Orthogonal Persistence for the Java™ Platform: Specification and Rationale

Mick Jordan and Malcolm Atkinson

Orthogonal Persistence for the Java™ Platform: Specification and Rationale

Mick Jordan and Malcolm Atkinson

SMLI TR-2000-94 December 2000

Abstract:

Orthogonal persistence provides the programmer with persistence for all data types, with minimal impact on the programing model or development process. We motivate the addition of orthogonal persistence to the Java™ platform, and show how this results in a simple and appealing application development model. The overall goal is to provide the illusion of continuous computation in the face of system shutdowns, planned or unplanned. This is achieved by checkpointing the state of the system periodically to stable memory.

We describe how the principles of orthogonal persistence are applied to the Java™ programming language and specify the small set of changes to the Java language specification and core libraries necessary to fulfill these principles. We describe the rationale for our particular choices, informed by the experience with the PJama prototype implementations. Finally, the programming model for managing state that is external to the Java™ virtual machine is discussed in detail.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
mick.jordan@sun.com
mpa@dcs.gla.ac.uk

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>. The entire technical report collection is available online at http://www.sun.com/research.

Orthogonal Persistence for the Java™ Platform: Specification and Rationale

Mick Jordan[1] and Malcolm Atkinson[2]

1. Sun Microsystems Laboratories, M/S MTV 29-112, 901 San Antonio Road, Palo Alto, CA 94303-4900, USA.
Email: mick.jordan@sun.com
2. University of Glasgow, Department of Computing Science, 8-17 Lilybank Gardens, Glasgow G12 8QQ, Scotland.
Email: mpa@dcs.gla.ac.uk

Contents

Chapter 1

Motivation

1.1 Introduction

This chapter describes the motivation for orthogonal persistence for the Java™ platform (OPJ). The seminal paper on this approach to persistence is entitled "Orthogonally Persistent Object Systems" by Atkinson and Morrison [4]. According to Atkinson and Morrison, orthogonal persistence is characterized by three simple principles:

- ***Type Orthogonality***: persistence is available for all objects irrespective of type.
- ***Persistence by Reachability***: the lifetime of each object is determined by reachability from a set of *root* objects.
- ***Persistence Independence***: code is identical whether it is operating on short-lived or long-lived objects.

The principles are language-independent and can be applied to any programming language. However, the effect is broadly the same, which is to say that persistence is completely transparent to the application programmer. This feature greatly enhances software reuse, and has a productivity impact similar to the provision of virtual memory and automatic main-memory management (garbage collection).

Central to the model provided by orthogonal persistence is the abstraction of a durable *single-level* object store.[1] This is in marked contrast to other persistence mechanisms in which the separation between programming language memory and a durable external store is explicit in the programming model.

1.2 Orthogonal Persistence and the Java™ Programming Language

The Java™ programming language provides the developer with a simple but powerful object model, a strong type system, automatic main-memory storage management, and concurrency through lightweight threads. Within the closed world of an executing Java™ technology-based application ("Java application"), these properties are extremely useful, vital even, in the timely development of reliable and robust applications. However, at present, the Java platform provides no satisfactory way of maintaining these properties beyond the single execution of a Java™ virtual machine. Instead, the programmer must deal explicitly with saving the state of an application, using one of a variety of ad hoc persistence mechanisms—file input/output, object serialization or relational database connectivity—none of which approach complete support for the full computational model. This lack of completeness, while only a minor nuisance for simple applications, becomes a serious problem as application complexity increases. In effect, without orthogonal persistence, the full benefits of the Java programming language model are diminished because long-term data has to be defined with a separate model, which, because of its longevity, becomes dominant.

1.2.1 Current Approaches to Persistence

At the time of writing, the dominant persistent data model in the enterprise remains the relational model, represented in practice by the SQL [18] language. The Java technology programmer interfaces with SQL either directly through JDBC™ [25] or SQLJ [5]; indirectly through some object-relational mapping software [23]; or indirectly through a framework such as Sun's Enterprise JavaBeans™ architecture [22], or IBM's San Francisco Framework [19].

In the arena of file-based persistent storage, ad hoc file formats remain widespread. The Java platform has contributed a significant number of additional file formats including property files and JAR archives, despite also providing a gen-

1. Similar to the abstraction provided by virtual memory but also providing *durability.*

eral object persistence mechanism through Java™ technology-based object serialization (“Java object serialization”) [26]. Unfortunately, Java object serialization suffers from a number of problems as a persistence mechanism, most notably that it is not completely transparent, does not scale well, and does not support class evolution adequately. Ironically, lack of orthogonality is the root cause of these problems.

Recently the eXtensible Markup Language (XML) [28] has recently become popular as a common framework for file formats. In particular, XML has begun to subsume the use of Java object serialization. However, XML still requires a parser and additional “boilerplate” code in order to be utilized by a Java application.

1.2.2 The Benefits of Orthogonal Persistence

It cannot be denied that the Java platform has been remarkably successful at providing an integration mechanism for applications that require access to the external systems and file-based or SQL-based databases which dominate the enterprise today. The value of bundling powerful user-interface support with convenient access to such data, in a platform that is also portable, is considerable.

However, for applications that are predominantly object-based, the *requirement* to map to SQL or XML simply to persist an object graph, is an added burden both during development and deployment. In addition, the fact that the mapping is typically incomplete[1] requires developers to carefully work around the missing language features. Finally, the additional layers may add considerable runtime cost to the application.

OPJ differs from these approaches by supporting persistence of the complete computational model of the Java™ programming language. It achieves this with very few modifications to the JLS [11]. In essence, OPJ extends the automatic memory management of the Java platform to encompass stable memory. OPJ is also unique among persistence mechanisms for the Java platform in requiring support for persistent threads. This results in a very powerful and intuitive programming model that we call *continuous computation*, which directly supports the 7*24 requirements of modern enterprise systems.

1. I.e., it is not orthogonal to type.

Chapter 2

Specification

2.1 Introduction

Orthogonal persistence for the Java platform (OPJ) provides persistence for the full computational model that is defined by the Java programming language specification (JLS) [11]. Persistence is defined as the ability for the computation state to survive in stable storage, across multiple executions of a Java virtual machine [15], and in the face of system and application failures. The aim of orthogonal persistence is to provide a single, uniform, computational model for all aspects of an application that deals with long-lived data. The model covers the definition of data and its behavior, data and application storage and application development.

Orthogonal persistence [4] is characterized by three simple principles:

- ***Type Orthogonality***: persistence is available for all objects irrespective of type.
- ***Persistence by Reachability***: the lifetime of each object is determined by reachability from a set of *root* objects.
- ***Persistence Independence***: code is identical whether it is operating on short-lived or long-lived objects.

This specification defines how these principles are applied to the Java programming language and the Java virtual machine.

The OPJ specification derives from earlier research into persistence [4] and the experience gained over several years with a succession of prototypes of orthogonal persistence for the Java platform. It has also been influenced by other persistence models for the Java platform. However, these models were most helpful in demonstrating the problems that arise from a lack of orthogonality rather than contributing directly to the specification. In general, our experience has led us to simplify the specification and move closer to the fundamental computational model of the Java programming language, rather than expanding the Application Programmer Interface (API). Consequently, in contrast to many Java technology-based APIs, this specification does not attempt to embrace a wide range of existing persistence mechanisms. Rather, in the same spirit as the Java programming language specification, it attempts to simplify a complex area by providing support at the Java programming language level.

2.2 Intent of the Specification

The general intent of this specification is that the complete computational environment of an executing OPJ virtual machine can be made persistent, in the form of a *suspended computation*, by an operation referred to as a *checkpoint*, and then resumed at some later time, by an operation referred to as a *resumption*, possibly on a different machine. Except where explicitly provided for in the specification, it is intended that the checkpoint operation be transparent to an application. The specification does not prescribe the mechanism by which a checkpoint is achieved, in order to provide flexibility of implementation. However, it is expected that the checkpoint operation will have a latency that is proportional to the amount of data modified or added since any previous checkpoint, independent of the total amount of reachable data. Note that unless explicitly modified by the OPJ specification, the JLS is in effect. Throughout this document we use the term OPJ virtual machine to denote a virtual machine that implements this specification.

2.3 Form of the Application Programming Interface

Since orthogonal persistence requires a broad layer of support from the Java virtual machine, the most appealing way

to specify the API is to augment the standard classes. The `Runtime` class is the most obvious choice, and for simplicity and clarity, this specification takes that approach. However, in Appendix A we present alternative specifications that define additional classes in the style of a standard extension and as a third-party API. The reference implementation of OPJ adopts the third-party API style using a package name of `org.opj`.

2.4 Applying the Principles of Orthogonal Persistence

2.4.1 Type Orthogonality

All the types defined in the JLS—including all user-defined subclasses of `java.lang.Object`—can be used in an OPJ application, including all of the core classes, in particular `Class`, `ClassLoader` and `Thread`. Arrays and arbitrary graphs of classes and instances are supported. The binding between an object instance and its associated `Class` instance is preserved across resumptions, as is the binding between a `Class` and its associated `ClassLoader`. The consequence of persistent `Thread` instances is that, on resumption, active threads continue from their state at the previous checkpoint. Special mechanisms are defined for handling threads and classes that refer to state that is external to the OPJ virtual machine.

2.4.2 Persistence by Reachability

An object is *persistence reachable* if and only if it is reachable by the standard rules for object reachability in the Java programming language *and* the object is not *transient*. An object is transient if and only if it is reachable solely through chains of references that pass through instance or static variables declared with the `transient` modifier.[1] An object is made persistent at a checkpoint if it is persistence reachable. Note that this is intentionally not "if and only if". An implementation is encouraged to avoid making unreachable objects persistent, but it is not required to do so, as such objects will eventually be garbage collected.

A class `Class` instance is persistence reachable if: a) it is directly referenced under the standard rules for object reachability as described above; or b) is referenced by a `ClassLoader` instance that is itself persistently reachable; or c) if at least one of its instances is persistently reachable; or d) or if some array of its instances is persistently reachable. A recent clarification to the JLS [11] states that the *bootstrap classloader* instance is always reachable. It follows that all classes loaded by the bootstrap classloader (bootstrap classes) are persistence reachable, as are all the objects that are persistence reachable from static variables in those classes.

Any *resolved* [15], Chapter 5, symbolic reference to a class `C` in the definition of some class `CC` will ensure that if `CC` is persistence reachable, then so is `C`. Note that this constraint does not restrict an implementation in its choice of eager or lazy resolution of symbolic references. However, once resolution has occurred the binding must be maintained.

The `transient` modifier acts to prune the reachability graph. On a resumption, and before any active use of an object, all member variables marked `transient` are reset to their default values. An implementation is expected to exploit this to exclude objects that are reachable only through transient variables in the set that is made persistent, but this is not a requirement.

2.4.3 Persistence Independence

An OPJ virtual machine must accept standard classfiles as generated by any compliant Java™ technology-based compiler ("Java compiler"). In particular, special classfile (bytecode) post processing for execution in an OPJ environment must not be required. This constraint supports the principle of persistence independence, since it ensures that any Java™ classfile that conforms to the JLS can be made persistent.

1. See section 2.7 for a clarification of conflicts between the use of `transient` by Java object serialization and OPJ. Note also that formal parameters and local variables of methods may not be declared with the transient modifier.

2.5 Execution

The JLS chapter 12 entitled Execution, discusses the life-cycle of a Java virtual machine, including start-up and exit. Orthogonal persistence augments this model by allowing the state of the computation to be captured at certain points by a *checkpoint* operation. The outcome of a successful checkpoint operation is a durable representation of the computation that may be *resumed* (continued) at some future time. The durable representation of the computation is referred to as a *suspended computation* and is stored in a *persistent store*. Typically, the persistent store will reside on a disk, although the specification permits other mechanisms as long as they provide adequate durability. This specification does not prescribe the format of the persistent store, which is implementation dependent. The expectation is that a computation may be suspended on one machine and resumed on a different machine, in keeping with the general commitment to architecture neutrality of the Java platform, but this is not a requirement. The mechanism by which this is achieved is implementation dependent.

OPJ introduces a subtlety in the meaning of virtual machine exit, due to the separate notions of the suspended computation and the computer process that is used to execute the virtual machine. When an OPJ virtual machine suspends operation, the associated computer process does indeed exit. Abstractly, however, the computation still exists in its suspended form in the persistent store.[1]

2.5.1 OPJ Virtual Machine Start-Up

The JLS specifies that "a Java virtual machine starts execution by invoking the main method of some specified class, passing it a single argument, which is an array of strings". The JLS does not define the details of how this class is specified to the Java virtual machine, nor how other environment-related information is communicated. OPJ distinguishes such an *initial* execution from a *resumption* execution of a suspended computation.

An OPJ virtual machine will typically require the location of the persistent store to be specified for both an initial execution and a resumption execution, although the location might be implicit. The way that the persistent store is specified is implementation dependent.

It is expected that, in the common case, users of OPJ applications will only experience resumption executions because applications will be deployed in persistent stores as a suitable suspended computation. Such stores can act as pre-configured applications, akin to Java™ technology-based archive files, but more robust and faster to start, due to the strong consistency guarantees provided by OPJ.

2.5.1.1 Initial Execution

An initial execution corresponds directly to the start-up of a normal Java virtual machine, in which case the associated persistent store is conceptually empty; that is, it does not contain a suspended computation.

2.5.1.2 Resumption Execution

A resumption execution resumes a suspended computation from a persistent store. An OPJ virtual machine may resume a computation in one of two ways depending on whether the suspended computation contains any active threads. If there are no active threads, a new main class *must* be specified to the OPJ virtual machine and a new thread is created in which to execute its main method. If active threads do exist in the suspended computation, then a new main class is optional and the computation resumes by continuing one of the active threads. In either case, any registered listeners for resume events (see section 2.8) are invoked, possibly concurrently with other threads.

2.5.2 OPJ Virtual Machine Checkpoint

A Java application may initiate a checkpoint explicitly or it may occur implicitly as part of the standard termination mechanisms provided by the Java platform. To explicitly initiate a checkpoint the `Runtime` class is augmented with two methods, `checkpoint` and `suspend`. Both of these methods generate suspended computations. They differ in that after `checkpoint` the computer process executing the OPJ virtual machine continues, whereas after `suspend` it terminates.

1. There is no programming interface to terminate the persistent form of the computation.

The checkpoint operation consists of two phases. In the first phase, any registered listeners for `CHECKPOINT` events (see section 2.8) are invoked. This phase may proceed concurrently with other threads. After all listeners have returned, the resulting state of the Java™ technology-based computation ("Java computation") is atomically made durable in the persistent store by a process called *stabilization*. After stabilization the OPJ virtual machine will continue execution if the `checkpoint` method was invoked. In the case of `suspend`, all registered shutdown listeners are invoked and, after they have all returned, the OPJ virtual machine process terminates.

A stabilization initiated by one thread must be atomic with respect to all other threads in the same OPJ virtual machine. That is, the complete state of the computation across all threads is captured by the stabilization before the other threads are permitted to alter the state. This can be achieved by simply suspending the other threads, but other more efficient implementations are possible. After a successful checkpoint, the computation resumes by continuing some active thread. Note that at a stabilization, threads may be suspended holding locks or waiting for notifications and this state must be preserved correctly in the suspended computation. As part of the checkpoint operation, each thread's working memory is flushed to main memory, using the terminology of chapter 17 of the JLS. In correctly synchronized programs, this will have no visible effect, but may cause additional non-determinism in programs that do not correctly synchronize access to shared variables.

When one thread initiates a checkpoint, it is possible that another thread may either be executing native code or be executing a Java™ method that was invoked from native code. Generally, it will not be possible for an OPJ virtual machine to capture the state of such threads in a suspended computation, as there will not be sufficient information available in the contents of the native stack frames. This situation is handled by capturing the state of the thread such that the stack is cut back to the frame of the caller of the native method. The state of the thread is recorded in such a way that, when resumed, it will return from the native method with a `CheckpointInNativeMethodException`, which is a subclass of `RuntimeException`. A class that invokes a native method should be prepared for this eventuality and provide appropriate recovery code, typically in conjunction with the event handling mechanisms described in section 2.8.

2.5.3 OPJ Virtual Machine Abort

The standard `Runtime.halt` method provides a way to terminate the computer process that is executing the OPJ virtual machine without invoking a checkpoint operation. In consequence, all changes to the computation state since the last checkpoint—or from the last resumption if no checkpoint has occurred—are lost.

2.5.4 OPJ Virtual Machine Exit

In addition to `Runtime.suspend` method provided by OPJ, the JLS specifies other ways that a Java virtual machine can exit:

"The JLS defines that a Java virtual machine terminates all activity and *exits* when one of two things happens:

- All threads that are not daemon threads terminate.
- Some thread invokes the `exit` method of the class `Runtime` or class `System` and the `exit` method is not forbidden by the security manager."

OPJ interprets "terminates all activity and *exits*" as terminating the computer process executing the OPJ virtual machine, as opposed to destroying the persistent state of the computation. Active threads are retained, with the understanding that the threads will be resumed automatically when the computation is resumed in a new OPJ virtual machine. Although daemon threads do not prevent an exit from occurring, they are retained in the persistent state of the computation and will be resumed.

The termination is *abnormal* if the argument to the `exit` method is non-zero or if the last thread throws an uncaught exception; otherwise, the termination is *normal*. In the case of an abnormal termination, the effect is as if the `Runtime.halt` method had been called. In the case of a normal termination, the effect is as if the `Runtime.suspend` method had been called. Note, therefore, that a call to `exit` with a zero argument will return to the caller if and when the suspended computation is resumed.

Note that in the special case of an OPJ virtual machine exit with no active threads, the static variables of the bootstrap classes act as the sole roots of persistence by the definition of persistence reachability. In this situation, in order to

retain objects created by classes loaded by application classloaders, an object may be registered explicitly as a root of persistence using the `Runtime.roots` set ().

2.5.5 Transactional Behavior

The sequence of suspended computations in a persistent store must conform to the standard ACID transactional properties [12]—namely *atomicity, consistency, isolation* and *durability*—from the perspective of an external observer. For this purpose we define a *VM transaction* as the unit of OPJ virtual machine computation between two suspended computations in a persistent store.

- **Atomic**: a VM transaction's changes to the OPJ virtual machine computational state either succeeds completely or has no effect.
- **Consistent**: A VM transaction is a correct transformation of the OPJ virtual machine computational state with respect to the semantics of the Java programming language.
- **Isolation**: Even though VM transactions may execute concurrently, it appears to each VM transaction, T, that others executed either before T or after T, but not both.
- **Durable**: Once a VM transaction completes successfully, its changes to the OPJ virtual machine computational state must survive system failures.

The OPJ specification does not prescribe how these properties are achieved and several implementations are possible. A straightforward approach is to restrict resumptions from a persistent store to one OPJ virtual machine at a time, using a store locking mechanism. However, it is recommended to allow multiple OPJ virtual machines to resume from a persistent store in read-only mode, but to reject or ignore any attempt to invoke a checkpoint in such an execution.

2.6 HashCodes

The JLS specification of the methods `java.lang.Object.hashCode` and `System.identityHashCode` is clarified for OPJ. It is required that the value returned by these methods be invariant for the lifetime of an object, irrespective of whether the lifetime spans one or more suspended computations.

2.7 The `transient` Modifier

In OPJ, the specification of the `transient` modifier is altered and augmented by additional sub-modifiers, `storage` and `serial`.[1] These sub-modifiers are interpreted by OPJ and by Java object serialization [26], respectively, and are needed because a common idiom is to use the `transient` modifier to mark variables that actually are persistent as needed custom serialization.

The combination `transient serial` means that the variable is interpreted as transient by Java object serialization only. Similarly, the combination `transient storage` means that the variable is interpreted as transient by OPJ only. An occurrence of the `transient` modifier without any sub-modifiers is equivalent to `transient serial storage` in which case the variable is interpreted as transient by both OPJ and Java object serialization.

A variable that is modified with `transient storage` is reset to its default value on a resumption, prior to any active use of the object. Note that the value must not be reset by the checkpoint operation, only by a resumption, as the value is required if the computation continues in the same process after a checkpoint.

2.8 Runtime Events

The OPJ programming model provides the convenient illusion of a continuously executing OPJ virtual machine. Persistence is therefore transparent for all objects and classes created in that virtual machine. However, for program state

1. This proposal was originally suggested by Gemstone Inc. Until this proposal, or an equivalent, is adopted an implementation of OPJ may substitute an API-based alternative as shown in Appendix A.

that is not represented using the Java programming language—for example values in C associated with native methods, or variables holding proxies for such state, including RMI references—persistence cannot be transparent in general. To cater for this and similar situations, OPJ provides a general event mechanism that allows a class to register interest in checkpoints and resumptions, in order to manage external state as transparently as possible. Events are represented by a subclass, `RuntimeEvent`, of the `java.util.EventObject` class and utilize the standard programming model for event handling in the Java platform.

`RuntimeEvent` instances are generated for significant events in the life-cycle of an OPJ virtual machine; these are checkpoint, resume and shutdown. A listener for a `RuntimeEvent` is registered with the `Runtime` class. `RuntimeEvent` instances are delivered when triggered by an OPJ virtual machine life-cycle event. The typical use of a checkpoint listener is to convert a proxy for an external resource, for example, a socket connection, into a value that can be restored by the corresponding resume listener. A shutdown listener is typically used to free up external resources or generate user notifications of the shutdown.

It is only after all checkpoint listeners have returned that the state of the computation is stabilized. Consequently, any side effects of a checkpoint listener are included in the suspended computation. In contrast, the side effects of shutdown listeners are lost. The general expectation is that `RuntimeEvent` listeners will complete in a timely fashion, so as to minimize the latency of a checkpoint operation.

A `RuntimeEvent` is a multicast event and is delivered in an undefined order, possibly concurrently, to registered listeners. Other threads may be active during the delivery phase and appropriate synchronization must be used to cope with other threads attempting to access an object while a listener is handling a checkpoint or resume event. To properly support such synchronization across the stabilization phase of a checkpoint, it is required that the implementation synchronize on the `Runtime` object prior to invoking the listener methods.

In general, programmers are advised to keep `RuntimeEvent` listeners simple to avoid unnecessary interactions between classes. Because the order of event delivery is undefined, large subsystems of classes may require explicit coordination, particularly in the resumption phase.

If an event listener throws an uncaught exception, this will be caught by the event dispatch mechanism, a stack backtrace will be output, and then execution will continue.

Registering an object as an event listener does *not* by itself cause the object to become persistence reachable. A `RuntimeEvent` will be delivered to a registered listener if it is persistence reachable. This is deliberately not “if and only if”, since it would be unreasonable on latency grounds to require full reachability analysis as part of the checkpoint operation.

2.9 Class Evolution

An implementation of this specification will normally provide tools that allow the definition of classes to be altered, either by off-line modifications to the representation of a suspended computation in a persistent store, or by an API that operates on-line. Such mechanisms must conform to the transactional properties described in section 2.5.5.

2.10 Impact on Core Classes

It is a requirement that any core platform classes that deal with native or external state be modified not to fail under the checkpoint/resume model of OPJ. In particular, a minimum requirement is that variables that denote state external to the OPJ virtual machine must be reinitialized on a resumption. An implementation is strongly encouraged to go further and fully support the continuous computation model by, for example, providing modified implementation of the `java.awt` and the `java.rmi` packages.

2.11 Additions to the java.lang Package

We show here only the new classes and the additions to the classes for the OPJ specification. Note that the majority of the additions relate to the event handling mechanisms. The use of these mechanisms is expected to be confined to the platform classes and a very small number of specialized user classes. The vast majority of application classes will

require no changes at all to use with OPJ and the few that do should only need to call the `Runtime.checkpoint` or `Runtime.suspend` methods.

2.11.1 The Class java.lang.RuntimeEvent

The `RuntimeEvent` class is used to denote events that may occur during the execution of an OPJ virtual machine. In keeping with other areas of the Java platform specification, integer codes are used to distinguish the specific events. The `RuntimeEvent` class is itself a subclass of `java.util.EventObject`.

```
public class RuntimeEvent extends java.util.EventObject {
  public int getID();
  public static final int CHECKPOINT = 0;
  public static final int RESUME = 1;
  public static final int SHUTDOWN = 2;
}
```

`public int getID()`

This method returns a value that identifies the kind of event, namely one of the constant values defined above.

2.11.2 The interface RuntimeListener

This is a tagging interface that is the superinterface of all runtime listener interfaces.

```
public interface RuntimeListener extends java.util.EventListener {
}
```

2.11.3 The interface CheckpointListener

This interface must be implemented by a class that wishes to be notified of `CHECKPOINT` events.

```
public interface CheckpointListener extends RuntimeListener {
  void checkpoint(RuntimeEvent event);
}
```

`void checkpoint(RuntimeEvent event)`

This method is invoked as a result of a call to `Runtime.checkpoint`, before the state of the computation is stabilized. The value returned by `event.getID()` will be `CHECKPOINT`. The method allows a class to prepare its state so that it is possible to resume correctly if and when the suspended computation is resumed in a new OPJ virtual machine process. This typically involves generating persistent encoding of any `transient` variables denoting external state. This method is called with the lock on the `Runtime` object held.

2.11.4 The interface ResumeListener

This interface must be implemented by a class that wishes to be notified of `resume` events.

```
public interface ResumeListener extends RuntimeListener {
  void resume(RuntimeEvent event);
}
```

`void resume(RuntimeEvent event)`

The `resume` method is invoked whenever a new OPJ virtual machine resumes a suspended computation from a persistent store. The value returned by `event.getID()` will be `RESUME`. The method provides an opportunity to re initialize any `transient` variables denoting external state that were in effect before the computation was last suspended. This method is called with the lock on the `Runtime` object held.

2.11.5 The interface ShutdownListener

This interface must be implemented by a class that wishes to be notified when the OPJ virtual machine shuts down.

```
public interface ShutdownListener extends RuntimeListener {
  void shutdown(RuntimeEvent event);
}
```

`void shutdown(RuntimeEvent event)`

The `shutdown` method is invoked before normal termination of the OPJ virtual machine, and provides an opportunity to release external resources. Note that any state changes to objects that are caused by this method will not be included in the suspended computation. This method is called with the lock on the `Runtime` object held.

2.11.6 The Class java.lang.Runtime

The following methods are added to the class.

```
public synchronized void addRuntimeListener(RuntimeListener listener);
public synchronized int checkpoint() throws CheckpointException;
public synchronized void suspend() throws CheckpointException;
public synchronized void removeRuntimeListener(RuntimeListener listener);
```

The following field is added to the class.

```
public final java.util.Set roots;
```

In addition, the specification of the `exit` method is altered slightly to take account of checkpoints (see 2.5.2).

`public synchronized int checkpoint() throws CheckpointException`

The checkpoint method first invokes any registered listeners for checkpoint events and then stabilizes the resulting state of the computation. This method does not terminate the currently executing OPJ virtual machine. However, since the computation could later be resumed from the associated suspended computation, the return value is used to distinguish whether this is a return in the currently running OPJ virtual machine process or in a new OPJ virtual machine process. The return values are `CHECKPOINT` and `RESUME`, respectively, as declared in the `RuntimeEvent` class. In addition, a checkpoint may fail with a `CheckpointException`, for example, if there is an error in stabilizing the suspended computation in the persistent store. This is signalled as a runtime exception because a stabilization failure, while expected to be a low-probability occurrence, is a serious error that ought not to be accidentally ignored, for example,

```
try {
  int r = Runtime.getRuntime().checkpoint();
  if (r == RuntimeEvent.RESUME) {
      System.out.println("resuming from checkpoint in a new VM activation");
  } else if (r == RuntimeEvent.CHECKPOINT) {
      System.out.println("continuing from checkpoint in the same VM activation");
  } else { System.out.println("unexpected value returned by checkpoint"); }
 } catch (CheckpointException ex) {
      System.out.println("checkpoint failed");
 }
```

The first time a thread enters the try block it will invoke the `checkpoint` method. If it is unsuccessful, the exception will be caught and the program will take some remedial action, for example, retrying the operation. If the checkpoint is successful, it will return with a value of `RuntimeEvent.CHECKPOINT` and output the message in the second arm of the if statement. Now imagine that sometime later, the OPJ virtual machine exits abnormally. This

leaves the persistent store containing the suspended computation with the invoking thread stopped just before the return from the call to `Runtime.checkpoint`. On resumption, this thread will return with the value `RuntimeEvent.RESUME` and thus output the message from the first arm of the if statement.

`public synchronized void suspend() throws CheckpointException`

First, if there is a security manager, its `checkExit` method is called and this may result in a `SecurityException`.

The method then invokes any registered listeners for checkpoint events and then stabilizes the resulting state of the computation. If the stabilization fails, the method completes abnormally by throwing an instance of `CheckpointException`. Otherwise, all registered listeners for shutdown events are invoked, and then the OPJ virtual machine process terminates.

`public synchronized void addRuntimeListener(RuntimeListener listener)`

This method adds `listener` to the set of registered runtime event listeners. It has no effect if the listener is already contained in the set. This method typically should be called from the constructor for the listener object.

`public synchronized void removeRuntimeListener(RuntimeListener listener)`

This method removes `listener` from the set of registered runtime event listeners. It has no effect if `listener` is not registered.

`public final java.util.Set roots`

This field provides an explicit mechanism for registering objects as roots of persistence. An object `obj` becomes a root of persistence by invoking `roots.add(obj`. This is most useful when a suspended computation contains no active threads. Note that it is not expected that applications will query this set to access persistent objects. Instead it is expected that they will register a `java.lang.Class` object as a root and use static variables of that class as implicit roots. For example:

```
public class MyRoots {
  static { Runtime.roots.add(MyRoots.class); }
  static Object myPersistentRootTable = new HashMap();
 }
```

It is possible to use this mechanism to create arbitrary schemes for finding root objects, for example, maps from strings to objects.

2.11.7 The Class CheckpointException

This exception is thrown if there is an error during an invocation of the `checkpoint` or `suspend` methods.

```
public class CheckpointException extends RuntimeException
```

2.11.8 The Class CheckpointInNativeMethodException

```
public class CheckpointInNativeMethodException extends RuntimeException
```

On return from `checkpoint` or `suspend`, any threads that were executing native method calls at the time of the checkpoint will resume by returning abruptly with this exception from the earliest native call on the stack.

Chapter 3

Discussion and Rationale

This chapter addresses the rationale for our particular design choices in the OPJ specification, which evolved over the a period of several years through the PJama prototype implementations [3], [14].

3.1 Application of the Principles to the Java Programming Language

Certain aspects of the Java programming language specification complicate the application of the principles of orthogonal persistence. The most notable are the provision of concurrency through lightweight threads and the dynamic loading of classes, especially the wide spectrum of binding times for symbolic resolution. In this section we discuss the application of the principles in detail.

3.1.1 Type Orthogonality

The principle of orthogonality has been a recurrent theme throughout programming language history. It is defined as the ability for a given language feature to compose with all other properties of a language, and such a feature is sometimes referred to as being “first-class”. It is generally held that orthogonality yields languages that are powerful and easy to use but which may be difficult or inefficient to implement. In consequence, all the major programming languages in widespread use today fall short of orthogonality, and this contributes to much fine print in the reference manuals detailing the limitations. The Java programming language is no exception to this, although it has less restrictions than many of its contemporaries.

The principle that persistence be orthogonal to type simply requires that any type can persist, without exception. Several persistence mechanisms for the Java platform, for example object serialization, do approach this goal. However, they outlaw the persistence of certain classes, for example `Thread`, or only provide partial support, for example, in the case of `Class`.

Generally, orthogonal persistence cannot be achieved in any language without some degree of support from the language implementation, and the disallowed types typically occur because the required support is lacking. In recognition of this situation, the OPJ specification is written as an extension to the Java platform, rather than as a layered API. We make no apology for this, although we have been criticized many times for taking this position. There is no evidence to suggest that OPJ can be completely implemented as a layered API.[1] It should also be noted that many of the other extensions that have been made to the Java platform since its inception, for example reflection, have required changes to the specification and the platform implementation.

The problematic areas in the Java programing language for OP are 1) the binding of code and data that is mandated by its object-orientation; 2) the support for user-defined class loaders; 3) static variables and 4) the support for concurrency by lightweight threads. We will discuss each of these in turn.

3.1.1.1 Persistence of Class `Class`

It is a generally accepted property of an object-oriented system that the state of an object and the code that implements its behavior are fundamentally intertwined. As a client of an object, one cannot tell whether state is physically represented as instance variables or dynamically generated in response to a method invocation. With this perspective, it is clear that the behavior must persist along with the state; otherwise the public invariants of the object might change each time the object is activated. Or, worse, the code might not be available, rendering the data unusable.

1. One can come close, but only by giving up on orthogonality. The debate hinges on whether that is important in practice.

Despite this argument, there remains a strong desire by many developers to treat code and data as separate entities when it comes to persistence.[1] As an example, Java object serialization, essentially the "official" persistence solution, separates and recombines code and data when serializing and deserializing objects. What this means in practice is that the type information associated with the class is made persistent but the bytecodes, which provide the behavior, are not. One argument for not saving the bytecodes is that it is too expensive to do so, both in time and space. There is some merit to this argument in the case where an application creates many small files of serialized objects that share common classes, as this would result in excessive duplication. Also, since serialization is used for marshalling arguments in Java™ technology-based remote method invocation ("Java RMI"), routinely including the bytecodes would increase communication costs. Another argument is that, without special tools, it is hard to modify the code once it is encapsulated with the data. We do not dispute this fact, but prefer to invest in the appropriate tools to support class evolution. It is revealing that one of the reasons for the demise of serialization as a persistence mechanism is that, even without the bytecodes, the state that is saved is inevitably tied closely to a specific implementation. This renders it unusable with a revised implementation, even with the explicit (but limited) support for class versioning in the serialization specification. The irony is that it is the lack of complete information about a class, namely the classfile, that ultimately prevents a general solution.

Orthogonality demands that OPJ maintain the binding between an instance and its associated `Class` instance for the lifetime of the instance. This provides strong guarantees that the behavior will not change accidently, which we deem to be very important. However, it does require explicit support for class evolution, for which there is only minimal support in the standard Java platform.[2] OPJ finesses the class versioning problem by ensuring that class instance data is never separated from its code.

There are no insurmountable implementation problems with maintaining the binding between an instance and its `Class` in an OPJ persistent store and this has been provided in all the OPJ prototypes. The provision of user-defined classloaders, however, is a notable complication. The full implications of classloaders on the implementation was poorly understood in early versions of the platform, and the subject of several loopholes in the type system. We are not persuaded of the value of classloaders,[3] but the principle of orthogonality demands that we support them. At a minimum, OPJ must retain the information on which classloader was used to load a class. In fact, considerably more meta-data must be also be made persistent in order to correctly preserve the semantics of the type system across checkpoints.

3.1.1.2 Static Variables

The existence of class static variables poses a pragmatic dilemma for the OPJ specification. On the one hand, there are clearly cases where a static variable contains a value that is crucial to the invariant of the class. For example, a class that hands out sequential, unique, integers using a static variable to hold the current value will fail to uphold its contract unless the static variable is made persistent with the class. On the other hand, static variables are often used for values that are inherently transient and should be re-initialized each time the application is resumed. Ironically, the language contains a mechanism to achieve this by declaring such variables `static transient`.[4] However, this is not used by programmers, if only because the standard persistence mechanism, Java object serialization, does not support static variables. This leads to the presumption that static variables are inherently transient and need no qualification. The dilemma for OPJ is that supporting persistent static variables might cause legacy code that does not use the transient modifier to break. This concern led Gemstone/J [9], which otherwise comes very close to providing OPJ, to not support persistent statics. We believe, however, that there are potentially many more classes that use statics to hold long-lived invariants and that this style of usage must be supported. Certainly we have successfully reused many classes in OPJ applications that, having been written without regard to persistence, would fail if static variables were not persistent. In addition, we believe that classes that require statics to be re-initialized on a resumption, in fact, require additional changes to participate correctly in a persistent application. This is discussed in considerable detail in 3.6. These considerations led us to require that persistent static variables be supported in OPJ.

This matter is closely related to the issue of class initialization since static variables are typically given their values by

1. There are many reasons for this, some technical and some sociological. A full analysis is beyond the scope of this rationale.
2. The Java Platform Debugging Interface [24] provides a mechanism to replace a class.
3. Classloaders produce islands of objects whose types are incompatible, even if they are structurally equivalent.
4. At one time this combination was flagged as illegal by the JLS. We argued successfully for it to be permitted in order to support OPJ.

static initialization blocks. Again, many persistence providers assume that classes will be (re)initialized on each execution. This is consistent with the class bytecodes not being stored with data, and being loaded afresh each time. However, it is not consistent with a model of continuous computation, in which a class is initialized (and finalized) precisely once. The OPJ specification therefore adopts this latter model.

3.1.1.3 Persistence of Class `Thread`

The persistence of threads, which supports the persistence of active execution state, is considered by many to be unnecessary and, indeed, somewhat eccentric. Even within our own research group, the need for persistent threads was much debated.

One reason for this view is that, in practice, many applications can be written without support for persistent threads, particularly single-user applications that can enumerate and manage their threads explicitly. Some amount of additional code has to be written to create new threads on resumption, but this is usually manageable in part because the number of threads is typically small. The key requirement in such an application is that there be a single point of control for the checkpoint operation, and that all persistent state be captured on the heap and not in thread stacks.

Problems arise with application systems that are composed of *independent* threads, especially if they are operating over the same object space. In this case, when one thread invokes a checkpoint operation, the other threads will generally not be in a state to comply. Since they may be executing arbitrary code at the time of the checkpoint, it is generally impossible to program for this situation. For example, a thread may be inside a critical section holding locks. If persistent threads are supported, then the thread will resume inside the critical section still holding the lock and therefore proceed correctly. If, however, persistent threads are not supported, there is no easy way to re-establish the invariants, and therefore the state of the objects in the suspended computation may be in an inconsistent state. Application systems containing multiple independent threads occur frequently in server systems that support a large object space. Generally speaking, as the size of the object space increases, so does the requirement for concurrent applications. Since the value of OPJ increases in step with the size and complexity of the object space, lack of support for persistent threads is felt most just when OPJ is most valuable! In these situations, the lack of persistent threads requires the programmer to explicitly denote sections of code where checkpoints must not occur, which violates persistence independence, and requires detailed global knowledge and a global checkpoint coordinator. As we move closer towards a world of concurrently active applications composed of independently developed and complex parts, with 7*24 uptime requirements, support for persistent threads becomes ever more important.

Unfortunately, the implementation of persistent threads is very challenging. Even more so than a `Class` instance, a `Thread` instance is a complicated structured object, the state of which also changes very rapidly. Furthermore, it is often partly implemented by native operating system support that does not provide adequate information on the current state. Experience shows that provision for thread persistence must be designed into the OPJ virtual machine from the beginning, as it is almost impossible to retrofit. To date, the only OPJ virtual machines that support persistent threads are those in which the virtual machine provides the entire thread implementation, and where exact information is available on the contents of thread stacks. These are the JavaInJava virtual machine [27], which is too slow for production use, and an experimental variant of the Spotless virtual machine [17] for the Palm Pilot. It is somewhat ironic that, despite having no durable persistent storage, the Palm Pilot is otherwise a perfect example of the continuous computation application model.

The solution to this problem lies in a VM design that from the outset supports persistence threads. The current generation of Java virtual machines has paid close attention to the exact identification of object references to support garbage collection, but fall some way short of the requirements for persistent threads. Extending this support to complete identification of all objects in the thread state is a relatively small increment that would support orthogonal persistence in the next generation of Java virtual machines.

3.1.2 Persistence by Reachability

The general idea of persistence by reachability is now widely accepted in the object community. The debate has moved on from the principle itself to the details of how to define the roots of persistence and gray areas in the definition of reachability.

Similar to the discussion above about the separation of code and data, there is also a concern about separating tran-

sient objects from persistent objects. Very often, this is related to external state which we cover in detail in section 3.6. Sometimes it results from the programmer "knowing" that objects are "transient", because they are caches of other state from other objects, or some such. Or it comes from a concern that "too much" data will be made persistent, either for latency concerns at a checkpoint, or because disk space is a scarce resource.[1] None of these arguments are fundamental. They have a familiar ring to those advocated by believers in manual rather than automatic storage management, a battle that appears finally to have been won.

In practice, the effect of persistence by reachability is much more pervasive than most programmers realize. Even if the roots of persistence are explicitly specified, references from these roots tend to reach a surprisingly large subset of the reachable objects. Once persistent threads are factored in, the pretence of separation is essentially over, since the language defines threads as the principal roots of reachability.[2]

Generally, if programmers think in a traditional "read-data, compute, write-data" model, they will be led towards attempting to separate transient and persistent objects. In contrast, if they are applying object-orientation to model (simulate) a real-world system, their design will be led towards the persistence of the entire computation state and will only be concerned about the relative lifetimes of objects.

3.1.2.1 The transient Modifier

Nevertheless, particularly when dealing with objects that are proxies for external state, there is a case for a mechanism to denote that a value has no validity or purpose in the persistent state of the computation. One such mechanism in the Java programming language is the ability to use the `transient` modifier to control reachability for persistence purposes. The meaning of `transient` was only loosely defined by the JLS and, unfortunately, has since been given an incompatible interpretation by the Java object serialization system.

In Java object serialization, the `transient` modifier *may* mark a variable that will cut the reachability, but it also may mark a variable that will receive *custom* serialization by the class implementor. Examples of the second usage occur frequently in the standard collection classes, and the idiom is widespread. Interpreting such variables as non-persistent in an orthogonal persistence context results in the loss of critical data. This problem is sufficiently serious that the OPJ prototypes currently ignore the transient modifier entirely, and provide an ad-hoc solution[3] for indication that a field is transient in the persistence sense. This issue is discussed in more detail in [21], [13]. The OPJ specification contains a proposal to remedy this situation that defines separate sub-modifiers, `serial` and `storage`, for serialization and persistence, respectively.

Given that the `transient` keyword already exists in the Java programming language, it is compelling to exploit it to define the concept of persistence reachable. However, we should note that it requires careful programming in the face of checkpoints invoked by concurrent threads, and this is discussed in detail in section 3.6.5.

3.1.2.2 Reachability of class Class

The issue of the reachability of class `Class` instances has been the subject of much debate in the OPJ community, because the Java programming language is unusually complicated in this area. The flexibility given to an implementation by the JLS in the time at which a symbolic reference[4] is resolved and the provision of user-defined classloaders are at the heart of the complexity. Initially, we felt that it was vital that an OPJ system eagerly resolve all symbolic references to classes, on the grounds that a "dangling" symbolic reference ran counter to the spirit of encapsulation provided by an orthogonally persistent system. In fact we went further and transitively resolved *and initialized* all referenced classes at a checkpoint. The JLS was subsequently written to permit eager resolution, but require that initialization be delayed until the first active use of a class, with which we complied.[5] Also, we finally decided to make the OPJ specification simply follow the JLS with respect to the flexibility of lazy or eager resolution. The rationale was

1. Disk storage is currently growing faster than main memory size. Also, it is often not realized that orthogonal persistence requires that persistent stores are garbage collected as for main memory.
2. Static variables in the bootstrap classes are the other roots.
3. This involves calling a method `Transient.mark(String fieldName)` in a static initializer of the class defining `fieldName`.
4. This means a symbolic reference in the classfile, not a `String` in the source code.
5. There were enough pathologies with class initialization occurring implicitly at a checkpoint that we came to regret our initial position.

that, since class names can be generated at run-time as `String`-valued expressions, it is impossible in general to capture all the classes that might be needed by an application. Although this can lead to persistent stores that are incomplete, we note that a given application can always force the loading of a closed world of classes if necessary.

There was also confusion over exactly when a class `Class` instance became unreachable and therefore available for garbage collection and "unloading". This centered around whether the static variables of classes should have some reachability independent from the active threads. This is pertinent to orthogonal persistence because static variables are the obvious roots of persistence for objects that exist independent of computations. Eventually this issue was decided in terms of reachability of the classloader that loaded the class and by defining the bootstrap class loader as always reachable.

Early versions of the OPJ prototype had an explicit `String` to `Object` map to establish roots of persistence. A consequence of this was that a small portion of the code had to be written specially for persistence in order to access the table (for the majority of the code there was no infringement of the third principle).[1] This was particularly problematic when a root needed to be established in a library class, if only because it requires a global naming convention for establishing root names. When implementing a reusable library class, no such convention can be assumed, other than that already established by the language, namely, the fully qualified class name and the name of the root variable. It was this insight that led us to the idea of using `Class` objects as persistent roots.

In the final specification, OPJ defines the notion of *persistence reachable*, which is a simply a slight modification to the standard Java programming language rules for reachability, that takes account of the `transient` modifier. An explicit root table is still available, as one of the bootstrap classes, thus ensuring its reachability, but it is now only needed to retain objects that are not reachable from active threads. Furthermore, the root table is defined simply as a *set* of root objects, rather than as a map. The expectation is that applications will simply place class `Class` objects[2] in the root set and then use static variables in those classes to access persistent objects. One final feature, which falls strictly outside of the OPJ specification, is a convention that an OPJ virtual machine will provide a mechanism, perhaps via a command line argument, that a given class should automatically be added to the root table. Under these circumstances, many programs require no changes at all to exploit orthogonal persistence.

3.2 Persistence Independence

Given the definition of persistence reachable, the vast majority of application source code requires no change to work with either a Java virtual machine or an OPJ virtual machine. This allows re-use of both source and byte code in both directions. Many other persistence mechanisms for the Java platform achieve a degree of persistence independence by post-processing Java classfiles to interpose instructions that manage access to persistent objects. This violates persistence independence in a different and less intrusive way than modifying source code, but has drawbacks in the development and deployment process. In general, the only way to handle the post-processing transparently is to perform it in a user-defined classloader. Unfortunately, this requires the application to be written in a particular way and precludes the use of classes loaded by the bootstrap or system classloader. The virtue is that this scheme can work without modifying the Java virtual machine to support persistence. But we have to be careful how we define "work". The performance overhead of supporting persistence this way can easily be an order of magnitude or more, which can render this approach unusable in practice. However, a recent report [16] suggest that this overhead might be held within acceptable limits.

3.3 Hashcodes

The requirement that object hashcodes remain invariant across checkpoints and resumptions follows directly from the goal of continuous computation. It is the source of some implementation complexity as many Java virtual machines attempt to optimize the run-time space requirements of object hashcodes as, in practice, many objects do not ever have their hashcode method invoked.

It is expected that many OPJ applications will operate in the context of a very large object space, well beyond the

1. It also required the establishment of a separate, globally managed, namespace for the names of roots.
2. Which have the virtue of being unique (singletons). Note that, because of user-defined classloaders, a `String`-valued class name is not necessarily unique.

numerical range of the `int` return type of the `System.identityHashcode` method. This is a concern for the situations where the identity hashcode is used as an approximation for object identity. We considered proposing an additional method `System.longIdentityHashcode` that would provide an adequate range to distinguish objects across a distributed federation of massive object stores, for example, 128 bits. Efficiently supporting such a large identity hashcode on any object would require some sophistication in an OPJ virtual machine, since it would be unreasonable to pre-allocate the space in every object. At present the need for this feature is not sufficiently proven. We also note that the inevitable trend towards 64-bit machines will ease the implementation issues should the need arise in the future.

3.4 Virtual Machine Life-Cycle

The continuous computation model necessarily requires a re-examination of the life-cycle of an OPJ virtual machine. A standard Java virtual machine is limited to two states: active or shutdown. The startup action transitions from shutdown to active and the exits action transitions from active to shutdown. In particular, there is effectively no distinction between the computer process[1] that is executing the virtual machine and the state of the computation itself. When the Java virtual machine exits, the computation dies with it.

The situation is more complex for an OPJ virtual machine. When a computation suspends, the process executing the virtual machine terminates, but the computation continues to exist in its suspended form in the persistent store. The resumption action transitions from the suspended state to the active state. Both the startup action and the resumption action involved the creation of a new process to execute the virtual machine, but the resulting behavior is quite different. Note that we can unify the startup and resumption actions by considering startup as the resumption of the empty computation. This leaves the issue of how the binding between the virtual machine process and the persistent store is established and this is discussed in section 3.4.1.

It is reasonable to ask if there should be a way to exit the virtual machine such that the persistent state of the computation reverts to the initial empty state. The short answer is no. An OPJ virtual machine behaves rather like the “big bang” universe: it starts and then just keeps on going. In addition, there is the distinction between states that contain active threads and those that just contain passive data, which leads to the issue of “injecting” a new thread into an existing state. These issues are discussed in more detail in section 3.4.2.

Finally, the above discussion tacitly assumes that the virtual machine can be treated separately from the computation that it is executing. For contemporary implementation of the Java virtual machine, this assumption turns out to be false, which complicates the specification of the persistent state of the computation. This is discussed in more detail in section 3.4.3.

3.4.1 Binding to the Persistent Store

We choose to associate one persistent store with one OPJ virtual machine activation because it is consistent with providing persistence for the complete computational state. This represents a notable difference from the convention in other object persistence mechanisms for the Java platform, in which it is normal for the application to explicitly open and close “databases”. The rationale for our choice comes from considering the implications of the continuous computation model and of `Thread` and class `Class` persistence, in particular. In the continuous computation model, the complete state of the computation, represented by the complete state of the *all* the “applications” and their associated classes, is required to be made persistent. While it is sometimes possible to identify subsets[2] of the computation that can be stored separately, in the general case it is very difficult.

Another argument for the 1-1 relationship is the issue of class versioning. If one permits the incremental addition of persistent stores under application control at runtime, each store might contain a different version of the same named class. The virtual machine has to be able, at a minimum, to verify that the classes in any store that is being opened are compatible with any that have already been loaded in the virtual machine, and then suppress their (re)loading. Additional complications arise when applications attempt to connect objects that belong to different stores, which can eas-

1. The term process is used loosely here. It could mean “operating system process” but it could also mean a dedicated piece of hardware, where the OPJ virtual machine *was* the operating system.
2. Bounding such subsets is difficult in practice because of persistence by reachability; see 3.1.2.

ily happen in a typical Java application. We do not feel that it is acceptable to simply warn programmers in the fine print of the manual that such code "should be avoided", as is often the case for other persistence mechanisms. We also note that, conceptually, multiple suspended computations in separate persistent stores seem more like multiple, distributed, virtual machines than a single machine. Since distributed programming explicitly follows a federated model in Java RMI, with only limited transparency, we believe that it would be a mistake to introduce another model under the auspices of persistence.

Overall, we believe that the consistency with the JLS that arises from the choice of one persistent store to one virtual machine far outweighs the perceived benefits of the multiple store approach. The 1-1 relationship is also consistent with the goal of a single-level store, which is itself consistent with extending the memory model to encompass stable storage.

3.4.2 OPJ Virtual Machine Exit

The JLS defines the conditions under which the virtual machine can exit, namely, an explicit call, for example to `Runtime.exit`, or implicitly through the termination of all active threads. It is necessary to define the OPJ semantics of this form of exit and there are essentially two choices. We can treat exit as a variant of the OPJ suspend operation, thus preserving the computational state in the persistent store, or we can define exit to destroy all computation state, effectively reverting to the initial empty state before the first startup. The situation is different for explicit and implicit exit, and we discuss these in turn.

3.4.2.1 Explicit Exit

There is some justification for exit to result in the empty state in the first edition of the JLS [10], chapter 20, in which `Runtime.exit` is specified to stop all active threads. The argument that the programmer expects all threads to be stopped, rather than suspended, is compelling for compatibility with standard Java applications. However, from a traditional database perspective, it would be very strange if, when the computation terminated, all the (passive) data also disappeared! Prior to the decision that bootstrap classes could not be garbage collected, the JLS clearly permitted this to occur, since it stated that an object was reachable if it could be accessed in any potential continuing computation from any live thread. So if there are no live threads, nothing is reachable and everything could be garbage collected, resulting in the empty state. We believe that this consequence to be an unacceptable position for OPJ and so choose to treat `Runtime.exit` like suspend.

However, there is a further dimension to exit, namely the notion of normal versus abnormal termination. Abnormal termination is indicated by a zero argument to the `Runtime.exit` method For consistency with implicit exit, we choose to terminate the virtual machine immediately without a checkpoint. This has the slightly strange consequence that a call with a non-zero argument never returns, but that call with a zero argument does, because it is equivalent to a request to suspend.

3.4.2.2 Implicit Exit

Since an explicit call to `Runtime.exit` occurs in an active thread, that thread when suspended acts as a root of persistence. In the case of implicit exit because all threads have terminated, that is not the case. Therefore, a case can still be made that this should result in an empty persistent computational state. Again, however, this is unacceptable because it prevents the existence of passive persistent stores.

To resolve this requires a mechanism for the programmer to identify objects that must remain reachable and therefore persistent. The mechanism we chose is the persistent root table in the `Runtime` class. We can keep this table reachable by definition or by arranging for a hidden system thread to hold a reference to it. We chose the former, leveraging the JLS clarification that classes loaded by the bootstrap loader can never be unloaded. That is, even in the absence of any live threads, we treat this requirement on the bootstrap classes as justification that they are always persistence reachable.

As with explicit exit we must distinguish normal from abnormal termination, that is, when the last thread encounters an unhandled exception. In this case, the invariants of any objects that were being accessed by that thread will be suspect and it would therefore be inappropriate to checkpoint the system in this state. Therefore, in this case it is essential to terminate the virtual machine immediately without a checkpoint.

3.4.2.3 Injecting New Threads

If a persistent store contains no active threads, then there is clearly a need to create a new thread on resumption. Since it must be possible to create a new thread in this case, the question arises whether it should be legal to create or inject an additional thread in the case of a suspended computation that already contains active threads. After all, it will not generally be easy for a user to tell whether a store is active or passive. For this reason we decided to permit new threads to be injected. However, questions arise about the relationship of the injected thread and its associated main class, to the existing threads; for example, what classloader should be used to load the new main class.

The most recent version of the JLS defers the specification of startup to the Java virtual machine specification, and this now offers considerable flexibility to an implementation over how startup actually occurs. The only specified behavior is that the initial class is created using the bootstrap classloader, is linked and initialized, and then has its main method invoked. This class may be user supplied as in JDK™ Version 1.1 or implementation supplied as in JDK™ Version 1.2. In the latter case a user-supplied class is actually created in a new classloader.

For OPJ, the key decision is whether to define thread injection in terms of the bootstrap classloader, as per the initial startup, or leave it to the implementation defined startup system; that is, leave it as implementation undefined. The space of implementation-defined possibilities is quite large. For example, an implementation could choose to create a separate classloader for each injected thread, producing a kind of multi-programming system. At the time of writing our experience in this area is limited, so we feel justified in leaving the choice to the Java virtual machine implementation.

3.4.2.4 Daemon Threads

The existence of daemon threads causes an additional complexity. While it is certainly harmless for a Java virtual machine to exit with daemon threads alive, it is not so obvious what should happen in an OPJ virtual machine. One can argue that if the existence of daemon threads does not prevent termination of the virtual machine, then their contribution to computation state is not worth preserving. Unfortunately, it is not possible to arbitrarily stop the threads, for the same reasons that led to `Thread.stop` being deprecated; namely, that this could leave objects with invalid invariants and therefore in an unusable state in a subsequent resumption. For this reason daemon threads are retained in the persistent state but, for compatibility, do not prevent termination of the virtual machine process.

3.4.3 Distinguishing the Virtual Machine, the Platform and the Application

There are three conceptually different aspects to an executing Java application. The first is the virtual machine itself, which has a precise specification [15]. The second is the layer that includes all the “core” classes, which by now comprise a very large number. Together the virtual machine and the core classes constitute the Java platform. Finally there is the application itself, represented by its additional loaded classes and its computational state.

In current Java platforms, these three aspects are not very clearly separated. Many of the core classes that can be used by an application actually participate in the implementation of the virtual machine. This is especially marked, for example, in the `Thread` class. The transitive closure of classes that are reachable from the `Thread` class turns out to be surprisingly large and reaches into the upper layers of the core classes.

This has the undesirable consequence of producing a dependence between an application and the implementation of the virtual machine. While this is not a problem for a standard Java virtual machine, it can result in the suspended form of an OPJ computation being dependent on a particular implementation of an OPJ virtual machine. This makes it impossible in general to resume a suspended computation with a different implementation and is therefore a fundamental barrier to the portability of suspended OPJ applications. It is essentially for this reason that the OPJ specification does not require that a suspended computation be resumable on a different machine. Even if the format of the persistent store were completely architecture neutral, the dependence between the implementation of the virtual machine and the core class libraries could still prevent portability.

3.5 The Relationship with Transactions

It is common to find that a persistence mechanism is strongly associated with the concept of a transaction [12], defined by the ACID properties, namely atomicity, consistency, isolation and durability. Indeed, our first paper [1]

contained a proposal for a sophisticated extensible transaction model, that has not been implemented in any of our released prototypes. However, only the durability property of a transaction is strictly related to persistence. Furthermore, isolation is already supported in part in the Java programming language by the thread synchronization mechanism, which is an explicit locking mechanism, as opposed to the implicit locking that is normal for transactions. Many aspects of the consistency requirement are also supported directly by the type system and automatic transient memory management of the Java programming language.

Many of the transactional mechanisms of other persistence solutions for the Java platform effectively introduce a separate memory model from that defined by the JLS—for example, the notion of an object in a "hollow" state, one that has an identity but whose instance variables are not yet populated from the external store. Objects can change state simply in response to transaction begin and commit which, arguably, represents a violation of encapsulation.[1] In any event, programmers must effectively understand this additional memory model in order to program correct applications.

Because of these concerns, the OPJ specification does not include an explicit notion of a transaction in the programming model. Indeed to do so would change the Java programming model and require many more significant changes to the JLS. We are continuing to research this area but believe that it is premature to standardize on a specific model. Again, this separates us from other persistence mechanisms for the Java platform, where the use of a transaction API is typically mandatory, usually arising as a direct consequence of allowing a many-to-one relationship between a virtual machine and a persistent store. However, it is important to note that, as specified in section 2.5.5, the behavior of the OPJ virtual machine as a whole maintains transactional properties from the perspective of an external agent.

Our current belief is that even if we added a transaction API, it would not replace orthogonal persistence. Rather it would provide additional facilities like concurrency control and rollback. In fact, the orthogonality principle would also apply to transactions, thus supporting long-lived transactions in a straightforward manner.

3.6 Handling Classes that Capture External State

For OPJ to be a usable specification, some classes, particularly those that capture native or other external state, must be modified to comply with the checkpoint/resume model of OPJ; else they will not function correctly after a resumption. Of particular importance are the AWT and RMI subsystems. In addition, since the Java platform is "open" by virtue of the native method mechanism, OPJ requires a general mechanism for handling state that, while logically forming part of the application, cannot be handled automatically by the OPJ platform.

3.6.1 Background

The provision for `native` methods is widely used to implement input/output classes and graphics classes that have, in previous, closed, orthogonally persistent systems, been defined as language primitives. The open nature of the Java platform presents an immediate problem for orthogonal persistence since a part of the total system computational state will likely be defined using a separate language such as C or assembler code. While it was clear from the beginning that OPJ would have to deal with the core classes,[2] the success and phenomenal evolution of the Java platform, coupled with the poor performance of early virtual machines, has resulted in far more native code than we ever imagined possible. Indeed, at the outset we rather naively suggested outlawing user-level native code in an OPJ application because of the risk that buggy native code might unknowingly corrupt the persistent store.

While we might hope for a reduction over time in the use of native code to address performance issues, the need for special code to manage devices that connect with the external world—for example, network adaptors and graphics devices—will always remain. The fundamental problem is that some state may inherently be beyond our ability to capture in a local checkpoint. To handle this problem, we have explored several mechanisms for allowing user-supplied code to execute on significant events like checkpoint and resumption. The general approach is to allow user-supplied code to register interest in specific runtime events in the OPJ virtual machine: specifically, checkpoint, shutdown and resume. There are several possible variants of the general approach, with different trade-offs, which we discuss in more detail below. It is interesting to note in passing that this issue is a good example of the 90/10 rule.

1. There is an analogy here with the way that `transient` variables can be reinitialized across a checkpoint in OPJ.
2. In a sense this is equivalent to treating these as if they were language primitives as other OP systems have done.

While we expect that relatively few classes will need to use these facilities, they form by far the largest piece of the OPJ API.

3.6.2 Goals for External State Handling

Recall that the overriding goal of OPJ is to provide the illusion of continuous computation in the face of system shutdowns, planned or unplanned. This is achieved by checkpointing the state of the application periodically to stable memory. In order to achieve this, an OPJ virtual machine must be able to trace all reachable aspects of the computation. This is only possible because of the strongly-typed nature of the Java programming language.

3.6.2.1 Representing External State

A class that exploits the native programming interface to access external state typically will maintain one or more variables that act as proxies for the external state. Examples might include an `int` variable that denotes an open file descriptor, acquired through the native interface to the operating system, or a `long` that, in fact, is the address of an entity in the external domain, for example, the address of a C structure. Such values cannot properly be interpreted according to their declared type, and are restricted to being stored, copied and passed as tokens through the native interface.

Since the actual state that is denoted by such proxies cannot be interpreted by the OPJ virtual machine, it cannot be made persistent by the OPJ checkpointing mechanism. Also, since the state of the external domain may well have changed when such a checkpointed computation is resumed, it should be clear that the proxy values themselves may also be meaningless, and will need to be regenerated prior to any use by clients of the class. We refer to such values as *transient proxies for external state* or simply *external state* when the context is clear.

Note that failure to address this issue may compromise integrity constraints at all levels of abstraction, including the underlying runtime system. For example, a program crash may occur due to dereferencing an outdated pointer in external code. We should note that the Java programming language offers no help in indicating the existence of such hidden external pointers to the programmer, unlike, say, Modula-3 [20], which provides an `ADDRESS` type and an `UNSAFE` keyword to act as clues. However, in a Java class definition, any appearance of the `native` keyword is a strong hint that such transient proxies might exist.

3.6.2.2 Scope and Goals

The scope of the OPJ mechanism is restricted to supporting the reconstruction of such transient proxies for external state. We do not directly address the deeper issue of consistency between the internal state and the actual external state denoted by a proxy. Our goals, ordered in importance, are as follows:

- *To provide a simple mechanism that allows a class to reconstruct its external state, such that it appears to be persistent to clients of the class.* In other words, the mechanism should make it appear to the vast majority of code that the suspend/resume never happened.
- *To minimize the latency of the checkpoint operation because, like garbage collection, high-latency can be intrusive to the application.* It is less important to minimize the latency of the resume operation, because we assume that resumptions are much less frequent than checkpoints, since resumptions occur only after a crash or an explicit shutdown of the system. Implicit in this model is that an OPJ system acts somewhat like an operating system. We are not trying hard to support a model where OPJ virtual machines are frequently being created and destroyed as is the case with Java applications in a conventional operating system environment. Nevertheless, we would like an OPJ Virtual Machine to execute the resume operation relatively quickly.
- *To contain the use of the mechanism to those classes that really require it.* This is consistent with the principle of persistence independence. Any interposition mechanism can be misused, often to the detriment of the system as a whole, particularly if the interposition occurs in a critical path of the implementation. In consequence, we deliberately chose to restrict the set of events that can be handled to those that relate directly to the continuous computation model provided by OPJ. For example, with this criteria, one cannot make the case for interposition on the swapping of an object to and from the persistent store, since this can occur at arbitrary moments independent of the checkpoint/resume operations.

We chose to use the standard Java™ technology-based event model as the basis of the design, even though it contains some redundant features, for reasons of familiarity and compatibility.

3.6.3 A Motivating Example

In this section, we introduce a simple example that we will use to illustrate the issues and techniques that we use to manage external state. The example, which is fictitious, is a class that provides a simple input interface to a native file system. For the purposes of illustration the interface is extremely simple; all of the implementation details are hidden in a separate class, and all error handling code is omitted.

First we show the class as it might be written without concern for the continuous computation model provided by OPJ.

```
public class FileInput {
  private int fd;
  public FileInput(String path) {
    fd = NativeFile.open(path);
  }
  public void seek(long newoffset) {
    NativeFile.seek(fd, newoffset);
  }
  public void close() {
    NativeFile.close(fd);
  }
  public int read() {
    return NativeFile.read(fd); // implicitly updates offset
  }
}
```

The `FileInput` class is merely a thin veneer on the `NativeFile` class which provides native methods that contain the real implementation. As is common in such situations, the transient proxy for the external state of the native class is represented as an integer, which typically indexes a table maintained by the native class.[1] Because this table and any related state cannot be made persistent by the OPJ virtual machine, it is desirable to modify the code so that to a client of `FileInput`, the file appears to remain open across a checkpoint/resume operation. We will now modify the code to achieve this, initially using standard language features.

In order to support the illusion of continuity, it is necessary to re-open the file when the computation resumes, and seek to the correct offset in the file. Notice that this requires that we record additional state in the class, namely the `path` variable passed to the constructor and the offset, which is implicit in the above example.[2]

Note first that the `fd` field holds an inherently transient value which we can indicate by the `transient` modifier. The effect of this is primarily for documentation purposes in this instance, because the type of `fd` is `int`. However, were the type some more complex object structure, then marking it `transient` would indicate to the OPJ virtual machine that the object structure did not need to be checkpointed to the persistent store, thus saving time and space.

The remaining issue is to handle the re-opening of the file and the seek to the correct offset. To achieve this, we employ a general technique that leverages the semantics of the `transient` modifier. The OPJ specification states that on a resumption, a field marked as `transient` will be reset to its default value before the first active use of the object containing the field. This guarantee can be used to test for a resumption using the following code pattern:

```
transient boolean constructed;
```

1. It might also denote an address in the native language environment.
2. It will likely be an explicit variable in the `NativeFile` code.

```
...
if (!constructed) {
  ... // construct the value
  constructed = true;
 }
```

The `constructed` variable is called an *access guard* and is a widely applicable technique. Usually it is not necessary to use a separate variable for this purpose, as one of the existing state variables can be used. This is certainly true for object-valued variables as the default value of `null` can never indicate a valid value. For scalar variables, however, the default value of zero may overlap with a valid value, so an explicit guard variable is required. Applying this idiom to the `FileInput` class gives:

```
public class FileInput {
  private transient int fd;
  private transient boolean constructed; // initialization/resumption sets to false
  private String path;
  private long offset;
  private void checkOpen() {
    if (!constructed) {
      fd = NativeFile.open(path);
      isOpen = true;
      NativeFile.seek(fd, offset);
    }
  }
  public FileInput(String path) {
    fd = NativeFile.open(path);
    this.path = path;
  }
  public void seek(long newoffset) {
    checkOpen();
    offset = NativeFile.seek(fd, newoffset); // assume call returns new file offset
  }
  public void close() {
    if (constructed) NativeFile.close(fd);
  }
  public int read() {
    checkOpen();
    int result = NativeFile.read(fd);
    offset++;
    return result;
  }
}
```

The modified and additional code is shown in bold font. The combination of the `transient` modifier and the access guard handles this simple case adequately. This technique has the virtue of supporting the principle of persistence

independence, since it works equally well for transient programs.

However, access guards incur a cost both in performance (slight) and in increased code complexity. In a more complex example there would be a real possibility of errors resulting from a failure to guard every access. Therefore, it would be preferable to guarantee that the value of `fd` was valid before any possible use. To achieve this requires some kind of callback mechanism to indicate that a resumption has occurred, which led us to the event-based model in the OPJ specification.

The access guard pattern is also vulnerable to race conditions in multi-threaded programs. If a checkpoint can occur after checking the guard but before a use of the guarded variable, a subsequent resumption may crash because the invariant established by the guard will not hold due to the transient variable being reset by the resumption.

3.6.4 Using the OPJ Event Handling API

We can use the OPJ API to separate the essential logic of the `FileInput` class from the logic needed to support resumption. First, we move the code to re-open the file and seek to the required offset into the method that handles the `RESUME` event. Second, instead of updating the file offset every time an element is read, we acquire the offset in a method that handles the `CHECKPOINT` event. On the reasonable assumption that checkpoints are less frequent than calls to the `read` method, this also improves performance. Finally, we ensure that the file is closed in a method that responds to the `SHUTDOWN` event.

```
public class FileInput implements CheckpointListener, ResumeListener,
                                  ShutdownListener {
  private transient int fd;
  private String path;
  private long offset;
  public FileStream(String path) {
    this.path = path;
    fd = NativeFile.open(path);
    Runtime.addListener(this);
  }
  public void seek(long newoffset) {
    NativeFile.seek(fd, newoffset);
  }
  public void close() {
    NativeFile.close(fd);
    Runtime.removeListener(this);
  }
  public int read() {
    return NativeFile.read(fd);
  }
}

public void checkpoint(RuntimeEvent ev) {
  offset = NativeFile.tell(fd);
}
public void resume(RuntimeEvent ev) {
```

```
    fd = NativeFile.open(path);
    NativeFile.seek(offset);
  }
  public void shutdown(RuntimeEvent ev) {
    close();
  }
```

Again bold font indicates code modified or added from the original class. Note how the use of the OPJ API has the added virtue of separating the majority of the code for managing resumption from the original logic of the class. Compared to using access guards this improves readability and maintainability. Given that this example is highly simplified for pedagogical reasons, the difference would be even more pronounced in a real class.

3.6.5 Concurrency Issues

In the discussion so far we have ignored the fact that a Java application can contain multiple threads of control. Multi-threading substantially complicates the management of transient external state.

First, consider an ostensibly single threaded application. In this context a simple and appealing model of the checkpoint/resume process is as follows. At a checkpoint, the application thread is stopped, then the checkpoint listeners are invoked serially, and then the computation is stabilized to the persistent store. On a resumption, resume listeners are invoked serially, then the application thread is again made active. In this model, there are no concurrency issues related to transient external state.

However, in practice there are no single threaded Java applications because the Java platform itself uses threads internally. Furthermore, it is difficult to distinguish application threads from system threads, or even application objects from system objects.[1] This problem exists in part because much of the platform is implemented in the Java programming language. Therefore, in general, if all threads are suspended, other than the one running the checkpoint/resume listener, there is a real possibility that listeners will be unable to complete their tasks because a thread required to support the platform will also be suspended.

The situation is worse if the application itself is also multi-threaded and, in general, authors of classes that manage external state must assume that the code might execute in a multi-threaded application. Even if the system threads could be determined and kept running, it is still possible that an application thread might be required to be active for the listener to complete.[2] Another concern is that, in a system consisting of a large number of applications, suspending all application threads maximizes the latency of the checkpoint operation to an external observer such as an interactive user.

These considerations led us to a liberal specification concerning the concurrent activation of checkpoint/resume listeners. They are allowed to execute concurrently with other threads. The rationale is that the existence of multiple application threads already requires that a class managing external state defend itself against a checkpoint being invoked in one thread while another thread is executing a method of the class. No significant additional complications arise from allowing the event delivery to run concurrently with application threads. Both require that classes that implement checkpoint/resume listeners defend themselves against concurrent access from application threads. In the limit, this strategy becomes equivalent to suspending application threads, but has the virtue that it suspends the minimal number of threads, and allows classes to optimize the concurrency control based on their particular needs.

The fundamental requirement is for all access to variables denoting transient external state to take place in a critical region protected by the checkpoint lock that is associated with the `checkpoint` method in the `Runtime` class. For example, consider the `seek` method in the previous version of the `FileInput` example:

```
public void seek(long newoffset) {
  NativeFile.seek(fd, newoffset);
}
```

1. A system thread or object is defined as one that is involved in supporting the implementation of the Java platform itself.
2. This problem is one of the motivations for the thread-based shutdown hooks interface in JDK 1.3.

Absent a critical region, it is possible that another thread might invoke `Runtime.checkpoint` just after the value of `fd` is pushed onto the call stack, but before the call takes place. If that checkpoint is ever resumed, the value of `fd` passed to the call will be that from the previous execution and be meaningless in the new context. The correct code that prevents this is as follows:

```
public void seek(long newoffset) {
   synchronized (Runtime.getRuntime()) {
     NativeFile.seek(fd, newoffset);
     }
}
```

The code sequences that create and destroy transient external state need special care. It is important that the combined operations of state creation/destruction and the registration/deregistration of the associated listener are atomic with respect to checkpoints. For example, imagine that the state is created but a checkpoint occurs before the listener is registered. When that checkpoint is resumed, no resume event will be delivered because the listener was not registered, and this will leave the state incorrectly set to its previous value. Similar considerations apply to state destruction, and these lead to the following code for the `FileInput.close` method:

```
public void close() {
   synchronized (Runtime.getRuntime()) {
      NativeFile.close(fd);
      Runtime.removeListener(this);
   }
 }
```

It is important that all state that pertains to the checkpoint/resume handlers is appropriately protected with the checkpoint lock, whether the state be transient or not. For example, the `offset` field is not marked `transient`, indeed it is vital that it not be, since its value is important to the resume listener. Once the value of `offset` is captured by the checkpoint listener, it is important that no further calls are permitted that might invalidate it, for example by a call to `read`. In fact, in the case where the checkpoint is resumed, calls must be blocked until the resume listener has completed, since it uses the value of `offset`. This is ensured because the checkpoint lock is specified to be held across all three phases of a checkpoint, namely, checkpoint event delivery, stabilization, and resume event delivery.

With these considerations, the final form of the `FileInput` class is as follows:

```
public class FileInput implements CheckpointListener, ResumeListener,
                                  ShutdownListener {
   private transient int fd;
   private String path;
   private long offset;
   public FileStream(String path) {
     this.path = path;
     synchronized (Runtime.getRuntime()) {
       fd = NativeFile.open(path);
       Runtime.addListener(this);
       }
   }
   public void seek(long newoffset) {
     NativeFile.seek(fd, newoffset);
   }
```

```
  public void close() {
    synchronized (Runtime.getRuntime()) {
      NativeFile.close(fd);
      Runtime.removeListener(this);
      }
  }
  public int read() {
    synchronized (Runtime.getRuntime()) {
      return NativeFile.read(fd);
      }
  }
}

public void checkpoint(RuntimeEvent ev) {
  offset = NativeFile.tell(fd);
}
public void resume(RuntimeEvent ev) {
  fd = NativeFile.open(path);
  NativeFile.seek(offset);
}
public void shutdown(RuntimeEvent ev) {
  close();
}
```

Note that there is no need to explicitly synchronize in the checkpoint and resume listeners as the lock is already held by the thread making the callback.

3.6.6 Exception Handling in Event Listeners

If an event listener fails with an uncaught exception, it is not obvious what the correct response by the dispatch code should be. For checkpoint listeners, one could take the position that any failure should result in a `CheckpointFailed` exception being thrown back to the caller of `Runtime.checkpoint()`. Unfortunately, it is not obvious how the caller should handle this case. A similar problem arises with resume listeners. It is possible that an unhandled exception in a resume listener may leave objects in an unusable state with their invariants broken. Also, in contrast to checkpoint failure, there is no control point in the application code to handle resume failure. So the only choices are to ignore the failure or halt the OPJ virtual machine process. The latter choice would then require some external mechanism, such as off-line evolution, to repair the suspended computation such that the resume failure no longer occurs.

The OPJ specification currently takes an optimistic view of checkpoint/resume listener failures requiring them to be caught, a message logged to the standard error stream, and otherwise ignored by the event dispatch code. Note, however, that nothing prevents a listener from performing explicit checks and invoking `Runtime.halt`.

3.6.7 Relationship with Shutdown Hooks

Revision 1.3 of the Java platform contains a mechanism for *shutdown hooks* that is similar to the OPJ event API. It differs in one significant respect. Whereas the OPJ API is based on the event model, shutdown hooks follow a different pattern which is based on registering a thread rather than an event listener. In the shutdown hooks approach, the

runtime system starts the registered threads instead of delivering an event to the listener object in an anonymous thread. Shutdown hooks therefore make the concurrency of hooks and other threads crystal clear. The use of separate threads also clarifies the handling of exceptions.

The shutdown hook proposal was released when we were in the final stages of releasing the first OPJ prototype that met the revised OPJ specification, and there was insufficient time to unify the two proposals. We made a small change to establish a shutdown event, rather than folding this into the suspend event, and adopted the new `Runtime.halt` method instead of an OPJ-specific `abort` method. These changes that could facilitate a future unification. One problem that we face with the shutdown hook proposal is that it *only* addresses shutdown. In particular, there is no way to communicate different events with the proposal as it stands. It would be clumsy and rather resource intensive to require a separate thread for checkpoint, resume and shutdown hooks. Another problem is that a thread is a use-once entity,[1] which would require a new thread to be registered as part of the hook code in order to handle the next occurrence of the event. Unfortunately, this is forbidden in the current shutdown hooks interface.

3.6.8 Problems of Scale, Reachability and Legacy Code

The OPJ event mechanism has the virtue of simplicity, both for the programmer and for the OPJ platform implementor. However, there are issues of scale, reachability and legacy code that suggest that it might not be sufficient.

3.6.8.1 Scale

Very large applications could conceivably contain many objects with registered checkpoint/resume listeners. Large systems could even consist of multiple applications, some of which might not even be active[2] at the time of a checkpoint. The API that we have presented so far will invoke all of the registered listeners regardless of whether they are actually needed by the current application working set, simply by virtue of being registered with the `Runtime` object. For example, imagine that every exported Java RMI object was registered as a listener. A server might well export a very large number of such objects, all of which would be invoked on the occurrence of a runtime event. Another example comes from AWT: potentially very many font objects contain native peers that require reconstruction on a resume. Clearly, in certain situations, such large numbers of runtime listeners might cause unacceptable latency in the checkpoint or resume operations.

3.6.8.2 Reachability

Ideally, if a listener object is unreachable, then it should not be notified of runtime events. However, under the normal language rules, the act of registering a listener with the `Runtime` object would cause it to remain reachable until such time as it is explicitly removed. There are certainly objects for which this is inconvenient and that would prefer to have their reachability determined irrespective of being registered as a listener. To support this, the OPJ specification states that registering as a listener does not by itself make an object persistence reachable. This can be achieved by using weak references or some equivalent technique for the listener data structure that is managed by the `Runtime` object.

Furthermore, the OPJ specification does not require that object reachability be accurately determined at a checkpoint, because of the latency involved in performing the analysis on large systems.[3] The effect of this trade-off is that a listener that has recently become unreachable could still have its listener methods invoked on a checkpoint and subsequent resumption. We are not entirely comfortable with this position but it is not clear from our experiences to date whether this choice can cause serious problems.

3.6.8.3 Legacy Code

The Java platform contains a significant amount of legacy code that deals with external state, legacy in the sense of being unaware of the requirements of the checkpoint/resume model. Important examples again are the AWT and RMI subsystems, that contain programming idioms that do not lend themselves directly to the defensive programming strategy required for checkpoint/resume listeners. For example, direct access to fields of a class precludes the strategy

1. A thread cannot be restarted. Calling the start method has no effect on a dead thread.
2. Therefore, their objects and classes might not have been faulted in to main memory.
3. It effectively requires the marking phase of a full garbage collection.

of using access methods to guard transient external state. Such code requires rewriting to introduce accessor functions. There are also many examples of the following idiom:

```
class Escape {
  private transient ProblemState problemState;
  public ProblemState getProblemState() {
    return problemState;
  }
}
```

In this example, the transient proxy for external state can escape the class by the `getProblemState` method, and could be stored in any other object including a thread's stack. Clearly the access to the escaped object cannot be protected by the mechanisms discussed so far. In fact, this problem has no general solution. It requires an exhaustive analysis of all of the uses of `getProblemState` to discover whether the value is stored and subsequently used to access the object directly. It is tempting to believe that code of the following form is safe:

```
getProblemState().useProblem()
```

Unfortunately, this code is just as vulnerable to race conditions with checkpoints invoked by other threads as direct access to the object. To be safe, acquisition *and* use of the proxy must be bracketed by a critical region protected by the checkpoint lock.

There are also a considerable number of static variables that cache external state and these are often accessed directly, or accessed from other static methods. The listener solution sometimes can be adapted to this case by creating an instance of an anonymous inner class to act as the listener. For example:

```
class StaticExample {
  private static transient Proxy proxy;
  static {
    Runtime.addListener(
      new ResumeListener() {
        public void resume(RuntimeEvent ev) {
          proxy = new Proxy();
        }
      });
  }
}
```

3.6.8.4 Event Ordering

One final issue that is often present in legacy code but can also affect any complex system is the order of event delivery, particularly of resumption events. The Java event model is specific that listeners are notified in an undefined order. This requires that subsystems containing multiple listeners manage their order of resumption explicitly through a controller if necessary. The disadvantage of deviating from the event standard model and specifying an order of delivery, for example—in reverse order of listener registration—is that it removes the freedom to deliver events in parallel, which can help to reduce latency in large systems. It is also not clear that it is any easier to arrange to register the listeners in the correct order than it is to write a controller that explicitly orders the events.

Ordering problems occur most frequently in legacy code that makes extensive use of static variables and methods, because this can result in there being no obvious place to interpose the appropriate resumption code.

3.6.9 Just-in-time Resumption

A general solution to all of the problems discussed in the previous section is provided by *just-in-time* resumption,

with which we have experimented in the context of the AWT subsystem. AWT is particularly prone to the problems discussed in the previous section. Unfortunately, the solution adds complexity to an OPJ implementation and it is not clear whether the benefits outweigh the costs.

The essential ideas behind just-in-time resumption are that an individual object (or class) carries the information about its own resumption and that the associated method is guaranteed to be invoked before the first active use of the object (or class) after a resumption.

A class that requires just-in-time resumption indicates this by implementing the `Resumable` interface. This is a sub-interface of the `ResumeListener` interface, and so inherits the `resume` method, but also acts like a marker[1] on all instances of classes that implement the interface. Following a resumption, whenever such an object is about to incur its first active use, its `resume` method will be invoked. Conceptually this approach is similar to that defined in the JLS for class initialization. The OPJ virtual machine takes responsibility for detecting instances of `Resumable` classes and handles the concurrency control issues directly. This is important because several threads might simultaneously attempt the active use of an instance. Since active use is not limited to method invocation, concurrency control cannot be handled by simply requiring the resume method to be marked as `synchronized`.

Managing the just-in-time resumption of static fields requires a slight modification to this approach. Since it is not possible for the programmer to define a meta-class-specific `resume` method, a solution similar to that for class finalization (`classFinalize`) is required.[2] For our experiments we chose to define a special adaptor class, `ClassResumeAdaptor`, which must be instantiated in a static initializer of the associated class. The implementation of this class uses a private interface to the OPJ virtual machine to mark the class as needing just-in-time resumption.

The algorithm for just-in-time resumption closely parallels that for class initialization defined in section 12.4.2 of the JLS. It guarantees that the meta-class `resume` method is invoked before the instance `resume` method, and that the class hierarchy is handled in top to bottom order, and, most importantly, ensuring that locks are held appropriately to prevent concurrent invocation. In addition, any active use of other objects during the invocation of a resume method will cause the algorithm to be applied to those objects, recursively. This feature is the principal benefit to the programmer, because it automates the ordering of the multiple resume handles that must be written by hand otherwise.

Just-in-time resumption solves the problems discussed in the previous section in the following ways.

- Since an object is not registered with the `Runtime` object, there is no global data structure to cause reachability problems. Furthermore, even if, despite being unreachable, such an object does get included in a checkpoint, since it will never be actively used again, its `resume` method will never be invoked.
- Even if there are very large numbers of such objects, their `resume` methods will not be invoked until they are actively used, thus the latency of resumption is reduced.
- The technique offers a (partial) solution to the ordering problem since the resumption follows the access patterns of a class automatically, by virtue of following the algorithm outlined in the previous section.

3.6.9.1 Implementation Issues with the Just-In-Time Approach

Unfortunately, the just-in-time approach complicates and constrains an OPJ virtual machine implementation, and fails to prevent all resumption problems.

First, note that the specification of class initialization in the JLS explicitly permits cycles between classes and considers them benign, although it contains a warning that a class involved in a cycle may see a default value, rather than the final value assigned by the initialization procedure. The resumption procedure also breaks cycles with similar consequences. Clearly, this can lead to subtle bugs in complex systems, although, empirically, the vast majority of ordering problems are resolved correctly by this algorithm with no further effort on the part of the programmer. This must be contrasted with the effort and ongoing maintenance required to program initialization or resumption with explicit ordering.

A fundamental requirement of the just-in-time resumption model is that the OPJ virtual machine can interpose on first

1. Similar to the `java.io.Serializable` interface.
2. Class finalization is no longer supported.

active use of an object after a resumption. In a system implemented with software object faulting, this is not too onerous, as there is an obvious interposition point, although it puts an extra check in the object access path. Note also that the `resume` method must only be invoked on *first* active use, regardless of how many times an object may be faulted in from the store. Also, hardware support for OPJ, which we might hope for eventually, would also be required to provide an equivalent mechanism such as some kind of trap mechanism.

Just-in-time resume methods that throw unhandled exceptions are problematic, because there is no reasonable place to handle the error. As with class initialization, the thread that invoked the active use of the resumable object will receive the exception, as a subclass of `Error`, but this thread is unlikely to be in a position to handle the exception as it could occur anywhere in the code. This situation should be contrasted with the previously described eager event-based model in which the system code that invokes the resume listener is at least in a position to catch any exceptions, even if it still has some difficulty in deciding on the appropriate response. It is also the case that, unlike the eager event delivery, which happens on system startup, just-in-time delivery may happen at any point in the lifetime of the process.

Overall, it is not at all clear that the benefits of just-in-time resumption are worth the cost in OPJ virtual machine implementation complexity. Therefore, although we have implemented the mechanism in our prototypes, we have left it out of the public specification. Currently, it is used mainly within the AWT subsystem. It remains an open question whether AWT could be restructured to use the simpler model. The other subsystem that suffered some of the issues discussed in section 3.6.8 was Java RMI, which we had modified so that exported objects were exported transparently across a checkpoint/resume. Initially each exported RMI object registered itself as a listener for resume events, thus exposing the issues of scale and reachability, when the number of exported objects was large. However, it did not prove difficult to redesign the system to register a single object for resumption purposes, that manually handled the re-export of the RMI references on an as needed basis. Unfortunately, AWT does not lend itself to such simple solutions.

3.7 Evolution

The Java platform does not currently provide an API for altering the definition of a class once it has been loaded. Since the OPJ specification states that the state of the computation is preserved across a checkpoint and resume operation, special mechanisms are required in order to alter the definition of a class and its associated instances. Ideally these mechanisms should form part of the OPJ specification and permit online evolution. Indeed, it is quite likely that the Java platform will eventually evolve to provide run-time evolution facilities. Partial solutions already exist, for example in the JPDA debugging framework [24]. However, the implementation issues for evolution in the context of orthogonal persistence are much more complex because of the performance and correctness issues involved in transforming the very large numbers of objects that can accumulate in a persistent store.

We have built an experimental system that supports scalable and atomically safe offline evolution in our OPJ prototypes [2], [6], [7], [8]. This allows all possible changes to the class definitions and supports a general mechanism for default and developer-defined transformations of the states of evolving class instances. We validate the set of changes relative to the other classes in use and the set of stored instances before proceeding. We have experimented with integrating recompilations with evolution in an evolution-build tool. Evaluation of these evolution mechanisms and further implementation experiments are needed before a specification of required evolution semantics is included in the OPJ specification.

Acknowledgements

The editors would like to thank all the members of the Forest project at Sun Microsystems Laboratories—Laurent Daynès, Neal Gafter, Brian Lewis, Bernd Mathiske, Michael Van de Vanter—and the Persistence and Distribution group at Glasgow University, past and present—Misha Dmitriev, Craig Hamilton, Tony Printezis, Daniel Schneider, Susan Spence—for their contributions and feedback. Special thanks to Jeanie Treichel for very careful proofreading and editorial suggestions.

Appendix A

The API as a Standard Extension or Third-Party Package

This section describes how the OPJ API would be presented in the form of a standard extension or third-party API. Note, however, that, given the current generation of Java virtual machines, it is not actually possible to implement the API using the mechanisms provided for standard extensions.

As a standard extension, the package name would be `javax.op`, and as a third-party API it would be `org.opj`. As the reference implementation uses the third-party API form, we use that form in this section.

For each class `XYZ` that in the platform-level specification, there would be a parallel class in the `org.opjorg.opj` package, named `OPXYZ`. For example, the class `org.opj.OPRuntime` would contain the `checkpoint` method. The rationale for the `OP` prefix is to allow unqualified import of the class without a name clash with the platform class. For consistency, all classes use the `OP` prefix, even if there would be no clash with the platform classes. Since we have to define a separate `OPRuntime` class, we choose to make its methods static since there can only ever be one instance.

There is one additional class defined in this form, `OPTransient`, which provides an alternative way to denote that a variable has the `transient storage` modifiers.

A.1 The Class org.opj.OPRuntime

```
public class OPRuntime {
  public static synchronized void addRuntimeListener(OPRuntimeListener listener);
  public static synchronized int checkpoint() throws OPCheckpointException;
  public static synchronized int suspend() throws OPCheckpointException;
  public static synchronized void removeRuntimeListener(OPRuntimeListener listener);
  public static final java.util.Set roots;
}
```

A.2 The Class org.opj.OPCheckpointException

```
public class OPCheckpointException extends RuntimeException
```

A.3 The Class org.opj.OPCheckpointInNativeMethodException

```
public class CheckpointInNativeMethodException extends RuntimeException
```

A.4 The Class org.opj.OPRuntimeEvent

```
public class OPRuntimeEvent extends java.util.EventObject {
  public int getID();
  public static final int CHECKPOINT = 0;
  public static final int SHUTDOWN = 1;
  public static final int RESUME = 2;
}
```

A.5 The Class org.opj.OPRuntimeListener

```
public interface OPRuntimeListener extends java.util.EventListener {
  public void createTransientState();
  public void destroyTransientState();
}
```

A.6 The Class org.opj.OPCheckpointListener

```
public interface OPCheckpointListener extends OPRuntimeListener {
  void checkpoint(OPRuntimeEvent event);
}
```

A.7 The Class org.opj.OPShutdownListener

```
public interface OPCheckpointListener extends OPRuntimeListener {
  void shutdown(OPRuntimeEvent event);
}
```

A.8 The Class org.opj.OPResumeListener

```
public interface OPResumeListener extends OPRuntimeListener {
  void resume(OPRuntimeEvent event);
}
```

A.9 The Class org.opj.OPTransient

```
public class OPTransient {
  public void mark(String fieldName);
}
```

This method may only be called in a static initializer of the class with a field called `fieldName`. It tells the OPJ virtual machine that the associated field of the class should be considered as if it were declared with the `transient storage` modifiers. An `IllegalStateException` is thrown if `mark` is called outside of a static initializer or if `fieldName` is not a field of the class being initialized.

Bibliography

[1] M.P. Atkinson, L. Daynès, M.J. Jordan, S. Spence, Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, Proceedings of the 7th International Conference on Persistent Object Systems, Cape May, New Jersey, May 1996.

[2] M.P. Atkinson, M. Dmitriev, C. Hamilton and T. Printezis, Scalable and Recoverable Implementation of Object Evolution for the PJama Platform, Proceedings of the The Ninth International Workshop on Persistent Object Systems, Lillehammer, Norway, September 2000.

[3] M.P. Atkinson and M.J. Jordan, A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform, Technical Report TR-2000-90, Sun Microsystems Laboratories, M/S 29-01, 901 San Antonio Road, Palo Alto, CA 94303, USA, June 2000

[4] M.P. Atkinson and R. Morrison, Orthogonally Persistent Object Systems, VLDB Journal, 4(3), pp319-401, 1995.

[5] G. Clossman, P. Shaw, M. Hapner, J. Klein, R. Pledereder and B. Becker, Java and Relational Databases: SQLJ, ACM SIGMOD Record, 27(2):500, June 1998.

[6] M. Dmitriev, The First Experience of Class Evolution Support in PJama, The Third International Workshop on Persistence and Java, Tiburon, CA, September 1998.

[7] M. Dmitriev, M. P. Atkinson, Evolutionary Data conversion in the PJama Persistent Language. In the “Proceedings of the 1st ECOOP Workshop on Object-Oriented Databases”. In Association with 13th European Conference on Object-Oriented Programming, Lisbon, Portugal, June 14 - 18, 1999.

[8] M. Dmitriev, Safe Class and Data Evolution in Large and Long-Lived Java Applications, Ph.D Thesis, University of Glasgow, Department of Computing Science, Submitted March 2001.

[9] Gemstone Systems Inc., The Gemstone/J iCommerce Platform, http://www.gemstone.com/products/j/main.html, April 2000.

[10] J. Gosling, W.N. Joy, G. Steele, The Java Language Specification, Addison-Wesley, 1996, ISBN 0-201-63451-1.

[11] J. Gosling, W.N. Joy, G. Steele, The Java Language Specification, Second Edition, Addison-Wesley, 2000, ISBN 0-201-31008-2.

[12] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993, ISBN 1-55860-190-2.

[13] M.J. Jordan, M.P. Atkinson, Orthogonal Persistence for Java - A Mid-term Report, The Third International Workshop on Persistence and Java, Tiburon, CA, Morgan Kaufmann, September 1998.

[14] B. Lewis, B. Mathiske and N. Gafter, Architecture of the PEVM: A High-Performance Orthogonally Persistent Persistent Java Virtual Machine, Technical Report TR-2000-93, Sun Microsystems Laboratories, M/S 29-01, 901 San Antonio Road, Palo Alto, CA 94303, USA, October 2000

[15] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1996, ISBN 0-201-63452-X.

[16] A. Marquez, S. Blackburn, G. Mercer and J. Zigman, Implementing Orthogonally Persistent Java, Proceedings of the The Ninth International Workshop on Persistent Object Systems, Lillehammer, Norway, September 2000.

[17] B. Mathiske and D. Schneider, Automatic Persistent Memory Management for the Spotless Java Virtual Machine, Sun Microsystems Laboratories Technical Report SMLI TR-2000-89.

[18] J. Melton and A. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, 1993, ISBN 1-55860-245-3.

[19] P. Monday, J. Carey and M. Dangler, San Francisco Component Framework: An Introduction, Addison-Wesley, ISBN 0-201-61587-8.

[20] G. Nelson (ed), Systems Programming with Modula-3, Prentice Hall, 1991, ISBN 0-13-590464-1.

[21] T. Printezis, M.P. Atkinson and M.J. Jordan, Defining and Handling Transient Fields in PJama, DBPL 99.

[22] Sun Microsystems Inc., Enterprise Java Beans Specification 1.1, http://java.sun.com/products/ejb/docs.html, April 2000.

[23] Sun Microsystems Inc., Java Blend, http://www.sun.com/software/javablend/index.html, April 2000.

[24] Sun Microsystems Inc., Java Platform Debugging Architecture, http://java.sun.com/products/jpda/.

[25] S. White, M. Fisher, R. Cattell, G. Hamilton and M. Hapner, JDBC API Tutorial and Reference, Second Edition, Addison-Wesley, ISBN 0-201-43328-1, 1999.

[26] Sun Microsystems Inc., Java Object Serialization Specification, Revision 1.2, December 1996.

[27] Antero Taivalsaari, Implementing a Java Virtual Machine in the Java Programming Language, Sun Microsystems Laboratories Technical Report, SMLI TR-98-64, 1998.

[28] W3C, Extensible Markup Language (XML) 1.0 Second Edition, http://www.w3.org/TR/2000/REC-xml-20001006, Oct 2000.

About the Authors

Malcolm Atkinson is a full Professor at the University of Glasgow, Scotland since 1984 and has been a Visiting Professor at Sun Microsystems Laboratories. He has sought to improve the context for the construction of large and complex applications via the provision of better integration between programming languages and databases, since working in Neil Wiseman's Rainbow group on CAD and Graphics with Mick Jordan in the early 1970s. He identified the value of orthogonal persistence at VLDB in 1978 and led the team that built the first orthogonally persistent programming language, PS-algol, in 1980. He currently leads research projects in bioinformatics, cultural computing, interpretation of remote-observations of users' actions, computer support for distance learning and persistence. He is a Fellow of the British Computer Society and a Fellow of the Royal Society of Edinburgh. He received his Ph.D from the University of Cambridge, England in 1974.

Mick Jordan is a Senior Staff Engineer at Sun Microsystems Laboratories. His interests include programming languages, programming environments, software configuration management and persistent object systems. He was a member of the team that designed and implemented the Modula-3 programming language. He received his Ph.D from the University of Cambridge, England in 1982.