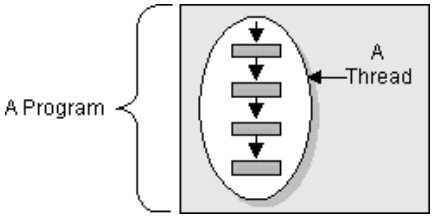
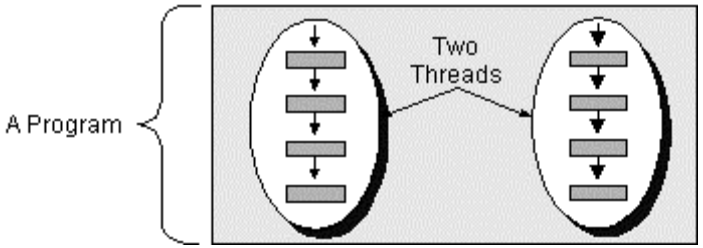
Thread

Un ***thread*** (*lightweight process*) è un singolo flusso sequenziale di controllo all'interno di un processo





Un ***thread***:

- esegue all'interno del contesto di esecuzione di un **unico processo/programma**
- **NON** ha uno spazio di indirizzamento riservato: tutti i thread appartenenti allo stesso processo condividono lo **stesso spazio di indirizzamento**
- ha *execution stack* e *program counter* **privati**

Java Thread

Due modalità per implementare ***thread*** in Java:

1. come sottoclasse della classe **`Thread`**
2. come classe che implementa l'interfaccia **`Runnable`**

1) come sottoclasse della classe **`Thread`**

- **`Thread`** possiede un metodo **`run()`** che la sottoclasse deve ridefinire
- si crea un'istanza della sottoclasse tramite **`new`**
- si esegue un thread chiamando il metodo **`start()`** che a sua volta richiama il metodo **`run()`**

Esempio di classe `Simplethread` che è ***sottoclasse*** di `Thread` (modalità 1):

```
public class SimpleThread extends Thread {

   public SimpleThread(String str) {
      super(str);
      }

      public void run() {
        for (int i = 0; i < 10; i++) {
           System.out.println(i + " " +
               getName());
           try {
               sleep((int)(Math.random()*
                  1000));
          } catch (InterruptedException e){}
           }
        System.out.println("DONE! " +
            getName());
              }


public class TwoThreadsTest {

   public static void main (String[] args) {
     new SimpleThread("Jamaica").start();
     new SimpleThread("Fiji").start();
     }
  }
```


E se occorre definire thread che non siano **necessariamente** sottoclassi di Thread?

2) come classe che implementa l'interfaccia **`runnable`**

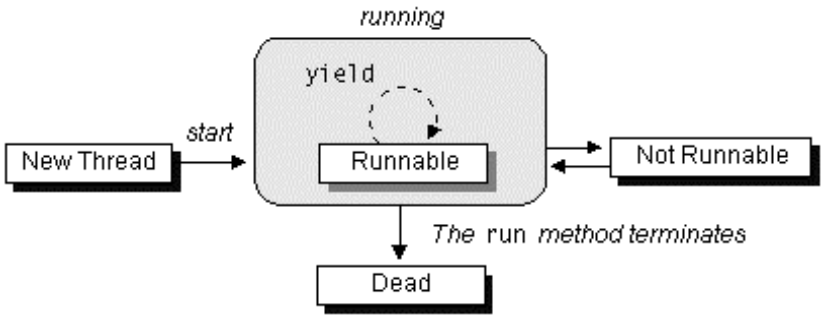
- implementare il metodo **`run()`** nella classe
- creare un'istanza della classe tramite **`new`**
- creare un'istanza della classe **`Thread`** con un'altra **`new`**, passando come parametro l'istanza della classe che si è creata
- invocare il metodo **`start()`** sul thread creato, producendo la chiamata al suo metodo **`run()`**

Interfaccia `Runnable`:
maggiore **flessibilità** derivante dal poter essere sottoclasse di qualsiasi altra classe

Esempio di classe `EsempioRunnable` che ***implementa l'interfaccia `Runnable`*** ed è sottoclasse di `MiaClasse` (modalità 2):

```
class EsempioRunnable extends MiaClasse
   implements Runnable {

   // non e' sottoclasse di Thread

   public void run() {
     for (int i=1; i<=10; i++)
        System.out.println(i + " " + i*i);
   }
}

public class Esempio {

   public static void main(String args[]){

      EsempioRunnable e =
           new EsempioRunnable();
      Thread t = new Thread (e);
      t.start();
   }
}
```


Il ciclo di vita di un thread



- **Creato**
  subito dopo l'istruzione **`new`**
  le variabili sono state allocate e inizializzate; il thread è in attesa di passare allo stato di eseguibile
- **Runnable**
  thread è in esecuzione, o in coda d'attesa per ottenere l'utilizzo della CPU
- **Not Runnable**
  il thread non può essere messo in esecuzione dallo scheduler. Entra in questo stato quando in **attesa** di un'operazione di I/O, o dopo l'invocazione dei metodi **`suspend()`**, **`wait()`**, **`sleep()`**
- **Dead**
  al termine "naturale" della sua esecuzione o dopo l'invocazione del suo metodo **`stop()`** da parte di un altro thread

Metodi per il controllo di thread

`start()` fa ***partire*** l'esecuzione di un thread. La macchina virtuale Java invoca il metodo `run()` del thread appena creato

`stop()` ***forza*** la ***terminazione*** dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente ***liberate*** (lock inclusi), come effetto della propagazione dell'eccezione `ThreadDeath`

`suspend()` ***blocca*** l'esecuzione di un thread in attesa di una successiva operazione di ***resume***. Non libera le risorse impegnate dal thread (possibilità di ***deadlock***)

`resume()` ***riprende*** l'esecuzione di un thread precedentemente ***sospeso***. Se il thread riattivato ha una priorità maggiore di quello correntemente in esecuzione, avrà subito accesso alla CPU, altrimenti andrà in coda d'attesa

`sleep(long t)` blocca per un ***tempo specificato*** (`time`) l'esecuzione di un thread. Nessun *lock* in possesso del thread viene rilasciato.

`join()` ***blocca*** il thread chiamante in attesa della ***terminazione*** del thread di cui si invoca il metodo. Anche con ***timeout***

`yield()` ***sospende*** l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in ***coda d'attesa***

I metodi precedenti interagiscono ***ovviamente*** con il ***gestore della sicurezza*** della macchina virtuale Java (`SecurityManager`, `checkAccess()`, `checkPermission()`)

Altri metodi fondamentali per le operazioni di ***sincronizzazione*** fra thread Java:
`wait()`, **`notify()`**, **`notifyAll()`**

(vedi lucido 11 e seguenti)

Il problema di `stop()` e `suspend()`

`stop()` e `suspend()` rappresentano azioni "brutali" sul ciclo di vita di un thread
=> rischio di determinare situazioni di blocco critico (***deadlock***)

Infatti:

- se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva***, tale risorsa rimane ***bloccata*** e non è utilizzabile da altri, perché il thread sospeso non ha avuto modo di rilasciare il *lock* su di essa
- se il ***thread interrotto*** stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera ***atomica***, l'interruzione può condurre ad uno ***stato inconsistente*** del sistema

➔ JDK 1.2, pur supportandoli ancora per ragioni di *back-compatibility*, ***sconsiglia*** l'utilizzo dei metodi `stop()`, `suspend()` e `resume()` (***metodi deprecated***)

Si consiglia invece di realizzare tutte le azioni di ***controllo*** e ***sincronizzazione*** fra thread tramite i metodi **`wait()`** e **`notify()`** su variabili condizione (astrazione di ***monitor***)

Priorità dei thread in Java

Scheduling: esecuzione di una molteplicità di thread su una singola CPU, in un qualche ordine

Macchina virtuale Java (JVM)
Fixed Priority Scheduling
algoritmo di scheduling molto semplice e deterministico

- JVM sceglie il thread in stato ***runnable*** con ***priorità*** più ***alta***

- Se più thread in attesa di eseguire hanno ***uguale priorità***, la scelta della JVM avviene con una modalità di tipo ***round-robin***.

La classe `Thread` fornisce i metodi:

- `setPriority(int num)`
- `getPriority()`

con valori di `num` compresi fra `MIN_PRIORITY` e `MAX_PRIORITY` (costanti definite anch'esse nella classe `Thread`)

Il ***thread*** messo ***in esecuzione*** dallo scheduler viene ***interrotto*** se e solo se:

- un thread con priorità ***più alta*** diventa ***runnable***;
- il metodo `run` ***termina l'esecuzione*** o il thread esegue un `yield`;
- il ***quanto*** di tempo assegnato si è ***esaurito*** (solo su sistemi che supportano *time-slicing*, come *Windows95/NT*)

```
public void run() {
   while (tick < 200000) {
         tick++;
         if ((tick % 50000) == 0)
            System.out.println("Thread #" +
              num + ", tick = " + tick);
   }
}
```

Time-Sliced System	***Non Time-Sliced System***
Thread #1, tick = 50000	Thread #0, tick = 50000
Thread #0, tick = 50000	Thread #0, tick = 100000
Thread #0, tick = 100000	Thread #0, tick = 150000
Thread #1, tick = 100000	Thread #0, tick = 200000
Thread #1, tick = 150000	Thread #1, tick = 50000
Thread #1, tick = 200000	Thread #1, tick = 100000
Thread #0, tick = 150000	Thread #1, tick = 150000
Thread #0, tick = 200000	Thread #1, tick = 200000

Sincronizzazione di thread

Differenti thread che fanno parte della stessa applicazione Java condividono lo ***stesso spazio*** di memoria

➔ è possibile che ***più thread*** accedano ***contemporaneamente*** allo stesso metodo o alla stessa sezione di codice di un oggetto

Servono meccanismi di sincronizzazione

JVM supporta la definizione di ***monitor*** per la sincronizzazione nell'accesso a risorse tramite la keyword **`synchronized`**

`synchronized` su:

- singolo metodo, o
- blocco di istruzioni

In pratica:

- a ogni oggetto Java è automaticamente associato un ***lock***
- per accedere a un metodo o una sezione `synchronized`, un thread deve prima ***acquisire il lock*** dell'oggetto
- il ***lock*** è automaticamente ***rilasciato*** quando il thread esce dalla sezione `synchronized`, o se viene interrotto da un'eccezione
- un thread che non riesce ad acquisire un *lock* ***rimane sospeso*** sulla richiesta della risorsa fino a che il *lock* non è disponibile

NOTA:

ad ogni oggetto contenente metodi o blocchi `synchronized` viene assegnata ***una sola variabile condizione***

➔ Due thread non possono accedere contemporaneamente a ***due sezioni `synchronized` diverse di uno stesso oggetto***

L'esistenza di ***una sola variabile condizione*** per ogni oggetto rende il modello Java ***meno espressivo*** di un *vero monitor*, che presuppone la possibilità di definire più sezioni critiche per uno stesso oggetto

Ogni oggetto Java (istanza di una sottoclasse qualsiasi della classe `Object`) fornisce i metodi di ***sincronizzazione***:

- **`wait()`**

  blocca l'esecuzione del thread invocante in attesa che un altro thread invochi i metodi `notify()` o `notifyAll()` per quell'oggetto. Il thread invocante deve essere in possesso del *lock* sull'oggetto; il suo blocco avviene dopo aver rilasciato il *lock*. Anche varianti con specifica di ***timeout***

- **`notify()`**

  risveglia un ***unico thread*** in attesa sul monitor dell'oggetto in questione. Se più thread sono in attesa, la scelta avviene in maniera ***arbitraria***, dipendente dall'implementazione della macchina virtuale Java. Il thread risvegliato compete con ogni altro thread, come di norma, per ottenere la risorsa protetta

- **`notifyAll()`**

  esattamente come `notify()`, ma risveglia ***tutti i thread*** in attesa per l'oggetto in questione. È necessario tutte le volte in cui ***più thread*** possono essere ***sospesi su differenti sezioni*** critiche dello stesso oggetto (***unica coda d'attesa***)

Esempio (*Produttori e Consumatori*):

```
public synchronized int get() {
   while (available == false)
      try {
         // attende un dato dai Produttori
         wait();
      } catch (InterruptedException e) {}
   }
   available = false;
   // notifica i produttori del consumo
   notifyAll();
…
}

public synchronized void put(int value) {
   while (available == true)
      try {
         // attende il consumo del dato
         wait();
      } catch (InterruptedException e) {}
   }
   …
   available = true;
   // notifica i consumatori della
   // produzione di un nuovo dato
   notifyAll();
}
```


Gruppi di thread

Obiettivo: raccogliere una ***molteplicità di thread*** all'interno di un ***solo gruppo*** per facilitare operazioni di ***gestione*** (ad es. sospendere/ interrompere/far ripartire l'esecuzione di un insieme di thread con un'unica invocazione)

JVM associa ***ogni thread*** ad ***un gruppo*** all'atto della ***creazione*** del thread
Questa associazione è ***permanente*** e non può essere modificata

```
ThreadGroup myThreadGroup =
    new ThreadGroup("Mio Gruppo");
Thread myThread =
    new Thread(myThreadGroup, "MioT");
```

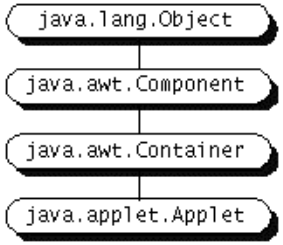
Scelta di ***default***:

stesso gruppo a cui appartiene il thread creatore (alla partenza di un'applicazione, la JVM crea un gruppo di thread chiamato ***"main"***)

Applet Java

Meccanismo per realizzare piccole applicazioni ***cliente*** in Java, capaci di essere scaricate dinamicamente su una ***connessione HTTP*** e di essere eseguite nella macchina virtuale messa a disposizione da un ***Web browser*** qualunque (Netscape, Internet Explorer, Appletviewer, …)

- Ogni applet sviluppata deve essere ***sottoclasse*** della classe **`Applet`**



- Viene inserita all'interno di un file HTML con:

```
<applet code=EsempioApplet.class
      width=xxx height=yyy>
<param name=ParamName value=Paramvalue>
     .....
<param name=ParamName value=ParamValue>
AlternateContent
</applet>
```


Metodi di Applet

Ogni applet ***eredita*** dalla classe **`Applet`** una serie di metodi che lo sviluppatore deve sovrascrivere:

- **`init()`** metodo invocato all'atto del ***caricamento*** (*loading/reloading*) da parte del browser. Si deve sovrascrivere qualora si debbano effettuare ***inizializzazioni*** particolari e non di pertinenza del metodo costruttore dell'applet (ad es. creare un certo numero di thread di animazione da avviare in seguito)

- **`start()/stop()`** metodo invocato quando l'applet ***entra/esce*** dalla zona di ***visualizzazione*** della finestra del browser (scorrimento, cambio di pagina). Questi metodi vanno sovrascritti quando si vogliono avviare/sospendere le attività dell'applet in funzione della sua visibilità sullo schermo (ad es. per avviare/arrestare i thread di animazione quando la finestra diventa visibile/nascosta)

- **`destroy()`** metodo invocato alla ***terminazione*** dell'applicazione browser che ha caricato l'applet. Va riscritto quando l'applet deve compiere alcune operazioni (***annullare*** gli effetti della fase di ***inizializzazione***) prima di essere eliminata dalla macchina virtuale del browser (ad es. distruggere i thread di animazione creati che, altrimenti, rimarrebbero sospesi per sempre)

Altri metodi di un'applet sono ereditati dalla classe **`Component`**:
(negli esempi, *g* rappresenta un'istanza della classe *Graphics*)

- ***`paint(g)`*** disegna un rettangolo grigio nell'area di visualizzazione assegnata
- ***`update(g)`*** cancella l'area di visualizzazione e richiama il metodo `paint(g)`

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

public class EsempioApplet extends Applet {
  public void paint(Graphics g) {
      Dimension r = size();
      g.setColor(Color.red);
      g.drawRect(10,10,r.width-20,
           r.height-20);
      System.out.println ("paint called");
      // dove finisce la stringa stampata??
     }
}
```


Passaggio dei Parametri

È possibile trasferire valori attuali per i ***parametri di invocazione*** dal file HTML in cui l'applet è contenuta al *runtime environment* dell'applet stessa:

- tag HTML <**`param name`**=xxx **`value`**=yyy>
- metodo di Java Applet **`getParameter()`**

Ad esempio:

```
…
public void init {
    String fontsize =
        getParameter("fontsize");
    if (fontsize=null) { fontsize=12; }
        else { fontsize =
               Integer.parseInt(fontsize);
             }
}
…
```

```
<applet code=ResizeEsempioApplet.class
        height=200 width=400>
<param name=fontsize value=16>
Esempio di trasferimento del valore di un
parametro da HTML a classe Java all'atto
dell'invocazione
</applet>
```

Java applet package

Mette a disposizione una ***API*** (*Application Program Interface*) che fornisce una ***serie di funzionalità ereditate*** automaticamente da ogni applet.

Ad esempio, ogni applet:

- può facilmente mettere in esecuzione ***file audio***
  `getAudioClip()`, `play()`, `stop()`, `loop()`
- può interagire con il Web browser ospitante causando il ***caricamento di nuovi documenti HTML*** `showDocument(URL)`
- può invocare ***metodi*** ***`public`*** di altre applet presenti nella stessa pagina Web
  `getAppletContext()`, `getApplet(Name)`, `getApplets()`
- può ***scrivere*** brevi stringhe sulla barra di ***stato*** del browser ospitante
  `showStatus(String)`

Esempio di Applet

Riepilogando, per realizzare una propria applet, occorre:

1) scrivere il ***sorgente*** della classe corrispondente (`EsempioApplet.java`)

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

public class EsempioApplet extends Applet {
    public void paint (Graphics g) {
        g.drawString("Hello world!",25,25);
     }
}
```

2) ***Compilare*** `EsempioApplet.java` con ***javac*** ottenendo `EsempioApplet.class`

3) ***Scrivere*** un file ***HTML*** in cui l'applet realizzata viene inserito (vedi lucido precedente)

4) Utilizzare un ***Web browser*** qualunque per visualizzare il file HTML

Appletviewer (SUN)

"***Minibrowser***" HTML fornito dalla SUN per la visualizzazione di applet. Mette a disposizione un ***ambiente di esecuzione*** per ***applet*** del tutto analogo a quello fornito da più complessi browser Web.

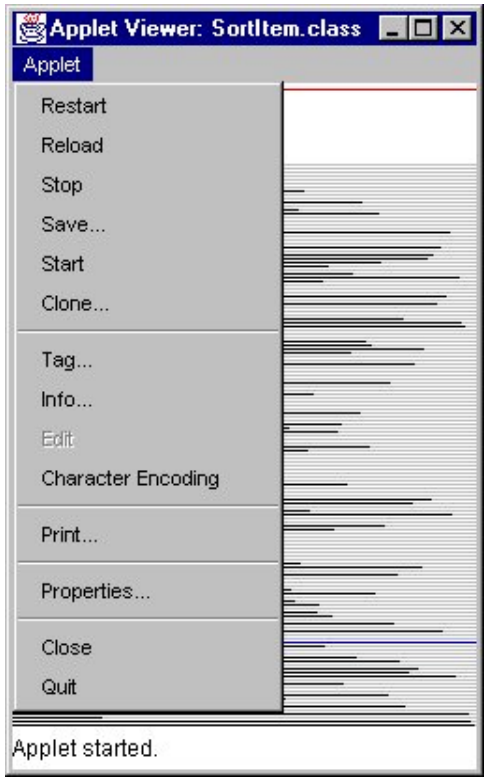
- Per mettere in esecuzione ***appletviewer*** nell'ambiente shell, è sufficiente digitare
  `appletviewer [-debug]  URL/file .....`

- I ***tag HTML diversi*** da `<applet ...>` presenti nei vari URL/file vengono ignorati

- Per ciascun tag `<applet ...>` nell'URL/file viene generata una ***finestra di visualizzazione*** corredata da un menu di utilità ***Applet***

- Ogni applet esegue le proprie ***operazioni di I/O*** nella finestra di competenza

Appletviewer (SUN)



- Lo sviluppatore può ***controllare l'esecuzione*** dell'applet selezionando funzioni messe a disposizione dal menu ***Applet***

Applet: Restrizioni di Sicurezza

Ogni Web browser mette in atto ***politiche di sicurezza*** per evitare che applet caricate possano compromettere il sistema.

L'implementazione delle politiche di sicurezza è una ***scelta progettuale*** del browser
➔ le restrizioni di sicurezza possono essere ***differenziate*** a seconda del browser utilizzato, ma anche dell'ambiente di utilizzo (*trusted/untrusted environment*)

Evoluzione del ***modello di sicurezza*** di Java:

JDK 1.0.x ***sandbox***

JDK 1.1.x possibilità di avere ***applet firmate***
firmate -> stessi diritti delle applicazioni
non firmate -> sandbox

JDK 1.2.x ***superamento dell'accesso on/off*** precedente
applet firmate sottostanno alla politica del `SecurityManager` di sistema, come tutte le applicazioni Java in esecuzione
granularità fine nella gestione dei permessi

Per altre informazioni su ***applet e sicurezza***:
`http://java.sun.com/sfaq/`

I browser oggi più diffusi applicano le seguenti restrizioni sulle ***applet non firmate*** caricate dalla rete:

- NON si possono ***caricare librerie*** o definire ***metodi nativi***
- generalmente, NON è possibile ***leggere o scrivere file*** sulla macchina che ospita il browser, né farvi ***partire l'esecuzione*** di programmi
- NO ***connessioni di rete***, se non con la macchina da cui il codice dell'applet proviene
- NO lettura di ***proprietà di sistema***
- le ***finestre di visualizzazione*** di un applet hanno un ***aspetto differente*** da quelle utilizzate da un'applicazione

Applet caricate dal ***file system locale*** (da un direttorio nel `CLASSPATH` dell'utente) non sono soggette alle restrizioni precedenti

Ogni browser implementa un proprio oggetto **`SecurityManager`** che mette in atto le ***politiche di sicurezza*** richieste
L'oggetto `SecurityManager`, una volta istanziato dal Web browser, ***non può più venire rimosso***

Quando la richiesta di una applet (ad es. invocazione di un metodo) è ***incompatibile*** con la politica corrente, il `SecurityManager` lancia ***un'eccezione*** (`SecurityException`), che l'applet può catturare e gestire

Il Class Loader (1)

Il `Class Loader` di Java opera quella serie di operazioni che vanno dal recupero dei dati di una classe alla restituzione dell'istanza di `Class`.

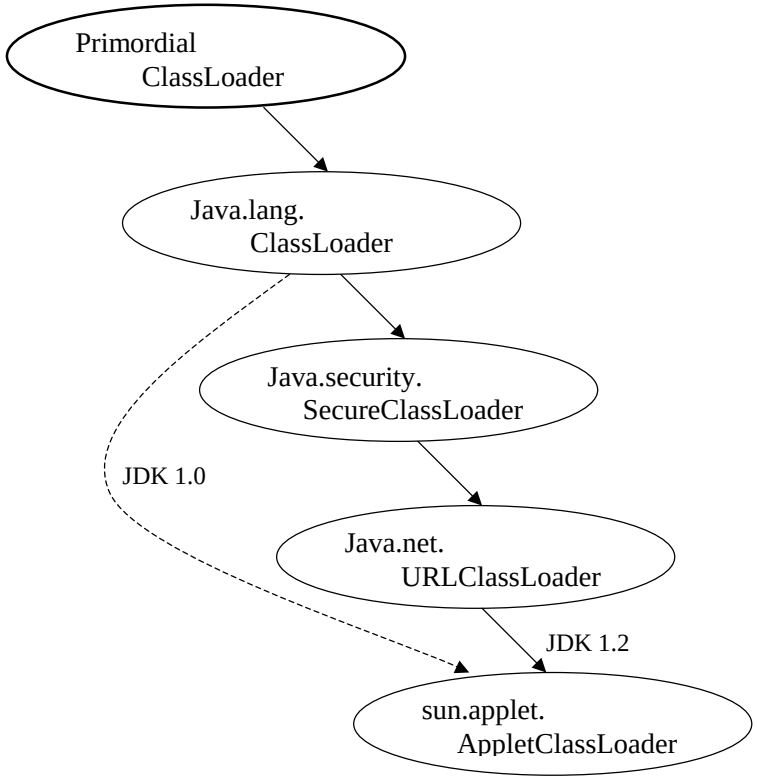
Coordinandosi con il `Security Manager` crea **spazi di nomi separati**, necessari per implementare le politiche di sicurezza.

Tipi di Class Loader:

- *class loader interno*; in JDK 1.2 usato solo per caricare le classi di Java;
- *secure class loader*; in JDK 1.2 permette di implementare politiche di sicurezza più granulari;
- *URL class loader*; permette di caricare classi da un insieme di URL distinte
- *Class loader specifici* definiti dalle applicazioni applet (es. ogni browser implementa un proprio applet class loader);

Il Class Loader (2)

I programmatori possono definire class loader propri purchè tutti estendano i tipi predefiniti:



Il Class Loader (3)

Es. **Appletviewer**
Usa una classe privata definita dalla Sun (sun.applet.AppletClassLoader) per caricare le applet

Es. **Applicazione utente**
Un utente può creare un proprio class loader di rete per scaricare classi da un server:

```
ClassLoader loader = new NetworkClassLoader(host, port);
Object main =
           loader.loadClass("Main", true).newInstance();
              . . .
```

Il class loader di rete deve definire i metodi findClass() e loadClassData() per caricare una classe dalla rete, dopodiché può usare il defineClass() per creare un'istanza della classe. Un esempio di implementazione potrebbe essere:

```
class NetworkClassLoader extends ClassLoader {
   String host;
   int port;

   public Class findClass(String name) {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
   }

   private byte[] loadClassData(String name) {
         // load the class data from the connection
                  . . .
   }
}
```