Tema 6

Applets. Applets Firmados. Encriptación.

Departament d'Informàtica. Universitat de València

Índice

1. Applets **2**

1.1. Un primer ejemplo . . . 6

1.2. Otro ejemplo con parámetros . . . 7

1.3. Formas de visualizar un applet . . . 9

1.4. El plugin de Java . . . 10

2. Applets firmados **11**

2.1. Seguridad en Java . . . 11

2.1.1. Administradores de seguridad y permisos . . . 12

3. Seguridad en la plataforma Java 2 **14**

3.1. Archivos de política de seguridad . . . 15

3.2. Firmas digitales . . . 20

3.2.1. Compendio de mensaje . . . 20

3.2.2. Mensajes firmados . . . 21

3.2.3. Autenticación de usuarios . . . 22

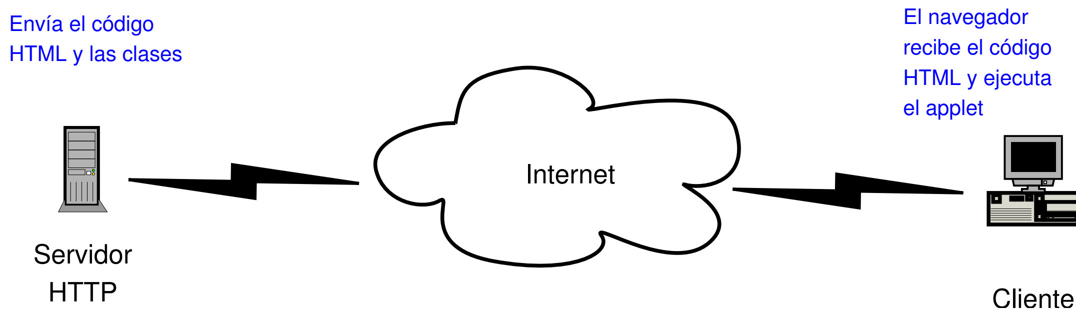
4. Encriptación **28**

4.1. Codificadores simétricos sobre bloques . . . 28

4.2. Codificadores asimétricos o de clave pública 33

1. Applets

- Un applet está formado por una clase o conjunto de clases escritas en Java que está insertado en una página Web.
- Cuando un usuario carga la página en la que está el applet, éste se ejecuta localmente (en la máquina cliente donde se está ejecutando el navegador Web) y no remotamente (sobre el sistema que está ejecutando el servidor HTTP).



- El hecho de que el código se ejecute localmente implica que la seguridad sea crucial. Los applets están sometidos a unas restricciones de seguridad por defecto.
- Por ejemplo, nadie desea que un applet cargado visitando una página Web borre sus ficheros. O que ese applet realice conexiones a otras máquinas para transferir información.
- Estas restricciones están impuestas por un objeto del tipo SecurityManager en el sistema cliente, que especifica qué acciones están permitidas y cuales no.

- A partir de la versión 1.1 de la plataforma Java, las clases se pueden firmar digitalmente, estas clases (si el usuario confía en quien las firma) pueden realizar acciones no permitidas a las clases no firmadas.

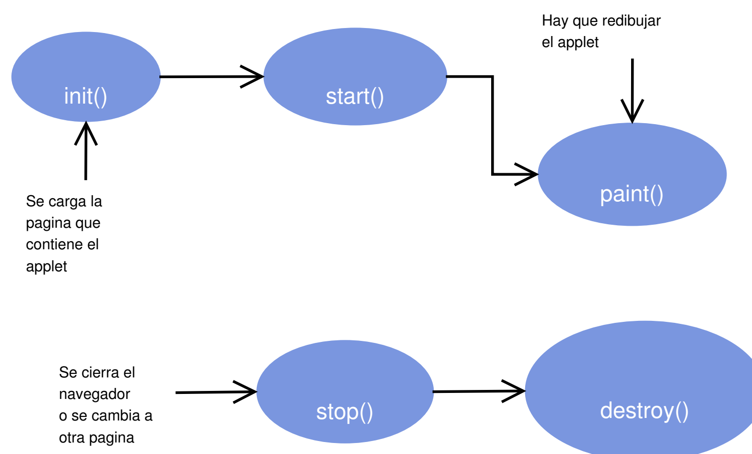
- Por defecto, el gestor de seguridad de los navegadores verifica entre otras cosas que los applets
  - No puedan leer los ficheros locales.
  - No puedan escribir en los discos locales.
  - No puedan establecer conexiones a otras máquinas salvo con el servidor que contiene el applet.
  - No ejecutar programas locales.
  - No puedan obtener información privada sobre el usuario.

- Los applets deben ser subclases de la clase Applet (si se van a utilizar clases de AWT) o de JApplet (si se van a utilizar clases de Swing).

- A diferencia de las aplicaciones vistas hasta el momento, la ejecución de un applet no comienza en el método main(.).

- Hay una serie de métodos que son llamados cuando ocurren determinadas circunstancias. Estos métodos de Applet o de JApplet son los que hay que sobreescribir.

La siguiente figura muestra cuando llama el navegador a estos métodos:



El esqueleto de un applet basado en la clase Applet se muestra a continuación:

```
import java.applet.Applet;
import java.awt.*;
```

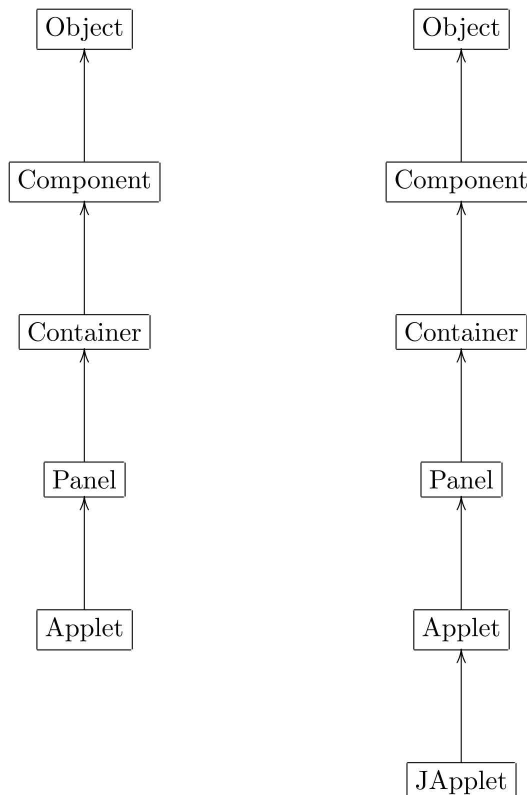
```
class UnApplet extends Applet{
   // Declaración de atributos

   public void init(){
      // Iniciación de las variables
      // Construccion de la GUI
   }

   public void start(){
      // Sentencias
   }

   public void stop(){
      // Sentencias
   }

   public void destroy(){
      // Sentencias
   }

   public void paint(Graphics g){
      // Operaciones de dibujo (si las hay)
   }
}
```

Un applet concreto puede sobreescribir todos estos métodos o solamente algunos de ellos en función de la tarea que deba realizar.

En los ejemplos de código proporcionados, en el directorio `ejemplos/applets/OrdenMetodos` hay un applet firmado (para poder acceder al disco local) que guarda en el fichero que se le pasa como parámetro desde el código HTML la secuencia de llamadas a los diferentes métodos.

Este applet se puede utilizar para ver las posibles diferencias entre los navegadores a la hora de llamar a estos métodos.

La jerarquía de las clases Applet y JApplet es la siguiente:



Además de los métodos que controlan el ciclo de vida del applet hay otros métodos disponibles, entre los cuales destacan:

```
public AudioClip getAudioClip(URL urlClip)
public AudioClip getAudioClip(URL url, String fichero)
```

Obtienen un fichero de audio y MIDI desde una URL y lo asignan a un objeto del tipo AudioClip que permite los mensajes play(), loop() y stop(). La versión JDK 1.1 soporta ficheros en formato `.au`. A partir de la versión 1.2 se soportan `.aiff` y `.wav` además de MIDI tipo 0, MIDI tipo 1 y ficheros en formato RMF.

```
public Image getImage(URL urlImagen)
public Image getImage(URL url, String fichero)
```

Estos métodos obtienen una imagen desde una URL y crean un objeto del tipo Image.

```
public void showStatus(String mensaje)
```

Este método muestra la cadena pasada como argumento en la barra de estado (en la parte inferior del navegador).

```
public void showDocument(URL htmlDoc)
public void showDocument(URL htmlDoc, String frame)
```

Estos métodos se pueden utilizar para indicar al navegador que muestre una determinada página Web. En realidad están definidos en la clase AppletContext por lo que para utilizarlos hay que hacer lo siguiente: getAppletContext().showDocument(...)

Como ya se ha comentado, el applet debe estar incluido en una página Web.

El lenguaje de marcado HTML contiene etiquetas para indicar que un elemento es un applet.

El esqueleto de código HTML de una página que contenga un applet se muestra a continuación:

```
<HTML>
   <HEAD>
      <TITLE>
         Applet
      </TITLE>
   </HEAD>
   <BODY>
      <APPLET
         ARCHIVE= fichero.jar
         CODE= Ejemplo.class
         CODEBASE= http://informatica.uv.es
         WIDTH=400
         HEIGHT=400
         ALIGN=MIDDLE>
         <PARAM NAME="FICHERO" VALUE="c:/tmp/seleccion.txt"/>
      </APPLET>
   </BODY>
</HTML>
```

Elementos	*Significado*
ARCHIVE	*Especifica el archivo jar donde se encuentran las clases y otros recursos que utilice el applet.*
CODE	*Especifica la clase que extiende a Applet o a JApplet. Se asume que la clase se puede encontrar en el mismo sitio que la página web a no ser que se proporcione CODEBASE.*
CODEBASE	*Designa la URL donde encontrar la clase o el fichero jar.*
WIDTH	*Especifica la anchura que tendrá el applet.*
HEIGHT	*Especifica la altura que tendrá el applet.*
ALIGN	*Especifica cómo se debe colocar el applet dentro del espacio disponible (LEFT, RIGHT, TOP, BOTTOM, MIDDLE).*
PARAM	*Se puede utilizar para pasar parámetros al applet. En NAME se especifica el nombre del parámetro y en VALUE se especifica el valor.*

1.1. Un primer ejemplo

```
import java.awt.*;
import java.applet.*;


/** Esta clase es un Banner en el que aparecera texto desplazandose
  */
public class Banner extends Applet implements Runnable{
    private String msg = "Esta frase se desplaza        Esta frase se desplaza       ";
    private Thread t = null;
    private boolean parado;

    /** Ocultamos el metodo init de Applet <br>
      * Se establecen los colores del fondo y de la fuente
      * ademas del tipo de fuente
      */
    public void init(){
        setBackground(Color.blue);
        setForeground(Color.white);
        setFont(new Font("Arial",Font.BOLD,18));
    }

    /** Ocultamos el metodo start() de Applet <br>
      * Se crea un hilo y se lanza
      */
    public void start(){
        // Esto se puede hacer ya que esta clase implementa a Runnable
        t = new Thread(this);
        parado = false;
        t.start();
    }

    /** Implementamos el metodo run() ya que la clase Banner
      * implementa a la interfaz Runnable. <br>
      * En este metodo hay un bucle infinito deltro del cual
      * se repinta el applet, se duerme durante
      * 250 milisegundos, y el primer caracter se pone al final
      * de la cadena. <br>
      * Si se debe parar el hilo se sale del bucle.
      */
    public void run(){
        char ch;

        for ( ; ; ){
            try{
                repaint();
                Thread.sleep(250);
                ch = msg.charAt(0);
                msg = msg.substring(1,msg.length());
                msg = msg + ch;
                if (parado)
                    break;
            }catch(InterruptedException e){}
        }
    }


    /** Ocultamos el metodo stop() de Applet <br>
      * Si se debe parar el applet se pone el indicador a false
      * y se asigna la referencia al hilo a null.
```

```
    */
  public void stop(){
    parado = true;
    t = null;
  }

  /** Ocultamos el metodo paint(Graphics g) de Applet <br>
    * Se limita a pintar el mensage. Este metodo sera llamado
    * cuando se llame a repaint()
    */
  public void paint(Graphics g){
    g.drawString(msg,50,30);
  }
}
```

Y el código HTML donde se utiliza este applet es:

```
<HTML>
  <HEAD>
    <TITLE> Ejemplo de applet </TITLE>
  </HEAD>

  <BODY>
    <H1> Ejemplo de applet </H1>

    <APPLET
      CODE = Banner.class
      WIDTH= 500
      HEIGHT= 40
      ALIGN = CENTER
    />
  </BODY>
</HTML>
```

Este ejemplo se encuentra en el directorio `ejemplos/applets/Ejemplo1`

1.2. Otro ejemplo con parámetros

Ahora vamos a modificar el código anterior para pasar como argumentos al applet la siguiente información:

- El texto a mostrar
- El intervalo de tiempo entre actualizaciones
- Las coordenadas donde debe mostrar el texto
- El sentido de desplazamiento del texto

```
import java.awt.*;
import java.applet.*;


/** Esta clase es un Banner en el que aparecera texto desplazandose
  */
public class BannerParametros extends Applet implements Runnable{
  private String msg;
  private int tiempo;
  private int cx;
  private int cy;
  private String sentido;
  private Thread t = null;
  private boolean parado;


  /** Ocultamos el metodo init de Applet <br>
    * Se establecen los colores del fondo y de la fuente
    * ademas del tipo de fuente
```

```
    */
    public void init(){
        setBackground(Color.blue);
        setForeground(Color.white);
        setFont(new Font("Arial",Font.BOLD,18));
    }

    /** Ocultamos el metodo start() de Applet <br>
     * Se obtienen los parametros, se crea un hilo y se lanza
     */
    public void start(){
        String parametro;

        // Obtencion de los parametros

        msg = getParameter("TEXTO");
        if (msg==null)
            msg = "Texto por defecto";

        msg = "      " + msg + "      ";

        sentido = getParameter("SENTIDO");
        if (sentido==null)
            sentido = "I";

        parametro = getParameter("TIEMPO");
        try{
            if (parametro!= null)
                tiempo = Integer.parseInt(parametro);
            else
                tiempo = 250; // Por defecto
        }catch(NumberFormatException e){
            // Si se ha proporcionado el parametro pero no es un numero valido
            tiempo = 250;
        }

        parametro = getParameter("CX");
        try{
            if (parametro!= null)
                cx = Integer.parseInt(parametro);
            else
                cx = 50; // Por defecto
        }catch(NumberFormatException e){
            // Si se ha proporcionado el parametro pero no es un numero valido
            cx = 50;
        }

        parametro = getParameter("CY");
        try{
            if (parametro!= null)
                cy = Integer.parseInt(parametro);
            else
                cy = 30; // Por defecto
        }catch(NumberFormatException e){
            // Si se ha proporcionado el parametro pero no es un numero valido
            cy = 30;
        }

        // Esto se puede hacer ya que esta clase implementa a Runnable
        t = new Thread(this);
        parado = false;
        t.start();
    }

    /** Implementamos el metodo run() ya que la clase Banner
     * implementa a la interfaz Runnable. <br>
     * En este metodo hay un bucle infinito deltro del cual
     * se repinta el applet, se duerme durante
     * 250 milisegundos, y el primer caracter se pone al final
     * de la cadena. <br>
     * Si se debe parar el hilo se sale del bucle.
     */
    public void run(){
        char ch;

        for ( ; ; ){
            try{
                repaint();
                Thread.sleep(tiempo);

                // En funcion de sentido desplazamos la frase a la
                // izquierda o a la derecha

                if (sentido.compareTo("I")==0){
                    ch = msg.charAt(0);
                    msg = msg.substring(1,msg.length());
                    msg = msg + ch;
                }
                else{
                    ch = msg.charAt(msg.length()-1);
                    msg = msg.substring(0,msg.length()-1);
                    msg = ch + msg;
                }
```

```
            if (parado)
               break;
         }catch(InterruptedException e){}
      }
   }

   /** Ocultamos el metodo stop() de Applet <br>
     * Si se debe parar el applet se pone el indicador a false
     * y se asigna la referencia al hilo a null.
     */
   public void stop(){
      parado = true;
      t = null;
   }

   /** Ocultamos el metodo paint(Graphics g) de Applet <br>
     * Se limita a pintar el mensage. Este metodo sera llamado
     * cuando se llame a repaint()
     */
   public void paint(Graphics g){
      g.drawString(msg,cx,cy);
   }
}
```

Y el código HTML donde se utiliza este applet y se le pasan los parámetros es:

```
<HTML>
   <HEAD>
      <TITLE> Ejemplo de applet con parametros</TITLE>
   </HEAD>
   <BODY>
      <H1> Ejemplo de applet con parametros</H1>

      <APPLET
         CODE = BannerParametros.class
         WIDTH= 300
         HEIGHT= 30
         ALIGN = CENTER>
         <PARAM NAME="TEXTO" VALUE="Texto en la parte superior"/>
         <PARAM NAME="TIEMPO" VALUE="150"/>
         <PARAM NAME="CX" VALUE="70"/>
         <PARAM NAME="CY" VALUE="30"/>
         <PARAM NAME="SENTIDO" VALUE="I"/>
      </APPLET>

      <BR><HR/><BR>

      <APPLET
         CODE = BannerParametros.class
         WIDTH= 300
         HEIGHT= 30
         ALIGN = CENTER>
         <PARAM NAME="TEXTO" VALUE="Texto en la parte inferior"/>
         <PARAM NAME="TIEMPO" VALUE="200"/>
         <PARAM NAME="CX" VALUE="70"/>
         <PARAM NAME="CY" VALUE="30"/>
         <PARAM NAME="SENTIDO" VALUE="D"/>
      </APPLET>
   </BODY>
</HTML>
```

Este ejemplo se encuentra en el directorio `ejemplos/applets/Ejemplo2`

1.3. Formas de visualizar un applet

Un applet se puede visualizar en un navegador Web (Internet Explorer, Netscape, ...) o utilizando la utilidad `appletviewer` que se distribuye en el JSDK.

Para visualizar un applet utilizando `appletviewer` hay que pasarle a esta utilidad el nombre de la página HTML que lo contiene, por ejemplo:

```
appletviewer Banner.html
```

Esta utilidad muestra solamente los applets descartando el resto del código que puede haber en la página.

Si hay más de un applet en la página, cada uno de ellos se mostraría en una ventana diferente.

```
appletviewer BannerParametros.html
```

También es posible visualizar con `appletviewer` applets que estén en páginas remotas. En la siguiente página hay un applet que muestra la convolución discreta.

```
appletviewer http://www.jhu.edu/~signals/discreteconv2/index.html
```

1.4. El plugin de Java

A la hora de cargar un applet en un navegador Web hay dos posibilidades:

- Cargarlo en la máquina virtual que proporciona el propio navegador o,
- Cargarlo en el plugin que proporciona Sun (lo cual implica que debe estar instalado el plugin para el navegador).

La primera aproximación tiene la ventaja de que el cliente podrá ejecutar el applet sin necesidad de haber instalado nada adicional en el navegador.

Pero tiene el inconveniente de que si el proveedor del navegador no actualiza la máquina virtual es posible que no se pueda cargar código desarrollado con las últimas versiones del JSDK.

Si el applet que hemos desarrollado merece la pena y necesita las últimas bibliotecas de clases se puede forzar a que el cliente utilice el plugin pero si es una mera decoración se recomienda no forzar a utilizar el plugin.

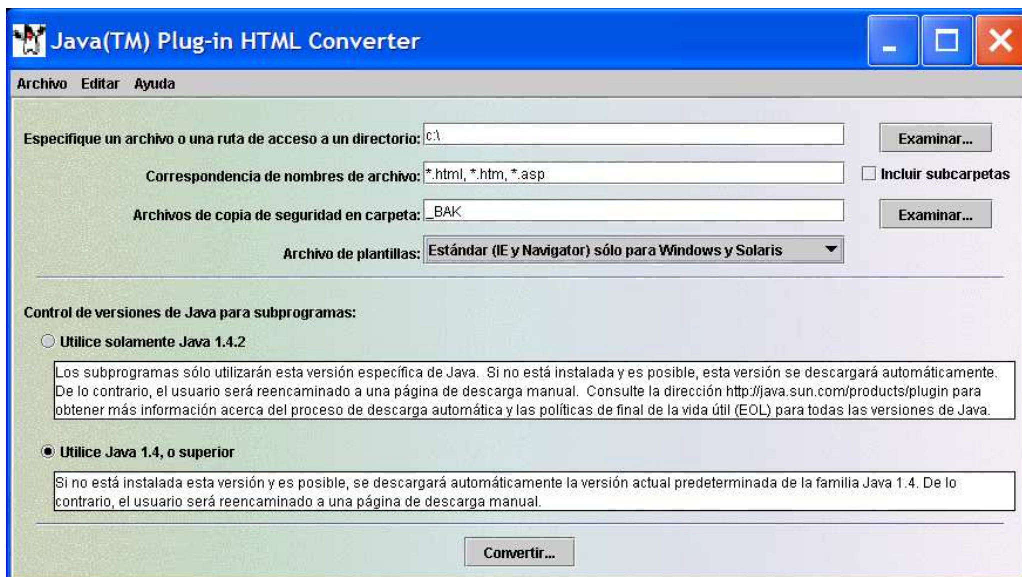
Si se desea forzar al navegador a utilizar el plugin, el (sencillo) código HTML mostrado a lo largo de este tema se complica ya que hay que tener en cuenta las particularidades de cada navegador (y del appletviewer) para contemplar todas las posibilidades.

El código de la derecha muestra la etiqueta <applet> tal y como lo hemos visto hasta ahora y el de la derecha muestra la transformación para que se ejecute en el plugin (hay que tener en cuenta que si el cliente no tiene instalado el plugin este código intenta instalarlo).

```
<APPLET CODE = Banner.class
    WIDTH= 500
    HEIGHT= 40
    ALIGN = CENTER
/>
```

```
<OBJECT
    classid = "clsid:CAFEEFAC-0014-0002-0000-ABCDEFFEDCBA"
    codebase = "http://java.sun.com/products/plugin/autodl/jinstall-1_4_2-
        windows-i586.cab#Version=1,4,2,0"
    WIDTH = 500 HEIGHT = 40 ALIGN = CENTER >
    <PARAM NAME = CODE VALUE = Banner.class >
    <PARAM NAME = "type" VALUE = "application/x-java-applet;jpi-version=1.4.2
        ">
    <PARAM NAME = "scriptable" VALUE = "false">

    <COMMENT>
    <EMBED
        type = "application/x-java-applet;jpi-version=1.4.2" \
        CODE = Banner.class \
        WIDTH = 500 \
        HEIGHT = 40 \
        ALIGN = CENTER \
        scriptable = false \
        pluginspage = "http://java.sun.com/products/plugin/index.html#download
            ">
        <NOEMBED>

        </NOEMBED>
    </EMBED>
    </COMMENT>
</OBJECT>
```

Por suerte esto no hay que realizarlo cada vez que se desee insertar un applet dentro de una página HTML, ya que junto con el JSDK se proporciona una utilidad llamada `htmlconverter` que convierte un fichero HTML en el que aparezca el elemento <applet> en otro que utilice el elemento <objet>.



2. Applets firmados

2.1. Seguridad en Java

Hay tres mecanismos en Java que ayudan a garantizar la seguridad:

- Las características del diseño del lenguaje (verificación de los límites de los vectores, comprobación de las conversiones de tipo, ...)
- Un mecanismo de control de acceso que controla lo que el código puede hacer (acceso a archivos, acceso a la red,...)
- Firma de código, con lo que es posible utilizar los algoritmos de criptografía estándar para autenticar el código. El usuario de este código puede determinar quién creó el código y si el código ha sido alterado después de haber sido firmado.

Para ejecutar una aplicación en la máquina virtual lo primero que hay que hacer es cargar las clases necesarias para lo cual la máquina virtual utiliza cargadores de clase (que se pueden personalizar).

Una vez que los archivos de clases se han cargado en la máquina virtual, se verifica la integridad del *bytecode* comprobando:

- Que las variables están inicializadas antes de ser utilizadas.
- Que en las llamadas a un método los argumentos son del tipo correcto.
- Que no se han violado las reglas para el acceso a los métodos y datos privados.
- Que los accesos a las variables locales caen dentro de la pila `runtime` (la pila de llamadas a métodos)...

Pero... si el compilador ya comprueba todo esto ¿para qué es necesario volverlo a comprobar?

El problema puede surgir si se modifica el *bytecode* una vez que haya sido compilado (modificando directamente el fichero `.class`).

De este modo se podría modificar una clase de forma que contenga instrucciones cuya sintaxis sea correcta pero inseguras para la máquina virtual de Java.

2.1.1. Administradores de seguridad y permisos

Una vez que la clase ha sido cargada y verificada, entra en acción el tercer mecanismo de la plataforma Java: el **administrador de seguridad**.

El administrador de seguridad es una clase que controla si el código realiza operaciones que no están permitidas (en tal caso se lanza una excepción).

Entre las operaciones que se comprueban están las siguientes:

- Si el hilo actual puede crear un nuevo cargador de clases.
- Si el hilo actual puede crear un subproceso.
- Si el hilo actual puede detener la máquina virtual.
- Si úna clase puede acceder a un miembro de otra clase.
- Si el hilo actual puede acceder a un paquete específico.

- Si el hilo actual puede acceder o modificar las propiedades del sistema.
- Si el hilo actual puede leer desde o escribir en un determinado archivo.
- Si el hilo actual puede eliminar un archivo.
- Si el hilo actual puede aceptar una conexión *socket* desde un *host* y número de puerto específicos.
- Si el hilo actual puede realizar una conexión *socket* a un *host* y número de puerto específicos.
- Si el hilo actual puede esperar una solicitud de conexión en un número de puerto local específico.
- Si el hilo actual puede llamar a los métodos stop(), suspend(), resume(), destroy(), setPriority() / setMaxPriority(), setName(), o setDaemon() de un hilo dado o de un grupo de hilos.

- Si el hilo actual puede iniciar un trabajo de impresión.
- Si una clase puede acceder al portapapeles del sistema.
- Si una clase puede acceder a la cola de eventos de AWT.

El comportamiento por defecto al ejecutar aplicaciones Java es que no se instala ningún administrador se seguridad, de modo que todas estas operaciones están permitidas.

Por otra parte la utilidad `appletviewer` instala inmediatamente un administrador se seguridad (denominado AppletSecurity) que es bastante restrictivo.

Ejemplos de restricciones de seguridad para los applets:

¿Puede un applet leer propiedades sobre el JRE y el sistema? [1]

¿Puede un applet leer propiedades sobre el usuario? [2]

¿Puede un applet cerrar el navegador? [3]

¿Puede un applet lanzar una aplicación? [4]

[1] El código se encuentra en el directorio `ejemplos/Seguridad/Ejemplo1`
[2] El código se encuentra en el directorio `ejemplos/Seguridad/Ejemplo2`
[3] El código se encuentra en el directorio `ejemplos/Seguridad/Ejemplo3`
[4] El código se encuentra en el directorio `ejemplos/Seguridad/Ejemplo4`

3. Seguridad en la plataforma Java 2

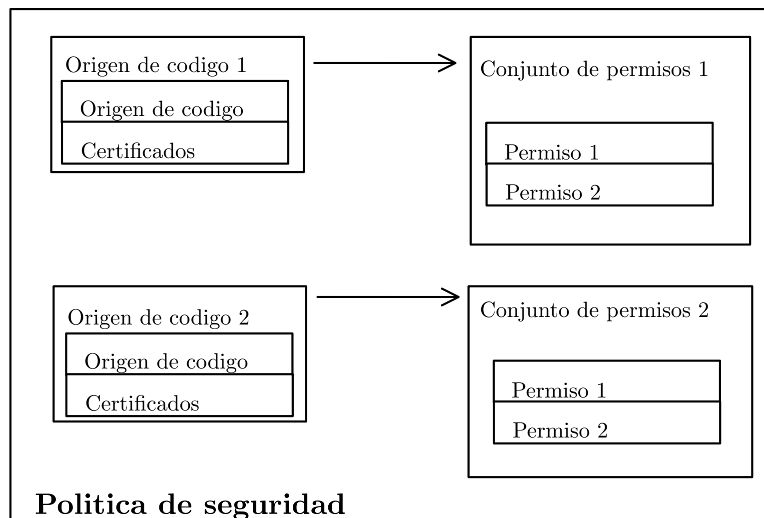
El JDK 1.0 tenía un modelo de seguridad muy sencillo: las clases locales tienen todos los permisos y las clases remotas están confinadas a la *sandbox*

Conjunto de medidas de seguridad. La *sandbox* crea un entorno en el que se imponen limitaciones sobre lo que puede realizar el código. Son utilizadas cuando el código proviene de fuentes desconocidas o en las que no se tiene confianza, permitiendo al usuario ejecutar código con ciertas garantías.

El JDK 1.1 implementaba una ligera modificación: al código remoto firmado por una entidad de confianza se le otorgaban los mismos permisos que a las clases locales.

Por lo tanto, ambas versiones del JDK proporcionaban control del tipo todo o nada. Los programas tenían todos los permisos o tenían los permisos otorgados por la *sandbox*.

La plataforma Java 2 tiene un mecanismo más flexible: una **política de seguridad** asigna un **conjunto de permisos** a **orígenes de código**.



El **origen de código** consta de:

1. La ubicación del código que puede ser un URL o un fichero JAR, y
2. Certificados (que veremos más adelante).

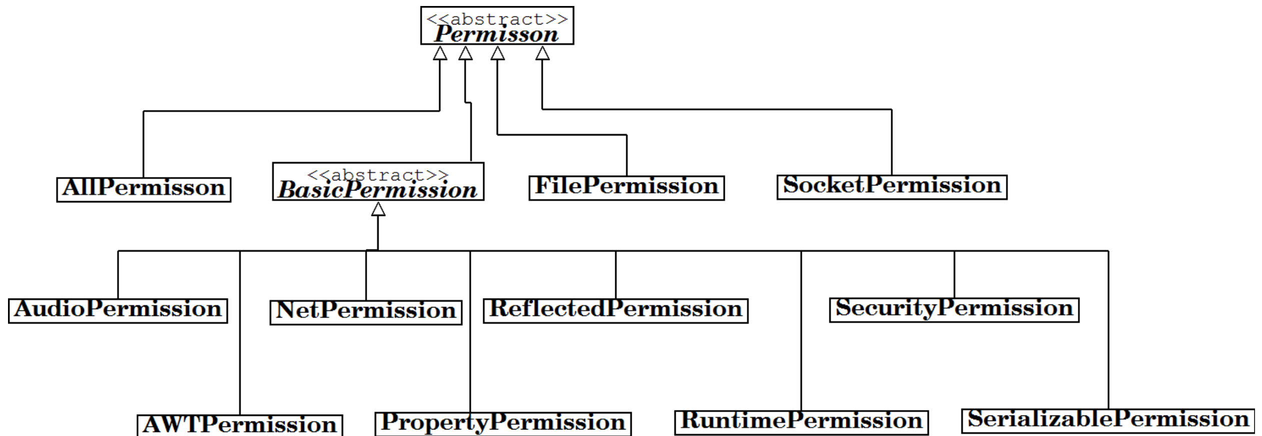
El administrador de seguridad comprueba los *permisos* en tiempo de ejecución.

Hay varias clases de permisos cada una de las cuales encapsula los detalles de un permiso particular. Por ejemplo la siguiente sentencia:

```
FilePermission p = new FilePermission("/tmp/*","read,write");
```

establece que es posible leer y escribir cualquier archivo del directorio `/tmp/`.

La jerarquía de permisos se muestra en el siguiente diagrama:



3.1. Archivos de política de seguridad

Cuando se utiliza un administrador de seguridad, en el instante en el que se cargan las clases, se asignan permisos solicitando a un objeto del tipo Policy que asigne los permisos para el origen de código de cada clase.

La clase de política está establecida en el archivo `java.security` del subdirectorio `jre/lib/security` del directorio donde está instalado de JRE. Por defecto dentro de este archivo está la siguiente línea:

```
policy.provider = sun.security.provider.PolicyFile
```

Es decir, los permisos se deben especificar en un fichero. La siguiente línea muestra un ejemplo de permiso en un fichero de política de seguridad:

```
permission java.io.FilePermission "/tmp/*","read,write";
```

Se pueden crear archivos de políticas en localizaciones prefijadas. Por defecto existen dos localizaciones:

- el archivo `java.policy` en el directorio `jre/lib/security`.
- el archivo `.java.policy` en el directorio raíz del usuario.

Estas localizaciones están establecidas por defecto en el fichero `java.security` de la siguiente forma:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Durante la fase de desarrollo es mejor no modificar estos ficheros estándar `java.policy` o `.java.policy` (ya que los podríamos dejar en un estado peligroso).

Es mejor especificar de forma explícita el archivo de políticas necesario para cada aplicación.

Para ello hay que colocar los permisos en un determinado archivo (por ejemplo `aplicacion.policy` e iniciar la máquina virtual del siguiente modo:

```
java -Djava.security.policy=aplicacion.policy Aplicacion
```

Para applets cargados con `appletviewer` se utiliza (si el fichero de política se llama `applet.policy`).

```
appletviewer -J-Djava.security.policy=applet.policy applet.html
```

El archivo de políticas especificado se añade a las políticas especificadas en los ficheros `java.policy` en el directorio `jre/lib/security` y `.java.policy` en el directorio raíz del usuario. Si se añade un segundo signo igual entonces **solo** se utiliza el archivo de política especificado, ignorando los otros.

```
appletviewer -Djava.security.policy==aplicacion.policy Aplicacion
```

Como ya se comentó anteriormente, las aplicaciones Java ejecutadas localmente, por defecto, no instalan ningún administrador de seguridad. Para ver el efecto de los ficheros de políticas en las aplicaciones hay que utilizar un administrador de seguridad, para ello hay dos posibilidades:

1. Añadir la siguiente línea en el método main(.):

   ```
   System.setSecurityManager(new SecurityManager());
   ```

   y especificar el fichero de política en la línea de órdenes,

2. o realizarlo todo desde la línea de órdenes:

   ```
   java -Djava.security.manager -Djava.security.policy=aplicacion.policy Aplicacion
   ```

En el directorio `ejemplos/Seguridad/Ejemplo5` hay un ejemplo de fichero de política de seguridad.

Vamos a ver ahora como crear ficheros de política de seguridad.

Un archivo de política contiene una secuencia de entradas que comienzan con grant. Cada una de ellas tiene la forma:

```
grant codigoFuente{
   permiso_1;
   permiso_2;
      ...
};
```

codigoFuente contiene un código base (que puede omitirse en el caso en que los permisos se apliquen a todas las fuentes) y los nombres de las entidades certificadoras seguras (que pueden omitirse si no son necesarias las firmas para esta entrada).

A continuación se muestran varios ejemplos de código base:

```
grant codeBase "http://informatica.uv.es/it3guia/clases/" {...}
grant codeBase "http://informatica.uv.es/it3guia/clases/fichero.jar" {...}
grant codeBase "file:c:\\java\\programas\\clases\\" {...}
```

En la primera línea se conceden los permisos al código que esté en el directorio `/it3guia/clases/` en la máquina
`http://informatica.uv.es`.

En la segunda línea se conceden los permisos al código que se encuentre en el fichero

`/it3guia/LP/clases/fichero.jar` en la máquina
`http://informatica.uv.es` pero no a ningún otro fichero en el directorio `/it3guia/clases/`

En la tercera línea se conceden los permisos al código que se encuentre en el directorio
`c:/java/programas/clases/` en la máquina local.

En cuanto a los permisos, estos tienen la siguiente estructura:

```
permission nombreClase nombreDestino, listaAcciones
```

- `nombreClase` es el nombre completo (incluyendo la información del paquete al que pertenece) de una clase de permisos (por ejemplo java.net.SocketPermission).
- `nombreDestino` es una especificación del permiso, en el ejemplo anterior sería especificar el servidor y quizá un numero de puerto.

- `listaAcciones` es también una especificación de permiso, se trata de una lista de acciones como `read` o `connect` separadas por comas.

Algunas clases de permisos no necesitan que se especifique `nombreDestino` o `listaAcciones`.

La siguiente tabla muestra algunas clases de permisos:

java.io.FilePermission	Para asignar permisos de acceso, creación, borrado o ejecución a ficheros.
java.net.SocketPermission	Para asignar permisos para aceptar conexiones, conectarse, escuchar o resolve de sockets.
java.util.PropertyPermission	Para establecer que propiedades se pueden leer o escribir.
java.lang.RuntimePermission	Para establecer si se puede crear un cargador de clases, si se puede obtener el que se está utilizando, si se puede finalizar la máquina virtual, si se puede imprimir, si se puede parar un hilo,...
java.awt.AWTPermission	Para establecer si se puede acceder al portapapeles, si se puede acceder a la cola de eventos,...
java.net.NetPermission	
java.security.SecurityPermission	Para establecer si se puede tener acceso a objetos relativos con la seguridad como por ejemplo Security, Policy, Provider ...
java.security.AllPermision	Otorga todos los permisos, por lo tanto debe ser utilizada sólo durante pruebas.

Vamos a ver con detalle algunos de estos permisos.

java.io.FilePermission

Destinos válidos:

archivo	Un archivo
directorio/*	Todos los archivos de un directorio
*	Todos los archivos del directorio actual
directorio/-	Todos los archivos y subdirectorios del directorio
-	Todos los archivos y subdirectorios del directorio actual
<<ALL_FILES>>	Todos los archivos del sistema de archivos

Acciones posibles: `read, write, delete, execute`

Por ejemplo, java.io.FilePermission(/tmp/*", read") implica java.io.FilePermission(/tmp/a.txt", read") pero no implica nada sobre java.net.NetPermission

Supongamos que se permite a un applet escribir en todo el sistema de ficheros. Esto puede permitir al applet reemplazar cualquier fichero, incluyendo la posibilidad de reemplazar la máquina virtual y los ficheros de configuración por lo que a efectos prácticos es como haber asignado todos los permisos al applet.

java.net.SocketPermission

Destinos para la especificación de la máquina:

hostname o IPaddress	Nombre o dirección IP de la máquina
localhost o cadena vacía	La máquina local
*.sufijoDominio	Cualquier máquina cuyo dominio finalice con el sufijo indicado
*	Cualquier máquina

Destinos para la especificación del rango de puertos (es opcional)

:n	Un número de puerto
:n-	Cualquier puerto a partir de n
:-n	Cualquier puerto anterior a n
:n_1-n_2	Cualquier puerto entre n_1 y n_2

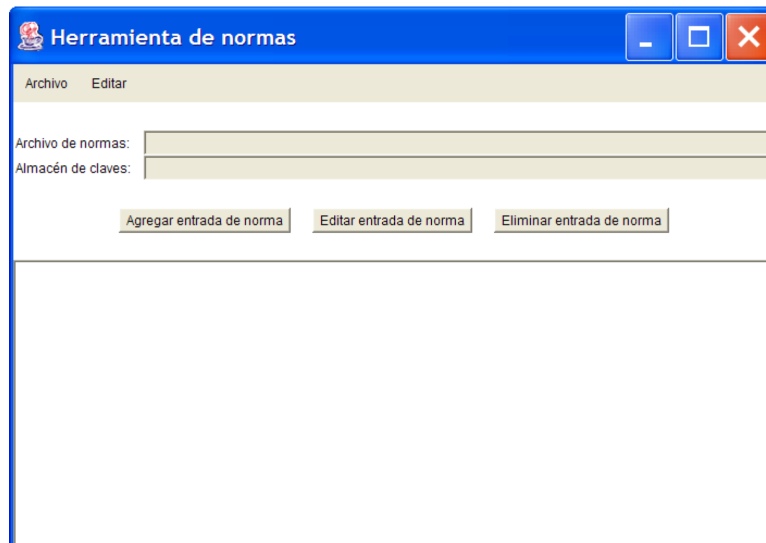
Acciones posibles: `accept, connect, listen, resolve`.

java.lang.RuntimePermission

Algunos destinos válidos son:

createClassLoader, getClassLoader, setContextClassLoader, setSecurityManager, createSecurityManager, exitVM, setFactory, setIO, modifyThread, stopThread, modifyThreadGroup, getProtectionDomain, readFileDescriptor, writeFileDescriptor, loadLibrary.library name, accessClassInPackage.package name, defineClassInPackage.package name, accessDeclaredMembers.class name, queuePrintJob

El JSDK proporciona una herramienta para crear o editar ficheros de políticas de seguridad llamada `policytool`.



3.2. Firmas digitales

En esta sección veremos cómo se puede asegurar la integridad de los datos (mediante *message digest*) y cómo se pueden utilizar las firmas digitales para comprobar la identidad del firmante.

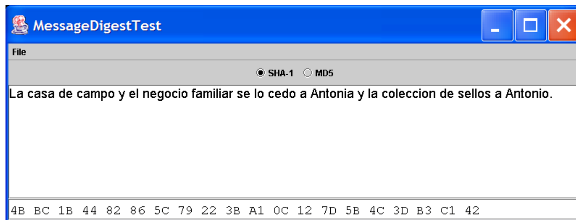
Veremos cómo se puede firmar el código y cómo se puede utilizar esto para desarrollar applets firmados.

3.2.1. Compendio de mensaje

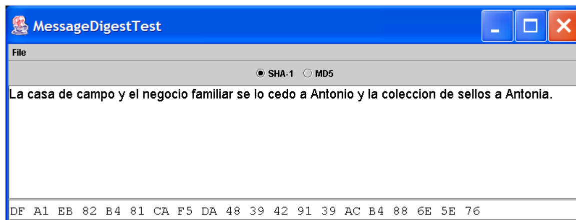
Un compendio de mensaje (*message digest*) es una marca digital obtenida a partir de un bloque de datos.

Se pueden utilizar para detectar cambios en los archivos de datos.

Ejemplo:



Este es el texto del testamento con el message digest obtenido mediante el algoritmo SHA1. El texto se deposita en el notario A y el message digest en el notario B.



Antonio descubre el texto y contrariado entra por la noche en el despacho del notario A y cambia el texto. Antes de la lectura del testamento se comprueba que el message digest no coincide con el depositado en el notario B, determinando que el texto ha sido modificado.

Las propiedades que debe cumplir un algoritmo de compendio de mensaje son:

- Si se modifican los datos entonces el compendio debe cambiar.
- Debe ser extremadamente improbable obtener un mensaje cuyo compendio sea igual al de otro mensaje.

Java proporciona dos algoritmos:

- SHA1 (Secure Hash Algorithm) desarrollado por el National Institute of Standards and Technology.
- MD5 (Message Digest 5) desarrollado por Ronald Rivest.

3.2.2. Mensajes firmados

Para firmar mensajes se utilizan firmas digitales.

Las firmas digitales se basan en la criptografía de clave pública.

En la criptografía de clave pública se utilizan un par de claves una pública (que puede ser distribuida a los demás usuarios) y otra privada (que debe ser conocida únicamente por el usuario). Entre las dos claves existe una relación matemática.

Hay diversos algoritmos contemplados en el DSS (Digital Signature Standard):

- DSA (Digital Signature Algorithm) Java incluye clases para el trabajo con este método.
- RSA (Rivest, Shamir, Adleman).
- ECDSA (Elliptic Curve Digital Signature Algorithm)

Java proporciona tres algoritmos para trabajar con DSA:

1. Para generar el par de claves
2. Para firmar un mensaje
3. Para verificar una firma

El par de claves sólo se genera una vez y se utilizarían para firmar y verificar cualquier mensaje.

Para generar las claves se utiliza la clase SecureRandom que genera números aleatorios más seguros (más aleatorios) que los generados utilizando la clase Random. Uno de los métodos de SecureRandom es setSeed(byte[] seed) que se puede utilizar para proporcionar una semilla para iniciar la secuencia de números.

Si no se proporciona semilla genera una semilla de 20 bytes lanzando hilos, poniéndolos a dormir y midiendo el tiempo exacto en el que estos despiertan.

La implementación típica de la firma digital comprende un algoritmo de compendio de mensaje y un algoritmo de clave pública para encriptar el compendio. Supongamos dos usuarios Alicia y Roberto:

- Alicia reduce un mensaje m a un compendio c, a continuación encripta el compendio c con su clave privada, obteniendo un compendio encriptado σ. Envía el mensaje m junto con el compendio encriptado σ a Roberto. Las dos partes forman el mensaje firmado digitalmente.
- Roberto desencripta el message digest encriptado σ con la clave pública de Alicia, obteniendo el compendio c; a continuación a partir del mensaje m, obtiene el compendio c' y lo compara con el desencriptado c. Si los dos coinciden, acepta el mensaje.

Una vez que se dispone del par de claves, se firma el mensaje con la clave privada y se entrega. Alguien que disponga de la clave pública puede comprobar la autenticidad de ese mensaje.

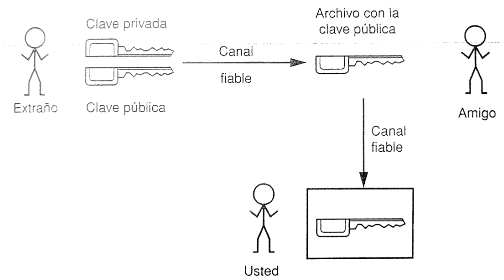
Este esquema todavía tiene un problema:

Supongamos que tenemos por un lado un mensaje firmado con la clave privada y por otro lado la clave pública. Lo único que podemos comprobar es si el mensaje fue firmado con la clave privada adecuada y que no ha sido modificado. Pero todavía no se tiene ni idea de quién es la persona que envió en mensaje ya que cualquiera podría haber generado un par de claves, firmado el mensaje con la clave privada y enviar el mensaje junto con la clave pública.

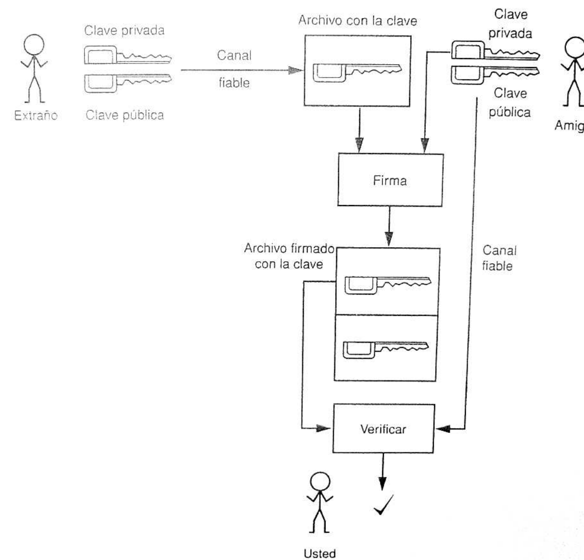
El problema de determinar la identidad de la persona que realizó el envío recibe el nombre de autenticación.

3.2.3. Autenticación de usuarios

Una posibilidad inmediata de solventar el problema de la autenticación es la siguiente:



Otra posibilidad es utilizar a alguien de confianza:



La certificación digital consiste en que una autoridad de certificación (CA) firma un mensaje especial m que contiene la identificación de un usuario y su clave pública de forma que cualquiera puede verificar que un mensaje fue firmado por una autoridad de certificación y pueda por tanto confiar en la clave pública del usuario.

El proceso (con Alicia, Roberto y la CA) se describe a continuación:

- Alicia envía una petición de certificado que contiene su identificación y su clave pública a una CA.
- La CA forma un mensaje especial m a partir de la petición y lo firma con su clave privada obteniendo una firma δ. La CA envía el mensaje m y la firma δ a Alicia. Las dos partes forman un certificado.
- Alicia envía a Roberto el certificado para que éste confíe en su clave pública.
- Roberto verifica la firma δ utilizando la clave pública de la CA. Si todo es correcto, acepta la clave pública de Alicia.

os de certificado más extendido es el formato X.509. Este estándar es parte de las series X.500 del CCITT.

En el formato más sencillo, un certificado X.509 contiene los siguientes datos:

- Versión del formato de certificado
- Número de serie del certificado
- Identificador del algoritmo de firma
- Nombre del firmante del certificado
- Periodo de validez (inicio y final)
- Nombre de la identidad que se certifica
- Clave pública de la identidad que se certifica (Algoritmo, parámetros del algoritmo y clave pública).
- Firma (de todo lo anterior codificados con la clave privada del firmante)

Sun proporciona una herramienta para la generación de claves y almacenes: `keytool`:

Algunas de las opciones que admite son:

-genkey	genera un par de claves y las guarda en un almacén
-list	lista el contenido de un almacén
-export	para exportar la clave pública a un archivo de certificado. El certificado resultante está autofirmado.
-printcert	Para visualizar la información de un certificado
-import	Si se confía en un certificado se puede añadir al almacén
-certreq	Genera una petición de firma de certificado (*Certificate Signing Request*) en formato PKCS#10.
-delete	Elimina una entrada del almacén.

Ejemplos de utilización:

```
keytool -genkey -keystore LP.almacen -alias LP

Escriba la contraseña del almacen de claves:  contraseña
+Cuales son su nombre y su apellido?
  [Unknown]:  Nombre Apellidos
+Cual es el nombre de su unidad de organizacion?
  [Unknown]:  Unidad
+Cual es el nombre de su organizacion?
  [Unknown]:  Organizacion
+Cual es el nombre de su ciudad o localidad?
  [Unknown]:  Ciudad
+Cual es el nombre de su estado o provincia?
  [Unknown]:  Provincia
+Cual es el codigo de pais de dos letras de la unidad?
  [Unknown]:  ES
+Es correcto CN=Nombre Apellidos, OU=Unidad, O=Organizacion, L=Ciudad, ST=Provincia, C=ES?
  [no]:  y

Escriba la contraseña clave para <LP>
  (INTRO si es la misma contraseña que la del almacen de claves):  LPcontraseña
```

Genera el fichero `LP.almacen` con claves (privada y pública) cuyo alias será `LP`.

Para ver el contenido del almacén:

```
keytool -list -keystore LP.almacen

Escriba la contraseña del almacen de claves:  contraseña

Tipo de almacen de claves: jks
Proveedor de almacen de claves: SUN

Su almacen de claves contiene entrada 1

lp, 04-may-2004, keyEntry,
Huella digital de certificado (MD5): A6:CB:58:D0:5E:6C:5D:17:87:FC:9A:85:BC:93:FD:EA
```

Para generar un Certificate Signing Request:

```
keytool -certreq -keystore LP.almacen -alias LP -file LP_CSR.txt
Escriba la contraseña del almacen de claves:  contraseña
Escriba la contraseña clave para <LP> LPcontraseña
```

Genera el fichero `LP_CSR.txt`:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIICeTCCAjcCAQAwdTELMAkGA1UEBhMCRVMxEjAQBgNVBAgTCVByb3ZpbmNpYTEPMA0GA1UEBxMG
Q2l1ZGFkMRUwEwYDVQQKEwxPcmdhbml6YWNpb24xDzANBgNVBAsTBlVuaWRhZDEZMBcGA1UEAxMQ
Tm9tYnJlIEFwZWxsaWRvczCCAbcwggEsBgcqhkjOOAQBMIIBHwKBgQD9f1OBHXUSKVLfSpwu7OTn
9hG3UjzvRADDHj+AtlEmaUVdQCJR+1k9jVj6v8X1ujD2y5tVbNeBO4AdNG/yZmC3a5lQpaSfn+gE
exAiwk+7qdf+t8Yb+DtX58aophUPBPuD9tPFHsMCNVQTWhaRMvZ1864rYdcq7/IiAxmd0UgBxwIV
AJdgUI8VIwvMspK5gqLrhAvwWBz1AoGBAPfhoIXWmz3ey7yrXDa4V7l5lK+7+jrqgvlXTAs9B4Jn
UVlXjrrUWU/mcQcQgYC0SRZxI+hMKBYTt88JMozIpuE8FnqLVHyNKOCjrh4rs6Z1kW6jfwv6ITVi
8ftiegEkO8yk8b6oUZCJqIPf4VrlnwaSi2ZegHtVJWQBTDv+z0kqA4GEAAKBgGnwRmEhlvFcqa75
F46o8hcbaT//GOF01AUgtt159kdGyCuAjKsvBR0GLj8YfyFkwmX/XbbOBPXGQGxUmGl7QEiMjdyJ
bT+p6fRdxkCC9CimcbIK3bxbDiBxdNnQtmWBADGunbzxIw8TnsJiC1esUAv2dzEd8OyavE8/6+Ks
O9QMoAAwCwYHKoZIzjgEAwUAAy8AMCwCFFaO6AktYVOYkH2BtNncm3J1oQY8AhQ6NBYnrJtaGWz9
xQnF3DdLC/3FNQ==
-----END NEW CERTIFICATE REQUEST-----
```

Este fichero se puede enviar a una entidad certificadora para que lo firme.

Para guardar el certificado en el fichero LP.certificado en formato binario

```
keytool -export -keystore LP.almacen -file LP.certificado -alias LP
Escriba la contraseña del almacen de claves:  contraseña
Certificado almacenado en el archivo <LP.certificado>
```

Para guardar el certificado en el fichero LP.certificado en formato de codificación imprimible RFC 1421:

```
keytool -export -keystore LP.almacen -file LP.certificado -alias LP -rfc
Escriba la contraseña del almacen de claves:  contraseña
Certificado almacenado en el archivo <LP.certificado>
```

A partir cualquiera de estos ficheros se puede ver la información del certificado:

```
keytool -printcert -file LP.certificado
Propietario: CN=Nombre Apellidos, OU=Unidad, O=Organizacion, L=Ciudad, ST=Provincia, C=ES
Emisor: CN=Nombre Apellidos, OU=Unidad, O=Organizacion, L=Ciudad, ST=Provincia,C=ES
Numero de serie: 40974dc3
Valido desde: Tue May 04 10:01:07 CEST 2004 hasta: Mon Aug 02 10:01:07 CEST 2004

Huellas digitales del certificado:
   MD5:  3A:F9:85:6B:04:75:96:97:12:46:A0:11:2E:7E:18:75
   SHA1: A7:28:1F:C2:46:22:D5:82:BB:A7:55:99:9D:BC:6B:EA:74:E5:44:96
```

Si entregamos el fichero con el certificado a otra persona y tiene plena confianza en que es correcto, puede importarlo a un almacén de certificados:

```
keytool -import -keystore certificados.almacen -alias LP -file LP.certificado
Escriba la contraseña del almacen de claves: contraseña
Propietario: CN=Nombre Apellidos, OU=Unidad, O=Organizacion, L=Ciudad, ST=Provincia, C=ES
Emisor: CN=Nombre Apellidos, OU=Unidad, O=Organizacion, L=Ciudad, ST=Provincia,C=ES
Numero de serie: 40974dc3
Valido desde: Tue May 04 10:01:07 CEST 2004 hasta: Mon Aug 02 10:01:07 CEST 2004

Huellas digitales del certificado:
         MD5:  3A:F9:85:6B:04:75:96:97:12:46:A0:11:2E:7E:18:75
         SHA1: A7:28:1F:C2:46:22:D5:82:BB:A7:55:99:9D:BC:6B:EA:74:E5:44:96
+Confiar en este certificado? [no]:  y
Se ha añadido el certificado al almacen de claves
```

Si lista el contenido de su almacén verá lo siguiente:

```
keytool -list -keystore certificados.almacen
Escriba la contraseña del almacen de claves: contraseña

Tipo de almacen de claves: jks
Proveedor de almacen de claves: SUN

Su almacen de claves contiene entrada 1

lp, 04-may-2004, trustedCertEntry,
Huella digital de certificado (MD5): 3A:F9:85:6B:04:75:96:97:12:46:A0:11:2E:7E:1
8:75
```

Una vez que se dispone de las claves se puede firmar el código utilizando `jarsigner`.

Supongamos que se ha desarrollado una aplicación con una serie de clases. Estas clases de empaquetan en un fichero jar:

```
jar -cvf UnApplet.jar *.class
```

El fichero JAR resultante se puede firmar del siguiente modo:

```
jarsigner -keystore LP.almacen -signedjar AppletFirmado.jar UnApplet.jar LP
Enter Passphrase for keystore: contraseña
Enter key password for LP: LPcontraseña
```

Supongamos ahora que este applet (aunque lo mismo se puede aplicar si es una aplicación) lo desea ejecutar alguien en su máquina y que ha obtenido el certificado y que confía en él (sea por que lo ha firmado una CA o porque se lo ha entregado alguien de confianza).

Vamos a plantear dos situaciones:

- Va a ejecutar el applet con `appletviewer`
- Va a ejecutar el applet dentro del navegador.

En el caso del appletviewer hace lo siguiente:

1. Importa el certificado a un almacen (voy a suponer que se llama certificados.almacen).

2. Crea un fichero de política de seguridad en la que permita al código firmado por quien haya emitido el certificado realizar las tareas que considere oportunas.

   Este fichero será algo así:

```
keystore "certificados.almacen";

grant signedBy "LP"
{
   permission java.io.FilePermission "c:\\tmp\\-","read,write";
};
```

3. Lanza appletviewer proporcionándole el archivo de política.

```
appletviewer -J-Djava.security.policy=.java.policy UnApplet.html
```

En el caso de ejecutar el applet dentro del plug-in del navegador la situación es un poco más compleja.

Por defecto, los applets firmados una vez que se acepta que se ejecuten tienen todos los permisos, por lo tanto no podremos aplicar un fichero con la política de seguridad.

Este comportamiento por defecto se puede modificar si en una de las localizaciones donde Java busca los ficheros de políticas añadimos las siguientes líneas:

```
grant {
   permission java.lang.RuntimePermission "usePolicy";
};
```

donde se indica al plug-in que para los applets firmados utilice también los ficheros de políticas de seguridad.

Una vez añadidas estas líneas ya no nos aparece el mensaje inicial indicando si deseamos ejecutar el applet, sino que intentará ejecutarlo y si no tiene permisos asociados fallará en la ejecución y si tiene los permisos necesarios lo ejecutará.

En este caso vamos a asumir que se decide modificar (o crear) `.java.policy` del directorio `$user.home`.

Este fichero con la modificación comentada anteriormente y con las líneas para asignar permisos al código firmado por `LP` sería

```
grant {
   permission java.lang.RuntimePermission "usePolicy";
};

keystore "certificados.almacen";

grant signedBy "LP" {
  permission java.io.FilePermission "c:\\tmp\\-", "read,write";
};
```

Además colocaremos el fichero certificados.almacen en este mismo directorio.

Ahora ya se puede cargar el applet y se ejecutará correctamente si el permiso otorgado es el que necesita el applet.

La siguiente tabla muestra la secuencia de acciones a realizar por los tres actores (CA, Desarrollador y Usuario):

CA	Desarrollador	Usuario
	1) Crear código fuente	
	2) Compilar código fuente	
	3) Empaquetar código fuente	
	4) Crear un par de claves	
	5) Generar un CSR	
	6) Enviar el CSR a una CA	
7) Comprobar identidad		
8) Firmar el certificado y enviarlo		
	9) Importar el certificado al almacén de claves	
	10) Firmar el fichero jar	
	11) Exportar el certificado	
		12) Obtener fichero jar y certificado
		13) Verificar el fichero jar
		14) Ver información sobre el certificado
		15) Importar certificado a un almacén
		16) Crear archivo de política de seguridad
		17) Ejecutar aplicación

4. Encriptación

La biblioteca de clases del J2SDK de Java incluye una serie de clases para trabajar con encriptación de datos (forman lo que se conoce como JCE *Java Cryptography Extension*).

Vamos a ver cómo trabajar con dos tipos de encriptación:

- Codificadores simétricos.
- Codificadores asimétricos o de clave pública.

4.1. Codificadores simétricos sobre bloques

En este tipo de codificación la secuencia de bits de entrada se divide en bloques de tamaño fijo y el algoritmo de codificación produce un bloque de datos codificados.

Hay diversos modos de funcionamiento:

- Modo ECB *Electronic Code Book*. Los datos se dividen en bloques y se codifica cada uno de ellos de forma independiente. Bloques de datos de entrada idénticos se transforman en bloques codificados idénticos.

- Modo CFB *Cipher Feedback Book* o Modo CBC *Cipher Block Chaining*. En este caso la codificación de cada bloque individual depende de los bloques precedentes, por lo que bloques de datos idénticos dan lugar (con alta probabilidad) a bloques codificados diferentes.

Java proporciona la clase Cipher, a través de la cual se obtiene el objeto que realizará la codificación.

```
Cipher codificador = Cipher.getInstance(transformacion);

//o

Cipher codificador = Cipher.getInstance(transformacion, nombreProveedor);
```

El J2SDK proporciona codificadores del proveedor SunJCE que es el predeterminado.

La transformación es un String que puede ser:

- *algoritmo/modo/padding*, por ejemplo

```
    // Algoritmo: DES
    // Modo: CBC
    // Padding: PKCS5Padding
    Cipher codificador = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

- *algoritmo*, por ejemplo

```
    // Algoritmo: DES
    Cipher codificador = Cipher.getInstance("DES");
```

El algoritmo DES se considera obsoleto, se recomienda utilizar AES pero el proveedor SunJCE no lo proporciona por lo que habría que utilizar otros proveedores.

Una vez se dispone del objeto que va a realizar el cifrado hay que establecer la forma de trabajo y generar una clave.

La forma de trabajo puede ser una de las siguientes opciones:

Flujo de entrada	Descripción
Cipher.ENCRYPT_MODE	Para encriptar un mensaje
Cipher.DECRYPT_MODE	Para desencriptar un mensaje.
Cipher.WRAP_MODE	Para encriptar una clave.
Cipher.UNWRAP_MODE	Para desencriptar una clave.

En cuanto a la generación de una clave veremos dos opciones:

Opción 1	
	1) Crear un objeto del tipo KeyGenerator especificando el tipo de algoritmo.
	2) Crear un objeto del tipo SecureRandom (semilla aleatoria).
	3) Establecer la semilla.
	4) Generar un objeto del tipo Key que contiene la clave.

```
String algoritmo = ...;
KeyGenerator keygen = KeyGenerator.getInstance(algoritmo);
SecureRandom random = new SecureRandom();
keygen.init(random);
Key clave = keygen.generateKey();
```

Opción 2	
	1) Crear un objeto del tipo SecretKeyFactory especificando el tipo de algoritmo. 2) Generar un array de bytes (semilla) 3) Crear un objeto del tipo SecretKeySpec a partir de la semilla y especificando el algoritmo. 4) Generar un objeto del tipo Key que contiene la clave.

```
String algoritmo = ...;
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(algoritmo);
byte[] semilla = ...;
SecretKeySpec  espec = new SecretKeySpec(semilla, algoritmo);
Key clave = keyFactory.generateSecret(espec);
```

Una vez que se dispone de la clave y se ha fijado el modo de trabajo hay que inicializar el objeto del tipo Cipher del siguiente modo:

```
codificador.init(modo, clave);
```

Una vez inicializado el codificador, ya se puede llamar al los métodos update(...) y doFinal(...) para realizar la codificación.

```
int tamEntrada = codificador.getBlockSize();
byte[] entrada = new byte[tamEntrada];

int tamSalida = codificador.getOutputSize(tamEntrada);
byte[] salida = new byte[tamSalida];

mientras queden datos para codificar
   entrada ← leer datos
   int longitud = codificador.update(entrada,0,tamEntrada,salida);
   guardar salida
fin_mientras

si quedan num_bytes < tamEntrada por codificar
   salida = codificador.doFinal(entrada,0,num_bytes);
sino
   salida = codificador.doFinal();
fin_si

guardar salida
```

Vamos a ver un ejemplo completo formado por 4 clases:

Clase	Descripción
GeneraClave	Genera una clave y la almacena en un fichero
Cifra	Clase de utilidad con un método estático cifra que realiza la tarea especificada en el objeto del tipo Cipher.
Encripta	Para encriptar datos.
Desencripta	Para desencriptar datos.

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class GeneraClave{
   public static void main(String[] args){

      if (args.length!=2){
          System.out.println("\nNumero incorrecto de entradas.");
          System.out.println("Uso:");
          System.out.println("java GeneraClave algoritmo ficheroClave\n");
      }else{
         try{
            KeyGenerator keyGen = KeyGenerator.getInstance(args[0]);
            SecureRandom aleat = new SecureRandom();
            keyGen.init(aleat);
            SecretKey clave = keyGen.generateKey();

            // Escritura de la clave en un ObjectOutputStream

            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[1]));
            out.writeObject(clave);
            out.close();
         }catch(IOException e){
            System.err.println("Error E/S");
         }catch(NoSuchAlgorithmException e){
            System.err.println("Algoritmo de codificacion inexistente");
         }
      }

   }
}
```

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

/** Clase Cifra con un metodo de utilidad
  */
public class Cifra{

   /** Metodo cifra.
     * @param in Origen de los datos a encriptar
     * @param out Fuente de los datos encriptados
     * @param codificador objeto del tipo Cipher que realiza el cifrado.
     */
   public static void cifra(InputStream in, OutputStream out,Cipher codificador) throws
        IOException, GeneralSecurityException{

      int tamEntrada = codificador.getBlockSize();
      byte[] entrada = new byte[tamEntrada];

      System.out.println("Tamaño del bloque de entrada: " + tamEntrada);


      int tamSalida = codificador.getOutputSize(tamEntrada);
```

```
      byte[] salida = new byte[tamSalida];

      System.out.println("Tamaño del bloque de salida: " + tamSalida);

      int numLeidos = 0;

      boolean finalizar = false;

      while (!finalizar){

         numLeidos = in.read(entrada);

         if (numLeidos == tamEntrada){
            int longitud = codificador.update(entrada,0,tamEntrada,salida);
            out.write(salida,0,longitud);
         }
         else
            finalizar = true;
      }

      if (numLeidos>0)
         salida = codificador.doFinal(entrada,0,numLeidos);
      else
         salida = codificador.doFinal();

      out.write(salida);

   }
}
```

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class Encripta{
   public static void main(String[] args){
      // Modo de operacion: encriptar
      int modo = Cipher.ENCRYPT_MODE;

      if (args.length!=4){
          System.out.println("\nNumero incorrecto de entradas.");
          System.out.println("Uso:");
          System.out.println("java Encripta algoritmo ficheroClave ficheroEntrada
               ficheroCodificado\n");
      }else{
         try{
            // Lectura de la clave
            ObjectInputStream cl = new ObjectInputStream(new FileInputStream(args[1]));
            Key clave = (Key)cl.readObject();
            cl.close();

            InputStream in = new FileInputStream(args[2]);
            OutputStream out = new FileOutputStream(args[3]);

            Cipher codificador = Cipher.getInstance(args[0]);
            codificador.init(modo,clave);

            Cifra.cifra(in,out,codificador);
            in.close();
            out.close();
         }catch(IOException e){
            System.err.println("Error E/S");
         }catch(GeneralSecurityException e){
            System.err.println("Error en el algoritmo de encriptacion.");
         }catch(ClassNotFoundException e){
            System.err.println("Error en la recuperacion de la clave");
```

```
            }
        }
    }

}
```

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class Desencripta{
    public static void main(String[] args){
        // Modo de operacion: desencriptar
        int modo = Cipher.DECRYPT_MODE;

        if (args.length!=4){
            System.out.println("\nNumero incorrecto de entradas.");
            System.out.println("Uso:");
            System.out.println("java Desencripta algoritmo ficheroClave ficheroCodificado
                ficheroDecodificado\n");
        }else{
            try{
                // Lectura de la clave
                ObjectInputStream cl = new ObjectInputStream(new FileInputStream(args[1]));
                Key clave = (Key)cl.readObject();
                cl.close();

                InputStream in = new FileInputStream(args[2]);
                OutputStream out = new FileOutputStream(args[3]);

                Cipher codificador = Cipher.getInstance(args[0]);
                codificador.init(modo,clave);

                Cifra.cifra(in,out,codificador);
                in.close();
                out.close();
            }catch(IOException e){
                System.err.println("Error E/S");
            }catch(GeneralSecurityException e){
                System.err.println("Error en el algoritmo de encriptacion.");
            }catch(ClassNotFoundException e){
                System.err.println("Error en la recuperacion de la clave");
            }
        }
    }

}
```

4.2. Codificadores asimétricos o de clave pública

La idea básica es que se dispone de un par de claves: una clave pública y otra privada.

Alguien que desee enviar un mensaje secreto a otra persona o entidad debe utilizar la clave pública de la otra parte para encriptar el mensaje.

La mayor parte de los algoritmos de clave pública son codificadores que actúan sobre bloques de datos, que tratan el mensaje como una secuencia de enteros y se basan su seguridad en la dificultad de resolver un determinado problema matemático.

El algoritmo más conocido es el RSA (Rivest, Shamir y Adleman) en el que el problema matemático es el de la factorización de un número en dos números primos.

Para que el código resultante sea seguro, el tamaño de las claves y los bloques debe se grande.

El tamaño de bloque típico en codificadores simétricos es de 64 o 128 bits, mientras que el tamaño de clave típico para RSA es de al menos 640 bits y el tamaño de los bloques de 1024 o 2048 bits.

Debido al mayor tamaño de los bloques y las claves, los codificadores asimétricos son más lentos que los simétricos por lo que son más utilizados para firmas digitales y para codificar claves simétricas.

En el directorio `Ejemplos/Encriptacion/Asimetrico` hay una serie de directorios que contienen las clases que se detallan a continuación. Estas clases muestran el uso de conjunto de la criptografía de clave simétrica y asimétrica.

Clase	Descripción
GeneraClavesRSA	Genera una clave pública y otra privada utilizando RSA y las almacena en ficheros.
EncriptaMensaje	Clase que genera una clave simétrica, encripta un fichero con esta clave y encripta la clave simétrica utilizando la clave pública.
DesencriptaMensaje	Desencripta la clave simétrica utilizando la clave privada. Una vez ha desencriptado la clave simétrica desencripta el mensaje.

La secuencia de acciones a realizar por los dos actores (Destinatario y Remitente) es:

Destinatario	Remitente
1) Genera claves pública y privada	
	2) Obtiene clave pública del destinatario
	3) Genera clave simétrica y encripta el mensaje usando la clave simétrica
	4) Encripta la clave simétrica con la clave pública del destinatario
	5) Envía la clave simétrica encriptada con la clave pública y el mensaje encriptado con la clave simétrica.
6) Desencripta la clave simétrica con la clave privada	
7) Desencripta el mensaje	