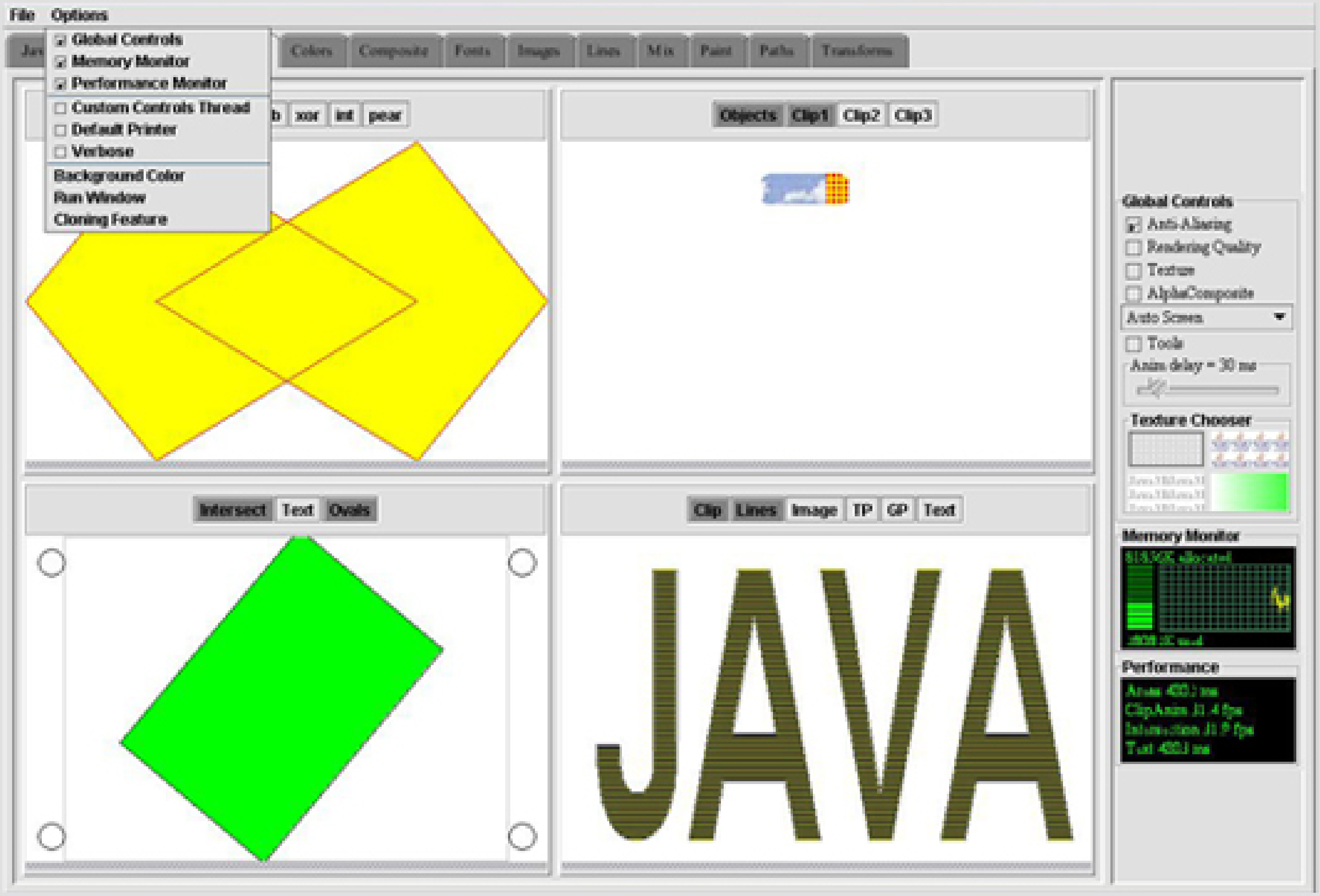
Java Computing Platform & Java Virtual Machines

(Concepts and Applications)



Emilia Naranjo
Lynn Escobar

First Edition, 2012

ISBN 978-81-323-1284-0

© All rights reserved.

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Java Software Platform

Java

The Java technology logo.

Original author(s)	James Gosling
Developer(s)	Sun Microsystems (Owned by Oracle Corporation)
Stable release	Version 6 Update 23 / December 7, 2010; 7 days ago
Operating system	Cross-platform
Type	Software platform
License	GNU General Public License / Java Community Process

Java refers to a number of computer software products and specifications from Sun Microsystems, a subsidiary of Oracle Corporation, that together provide a system for developing application software and deploying it in a cross-platform environment. Java is used in a wide variety of computing platforms from embedded devices and mobile phones on the low end, to enterprise servers and supercomputers on the high end. Java is used in mobile phones, Web servers and enterprise applications, and while less common on desktop computers, Java applets are often used to provide improved and secure functionalities while browsing the World Wide Web.

Writing in the Java programming language is the primary way to produce code that will be deployed as Java bytecode, though there are bytecode compilers available for other languages such as Ada, JavaScript, Python, and Ruby. Several new languages have been designed to run natively on the Java Virtual Machine (JVM), such as Scala, Clojure and Groovy. Java syntax borrows heavily from C and C++, but object-oriented features are modeled after Smalltalk and Objective-C. Java eliminates certain low-level constructs such as pointers and has a very simple memory model where every object is allocated on the heap and all variables of object types are references. Memory management is handled through integrated automatic garbage collection performed by the JVM.

On November 13, 2006, Sun Microsystems made the bulk of its implementation of Java available under the GNU General Public License, although there are still (a very few) parts distributed as precompiled binaries due to copyright issues with Sun-licensed (not owned) code.

Platform

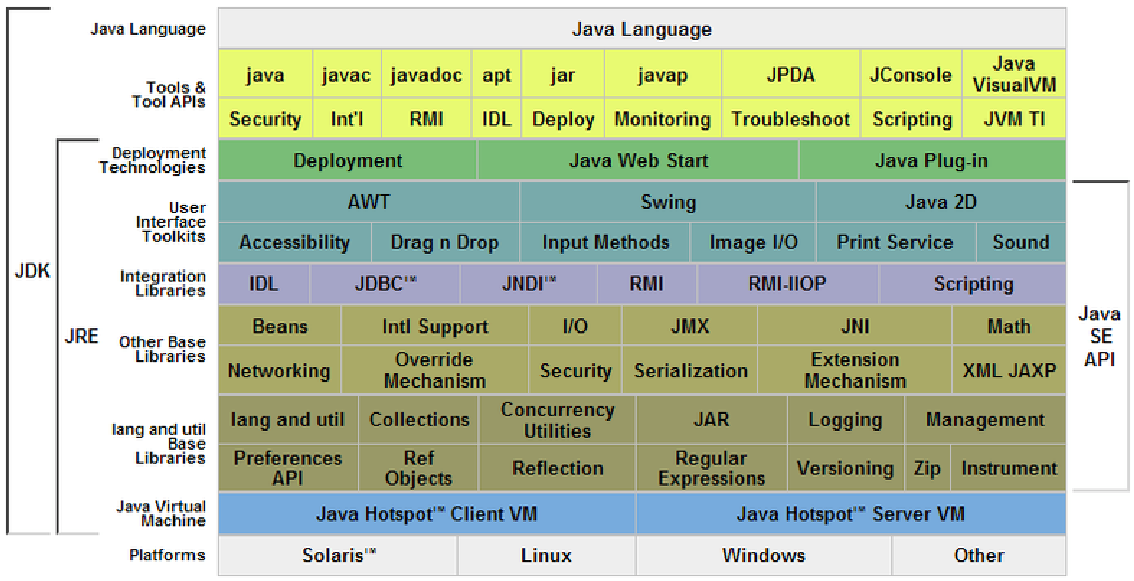
An edition of the *Java platform* is the name for a bundle of related programs from Sun which allow for developing and running programs written in the Java programming language. The platform is not specific to any one processor or operating system, but rather an execution engine (called a virtual machine) and a compiler with a set of libraries that are implemented for various hardware and operating systems so that Java programs can run identically on all of them.

- Java Card: A technology that allows small Java-based applications (applets) to be run securely on smart cards and similar small-memory-footprint devices.
- Java ME (Micro Edition): Specifies several different sets of libraries (known as profiles) for devices which are sufficiently limited that supplying the full set of Java libraries would take up unacceptably large amounts of storage.
- Java SE (Standard Edition): For general-purpose use on desktop PCs, servers and similar devices.
- Java EE (Enterprise Edition): Java SE plus various APIs useful for multi-tier client–server enterprise applications.

As of September 2009, the current version of the Java Platform is specified as either 1.6.0 or 6 (both refer to the same version). Version 6 is the product version, while 1.6.0 is the developer version.

The Java Platform consists of several programs, each of which provides a distinct portion of its overall capabilities. For example, the Java compiler, which converts Java source code into Java bytecode (an intermediate language for the Java Virtual Machine (JVM)), is provided as part of the Java Development Kit (JDK). The Java Runtime Environment (JRE), complementing the JVM with a just-in-time (JIT) compiler, converts intermediate bytecode into native machine code on the fly. Also supplied are extensive libraries, pre-compiled in which are several other components, some available only in certain editions.

The essential components in the platform are the Java language compiler, the libraries, and the runtime environment in which Java intermediate bytecode "executes" according to the rules laid out in the virtual machine specification.



Java Platform diagram from Sun

Languages

The word Java, by itself, usually refers to the Java programming language which was designed for use with the Java Platform. Programming languages are typically outside of the scope of the phrase "platform", although the Java programming language is listed as a core part of the Java platform. The language and runtime are therefore commonly considered a single unit.

Nevertheless, third parties have produced a number of compilers or interpreters which target the JVM. Some of these are for existing languages, while others are for extensions to the Java language itself. These include:

- Clojure
- Groovy
- JRuby, a Ruby interpreter
- Jython, a Python interpreter that include jythonc, a Python-to-Java bytecode compiler
- Rhino, a JavaScript interpreter
- Scala

Similar platforms

The success of Java and its write once, run anywhere concept has led to other similar efforts, notably the Microsoft .NET platform, appearing since 2002, which incorporates many of the successful aspects of Java. .NET in its complete form (Microsoft's implementation) is currently only fully available on Windows platforms, whereas Java is fully available on many platforms. .NET was built from the ground-up to support multiple programming languages, while the Java platform was initially built to support only the Java language (although many other languages have been made for JVM since).

.NET includes a Java-like language called Visual J# (formerly known as J++) that is not compatible with the Java specification, and the associated class library mostly dates to the old JDK 1.1 version of the language; for these reasons, it is more a transitional language to switch from Java to the Microsoft .NET platform, than a first class Microsoft .NET language. Visual J# has been discontinued with the release of Microsoft Visual Studio 2008. The existing version shipping with Visual Studio 2005 will be supported until 2015 as per the product life-cycle strategy.

History

The Java platform and language began as an internal project at Sun Microsystems in December 1990, providing an alternative to the C++/C programming languages. Engineer Patrick Naughton had become increasingly frustrated with the state of Sun's C++ and C APIs (application programming interfaces) and tools. While considering moving to NeXT, Naughton was offered a chance to work on new technology and thus the *Stealth Project* was started.

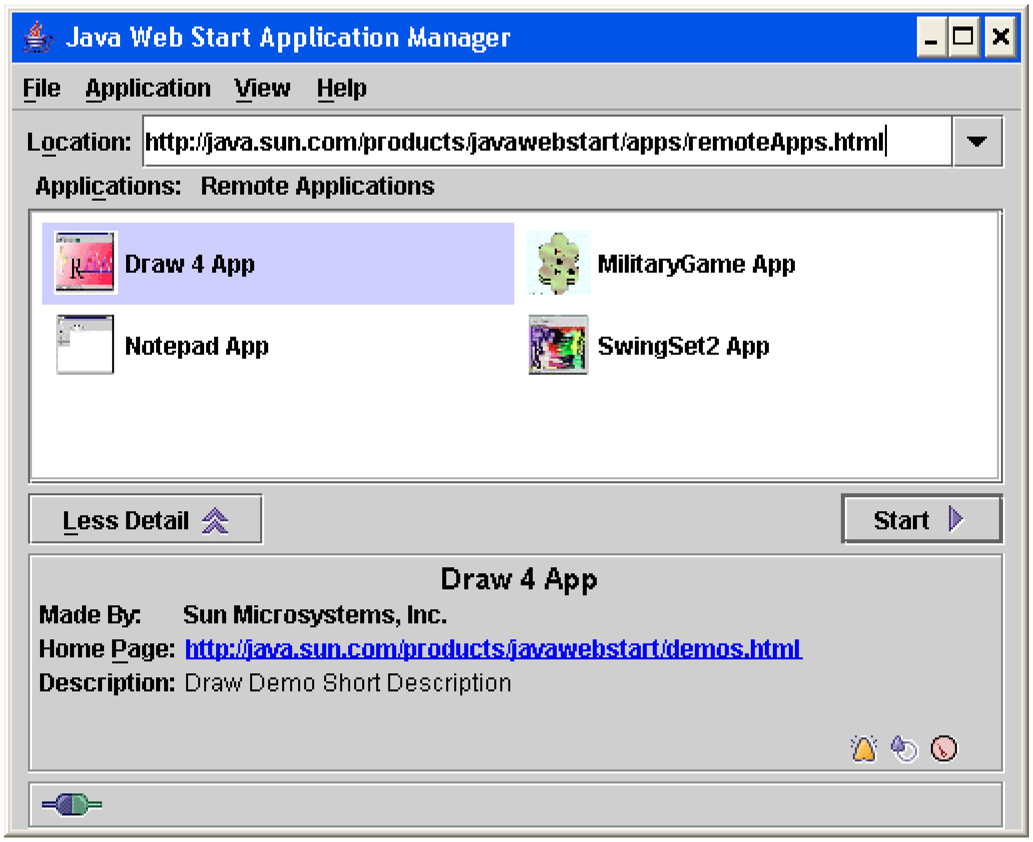
The Stealth Project was soon renamed to the *Green Project* with James Gosling and Mike Sheridan joining Naughton. Together with other engineers, they began work in a small office on Sand Hill Road in Menlo Park, California. They were attempting to develop a new technology for programming next generation smart appliances, which Sun expected to be a major new opportunity.

The team originally considered using C++, but it was rejected for several reasons. Because they were developing an embedded system with limited resources, they decided that C++ demanded too large a footprint and that its complexity led to developer errors. The language's lack of garbage collection meant that programmers had to manually manage system memory, a challenging and error-prone task. The team was also troubled by the language's lack of portable facilities for security, distributed programming, and threading. Finally, they wanted a platform that could be easily ported to all types of devices.

Bill Joy had envisioned a new language combining Mesa and C. In a paper called *Further*, he proposed to Sun that its engineers should produce an object-oriented environment based on C++. Initially, Gosling attempted to modify and extend C++ (which he referred to as "C++ ++ --") but soon abandoned that in favor of creating an entirely new language, which he called **Oak**, after the tree that stood just outside his office.

By the summer of 1992, they were able to demonstrate portions of the new platform including the Green OS, the Oak language, the libraries, and the hardware. Their first attempt, demonstrated on September 3, 1992, focused on building a PDA device named *Star7* which had a graphical interface and a smart agent called "Duke" to assist the user. In November of that year, the Green Project was spun off to become **firstperson**, a wholly owned subsidiary of Sun Microsystems, and the team relocated to Palo Alto, California. The firstperson team was interested in building highly interactive devices, and when Time Warner issued an RFP for a set-top box, firstperson changed their target and responded with a proposal for a set-top box platform. However, the cable industry felt that their platform gave too much control to the user and firstperson lost their bid to SGI. An additional deal with The 3DO Company for a set-top box also failed to materialize. Unable to generate interest within the TV industry, the company was rolled back into Sun.

Java meets the Internet


Java Web Start Application Manager
File
Application
View
Help
Location: http://java.sun.com/products/javawebstart/apps/remoteApps.html
Applications: Remote Applications
Draw 4 App
MilitaryGame App
Notepad App
SwingSet2 App
Less Detail
Start
Draw 4 App
Made By: Sun Microsystems, Inc.
Home Page: http://java.sun.com/products/javawebstart/demos.html
Description: Draw Demo Short Description

Java Web Start allows provisioning applications over the Web

In June and July 1994, after three days of brainstorming with John Gage, the Director of Science for Sun, Gosling, Joy, Naughton, Wayne Rosing, and Eric Schmidt, the team re-targeted the platform for the World Wide Web. They felt that with the advent of graphical web browsers like Mosaic, the Internet was on its way to evolving into the same highly interactive medium that they had envisioned for cable TV. As a prototype, Naughton wrote a small browser, WebRunner (named after the movie Blade Runner), later renamed HotJava.

That year, the language was renamed *Java* after a trademark search revealed that *Oak* was used by Oak Technology. Although Java 1.0a was available for download in 1994, the first public release of Java was 1.0a2 with the HotJava browser on May 23, 1995, announced by Gage at the SunWorld conference. His announcement was accompanied by a surprise announcement by Marc Andreessen, Executive Vice President of Netscape Communications Corporation, that Netscape browsers would be including Java support. On January 9, 1996, the JavaSoft group was formed by Sun Microsystems in order to develop the technology.

Usage

Desktop use

According to Sun, the Java Runtime Environment is found on over 700 million PCs. Microsoft has not bundled a Java Runtime Environment (JRE) with its operating systems since Sun Microsystems sued Microsoft for adding Windows-specific classes to the bundled Java runtime environment, and for making the new classes available through Visual J++. A Java runtime environment is bundled with Apple's Mac OS X (although as of the Java for Mac OS X 10.6 Update 3 release, the Apple-supplied runtime is deprecated and may be removed from future OS releases), and many Linux distributions include the partially compatible free software package GNU Classpath.

Some Java applications are in fairly widespread desktop use, including the NetBeans and Eclipse integrated development environments, and file sharing clients such as LimeWire and Vuze. Java is also used in the MATLAB mathematics programming environment, both for rendering the user interface and as part of the core system.

Mobile devices

Java ME has become popular in mobile devices, where it competes with Symbian, BREW, and the .NET Compact Framework.

The diversity of mobile phone manufacturers has led to a need for new unified standards so programs can run on phones from different suppliers - MIDP. The first standard was MIDP 1, which assumed a small screen size, no access to audio, and a 32kB program limit. The more recent MIDP 2 allows access to audio, and up to 64kB for the program size. With handset designs improving more rapidly than the standards, some manufacturers relax some limitations in the standards, for example, maximum program size.

Google's Android Operating System uses the Java language, but not its class libraries, therefore the Android platform cannot be called Java. Android executes the code on the Dalvik VM instead of the Java VM.

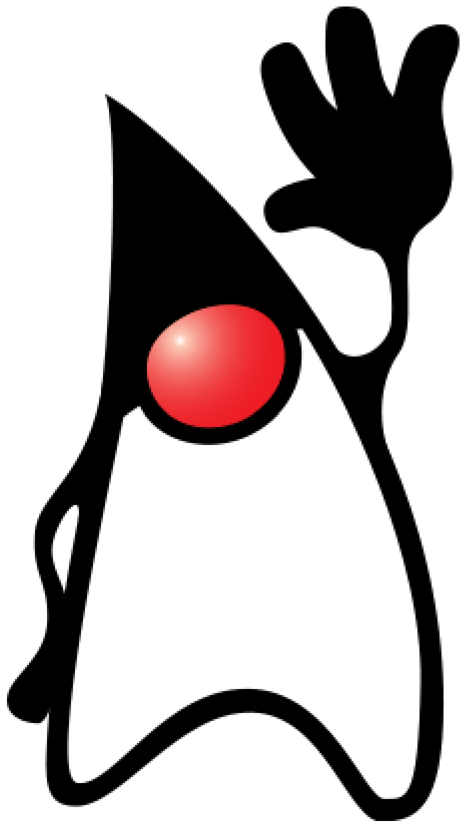
Web server and enterprise use

The Java platform has become a mainstay of enterprise IT development since the introduction of the Enterprise Edition in 1998, in two different ways:

1. Through the coupling of Java to the web server, the Java platform has become a leading platform for integrating the Web with enterprise backend systems. This has allowed companies to move part or all of their business to the Internet environment by way of highly interactive online environments (such as highly dynamic websites) that allow the customer direct access to the business processes (e.g. online banking websites, airline booking systems and so on). This trend has continued from its initial Web-based start:
    - The Java platform has matured into an Enterprise Integration role in which legacy systems are unlocked to the outside world through bridges built on the Java platform. This trend has been supported for Java platform support for EAI standards like messaging and web services and has fueled the inclusion of the Java platform as a development basis in such standards as SCA, XAM and others.
    - Java has become the standard development platform for many companies' IT departments, which do most or all of their corporate development in Java. This type of development is usually related to company-specific tooling (e.g. a booking tool for an airline) and the choice for the Java platform is often driven by a desire to leverage the existing Java infrastructure to build highly intelligent and interconnected tools.
2. The Java platform has become the main development platform for many software tools and platforms that are produced by third-party software groups (commercial, open source and hybrid) and are used as configurable (rather than programmable) tools by companies. Examples in this category include web servers, application servers, databases, enterprise service buses, BPM tools and content management systems.

Enterprise use of Java has also long been the main driver of open source interest in the platform. This interest has inspired open source communities to produce everything from simple function libraries to program development frameworks (e.g. the Spring Framework, Wicket, Dojo, Hibernate) to open source implementations of standards and tools (e.g. Apache Tomcat, the Glassfish application server, the Mule and ServiceMix ESBs).

Mascot



Duke, the Java mascot

Duke is Java's mascot.

When Sun announced that Java SE and Java ME would be released under a free software license (the GNU General Public License), they released the Duke graphics under the free BSD license at the same time.

Licensing

The source code for Sun's implementations of Java (which is the de-facto reference implementation) has been available for some time, but until recently the license terms severely restricted what could be done with it without signing (and generally paying for) a contract with Sun. As such these terms did not satisfy the requirements of either the Open Source Initiative or the Free Software Foundation to be considered open source or free software, and Sun Java was therefore a proprietary platform.

While several third-party projects (e.g. GNU Classpath and Apache Harmony) created free software partial Java implementations, the sheer size of the Sun libraries combined with the use of clean room techniques meant that their implementations of the Java libraries (the compiler and vm are comparatively small and well defined) were incomplete and not fully compatible. These implementations also tended to be a long way behind Sun's in terms of optimization.

Free software

Sun announced in JavaOne 2006 that Java would become free and open source software, and on October 25, 2006, at the Oracle OpenWorld conference, Jonathan I. Schwartz said that the company was set to announce the release of the core Java Platform as free and open source software within 30 to 60 days.

Sun released the Java HotSpot virtual machine and compiler as free software under the GNU General Public License on November 13, 2006, with a promise that the rest of the JDK (which includes the JRE) would be placed under the GPL by March 2007 ("except for a few components that Sun does not have the right to publish in source form under the GPL"). According to Richard Stallman, this would mean an end to the "Java trap". Mark Shuttleworth called the initial press announcement, "*A real milestone for the free software community*".

Sun released the source code of the Class library under GPL on May 8, 2007, except some limited parts that were licensed by Sun from 3rd parties who did not want their code to be released under a free software and open-source license. Some of the encumbered parts turned out to be fairly key parts of the platform such as font rendering and 2D rasterisation, but these were released as open-source later by Sun.

Sun's goal is to replace the parts that remain proprietary and closed-source with alternative implementations and make the class library completely free and open source. A third party project called IcedTea has created a completely free and highly usable JDK by replacing encumbered code with either stubs or code from GNU Classpath. IcedTea is currently available on Fedora 7 and Ubuntu.

In June 2008, it was announced that IcedTea6 (as the packaged version of OpenJDK on Fedora 9) has passed the Technology Compatibility Kit tests and can claim to be a fully compatible Java 6 implementation.

Chapter 2

Java Virtual Machine and Java Class Library

Java Virtual Machine

A **Java Virtual Machine** (**JVM**) enables a set of computer software programs and data structures to use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of computer intermediate language commonly referred to as Java bytecode. This language conceptually represents the instruction set of a stack-oriented, capability architecture. Sun Microsystems claims there are over 4.5 billion JVM-enabled devices.

Overview

A JVM can also implement programming languages other than Java. For example, Ada source code can be compiled to Java bytecode, which may then be executed by a JVM. JVMs can also be released by other companies besides Sun (the developer of Java) — JVMs using the "Java" trademark may be developed by other companies as long as they adhere to the JVM specification published by Sun and to related contractual obligations.

Java was conceived with the concept of WORA: "write once, run anywhere". This is done using the Java Virtual Machine. The JVM is the environment in which java programs execute. It is software that is implemented on non-virtual hardware and on standard operating systems.

JVM is a crucial component of the Java platform, and because JVMs are available for many hardware and software platforms, Java can be both middleware and a platform in its own right, hence the trademark write once, run anywhere. The use of the same bytecode for all platforms allows Java to be described as "compile once, run anywhere", as opposed to "write once, compile anywhere", which describes cross-platform compiled languages. A JVM also enables such features as automated exception handling, which provides "root-cause" debugging information for every software error (exception), independent of the source code.

A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together form the Java Runtime Environment (JRE).

Execution environment

Programs intended to run on a JVM must be compiled into a standardized portable binary format, which typically comes in the form of .class files. A program may consist of many classes in different files. For easier distribution of large programs, multiple class files may be packaged together in a .jar file (short for Java archive).

The JVM runtime executes `.class` or `.jar` files, emulating the JVM instruction set by interpreting it, or using a just-in-time compiler (JIT) such as Sun's HotSpot. JIT compiling, not interpreting, is used in most JVMs today to achieve greater speed. There are also ahead-of-time compilers that enable developers to precompile class files into native code for particular platforms.

Like most virtual machines, the Java Virtual Machine has a stack-based architecture akin to a microcontroller/microprocessor. However, the JVM also has low-level support for Java-like classes and methods, which amounts to a highly idiosyncratic memory model and capability-based architecture.

The JVM, which is the instance of the 'JRE' (*Java Runtime Environment*), comes into action when a Java program is executed. When execution is complete, this instance is garbage collected. JIT is the part of the JVM that is used to speed up the execution time. JIT compiles parts of the bytecode that have similar functionality at the same time, and hence reduces the amount of time needed for compiling.

JVM languages

Versions of non-JVM languages

Language	**On JVM**
Erlang	Erjang
JavaScript	Rhino
PHP	Resin
Python	Jython
REXX	NetRexx
Ruby	JRuby
Tcl	Jacl
Languages designed expressly for JVM	
MIDletPascal	
Clojure	
Groovy	
Scala	

Although the JVM was primarily aimed at running compiled Java programs, many other languages can now run on top of it. The JVM has currently no built-in support for dynamically typed languages: the existing JVM instruction set is statically typed, although the JVM can be used to implement interpreters for dynamic languages. The JVM has a limited support for dynamically modifying existing classes and methods; this currently only works in a debugging environment. However, new classes and methods can be added dynamically. Built-in support for dynamic languages is currently planned for Java 7.

Bytecode verifier

A basic philosophy of Java is that it is inherently "safe" from the standpoint that no user program can "crash" the host machine or otherwise interfere inappropriately with other operations on the host machine, and that it is possible to protect certain functions and data structures belonging to "trusted" code from access or corruption by "untrusted" code executing within the same JVM. Furthermore, common programmer errors that often lead to data corruption or unpredictable behavior such as accessing off the end of an array or using an uninitialized pointer are not allowed to occur. Several features of Java combine to provide this safety, including the class model, the garbage-collected heap, and the verifier.

The JVM *verifies* all bytecode before it is executed. This verification consists primarily of three types of checks:

- Branches are always to valid locations
- Data is always initialized and references are always type-safe
- Access to "private" or "package private" data and methods is rigidly controlled.

The first two of these checks take place primarily during the "verification" step that occurs when a class is loaded and made eligible for use. The third is primarily performed dynamically, when data items or methods of a class are first accessed by another class.

The verifier permits only some bytecode sequences in valid programs, e.g. a jump (branch) instruction can only target an instruction within the same function or method. Furthermore, the verifier ensures that any given instruction operates on a fixed stack location, allowing the JIT compiler to transform stack accesses into fixed register accesses. Because of this, that the JVM is a stack architecture does not imply a speed penalty for emulation on register-based architectures when using a JIT compiler. In the face of the code-verified JVM architecture, it makes no difference to a JIT compiler whether it gets named imaginary registers or imaginary stack positions that must be allocated to the target architecture's registers. In fact, code verification makes the JVM different from a classic stack architecture whose efficient emulation with a JIT compiler is more complicated and typically carried out by a slower interpreter.

Code verification also ensures that arbitrary bit patterns cannot get used as an address. Memory protection is achieved without the need for a memory management unit (MMU).

Thus, JVM is an efficient way of getting memory protection on simple architectures that lack an MMU. This is analogous to managed code in Microsoft's .NET Common Language Runtime, and conceptually similar to capability architectures such as the Plessey 250, and IBM System/38.

Bytecode instructions

The JVM has instructions for the following groups of tasks:

- Load and store
- Arithmetic
- Type conversion
- Object creation and manipulation
- Operand stack management (push / pop)
- Control transfer (branching)
- Method invocation and return
- Throwing exceptions
- Monitor-based concurrency

The aim is binary compatibility. Each particular host operating system needs its own implementation of the JVM and runtime. These JVMs interpret the bytecode semantically the same way, but the actual implementation may be different. More complex than just emulating bytecode is compatibly and efficiently implementing the Java core API that must be mapped to each host operating system.

Secure execution of remote code

A virtual machine architecture allows very fine-grained control over the actions that code within the machine is permitted to take. This is designed to allow safe execution of untrusted code from remote sources, a model used by Java applets. Applets run within a VM incorporated into a user's browser, executing code downloaded from a remote HTTP server. The remote code runs in a restricted "sandbox", which is designed to protect the user from misbehaving or malicious code. Publishers can purchase a certificate with which to digitally sign applets as "safe", giving them permission to ask the user to break out of the sandbox and access the local file system, clipboard or network.

C to bytecode compilers

From the point of view of a compiler, the Java Virtual Machine is just another processor with an instruction set, Java bytecode, for which code can be generated. The JVM was originally designed to execute programs written in the Java language. However, the JVM provides an execution environment in the form of a bytecode instruction set and a runtime system that is general enough that it can be used as the target for compilers of other languages.

Because of its close association with the Java language, the JVM performs the strict runtime checks mandated by the Java specification. That requires C to bytecode compilers to provide their own "lax machine abstraction", for instance producing compiled code that uses a Java array to represent main memory (so pointers can be compiled to integers), and linking the C library to a centralized Java class that emulates system calls. Most or all of the compilers listed below use a similar approach.

Several C to bytecode compilers exist:

- NestedVM translates C to MIPS machine language first before converting to Java bytecode.
- Cybil works similarly to NestedVM but targets J2ME devices.
- LLJVM compiles C to LLVM IR, which is then translated to JVM bytecode.
- C2J is also GCC-based, but it produces intermediary Java source code before generating bytecode. Supports the full ANSI C runtime.
- Axiomatic Multi-Platform C supports full ANSI C 1989, SWT, and J2ME CDC 1.1 for mobile devices.
- Java Backend for GCC, possibly the oldest project of its kind, was developed at The University of Queensland in 1999.
- Javum is an attempt to port the full GNU environment to the JVM, and includes one of the above compilers packaged with additional utilities.
- egcs-jvm appears to be an inactive project.

Compilers targeting Java bytecode have been written for other programming languages, including Ada and COBOL.

Licensing

Starting with J2SE 5.0, changes to the JVM specification have been developed under the Java Community Process as JSR 924. As of 2006, changes to specification to support changes proposed to the class file format (JSR 202) are being done as a maintenance release of JSR 924. The specification for the JVM is published in book form, known as "blue book". The preface states:

We intend that this specification should sufficiently document the Java Virtual Machine to make possible compatible clean-room implementations. Sun provides tests that verify the proper operation of implementations of the Java Virtual Machine.

Sun's JVM is called HotSpot. Clean-room Java implementations include Kaffe, IBM J9 and Dalvik. Sun retains control over the Java trademark, which it uses to certify implementation suites as fully compatible with Sun's specification.

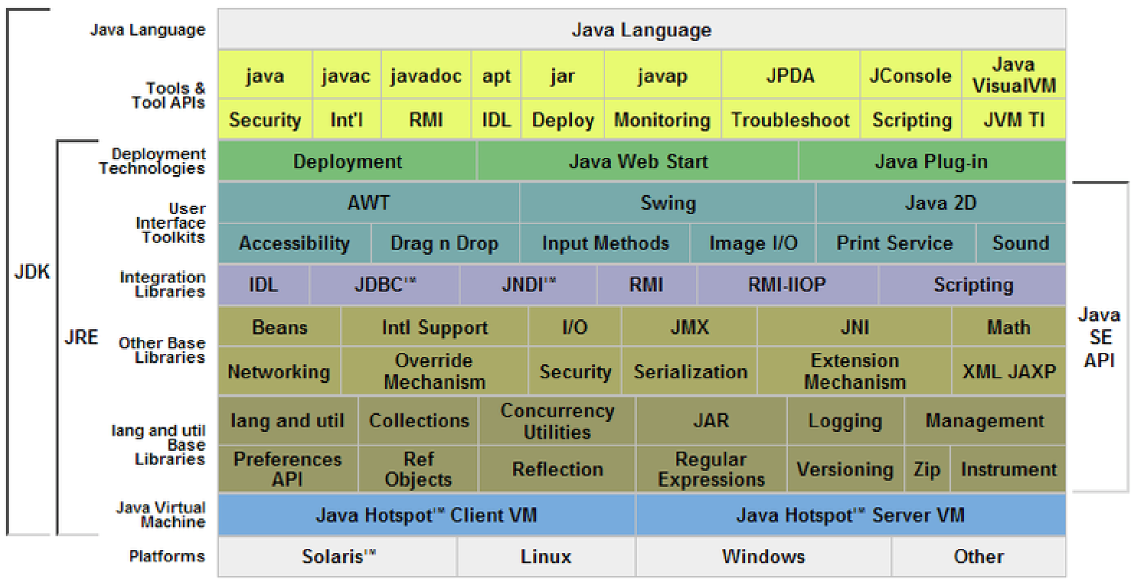
Heap

The *Java Virtual Machine heap* is the area of memory used by the JVM (and specifically HotSpot) for dynamic memory allocation. The heap is split up into "generations":

- The *young generation* stores short-lived objects that are created and immediately garbage collected.
- Objects that persist longer are moved to the *old generation* (also called the *tenured generation*).
- The *permanent generation* (or *permgen*) is used for class definitions and associated metadata.

Originally there was no permanent generation, and objects and classes were stored together in the same area. But as class unloading occurs much more rarely than objects are collected, moving class structures to a specific area allows significant performance improvements.

Java Class Library



Java Platform diagram

The **Java Class Library** is a set of dynamically loadable libraries that Java applications can call at runtime. Because the Java Platform is not dependent on any specific operating system, applications cannot rely on any of the existing libraries. Instead, the Java Platform provides a comprehensive set of standard class libraries, containing much of the same reusable functions commonly found in modern operating systems.

The Java class libraries serve three purposes within the Java Platform:

- Like other standard code libraries, they provide the programmer a well-known set of functions to perform common tasks, such as maintaining lists of items or performing complex string parsing.
- In addition, the class libraries provide an abstract interface to tasks that would normally depend heavily on the hardware and operating system. Tasks such as network access and file access are often heavily dependent on the native capabilities of the platform.
- Finally, some underlying platforms may not support all of the features a Java application expects. In these cases, the class libraries can either emulate those features using whatever is available, or provide a consistent way to check for the presence of a specific feature.

Architecture

The Java Class Library is almost entirely written in Java itself, except for the parts that need direct access to the hardware and operating system (such as for I/O, or bitmap graphics). The classes that give access to these functions commonly use native interface wrappers to access the API of the operating system.

Almost all of the Java Class Library is stored in a single Java archive file called "rt.jar", which is provided with JRE and JDK distributions. The Java Class Library (rt.jar) is located in the default bootstrap classpath, and does not have to be found in the classpath declared for the application.

Conformance

Any Java implementation must pass the Java Technology Compatibility Kit tests for compliance.

Main Features

Features of the Class Library are accessed through classes grouped by packages.

- `java.lang` contains fundamental classes and interfaces closely tied to the language and runtime system.
- I/O and networking: access to the platform file system, and more generally to networks, is provided through the `java.io`, `java.nio`, and `java.net` packages.
- Mathematics package: `java.math` provides regular mathematical expressions, as well as arbitrary-precision decimals and integers numbers.
- Collections and Utilities : provide built-in Collection data structures, and various utility classes, for Regular expressions, Concurrency, logging and Data compression.
- GUI and 2D Graphics: the `java.awt` package supports basic GUI operations and binds to the underlying native system. It also contains the 2D Graphics API. The `javax.swing` package is built on AWT and provides a platform independent

widget toolkit, as well as a Pluggable look and feel. It also deals with editable and non-editable text components.

- Sound: provides interfaces and classes for reading, writing, sequencing, and synthesizing of sound data.
- Text: the `java.text` package deals with text, dates, numbers, and messages.
- Image package: `java.awt.image` and `javax.imageio` provide APIs to write, read, and modify images.
- XML: built-in classes handle SAX, DOM, StAX, XSLT transforms, XPath, and various APIs for Web services, as SOAP protocol and JAX-WS.
- CORBA and RMI APIs, including a built-in ORB
- Security and Cryptography
- Databases: access to SQL databases is provided through the `java.sql` package.
- Access to Scripting engines: the `javax.script` package gives access any Scripting language that conforms to this API.
- Applets: `java.applet` allows applications to be downloaded over a network and run within a guarded sandbox
- Java Beans: `java.beans` provides ways to manipulate reusable components.

Licensing

Following their promise to release a fully buildable JDK based almost completely on free and open source code in the first half of 2007 , Sun released the complete source code of the Class Library under the GPL on May 8, 2007, except some limited parts that were licensed by Sun from third parties who did not want their code to be released under a free and open-source license. Sun's goal is to replace the parts that remain proprietary and closed source with alternative implementations and make the Class Library completely free and open source.

As of May 2008, the only part of the Class library that remain proprietary and closed-source (4% as of May 2007 for OpenJDK 7, and less than 1% as of May 2008 and OpenJDK 6) is :

- The SNMP implementation.

Since the first May 2007 release, Sun Microsystems, with the help of the community, has released as Open-source or replaced with Open-source alternatives almost all the encumbered code:

- All the audio engine code, including the software synthetizer, has been released as Open-source. The closed-source software synthesizer has been replaced by a new synthesizer developed specifically for OpenJDK called *Gervill*,
- All cryptography classes used in the Class library have been released as Open-source,
- The code that scales and rasterizes fonts has been replaced by FreeType

- The native color management system has been replaced by LittleCMS . There is a pluggable layer in the JDK, so that the commercial version can use the old color management system and OpenJDK can use LittleCMS.
- The anti-aliasing graphics rasterizer code has been replaced by the Open-sourced Pisces renderer used in the phoneME project. This code is fully functional, but still needs some performance enhancements ,
- The Javascript plugin has been open-sourced (the JavaScript engine itself was open-sourced from the beginning).

Alternative implementations

GNU Classpath is the other main free software class library for Java. Contrary to other implementations, it only implements the Class Library, and is used by many free Java runtimes (like Kaffe, SableVM, JamVM, CACAO).

Apache Harmony is another free software class library. Its aim is to also implement the other parts of the Java stack (Virtual Machine, Compiler, and other tools required for any Java implementation).

Chapter 3

JavaFX

JavaFX

Developer(s)	Sun Microsystems
Stable release	1.3 / April 22, 2010; 7 months ago
Operating system	Java Runtime Environment
Platform	Cross-platform
Available in	JavaFX Script
Type	Rich Internet applications
License	EULA

JavaFX is a Java platform for creating and delivering rich Internet applications that can run across a wide variety of connected devices. The current release (JavaFX 1.3, April 2010) enables building applications for desktop, browser and mobile phones. TV set-top boxes, gaming consoles, Blu-ray players and other platforms are planned.

To build JavaFX apps developers use a statically typed, declarative language called JavaFX Script; Java code can be integrated into JavaFX programs. JavaFX is compiled to Java bytecode, so JavaFX applications run on any desktop and browser that runs the Java Runtime Environment (JRE) and on top of mobile phones running Java ME.

On desktop, the current release supports Windows XP, Windows Vista and Mac OS X operating systems. Beginning with JavaFX 1.2 Oracle has released beta versions for Linux and OpenSolaris. On mobile, JavaFX is capable of running on multiple mobile operating systems, including Symbian OS, Windows Mobile, and proprietary real-time operating systems.

WORLD TECHNOLOGIES

Commentators have speculated JavaFX will compete on the desktop with Adobe Flash Player, Adobe AIR, OpenLaszlo and Microsoft Silverlight.

Technical highlights

Common profile. JavaFX is based on the concept of a “Common profile” that is intended to span across all devices supported by JavaFX. This approach makes it possible for developers to use a common programming model while building an application targeted for both desktop and mobile devices and to share much of the code, graphics assets and content between desktop and mobile versions. To address the need for tuning applications on a specific class of devices, the JavaFX 1.1 platform includes APIs that are desktop or mobile-specific. For example JavaFX Desktop profile includes Swing and advanced visual effects.

Drag-to-Install. From the point of view of the end user “Drag-to-Install” allows them to drag a JavaFX widget (or application residing in a website and is visible within the browser window) and drop it onto their desktop. The application will not lose its state or context even after the browser is closed. An application can also be re-launched by clicking on a shortcut that gets created automatically on the user's desktop. This behavior is enabled out-of-the-box by the Java applet mechanism and is leveraged by JavaFX from the underlying Java layer. Sun touts “Drag-to-Install” as opening up of a new distribution model and allowing developers to “break away from the browser”.

Integrating graphics created with third-party tools. JavaFX includes a set of plug-ins for Adobe Photoshop and Illustrator that enable advanced graphics to be integrated directly into JavaFX applications. The plug-ins generate JavaFX Script code that preserves layers and structure of the graphics. Developers can then easily add animation or effects to the static graphics imported. There is also an SVG graphics converter tool (a.k.a. Media Factory) that allows for importing graphics and previewing assets after the conversion to JavaFX format.

Design highlights

Sun Microsystems licensed a custom typeface called Amble for use on JavaFX powered devices. The font family was designed by mobile user interface design specialists Punchcut and is available as part of the JavaFX SDK 1.3 Release.

JavaFX platform components

Current release of JavaFX platform includes the following components:

1. The JavaFX SDK: JavaFX compiler and runtime tools. Graphics, media web services, and rich text libraries
2. NetBeans IDE for JavaFX: NetBeans with drag-and-drop palette to add objects with transformations, effects and animations plus set of samples and best

practices. For Eclipse users there is a community-supported plugin hosted on Project Kenai

3. Tools and plugins for creative tools (a.k.a. Production Suite): Plugins for Adobe Photoshop and Adobe Illustrator that can export graphics assets to JavaFX Script code, tools to convert SVG graphics into JavaFX Script code and preview assets converted to JavaFX from other tools

History

JavaFX Script, the scripting component of JavaFX, began life as a project by Chris Oliver called F3.

Sun Microsystems first announced JavaFX at the JavaOne Worldwide Java Developer conference on May 2007.

In May 2008 Sun Microsystems announced plans to deliver JavaFX for the browser and desktop by the third quarter of 2008, and JavaFX for mobile devices in the second quarter of 2009. Sun also announced a multi-year agreement with On2 Technologies to bring comprehensive video capabilities to the JavaFX product family using the company's TrueMotion Video codec.

Since end of July 2008, developers could download a preview of the JavaFX SDK for Windows and Macintosh, as well as the JavaFX plugin for NetBeans 6.1. On December 4, 2008 Sun released JavaFX 1.0.

JavaFX 1.1

JavaFX for mobile development was finally made available as part of the JavaFX 1.1 release announced officially on February 12, 2009.

JavaFX 1.2

JavaFX 1.2 was released at JavaOne on June 2, 2009. This release introduced:

- Beta support for Linux and Solaris,
- Built-in controls and layouts,
- Skinnable CSS controls,
- Built-in chart widgets,
- JavaFX I/O management, masking differences between desktop and mobile devices,
- Speed improvements.
- Windows Mobile Runtime with Sun Java Wireless Client.

JavaFX 1.3

JavaFX 1.3 was released on April 22, 2010. This release introduces:

- Performance improvements
- Support of additional platforms
- Improved support for user interface controls

Current release

JavaFX 1.3.1

This version is an updated version of JavaFX released on 21 Aug 2010 . Features:

- Quick startup time of JavaFX application.
- Custom progress bar for application startup.

Future

JavaFX 2.0

JavaFX 2.0 has been scheduled for release in the second half of 2011. A notable change in this release is that the JavaFX scripting language will be discontinued and its functionality moved to regular Java APIs.

License

There are currently various licenses for the modules that compose the JavaFX runtime:

- The core JavaFX runtime is still proprietary software and its code has not yet been released to the public,
- The JavaFX compiler and an older version of the 2D Scene graph are released under a GPL v2 license,
- The NetBeans plugin for JavaFX is dual licensed under GPL v2 and CDDL.

During development, Sun explained they will roll out their strategy for the JavaFX licensing model for JavaFX first release. After the release, Jeet Kaul, Sun's Vice president for Client Software, explained that they will soon publish a specification for JavaFX and its associated file formats, and will continue to open source the JavaFX runtime, and decouple this core from the proprietary parts licensed by external parties.

Chapter 4

Java Bytecode

Java bytecode is the form of instructions that the Java virtual machine executes. Each bytecode opcode is one byte in length, although some require parameters, resulting in some multi-byte instructions. Not all of the possible 256 opcodes are used. In fact, Sun Microsystems, the original creators of the Java programming language, the Java virtual machine and other components of the Java Runtime Environment (JRE), have set aside 3 values to be permanently unimplemented.

Relation to Java

A Java programmer does not need to be aware of or understand Java bytecode at all. However, as suggested in the IBM developerWorks journal, "Understanding bytecode and what bytecode is likely to be generated by a Java compiler helps the Java programmer in the same way that knowledge of assembler helps the C or C++ programmer."

Instructions

As each byte has 256 potential values, there are 256 possible opcodes. Of these, 0x00 through 0xca, 0xfe, and 0xff are assigned values. 0xba is unused for historical reasons. 0xca is reserved as a breakpoint instruction for debuggers and is not used by the language. Similarly, 0xfe and 0xff are not used by the language, and are reserved for internal use by the virtual machine.

Instructions fall into a number of broad groups:

- Load and store (e.g. aload_0, istore)
- Arithmetic and logic (e.g. ladd, fcmpl)
- Type conversion (e.g. i2b, d2i)
- Object creation and manipulation (new, putfield)
- Operand stack management (e.g. swap, dup2)
- Control transfer (e.g. ifeq, goto)
- Method invocation and return (e.g. invokespecial, areturn)

There are also a few instructions for a number of more specialized tasks such as exception throwing, synchronization, etc.

Many instructions have prefixes and/or suffixes referring to the types of operands they operate on. These are as follows:

Prefix/Suffix	**Operand Type**
`i`	integer
`l`	long
`s`	short
`b`	byte
`c`	character
`f`	float
`d`	double
`a`	reference

For example, "iadd" will add two integers, while "dadd" will add two doubles. The "const", "load", and "store" instructions may also take a suffix of the form "_*n*", where *n* is a number from 0-3 for "load" and "store". The maximum *n* for "const" differs by type.

The "const" instructions push a value of the specified type onto the stack. For example "iconst_5" will push an integer 5, while "dconst_1" will push a double 1. There is also an "aconst_null", which pushes "null". The *n* for the "load" and "store" instructions specifies the location in the variable table to load from or store to. The "aload_0" instruction pushes the object in variable 0 onto the stack (this is usually the "this" object). "istore_1" stores the integer on the top of the stack into variable 1. For variables with higher numbers the suffix is dropped and operators must be used .

Model of computation

The model of computation of Java bytecode is that of a stack-oriented programming language. For example, assembly code for an x86 processor might look like this:

```
add eax, edx
mov ecx, eax
```

This code would add two values and move the result to a different location. Similar disassembled bytecode might look like this:

```
0 iload_1
1 iload_2
2 iadd
3 istore_3
```

Here, the two values to be added are pushed onto the stack, where they are retrieved by the addition instruction, summed, and the result placed back on the stack. The storage instruction then moves the top value of the stack into a variable location. The numbers in front of the instructions simply represent the offset of each instruction from the beginning of the method. This stack-oriented model extends to the object oriented aspects of the language as well. A method call called "getName()", for example, may look like the following:

```
Method java.lang.String getName()
0 aload_0       // The "this" object is stored in location 0 of the
variable table
1 getfield #5 <Field java.lang.String name>
                // This instructon pops an object from the top of the
stack, retrieves the specified
                // field from it, and pushes the field onto the stack.
                // In this example, the "name" field is the fifth field
of the class.
4 areturn       // Returns the object on top of the stack from the
method.
```

Example

Consider the following Java code:

```
  outer:
  for (int i = 2; i < 1000; i++) {
      for (int j = 2; j < i; j++) {
          if (i % j == 0)
              continue outer;
      }
      System.out.println (i);
  }
```

A Java compiler might translate the Java code above into byte code as follows, assuming the above was put in a method:

```
  0:   iconst_2
  1:   istore_1
  2:   iload_1
  3:   sipush  1000
  6:   if_icmpge       44
  9:   iconst_2
  10:  istore_2
  11:  iload_2
  12:  iload_1
  13:  if_icmpge       31
  16:  iload_1
  17:  iload_2
  18:  irem
  19:  ifne    25
  22:  goto    38
  25:  iinc    2, 1
  28:  goto    11
  31:  getstatic       #84; //Field
java/lang/System.out:Ljava/io/PrintStream;
  34:  iload_1
  35:  invokevirtual   #85; //Method java/io/PrintStream.println:(I)V
  38:  iinc    1, 1
  41:  goto    2
  44:  return
```

Generation

The most common language targeting Java Virtual Machine by producing Java bytecode is Java. Originally only one compiler existed, the javac compiler from Sun Microsystems, which compiles Java source code to Java bytecode; but because all the specifications for Java bytecode are now available, other parties have supplied compilers that produce Java bytecode. Examples of other compilers include:

- Jikes, compiles from Java to Java bytecode (developed by IBM, implemented in C++)
- Espresso, compiles from Java to Java bytecode (Java 1.0 only)
- GCJ, the GNU Compiler for Java, compiles from Java to Java bytecode; it is also able to compile to native machine code and is available as part of the GNU Compiler Collection (GCC).

Some projects provide Java assemblers to enable writing Java bytecode by hand. Assembler code may be also generated by machine, for example by compiler targeting Java virtual machine. Notable Java assemblers include:

- Jasmin, takes textual descriptions for Java classes, written in a simple assembler-like syntax using Java Virtual Machine instruction set and generates a Java class file
- Jamaica, a macro assembly language for the Java virtual machine. Java syntax is used for class or interface definition. Method bodies are specified using bytecode instructions.

Others have developed compilers, for different programming languages, in order to target the Java virtual machine, such as:

- JRuby and Jython, two scripting languages based on Ruby and Python
- Groovy, a scripting language based on Java
- Scala, a type-safe general-purpose programming language supporting object-oriented and functional programming
- JGNAT and AppletMagic, compile from the Ada programming language to Java bytecode
- C to Java byte-code compilers
- Clojure
- MIDletPascal

JavaFX code is also compiled to Java bytecode.

Execution

Java bytecode is designed to be executed in a Java virtual machine. There are several virtual machines available today, both free and commercial products.

If executing Java bytecode in a Java virtual machine is not desirable, a developer can also compile Java source code or Java bytecode directly to native machine code with tools such as the GNU Compiler for Java. Some processors can execute Java bytecode natively. Such processors are known as *Java processors*.

Support for dynamic languages

The Java Virtual Machine has currently no built-in support for dynamically typed languages, because the existing JVM instruction set is statically typed - in the sense that method calls have their signatures type-checked at compile time, without a mechanism to defer this decision to run time, or to choose the method dispatch by an alternative approach.

JSR 292 (*Supporting Dynamically Typed Languages on the JavaTM Platform*) proposes to add a new `invokedynamic` instruction at the JVM level, to allow method invocation relying on dynamic type checking (instead of the existing statically type-checked `invokevirtual` instruction). The Da Vinci Machine is a prototype virtual machine implementation that hosts JVM extensions aimed at supporting dynamic languages.

Chapter 5

Class (File Format) and Concutest

Class (file format)

	Class
Filename extension	`.class`
Developed by	Sun Microsystems
Type of format	Bytecode

In the Java programming language, source files (.java files) are compiled into (virtual) machine-readable **class files** which have a .class extension. Since Java is a platform-independent language, source code is compiled into an output file known as bytecode, which it stores in a .class file. If a source file has more than one class, each class is compiled into a separate .class file. These .class files can be loaded by any Java Virtual Machine (JVM).

Since JVMs are available for many platforms, the .class file compiled in one platform will execute in a JVM of another platform. This makes Java platform-independent.

History

As of 2006, the modification of the class file format is being considered under Java Specification Request (JSR) 202.

File layout and structure

Sections

There are 10 basic sections to the Java Class File structure:

- **Magic Number**: 0xCAFEBABE
- **Version of Class File Format**: the minor and major versions of the class file
- **Constant Poolabstract, static, etc**

- **This Class**: The name of the current class
- **Super Class**: The name of the super class
- **Interfaces**: Any interfaces in the class
- **Fields**: Any fields in the class
- **Methods**: Any methods in the class
- **Attributes**: Any attributes of the class (for example the name of the sourcefile, etc)

There is a handy mnemonic for remembering these 10: **M**y **V**ery **C**ute **A**nimal **T**urns **S**avage **I**n **F**ull **M**oon **A**reas.

Magic, Version, Constant, Access, This, Super, Interfaces, Fields, Methods, Attributes (MVCATSIFMA)

General layout

Because the class file contains variable-sized items and does not also contain embedded file offsets (or pointers), it is typically parsed sequentially, from the first byte toward the end. At the lowest level the file format is described in terms of a few fundamental data types:

- **u1**: an unsigned 8-bit integer
- **u2**: an unsigned 16-bit integer in big-endian byte order
- **u4**: an unsigned 32-bit integer in big-endian byte order
- **table**: an array of variable-length items of some type. The number of items in the table is identified by a preceding count number, but the size in bytes of the table can only be determined by examining each of its items.

Some of these fundamental types are then re-interpreted as higher-level values (such as strings or floating-point numbers), depending on context. There is no enforcement of word alignment, and so no padding bytes are ever used. The overall layout of the class file is as shown in the following table.

byte offset	**size**	**type or value**	**description**
0	4 bytes	u1 = 0xCA hex	magic number (CAFEBABE) used to identify file as conforming to the class file format
1		u1 = 0xFE hex	
2		u1 = 0xBA hex	

3		u1 = 0xBE hex	
4 5	2 bytes	u2	minor version number of the class file format being used
6 7	2 bytes	u2	major version number of the class file format being used. J2SE 6.0 = 50 (0x32 hex), J2SE 5.0 = 49 (0x31 hex), JDK 1.4 = 48 (0x30 hex), JDK 1.3 = 47 (0x2F hex), JDK 1.2 = 46 (0x2E hex), JDK 1.1 = 45 (0x2D hex).
8 9	2 bytes	u2	constant pool count, number of entries in the following constant pool table. This count is at least one greater than the actual number of entries.
10	*cpsize* (variable)	table	constant pool table, an array of variable-sized constant pool entries, containing items such as literal numbers, strings, and references to classes or methods. Indexed starting at 1, containing (*constant pool count* - 1) number of entries in total.
10+*cpsize* 11+*cpsize*	2 bytes	u2	access flags, a bitmask
12+*cpsize* 13+*cpsize*	2 bytes	u2	identifies *this* class, index into the constant pool to a "Class"-type entry
14+*cpsize* 15+*cpsize*	2 bytes	u2	identifies *super* class, index into the constant pool to a "Class"-type entry
16+*cpsize* 17+*cpsize*	2 bytes	u2	interface count, number of entries in the following interface table
18+*cpsize*	*isize* (variable)	table	interface table, an array of variable-sized interfaces
18+*cpsize*+*isize* 19+*cpsize*+*isize*	2 bytes	u2	field count, number of entries in the following field table
20+*cpsize*+*isize*	*fsize* (variable)	table	field table, variable length array of fields

...			
20+*cpsize*+*isize*+*fsize*	2 bytes	u2	method count, number of entries in the following method table
21+*cpsize*+*isize*+*fsize*			
22+*cpsize*+*isize*+*fsize*			
...	*msize* (variable)	table	method table, variable length array of methods
...			
...			
22+*cpsize*+*isize*+*fsize*+*msize*	2 bytes	u2	attribute count, number of entries in the following attribute table
23+*cpsize*+*isize*+*fsize*+*msize*			
24+*cpsize*+*isize*+*fsize*+*msize*			
...	*asize* (variable)	table	attribute table, variable length array of attributes
...			
...			

C programming language representation

This sample text for testing

```
struct Class_File_Format {
   u4 magic_number;

   u2 minor_version;
   u2 major_version;

   u2 constant_pool_count;

   cp_info constant_pool[constant_pool_count - 1];

   u2 access_flags;

   u2 this_class;
   u2 super_class;

   u2 interfaces_count;

   u2 interfaces[interfaces_count];

   u2 fields_count;
   field_info fields[fields_count];

   u2 methods_count;      method_info methods[methods_count];

   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

The constant pool

The constant pool table is where most of the literal constant values are stored. This includes values such as numbers of all sorts, strings, identifier names, references to classes and methods, and type descriptors. All indexes, or references, to specific constants in the constant pool table are given by 16-bit (type u2) numbers, where index value 1 refers to the first constant in the table (index value 0 is invalid).

Due to historic choices made during the file format development, the number of constants in the constant pool table is not actually the same as the constant pool count which precedes the table. First, the table is indexed starting at 1 (rather than 0), so the count should actually be interpreted as the maximum index. Additionally, two types of constants (longs and doubles) take up two consecutive slots in the table, although the second such slot is a phantom index that is never directly used.

The type of each item (constant) in the constant pool is identified by an initial byte *tag*. The number of bytes following this tag and their interpretation are then dependent upon the tag value. The valid constant types and their tag values are:

Tag byte	Additional bytes	Description of constant
1	2+*x* bytes (variable)	UTF-8 (Unicode) string: a character string prefixed by a 16-bit number (type u2) indicating the number of bytes in the encoded string which immediately follows (which may be different than the number of characters). Note that the encoding used is not actually UTF-8, but involves a slight modification of the Unicode standard encoding form.
3	4 bytes	Integer: a signed 32-bit two's complement number in big-endian format
4	4 bytes	Float: a 32-bit single-precision IEEE 754 floating-point number
5	8 bytes	Long: a signed 64-bit two's complement number in big-endian format (takes two slots in the constant pool table)
6	8 bytes	Double: a 64-bit double-precision IEEE 754 floating-point number (takes two slots in the constant pool table)
7	2 bytes	Class reference: an index within the constant pool to a UTF-8 string containing the fully-qualified class name (in *internal format*)
8	2 bytes	String reference: an index within the constant pool to a UTF-8 string
9	4 bytes	Field reference: two indexes within the constant pool, the first pointing to a Class reference, the second to a Name and Type descriptor.
10	4 bytes	Method reference: two indexes within the constant pool, the first pointing to a Class reference, the second to a Name and Type descriptor.
11	4 bytes	Interface method reference: two indexes within the constant pool, the first pointing to a Class reference, the second to a Name and Type

		descriptor.
12	4 bytes	Name and type descriptor: two indexes to UTF-8 strings within the constant pool, the first representing a name (identifier) and the second a specially-encoded type descriptor.

There are only two integral constant types, integer and long. Other integral types appearing in the high-level language, such as boolean, byte, and short must be represented as an integer constant.

Class names in Java, when fully qualified, are traditionally dot-separated, such as "java.lang.Object". However within the low-level Class reference constants, an internal form appears which uses slashes instead, such as "java/lang/Object".

The Unicode strings, despite the moniker "UTF-8 string", are not actually encoded according to the Unicode standard, although it is similar. There are two differences. The first is that the codepoint U+0000 is encoded as the two-byte sequence `C0 80` (in hex) instead of the standard single-byte encoding `00`. The second difference is that supplementary characters (those outside the BMP at U+10000 and above) are encoded using a surrogate-pair construction similar to UTF-16 rather than being directly encoded using UTF-8. In this case each of the two surrogates is encoded separately in UTF-8. For example U+1D11E is encoded as the 6-byte sequence `ED A0 B4 ED B4 9E`, rather than the correct 4-byte UTF-8 encoding of `f0 9d 84 9e`.

Concutest

Concutest is a specialized unit testing framework for the Java programming language. Created by Mathias Ricken while at the JavaPLT (Java Programming Languages Team) at Rice University, Concutest provides a set of unit testing-like features for the analysis of multithreaded programs.

Unit Testing

Traditional unit testing is covered in greater detail in unit testing; however the basic premise is to verify the goodness of a program by analyzing specific attributes of an execution of a program. Specifically, the unit tests of some program will execute a function then verify that its result is correct. For example, one might unit test the mathematical function "f(x,y) = 3x+2y" by verifying that "f(0,0) == 0", "f(1,1) == 5", and others.

Difficulty of Thread Scheduling

One of the aspects that makes modern multithreaded programming so difficult is the fact that thread schedules are nondeterministic. Specifically, there are many possible schedules of execution, and modern operating systems do not selectively pick between them except according to very vague heuristics (such as thread priority). For example, consider a multithreaded processes:

```
Initial conditions:
I = 1
J = 2
Thread A:
1 - Increment I
2 - Increment J
Thread B:
1 - Multiply J by 2
2 - Multiply I by 0
```

The results of this system depend upon the order in which the threads' steps are scheduled. One example schedule is this (which corresponds to Thread A running to completion followed by Thread B):

```
Increment I
Increment J
Multiply J by 2
Multiply I by 0
Final conditions:
I = 0
J = 6
```

However, this is another legitimate schedule:

```
Multiply J by 2
Multiply I by 0
Increment I
Increment J
Final conditions:
I = 1
J = 5
```

There is a problem with our program, because various scheduling of its threads produce different results (we want the program to produce the same results no matter what). It is often difficult to determine that a program has such a problem (known as a race condition) because many schedules, although valid, would not occur during most normal conditions. Certain thread schedules only occur during exceptional conditions, including conditions not specifically known to the software developer (due to the immense complexity of software), and thus cannot be tested or accounted for directly.

Concurrent Unit Testing

Concutest provides a solution to this difficulty through concurrent unit testing. Concutest analyzes a candidate program to determine its sequence points; effectively, it divides a program up into a series of discrete steps, such as those provided for illustration above, and then analyzes all possible control flows.

Concutest, after determining a correct division of a program's steps, analyzes the control flow to determine if the results of the program are the same no matter which schedule is selected. Since in production code, any valid schedule may be executed by the operating system, a valid program must achieve the same result regardless of schedule. Therefore, if multiple runs of a program under different Concutest schedules produce different results, the program has flawed concurrent behavior.

Effectively, this analysis of control flow is a type of unit test; before a program is considered valid and ready for release, software developers run their software through a series of unit tests to ensure its correctness. With Concutest, one of the test steps against which a program is validated is that multiple, various possible thread executions achieve the same result. As part of a comprehensive testing package, Concutest helps to ensure that programs are valid.

Concutest cannot detect all problems that arise from the nondeterminism of concurrency; only those that arise from disparate scheduling can be detected by Concutest. Furthermore, the division of the program into blocks separated by sequence points assumes that the program is free of race conditions. Therefore, it is necessary that race condition detection, e.g. using a lockset algorithm, is used along with Concutest. Concutest is not a replacement for traditional unit testing, and it is advised that other testing be used in the context of Concutest to ensure that the unit tests themselves pass even under disparate thread schedules.

Mechanism

Concutest divides programs according to their sequence points. In the Java platform, sequence points are defined as any point at which a thread interacts with a synchronization object. For example, acquiring a mutex or releasing it would each be sequence point operations. Incrementing an integer would not be a sequence point because it is not involved in defined, protected operations.

Concutest requires that all modifiable resources in the program are protected by synchronization primitives. Concutest cannot determine certain kinds of race conditions: it can only determine whether multiple various schedules of the program result in different behaviors; it cannot determine whether multiple threads have a race condition in their accessing of a single object unless those differences show up due to the schedulings that Concutest checks.

Funding

It is partially funded by the National Science Foundation and the Texas Advanced Technology Program.

____________________WORLD TECHNOLOGIES____________________

Chapter 6

OSGi and GNU Compiler for Java

OSGi

OSGi Service Platform

Developer(s)	OSGi Alliance
Stable release	4.2 / September 2009
Operating system	Java
Type	standards organization
License	OSGi Specification License

The **OSGi framework** is a module system and service platform for the Java programming language that implements a complete and dynamic component model, something that does not exist in standalone Java/VM environments. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot; management of Java packages/classes is specified in great detail. Application life cycle management (start, stop, install, etc.) is done via APIs which allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.

The OSGi specifications have moved beyond the original focus of service gateways, and are now used in applications ranging from mobile phones to the open source Eclipse IDE. Other application areas include automobiles, industrial automation, building automation,

WORLD TECHNOLOGIES

PDAs, grid computing, entertainment (e.g. iPronto), fleet management and application servers.

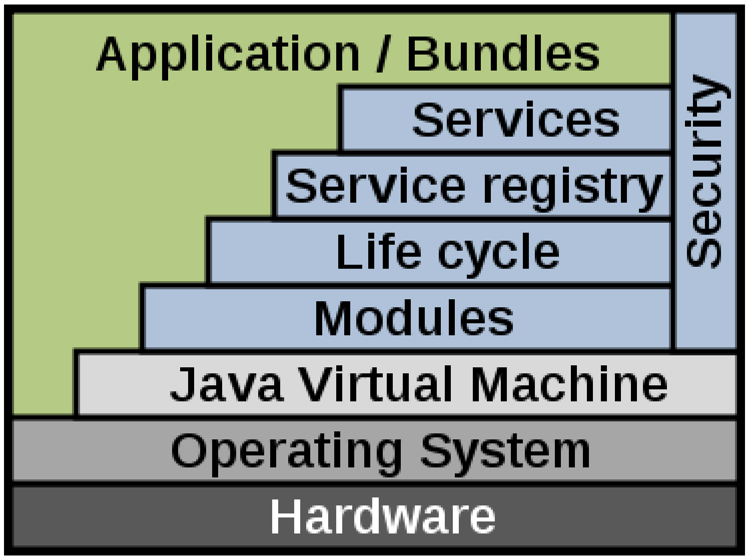
Sponsoring organization

The OSGi Alliance (formerly known as the Open Services Gateway initiative, now an obsolete name) is an open standards organization founded in March 1999 that originally specified and continues to maintain the OSGi standard.

Specification process

The OSGi specification is developed by the members in an open process and made available to the public free of charge under the OSGi Specification License . The OSGi Alliance has a compliance program that is open to members only. As of November 2010, there are seven certified OSGi framework implementations. A separate page lists both certified and non-certified OSGi Specification Implementations, which include OSGi frameworks and other OSGi specifications.

Architecture



OSGi Service Gateway Architecture

Any framework that implements the OSGi standard provides an environment for the modularization of applications into smaller bundles. Each bundle is a tightly-coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies (if any).

The framework is conceptually divided into the following areas:

Bundles
    Bundles are normal jar components with extra manifest headers.

Services
: The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects (POJO).

Services Registry
: The API for management services (ServiceRegistration, ServiceTracker and ServiceReference).

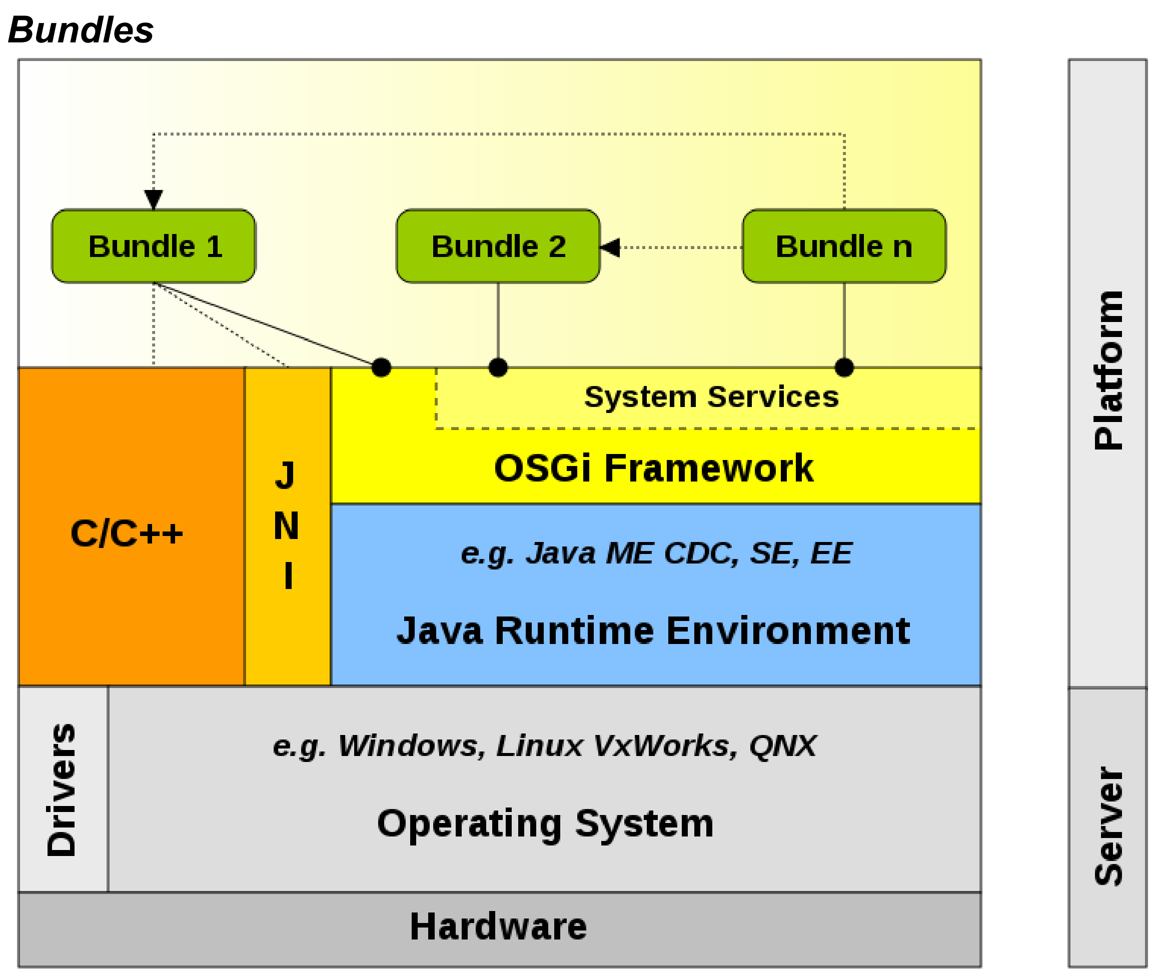
Life-Cycle
: The API for life cycle management for (install, start, stop, update, and uninstall) bundles.

Modules
: The layer that defines encapsulation and declaration of dependencies (how a bundle can import and export code).

Security
: The layer that handles the security aspects by limiting bundle functionality to pre-defined capabilities.

Execution Environment
: Defines what methods and classes are available in a specific platform. There is no fixed list of execution environments, since it is subject to change as the Java Community Process creates new versions and editions of Java. However, the following set is currently supported by most OSGi implementations:

- CDC-1.0/Foundation-1.0
- CDC-1.1/Foundation-1.1
- OSGi/Minimum-1.0
- OSGi/Minimum-1.1
- JRE-1.1
- From J2SE-1.2 up to J2SE-1.6



Classification: OSGi & System-Layering

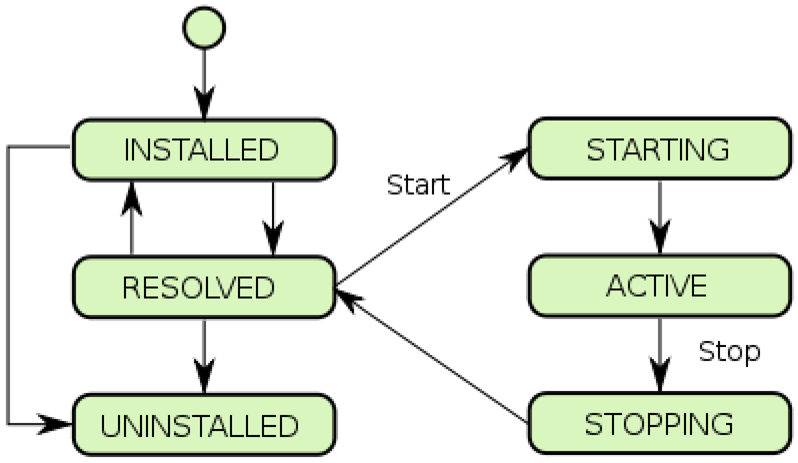
A bundle is a group of Java classes and additional resources equipped with a detailed manifest `MANIFEST.MF` file on all its contents, as well as additional services needed to give the included group of Java classes more sophisticated behaviors, to the extent of deeming the entire aggregate a component.

The meaning of the contents in the example is as follows :

- **Bundle-Name:** Defines a human-readable name for this bundle, Simply assigns a short name to the bundle.
- **Bundle-SymbolicName:** The only required header, this entry specifies a unique identifier for a bundle, based on the reverse domain name convention (used also by the java packages).
- **Bundle-Description:** A description of the bundle's functionality.
- **Bundle-ManifestVersion:** This little known header indicates the OSGi specification to use for reading this bundle.
- **Bundle-Version:** Designates a version number to the bundle.

- **Bundle-Activator:** Indicates the class name to be invoked once a bundle is activated.
- **Export-Package:** Expresses what Java packages contained in a bundle will be made available to the outside world.
- **Import-Package:** Indicates what Java packages will be required from the outside world, in order to fulfill the dependencies needed in a bundle.

Life-cycle



OSGi Bundle Life-Cycle

A Life Cycle layer adds bundles that can be dynamically installed, started, stopped, updated and uninstalled. Bundles rely on the module layer for class loading but add an API to manage the modules in run time. The lifecycle layer introduces dynamics that are normally not part of an application. Extensive dependency mechanisms are used to assure the correct operation of the environment. Life cycle operations are fully protected with the security architecture.

Bundle State	Description
INSTALLED	The bundle has been successfully installed.
RESOLVED	All Java classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or has stopped.
STARTING	The bundle is being started, the `BundleActivator.start` method will be called, and this method has not yet returned. When the bundle has an activation policy, the bundle will remain in the STARTING state

	until the bundle is activated according to its activation policy.
ACTIVE	The bundle has been successfully activated and is running; its Bundle Activator start method has been called and returned.
STOPPING	The bundle is being stopped. The `BundleActivator.stop` method has been called but the stop method has not yet returned.
UNINSTALLED	The bundle has been uninstalled. It cannot move into another state.

Services

Standard services

The OSGi Alliance has specified many services. Services are specified by a Java interface. Bundles can implement this interface and register the service with the Service Registry. Clients of the service can find it in the registry, or react to it when it appears or disappears.

The table below shows a description of OSGi System Services:

System Services	Description
Logging	The logging of information, warnings, debug information or errors is handled through the Log Service. It receives log entries and then dispatches these entries to other bundles that subscribed to this information.
Configuration Admin	This service allows an operator to set and get the configuration information of deployed bundles
Device Access	Facilitates the coordination of automatic detection and attachment of existing devices. This is used for Plug and Play scenarios.
User Admin	This service uses a database with user information (private and public) for authentication and authorization purposes.
IO Connector	The IO Connector Service implements the CDC/CLDC `javax.microedition.io` package as a service. This service allows bundles to provide new and alternative protocol schemes.
Preferences	Offers an alternative, more OSGi-friendly mechanism to using Java's default `Properties` for storing preferences.
Component Runtime	The dynamic nature of services—they can come and go at any time—makes writing software harder. The Component Runtime specification can simplify handling these dynamic aspects by providing an XML based declaration of the dependencies.
Deployment Admin	Standardizes access to some of the responsibilities of the management agent.
Event Admin	Provides an interbundle communication mechanism based on a publish-and-subscribe model.

Application Admin	Simplifies the management of an environment with many different types of applications that are simultaneously available.

The table below shows a description of OSGi Protocol Services:

Protocol Services	Description
HTTP Service	Allows information to be sent and received from OSGi using HTTP.
UPnP Device Service	Specifies how OSGi bundles can be developed to interoperate with Universal Plug and Play (UPnP) devices.
DMT Admin	Defines an API for managing a device using concepts from the Open Mobile Alliance (OMA) device management specifications.

The table below shows a description of OSGi Miscellaneous Services:

Miscellaneous Services	Description
Wire Admin	Allows the connection between a Producer service and a Consumer service.
XML Parser	The XML Parser service allows a bundle to locate a parser with desired properties and compatibility with JAXP.
Measurement and State	The Measurement and State service allows and simplifies the correct handling of measurements in an OSGi service platform.

Organization

The OSGi Alliance was founded by Ericsson, IBM, Motorola, Sun Microsystems and others in March 1999 (before incorporating as a nonprofit corporation it was called the Connected Alliance).

Among its members are (as of May 2007) more than 35 companies from quite different business areas, for example IONA Technologies, Ericsson, Deutsche Telekom, IBM, Makewave (formerly Gatespace Telematics), Motorola, Nokia, NTT, Oracle, ProSyst, Red Hat, Samsung Electronics, Siemens, VMware (SpringSource), and Telefonica.

The Alliance has a Board of Directors which provides the organization's overall governance. OSGi Officers have various roles and responsibilities in supporting the Alliance. Technical work is conducted within Expert Groups (EGs) chartered by the Board of Directors, and non-technical work is conducted in various Working Groups and Committees. The technical work conducted within Expert Groups include developing specifications, reference implementations, and compliance tests. These Expert Groups have produced four major releases of the OSGi specifications (as of 2007).

There are dedicated Expert Groups for the Enterprise, Mobile, Vehicle and the Core Platform areas.

The Enterprise Expert Group (EEG) is the newest EG and is addressing Enterprise / Server-side applications. In November 2007 the Residential Expert Group (REG) started to work on specifications to remotely manage residential/home-gateways. In October 2003, Nokia, Motorola, IBM, ProSyst and other OSGi members formed a Mobile Expert Group (MEG) that will specify a MIDP-based service platform for the next generation of smart mobile phones, addressing some of the needs that CLDC cannot manage - other than CDC. MEG became part of OSGi as with R4.

Community

Also in 2003 Eclipse selected OSGi as the underlying runtime for the plug-in architecture used for the Eclipse Rich Client Platform and the IDE platform. Eclipse itself includes sophisticated tooling for developing OSGi bundles and there are a number of other Eclipse plug-ins aimed at supporting OSGi behaviour (e.g. both ProSyst and Knopflerfish have Eclipse plug-ins available specifically for OSGi developers).

There is a vibrant free software community revolving around the OSGi specification. Some widely-used open source implementations are Equinox OSGi, Apache Felix, Knopflerfish OSGi project as well as the mBedded Server Equinox Edition. Regarding tooling, build system support and testing, the OPS4J Pax projects provide a lot of useful components and expertise.

Specification versions

- OSGi Release 1 (R1): May 2000
- OSGi Release 2 (R2): October 2001
- OSGi Release 3 (R3): March 2003
- OSGi Release 4 (R4): October 2005 / September 2006
    - Core Specification (R4 Core): October 2005
    - Mobile Specification (R4 Mobile / JSR-232): September 2006
- OSGi Release 4.1 (R4.1): May 2007 (AKA JSR-291)
- OSGi Release 4.2 (R4.2): September 2009

New in OSGi Release 4

The new features of OSGi R4 in brief are as follows:

- New modularization capabilities providing enhanced encapsulation of networked services that can share a single Virtual Machine (VM).
- Modularized class sharing and hiding of implementation details.
- Methods for handling multiple versions of the same classes so old and new applications can execute within the same VM.
- Localization of OSGi bundle manifests enabling service deployment anywhere.

- Enhancements in security and policies: The new Conditional Permission Admin service provides an elegant and simple way to manage networked services securely. It also supports dynamic policies that can depend on external (custom) conditions. Combined with R4 support for digital signatures, this provides a central security solution to large deployments of products using the OSGi Service Platform.
- A Declarative Services specification that addresses memory footprint issues that can prevent small embedded devices from using a service oriented architecture to support multiple applications. Additionally, it significantly simplifies the service-oriented programming model by declaratively handling the dynamics of services.
- Compatibility with Release 3, requiring no changes for existing OSGi bundles, applications, or services.

New in Release 4.1

OSGi R4.1 was a minor revision intended to clarify certain aspects of bundle initialization and loading in order to improve third party usage. It added no new services or major features.

New in Release 4.2

OSGi R4.2 was a significant release which added several new services and capabilities, including:

- **Framework launching**: Standardized means to launch OSGi from various providers
- **Remote Services**: Allows the exporting of services to remote VMs (formerly known as Distribute OSGi)
- **Blueprint Service**: Dependency injection and inversion of control (similar to Spring) that allows external configuration of bundle dependencies
- **Bundle Tracker**: Track and respond to changes in bundle presence and state
- **Service Hooks**: Allow introspection and behavior modification of service calls to inject security or dynamicism
- **Conditional Permissions**: Support negative permissions, forbidding specific actions instead of just allowing them

More information can also be specified in each bundle header, such as license information, MIME types and icons. Additionally, changes to Declarative Services allow the easier setting of permissions. Finally, OSGi bundles can now have their return values read.

Related RFCs and Java Specifications

- RFC-2608 (Service Location Protocol)
- Sun JINI (Java Intelligent Network Infrastructure)
- Sun JCP JSR-8 (Open Services Gateway Specification)

- Sun JCP JSR-232 (Mobile Operational Management)
- Sun JCP JSR-246 (Device Management API)
- Sun JCP JSR-249 (Mobile Service Architecture for CDC)
- Sun JCP JSR-277 (Java Module System)
- Sun JCP JSR-291 (Dynamic Component Support for Java SE - AKA OSGi 4.1)
- Sun JCP JSR-294 (Improved Modularity Support in the Java Programming Language)

GNU Compiler for Java

GNU Compiler for Java

Developer(s)	The GNU Project
Stable release	4.5.1 / July 31, 2010; 4 months ago
Operating system	Unix-like
Type	Compiler
License	GNU GPL

The **GNU Compiler for Java** (**GCJ** or **gcj**) is a free software compiler for the Java programming language and a part of the GNU Compiler Collection.

GCJ can compile Java source code to either Java Virtual Machine bytecode, or directly to machine code for any of a number of CPU architectures. It can also compile class files containing bytecode or entire JARs containing such files into machine code.

History

Almost all of the runtime-libraries used by gcj come from the GNU Classpath project (but compare the `libgcj` library). As of gcj 4.3, gcj integrates with ecj, the Eclipse Compiler for Java.

As of 2007 a lot of work has gone in to getting GNU Classpath to support Java's two graphical APIs: AWT and Swing. Work on supporting AWT is ongoing, after which support for Swing will follow.

As of 2009 there have been no new developments announced from gcj. The product is currently in maintenance mode.

Performance

Java code compiled into machine code by GCJ should have faster start-up time than the equivalent bytecode launched in a JVM.

However, after start-up, Java code compiled by GCJ does not necessarily execute any faster than bytecode executed by a modern JIT-enabled JVM. This is true even when GCJ is invoked with advanced optimization options such as *-fno-bounds-check -O3 -mfpmath=sse -msse2 -ffast-math -march=native* -- in this case, the compiled program may or may not surpass JVM performance, depending on the operations performed by the code in question.

CNI (Compiled Native Interface)

The CNI (**Compiled Native Interface**, previously Cygnus Native Interface), a software framework for the gcj, allows Java code to call and be called by native applications (programs specific to a hardware and operating-system platform) and libraries written in C++.

CNI closely resembles the Java Native Interface (JNI) framework which comes as standard with various Java virtual machines. However the CNI authors claim various advantages over JNI:

> “ We use CNI because we think it is a better solution, especially for a Java implementation that is based on the idea that Java is just another programming language that can be implemented using standard compilation techniques. Given that, and the idea that languages implemented using Gcc should be compatible where it makes sense, it follows that the Java calling convention should be as similar as practical to that used for other languages, especially C++, since we can think of Java as a subset of C++. CNI is just a set of helper functions and conventions built on the idea that C++ and Java have the *same* calling convention and object layout; they are binary compatible. (This is a simplification, but close enough.) ”

CNI depends on Java classes appearing as C++ classes. For example, given a Java class,

```
public class Int {
   public int i;
   public Int (int i) { this.i = i; }
   public static Int zero = new Int(0);
}
```

one can use the class thus:

```
#include <gcj/cni.h>
#include <Int>

Int* mult(Int* p, jint k)
{
  if (k == 0)
    return Int::zero;  // Static member access.
  return new Int(p->i * k);
}
```

Chapter 7

Grails and Griffon (Framework)

Grails (framework)

Grails

Stable release	1.3.5 / October 4, 2010; 2 months ago
Written in	Groovy
Operating system	Cross-platform
Platform	Cross-platform (JVM)
Type	Web application framework
License	Apache License 2.0

Grails is an open source web application framework which uses the Groovy programming language (which is in turn based on the Java platform). It is intended to be a high-productivity framework by following the "coding by convention" paradigm, providing a stand-alone development environment and hiding much of the configuration detail from the developer.

Grails was previously known as 'Groovy on Rails'; in March 2006 that name was dropped in response to a request by David Heinemeier Hansson, founder of the Ruby on Rails framework. Work began in July 2005, with the 0.1 release on March 29, 2006 and the 1.0 release announced on February 18, 2008.

G2One - The Groovy Grails Company - was acquired by SpringSource in November, 2008, and it was later acquired by VMware.

Overview

Grails has been developed with a number of goals in mind:

- Provide a high-productivity web framework for the Java platform.
- Re-use proven Java technologies such as Hibernate and Spring under a simple, consistent interface
- Offer a consistent framework which reduces confusion and is easy to learn.
- Offer documentation for those parts of the framework which matter for its users.
- Provide what users expect in areas which are often complex and inconsistent:
    - Powerful and consistent persistence framework.
    - Powerful and easy to use view templates using GSP (Groovy Server Pages).
    - Dynamic tag libraries to easily create web page components.
    - Good Ajax support which is easy to extend and customize.
- Provide sample applications which demonstrate the power of the framework.
- Provide a complete development mode, including web server and automatic reload of resources.

Grails has been designed to be easy to learn, easy to develop applications and extensible. It attempts to offer the right balance between consistency and powerful features.

High productivity

Grails has three properties which attempt to increase productivity when compared to traditional Java web frameworks:

- No XML configuration
- Ready-to-use development environment
- Functionality available through mixins

No XML configuration

Creating web applications in Java traditionally involves configuring environments and frameworks at the start and during development. This configuration is very often externalized in XML files to ease configuration and avoid embedding configuration in application code.

XML was initially welcomed as it provided greater consistency to configure applications. In recent years however it has become apparent that although XML is great for configuration it can be tedious to set up an environment. This may take away productivity as developers spend time understanding and maintaining framework configuration as the application grows. Adding or changing functionality in applications which use XML configuration adds an extra step to the change process next to writing application code which slows down productivity and may take away the agility of the entire process.

Grails takes away the need to add configuration in XML files. Instead the framework uses a set of rules or conventions while inspecting the code of Grails-based applications. For example, a class name which ends with `Controller` (for example `BookController`) is considered a web controller.

Ready-to-use development environment

When using traditional Java web toolkits, it's up to developers to assemble development units, which can be tedious. Grails comes with a complete development environment which includes a web server to get developers started right away. All required libraries are part of the Grails distribution and Grails prepares the Java web environment for deployment automatically.

Functionality available through mixins

Grails features dynamic methods on several classes through mixins. A mixin is a method which is added to a class dynamically as if the functionality was compiled in the program.

These dynamic methods allow developers to perform operations without having to implement interfaces or extend base classes. Grails provides dynamic methods based on the type of class. For example domain classes have methods to automate persistence operations like save, delete and find.

Web framework

The Grails web framework has been designed according to the MVC paradigm.

Controllers

Grails uses controllers to implement the behaviour of web pages. Below is an example of a controller:

```
class BookController {
   def list = {
      [ books: Book.findAll() ]
   }
}
```

The controller above has a `list` action which returns a model containing all books in the database. To create this controller the `grails` command is used, as shown below:

```
grails create-controller
```

This command requests the controller name and creates a class in the `grails-app/controller` directory of the Grails project. Creating the controller class is sufficient to have it recognized by Grails. The `list` action maps to `http://localhost:8080/book/list` in development mode.

Views

Grails supports JSP and GSP. The example below shows a view written in GSP which lists the books in the model prepared by the controller above:

```
<html>
  <head>
    <title>Our books</title>
  </head>
  <body>
    <ul>
      <g:each in="${books}">
        <li>${it.title} (${it.author.name})</li>
      </g:each>
    </ul>
  </body>
</html>
```

This view should be saved as `grails-app/views/book/list.gsp` of the Grails project. This location maps to the `BookController` and `list` action. Placing the file in this location is sufficient to have it recognized by Grails.

There is also a GSP tag reference available.

Ajax support

Grails supports several Ajax libraries including OpenRico, Prototype, Dojo, YUI and ZKGrails. You can use existing tag libraries which create HTML with Ajax code. You can also easily create your own tag libraries.

Dynamic tag libraries

Grails provides a large number of tag libraries out of the box. However you can also create and reuse your own tag libraries easily:

```
 def formatDate = { attrs ->
    out << new
java.text.SimpleDateFormat(attrs.format).format(attrs.date)
 }
```

The `formatDate` tag library above formats a `java.util.Date` object to a `String`. This tag library should be added to the `grails-app/taglib/ApplicationTagLib.groovy` file or a file ending with `TagLib.groovy` in the `grails-app/taglib` directory.

Below is a snippet from a GSP file which uses the `formatDate` tag library:

```
<g:formatDate format="yyyyMMdd" date="${myDate}"/>
```

To use a dynamic tag library in a GSP no import tags have to be used. Dynamic tag libraries can also be used in JSP files although this requires a little more work.

Persistence

Model

The domain model in Grails is persisted to the database using GORM (Grails Object Relational Mapping). Domain classes are saved in the `grails-app/domain` directory and can be created using the `grails` command as shown below:

```
grails create-domain-class
```

This command requests the domain class name and creates the appropriate file. Below the code of the `Book` class is shown:

```
 class Book {
    String title
    Person author
 }
```

Creating this class is all that is required to have it managed for persistence by Grails. With Grails 0.3, GORM has been improved and e.g. adds the properties id and version itself to the domain class if they are not present.

Methods

The domain classes managed by GORM have 'magic' dynamic and static methods to perform persistence operations on these classes and its objects.

Dynamic Instance Methods

The `save()` method saves an object to the database:

```
 def book = new Book(title:"The Da Vinci Code",
author:Author.findByName("Dan Brown"))
 book.save()
```

The `delete()` method deletes an object from the database:

```
 def book = Book.findByTitle("The Da Vinci Code")
 book.delete()
```

The `refresh()` method refreshes the state of an object from the database:

```
 def book = Book.findByTitle("The Da Vinci Code")
```

```
book.refresh()
```

The `ident()` method retrieves the object's identity assigned from the database:

```
def book = Book.findByTitle("The Da Vinci Code")
def id = book.ident()
```

Dynamic Static (Class) methods

The `count()` method returns the number of records in the database for a given class:

```
def bookCount = Book.count()
```

The `exists()` method returns true if an object exists in the database with a given identifier:

```
def bookExists = Book.exists(1)
```

The `find()` method returns the first object from the database based on an object query statement:

```
def book = Book.find("from Book b where b.title = ?", [ 'The Da Vinci
Code' ])
```

Note that the query syntax is Hibernate HQL.

The `findAll()` method returns all objects existing in the database:

```
def books = Book.findAll()
```

The `findAll()` method can also take an object query statement for returning a list of objects:

```
def books = Book.findAll("from Book")
```

The `findBy*()` methods return the first object from the database which matches a specific pattern:

```
def book = Book.findByTitle("The Da Vinci Code")
```

Also:

```
def book = Book.findByTitleLike("%Da Vinci%")
```

The `findAllBy*()` methods return a list of objects from the database which match a specific pattern:

```
def books = Book.findAllByTitleLike("The%")
```

The `findWhere*()` methods return the first object from the database which matches a set of named parameters:

```
def book = Book.findWhere(title:"The Da Vinci Code")
```

Scaffolding

Grails supports scaffolding to support CRUD operations (Create, Read, Update, Delete). Any domain class can be scaffolded by creating a scaffolding controller as shown below:

```
class BookController {
   def scaffold = true
}
```

By creating this class you can perform CRUD operations on `http://localhost:8080/book`. Currently Grails does not provide scaffolding for associations.

Legacy Database Models

The persistence mechanism in GORM is implemented via Hibernate. As such, legacy databases may be mapped to GORM classes using standard Hibernate mapping files.

Creating a Grails project

Download and installation guidelines for Grails are available on the Grails web site .

Grails provides support scripts to create and execute projects as follows:

- Grails will create a complete outline application in response to the command

```
grails create-app
```

This command will request the name of the project and creates a project directory with the same name. Further `grails` commands can be issues in this directory to create the classes and web pages of the application.

At the command `grails run-app` Grails will run the application on a webserver at the `http://localhost:8080/` URL.

Target audience

The target audience for Grails is:

- Java developers who are looking for an integrated development environment to create web based applications.

- Developers without Java experience looking for a high-productivity environment to build web based applications.

Integration with the Java platform

Grails is built on top of and is part of the Java platform meaning it's very easy to integrate with Java libraries, frameworks and existing code bases. The most prominent feature Grails offers in this area is the transparent integration of classes which are mapped with the Hibernate ORM framework. This means existing applications which use Hibernate can use Grails without recompiling the code or reconfiguring the Hibernate classes while using the dynamic persistence methods discussed above.

One consequence of this is that scaffolding can be configured for Java classes mapped with Hibernate. Another consequence is that the capabilities of the Grails web framework are fully available for these classes and the applications which use them.

Griffon (framework)

Griffon

Original author(s)	Danno Ferrin, Andres Almiray, James Williams
Initial release	September 19, 2008
Stable release	0.9.2-beta-2 / November 23, 2010; 18 days ago
Written in	Groovy
Operating system	Cross-platform
Platform	Cross-platform (JVM)
Available in	English
Type	Rich Client Platform
License	Apache License 2.0

Griffon is an open source Rich Client Platform framework which leverages the Groovy programming language (which is in turn based on the Java platform). Griffon is intended to be a high-productivity framework by rewarding use of the Model-View-Controller paradigm, providing a stand-alone development environment and hiding much of the configuration detail from the developer. A significant portion of the build environment is directly derived from the Grails codebase and hence follows many of its conventions.

The first release is the fruit of the effort by the Groovy Swing team and an attempt to take the best of rapid application development, as indicated by its Grails-like structure, the agility of the Groovy, and the availability of components for Swing.

Overview

Griffon aims to reduce the typical confusion that occurs with traditional Swing development. Due to the MVC structure of Griffon, developers never have to go searching for files or be confused on how to start a new project. Everything begins with:

```
griffon create-app <APP_NAME>
```

The generated project follows this structure:

```
%PROJECT_HOME%
    + griffon-app
       + conf                ---> location of configuration artifacts
like builder configuration
           + keys                  ---> keys for code signing
           + webstart              ---> webstart and applet config
       + controllers         ---> location of controller classes
       + i18n                ---> location of message bundles for i18n
       + lifecycle           ---> location of lifecycle scripts
       + models              ---> location of model classes
       + resources           ---> location of non code resources
(images, etc)
       + views               ---> location of view classes
   + lib
   + scripts                 ---> scripts
   + src
       + main                ---> optional; location for Groovy and
Java source files
                                  (of types other than those in
griffon-app/*)
```

The builder infrastructure enables seamless integration of different widget libraries such as Swing, JIDE, and SwingX.

Griffon’s built-in scripts include targets for desktop, webstart, and applets. The baseline requirement is Java 5 or higher.

In the first release, three sample applications are included :

- Greet, a Groovy Twitter client featured in the JavaOne 2009 Script Bowl,
- FontPicker, an application to view the available fonts on one's machine,
- SwingPad, a lightweight designer application for Griffon user interfaces.

Plugins

Griffon can be extended with the use of plugins. Plugins provide run-time access to testing libraries such as Easyb and FEST, and all widget libraries besides core Swing are provided as plugins. The plugin system allows for a wide range of additions, for example

- Polyglot Programming with Clojure, Scala, JavaFX and Erlang.
- Additional UI toolkits - SWT, JavaFX, Pivot, GTK
- SQL and NoSQL datastores like Berkleydb, CouchDB, Db4O, Neo4j, NeoDatis, Memcached and Riak.

Documentation

Official manual

The Griffon Guide (available in PDF too) is the official manual of the Griffon framework. It describes all that a developer needs to know to get started, including a reference to all properties and method additions.

Books

Features that would eventually become integral parts of Griffon (UI builders) were featured in these books:

- Groovy In Action
- Beginning Groovy and Grails

Books currently in development include:

- Griffon In Action

Magazine

- GroovyMag for Groovy and Grails developers

Refcard

Dzone published a Getting started with Griffon Refcard as part of their Refcardz program.

Screencasts

Todd Costella produced a series of screencasts dubbed Griffoncast that show how to get started with Griffon.

Twitter

News, links and announcements of new releases and features are regularly posted on Twitter (@theaviary).

Chapter 8

Hibernate (Java) and JDBC Driver

Hibernate (Java)

Hibernate

Developer(s)	Red Hat
Stable release	3.6.0 Final / October 13, 2010; 58 days ago
Development status	Active
Written in	Java
Operating system	Cross-platform (JVM)
Platform	Java Virtual Machine
Type	Object-relational mapping
License	GNU Lesser General Public License

Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions.

Hibernate is free software that is distributed under the GNU Lesser General Public License.

Hibernate's primary feature is mapping from Java classes to database tables (and from Java data types to SQL data types). Hibernate also provides data query and retrieval facilities. Hibernate generates the SQL calls and relieves the developer from manual result set handling and object conversion, keeping the application portable to all supported SQL databases, with database portability delivered at very little performance overhead.

Mapping

Mapping Java classes to database tables is accomplished through the configuration of an XML file or by using Java Annotations. When using an XML file, Hibernate can generate skeletal source code for the persistence classes. This is unnecessary when annotation is used. Hibernate can use the XML file or the annotation to maintain the database schema.

Facilities to arrange one-to-many and many-to-many relationships between classes are provided. In addition to managing association between objects, Hibernate can also manage reflexive associations where an object has a one-to-many relationship with other instances of its own type.

Hibernate supports the mapping of custom value types. This makes the following scenarios possible:

- Overriding the default SQL type that Hibernate chooses when mapping a column to a property.
- Mapping Java Enum to columns as if they were regular properties.
- Mapping a single property to multiple columns.

Persistence

Hibernate provides transparent persistence for Plain Old Java Objects (POJOs). The only strict requirement for a persistent class is a no-argument constructor, not necessarily *public*. Proper behavior in some applications also requires special attention to the *equals()* and *hashCode()* methods.

Collections of data objects are typically stored in Java collection objects such as Set and List. Java generics, introduced in Java 5, are supported. Hibernate can be configured to lazy load associated collections. Lazy loading is the default as of Hibernate 3.

Related objects can be configured to *cascade* operations from one to the other. For example, a parent such as an Album object can be configured to cascade its save and/or delete operation to its child Track objects. This can reduce development time and ensure referential integrity. A *dirty checking* feature avoids unnecessary database write actions by performing SQL updates only on the modified fields of persistent objects.

Hibernate Query Language (HQL)

Hibernate provides an SQL inspired language called Hibernate Query Language (HQL) which allows SQL-like queries to be written against Hibernate's data objects. *Criteria Queries* are provided as an object-oriented alternative to HQL.

Integration

Hibernate can be used both in standalone Java applications and in Java EE applications using servlets or EJB session beans. It can also be included as a feature in other programming languages. For example, Adobe integrated Hibernate into version 9 of ColdFusion (which runs on J2EE app servers) with an abstraction layer of new functions and syntax added into CFML.

Entities and components

In Hibernate jargon, an *entity* is a stand-alone object in Hibernate's persistent mechanism which can be manipulated independently of other objects. In contrast, a *component* is subordinate to other entities and can be manipulated only with respect to other entities. For example, an Album object may represent an entity but the Tracks object associated with the Album objects would represent a *component* of the Album entity if it is assumed that Tracks can only be saved or retrieved from the database through the Album object.

History

Hibernate was started in 2001 by Gavin King as an alternative to using EJB2-style entity beans. Its mission back then was to simply offer better persistence capabilities than offered by EJB2 by simplifying the complexities and allowing for missing features.

Early in 2003, the Hibernate development team began Hibernate2 releases which offered many significant improvements over the first release and would go on to catapult HIbernate as the "de facto" standard for persistence in Java...

JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers and worked with them in supporting Hibernate.

As of 2010 the current version of Hibernate is Version 3.x. This version introduced new features like a new Interceptor/Callback architecture, user defined filters, and JDK 5.0 Annotations (Java's metadata feature). As of 2010 Hibernate 3 (version 3.5.0 and up) is a certified implementation of the Java Persistence API 2.0 specification via a wrapper for the Core module which provides conformity with the JSR 317 standard.

Application programming interface

The Hibernate API is provided in the Java package org.hibernate.

org.hibernate.SessionFactory interface

References immutable and threadsafe object creating new Hibernate sessions. Hibernate-based applications are usually designed to make use only of a single instance of the class implementing this interface (often exposed using a singleton design pattern).

org.hibernate.Session interface

Represents a Hibernate session i.e. the main point of the manipulation performed on the database entities. The latter activities include (among the other things) managing the persistence state (transient, persisted, detached) of the objects, fetching the persisted ones from the database and the management of the transaction demarcation.

A session is intended to last as long as the logical transaction on the database. Due to the latter feature Session implementations are not expected to be threadsafe nor to be used by multiple clients.

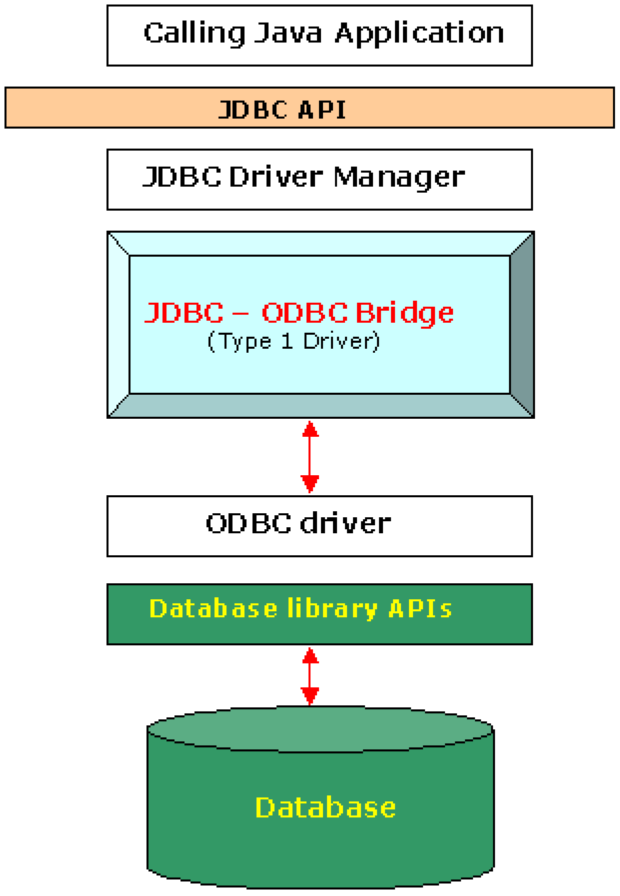
JDBC driver

A **JDBC driver** is a software component enabling a Java application to interact with a database.

To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

JDBC technology drivers fit into one of four categories.

Type 1 Driver - JDBC-ODBC bridge



Schematic of the JDBC-ODBC bridge

The JDBC type 1 driver, also known as the JDBC-ODBC bridge, is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls.

The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon. Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver. The use of this driver is discouraged if the alternative of a pure-Java driver is available. The other implication is that any application using a type 1 driver is non-portable given the binding between the driver and platform. This technology isn't suitable for a high-transaction environment. Type 1 drivers also don't support the complete Java command set and are limited by the functionality of the ODBC driver.

Functions

- Translates query obtained by JDBC into corresponding ODBC query, which is then handled by the ODBC driver.
- Sun provides a JDBC-ODBC Bridge driver. sun.jdbc.odbc.JdbcOdbcDriver. This driver is native code and not Java, and is closed source.
- Client -> JDBC Driver -> ODBC Driver -> Database

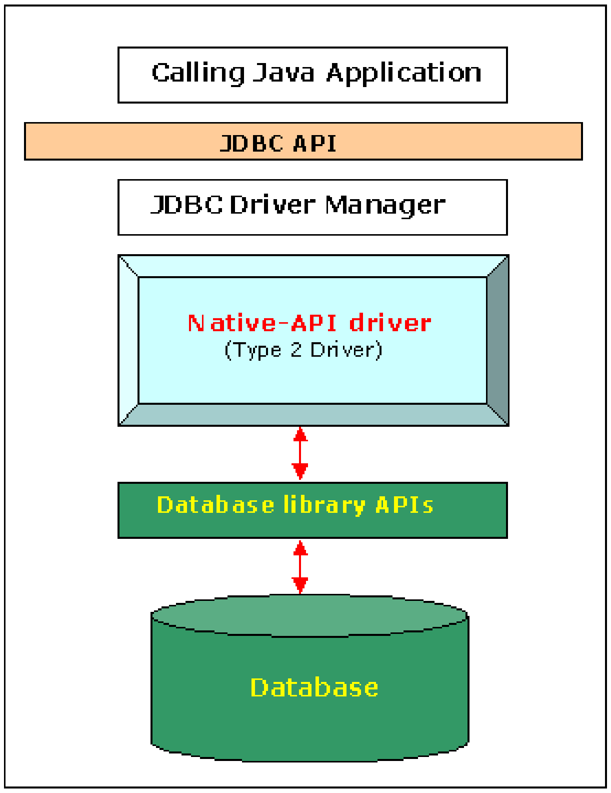
Advantages

- Easy to connect.
- directly connected to the database

Disadvantages

- Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native db connectivity interface.
- The ODBC driver needs to be installed on the client machine.
- Considering the client-side software needed, this is not suitable for applets.

Type 2 Driver - Native-API Driver specification



Schematic of the Native API driver

The JDBC type 2 driver, also known as the Native-API driver, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

The type 2 driver is not written entirely in Java as it interfaces with non-Java code that makes the final database calls. The driver is compiled for use with the particular operating system. For platform interoperability, the Type 4 driver, being a full-Java implementation, is preferred over this driver.

However the type 2 driver provides more functionality and better performance than the type 1 driver as it does not have the overhead of the additional ODBC function calls.

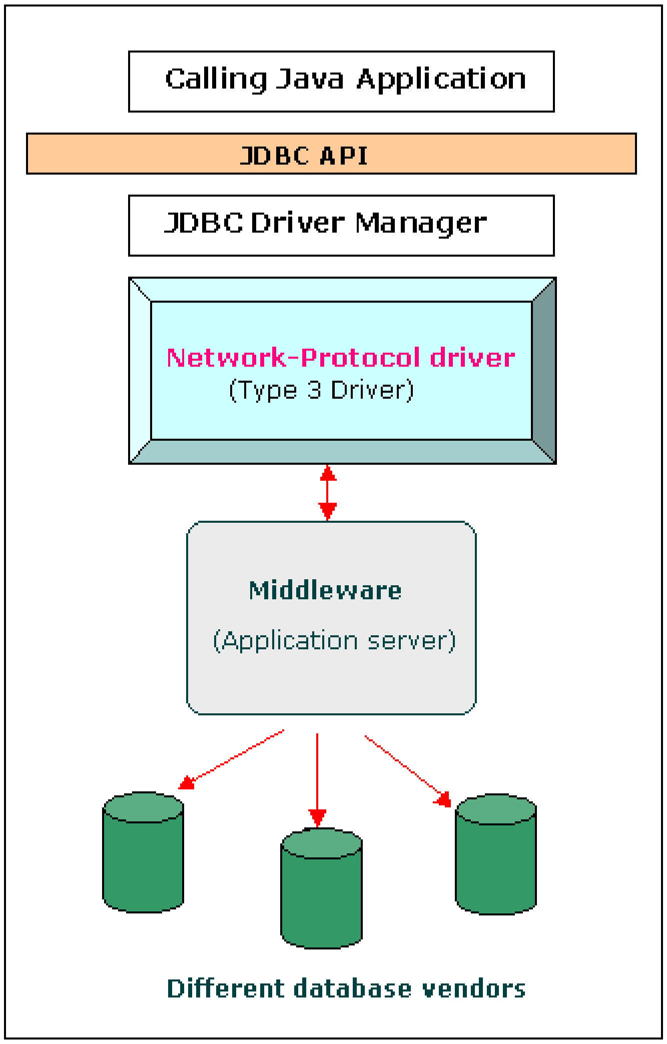
Advantages

- Better performance than Type 1 Driver (JDBC-ODBC bridge).
- Provides Fastest performance than all 3 drivers as it calls native APIs(MySQL,Oracle...etc).

Disadvantages

- The vendor client library needs to be installed on the client machine.
- Not all databases have a client side library
- This driver is platform dependent
- This driver supports all java applications except Applets

Type 3 Driver - Network-Protocol Driver



Schematic of the Network Protocol driver

The JDBC type 3 driver, also known as the Pure Java Driver for Database Middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

This differs from the type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier. Like type 4 drivers, the type 3 driver is written entirely in Java. The same driver can be used for multiple databases. It depends on the number of databases the middleware has been configured to support. The type 3 driver is platform-independent as the platform-related differences are taken care by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.

Functions

- Follows a three tier communication approach.
- Can interface to multiple databases - Not vendor specific.
- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
- Thus the client driver to middleware communication is database independent.
- Client -> JDBC Driver -> Middleware-Net Server -> Any Database,...

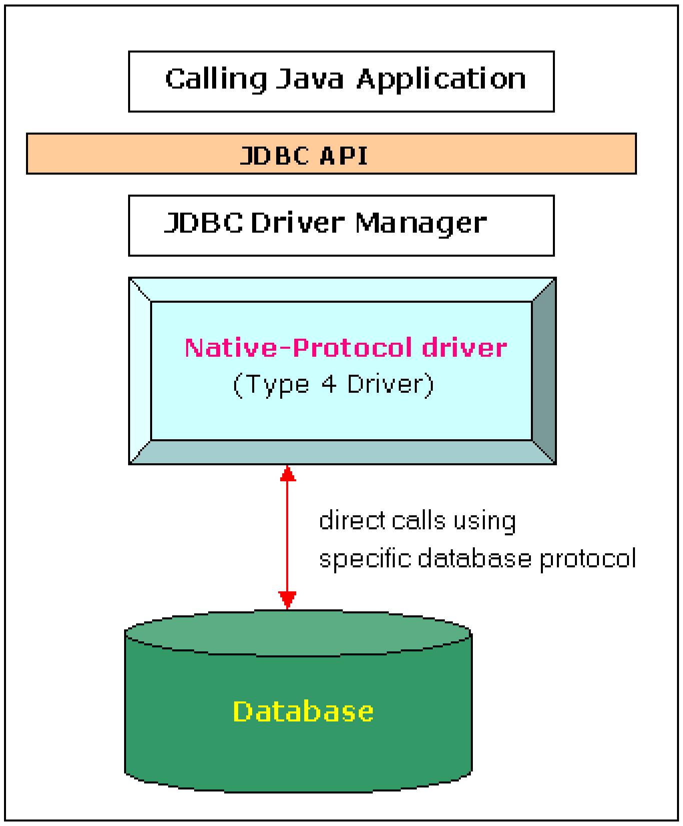
Advantages

- Since the communication between client and the middleware server is database independent, there is no need for the vendor db library on the client machine. Also the client to middleware need not be changed for a new database.
- The Middleware Server (which can be a full fledged J2EE Application server) can provide typical middleware services like caching (connections, query results, and so on), load balancing, logging, auditing etc.
- Eg. for the above include jdbc driver features in Weblogic.
    - Can be used in internet since there is no client side software needed.
    - At client side a single driver can handle any database. (It works provided the middleware supports that database!)

Disadvantages

- Requires database-specific coding to be done in the middle tier.
- An extra layer added may result in a time-bottleneck. But typically this is overcome by providing efficient middleware services.

Type 4 Driver - Native-Protocol Driver



Schematic of the Native-Protocol driver

The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol.

Written completely in Java, type 4 drivers are thus platform independent. They install inside the Java Virtual Machine of the client. This provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the type 3 drivers, it does not need associated software to work.

As the database protocol is vendor-specific, the JDBC client requires separate drivers, usually vendor-supplied, to connect to different types of databases.

Functions

- Type 4 drivers, coded entirely in Java, communicate directly with a vendor's database, usually through socket connections. No translation or middleware layers are required, improving performance.

- The driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server.
- Completely implemented in Java to achieve platform independence.
- This type includes (for example) the widely-used Oracle thin driver - oracle.jdbc.driver.OracleDriver which connects using a format configuration of jdbc:oracle:thin:@URL
- Client -> Native-protocol JDBC Driver -> database server

Advantages

- These drivers don't translate the requests into an intermediary format (such as ODBC), nor do they need a middleware layer to service requests. This can enhance performance considerably.
- The JVM can manage all aspects of the application-to-database connection; this can facilitate debugging.

Disadvantages

- Clients require a separate driver for each database.
- Drivers are database dependent.

Chapter 9

Apache Harmony and Dalvik (Software)

Apache Harmony

Apache Harmony	
Developer(s)	Apache Software Foundation
Stable release	5.0M15 6.0M3 / September 15, 2010; 2 months ago
Development status	Active
Written in	C++ and Java
Operating system	Windows and Linux
Type	Java Virtual Machine and Java Library
License	Apache License 2.0

Apache Harmony is an open source / free Java implementation from the Apache Software Foundation, starting with Java SE 5 and 6. It will be licensed under the Apache License, Version 2. It was announced in early May 2005 and on October 25, 2006, the Board of Directors voted to make Apache Harmony a top-level project.

History

Initiation

The Harmony project was initially conceived as an effort to unite all developers of the Free Java implementations. Many developers expected that it would be the project above the GNU, Apache and other communities. GNU developers were invited into and participated during the initial, preparatory planning.

Incompatibility with GNU Classpath

Despite the impression given by the preparatory planning, it was decided not to use the code from GNU Classpath, and that Harmony would use an incompatible license; therefore blocking the collaboration between Harmony and existing free Java projects. Apache developers would then write the needed classes from scratch and expect necessary large code donations from software companies. Various misunderstandings at the beginnings of the project, and the fact that major companies like IBM proposed to give large amount of existing code, created some confusion in the free Java community about the real objectives of the project

One major point of incompatibility between the GNU Classpath and Apache Harmony projects was their incompatible licenses: Classpath's GNU General Public License with the linking exception versus Harmony's Apache License

Difficulties to obtain a TCK license from Sun

On April 10, 2007, the Apache Software Foundation sent a letter to Sun Microsystems CEO, Jonathan Schwartz regarding their inability to acquire an acceptable license for the Java SE 5 Technology Compatibility Kit (TCK), a test kit needed by the project to demonstrate compatibility with the Java SE 5 specification, as required by the Sun specification license for Java SE 5. What makes the license unacceptable for ASF is the fact that it imposes rights restrictions through limits on the "field of use" available to users of Harmony, not compliant with the Java Community Process rules.

Sun answered on a company blog that it intended to create an open source implementation of the Java platform under GPL, including the TCK, but that their current priority was to make the Java Platform accessible to the GNU/Linux community under GPL as quickly as possible.

This answer triggered some reactions, either criticizing Sun for not responding "in a sufficiently open manner" to an open letter , or rather Apache Software Foundation; some think that ASF acted unwisely to aggressively demand something they could have obtained with more diplomacy from Sun, especially considering the timescale of the opening class library .

Since Sun's release of OpenJDK, Sun has released a specific license to allow to run the TCK in the OpenJDK context for any GPL implementation deriving substantially from OpenJDK.

On December 9, 2010, the Apache Software Foundation resigned from the Java Community Process Executive Committee , in protest over the difficulty in obtaining a license acceptable to Apache for use with the Harmony project.

Use in Android SDK

Dalvik, the Virtual Machine used in Google's Android platform, uses a subset of Harmony for the core of its Class Library. However, Dalvik does not align to Java SE nor Java ME Class Library profiles (for example J2ME classes, AWT and Swing are not supported). Instead it uses its own library, built on top of a subset of Harmony.

Disengagement from IBM

On 11 October 2010, IBM, by far the biggest participant in the project, decided to join Oracle on the OpenJDK project, effectively shifting its efforts from Harmony to the Oracle reference implementation. Bob Sutor, IBM's head of Linux and open source, blogged that "IBM will be shifting its development effort from the Apache Project Harmony to OpenJDK"..

Development team

In the beginning Apache Harmony received some large code contributions from several companies. Development discussions have taken place on open mailing lists. Later the Apache Software foundation mentors put a lot of effort into bringing the development process more in line with "the Apache way," and it seems that their efforts were highly successful. In November 1, 2006, the current team of committers consisted of 16 developers, 12 of them from IBM and Intel.

Recent development status

Apache Harmony was accepted among the official Apache projects on 29 October 2006.

Architecture

The Dynamic Runtime Layer virtual machine consists of the following components:

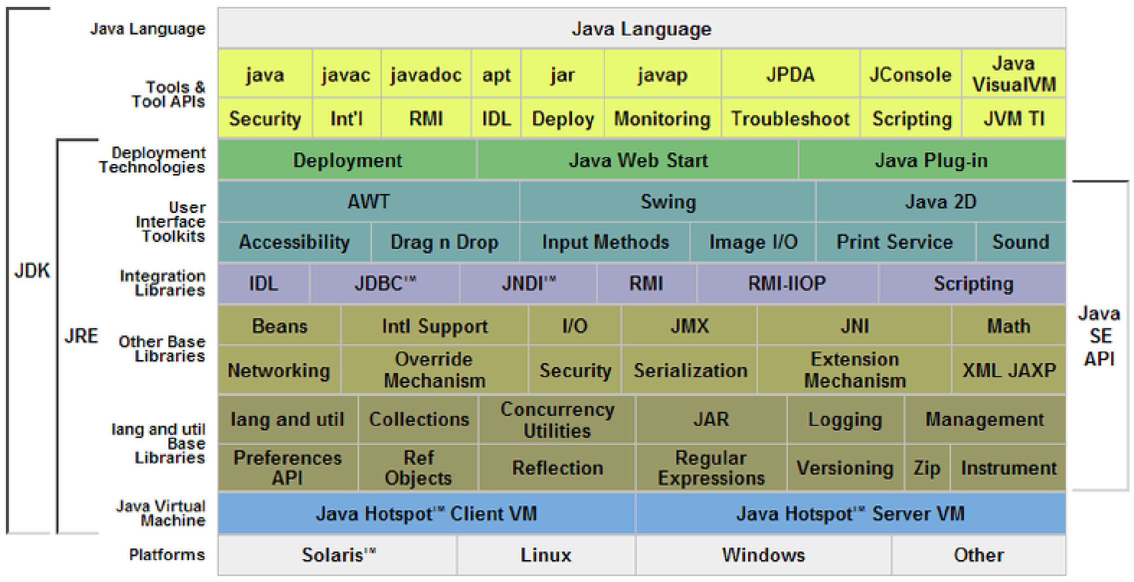
1. **The VM core:** with its subcomponents concentrates most of the JVM control functions.
2. **The porting layer**: hides platform-specific details from other VM components behind a single interface and is based on the Apache Portable Runtime layer.
3. **The Garbage Collector**: allocates Java objects in the heap memory and reclaims unreachable objects using various algorithms
4. **Execution Manager**: selects the execution engine for compiling a method, handles profiles and the dynamic recompilation logic.
5. **Class Library**: is a Java standard library.
6. **The thread manager** that handle Operating system threading
7. **The execution engine:** consists of the following:
   1. The just-in-time compiler for compilation and execution of method code.
   2. The interpreter for easier debugging.

Support platform and operating system

The project provide a portable implementation that ease development on many platforms and operating systems. The main focus is on Windows and Linux operating systems on x86 and x86-64 architectures.

	Windows 2000	Windows XP, Server 2003, Vista	Linux RHEL, SLES, Debian, Gentoo, Fedora	FreeBSD	AIX	Mac OS X
IA32 (Pentium III or better)	In progress	Yes	Yes	In progress	N/A	N/A
x86-64 (Intel 64, AMD64)	N/A	Yes	Yes	N/A	N/A	N/A
Itanium (IA64, IPF)	N/A	In progress	Yes	N/A	N/A	N/A
PPC32	N/A	N/A	In progress	N/A	N/A	N/A
PPC64	N/A	N/A	In progress	N/A	In progress	N/A
zSeries (31 bit)	N/A	N/A	In progress	N/A	N/A	N/A

Class Library coverage



Java Platform diagram showing Class Library

The expected donations from software companies were actually received. The Apache Harmony now contains the working code, including the Swing, AWT and Java 2D code which were contributed by Intel.

The part of the implemented classes is still smaller than in GNU Classpath (97.7% in the trunk versus almost 100% as of July 2007), despite some non-trivial applications were shown being running in 2006 JavaOne international conference.

Also Harmony's test suite is less extensive than GNU Classpath's for now (20000 tests versus 50000 as of October 2006).

The progress of the Apache Harmony project can be tracked against J2SE 1.4 and Java SE 5.0. Also, there is a branch for Harmony v6.0 in development for Java SE 6.0.

Apache Harmony developers integrate several existing, field-tested open-source projects to meet their goal (not reinventing the wheel). Many of these projects are mature and well known and other part of the library need be writing from scratch.

This is a list of existing open source components that are used in the Apache Harmony project; some of them were in use before the project started.

Component	Description
ICU	Mature C/C++ and Java libraries for Unicode support, software internationalization and software globalization.
Apache Xalan	An XSLT stylesheet processor for Java and C++ which implements the XPath language.
Apache Xerces	An XML parser library for Java, C++ and Perl
Apache Portable Runtime	Cross-platform abstraction library provides platform independence.
Apache CXF	Robust and high performance Web Services framework work over protocols such as SOAP, XML/HTTP, RESTful HTTP, and CORBA.
BCEL	Libraries for decomposing, modifying, and recomposing binary Java classes (i.e., bytecode)
MX4J	Java Management Extensions (JMX) tools for managing and monitoring applications, system objects, devices and service oriented networks.
vmmagic	Set of extensions to the Java language to facilitate systems programming in Java by adding direct memory operations etc.
Bouncy Castle	Libraries collection of lightweight cryptography for Java and C#.
ANTLR	language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages.

Documentation

Harmony is currently less documented than the alternative free Java implementations. For instance, in GNU Classpath every method of the central CORBA class (ORB) has the explaining comment both in the standard abstract API class and implementation . In the Yoko project, used by Harmony , the majority of methods both in the standard declaration and implementing class were not documented (at the end of October, 2006). Also, GNU Classpath supported both older and current CORBA features (same as Sun's implementation). Harmony, differently, left the central method of the older standard (ORB.connect(Object)) fully unimplemented.

Tools

A complete implementation of the Java platform also requires a compiler that translates Java source code into bytecodes, a program that manages JAR files, a debugger, and an applet viewer and web browser plugin, to name a few. Harmony currently has the compiler, appletviewer, jarsigner, javah, javap, keytool, policytool, and unpack200 .

Virtual machine support

Harmony currently has seven virtual machine implementations that run Harmony Class Library, all of which were donations by external groups:

- JC Harmony Edition VM, "JCHEVM," based on the JCVM's interpreter, contributed by the author, Archie Cobbs.
- BootJVM, a simple bootstrapping virtual machine, contributed by Daniel Lydick.
- SableVM, an advanced, portable interpreter, contributed by authors from the Sable Research Group; and the Dynamic Runtime Layer Virtual Machine.
- DRLVM, a just-in-time compiler contributed by Intel.
- BEA announced the availability of an evaluation version of JRockit VM running Apache Harmony Class Library.
- JikesRVM, an open-source meta-circular JVM that use the Apache Harmony Class Library .
- Ja.NET SE - the open source project is providing a Java 5 JDK (class libraries, tools, etc.) that run on the .NET CLR. Ja.NET SE is based on the Apache Harmony Class Libraries.

In the end of November, 2006, the language support provided by these virtual machine was still incomplete, and the build instructions recommended to use IBM's proprietary J9 instead to run the class library test suite. However, this is not necessary anymore (as of July 2007). The DRLVM virtual machine is currently (as of July 2006) under heavy development, so a fast improvement of its features can be expected.

Applications status

Since its conception, Harmony has steadily grown in its ability to execute non-trivial Java applications. As of July 2007, supported applications include:

- Eclipse : 99.3% of the 36000 RI test pass on Harmony's DRLVM + class library. .
- Apache Tomcat : 100% of the RI tests pass .
- JUnit : 100% of the RI tests pass .
- Apache Ant : 97% of the RI tests pass. .
- other applications pass with a high success rate, such as Apache Derby, Apache Axis, Log4j, Apache Velocity, Apache Cocoon, jEdit, and Apache Commons.

However, Harmony's incomplete library prevents it from launching some other applications:

- ArgoUML: because it requires a Java applet implementation, which is still not available in Harmony.
- Apache Geronimo runs on Apache Harmony with some issues and workarounds .
- Vuze, formerly Azureus, because of unimplemented security classes.

Dalvik (software)

Dalvik	
Original author(s)	Dan Bornstein
Operating system	Linux
Platform	Android
Type	Virtual machine
License	Apache License 2.0

Dalvik is the name of the virtual machine (VM) in Google's Android operating system. Dalvik is thus an integral part of Android, which is typically used on mobile devices such as mobile phones, tablet computers and netbooks. Before execution, Android applications are converted into the compact **Dalvik Executable** (**.dex**) format, which is designed to be suitable for systems that are constrained in terms of memory and processor speed.

Dalvik, like the rest of Android, is open-source software. It was originally written by Dan Bornstein, who named it after the fishing village of Dalvík in Eyjafjörður, Iceland, where some of his ancestors lived.

Architecture

Unlike Java VMs, which are stack machines, the Dalvik VM is a register-based architecture.

The relative merits of stack machines versus register-based approaches is a subject of ongoing debate. Generally, stack-based machines must use instructions to load data on the stack and manipulate that data, and, thus, require more instructions than register machines to implement the same high level code, but the instructions in a register machine must encode the source and destination registers and, therefore, tend to be larger. This difference is primarily of importance to VM interpreters for whom opcode dispatch tends to be expensive along with other factors similarly relevant to Just-in-time compilation.

A tool called **dx** is used to convert some (but not all) Java .class files into the .dex format. Multiple classes are included in a single .dex file. Duplicate strings and other constants used in multiple class files are included only once in the .dex output to conserve space. Java bytecode is also converted into an alternate instruction set used by the Dalvik VM. An uncompressed .dex file is typically a few percent smaller in size than a compressed .jar (Java Archive) derived from the same .class files.

The Dalvik executables may be modified again when installed onto a mobile device. In order to gain further optimizations, byte order may be swapped in certain data, simple data structures and function libraries may be linked inline, and empty class objects may be short-circuited, for example.

As of Android 2.2, Dalvik has a just-in-time compiler.

Being optimized for low memory requirements, Dalvik has some specific characteristics that differentiate it from other standard VMs:

- The VM was slimmed down to use less space
- The constant pool has been modified to use only 32-bit indexes to simplify the interpreter
- It uses its own bytecode, not Java bytecode

Moreover, Dalvik has been designed so that a device can run multiple instances of the VM efficiently.

Class library

Dalvik does not align to Java SE nor Java ME Class Library profiles (e.g., Java ME classes, AWT or Swing are not supported). Instead it uses its own library built on a subset of the Apache Harmony Java implementation.

Licensing

Dalvik is said to be a clean-room implementation rather than a development on top of a standard Java runtime, which would mean it does not inherit copyright-based license restrictions from either the standard-edition or open-source-edition Java runtimes. Dalvik is published under the Apache 2 license.

Controversy

On the 12th of August 2010, Oracle, owner of Java since it acquired Sun Microsystems in April 2009, sued Google over claimed infringement of copyrights and patents. In developing Android, it is alleged that Google knowingly, directly and repeatedly infringed Oracle's Java-related intellectual property. Oracle has named Boies, Schiller & Flexner as part of its legal team.

Specifically the patent infringement claim references seven patents including US Patent No. 5966702 "Method And Apparatus For Preprocessing And Packaging Class Files", and US Patent No. 6910205 "Interpreting Functions Utilizing A Hybrid Of Virtual And Native Machine Instructions". It also references US Patent No. RE38104 "Method And Apparatus For Resolving Data References In Generated Code".

A 2008 Federal Court of Appeals case addressing gratis (“$0”) copyright license breach may have an impact on the controversy.

Chapter 10

HotSpot

HotSpot	
Developer(s)	Oracle Corporation (previously Sun Microsystems)
Stable release	16.3-b01
Written in	C++
Operating system	Cross-platform
Type	Java Virtual Machine
License	GNU General Public License

HotSpot is the primary Java Virtual Machine for desktops and servers, produced by Oracle Corporation. It features techniques such as just-in-time compilation and adaptive optimization designed to improve performance.

History

HotSpot, first released April 27, 1999, was originally developed by Longview Technologies, LLC which was doing business as Animorphic, a small startup company formed in 1994. Animorphic's virtual machine technology had earlier been successfully used in a Sun research project, the Self programming language. In 1997, Longview Technologies, LLC (DBA Animorphic) was purchased by Sun Microsystems. Initially available as an add-on for Java 1.2, HotSpot became the default Sun JVM in Java 1.3.

Its name derives from the fact that as it runs Java bytecode, it continually analyzes the program's performance for "hot spots" which are frequently or repeatedly executed. These are then targeted for optimization, leading to high performance execution with a minimum of overhead for less performance-critical code. In some cases, it is possible for adaptive optimization of a JVM to exceed the performance of hand-coded C++ or C code.

Features

Sun's JRE features 2 virtual machines, one called *Client* and the other *Server*. The Client version is tuned for quick loading. It makes use of interpretation, compiling only often-run methods. The Server version loads more slowly, putting more effort into producing highly optimized JIT compilations, that yield higher performance.

The HotSpot Java Virtual Machine is written in C++. As stated in HotSpot web page, the source contains approximately 250,000 lines of code. Hotspot provides:

- A class loader,
- A bytecode interpreter,
- *Client* and a *Server* virtual machines, optimised for their respective uses
- Several garbage collectors,
- A set of supporting runtime libraries.

JVM flags

HotSpot supports many Command-line arguments for tweaking the Virtual Machine at launch. Some are standard and must be found in any conforming JVM, others are specific to HotSpot and may not be found in other JVMs (options that begin with -X or -XX are non-standard).

License

On 13 November 2006, the Sun JVM and JDK were licensed under the GPL version 2. This is the code that will become Java 7.

Supported platforms

Maintained by Oracle

As for the whole JDK, HotSpot is currently supported by Oracle on Microsoft Windows, Linux, and Solaris. Supported ISAs are IA-32, x86-64 and SPARC (exclusive to Solaris).

Ports by third parties

Ports are also available by third parties for Mac OS X and various other Unix Operating systems. A number of different hardware architectures are also supported, including x86, PowerPC, and SPARC (Solaris only).

Ports of HotSpot are difficult because the code, while primarily written in C++, contains a lot of assembly. To remedy this, the IcedTea project has developed a generic port of the HotSpot interpreter called *zero-assembler Hotspot* (or *zero*), with almost no assembly code. This port is intended to allow the interpreter part of HotSpot to be very easily adapted to any Linux processor architecture, potentially making it infinitely portable. The code of *zero-assembler Hotspot* is used for all the non-x86 ports of HotSpot (PPC, IA64, S390 and ARM) since version 1.6.

Gary Benson, an IcedTea developer, is now developing a platform-independent Just-in-time compiler called *Shark* for HotSpot, using LLVM, to complement *zero*.

Chapter 11

Microsoft Java Virtual Machine and IcedTea

Microsoft Java Virtual Machine

The **Microsoft Java Virtual Machine** was a proprietary Java Virtual Machine computer program from Microsoft. It was first made available for Internet Explorer version 3 so that users could run Java applets when browsing on the World Wide Web. It was the fastest Windows-based implementation of a Java virtual machine for the first two years after its release. Sun Microsystems, the creator of Java, sued Microsoft in October 1997 for incompletely implementing the Java 1.1 standard. It was also named in the United States v. Microsoft antitrust civil actions, as an implementation of Microsoft's Embrace, extend and extinguish strategy. In 2001, Microsoft settled the lawsuit with Sun and discontinued its Java implementation.

History

Performance

The Microsoft JM won the PC Magazine Editor's choice awards in 1997 and 1998 for best Java support. In 1998 a new release included the Java Native Interface which supplemented Microsoft's proprietary Raw Native Interface (RNI) and J/Direct. Microsoft claimed to have the fastest Java implementation for Windows, although IBM also made that claim in 1999 and beat the Microsoft and Sun virtual machines in the JavaWorld Volano test.

Antitrust trial

Microsoft's proprietary extensions to Java were used as evidence in the United States v. Microsoft antitrust civil actions.

A Memorandum of the United States in Support of Motion for Preliminary Injunction in the case of United States of America vs. Microsoft claimed that Microsoft wanted to kill Java in the marketplace.

In short, Microsoft feared and sought to impede the development of network effects that cross-platform technology like Netscape Navigator and Java might enjoy and use to challenge Microsoft's monopoly. Another internal Microsoft document indicates that the plan was not simply to blunt Java/browser cross-platform momentum, but to destroy the cross-platform threat entirely, with the "Strategic Objective" described as to "Kill cross-platform Java by grow[ing] the polluted Java market."

Sun vs. Microsoft

In October 1997, Sun Microsystems, the creator of Java, sued Microsoft for incompletely implementing the Java 1.1 standard.

In January 2001, Sun and Microsoft settled the suit. Microsoft paid Sun $20 million and the two agreed to a plan for Microsoft to phase out products that included the older version of Microsoft Java that allegedly infringed on Sun's Java copyrights and trademarks.

- Office XP Developer
- Office 2000 Developer
- Office 2000 Premium Service Release 1
- Microsoft BackOffice Server 2000
- Internet Security and Acceleration Server(ISA) 2000
- Internet Explorer 5.5
- Visual Studio 6 Microsoft Developer Edition
- Windows 98 and Windows Me

The Microsoft Java Virtual Machine was discontinued in 2001 in response to the Sun Microsystems lawsuit. Microsoft continued to offer support until June 30, 2009.

Windows XP

The initial edition Windows XP RTM did not ship with a Java virtual machine 2001, due to the settlement. This required users that wanted to run Java Applets in Internet Explorer to download and install either the standard Sun Java virtual machine, or locate a copy of the Microsoft Java virtual machine elsewhere.

Service Pack 1 (SP1) for Windows XP was released on September 9, 2002. It contained post-RTM security fixes and hot-fixes, compatibility updates, optional .NET Framework support, and enabled technologies for new devices such as Tablet PCs. It also included the Microsoft Java virtual machine.

On February 3, 2003, Microsoft released Service Pack 1 (SP1) again as Service Pack 1a (SP1a). This release removed Microsoft's Java virtual machine as a result of the lawsuit with Sun Microsystems.

IcedTea

IcedTea6

Developer(s)	Red Hat / GNU Classpath
Stable release	1.9 / September 10, 2010; 3 months ago
Written in	C and Java
Operating system	Cross-platform
Type	Java Virtual Machine and Java Library
License	GPL+linking exception

IcedTea7

Developer(s)	Red Hat / GNU Classpath
Stable release	1.13 (OpenJDK7 Milestone 7) / July 29, 2010; 4 months ago
Written in	C and Java
Operating system	Cross-platform
Type	Java Virtual Machine and Java Library
License	GPL+linking exception

IcedTea is a software development and integration project launched by Red Hat in June 2007. The initial goal was to make the Java OpenJDK software which Sun Microsystems released as free software in 2007 usable without requiring any other software that is not free software and hence make it possible to add OpenJDK to Fedora and other Linux distributions that insist on free software. This was met and a version of IcedTea based on OpenJDK was packaged with Fedora 8 in November 2007.

April 2008 saw the first release of a new variant, **IcedTea6** which is based on Sun's build drops of OpenJDK6, a fork of the OpenJDK with the goal of being compatible with the existing JDK6. This was released in Ubuntu and Fedora in May 2008. The IcedTea package in these distributions has been renamed to OpenJDK using the OpenJDK trademark notice. In June 2008, the Fedora build passed Sun's rigorous TCK testing on x86 and x86-64.

The IcedTea project also provides an easier to use build system and a home for various new features that are not yet ready to be integrated into the main OpenJDK tree, either because of immaturity or because of the need for OpenJDK contributors to sign the Sun Contributor Agreement.

History

This project was created following Sun's release of its HotSpot Virtual Machine and Java compiler in November 2006, and most of the source code of the class library in May 2007. However, parts of the class library, such as font rendering, colour management and sound support, were only provided as proprietary binary plugins. This was because the source code for these plugins was copyrighted to 3rd parties, rather than Sun Microsystems. The released parts were published under the terms of the GNU General Public License, a free software licence.

Because of these missing components, it was not possible to build OpenJDK only with Free software components. Sun aimed to negotiate with the license holders to allow this code to be released under a Free license, or failing that, to replace these proprietary elements with alternative implementations. With the plugins replaced, the class library would then be completely Free. Sun has continued to use the proprietary code in their certified binary releases.

Following the announcement, the IcedTea project was started and was formally announced on June 7, 2007, with a build repository provided by the GNU Classpath team. The team could not call their software product *"OpenJDK"* because this is a trademark owned by Sun Microsystems. They instead decided to use the temporary name *"IcedTea"*.

On November 5, 2007, Red Hat signed both the Sun Contributor Agreement and the OpenJDK Community TCK License. The press release suggested that this would benefit the IcedTea project. Simon Phipps suggested the possibility of IcedTea being hosted on openjdk.java.net, and Mark Reinhold noted that signing the copyright assignment could allow Red Hat to contribute parts of IcedTea to Sun for inclusion in the mainstream JDK.

Since then, a number of patches from IcedTea6 have made their way into OpenJDK6.

On June 2008, it was announced that IcedTea6 (as the packaged version of OpenJDK on Fedora 9) has passed the Technology Compatibility Kit tests and can claim to be a fully compatible Java 6 implementation. The project continues to track both OpenJDK6 and OpenJDK7 development in separate repositories, and contribute patches back upstream where possible.

The aims

Specifically, the IcedTea project started with two aims. One was to make it possible for the GNU Compiler for Java to compile the OpenJDK code. The problem was that the only program which could compile the OpenJDK software was the existing proprietary Sun JDK. Free distributions like Fedora can't depend on proprietary tools in order to build packages, so the IcedTea project had to make it possible to compile the code using Free software. When this was done, the resulting IcedTea version of OpenJDK could be used to compile itself, thus escaping the need to use non-Free software for future compiling.

The second task was to provide Free equivalents of the binary plugins that existed in OpenJDK because Sun was unable to release all the source code. As of March 2008, this is no longer necessary for IcedTea6, as the OpenJDK6 build drops can be built with no binary plugins. With the release of b10, which replaces the proprietary sound support with that from the Gervill project, a full implementation of Java 1.6 can be built without binary plugins. The only remaining binary plug is for SNMP support, which is an optional provider for the JMX architecture and not part of the specification. As of b53 in April 2009, the same is true for OpenJDK7.

Other benefits

IcedTea provides the only working free software Java Web browser plugin. It was the first to work in 64-bit browsers under 64-bit Linux, a feature Sun's proprietary JRE later addressed. This makes it suitable to enable support for Java applets in 64-bit Mozilla Firefox, among others. IcedTea also provides the only Free Java Web Start implementation by means of continued development of Netx. Sun has made continued promises about releasing their plugin and Web Start implementation as part of OpenJDK, but have so far failed to deliver, despite continued pressure from the community. Development on the IcedTea plugin continues rapidly, with the latest version of the next-generation plugin supporting Google's Chromium.

IcedTea also provides a more familiar build system, by providing a wrapper around the OpenJDK makefiles using the GNU autotools. This removes the need to remember a large number of environment variables for configuring the build. (The current IcedTea builds set roughly forty such variables for the underlying OpenJDK build.) It has also provided a place for early work on features which will eventually appear in the main OpenJDK builds such as Gervill and for work on ports to other platforms.

Progress and availability

From June 2007, IcedTea was able to build itself and pass a significant portion of Mauve, the GNU Classpath test suite. In May 2008, support was added to IcedTea for running the Sun JTreg regression tests.

Currently:

- IcedTea is the default JVM in Ark Linux, and the suggested JVM in Arch Linux.

- IcedTea is available in CRUX PPC from release 2.4, and actually there are binary ppc and ppc64 packages too.

- IcedTea is available in Ubuntu 7.10 (Gutsy Gibbon), from the "universe" repository, and IcedTea6 in 8.04 (Hardy Heron).

- IcedTea was available in Fedora 8 and IcedTea6 appears in Fedora 9, 10, 11, 12 and 13 as java-1.6.0-openjdk.

- A binary package for IcedTea6 is available in Gentoo's official repository, but was marked unstable until recently. Continued development on source ebuilds for Gentoo for both IcedTea6 and IcedTea7 occurs in the java overlay. Installing a Java application by default pulls in IcedTea6 instead of sun-jdk.

- It can be built and run under Debian. Binary packages were submitted on 20 April 2008 but were rejected due to licensing problems with some files. A new package was uploaded in June 2008 and entered unstable on 12 July 2008.

Architecture

OpenJDK contained approximately (on release in May 2007) 4 % encumbered code, which was only packaged as binary plugins. These were required to build and use the JDK. OpenJDK6 was released with only 1 % encumbered code, and this sound support has also since been replaced. IcedTea6 is based on this release. IcedTea6 still provides its own web browser plugin and Web Start support, as Sun's implementation remains proprietary.

IcedTea can compile OpenJDK using GNU Classpath-based solutions such as GCJ and optionally bootstraps itself using the HotSpot Java Virtual Machine and the javac Java compiler it just built.

Zero and Shark

Ports of HotSpot (OpenJDK's Virtual Machine) are difficult because the code contains a lot of assembly in addition to the C++ core. The IcedTea project has developed a generic port of the HotSpot interpreter called *zero-assembler Hotspot* (or *zero*), with almost no assembly code. This port is intended to allow the interpreter part of HotSpot to be very easily adapted to any Linux processor architecture. The code of *zero-assembler Hotspot* is used for all the non-x86 ports of HotSpot (PPC, IA64, S390 and ARM) since version 1.6 of IcedTea7.

The IcedTea project is now developing a platform-independent just-in-time compiler called *Shark* for HotSpot, using Low Level Virtual Machine, to complement *Zero*. Completing this work will make the Java Virtual Machine independent of the CPU architecture.

WORLD TECHNOLOGIES

Chapter 12

Jikes RVM and Jazelle

Jikes RVM

	Jikes RVM
Developer(s)	Jikes RVM Project Organization
Stable release	3.1.1 / July 5, 2010; 4 months ago
Development status	Active
Written in	Java
Operating system	Unix-like
Type	Java Virtual Machine and Java Library
License	Eclipse Public License

Jikes RVM (Jikes Research Virtual Machine) is a mature open source virtual machine that runs Java programs. Unlike most other JVMs it is written in Java, a style of implementation known as meta-circular.

History

- Nov, 1997, the Jalapeño project starts as an internal research project at IBM's Thomas J. Watson Research Center.
- 1999, 2000, research papers describing novel aspects of Jikes RVM are published by IBM researchers and several universities are given access to the source code.
- Oct, 2001, Jikes RVM version 2 is released as an open source project under the Common Public License. The release supports PowerPC and Intel architectures and a range of different garbage collection algorithms.
- 2002, Jikes RVM 2.2 is released with the precise garbage collectors now refactored into the popular *Memory Management Toolkit* precise garbage collectors.

- 2004, Jikes RVM 2.4 is released with increased stability and performance partly, particular focus was made of running eclipse.
- 2007, Jikes RVM 2.9 development starts with the code base extensively refactored to use features of Java 5.0 and to use an ant build system.
- 2008, Jikes RVM 3.0 released marking the end of 2.9 development and new stability in the code base.
- 2009, Jikes RVM 3.1 released under the Eclipse Public License (EPL) with significantly improves over the performance of 3.0.1 and switching to native threading.

Bootstrap

Being meta-circular Jikes RVM requires a bootstrap JVM to run upon to create a boot image. The boot image is a view of the objects Jikes RVM requires to boot created using reflection in the bootstrap JVM. A small C loader is responsible for loading the boot image at runtime.

VM Magic

VM Magic is where the compiler generates different code for a class than the bytecodes within that class should perform. VM Magic classes allow direct access to memory and are key to the Memory Management Toolkits performance. The VM Magic classes reside in the *org.vmmagic* package and have been reused in other Java projects.

Memory Management Toolkit

The Memory Management Toolkit (MMTk) is a collection of precise garbage collectors that have been used within Jikes RVM and other projects such as the singularity operating system and the Rotor software project. As with the rest of the Jikes RVM the implementation is in Java, but the main dependence is on VM Magic.

Class Libraries

Either Apache Harmony or GNU Classpath class libraries can be used with Jikes RVM, with experimental support for OpenJDK's class library.

Compilers

Jikes RVM uses a fast baseline compiler to quickly generate code for a particular architecture. Adaptive compilation then recompiles code with an optimizing compiler with features such as on stack replacement. The adaptive compilation system uses a cost-benefit analysis model.

Runtime

Jikes RVM's runtime has many innovative features including mechanisms for fast locking, a collaborative scheduling mechanism and support for fast exception gathering and dispatch.

Architectures

The PowerPC (or ppc) and ia32 (or Intel x86 - 32bits) instruction set architectures are supported by Jikes RVM.

Research

Since it is a research project, the emphasis of Jikes RVM is on researching new technologies, as is apparent from the scientific publications it has spawned (over 190 papers as of 2008).

Jikes RVM has also appeared in the Google Summer of Code 2007 and 2008.

Jazelle

Jazelle DBX (Direct Bytecode eXecution) allows some ARM processors to execute Java bytecode in hardware as a third execution state alongside the existing ARM and Thumb modes. Jazelle functionality was specified in the ARMv5TEJ architecture and the first processor with Jazelle technology was the **ARM926EJ-S**. Jazelle is denoted by a 'J' appended to the CPU name, except for post-v5 cores where it is required (albeit only in trivial form) for architecture conformance.

Jazelle RCT (Runtime Compilation Target) is a different technology and is based on ThumbEE mode and supports ahead-of-time (AOT) and just-in-time (JIT) compilation with Java and other execution environments.

The published specifications are very incomplete, being only sufficient for writing operating system code that can support a JVM that uses Jazelle. The declared intent is that only the JVM software needs to (or is allowed to) depend on the hardware interface details. This tight binding facilitates that the hardware and JVM can evolve together without affecting other software. In effect, this gives ARM Holdings considerable control over which JVMs are able to exploit Jazelle. It also prevents open source JVMs from using Jazelle. These issues do not apply to the ARMv7 ThumbEE environment, the nominal successor to Jazelle DBX.

Implementation

The Jazelle extension uses low-level binary translation, implemented as an extra stage between the fetch and decode stages in the processor instruction pipeline. Recognised bytecodes are converted into a string of one or more native ARM instructions.

The Jazelle mode moves JVM interpretation into hardware for the most common simple JVM instructions. This is intended to significantly reduce the cost of interpretation. Among other things, this reduces the need for JIT and other JVM accelerating techniques. JVM instructions that are not implemented in Jazelle hardware cause appropriate routines in the Jazelle-aware JVM implementation to be invoked. Details are not published, since all JVM innards are transparent (except for performance) if correctly interpreted.

Jazelle mode is entered via the BXJ instructions. A hardware implementation of Jazelle will only cover a subset of JVM bytecodes. For unhandled bytecodes—or if overridden by the operating system—the hardware will invoke the software JVM. The system is designed so that the software JVM does not need to know which bytecodes are implemented in hardware and a software fallback is provided by the software JVM for the full set of bytecodes.

Instruction set

The instruction set used in Jazelle mode is documented—it is Java bytecode after all. However, ARM have chosen to remain quiet on the exact execution environment details; the documentation provided with Sun's HotSpot Java Virtual Machine goes as far as to state: *For the avoidance of doubt, distribution of products containing software code to exercise the BXJ instruction and enable the use of the ARM Jazelle architecture extension without [..] agreement from ARM is expressly forbidden.*.

Employees of ARM have in the past published several white papers that do give some good pointers about the processor extension. Versions of the ARM Architecture Reference Manual available from 2008 have included pseudocode for the 'BXJ' (Branch and eXchange to Java) instruction, but with the finer details being shown as "SUB-ARCHITECTURE DEFINED" and documented elsewhere.

Application binary interface (ABI)

The Jazelle state relies on an agreed calling convention between the JVM and the Jazelle hardware state. This application binary interface is not published by ARM, rendering Jazelle an undocumented feature for most users and Free Software JVMs.

The entire VM state is held within normal ARM registers, allowing compatibility with existing operating systems and interrupt handlers unmodified. Restarting a bytecode (such as following a return from interrupt) will re-execute the complete sequence of related ARM instructions.

Specific registers are designated to hold the most important parts the JVM state, registers r0-r3 hold an alias of the top of the Java stack, r4 holds Java local operand zero (pointer to `*this`) and r6 contains the Java stack pointer.

Jazelle reuses the existing Program Counter register r15. A pointer to the *next* bytecode goes in r14, so the use of the PC is not generally user-visible except during debugging.

CPSR: Mode indication

Java bytecode is indicated as the current instruction set by a combination of two-bits in the ARM CPSR (Current Program Status Register). The 'T'-bit must be cleared and the 'J'-bit set.

Bytecodes are decoded by the hardware in two stages (versus a single stage for Thumb and ARM code) and switching between hardware and software decoding (Jazelle mode and ARM mode) takes ~4 clock cycles..

For entry to Jazelle hardware state to succeed, the JE (Jazelle Enable) bit in the CP14:c0(c2)[bit 0] register must be set; clearing of the JE bit by a [privileged] operating-system provides a high-level override to prevent application programs from using the hardware Jazelle acceleration, additionally the CV (Configuration Valid) bit found in CP14:c0(c1)[bit 1] must be set to show that there is a consistent Jazelle state setup for the hardware to use.

BXJ: Branch to Java

The BXJ instruction attempts to switch to Jazelle state, and if allowed and successful, sets the 'J' bit in the CPSR; otherwise "falling through" and acting as a standard BX (Branch) instruction. The only time when an operating system, or debugger must be fully aware of the Jazelle mode is when decoding a faulted or trapped instruction. The Java program counter (PC) pointing to the next instructions must be placed in the Link Register (r14) before executing the BXJ branch request, as regardless of hardware or software processing, the system must know where to begin decoding.

Because the current state is held in the CPSR, the bytecode instruction set is automatically reselected after task-switching and processing of the current Java bytecode is restarted.

Following an entry into the Jazelle state mode, bytecodes can be processed in one of three ways; decoded and executed natively in hardware, handled in software (with optimised ARM/ThumbEE JVM code), or treated as an invalid/illegal opcode. The third case will cause a branch to an ARM exception mode, as will a Java bytecode of 0xff, which is used for setting JVM breakpoints.

Execution will continue in hardware until an unhandled bytecode is encountered, or an exception occurs. Between 134 and 149 bytecodes (out of 203 bytecodes specified in the JVM specification) are translated and executed directly in the hardware.

Low-level registers

Low-level configuration registers, for the hardware virtual machine, are held in the ARM Co-processor "CP14 register c0". The registers allow detecting, enabling or disabling the hardware accelerator—if it is available.

- The Jazelle Identity Register in register CP14:c0(c0) is read-only accessible in all modes.
- The Jazelle OS Control Register at CP14:c0(c1) is only accessible in kernel mode and will cause an exception when accessed in user-mode.
- The Jazelle Main Configuration Register at CP14:c0(c2) is write-only in user-mode and read-write in kernel mode.

A "trivial" hardware implementation of Jazelle (as found in the QEMU emulator) is only required to support the BXJ opcode itself (treating BXJ as a normal BX instruction) and to return RAZ (Read-As-Zero) for all of the CP14:c0 Jazelle-related registers.

Successor: ThumbEE

The ARMv7 architecture has de-emphasized Jazelle and *Direct Bytecode Execution* of JVM bytecodes. In implementation terms, only trivial hardware support for Jazelle is now required: support for entering and exiting Jazelle mode, but not for executing any Java bytecodes.

Instead, the *Thumb Execution Environment* (ThumbEE) is now preferred. Support for this is mandatory in ARMv7-A processors (such as the Cortex-A8 and Cortex-A9), and optional in ARMv7-R processors. ThumbEE targets compiled environments, perhaps using JIT technologies. It is not at all specific to Java, and is fully documented; much broader adoption is anticipated than Jazelle was able to achieve.

ThumbEE is a variant of the Thumb2 16/32-bit instruction set. It integrates null pointer checking; defines some new fault mechanisms; and repurposes the 16-bit LDM and STM opcode space to support a few instructions such as range checking, a new handler invocation scheme, and more. Accordingly, compilers that produce Thumb or Thumb2 code can be modified to work with ThumbEE-based runtime environments.

Chapter 13

OpenJDK and Clojure

OpenJDK

	OpenJDK6
Developer(s)	Sun Microsystems
Initial release	OpenJDK6 Build b05 February 12, 2008; 2 years ago
Stable release	OpenJDK6 Build b20 / June 21, 2010; 5 months ago
Written in	C++ and Java
Operating system	Cross-platform
Type	Library
License	GPL+linking exception

	OpenJDK7
Developer(s)	Sun Microsystems
Preview release	OpenJDK7 Build b119 / November 23, 2010; 19 days ago
Written in	C++ and Java
Operating system	Cross-platform
Type	Library
License	GPL+linking exception

OpenJDK (aka **Open Java Development Kit**) is a free and open source implementation of the Java programming language. It is the result of an effort Sun Microsystems began in 2006. The implementation is licensed under the GNU General Public License (GPL) with a linking exception, which exempts components of the Java class library from the GPL licensing terms.

History

Sun's promise and initial release

Sun announced in JavaOne 2006 that Java would become open-source software, and on October 25, 2006, at the Oracle OpenWorld conference, Jonathan Schwartz said that the company intended to announce the open-sourcing of the core Java Platform within 30 to 60 days.

Sun released the Java HotSpot virtual machine and compiler as free software under the GNU General Public License on November 13, 2006, with a promise that the rest of the JDK (which includes the Java Runtime Environment) would be placed under the GPL by March 2007, "except for a few components that Sun does not have the right to publish in source form under the GPL". According to computer scientist and free-software advocate Richard Stallman, this would end the "Java trap", the vendor lock-in that he argues applied to Java and programs written in Java. Software entrepreneur Mark Shuttleworth called the initial press announcement "A real milestone for the free software community".

Release of the class library

Following their promise to release a Java Development Kit (JDK) based almost completely on free and open source code in the first half of 2007 , Sun released the complete source code of the Java Class Library under the GPL on May 8, 2007, except for some limited parts that some third parties licensed to Sun that rejected the terms of the GPL. Included in the list of encumbered parts were several major components of the Java graphical user interface (GUI). Sun stated that it planned to replace the remaining proprietary components with alternative implementations and to make the class library completely free.

Community improvements

On November 5, 2007, Red Hat announced an agreement with Sun, signing Sun's broad contributor agreement (which covers participation in all Sun-led free and open source software projects by all Red Hat engineers) and Sun's OpenJDK Community TCK License Agreement (which gives the company access to the test suite that determines whether a project based on openJDK complies with the Java SE 6 specification).

Also on November 2007, the *Porters Group* was created on OpenJDK to aid in efforts to port OpenJDK to different processor architectures and operating systems. The BSD porting projects , led by Kurt Miller and Greg Lewis and the Mac OS X porting project (based on the BSD one) SoyLatte led by Landon Fuller have expressed interest in joining OpenJDK via the Porters Group and as of January 2008 are part of the mailing list discussions. Another project pending formalization on the Porters Group is the Haiku Java Team, led by Bryan Varner.

On December 2007, Sun moved the revision control of OpenJDK from TeamWare to Mercurial, as part of the process of releasing it to open source communities.

OpenJDK has comparatively strict procedures of accepting code contributions: every proposed contribution must be reviewed by two of Sun's engineers and the contributor must have signed the Sun/Oracle Contributor Agreement.(SCA/OCA) Preferably, there should also be a JTreg test demonstrating that the bug has been fixed. Initially, the external patch submission process was slow and commits to the codebase were only made by Sun engineers, until September 2008. The process has improved and, as of 2010, simple patches and backports from OpenJDK7 to OpenJDK6 can take place within hours rather than days.

On 2010-10-11, IBM and Oracle announced that both companies will collaborate to further develop OpenJDK.

On 2010-11-12, Apple and Oracle announced the OpenJDK project for Mac OS X.

Status

Supported JDK versions

OpenJDK was initially based only on the JDK 7.0 version of the Java platform.

Since February 15, 2008, there are two separate OpenJDK projects:

- The main OpenJDK project, which is based on the JDK 7.0 version of the Java platform,
- The JDK 6 project, which provides an Open-source version of Java 6.0.

Compiler and Virtual Machine

Sun's Java compiler, javac, and HotSpot (the virtual machine), are now under a GPL license.

Class library

As of the first May 2007 release, 4% of the OpenJDK class library remained proprietary. By the appearance of OpenJDK 6 in May 2008, less than 1% (the SNMP implementation, which is not part of the Java specification) remained, making it possible to build OpenJDK without any binary plugs. The binary plug requirement was later dropped from OpenJDK7 as part of b53 in April 2009.

This was made possible, over the course of the first year, by the work of Sun Microsystems and the OpenJDK community. Each encumberance was either released as free and open source software or replaced with an alternative:

- All the audio engine code, including the software synthesizer, has been released as open source. The proprietary software synthesizer has been replaced by a new synthesizer developed specifically for OpenJDK called *Gervill*,
- All cryptography classes used in the class library have been released as open source,
- The code that scales and rasterizes fonts has been replaced by FreeType
- The native color management system has been replaced by LittleCMS. There is a pluggable layer in the JDK, so that the proprietary version can use the old color management system and OpenJDK can use LittleCMS.
- The anti-aliasing graphics rasterizer code has been replaced by the open source Pisces renderer used in the phoneME project. This code is fully functional, but still needs some performance enhancements

Sun has made continued promises about releasing their plugin and Web Start implementation as part of OpenJDK, but have so far failed to deliver. The only currently available free plugin and Web Start implementation is that provided by IcedTea.

IcedTea and Inclusion in Software Distributions

To be able to bundle OpenJDK in Fedora and other free Linux distributions, OpenJDK needed to be buildable using only free software components. Due to the encumbered components in the class library and implicit assumptions within the build system that the JDK being used to build OpenJDK was a Sun JDK, this was not possible. In order to achieve this goal, a project called IcedTea was started by Red Hat in June 2007. It began life as an OpenJDK/GNU Classpath hybrid that could be used to bootstrap OpenJDK, replacing the encumbrances with code from GNU Classpath.

On November 5, 2007, Red Hat signed both the Sun Contributor Agreement and the OpenJDK Community TCK License. One of the first benefits of this agreement is tighter alignment with the IcedTea project, which brings together Fedora, the Linux distribution, and JBoss, the application server, technologies in a Linux environment. IcedTea is providing free software alternatives for the few remaining proprietary sections in the OpenJDK project.

In May 2008, the Fedora 9 and Ubuntu 8.04 distributions included IcedTea 6, based completely on free and open source code. Fedora 9 was the first version to ship with IcedTea6, based on the OpenJDK6 sources from Sun rather than OpenJDK7. It was also the first to use OpenJDK for the package name (via the OpenJDK trademark agreement) instead of IcedTea.Ubuntu also first packaged IcedTea7 before later moving to IcedTea6. Packages for IcedTea6 were also created for Debian and included in *lenny*. On July 12, 2008, Debian accepted OpenJDK-6 in unstable, and it is now in stable. OpenJDK is also available on openSUSE, Red Hat Enterprise Linux and RHEL derivatives such as CentOS.

In June 2008, Red Hat announced that the packaged binaries for OpenJDK on Fedora 9, built using IcedTea 6, had passed the Technology Compatibility Kit tests and could claim to be a fully compatible Java 6 implementation. In July 2009, an IcedTea 6 binary build for Ubuntu 9.04 passed all of the compatibility tests in the Java SE 6 JCK.

Since August 2008, OpenJDK 7 is runnable on Mac OS X and other BSD distributions.

Collaboration with IBM

On October 11, 2010, IBM, by far the biggest participant in the Apache Harmony project, decided to join Oracle on the OpenJDK project, effectively shifting its efforts from Harmony to OpenJDK. Bob Sutor, IBM's head of Linux and open source, blogged that "IBM will be shifting its development effort from the Apache Project Harmony to OpenJDK"..

Java on Mac OS X

On November 12, 2010, just three weeks after deprecating its own Java runtime port, Apple and Oracle announced the OpenJDK project for Mac OS X. Apple will contribute most of the key components, tools and technology required for a Java SE 7 implementation on Mac OS X, including a 32-bit and 64-bit HotSpot-based Java virtual machine, class libraries, a networking stack and the foundation for a new graphical client.

Clojure

Clojure

Paradigm	functional, multiparadigm
Appeared in	2007
Designed by	Rich Hickey
Stable release	1.2.0 (2010-08-19)

Typing discipline	dynamic, strong
Influenced by	Lisp, Prolog, ML, Haskell, Erlang
Platform	JVM and CLR
License	Eclipse Public License

Clojure (pronounced "closure") is a modern dialect of the Lisp programming language. It is a general-purpose language supporting interactive development that encourages a functional programming style, and simplifies multithreaded programming.

Clojure runs on the Java Virtual Machine and the Common Language Runtime. Clojure honors the code-as-data philosophy and has a sophisticated Lisp macro system.

Philosophy

Rich Hickey developed Clojure because he wanted a modern Lisp for functional programming, symbiotic with the established Java Platform, and designed for concurrency.

Clojure's approach to concurrency is characterized by the concept of identities, which represent a series of immutable states over time. Since states are immutable values, any number of workers can operate on them in parallel, and concurrency becomes a question of managing changes from one state to another. For this purpose, Clojure provides several mutable reference types, each having well-defined semantics for the transition between states.

Syntax

Like any other Lisp, Clojure's syntax is built on S-expressions that are first parsed into data structures by a reader before being compiled. Clojure's reader supports literal syntax for maps, sets and vectors in addition to lists, and these are given to the compiler as they are. In other words, the Clojure compiler does not compile only list data structures, but supports all of the mentioned types directly. Clojure is a Lisp-1, and is not intended to be code-compatible with other dialects of Lisp.

Macros

Clojure's macro system is very similar to that in Common Lisp with the exception that Clojure's version of the backquote (called "syntax quote") qualifies symbols with their namespace. This helps prevent unintended name capture as binding to namespace-qualified names is forbidden. It is possible to force a capturing macro expansion, but this must be done explicitly. Clojure also disallows rebinding global names in other namespaces that have been imported into the current namespace.

Language features

- A compiled language producing JVM bytecode
    - Tight Java integration: By compiling into JVM Byte code, Clojure applications can be easily packaged and deployed to JVMs and app servers without added complexity. The language also provides macros which make it simple to use existing Java APIs. Clojure's data structures all implement standard Java Interfaces, making it easy to run code implemented in Clojure from Java.
- Dynamic development with a read-eval-print loop
- Functions as first-class objects
- Emphasis on recursion instead of side-effect-based looping
- Lazy sequences
- Provides a rich set of immutable, persistent data structures
- Concurrent programming through software transactional memory, an agent system, and a dynamic var system
- Multimethods to allow dynamic dispatch on the types and values of any set of arguments (cf. the usual object-oriented polymorphism which dispatches on the type of (what is effectively) the first method argument)

Examples

Hello world:

```
(println "Hello, world!")
```

GUI Hello World:

```
(javax.swing.JOptionPane/showMessageDialog nil "Hello World" )
```

A thread-safe generator of unique serial numbers:

```
(let [i (atom 0)]
  (defn generate-unique-id
    "Returns a distinct numeric ID for each call."
    []
    (swap! i inc)))
```

An anonymous subclass of `java.io.Writer` that doesn't write to anything, and a macro using that to silence all prints within it:

```
(def bit-bucket-writer
  (proxy [java.io.Writer] []
    (write [buf] nil)
    (close []    nil)
    (flush []    nil)))

(defmacro noprint
  "Evaluates the given expressions with all printing to *out*
silenced."
  [& forms]
  `(binding [*out* bit-bucket-writer]
     ~@forms))

(noprint
 (println "Hello, nobody!"))
```

10 Threads manipulating one shared data structure, which consists of 100 vectors each one containing 10 (initially sequential) unique numbers. Each thread then repeatedly selects two random positions in two random vectors and swaps them. All changes to the vectors occur in transactions by making use of clojure's software transactional memory system. That's why even after 100 000 iterations of each thread no number got lost.

```
(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (vec (map (comp ref vec)
                           (partition nitems (range (* nvecs
nitems)))))
        swap #(let [v1 (rand-int nvecs)
                    v2 (rand-int nvecs)
                    i1 (rand-int nitems)
                    i2 (rand-int nitems)]
                (dosync
                 (let [temp (nth @(vec-refs v1) i1)]
                   (alter (vec-refs v1) assoc i1 (nth @(vec-refs v2)
i2))
                   (alter (vec-refs v2) assoc i2 temp))))
        report #(do
                 (prn (map deref vec-refs))
                 (println "Distinct:"
                          (count (distinct (apply concat (map deref
vec-refs))))))]
    (report)
    (dorun (apply pcalls (repeat nthreads #(dotimes [_ niters]
(swap)))))
    (report)))

(run 100 10 10 100000)
```

Output of previous example:

```
([0 1 2 3 4 5 6 7 8 9] [10 11 12 13 14 15 16 17 18 19] ...
 [990 991 992 993 994 995 996 997 998 999])
Distinct: 1000

([382 318 466 963 619 22 21 273 45 596] [808 639 804 471 394 904 952 75
289 778] ...
 [484 216 622 139 651 592 379 228 242 355])
Distinct: 1000
```

Chapter 14

Java Pathfinder

	Java Pathfinder
Developer(s)	NASA
Stable release	6.0 / November 30, 2010; 12 days ago
Written in	Java
Operating system	Cross-platform
Size	1.6 MB (archived)
Type	Model checking
License	NASA Open Source Agreement version 1.3

Java Pathfinder (JPF) is a system to verify executable Java bytecode programs. JPF was developed at the NASA Ames Research Center and open sourced in 2005. The acronym JPF is not to be confused with the unrelated *Java Plugin Framework* project.

The core of JPF is a Java Virtual Machine that is also implemented in Java. JPF executes normal Java bytecode programs and can store, match and restore program states. Its primary application has been Model checking of concurrent programs, to find defects such as data races and deadlocks. With its respective extensions, JPF can also be used for a variety of other purposes, including

- model checking of distributed applications
- model checking of user interfaces
- test case generation by means of symbolic execution
- low level program inspection
- program instrumentation and runtime monitoring

JPF has no fixed notion of state space branches and can handle both data and scheduling choices.

Example

The following system under test contains a simple race condition between two threads accessing the same variable `d` in statements (1) and (2), which can lead to a division by zero exception if (1) is executed before (2)

```
public class Racer implements Runnable {
     int d = 42;

     public void run () {
          doSomething(1001);
          d = 0;                                    // (1)
     }

     public static void main (String[] args){
          Racer racer = new Racer();
          Thread t = new Thread(racer);
          t.start();

          doSomething(1000);
          int c = 420 / racer.d;                    // (2)
          System.out.println(c);
     }

     static void doSomething (int n) {
          try { Thread.sleep(n); } catch (InterruptedException ix) {}
     }
}
```

Without any additional configuration, JPF would find and report the division by zero. If JPF is configured to verify absence of race conditions (regardless of their potential downstream effects), it will produce the following output, explaining the error and showing a counter example leading to the error

```
JavaPathfinder v6.0 - (C) RIACS/NASA Ames Research Center
====================================================== system under
test
application: Racer.java
...
====================================================== error #1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Racer@13d.d
  main at Racer.main(Racer.java:16)
               "int c = 420 / racer.d;               "  : getfield
  Thread-0 at Racer.run(Racer.java:7)
               "d = 0;                               "  : putfield

====================================================== trace #1
------------------------------------------------------ transition #0
thread: 0
...
------------------------------------------------------ transition #3
thread: 1
```

```
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet[id="sleep",isCascaded:false
,{main,>Thread-0}]
      [3 insn w/o sources]
  Racer.java:22                  : try { Thread.sleep(n); } catch
(InterruptedException ix) {}
  Racer.java:23                  : }
  Racer.java:7                   : d = 0;
...
------------------------------------------------------- transition #5
thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet[id="sharedField",isCascaded
:false,{>main,Thread-0}]
  Racer.java:16                  : int c = 420 / racer.d;
```

Extensibility

JPF is an open system that can be extended in a variety of ways. The main extension constructs are

- *listeners* - to implement complex properties (e.g. temporal properties)
- *peer classes* - to execute code at the host JVM level (instead of JPF), which is mostly used to implement native methods
- *bytecode factories* - to provide alternative execution semantics of bytecode instructions (e.g. to implement symbolic execution)
- *choice generators* - to implement state space branches such as scheduling choices or data value sets
- *serializers* - to implement state abstractions
- *publishers* - to produce different output formats
- *search policies* - to use different program state space traverse algorithms

JPF includes a runtime module system to package such constructs into separate *JPF extension projects*. A number of such projects are available from the main JPF server, including a symbolic execution mode, numeric analysis, race condition detection for relaxed memory models, user interface model checking and many more.

Limitations

- JPF cannot analyze Java native methods. If the system under test calls such methods, these have to be provided within peer classes, or intercepted by listeners
- as a model checker, JPF is susceptible to Combinatorial explosion, although it performs on-the-fly Partial order reduction
- the configuration system for JPF modules and runtime options can be complex

Chapter 15

Strictfp, Groovy (Programming Language) and Java Native Interface

strictfp

`strictfp` is a keyword in the Java programming language that restricts floating-point calculations to ensure portability. It was introduced into Java with the Java virtual machine (JVM) version 1.2.

Basis

The IEEE standard, IEEE 754, specifies a standard method for both floating-point calculations and storage of floating-point values in either single (32-bit, used in Java `float`s) or double (64-bit, used in Java `double`s) precision. Prior to JVM 1.2, floating-point calculations were strict; that is, all intermediate floating-point results were represented as IEEE single or double precisions. As a consequence, errors of calculation, overflows and underflows, could occur. Whether or not an error had occurred, the calculation would always return a valid number; if an overflow or underflow had occurred, that number would be incorrect. Hence, whether an error had occurred was typically not obvious. Since JVM 1.2, intermediate computations are not limited to the standard 32- and 64- bit precisions. On platforms that can handle other representations, those representations can be used, preventing overflows and underflows, thereby increasing precision. For some applications, a programmer might need every platform to have precisely the same floating-point behavior, even on platforms that could handle greater precision. The `strictfp` modifier accomplishes this by truncating all intermediate values to IEEE single- and double- precision, as occurred in earlier versions of the JVM.

Usage

Programmers can use the modifier `strictfp` to ensure that calculations are performed as in the earlier versions; that is, only with IEEE single and double precision types used. Using strictfp guarantees that results of floating-point calculations are identical on all platforms. This can be extremely useful when comparing floating-point numbers.

It can be used on classes, interfaces and non-abstract methods. When applied to a method, it causes all calculations inside the method to use strict floating-point math. When applied to a class, all calculations inside the class use strict floating-point math. Compile-time constant expressions must always use strict floating-point behavior

Examples

```
public strictfp class MyFPclass {
    // ... contents of class here ...
}
```

The Java package java.lang.Math class contains these strictfp methods:

```
 public static strictfp double abs(double);
 public static strictfp int max(int, int);
 public static strictfp long max(long, long);
 public static strictfp float max(float, float);
 public static strictfp double max(double, double);
 public static strictfp int min(int, int);
```

Ideas for further improvement of Java's floating-point performance

The introduction of the strictfp keyword allowed Java to take advantage of the speed and precision of the extended precision floating-point operations supported by x86 CPUs (for all code not explicitly using the strictfp keyword), while those wanting total platform-independency could still have it by using the strictfp keyword. There have been ideas that Java's floating-point behavior should be modified even more to allow faster or more precise calculations on platforms where this is possible.

Groovy (programming language)

Groovy

Paradigm	Object-oriented, scripting
Appeared in	2003

Designed by	JCP
Developer	Guillaume Laforge (Project Manager and JSR-241 Spec Lead)
Stable release	1.7.5 (September 16, 2010; 2 months ago)
Typing discipline	Dynamic, strong, duck
Influenced by	Java, Python, Ruby, Perl, Smalltalk, Objective-C
Platform	JVM
OS	Cross-platform
License	Apache License v2.0

Groovy is an object-oriented programming language for the Java platform. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. It can be used as a scripting language for the Java Platform.

Groovy uses a Java-like bracket syntax. It is dynamically compiled to Java Virtual Machine (JVM) bytecode and interoperates with other Java code and libraries. Most Java code is also syntactically valid Groovy.

Groovy 1.0 was released on January 2, 2007.

History

James Strachan first talked about the development of Groovy in his blog in August 2003. Several versions were released between 2004 and 2006. After the JCP standardization process began, the version numbering was changed and a version called "1.0" was released on Tuesday, January 2, 2007. After various betas and release candidates numbered 1.1, on December 7, 2007, Groovy 1.1 Final was released and immediately rebranded as Groovy 1.5 as a reflection of the many changes that were made.

In July 2009, Strachan wrote on his blog that "I can honestly say if someone had shown me the *Programming in Scala* book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy." Strachan left the project silently long before the Groovy 1.0 release in 2007.

Features

Many (but not all) valid Java files are also valid Groovy files. Although the two languages are similar, Groovy code can be more compact, because it does not require all the elements that Java requires. This makes it possible for Java programmers to gradually learn Groovy by starting with familiar Java syntax before acquiring more Groovy idioms.

Groovy features not available in Java include both static and dynamic typing (with the *def* keyword), closures, operator overloading, native syntax for lists and associative arrays (maps), native support for regular expressions, polymorphic iteration, expressions embedded inside strings, additional helper methods, and the safe navigation operator "?." to automatically check for nulls (for example, "variable?.method()", or "variable?.field").

Groovy's syntax can be made far more compact than Java. For example, a declaration in Standard Java 5+ such as:

```
 for (String it : new String[] {"Rod", "Carlos", "Chris"})
     if (it.length() <= 4)
         System.out.println(it);
```

can be expressed in Groovy as:

```
 ["Rod", "Carlos", "Chris"].findAll{it.size() <= 4}.each{println it}
```

Groovy provides native support for various markup languages such as XML and HTML, accomplished via an inline DOM syntax. This feature enables the definition and manipulation of many types of heterogeneous data assets with a uniform and concise syntax and programming methodology.

Unlike Java, a Groovy source code file can be executed as an (uncompiled) script, if it contains code outside any class definition, or if it is a class with a *main* method, or is a *Runnable* or *GroovyTestCase*. But, unlike some script languages such as Ruby, a Groovy script is fully parsed, compiled, and generated before execution. (This occurs under the hood, and the compiled version is not saved as an artifact of the process.)

GroovyBeans are Groovy's version of JavaBeans. Groovy implicitly generates accessor and mutator methods. In the following code, setColor(String color) and getColor() are implicitly generated; and the last two lines, which appear to access color directly, are actually calling the implicitly generated methods.

```
class AGroovyBean {
  String color
}

def myGroovyBean = new AGroovyBean()

myGroovyBean.setColor('baby blue')
assert myGroovyBean.getColor() == 'baby blue'
```

```
myGroovyBean.color = 'pewter'
assert myGroovyBean.color == 'pewter'
```

Groovy offers simple, consistent syntax for handling *lists* and *maps*, reminiscent of Java's *array* syntax.

```
def movieList = ['Dersu Uzala', 'Ran', 'Seven Samurai']  //looks like
an array, but is a list
assert movieList == 'Seven Samurai'
movieList = 'Casablanca'  //adds an element to the list
assert movieList.size() == 4

def monthMap = [ 'January' : 31, 'February' : 28, 'March' : 31 ]
//declares a map
assert monthMap['March'] == 31  //accesses an entry
monthMap['April'] = 30  //adds an entry to the map
assert monthMap.size() == 4
```

IDE support

Many integrated development environments support Groovy:

- Eclipse, through a plugin
- IntelliJ IDEA, through the Jet Groovy Plugin
- NetBeans, since version 6.5
- TextMate
- JDeveloper, for use with Oracle ADF
- Xappworks, a cloud-based IDE currently in beta & scheduled to enter the market early 2011

Java Native Interface

The **Java Native Interface** (**JNI**) is a programming framework that allows Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly.

Purpose and features

JNI allows one to write native methods to handle situations when an application cannot be written entirely in the Java programming language, e.g. when the standard Java class library does not support the platform-specific features or program library. It is also used to modify an existing application—written in another programming language—to be accessible to Java applications. Many of the standard library classes depend on JNI to provide functionality to the developer and the user, e.g. file I/O and sound capabilities. Including performance- and platform-sensitive API implementations in the standard library allows all Java applications to access this functionality in a safe and platform-independent manner.

The JNI framework lets a native method utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code.

JNI is sometimes referred to as the "escape hatch" for Java developers because it allows them to add functionality to their Java application that the standard Java APIs cannot otherwise provide. It can be used to interface with code written in other languages, such as C and C++. It is also used for time-critical calculations or operations like solving complicated mathematical equations, since native code can be faster than JVM code.

Pitfalls

- subtle errors in the use of JNI can destabilize the entire JVM in ways that are very difficult to reproduce and debug;
- only applications and signed applets can invoke JNI;
- an application that relies on JNI loses the platform portability Java offers (a workaround is to write a separate implementation of JNI code for each platform and have Java detect the operating system and load the correct one at runtime);
- the JNI framework does not provide any automatic garbage collection for non-JVM memory resources allocated by code executing on the native side. Consequently, native side code (such as C, C++, or assembly language) must assume the responsibility for explicitly releasing any such memory resources that it itself acquires;
- error checking is a MUST or it has the potential to crash the JNI side and the JVM.
- on Linux and Solaris platforms, if the native code registers itself as a signal handler it could intercept signals intended for the JVM. Signal chaining should be used to allow native code to better interoperate with JVM.
- on Windows platforms Structured Exception Handling (SEH) may be employed to wrap native code in SEH try/catch blocks so as to capture machine (CPU/FPU) generated software interrupts (such as NULL pointer access violations and divide-by-zero operations) and to handle these situations before the interrupt is propagated back up into the JVM (i.e. Java side code) and in all likelihood resulting in an unhandled exception.
- The encoding used for the NewStringUTF, GetStringUTFLength, GetStringUTFChars, ReleaseStringUTFChars, GetStringUTFRegion functions is not standard UTF-8, but modified UTF-8. The null character (U+0000) and codepoints greater than or equal to U+10000 are encoded differently in modified UTF-8. Many programs actually use these functions incorrectly and treat the UTF-8 strings returned or passed into the functions as standard UTF-8 strings

instead of modified UTF-8 strings. Programs should use the NewString, GetStringLength, GetStringChars, ReleaseStringChars, GetStringRegion, GetStringCritical, and ReleaseStringCritical functions, which use UTF-16LE encoding on little-endian architectures and UTF-16BE on big-endian architectures, and then use a UTF-16 to standard UTF-8 conversion routine.

How the JNI works

In the JNI framework, native functions are implemented in separate .c or .cpp files. (C++ provides a slightly cleaner interface with JNI.) When the JVM invokes the function, it passes a `JNIEnv` pointer, a `jobject` pointer, and any Java arguments declared by the Java method. A JNI function may look like this:

```
JNIEXPORT void JNICALL Java_ClassName_MethodName
  (JNIEnv *env, jobject obj)
{
    /*Implement Native Method Here*/
}
```

The *`env`* pointer is a structure that contains the interface to the JVM. It includes all of the functions necessary to interact with the JVM and to work with Java objects. Example JNI functions are converting native arrays to/from Java arrays, converting native strings to/from Java strings, instantiating objects, throwing exceptions, etc. Basically, anything that Java code can do can be done using `JNIEnv`, albeit with considerably less ease.

For example, the following converts a Java string to a native string:

```
//C++ code
JNIEXPORT void JNICALL Java_ClassName_MethodName
  (JNIEnv *env, jobject obj, jstring javaString)
{
    //Get the native string from javaString
    const char *nativeString = env->GetStringUTFChars(javaString, 0);

    //Do something with the nativeString

    //DON'T FORGET THIS LINE!!!
    env->ReleaseStringUTFChars(javaString, nativeString);
}
/*C code*/
JNIEXPORT void JNICALL Java_ClassName_MethodName
  (JNIEnv *env, jobject obj, jstring javaString)
{
    /*Get the native string from javaString*/
    const char *nativeString = (*env)->GetStringUTFChars(env,
javaString, 0);

    /*Do something with the nativeString*/

    /*DON'T FORGET THIS LINE!!!*/
    (*env)->ReleaseStringUTFChars(env, javaString, nativeString);
```

```
 }
 /*Objective-C code*/
 JNIEXPORT void JNICALL Java_ClassName_MethodName(JNIEnv *env, jobject
obj, jstring javaString)
 {
     /*DON'T FORGET THIS LINE!!!*/
     JNF_COCOA_ENTER(env);

     /*Get the native string from javaString*/
     NSString* nativeString = JNFJavaToNSString(env, javaString);

     /*Do something with the nativeString*/

     /*DON'T FORGET THIS LINE!!!*/
     JNF_COCOA_EXIT(env);
 }
```

Note that C++ JNI code is syntactically slightly cleaner than C JNI code because like Java, C++ uses object method invocation semantics. That means that in C, the `env` parameter is dereferenced using `(*env)->` and `env` has to be explicitly passed to `JNIEnv` methods. In C++, the `env` parameter is dereferenced using `env->` and the `env` parameter is implicitly passed as part of the object method invocation semantics.

Native data types can be mapped to/from Java data types. For compound types such as objects, arrays and strings the native code must explicitly convert the data by calling methods in the `JNIEnv`.

Mapping types

The following table shows the mapping of types between Java and native code.

Native Type	Java Language Type	Description	Type signature
unsigned char	jboolean	unsigned 8 bits	Z
signed char	jbyte	signed 8 bits	B
unsigned short	jchar	unsigned 16 bits	C
short	jshort	signed 16 bits	S
long	jint	signed 32 bits	I
long long __int64	jlong	signed 64 bits	J
float	jfloat	32 bits	F
double	jdouble	64 bits	D

In addition, the signature "L fully-qualified-class ;" would mean the class uniquely specified by that name; e.g., the signature "Ljava/lang/String;" refers to the class java.lang.String. Also, prefixing `[` to the signature makes the array of that type; for example, `[I` means the int array type.

Here, these types are interchangeable. You can use `jint` where you normally use an `int`, and vice-versa, without any typecasting required.

However, mapping between Java Strings and arrays to native strings and arrays is different. If you use a `jstring` in where a `char *` would be, your code could crash the JVM.

```
JNIEXPORT void JNICALL Java_ClassName_MethodName
        (JNIEnv *env, jobject obj, jstring javaString) {
    // printf("%s", javaString);        // INCORRECT: Could crash VM!

    // Correct way: Create and release native string from Java string
    const char *nativeString = env->GetStringUTFChars(javaString, 0);
    printf("%s", nativeString);
    env->ReleaseStringUTFChars(javaString, nativeString);
}
```

This is similar with Java arrays, as illustrated in the example below that takes the sum of all the elements in an array.

```
JNIEXPORT jint JNICALL Java_IntArray_sumArray
        (JNIEnv *env, jobject obj, jintArray arr) {
    jint buf;
    jint i, sum = 0;
    // This line is necessary, since Java arrays are not guaranteed
    // to have a continuous memory layout like C arrays.
    env->GetIntArrayRegion(arr, 0, 10, buf);
    for (i = 0; i < 10; i++) {
        sum += buf[i];
    }
    return sum;
}
```

Of course, there is much more to it than this. Look for links below for more information.

JNIEnv*

A JNI interface pointer (`JNIEnv*`) is passed as an argument for each native function mapped to a Java method, allowing for interaction with the JNI environment within the native method. This JNI interface pointer can be stored, but remains valid only in the current thread. Other threads must first call `AttachCurrentThread()` to attach themselves to the VM and obtain a JNI interface pointer. Once attached, a native thread works like a regular Java thread running within a native method. The native thread remains attached to the VM until it calls `DetachCurrentThread()` to detach itself.

To attach to the current thread and get a JNI interface pointer:

```
JNIEnv *env;
(*g_vm)->AttachCurrentThread (g_vm, (void **) &env, NULL);
```

To detach from the current thread:

```
(*g_vm)->DetachCurrentThread (g_vm);
```

Advanced uses

Native AWT painting

Not only can native code interface with Java, it can also draw on a Java `Canvas`, which is possible with the Java AWT Native Interface. The process is almost the same, with just a few changes. The Java AWT Native Interface is only available since J2SE 1.3.

Access to assembly code

JNI also allows direct access to assembly code, without even going through a C bridge. Accessing Java applications from assembly is also possible in the same way.

Microsoft's RNI

Microsoft's proprietary implementation of a Java Virtual Machine (Visual J++) had a similar mechanism for calling native Windows code from Java, called the **Raw Native Interface** (**RNI**). However, following the Sun - Microsoft litigation about this implementation, Visual J++ is no longer maintained.