The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

Pickling State in the Java(tm) System

Roger Riggs, Jim Waldo, and Ann Wollrath
Sun Microsystems, Inc.

Pickling State in the Java™ System

Roger Riggs, Jim Waldo, Ann Wollrath, Krishna Bharat
JavaSoft
{Roger.Riggs, Jim.Waldo, Ann.Wollrath}@Sun.COM, kb@cc.gatech.edu

Abstract

The Java™[1] system (hereafter referred to simply as "Java") inherently supports the transmission of stateless computation in the form of object classes. In this paper we address the related task of capturing the state of a Java object in a serialized form for the purposes of transmission or storage, to be used later in reconstituting an equivalent object. This is accomplished by a mechanism known as *pickling*[2][3][5].

Pickling is the process of creating a serialized representation of objects. Pickling defines the serialized form to include meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated. Unpickling is the complementary process of recreating objects from the serialized representation.

Pickling and unpickling extract from the Java Virtual machine, at runtime, any meta information needed to pickle the fields of objects. Class specific methods are only required to customize the pickling process.

1 Introduction

The Java™[1][4] system supports the transmission of stateless computation in the form of object classes. These can be dynamically loaded into a Java Virtual Machine, for immediate linking and execution. This enables a computation to be "cold-started" at new hosts, always starting with the same, initial state and at the same point in the computation. However, if we need to resume the computation with the state it had prior to transmission, or need to communicate state between cooperating processes, we would need to transmit objects as well.

Pickling is the process of creating a serialized representation of objects. Pickling defines the serialized form to include the meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated.

Unpickling is the complementary process of recreating objects from the serialized representation. The meta information needed for pickling and unpickling is extracted directly from the Java Virtual machine at runtime. Class specific methods are only required to customize the pickling process.

There are many applications for pickling, including:

- Checkpointing of application state; support for persistent objects.
- Marshalling of objects as arguments to and returns from remote-method invocations.
- Serializing objects or object graphs so that they may be stored and retrieved as blobs in databases.

Goals for Pickling

We identify the following goals for pickling:

- Build a simple yet extensible mechanism for serializing and deserializing Java objects.
- Maintain the Java language's object semantics and extend the type and safety properties guaranteed by the Java language and type system to the serialized form.
- Be extensible to support, for example, marshalling and unmarshaling needed for remote object systems.
- Provide simple persistence of Java objects and be extensible to allow more sophisticated Java object storage.

In this paper we describe pickling in a Java system. Section 2 describes the Java Object Model. Section 3 describes the implementation. Section 4 describes type fingerprinting and Section 5 the structure of pickle streams. Section 6 looks at the integrity of sensitive data. Section 7 describes future work.

1. Java and other Java-based names and logos are trademarks of Sun Microsystems, Inc., and refer to Sun's family of Java-branded products and services.

2 The Java Object Model

The pickling of Java objects is based on the availability at runtime of descriptions of Java objects. The Java Virtual Machine retains descriptions of Java object classes for its own use in loading and verifying classes. Pickling uses this information about objects to save and restore the state of objects.

The Java language is described in full elsewhere[4] so only those aspects relevant to saving an object's state will be described here. The Java language is a strongly typed object-oriented language with a syntax similar to C.

Java classes inherit implementations from at most one other class. All classes extend the base class java.lang.Object. Base classes are extended to define subclasses and may add methods and fields.

The Java language defines an interface class as an abstract class declaration that has no implementation. It contains only methods and constants. Java classes may implement multiple interfaces. To implement an interface the class must implement all of the methods specified by the interface. Pickling is only concerned with an object's state so the presence or absence of interfaces only contributes to the identification of the class, not it storage.

Each class may define zero or more methods and fields. Methods are procedures that apply to the class. There are two kinds of methods, those that operate on a specific object and class methods that do not operate on an instance when they are invoked.

The fields of a class hold the state of the class. Each field is strongly typed as either a reference to an object or to one of the built-in primitives types including integer, floating-point, character, and booleans. In the Java system strings and arrays are object types and have support in the language. Fields can be of two kinds, class and instance. Class fields are shared among all instances, instance fields are stored for each object.

Access to the fields of an object is specified as public, package, protected, or private. Public fields are accessible via any reference to the object. By default, the fields of an object are accessible to any object within the package of the referenced object. Protected fields are accessible to subclasses of the class that define them as well as to other classes in the same package. Private fields are only accessible to methods of the object.

The only mechanism to manipulate objects is by object references which are strongly typed. All non-primitive fields of an object refer explicitly to some object class, so the object is known to implement that classes behavior. A reference to an object may be null, signifying that it does not refer to any object.

When a class is extended the subclass is given access to the superclass's public, package, and protected fields, but not the private fields. Subclasses may use all of the accessible fields of their superclasses.

The complete state of an object is held in its class and instance fields and in the class and instance fields of all of its superclasses. Since the class fields are shared among all instances, difficulties arise in transporting and restoring their contents. It is the state of the instance fields that must be written and read from a stream in order to reconstruct the complete object.

3 Simplified Pickling

We present a simple and flexible approach to picklingJava objects. The approach extends the support provided by the Java system for primitive data types. The DataOutput and DataInput interfaces define methods for writing and reading integer types, floating point types, booleans, characters, strings, and arrays of bytes. The DataOutputStream and DataInputStream classes implement these interfaces and perform the encoding and serialization of the primitive types.

For pickling, the ObjectOutput and ObjectInput interfaces extend DataOutput and DataInput interfaces respectively to include writeObject and readObject methods. The classes ObjectOutputStream and ObjectInputStream extend DataOutputStream and DataInputStream respectively to implement pickling of all object types. Special handling is required for Strings, arrays, and Class objects.

An example use of pickling and unpickling is shown in Figure 1.

```
// Pickle today's date to a file.
FileOutputStream os =
                new FileOutputStream("tmp");
ObjectOutputStream p = new ObjectOutputStream(os);

p.writeObject("Today");
p.writeObject(new Date());
p.flush();

// Unpickle a string and date from a file.
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream p = new ObjectInputStream(in);

String today = (String)p.readObject();
Date date = (Date)p.readObject();
```

Figure 1: Pickling and Unpickling Example

The application sets up an output stream and writes a sequence of objects or primitives types. Pickling serializes the objects and objects reachable from them. The objects are written to a stream along with type information so that they can be reinstantiated as equivalent objects by the complementary, *unpickling* mechanism. Shared references within the set of objects are preserved and complex data-graphs are restored with the same structure.

Pickling Interfaces

Pickling utilizes three interfaces:

- The client interface drives the pickling process. It is called with individual objects or primitives that are to be pickled.
- The specials interface allows methods of the object to implement the serialization and deserialization for its own fields.
- The subclass interface allows pickling to be extended to allow additional information about classes and objects to be written to and read from the stream.

Pickling Client Interface

The client interface consists of ObjectOutput and ObjectInput interfaces which are extensions of the basic DataOutput and DataInput interfaces. These pickling interfaces are shown in Figure 2.

```
interface ObjectOutputextends java.io.DataOutput
{
  // Write an object, array or String to the stream
  public void writeObject(Object o)
        throws IOException;
}

interface ObjectInputextends java.io.DataInput
{
  // Reads an object from the stream.
  public Object readObject()
        throws IOException;
}
```

Figure 2: Output and Input Interfaces

Pickling Process

To pickle an object, an ObjectOutputStream is created with an OutputStream to which the serialized form is written. Then, for primitives, the methods of the class DataOutputStream, such as writeInt or writeUTF, can be used. For objects, the writeObject method of the class ObjectOutputStream is used.

The writeObject method pickles the specified object and traverses its references to other objects in the graph recursively to create a complete serialized representation of the graph. The pickle of each object consists of the pickle of the class of the object followed by the object's fields.

Within a pickle stream the first reference to any object results in the object being serialized and the assignment of a handle. Subsequent references to that object record only the handle. Using object handles preserves sharing and circular references that occur naturally in object graphs. Repeat references to an object use only the handle allowing a very compact representation. Each field of the object is written appropriately depending on its type. Fields declared static or transient are not pickled.

The state of the object is then saved class by class from the base class through the most derived class. For each class the special methods are called, if they are defined by the class, or uses the default mechanism to pickle the fields of the object.

The client interfaces to write objects are implemented by the ObjectOutputStream class shown in Figure 3.

```
public class ObjectOutputStream
  extends java.io.DataOutputStream
  implements ObjectOutput
{
  // Creates a new context for an output stream.
  public ObjectOutputStream(OutputStream out)
        throws IOException {...}

  // Write an object, array or String to the stream
  public final void writeObject(Object o)
        throws IOException {...}

  // Called for each class written to the stream.
  protected void annotateClass(Class classs)
        throws IOException {...}

  // Called for each object written to the stream
  protected Object replaceObject(Object obj)
        throws IOException {...}
}
```

Figure 3: ObjectOutputStream Interface

Unpickling Process

To retrieve objects an instance of ObjectInputStream is primed with the stream containing the serialized representation. Primitives and objects are read from the stream in the same order as they were written. The readObject method operates recursively, retracing the sequence generated by writeObject, and following references in a corresponding manner. It creates a new instance for every object in the pickle and restores the object with inter-object references preserved.

When a new object is to be read from the stream, the ObjectInputStream reads the class of the object, creates a new instance of it and adds a mapping from the handle in the pickle to the new object. If the same handle is encountered later in the pickle, it returns a reference to the corresponding unpickled object.

The state of the object is then restored class by class from the base class through the most derived class. For each class the special methods are called, if they are defined by the class, or uses the default mechanism to unpickle the fields of the object.

The client interfaces to read objects are implemented by the ObjectInputStream class shown in Figure 4.

Pruning Object Graphs

There are two mechanisms for pruning the graph of objects to be pickled. In the first, the transient keyword is used to mark fields to indicate that they are not part of the persistent state of an object. The default pickling mechanism will not use transient fields.

The second form uses the specials mechanism, in which the programmer explicitly saves only that part of the object's state that is important. This usage is quite a bit more flexible and gives the programmer direct control over what is saved. The programmer may conditionally prune the graph by writing only some of the references to other objects. During unpickling the equivalent state can be recreated.

```
public class ObjectInputStream
  extends java.io.DataInputStream
  implements ObjectInput
{
  // Creates a new context for a input stream.
  public ObjectInputStream(InputStream in)
      throws IOException {...}

  // Reads an object from the stream.
  // The result should be typecast as desired
  public final Object readObject()
      throws IOException {...}

  // Locate the local class for the named class.
  protected Class resolveClass(String classname)
      throws IOException, ClassNotFoundException
      {...}

  // Called when an object has been unpickled.
  protected Object resolveObject(Object obj)
      throws IOException {...}
}
```

Figure 4: ObjectInputStream Interface

Class Specific Pickling

Special methods can be used to override the default pickling/unpickling mechanism on a per-class basis. These special methods are needed in cases where the class itself knows best how to save the persistent state or when the default mechanism cannot or does not save the needed state. For example:

- A class that defines a large data structure having a more compact representation would be a good candidate for writing special pickling methods.
- In the case of a native implementation with native state, the default pickling mechanism cannot access the state and appropriate specials would be needed to save the object's state.
- If the state of a class is a function of the local operating environment, then it should pickle itself with the information that will allow the unpickling to reestablish that state relative to the new local environment.

The specials interface is defined by three methods with the signatures shown in Figure 5.

```
// Write this classes fields to the stream.
private void writeObject(ObjectOutputStream out)
      throws IOException;

// Read this classes fields from the stream.
private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException;

// Called when unpickling is aborted.
private void readObjectCleanup(ObjectInputStream
                               in)
      throws Exception;
```

Figure 5: Special Method Signatures

The special methods are private and can be called only by the pickling/unpickling run-time. If this were not so, it would be possible to call the writeObject or readObject methods to examine or modify the private state of objects. This would be a serious threat to the integrity of objects.

A class takes over pickling for itself by implementing both the writeObject and readObject methods. An exception is raised by the pickle stream if only one is implemented. Having only one of these methods would not make sense because the default mechanism cannot read data written by the special method.

The readObjectCleanup method is useful if a readObject method has taken some action that would need to be reversed if the unpickle of the graph fails. Usually when an unpickle fails, none of the objects are referenced (except by the pickle stream) and these will be garbage collected when the stream itself is reclaimed. However, if a readObject method has created references in the environment to an object in the incomplete graph, those references must be removed. A readObjectCleanup method is needed in this case to remove the references to the object so that it may be garbage collected.

Specials have the following call semantics:

- Specials are invoked at each level in the inheritance hierarchy starting from the base class and proceeding to the most derived class. If no special is defined for a class, the default mechanism is used.
- The special for a class is only responsible for pickling and unpickling its own fields. It cannot invoke specials in other classes and it cannot prevent other classes from being pickled.
- As with constructors, a readObject method can assume the current object has the type of its declared supertype. It should not, however, rely on the behavior of dispatched methods that might access the fields of its (yet to be unpickled) subclasses. This ordering allows the readObject method, if necessary, to override the actions of its superclasses with respect to public and protected fields.
- If a special raises an exception, the top-level operation is aborted with an exception. Typically this is done by classes that do not wish to be pickled.

Abort Processing during Pickling/Unpickling

The pickling and unpickling process involves many objects and interactions with the underlying system. Since many exceptional conditions may occur, some discipline is needed to handle them and define the state of the stream. The two usual cases are exceptions caused by I/O errors on the target/source stream or exceptions raised by pickle specials to prevent themselves from being pickled. These exceptions are caught by the pickle stream and mark the stream as aborted. Any subsequent attempt to write to the stream will fail by throwing an exception. Pickle special methods cannot override this behavior since the stream is marked aborted on receipt of the first exception and the stream will check this condition after every special terminates and rethrow the exception.

A pickle stream that has had an exception raised is not left in a usable state since an unknown amount of the object and the referenced graph has been pickled or unpickled. Due to the recursive and nested structure of the pickle no statement can be made about what data is in the pickle except that it is incomplete and unusable. It is up to the caller to determine how to continue.

Extending Pickling via the Subclass Interface

ObjectOutputStream and ObjectInputStream can be extended to allow per class and per object data to be pickled.

As every object is pickled, its class must be identified in the stream. This is done by pickling the class itself, using its name and fingerprint. When a class object is pickled, it may be desirable to pickle additional information for the class. For example, if a class was loaded from the network, saving the network address of the code in the stream will allow the class to be loaded on demand during unpickling. The annotateClass method is called after the class name and fingerprint have been pickled. Within this method additional information can be written to the stream.

During unpickling, the class name and fingerprint are retrieved from the stream and the resolveClass method is invoked. The resolveClass method needs to find the named class and return it. It should read any additional information written by the corresponding annotateClass method from the stream and load the class as needed. By default resolveClass just invokes Class.forName. The fingerprint, described in more detail later, is needed to confirm the semantic and structural equivalence of the class used to pickle the object as compared to the locally available class.

Another need that arises is to be able to substitute one object for another during pickling. To make this substitution possible ObjectOutputStream supports a protected replaceObject method. A subclass can define this method to substitute an alternate object. The replaceObject method is called once for each object; it may return either a substitute object or the original. Care must be taken to replace the object with a compatible object or later unpickling will be aborted with a ClassCastException when the incompatible object is assigned to a field or array element. Null may be returned by replaceObject but should be used carefully since setting references to null in arbitrary classes may cause unexpected exceptions in methods that act on the reconstructed graph.

Whenever possible, the writeObject and readObject methods should be used to pickle objects in an appropriate form. However, in cases where the objects implementation is not available, subclasses of the pickling implementations can be created to substitute objects to do a dynamic substitution. For example, objects that are instances of FontData could be replaced by the writeObject method with instances of a FontName class. The corresponding resolveObject method could replace instances of FontName with the corresponding local FontData class. Object substitution during pickling and unpickling can be a very powerful tool for dynamic binding of objects that are part of the environment.

Default Pickling of Objects

The pickling of objects is driven by the class of the object. Strings, Arrays, and Class objects are handled with specific mechanisms, described below, either because they have specific representations or are of special significance to pickling. For all other object types the default pickling mechanism is used.

The pickle stream implementations interpret the class information dynamically to locate and access the fields of an object. Private native methods access the class meta information and fields as follows:

- The fields of the object, except for static and transient fields, are put in canonical order (sorted) so as to be insensitive to reordering of declarations.
- The sorted fields are then written to or read from the stream dispatched by their signatures.
- Fields of primitive types invoke the corresponding DataOutput and DataInput methods.
- Arrays, Strings, and objects invoke the writeObject method on ObjectOutputStream, effectively recursing.
- During unpickling, new objects are created and assignments to each field are type checked.

Pickling and Unpickling of Class Objects

The integrity of the type system is maintained by treating objects of type Class specially and putting information in the pickle to correctly match the type of the object in the pickle with the class available during unpickling. This type matching is done using *secure fingerprinting*. Whenever a Class object is pickled, only its name and secure fingerprint are written to the stream. The details of the fingerprint are described in the section on FingerPrinting Java types.

Correct identification of classes is crucial since the structure of the stream is derived from the definition of the class. Any change of names or types of fields in the class would cause the stream to be misinterpreted. Making the streams self describing would greatly increase the overhead.

ObjectInputStream verifies that the structure implicit in the stream is the same as that defined by the currently available class. This allows primitive fields to be read without additional type checks. For fields that contain references, every assignment of an unpickled object is type checked against the type of the field and for each assignment to an array element.

Pickling of Arrays and Strings

Strings and arrays are treated specially in pickles in order to deal with their specific representations. Both are objects so reference sharing must also be preserved.

Pickling of array objects writes the array signature, length and then iterates over the contents of the array to pickle each element according to its type.

Reading arrays is complementary to writing. First the array signature and length are read, and the first level index is created. Each of the elements is then read and assigned. If the elements are arrays the process recurses naturally. For arrays of primitive types, the functions of DataInputStream are used to read the elements. For object types, Strings, sub-arrays and class objects, readObject is used. This design maintains any potential reference sharing to arrays and subarrays.

Strings are pickled in their Universal Transfer Format (UTF) form but require special handling since they have a native implementation.

Modal Pickling

In practice there are many different situations in which pickling is required, and a different scheme may be required in each. Although there is just a single set of specials for each class, modal pickling can be supported by having the specials switch on the type of the stream which invokes them. Whenever modal-pickling is necessary, appropriate subclasses of the pickling streams would be defined. This allows for forward compatibility, as in any subtyping scheme.

An example of modal pickling is its use in a remote method invocation mechanism[8]. Remote objects are defined by Java interfaces. References to remote objects are declared using these remote object interfaces, not the implementations that support the interfaces. In remote method invocation two types of objects need special handling, the surrogate for a remote object and the remote object itself. Pickling of the surrogate is handled using a writeObject special to preserve the remote reference information. However, a reference to the remote object implementation should not be pickled as itself since that would include all of its implementation state. Instead, it should be pickled as a surrogate object with the matching interfaces and the necessary remote reference information. The marshaling subclass of the pickle output stream is responsible for detecting the implementation object and finding or creating the appropriate surrogate. Both the surrogate and the remote object implementation support the remote object interfaces, and as such both are type compatible with the field and array elements of the defined remote object interfaces.

4 Fingerprinting Java Types

A fingerprint is a concise representation of a piece of data, typically created by a hashing scheme that satisfies the following:

- Low collision: Two pieces of data will hash to the same pattern with extremely low probability (low enough to be ignored).
- Rehashable: Hashes themselves can be input to the hash algorithm.
- Fixed length: The representation is of fixed length.

Java types are fingerprinted by computing a shallow fingerprint for the class and each of its superclasses up to java.lang.Object. The shallow fingerprint consists of:

- The class name encoded in Universal Transfer Format (UTF).
- The class access flags as an integer, (such as public, interface,...).
- The sorted list of interfaces supported by the object encoded as UTF.
- The sorted list of fields, including field name, signature and access modes (such as public, static, protected, private, transient...).
- The sorted list of methods, including method name, signature and access modes (such as public, private, static,...).

The shallow fingerprints are computed using the NIST Secure Hash Algorithm (SHA)[7]. The fingerprint is the rehash of the concatenation of the hashes for each class up the supertype chain. The result is the secure hash of the class. The hash will differ if any changes are made to any of the classes.

This "shallow" fingerprinting scheme provides more assurance than merely hashing the name of the type. Types need to be equivalent in structure as well as name. However, the cost associated with a scheme that hashes the entire hierarchy is not justified, given that the rest of the Java system environment uses name equivalence, and that all classes are verified for consistency as they are loaded by the class loader.

The shallow fingerprinting scheme provides the *same* assurances as a deep fingerprinting scheme if the fingerprints of all referenced object types are matched as well (transitively). Classes are type matched only when an object of that type is pickled/unpickled. The type matching explicitly includes the signatures of all of the supertypes. This insures the structural equivalence of the object and corresponding sequence of bytes in the stream needed to unpickle the object. Other classes this object depends upon will be type matched when the first object of that class appears in the pickle. Objects are polymorphic so the actual types must be represented in the pickle as well as the declared type. The declared types are included in the fingerprint as one of the supertypes of the actual type.

Pickling the `null` reference is interesting because it carries no type information. However this does not present a problem for pickling. The usual interclass dependency checking performed by the class verifier will validate the classes effectively so that they may operate correctly.

Secure Hash Algorithm (SHA)

The National Institute of Standards and Technology (NIST) designed the Secure Hash Algorithm (SHA) for use in the Secure Hash Standard (SHS), to support the Digital Signature Algorithm (DSA). SHA produces secure 160 bit hash values from data of arbitrary length. SHA provides all the guarantees mentioned above. SHA is described in detail in [7]. We use SHA Version 3.

Java Type Specifiers

The strings and values that are input to the hash algorithm are all defined by the formats and constants defined by the Java Virtual Machine Specification[5]. Class names, method names, method signatures, field names and signatures are strings. Method and field signatures use the same encoding as defined by the type system in the Java Virtual Machine. The access flags for the classes, methods, and fields use only the significant bits as defined in the Java Virtual Machine specification.

5 Structure of Pickles

A *pickle* is a representation of the state of a sequence of values (objects, arrays, Strings, Classes, or primitives) corresponding to a sequence of writes. It begins with a magic number of 16 bits and a version number of 16 bits representing the version of the pickling run-time which produced the pickle. The unpickling run-time should ensure that it can process the pickle before it proceeds.

Primitive types are written to the pickle without any preamble or type-checking information by the methods of java.io.DataOutputStream, a core class that encodes primitive types into a stream in a portable fashion. Those methods are also used to write out the fields and values of objects. Reads correspondingly make direct use of the methods in the standard java.io.DataInputStream class. Since such types have a fixed format which is common to all implementations of the Java System, there is no reason to add type checking information, which helps keep pickles small. Since these are not reference types, the caller is required to know the type of the value before reading it for the purposes of assignment. Consequently a preamble containing type information is not needed.

Handles are assigned to objects written to the stream. They are assigned in ascending order using 32 bits and have a base offset value to make them more unique in the stream. However, these handles do not appear in the stream except when a reference is made to a previously pickled object. The writeObject and readObject methods are assumed to be synchronous in assigning handles. This saves space in the pickles, especially in the cases where there are no back references (as occurs in argument marshaling).

Markers are inserted into the stream to validate the alignment of the stream. Markers are 16 bit counters that can wrap around as necessary. Markers are put into the stream at the end of each object, Array, and Class. Markers are also inserted following each call to a special writeObject method. This improves the probability that errors in class specific writeObject and readObject combinations will be detected as soon as possible.

The following description of the pickle format is not rigorous but will provide a flavor for the layout of objects in the pickle. Objects, arrays, strings, and classes that are being written to the pickle for the first time are formatted as follows:

```
<OBJECT object>:= <CODE for OBJECT>
  <Reference to Class of o>
  <Implicit assignment of handle for o>
  <State of the object's base class>
  [Marker if state was not saved by default]
  ...
  <State of the object's most derived class>
  [Marker]

<String object>:= <CODE for String>
  <Implicit assignment of handle for object>
  <UTF of String>

<Array object>:= <CODE for ARRAY>
  <UTF of classname of array>
  <Implicit assignment of handle for object>
  <Length of first index of array (32 bits)>
  <length> objects
  [Marker]

<Class object>:= <CODE for CLASS>
  <Classname in UTF format>
  <20 byte signature of class>
  <Implicit assignment of handle for object>
  <Call to protected annotateClass method>
  [Marker]
```

Other references to objects are encoded as either NULL or as reference to a previously pickled object. The encoding is:

```
<Object null>:= <CODE for NULL>

<Object object>:= <CODE for REFERENCE>
  <Handle as previously assigned to the object>
```

6 Integrity Validations

Since the source of code commonly present in the Java runtime environment is unknown and it may initiate or be involved in pickling, the pickling mechanism needs to be made as tamper proof as possible.

In the attempt to make the pickling mechanism robust the Java language's inherent safety features have helped maintain the integrity of classes.

- The pickling classes provide read and write methods which are declared final and hence can be trusted. This allows the programmer of a class to be sure that the fact that ObjectOutputStream and ObjectInputStream can be subclassed does not mean the specials will behave differently in different circumstances.
- Similarly the programmer of a class can be sure that subclasses of the class will not be able to prevent their specials from being called when an instance is being pickled/unpickled. This allows the special to raise an exception if it wishes, to enforce the class's right not to be pickled.
- The pickling run-time cannot be replaced with another implementation. This is prevented by the class loader and security manager. For the same reason, the native code which provides internal access to objects is only accessible privately to ObjectOutputStream and ObjectInputStream classes and cannot be accessed otherwise.
- It is not possible to corrupt objects using the readObject method, i.e. it is not possible to get the unpickling run-time to reinitialize a pre-existing object from a pickle-stream. This is because the special methods are private and ObjectInputStream only calls special methods on objects it has created.

Remaining Concerns about Data Integrity

Not all concerns have been eliminated however:

- Pickles have a publicly known format and can be tampered with. Just as sensitive fields of the object must be marked private, those same fields must not be pickled carelessly. Marking them transient is an easy mechanism to keep them out of the pickle. If sensitive data must be pickled there is no solution short of encryption with a shared key that will protect the information.

- Fingerprinting compares classes for both structural and name equivalence. This does not guarantee that the code implementing the class does what it is expected to. Support for identifying and authenticating classes is a basic requirement and is outside the scope of pickling.
- It is possible to create an object that could never exist, by unpickling it from a concocted pickle. Since all classes in the hierarchy need to initialize themselves, specials need to be satisfied with the data they read, and marker boundaries need to be respected, this is not an easy thing to do. However, it is technically possible and can produce an object that violates internal invariants between data fields which may be integral for the object's correct operation. This potential for error can be mitigated to some extent by having read-specials explicitly test invariants.
- It is possible to violate opaqueness by using unpickling to "look" inside objects to examine private state. This is typically a concern faced only by private, critical classes. The class can prevent access by marking sensitive fields transient, by using the specials mechanism to control access or raising exceptions as appropriate.

Guidelines for Safe Pickling of Sensitive Classes

When writing a class that provides controlled access to resources, care must be taken to protect the mechanisms that access those functions. During unpickling (by default) the private state of the object is restored. For example, a file descriptor is a handle that provides access to an operating system stream. Being able to forge a file descriptor would allow some forms of illegal access, since restoring state is done from an insecure stream. To avoid compromising access control, the related state of an object must not be restored from the pickle or it must be reverified by the class. Use one of the following techniques to protect sensitive data in classes:

- The easiest technique is to mark fields that contain sensitive data as "private transient". Transient and static fields are not pickled or unpickled. Simply marking the field will prevent the state from appearing in the pickle and from being restored during unpickling. Since pickling and unpickling (of private fields) cannot be taken over outside of the class the classes transient fields are safe.
- Particularly sensitive classes should not be pickled at all. To accomplish this writeObject and readObject methods (with signatures as described above) should be implemented to throw a NoAccessException passing its class name. Throwing an exception will abort the entire pickling process before any state from the class will be pickled or unpickled.
- Some classes may find it beneficial to allow pickling/unpickling but specifically handle and revalidate the state as it is unpickled. The class should implement writeObject and readObject methods to save and restore only the appropriate state. If access should be denied throwing a new NoAccessException will prevent further access.

7 Future Work

While the current implementation is robust and extensible some additional work is needed. We have not looked at the performance aspects of this scheme but believe that it needs to be characterized and improvements made. Also, the current recursive traversal is suitable for only modest size graphs and will need to be extended to accommodate very large graphs.

No attempt has been made in this work to address the evolution of classes. The general object versioning problem has taken many forms and is very complex. We are investigating what is needed in pickling to support the evolution of classes.

The security of private information is of vital concern to the Internet community. The current implementation requires the programmer to go beyond just putting data in private fields to keep data secret. More conservative approaches are needed for production systems.

Availability

Java Object Serialization will be released with JDK 1.1. Early access versions of this system can be obtained from http://java.sun.com.

References

[1] Arnold, Ken, and James Gosling, *The Java Programming Language,* Addison-Wesley (1996).

[2] Birrell, Andrew, Michael B. Jones, and Edward P. Wobber, *A simple and efficient implementation for small databases,* Digital Equipment Corporation Systems Research Center Technical Report 24 (1987).

[3] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, *Network Objects*. Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).

[4] Gosling, James, and Bill Joy, Guy Steele, *The Java™ Language Specification*, in preparation.

[5] Herlihy, M and B. Liskov, *A Value Transmission Method for Abstract Data Types,* ACM Transactions on Programming Languages and Systems, Volume 4, Number 4, (1982).

[6] Lindholm, Tim and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley (1996), in preparation.

[7] Schneier, Bruce, *Applied Cryptography*, John Wiley & Sons, Inc (1994).

[8] Wollrath, Ann and Roger Riggs, Jim Waldo, *A Distributed Object Model for the Java™* system, Proceedings of the USENIX 2nd Conference on Object-Oriented Technologies and Systems (1996).

Authors

Roger Riggs (roger.riggs@Sun.com) is a Staff Engineer at JavaSoft, working on large scale distribution and reliable distributed systems. Prior to joining the group, he worked on client-server text retrieval and CORBA based information retrieval, and scalability issues on the Web.

Jim Waldo (jim.waldo@Sun.com) is a Senior Staff Engineer at JavaSoft, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked in distributed systems and object-oriented software development at Apollo Computer (later Hewlett Packard), where he was one of the original designers of what has become the Common Object Request Broker Architecture (CORBA).

Ann Wollrath (ann.wollrath@Sun.com) is a Staff Engineer at JavaSoft, working in the area of reliable large-scale distributed systems. Prior to joining Sun, she worked in the Parallel Computing Group at the MITRE Corporation investigating optimistic execution schemes for parallelizing sequential object oriented programs.

Krishna Bharat (kb@cc.gatech.edu) is a Ph.D. student at the College of Computing, Georgia Tech, working in the area of environment and system support for constructing distributed user interface applications. This work was begun when he was a research intern at Sun Microsystems Laboratories (East). He plans to finish his dissertation in June '96, and will be joining the research staff of Digital, Systems Research Center.