ANÁLISIS, PRUEBA Y DOCUMENTACIÓN DEL USO DEL LENGUAJE JAVA EN APLICACIONES SEGURAS

ROBERTO ENRIQUE RAMIREZ ARRIETA

ANTONIO ALFREDO REVOLLO CARRILLO

TECNOLOGICA DE BOLIVAR INSTITUCIÓN UNIVERSITARIA

FACULTAD DE INGENIERIA DE SISTEMAS

CARTAGENA D.T Y C.

2.000

ANÁLISIS, PRUEBA Y DOCUMENTACIÓN DEL USO DEL LENGUAJE JAVA EN APLICACIONES SEGURAS

ROBERTO ENRIQUE RAMÍREZ ARRIETA

ANTONIO ALFREDO REVOLLO CARRILLO

Trabajo de grado presentado como requisito para optar el título de Ingenieros de Sistemas

Director:
CARLOS VALENCIA MARTÍNEZ
Ingeniero Electricista

TECNOLOGICA DE BOLIVAR INSTITUCIÓN UNIVERSITARIA

FACULTAD DE INGENIERIA DE SISTEMAS

CARTAGENA D.T Y C.

2.000

Cartagena, Abril 10 del 2000.

Señores:
COMITÉ DE EVALUACIÓN DE PROYECTOS
TECNOLÓGICA DE BOLÍVAR INSTITUCIÓN UNIVERSITARIA
L. C.

Respetados señores.

A petición de los estudiantes **ROBERTO ENRIQUE RAMIREZ ARRIETA** y **ANTONIO ALFREDO REVOLLO CARRILLO**, matriculados en el programa de Ingeniería de Sistemas, acepté participar como director de tesis para la elaboración del proyecto **"ANÁLISIS, PRUEBA Y DOCUMENTACIÓN DEL USO DEL LENGUAJE JAVA EN APLICACIONES SEGURAS "**, como propuesta de tesis para optar al título de Ingenieros de Sistemas.

Atentamente,

_____
Ing. CARLOS VALENCIA
c.c. # 73'148.708 de Cartagena

Cartagena, Abril 10 del 2000.

Señores:
COMITÉ DE EVALUACIÓN DE PROYECTOS
TECNOLÓGICA DE BOLÍVAR INSTITUCIÓN UNIVERSITARIA
L. C.

Respetados señores.

La presente es con el fin de plantearle a ustedes nuestro tema de tesis de grado **"Análisis, Prueba y Documentación del Uso del Lenguaje Java en Aplicaciones Seguras ",** sugerido a partir de las investigaciones ya realizadas en el proyecto de maestría **"Software de Correo Electrónico Seguro Basado en Secuencias de Lucas"**, llevado a cabo por los Ingenieros Carlos Valencia y Giovanny Vasquez.

Se tuvieron en cuenta las siguientes razones para llevar a cabo este proyecto:

a) La importancia que ha adquirido el uso del lenguaje Java y la muy poca documentación de uso de librerías para aplicaciones seguras en este lenguaje.
b) Apoyo a la investigación del proyecto de maestría anteriormente mencionado.
c) Apropiación de esta tecnología.
d) Dejar una base para próximas investigaciones por parte de otros estudiantes que se interesen en este tema.

Agradeciendo de antemano la atención prestada a la presente.

Atentamente,

_____	_____
Antonio A. Revollo Carrillo	Roberto E. Ramírez Arrieta
c.c # 73’572.534 C/gena	c.c # 73’155.091 C/gena

ARTICULO 105.

La Institución se reserva el derecho de propiedad intelectual de todos los trabajos de grado aprobados, los cuales no pueden ser explotados comercialmente sin su aprobación.

DEDICATORIA

Este trabajo se lo dedico a mis padres, *Antonio y Nelly* ... a mi hermana *Nelly,* y a mi tía *Myriam* que me apoyaron y me ayudaron constantemente.

A mis abuelos *Francisca Montero* y *Horacio Revollo* que fueron , son y serán los viejitos que más estimo.

Antonio Revollo Carrillo

AGRADECIMIENTOS

Agradezco a Dios nuestro Señor por mantenerme hasta este momento con vida y salud.

A mi familia por contribuir indirecta o directamente en la persona que soy ahora mismo.

A la **Tecnológica de Bolívar** por darme la oportunidad de formarme como profesional. Al Ing. *Carlos Valencia Martínez* por su colaboración para el desarrollo de ésta Tesis de Grado.

A mi compañero de tesis Roberto Ramírez por sus aportes y apoyo en todo este tiempo.

Y a todos mis amigos que me brindaron su mano hasta último momento.

Antonio Revollo Carrillo

DEDICATORIA

Este trabajo se lo dedico a quien para mi son las mejores personas del mundo, mis padres, *Roberto Ramírez Nieto y Carmen Arrieta Flores.*

A mis hermanas y sobrinas con quienes día a día comparto los buenos y malos ratos que nos da la vida.

A Dios por acompañarme cada día.

Roberto Ramírez Arrieta

AGRADECIMIENTOS

A mis padres por su apoyo incansable.

A *Mavel de la Espriella*, porque sus consejos me ayudaron a continuar con lo que hoy me enorgullece.

A *Carlos Valencia*, nuestro director, por su valioso apoyo en este trabajo.

Al mejor compañero y amigo, *Antonio Revollo* por su respetable filosofía.

A la familia tecnológica, *Miriam, José, Mantilla, Garzón, Nelson, Migue, Clari,...* y todos sus demás integrantes quienes hicieron de mi estadía en la "U" algo inolvidable.

Roberto Ramírez Arrieta

Nota de Aceptación

_____

_____

_____

_____
Presidente del Jurado

_____
Jurado

_____
Jurado

Cartagena, (día, mes, año)

CONTENIDO

LISTA DE FIGURAS

INTRODUCCIÓN

Si se quiere una aplicación segura en un lenguaje de programación cualquiera, el programador debe idear sus propios algoritmos y mecanismos para lograrlo o algunos veces busca la forma de adquirir modelos adicionales para tal fin.

Sin embargo, cuando ese lenguaje de desarrollo de aplicaciones es Java, el programador cuenta con herramientas y un ambiente de trabajo adecuado que le permite el desarrollo de aplicaciones altamente seguras.

Se debe tener en cuenta que las herramientas proporcionadas por Java no hacen que la aplicación sea segura por si sola. Esta característica se consigue mediante el buen uso de las librerías de seguridad que proporciona el lenguaje y además teniendo en cuenta que una aplicación Java puede interactuar con cualquier sistema o traficar por redes (applets), es necesario hacer revisión adecuada y permanente de la seguridad de las aplicaciones y el ambiente donde estos actúan.

Este documento es orientado hacía lectores que conozcan el leguaje de programación Java y desee explotar las herramientas de seguridad, poco conocidas en el ambiente universitario de la CUTB, que convierte a Java en más de un lenguaje de programación de aplicaciones Orientado a Objetos.

1. MANIPULACIÓN DE LA MEMORIA POR PARTE DE JAVA

1.1. CÓMO JAVA MANEJA LA MEMORIA?

1.1.1. Hospedaje de Objetos: Cuando Java crea un objeto por medio de la palabra clave **new,** no existe ninguna palabra clave para deshacerse de ellos, entonces pareciera que Java está lleno de fugas de memoria, ya que nunca se llama a ningún método para liberar la memoria que está ocupada.

Sin embargo Java usa una técnica llamada *garbage collection* (recolección de basura) para detectar y liberar automáticamente los objetos en desuso, esto es, que nunca tendrá de preocuparse por liberar memoria o destruir objetos, pues el recolector de basura se encargará de ello.

Es una técnica que ha estado en uso por años en lenguajes como Lisp. Debido a que el intérprete de Java sabe qué objetos ha asignado, qué variables se refieren a qué objetos y cuáles objetos hacen referencias a otros objetos, puede saber también cuándo un objeto asignado ya no está referenciado por otro objeto o variable. Una vez que encuentra dicho objeto, lo destruye sin peligro. El recolector

de basura también puede detectar y destruir "ciclos" de objetos que se referencien entre sí, pero que no sean referenciados por ningún otro objeto.

Para realizar toda esta técnica la Maquina Virtual de Java cuenta con una pila como estructura de datos, en la cual se guardan los objetos de Java durante la ejecución del programa. La cantidad de objetos que pueden ser guardados en la pila es una información pertinente al fabricante de la Máquina Virtual de Java, lo cual agrega un nivel de seguridad debido a que un Hacker no tiene idea de como son representados los objetos en memoria. Esto hace más difícil montar un ataque que dependa del acceso a la memoria directamente. Cuando un objeto no es usado más, la máquina virtual lo marca para la colección de basura.

El recolector de basura de Java se ejecuta como un hilo de baja prioridad y hace casi todo su trabajo cuando no hay tareas en proceso. Por lo general, se ejecuta en momentos de inactividad, cuando el usuario introduce datos mediante el teclado o movimientos del ratón, la única ocasión en que el recolector de basura se ejecuta sin importar que algo de alta prioridad se esté llevando a cabo (en este caso, disminuye realmente la velocidad del sistema), es cuando el intérprete queda sin memoria. Esto no ocurre con frecuencia, ya que el hilo de baja prioridad se encarga de la limpieza en el plano secundario.

Este esquema puede parecer extremadamente lento y un desperdicio de memoria, sin embargo, los buenos recolectores de basura pueden ser muy eficaces y hacer

que la programación sea más sencilla y menos propensa a problemas, aunque en verdad nunca sean tan buenos como una bien escrita asignación y desasignación de memoria Pero en casi todos los programas existentes, un desarrollo rápido, la inexistencia de errores y un mantenimiento sencillo son características más importantes que la velocidad real o la eficiencia de la memoria.

Al final del documento se presenta el manual del programador de un sencillo programa que simula la técnica del Recolector de Basura.

1.1.2. Hospedaje de Clases: El área de clases es donde la JVM guarda información específica de clases tal como métodos y campos estáticos. Cuando una clase es cargada y ha sido verificada, la JVM crea una entrada en el área de clases para esa clase.

Generalmente el área de clases es simplemente una parte de la pila. En este caso, las clases pueden ser también basura recopilada una vez no sean usadas. Alternativamente, el área de clase puede ser una parte separada de la memoria y requerirá de lógica adicional en la parte del realizador de la JVM para limpiar las clases que no están en uso.

1.2. JAVA NO USA PUNTEROS

En Java la referencia y desreferencia de objetos se maneja automáticamente. Java no permite que el usuario manipule punteros o direcciones de memoria de ningún tipo:

- No acepta que usted lance referencias de objetos o arreglo a enteros o viceversa.
- No permite que usted haga aritmética de puntero.
- No deja que usted compute el tamaño en bytes de ningún tipo primitivo u objeto.

Existen dos razones para estas restricciones:

- Los punteros constituyen una fuente notoria de errores. Al eliminarlos se simplifica el lenguaje y se acaban muchos errores potenciales.

- Los punteros y la aritmética de punteros se podrían usar para evitar que ocurran verificaciones al tiempo de la ejecución y que funcionen los mecanismos de seguridad en Java. Eliminar punteros permite que Java proporcione garantías de seguridad.

1.3. EL SANDBOX DE JAVA

Los Applets cuentan (al menos en teoría) con un lugar muy seguro para correr un programa, este lugar es llamado ***el sandbox***, el cual fue diseñado teniendo en cuenta los siguientes objetivos:

- Evitar daños al browser del sistema ocasionados por la actualización de archivos o la ejecución de comandos del sistema.
- Evitar la recuperación de datos no deseados ya sea leyendo archivos o extrayendo información del medio.
- Evitar que el browser sea usado como una plataforma para el ataque a otros sistemas.
- Evitar la incorporación de clases Java de confianza en el browser ya que puede sobrepasarse del limite o ser alterado.

Este último objetivo es la clave a todos los otros. Porque el Administrador de Seguridad es, así mismo, una clase incorporada ya que si un atacante puede alterarlo o desviarlo, todo el control se pierde.

El Administrador de Seguridad es la parte del código local del browser, debido a que la implementación de las restricciones de cajón es responsabilidad de cada proveedor de browser. Sin embargo, todos ellos tienen los mismos objetivos, de tal

forma que el resultado es un conjunto de restricciones que son comunes para la mayoría los vendedores de estos productos:

- Ningún applet accede a un disco local.
- Muy limitado a la información del medio.
- El único host que un applet puede reconocer para una conexión de red es el primero desde el cual fue cargado.
- Ninguno enlaza al código local.
- Ninguno imprime.

Para crear aplicaciones cliente/servidor efectivas usando Java requiere que se le de al applet alguna libertad de la seguridad del sanbox.

El modelo de seguridad de Java es construido alrededor del concepto de un dominio de protección. El ***sandbox*** del applet es un dominio de protección con controles muy estrictos, en contraste, el ambiente de aplicaciones de Java es un dominio de protección sin ningún control, que el impuesto por el sistema operativo. Lo que se busca es un dominio de protección que se sitúe en medio de los dos.

El JDK 1.1 ofrece applets firmadas como forma de escape de las restricciones del sandbox. Las applets firmadas proporcionan el mecanismo para el dominio de protecciones deseado.

1.4. LA MÁQUINA VIRTUAL DE JAVA

Java es una programa tan robusto que hacer una estructura detallada de como quedaran los diferentes segmentos dentro de la memoria es algo complicado debido a que cada proveedor de software tendrá su forma particular para gestionar la memoria por medio de la Maquina Virtual de Java (JVM), es de ahí, que cualquier aplicación hecha con esta herramienta se hace confiable y a su vez segura hacía los ataques que pueda recibir por parte de intrusos.

En la figura 1, el área resaltada con puntos discontinuos muestra la Máquina Virtual de Java (JVM). Cada uno de los componentes que se muestran agrega un nivel de seguridad al sistema donde interactua dicha máquina.

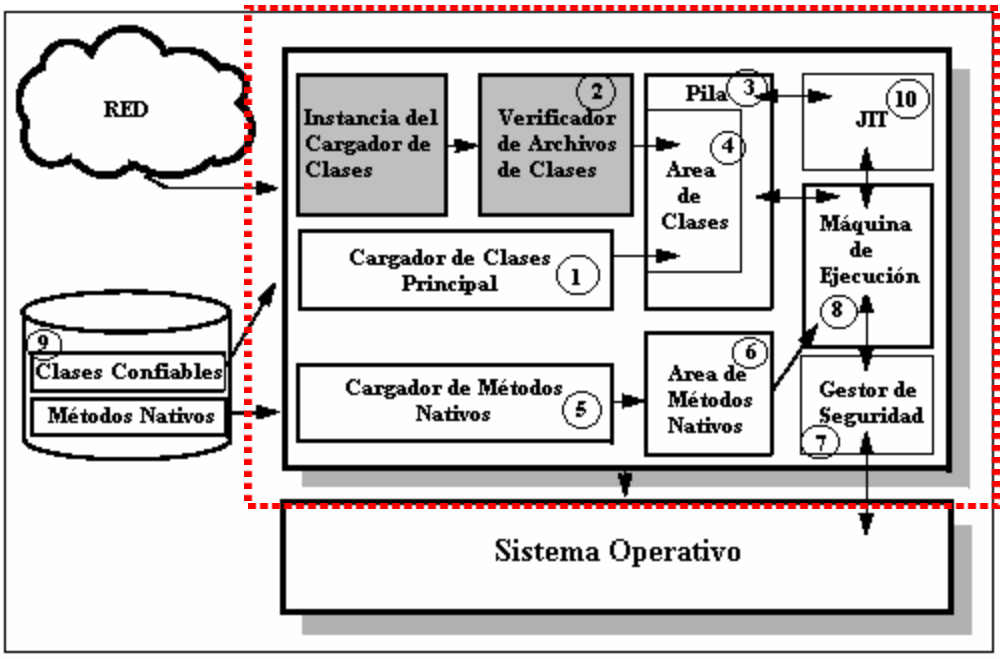


Figura 1. Aspectos de la Máquina Virtual de Java

Estos componentes son generalmente encontrados en Browser de Web.

1) El Cargador de Clases Principal

Antes de que la máquina virtual de java pueda ejecutar una programa java, necesita localizar y cargar las clases que contienen esas clases en memoria. En un ambiente de ejecución tradicional, éste servicio es proporcionado por el sistema operativo, el cual carga el código del sistema en la forma de la plataforma particular.

El sistema operativo tiene acceso a todas las funciones de entrada/salida de bajo nivel y tiene un conjunto de lugares en el sistema de archivos en donde el busca programas o librerías de código ocultas. En los sistemas UNIX y PC ésta es alguna combinación de las configuraciones del PATH (Rutas) la cual especifica una lista de directorios para la búsqueda de archivos. En ambientes *mainframe* la misma función es proporcionada por lo que se conoce como LINKLIST.

En el ambiente de tiempo de ejecución de Java las cosas son un poco más complicadas por el hecho de que no todos los archivos de clase son cargados del mismo tipo de lugares o direcciones. En general las clases pueden ser divididas en tres categorías:

- **Clases que conforman el centro del API de Java**

  Estas son las clases integradas con la máquina virtual de java las cuales proporcionan acceso a redes, GUI y funciones de hilos. Son integradas con las implementaciones del JVM y son parte de la especificación de Java. Estas clases son altamente confiables y no están sujetas al mismo grado de examinación en tiempo de ejecución como las clases enviadas al JVM desde una fuente externa.

- **Clases instaladas en el sistema de archivos local**

  Mientras que no sean parte del conjunto central de clases de Java, éstas clases son asumidas como seguras, mientras que el usuario las halla instalado en algún punto de su máquina y presumiblemente aceptando los riesgos asociados. En muchos casos éstas clases son tratadas en la misma manera que aquellas que se encuentran en el corazón de Java.

- **Clases cargadas de otras fuentes**

  En un browser de Web éstas serian las clases que constituyen un applet cargada de un servidor de Web remoto. Éstas son las menos confiables de

todas ya que ellas están siendo enviadas al ambiente seguro del JVM desde fuentes potencialmente dañinas y por lo general sin el consentimiento del usuario por esta razón éstas clases deben estar sujetas a un alto grado de chequeo antes de ser puestas a disposición del usuario en el JVM.

Dado el rango diverso de fuentes posibles para archivos de clase y los diferentes requerimientos de chequeo con el JVM, es claro que diferentes mecanismos serán requeridos para localizar y cargar clases. El cargador de clases viene en varios motivos, cada uno responsable de localizar y cargar un tipo de archivo de clase.

Los usuarios pueden también implementar su propio *Cargador de Clases Principal* para realizar una serie de pruebas a los archivos de clases que son considerados como potencialmente seguros.

2) El Verificador de Archivos de Clases

Algunos de los archivos de clases cargados por el JVM vienen de fuentes no confiables, estos deben ser chequeados antes de ejecutarlos para procurar que no se amenace contra la integridad del JVM.

El *Verificador de Archivos de Clases* es invocado por el *Cargador de Clases* para realizar una serie de pruebas a los archivos de clases que son considerados como potencialmente inseguros.

Esas pruebas examinan todos los aspectos de un archivo de clase desde su tamaño y estructura hasta sus características de tiempo de ejecución. Solo cuando esos exámenes han sido pasados el archivo se hace disponible para su uso.

3) La Pila

La *pila* es un área de memoria usada por el JVM para guardar objetos de Java durante la ejecución de un programa. Exactamente cuántos objetos son guardados en la pila? Es implementación especifica del fabricante y esto agrega otro nivel de seguridad ya que esto significa que el *hacker* no puede tener idea de cómo el JVM representa los objetos en memoria. Esto hace aún más difícil montar un ataque que dependa del acceso a la memoria directamente.

Cuando un objeto no es usado más, el JVM lo marca para el *Recolector de Basura* y en algún punto de la memoria, la pila es liberada para su uso.

4) El Área de Clases

El área de clases es donde el JVM guarda la información especifica de clases tal como métodos y campos estáticos. Cuando una clase es cargada y ha sido verificada, el JVM crea una entrada en el área de clases para esa clase.

Generalmente el *área de clases* es simplemente una parte de la pila. En este caso las clases pueden ser también basura recopilada una vez no sean usadas.

Alternativamente el *área de clases* puede ser una parte separada de la memoria y requerirá de lógica adicional por parte del diseñador del JVM para limpiar las clases que no están en uso. Cuando un compilador JIT (*Just In Time, Justo a Tiempo*) está presente, el código nativo generado por los métodos de clases es también almacenado en el área de clases.

5) El Cargador de Métodos Nativos

Muchas de las clases del corazón de Java tales como las que representan los elementos de la GUI o características de redes, requieren implementaciones de código nativo para acceder a las funciones del sistema operativo. Los programadores pueden también implementar sus propios métodos nativos, asumiendo por supuesto, que no quieren que su código sea portable. Esos métodos nativos están compuestos por una cubierta de Java (la que especifica la

firma del método) y una implementación en código nativo (casi siempre una DLL o librería oculta).

Las clases del corazón de Java no son entorpecidas por el hecho de que ellas usan un código nativo; ellas son una parte de la implementación del JVM para un sistema operativo en particular. Las applets y aplicaciones, por otro lado, son de mayor uso si son portables, pero son portables si ellas evitan métodos nativos.

El método nativo almacenado es responsable de almacenar y cargar estas librerías ocultas en el JVM. Note que no es posible que el JVM realice alguna validación o verificación de código nativo e instalar tal código es exponerse a los riesgos asociados con correr programas no confiables en la máquina.

6) El Área de Métodos Nativos

Una vez que el código nativo ha sido cargado, es guardado en el área de métodos nativos para acceso rápido cuando se requiera.

7) El Gestor de Seguridad

Aún cuando el código no confiable ha sido verificado, está sujeto a restricciones de tiempo de ejecución. El gestor de seguridad es el responsable de hacer valer esas restricciones. En un browser de Web, el gestor de seguridad es proporcionado por el fabricante del mismo y es el componente del JVM quien evita que las applets lean o escriban al sistema de archivos, accedan a la red de forma insegura, hagan investigaciones a cerca del ambiente de ejecución, impriman y mucho más.

Por defecto en una implementación *stand-alone* del JVM (implementación usada para aplicaciones que no corren en un Browser de Web o aplicaciones locales) no hay gestor de seguridad, siempre y cuando no hayan mecanismos para cargar clases de una fuente no confiable. Sin embargo, es posible para un desarrollador de aplicaciones implementar su gestor de seguridad para hacer valer sus políticas de seguridad.

8) La Máquina de Ejecución

Es el corazón del JVM. Es el *procesador virtual* que ejecuta *bytecode*, realiza el manejo de la memoria, de hilos y las llamadas a métodos nativos.

9) Las Clases Confiables

Éstas clases son las que se integran como parte de la implementación del JVM. Incluye tanto todas las clases en paquetes que comienzan por "java." y "sun." como las proporcionadas por los vendedores usadas para implementar las partes de las clases centrales de plataforma especifica (como los componentes GUI). Ellas son generalmente grabadas en el sistema de archivos (usualmente en un archivo llamado *classes.zip*) pero puede ser considerado como parte del JVM ejecutable por el mismo.

10) El Compilador Justo a Tiempo (JIT)

Debido a que los bytecodes de Java son interpretados en tiempo de ejecución, en la máquina de ejecución, los programas de java se ejecutan más lentamente que el equivalente al código nativo de la plataforma en uso. Esto ocurre porque cada *bytecode* debe ser traducido a una o más instrucciones nativas siempre que son encontrados. El desempeño de Java es aún mejor que otros lenguajes interpretados, ya que las instrucciones de *bytecode* fueron diseñadas para ser de muy bajo nivel (las más simples instrucciones tienen una correlación uno a uno con las instrucciones de máquina nativa).

Sin embargo, SUN vio que había una necesidad de mejorar el desempeño de la ejecución de Java y hacerlo de manera que no comprometiera tampoco el objetivo de "escribir una vez, correr en cualquier lado" y no minara la seguridad del JVM.

Ya que las instrucciones de *bytecode* son finalmente traducidas al código nativo de la máquina, las formas principales de desempeño rápido serían hacer la traducción de los mismos tan rápido como sea posible y ejecutarlo en menor tiempo que se pueda.

Por otra parte, la seguridad y portabilidad de Java son dependientes del formato del archivo de clases y de los *bytecodes* que permiten al código correr en cualquier JVM y ser rigurosamente examinado para procurar que sea seguro antes de ejecutarse. Además, cualquier traducción debe ocurrir después de que un archivo de clases ha sido cargado y ejecutado.

Dos opciones se presentan por sí mismas:

1. Traducir todos los archivos de clases a código nativo tan pronto son cargados y verificados.

2. Traducir los archivos de clases, método por método a medida que se necesiten.

La primera opción parece bastante atractiva, pero puede que hallan métodos en los archivos de clases que jamas se ejecuten, entonces, el tiempo utilizado para traducirlos se desperdiciaría. La segunda opción fue la elegida por SUN. En este caso, la primera vez que es llamado un método, éste es traducido a código nativo, y luego guardado en el *área de clases.* La especificación de clases es actualizada de forma que las futuras llamadas al método corran el código nativo en vez del *bytecode* original.

Esto satisface nuestros requerimientos de que el b*ytecode* debe ser traducido en tan poco tiempo como sea necesario (una vez que el código sea ejecutado y no en el caso del código que no es ejecutado).

El proceso de traducir el *bytecode* a método nativo al vuelo es conocido como *Compilación Justo a Tiempo* (JIT) y es realizada por el compilador JIT. SUN proporciona una especificación para cómo y cuándo debe usarse un compilador JIT y deja en libertad de que los vendedores implementen sus propios compiladores JIT como ellos elijan.

El código compilado JIT se ejecuta de 10 a 50 veces más rápido que el *bytecode* regular sin quitar las características de portabilidad y seguridad.

1. MANIPULACIÓN DE LA MEMORIA POR PARTE DE JAVA

1.1. CÓMO JAVA MANEJA LA MEMORIA?

1.1.1. Hospedaje de Objetos: Cuando Java crea un objeto por medio de la palabra clave **new,** no existe ninguna palabra clave para deshacerse de ellos, entonces pareciera que Java está lleno de fugas de memoria, ya que nunca se llama a ningún método para liberar la memoria que está ocupada.

Sin embargo Java usa una técnica llamada *garbage collection* (recolección de basura) para detectar y liberar automáticamente los objetos en desuso, esto es, que nunca tendrá de preocuparse por liberar memoria o destruir objetos, pues el recolector de basura se encargará de ello.

Es una técnica que ha estado en uso por años en lenguajes como Lisp. Debido a que el intérprete de Java sabe qué objetos ha asignado, qué variables se refieren a qué objetos y cuáles objetos hacen referencias a otros objetos, puede saber también cuándo un objeto asignado ya no está referenciado por otro objeto o variable. Una vez que encuentra dicho objeto, lo destruye sin peligro. El recolector

de basura también puede detectar y destruir "ciclos" de objetos que se referencien entre sí, pero que no sean referenciados por ningún otro objeto.

Para realizar toda esta técnica la Maquina Virtual de Java cuenta con una pila como estructura de datos, en la cual se guardan los objetos de Java durante la ejecución del programa. La cantidad de objetos que pueden ser guardados en la pila es una información pertinente al fabricante de la Máquina Virtual de Java, lo cual agrega un nivel de seguridad debido a que un Hacker no tiene idea de como son representados los objetos en memoria. Esto hace más difícil montar un ataque que dependa del acceso a la memoria directamente. Cuando un objeto no es usado más, la máquina virtual lo marca para la colección de basura.

El recolector de basura de Java se ejecuta como un hilo de baja prioridad y hace casi todo su trabajo cuando no hay tareas en proceso. Por lo general, se ejecuta en momentos de inactividad, cuando el usuario introduce datos mediante el teclado o movimientos del ratón, la única ocasión en que el recolector de basura se ejecuta sin importar que algo de alta prioridad se esté llevando a cabo (en este caso, disminuye realmente la velocidad del sistema), es cuando el intérprete queda sin memoria. Esto no ocurre con frecuencia, ya que el hilo de baja prioridad se encarga de la limpieza en el plano secundario.

Este esquema puede parecer extremadamente lento y un desperdicio de memoria, sin embargo, los buenos recolectores de basura pueden ser muy eficaces y hacer

que la programación sea más sencilla y menos propensa a problemas, aunque en verdad nunca sean tan buenos como una bien escrita asignación y desasignación de memoria Pero en casi todos los programas existentes, un desarrollo rápido, la inexistencia de errores y un mantenimiento sencillo son características más importantes que la velocidad real o la eficiencia de la memoria.

Al final del documento se presenta el manual del programador de un sencillo programa que simula la técnica del Recolector de Basura.

1.1.2. Hospedaje de Clases: El área de clases es donde la JVM guarda información específica de clases tal como métodos y campos estáticos. Cuando una clase es cargada y ha sido verificada, la JVM crea una entrada en el área de clases para esa clase.

Generalmente el área de clases es simplemente una parte de la pila. En este caso, las clases pueden ser también basura recopilada una vez no sean usadas. Alternativamente, el área de clase puede ser una parte separada de la memoria y requerirá de lógica adicional en la parte del realizador de la JVM para limpiar las clases que no están en uso.

1.2. JAVA NO USA PUNTEROS

En Java la referencia y desreferencia de objetos se maneja automáticamente. Java no permite que el usuario manipule punteros o direcciones de memoria de ningún tipo:

- No acepta que usted lance referencias de objetos o arreglo a enteros o viceversa.
- No permite que usted haga aritmética de puntero.
- No deja que usted compute el tamaño en bytes de ningún tipo primitivo u objeto.

Existen dos razones para estas restricciones:

- Los punteros constituyen una fuente notoria de errores. Al eliminarlos se simplifica el lenguaje y se acaban muchos errores potenciales.

- Los punteros y la aritmética de punteros se podrían usar para evitar que ocurran verificaciones al tiempo de la ejecución y que funcionen los mecanismos de seguridad en Java. Eliminar punteros permite que Java proporcione garantías de seguridad.

2.3. EL SANDBOX DE JAVA

Los Applets cuentan (al menos en teoría) con un lugar muy seguro para correr un programa, este lugar es llamado ***el sandbox***, el cual fue diseñado teniendo en cuenta los siguientes objetivos:

- Evitar daños al browser del sistema ocasionados por la actualización de archivos o la ejecución de comandos del sistema.
- Evitar la recuperación de datos no deseados ya sea leyendo archivos o extrayendo información del medio.
- Evitar que el browser sea usado como una plataforma para el ataque a otros sistemas.
- Evitar la incorporación de clases Java de confianza en el browser ya que puede sobrepasarse del limite o ser alterado.

Este último objetivo es la clave a todos los otros. Porque el Administrador de Seguridad es, así mismo, una clase incorporada ya que si un atacante puede alterarlo o desviarlo, todo el control se pierde.

El Administrador de Seguridad es la parte del código local del browser, debido a que la implementación de las restricciones de cajón es responsabilidad de cada proveedor de browser. Sin embargo, todos ellos tienen los mismos objetivos, de tal

forma que el resultado es un conjunto de restricciones que son comunes para la mayoría los vendedores de estos productos:

- Ningún applet accede a un disco local.
- Muy limitado a la información del medio.
- El único host que un applet puede reconocer para una conexión de red es el primero desde el cual fue cargado.
- Ninguno enlaza al código local.
- Ninguno imprime.

Para crear aplicaciones cliente/servidor efectivas usando Java requiere que se le de al applet alguna libertad de la seguridad del sanbox.

El modelo de seguridad de Java es construido alrededor del concepto de un dominio de protección. El ***sandbox*** del applet es un dominio de protección con controles muy estrictos, en contraste, el ambiente de aplicaciones de Java es un dominio de protección sin ningún control, que el impuesto por el sistema operativo. Lo que se busca es un dominio de protección que se sitúe en medio de los dos.

El JDK 1.1 ofrece applets firmadas como forma de escape de las restricciones del sandbox. Las applets firmadas proporcionan el mecanismo para el dominio de protecciones deseado.

1.4. LA MÁQUINA VIRTUAL DE JAVA

Java es una programa tan robusto que hacer una estructura detallada de como quedaran los diferentes segmentos dentro de la memoria es algo complicado debido a que cada proveedor de software tendrá su forma particular para gestionar la memoria por medio de la Maquina Virtual de Java (JVM), es de ahí, que cualquier aplicación hecha con esta herramienta se hace confiable y a su vez segura hacía los ataques que pueda recibir por parte de intrusos.

En la figura 1, el área resaltada con puntos discontinuos muestra la Máquina Virtual de Java (JVM). Cada uno de los componentes que se muestran agrega un nivel de seguridad al sistema donde interactua dicha máquina.

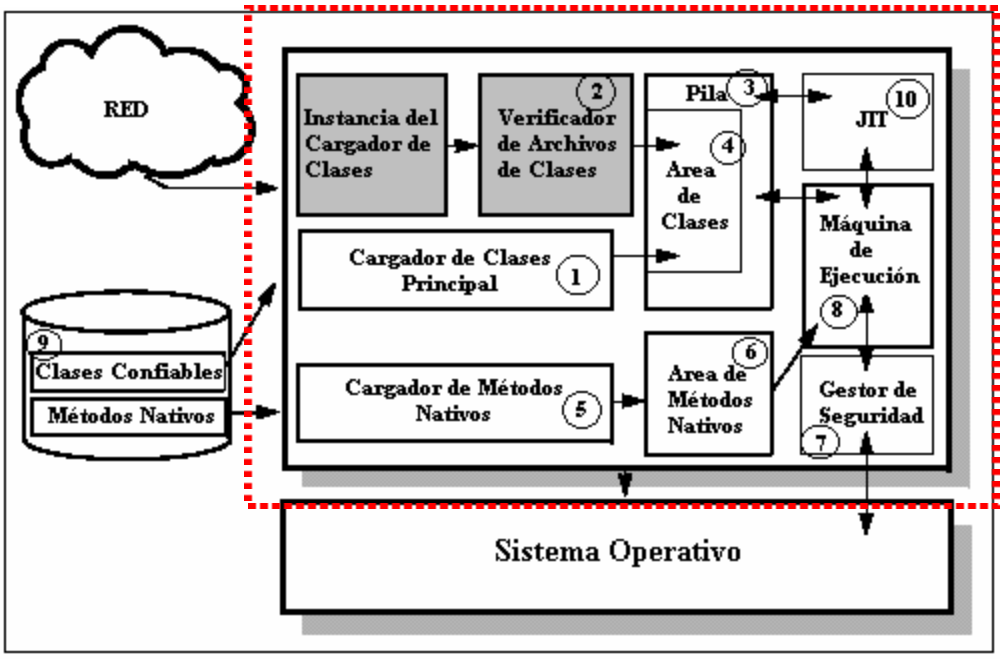


Figura 1. Aspectos de la Máquina Virtual de Java

Estos componentes son generalmente encontrados en Browser de Web.

1) El Cargador de Clases Principal

Antes de que la máquina virtual de java pueda ejecutar una programa java, necesita localizar y cargar las clases que contienen esas clases en memoria. En un ambiente de ejecución tradicional, éste servicio es proporcionado por el sistema operativo, el cual carga el código del sistema en la forma de la plataforma particular.

El sistema operativo tiene acceso a todas las funciones de entrada/salida de bajo nivel y tiene un conjunto de lugares en el sistema de archivos en donde el busca programas o librerías de código ocultas. En los sistemas UNIX y PC ésta es alguna combinación de las configuraciones del PATH (Rutas) la cual especifica una lista de directorios para la búsqueda de archivos. En ambientes *mainframe* la misma función es proporcionada por lo que se conoce como LINKLIST.

En el ambiente de tiempo de ejecución de Java las cosas son un poco más complicadas por el hecho de que no todos los archivos de clase son cargados del mismo tipo de lugares o direcciones. En general las clases pueden ser divididas en tres categorías:

- **Clases que conforman el centro del API de Java**

  Estas son las clases integradas con la máquina virtual de java las cuales proporcionan acceso a redes, GUI y funciones de hilos. Son integradas con las implementaciones del JVM y son parte de la especificación de Java. Estas clases son altamente confiables y no están sujetas al mismo grado de examinación en tiempo de ejecución como las clases enviadas al JVM desde una fuente externa.

- **Clases instaladas en el sistema de archivos local**

  Mientras que no sean parte del conjunto central de clases de Java, éstas clases son asumidas como seguras, mientras que el usuario las halla instalado en algún punto de su máquina y presumiblemente aceptando los riesgos asociados. En muchos casos éstas clases son tratadas en la misma manera que aquellas que se encuentran en el corazón de Java.

- **Clases cargadas de otras fuentes**

  En un browser de Web éstas serian las clases que constituyen un applet cargada de un servidor de Web remoto. Éstas son las menos confiables de

todas ya que ellas están siendo enviadas al ambiente seguro del JVM desde fuentes potencialmente dañinas y por lo general sin el consentimiento del usuario por esta razón éstas clases deben estar sujetas a un alto grado de chequeo antes de ser puestas a disposición del usuario en el JVM.

Dado el rango diverso de fuentes posibles para archivos de clase y los diferentes requerimientos de chequeo con el JVM, es claro que diferentes mecanismos serán requeridos para localizar y cargar clases. El cargador de clases viene en varios motivos, cada uno responsable de localizar y cargar un tipo de archivo de clase.

Los usuarios pueden también implementar su propio *Cargador de Clases Principal* para realizar una serie de pruebas a los archivos de clases que son considerados como potencialmente seguros.

2) El Verificador de Archivos de Clases

Algunos de los archivos de clases cargados por el JVM vienen de fuentes no confiables, estos deben ser chequeados antes de ejecutarlos para procurar que no se amenace contra la integridad del JVM.

El *Verificador de Archivos de Clases* es invocado por el *Cargador de Clases* para realizar una serie de pruebas a los archivos de clases que son considerados como potencialmente inseguros.

Esas pruebas examinan todos los aspectos de un archivo de clase desde su tamaño y estructura hasta sus características de tiempo de ejecución. Solo cuando esos exámenes han sido pasados el archivo se hace disponible para su uso.

3) La Pila

La *pila* es un área de memoria usada por el JVM para guardar objetos de Java durante la ejecución de un programa. Exactamente cuántos objetos son guardados en la pila? Es implementación especifica del fabricante y esto agrega otro nivel de seguridad ya que esto significa que el *hacker* no puede tener idea de cómo el JVM representa los objetos en memoria. Esto hace aún más difícil montar un ataque que dependa del acceso a la memoria directamente.

Cuando un objeto no es usado más, el JVM lo marca para el *Recolector de Basura* y en algún punto de la memoria, la pila es liberada para su uso.

4) El Área de Clases

El área de clases es donde el JVM guarda la información especifica de clases tal como métodos y campos estáticos. Cuando una clase es cargada y ha sido verificada, el JVM crea una entrada en el área de clases para esa clase.

Generalmente el *área de clases* es simplemente una parte de la pila. En este caso las clases pueden ser también basura recopilada una vez no sean usadas.

Alternativamente el *área de clases* puede ser una parte separada de la memoria y requerirá de lógica adicional por parte del diseñador del JVM para limpiar las clases que no están en uso. Cuando un compilador JIT (*Just In Time, Justo a Tiempo*) está presente, el código nativo generado por los métodos de clases es también almacenado en el área de clases.

5) El Cargador de Métodos Nativos

Muchas de las clases del corazón de Java tales como las que representan los elementos de la GUI o características de redes, requieren implementaciones de código nativo para acceder a las funciones del sistema operativo. Los programadores pueden también implementar sus propios métodos nativos, asumiendo por supuesto, que no quieren que su código sea portable. Esos métodos nativos están compuestos por una cubierta de Java (la que especifica la

firma del método) y una implementación en código nativo (casi siempre una DLL o librería oculta).

Las clases del corazón de Java no son entorpecidas por el hecho de que ellas usan un código nativo; ellas son una parte de la implementación del JVM para un sistema operativo en particular. Las applets y aplicaciones, por otro lado, son de mayor uso si son portables, pero son portables si ellas evitan métodos nativos.

El método nativo almacenado es responsable de almacenar y cargar estas librerías ocultas en el JVM. Note que no es posible que el JVM realice alguna validación o verificación de código nativo e instalar tal código es exponerse a los riesgos asociados con correr programas no confiables en la máquina.

6) El Área de Métodos Nativos

Una vez que el código nativo ha sido cargado, es guardado en el área de métodos nativos para acceso rápido cuando se requiera.

7) El Gestor de Seguridad

Aún cuando el código no confiable ha sido verificado, está sujeto a restricciones de tiempo de ejecución. El gestor de seguridad es el responsable de hacer valer esas restricciones. En un browser de Web, el gestor de seguridad es proporcionado por el fabricante del mismo y es el componente del JVM quien evita que las applets lean o escriban al sistema de archivos, accedan a la red de forma insegura, hagan investigaciones a cerca del ambiente de ejecución, impriman y mucho más.

Por defecto en una implementación *stand-alone* del JVM (implementación usada para aplicaciones que no corren en un Browser de Web o aplicaciones locales) no hay gestor de seguridad, siempre y cuando no hayan mecanismos para cargar clases de una fuente no confiable. Sin embargo, es posible para un desarrollador de aplicaciones implementar su gestor de seguridad para hacer valer sus políticas de seguridad.

8) La Máquina de Ejecución

Es el corazón del JVM. Es el *procesador virtual* que ejecuta *bytecode*, realiza el manejo de la memoria, de hilos y las llamadas a métodos nativos.

9) Las Clases Confiables

Éstas clases son las que se integran como parte de la implementación del JVM. Incluye tanto todas las clases en paquetes que comienzan por "java." y "sun." como las proporcionadas por los vendedores usadas para implementar las partes de las clases centrales de plataforma especifica (como los componentes GUI). Ellas son generalmente grabadas en el sistema de archivos (usualmente en un archivo llamado *classes.zip*) pero puede ser considerado como parte del JVM ejecutable por el mismo.

10) El Compilador Justo a Tiempo (JIT)

Debido a que los bytecodes de Java son interpretados en tiempo de ejecución, en la máquina de ejecución, los programas de java se ejecutan más lentamente que el equivalente al código nativo de la plataforma en uso. Esto ocurre porque cada *bytecode* debe ser traducido a una o más instrucciones nativas siempre que son encontrados. El desempeño de Java es aún mejor que otros lenguajes interpretados, ya que las instrucciones de *bytecode* fueron diseñadas para ser de muy bajo nivel (las más simples instrucciones tienen una correlación uno a uno con las instrucciones de máquina nativa).

Sin embargo, SUN vio que había una necesidad de mejorar el desempeño de la ejecución de Java y hacerlo de manera que no comprometiera tampoco el objetivo de "escribir una vez, correr en cualquier lado" y no minara la seguridad del JVM.

Ya que las instrucciones de *bytecode* son finalmente traducidas al código nativo de la máquina, las formas principales de desempeño rápido serían hacer la traducción de los mismos tan rápido como sea posible y ejecutarlo en menor tiempo que se pueda.

Por otra parte, la seguridad y portabilidad de Java son dependientes del formato del archivo de clases y de los *bytecodes* que permiten al código correr en cualquier JVM y ser rigurosamente examinado para procurar que sea seguro antes de ejecutarse. Además, cualquier traducción debe ocurrir después de que un archivo de clases ha sido cargado y ejecutado.

Dos opciones se presentan por sí mismas:

2. Traducir todos los archivos de clases a código nativo tan pronto son cargados y verificados.

3. Traducir los archivos de clases, método por método a medida que se necesiten.

La primera opción parece bastante atractiva, pero puede que hallan métodos en los archivos de clases que jamas se ejecuten, entonces, el tiempo utilizado para traducirlos se desperdiciaría. La segunda opción fue la elegida por SUN. En este caso, la primera vez que es llamado un método, éste es traducido a código nativo, y luego guardado en el *área de clases.* La especificación de clases es actualizada de forma que las futuras llamadas al método corran el código nativo en vez del *bytecode* original.

Esto satisface nuestros requerimientos de que el b*ytecode* debe ser traducido en tan poco tiempo como sea necesario (una vez que el código sea ejecutado y no en el caso del código que no es ejecutado).

El proceso de traducir el *bytecode* a método nativo al vuelo es conocido como *Compilación Justo a Tiempo* (JIT) y es realizada por el compilador JIT. SUN proporciona una especificación para cómo y cuándo debe usarse un compilador JIT y deja en libertad de que los vendedores implementen sus propios compiladores JIT como ellos elijan.

El código compilado JIT se ejecuta de 10 a 50 veces más rápido que el *bytecode* regular sin quitar las características de portabilidad y seguridad.

2. INTERFACE DE PROGRAMACIÓN DE APLICACIONES (API) DE JAVA ENCARGADA PARA PROVEER SEGURIDAD

2.1. EL PAQUETE java.security

2.1.1. Interfaces

- **Interface java.security.Certificate**

Interface pública **Certificate.**

Esta es una interface de métodos abstractos para el manejo de identidad de certificados. Una identidad de certificados es garantizada por un principal y una clave pública es garantizada por otro principal. (Un principal representa una entidad tal como un usuario individual o un grupo.)

En particular, la interface esta proyectada para ser una abstracción común para constructores que tienen formatos diferentes pero importantes usos comunes. Por ejemplo, tipos diferentes de certificados, tal como certificados X.509 y certificados

PGP, compartición general de la funcionalidad de certificados (la necesidad de codificar y decodificar certificados) y algunos tipos de información, tales como una clave pública, el principal quién seria la clave, y el garante garantizando que la clave pública es aquella del principal especificado. Así una implementación de certificados X.509 y una implementación de certificados PGP pueden ambas utilizar la interface *Certificate*, aun cuando sus formatos, tipos adicionales y alcances de información almacenados sean diferentes.

Importante

La interface es útil para catalogar y agrupar objetos ocultando usos comunes de seguridad. No tiene semántica alguna propia. En particular, un objeto *Certificate* no construye sentencia alguna como la de validez que es obligatoria. Lo correspondiente a la implementación de esta interface de aplicación es verificar la certificación y su misma satisfacción de su validez.

Métodos de la Clase Certificate

- ✓ `decode(InputStream)`

Decodifica un certificado desde un flujo de entrada.

Definición

`public abstract void decode (InputStream stream) arroja KeyException, IOException`

Parámetros:

`stream` – esta es la salida clasificada desde el cual se busca el dato a ser decodificado.

Excepciones que arroja:

`KeyException` - Si el certificado no es bien inicializado, o si el dato es perdido.

`IOException` - Si un excepción ocurre mientras se trata de entrar el certificado decodificado desde la entrada clasificada.

- ✓ `encode(OutputStream)`

Codifica el certificado hacia un flujo de entrada en un formato que puede ser decodificado por el método decode.

Definición

`public abstract void encode (OutputStream stream) throws KeyException, IOException`

Parámetros:

`stream` – esta es la salida clasifica para codificar el certificado.

Excepciones que arroja:

`KeyException -` Si el certificado no es bien inicializado, o si el dato es perdido.

`IOException -` Si un excepción clasificada ocurre mientras esta tratando de salir el certificado codificado a la salida clasificada.

✓ `getFormat()`

Retorna el nombre del formato codificado.

Este es usado como un indicio para hallar un parser apropiado. Este podría ser "X.509","PGP", etc. Este formado es producido y entendido por los métodos encode y decode.

Definición:

```
public abstract String getFormat()
```

✓ `getGuarantor()`

Retorna el generador del certificado, que es, la principal garantía que la clave pública es asociada con este certificado, es decir el principal se asocia con este certificado. Los certificados X.509 por ejemplo, usualmente son generados por una Autoridad de Certifcación (tal como el Servicio Postal de USA o Verisign, Inc)

Definición:

```
public abstract Principal getGuarantor()
```

✓ `getPrincipal()`

Retorna el principal del par de claves principales siendo garanrtizado por el generador del Certificado.

Definición:

```
public abstract Principal getPrincipal()
```

✓ `getPublicKey()`

Retorna la clave pública del par de la claves principales siendo garantiza por el generador del Certificado. Es decir, retorna la clave pública que el certificado garantiza que corresponde a un principal en particular

Definición:

```
public abstract PublicKey getPublicKey()
```

✓ `toString(boolean)`

Retorna una cadena que representa el contenido del certificado.

Definición:

```
public abstract String toString(boolean detailed)
```

Parámetros:

`detailed` – si se da o no información detallada acerca del certificado.

- **Interface java.security.Key**

Key es una Interface pública.

Subclase de la clase `Serializable`.

La interface *Key* es la interface de mayor nivel para toda las claves. En esta se define la funcionalidad compartida por todas los objetos de clave. Todas las claves tienen tres características:

1. *Un algoritmo*

Este es el algoritmo adecuado para esa clave. Este algoritmo adecuado es usualmente de encripción o de operación asimétrica (tal como, el DSA o RSA), la clave trabajará con estos algoritmos y otros relacionales (tal como el MD5 con RSA, SHA-1 con RSA, el DSA en su estado natural, etc.) el nombre del algoritmo de una sola clave se obtiene usando el método *getAlgorithm*.

2. *Una forma decodificada*

Esta es una forma decodificada externa usada por la clave cuando una representación estándar de la clave es necesitada fuera de la JVM (Java Virtual Machine), como trasmitiendo la clave a algunos otros grupos. La clave es

decodificada según un formato estándar (tal como X.509 o PKCS#8), y es retornada usando el método *getEncoded.*

3. Un formato

Este es el nombre del formato de la clave decodificada. Este es retornado por el método *getFormat.*

Las claves son generalmente obtenidas directamente por el generador de claves, certificados o diversidad de identidad de clases usando el manejador de clases. No hay suministrado en este orden por el parseo de claves decodificadas y certificados.

Métodos de la Clase java.security.Key

✓ `getAlgorithm()`

Retorna el nombre estándar del algoritmo para esta clave. Por ejemplo, "DSA" indicaría que esta clave es una clave DSA. Note que este método puede retornar nulo (null), cuando el algoritmo para esta clave es desconocido.

Definición:

```
public abstract String getAlgorithm()
```

- ✓ `getEncoded()`

Retorna la clave decodificada o nulo si la clave no soporta decodificación.

Definición:

```
public abstract byte[] getEncoded()
```

- ✓ `getFormat()`

Retorna el formato usado para decodificar la clave o nulo (null) si la clave no soporta decodificación.

Definición:

```
public abstract String getFormat()
```

- ➢ **Interface java.security.Principal**

Principal es una Interface pública.

Esta interface representa un principal. Un principal puede ser un individuo, una corporación, un hilo de programa; cualquier cosa que pueda tener una identidad.

Métodos de la Clase java.security.Principal

✓ `equals(Object)`

Compara este principal con el objeto especificado. Retorna *true* (verdadero) si el objeto pasó acompañando al principal representado por la implementación de esta interface.

Definición:

```
public abstract boolean equals(Object another)
```

Parámetros:

`another` - con el cual es comparado el principal.

Retorna:

Verdadero (true) si el principal que ha pasado es el mismo que aquel que se encapsuló por el este principal, falso (false) si es algo distinto.

Predomina dentro de:

La clase `Objet`

✓ `getName()`

Retorna el nombre de este principal.

Definición:

```
public abstract String getName()
```

✓ `hashCode()`

Retorna el código hash para este principal.

Definición:

```
public abstract int hashCode()
```

✓ `toString()`

Retorna la representación en cadena de este principal.

Definición:

```
public abstract String toString()
```

➢ **Interface java.security.PrivateKey**

PrivateKey es una Interface pública.

Es subclase de la clase **Key**

Es una clave privada. Esta interface no contiene métodos ni constantes. Esta simplemente sirve para agrupar (y proveer tipo de seguridad para) todas las interfaces de clave privada. Nota: Las interfaces especializadas de clave privada se extienden de esta interface. Vea, por ejemplo, la interface `DSAPrivateKey` en java.security.interfaces.

- **Interface java.security.PublicKey**

PublicKey es una Interface pública

Es subclase de la clase **Key**

Es una clave pública. Esta interface no contiene métodos o constantes. Está simplemente para agrupar (provee tipos de seguridad para) todas las interfaces de clave pública. Nota: Las interfaces especializadas de clave pública se extienden de esta interface. Ver, por ejemplo, la interfaz DSAPublicKey en java.security.interfaces.

2.1.2. Clases

- **La Clase java.security.DigestInputStream**

```
java.lang.Object
  |
  +----java.io.InputStream
         |
         +----java.io.FilterInputStream
                |
                +----java.security.DigestInputStream
```

DigestInputStream es una Clase pública.

Subclase de la clase **java.io.FilterInputStream**.

Es un flujo transparente que actualiza los mensajes resumidos asociados, usando los bits que van directamente al flujo.

Para completar la computación del mensaje resumido, se llama uno de los métodos *digest* en el mensaje resumido asociado, después llamas a uno de los métodos *read* de flujo resumido de entrada .

Es posible que esta clase coloque el flujo en **`on/off`** . Cuando esta está en **`on`**, una llamada a *read* conllevaría a una actualización sobre el resumen de mensaje.

Pero cuando está en **`off,`** el resumen de mensaje no es actualizado. Por defecto esta clase se encuentra en **on** para actuar sobre el flujo.

Hay que hacer notar que los objetos de resumen pueden computar un solo resumen (vea *MessageDigest*), de manera que dentro del orden de computo de resúmenes intermedios, el que hace el llamado debería mantener en reserva un encabezado del objeto de resumen, y hacerlo identico por cada resumen a ser computado, dejando el resumen original inalterado.

Variable de la Clase DigestInputStream

- ✓ `digest`

Este variable guarda el resumen de mensaje asociado con este flujo.

Definición:

```
protected MessageDigest digest
```

Constructor de la Clase DigestInputStream

- ✓ `DigestInputStream(InputStream, MessageDigest)`

Este constructor crea un resumen de flujo de entrada, usando el flujo de entrada y el resumen de mensaje especificados.

Definición:

```
public DigestInputStream(InputStream stream, MessageDigest digest)
```

Parámetros:

`stream` – el flujo de entrada.

`digest` – el resumen de mensaje asociado con este flujo.

Métodos de la Clase DigestInputStream

✓ `getMessageDigest()`

Este método retorna el resumen de mensaje asociado con este flujo.

Definición:

```
public MessageDigest getMessageDigest()
```

✓ `setMessageDigest(MessageDigest)`

Este método asocia el resumen de mensaje especificado con el flujo

Definición:

```
public void setMessageDigest (MessageDigest digest)
```

Parámetros:

`digest` – argumento que guarda el mensaje resumido a ser asociado con este flujo.

✓ `read()`

Lee un byte, y actualiza los resúmenes de mensajes (si la función de resumen esta en **on**). Es decir, este método lee un byte desde el flujo de entrada, bloqueando el siguiente byte hasta que el anterior a ese sea leído. Si la función de resumen está en **on**, este método entonces llamará a ***update*** *(método de la clase MessageDigest*)sobre el resumen de mensaje asociado con este flujo, pasándole el byte leído.

Definición:

```
public int read() throws IOException
```

Retorna:

El byte leído.

Excepciones que arroja:

`IOException,` Si un error de Entrada/Salida ocurre.

✓ `read(byte, int, int)`

Lee un byte dentro de un arreglo, y actualiza los resúmenes de mensajes (si la función de resumen está en **on**). Esto es, este método lee en lo posible **len** bytes del flujo de entrada dentro del arreglo **b**, iniciando en la posición (offset) **off.** Este método efectúa un bloqueo hasta que el dato es leído. Si la función está en **on,** este método entonces llamará a ***update*** sobre el resumen de mensaje asociado con este flujo, pasándole el dato.

Definición:

```
public int read(byte b[], int off, int len) throws IOException
```

Parámetros:

`b` – Arreglo dentro del cual el dato es leído.

`off` – inicio del offset dentro de b de donde el dato deberá ser localizado.

`len` – Máximo número de bytes a ser leídos desde el flujo de entrada dentro de `b`, iniciando el offset en `off`.

Retorna:

El número actual de bytes leídos. Este es menor de *len* si el final del flujo es alcanzado antes de leer los **len** bytes. Retorna *–1* si no se leyó byte alguno porque el final del flujo tuvo que ser ya alcanzado cuando se hizo la llamada.

Excepciones que arroja:

`IOException,` Si un error de Entrada/Salida ocurre.

✓ `on(boolean)`

Coloca la función de resumen en **on/off.** Por defecto está en **on.** Cuando esta está en **on,** una llamada a *read* conlleva a una actualización sobre el resumen de mensaje. Pero, cuando está en **off,** el resumen de mensaje no es actualizado.

Definición:

```
public void on (boolean on)
```

Parámetros:

`on` - si es true (verdadero) coloca la función de resumen en `on`**,** si es false (falsa) la coloca en `off`.

✓ `ToString()`

Imprime la representación de cadena de este resumen de flujo de entrada y sus objetos de resumen de mensaje asociados.

Definición:

```
public String toString()
```

- **La Clase java.security. DigestOutputStream**

```
java.lang.Object
  |
  +----java.io.OutputStream
         |
         +----java.io.FilterOutputStream
                |
                +----java.security.DigestOutputStream
```

DigestOutputStream es una clase pública.

Subclase de la clase **java.io.FilterOutputStream.**

Es un flujo transparente que actualiza los mensajes resumidos asociados, usando los bits que van directamente al flujo.

Para completar la computación del mensaje resumido, se llama uno de los métodos *digest* sobre el mensaje resumido asociado, después llamas a uno de los métodos *write* de resumen de flujo de salida.

Es posible que esta clase coloque el flujo en **on/off.** Cuando este está en **on**, una llamada a *write* conllevaría a una actualización sobre el resumen de mensaje. Pero cuando está en **off,** el resumen de mensaje no es actualizado. Por defecto este de encuentra en **on** para actuar sobre el flujo.

Variable de la Clase DigestOutputStream

✓ `digest`

Este variable guarda el resumen de mensaje asociado con este flujo.

Definición:

```
protected MessageDigest digest
```

Constructor de la Clase DigestOutputStream

✓ `DigestOutputStream(OutputStream, MessageDigest)`

Este constructor crea un resumen de flujo de salida, usando el flujo de salida y el resumen de mensaje especificados.

Definición:

```
public DigestOutputStream (OutputStream stream, MessageDigest digest)
```

Parámetros:

`stream` – el flujo de salida.

`digest` – el resumen de mensaje asociado con este flujo.

Métodos de la Clase DigestOutputStream

✓ `getMessageDigest`

Este método retorna el resumen de mensaje asociado con este flujo.

Definición:

```
public MessageDigest getMessageDigest()
```

✓ `setMessageDigest`

Este método asocia el resumen de mensaje especificado con el flujo

Definición:

```
public void setMessageDigest (MessageDigest digest)
```

Parámetros:

`digest` – argumento que guarda el mensaje resumido a ser asociado con este flujo.

✓ `write`

Actualiza el resumen de mensaje (si la función de resumen está en **on**) usando el byte especificado y en algunos casos escribe el byte al flujo de salida. Esto es, si

la función de resumen está en **on**, este método entonces llama a ***update*** *(método de la clase MessageDigest*)sobre el resumen de mensaje asociado con este flujo, pasándole el byte *b*. Este método entonces escribe el byte al flujo de salida, haciendo un bloqueo hasta que el byte es escrito.

Definición:

```
public void write(int b) throws IOException
```

Parámetros:

`b` – Que es el byte a ser usado para actualizar y escribir al flujo de salida.

Excepciones que arroja:

`IOException,` Si un error de Entrada/Salida ocurre.

✓ `write`

Actualiza el resumen de mensaje (si la función de resumen está en **on**) usando la subcadena especificada, y en cualquier caso escribe la subcadena al flujo de salida . Esto es, si la función de resumen está en **on,** este método entonces llama al método ***update*** sobre el resumen de mensaje asociado con este flujo, pasándole las especificaciones de la subcadena. Este método entonces escribe la subcadena de bytes al flujo de salida, haciendo un bloqueo hasta que los bytes son escritos.

Definición:

```
public void write(byte b[], int off, int len) throws IOException
```

Parámetros:

`b –` Arreglo que contiene el subarreglo a ser usado para actualizar y escribir al flujo de salida.

`off –` el offset dentro de **b** del primer byte a ser actualizado y escrito.

`len –` el número de bytes de datos a ser actualizados y escritos desde b, iniciándose en el offset **off.**

Excepciones que arroja:

`IOException,` Si un error de Entrada/Salida ocurre.

✓ on

Coloca la función de resumen en **on/off**. Por defecto está en **on.** Cuando esta está en **on,** una llamada a *write* conlleva a una actualización sobre el resumen de mensaje. Pero, cuando está en **off,** el resumen de mensaje no es actualizado.

Definición:

```
public void on (boolean on)
```

Parámetros:

`on -` si es true (verdadero) coloca la función de resumen en **on,** si es false (falsa) la coloca en **off**.

✓ `toString`

Imprime la representación de cadena de este resumen de flujo de salida y sus objetos de resumen de mensaje asociados.

Definición:

```
public String toString()
```

➢ **La Clase java.security.Identity**

```
java.lang.Object
  |
  +----java.security.Identity
```

Identity es una clase pública abstracta. Subclase de la clase **Object.**

Esta clase representa identidades: objetos del mundo real tal como personas, compañías o organizaciones cuyas identidades pueden ser autenticadas usando sus claves públicas. Las identidades también pueden ser construres abstractos o concretos, tal como hilos de demonio o tarjetas inteligentes.

Todos los objetos *Identity* tienen un nombre y una clave pública. Los nombres son invariables. Las identidades también pueden tener un alcance. Es decir, si una Identidad es especificada para tener un alcance en particular, entonces el nombre y la clave pública de la Identidad son únicos dentro de ese alcance.

Una Identidad también puede tener un conjunto de certificados (todos certificando sus propias claves públicas). El nombre Principal especificado en estos certificados no necesita ser el mismo, solamente la clave.

Una Identidad puede ser subclase, incluir postales y dirección de e-mail, números telefónicos, imágenes de rostros y logotipos, etc.

Constructores de la Clase java.security.Identity

✓ `Identy()`

Constructor para Serialización solamente.

Definición:

```
protected Identity()
```

✓ `Identy (String, IdentityScope)`

Constructor de una identidad con el nombre y el ámbito especificado.

Definición:

```
public Identity(String name, IdentityScope scope) throws
                                                    KeyManagementException
```

Parámetros:

`name` - El nombre de la identidad

`scope` - El ámbito de la identidad.

Excepciones que arroja:

`KeyManagementException,` Si ya existe una identidad con el mismo nombre en el ámbito

✓ `Identity (String)`

Constructor de una identidad con el nombre y ámbito especificado.

Definición:

```
public Identity(String name)
```

Parámetros:

`name` – El nombre de la identidad.

Métodos de la Clase java.security.Identity

✓ `addCertificate(Certificate)`

Añade un certificado para la identidad. Si la identidad tiene una clave pública, la clave pública en el certificado debe ser la misma, y si la identidad no tiene una clave pública, la clave pública de la identidad es preparada para que sea especificada en el certificado.

Declaración:

```
public void addCertificate(Certificate certificate) throws
KeyManagementException
```

Parámetros:

`certificate` – el certificado a ser añadido.

Excepciones que arroja:

`KeyManagementException,` Si el certificado no es valido, si la clave pública en el certificado genera conflictos con esta clave pública de la identidad, o ocurre otra excepción.

✓ `certificates()`

Retorna una copia de todos los certificados para esta identidad.

Declaración:

```
public Certificate[] certificates().
```

✓ `equals(Object)`

Analiza la igualdad entre el objeto especificado y esta identidad. Este primero prueba a ver si las entidades actualmente hacen referencia al mismo objeto, en tal caso este retorna **true.** Luego, este chequea a ver si las entidades tienen el mismo nombre y el mismo ámbito. Si ocurre esto, el método retorna **true.** De otro modo, este llama a **identityEquals**, cuyas subclases se extralimitaran.

Declaración:

```
public final boolean equals(Object identity)
```

Parámetros:

`identity` - El objeto a probar para igualdad con esta identidad.

Retorna:

Verdadero si el objeto es considerado igual, falso en caso contrario.

Preválese sobre:

`equals` en la clase `Object`

- ✓ `getInfo()`

Retorna la información general previamente especificada por esta identidad.

Declaración:

```
public String getInfo()
```

- ✓ `getName()`

Retorna el nombre de esta identidad.

Declaración

```
public final String getName()
```

- ✓ `getPublicKey()`

Retorna la clave pública de esta identidad.

Declaración

```
public PublicKey getPublicKey()
```

✓ `getScope()`

Retorna el ámbito de esta identidad

Declaración:

```
public final IdentityScope getScope()
```

✓ `hashCode()`

Retorna un código hash para esta identidad.

Declaración:

```
public int hashCode()
```

Prevalese sobre:

`hashCode` en la clase `Object`

✓ `identityEquals(Identity)`

Prueba la igualdad entre la identidad especificada y esta identidad. Este método deberá predominar sobre subclases que prueben ser iguales. El comportamiento por defecto es retornar verdadero si los nombres y las claves públicas son iguales.

Declaración:

```
protected boolean identityEquals(Identity identity)
```

Parámetros:

`identity` - la identidad a probar por igualdad con esta identidad.

Retorna:

Verdadero si las identidades son consideradas iguales, de lo contrario retornará falso.

✓ `removeCertificate`

Remueve un certificado desde la identidad.

Declaración:

```
public void removeCertificate(Certificate certificate) throws
KeyManagementException
```

Parámetros:

`certificate` - el certificado a ser borrado.

Excepciones que arroja:

`KeyManagementException,` Si el certificado está desaparecido, o si ocurre otra excepción.

✓ `setInfo`

Especifica una cadena para información general de esta identidad.

Declaración:

```
public void setInfo(String info)
```

Parámetros:

`info` - la cadena de información.

✓ `setPublicKey`

Muestra la clave pública de esta identidad. La clave antigua y todos los certificados de esta identidad son borrados por esta operación.

Declaración:

```
public void setPublicKey(PublicKey key) throws
                                        KeyManagementException
```

Parámetros:

`key` - la clave pública para esta identidad.

Excepciones que arroja:

`KeyManagementException,` Si otra identidad dentro del ámbito de esta identidad tiene la misma clave pública, o si otra excepción ocurre.

✓ `toString()`

Retorna una cadena corta que describe esta identidad, su nombre y su ámbito eficazmente (de haberlo).

Declaración:

```
public String toString()
```

Retorna:

Información a cerca de la identidad, tal como su nombre y su ámbito (de haberlo).

Preválese sobre:

`toString` en la clase `Object`

- ✓ `toString(boolean)`

Retorna una representación de cadena de esta identidad, con más detalles opcionales que los suministrados por el método toString sin ningún argumento.

Declaración:

```
public String toString(boolean detailed)
```

Parámetros:

`detailed` - de alguna manera suministra información detallada.

Retorna:

Información acerca de esta identidad. Si detailed es verdadero, entonces este método retorna más información que la suministrada por el método toString sin ningún argumento.

- **La Clase java.security.IdentityScope**

```
java.lang.Object
  |
  +----java.security.Identity
        |
        +----java.security.IdentityScope
```

IdentityScope es una clase pública abstracta.

Subclase de la clase **java.security.Identity**

Esta clase representa un ámbito para identidades. Esta por si misma es una identidad, y por tanto tiene un nombre y puede tener un ámbito. Esta también puede tener opcionalmente una clave pública y certificados asociados.

Una clase IdentityScope puede contener Objetos Identity de todos los tipos, incluyendo Firmadores (Signers). Todos los tipos de objetos Identity pueden ser recuperados, añadidos, y borrados usando los mismos métodos. Note que esto es posible, y de hecho esperado, que diferentes tipos de ámbitos de identidades apliquen diferentes políticas para sus distintas operaciones sobre los distintos tipos de Identidades.

Hay un exacto mapeado entre las claves y las identidades, y allí puede solo haber una copia de una clave por ámbito. Por ejemplo, suponga que Acme Software Inc, es un software de publicidad conocido por un usuario. Suponga que este es una identidad, por tanto, este tiene una clave pública, y un conjunto de certificados

asociados. Este es nombrado dentro del ámbito usando el nombre "Acme Software". Otra identidad no se nombró dentro del ámbito con la misma clave pública. Por supuesto, nadie tiene el mismo nombre también.

Constructores de la Clase java.security.IdentityScope

✓ `IdentityScope()`

Este constructor es usado para Serialización solamente y debe no ser usado por subclases.

Declaración:

```
protected IdentityScope()
```

✓ `IdentityScope(String)`

Construye una nuevo ámbito de identidad con el nombre especificado.

Declaración:

```
public IdentityScope(String name)
```

Parámetros:

`name` - el nombre del ámbito.

✓ `IdentityScope(String, IdentityScope)`

Construye un nuevo ámbito de identidad con el nombre y ámbito especificado.

Declaración:

```
public IdentityScope(String name, IdentityScope scope) throws
                                                KeyManagementException
```

Parámetros:

`name` - Nombre del ámbito.

`scope` - el ámbito para el nuevo ámbito de identidad.

Excepciones que arroja:

`KeyManagementException`, Si ya existe una identidad con el mismo nombre dentro del ámbito.

Métodos de la Clase java.security.IdentityScope

✓ `addIdentity(Identity)`

Añade una identidad a este ámbito de identidad

Declaración:

```
public abstract void addIdentity(Identity identity) throws
KeyManagementException
```

Parámetros:

`identity` - la identidad a ser añadida.

Excepciones que arroja:

`KeyManagementException,` Si la identidad no es valida, la ocurrencia de un conflicto de nombre, otra identidad tiene la misma clave pública que la identidad a ser añadida, o otra ocurrencia de excepción.

✓ `getIdentity(Principal)`

Recupera la identidad cuyo nombre es el mismo que del principal especificado. (Nota: Implementada Identidad Principal.)

Declaración:

```
public Identity getIdentity(Principal principal)
```

Parámetros:

`principal -` el principal correspondiente a la identidad a ser recuperada.

Retorna:

La identidad cuyo nombre es la misma que la del principal, o nulo si no hay identidades del mismo nombre dentro del ámbito.

✓ `getIdentity(PublicKey)`

Recupera la identidad con la clave pública especificada.

Declaración:

```
public abstract Identity getIdentity(PublicKey key)
```

Parámetros:

`key -` la clave pública para la identidad a ser retornada.

Retorna:

La identidad con la clave dada, o nulo si no hay identidades dentro de este ámbito con esa clave.

✓ `getIdentity(String)`

Retorna la identidad dentro de este ámbito con el nombre especificado (de haberlo).

Declaración:

```
public abstract Identity getIdentity(String name)
```

Parámetros:

`name -` el nombre de la identidad a ser recuperada.

Retorna:

El nombre de la identidad (name), o nulo si no hay identidades nombradas con name dentro de este ámbito.

✓ `getSystemScope()`

Retorna el "ámbito de identidad del sistema".

Declaración:

```
public static IdentityScope getSystemScope()
```

✓ `identities()`

Retorna una enumeración de todas las identidades dentro de este ámbito de identidad.

Declaración:

```
public abstract Enumeration identities()
```

✓ `removeIdentity(Identity)`

Borra una identidad desde este ámbito de Identidad.

Declaración:

```
public abstract void removeIdentity(Identity identity) throws
                                          KeyManagementException
```

Parámetros:

`identity` - la identidad a ser borrada.

Excepciones que arroja:

`KeyManagementException ;` si la identidad esta perdida, o si ocurre otra excepción.

✓ `setSystemScope(IdentityScope)`

Muestra el ámbito de identidad del sistema (system's identity scope).

Declaración:

```
protected static void setSystemScope(IdentityScope scope)
```

Parámetros:

`scope` - el ámbito a determinar.

✓ `size()`

Retorna el número de identidades dentro del ámbito de identidad.

Declaración:

```
public abstract int size()
```

✓ `toString()`

Retorna una representación de cadena de este ámbito de identidad, incluyendo su nombre, su nombre de ámbito, y el número de identidades en este ámbito de identidad.

Declaración:

```
public String toString()
```

Predomina sobre:

`toString` dentro de la clase `Identity`

➢ La Clase java.security.KeyPair

java.lang.Object
  |
  +----java.security.KeyPair

KeyPair es una clase pública declarada como final(no modificable)

Se trata simplemente de un contenedor para una pareja de claves (una pública y una privada). No impone ninguna seguridad, y , cuando es inicializada debe ser tratada semejante a PrivateKey. Tiene dos métodos públicos, uno para devolver la clave pública y el otro para devolver la clave privada.

Constructores de la Clase java.security.KeyPair

✓ `KeyPair(PublicKey, PrivateKey)`

Construye una clave con la clave pública y clave privada especificada.

Declaración:

```
public KeyPair(PublicKey publicKey, PrivateKey privateKey)
```

Parámetros:

`publicKey` - clave pública.

`privateKey` - clave privada.

Métodos de la Clase java.security.KeyPair

✓ `getPrivate()`

Retorna la clave privada del par de claves.

Declaración:

```
public PrivateKey getPrivate()
```

✓ `getPublic()`

Retorna la clave pública del par de claves.

Declaración:

```
public PublicKey getPublic()
```

- **La Clase java.security.KeyPairGenerator**

java.lang.Object
  |
  +----java.security.KeyPairGenerator

KeyPairGenerator es una clase pública abstracta.

Es subclase de la clase *Objet*

La clase KeyPairGenerator es usada para generar un par de claves, pública y privada. Los generadores de claves son construidos usando los métodos fabricados `getInstance` (métodos estáticos que retornan instancias de una clase dada).

La generación de claves es una de las áreas que no siempre permite independencia del algoritmo. Por ejemplo, es posible generar una pareja de claves DSA especificando los parámetros de la familia de claves (p, q y g), pero esto no puede hacerse para una pareja de claves RSA. Es decir, esos parámetros pueden aplicarse a DSA pero no a RSA.

Por lo tanto, habrá dos formas de generar una pareja de claves: una independiente del algoritmo, y otra dependiente del mismo. La única diferencia entre las dos es la inicialización del objeto. La segunda se realiza a través de interfaces estándar específicas para el algoritmo en cuestión.

Todos los generadores de claves comparten dos aspectos: el de la "fuerza" de la clave y una fuente de aleatoriedad. El concepto de "fuerza" de la clave es compartido por todos los algoritmos de clave pública, aunque se interpreta de forma diferente en cada uno de ellos. El método *initialize* perteneciente a esta clase KeyPairGenerator, permite inicializar el generador de claves, especificando la fortaleza de la clave deseada por el usuario.

Ya que otros parámetros no son especificados cuando llamas este método *initialize* de algoritmo independiente, otros valores, tal como parámetros de algoritmos, exponente público, etc., son por defecto valores estándar.

Es algunas veces conveniente inicializar un objeto generador de un para de claves usando un algoritmo de semántica especifico. Por ejemplo, pueda que usted quiera inicializar un generador de claves DSA dado un conjunto de parámetros p, q y g, o un generador de claves RSA dado un exponente público.

Esta es la terminación completamente de la interface estándar del algoritmo especifico. Claro que llamando al método initialize del algoritmo independiente, el generador del par de claves se añade a una interface del algoritmo especificado así que uno de sus métodos de inicialización de parámetros especializados puede ser llamado. Un ejemplo es la interface *DSAKeyPairGenerator* (del paquete *java.security.interfaces*)

Constructor de la Clase java.security.KeyPairGenerator

✓ `KeyPairGenerator(String)`

Crea un objeto KeyPairGenerator para el algoritmo especificado.

Declaración:

```
protected KeyPairGenerator(String algorithm)
```

Parámetros:

`algorithm` - cadena que contiene el nombre estándar del algoritmo.

Métodos de la Clase java.security.KeyPairGenerator

✓ `generateKeyPair()`

Genera un par de claves. A menos que el método de inicialización sea llamado usando la interface KeyPairGenerator, el algoritmo especifico por defecto será usado. Este generará un nuevo par de claves cada vez que sea llamado.

Declaración:

```
public abstract KeyPair generateKeyPair()
```

✓ `getAlgorithm()`

Retorna el nombre estándar del algoritmo para el generador de claves.

Declaración:

```
public String getAlgorithm()
```

✓ `getInstance(String)`

Genera un objeto *KeyPairGenerator* que implementa el algoritmo requerido. Esto se permite dentro del ambiente de desarrollo.

Declaración:

```
public static KeyPairGenerator getInstance(String algorithm) throws
NoSuchAlgorithmException
```

Parámetros:

`algorithm -` cadena que contiene el nombre estándar del algoritmo.

Excepciones que arroja:

`NoSuchAlgorithmException;` si el algoritmo no está disponible dentro del ambiente de desarrollo.

✓ `getInstance(String, String)`

Genera un objeto KeyPairGenerator implementando el algoritmo especificado, como el suministrado por el proveedor especificado, si éste es un algoritmo valido del mismo.

Declaración:

```
public static KeyPairGenerator getInstance(String algorithm,
                        String provider) throws NoSuchAlgorithmException,
                                                NoSuchProviderException
```

Parámetros:

`algorithm` - cadena que contiene el nombre estándar del algoritmo

`provider` – cadena que contiene el nombre del proveedor.

Excepciones que arroja:

`NoSuchAlgorithmException,` si el algoritmo no está disponible por el proveedor.

`NoSuchProviderException;` si el proveedor no está disponible dentro del ambiente de desarrollo.

✓ `initialize(int)`

Inicializa el generador del par de claves para una "fuerza" segura usando una fuente de aleatoriedad suministrada por el sistema.

Declaración:

```
public void initialize(int strength)
```

Parámetros:

`strength` - "fuerza" de la clave. Este es un algoritmo especifico de tamaño considerable, tal como aquellos donde sus módulos son grandes.

✓ `initialize(int, SecureRandom)`

Inicializa el generador del par de claves para un "fuerza" segura.

Declaración:

```
public abstract void initialize(int strength,  SecureRandom random)
```

Parámetros:

`strength` - "fuerza" de la clave. Este es un algoritmo especifico de tamaño considerable, tal como aquellos donde sus módulos son grandes.

`random` - la fuente de aleatoriedad para este generador.

- **La Clase java.security.MessageDigest**

```
java.lang.Object
  |
  +----java.security.MessageDigest
```

MessageDigest es una clase pública abstracta. Subclase de *Object*

Esta clase suministra la funcionalidad de un algoritmo de resumen de mensaje, tal como MD5 o SHA. Los resúmenes de mensajes son asegurados por la función hash de una sola vía, esta toma un tamaño arbitrarios de datos y su salida es el valor hash de longitud fija.

Así mismo, otros algoritmos basados es clases se encuentran en Java Security. *MessageDigest* tiene dos componentes principales:

- **La API (Interface de Programación de Aplicaciones) de Resumen de Mensaje**

Esta es la interface de métodos llamada por aplicaciones que necesitan servicios de resumen de mensajes. La API consiste de todos los métodos públicos.

- **La SPI (Interface de Proveedor de Servicio) de Resumen de Mensaje**

Esta es la interface implementada por proveedores que suplen algoritmos específicos. Esta consiste de todos los métodos cuales son prefijados por *engine* (motor). Cada método es llamado por un método público de la API correspondientemente nombrado. Por ejemplo, el método *engineReset* es llamado por el método *reset.* Los métodos SPI son abstractos, proveedores que deberán suplir a implementaciones concretas.

Un objeto MessageDigest sale inicializado. El dato es procesado directamente usando el método *update.* En cualquier instante reset puede ser llamado para resetear el resumen. Una vez que todo el dato es actualizado, este tuvo que ser actualizado, uno de los métodos *digest* deben ser llamados para completar la computación hash.

El método digest puede ser llamado una vez para un número dado de actualizaciones. Después que fue llamado digest, el objeto *MessageDigest* es reseteado a su estado de inicialización.

Las aplicaciones son libres de implementar la interface *Cloneable* que se encuentra dentro del paquete *java.lang.*, y si lo hacen permitirán probar la clonabilidad de aplicaciones cliente usando la instancia *instanceof Cloneable* antes clonando:

```
MessageDigest md = MessageDigest.getInstance("SHA");
 if (md instanceof Cloneable) {
     md.update(toChapter1);
```

```
MessageDigest tc1 = md.clone();
```

```
    byte[] toChapter1Digest = tc1.digest;
    md.update(toChapter2);
    ...etc.
} else {
    throw new DigestException("couldn't make digest of partial content");
}
```

Hay que anotar que si una aplicación dada no es clonable, ésta aún tiene la posibilidad de computar resúmenes intermedios por diversos instanciamientos de objetos, si el número de resúmenes se conocen con anterioridad.

Constructores de la Clase java.security.MessageDigest

✓ `MessageDigest(String)`

Crea un mensaje de resumen con el algoritmo de nombre especificado.

Declaración:

```
protected MessageDigest(String algorithm)
```

Parámetros:

`algorithm -` nombre estándar del algoritmo de resumen.

Métodos de la Clase java.security.MessageDigest

✓ `clone()`

Retorna un clone si la inplementación es clonable.

Declaración:

```
public Object clone() throws CloneNotSupportedException
```

Excepciones que arroja:

`CloneNotSupportedException,` si esta es llamada en una implentación que no soporta la clonación.

✓ `digest()`

Completa la computación hash para levar a cabo operaciones finales tal como padding (relleno). El método digest es reseteado después que este se llame.

Declaración:

```
public byte[] digest()
```

Retorna:

El arreglo de bytes para el resultado del valor hash.

✓ `digest(byte[])`

Lleva a cabo una actualización final en el resumen usando el arreglo de bytes especificado, luego completa la computación del resumen. Esto es, el método primero llama a `update` en el arreglo, y luego llama a digest().

Declaración:

```
public byte[] digest(byte input[])
```

Parámetros:

`input -` la entrada a ser actualizada antes que el resumen es completado.

Retorna:

El arreglo de bytes para el resultado del valor hash.

✓ `engineDigest()`

SPI: completa la computación hash para llevar a cabo operaciones finales tal como rellenados (padding). Una vez que *engineDigest* fue llamado, el motor debe ser reseteado. Resetear es responsabilidad del motor implementado.

Declaración:

```
protected abstract byte[] engineDigest()
```

Retorna:

El arreglo de bytes para el resultado del valor hash.

✓ `engineReset()`

SPI: Resetea el resumen para otro uso.

Declaración:

```
protected abstract void engineReset()
```

✓ `engineUpdate(byte)`

SPI: Actualiza el resumen usando el byte especifico.

Declaración:

```
protected abstract void engineUpdate(byte input)
```

Parámetros:

`input -` el byte a usar para la actualización

✓ `engineUpdate(byte[], int, int)`

SPI: Actualiza el resumen usando el arreglo de bytes especificado, empezando en el offset especificado. Este debe ser no-op si el resumen fue inicializado.

Declaración:

```
protected abstract void engineUpdate(byte input[], int offset, int len)
```

Parámetros:

`input` - el arreglo de bytes a usarse para la actualización.

`offset` - el offset, inicio dentro el arreglo de bytes.

`len` - el número de bytes a usar, iniciados en el offset.

✓ `getAlgorithm()`

Retorna una cadena que identifica al algoritmo, independiente de la implementación detallada. El nombre un nombre estándar de Java Security (tal como "SHA", "MD5", etc.).

Declaración:

```
public final String getAlgorithm()
```

✓ `getInstance(String)`

Genera un objeto MessageDigest que implementa el algoritmo de resumen especificado. Si el paquete del proveedor por defecto contiene una subclase MessageDigest que implementa el algoritmo, una instancia de esa subclase es retornada. Si el algoritmo no está disponible en el paquete por defecto, otro paquete es buscado.

Declaración:

```
public static MessageDigest getInstance(String algorithm) throws
                                                    NoSuchAlgorithmException
```

Parámetros:

`algorithm` - el nombre del algoritmo requerido.

Retorna:

Un objeto MessageDigest implementando el algoritmo especificado.

Excepciones que arroja:

`NoSuchAlgorithmException,` Si el algoritmo no está permitido en el ambiente de quien lo llame.

✓ `getInstance(String, String)`

Genera un objeto MessageDigest implementando el algoritmo especificado, como el soportado desde el proveedor especificado, si tal algoritmo está disponible desde el proveedor.

Declaración:

```
public static MessageDigest getInstance(String algorithm String provider)
                                        throws NoSuchAlgorithmException,
                                               NoSuchProviderException
```

Parámetros:

`algorithm` - el nombre del algoritmo requerido.

`provider` - el nombre del proveedor.

Retorna:

Un objeto Resumen de Mensaje implementando el algoritmo especificado.

Excepciones que arroja:

- `NoSuchAlgorithmException`, si no está disponible dentro del paquete suministrado por el proveedor requerido.

- `NoSuchProviderException,` si el proveedor no está permitido dentro del ambiente.

✓ `isEqual(byte[], byte[])`

Compara dos resúmenes para igualdad. Compara un byte sencillo.

Declaración:

```
public static boolean isEqual(byte digesta[], byte digestb[])
```

Parámetros:

`digesta` - uno de los resúmenes a comparar.

`digestb` - el otro resumen a comparar.

Retorna:

Verdadero (true) si los resúmenes son iguales, falso (false) en caso contrario.

✓ `reset()`

Resetea el resumen para otro uso.

Declaración:

```
public void reset()
```

✓ `toString()`

Retorna una representación de cadena de este objeto de resumen de mensaje

Declaración:

```
public String toString()
```

✓ `update(byte)`

Actualiza el resumen usando el byte especificado.

Declaración:

```
public void update(byte input)
```

Parámetros:

`input -` el byte con el cual se actualiza el resumen.

✓ `update(byte[])`

Actualiza el resumen usando el arreglo de bytes especificado.

Declaración:

```
public void update(byte input[])
```

Parámetros:

`input` - el arreglo de bytes

✓ `update(byte[], int, int)`

Actualiza el resumen usando el arreglo de bytes especificado, iniciándolo en el offset especificado.

Declaración:

```
public void update(byte input[], int offset, int len)
```

Parámetros:

`input -` el arreglo de bytes.

`offset` - el offset de inicio del arreglo de bytes.

`len -` el número de bytes a usar, iniciados en el offset.

- **LA CLASE java.security.Provider**

```
java.lang.Object
  |
  +----java.util.Dictionary
        |
        +----java.util.Hashtable
              |
              +----java.util.Properties
                    |
                    +----java.security.Provider
```

Provider es una clase pública abstracta. Subclase de ***Properties*****.**

Esta clase representa un "proveedor" para la API de Java Security. Un proveedor implementa alguna o todas las partes de Java Security, incluyendo:

- Algoritmos (tales como DSA, RSA, MD5 o SHA-1).
- Generación de claves y manejador de facilidades (tal como claves para algoritmos específicos).

Cada proveedor tiene un nombre y un número de versión, y lo configuran en cada tiempo de ejecución dentro donde se instale.

Hay un proveedor por defecto que viene estándar con el JDK. Es llamado el proveedor SUN .

Constructores de la Clase java.security.Provider

✓ `Provider (String, double, String)`

Construye un proveedor con el nombre especificado, número de versión, y información.

Declaración:

```
protected Provider(String name, double version, String info)
```

Parámetros:

`name` - el nombre del proveedor.

`versión` - el número de versión del proveedor.

`info` - una descripción del proveedor y sus servicios.

Métodos de la Clase java.security.Provider

✓ `getInfo()`

Retorna una descripción legible para el usuario del proveedor y sus servicios. Este puede retornar una página HTML, con enlaces relacionados.

Declaración:

```
public String getInfo()
```

Retorna:

Una descripción del proveedor y sus servicios.

✓ `getName()`

Retorna el nombre de este proveedor.

Declaración:

```
public String getName()
```

✓ `getVersion()`

Retorna el número de versión para este proveedor.

Declaración:

```
public double getVersion()
```

✓ `toString()`

Retorna una cadena con el nombre y el número de versión de este proveedor

Declaración:

```
public String toString()
```

- **La Clase java.security.SecureRandom**

```
java.lang.Object
  |
  +----java.util.Random
         |
         +----java.security.SecureRandom
```

SecureRandom es una clase pública. Subclase de Random

Esta clase genera un número pseudo-aleatorio criptograficamente fuerte basado en el algoritmo hash SHA-1.

La llamada heredada de Random será implementada en términos del fortalecimiento de la funcionalidad.

Constructor de la Clase java.security.SecureRandom

- ✓ `SecureRandom()`

Este vacía el generador de semillas del constructor automáticamente. Se tratará de proporcionar suficientes bytes semilla completamente aleatorias en el estado interno del generador (20 bytes).

La primera vez que este constructor es llamado en una Máquina Virtual dada, este puede tomar varios segundos de tiempo de CPU para el generador de semillas, dependiendo fundamentalmente del hardware. Sucesivas llamadas se ejecutan rápidamente porque ellas dependen del mismo generador de números pseudo-aletorios para sus bits semillas.

El procedimiento de semilla implementado por este constructor aseguran que la secuencia de bytes pseudo-aletorios producidos por cada instancia SecureRandom producen información no útil acerca de las secuencia de bytes producidos por cualquier otra instancia. Si de cualquier manera, el usuario quiera producir múltiples instancias con semillas no relacionadas, el siguiente código produce el resultado deseado (en un costo substancial de CPU por instancia).

```
SecureRandom rnd = new SecureRandom(SecureRandom.getSeed(20));
```

Declaración:

```
public SecureRandom()
```

✓ `SecureRandom(byte[])`

Este constructor usa un usuario proveedor de semillas en preferencia a su mismo algoritmo de semilla referenciado en la descripción del constructor vacío. Este puede preferiblemente estar en el constructor vacío si el que llama tiene acceso a una alta calidad de bytes aleatorios desde algún dispositivo físico (por ejemplo, un detector de radiación o diodo ruidoso).

Declaración:

```
public SecureRandom(byte seed[])
```

Parámetros:

`seed` - la semilla.

✓ `getSeed(int)`

Retorna el número de bytes semilla dados. Los computa usando el algoritmo generador de semilla que esta clase usa para tal fin por si misma. Esta llamada puede ser usada para ensemillar otros generadores de números aleatorios. Cuando se intenta retornar una secuencia de bytes "verdaderamente aleatoria", no se sabe con exactitud como son aleatoriamente los bytes retornados por la llamada (vea el constructor vacío SecureRandom para más información).

Declaración:

```
public static byte[] getSeed(int numBytes)
```

Parámetros:

`numBytes` - número de bytes semillas a generar.

Retorna:

Los bytes semillas.

✓ `next(int)`

Genera un entero que contiene el número de bits pseudo-aleatorio del usuario especificado (encabezado por ceros). Este método desborda un método java.util.Random, y sirve para suministrar una fuente de aleatoriedad de bits para todos los métodos heredados desde esta clase (por ejemplo, nextInt, nextLong, y nextFloat).

Declaración:

```
protected final int next(int numBits)
```

Parámetros:

`numBits` - número de bits pseudo-aleatorios a ser generados, donde 0 <= numBits <=32.

Preválese sobre:

`next` en la clase `Random`

✓ `nextBytes(byte[])`

Genera un número de bytes aleatorios del usuario especificado. Este método es usado como la base de todas las entidades aleatorias retornadas por esta clase (exceptuando los bytes semillas). Así, este puede predominar cambiando el comportamiento de las clases.

Declaración:

```
public synchronized void nextBytes(byte result[])
```

Parámetros:

`bytes` - el arreglo a ser llenado con los bytes aleatorios.

Preválese sobre:

`nextBytes` en la clase `Random`

✓ `setSeed(byte[])`

Reensemilla este objeto aleatorio. La semilla complementaria dada, reemplaza la semilla existente. Así, repetidas llamadas nunca son garantizadas para reducir la aleatoriedad.

Declaración:

```
public synchronized void setSeed(byte seed[])
```

Parámetros:

`seed` - la semilla.

✓ `setSeed(long)`

Reensemilla este objeto aleatorio, usando los ocho bytes contenidos en el long seed dado.

La semilla complementaria dada, reemplaza la semilla existente. Así, repetidas llamadas nunca son garantizadas para reducir la aleatoriedad.

Este método es definido por compatibilidad con java.util.Random.

Declaración:

```
public void setSeed(long seed)
```

Parámetros:

`seed` - la semilla.

Preválese sobre:

`setSeed` en la clase `Random`

- **La Clase java.security.Security**

java.lang.Object
  |
  +----java.security.Security

Security es una clase pública declarada como final. Subclase de *Object*

Esta clase centraliza todas las propiedades y métodos de seguridad comunes. Uno de sus principales usos es el manejo de proveedores.

Métodos de la Clase java.security.Security

✓ `addProvider(Provider)`

Añade un proveedor a la siguiente posición disponible

Declaración:

```
public static int addProvider(Provider provider)
```

Parámetros:

`provider` - proveedor a ser añadido.

Retorna:

La posición de preferencia en la cual el proveedor fue añadido, o -1 si el proveedor no fue añadido porque este ya está instalado.

✓ `getAlgorithmProperty(String, String)`

Obtiene una propiedad especificada para un algoritmo. El nombre del algoritmo debe ser debe ser un nombre estándar. Un posible uso es para el análisis gramatical del algoritmo especializado, el cual puede mapear clases a algoritmos que ellos entiendan (mucho gusta hacer el análisis gramatical de claves).

Declaración:

```
public static String getAlgorithmProperty(String algName, String
```

`propName)`

Parámetros:

`algName` - nombre del algoritmo.

`propName` - nombre de la propiedad obtenida.

Retorna:

El valor de la propiedad especificada.

✓ `getProperty(String)`

Obtiene una propiedad de seguridad.

Declaración:

```
public static String getProperty(String key)
```

Parámetros:

`key` - la clave de la propiedad estando recuperada.

Retorna:

El valor de la propiedad de seguridad correspondiente a la clave.

✓ `getProvider(String)`

Retorna el proveedor instalado con el nombre especificado, puede ser cualquiera.

Retorna nulo si el nombre del proveedor no está instalado.

Declaración:

```
public static Provider getProvider(String name)
```

Parámetros:

`name` - nombre del proveedor obtenido.

✓ `getProviders()`

Retorna todos los proveedores actualmente instalados.

Declaración:

```
public static Provider[] getProviders()
```

Retorna:

Una arreglo de todos los proveedores actualmente instalados.

✓ `insertProviderAt(Provider, int)`

Adiciona un nuevo proveedor, en la posición especificada. La posición es el orden de preferencia en el cual los proveedores son buscados para el algoritmo requerido. Note que este no garantiza que estas preferencias serán respetadas. La posición base es la 1, ya que la 1 es la de mayor preferencia, seguida por la 2, y así sucesivamente. Algunas veces añadir este un proveedor será legal, pero solo en la última posición, en el cual el argumento *position* será ignorado.

Si el proveedor dado es instalado en la posición requerida, el proveedor que estaba en esa posición, y todos los demás en una posición más avanzada, son corridos una posición hacia arriba(hacia el final de la lista de los proveedores instalados).

Un proveedor no puede ser adicionado si ya está instalado.

Declaración:

```
public static int insertProviderAt(Provider provider, int position)
```

Parámetros:

`provider:` el proveedor a ser adicionado

`position:` La posición preferida el llamador quiere para ese proveedor

Valores que retorna:

La posición de preferencia actual en la cual el proveedor fue adicionado, o -1 si el proveedor no fue adicionado debido a que está instalado.

- ✓ `removeProvider(String)`

Borra el proveedor con el nombre especificado. Cuando el proveedor especificado es borrado, todos los proveedores localizados en una posición mayor que donde el proveedor especificado se encuentra son corridas hacia abajo una posición (hacia la cabeza de la lista de proveedores instalados).

Este método retorna con discreción (no pasa nada) si el proveedor no está instalado.

Declaración:

```
public static void removeProvider(String name)
```

Parámetros:

`name:` el nombre del proveedor a remover

✓ `setProperty(String, String)`

Configura una propiedad de seguridad

Declaración:

```
public static void setProperty(String key, String datum)
```

Parámetros:

`key:` El nombre de la propiedad a ser configurada.

datum: El valor de la propiedad a ser configurada.

- **La Clase java.security.Signature**

```
java.lang.Object
  |
  +----java.security.Signature
```

Signature Clase pública abstracta.

Subclase de *Object*

Esta clase de firma es usada para proporcionar la funcionalidad de un algoritmo de firma digital, tal como RSA con MD5 o DSA. Las firmas digitales son usadas para autenticación y aseguramiento de la integridad de los datos digitales.

Como otras clases basadas en algoritmo en java security, la clase Signature tiene dos componentes principales:

- Digital Signature API (Interface de programación de aplicaciones)

Esta es la interface de métodos llamados por aplicaciones que necesitan de servicios de firma digital. La API está conformada por todos los métodos públicos.

- Digital Signature SPI (Interface del proveedor del servicio)

Esta es la interface implementada por los proveedores que proporcionan algoritmos específicos. Está conformada por todos los métodos cuyos nombres son comenzados por engine. Cada uno de esos métodos son llamados por un método del API público nombrado correspondientemente. Por ejemplo, el método *engineSign* es llamado por el método *sign.* Los métodos SPI son abstractos; los proveedores deben suministrar una implementación concreta.

Así como otras clases basadas en algoritmos en java security, Signature proporciona algoritmos independientes de la implementación, por medio de la cual un llamador(código de aplicación) solicita un algoritmo de firma particular y es pasada un objeto Signature inicializado correctamente. Es también posible, si se desea, solicitar un algoritmo particular de un proveedor particular.

Además, hay dos formas de solicitar un objeto de algoritmo Signature: ya sea especificando solo un nombre de algoritmo o un nombre de algoritmo acompañado por un proveedor de paquete.

- Si solo se especifica el nombre del algoritmo, el sistema determinará si hay una implementación del algoritmo solicitado disponible en el ambiente de trabajo, y si hay más de uno, si hay uno preferido.

- Si se especifican tanto nombre del algoritmo como el proveedor del paquete, el sistema determinará si hay una implementación del algoritmo en el paquete solicitado, y lanza una excepción si no hay.

Un objeto Signature puede ser usado para generar y verificar firmas digitales.

Hay tres fases para usar un objeto Signature ya sea para firmar datos o para verificar una firma:

1. Inicialización, ya sea con una clave pública, la cual inicializa la firma para verificación, o una clave pública, la cual inicializa la firma para firmado.

2. Actualización, dependiendo del tipo de inicialización, esta actualizará los bytes a ser firmados o verificados.

3. Firmando o verificando, una firma sobre todos los bytes actualizados.

Variables de la Clase java.security.Signature

✓ `SIGN`

Posible valor de "state", significando que este objeto de firma ha sido inicializado para firma.

Declaración:

```
protected static final int SIGN
```

✓ `state`

Estado actual de este objeto de firma

Declaración:

```
protected int state
```

✓ `UNINITIALIZED`

Posible valor de "state", significando que este objeto de firma no ha sido inicializado aún.

Declaración:

```
protected static final int UNINITIALIZED
```

✓ `VERIFY`

Posible valor de "state", significando que este objeto de firma ha sido inicializado para verificación.

Declaración:

```
protected static final int VERIFY
```

Constructor de la Clase java.security.Signature

✓ `Signature(String)`

Crea un objeto Signature para el algoritmo especificado

Declaración:

```
protected Signature(String algorithm)
```

Parámetros:

`algorithm:` El nombre de cadena estándar de algoritmo.

Métodos de la Clase java.security.Signature

✓ `clone()`

Devuelve un clone si la implementación es clonable.

Declaración:

```
public Object clone() throws CloneNotSupportedException
```

Valores que devuelve:

Un clone si la implementación es clonable.

Excepciones que arroja:

`CloneNotSupportedException`, si este es llamado sobre una implementación que no soporta clonable.

Preválese sobre:

`clone` en la clase `Object.`

✓ `engineGetParameter(String)`

SPI: Obtiene el valor del parámetro del algoritmo especificado. Este método proporciona un mecanismo de propósito general a través del cual es posible obtener varios de los parámetros de este objeto. Un parámetro podría ser un parámetro configurable para el algoritmo, tal como un tamaño del parámetro, o una fuente de bits aleatorios para generación de firmas (si es apropiado), o una indicación de si se realiza una computación opcional o específica. Un esquema de llamado uniforme de algoritmo específico para cada parámetro es deseable pero se deja sin especificar en este momento.

Declaración:

```
protected abstract Object engineGetParameter(String param)
                                        throws InvalidParameterException
```

Parámetros:

`param:` El nombre de la cadena del parámetro

Valores que devuelve:

El objeto que representa el valor del parámetro, o nulo si no hay alguno.

Excepciones que arroja:

`InvalidParameterException,` si el parámetro es inválido para este motor, u otra excepción ocurre mientras se intenta obtener este parámetro.

✓ `engineInitSign(PrivateKey)`

SPI: Inicializa este objeto de firma con la clave especificada privada para operaciones de firmado.

Declaración:

```
protected abstract void engineInitSign(PrivateKey privateKey) throws
InvalidKeyException
```

Parámetros:

`privateKey:` La clave privada de la identidad para quien será generada la firma.

Excepciones que arroja:

`InvalidKeyException`, si la clave no es codificada en la forma debida, se han olvidado parámetros, y demás.

✓ `engineInitVerify(PublicKey)`

SPI: Inicializa este objeto de firma con la clave pública especificada para operaciones de verificación.

Declaración:

```
protected abstract void engineInitVerify(PublicKey publicKey) throws
                                                    InvalidKeyException
```

Parámetros:

`publicKey:` La clave pública de la identidad a quien se va a verificar la firma.

Excepciones que arroja:

`InvalidKeyException`, si la clave es decodificada de manera indebida, son olvidados parámetros, etc.

✓ `engineSetParameter(String, Object)`

SPI: coloca en el parámetro del algoritmo especificado el valor dado. Este método proporciona un mecanismo de propósito general a través del cual es posible configurar varios de los parámetros de este objeto. Un parámetro podría ser un parámetro configurable para el algoritmo, tal como un tamaño del parámetro, o una fuente de bits aleatorios para generación de firmas (si es apropiado), o una indicación de si se realiza una computación opcional o específica. Un esquema de llamado uniforme de algoritmo específico para cada parámetro es deseable pero se deja sin especificar en este momento.

Declaración:

```
protected abstract void engineSetParameter(String param, Object value)
                                           throws InvalidParameterException
```

Parámetros:

`param:` El identificador de cadena del parámetro.

`value:` El valor del parámetro.

Excepciones que arroja:

`InvalidParameterException,` Si el parámetro no es válido para este motor de algoritmo de firma, el parámetro ya ha sido configurado y no puede ser configurado otra vez, ocurre una excepción de seguridad, etc.

✓ `engineSign()`

SPI: Devuelve los bytes de firma de todos los datos configurados hasta el momento. La firma devuelta es X.509-decodificada.

Declaración:

```
protected abstract byte[] engineSign() throws SignatureException
```

Datos que devuelve:

Los bytes de la firma que resultan de una operación de firmado

Excepciones que arroja:

`SignatureException,` si el motor no es inicializado en la forma debida.

✓ `engineUpdate(byte)`

SPI: Actualiza los datos a ser firmados o verificados usando el byte específico.

Declaración:

```
protected abstract void engineUpdate(byte b) throws SignatureException
```

Parámetros:

`b:` El byte a usar para la actualización

Excepciones que arroja:

`SignatureException,` Si el motor no es inicializado en forma correcta.

✓ `engineUpdate(byte[], int, int)`

SPI: Actualiza los datos a ser firmados o verificados, usando el arreglo de bytes especificado, comenzando por el offset especificado

Declaración:

```
protected abstract void engineUpdate(byte b[], int off, int len) throws
                                                        SignatureException
```

Parámetros:

`data:` El arreglo de bytes

`off:` El offset desde donde se comenzará en el arreglo de datos.

`len:` El número de bytes a usar, comenzando en el offset.

Excepciones que arroja:

`SignatureException,` Si el motor no es inicializado en forma correcta.

✓ `engineVerify(byte[])`

SPI: Verifica la firma que es pasada. Los bytes de firma se espera que sean X.509- decodificados.

Declaración:

```
protected abstract boolean engineVerify(byte sigBytes[]) throws
                                                    SignatureException
```

Parámetros:

`sigBytes:` Los bytes de firma a ser verificados.

Datos que devuelve:

true si la firma fue verificada, false si no.

Excepciones que arroja:

`SignatureException,` Si el motor no es inicializado correctamente, o la firma pasada es decodificada de forma incorrecta o es de tipo errado, etc.

✓ `getAlgorithm()`

Devuelve el nombre del algoritmo para este objeto de firma.

Declaración:

```
public final String getAlgorithm()
```

Valores que devuelve:

El nombre del algoritmo para este objeto de firma.

✓ `getInstance(String)`

Genera un objeto Signature que implementa el algoritmo especificado. Si el paquete del proveedor por defecto contiene una subclase de Signature implementando el algoritmo, una instancia de esa subclase es devuelta. Si el algoritmo no está disponible en el paquete por defecto, son revisados otros paquetes.

Declaración:

```
public static Signature getInstance(String algorithm)
                                    throws NoSuchAlgorithmException
```

Parámetros:

`algorithm:` El nombre estándar del algoritmo solicitado.

Valores que devuelve:

El nuevo objeto Signature

Excepciones que arroja:

`NoSuchAlgorithmException`, si el algoritmo no está disponible en el ambiente de trabajo.

✓ `getParameter(String)`

Genera un objeto Signature implementando el algoritmo específico, suministrado por el proveedor especificado, si tal algoritmo está disponible desde ese proveedor.

Declaración:

```
public static Signature getInstance(String algorithm,String provider)
                                        throws NoSuchAlgorithmException,
                                               NoSuchProviderException
```

Parámetros:

`algorithm:` El nombre del algoritmo solicitado.

`provider:` El nombre del proveedor.

Valores que devuelve:

El nuevo objeto Signature

Excepciones que arroja:

- `NoSuchAlgorithmException,` si el algoritmo no está disponible en el paquete suministrado por el proveedor solicitado.

- `NoSuchProviderException,` si el proveedor no está disponible en el ambiente de trabajo.

✓ `getParameter(String)`

Obtiene el valor del parámetro del algoritmo especificado. Este método proporciona un mecanismo de propósito general a través del cual es posible obtener varios parámetros de este objeto. Un parámetro podría ser algún parámetro configurable para el algoritmo, tal como un tamaño del parámetro, o una fuente de bits aleatorios para generación de firmas (si es apropiado), o una indicación de si se realiza una computación opcional o específica.

Un esquema de llamado uniforme de algoritmo específico para cada parámetro es deseable pero se deja sin especificar en este momento.

Declaración:

```
public final Object getParameter(String param) throws
                                                InvalidParameterException
```

Parámetros:

`param:` El nombre de la cadena del parámetro.

Valores que devuelve:

El objeto que representa el valor del parámetro, o null si no hay alguno.

Excepciones que arroja:

`InvalidParameterException,` si param es un parámetro inválido para este motor, u otra excepción ocurre mientras se intenta obtener este parámetro.

✓ `initSign(PrivateKey)`

Inicializa este objeto para firma. Si este método es llamado otra vez con un argumento diferente, niega el efecto de esta llamada.

Declaración:

```
public final void initSign(PrivateKey privateKey) throws
InvalidKeyException
```

Parámetros:

`priveteKey:` la clave privada de la identidad para quien la firma será generada.

Excepciones que arroja:

`InvalidKeyException,` si la clave es inválida.

✓ `initVerify(PublicKey)`

Inicializa este objeto para verificación. Si este método es llamado otra vez con un argumento diferente, niega el efecto de esta llamada.

Declaración:

```
public final void initVerify(PublicKey publicKey) throws
                                                    InvalidKeyException
```

Parámetros:

`publicKey:` La clave pública de la identidad para quien la firma será verificada.

Excepciones que arroja:

`InvalidKeyException`, si la clave es inválida.

✓ `setParameter(String, Object)`

Configura el parámetro del algoritmo especificado con el valor dado. Este método proporciona un mecanismo de propósito general a través del cual es posible configurar varios de los parámetros de este objeto. Un parámetro podría ser algún parámetro configurable para el algoritmo, tal como un tamaño del parámetro, o una fuente de bits aleatoria para generación de firmas (si es apropiado), o una indicación de si se realiza una computación opcional o específica. Un esquema denominado algoritmo específico uniforme por cada parámetro es considerado pero se deja sin especificar en este momento.

Declaración:

```
public final void setParameter(String param, Object value) throws
```

`InvalidParameterException`

Parámetros:

`param:` El identificador de cadena del parámetro.

`value:` El valor del parámetro.

Excepciones que arroja:

`InvalidParameterException,` si param es un parámetro inválido para este motor de algoritmo de firma, el parámetro ya está configurado y no puede ser configurado nuevamente, ocurre una excepción de seguridad, etc.

✓ `sign()`

Devuelve los bytes de firma de todos los datos actualizados. La firma devuelta es X.509-decodificada.

Una llamada a este método pone el objeto de firma en el estado que estaba cuando fue previamente inicializado para firmado a través de una llamada a initSign(PrivateKey). Es decir el objeto se resetea y se vuelve disponible para generar otra firma del mismo firmante, si se desea a través de nuevas llamadas a update y sign.

Declaración:

```
public final byte[] sign() throws SignatureException
```

Valores que devuelve:

Los bytes de firma que resultan una operación de firmado.

✓ `toString()`

Devuelve una representación en string del objeto de firma, proporcionando información que incluye el estado del objeto y el nombre del algoritmo usado.

Declaración:

```
public String toString()
```

Valor que devuelve:

Una representación en string de este objeto de firma.

Predomina dentro de:

`toString` en la clase `Object`

✓ `update(byte)`

Configura los datos a ser firmados o verificados por n byte.

Declaración:

```
public final void update(byte b) throws SignatureException
```

Parámetros:

`b:` El byte a usar para la actualización

Excepciones que arroja:

`SignatureException,` si este objeto de firma no es inicializado de manera correcta.

✓ `update(byte[])`

Configura los datos a ser firmados o verificados usando el arreglo específico de bytes.

Declaración:

```
public final void update(byte data[]) throws SignatureException
```

Parámetros:

`data:` El arreglo de bytes a usar para la actualización.

Excepciones que arroja:

`SignatureException,` si este objeto de firma no es inicializado correctamente.

✓ `update(byte[], int, int)`

Configura los datos a ser firmados o verificados usando el arreglo específico de bytes. Empezando en el offset especificado.

Declaración:

```
public final void update(byte data[], int off, int len)
                                                    throws SignatureException
```

Parámetros:

`data:` El arreglo de bytes a usar para la actualización.

`off:` El offset desde donde se va a comenzar en el arreglo de bytes.

`len:` El número de bytes a usar, comenzando en el offset.

Excepciones que arroja:

`SignatureException,` si este objeto de firma no es inicializado correctamente.

✓ `verify(byte[])`

Verifica la firma pasada. Los bytes de firma se espera que sean X.509-decodificados.

Una llamada a este método resetea este objeto de firma al estado donde se encontraba previamente al inicializarlo a través de una llamada a *initVerify(PublicKey).* Es decir, el objeto se resetea y se hace disponible para otra firma de la identidad para quien la clave pública fue especificada en la llamada a initVerify.

Declaración:

```
public final boolean verify(byte signature[]) throws SignatureException
```

Parámetros:

`signature:` Los bytes de firma a ser verificados.

Valores que devuelve:

true si la firma fue verificada, false si no.

Excepciones que arroja:

`SignatureException,` si el objeto de firma no es inicializado correctamente, o la firma que se pasa esta decodificada de manera inadecuada o el tipo está errado, etc.

- **La Clase java.security.Signer**

```
java.lang.Object
  |
  +----java.security.Identity
         |
         +----java.security.Signer
```

Signer Clase pública abstracta.

Subclase de Identity

Esta clase es usada para representar una Identidad que puede también digitalmente firmar datos.

El manejo de una clave privada de un firmante es un asunto importante y sensible que debería ser manejado por subclases en cuanto a sus adecuados usos futuros.

Constructores de la Clase java.security.Signer

✓ `Signer()`

Crea un firmante. Este constructor debe solo ser usado para serialización.

Declaración:

```
protected Signer()
```

✓ `Signer(String)`

Crea un firmante con el nombre de identidad especificado.

Declaración:

```
public Signer(String name)
```

Parámetros:

`name` - nombre de identidad.

✓ `Signer(String, IdentityScope)`

Crea un firmante con el nombre de identidad y ámbito especificados.

Declaración:

```
public Signer(String name, IdentityScope scope) throws
                                                    KeyManagementException
```

Parámetros:

`name` - el nombre de identidad.

`scope` - el ámbito de la identidad.

Excepciones que arroja:

`KeyManagementException,` si ya existe una identidad con el mismo nombre dentro del ámbito.

Métodos de la Clase java.security.Signer

✓ `getPrivateKey()`

Devuelve la clave privada de este firmante.

Declaración:

```
public PrivateKey getPrivateKey()
```

Valores que devuelve:

La clave privada de este firmante, o nulo si la clave privada aún no es establecida.

✓ `setKeyPair(KeyPair)`

Establece el par de claves (pública y privada) para este firmante.

Declaración:

```
public final void setKeyPair(KeyPair pair) throws
InvalidParameterException, KeyException
```

Parámetros:

`pair` - la inicialización de un par de claves.

Excepciones que arroja:

- `InvalidParameterException`, si el par de claves no fueron debidamente inicializadas.

- `KeyException,` si el par de claves no pudieron ser establecidas por alguna razón.

✓ `toString()`

Devuelve una cadena de información acerca del firmante.

Declaración:

```
public String toString()
```

Valores que devuelve:

Devuelve una cadena de información acerca del firmante.

Predomina sobre:

toString en la clase Identity

2.1.3. EXCEPCIONES

➢ **La Clase java.security.DigestException**

```
java.lang.Object
  |
  +----java.lang.Throwable
         |
         +----java.lang.Exception
                |
                +----java.security.DigestException
```

DigestException es una clase pública.
Subclase de *Exception*

Esta es la excepción genérica de resumen de mensajes (*Message Digest)*

Constructores de la Clase java.security.digestexception

- ✓ `DigestException()`

Construye una DigestException sin mensaje de detalle(Un mensaje de detalle es una cadena que describe esta excepción particular).

Declaración:

```
public DigestException()
```

- ✓ `DigestException(String)`

Construye una DigestException con el mensaje de detalle especificado

Declaración:

```
public DigestException(String msg)
```

Parámetros:

`msg:` el mensaje de detalle.

➢ La Clase java.security.InvalidKeyException

```
java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.security.KeyException
                    |
                    +----java.security.InvalidKeyException
```

InvalidKeyException es una clase pública

Subclase de *KeyException*.

Esta es la excepción para claves inválidas (codificación o decodificación inválidas, longitud errada, sin inicializar, etc.).

Constructores de la Clase java.security.InvalidKeyException

✓ `InvalidKeyException()`

Construye una InvalidKeyException sin mensaje de detalle.

Declaración:

```
public InvalidKeyException()
```

✓ `InvalidKeyException(String)`

Construye una InvalidKeyException con el mensaje de detalle específico.

Declaración:

```
public InvalidKeyException(String msg)
```

Parámetros:

`msg:` El mensaje de detalle.

- **La Clase java.security.InvalidParameterException**

```
java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.lang.RuntimeException
                    |
                    +----java.lang.IllegalArgumentException
                          |
                          +----java.security.InvalidParameterException
```

InvalidParameterException es una clase pública

Subclase de *IllegalArgumentException*

Esta excepción es arrojada cuando un parámetro inválido es pasado a un método.

Constructores de Clase java.security.InvalidParameterException

- ✓ `InvalidParameterException()`

Construye una InvalidParameterException sin mensaje de detalle.

Declaración:

```
public InvalidParameterException()
```

✓ `InvalidParameterException(String)`

Construye una InvalidParameterException con el mensaje de detalle específico.

Declaración:

```
public InvalidParameterException(String msg)
```

Parámetros:

`msg:` el mensaje de detalle.

➢ **La Clase java.security.KeyException**

```
java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.security.KeyException
```

KeyException es una clase pública.

Subclase de *Exception*

Esta es la excepción de claves básica.

Constructores de la Clase java.security.KeyException

✓ `KeyException()`

Construye una KeyException sin mensaje de detalle.

Declaración:

```
public KeyException()
```

✓ `KeyException(String)`

Construye una KeyException con el mensaje de detalle específico.

Declaración:

```
public KeyException(String msg)
```

Parámetros:

`msg:` el mensaje de detalle.

- **La Clase java.security.KeyManagementException**

```
java.lang.Object
  |
  +----java.lang.Throwable
         |
         +----java.lang.Exception
                |
                +----java.security.KeyException
                       |
                       +----java.security.KeyManagementException
```

KeyManagementException es una clase pública

Subclase de *KeyException*

Esta es la excepción de manejo de claves general, para todas las operaciones relacionadas con el manejo de claves. Las subclases deben incluir:

- KeyIDConflict
- KeyAuthorizationFailureException
- ExpiredKeyException

Constructores de la Clase java.security.KeyManagementException

✓ `KeyManagementException()`

Construye una KeyManagementException sin mensaje de detalle.

Declaración:

```
public KeyManagementException()
```

✓ `KeyManagementException(String)`

Construye una KeyManagementException con el mensaje de detalle específico

Declaración:

```
public KeyManagementException(String msg)
```

Parámetros:

`msg:` el mensaje de detalle.

- **La Clase java.security.NoSuchAlgorithmException**

```
java.lang.Object
    |
    +----java.lang.Throwable
          |
          +----java.lang.Exception
                |
                +----java.security.NoSuchAlgorithmException
```

NoSuchAlgorithmException es una clase pública.

Subclase de *Exception*

Esta excepción es arrojada cuando un algoritmo criptográfico particular es solicitado pero no está disponible en el ambiente de trabajo.

Constructores de la Clase java.security.NoSuchAlgorithmException

✓ `NoSuchAlgorithmException()`

Construye una NoSuchAlgorithmException sin mensaje de detalle.

Declaración:

```
public NoSuchAlgorithmException()
```

✓ `NoSuchAlgorithmException(String)`

Construye una NoSuchAlgorithmException con el mensaje de detalle específico. Un mensaje de detalle es una cadena que describe esta particular excepción, la cual puede, por ejemplo, especificar cual algoritmo no está disponible.

Declaración:

```
public NoSuchAlgorithmException(String msg)
```

Parámetros:

`msg`: el mensaje de detalle.

➢ La Clase java.security.NoSuchProviderException

```
java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.security.NoSuchProviderException
```

NoSuchProviderException es una clase pública.
Subclase de ***Exception***

Esta excepción es arrojada cuando un proveedor de seguridad particular es solicitado pero no está disponible en ese ambiente.

Constructores de la Clase java.security.NoSuchProviderException

✓ `NoSuchProviderException()`

Construye una NoSuchProviderException sin mensaje de detalle.

Declaración:

```
public NoSuchProviderException()
```

✓ `NoSuchProviderException(String)`

Construye una NoSuchProviderException con el mensaje de detalle específico.

Declaración:

```
public NoSuchProviderException(String msg)
```

Parámetros:

`msg:` el mensaje de detalle.

- **La Clase java.security.ProviderException**

```
java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.lang.RuntimeException
                    |
                    +----java.security.ProviderException
```

ProviderException es una clase pública.

Subclase de ***RuntimeException***

Una excepción de tiempo de ejecución para excepciones de proveedores (tales como errores de mala configuración), la cual podría ser utilizada como subclase por proveedores para el arrojo de errores de tiempo de ejecución especializados y específicos del proveedor.

Constructores de la Clase java.security.ProviderException

✓ `ProviderException()`

Construye una ProviderException sin mensaje de detalle.

Declaración:

```
public ProviderException()
```

✓ `ProviderException(String)`

Construye una ProviderException con el mensaje de detalle específico.

Declaración:

```
public ProviderException(String s)
```

Parámetros:

`s` - el mensaje de detalle.

- **La Clase java.security.SignatureException**

java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.security.SignatureException

SignatureException es una clase pública.

Subclase de *Exception*

Esta es la excepción genérica de firmas.

Constructores de la Clase java.security.SignatureException

✓ `SignatureException()`

Construye una SignatureException sin mensaje de detalle.

Declaración:

```
public SignatureException()
```

✓ `SignatureException(String msg)`

Construye una SignatureException con el mensaje de detalle específico.

Declaración:

```
public SignatureException(String msg)
```

Parámetros:

`msg:` el mensaje de detalle.

2.2. EL PAQUETE java.security.acl

2.2.1. Interfaces

- **La Interface java.security.acl.Acl.**

Esta interface representa una lista de control de acceso(ACL). Una lista de control de acceso es una estructura de datos usada para proteger el acceso a recursos. Un ACL puede ser visto como una estructura de datos con múltiples entradas ACL. Cada una de estas entradas, de interfaces tipo AclEntry, contiene un conjunto de permisos asociados con un "principal particular". Adicionalmente, cada entrada ACL es especificada para ser positiva o negativa. Si es positiva, los permisos deben ser otorgados al principal asociado. Si es negativo, los permisos le son negados.

Las entradas ACL en cada ACL acatan las siguientes reglas:

- Cada Principal puede tener a lo más una entrada ACL positiva o negativa; es decir, múltiples entradas, positivas o negativas, no son permitidas para ningún principal. Cada entrada especifica el conjunto de permisos que serán concedidos(si es positivo) o negados(si es negativo).
- Si no hay entradas para un principal en particular, se considera que ese principal tiene un conjunto de permisos nulo o vacío.

- Si hay una entrada positiva que concede, a un principal, un permiso en particular y una entrada negativa que niega a dicho principal el mismo permiso, el resultado es tomado como si el permiso nunca fue negado o concedido.
- Los permisos individuales siempre sobre escriben los permisos de grupo(s) al cual(es) el principal pertenece, es decir, los permisos negativos individuales(que especifican negación de permisos) reescriben los permisos positivos de los grupos y los permisos positivos individuales sobreescriben los permisos de negación de los grupos.

El paquete java.security.acl proporciona las interfaces para el ACL y las estructuras de datos relacionadas (entradas ACL, grupos, permisos, etc.) y las clases sun.security.acl proporciona una implementación por defecto de las interfaces. Por ejemplo, java.security.acl.Acl proporciona la interface para una ACL y la clase *sun.security.acl.AclImpl* proporciona la implementación por defecto de la interface.

La interface java.security.acl.Acl se extiende de la interface java.security.acl.Owner. La interface Owner es usada para mantener una lista de dueños para cada ACL. Solo los dueños tienen permiso para modificar una ACL. Por ejemplo solo un dueño puede llamar el método addEntry de las ACL para adicionar una nueva entrada al ACL.

Métodos de la Interface java.security.acl.Acl.

✓ `addEntry(Principal, AclEntry):`

Adiciona una entrada ACL a esta ACL. Una entrada asociada a un principal(por ejemplo, un individual o un grupo) con un conjunto de permisos. Si ya hay una entrada ACL del mismo tipo(negativa o positiva) en el ACL, se retorna falso.

Declaración:

```
public abstract boolean addEntry(Principal caller, AclEntry entry) throws
                                                                NotOwnerException
```

Parámetros:

`caller:` El "principal" que invoca el método

`entry:` La entrada ACL que será adicionada a esta ACL.

Valores que Devuelve:

t*rue* cuando se tuvo éxito, *false* si hay una entrada del mismo tipo(positiva o negativa) para el mismo principal en esta ACL.

Excepciones que arroja:

`NotOwnerExcepcion,` Si el llamador(caller) principal no es un dueño de esta acl.

✓ `CheckPermission(Principal, Permission)`

Chequea si el principal especificado tiene o no el permiso especificado. Si es así, se devuelve verdadero, de lo contrario se devuelve falso. Más específicamente, este método chequea si el permiso pasado es un miembro del conjunto de permisos concedidos del principal especificado. El conjunto de permisos concedidos es determinado por el mismo algoritmo tal como se usa por el método getPermissions.

Declaración:

```
public abstract boolean checkPermission(Principal principal,
                                        Permission permission)
```

Parámetros:

`principal:` el principal, asumido como un principal autenticado válido.

`permission:` el permiso por el cual se está chequeando.

Val*ores que Devuelve:*

true si el principal tiene el permiso especificado, de otro modo es false.

✓ `entries()`

Devuelve una enumeración de las entradas en esta ACL. Cada elemento en la enumeración es del tipo AclEntry.

Declaración:

```
public abstract Enumeration entries()
```

Valores que Devuelve:

Una enumeración de las entradas en ésta ACL.

✓ `getName()`

Retorna el nombre de esta ACL.

Declaración:

```
public abstract String getName()
```

✓ `getPermisions(Principal)`

Devuelve una enumeración para el conjunto de permisos concedidos para el principal especificado(representando una entidad individual o un grupo). Este conjunto de permisos concedidos es calculado como sigue:

Si no hay entrada en esta Lista de Control de Acceso ACL para el principal específico, un conjunto de permisos vacíos es devuelto. De otro modo, los conjuntos de permisos del grupo del principal están determinados. (Un principal puede pertenecer a uno o más grupos, donde un grupo es un grupo de principales, representado por la interface del grupo.) El conjunto de permisos positivos del grupo es la unión de todos los permisos positivos de cada grupo al cual el principal pertenece. El conjunto de permisos negativos del grupo es la unión de todos los permisos negativos de cada grupo al cual el principal pertenece. Si hay un permiso específico que ocurra tanto en el conjunto de permisos positivos como en el negativo, éste es borrado de ambos.

Los conjuntos de permisos positivos y negativos individuales también son determinados. El conjunto de permisos positivos contiene los permisos especificados en la entrada ACL positiva (si la hay) para el principal. Similarmente, el conjunto de permisos negativos contiene los permisos especificados en la entrada negativa ACL (si la hay) para el principal. El conjunto de permisos positivos (o negativos) es considerado nulo si no hay una entrada ACL positiva (o negativa) para el principal en esta ACL.

El conjunto de permisos concedidos al principal es luego calculado usando la simple regla que los permisos individuales siempre sobre-escriben los permisos de grupo. Es decir, el conjunto de permisos negativos del principal (negación específica de permisos) sobre-escribe el conjunto de permisos positivos del grupo, y el conjunto de permisos positivos individuales del principal sobre-escribe el conjunto de permisos negativos del grupo.

Declaración:

```
public abstract Enumeration getPermissions(Principal user)
```

Parámetros:

`user`: El principal de quien el conjunto de permisos será devuelto.

Valores que Devuelve:

El conjunto de permisos que especifica los permisos a los que al principal se han concedido.

✓ `removeEntry(Principal, AclEntry)`

Borra una entrada ACL de ésta ACL.

Declaración:

```
public abstract boolean removeEntry(Principal caller, AclEntry entry)
                                                    throws NotOwnerException
```

Parámetros:

`Caller:` El principal que invoca este método. Debe ser un dueño de esta ACL.

`entry:` La entrada ACL a ser borrada de esta ACL.

Valores que Devuelve:

Verdadero cuando la función utilizada realiza su tarea con éxito, falso si la entrada no hace parte de esta ACL.

Excepciones que Arroja:

`NotOwnerException,` Si el principal caller no es un dueño de esta ACL.

✓ `setName(Principal, String)`

Determina el nombre de esta ACL

Declaración:

```
public abstract void setName(Principal caller, String name) throws
                                                        NotOwnerException
```

Parámetros:

`caller:` el principal invoca este método. Este debe ser un dueño de este ACL.

`name:` el nombre a ser dado a esta ACL.

Excepciones que arroja:

`NotOwnerException`, si el que llama al principal no es un dueño de esta ACL.

✓ `ToString()`

Devuelve una representación de tipo string del contenido del ACL.

Declaración:

```
public abstract String toString()
```

Valores que Devuelve:

Una representación string del contenido del ACL.

Predomina dentro de:

toStrings en la clase *Object.*

- **Interface java.security.acl.AclEntry**

AclEntry es una interface pública.

Subclase de *Cloneable*

Esta es la interface usada para representar una entrada en una lista de control de acceso (ACL).

Una ACL puede ser pensada como una estructura de datos con múltiples objetos de entrada ACL. Cada objeto de entrada ACL contiene un conjunto de permisos asociados a un principal particular. (Un principal representa una entidad tal como un usuario individual o un grupo).

Adicionalmente, cada entrada ACL es especificada para ser negativa o positiva. Si es positiva, los permisos serán otorgados al principal asociado. Si es negativa, los permisos serán negados.

Cada principal puede tener a lo más una entrada ACL positiva y una entrada ACL negativa; Es decir, múltiples entradas ACL positivas o negativas no son permitidas para ningún principal. Nota: Las entradas ACL son positivas por defecto. Una entrada llega a ser negativa solo si el método setNegativePermissions es llamado sobre ella.

Métodos de la Interface java.security.acl.AclEntry

- ✓ `addPermission(Permission permission)`

Adiciona el permiso especificado a esta entrada ACL. Nota: una entrada podría tener múltiples permisos.

Declaración:

```
public abstract boolean addPermission(Permission permission)
```

Parámetros:

`permission:` el permiso a ser asociado con el principal en esta entrada.

Valores que Devuelve:

true si el permiso fue adicionado, falso si el permiso fue ya parte de este conjunto de permisos de entrada.

- ✓ `checkPermission(Permission  permission)`

Chequea si el permiso especificado es parte del conjunto de permisos en esta entrada.

Declaración:

```
public abstract boolean checkPermission(Permission permission)
```

Parámetros:

`Permission:` El permiso a ser chequeado.

Valores que Devuelve:

true si el permiso es parte del conjunto de permisos en esta entrada, falso de lo contrario.

✓ `clone()`

Clona esta entrada ACL.

Declaración:

```
public abstract Object clone()
```

Valores que Devuelve:

Un clone de esta entrada ACL.

Predomina dentro de:

clone en la clase Object.

✓ `getPrincipal()`

Devuelve el principal para el cual los permisos son concedidos o negados por esta entrada ACL. Devuelve null si no hay aun un determinado principal para esta entrada.

Declaración:

```
public abstract Principal getPrincipal()
```

Valores que Devuelve:

El principal asociado con esta entrada.

- ✓ `isNegative()`

Devuelve true si esta es una entrada ACL negativa(una que esté negando al principal asociado el conjunto de permisos en la entrada), falso de otro modo.

Declaración:

```
public abstract boolean isNegative()
```

Valores que devuelve:

true si esta es una entrada ACL negativa, false si no lo es.

- ✓ `permissions()`

Devuelve una enumeración de los permisos en esta entrada ACL.

Declaración:

```
public abstract Enumeration permissions()
```

Valores que Devuelve:

Una enumeración de los permisos en esta entrada ACL.

✓ `removePermission(Permission  permission)`

Borra el permiso especificado de esta entrada ACL.

Declaración:

```
public abstract boolean removePermission(Permission permission)
```

Parámetros:

`permission:` El permiso a ser borrado de esta entrada.

Valores que Devuelve:

true si el permiso es borrado, false si el permiso no fue parte de este conjunto de permisos de entrada.

✓ `setNegativePermissions()`

Configura esta entrada ACL para ser negativa. Es decir, el principal asociado (ej. un usuario o un grupo) le será negado el conjunto de permisos especificados en la

entrada. Nota: Las entradas ACL son por defecto positivas. Una entrada llega a ser negativa solo si este método setNegativePermissions es llamado sobre ella.

Declaración:

```
public abstract void setNegativePermissions()
```

✓ `setPrincipal(Principal user)`

Especifica el principal para quien los permisos son concedidos o negados por esta entrada ACL. Si a un principal ya le fue aplicado un setPrincipal para esta entrada ACL, es devuelto false, de otra forma se devuelve true.

Declaración:

```
public abstract boolean setPrincipal(Principal user)
```

Parámetros:

`user` - El principal a ser determinado para esta entrada.

Valores que Devuelve:

true si el principal es determinado, *false* si ya un principal fue determinado para esta entrada.

✓ `toString()`

Devuelve una representación en string del contenido de esta entrada ACL.

Declaración:

```
public abstract String toString()
```

Valores que Devuelve:

Una representación en string de los contenidos.

Predomina dentro de:

La clase `Object`.

- **La Interface java.security.acl.Group**

Group es una interface pública

Subclase de *Principal*

Esta interface es utilizada para representar un grupo de principales. (Un principal representa una entidad tal como un usuario individual o una compañía).

Note que Group es subclase de Principal. Además tanto un principal como un grupo pueden ser pasados como un argumento a métodos que contienen un parámetro principal. Por ejemplo, Se puede adicionar ya sea un principal o un

grupo a un objeto Group llamando el método addMember del objeto, pasando el principal o el grupo.

Métodos de la Interface java.security.acl.Group

✓ `addMember(Principal user)`

Adiciona el miembro específico al grupo.

Declaración:

```
public abstract boolean addMember(Principal user)
```

Parámetros:

`user:` El principal a adicionarse a este grupo.

Valores que devuelve:

true si el miembro fue adicionado con éxito, false si el principal fue ya un miembro.

✓ `isMember(Principal member)`

Devuelve true si el principal pasado es un miembro del grupo. Este método hace una búsqueda recursiva, de forma que si un principal pertenece a un grupo el cual es un miembro de este grupo, se devuelve true.

Declaración:

```
public abstract boolean isMember(Principal member)
```

Parámetros:

`member:` El principal a quien se chequeará la acción de ser miembro.

Valores que devuelve:

true si el principal es un miembro de este grupo, de otra forma false.

✓ `members()`

Devuelve una enumeración de los miembros del grupo. los objetos retornados pueden ser instancias ya sea de Principal o Grupo (la cual es una subclase de Principal).

Declaración:

```
public abstract Enumeration members()
```

Valores que devuelve:

Una enumeración de los miembros del grupo.

✓ `removeMember(Principal user)`

Borra al miembro especificado del grupo.

Declaración:

```
public abstract boolean removeMember(Principal user)
```

Parámetros:

`user:` El principal a remover de este grupo.

Valores que devuelve:

true si el principal fue borrado, o false si no era miembro.

- **La Interfece java.security.acl.Owner**

Owner interface pública

Interface para manejo de dueños de listas de control de acceso (ACLs) o configuraciones de ACL. (Note que la interface ACL en el paquete java.security.acl extiende de esta interface Owner.) El dueño inicial Principal debe ser especificado como un argumento al constructor de los clase que esta implementando esta interface.

Métodos de la Interfece java.security.acl.Owner

✓ `addOwner(Principal, Principal)`

Adiciona un dueño. Solo los dueños pueden modificar el contenido del ACL. El llamador(caller) principal debe ser un dueño del ACL para llamar este método. Es decir, solo un dueño puede adicionar otro dueño. El dueño inicial es configurado el tiempo de construcción del ACL.

Declaración:

```
public abstract boolean addOwner(Principal caller, Principal owner)
                                                    throws NotOwnerException
```

Parámetros:

`caller:` El principal que invoca este método. Debe ser un dueño del ACL.

`owner:` El dueño que debe ser adicionado a la lista de dueños.

Valores que Devuelve:

True si se tiene éxito, false si el dueño es ya un dueño.

Excepciones que Arroja:

`NotOwnerException,` Si el llamador principal no es un dueño del ACL.

✓ `deleteOwner(Principal, Principal )`

Borra un dueño. Si éste es el último dueño en el ACL, se produce una excepción.

El llamador principal debe ser un dueño del ACL para que pueda invocar este método.

Declaración:

```
public abstract boolean deleteOwner(Principal caller, Principal owner)
                                throws NotOwnerException, LastOwnerException
```

Parámetros:

`caller:` El principal que invoca este método. Debe ser un dueño del ACL.

`owner:` El dueño a ser borrado de la lista de dueños.

Valores que Devuelve:

True si el dueño es borrado, false si ese dueño no hace parte de la lista de dueños.

Excepciones que arroja:

1. `NotOwnerException.` Si el llamador principal no es un dueño del ACL.
2. `LastOwnerException.` Si solo se ha dejado un dueño, de modo que deleteOwner dejará el ACL sin dueño.

✓ `isOwner(Principal owner)`

Devuelve true si el principal dado es un dueño del ACL.

Declaración:

```
public abstract boolean isOwner(Principal owner)
```

Parámetros:

`owner:` El principal a ser chequeado para determinar si es o no un dueño.

Valores que Devuelve:

true si el principal pasado está en la lista de dueños, false si no.

➢ **La Interface java.security.acl.Permission**

Permission es una interface pública.

Esta interface representa un permiso, tal como el usado para conceder un tipo particular de acceso a un recurso.

Métodos de la Interface java.security.acl.Permission

✓ `equals(Object another)`

Devuelve true si el objeto pasado coincide con el permiso representado en esta interface.

Declaración:

```
public abstract boolean equals(Object another)
```

Parámetros:

`another`: el objeto de permiso que se va a comparar.

Valores que devuelve:

true si los objetos de permiso son iguales, de otra forma false.

Predomina dentro de:

equals en la calase Object

✓ `toString()`

Imprime una representación string de este permiso.

Declaración:

```
public abstract String toString()
```

Valores que devuelve:

la representación en string del permiso.

Predomina dentro de:

`toString` en la clase `Object`

2.2.2. Excepciones

- **La Clase java.security.acl.AclNotFoundException**

```
java.lang.Object
  |
  +----java.lang.Throwable
         |
         +----java.lang.Exception
                |
                +----java.security.acl.AclNotFoundException
```

AclNotFoundException es una clase pública

Subclase de *Exception*

Esta es una excepción arrojada cuando se hace una referencia a una lista de control de acceso (ACL) que no existe.

Constructor de la Clase java.security.acl.AclNotFoundException

- ✓ `AclNotFoundException()`

Construye una AclNotFoundException.

Declaración:

```
public AclNotFoundException()
```

- **La Clase java.security.acl.LastOwnerException**

java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.security.acl.LastOwnerException

LastOwnerException clase pública.

Subclase de *Exception*

Esta es una excepción arrojada siempre que se intenta borrar el último dueño de una lista de control de acceso.

Constructor de la Clase java.security.acl.LastOwnerException

- ✓ `LastOwnerException()`

Construye una LastOwnerException.

Declaración:

public LastOwnerException()

- **La Clase java.security.acl.NotOwnerException**

```
java.lang.Object
  |
  +----java.lang.Throwable
        |
        +----java.lang.Exception
              |
              +----java.security.acl.NotOwnerException
```

NotOwnerException es una clase pública.

Subclase de *Exception*

Esta excepción es arrojada cuando la modificación de un objeto (tal como una ACL) es permitida solamente al dueño del objeto, pero el principal que intenta la modificación no es un dueño.

Cnstructor de la Clase java.security.acl.NotOwnerException

- ✓ `NotOwnerException()`

Construye una NotOwnerException.

Declaración:

```
public NotOwnerException()
```

2.2.3. Cálculo de Permisos Concedidos

Cuando se calcula los permisos de red que a un principal han sido otorgados, se usan las siguientes reglas:

1. Cada principal o grupo pueden tener a lo más una entrada ACL positiva y una negativa; es decir, las entradas ACL múltiples negativas o positivas no son permitidas para ningún principal o grupo.

2. Si no hay entrada para un principal particular o un grupo, entonces el principal o grupo tiene un conjunto de permisos nulo.

3. El conjunto de permisos positivos del grupo de red para un principal es la unión de todos los permisos positivos de cada grupo al cual el principal pertenece.

4. El conjunto de permisos negativos del grupo de red para un principal es la unión de todos los permisos negativos de cada grupo al cual el pincipal pertenece.

5. Si hay una entrada positiva que concede a un principal o un grupo un permiso particular y una entrada negativa que niega al principal o grupo el mismo

permiso, entonces ese permiso es borrado tanto del conjunto de permisos positivos como del conjunto de permisos negativos.

6. Permisos Individuales (permisos concedidos o negados a un principal específico) siempre prevalecen sobre los permisos de grupo. Específicamente, los permisos negativos individuales (negación específica de servicios) prevalecen sobre los permisos positivos de los grupos. Y los permios positivos individuales prevalecen sobre los permisos negativos de los grupos.

7. Asuma que el conjunto de permisos positivos de todos los grupos al cual el principal pertenece es g1 y el conjunto de permisos negativos de todos los grupos al cual el principal pertenece es g2. Asuma también que el conjunto de permisos positivos individuales para el principal es p1 y el conjunto de permisos negativos individuales para el principal es p2 . Entonces los permisos resultantes concedidos al principal son (p1+(g1-p2))-(p2+(g2-p1)).

2.2.4. Ejemplo de Cálculo de permisos

Asuma que un principal P pertenece a los grupos G1 y G2. La tabla siguiente muestra 5 columnas usando algunos ejemplos de permisos dados a G1, G2 y P. Los permisos resultantes concedidos a P son mostrados en la última columna.

	Permisos del Grupo G1	Permisos del Grupo G2	Unión (G1, G2) perms	Permisos Individuales	Permisos Resultantes
Positivo	A	B	A+B	C	A+B+C
Negativo	Nulo	Nulo	Nulo	Nulo	
Positivo	A	B	B	C	B+C
Negativo	-C	-A	-C	Nulo	
Positivo	A	B	A+B	C	B+C
Negativo	Nulo	Nulo	Nulo	-A	
Positivo	A	C	A	B	B
Negativo	-C	-B	-B	-A	

2.3. EL PAQUETE java.security.interfaces

2.3.1. Interfaces

- **Interface java.security.interfaces.DSAKey**

DSAKey interface pública.

Es la interface a una clave DSA publica o privada. DSA (Digital Signature Algorithm) es definido en NIST's FIPS-186.

Métodos de la Interface java.security.interfaces.DSAKey

✓ `getParams()`

Devuelve los parámetros de la clave específicos de DSA. Esos parámetros nunca son secretos.

Declaración:

```
public abstract DSAParams getParams()
```

Valores que devuelve:

Los parámetros de clave específicos del DSA.

- **Interface java.security.interfaces.DSAKeyPairGenerator**

DSAKeyPairGenerator interface pública

Es una interface a un objeto capaz de generar pares de claves DSA. Los métodos de inicialización pueden ser llamados cualquier número de veces. Si no es llamado ningún método de inicialización sobre un DSAKeyPairGenerator, se generan claves de 1024-bit por defecto, usando los parámetros precomputados p, q y g y una instancia de SecureRandom como fuente de bits aleatorios.

Los usuarios que desean indicar parámetros específicos de DSA y generar un par de claves apropiado para usarlo con el algoritmo DSA deben realizar los siguientes pasos:

1. Obtener un generador de claves (instancia) para el algoritmo DSA llamando el método getInstance del KeyPairGenerator con "DSA" como su argumento.

2. Inicializar el generador arrojando los resultados a un DSAKeyPairGenerator y llamando uno de los métodos initialize de ésta interface DSAKeyPairGenerator.

3. Generar un par de claves llamando el método generateKeyPair de la clase KeyPairGenerator.

Nota: No es necesario hacer la inicialización de algoritmo específico para un generador de par de claves DSA. Es decir, no es obligatorio que siempre se llame el método initialize en esta interface. La inicialización independiente del algoritmo usando el método initialize en la interface KeyPairGenerator es todo lo que se necesita cuando se acepta las opciones por defecto para los parámetros del algoritmo específico.

Métodos de la Interface java.security.interfaces.DSAKeyPairGenerator

✓ `initialize(DSAParams, SecureRandom)`

Inicializa el generador de par de claves usando p, q y g, los parámetros de la familia DSA.

Declaración:

```
public abstract void initialize(DSAParams params, SecureRandom random)
                                        throws InvalidParameterException
```

Parámetros:

`params:` Los parámetros a usar para generar las claves.

`random:` La fuente de bits aleatorios a usar para generar bts de clave.

Excepciones que arroja:

`InvalidParameterException,` si los parámetros pasados son inválidos o nulos.

✓ `initialize(int, boolean, SecureRandom)`

Inicializa el generador de par de claves para una longitud de módulo dado, sin parámetros.

Si *genParams* es true, este método generará nuevos parámetros p, q y g. Si es false, el método usará parámetros precomputados para la longitud de módulo solicitada. Si no hay parámetros precomputados para esa longitud de módulo, es arrojada una excepción. Es garantizado que siempre habrá parámetros por defecto para longitudes de módulo de 512 y 1024 bits.

Declaración:

```
public abstract void initialize(int modlen, boolean genParams,
                    SecureRandom random) throws InvalidParameterException
```

Parámetros:

`modlen:` La longitud de módulo, en bits. Los valores válidos son cualquier múltiplo de 8 entre 512 y 1024, inclusive.

`random:` La fuente de bits aleatoria a usar par generar bits de clave.

`genParams:` true o false si se quiere o no generar nuevos parámetros para la longitud de módulo solicitada.

Excepción que arroja:

`InvalidParameterException,` Si la longitud de módulo no está entre 512 y 1024, o si genParams es false y no hay parámetros precomputados para la longitud de módulo solicitada.

- **Interface java.security.interfaces.DSAParams**

DSAParams interface pública

Interface para un conjunto de parámetros de clave específico de DSA, el cual define una familia de claves DSA. DSA (Digital Signature Algorithm) es definido en NIST's FIPS-186.

Métodos de la Interface java.security.interfaces.DSAParams

- ✓ `getP`

Devuelve el primo, p.

Declaración:

```
public abstract BigInteger getP()
```

Datos que devuelve:

El primo, p.

- ✓ `getQ`

Devuelve el subprimo, q.

Declaración:

```
public abstract BigInteger getQ()
```

Datos que devuelve:

El subprimo, q.

- ✓ `getG`

Devuelve la base, g.

Declaración:

```
public abstract BigInteger getG()
```

Valores que devuelve:

La base, g.

- **Interface java.security.interfaces.DSAPrivateKey**

DSAPrivateKey interface pública

Subclase de *DSAKey, PrivateKey*

La interface standard para una clave privada DSA. DSA (Digital Signature Algorithm) es definido en NIST's FIPS-186.

Métodos de la Interface java.security.interfaces.DSAPrivateKey

✓ `getX`

Devuelve el valor de la clave privada, x.

Declaración:

```
public abstract BigInteger getX()
```

Valor que devuelve:

El valor de la clave privada, x.

- **Interface java.security.interfaces.DSAPublicKey**

DSAPublicKey interface pública.

Subclase de DSAKey, PublicKey

Es la interface a una clave pública DSA.

Métodos de la Interface java.security.interfaces.DSAPublicKey

✓ `getY()`

Devuelve el valor de la clave pública, y.

Declaración:

```
public abstract BigInteger getY()
```

Valor que devuelve:

El valor de la clave pública, y.

3. LOS APPLETS DE JAVA

3.1. DEFINICIÓN Y CARACTERÍSTICAS

Cuando de habla de applets, se hace referencia a todos esos programas hechos en lenguaje Java que se ejecutarán en la Web, proporcionando características dinámicas en las páginas donde se cargan, además por ser interpretados pueden correr fácilmente en diferentes plataformas de hardware.

Las primeras páginas en Internet eran estáticas: imágenes y texto, y si se quería que un programa corriera, debía enviarse un archivo de datos al servidor donde ese programa estaba. Se llenaba una forma en la pantalla, hacía clic en el botón **send** y esperaba el resultado.

Hoy, las applets de Java hacen de las páginas Web algo mucho más llamativo, por dos razones:

- Las páginas pueden contener enlaces, imágenes en movimiento con los beneficios de la multimedia, lo que las hace más amigables.

- Con un contenido ejecutable, cualquier transacción (un pedido con una tarjeta de crédito, llenado de una forma, etc.) puede completarse en la máquina del cliente y la transacción resultante es enviada a través de la Web. Esto evita la lentitud que produce el tener que comunicarse con el servidor Web en cada iteración de una transacción.

Existen dos formas para que los browser puedan mostrar el contenido de una página que contiene applets de Java, y son distinguibles debido a la herramienta utilizada para el desarrollo del navegador.

Es el caso por ejemplo de HotJava y Netscape ambos consultan páginas Web, escritas en HTM, de los servidores de la Web en Internet. Como HotJava está escrito en lenguaje Java emplea la API y el sistema de tiempo de ejecución de Java para mostrar las páginas web al usuario.

Nestcape, en cambio esta escrito, en C++, por lo que emplea funciones de C++ para mostrar los documentos HTML.

Cuando cualquiera de ambos navegadores encuentra una etiqueta miniaplicación (Applet tag) en un archivo HTML, solicita del servidor de la web el archivo de bytecode Java necesario para ejecutar el applet. HotJava carga y ejecuta el archivo de bytecode empleando el sistema de tiempo de ejecución Java. Netscape

ejecuta el archivo de bytecode mediante la versión que lleva incorporada del sistema de tiempo de ejecución Java.

En la *figura 2* se muestra el proceso anteriormente descrito:

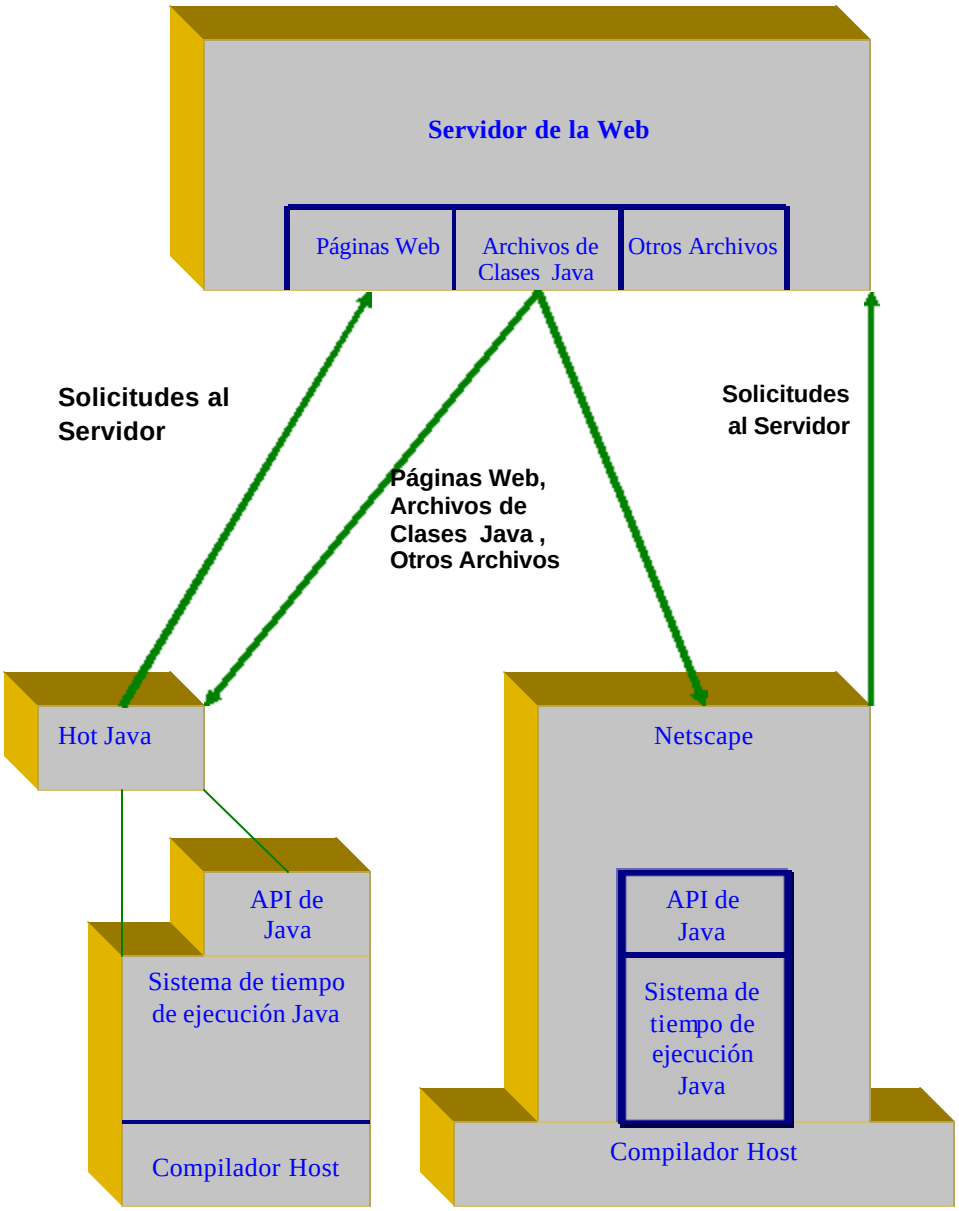


Figura 2. Cómo dan soporte HotJava y Netscape 40 a las miniaplicaciones Java

Algunas de las razones por las que las primeras web no usaban contenido ejecutable en el cliente eran las siguientes:

1. Se podrían cargar virus muy peligrosos que afectarán gravemente al sistema. Con contenido ejecutable usted no podrá descubrir que podría estar bajando código potencialmente peligroso.

2. La no portabilidad de algunas aplicaciones, lo cual atentaría con uno de los encantos de la web como lo era que cualquier sistema cliente que se eligiera estaría capacitado para bajar páginas ejecutándose en un sistema totalmente diferente.

Java hace posible el contenido ejecutable mientras resuelve los anteriores problemas mediante los siguientes componentes:

- Una máquina virtual Java (JVM) diseñada para prevenir que el código bajado (usualmente llamado applet) no sea estropeado por el sistema cliente. El applet corre en un espacio protegido, conocido informalmente como sandbox y tiene solamente un acceso limitado y estrictamente controlado al sistema circundante.

- Un conjunto de bytecodes - instrucciones de máquina virtual - las cuales son interpretadas por el JVM. Estos son necesarios para prevenir que el applet salga del sandbox. Dichos bytecodes tienen un beneficio propio ya que son independientes de la máquina, si usted tiene una JVM para su estación de trabajo, entonces puede correr cualquier applet de cualquier servidor.

- Un lenguaje de alto nivel, orientado a objetos en el cual escribir las clases que harán las applets. Este es un lenguaje similar en muchas formas a C++ con algunas funciones (como punteros) omitidas ya que ellas podrían usarse para escapar del "sandbox".

En estos momentos existe el JDK (el Kit del Desarrollador en Java), el cual contiene el JVM, el compilador, las clases básicas y está disponible para cualquier plataforma. Además los navegadores de web incluyen la máquina virtual de Java (JVM), lo cual hace que el contenido ejecutable sea hoy una realidad.

3.2. INCIDENCIA DE LOS APPLETS EN LA SEGURIDAD DEL SISTEMA

Los applets de Java pueden ser de gran ayuda para la seguridad de un sistema y puede fortalecer enlaces débiles, teniendo en cuenta que la distribución de código es un proceso de riesgo.

Muchas aplicaciones necesitan código ejecutándose en el cliente en cooperación con código ejecutándose en el servidor (como es el caso de marcadores a conexiones de redes telefónicas) y ese código debe estar instalado en este de alguna forma. La distribución de ese código es de algún modo un enlace débil en un sistema Online lo cual es mucho más fácil de atacar que desencriptar mensajes que fluyen a través de la web.

El peligro consiste en que se este código puede ser dañado, por ejemplo un número telefónico puede ser cambiado para que el cliente marque el sitio atacante en vez del servidor correcto. El cliente nunca descubrirá esto, ya que el atacante se adelanta al trafico de información entre el cliente y el servidor, leyéndola a medida que esta pasa. Igualmente podría ser introducido un virus. A diferencia de un download normal, las applets de Java son chequeadas a su llegada al browser y más aún pueden ser firmadas. La verificación y firma de los applets de Java son parte esencial para la seguridad de Java.

Existen 3 componentes para el chequeo de Applets:

1. El *Class Loader* es responsable de juntar las diferentes partes del programa de modo que pueda ser ejecutado.

2. El *Class File verifier* (el cual incluye el verificador de bytecode) verifica que el programa obedezca las reglas de la máquina virtual de Java (pero note que

esto no necesariamente significa que obedezca las reglas del lenguaje Java).

3. El *Security Manager* impone restricciones locales en las cosas que al programa se le permite hacer, es perfectamente posible personalizar esta parte para permitir a los applets acceso limitado a recursos controlados cuidadosamente, pero en la práctica, los vendedores de browsers han implementado una versión altamente restrictiva que proporciona Sun. Esta no permite acceso al sistema de archivos locales y acceso a redes solo a la localidad de la cual el applet o su página web vino.

La forma para permitir más amplio acceso es por medio de applets firmadas de JDK 1.1. Usted puede desear, por ejemplo, imprimir algo desde un applet. Es poco probable que quiera que el manejador de seguridad permita a cualquiera hacer esto, pero podrá permitirle el acceso especialmente a personas dignas de confianza. Entonces usted baja el applet; descubre que es encriptada con la clave privada de alguien; verifica el certificado de clave pública que te acompaña, para estar seguro que es valida, y define a alguien especialmente de confianza; desencripta el applet con la clave publica y luego le permite el acceso necesario.

Una cosa importante que distingue a Java de otras formas de contenido ejecutable, es que esta tiene tanto Web de confianza que permiten las firmas y tres componentes de seguridad para validar el código bajado. Estas precauciones son tomadas, no porque usuarios de Java sean menos fiables que otros, sino porque aún los más confiables proporcionadores de código algunas veces

cometen errores o pueden tener sus sistemas comprometidos. Sin la validación, una página Web de confianza puede llegar a ser una de página Web de alteraciones si alguno de los sitios confiables es crakeado con éxito.

Debido a que Java tiene la más fuerte seguridad para contenido ejecutable, los especialistas en este campo ven esta herramienta como un reto y se dedican a encontrar huecos en ella a través de applets de ataque, las cuales pretenden hacer daño en el sistema que las cargue.

Hasta el momento, todos los applets de ataque reportados, son hechos por especialistas, y no por atacantes peligrosos. Estas personas, al encontrar el hueco, reportan a *Sun* el resultado de la investigación y la cura, luego de haberse proporcionado la solución al error, entonces, es publicada la falla en Java, para que los usuarios se actualicen.

Existe sin embargo una amenaza contra la cual Java no tiene defensas fuertes. La verdadera esencia de Java es que es una programa que viene de un servidor, es cargado en la máquina cliente, con poca intervención del usuario.

Que tal si es un programa indeseable? Existen applets maliciosos que pueden usar mucho el tiempo de máquina del sistema cliente de modo que este no puede funcionar normalmente. Como este tipo de molestias no pueden ser prevenidas, todo lo que se puede hacer es descubrir el responsable tomar acción después del evento.

3.2. APPLETS Y CRIPTOGRAFIA

Existen Applets las cuales son consideradas como un bien o activo, piezas de propiedad intelectual, las cuales necesitan ser protegidas de intrusos.

La mayoría de las applets que se encuentran en la web están disponibles para cualquiera, de forma gratuita. Usualmente , el dueño de la página web las usa para hacer más atractivas las páginas o programar una función que se quiera usar, tal como una aplicación de plan de inversión comprometida a hacer más competitiva una empresa. Algunas veces, sin embargo, es el usuario quien usa el applet, el cual puede ser un juego o una revista famosa que se quiera leer. En este caso el dueño del applet desea cobrar por su uso.

Si se es dueño de un applet, existen tres obstáculos por vencer:

1. Si desea enviar el applet a otra persona en una forma protegida, de modo que nadie - incluyendo esa persona - pueden ejecutarla. Además se quiere enviar información acerca de cuanto se desea cobrar para su uso, lo que ella hace es información adicional (técnicamente se llama Metadatos)

2. El dueño del applet debe estar preparado para aceptar alguna forma de pago del cliente que quiere usar el applet. Se debe permitir pagar diferentes sumas,

dependiendo de cuanto se desea usar.

Por ejemplo, se podría cargar una sola suma para uso ilimitado, otra suma diferente para un solo uso y otra servirá para usar el applet con un determinado periodo de tiempo.

3. Se debe estar preparado para conceder los derechos de uso para los cuales se ha pagado sin permitir ningún derecho adicional que no se halla pagado.

Por supuesto, se podría encriptar el código del applet y vender la clave, que permitirá desencriptarla. Lo anterior reuniría los requerimientos 1 algo del 2 y quedaría corto para el 3, sin embargo, a penas se haya desencriptado los archivos de clase del applet, se estaría en libertad para distribuirlos a través de los amigos, además de esta falla fundamental, hay algo profundamente insatisfecho a cerca del modelo de pago, el cual carece de perspicacia y flexibilidad: se tiene también el código y puede ser usado a favor o no.

3.3.1. Cómo firmar Applets para Netscape Communicator

Lo primero que se necesita es hacerse con las herramientas necesarias. En primer lugar, hay que notar que para poder firmar un applet que va a acceder a recursos restringidos, como el sistema de ficheros, propiedades del sistema o conexiones de red, se necesita alterar el código fuente del applet, incluyendo llamadas a ciertas funciones para garantizar los privilegios requeridos. Estas clases

constituyen lo que se denomina el *Netscape Java Capabilities API.* Por otro lado, se requiere la herramienta de firmado, conocida como *SignTool 1.1*. Una vez armados con estas herramientas, se puede proceder ya a firmar applets, siguiendo los siguientes pasos.

- **PASO 1: CREAR UN CERTIFICADO PARA FIRMAR APPLETS**

El primer paso consiste también en adquirir un certificado adecuado de una autoridad de certificación. Hay que asegurarse de que la autoridad elegida emite certificados que permitan firmar software, para uso personal se debe solicitar un certificado de clase 2 y mientras que para uso comercial se necesita uno de clase 3, lógicamente que incurre en un costo la obtención de este certificado solo basta en dirigirse a ellos y le dirán cuanto le puede costar anualmente este.

Existen muchas autoridades de certificación (CA) que emiten certificados, una de las más reconocidas es, *VeriSign*. Se pueden consultar más autoridades propuestas por Netscape, aptas para certificar sus productos.

En este caso, y como sólo se necesita para hacer pruebas, se creará un certificado de prueba, que es gratis, utilizando la citada herramienta SignTool 1.1.

Antes de instalar claves y certificados nuevos en las bases de datos de Netscape Communicator, es imprescindible crear una contraseña para acceder a la base de

datos. Para hacerlo, se pulsa el botón de Seguridad en la barra de herramientas de Communicator, se elige *Passwords* y después se pulsa el botón de *Establecer Passwords* para crearlo como se muestra en la *figura 3.*

Figura 3. Ventana de la Sección de Seguridad del Navegador Netscape: Elección de la Contraseña para el certificado

Otro aviso importante es que para ejecutar SignTool se debe cerrar antes todas las ventanas abiertas de Communicator, ya que en caso contrario se corre el riesgo de corromper las bases de datos de claves y certificados.

Para generar un certificado de prueba se utiliza la opción **-G** seguida de un nombre y la opción **-d** para indicar el directorio donde se encuentra la base de datos de certificados (el fichero cert7.db) y claves (el fichero key3.db) de

Communicator. Por ejemplo, para crear un certificado para ***antonio*** de la ***CUTB***, se puede escribir lo siguiente:

```
signtool -G antonio -d c:\netscape\users\tony20r
using certificate directory: c:\netscape\users\antonio
Enter certificate information. All fields are optional. Acceptable
characters are numbers, letters, spaces, and apostrophes.
certificate common name: Certificado para pruebas
organization: CUTB
organization unit: RR
state or province: Cartagena
country (must be exactly 2 characters): CO
username: antonio
email address: antoniorev@latinmail.com
Enter Password or Pin for "Communicator Certificate DB": [no se produce
eco]
generated public/private key pair
certificate request generated
certificate has been signed
certificate "antonio" added to database
Exported certificate to x509.raw and x509.cacert.
```

Si en el paso anterior se especificó la opción **-d** junto con el camino donde se encuentra la base de datos de claves y de certificados de Communicator, la base ya ha quedado automáticamente actualizada. La próxima vez que se arranque Communicator, si se pulsa el botón de *Seguridad (Security))*de la barra de herramientas, en *Certificados (Certificates)* y a continuación en *Propios (Yours)* se comprobará que aparece el recién creado, en la *figura 4* se muestra qué se verá en la ventana .

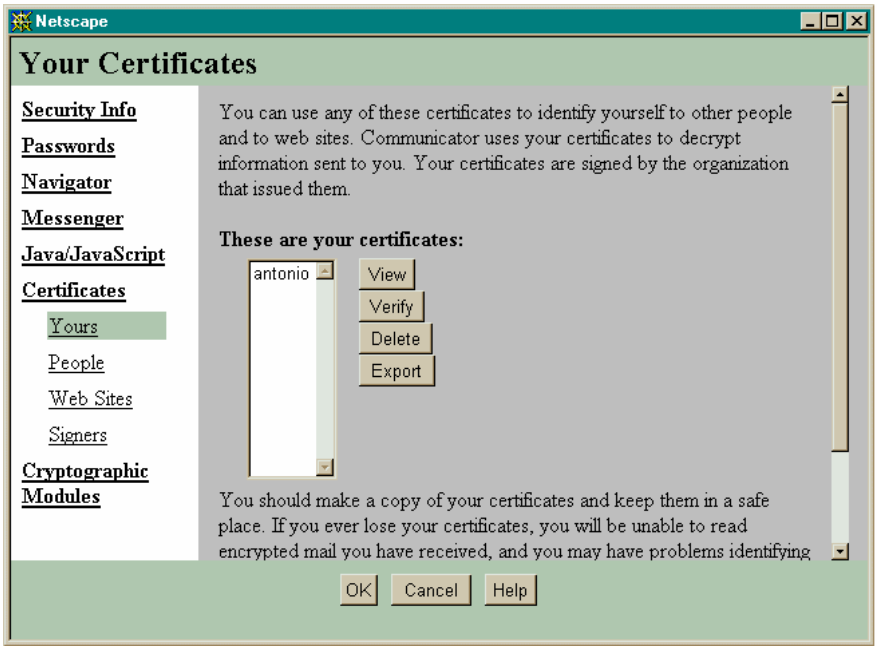


Figura 4. Ventana de la sección de Seguridad del Navegador Netscape: Muestra los Certificados creados por el usuario

En el momento que el applet se cargue en el Browser y desea confirmar el dueño del Certificado le aparecerá una ventana parecida a la mostrada en la *figura 5.*

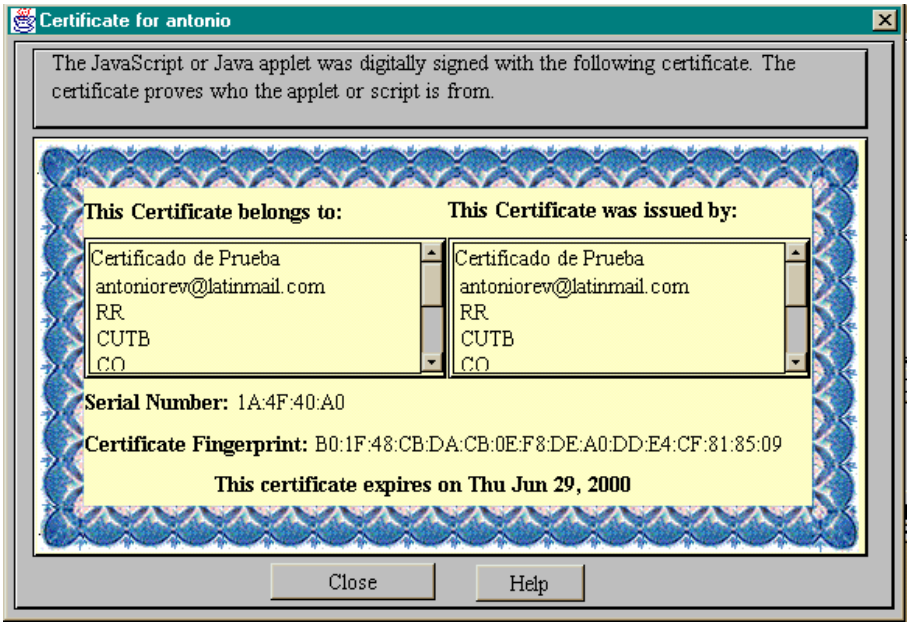


Figura 5. Ventana que muestra la información del Certificado cuando se carga el applet en el Navegador Netscape.

Se puede comprobar que el certificado se ha instalado correctamente también mediante ***signtool,*** con la opción **-l:**

```
signtool –l antonio –d c:\nestcape\users\tony20r
using certificate directory: c:\netscape\users\tony20r
Object signing certificates
-------------------------------------
antonio
    Issued by: antonio (Certificado de Prueba)
    Expires: Thu Jun 29, 2000
-------------------------------------
For a list including CA's, use "signtool -L"
```

Con este certificado ya se está listo para firmar applets. Ahora bien, no basta con firmarlos. Para que los usuarios puedan ejecutar applets firmados es necesario que instalen en su versión de Communicator el certificado que se acaba de generar. Si se hace un listado del directorio donde se está trabajando, se observará que *signtool* ha creado dos ficheros automáticamente, **x509.cacert** y **x509.raw.** El primero de ellos contiene el certificado en formato *base64.* Para que quede a disposición de cualquiera que lo necesite, se seguirán los siguientes pasos:

- **Crear un enlace al fichero x509.cacert en una página Web**

  Por ejemplo, puede añadir la siguiente línea en HTML a su página personal para que los visitantes que deseen ejecutar sus applets sepan dónde obtener su certificado:

  Primero debes instalar mi certificado.

- **Asegurarse de que el servidor Web exporta el fichero como MIME-type application/x-x509-ca-cert**

  Los servidores de Netscape están configurados así por defecto y también algunos servidores Unix. Cosa que no sucede con el Internet Information Server de NT, por lo que habrá que configurarlo para que asocie ese tipo MIME a la extensión *.cacert*.

- **Instalar el certificado**

  Una vez que esos pasos han sido seguidos, basta con pinchar en el enlace anterior para que automáticamente se lance el proceso de instalación de certificados de Netscape. Siguiendo las instrucciones del proceso, muy sencillo, se instalará el certificado en el navegador del usuario, que podrá a partir de ahora ejecutar correctamente todos los applets firmados con este certificado.

♦ **PASO 2: CREAR UN APPLET CON PRIVILEGIOS**

Antes de traspasar los límites del recinto de seguridad, los applets tienen que pedir permiso educadamente. La manera como se consigue es mediante el API de capacidades. Cada vez que un applet quiere efectuar alguna acción restringida, debe pedir permiso al navegador para ver si éste se lo concede, para lo cual se sirve de las funciones contenidas en esta API. Cuando el applet le pide permiso al

navegador, éste busca en su base de datos si existe una entrada para el firmante del applet y si la acción le está permitida. Caso de no existir ninguna entrada, el navegador presenta una ventana pidiendo confirmación antes de conceder el permiso. El usuario decide si garantiza o deniega el permiso, con la posibilidad de almacenar la decisión en la base de datos para futuras referencias.

La forma de modificar el applet consiste en preceder todas las llamadas a recursos protegidos de una llamada al gestor de privilegios.

Por ejemplo, para solicitar privilegios antes de leer propiedades del sistema, la función en Java que lee los privilegios irá precedida de:

```
PrivilegedManager.enablePrivilege("UniversalPropertyRead");
```

Para leer/crear/escribir ficheros, se precedería el bloque de funciones correspondiente por:

```
PrivilegedManager.enablePrivilege("UniversalFileAccess");
```

Y así con distintas llamadas para cada tipo de recurso. Una vez terminadas las operaciones que requieren privilegios, constituye una buena práctica revocar los privilegios concedidos, utilizando la siguiente función:

```
PrivilegeManager.revertPrivilege("UniversalPropertyRead");
```

Al final hay un listado de las diferentes páginas web que puedes visitar para que te informes más al respecto sobre este tema.

El mayor inconveniente de este enfoque es que exige la modificación del applet y además, como consecuencia de los cambios introducidos, no funcionará ya en ningún otro navegador, tirando por tierra la filosofía de Java de ***"Escribir una vez, ejecutar en cualquier sitio".***

- **PASO 3: FIRMAR Y EMPAQUETAR LOS FICHEROS CON LAS CLASES**

Para firmar el fichero con todas las clases compactadas que emplee el applet, primero se crea un directorio al que se copian todas las clases que componen el applet y a continuación se firma el directorio utilizando la herramienta ***SignTool***.

La propia herramienta se encarga de generar un fichero comprimido con todas las clases y demás ficheros que se hayan copiado en el directorio. Este fichero recibe el nombre de Archivo Java ***(JAR)***. En versiones anteriores esta herramienta se llamaba *zigbert,* pero con la versión actual de *SignTool* ha quedado obsoleta.

Por ejemplo, se tiene un applet llamada ***AppletConfiable.java*** y en el caso supuesto de que se quiera firmar su archivo compilado ***AppletConfiable.class*** con todas la clase que este applet emplee, se copiarán a un directorio al que se llamará ***EmpleoClases*** y se ejecutará el programa ***SignTool,*** pasándole la clave del certificado almacenada en la base de datos de Communicator, que se acaba de generar en el *paso 1.* Por ejemplo:

```
signtool -k antonio –d c:\netscape\users\tony20r -Z EmpleoClases.jar
EmpleoClases
using certificate directory: .
```

Alguna parte del proceso se verá como:

```
Generating abednego/META-INF/manifest.mf file..
--> AppletConfiable.class
adding AppletConfiable/jar to EmpleoClases.jar...(deflated 0%)
Generating zigbert.sf file..
Aquí pide el password:
Enter Password or Pin for "Communicator Certificate DB": [no se produce
eco]
Si se ha introducido correctamente, continúa:
adding EmpleoClases/META-INF/manifest.mf to EmpleoClases.jar...(deflated
14%)
adding EmpleoClases/META-INF/zigbert.sf to EmpleoClases.jar...(deflated
27%)
adding EmpleoClases/META-INF/zigbert.rsa to EmpleoClases.jar...(deflated
43%)
tree " EmpleoClases" signed successfully
```

Ahora ya está disponible el fichero ***EmpleoClases.jar*** para distribuirlo con su firma incorporada. Se puede verificar que el proceso se ha realizado correctamente utilizando la opción -**v:**

```
signtool -v EmpleoClases.jar
using certificate directory: .
archive "EmpleoClases.jar" has passed crypto verification.
status path

----------------------------

verified AppletConfiable.class
```

- **PASO 4: INCRUSTAR EL JAR EN LA PÁGINA WEB Y LEER EL ARCHIVO EN EL DISCO DESDE LA MISMA PÁGINA**

En este caso la etiqueta <applet> también cambia ligeramente para permitir la inserción del archivo JAR y leer el archivo desde disco, supongamos que ese archivo sea *prueba.txt*:

```
<applet code=AppletConfiable.class  archive=EmpleoClases.jar  width=480
height=200>
<PARAM name="Fichero" value="c:\prueba.txt">
</applet>
```

Es importante eliminar el fichero `AppletConfiable.class` original del directorio, ya que en caso contrario el navegador lo cargaría, en lugar del firmado. Ahora ya se puede visualizar correctamente el applet en Communicator. En cada ocasión en la que el applet necesite acceder a un recurso protegido, al usuario se le presentará una ***ventana de advertencia,*** en la que se le informa de los permisos que está solicitando ese applet, acompañados de una estimación del riesgo potencial derivado de su concesión. Siempre y cuando no se active la casilla para recordar la decisión, esta ventana aparecerá cada vez que el applet intente acceder de nuevo a ese recurso.

En las *figuras 6 y 7* se muestran las ventanas de advertencias del Navegador Netscape Commnicator cuando se carga un applet que intenta acceder a los recursos del sistema:

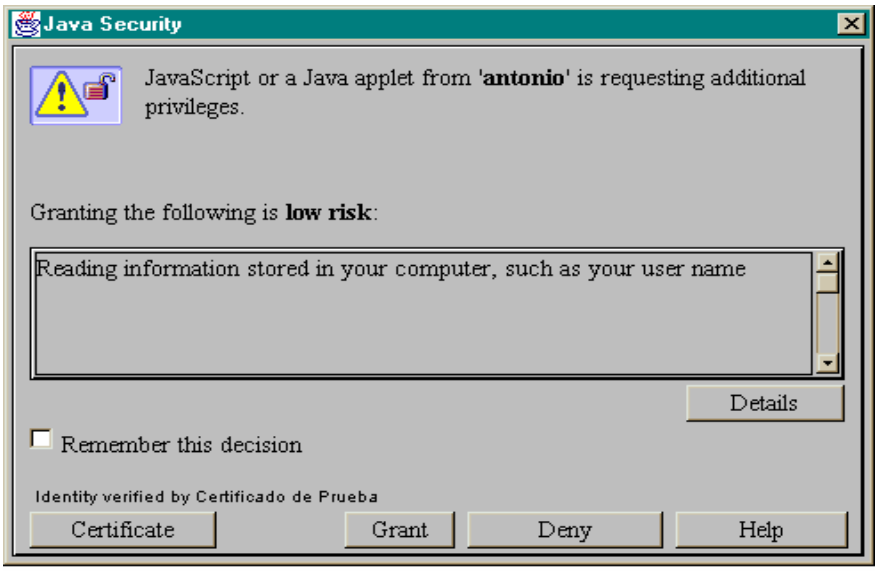


Figura 6. Ventana de Advertencia del Navegador Netscape: el applet intenta leer información dentro del Sistema como el ***nombre de usuario.***

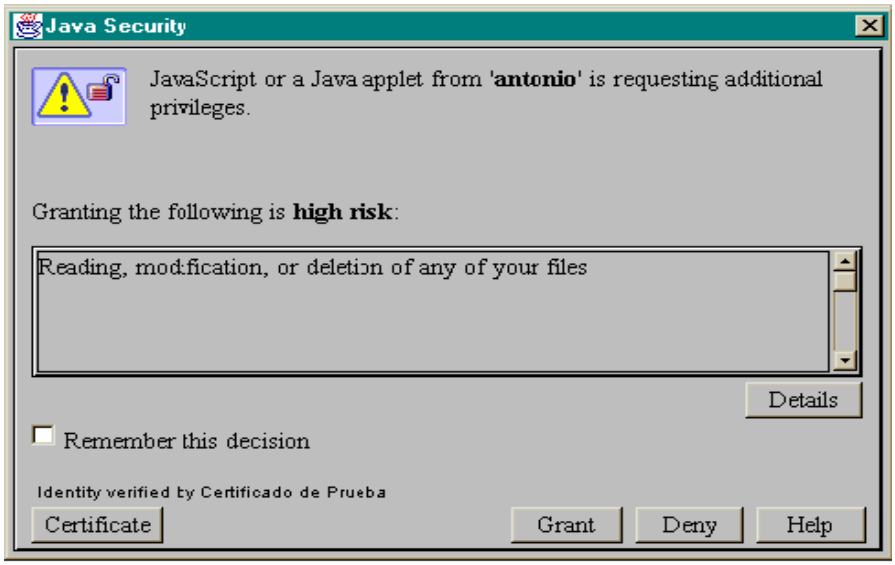


Figura 7. Ventana de Advertencia del Navegador Netscape: el applet intenta leer, modificar o borrar cualquier archivo dentro del Sistema

En la *figura 8* se muestra el applet cargada en el Navegador Nestcape después de haberle otorgado permiso el usuario.

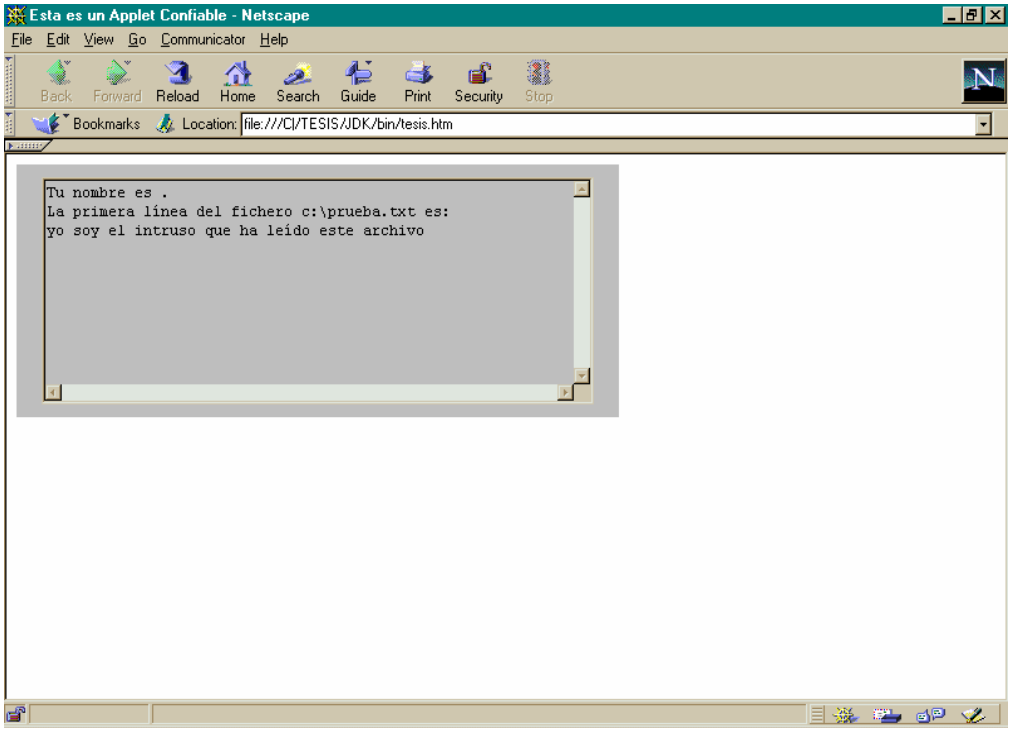


Figura 8. Ventana del Navegador Netscape: se muestra el applet cargada habiéndole otorgando el usuario permiso para hacerlo.

3.3.2. No todo son Ventajas

A pesar de que la firma digital de applets supera las drásticas limitaciones impuestas por el modelo del recinto de seguridad de Java, se presentan varios obstáculos para su implantación:

- Los distintos navegadores utilizan distintos esquemas de firmado, mutuamente incompatibles. Este inconveniente se puede superar creando applets firmadas para los dos navegadores y detectar en la página qué tipo de navegador se utiliza, presentándole un applet u otro, aunque aumenta la complejidad.

- Las herramientas de firmado son bastante crípticas y complicadas de utilizar. La única solución consiste en que el desarrollador se arme de paciencia y trabaje con precisión.

- Los usuarios pueden verse abrumados por tantas ventanas cuyo significado a menudo no comprenden con claridad. Debería explicárseles con sencillez qué permisos necesita de verdad un applet para hacer qué cosa y requerir el mínimo de interacción por su parte. Cuanto más granular sea el acceso a los recursos, menor será el posible daño en caso de que el usuario garantice más permisos de los debidos.

- Por último, es importante resaltar que el hecho de que un applet esté firmado no implica que sea seguro, sino simplemente que se puede identificar al autor. En el caso de un autor desconocido, persiste la incertidumbre al decidir si se le concederán o no privilegios para acceder a recursos restringidos en la máquina del usuario.

4. APLICACIONES

4.1. APLICACIÓN PARA EL CÁLCULO DE FUNCIONES HASH (con el paquete java.security)

Incluimos a continuación un ejemplo sencillo que calcula la *función Hash* de un vector de bytes. El programa define la clase **EjecutaHash** donde se inicializa el vector de bytes para el que se calculará la *función Hash* (metodo IniciaMensaje) y se calcula dicha función (método CalculaHash):

```
import java.security.*;
import java.io.*;
class EjecutaHash{
    private static void IniciaMensaje(byte mensaje[], int tamaño){
        for (int i=0;i<tamaño;i++){
            mensaje[i]=2;
        }
    }
    public static byte[] CalculaHash(byte mensaje[],String algoritmo, int tamaño) throws
                                        NoSuchAlgorithmException, DigestException{

        MessageDigest prevHash;
         byte[] hash = new byte[16];
         IniciaMensaje(mensaje, tamaño);
         prevhash=MessageDigest.getInstance(algoritmo);
         prevHash.update(mensaje);
         hash=prevHash.digest();
         System.out.println("\n"+ prevHash.toString());
         return hash;
     }

     public static void main(String args[]) throws IOException,
                                        NoSuchAlgorithmException ,    DigestException {

         byte m1[]=new byte[1024];
         byte prueba[]= new byte[16];
         prueba= CalculaHash(m1,"MD5",1024);
     }
```

```
}// fin de la clase EjecutaHash
```

Lógicamente, en una versión más realista los datos podrían obtenerse de un fichero - que hemos denominado "datos.txt" - e ir almacenándose en el vector a medida que se van leyendo:

```
FileInputStream f = new FileInputStream("datos.txt");
while (f.available( )> 0) {
   prevHash.update( (byte) f.read( ));
}
```

Otra cuestión interesante es generar un fichero de salida con la función Hash del mensaje, por si deseamos transmitirla a través de la red o almacenarla en un lugar seguro. En general, la escritura de objetos en un fichero - si no deseamos preocuparnos por su formato - puede llevarse a cabo fácilmente, si estos admiten serialización, mediante la clase **ObjectOutputStream.**

```
try {
     FileOutputStream fHash = new FileOutputStream("Hash.dat");
     ObjectOutputStream ost = new ObjectOutputStream(fHash);
     ost.write(hash);
     ost.flush(); //vaciar el buffer
     ost.close();
     fHash.close();
}
catch (IOException e) {
            System.err.println(e);
}
```

4.2. APLICACIÓN PARA LA GENERACIÓN DE CLAVES (con el paquete java.security)

Veamos ahora, un programa ejemplo que genera un par de claves: privada y pública y las almacena en fichero para poderlas utilizar posteriormente con el fin que se desee: firmar otros ficheros o verificar firmas.

La aplicación se ha estructurado como una única clase **generadorClaves** que posee 3 métodos: **VisualizaClave, CreaFicherosDeClaves** y **main**.

- **VisualizaClave:** Permite imprimir en hexadecimal las clave generadas.

- **CreaFicherosDeClaves:** Crea dos ficheros: ClavePublica.dat y ClavePrivada.dat y almacena en ellos las claves pública y privada respectivamente. Se ha elegido la opción de serializar los objetos PublicKey y PrivateKey y almacenarlos, puesto que después a la hora de generar una firma digital los métodos initSign e initVerify emplean como parámetros objetos de tipo PrivateKey y PublicKey respectivamente.

- **Main:** Este método es el más interesante, puesto que aquí se genera la pareja de claves y se obtiene, mediante el método *getEncoded* la clave pública codificada como un array de bytes, lo que nos permite visualizarla por pantalla.

El primer paso es conseguir un objeto generador de parejas de claves para generar las claves con el algoritmo de firma DSA.

```
KeyPairGenerator GenClaves = KeyPairGenerator.getInstance ("DSA");
```

El paso siguiente es inicializar el generador de parejas de claves. Todos los generadores de parejas de claves comparten los conceptos de "fuerza" y aleatoriedad. El generador de parejas de claves DSA puede emplear uno o los dos argumentos.

La "fuerza" para el generador de una clave DSA es la longitud de la clave (en bits). La fuente de aleatoriedad debe ser una instancia de la clase Securerandom.

```
GenClaves.initialize(512, new SecureRandom());
```

El paso siguiente es generar la pareja de claves y almacenarla en una instancia de la clase KeyPair.

```
KeyPair ParClaves = GenClaves.generateKeyPair();
```

Se han utilizado después los dos métodos de la clase KeyPair: getPublic y getPrivate, que permiten obtener las claves pública y privada, y codificarlas como un array de bytes mediante el método getEncoded para poderlas visualizar. (En nuestro caso el programa sólo visualiza la clave pública).

A continuación se muestra el código completo del programa:

```
import java.security.*;
import java.io.*;
import java.lang.*;

//Crea ficheros de claves y visualiza la clave pública por pantalla
class generadorClaves {
        public static void VisualizaClave(byte[] clave) {
         int longClave = clave.length,i;
         String s = new String();
            for (i=0; i<longClave; i++) {
                    String k= "00"+ Integer.toHexString((int)clave[i]);
                    k=k.substring(k.length()-2,k.length());
                    s=s+":" + k;
               }
                     System.out.println(s);
        } // fin VisualizaClaves

public static void CreaFicherosDeClaves(KeyPair Claves){
```

```
            PublicKey ClavePublica = Claves.getPublic();
            PrivateKey ClavePrivada = Claves.getPrivate();

           //f1 fichero en el que escribiremos la clave publica
            try{
                  FileOutputStream f1=new FileOutputStream("ClavePublica.dat");
                  ObjectOutputStream ost = new ObjectOutputStream(f1);
                  ost.writeObject(ClavePublica);
                  ost.flush(); //vaciar el buffer
                  ost.close();
                   f1.close();

                   //f2 fichero en el que escribiremos la clave privada
                   FileOutputStream f2=new FileOutputStream("ClavePrivada.dat");
                   ObjectOutputStream ost2 = new ObjectOutputStream(f2);
                   ost2.writeObject(ClavePrivada);
                   ost2.flush(); //vaciar el buffer
                   ost2.close();
                   f2.close();
              }
              catch (IOException e) {
                        System.err.println(e);
              }
} //Fin CreaFicherosDeClaves

public static void main (String a[]) throws NoSuchAlgorithmException, IOException {

            KeyPairGenerator GenClaves = KeyPairGenerator.getInstance ("DSA");
           GenClaves.initialize(512, new SecureRandom()); //inicializo el //generador con 512
           KeyPair ParClaves = GenClaves.generateKeyPair(); //genero pareja de //claves
           PublicKey ClavePublica = ParClaves.getPublic();
           PrivateKey ClavePrivada = ParClaves.getPrivate();
           byte[] encKey = ClavePublica.getEncoded();
           System.out.println("Formato clave:" + ClavePublica.getFormat());
           VisualizaClave(encKey);
           CreaFicherosDeClaves(ParClaves);
} //fin main

}//fin de la clase generadorClaves
```

4.3. APLICACIÓN PARA LA GENERACIÓN DE PERMISOS DE USUARIO (con el paquete java.security.acl)

Nota: Este programa de ejemplo es hecho con el propósito de mostrar algunas de las cosas que pueden ser realizadas con una implementación de las interfaces de java.security.acl. Este ejemplo usa la implementación suministrada por el paquete sun.security.acl.

```
import java.security.Principal;
import java.security.acl.*;
import sun.security.acl.*;
import java.util.Enumeration;

public class AclEx {

  public static void main(String argv[])
    throws Exception
  {

    Principal p1 = new PrincipalImpl("usuario1");
    Principal p2 = new PrincipalImpl("usuario2");
    Principal owner = new PrincipalImpl("dueño");

    Permission read = new PermissionImpl("READ");
    Permission write = new PermissionImpl("WRITE");

    System.out.println("Creando un grupo nuevo con dos miembros: usuario1 y
     usuario2");
    Group g = new GroupImpl("grupo1");
    g.addMember(p1);
    g.addMember(p2);

    //
    // crea una nueva acl con el nombre "ejemploAcl"
    //
    System.out.println("Creando una nueva Acl llamada ' ejemploAcl '");
    Acl acl = new AclImpl(owner, " ejemploAcl ");

    //
    // Permite al grupo todos los permisos
    //
    System.out.println("Creando una nueva entrada ACL en ejemploAcl para el grupo, ");
    System.out.println("  Con permisos de lectura & escritura");
    AclEntry entry1 = new AclEntryImpl(g);
    entry1.addPermission(read);
    entry1.addPermission(write);
```

```
    acl.addEntry(owner, entry1);

    //
    // Quita los permisos de escritura ( WRITE) para
    // usuario1. Todos los demás aun tienen
    // privilegios de escritura (WRITE) .
    //
    System.out.println("Creando una nueva entrada Acl en ejemploAcl para usuario1");
    System.out.println("  sin permiso de escritura");
    AclEntry entry2 = new AclEntryImpl(p1);
    entry2.addPermission(write);
    entry2.setNegativePermissions();
    acl.addEntry(owner, entry2);

    //
    // Esta es una enumeración de
    // interfaces de permisos. Debe devolver
    // solo permiso de lectura "READ".
    Enumeration e1 = acl.getPermissions(p1);
    System.out.println("Los permisos para el usuario1 son:");
    while (e1.hasMoreElements()) {
        System.out.println("  " + e1.nextElement());
        };

    //
    // Esta enumeración debe tener permisos de lectura"READ" y escritura "WRITE".
    //
    Enumeration e2 = acl.getPermissions(p2);
    System.out.println("Los permisos para el usuario2 son:");
    while (e2.hasMoreElements()) {
        System.out.println("  " + e2.nextElement());
        };

    // Esto debe devolver false.
    boolean b1 = acl.checkPermission(p1, write);
    System.out.println("usuario1 tiene permiso de escritura "write": " + b1);

    // Todo esto debe devolver verdadero "true";
    boolean b2 = acl.checkPermission(p1, read);
    boolean b3 = acl.checkPermission(p2, read);
    boolean b4 = acl.checkPermission(p2, write);
    System.out.println("El usuario1 tiene permiso de lectura: " + b2);
    System.out.println("El usuario2 tiene permiso de lectura: " + b3);
    System.out.println("El usuario2 tiene permiso de escritura: " + b4);
  }
}
```

4.4. IMPLEMENTACION DE APPLET FIRMADA PARA EL NAVEGADOR DE WEB NETSCAPE

Aquí se muestra el código del applet *AppletConfiable.java* la cual fue firmada con la herramienta Signtool 1.1, para el navegador Netscape. Este applet accede al sistema cuando se le otorga el permiso por parte del usuario.

```
//
// AppletConfiable: applet que va más allá de los límites del recinto de seguridad de Java
//
// Para ello, utiliza el API de Capacidades de Netscape. Debido al uso de estas clases,
// este applet sólo funcionará con Netscape Communicator
// Se importan las clases que se necesitarán
import java.applet.*;
import java.awt.*;
import java.io.*;
import java.lang.*;
import netscape.security.*;
/* Aquí se importa el Netscape Capabilities API. El fichero capsapi_classes.zip con las clases del
API
 debe estar presente en el camino, para que sea encontrado */

public class AppletConfiable extends Applet {
    private TextArea display;

        public AppletConfiable() {
                // El constructor prepara la pantalla para que la distribución de elementos
                // sea satisfactoria
            GridBagLayout gbl = new GridBagLayout();
            GridBagConstraints gbc = new GridBagConstraints();
             setLayout(gbl);

             display = new TextArea();
             display.setEditable(false);
             display.setFont(new Font("Courier", Font.PLAIN, 12));

             gbc.anchor = GridBagConstraints.WEST;
             gbc.gridx = 1;
             gbc.gridy = 1;
             gbc.insets = new Insets(2,2,2,2);
             gbl.setConstraints(display, gbc);

             add(display);
```

```
            validate();
        } // fin de la clase AppletConfiable

        public void init( )
        {
                DataInputStream dis;
                String linea;
                FileInputStream elFichero;
                String nombrefichero = getParameter("Fichero") ;

                // Se le solicita permiso al Gestor de Privilegios para acceder a
                // las propiedades del sistema
                try {
                PrivilegeManager.enablePrivilege("UniversalPropertyRead");
                display.appendText( "Tu nombre es " + System.getProperty("user.name") + ".\n" );
                 PrivilegeManager.revertPrivilege("UniversalPropertyRead") ;
                        // Una vez usado el privilegio, se revoca
                }
                catch ( SecurityException e ) {
                display.appendText( "No puedo obtener tu nombre, por culpa de una
                                                                        SecurityException.\n"
);
                 }// fin

                // Se le solicita permiso al Gestor de Privilegios para acceder a
                // al sistema de ficheros. Aquí se ha solicitado permiso total, aunque se        podría
                 //haber  pedido permiso de lectura nada más
                try {
                        PrivilegeManager.enablePrivilege("UniversalFileAccess");
                        elFichero = new FileInputStream(nombrefichero);
                        PrivilegeManager.revertPrivilege("UniversalFileAccess") ;
                        // Una vez usado el privilegio, se revoca
                        try {
                                dis = new DataInputStream(new BufferedInputStream(elFichero));
                                linea = dis.readLine();
                                display.appendText( "La primera línea del fichero " + nombrefichero
                                                                                + "  es:\n" + linea );
                        }
                        catch (IOException e) {
                                display.appendText( "IO Error:" + e.getMessage());
                           }

                }// fin del try
                catch ( FileNotFoundException e) {
                display.appendText( "No se ha encontrado el fichero: " + nombrefichero );
                }
                catch ( SecurityException e ) {
                                display.appendText( "No puedo leer el fichero, por culpa de una
                                                                        SecurityException." );
                }
        }// fin de la clase init
}// fin de la clase AppletConfiable
```

5. CONCLUSIONES

Del proyecto de tesis *ANÁLISIS, PRUEBA Y DOCUMENTACIÓN DEL USO DEL LENGUAJE JAVA EN APLICACIONES SEGURAS* se establecieron las siguientes conclusiones:

- A través de los navegadores Web se puede garantizar seguridad en cierto grado, si en algún momento dado, un applet de Java quiere tener acceso a los archivos de disco o a alguna propiedad del sistema, ya que se sabe quién es el dueño de éste más no si el applet es dañino. Esta seguridad se puede garantizar otorgándole o no permisos al applet en el instante en que se carga en el navegador. Esto se comprobó utilizando la herramienta *SignTool* (específicamente para *Netscape Navigator)* que permitió realizar certificados de prueba (las únicas entidades autorizadas para expedir certificados válidos son las autoridades de certificación como *Verisign*) acompañado con una serie de pasos concernientes para tal fin.

- Java administra la memoria RAM de una manera eficiente porque no usa punteros, emplea un área específica de la memoria para ejecutar las *applets*

(*sandbox*), y una técnica llamada *recolector de basura*, que se basa en limpiar de la memoria de forma automática los objetos que no están siendo utilizados en la etapa de ejecución del programa. Por tanto, esto conlleva a que como los objetos son borrados automáticamente no exista riesgo de desbordamiento y a su vez garantiza seguridad ya que ningún intruso tiene la oportunidad de capturar información de la memoria.

Se investigó la manera de esquematizar la forma como Java administra la memoria cuando se ejecuta una aplicación, encontrando que la forma como esto sucede sólo es conocida por el fabricante del JVM (Máquina Virtual de Java) y no es revelado. Solo se da a conocer la existencia de diferentes niveles de seguridad que se aplican a las clases y métodos de los programas cargados en memoria.

El hecho de conocer poco acerca de la arquitectura de manejo de memoria le agrega mayor seguridad al sistema, ya que la persona que intente violarlo, no sabe donde comienza y termina el área de datos de las aplicaciones.

- Fueron probadas librerías del paquete *java.security.** , que se consideraron más relevantes, tales como las encargadas de generar y manejar resúmenes de mensaje por medio del algoritmo SHA o MD5, y aquellas que permitían crear un pareja de claves por medio del algoritmo DSA.

Otro paquete al cual se hizo mención fue *java.security.acl,* que permitió establecer para varios usuarios, una estructura de permisos para controlar los recursos de un sistema, esta estructura de control de acceso, se mostró útil para la creación de perfiles de usuario en sistemas confiables.

Con la implementación de las librerías de los paquetes anteriormente mencionados se comprobó las ventajas de seguridad que se encuentran implícitas en el JDK 1.1 listas para ser usadas por los desarrolladores de software en la creación de aplicaciones seguras. Cabe resaltar que para garantizar la seguridad del sistema se debe contar con los aspectos de generación de claves, código Hash, encripción simétrica y encripción asimétrica. En la versión 1.1 del JDK no se incluyen los algoritmos necesarios para ahondar en los aspectos de encripción.

- Es importante para los ambientes informáticos - empresariales contar con herramientas de desarrollo de software robustas y seguras. Un programador de aplicaciones que domine las características y herramientas de seguridad de Java, como lenguaje que se encuentra a la vanguardia en este tema, puede hacer de sus sistemas de software productos altamente competitivos y más confiables. Esto ha de traer con sigo reconocimiento y beneficios dentro del complejo mundo de la informática.

6. RECOMENDACIONES

Teniendo en cuenta los diferentes aspectos desarrollados en este trabajo de tesis y los problemas que se tuvieron para profundizar más sobre los objetivos planteados, se hacen las siguientes recomendaciones para investigaciones futuras referente a la seguridad a través del lenguaje Java:

- El JDK 1.1 no incluye librerías especializadas para encripción simétrica o asimétrica con algoritmos como el RSA, IDEA y DES, pero si se quiere trabajar con alguno de ellos, se pueden adquirir paquetes de librerías de este tipo, las cuales son vendidas por diferentes fabricantes para Java, incluyendo a Sun Microsystem empresa creadora de éste lenguaje de programación.

  Con dichas herramientas criptográficas adicionales al JDK 1.1 y las librerías de seguridad estudiadas en este proyecto, sumadas a la robustez del manejo de memoria de Java, se pueden desarrollar sistemas altamente confiables y portables.

- Estudiar y probar las librerías de seguridad de las nuevas versiones del JDK, las cuales prometen incluir herramientas adicionales a las estudiadas en este proyecto.

- Impartir dentro de la Tecnológica de Bolívar el estudio del Lenguaje Java, puesto que esta herramienta cada vez va tomando más auge en el mundo de las redes de computadoras y los sistemas distribuidos, debido a sus características de trabajo en esos ambientes computacionales.

Estudiar a fondo las librerías incluidas con los lenguajes de programación antes de realizar cualquier aplicación en ellos, ya que algunas veces estos lenguajes contienen herramientas de mucha utilidad listas para usarse. Esto, además de ahorrar tiempo de programación, permite trabajar con módulos de alta calidad que hacen de una aplicación algo más profesional.

BIBLIOGRAFIA

- Ayudas Del *MSDN Library Visual Studio 6.0* Concernientes a la Documentación del *Visual J++ 6.0.*

- BECERRA Santamaria, Cesar A. *Los 600 Principales Métodos de Java*. Kimpres Ltda. 1998. Pág. 10-36.

- DANEH, Arman. *Aprendiendo JAVA SCRIPT En Una Semana*. Madrid – España. Prentice Hall. 1999. Pág. 52-70.

- FLANAGAN, David. *Java En Pocas Palabras. Referencia Al Instante*. Mc Graw Hill.1999. Pág. 3-81, 127-142

- *JAVA, Al Descubierto, Cubre la versión 1.0*. Madrid. Prentice Hall. 1999. Pág. 6-48, 60-105

- JAWORSKI, Jamie. *Guía de Desarrollo. "Integre Applets Java Para Crear Aplicaciones, Interacciones Con Internet*. Madrid – España. Prentice Hall. 1999. Pág. 11- 120.

Páginas en Internet

- ✓ http://www.iec.csic.es/criptonomicon/java/
- ✓ http://swissnet.ai.mit.edu/-jbank/javapaper/javapaper.html
- ✓ ftp://ftp.javasoft.com/docs/whitepaper.ps.tar.z
- ✓ http://java.sun.com/sfaq/
- ✓ http://java.sun.com/1.0alpha3/doc/security/security.html
- ✓ http://java.sun.com/JDK-beta2/psfiles/javaspec.ps

- ✓ http://www.javasoft.com
- ✓ http://www.cert.org/
- ✓ http://www.ibm.com/java
- ✓ http://www.ibm.com/java/siteindex.html
- ✓ http://pantheon.yale.edu/~dff/java.html
- ✓ http://pantheon.yale.edu/help/programming/jdk1.1.1/docs/api
- ✓ http://www.trl.ibm.co.jp
- ✓ http://www.javasoft.com/security
- ✓ http://www.sun.com/java
- ✓ http://www.microsoft.com/java
- ✓ http://www.microsoft.com/java/security
- ✓ http://www.rstcorp.com
- ✓ http://www.cs.princeton.edu/sip
- ✓ http://www.math.gatech.edu/~mladue/HostileApplets.html
- ✓ http://ferret.Imh.ox.ac.uk/~david/java/bugs/public.html
- ✓ http://seclab.cs.ucdavis.edu/~samorodi/java/javasec.html
- ✓ http://www.cs.purdue.edu/homes/spaf/hotlists/csec-body.html#java00
- ✓ http://java.sun.com/docs/books/jls/.
- ✓ http://developer.netscate.com/library/documentation/signedobj/capabilities.
- ✓ http://java.sun.com/docs/books/tutorial/security1.1/index.html
- ✓ http://daffy.cs.yale.edu/java/java_sec/java_sec.html

ALGUNAS PÁGINAS WEB DONDE SE PUEDE CONSULTAR DEL TEMA DE LOS APPLETS:

Para bajar las clases del Netscape Java Capabilities API:

- ✓ http://developer.netscape.com/library/documentation/signedobj/capsapi_classes.zip

Para la bajar la herramienta *Signtool 1.1* para generar certificados y firmar applets:

- ✓ http://developer.netscape.com/software/signedobj/jarpack.html

Para consultar acerca de la Autoridad de Certificación *Verisign:*

- ✓ http://www.verisign.com/

Autoridades propuestas por Netscape aptas para certificar sus productos:

- ✓ http://certs.netscape.com/

Listado completo de los recursos que se pueden acceder al sistema por medio de un applet firmado por una herramienta para el navegador Netscape Communicator:

- ✓ http://developer.netscape.com/docs/manuals/signedobj/targets/index.htm

Documentación completa sobre este API de Capacidad de Java para Netscape:

- ✓ http://developer.netscape.com/docs/manuals/signedobj/capsapi.html

MANUAL DEL PROGRAMADOR

"Simulación de la técnica Recolector de Basura por parte del Lenguaje Java"

Requerimientos de sistemas para que el programa funcione en condiciones optimas:

- PC con sistema operativo Windows sea 95,98 o superior.
- Procesador con 300 Mhz como mínimo.
- Previa instalación de la Maquina Virtual de Java (JVM).
- Resolución de la pantalla de 800x600 a una alta densidad de colores (16 bits).

¿POR QUÉ LA SIMULACIÓN DE LA TÉCNICA DEL RECOLECTOR DE BASURA?

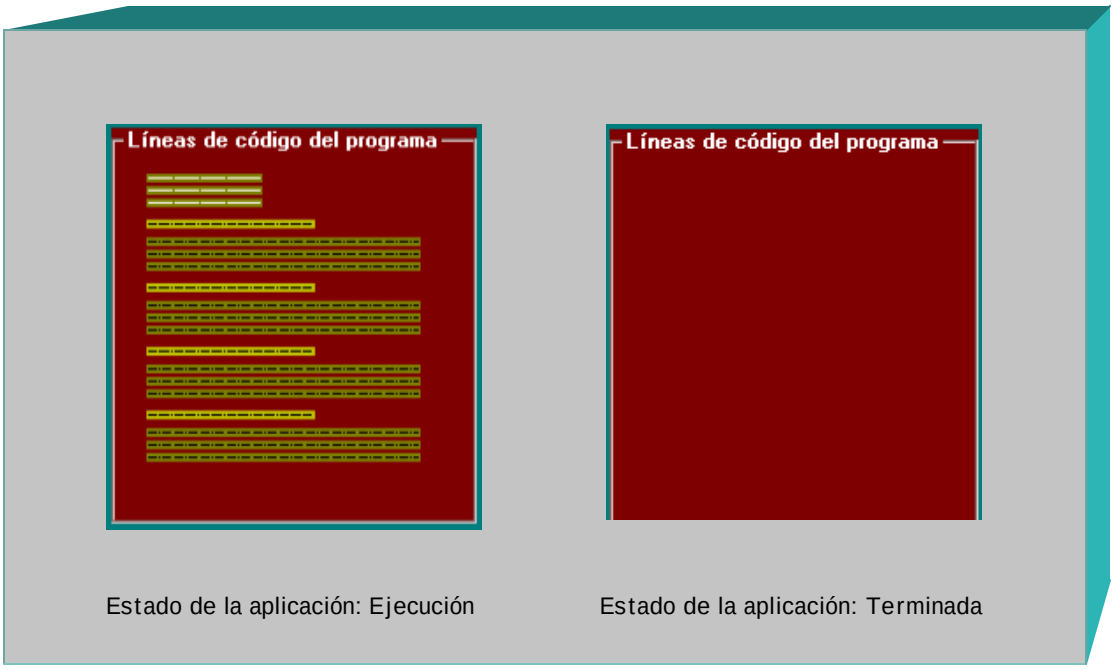
Se tuvo la idea de diseñar un programa sencillo que simulara el proceso que se sigue por la técnica del recolector de basura, porque se quiso mostrar todo este proceso de una forma gráfica para que los interesados en saber como Java maneja esta técnica tuvieran una concepción visual de ésta.

Es un programa muy didáctico para esquematizar cómo Java destruye los objetos que queden en memoria cuando no vayan a ser utilizados por la aplicación en el momento que deje de ejecutarse.

COMPONENTES DEL PROGRAMA

El programa consta de los siguientes componentes:

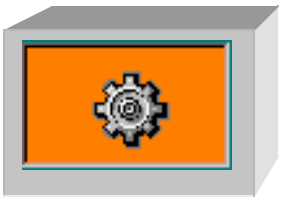
1. Un componente donde se visualiza la ejecución/terminación del programa.



Estado de la aplicación: Ejecución

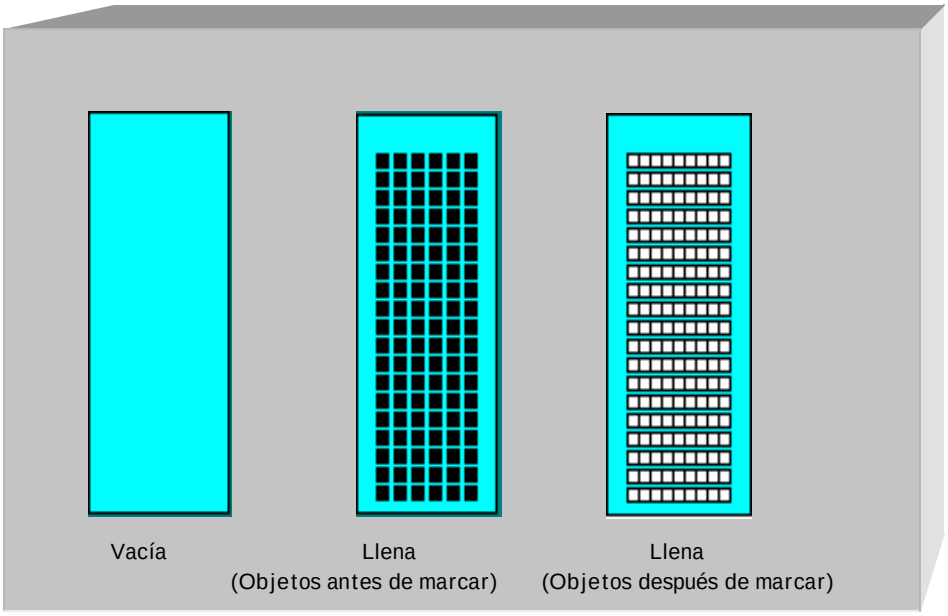
Estado de la aplicación: Terminada

2. Un componente que muestra la Maquina Virtual de Java, esta es la encargada de llevar cada objeto creado a la pila.



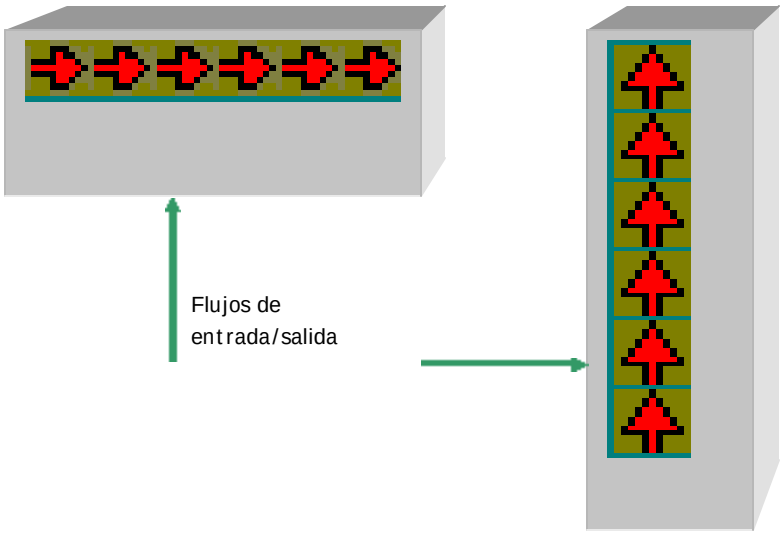
Maquina Virtual de Java

3. Un componente que muestra los diferentes estados de la Pila.

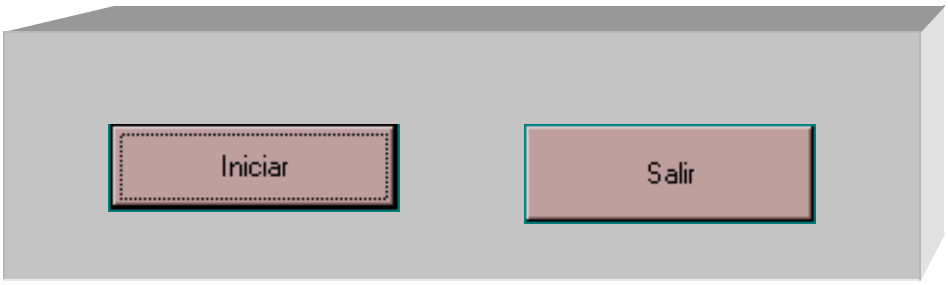


4. Un componente que muestra al Recolector de Basura en sus dos estados (lleno/vacío).

5. Una serie de flujos de entrada/salida representados por flechas para seguir la trayectoria de los objetos en este proceso de simulación de la técnica del recolector de basura.



6. Dos botones, **Iniciar** para dar inicio a la ejecución del simulador y **Salir** para cuando se desee salir del programa.



FILOSOFIA DEL PROGRAMA

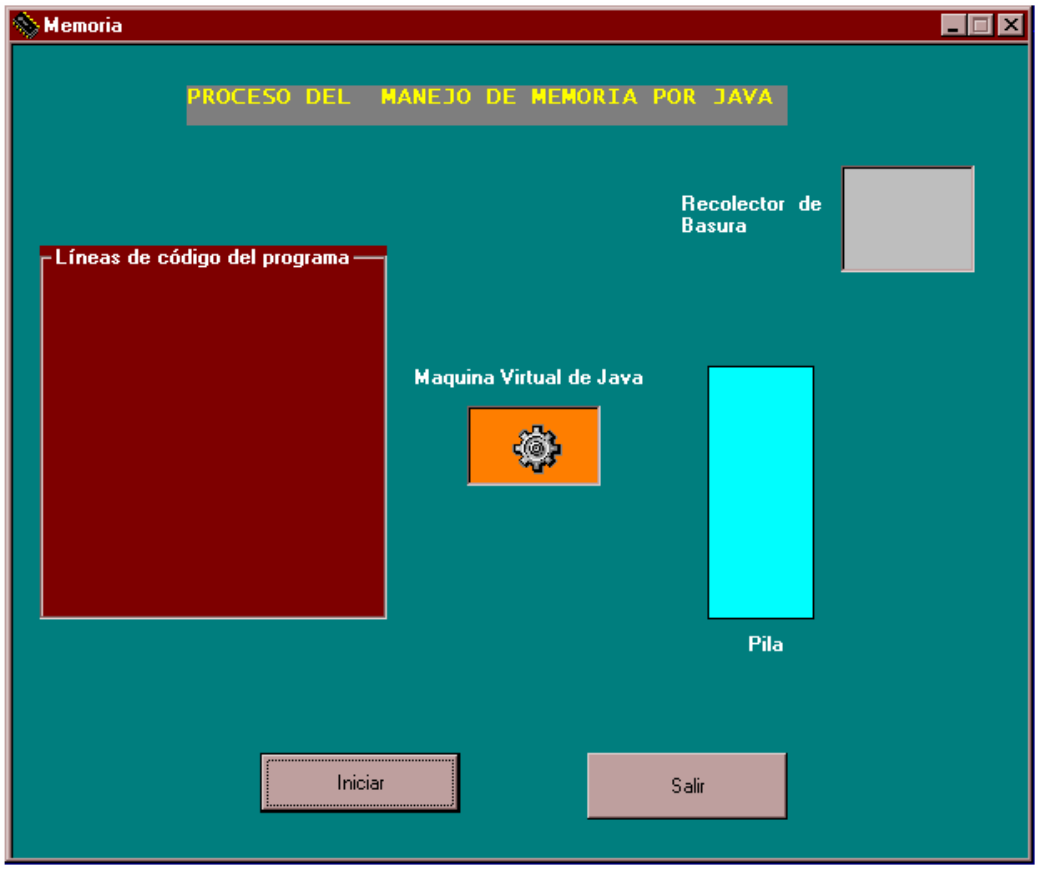
El programa de simulación se inicia oprimiendo el botón **Iniciar**, automáticamente empezará la ejecución del programa, primero se ejecutaran las líneas de código del programa, cada línea que se dibuja simulará un objeto, éste es trasladado a la pila por medio de la maquina virtual de Java que es la encargada de realizar este trabajo (se emplea una cuadricula de color negro para tal efecto).

Posteriormente cuando el programa deja de ejecutarse (se borra del panel ejecución), se marcan los objetos que dejen de ser referenciados dentro de la pila (esto se simula con una cuadricula de color blanco) para luego ser depositados en el recolector de basura de manera automática.

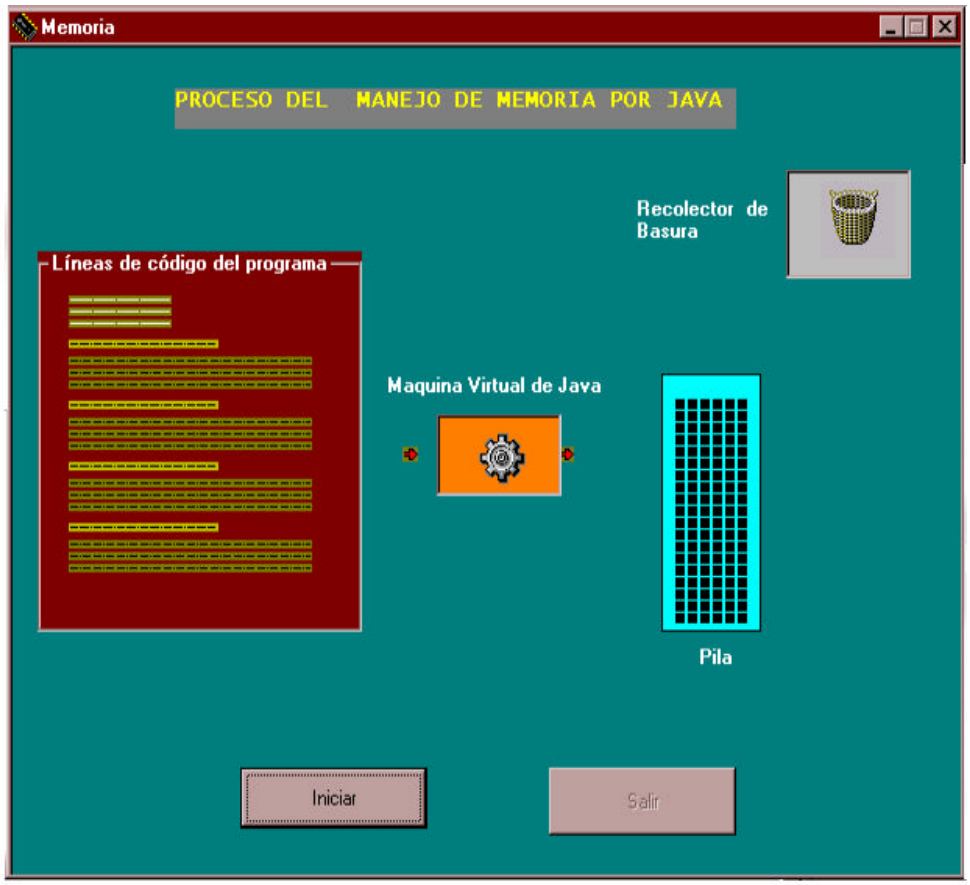
Si se desea nuevamente correr el programa basta con solo oprimir nuevamente el botón **Iniciar.**

Para salir del programa se oprime el botón **Salir.**

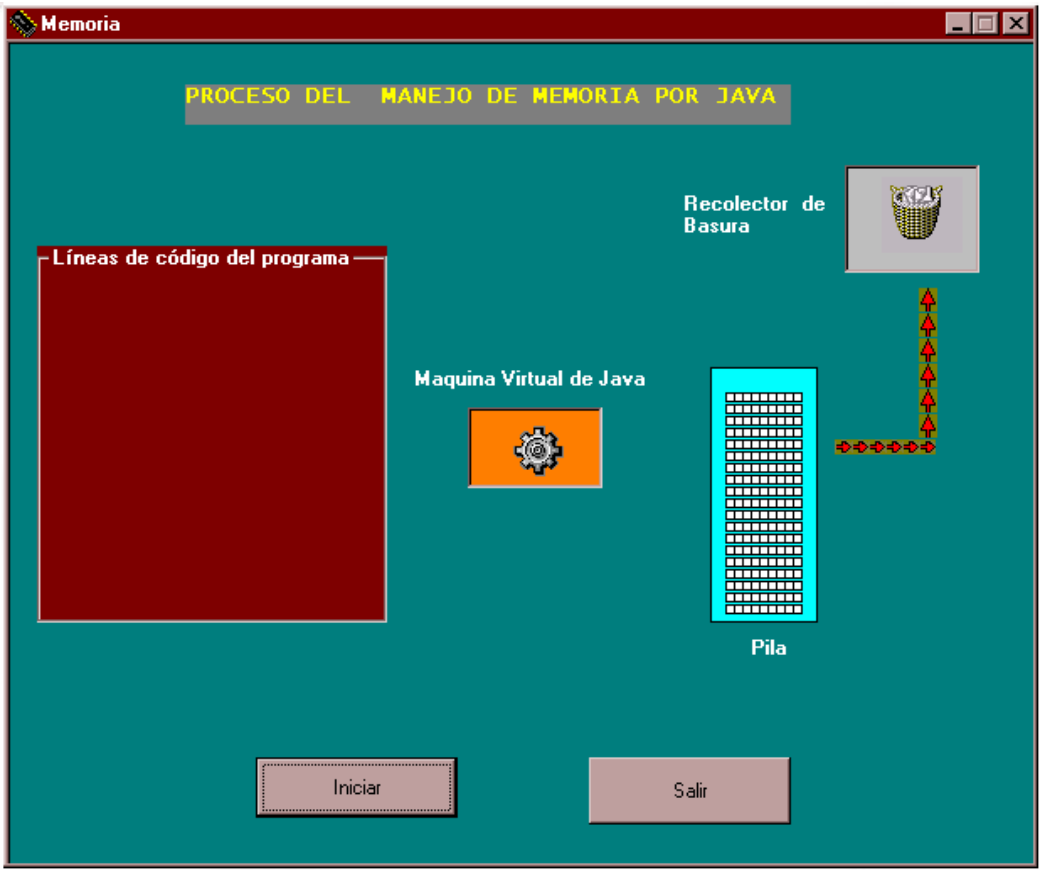
ENTORNO DEL PROGRAMA



Vista inicial de la ventana del programa



Vista de la fase de ejecución del programa y llenado de la pila con los objetos utilizados en la aplicación, este traslado de objetos lo hace la Maquina Virtual de Java



Vista de la fase del traslado de los objetos que dejen de ser referenciados, hacía el Recolector de Basura cuando son marcados dentro de la pila.