LogOn
Technology Transfer
Bleichstraße 5
61476 Kronberg, Germany
Tel.: +49-6173-9558-50
Fax.: +46-6173-9558-68
e-mail: logon@ltt.de

*LogOn announces the dates for the forthcoming Events:*

**LogOn Info Days 2004**

| | | |
|---|---|---|
| **OMG and Java™ Info Day Zurich** | May 24, 2004 | www.ltt.de/omg.java-days.2004/zurich |
| **Security, Storage and Linux Info Day Zurich** | May 25, 2004 | www.ltt.de/security.storage.linux-days/zurich |
| **OMG and Java™ Info Day London** | June 10, 2004 | www.ltt.de/omg.java-days.2004/london |
| **OMG and XML Info Day Budapest** | June 11, 2004 | www.ltt.de/omg.xml-days.2004/budapest |

More Events at **www.ltt.de/events/2004.shtml**

## Experts' Corner: Ciarán Bryce



*RESEARCH AND DEVELOPMENT*

### Isolates: A New Approach to Multi-Programming in Java Platforms (May '04)

***By Ciarán Bryce***

*This month's article describes recent research led by Sun Microsystems to develop Java technology for multi-application environments.*

The Java programming language has been popular for almost a decade, ever since it became possible to run Java programs within a World-Wide Web browser. The language is now used to program applications for a variety of environments, from hand-held devices, to desktops, to Web servers.

Several reasons explain the popularity of the Java language. It has safety and security features that are extremely useful for applications and which are absent from the C and C++ languages. A key safety feature is the absence of “pointer semantics” that allows programs to manipulate memory locations directly, and which is a source of many errors in C/C++ programs. Concerning security, the Java language and environment incorporates a protection model that allows an application to curtail the privileges of code extensions, like applets, that are linked into the application at runtime. Yet another reason for the success of Java is its “write-once, run anywhere” approach, which enables a compiled Java program to be run in any Java environment without re-compilation.

The Java language is likely to remain popular for a long time to come. One reason is that it has obtained a critical mass of users and programmers. There is now a huge variety of libraries available for Java applications, notably for graphical user interfaces (GUIs), and elaborate development environments exist, e.g., Eclipse (see http://www.eclipse.org).

Support for executing Java programs is provided via the Java environment. Sun Microsystems and its partners have developed several editions of the environment: the Java *Standard* edition for desktop machines (J2SE), a series of *Micro-editions* for small devices (J2ME), and finally an *Enterprise* edition for Web server computing (J2EE). The main difference between these is the core set of libraries supported, J2ME being an environment drastically reduced in size so that it can run on small-memory devices, and J2EE coming with a range of packages that are not part of the J2SE environment.

***Single User, Single Application***

The Java runtime environment thus executes in a variety of settings. Despite this variety, the aim of each environment remains the same: *support the execution of a Java program*. What this actually means is that a Java runtime environment executes a single Java program or application, on behalf of the user that launches the application. There is a one-to-one correspondence between a Java program and the Java environment in which it runs.

The idea of a Java environment running on behalf of a single user and application is evident also in the Java platform security model, originally known as the *sandbox*, and then evolved to a general protection domain model. The aim of this model is to reduce the privileges associated with code that is dynamically linked into the application. To put this into context, a feature of Java environments is that the user can specify where the application gets its code from, e.g., it can be loaded from disk or from a network site as is the case for applets. Code from remote sites might not necessarily be trusted, so the security policy of the environment specifies the restricted privileges that code possesses based on the origin and signatures bound to the code. For instance, it is typical for platforms to forbid untrusted code from manipulating files co-located on the platform. Thus, the spirit of the security model is clear: a user runs an application in a Java environment; the environment has the same privileges as the user, but code that the user does not trust is granted restricted access to resources.

Though the Java environment is single user and single application, it is being increasingly used in computer system environments that are multi-application and even multi-user. Take the example of a simple desktop. A Java environment can be included in several applications that are concurrently launched, e.g., a Web browser or a development environment. In a multi-user operating system environment, multiple Java programs can be concurrently run on behalf of different users.

So what happens when several Java applications are concurrently run? Each Java environment runs independently as a separate operating system (OS) process. This is denoted below as the *elementary multi-programming* approach, and is illustrated in Figure 1. Recall that in OS environments, a process represents the abstraction of a program in execution. Several processes can be run at the same time, and each can be started and stopped independently. Moreover, each OS process runs in a protected boundary, which is implemented using the address space concept that is supported by hardware. This ensures that a program (in a process) is unable to gain access to any data belonging to any other process in the system. This is a very important security guarantee in system environments where independent programs run, perhaps on behalf of different users.

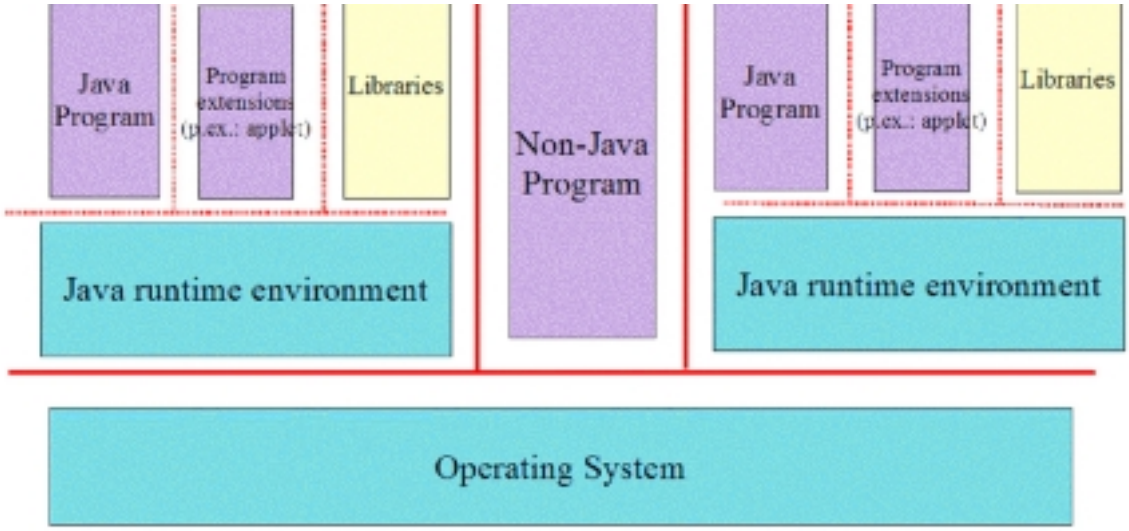


**Figure 1**

- Elementary Multi-Programming

**Figure 1** shows a platform running three processes over an operating system, two of these are Java environments. The complete lines between the processes, and between the operating system and the processes, represent a hardware-enforced protection boundary. Each Java environment process runs a Java program, which may use code extensions like applets and other libraries. The dotted line in the environments represent the protection boundary implemented by the Java security model that protects a program and platform resources from untrusted code. These protection boundaries are implemented in software.

There are advantages to running independent Java environments as independent OS processes in the manner illustrated in Figure 1. First, since the Java applications are generally independent, the approach enforces a high level of protection, since each application essentially exploits the protection features of the OS. Of course, protection is already a feature of Java environments, but OS address space protection goes further: it ensures that there is absolutely no data sharing between independent applications, so any error in one application can have no effect on other Java applications.

A second advantage of the elementary multi-programming approach is manageability. Applications that are independent can be treated as such. For instance, applications can be started and stopped independently of each other. Further, the platform privileges assigned to the process running one Java application can be orthogonal to the privileges granted to another application. This is particularly useful when the Java applications are run on behalf of different users. In short, *running each Java application as an independent process combines the advantages of the Java programming language with the advantages of OS multi-programming*.

### *Towards Real Multi-programming in Java*

There are limits however to the elementary approach to multi-programming just outlined. These limits are so significant that researchers from Sun Microsystems and other institutes have come up with an alternative approach. This approach is presented in the *Java Specification Request (JSR) 121*, which proposes the **Java Isolation API** (see http://bitser.net/isolate-interest). Before giving an overview of the API, the limitations of using OS multi-programming facilities for Java multi-programming are considered.

The first restriction of elementary multi-programming is that it is inappropriate for small devices where the developer may wish to run independent applications safely, but where the platform OS does not have, or cannot use, hardware-enforced protection via address spaces. Without address spaces, there is no simple way to deploy multiple independent Java programs on the device.

A second shortcoming of elementary multi-programming is that there is no Java programmable way of controlling all of the Java programs present on the platform. A user starts each application using a non-Java mechanism, usually with a shell command. In effect, the user steps out of the Java world to manage the Java programs. This is contrary to the spirit of Java computing, where increasingly the aim is to control platform resources from inside of Java programs. For instance, Java programs can create and open files, and communicate with programs on remote platforms using sockets. In a multi-programming environment, the natural requirement for an API is that it allow independent Java programs to be controlled. This is a key goal of the Isolation API.

Another restriction of elementary multi-programming is *efficiency* on current platforms. If *N* Java programs are simultaneously run on a platform, then there are *N* Java environments and consequently *N* OS processes simultaneously activated. The core of the Java runtime environment is quite large, and having this instantiated *N* times is costly. A further price paid is in execution time. In fact, the "slowest" part of a Java application is its start-up, as the whole environment must be put in place before the program can actually run. Again, this start-up cost is paid *N* times on a platform where *N* Java programs are present, even though advanced Java implementations can attempt to optimize sharing through careful use of the OS memory management features.

***Isolates: Java Program Containers***

The core idea of the Isolation API is to run all independent programs in the same Java runtime environment. Each independent program is run inside an *isolate*. An isolate is a Java program unit that shares no application data or mutable class state with other isolates. An isolate can be started and stopped independently of other isolates. Each isolate runs a program that has its own *main* method. The idea is that independent programs run in isolates thus have the same level of protection as when they are run in different OS processes.
The idea is illustrated in **Figure 2**. Here, there is a single Java environment running several Java

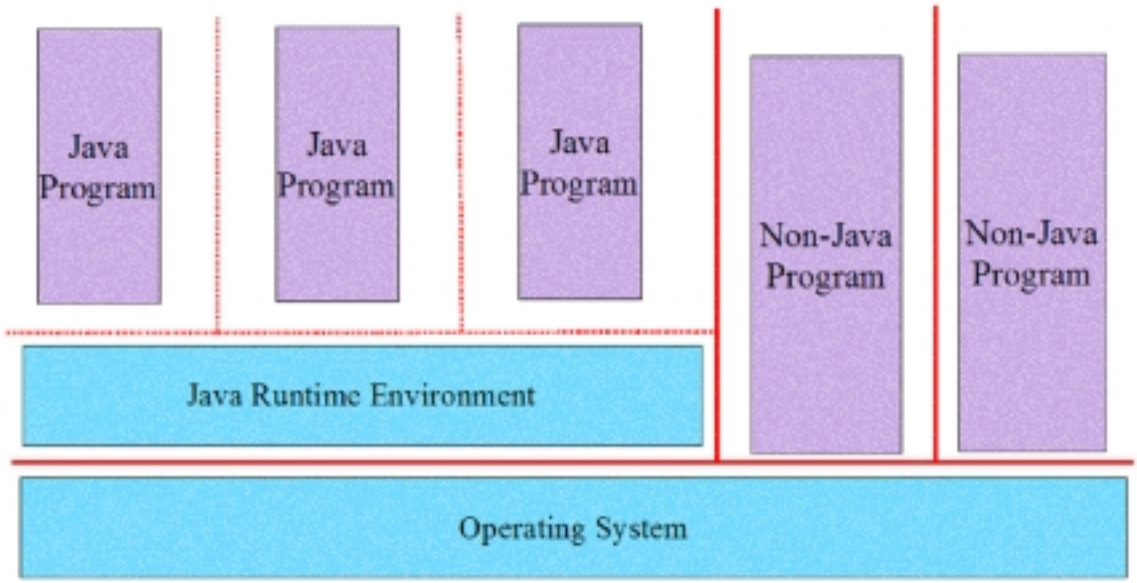


**Figure 2** - Programs and Isolates

programs. Each program runs in an isolate, so there is a protection boundary around each that forbids sharing with other program isolates. Several other programs can be run at the same time inside other OS processes. The boundaries that separate the OS processes are again those that are implemented with the help of the underlying hardware. The boundaries that separate the isolates inside of the Java runtime environment are those implemented by the Java environment. The Isolation API specification does not say how these barriers are implemented; the important point is that they be there. One implementation option is for the runtime to use software techniques to enforce isolation. An alternative is to have a Java environment that is distributed across several OS processes, and for the environment to use hardware address space boundaries to enforce isolation. Whatever the strategy chosen, the behavior of the Isolate application must be the same.

Details of the Isolation API can be found at the site (http://bitser.net/isolate-interest/). The main goal of the API is to allow the creation and termination of isolates, as well as for communication between isolates. As said, there is no direct sharing between isolates. Communication is done using traditional inter-process communication mechanisms such as sockets or files, or through an API communication mechanism called *links* which are simple data channels that also allow isolates to synchronize between themselves.

***Aside**: The expert Java programmer may wonder at this point why isolation cannot be implemented by the current Java security model. This model is, after all, designed to protect a program from another program (perhaps in the form of an applet). This model is based on the Java environment's class loading mechanism – which is responsible for dynamically loading and linking code into the application. In effect, a Java environment may employ several class loaders. A program linked by one class loader is prohibited from using code and data belonging to a program instantiated by another loader. There have been several research efforts to define isolated Java environments by extending this mechanism. However, the class loader security model is very controversial since class loaders do share basic classes. This sharing can be exploited by programs to exchange information with programs instantiated via other loaders. Further, it prevents the environment from safely removing programs that belong to one class loader, since this could cause another program's data to become inconsistent.*

***The Benefits of Isolates***

So fundamentally, what will all of this change for Java and Web application developers? There are several aspects to the answer.

One aspect is that multi-programmed and multi-part application environments become **platform neutral**, since the code to life-cycle manage these programs is pure Java, and no longer relies on scripts and the underlying OS. This is not just a management benefit, but it is also positive for **security** since mistrusting though cooperating programs can be run in the same Java environment. The security benefit is very important in J2EE environments, as these can be composed of application components from different sources, and which the administrator might prefer to run in different protection environments.

Another benefit for application developers is **scalability**. It was mentioned above that a disadvantage of the elementary multi-programming model is the replication of the runtime environment and the associated space and time costs. In the Isolate API approach, as far as the developer is concerned, there is only a single environment that runs all isolates, so the mentioned inefficiency can be removed.

A final advantage of the Isolation API approach is that, with the implementation of clear boundaries around Java programs and components of multi-part applications, it becomes possible to consider other features found in multi-programmed environments. An example of this is resource management, which is considered further on in the article.

***Sidebar: Threads and Isolates***

The familiar Java programmer may wonder about the relationship between threads and isolates. Indeed, there are many similarities between them. Each is concurrently instantiated with others. Both threads and isolates can cooperate to get a job done, and at the same time, compete for the CPU and other platform resources. Both threads and isolates are represented by a core class, *java.lang.Thread* and *javax.isolate.Isolate* respectively, that permit the life-cycle of the entities to be managed. Since threads exist in Java since its beginning, then one may ask why isolates are needed.

The crucial difference between the two abstractions is protection. Threads can share objects, and their goal is to make calculations on objects as efficient as possible. Isolates on the other hand do not share any objects; they can contain communication endpoints that reference the same socket or link. This allows programs in isolates to be completely independent since stopping and removing an isolate cannot possibly leave the data of another isolate in an inconsistent state. An isolate is considered "stopped" when all of its threads have been stopped, and further execution in the isolate is therefore impossible. This independence is not the case for threads, and underlines why class-loader based container systems are so unsatisfying – reliable lifecycle management of threads within a class-loader scope is difficult or impossible to guarantee.

Isolates and threads co-exist in a Java environment. An isolate can contain any number of threads that share the objects of the isolate. One can view an isolate as a traditional Java environment; a program written before the Isolation API can be run unmodified within an isolate. An isolate encourages a coarse grained concurrency within an application.

***Sidebar: Java Specification Request 121***

The Isolation API is formally defined in the *Java Specification Request, number 121,* produced by the expert group listed at http://bitser.net/isolate-interest/jsr-expert-group.html. A Java Specification Request, or JSR for short, is a document produced by an expert group that proposes an addition to future releases of Java platforms. In order to ensure that the proposed specification is in line with the needs of Java users, the process of producing a JSR is rigorously defined by a procedure known as the *Java Community Process (*JCP, see http://jcp.org*)*.

The JCP tries to maximize the involvement of Java developers in the JSR. The first step in a JSR is the formation of an Expert Group. This is mainly composed of representatives from major development companies, e.g., Sun Microsystems, Borland Software Corporation, Hewlett-Packard, IBM, Oracle, Apple Computer, IONA Technologies, Cisco Systems and Fujitsu. Individuals may also be expert group members, subject to a specific JCP agreement covering their participation.

Members of the Expert Group then proceed to produce a draft of the specification. This specification must be accompanied by at least one reference implementation and a validation suite, to show its feasibility. Expert Group members vote on the draft specification. If this vote succeeds, the specification is placed in the public domain for a public review. This review gives Java developers a chance to send comments on the JSR to the expert committee, which are then taken into account for the final version.

As a JSR nears completion, a decision is made about whether the JSR is ready to proceed or if more work is needed before the JSR becomes final. The general availability of the specification, RI and validation suite (known as the *Technical Compatibility Kit,* or TCK) coincides with the JSR reaching its final stages. When a JSR delivers an API for the core Java packages, as a *java.** package, the RI of the JSR is delivered as part of Sun's major J2SE release. Other editions typically include subsets of the new API (J2ME), or

include them as part of a larger specification (J2EE). For optional packages, those furnished as *javax.**, there may be no direct binding to a Sun Java release.

JSR 121 completed its public review but was unable to rendezvous with the major Sun J2SE release currently under beta test. Consequently, a new public review specification and RI are being prepared with emphasis on package boundaries that exactly correspond to the needs of J2ME/CLDC, J2ME/CDC and J2SE. The architectural rework is nearing completion, and details are available at http://bitser.net/isolate-interest.

***Current Research on Isolates***

An important implementation of the Isolation API is the *Multi-tasking Virtual Machine* (MVM), developed at Sun Microsystems in Mountain View, California (see http://research.sun.com/projects/barcelona). MVM is built from the code base of the Java HotSpot virtual machine, version 1.3.1. The implementation strategy chosen in MVM is to run the Java runtime within a single OS process. The HotSpot runtime was modified so that core classes in the environment are replicated per isolate, thus ensuring the inter-isolate protection.

One of the research and development efforts currently underway at Sun Microsystems is to adapt the Isolation API to the J2ME platform. The goal of this work is to be able to run independent application programs on cell phone devices without requiring several OS address spaces.

Another research effort around isolates is *Resource Management*. This is the means of making programs accountable for the platform resources they consume, which has been conspicuously absent from Java platforms until now. Resources include memory, CPU time, sockets, etc. In short, a resource is anything measurable that programs compete for, and for which a shortfall results in performance change. Resource management is particularly important in multi-programmed environments since a badly or maliciously coded program could attempt to needlessly consume resources, and therefore starve other programs of resources. Poor resource management in a multi-task environment can lead to sub-optimal performance, and creates the possibility for programs to launch denial of service attacks on others.

A proposed Java API for resource management can be found at http://research.sun.com/techrep/2003/abstract-124.html. The particularity of this API is that it not only deals with traditional resources such as memory and CPU, but also with higher-level resources such as database connections, and can handle any class of resource that the application programmer cares to define.

A further element of work on Isolates is to adapt the API specification to a multi-user environment. In this case, each isolate is bound to a different user ID, which means that the privileges of the isolate for the underlying platform resources reflect the privileges accorded to the isolate's controlling user. This is another significant step towards making the developers' Java environment even more independent of OS platform specific details (see http://research.sun.com/projects/barcelona).

Another early implementation of the Isolation API came from a group led by Jay Lepreau at the University of Utah. The system is the JanosVM (see http://www.cs.utah.edu/flux/janos/janosvm.html) and implements the public review version of the API.

Researchers at SAP are currently investigating “M-to-N” implementations of the Isolation API. This research looks at compromise implementation strategies for isolation in Java environments. The aim is to allow for a variable and configurable number of underlying OS processes that run the Java environment, and to map any of the M isolates in the Java application to any of the underlying N processes, where M is much greater than N.

**Acknowledgments**. I would like to thank Pete Soper and Grzegorz Czajkowski from Sun Microsystems for their valuable feedback on the contents of this paper.