JLegacyC.java

>javac JLegacyRO.java           >javac JLegacyIF.java

>ls                             >ls
JLegacy.class                   JLegacy.class
JLegacy.java                    JLegacy.java
JLegacyIF.class                 JLegacyC.class
JLegacyIF.java                  JLegacyC.java
JLegacyRO.class                 JLegacyIF.class
JLegacyRO.java                  JLegacyIF.java

>rmic JLegacyRO
```

```
Server                          Client
>ls
JLegacy.class
JLegacy.java
JLegacyIF.class
JLegacyIF.java
JLegacyRO.class
JLegacyRO.java
JLegacyRO_Skel.class
JLegacyRO_Stub.class

>javah -jni my.jlegacy.classes.JLegacyRO

>ls                             >ls
JLegacy.class                   JLegacy.class
JLegacy.java                    JLegacy.java
JLegacyIF.class                 JLegacyC.class
JLegacyIF.java                  JLegacyC.java
JLegacyRO.class                 JLegacyIF.class
JLegacyRO.java                  JLegacyIF.java
JLegacyRO_Skel.class            JLegacyRO_Stub.class
JLegacyRO_Stub.class
my_jlegacy_classes_JLegacyRO.h

>java my.jlegacy.classes.JLegacyRO hostname &
JLegacyRO: creating registry
JLegacyRO: bound in registry
```

Table 1: Compilation steps on the client server platforms

Legacy_Type structure returned by Get_Legacy_Data are char arrays or UTF-8 format in Java. The UTF-8 format encodes nonnull ASCII characters in the range 0x01 to 0x7F (hexadecimal) in a single byte. Characters above 0x7F are encoded using up to 3 bytes of storage. The JNI SetObjectField function requires a native type for the value of the indicated field, so the char arrays must be converted to java.lang.String objects before their instance fields can be set in the Java object. This translation may be performed using the JNI NewStringUTF function. Since a series of these instance fields have to be set, the steps needed to do this are generalized into another function, JL_SetStringField.

If an error condition arises during execution of the native method, the method will delete the local reference that the JLegacy object pointed to and then return a null object to the interface implemented by JLegacyRO. Freeing the local reference is a habitual practice of mine when I write C code, though it's not required in Java; I just think it's good programming style.

Now let's make everything talk to each other. First let's discuss compiling getN into the native shared-object library, libJLEG.so. In the makefile for libJLEG.so, legacy.so must be supplied as an argument to the link editor in order to resolve the symbol supplied by Get_Legacy_Data's object module for getN. In addition, Java 3.1 (Sun 1.1.5) assumes the runtime linker to load n32 libraries. If you attempt to load an o32 native library from the JLegacyRO class, a fatal error will be returned by rld. It can't successfully map the shared object name to the LD_LIBRARY_PATH despite the presence of the native library located at a path specified by the environment variable.

To facilitate loading an o32 library, two options are available. The first is to set the environment variable SGI_ABI to "-32" before starting JLegacyRO. The second is to pass the "-32" argument to the Java interpreter when starting JLegacyRO. On the Indy Web server the LD_LIBRARY_PATH variable must include the path for libJLEG.so and legacy.so, as well as <yourJAVA_HOMEpath>/lib/sgi/green_threads.

Apparently the JVM for the Silicon Graphics platform uses the default Green threads package as its user threading model. The Green threads package maps all Java threads into a single native thread, prohibiting concurrent execution of multiple threads in a Java application. In addition, the CLASSPATH variable on the Indy Web server must include the path that precedes the directory structure, defined by the package the classes were compiled in, so the Java interpreter can locate them. Finally, the applet class is served from the Indy Web server by setting the CODEBASE attribute accordingly in the HTML file.

I hope this article answers more questions than it raises. I know I learned a lot while working on this task and even more while writing about it. I hope you did, too.

Although all of these classes were served from a single Indy Web server, a summary illustrating the client and server classes running on different platforms might be useful to make clear on which platform each class belongs and each command-line step takes place (see Table 1). In this context *client* refers to the process (i.e., applet) invoking a method defined by a remote object and server refers to the remote object process. The rmic compiler is used on the server platform to create the stub and skeleton classes; the stub class is copied to the client platform before runtime. In addition, javah is used on the server platform to generate the header file that defines the C prototype for the native method declared by the remote object class; development of the source file that implements the C function is left to the user. The make of the native shared-object library on the server platform isn't illustrated, nor is browser startup on the client platform.

About the Author

Scott Howard, a staff analyst for New Technology, Inc., in Huntsville, Alabama, has developed software for private industry and the aerospace community for 13 years using FORTRAN 77, C and now Java. He's also a contributor to the Enhanced Huntsville Operations Support Center System Web infrastructure and Java Common User Interface designs at Marshall Space Flight Center. He can be contacted at Scott.Howard@UMS.MSFC.NASA.GOV.

UMS.MSFC.NASA.GOV

RMI: Pure Java Distributed Computing

A database that can be packaged with data and application logic and distributed over the Net

by Christopher Lambert

What Is RMI?

RMI, the acronym for Remote Method Invocation, is part of the core Java API. The central idea behind this technology is the ability to call the methods of a remote object, shielding the programmer from mundane Socket handling while promoting a cleaner software architecture.

Why Use RMI?

RMI allows a developer to create distributed applications while retaining 100% Java compatibility and reducing the overall complexity of a project. By using RMI, the programmer can get an instance of the server object and call its methods directly. By calling the server object's methods, we can avoid the use of large switch statements and proprietary protocols.

Comparing RMI to Sockets

RMI is actually an abstraction layer built over Sockets.

1. It uses Sockets to communicate data over the network.
2. It allows for persistent/stateless connections.
3. It uses Serialization to transport objects over streams.

Advantages of RMI

A primary advantage is simplicity and clean implementation, leading to more maintainable, robust and flexible applications. This isn't to say a system can't be written using Sockets in place of RMI, just that RMI removes a great deal of mundane tasks -- such as parsing and switch logic. Since RMI has the potential to reduce a great deal of code, more complex systems can be built with relative ease. The greatest benefits don't revolve around ease of use, however.

RMI allows us to create a distributed system while at the same time decoupling the client/server objects. RMI isn't the first API to put these benefits on the table, but it's a pure Java solution for doing so**.** This means that it's possible to create a zero-install client for your users. An example of this may be a Decision Support System (DSS) written as an applet that communicates with a "server interface" object using RMI. This server interface object could be designed to simply call methods from a server object that talks to the database (see Figure 1). By using such an architecture, you can build some extremely powerful applications that are easily maintained (relatively, of course!).

As you can see from Figure 1, a system can use RMI to its advantage in several ways:

1. There's no client installation needed, only a Java 1.1-capable browser (or a JRE for applications).
2. If the DBMS is changed (i.e., if you move to Oracle from Access), then only the server object needs to be recompiled, while the server interface and client remain the same.
3. All portions are easily distributed, and development teams can be given a "section" of the distributed architecture to work on. This simplifies coding and allows a group to leverage its talents better. For example, the GUI "expert" could focus on the client while the DBMS "expert" could focus on the server.

Disadvantages of RMI

RMI is slightly less efficient than Sockets because of the additional "layer" involved and because it must deal with the registry in order to communicate. Another concern is creating multithreaded servers safely; a common mistake is to assume the default threading will allow you to ignore code that ensures our server is thread-safe and robust. If you want to implement a concurrent user system, you'll still need to provide the proper structure for doing so.

How Does RMI Work?

RMI uses a registry to store information regarding servers that have been bound to it. This article uses the rmiregistry provided in the JDK; however, it's possible to write an RMI-based application without it.

Binding is done by calling the Naming.rebind() method in the server object's constructor (found in the java.rmi package). If the method fails, it'll throw one of the following exceptions – RemoteException, MalformedURLException or UnknownHostException. In the case of RemoteException, there was an error with the registry, often occurring because rmiregistry wasn't executed before the server object attempted to bind. Once the server has been bound to the registry, a client can do a Naming.lookup() to get an instance of the RMI server object.

After the client has an instance of the server object, it'll be able to call all the methods defined in the server's list of promised remote methods. These methods are defined in an interface that both the client and server objects implement. By using rmic, we can create a stub and skeleton to use for compiling our client object.

The stub sits in the client's codebase or classpath (the client's .class file usually resides in the same directory). This stub object is what tells the client what methods may be called from the server and handles all the details that allow us to call a remote object's method via the registry.

The skeleton is similiar to the stub, except it must be in the server's classpath (like the stub, the skeleton usually resides in the same directory as the .class file for the server). The skeleton handles incoming requests/parameters from clients and returns the results via the registry.

Figure 2 should help you visualize this whole process. It's important to realize that while we're looking at a one-to-one relationship here, anything is possible. You can have multiple clients associated with a single server or multiple servers, or even allow a client to be a server as well. The only requirements are that the stub/skeletons must be available to each object that needs them (client/server) and that the registry must be located on the server's machine (or an alternative must be accounted for).

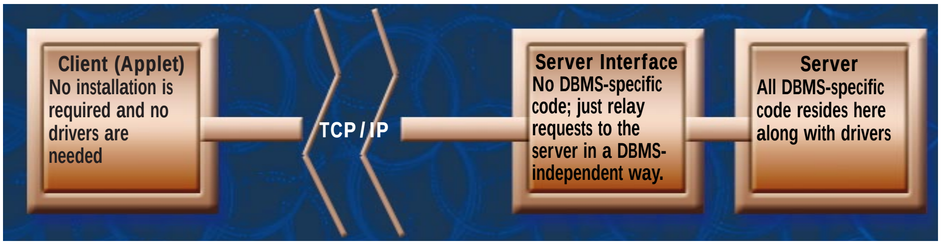

Client (Applet)
No installation is required and no drivers are needed
TCP / IP
Server Interface
No DBMS-specific code; just relay requests to the server in a DBMS-independent way.
Server
All DBMS-specific code resides here along with drivers

Figure 1: How a typical RMI application may look; notice that the client talks through the interface.

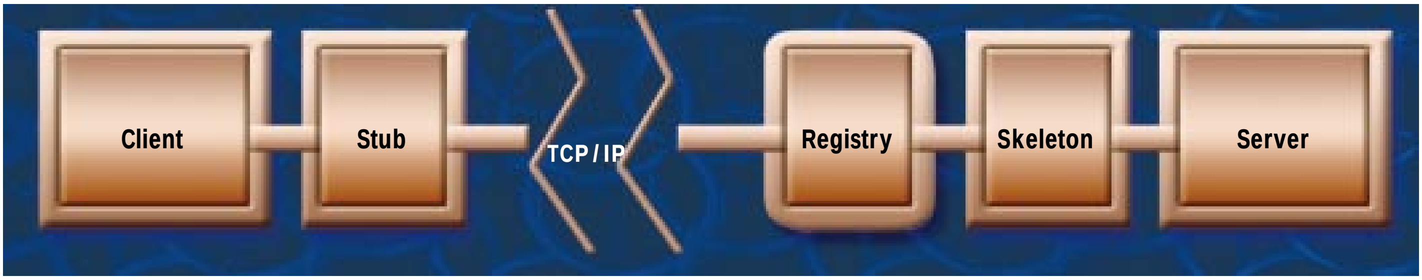


Figure 2: The relationship between each of the components in RMI

Putting It All Together

The following sections outline the steps involved.

1. ***Writing the Interface***

- Create an interface to be implemented by the server class. This interface must contain all public methods, each of which includes a throws RemoteException clause in its definition. This interface must also extend the java.rmi.Remote object. For an example, see Listing 1.

2. ***Writing the Server object*** (please refer to Listing 2.)

- The server must extend UnicastRemoteObject.
- The server must implement the interface.
- To make the server object concrete so that it may be instantiated, write the code and complete each of the methods defined in the interface.
- The security manager should be set, possibly in the driver's main() method or the constructor. To do this, create a new security manager object as an argument to the system classes setSecurityManager method, i.e.:

```
System.setSecurityManager (new
 RMISecurityManager());
```

- Attempt a Naming.rebind() to bind the server to the registry in the driver's main() or the constructor. To use Naming.rebind(), you must specify the server's name and pass through an instance to the server object; for example, the main() approach, i.e.:

```
MyServer myserver = new MyServer();
     Naming.rebind ("myserver", myserver);
```

3. ***Writing the Client object***

- Call the Naming.lookup() method to get an instance of the interface that the server implements. This interface contains all the promised methods (step 1). See Listing 3.

4. ***Getting things running***

- Compile the server object and use rmic to create the stub and skeleton .class files:

```
javac MyServer.java
rmic MyServer
```

- Compile the client object:

```
javac MyClient.java
```

- Run the RMI registry provided with the JDK:

```
rmiregistry
```

- Run your server object:

```
java MyServer
```

- Run your client object:

```
java MyClient
```

What Is Needed?

To access the RMI API, you must include the necessary packages from "java.rmi.*" and its child packages. You must also be using the JDK 1.1.x and not JDK 1.2.x, as 1.2 requires other considerations.

RMI with JDK 1.2.x vs JDK 1.1.x

Please note that, while in JDK 1.1.x, you simply need to create a new instance of RMISecurityManager to set your server with the appropriate permissions. JDK1.2 is a little different. Thankfully, the changes necessary aren't a big deal: simply write your own MySecurityManager class that inherits from SecurityManager and implement the following method (minimum):

```
public void checkPermission (Permission p)
   {
    return;
   } //checkPermission
```

Be aware that this probably isn't something you'd want to use for a corporate app; however, for learning RMI with JDK 1.2.x it works fine. Just so you know, what's happening here is that the checkPermission() method will throw a security exception if a SecurityManager object doesn't allow certain requests. Otherwise, if the request is okay, it returns (just as above). So the above method allows everything – please read up on Java security issues before doing corporate applications that may require (a little) more stringent security!

An Example Program Using RMI

Now that we've discussed RMI and its application, it's time to write a small example application to put RMI to the test. If you have any problems compiling or running this demo, you can e-mail me at chrislambert@pobox.com. I'm using Sun's JDK 1.1.6 to compile and run this application, but if you comment/uncomment where indicated, it will compile/run under JDK 1.2.x.

The example code will compile to create a Server and Client, and should include the Server.java, Client.java and ServerInterface.java source files. To compile and execute the example code do the following:

```
javac Server.java   //compile Server.java file
javac Client.java   //compile Client.java file
rmic Server         //create stub/skeleton
start/min rmiregistry
                    //start up the RMI registry
start java Server   //start up the Server
javaClient server   //start client and connect
                    to "server" (defaul registry
                    name //used by Server
```

Resources

To find out more on how to leverage the RMI API, I've listed some URLs and book titles to help you get started.

1. *Java in a Nutshell – Examples.* ISBN 1-56592-371-5
2. *Beginning Java.* ISBN 1-861000-27-8
3. *Java 1.1 Developers Handbook.* ISBN 0-7821-1919-0
4. http://adams.patriot.net/~tvalesky/easy-rmi.html
5. www.nada.kth.se/javadoc/JDK1.1/guide/rmi/index.html
6. www.javasoft.com/products/jdk/rmi/index.html

Conclusion

This article presents a quick and easy way to get started using RMI by example. I recommend that you refer to the URLs above for more information or consider the books listed. Enjoy using RMI and the best of luck in your programming endeavors!

▼▼▼▼▼▼ CODE LISTING ▼▼▼▼▼▼

The complete code listing for this article can be located at **www.JavaDevelopersJournal.com**

About the Author

Christopher Lambert, a graduate of Lambton College with a computer programmer analyst degree, is currently completing a BS in computer science at the University of Northern British Columbia. He works at Canfor, a software consulting company pecializing in Java/Oracle. He can be reached at chrislambert@pobox.com.

✉ chrislambert@pobox.com

WIDGET FACTORY

MORE ON Syntax Highlighting

Adding more advance features, including highlighting for strings, comments and numbers

by **Jim Crafton**

Last month we discussed the use of Swing's Document model to create a syntax-highlighting Document model that we could just plug into JTextPane and use. This month we'll continue with that and add complete support for comments, strings and numbers. We'll also cover how easy it is to actually use the model we've developed, and test things out as we go along.

To start things out let's try and use what we have so far. If we look at the code below we can see that plugging the Document we've created into a JTextPane component is quite easy. Defining the keywords is also very easy.

```
JTextPane editor = new JTextPane();
CodeDocument doc = new CodeDocument();
Vector keywords = new Vector();
keywords.addElement("abstract");
keywords.addElement("boolean");
...
doc.setKeywords(keywords);
  editor.setDocument(doc);
```

We could have just created the CodeDocument on the fly and passed it in as an argument to the constructor of the JTextPane class, but then the keywords for the Document wouldn't have been set and we'd have had to retrieve the Document and then add them.

By running the code above and typing in some text, we get the window in Figure 1 to come up.

A word of caution: If you use this and the first word you type in is in your keywords list, you'll get an exception. This is a known bug on the Swing Web site (for more information look at http://developer.java.sun.com/developer/bugParade/bugs/4128967.html) with no known workaround. You may also notice that when you type for a while and then move the cursor back to a previous position and start typing again, the JTextPane doesn't quite repaint itself correctly. This is also a known bug of the JTextPane (for more information see http://developer.java.sun.com/developer/bugParade/bugs/4127974.html). This has been fixed in Swing version 1.1.

Now you can test the modifications we'll be making to the CodeDocument class.

Changing the insertString Method

Some of you may have noticed that the Document Model code we developed in the previous article worked only if you actually typed in the text manually. If we had called the insertString method and passed in a string with a length longer than one character, the syntax highlighting wouldn't have worked. To fix that we'll make a simple change, moving the code that actually did the comparing to another method, and changing the insertString method to support any size string. We'll create a method called processChar that will actually do the work of checking the kind of character we're dealing with and, in turn, call the correct method to handle that character. To make the whole thing work, we can then just loop through each of the characters in the string that's passed into the insertString method, starting at the offset position (the offs variable passed into the insertString method) and ending at the offset plus the length of the string passed in. For each character we find we can call the processChar method, and *voila!* We can now handle any size string. To get an idea of the code, take a look below.

```
public void insertString(int offs,
                         String str,
                         AttributeSet a)
throws BadLocationException{
    super.insertString(offs, str, normal);

    int strLen = str.length();
    int endpos = offs + strLen;
    int strpos;
    for (int i=offs;i<endpos;i++){
      currentPos = i;
      strpos = i - offs;
      processChar(str.charAt(strpos));
    }
    currentPos = offs;
  }
```

The only problem with the solution above is that when reading large chunks of text you may notice a slowdown in JTextPane's performance while it loads and parses all the text. A possible solution would be to use a thread to handle any text not immediately visible to the user, parse it in the background and then...well, that'll be the subject of a future article!

Adding String Support

Now that we can support entered text of any size, let's add highlighting support for strings. Adding string support isn't too hard now that we have the basic pieces in place. We'll add a variable that we'll use to keep track of whether we're entering a string. If this flag is turned on, any text we enter following the double quote ("") will be colored in a different foreground color. Also, if this flag is turned on, no keyword processing will occur. A carriage return will automatically shut the flag off. We'll need to keep track of the start position where the flag was first turned on. This will allow us to type in some text representing a string, hit the carriage return and type other stuff, and then put the caret back on the line where the string was and resume entering the string text – all the while keeping the formats correct. We'll also need a variable that represents the style attribute we want for strings.

In addition to variables we'll need some new methods. For strings we'll create a checkForString() method that will determine whether we're inside a string. If it finds that we are, the mode variable will be set appropriately (this will be explained in more detail a few paragraphs down). We'll also need a method called insertTextString() that will

actually reinsert the properly formatted string.

Adding Number Support

Adding support for numbers is similar to adding support for strings. We'll add a variable to hold the style attribute for numbers, and also add two more methods: checkForNumber() and insertNumberString(). Like strings, the checkForNumber() method determines whether we're actually entering valid digits. If we are, it sets the CodeDocument's mode accordingly. Like insertTextString, insertNumberString inserts the properly formatted text as a number.

Adding Comment Support

Comments are handled in a similar manner. Two new methods are needed, checkForComment() and insertCommentString(). Like numbers and text, the checkForComments method determines if a comment block has been started (it checks for the "/*" combination to start a comment and the "*/" to end a comment block). If it has, it changes the attributes of the entered text by calling the insertCommentString() method. Again, like numbers and strings, the insertCommentString() changes the formatting attributes of the string accordingly, and then inserts into the document. Another private variable is needed to hold the formatting attributes for comments.

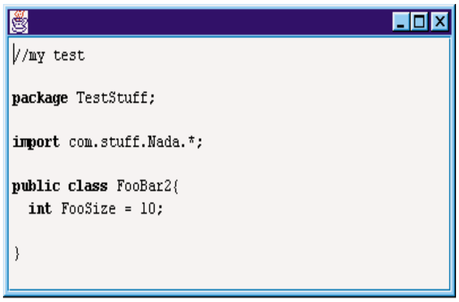
Putting It All Together

Now it's time to get down and dirty. We're finally going to look at the whole process, first in general terms and then in more detail using code examples. As we mentioned earlier, the original insertString method of the CodeDocument class was changed, and much of the logic was moved to a new method called processChar(), thus allowing us to handle not only single keystrokes, but also to programmatically insert multiple character strings. ProcessChar() works by making certain assumptions about what characters will follow other characters. Based on this, it sets the insert mode for the current position of incoming text. Based on this mode (which is simply a private integer variable), it can then determine which insertXXXString() method to call. For example, when it encounters the character '9', it figures there's a very good chance that this is the start of a number or that a number is currently in the process of being entered. To verify this it calls the checkForNumber() method. If checkForNumber() determines that we're still entering a number, it sets the mode to the number entry mode. If it discovers any other character present, however, such as a space, a parenthesis or a letter, it sets the mode to the default text entry mode (the number entry mode is represented by the static constant NUMBER_MODE, while the text entry mode is represented by the static constant TEXT_MODE). The other checkForXXX() methods work in a similar fashion. After the checkForXXX() method has returned, the mode will have been set correctly and, based on this, the proper insertXXXString() method can be called. If the mode is in TEXT_MODE, the formatting is left alone.

Now look at the code in Listing 1 for method processChar. The first thing the method does is to check if we're in COMMENT_MODE. Because comments can include anything (aside from the comment-terminating characters), we'll let the mode default to COMMENT_MODE; otherwise we'll change it to default to TEXT_MODE, which is the standard text entry mode (no formatting). Next, a switch statement is created based on the character passed to the processChar method. As mentioned before, the method works on the assumption that the character being entered belongs to one of five groups (generic text, keywords, strings, numbers or comments), and that the case statements are grouped accordingly. Characters equaling the numbers '1' through '9' cause a check for numbers; characters equaling '*' or '/' suggest the possible start or end of a comment block, and a check is made; and so on. Once the switch statement is finished executing, a final check is made for quoted strings if we're still in TEXT_MODE. Finally, depending on which mode we're in, a call is made to the appropriate insertXXXString method.

There are four check methods: checkForComment(), checkForKeyword(), checkForNumber() and checkForString(). While there are some differences from the original checkForKeywords() method, the basic idea, as discussed last month, is the same. We retrieve the current element, get the text from it, find our position in the element and then, starting at the end, we walk backward until a delimiter of some sort is found and then set the mode accordingly. Let's look at some of the checkForString() method's code to examine this more closely. We'll assume we already have the correct offset from the Document's element.

```
    ..
  ..
  int quoteCount = 0;
    if ((offs >= 0) && (offs <= strLen-1)){
      i = offs;
      while (i >0){
      //the while loop walks back until we hit a
delimiter

        char charAt = elementText.charAt(i);
        if ((charAt == '"')){
         quoteCount ++;
        }
        i--;
      }
      int rem = quoteCount % 2;
      mode = (rem == 0) ? TEXT_MODE:
STRING_MODE;
    }
```



```
//my test

package TestStuff;

import com.stuff.Nada.*;

public class FooBar2{
  int FooSize = 10;

}
```

Figure 1

The code is fairly simple: we loop backward until we run out of characters to process, each time comparing a character from the element text. If the character is a double quote (" "), we add one to the local variable called quoteCount. Once we're done with the loop, we check the remainder of the quoteCount divided by two and assign the results to another local variable called rem. Why do this? Again, this is another assumption about the way words are put together. Since quotes always come in pairs, if our remainder isn't equal to zero, we know we're "inside" a string quotation. Otherwise we can safely assume that we're "outside" and are just entering normal text.

And Finally...

I think at this point I've probably run out of column space. We now have a syntax-highlighting Document class that supports any user-defined set keywords, plus string, number and comment highlighting (see Listings 2 and 3). We've also seen how easy it is to incorporate our Document class into the JTextPane control to test our results as we go along. In the next article we'll add some more user-definable properties so you can change the color of highlighted string, comments, keywords and so on. We'll also look at adding support for a kind of "smart editor" à la Borland's Code Insight features in their IDE editors.

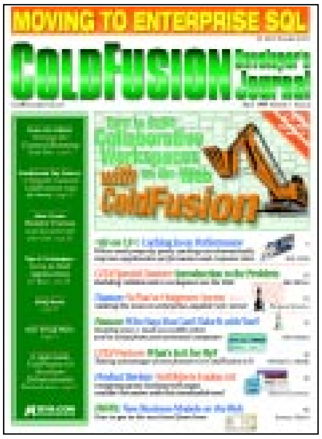
References

1. Topley, K. (1998). *Core Java Foundation Classes.* Prentice Hall PTR. Prentice-Hall Inc.
2. Eckstein, R., Loy, R, and Wood, D. (1998) *Java Swing.* O'Reilly and Associates Inc.

About the Author

Jim Crafton is a staff consultant with Computer Sciences Corporation where he specializes in object-oriented development. He also develops advanced graphics software for Windows and the BeOS. He can be reached at ddiego@one.net and has a Web site at www.one.net/~ddiego/.

ddiego@one.net


MOVING TO ENTERPRISE SQL
COLDFUSION Developer's Journal
How to Build Collaborative Workspaces on the Web with ColdFusion

LISTING1

```
private void processChar(String str){
  char strChar = str.charAt(0);
  if (mode != this.COMMENT_MODE){
   mode = TEXT_MODE;
  }
   switch (strChar){
    case ('{'):case ('}'):case
(' '): case('\n'):
    case ('('):case (')'):case
(';'):case ('.'):{
     checkForKeyword();
     if (mode == STRING_MODE &&
strChar == '\n'){
      mode = TEXT_MODE;
     }
    }
    break;
    case ('"'):{
     insertTextString(str, cur-
rentPos);
     this.checkForString();
    }
    break;
    case ('0'):case ('1'):case
('2'):case ('3'):case ('4'):
    case ('5'):case ('6'):case
('7'):case ('8'):case ('9'):{
     checkForNumber();
    }
    break;
    case ('*'):case ('/'):{
     checkForComment();
    }
    break;
   }
   if (mode == this.TEXT_MODE){
    this.checkForString();
   }
   if (mode == this.STRING_MODE){
    insertTextString(str, this.cur-
rentPos);
   }
   else if (mode ==
this.NUMBER_MODE){
    insertNumberString(str,
this.currentPos);
   }
   else if (mode ==
this.COMMENT_MODE){
    insertCommentString(str,
this.currentPos);
   }

 }
```

CODE LISTING

The complete code listing for this article can be located at **www.JavaDevelopersJournal.com**

Enterprise Database Access with JDBC 2.0

A look at the key features of version 2.0

by **Prasad Thammineni & Vasu Ramachandriah**

Java is the fastest-growing programming language today. The main reason this object-oriented language is so popular is that it's simple, easy to learn and portable.

Java has several core APIs, one of which is the JDBC API. JDBC is based on the X/Open SQL Call Level Interface (CLI) – the basis of ODBC. JDBC gives Java developers a common API to access most databases. This includes relational databases such as Oracle, DB2 UDB, Sybase and Informix as well as legacy databases like IMS. JDBC is used mainly to create *n*-tier client/server database applications or Web-enabled database applications.

The JDBC API allows developers to easily write applications that access data from a database. JDBC API users aren't required to understand low-level database-related functions like memory management and byte alignment. Not only is JDBC easy to use, it also gives programmers a powerful set of APIs they can use to quickly build sophisticated and real-world applications.

As the first in a multipart series focusing on the enterprise features of JDBC 2.0, this article explores several key features introduced in JDBC 2.0.

JDBC 1.0

Sun announced JDBC 1.0 in February of 1996. Since then, database vendors and independent software vendors (ISVs) have implemented JDBC drivers that conform to its specifications. If you needed to access data from an IBM DB2 UDB or Oracle database, you had to use the database vendor-supplied JDBC driver or a third-party driver like INTERSOLV's DataDirect SequeLink Java Edition.

As the number of Java database applications grew, application developers found the features available in JDBC 1.0 inadequate. To implement the desired database functions, application developers had to write a lot of code. For example, JDBC 1.0 supports retrieving records only in the forward direction. To develop an application allowing end users to scroll database records in both directions, the developer, using a JDBC driver, had to cache all the records as they were retrieved from the database locally on the client side. The application quickly became even more complex if it had to support modification of these records. Developing and maintaining these features, which should have been supported by JDBC 1.0, resulted in an unnecessary burden on application developers.

To make up for the deficiencies in JDBC, development tool and JavaBean vendors developed commercial products to support these features. Examples include IBM VisualAge for Java's Data Access Beans and Specialized Software's ROAD:BeanBox.

JDBC 2.0

With the introduction of the JDBC 2.0 API and its rich set of new features, developers can now concentrate on the overall development of applications, e.g., implementing business logic rather than writing nonbusiness-specific database functionality. Some of the new features include support for bidirectional result sets, batch updates, connection pools and connectionless result sets.

JDBC 2.0 is fully compatible with JDBC 1.0. Applications developed using JDBC 1.0 are upwardly compatible and don't require any programming changes. All interfaces and classes found in JDBC 1.0 are present in JDBC 2.0.

The JDBC 2.0 API consists of two main components from Sun: the Core API and the Standard Extension. The Core API can be found in the java.sql package and the Standard Extension API in the javax.sql package.

Release 2.0 for the JDBC Core API has many new features, including:

- Scrollable result sets
- Result sets that can be updated
- Batch updates
- SQL3 data-type support (SQL types ARRAY, BLOB, CLOB, DISTINCT, STRUCT and REF)
- Custom mapping of SQL3 user-defined types to Java classes
- Storing of Java objects in an object-relational database

The JDBC 2.0 Standard Extension introduces a wide variety of new features that address the needs of enterprise application developers. Using this API you can:

- Locate and maintain database connections using Java Naming and Directory Interface (JNDI) and DataSource objects.
- Use connection pooling to pool and share a set of database connections between a larger set of end users.
- Implement distributed transactional applications.

Overview of New Features in JDBC 2.0

Result Set Enhancements

JDBC 1.0 API provided result sets that scrolled only in a forward direction. Once a result set was created, users could only access information one record at a time. With the introduction of scrollable result sets in JDBC 2.0, you can now create applications that let you scroll both forward and backward through the contents of a result set. In addition, scrollable result sets allow for relative and absolute positioning. For example, it's now possible to move directly to the tenth row in a scrollable result set, or to the fifth row following the current row. These result sets can be updated as well.

Result Set Types

The JDBC 2.0 API supports three result set types: forward-only, scroll-insensitive and scroll-sensitive. They all support scrolling in one form or another, but differ in their ability to display changes while they are open.

A forward-only result set allows you to move "forward" in the rows returned. This can be one of the lightest-weight cursors you can build. Depending on the JDBC driver's implementation, a forward-only result set may take up the least amount of client-side resources and could dramatically improve the performance. Forward-only result sets are best suited for Web-enabled database applications where users are using a Web browser to query data.

A scroll-insensitive result set is generally not sensitive to changes made while it's open. When you create such a result set, you get a snapshot of the underlying data. The rows, order and column values are fixed when the result set is created. A scroll-insensitive result set is not your best choice for data that's constantly changing. However, this choice makes a lot of sense when you're accessing data from tables that contain values not likely to change.

A scroll-sensitive result set is sensitive to changes made while it's open and provides a dynamic view of the underlying data. For example, if you're viewing data from a table using a scroll-sensitive result set and somebody else

makes changes in the underlying values, the changes are made visible to you. Driver vendors typically implement this feature by constantly reexecuting the query used to generate the result set. Because of this repetitive activity, dynamic cursors are expensive to implement and are comparatively slow. This type of result set is best suited for applications that need to display the latest data.

Concurrency Types

A result set can have one of two different concurrency types: read-only or updatable. A result set that uses read-only concurrency doesn't allow updates of its contents, and, since locks aren't placed on read-only database records, the overall concurrency of transactions is increased. A result set that's updatable allows updates and may use write-locks to mediate access to the same data item by different transactions. Since only one write-lock can be held on a database item, this can reduce concurrency. Alternatively, you could use optimistic concurrency control if you think conflicting access to the data will be rare.

Tuning Data Access Performance

You can improve the performance of your application by indicating to the JDBC driver how you intend to use the data being accessed. One way to tune data access is to use the FetchSize property of the statement. This allows you to specify the number of rows to be fetched from the database each time more rows are requested. Instead of making a round-trip for a single row, the driver fetches FetchSize rows and works on these rows in memory. The moment your code steps outside this subset of rows, the driver makes a trip to the database to fetch a new set of FetchSize rows. You can improve the responsiveness of the query being executed by fine-tuning this property depending on your application needs. You should also remember that if you specify a large value for FetchSize property, data on the client could get stale very quickly. You can further fine-tune the performance of a JDBC driver by specifying the direction for processing the rows – forward, reverse or unknown. By setting these properties you can dramatically improve the performance of your applications. These two hints are just suggestions, and the driver can choose to ignore them.

Creating a Result Set

The following code example creates a scrollable result set that's sensitive to updates. The FetchSize property has been set to 50, meaning 50 rows of data will be fetched at a time. Note that we have specified the result set will be updatable by setting the concurrency type to CONCUR_UPDATABLE.

```
Connection con = DriverManager.getConnection
("jdbc:subprotocol: sampleDB");

PreparedStatement pstmt = con.prepareState-
ment ("SELECT * FROM DEPT",
Resultset.TYPE_SCROLL_SENSITIVE,
Resultset.CONCUR_UPDATABLE);
pstmt.setFetchSize(50);

Resultset rs = pstmt.executeQuery();
```

In some instances the actual result set returned might not be the one you wanted. For example, if the query contains a table join and the result set isn't updatable, the JDBC driver may not produce an updatable result set. When this occurs, the driver issues a SQLWarning. You can determine the actual result set type and concurrency type of a result set by calling result set's getType() and getConcurrency() methods, respectively.

Updating a Result Set

A result set is updatable if its concurrency type is set to CONCUR_UPDATABLE. You can update, insert or delete rows of an updatable result set. The example below updates the first row of a result set. The result set's updateXXX() methods are used to modify the value of an individual column in the current row. Calling these methods doesn't update the underlying database. The database is updated only when the updateRow() method is called. Names or numbers can be used to specify columns.

> *"With the introduction of the JDBC 2.0 API and its rich set of new features, developers can now concentrate on the overall development of applications"*

```
rs.first();

rs.updateString(1, "Hello World");
rs.updateFloat("distance", 100000.0f);

rs.updateRow();
```

If you move to another row after modifying individual columns but before you call updateRow(), the changes are discarded. You can explicitly cancel the changes made to individual columns of a row by calling the Resultset.cancelRowUpdates() method. This method should be called after calling the updateXXX() methods and before calling updateRow(); otherwise it has no effect.

The following example deletes the tenth row in the result set from the database.

```
rs.absolute(10);
rs.deleteRow( );
```

JDBC 2.0 introduced the concept of an insert row. The example below shows how to insert a new row into a result set.

```
rs.moveToInsertRow();
rs.updateString(1, "Insert example");
rs.updateFloat("distance", 100.10f);
rs.insertRow();
rs.first();
```

An insert row is associated with a result set and is used as a staging area before it's inserted into the result set itself. To position the result set's cursor on the insert row, you must call the result set's moveToInsertRow() method. Use the result set's updateXXX() and getXXX() methods to update and retrieve individual columns of the insert row. Immediately after moving to the insert row, using the moveToInsertRow( ) method, the contents of the insert row are undefined. Calling the getXXX() method on a column in the insert row immediately after calling the moveToInsertRow() would return an undefined value until the value is set by calling updateXXX() method.

Calling updateXXX() methods on an insert row doesn't affect the underlying database or the result set. For the changes to affect the underlying database, the insertRow() method should be called. When inserting a row, columns must allow null values. For example, if the column in the result set hasn't been assigned a value, or if a column in the result set being inserted isn't present in the underlying table and the columns don't accept null values, the insertRow() method will throw a SQLException.

Though different database implementations can produce either updatable or read-only result sets for the same SQL query, you can generally expect queries that meet the following criteria to produce an updatable result set:

- The query references only a single table in the database.
- The query does not contain any join operations.
- The query selects the primary key of the table it references.

In addition, a SQL query should also satisfy the conditions listed below if inserts are to be performed:

- The query selects all of the nonnullable columns in the underlying table.
- The query selects all columns that don't have a default value.

Moving Around a Result Set

Earlier we said that result sets in JDBC 2.0 support both forward and backward scrolling as well as relative and absolute positioning. In this section we'll discuss these features in more detail.

A JDBC 2.0 result set maintains an internal pointer called a cursor that indicates the row in the result set currently being accessed. A result set cursor is analogous to the cursor on a computer screen that indicates the current cursor position. The cursor maintained by a forward-only result set can only move forward through the contents of the result set.

In the JDBC 1.0 API, the only way to move the cursor was to call the method next(). This is still the appropriate mechanism to use in JDBC 2.0 when accessing rows in the forward direction. JDBC 2.0 also provides additional ways to move the cursor. The new method previous() moves the cursor in the backward direction, one row at a time, toward the beginning of the result set. Both the next() and previous() methods return false when you scroll beyond the last row or above the first row. The following code example loops through all the rows of a result set from first to last, and once it scrolls beyond the last row it loops in the reverse direction until it scrolls before the first row.

```
Statement stmt = con.createStatement(Result
   set.TYPE_SCROLL_SENSITIVE, Resultset.CON
   CUR_UPDATABLE);

Resultset rs = stmt.executeQuery("SELECT
   FIRSTNAME, LASTNAME FROM EMPLOYEES");
// print first name and last name from first
// to last order
while(rs.next()){
 String fname = rs.getString("FIRSTNAME");
 String lname = rs.getString("LASTNAME");
 System.out.println(fname + " " + lname);
}

// print first name and last name in the
// opposite order

while (rs.previous()) {
 String fname = rs.getString("FIRSTNAME");
 String lname = rs.getString("LASTNAME");
 System.out.println(fname + " " + lname);
}
```

Using the methods first(), last(), beforeFirst() and afterLast(), you can move the cursor to the row indicated in their names. The method absolute() will move the cursor to the row number indicated in the argument passed to it. If the number is positive, the cursor moves to the given row number from the beginning. If it's negative, the cursor moves to the given row number from the end. For example, absolute(1) puts the cursor on the first row and absolute(-1) puts the cursor on the last row.

Along with the next() and previous() methods, the reverse() method moves the cursor with respect to the current position. With the relative() method you can specify the number of rows you want to move the cursor from the current position. As in the absolute() method, specifying a positive number will move the cursor forward the given number of rows; specifying a negative number will move it backward a given number of rows. In the following example the cursor moves to the fifth row, then to the second row and finally to the fourth row.

```
rs.absolute(5); //cursor on the fifth row
rs.relative(-3); //cursor on the second row
rs.relative(2); //cursor on the fourth row
```

Other methods are available. For example, getRow(), isFirst() and isLast() can help you position and control the cursor better.

In the next article we'll continue to explore this API and the new features of JDBC 2.0. Meanwhile, for more information you can refer to the JDBC specification document available at Sun's Web site.

About the Authors

Prasad Thammineni is vice president of Java development at Specialized Software, a Java and e-business consulting firm. He can be reached at prasad@specializedsoftware.com.

Vasu Ramachandriah is a Java architect at Specialized Software with more than three years of Java experience. He can be reached at vasu@specializedsoftware.com.

Prasad@ and Vasu@specializedsoftware.com

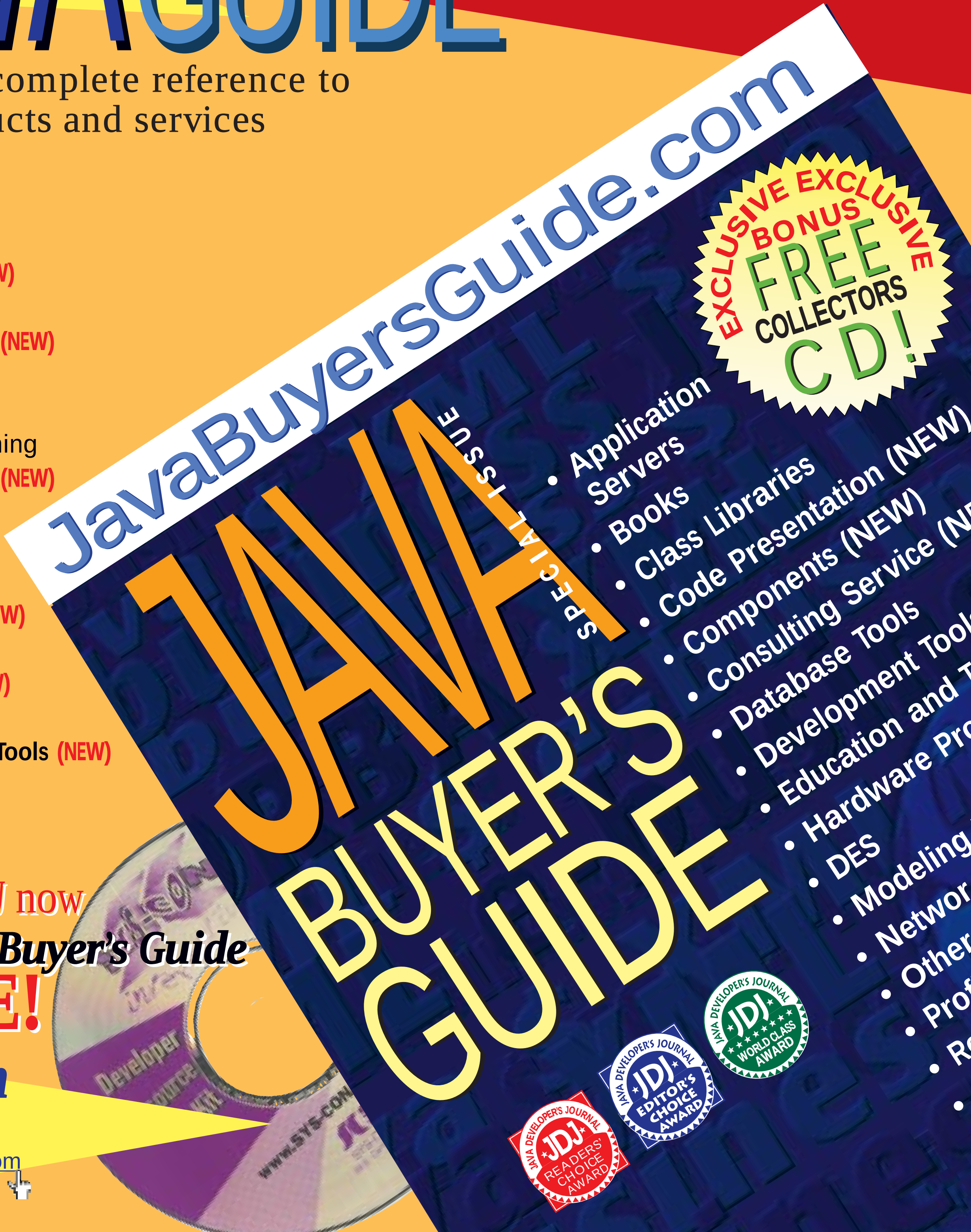
JAVA DEVELOPER'S JOURNAL *presents*

the Annual Print Edition of

JAVA BUYER'S GUIDE

the most complete reference to
Java products and services

- Application Servers
- **Books**
- Class Libraries
- **Code Protection** (NEW)
- Components (NEW)
- **Consulting Services** (NEW)
- Database Tools
- **Development Tools**
- Education and Training
- **Hardware Products** (NEW)
- IDES
- **Modeling Tools**
- Network Tools (NEW)
- **Other Java Tools** (NEW)
- Profilers (NEW)
- **Reporting Tools** (NEW)
- Sites (NEW)
- **Team Development Tools** (NEW)
- Testing Tools
- **Web Tools**

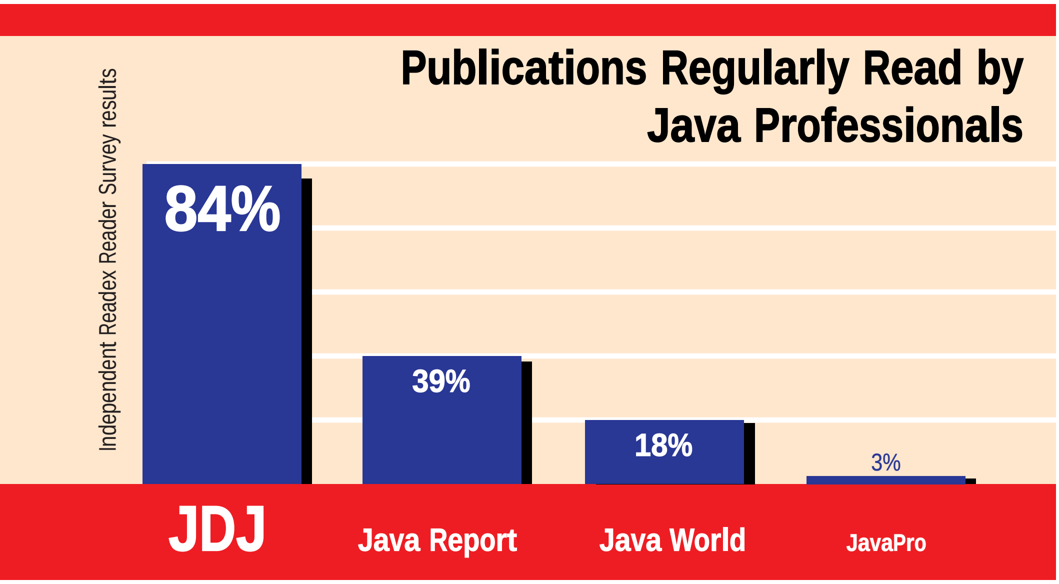


Subscribe to *JDI* now
& receive ***Java Buyer's Guide***
FREE!

Attention
Java vendors!
update your listings at
JavaBuyersGuide.com
for our next print edition.

Only Java Developer's Journal Readers are

100% Pure Java



Before you advertise in a publication, please ask how many real Java readers you're actually reaching!

JDJ is the only publication whose readers are 100% pure Java developers.

Your ad in ***Java Developer's Journal*** reaches 100% Java professionals who make decisions to purchase Java related products and services, not over 40% Visual Basic programmers who never asked to receive the publication you advertise in!

We built our circulation one subscriber at a time.

That's one of our secrets why your ad works in ***JDJ.***

www.JavaDevelopersJournal.com or call *914-735-0300*

Carmen Gonzalez
Vice President,
JDJ Advertising Sales

The World's Leading Java Resource

KL Group Ships JClass 4.0 Enterprise-Ready Java Components

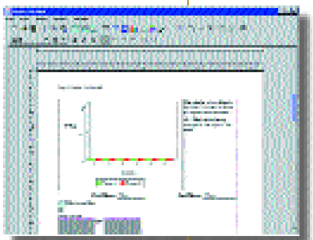
(Toronto, ON) -- KL Group Inc. has released its next generation of 100% Pure Java GUI components, JClass 4.0, featuring a substantially rewritten code base that provides optimized support for Swing and Sun Microsystems’ Java 2. Key features include leaner components, pluggable architecture, new input methods, and HTML and IDE support. www.klgroup.com/jclass.

Blue Sky Software Announces WebHelp3 and Support for Sun’s JavaHelp

(La Jolla, CA) -- Blue Sky Software Corp. has an improved version of its WebHelp solution, WebHelp3, part of the RoboHELP Office 2000 suite. Based on Dynamic HTML, WebHelp3 provides new functionality such as full-text search using Boolean operators, a customizable interface and extremely fast loading. www.blue-sky.com.

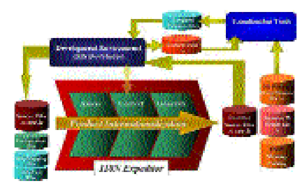
InetSoft Technology Offers Style Report 2.0

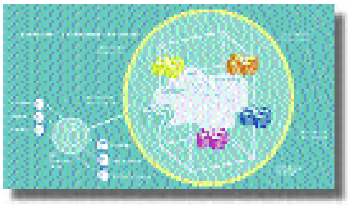
(Piscataway, NJ) -- Building on the success of Style Report 1.x, InetSoft Technology introduces Style Report 2.0 with JBuilder and Visual Café integrators. “Our customers have been increasingly asking for an integrated printing/reporting solution inside IDE. We are excited to meet those demands with Style Report 2.0,” said Luke Liang, the company’s marketing director. www.inetsoftcorp.com.



OneRealm Updates Its Internationalization Tools

(Boulder, CO) -- One Realm, Inc., has announced version 2.0 of its 118n Expeditor and 118n Reporter, which detect and correct software bugs related to internationalization (abbreviated 118n). By eliminating internationalization bugs early in the development process, OneRealm’s new tools reportedly improve the productivity of both software developers and globalization teams. 118n Expeditor is available now for either C/C++ or Java software source code on Microsoft Windows 95, Windows 98 or Windows NT. 118n Reporter is available running on Microsoft Windows 95/98/NT or Sun Solaris 2.6. Site licenses are available. www.onerealm.com





Theory Center Founded

(Boston, MA) -- The Theory Center, Inc., was launched on June 15 to provide EJB component-based solutions that enable Global 2000 companies to quickly turn their e-business visions into revenue-generating reality. The reusable, prebuilt components reduce the development time of new applications, protect an organization’s investment in legacy applications and leverage it to generate new and incremental income through the seamless integration of business functions among partners, suppliers and customers. www.theorycenter.com.

IBM Launches New Online Resource

(Cupertino, CA) -- IBM has announced IBM developerWorks, a fast, free and central online resource that allows developers to tap into the wealth of information, technology and support available from IBM. Each area -- including news, tools and code, standards and education -- is organized into zones focusing on Java technology, XML, security and Web development. A Linux zone is coming soon. www.ibm.com/developerWorks.

BetaBeans Service from Flashline.com Now Available

(Cleveland, OH) -- Flashline.com, a JavaBeans marketplace, announces the immediate availability of BetaBeans, a service for developers that generates third-party feedback on the quality and functionality of submitted JavaBeans. Developers who list a bean as a BetaBean receive exposure to thousands of Flashline’s targeted visitors. The service accelerates the cycle for creating high-quality JavaBeans as suggestions and improvements can be implemented before the commercial release. www.flashline.com.

Sales Vision Deploys Jsales on Linux-Based Operating System

(Charlotte, NC) -- Sales Vision, Inc., announces the availability of Jsales on Red Hat Linux 6.0 running Intel architecture servers and clients. Red Hat, Inc., is a leader in the development of Linux-based OS solutions. Sales Vision is currently the only vendor to offer an enterprise-class CRM solution deployable on Linux. www.salesvision.com.

Intuitive Systems Offers New Tool for Linux Platform

(Sunnyvale, CA) -- Intuitive Systems, Inc., has announced Optimizeit for Linux, the first complete Java performance tool available for the Linux platform. A comprehensive profiling tool, award-winning Optimizeit allows developers to test and improve the performance of Java applications, servlets, applets and JavaBeans whether they’re running on a Linux, Microsoft Windows or Sun Solaris operating system. www.optimizeit.com.

Instantiations Unveils Java Performance Product Family for E-Business

(Tualatin, OR) -- Instantiations, Inc., has introduced the JOVE Super Optimizing Deployment Environment, a product family that extends the company’s offerings further into e-business and high-end Java application solutions. The new products maximize performance and simplify the deployment of the back-end server applications that drive Internet commerce. System performance gains that would normally take weeks or months of engineering work can now be achieved in minutes. The package includes a sophisticated software optimization engine, a native Java compiler, and a scalable runtime platform that enables complex Java applications to be deployed as high-performance, robust, executable files. www.instantiations.com.

ADVERTISING INDEX

ADVERTISER	URL	PH	PG
9NETAVENUE, INC.	WWW.9NETAVE.COM	888.9NETAVE	49
BATEMAN, INC.	WWW.BATEMANINC.COM	805.383.3338	75
BEA WEBLOGIC	WWW.WEBLOGIC.BEASYS.COM	800.817.4BEA	2
BLUE SKY SOFTWARE	WWW.BLUE-SKY.COM	800.559.4423	13
BORLAND.COM	WWW.INTERBASE.COM/PRODUCTS/DEMOJDJ.HTML	800.451.7788 x7183	21
CAREER OPPORTUNITY ADVERTISERS		800.846.7591	76-81
CEREBELLUM SOFTWARE	WWW.CEREBELLUMSOFT.COM	888.862.9898	23
CLOUDSCAPE, INC.	WWW.CLOUDSCAPE.COM	888.59JAVA1	15
COMPUTER ASSOCIATES INTERNATIONAL, INC.	WWW.CAI.COM/ADS/JASMINE/DEV	888.7JASMINE	6
CYRUS INTERSOFT, INC.	WWW.CYRUSINTERSOFT.COM	612.331.6600	39
CYSCAPE	WWW.CYSCAPE.COM/FREE4J	800.932.6869	16
DEVELOPMENTOR	WWW.DEVELOP.COM	800.699.1932	63
ELIXIR TECHNOLOGY	WWW.ELIXIR.COM.SG	65 532.4300	35
ENTERPRISESOFT	WWW.ENTERPRISESOFT.COM	510.742.6700	11
FINDAHOST.COM	WWW.FINDAHOST.COM	440.257.6690	69
HOSTPRO	WWW.HOSTPRO.NET	213.252.9779	57
IMI SYSTEMS INC.	WWW.IMISYS.COM	800.828.0180	61
INETSOFT TECHNOLOGY CORP	WWW.INETSOFTCORP.COM	732.235.0137	47
INSIGNIA SOLUTIONS, INC.	WWW.INSIGNIA.COM	800.848.7677	33
INSTANTIATIONS INC.	WWW.INSTANTIATIONS.COM	800.808.3737	67
JAVA DEVELOPER'S JOURNAL	SUBSCRIBE@SYS-CON.COM	800.513.7111	73-75
JDJ STORE	WWW.JDJSTORE.COM	888.303.JAVA	59
KL GROUP INC.	WWW.KLGROUP.COM/JCLASS/POWER	888.328.9597	52,53
KL GROUP INC.	WWW.KLGROUP.COM/CULPRITS	888.328.9597	84
KL GROUP INC.	WWW.KLGROUP.COM/TRUTH	888.328.9597	25
NSICOM	WWW.NSICOM.COM	97.2.353.31976	63
OBJECT DOMAIN SYSTEMS, INC.	WWW.OBJECTDOMAIN.COM	919.461.4904	37
OBJECT INTERNATIONAL SOFTWARE	WWW.TOGETHERJ.COM	919.772.9350	41
OBJECTSPACE, INC.	WWW.OBJECTSPACE.COM/GO/UNIVERSAL	800.OBJECT1	82,83
OBJCETSPACE, INC.	WWW.OBJECTSPACE.COM	888.59JAVA1	45
ONEREALM, INC.	WWW.ONEREALM.COM/JDJ	800.633.1072 x23839	7
ORACLE CORPORATION	WWW.ORACLE.COM/INFO/32	800.633.1072 x23839	17
PROTOVIEW	WWW.PROTOVIEW.COM	800.231.8588	3
RIVERTON SOFTWARE CORPORATION	WWW.RIVERTON.COM	781.229.0070	29
SALES VISION	WWW.SALESVISION.COM	800.275.4314	51
SLANGSOFT	WWW.SLANGSOFT.COM	972.375.18127	55
SOFTWIRED INC.	WWW.SOFTWIRED.COM	(41)1.445.23101	38
SPECIALIZED SOFTWARE	WWW.SPECIALIZEDSOFTWARE.COM/JDJ/	800.328.2825 x6576	44
SUMMER INTERNET WORLD 99	WWW.INTERNET.COM/REGISTERSUMMER	800.632.5537	65
SUN MICROSYSTEMS, INC.	WWW.SUN.COM/SERVICE/SUNED	800.422.8020	4
THE THEORY CENTER	WWW.THEORYCENTER.COM	888.843.6791	43
TIDESTONE TECHNOLOGIES, INC.	WWW.TIDESTONE.COM	888.880.0665	31
TRAINING/ETC.COM	WWW.TRAININGETC.COM	410.531.9953	69
VISUALIZE INC.	WWW.VISUALIZEINC.COM	602.861.0999	63

Employment Ad

Employment Ad