Security Capabilities and Potentials of Java*

D. L. Smarkusky, S. A. Demurjian, Sr., M. C. Bastarrica, and T. C. Ting
Computer Science & Engrg. Dept.
The University of Connecticut
Storrs, Connecticut 06269-3155
{debs, steve, cecilia, ting}@engr.uconn.edu
860-486-4818, 860-486-4817 (fax)

Abstract

In the past two years, Java has exploded onto the computing landscape, offering an object-oriented language and environment that is suitable for a wide variety of application domains. Java is targeted for applications that include: advanced capabilities in WWW browsers through web pages that contain applets; enterprise computing with database connectivity, CORBA/ORB availability, remote method invocation services, etc.; usage in personal, commercial, and consumer market products such as smart phones, remote controls, touch screens, etc.; embedded computing applications, such as mobile phones, process control, network routers, etc., that often have real-time constraints; and, smart card technology, which is a credit-card sized computer that could contain personal data and health care data for emergency use. Many of these applications require security as an integral component, to control access and prevent misuse. The purpose of this paper is to focus on the security capabilities and potentials of Java, since there is growing evidence that Java will become a major force in the marketplace. There must be an understanding of the available security primitives in Java, which are significant, along with an investigation of the ability of Java to support existing object-oriented security approaches. Most importantly, it is critical to consider potential security solutions for distributed object computing applications that are likely to dominate the field into the 21st century.

1 Introduction

The Java object-oriented programming language and environment first appeared commercially in early 1996, and in just over 2 years time, there has been an explosive interest and growth of Java across the computing landscape. Java was originally envisioned to augment the capabilities of world-wide web (WWW) browsers through the ability to write applets that can be downloaded from the Internet and manipulated safely within Netscape or Microsoft Explorer. Java has evolved to the situation where:

- Java is taught in educational institutions from freshman undergraduates through graduate student offerings, with textbooks widely available from all major publishers.
- Java is utilized for distributed, Internet-based applications of all types, including graphical user interfaces (GUIs), programming environments, mixed-programming language applications, upgrading and interfacing to legacy systems, etc.
- Java is attractive for general purpose, single-CPU development, since it has the potential to easily evolve software to multiple and varied hardware/OS platforms.

From a security perspective, the usage of Java for the design and development of large-scale, multiprocessor, distributed applications, is of paramount concern. Successful distributed object computing

---

*The work of S. Demurjian and C. Bastarrica has been supported, in part, by a contract from the Mitre Corporation, Eatontown, NJ.
(DOC) can be addressed from three perspectives. First, when developing new applications, it is often the case that multiple programming languages and varied paradigms must work together, e.g., a Java GUI, a C I/O package, and an SQL database system. This motivates the second perspective, involving the integration of Java with commercial-off-the-shelf (COTS) systems. Of course, when integration occurs, it may be necessary to be innovative and creative to allow interactions with legacy applications, which is the third perspective. In all three perspectives, security must be considered, to insure that interoperating legacy, COTS, and database systems can satisfy the security policy of distributed applications. There have been efforts to begin to address security for DOC [6, 8, 15].

The purpose of this paper is to examine and detail the security capabilities and potentials of the Java language/environment. As interest in Java continues to grow, its utilization in security settings must be understood. Java provides a robust set of security capabilities as part of the Java Security Application Programmers Interface (API). These capabilities include digital signatures, message digests, key management, and access control lists. A first goal of this paper is to detail these capabilities, so that the security community can understand the available functionality. Java, as an object-oriented programming language, is of interest from a user role-based security (URBS) perspective, to determine the potential to realize discretionary access control (DAC). Many researchers, including ourselves, have studied this problem for object-oriented/C++ applications and systems [1, 3, 5, 7, 10, 11, 13]. A second goal of this paper is to consider the applicability of our previous approaches to Java. This leads to a third goal, an exploration of the unique features of Java that can be used to enhance existing URBS/DAC/OO approaches or to support new approaches for distributed object computing security.

To meet these goals, the remainder of this paper is organized into four sections and a conclusion. In Section 2, a brief overview of the Java language/environment is provided, for background. In Section 3, the security capabilities of Java, as provided in the Java Security API, are examined, targeting the first goal described previously. Section 4 explores the realization of URBS/DAC approaches in Java, that we have presented in prior work [3, 5], targeting the second goal. Using that material as a basis, Section 5 examines advanced security capabilities, concentrating on the potentials of Java, and targeting the third goal. Finally, Section 6 concludes this paper and outlines future work.

2 Background: An Overview of Java

Java is a third generation, general-purpose, platform-independent, concurrent, class-based, object-oriented language and associated environment, designed to be simple enough so that software engineers can quickly achieve fluency in the language and effectively utilize the language and environment for developing applets and applications. Java can be utilized in two distinct ways. In its original conceptualization, Java can be used to write special programs called applets that can be downloaded from the Internet and displayed/manipulated safely within a WWW browser. In addition, Java can also be used to design and develop standalone applications, with a wide range of capabilities and functionality. Platform independence is achieved in Java by using bytecode, which is similar in concept to the p-code used in early Pascal compilers that revolutionized compiler design and development. Thus, the Java compiler does not produce native executable code for a particular machine. Instead, the Java compiler produces a special format called Java bytecode that is finally interpreted to the native language of the host machine on the fly.

2.1 The Java Environment

Java has two main components, the Java Development Kit (JDK) and the Java Runtime Environment (JRE). The JDK component is a package of programs and support files which is needed to develop Java programs. Included in the support files are class source code and documentation for the complete Java class hierarchy. The JDK contains the command-line driven javac Java compiler. The Java Debugger (JDB) is included with the JDK. It is also command-line driven, and has syntax similar to that of the UNIX dbx and gdb debuggers. The JRE component is needed in order to execute Java applications. It consists of the bytecode interpreter and other files such as the code verifier. Version 1.1.5 of both the JDK [21] and the JRE [23] are available from Sun for Microsoft Windows 95/NT 4.0, SPARC Solaris
2.4, 2.5, 2.5.1, and 2.6, and Solaris 2.5, 2.5.1, and 2.6 x86. In addition, there are a wide variety of platforms available as third-party ports [22], including: AIX, Amiga, BeOS, Digital Unix, FreeBSD, HP 3000, HP-UX, IRIX, Linux, MacOS, Netware 4.1/IntranetWare, OpenServer 5, OS/2 Warp, OS/390, OS/400, Psion Series 3/EPOC 16, RiscOS, RiscBSD, and UnixWare.

In a normal compiled executable program, the object file contains the processor instructions to be executed, and the processor executes the instructions. In order to support platform independence, Java must provide an execution environment that can oversee the execution of applets and applications. The Java Virtual Machine (JVM) is utilized for this purpose. JVM is a program which runs on a particular hardware/OS platform (or ‘real’ machine) which interprets and executes a Java applet/application that is contained in a .class file. The .class file contains both executable JVM instructions (called bytecodes), and additional information such as the class structure, method and data member visibility, and superclass information. Since each JVM interprets the same set of bytecodes, true program portability is achieved by implementing JVMs for a wide variety of platforms.

2.2 A Brief Tour of Object-Oriented Capabilities of Java

This section contains a brief synopsis of the key object-oriented features and capabilities of the Java language. The main modeling capability is the Java class, which is similar to a C++ class. Within a Java class, a member (method or variable) can be tagged as private, public, protected, or package (default). A class for prescriptions in a health care application is given below:

```java
public class Prescription {
    public String Get_Prescription_No(...) { ... }
    public void Set_Prescription_No(...) { ... }
    public String Get_Phamacist_Name(...) { ... }
    public void Set_Phamacist_Name(...) { ... }
    public String Get_Medication(...) { ... }
    public void Set_Medication(...) { ... }

    private String prescription_no;
    private String pharmacist_name;
    private String medication;
}
```

Classes that are related to one another can be grouped together into the package abstraction. For example, a PatientInfo package:

```java
package PatientInfo;

    public class Prescription { ...; }
    public class PatientGUI { ...; }
    public class MedicalRecord { ...; }
...
```

could be declared with classes for prescriptions, a GUI for patients, and a patient’s medical record, to name a few. When a package needs to utilize another package or classes that are external, one or more import statements must be employed.

Inheritance is supported in Java by using the extends keyword when declaring a class. For example, a number of descendants of Item can be defined that form an inheritance hierarchy:

```java
class Test extends Item { ...; }                          Item
                       /   \
class Radiology extends Test { ...; }                      Test
                        |                        Prescription
class Prescription extends Item { ...; }  Radiology
```
2.3 The Java API Packages

Java, through its public interface capabilities and package concepts also requires a clear definition of the exported portion of all classes/packages, which requires software engineers to specifically enumerate which packages, classes, and/or methods are imported. Thus, like Modula-2 and Ada88 (and of course, Ada95), Java provides a set of application programming interface (API) packages. The Java Platform 1.1.5 Core API, available online [16], is shown below:

<table>
<thead>
<tr>
<th>java.applet</th>
<th>java.io</th>
<th>java.rmi.dgc</th>
<th>java.security.interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt</td>
<td>java.lang</td>
<td>java.rmi.registry</td>
<td>java.sql</td>
</tr>
<tr>
<td>java.awt.datatransfer</td>
<td>java.lang.reflect</td>
<td>java.rmi.server</td>
<td>java.text</td>
</tr>
<tr>
<td>java.awt.event</td>
<td>java.math</td>
<td>java.security</td>
<td>java.util</td>
</tr>
<tr>
<td>java.awt.image</td>
<td>java.net</td>
<td>java.security.acl</td>
<td>java.util.zip</td>
</tr>
<tr>
<td>java.beans</td>
<td>java.rmi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each API contains a complete description of the package, which includes the classes and public methods that can be imported and utilized when developing Java applications. For example, the package java.security [18] contains five classes: Certificate, Key, Principal, PrivateKey, and PublicKey. Each class is then described according to the methods in the public interface that are importable by other classes/packages. For instance, the Certificate class contains methods such as decode(), encode(), getFormat(), getPublicKey(), and so on.

3 Java and Security

Java provides transparent, general, and open security mechanisms which do not require any knowledge or action on the part of the software engineer. The sandbox is Java’s basic security mechanism, which forces downloaded applets to run in a confined portion of the system, and allows the software engineer to customize a security policy. One result of this approach is that the security policy is hard coded as part of the application, providing little or no flexibility either to modify the policy or to have discretionary access control. The Java language/environment has features that assist in protecting the integrity of the system and preventing several common attacks. Mainly, Java programs are not allowed to define their own memory pointers or to access physical memory directly. While applets solve many of the important problems in client/server and network-centric computing (code-once, use-many), they also raise concerns about security. Protective measures can go from a firewall between the Internet and the company’s intranet to obtaining software only from trusted sources to the use of anti-virus programs to check all of the software. There is a trade-off between the level of security achieved and the cost of the procedure, since it is impossible to find a security system that is 100% secure.

3.1 Sandboxes

An applet’s actions are restricted to its sandbox, an area of the Web browser dedicated to that applet. The applet may do anything it wants within its sandbox, but cannot read or alter any data outside it. The sandbox model supports the running of untrusted code in a trusted environment so that if a user accidentally imports a hostile applet, that applet cannot damage the local machine. To implement sandboxes, the Java platform relies on three major components: the class loader, the bytecode verifier, and the security manager. Each component plays a key role in maintaining the integrity of the system, assuring that only the correct classes are loaded, that the classes are in the correct format, and that untrusted classes will neither execute dangerous instructions nor access protected system resources. Java’s Protected Domains constitute an extension of the sandbox, and determine the domain and scope in which an applet can execute. Two different protected domains can interact only through trusted code, or by explicit consent of both parties.

The class loader forms the first line of defense in the Java security model by determining how and when applets can load classes, and is responsible for: fetching the applet’s code from the remote machine, creating and enforcing a namespace hierarchy, and preventing applets from invoking methods
that are part of the system's class loader. An executing Java environment permits multiple class 
loaders, each with its own namespace, to be simultaneously active. Namespaces allow the JVM to 
group classes based on where they originated (e.g., local or remote). This delineates and controls what 
other portions of the runtime environment the applet can access and modify. Java applications are free 
to create their own class loaders. In fact, the JDK provides a template for a class loader to facilitate 
customization. Before a class loader may permit a given applet to execute, its code must be checked by 
the bytecode verifier. The verifier insures that the applet's code, which may not have been generated 
by a Java compiler, adheres to all of the rules of the language. In fact, in order to do its job, the 
verifier assumes that all code is meant to crash or penetrate the system's security measures. Then, the 
verifier insures that the code conforms to language specifications and that the applet does not forge 
pointers, circumvent access restrictions, or access objects through an illegal cast. Using a bytecode 
verifier means that Java validates all untrusted code before permitting execution within a namespace. 
Thus, namespaces insure that one applet cannot affect the rest of the runtime environment, and code 
verification insures that an applet cannot violate its own namespace.

3.2 Security Managers

The security manager enforces the boundaries around the sandbox by implementing and imposing 
the security policy for applications. All classes in the Java packages cooperate by asking the security 
manager for permission to perform certain operations. SecurityManager is an abstract class of the 
java.lang API, and provides the programming interface and partial implementation for all Java security 
managers. By default, an application has no security manager, so all operations are allowed. But, 
if there is a security manager, all operations are disallowed by default. Existing browsers and applet 
viewers create their own security managers when starting up.

When there is a security manager, each operation or group of operations will have its own checkXXX 
method. There are checkXXX methods for operations on sockets, threads, files, networking, windows, 
etc. Generally, the application code does not use them, given that the Java runtime does it at a lower 
level. However, they are necessary when overriding some methods. To write a security manager, it 
is necessary to create a subclass of SecurityManager and override most or all of its methods: class 
MyPolicy extends SecurityManager { ... }. Once a new security manager is created, it can 
be installed with the setSecurityManager method from the System class. The security manager 
will remain active until the end of the application. A method that opens a file for reading invokes the 
checkRead method of the security manager. A method that opens a file for writing invokes the 
checkWrite method. If the security manager approves the operation, the checkXXX method returns, 
otherwise, it throws a SecurityException.

3.3 Advanced Security Features

The Java Security API [17], which is part of the java.security package [18], has classes that support 
digital signatures, message digests, key management, and access control lists. Users can rely on the 
default package, or purchase and/or create one of their own. Thus, an organization can select a security 
level commensurate with their risk tolerance. The remainder of this section reviews the capabilities of 
the Java Security API.

Digital Signatures and JAR files

If a particular publisher is trusted, and a signed applet from that publisher has arrived over the Internet 
and been authenticated, then the Java Security Manager could allow that applet out of the sandbox, 
and treat it as an application. The first task of any security system is to be able to assure that who or 
whatever is on the other side of a connection is who or what the user expected to be there, i.e., the host 
that they have connected to is the host they contacted and not an impostor, or the module that they 
have loaded is really the one they expected to run and not a substitute. This is of particular concern 
in downloaded environments where there is a constant threat of a Trojan Horse.
The man-in-the-middle or middleman is a type of attack to which all network-based systems might be vulnerable, and proceeds in a number of steps. First, a client application requests some service from a legitimate server. Unknown to both client and server, an attacker application observes this request and waits for the server to respond. When it does, the attacker intercepts the server's response and replaces it with one of its own, one that the client may assume came from the original server. The way to prevent this type of attack is to ship code contained within a digital shrink-wrap. Java environments can achieve this results using signed applets. To do this, a supplier bundles Java code (and any related files) into a JAR (a Java Archive). The supplier then signs the file with a digital signature. The client can verify the authenticity of the supplier by verifying the signature.

JavaKey [19] is the SUN security provider command-line tool whose primary use is to generate digital signatures for archive files. In order to generate a signature for a particular file, the signer must first have a public/private key pair associated with the file, along with one or more certificates that authenticates its public key. JAR files also provide increased performance. Current Web technology transmits files serially; using JAR files, a Web page can download everything it needs in a single request.

**Key Management**

The Java Security API provides support for integrated key management in Java programs and applets. All keys have three characteristics [20]:

**An Algorithm:** The key algorithm is usually an encryption or asymmetric operation algorithm (such as DSA or RSA) which will work with those algorithms and with related algorithms (such as MD5 with RSA, SHA-1 with RSA, RawDSA, etc.).

**An Encoded Form:** This is an external encoded form for the key that is used when transmitting it to some other party. The key is encoded according to a standard format (such as X.509 or PKCS#8).

**A Format:** This is the name of the format of the encoded key.

Keys are generally obtained through key generators, certificates, or the various Identity classes used to manage keys. There are no provisions yet for the parsing of encoded keys and certificates. An identity certificate is a guarantee by a principal that a public key is that of another principal. The KeyPairGenerator class is used to generate pairs of public and private keys. Key generation is an area that sometimes does not lend itself well to algorithm independence. There are therefore two ways to generate a key pair: in an algorithm-independent manner and in an algorithm-specific manner.

**Message Digests**

Using public key encryption algorithms to encrypt messages can be quite slow. As a result, cryptographers have developed a way to generate a short, unique representation of your message, called a message digest, that can be encrypted and then used as your digital signature. The MessageDigest class provides the functionality of a message digest algorithm, such as MD5 or SHA. Message digests are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value. Like other algorithm-based classes in the Java Security API, MessageDigest has two major components: the MessageDigest API, or the methods called by applications needing message digest services, and the MessageDigest SPI (Service Provider Interface), or the interface implemented by providers that supply specific algorithms.

**Access Control Lists**

Every authenticated principal will have a level of accessibility: highly trusted resources should be granted more access than those of more dubious origin. Access Control Lists (ACL) are data structures used to guard access to resources, and allow users to define read/write permissions based on users and groups. ACLs simplify the effort required for programs to maintain and access these lists. An ACL can be thought of as a data structure with multiple ACL entries. Each ACL entry contains a
set of positive or negative permissions associated with a particular principal (an individual user or a group). Individual permissions (either positive or negative) override the groups’ permissions. The java.security.acl package provides the interfaces to the ACL and related data structures (ACL entries, groups, permissions, etc.), and the sun.security.acl API provides a default implementation.

4 Java and User-Role Based Security

Security issues in distributed object computing are difficult to address since security of individual systems (legacy, COTS, database) must be supported and enhanced across the entire distributed and interoperating application. In Section 3, we addressed aspects of security that are directly supportable via primitives provided by the Java Security API. In this section, we consider the ability of Java to support user-role based security (URBS) approaches, where permissions and right to access are be assigned based on the individual roles, rather than to specific users. URBS is a realization of discretionary access control (DAC), that assigns rights and permissions to roles rather than to individual users, with users assigned to specific roles. Work on URBS/DAC for object-oriented/C++ systems and applications has been our major emphasis in the past five years [2, 3, 4, 5].

The premise of our efforts is that the public interface provided by object-oriented programming languages is not suited for the customized approach that is needed for supporting URBS/DAC. The public interface of a class is the union of all privileges (methods) needed by all users of each class. This allows methods intended for only specific users to be available to all users. Our past approaches have strengthened the public interface concept, promoting the idea that different subsets of the public interface are available to specific users based on role, thereby providing a means to realize URBS/DAC. We have detailed a number of extensible and reusable URBS/DAC enforcement mechanisms that utilize inheritance, generics, and exception handling for the automatic generation of code for DAC policies [3, 4, 5]. The goal of these efforts has been to minimize the amount of knowledge a software engineer must have on URBS/DAC by having mechanisms that are self-contained, class libraries, which supply all of the required code to define and enforce the desired security policy.

This section begins by providing background material on our URBS approach. Specifically, in our URBS model, we have a user-role definition hierarchy (URDH) to organize responsibilities and to establish privileges. Privileges can be assigned (can invoke a set of application methods) or prohibited (cannot invoke a set of application methods) to roles. Given this background, the remainder of this section reviews the realization and support of three of our prior URBS approaches in the Java language/environment. We complete this section with some remarks on the limitations of Java in support of our prior URBS approaches.

4.1 User-Role Based Security

To support URBS, the user-role definition hierarchy (URDH) characterizes the different kinds of individuals (and groups) who require different levels of access to an application. Figure 1 shows a partial URDH for a health care application (HCA). User roles (UR) (e.g., Staff RN, Education, etc.), can be grouped under a single user type (UT) (e.g., Nurse). When multiple UTs share privileges, a user class (UC) can be defined (e.g., MedicalStaff). To define, UCs, UTs, and URs, we utilize a node profile (NP): 1. a name for the node; 2. a prose description of its responsibility; 3. a set of assigned methods (the positive privileges); 4. a set of prohibited methods (the negative privileges); and 5. a set of consistency criteria for relating URDH nodes.

A node description for a UT in Figure 1 is: Nurse: Direct involvement with patient care on a daily basis. In addition, for each UR, the role-security requirements are defined, e.g., Staff RN: All clinical information for the patients that they are responsible for. Can write/modify portions of clinical information to track patient progress. Cannot change a Physician’s orders on a patient. To establish privileges, application methods are assigned to URDH nodes. Commonalities (shared methods) can be moved from a set of URs to their shared UT, and from a set of UTs to their shared UC. Methods shared by all UTs can be moved up to Users. Thus, commonalities flow up the URDH, while differences flow down. Prohibited methods are used to explicitly identify which methods cannot be accessed by
a URDH node. *Equivalence (subsumption) criteria* allow the security engineer to identify which URs (UTs/UCs) must have the same (subsumable) capabilities, as reflected in the assigned/prohibited methods. Conflicts between assigned and prohibited methods or violations of consistency criteria that can be automatically flagged.

### 4.2 User-Role Subclassing Approach

The user-role subclassing approach (URSA) utilizes inheritance to derive new user-role subclasses for each object type (OT) in an application. Each subclass of an OT contains the assigned and prohibited methods by the UR on that OT. In URSA, each class of the application has a group of subclasses, based on the different roles that have some subset of assigned and/or prohibited methods from the class. As subclasses, the basic concept is to inherit the methods that are assigned while simultaneously inheriting the prohibited methods. The key for URSA is to turn off the prohibited methods. In Java, all methods are virtual unless tagged as final.

```java
public class Prescription { // As given in Section 2.2 }

public class Staff_MD_Prescription extends Prescription
{
    public void Set_Prescription_No(...)  
        { return; // Prohibit access to this method - Turn Off }

    public void Set_Pharmacist_Name(...)  
        { return; // Prohibit access to this method - Turn Off }

    public void Set_Medications(...)      
        { return; // Prohibit access to this method - Turn Off }
}

public class Attending_MD_Prescription extends Prescription
{
    public void Set_Pharmacist_Name(...)  
```
If the Set_Prescription_No method is requested by an individual whose role is Staff_RN, then the method associated with the Set_Prescription_No of the Staff_RN_Prescription subclass is executed and no value is returned.

4.3 URDH Class Library Approach

The URDH class library approach (UCLA), employs inheritance to implement the enforcement mechanism from the combined perspective of the application’s URDH and the OT/class library. In UCLA, a new class hierarchy is provided, where each class represents a URDH node and the access rights of the node are specified via a set of validation methods. For each URDH node, positive methods access is defined based on the assigned methods that have been specified. As the application executes, each method must validate against the current UR.

```java
public class Root: All Check Methods defined to return False;
public class Users extends Root {}
public class Medical_Staff extends Root {}

public class Nurse extends Medical_Staff
    { public boolean Check_Prescription_Get_Medication()
        { return True; } }

public class Staff_RN extends Nurse
    { public boolean Check_Prescription_Get_Prescription_No()
        { return True; }

        public boolean Check_Prescription_Get_Pharmacist_Name()
        { return True; } }
```

The Root class includes new Check methods which are defined for all application methods from all classes to return False. These check methods will be turned on at lower levels (UC/UT/UR) by the assigned methods of the URDH. These Check methods are also utilized to change the code that can be generated for each class:

```java
class Prescription extends Item
{
    public Prescription( String name, String D, int No, String Name1, String Med)
    { // initialize variables }

    public int Get_Prescription_No()
    { if (current_user.Check_Prescription_Get_Prescription_No())
       return (Prescription_No);
    else
       return NULL;
    }

    public void Set_Prescription_No(int No)
    { if (current_user.Check_Prescription_Set_Prescription_No())
       Prescription_No = No;
    }
}
```

Once the user role has been determined, a new global current_user object will be created at run-time and casted to the selected user role.
4.4 Basic Exception Approach

Exception handling in Java is similar to that of C++, where the try construct is utilized to encapsulate a block of code (in our case, a method call) that has the potential to raise an exception. As the code within the try block is executing, various conditions can be checked, and when the correct situation occurs (in our case, an attempt to access unauthorized UR), an exception can be raised using the throw construct. This thrown exception is then processed by the catch block which typically follows the original try block. In our case, the catch block will be used to process the security violation.

In the basic exception approach (BEA), each class is modified to include a set of methods for exception handling. This is illustrated below. Note that the import and variable definitions have been omitted to both simplify and clarify the presentation.

```java
public class Prescription extends Item {
    // Private data has been omitted

    public Prescription(String Name, String D, int No, String Name1, String Med) {
        // Assign Prescription variables, call Item constructor
    }

    public int Get_Prescription_No() {
        return rtn_int_check_valid_URI(Prescription_No);
    }

    // All Other Prescription methods

    public int rtn_int_check_valid_URI(int rtn_int_ck) {
        try {
            Check_URI();
        } catch (Unauthorized_URI UR_exception) {
            System.out.println("Attempt to access unauthorized UR");
        }
    }

    // All other data type check_valid_URI methods

    public void Check_URI() throws Unauthorized_URI {
        if (!compareTo(current_user.Get_User_Role(), "Staff_RW")
            & (compareTo(current_user.Get_User_Role(), "Attending_MD")
                != 0))
            throw new Unauthorized_URI; // throw raises exception
    }

    Check_URI is needed to verify that the current UR can invoke the desired method. For the purpose of this first example, we'll assume a simple table lookup. The class Unauthorized_URI is an exception handling class. While the functionality in this case is null, it can be totally expanded to handle complex situations based on the exception that is raised.

4.5 Limitations of Java in Support of URBS

While Java appears to easily support the various approaches given in Sections 4.2, 4.3, and 4.4, in actuality, Java falls short in one regard, and is also limited with respect to generics-based approaches that were not presented. These two limitations are reviewed below:

• UCLA, as presented in Section 4.3, is in fact, not fully supported. UCLA as originally conceptualized and realized via C++ [3] requires the availability of multiple inheritance. Multiple inheritance is utilized in the URDH, since a user type inherits from both User and a user class
(see Figure 1 again). While Java can realize UCLA through the replication of privileges from User into either the user types or user classes, it is not an ideal solution. The interface capability of Java, which supports design-level multiple inheritance, is also not appropriate, since interfaces do not allow implementations to be inherited, but only provide the signatures of methods without implementations.

• UCLA and BEA, as presented in Sections 4.3 and 4.4, respectively, have corresponding approaches (GUCLA and GEA) that utilize generics [5]. We have purposely not presented these approaches herein, since Java does not support a generic class. Without generics, the ability to reuse security definition and enforcement code is a major drawback of the language.

While Java appears to have stabilized from a language design perspective (i.e., no major changes to the language definition in quite some time), the user community may call of the inclusion of both multiple inheritance and generics, since both concepts are fundamental to software reuse.

5 Advanced Security Features and URBS

This section focuses on the third goal of the paper, the ability to utilize security features of Java for URBS, thereby truly exploring the potentials of the language. Thus, we focus on capabilities of Java that exist in the Java Security API (as presented in Section 3) as well as functionality that is provided as part of the Java runtime environment. The remainder of this section considers four advanced capabilities of Java and their potential for supporting URBS: packages for encapsulating security definition and enforcement code; access control lists which are part of the Java Security API; the Class class of the Java Language API that allows dynamic queries on the methods defined on application classes; and, software agents which are supported by Java aglets.

5.1 Packages in Java

The highest level of abstraction/encapsulation in Java is the package, as briefly reviewed in Section 2.2, which allows collections of one or more classes to be bound into a single named unit. For example, consider the two declarations of a PatientInfo package:

```java
package PatientInfo;

class Prescription { ... };
class PatientGUI { ... };
class MedicalRecord { ... };
...
```

```java
package PatientInfo;

public class Prescription { ... };
public class PatientGUI { ... };
public class MedicalRecord { ... };
...
```

In the version on the left, the classes are only visible within the package in which they are defined. In the version on the right, classes tagged with the public qualifier are visible within the package and externally. A package without at least one public class isn’t reasonable, since it does not provide an entry into the package. The package access specifier supports sharing across classes within a single package, while simultaneously hiding this information from classes in different packages.

The package construct can be instrumental in encapsulating the security definition and enforcement code that is required for the different URBS approaches. For example, in UCLA (see Section 4.1 again), the entire URDH class library can be encapsulated into a single package, allowing changes to the URDH to be localized to a single, controlled package. In the case of URSA (see Section 4.2 again), the classes Prescription, Staff BN Prescription, and Attending MD Prescription can be encapsulated into a single package, with Prescription not tagged as a public class. This would mean that only the other two user-role subclasses, tagged as public, are visible externally, which would further protect unauthorized access to Prescription, since all access must go through the user-role subclasses. Packages could be established for each application class (or inheritance hierarchy) and its user-role subclasses. Overall, the use of packages, when coupled with inheritance and the visibility primitives available in Java for classes, greatly enhances the encapsulation and management of the security definition and enforcement code.
5.2 Access Control Lists

Recall that in Section 4.1 we presented the user-role definition hierarchy, URDH, as a means to define the user roles. As discussed in Sections 4.2 and 4.3, the main purpose of the URDH is to allow methods that are defined on classes throughout the application to be assigned and/or prohibited to various user classes, user types, and user roles. The privileges associated with the URDH are directly supported in Java, via the Access Control List (ACL) [25], which is a data structure used to guard access to resources. This data structure contains ACL entries. Each ACL entry contains a set of permissions (access to methods) and for a particular principal (UR or UT). Privileges are assigned when the principal is allowed to access a method and prohibited otherwise. The individual permissions (for URBS, the UR) will override permissions of the group (for URBS, the UT) to which an individual belongs. Permissions for the URBS would be a set of all possible methods. So, we could not only assign methods to a user role, but also to a user type.

The following methods from the ACL interface, java.security.acl.ACL, are required for the support of URBS:

1. addEntry(): Adds an ACL entry to the Access Control List. This entry contains the specified user and a list of methods which are assigned or prohibited for this user. Remember that there can only be one list of assigned and prohibited methods, which is dynamically determined by the user role that is being played.

2. checkPermission(): Returns true if the input user has permission to access the input method, false otherwise.

3. getPermission(): Returns an enumeration of all methods which are assigned or prohibited for the input user. The assigned/prohibited methods are determined by first obtaining the assigned/prohibited methods for the group (in our case the UT) and then determining the assigned/prohibited methods for the individual (in our case the UR). The final permissions are then determined by allowing the individual permissions to override the group permissions for both the assigned and prohibited methods. The assigned permission set is returned.

4. removeEntry(): Removes an ACL entry from the Access Control List.

The following methods from the ACLEntry interface, java.security.acl.ACLEntry, are required to build the ACL Entry for a URBS ACL:

1. addPermission(): Adds a permission (method) to the ACL Entry. There can be more than one method in each ACL Entry.

2. checkPermission(): Determines if a permission (method) is already part of the ACL Entry.

3. removePermission(): Removes a permission (method) from the entry.

4. setPrincipal(): Specifies the user role or user class for which the permissions (methods) are assigned or prohibited.

5. getPrincipal(): Returns the user role or user class for which these permissions (methods) are assigned or prohibited.

6. setNegativePermissions(): Set the ACL entry to be a list of negative permissions (prohibited methods). All ACL entries are by default positive (assigned).

For URBS, we would specify the permissions of all methods so that the checkPermission() method could be invoked to accurately determine both the assigned and prohibited methods. As we stated earlier, the URs inherit all of the permissions of the parent UT. Therefore, the Java API for Group, java.security.acl.Group, could be utilized to assign URs to the UTs, via the addMember() and removeMember() methods, where the UT would be the Group.
The ACL capabilities of Java would be instrumental in supporting the three URBS approaches (URSA, UCLA, and BEA) given in Sections 4.2, 4.3, and 4.4. For URSA, UCLA, and BEA, ACL can be utilized to track the information required for an authorization list, that would bind users to their associated roles upon login. For BEA, the ACL has the most significant potential use, namely for the Check.JR method, that is able to verify which URs have access to which methods. Using an ACL, this information could be dynamically changed, whenever the security requirements cause the addition/deletion of roles or changes in application classes. This would be one of the major benefits of using Java's ACL in role-based security. While an ACL can be implemented in any language, having one designed, implemented, tested, and with a standard interface, is a definite advantage to Java.

5.3 The Class Class in Java

In the java.lang API, the Object and Class classes have a large set of methods defined that are accessible to software engineers for obtaining information about any system- or user-defined class in Java. For instance, Class has methods that can be invoked to return, for a specific user or system class, a list of its public methods, member variables, declared constructors, etc.

The Class class can be used by URSA, UCLA, and BEA for the retrieval of all public methods for each class. This would reduce the overhead of having to statically determine all methods and their signatures, but would allow the methods to be obtained dynamically. The retrieved methods would have a default permission of assigned and only the links to the prohibited methods would need to be removed. This is a powerful capability, since it can be utilized in different ways by the various approaches. For example, the Check.JR method of BEA, if implemented with ACL as described in Section 5.2, could utilize the getMethods method of Class whenever the security policy was updated. This would allow the revised/updated entries of the ACL (that contain, for each role, the assigned methods) to be automatically and dynamically compared against the actual methods defined on each class. Similarly, whenever a class was altered, this verification could also occur. In both situations, the maintenance of the security policy is greatly simplified.

5.4 Java and Aglets

Mobile software agents are defined in formal terms as objects that have behavior, state and location [14]. Agents can move from place to place and have a specific function or responsibility to perform. Agents are like other objects in that they can be created and destroyed, but they can also migrate to a new location, execute their required responsibilities, and process incoming messages from other agents. Agents cannot interact by invoking each others methods, rather, they communicate via message passing.

IBM terms these mobile agents of Java, "aglets", combining the terms of "agent" and "applet" [26]. Unlike a Java applet, an aglet continues execution where it left off (upon reaching a new location). This is possible because an aglet is an executable object (containing code and state data) which moves from host to host across a network [24]. Karjoth describes a proxy as a representative of an aglet which serves as a shield to protect the aglet from direct access to its public methods [9]. The proxy used for the aglet is similar in function to URBS, as both have the responsibility to prevent the access of unauthorized users (or agents).

Like applets, aglet actions should be restricted to a sandbox (see Section 3.1 again). The sandbox model supports the running of untrusted code in a trusted environment so that if a hostile aglet is received, that aglet cannot damage the local machine. For applets, this security is enforced through three major components: the class loader, the bytecode verifier, and the security manager. Aglets would require the same level of security as applets. The aglets would need to ask permission from the security manager before performing operations, thus allowing the security manager to know the identity of the aglet.

Mobile aglet security is progressing with the use of the Java sandbox mechanism and separation execution environments [9]. Java security mechanisms such as cryptography and authentication are also be investigated to ensure security of both the aglet and the messages transported between aglets. Aglets offer the opportunity to rethink our URBS security approaches, which are class/method based
for user roles, and whose definition process is focused on type-level concerns. In distributed object computing, it is critical to explore the security of runtime objects, as they are accessed by users playing roles. Aglets may provide active objects that monitor and/or enforce security, from the perspective of the user, the user role, the object, or any/all combinations. As security needs change, security aglets can be dynamically updated to maintain their oversight and enforcement capability. Aglets in Java must be examined for their potential to support security in distributed object computing.

6 Concluding Remarks and Future Work

This paper has examined the security capabilities and potentials of the Java object-oriented language/environment. Java has many attractive features, as presented in Section 2, the most important of which is its platform independence. Java takes a module approach to software development, providing a robust set of APIs (packages - see Section 2.3) to software engineers as tools for GUI development, communications, etc., and for our purposes, security. There are a wide range of security capabilities, provided in the Java Security API, including digital signatures, message digests, key management, and access control lists, which all function under the control of a security manager, as described in Section 3. From an object-oriented/programming language perspective, Section 4 examined the ability of Java to support our previous URBS approaches. While some of the approaches were realizable in Java, others that utilize multiple inheritance and generics could not be fully attained, as discussed in Section 4.5. Since Java provides advanced security and language capabilities (see Section 3) not offered by C++, Section 5 examined the potential for utilizing these capabilities in support of URBS. Specifically, we considered the package abstraction which can be utilized to encapsulate URBS code (see Section 5.1), Java's access control lists for realizing important components of our URBS approaches (see Section 5.2), and the Class class for performing automatic and dynamic verification of security privileges (see Section 5.3).

One of the more interesting potentials of Java is related to our future work, namely, the utilization of Java agents, or aglets, for supporting security in distributed object computing, as seen in Section 5.4. We have begun to explore this capability, and will consider it in conjunction with our URBS approaches (see Section 4 again). Another future related area is the support of security within the CORBA/ORB framework, which is the only available standard for distributed computing. A third related area is security capabilities offered by emerging object-oriented database platforms, including the recently announced Jasmine by CAI. Our effort on software architectural alternatives for URBS policies [6], and other research [8, 15], is relevant in these three areas of future work.

References


