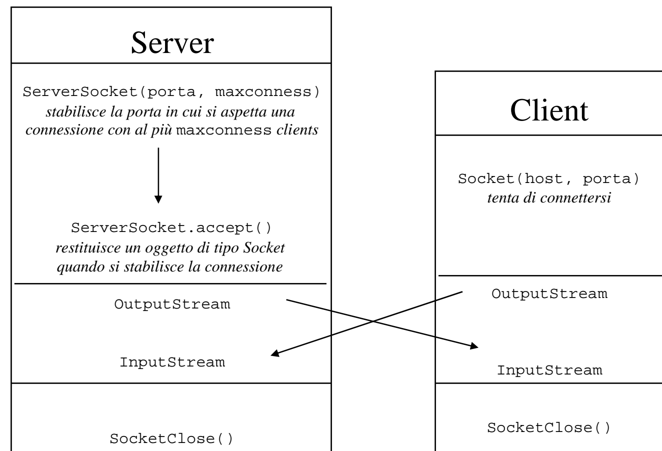
Networking

- Java permette comunicazioni in rete basate sul concetto di socket, che permette di vedere la comunicazione in termini di flusso (stream), in modo analogo all'input-output di file, usando
  - *Stream sockets*
    un processo stabilisce una comunicazione con un altro processo (stream continuo), usando come protocollo TCP (Transmission Control Protocol)
  - *Datagram sockets*
    vengono trasmessi pacchetti di informazione, usano il protocollo UDP (User Datagram Protocol)
- La libreria di riferimento è il package `java.net`

Connessione

- Un socket mantiene *due flussi*: input e output.
  - Un processo *invia dei dati* ad un altro processo attraverso la rete semplicemente scrivendo nel flusso di output associato al socket
  - Un processo *riceve dei dati* scritti da un altro processo leggendo il flusso di input associato al socket
- Per stabilire una connessione
  - una macchina deve far girare un programma che è "in attesa" di connessione, e un'altra macchina deve cercare di connettersi con essa, in modo analogo alla trasmissione di un fax
  - bisogna conoscere l'indirizzo della macchina con cui ci si vuole connettere
  - bisogna conoscere un numero di porta (in TCP/IP un numero tra 1025 e 65535). Sia il client che il server devono convenire sul numero di porta da usare

Connessione TCP/IP



Server TCP/IP

```
// Inizializza un server che riceverà una connessione da un client, manderà
// una stringa al client, ne ricevera' una, e chiuderà la connessione

import java.io.*;
import java.net.*;
import java.awt.*;

public class Server extends Frame {
   private TextArea display;

   public Server(){
      super( "Server" );
      display = new TextArea( "", 0, 0, TextArea.SCROLLBARS_VERTICAL_ONLY );
      add( display, BorderLayout.CENTER );
      setSize( 300, 150 );
      setVisible( true );
   }

   public static void main( String args[] ){
      Server s = new Server();

      s.addWindowListener( new CloseWindowAndExit() );    // da implementare
      s.runServer();
   }

   public void runServer(){
      ServerSocket server;
      Socket connection;
      DataOutputStream output;
      DataInputStream input;
      int counter = 1;
```

```
      try {
         // Passo 1 Crea un ServerSocket
         server = new ServerSocket( 5000, 100 );

         while ( true ) {
            // Passo 2 Resta in attesa di una connessione
            connection = server.accept();
            display.append( "Connessione " + counter + " richiesta da : " +
               connection.getInetAddress().getHostName() );

            // Passo 3: Inizializza i flussi di input e output
            input  = new DataInputStream( connection.getInputStream() );
            output = new DataOutputStream( connection.getOutputStream() );
            display.append( "\nI Flussi di I/O sono a posto\n" );

            // Passo 4: Comandi durante la connessione.
            display.append("Mando il messaggio \"Connessione perfetta!\"\n" );
            output.writeUTF( "Connessione perfetta!" );                 // invia
            display.append( "Messaggio del client: " + input.readUTF() );      // riceve

            // Passo 5: Chiude la connessione.
            display.append( "\nTrasmissione finita " + "Chiudo il socket.\n\n" );
            connection.close();
            ++counter;
         }
      }
      catch ( IOException e ) {
         e.printStackTrace();
      }                                                  // nb: UTF=Unicode Text Format
   }
}
```


Client TCP/IP

```
// Inizializza un Client che leggerà l'informazione spedita dal
// Server e la mostrerà sul display.

import java.io.*;
import java.net.*;
import java.awt.*;

public class Client extends Frame {
   private TextArea display;

   public Client(){
      super( "Client" );
      display = new TextArea( "", 0, 0, TextArea.SCROLLBARS_VERTICAL_ONLY );
      add( display, BorderLayout.CENTER );
      setSize( 300, 150 );
      setVisible( true );
   }

   public static void main( String args[] ){
      Client c = new Client();

      c.addWindowListener( new CloseWindowAndExit() );   // da implementare
      c.runClient();
   }
```

```
public void runClient(){
      Socket client;
      DataInputStream input;
      DataOutputStream output;

      try {
         // Passo 1: Crea un Socket per entrare in connessione.
         client = new Socket( InetAddress.getLocalHost(), 5000 );
         display.append( "Connessione con: " + client.getInetAddress().getHostName());

         // Passo 2: Inizializza i flussi di input e di output.
         input  = new DataInputStream( client.getInputStream() );
         output = new DataOutputStream( client.getOutputStream() );
         display.append( "\nI Flussi di I/O sono a posto\n" );

         // Passo 3: Processi durante la connessione
         display.append( "Messaggio del server: " + input.readUTF() ); // riceve
         display.append( "Invio il messaggio \"Grazie mille.\"\n" );
         output.writeUTF( "Grazie mille." );                           // invia

         // Passo 4: Chiude la connessione.
         display.append( "Trasmissione completata. " +"Chiudo la connessione.\n" );
         client.close();
      }
      catch ( IOException e ) {
         e.printStackTrace();
      }
    }
}
```


La classe Socket

- `Socket (InetAddress indirizzo, int porta)`

  Crea uno stream socket e lo connette alla porta e all'indirizzo specificato
- `Socket (String indirizzo, int porta)`

  Come sopra ma l'indirizzo è dato come stringa
- `getInetAddress()`

  Ritorna l'indirizzo al quale il socket è connesso
- `getInputStream() e getOutputStream()`

  Ritornano l'input e l'output stream per un socket
- `getLocalAddress()`

  Ritorna l'indirizzo locale al quale il socket è connesso (nota: un host potrebbe avere piu' indirizzi …)

Esecuzione socket TCP

SERVER	CLIENT
	Connessione con: ihoh I Flussi di I/O sono a posto
Connessione 1 richiesta da : ihoh I Flussi di I/O sono a posto Mando il messaggio "Connessione perfetta!"	
	Messaggio del server: Connessione perfetta! Invio il messaggio "Grazie mille."
Messaggio del client: Grazie mille. Trasmissione finita Chiudo il socket.	
	Trasmissione completata. Chiudo la connessione.

Socket UDP

- UDP: protocollo che non richiede comunicazione sincrona (TCP : telefono / UDP : posta).
- La comunicazione avviene per pacchetti.
- `DatagramPacket` ha due costruttori
  - uno per ricevere dati
    `DatagramPacket(byte[] riceve, int lungh)`
  - uno per inviare dati
    ```
    DatagramPacket(byte[] invia, int lungh,
                   InetAddress indirizzo, int porta)
    ```
- `DatagramSocket` ha tre costruttori
  - `DatagramSocket()` ogni porta sul local host
  - `DatagramSocket(int porta)` porta specificata sul local host
  - `DatagramSocket(int porta, inetAddress indirizzo)` porta specificata sull'indirizzo specificato

UDP Server

```
import java.io.*;
import java.net.*;
import java.awt.*;

public class Server extends Frame {
   private TextArea display;
   private DatagramPacket sendPacket, receivePacket;
   private DatagramSocket socket;

   public Server(){
      super( "Server" );
      display = new TextArea();
      add( display, BorderLayout.CENTER );
      setSize( 400, 300 );
      setVisible( true );
      try {
         socket = new DatagramSocket( 5000 );
      }
      catch( SocketException se ) {
         se.printStackTrace();
         System.exit( 1 );
      }
   }

   public static void main( String args[] ){
      Server s = new Server();
      s.addWindowListener( new CloseWindowAndExit() ); // da implementare
      s.waitForPackets();
   }
```



```
public void waitForPackets(){
     while ( true ) {
        try {
           // inizializza un pacchetto
           byte data[] = new byte[ 100 ];
           receivePacket = new DatagramPacket( data, data.length );
           // aspetta un pacchetto
           socket.receive( receivePacket );               // riceve
           // processa il pacchetto
           display.append( "\nPacket received:" +
              "\nFrom host: " + receivePacket.getAddress() +
              "\nHost port: " + receivePacket.getPort() +
              "\nLength: " + receivePacket.getLength() +
              "\nContaining:\n\t" + new String( receivePacket.getData() ) );
           // risponde al client facendo echo delle informazioni sul pacchetto
           display.append( "\n\nEcho data to client...");
           sendPacket = new DatagramPacket(
                              receivePacket.getData(),
                              receivePacket.getLength(),
                              receivePacket.getAddress(),
                              receivePacket.getPort() );
           socket.send( sendPacket );                     // invia
           display.append( "Packet sent\n" );
        }
        catch( IOException io ) {
           display.append( io.toString() + "\n" );
           io.printStackTrace();
        }
     }
  }
}
```

UDP Client

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class Client extends Frame implements ActionListener {
   private TextField enter;
   private TextArea display;
   private DatagramPacket sendPacket, receivePacket;
   private DatagramSocket socket;

   public Client(){
      super( "Client" );
      enter = new TextField( "Type message here" );
      enter.addActionListener( this );
      add( enter, BorderLayout.NORTH );
      display = new TextArea();
      add( display, BorderLayout.CENTER );
      setSize( 400, 300 );
      setVisible( true );
      try {
         socket = new DatagramSocket();
      }
      catch( SocketException se ) {
         se.printStackTrace();
         System.exit( 1 );
      }
   }
```



```
public void waitForPackets(){
      while ( true ) {
         try {
            // set up packet
            byte data[] = new byte[ 100 ];
            receivePacket = new DatagramPacket( data, data.length );

            // wait for packet
            socket.receive( receivePacket );

            // process packet
            display.append( "\nPacket received:" +
               "\nFrom host: " + receivePacket.getAddress() +
               "\nHost port: " + receivePacket.getPort() +
               "\nLength: " + receivePacket.getLength() +
               "\nContaining:\n\t" + new String( receivePacket.getData() ) );
         }
         catch( IOException exception ) {
            display.append( exception.toString() + "\n" );
            exception.printStackTrace();
         }
      }
   }
```

```
public void actionPerformed( ActionEvent e )
   {
      try {
         display.append( "\nSending packet containing: " + e.getActionCommand() + "\n" );

         String s = e.getActionCommand();
         byte data[] = s.getBytes();

         sendPacket = new DatagramPacket( data, data.length,
            InetAddress.getLocalHost(), 5000 );
         socket.send( sendPacket );
         display.append( "Packet sent\n" );
      }
      catch ( IOException exception ) {
         display.append( exception.toString() + "\n" );
         exception.printStackTrace();
      }
   }

   public static void main( String args[] )
   {
      Client c = new Client();

      c.addWindowListener( new CloseWindowAndExit() );  // da implementare
      c.waitForPackets();
   }
}
```


Esecuzione socket UDP

SERVER

```
Packet received:
From host: sally/157.138.20.104
Host port: 1031
Length: 28
Containing:
     ciao!!

Echo data to client...Packet sent

Packet received:
From host: sally/157.138.20.104
Host port: 1031
Length: 33
Containing:
     questa e' una seconda prova di
     trasmissione

Echo data to client...Packet sent
```

CLIENT

```
Sending packet containing: ciao!!
Packet sent

Sending packet containing: questa e'
     una seconda prova di trasmissione
Packet sent

From host: sally/157.138.20.104
Host port: 5000
Length: 28
Containing:
     ciao!!
Packet received:
From host: sally/157.138.20.104
Host port: 5000
Length: 33
Containing:
     questa e' una seconda prova di
     trasmissione
```

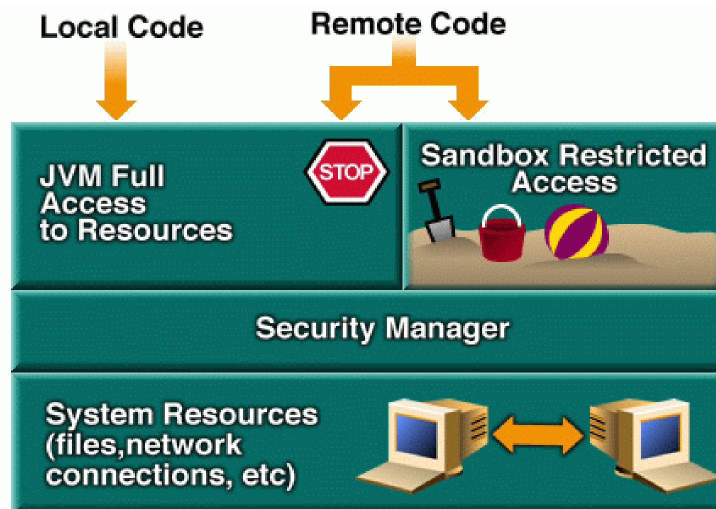
Sicurezza in Java 1.2

- Problema: limitare l'accesso alle risorse di sistema da parte di codice scaricato dalla rete e quindi potenzialmente insicuro (*untrusted*)
- JDK 1.2 migliora notevolmente la gestione della sicurezza rispetto alle versioni precedenti:
  - controllo degli accessi *policy-based,* facilmente configurabile e *fine-grained*
  - nuove classi e interfacce per crittografia, gestione dei certificati e delle chiavi
  - tre nuovi tool per la gestione della sicurezza
- Vediamo un po' di storia

Sicurezza in JDK 1.0: Sandbox

- Il primo modello di Java era piuttosto restrittivo e si basava sul concetto di *Sandbox*
- La *Sandbox* rappresenta un ambiente isolato in cui eseguire il codice non sicuro (untrusted) scaricato dalla rete
- ***Security Manager*** garantisce che: il codice locale ha accesso completo alle risorse del sistema, mentre il codice scaricato dalla rete puo' essere eseguito **solo all'interno della *Sandbox***

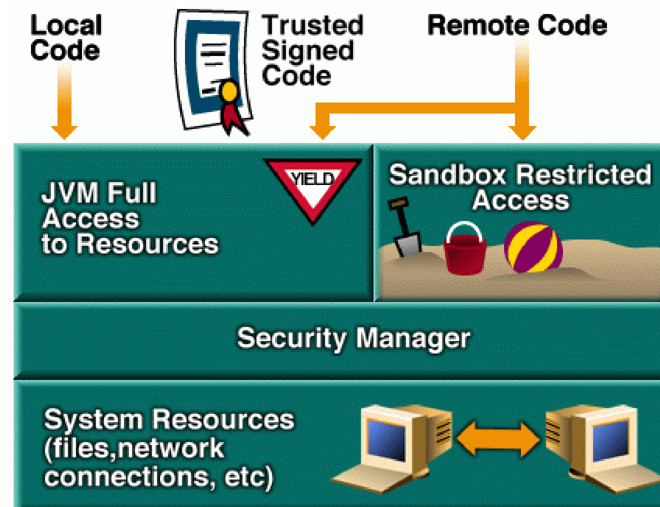
Modello di sicurezza in JDK 1.0



Sicurezza in JDK 1.1

- JDK 1.1 ha introdotto il concetto di *applet firmata*
- Le *applet firmate* sono trattate come codice locale (trusted) ed hanno quindi accesso a tutte le risorse
- Una applet e' *firmata correttamente* se la chiave publica usata per verificarne la firma e' *affidabile* (*trusted*)
- Una applet *non firmata* viene eseguita ancora nella *Sandbox*

Modello di sicurezza in JDK 1.1



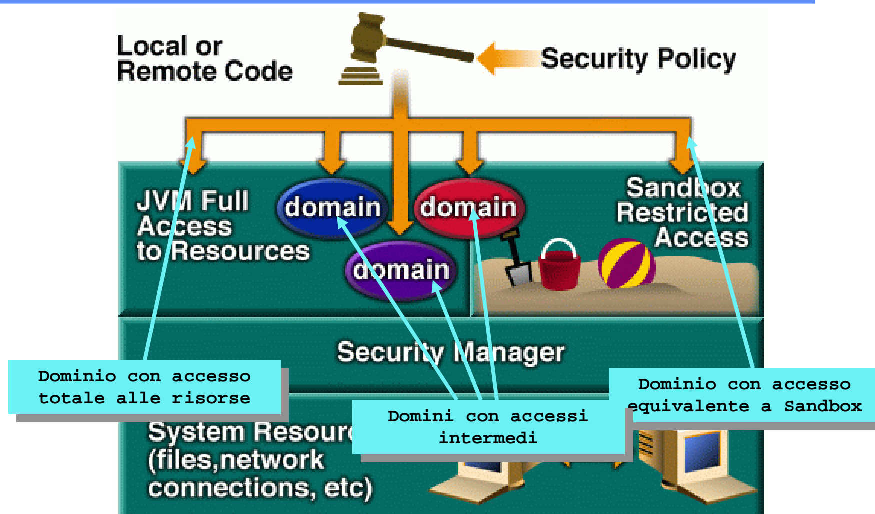
Sicurezza in JDK 1.2

- Migliora molto la gestione della sicurezza!
- tutto il codice (trusted e untrusted) puo' essere gestito tramite una *politica di sicurezza*
- La *politica di sicurezza* stabilisce i privilegi di accesso del codice in base a vari parametri:
  - ***indirizzo di rete di provenienza***
  - ***firma*** (chi ha firmato l'applet)
- Specifica in dettaglio quali sono gli accessi a quali risorse
  - Esempio: ***read/write*** su una particolare directory
  - Esempio: ***connect access*** a un certo **host** su una certa porta

JDK 1.2: i Domini

- Il codice e' organizzato in *domini*
- Ogni *dominio* include un insieme di classi; le istanze delle classi di uno stesso *dominio* hanno gli stessi privilegi di accesso
- Naturalmente, un *dominio* puo' essere configurato come una *Sandbox*: la politica JDK 1.0 e' quindi ancora realizzabile come sottocaso
- Le applicazioni locali per default hanno completo accesso alle risorse ma ora e' anche possibile "filtrarle" tramite una politica di sicurezza

Modello di sicurezza in JDK 1.2



Esempio: osservare le restrizioni

- Provare ad eseguire un'applet che cerca di creare un nuovo file `writetest` nella directory corrente
- Dal sito javasoft:

```
appletviewer
   http://java.sun.com/docs/books/tutorial/
   security1.2/tour1/example-1dot2/WriteFile.html
```

- Il visualizzatore di applet (o il browser) mostra un messaggio riguardante una exception di sicurezza

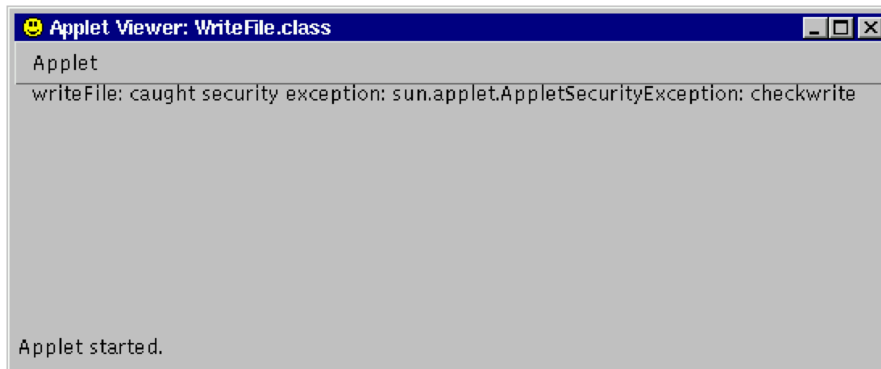
Exception di sicurezza

WriteFile applet

appletviewer

Exception:
WriteFile doesn't have permission
to write to writetest file.

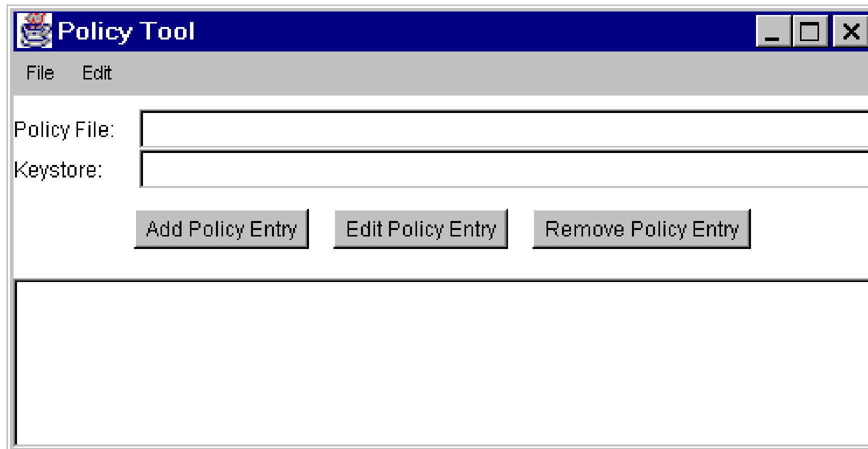
Esempio: osservare le restrizioni



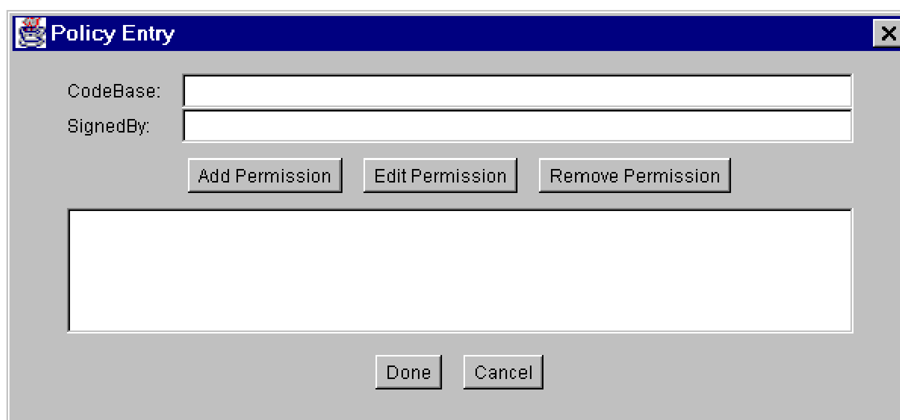
Creare una Politica di Sicurezza

- Si deve generare un *policy file* (**Nota**: il nome di questo file **dipende dal sistema operativo**!)
- Si puo' scrivere con un qualsiasi editor
- Si puo' utilizzare il *Policy Tool;* basta lanciare `policytool` dal prompt
- Evita errori nella generazione del file!
- Quando viene lanciato cerca di caricare la politica di sicurezza esistente; se non la trova mostra una schermata bianca

Policy Tool: schermata iniziale



Bottone *Add Policy*



Creare una politica di sicurezza

- **CodeBase**: indica la locazione di provenienza del codice; vengono garantiti accessi solo al codice che proviene da **CodeBase**
- **Inserire:** `http://java.sun.com/docs/books/tutorial/security1.2/tour1/example-1dot2/`
- **Attenzione**: Se lasciato vuoto concede gli accessi *a qualsiasi locazione*
- Per dare accesso a tutta una gerarchia di directory ("-" indica tutte le sotto-directory):
  `http://java.sun.com/docs/books/tutorial/security1.2/-`

Creare una politica di sicurezza

- **SignedBy**: indica un alias di un *certificato* per una particolare chiave pubblica; garantisce gli accessi *solo al codice firmato* con la corrispondente chiave privata
- Se lasciato vuoto non richiede nessuna firma
- In questo esempio lo lasciamo vuoto!
- Premendo "Add Permission" si passa alla schermata successiva dove e' possibile inserire il *tipo di Permission* (I/O), il *target* (nome file) *e l'azione* (write)

Aggiungere un privilegio

Permissions

Add New Permission:

File Permission	java.io.FilePermission
Target Name:	writetest
Actions:	write
Signed By:	

OK   Cancel

- Salvare il file dal menu “Save As”

Policy Tool

File   Edit

Policy File: C:\Test\mypolicy

Keystore:

Add Policy Entry   Edit Policy Entry   Remove Policy Entry

CodeBase "http://java.sun.com/docs/books/tutorial/security1.2/tour1/example-1dot2/"

Come “usare” una politica

- Deve essere specificata nel “security properties file”:
  - Windows: *java.home*\lib\security\java.security
  - Unix: *java.home*/lib/security/java.security
- Le entry sono della forma (n e’ un numero):

```
policy.url.<n>=file:/Test/mypolicy
```

- oppure esplicitamente:

```
appletviewer -J-Djava.security.policy=mypolicy
    http://java.sun.com/docs/books/tutorial/security1.2/tour1/
            example-1dot2/WriteFile.html
```

- Ora il file verra’ scritto correttamente dall’applet!