10
Modificatori

un modificatore la definizione ed il comportamento di classi, metodi e variabili

10.1 Controllo dell'accesso a metodi e campi

Incapsulazione:
nascondere le parti interne di un oggetto,
permettendo l'accesso all'oggetto solo attraverso le interfacce definite

Un nome viene cercato in tutti i file sorgente possibile

Package: gruppo di classi collegate

Protezione package: metodi e campi sono accessibili alle classi nello stesso package

Le classi non insertite esplicitamente in un package vengono inderite nello stesso package di default

private: metodi e campi privati possono essere utilizzati o richiamati solo all'interno della stessa classe;

public: metodi e campi pubblici possono essere utilizzati o richiamati anche all'esterno della classe o del package;

: protezione a livello di package

protected: metodi e campi protetti possono essere utilizzati da tutte le classi del package e dalle sottoclassi anche fuori dal package; (*)

non è possibile definire due classi pubbliche nello stesso file

Visibilità	public	protected		private
nella stessa classe	si	si	si	si
da una classe nello stesso package	si	si	si	no
da una classe fuori dal package	si	no (si solo nelle sottoclassi)*	no	no

nelle sottoclassi:

un metodo pubblico rimane pubblico
un metodo protetto rimane protetto o può diventare pubblico
un metodo privato non viene visto
un metodo senza protezione può diventare privato

In generale si dichiarano i campi privati e si manipolano tramite metodi di accesso (pubblici) (setProprieta (tipo valore); getProprieta();)

Se eredito una proprietà privata, lei c'è, ma non la vedo. La posso manipolare utilizzando i metodi accesso setProprieta (tipo valore); getProprieta();

Nel momento in cui ho definito le intestazioni dei metodi pubblici per la manipolazione di un oggetto, ne posso modificare l'implementazione senza che questo influenzi il resto del programma

10.2 Metodi e campi di classe

static: definisce campi e metodi di classe, non specifici ad ogni singolo oggetto, ma unici per tutta la classe

sono acceduti indicando il nome di un oggetto o della classe

Esempio

```
public class  CountInstances {
   private static int   numInstances = 0;

   protected static int getNumInstances() {return numInstances;}

   private static void  addInstance() { numInstances++;}

   CountInstances() {CountInstances.addInstance();}

   public static void  main(String args[]) {
      System.out.println("Starting with " +
        CountInstances.getNumInstances() + " instances");
      for (int  i = 0;  i < 10;  ++i)
         new CountInstances();
      System.out.println("Created " +
                CountInstances.getNumInstances() + " instances");
   }
}
```

10.3 Finalizzazione

final: non si possono creare sottoclassi di classi finali
una variabile finale è una costante
un metodo finale non può essere reimplementato nelle sottoclassi

l'uso di classi e metodi finali aumenta l'efficienza del codice: un metodo finale non è riscritto nelle sottoclassi e quindi ad esso non si applica il polimorfismo sui metodi.

10.5 Altri modificatori

syncronized, volatile: usati nei thread

native: usato nei metodi nativi

11
Package, interfacce e classi interne

11.1 Package

Un package permette di raggruppare insiemi di classi

Permettono di organizzare classi in unità,
di ridurre conflitti fra i nomi,
di proteggere classi, campi e metodi e
di identificare le classi con il proprio nome.

Un package è una collezione di classi, ma può contenere anche altri package:
java, java.awt, java.awt.Image, java.awt.event.*;

riferimento alle classi di un package:

- le classi di java.lang o del packge di default vengono indicate dal loro nome
- si può importare un package o una classe
  import java.util.Vector
  import java.awt.*
  vengono importate le classi pubbliche effettivamente utilizzate
  non vengono importati i sottopackage
- nomepackage.nomeclasse

se diversi package importati definiscono lo stesso nome si deve usare la specifica completa nomepackage.nome

Definizione package

- si sceglie il nome del package: it.unipg.dipmat.liste
- si definisce una struttora di directory corrispondente: it\unipg\dipmat\liste
- si inserisce all'inizio del file di definizione delle classi:
  package it.unipg.dipmat.liste

CLASSPATH specifica dove iniziare a cercare le directory dei package

Esempio

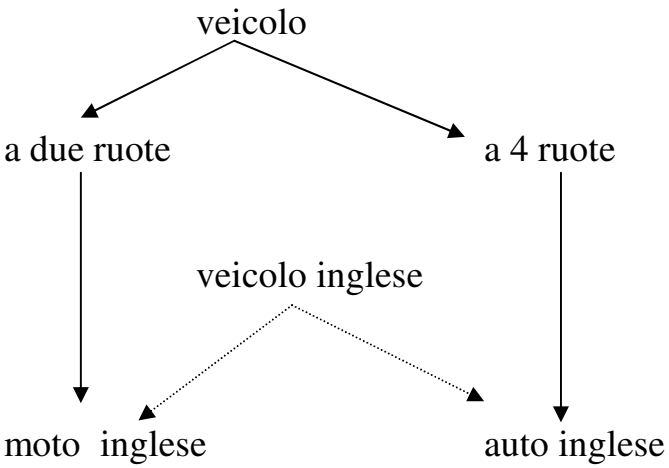
```
package  collections;

import  java.util.Enumeration;  // added

public class  LinkedList {
   private Node  root;
   public Enumeration  enumerate() {return new LinkedListEnumerator(root);}
}

class  Node {
   private Object  contents;
   private Node    next;

   Object  contents() { return contents;}
   Node    next() { return next;}
}

class  LinkedListEnumerator implements Enumeration {
   private Node  currentNode;

   LinkedListEnumerator(Node  root) { currentNode = root;}

   public boolean  hasMoreElements() { return currentNode != null;}

   public Object   nextElement() {
      Object  anObject = currentNode.contents();
      currentNode = currentNode.next();
      return  anObject;
   }
}
```

```
================================================
//  USO
class LinkedListTester {
        public static void main(String argv[]) {

        collections.LinkedList  aLinkedList = null;  /* createLinkedList(); */

        java.util.Enumeration   e = aLinkedList.enumerate();

        while (e.hasMoreElements()) {
                Object  anObject = e.nextElement();
                // do something useful with anObject
                }
        }
}
```

11.2 Interfacce

Un'interfaccia fornisce un modello di comportamento che altre classi devono implementare

Servono a superare i vincoli dell'ereditarietà singola



Un'interfaccia è costituita da un insieme di definizioni di metodi astratti (intestazioni senza implementazioni)

Si costituisce una gerarchia di interfacce distinta dalla gerarchia delle classi

Uso di interfacce

```
public class Neko extends java.applet.Applet implements Runnable {

public class Neko extends java.applet.Applet
        implements Runnable, MouseListener, Sortable, Observable {
```

eventuali sottoclassi ereditano le interfacce implementate

Creazione di interfacce

public interface Growable {. . . }

l'interfaccia può contenere definizioni di metodi (che vengono considerati pubblici e astratti) e campi (che vengono considerati pubblici e finali)

interfacce non pubbliche non convertono automaticamente i metodi in pubblici e astratti e le costanti in pubbliche

un'interfaccia non pubblica può essere utilizzata da classi e interfacce nello stesso package

Esempio

```
public interface Fruitlike
{void profuma(); void marcisce(); }
==================================================
public interface Spherelike
{void rimbalza(); void ruota(); }
==================================================
class Fruit implements Fruitlike
{. . . }

class Orange extends Fruit implements Spherelike
{. . . }
```

E' possibile dichiarare variabili di tipo interfaccia; in questo caso il valore della variabile sarà quello di un oggetto che implementa l'interfaccia:

```
Runnable aro = new MiaAnimazioneClass();
```

E' possibile fare casting di oggetti su interfacce

```
Orange unArancio = new Orange();
Fruit unFrutto =  unArancio;
Fruitlike unFruttolike =  unArancio;
Spherelike unaSpherelike =  unArancio;
Orange secondoarancio = (Orange) unFruttolike;

unFrutto.marcisce();        unFruttolike.profuma();   unSpherelike.rimbalza();
```

unFruttolike.rimbalza(); //errore in compilazione

((Orange) unFruttolike).rimbalza();

unArancio.marcisce(); unArancio.profuma(); unArancio.rimbalza();

si può specificare un parametro generico per il metodo di un'interfaccia indicando un parametro di tipo interfaccia:

```
public interface Fruitlike
{void cresce(Fruitlike self); . . . }

class Orange extends Fruit implements Spherelike
{
void cresce(Fruitlike self);
        {
        Orange o = (Orange) self;
        . . .
. . .  }
}
```

Estensione di interfacce

Le interfacce possono essere organizzate in gerarchie

public interface Fruitlike extends Foodlike{. . . }

non esiste un'interfaccia radice della gerarchia

E' ammessa l'ereditarietà multipla:

public interface Fruitlike extends Foodlike, Growable, Vegetable{. . . }

in questo caso i metodi devono differire per nome o per numero e tipo di parametri (non conta la differenza del tipo di ritorno)

11.3 Classi interne

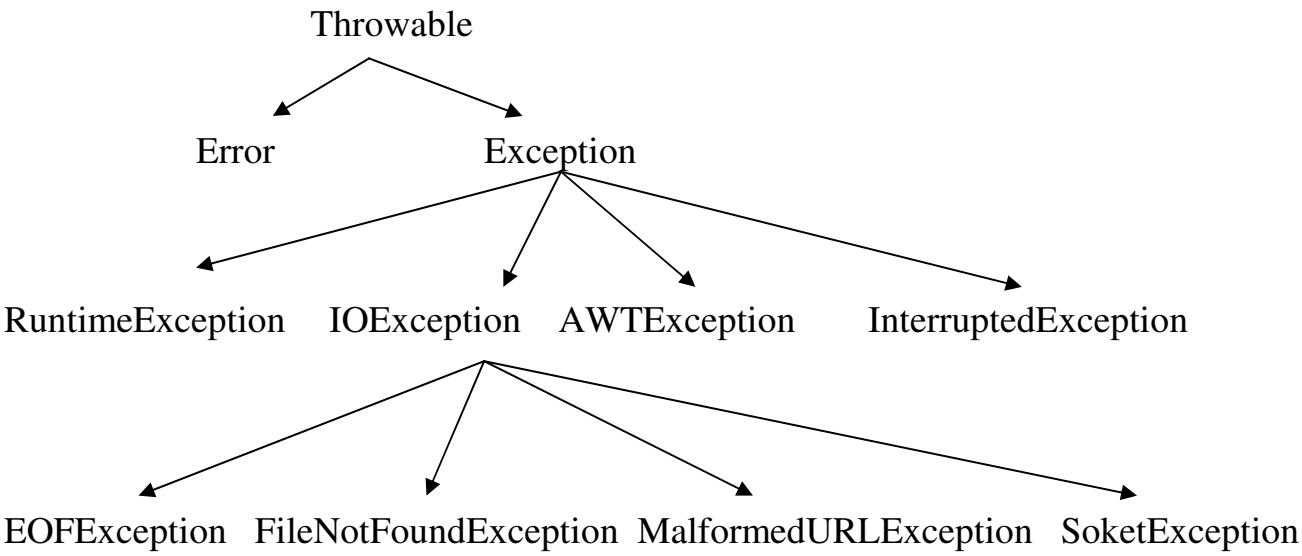
E' possibile definire una classe all'interno di un'altra classe, in un blocco o anonimamente

12
Eccezioni

eccezione: accadimento anomalo (o inaspettato) durante l'esecuzione del programma

le eccezioni vengono sollevate, in risposta ad un evento inusuale;
è possibile sollevare esplicitamente una eccezione

una eccezione è un oggetto di una classe che discende dalla classe Throwable



12.1 Protezione dalle eccezioni

```
try {
//codice da proteggere
. . .
}
catch (EccezioneTipo1 e1) { . . .}
catch (EccezioneTipo2 e2) {. . . }
catch (EccezioneTipo3 e3) {. . . }
finally  { . . . }
```

Se un metodo solleva una eccezione l'esecuzione del blocco try viene interrotta e si passa ad eseguire il primo catch il cui parametro è compatibile con l'eccezione sollevata. Poi si esegue il blocco finally, se è presente. L'esecuzione prosegue con la prima istruzione che segue il try catch finally

Non è obbligatorio gestire con un try catch i metodi che possono sollevare eccezioni che discendono da Error o da RuntimeException

Esempi

```
try { Thread.sleep(1000);}
catch (InterruptedException e) {}
```

```
try { Thread.sleep(1000);}
catch (InterruptedException e) {System.out.println("Errore"+e.getMessage());}
```

```
try {
        char mybuffer = new mybuffer[1024];
        numbytes = 0;
        FileInputStream s = new FileInputStream("pippo.txt");

        while (numbytes < mybuffer.length){
                mybuffer [numbytes]= s.read();
                numbytes++;}
}
catch (IOException e) { System.out.println("IOException; bytes letti: "+numbytes);}
```

I blocchi catch vanno messi in ordine, da quello che riguarda l'eccezione più specifica a quello che riguarda l'eccezione più generica; i catch vengono esaminati in ordine: il primo catch il cui parametro è compatibile con l'eccezione che è stata generata cattura l'eccezione; se sbaglio l'ordine il compilatore se ne accorge.

```
try {
. . .
}
catch (NullPointerException p) { . . .}        // in realtà questo non si dovrebbe fare!
catch (RuntimeException r) {. . . }             //in realtà questo non si dovrebbe fare!
catch (IOException m) {. . . }
catch (Exception e) {. . . }
```

```
catch (Throwable t) {. . . }
```

```
try {
        char mybuffer = new mybuffer[1024];
        int numbytes = 0;
        FileInputStream s = new FileInputStream("pippo.txt");

        while (numbytes <= mybuffer.length){
                mybuffer [numbytes]= s.read();
                numbytes++;}
}
catch (FileNotFoundException e) { System.out.println("File non trovato";}
catch (IOException e) { System.out.println("IOException; bytes letti: "+numbytes);}
catch (Exception e) { e.printStackTrace();}
```

Un blocco try catch si può trovare anche all'interno di un blocco finally

```
        FileInputStream s = null;
        try {
                s = new FileInputStream("pippo.txt");
                . . .
                }
        catch (IOException e) { . . .}
        finally  {
                if (s != null)
                try { s.close();}
                catch (IOException e) { }
        }
```

finally può essere usato anche dopo return, break o continue (si usa try e finally): il codice contenuto del blocco finally viene comunque eseguito dopo l'interruzione

Esempio

```
public class  MyFinalExceptionalClass /* extends ContextClass */ {
   public static void  main(String argv[]) {
      int  mysteriousState = Integer.parseInt(argv[0]); /* getContext(); */

      while (true) {
         System.out.print("Who ");
         try {
            System.out.print("is ");
            if (mysteriousState == 1) return;
            System.out.print("that ");
            if (mysteriousState == 2)  break;
            System.out.print("strange ");
            if (mysteriousState == 3) continue;
            System.out.print("but kindly ");
            if (mysteriousState == 4) throw new Error(); /* UncaughtException(); */
            System.out.print("not at all ");
         } finally { System.out.print("amusing ");}
         System.out.print("yet compelling ");
      }
      System.out.print("man?");
   }
}
```

12.2 Dichiarazione di metodi che possono sollevare eccezioni

public boolean mioMetodo (int x, int y) throws UnaEccezione { . . .}

public boolean metodo2 (int x) throws Eccezione1, Eccezione2, Eccezione3 { . . .}

public void metodo3 () throws IOException { . . .}

```
public void leggiFile (String nomefile) throws IOException
{
        char mybuffer = new mybuffer[1024];
        int numbytes = 0;
        FileInputStream s = new FileInputStream(nomefile);

        while (numbytes < mybuffer.length){
                mybuffer [numbytes]= s.read();
                numbytes++;}
…

}
```

in questo caso non è necessario inserire l'istruzione read in un blocco try catch: IOException viene passata al metodi chiamante, che la deve gestire

Quando un metodo viene ridefinito è possibile sollevare meno eccezioni del metodo originale, ma non di più

12.3 Creazione e sollevamento di eccezioni

eccezioni sono classi ed è possibile derivare nuove eccezioni

```
public class SunSpotException extends Exception{
        public SunSpotException(){ }
        public SunSpotException(String msg){super(msg); }
         }
```

le eccezioni vengono sollevate dall'istruzione throw:

throw new ServiceNotAvailableException();

throw new SunSpotException("Errore: comunicazione disturbata");

quando una eccezione viene sollevata il metodo esce immediatamente (ma se c'è una clausola finally il suo codice viene eseguito) e nessun valore viene restituito

se il metodo chiamante non ha il blocco try catch a proteggere la chiamata il programma potrebbe terminare

E' possibile sollevare un'eccezione all'interno di una blocco catch

Esempio

```
public class MyFirstException extends Exception {

  public MyFirstException(String msg) {super(msg);}
}

   public void  anExceptionalMethod() throws MyFirstException {
//…
       if (true /* someThingUnusualHasHappened() */) {
          throw new MyFirstException("Evento anomalo");
          // execution never reaches here
       }
//  …
 }


public void  responsibleExceptionalMethod() throws MyFirstException {
        try {
                anExceptionalMethod();
            }
        catch (MyFirstException m) {
                // do something responsible
                throw m;     // re-throw the exception
                }
}
```

Esempio

Definiamo una lista di interi ed un metodo che restituisce il valore massimo;
se la lista è vuota viene sollevata una eccezione;

```
class ListaVuota extends Exception{
        public ListaVuota(){
                super("La lista è vuota");
        }
}

class Elemento {
        int valore;
        Elemento next;}

class Lista{
        Elemento testa=null;

        Lista(int n){
        // …il costruttore produce una lista di n elementi casuali
        }

        public int massimo () throws ListaVuota {
                if (testa == null)
                        throw new ListaVuota();
                else {
                        int m = testa.valore;
                        for (Elemento e = testa.next; e != null; e = e.next)
                                if (e.valore > m)
                                        m = e.valore;
                        return m;
                       }
        }}

        //Nel main() il metodo può usare così:
public class ProvaEccezioni{
        public static void main(String args []){
                Lista lista = new Lista(100);
                try{
                        int m = lista.massimo();
                        System.out.println("Massimo = "+m);
                 }
                catch(ListaVuota e) { System.out.println(e.getMessage());}     }}
```

13
Multithreading

Processo: esecuzione di un programma all'interno del suo proprio spazio di indirizzi

Thread: singolo flusso di esecuzione all'interno di un processo

I Thread sono usati in particolare per effettuare più attività contemporaneamente

13.1 Creazione e uso di thread

- derivare dalla classe Thread
- implementare l'interfaccia Runnable

```
public class unThread extends Thread{
        public void run(){. . . }
        }

public static void main (String args []){

unThread t1 = new unThread();

t1.start();
…
}

t1.suspend();        t1.resume();        t1.interrupt();
```

==========

```
public class unAltroThread extends unaClasse implements Runnable{
        public void run(){. . . }
        }

public static void main (String args []){

unAltroThread aut = new unAltroThread();
Thread t1 = new Thread(aut);

t1.start();
…

}
t1.suspend();        t1.resume();        t1.interrupt();



new Tread(new unAltroThread()).start();
```

Esempio

```
 class SimpleRunnable implements Runnable {
   public void run() {
      System.out.println("in thread named '"+Thread.currentThread().getName() + "'");
   }
}

public class ThreadTester {
   public static void main(String argv[]) {
      SimpleRunnable aSR = new SimpleRunnable();

      while (true) {
         Thread t = new Thread(aSR);
         System.out.println("new Thread() " + (t == null ? "fail" : "succeed") + "ed.");
         t.start();
         try { t.join(); } catch (InterruptedException ignored) { }
                  // waits for thread to finish its run() method

      }
   }
}
```

```
new Thread() succeeded.
in thread named 'Thread-1'
new Thread() succeeded.
in thread named 'Thread-2'
new Thread() succeeded.
in thread named 'Thread-3'
new Thread() succeeded.
in thread named 'Thread-4'
new Thread() succeeded.
in thread named 'Thread-5'
new Thread() succeeded.
in thread named 'Thread-6'
new Thread() succeeded.
in thread named 'Thread-7'
new Thread() succeeded.
in thread named 'Thread-8'

//senza join
 new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
in thread named 'Thread-1'
new Thread() succeeded.
new Thread() succeeded.
in thread named 'Thread-2'
in thread named 'Thread-7'
in thread named 'Thread-3'
in thread named 'Thread-4'
in thread named 'Thread-5'
new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
new Thread() succeeded.
in thread named 'Thread-8'
in thread named 'Thread-11'
```

```
class SimpleRunnable1 extends Thread{
   public void run() {
      System.out.println("in thread named '"
                     + Thread.currentThread().getName() + "'");

   }
}

public class ThreadTester1 {
   public static void main(String argv[]) {

      while (true) {
       SimpleRunnable1 aSR = new SimpleRunnable1();
       System.out.println("new Thread() " + (aSR == null ?
                                  "fail" : "succeed") + "ed.");
      aSR.start();
      }
   }
}
```

```
/*
 Vediamo all'opera la concorrenza:
 *crea 10 thread
 *ogni theard si presenta in un ciclo infinito
 */


class SimpleRunnable2 extends Thread{
   public void run() {
      while (true){
         System.out.println("in thread named '"
                     + Thread.currentThread().getName() + "'");
//       yield();

     }
   }
}

public class ThreadTester2 {
   public static void main(String argv[]) {

      for (int i = 0; i < 10; i++) {
       SimpleRunnable2 aSR = new SimpleRunnable2();
       System.out.println("new Thread() " + (aSR == null ?
                                   "fail" : "succeed") + "ed.");
      aSR.start();

  //     try { aSR.join(); } catch (InterruptedException ignored) { }

      }
      while (true){System.out.println("Io sono il main"); }
   }
}
```

```
Thread t = new Thread( nomethread); //è possibile dare un nome a un thread


 Thread t = new Thread(aSR, nomethread); //è possibile dare un nome a un thread
```

Quando un thread muore viene sollevato un errore di classe ThreadDeath

```
public class test  {

public static void main(String argv[]) {
      Thread t = new Thread(new SimpleRunnable());

         try {
         t.start();
         metodoChePuoStoppareT(t);
         } catch (ThreadDeath td) {
               . . .     //operazioni di ripulitura
               throw td; //si lascia morire il thread
               }
     }
  }
```

13.2 Canali di comunicazione (pipe)

un thead produttore genera un flusso di dati
un thead consumatore legge ed elabora il flusso di dati

Esempio
produttore → filtro → consumatore

```
/* @version 1.20 1999-04-23 @author Cay Horstmann*/

import java.util.*;          import java.io.*;

public class PipeTest
{  public static void main(String args[])
   {  try
      {  /* set up pipes */
         PipedOutputStream pout1 = new PipedOutputStream();
         PipedInputStream pin1 = new PipedInputStream(pout1);

         PipedOutputStream pout2 = new PipedOutputStream();
         PipedInputStream pin2 = new PipedInputStream(pout2);

         /* construct threads */
         Producer prod = new Producer(pout1);
         Filter filt = new Filter(pin1, pout2);


         /* start threads */ Consumer cons = new Consumer(pin2);
          Thread t = new Thread(cons);
         prod.start();       filt.start();       t.start();
      }
      catch (IOException e){}
   }
}
```

```
class Producer extends Thread
{
  private DataOutputStream out;
  private Random rand = new Random();

public Producer(OutputStream os)
  {  out = new DataOutputStream(os);}


  public void run()
  {  while (true)
    {  try
      {  double num = rand.nextDouble();
        out.writeDouble(num);
        out.flush();
        sleep(Math.abs(rand.nextInt() % 1000));
      }
      catch(Exception e)
      {  System.out.println("Error: " + e);}
    }
  }
}
```

```
class Filter extends Thread
{  private DataInputStream in;
   private DataOutputStream out;

public Filter(InputStream is, OutputStream os)
   {  in = new DataInputStream(is);
      out = new DataOutputStream(os);
   }

   public void run()
   {
         double total = 0;
         int count = 0;

         for (;;)
      {  try
         {  double x = in.readDouble();
            total += x;
            count++;
            out.writeDouble(total / count);
          out.flush();
         }
        catch(IOException e)
         {  System.out.println("Error: " + e);}
      }
   }
}
```

```
class Consumer implements Runnable
{       private DataInputStream in;

public Consumer(InputStream is)
  {   in = new DataInputStream(is);}

  public void run()
  {
        double old_avg = 0;

        for(;;)
     {  try
       {  double avg = in.readDouble();
          if (Math.abs(avg - old_avg) > 0.01)
          {  System.out.println("Current average is " + avg);
             old_avg = avg;
          }
       }
       catch(IOException e)
       {  System.out.println("Error: " + e);}
     }
  }
}
```

13.3 Scheduling

- preemptivo (con prelazione)
- non preemptivo (senza prevaricazione)

yield(): cede il passo ad un altro thread
t.join(): si aspetta che il thread t abbia terminato

setPriority(int priorità) int getPriority()
Thread.MIN_PRIORITY Thread.NORM_PRIORITY Thread.MAX_PRIORITY

Esempi

```
public class ComplexThread1 extends Thread { //verifica il tipo di scheduling

    ComplexThread1(String name) {super(name);}

   public void run() {
      while (true) {System.out.println(Thread.currentThread().getName());}
   }

   public static void main(String argv[]) {
      new ComplexThread1("one potato").start();
      new ComplexThread1("two potato").start();
   }}
```

==========

```
public class ComplexThread2 extends Thread {//i due scrivono in modo alternato
   ComplexThread2(String name) {super(name);}

   public void run() {
      while (true) {
         System.out.println(Thread.currentThread().getName());
         Thread.currentThread().yield(); //si può anche scrivere Thread.yield();
      }
   }

   public static void main(String argv[]) {
      new ComplexThread2("one potato").start();
     new ComplexThread2("two potato").start();
   }
}
```

```
=====================================================

public class ComplexThread2bis extends Thread { //priorità
   ComplexThread2bis(String name) {super(name);}

   public void run() {
      while (true) {
         System.out.println(Thread.currentThread().getName());
      }
   }

   public static void main(String argv[]) {
     Thread t1 = new ComplexThread2bis("one potato");
     Thread t2 = new ComplexThread2bis ("two potato");
     Thread t3 = new ComplexThread2bis ("three potato");
     t1.start();  t2.start();
        try
      { t1.join();}
         catch (InterruptedException e) {System.out.println("Thread interrotto");}
       t3.start();
   }
  }

=====================================================

public class ComplexThread3 extends Thread { //priorità
   ComplexThread3(String name) {super(name);}

   public void run() {
      while (true) {
         System.out.println(Thread.currentThread().getName());
      }
   }

   public static void main(String argv[]) {
     Thread t1 = new ComplexThread3("one potato");
     Thread t2 = new ComplexThread3("two potato");
     t2.setPriority(t1.getPriority() +1);
     t1.start();  t2.start();
   }
  }
=====================================================
```

```
public class ComplexThread3bis extends Thread { //priorità
   ComplexThread3bis(String name) {super(name);}

   public void run() {
      while (true) {
         System.out.println(Thread.currentThread().getName());
         Thread.currentThread().yield(); //si può anche scrivere Thread.yield();
      }
   }

   public static void main(String argv[]) {
     Thread t1 = new ComplexThread3bis("one potato");
     Thread t2 = new ComplexThread3bis("two potato");
     t2.setPriority(t1.getPriority() +1);
     t1.start();  t2.start();
   }
}
```

==========

```
public class ComplexThread4 extends Thread {//sleep
   private int delay;

   ComplexThread4(String name, float seconds) {
super(name);         delay = (int) seconds * 1000;     start();
}

   public void run() {
      while (true) {
         System.out.println(Thread.currentThread().getName());
         try {Thread.sleep(delay);}
               catch (InterruptedException e) {return;}
      }
   }

   public static void main(String argv[]) {
      new ComplexThread4("one potato",   1.1F);
      new ComplexThread4("two potato",   1.3F);
      new ComplexThread4("three potato", 0.5F);
      new ComplexThread4("four",         0.7F);
   }
}
```

13.4 Gruppi di thread

```
String nomegruppo = . . .;
ThreadGroup g = new ThreadGroup(nomegruppo);
ThreadGroup g1 = new ThreadGroup(g, “sottogruppodig”);


Thread t = new Thread(g,nomeThread); //il thread è aggiunto al gruppo
Thread t1 = new Thread(g,oggettorunnable,nomeThread); //il thread è aggiunto al
                                                                    //gruppo
```

public int activeCount()
restituisce una stime superiore del numero di thread del gruppo attivi

public int enumerate(Thread list[])
copia nell’array specificato i thread attivi

public final void interrupt()
public final void suspend()
public final void resume()
sono applicati a tutti i thread del gruppo

13.5 Sincronizzazione

Quando più thread accedono in modifica allo stesso dato è necessario usare la sincronizzazione per evitare il rischio di modifiche inconsistenti.
La sincronizzazione ha lo scopo di assicurare che un solo thread alla volta acceda al dato.

Esempio

```
class ContoCorrente {
   private double c = 0;

   public  void incrementa() { c=c+1;}
   public  void decrementa() { c=c-1;}
   public  double saldo() { return c;}
}


class Paperone extends Thread {
        ContoCorrente conto;

        Paperone(ContoCorrente conto){
                this.conto=conto;
                }

         public  void run() {
                for (int i=0; i < 20000000; i++)
                        contatore.incrementa();
   }
}

class Paperino extends Thread {
        ContoCorrente conto;

        Paperino(ContoCorrente contatore){
                this.conto=conto;
                }

        public  void run() {
                for (int i=0; i < 20000000; i++)
                        conto.decrementa();
                }
}
```

```
public class EsempioMancanzaSincronizzazione{

	public static void main (String aa []){
		ContoCorrente c = new ContoCorrente();


		Paperone paperone = new Paperone(c);
		Paperino paperino = new Paperino(c);


		paperone.start();
		paperino.start();

		try{
			paperone.join();
			paperino.join();
		}
		catch(InterruptedException e) {e.printStackTrace();}

		System.out.println(c.saldo());
	}
}
```

Il saldo finale visualizzato dovrebbe essere 0.0, ma non sempre è così.

Questo dipende dal fatto che non siamo garantiti di come le microoperazioni svolte dai thread sono svolte. Questo può provocare delle modifiche inconsistenti ai dati nel caso in cui più thread accedono in modifica allo stesso dato.

Un ambiente thread-safe consente ai thread di coesistere senza conflitti

L'uso della sincronizzazione tramite la parola chiave *synchronized* permette di gestire situazioni in cui più thread accedono in modifica allo stesso dato.

Quando un thread esegue l'istruzione:
oggetto.metodo();
e metodo è stato dichiarato *synchronized* avviene la seguente cosa:
1) il thread verifica se *oggetto* è libero; se sì lo blocca (lock) ed esegue il metodo; altrimenti viene posto in attesa;
2) quando il thread ha concluso l'esecuzione di *metodo()*, *oggetto* viene sbloccato (unlock). Se ci sono thread in attesa di eseguire *oggetto.metodo()* uno di loro viene riattivato

oggetto, su coi viene fatto il lock e l'unlock per la sincronizzazione, viene detto *monitor*

```
class ContoCorrenteCS {
   private double c = 0;

   synchronized public  void incrementa() {c=c+1;}
   public  synchronized void decrementa() {c=c-1;}
   public  double saldo() {return c;}
}
```

```
class PaperoneCS extends Thread {
        ContoCorrenteCS conto;

        PaperoneCS(ContoCorrenteCS contatore){
                this.conto=conto;
        }

   public  void run() {
        for (int i=0; i < 10000000; i++)
                conto.incrementa();
   }
}

class PaperinoCS extends Thread {
        ContoCorrenteCS conto;
        PaperinoCS(ContoCorrenteCS contatore){
                this.conto=conto;
        }

           public  void run() {
                for (int i=0; i < 10000000; i++)
                        conto.decrementa();
   }
}

public class EsempioConSincronizzazione{

        public static void main (String aa []){
                ContoCorrenteCS c = new ContoCorrenteCS();

                PaperoneCS paperone = new PaperoneCS(c);
                PaperinoCS paperino = new PaperinoCS(c);
                paperone.start();
                paperino.start();

        try{
                paperone.join();
                paperino.join();
        }
        catch(InterruptedException e) {e.printStackTrace();}
                System.out.println(c.saldo());
        }
}
```

Esempio

Classe che costituisce un contatore per i thread generati

```
public class ThreadCounter {
        private int numerothread=0;

        public void contami()      {numerothread = numerothread +1;}
        public int quanti()  {return numerothread ;}
        }
////////////////////////////////////////////////////////////////////////////////////////////////////

public class SafeThreadCounter {
        int numerothread;

        public syncronized void contami()      {numerothread += 1;}
        public void quanti()        {return numerothread ;}
        }
```

L'uso della sincronizzazione può evitare modifiche inconsistenti dei dati, ma può provocare altri problemi:
colli di bottiglia e deadlock

Vediamo un altro esempio

La classe Punto rappresenta un punto dello schermo. Abbiamo due proprietà *x* ed *y* che rappresentano la posizione iniziale e tre metodi.
Il metodo *sposta()* il punto lungo la diagonale *y=x* incrementando entrambe le coordinate di una unità.
Il metodo *visualizza()* stampa sullo schermo le coordinate del punto.
Il metodo *controlla()* stampa sullo schermo le coordinate del punto se queste sono diverse. In teoria questo non dovrebbe mai accadere, poiché inizialmente le coordinate sono uguali ed ad ogni spostamento entrambe le coordinate vengono incrementate di uno.

L'applicazione crea un oggetto punto e lo passa a 3 thread.
Il primo contiene un ciclo infinito che richiama il metodo sposta del punto.
Il secondo contiene un ciclo infinito che richiama il metodo che visualizza le coordinate del punto (questo thread non è realmente eseguito perché la chiamata di start() è commentata).
Il terzo thread contiene un ciclo infinito che richiama il metodo che controlla la correttezza delle coordinate del punto.

Siamo in una situazione in cui abbiamo più thread che possono accedere ad un dato (cioè l'oggetto *punto* di classe Punto) ed almeno un thread può modificare il dato.
In questa situazione è necessaria la **sincronizzazione**, per evitare modifiche inconsistenti dei dati. In questa prima versione la sincronizzazione non è stata fatta, per cui capita che il terzo thread (ed anche il secondo, se fosse eseguito) visualizzi dei punti non esistenti della traiettoria.
Infatti può capitare che mentre la visualizzazione venga eseguita mentre il primo thread ha modificato una coordinata del punto, ma ancora non ha modificato l'altra.

Questa prima versione nopn usa la sincronizzazione. Eseguendo il programma il terzo metodo visualizza punti inesistenti della traiettoria

// (cartella: **EsempioNonSincronizzato)**

```
class Punto {
        int x = -1000000000, y = -1000000000;

        void sposta() { x++; y++; }

        void visualizza() {System.out.println("x=" + x + " y=" + y); }

        void controlla() {
                if(x-y !=0)
                        System.out.println("x=" + x + " y=" + y);
        }
}
```

```
class Muovi extends Thread
{
	Punto punto;

	Muovi (Punto punto){this.punto=punto;}

	public void run()
	{
		while (true)
			punto.sposta();
	}
}

class Disegna extends Thread
{
	Punto punto;

	Disegna (Punto punto){this.punto=punto; }

	public void run()
	{
		while (true)
			punto.visualizza();

	}
}

class Verifica extends Thread
{
	Punto punto;

	Verifica (Punto punto){   this.punto=punto;}

	public void run()
	{
		while (true)
			punto.controlla();

	}

}
```

```
class EsempioThreadSincronizzazione {
        public static void main(String [] s)
                {

                        Punto punto;
                        punto = new Punto();
                        Sposta t1=new Sposta (punto);
                        Disegna t2=new Disegna (punto);
                        Verifica t3=new Verifica (punto);

                        t1.start();
//                      t2.start();
                        t3.start();


                }

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Nei due esempi successivi la sincronizzazione è ottenuta utilizzando la parola chiave **synchronized**.

Nella cartella **EsempioSincronizzatoBlocchi** si utilizza **synchronized** per evitare che i thread possano accedere in contemporanea al codice che accede al dato comune che essi possono modificare. In questo primo caso la sincronizzazione è applicata a blocchi di codice.

Il blocco sincronizzato viene preceduto dalla parola chiave *synchronized* e da un parametro, detto monitor che ha il ruolo di chiave di accesso al blocco di istruzioni. Quando un thread prova ad eseguire il blocco sincronizzato verifica se il monitor è libero. Se il monitor è libero, il thread lo blocca ed esegue il blocco di istruzioni; infine libera il monitor. Se invece il monitor è già occupato da un altro thread, il nostro thread viene messo in attesa. Quando il thread che eseguiva il blocco libera il monitor, uno dei thread in attesa blocca il monitor ed esegue il blocco di istruzioni.
Perché la sincronizzazione sia corretta, tutti i thread che devono lavorare in mutua esclusione perché accedono in modifica (oppure, come in questo esempio in modifica ed anche in lettura) devono usare come monitor lo stesso oggetto. In questo caso tutti 3 i thread usano come monitor l'oggetto *punto* dichiarato e costruito nel main() e che viene poi passato a ciascun thread nel costruttore dei thread.
Come accade in questo esempio, l'oggetto **naturalmente candidato** a fare da monitor è lo stesso oggetto che i thread potrebbero andare a modificare contemporaneamente (oppure, come in questo esempio modificare e leggere le proprietà dell'oggetto).

```
class Punto {
        int x = -1000000000, y = -1000000000;
        void sposta() { x++; y++; }
        void visualizza() {System.out.println("x=" + x + " y=" + y);}
        void controlla() {
                if(x-y !=0)
                        System.out.println("x=" + x + " y=" + y);
        }
}


class Sposta extends Thread
{
        Punto punto;

        Sposta (Punto punto){     this.punto=punto;}

        public void run()
        {
                while (true)
                synchronized(punto){punto.sposta();}
        }

}

class Visualizza extends Thread
{
        Punto punto;

        Visualizza (Punto punto){this.punto=punto;}

        public void run()
        {
                while (true)
                {
                        synchronized(punto){punto.visualizza();}
                }
        }

}
```

```
class Controlla extends Thread
{
	Punto punto;

	Controlla (Punto punto){this.punto=punto;}

	public void run()
	{
		while (true)
			synchronized(punto){punto.controlla();}
	}

}
class EsempioThreadSincronizzazione {
	public static void main(String [] s)
		{

			Punto punto;
			punto = new Punto();
			Sposta t1=new Sposta (punto);
			Visualizza t2=new Visualizza (punto);
			Controlla t3=new Controlla (punto);

			t1.start();
//			t2.start();
			t3.start();

		}

}
```

//

Il terzo esempio, contenuto nella cartella **EsempioSincronizzatoMetodi**, mostra un'altra forma sintattica dell'uso della parola chiave *synchronized*.

In questo caso la parola chiave *synchronized* precede l'intestazione del metodo. In questo caso il monitor che ha il ruolo di chiave di accesso al metodo è il parametro implicito *this*, cioè l'oggetto a cui il metodo è applicato.

In questo caso la sincronizzazione è corretta: quando il primo thread prova ad eseguire il metodo
punto.sposta() o quando il secondo thread prova ad eseguire il metodo punto.visualizza() oppure il terzo thread prova ad eseguire il metodo punto.controlla(), il thread verifica se ll monitor (cioè l'oggetto punto nell'ambiente del thread) è libero: se sì lo blocca, esegue il metodo ed infine rilascia il monitor. Altrimenti viene messo in attesa finché il thread che sta bloccando il monitor non lo rilascia.

Qui la sincronizzazione è corretta perché la proprietà *punto* nell'ambiante di ciascuno dei tre thread si riferisce allo stesso oggetto, il *punto* dichiarato e costruito nel main() che poi viene passato ai thread nel loro costruttore.

```
class Punto {
        int x = -1000000000, y = -1000000000;

        synchronized void sposta() { x++; y++; }

        synchronized void visualizza() {System.out.println("x=" + x + " y=" + y);}

        synchronized void controlla() {
                if(x-y !=0)
                        System.out.println("x=" + x + " y=" + y);
        }
}

class Sposta extends Thread
{
        Punto punto;

        Sposta (Punto punto){ this.punto=punto; }


        public void run()
        {
                while (true)
                        punto.sposta();
        }
}
```

```
class Visualizza extends Thread
{
	Punto punto;

	Visualizza (Punto punto){ this.punto=punto; }

	public void run()
	{
		while (true)
			punto.visualizza();

	}
}

class Controlla extends Thread
{
	Punto punto;

	Controlla (Punto punto){ this.punto=punto; }

	public void run()
	{
		while (true)
			punto.controlla();
	}
}

class EsempioThreadSincronizzazione {
	public static void main(String [] s)
		{

			Punto punto;
			punto = new Punto();
			Sposta t1=new Sposta (punto);
			Visualizza t2=new Visualizza (punto);
			Controlla t3=new Controlla (punto);

			t1.start();
//			t2.start();
			t3.start();
		}
}
```

//

Il quarto ed ultimo esempio, contenuto nella cartella **EsempioSincronizzazioneErrata**, mostra un modo errato di fare la sincronizzazione.

```
class Punto {
	int x = -1000000000, y = -1000000000;
	void sposta() { x++; y++; }
	void visualizza() {System.out.println("x=" + x + " y=" + y); }
	void controlla() {
		if(x-y !=0)
			System.out.println("x=" + x + " y=" + y);
	}
}

class Sposta extends Thread
{
	Punto punto;

	Sposta(Punto punto){ this.punto=punto; }

	synchronized public void run()
	{
		while (true)
			punto.sposta();
	}
}

class Visualizza extends Thread
{
	Punto punto;

	Visualizza (Punto punto){ this.punto=punto; }

	synchronized public void run()
	{
		while (true)
			punto.visualizza();

	}
}
```

```
class Controlla extends Thread
{
	Punto punto;

	Controlla (Punto punto){ this.punto=punto; }

	synchronized public void run()
	{
		while (true)
			punto.controlla();
	}
}

class EsempioThreadSincronizzazione {
	public static void main(String [] s)
		{

			Punto punto;
			punto = new Punto();
			Sposta t1=new Sposta (punto);
			Visualizza t2=new Visualizza (punto);
			Controlla t3=new Controlla (punto);

			t1.start();
//			t2.start();
			t3.start();

		}
}
```

In questo caso ad essere sincronizzati sono i metodi run() di ciascun thread. In questo caso l'obiettivo della sincronizzazione non è raggiunto. Infatti ogni thread usa un monitor diverso: se stesso. Quindi ogni thread blocca se stesso e va avanti. Quindi in questo modo l'obiettivo di impedire l'accesso simultaneo da parte dei thread all'oggetto *punto* del main() non è raggiunto.

Vediamo un altro modo errato di fare la sincronizzazione

```
//Questa classe rappresenta un punto che si muove secondo una certa traiettoria

public class Point {      //redefines class Point from package java.awt
        private float x, y;   //OK since we're in a different package here

    public float x() { return x;}        // needs no synchronization

    public float y() {return y;}        // ditto

    . . .    // methods to set and change x and y
        syncronized sposta(…..)
 }


// classi per visualizzare il (concretamente per stampare le coordinate del) punto

 public class QuasiSafePointPrinter {
    public void print(Point p) {
        float safeX, safeY;
        syncronized(this)   //equivale a dichiarare print sincronizzato
                {
                safeX = p.x();
                safeY = p.y();
                }
       System.out.println("The point's x is " + safeX + " and y is " + safeY + ".");
    }
 }
```

Il problema è che i thread utilizzano di nuovo monitor diversi e quindi non si sincronizzano

per proteggere campi di classe:

```
public class ThreadCounter extends Thread{
        private static int numerothread=0;

        Color colore;

        public syncronized void contami()       {numerothread += 1;}
        }
```

il lock (che viene fatto su una singola istanza this) non basta nel caso in cui ci siano ad esempio almeno due oggetti di classe ThreadCounter ed un thread lavora con il primo oggetto e l'altro thread lavora con il secondo oggetto

```
public class ThreadCounter {
        private static int numerothread;

        public void contami(){
                syncronized (getClass())
                        {numerothread += 1;}
                }
        }
```

il lock viene fatto sulla classe stessa

un thread sincronizzato può eseguire la chiamata del metodo wait()
quando un altro thread richiama il metodo notifyAll() risveglia tutti i metodi che erano stati sospesi

13.6 Il modificatore *volatile*

i thread possono farsi delle copie locali nel proprio ambiente di esecuzione di campi dell'oggetto thread; questo può provocare aggiornamenti incomprensibili;
per evitare questo si può dichiarare la variabile *volatile*

```
public class StoppableTask extends Thread {
  private volatile boolean pleaseStop=false;

  public void run() {
    while (!pleaseStop) {
      // do some stuff...
    }
  }

  public void tellMeToStop() {
    pleaseStop = true;
  }

}
```

Se pleaseStop non fosse dichiarata volatile il thread potrebbe leggerne il valore all'inizio dell'esecuzione di run() e mai più ricontrollarlo, non accorgendosi che il valore della variabile è cambiato a causa di un altro thread che ha richiamato il metodo tellMeToStop() e provocando così un loop infinito

14
Flussi di input e di output

14.1 Classi astratte per l'input

InputStream: flusso di byte

Reader: flusso di caratteri

```
InputStream s = …;
Reader r = …;

byte bbuffer = new byte [1000];
char cbuffer = new char [1000];
```

Metodi: (tutti i metodi possono sollevare IOException)

- read()

```
s.read ();                          r.read ();
s.read (bbuffer);                   r.read (cbuffer);
s.read (bbuffer,100,300);           r.read (cbuffer,100,300);
```

restituisce –1 se su è giunti alla fine del flusso

Esempio

```
byte b;        char c;        int risb, risc;

while ((risb=s.read()) != -1) {
        b = (byte) risb;
                . . . }  //raggiunta fine flusso

while (risc=r.read()) != -1) {
        c = (char) risc;
        . . . }  //raggiunta fine flusso
```

```
if ((s.read(bbuffer) != bbuffer.length) || (risc=r.read(cbuffer) != cbuffer.length))
        System.out.println("letto meno di quanto atteso");
```

skip(n)

```
if ((s.skip(500) != 500) || (r.skip(600) != 600))
        System.out.println("saltato meno di quanto atteso");
```

available()

restituisce, se possibile, il numero di byte rimasti da leggere

ready()

restituisce, se possibile, se ci sono altri caratteri da leggere

mark() e reset()

```
if (s.markSupported() && r.markSupported())  {
        …
        s.mark(600);                    r.mark(600);
        …
        s.reset();              r.reset();
        . . . }
```

close()

```
try
{. . . }
catch (IOException e) {. . . }
finally {
        if (r != null)
        try{ r.close();}
           catch( IOException e) {System.out.println("Errore chiusura flusso r");}
        if (s != null)
        try{s.close();}
           catch( IOException e) {System.out.println("Errore chiusura flusso s");}
```

```
}
```

14.2 Flussi concreti di input

FileInputStream e FileReader

Flussi collegati a file

FileInputStream s = new FileInputStream(nomefile);
FileReader r = new FileReader(new File(nomefile));

System.in: standard input System.out: standard output

InputStreamReader può incapsulare ogni InputStream generando un flusso di caratteri

InputStreamReader filecaratteri = new InputStreamReader (s);
InputStreamReader tastiera = new InputStreamReader (System.in);

Flussi da socket e connessioni URL

InputStreamReader in = new InputStreamReader(**socket.getInputStream()**));
InputStreamReader in = new InputStreamReader(**connessioneURL.getInputStream()**));

ByteArrayInputStream e CharArrayReader

byte bbuffer = new byte [1000];
char cbuffer = new char [1000];

InputStream s = new ByteArrayInputStream(bbufer);
Reader r = new CharArrayReader(cbuffer);

ByteArrayInputStream s = new ByteArrayInputStream(bbufer);
CharArrayReader r = new CharArrayReader(cbuffer);

InputStream s1 = new ByteArrayInputStream(bbufer,100,300);
Reader r1 = new CharArrayReader(cbuffer,200,800);

StringBufferInputStream e StringReader

Analogo al precedente, basato su stringhe

14.3 Classi astratte di filtri

FilterInputStream e FilterReader
I filtri contengono definizioni di metodi e servizi
I filtri possono essere annidati

14.4 Classi concrete di filtri

BufferedInputStream e BufferedReader

Effettuano un buffering dell'input, disaccoppiando comunicazione ed elaborazione

BufferedReader non è una sottoclasse di FilterReader; implementa il metodo readLine() (restituisce null alla fine del flusso)

```
BufferedInputStream s=new BufferedInputStream (new FileInputStream(nomefile));
BufferedReader r = new BufferedReader( new FileReader(nomefile));
```

DataInputStream

Consente l'input formattato;
utilizza (come anche RandomAccessFile) l'interfaccia DataInput.

Metodi:

```
boolean readBoolean()    throws IOException;
byte readByte()          throws IOException;
int readUnsignedByte()   throws IOException;
short readShort()        throws IOException;
int readUnsignedShort()  throws IOException;
char readChar()          throws IOException;
int readInt()            throws IOException;
long readLong()          throws IOException;
float readFloat()        throws IOException;
double readDouble()      throws IOException;
String readLine()        throws IOException;   //per caratteri ASCII
String readUFT()         throws IOException;   //per caratteri Unicode UFT-8
```

readLine() restituisce null alla fine del flusso
readUFT() solleva UFTDataFormatException
gli altri metodi sollevano EOFException

LineNumberInputStream e LineNumberReader

Metodo:
getLineNumber();

PushBackInputStream e PushBackReader

Metodi: quelli di InputStream più:
unread(), inverso delle 3 forme di read

usato nei parser

PipedInputStream e PipedReader

Per la connessione fra thread

SequenceInputStream

Concatenano flussi

InputStream s = new FileInputStream(nomefile);
InputStream s1 = new FileInputStream(nomefile1);

InputStream s2 = new SequenceInputStream(s,s1);

ObjectInputStream

Consente l'input di oggetti; utilizza l'interfaccia ObjectInput.

Gli oggetti devono essere istanze di classi che implementano l'interfaccia Serializable
Metodi: quelli di DataInput più:
Object readObject () throws ClassNotFoundException, IOException;

Esempio

```
FileInputStream s = new FileInputStream(nomefile);
ObjectInputStream ois = new ObjectInputStream(s);

int i = ois.readInt();
String oggi = (String) ois.readObject ();
Date data = (Date) ois.readObject ();
s.close();
```

Classe scanner

import java.util.Scanner;

Permette l'input formattato di flussi di caratteri

Costruttori:
Scanner(InputStream source)
Scanner(File source)

Metodi:

boolean	nextBoolean();	boolean hasNextBoolean();
int	nextInt();	boolean hasNextInt();
long	nextLong();	boolean hasNextLong();
float	nextFloat();	boolean hasNextFloat ();
double	nextDouble();	boolean hasNextDouble ();
String	nextLine();	boolean hasNextLine ();

void close()

è un iteratore:

```
Scanner sc = new Scanner(new File("myNumbers"));
     while (sc.hasNextLong()) {
        long aLong = sc.nextLong();
     }
sc.close();
```

Esempio

```
//Dato un file di caratteri che contiene un numero per ogni riga, questo programma
//calcola la media dei valori contenuti nel file

import java.io.*;

public class MediaFileChar {

  public static void main(String argv[])
   {
      String nome=null;
      BufferedReader tastiera = new BufferedReader(new InputStreamReader(System.in));
      BufferedReader infile=null; //perche' l'apertura sta in blocco try catch
      int  n=0;
      double s=0;
      String linea;

      System.out.print("nome file da leggere: ");
      try {
          nome = tastiera.readLine();
          infile = new BufferedReader(new FileReader(nome));
          while((linea =infile.readLine())!= null)
              {
              System.out.println(linea);
              s+=Double.parseDouble(linea);
              n++;
              }
          }
       catch (FileNotFoundException e) {System.out.println("File "+nome+" non trovato");
                                        return;}
       catch (IOException e) {System.out.println("Errore: "+e.getMessage()); }
       finally {try {
                        if (infile != null) infile.close();
                        System.out.println("File chiuso ");
                      }
                  catch (IOException e) {System.out.println("Errore: "+e.getMessage()); }
                 }
        if (n!=0)
              System.out.println("media = "+ s/n);
        else
              System.out.println("File vuoto");
    }
```

```
 }
```

Esempio

```
//Dato un file BINARIO che contiene un numero  intero per ogni riga, questo programma
//calcola la media dei valori contenuti nel file; il file sarà stato scritto col programma
// scrivifile.java presentato alla fine del capitolo

import java.io.*;

public class MediaFileBinario {

  public static void main(String argv[])
   {
      String nome=null;
      Scanner tastiera = new Scanner(System.in);
      DataInputStream infile=null; //perche' l'apertura sta in blocco try catch
      int num, n=0;
      double s=0;

      System.out.print("nome file da leggere: ");
      try {
          nome = tastiera.nextLine();
     infile = new DataInputStream(new BufferedInputStream(new FileInputStream(nome)));
          }
      catch (FileNotFoundException e1)
          {System.out.println("File "+nome+" non trovato");
           return;}
      catch (IOException e1)
          {System.out.println("Errore: "+e1.getMessage());
           return;}
```

```
  try {
      while(true)
          {
          num = infile.readInt();
          System.out.println("num = "+num);
          s+=num;
          n++;
           }
         }
     catch (EOFException e2)
          {System.out.println("Letta fine file "); }
     catch (IOException e1)
          {System.out.println("Errore: "+e1.getMessage()); }

     finally { try
              {infile.close();}
              catch (IOException e1)
                 {System.out.println("Errore: "+e1.getMessage()); }

            }
     if (n!=0)
                System.out.println("media = "+ s/n);
        else
                System.out.println("File vuoto");
   }
}
```

Esempio

```
//Dato un file di caratteri che contiene un numero per ogni riga, questo programma
//calcola la media dei valori contenuti nel file

import java.util.Scanner;          import java.io.*;

public class MediaFileCharScanner {

  public static void main(String argv[])
   {
      String nome=null;
      Scanner tastiera = new Scanner(System.in);
      Scanner infile=null; //perche' l'apertura sta in blocco try catch
      int  n=0;
      double s=0, num;

      System.out.print("nome file da leggere: ");

          nome = tastiera.nextLine();
        try{
          infile = new Scanner(new File (nome));

          while (infile.hasNextDouble()) {
                num =infile.nextDouble();
                System.out.println( num);
                s+=num;
                n++;
          }
          infile.close();

        if (n!=0)
              System.out.println("media = "+ s/n);
        else
              System.out.println("File vuoto");
        }
      catch (FileNotFoundException e)
          {System.out.println("File "+nome+" non trovato"); }

    }
}
```

14.5 Classi astratte per l'output

OutputStream: flusso di byte

Writer: flusso di caratteri

```
OutputStream s = …;
Writer r = …;
```

Metodi: (tutti i metodi possono sollevare IOException)

- write()

s.write (b);	r.write (c);
s.write (bbuffer);	r.write (cbuffer);
s.write (bbuffer,100,300);	r.write (cbuffer,100,300);

flush()

close()

```
try {. . . }
finally {s.close();   r.close();}
```

14.6 Flussi concreti di output

ByteArrayOutputStream e CharArrayWriter

```
OutputStream s = new ByteArrayOutputStream();
Writer r = new CharArrayWriter();

OutputStream s = new ByteArrayOutputStream(1024);
Writer r = new CharArrayWriter(1024*1024);
```

l'array viene ingrandito mano a mano che ci si scrive, si può predefinire una dimensione iniziale

alla fine si può copiare il contenuto su un altro flusso o estrarlo su un array o una stringa

s.writeTo(unaltroflusso); w.writeTo(unaltroreader);

byte[] bbuffer = s.toByteArray();
char[] cbuffer = w.toCharArray();
String s1 = s.toString();
String s2 = w.toString();
String s3 = s.toString(String NomeCodifica); //strings Unicode

Metodi:
size() reset()

FileOutputStream e FileWriter

Flussi collegati a file

FileOutputStream s = new FileOutputStream(nomefile);
FileWriter r = new FileWriter(new File(nomefile));

System.out: standard output
System.err: standard error

sono sottoclassi di OutputStreamWriter che può incapsulare ogni OutputStream generando un flusso di caratteri

Flussi da socket e connessioni URL

OutputStreamWriter o = new OutputStreamWriter(socket.getOutputStream()));
OutputStreamWriter o = new OutputStreamWriter(connessioneURL.getOutputStream()));

14.7 Classi astratte di filtri

FilterOutputStream e FilterWriter

I filtri contengono definizioni di metodi e servizi

I filtri possono essere annidati

14.8 Classi concrete di filtri

BufferedOutputStream e BufferedWriter

Effettuano un buffering dell'output, disaccoppiando comunicazione ed elaborazione

BufferedWriter non è una sottoclasse di FilterWriter; implementa il metodo newLine(linea)

```
OutputStream s = new BufferedOutputStream (new FileOutputStream(nomefile));
Writer r = new BufferedWriter( new FileWriter(nomefile));
```

DataOutputStream

Consente l'output formattato;
utilizza (come anche RandomAccessFile) l'interfaccia DataOutput.

Metodi:

```
void writeBoolean(boolean b)   throws IOException;
void writeByte(int i)          throws IOException;
void writeShort(int i )        throws IOException;
void writeChar(int i)          throws IOException;
void writeInt(int i)           throws IOException;
void writeLong(long l)         throws IOException;
void writeFloat(float f)       throws IOException;
void writeDouble(double d)     throws IOException;

void writeBytes(String s)      throws IOException;   //per caratteri 8-bit
void writeChars(String s)      throws IOException;   //per caratteri 16-bit Unicode
void writeUFT(String s)        throws IOException;   //per caratteri Unicode UFT-8
```

Esempio

```
import java.io.*;

public class CopiaFile {

static void copia(String sorgente, String destinazione){
DataInputStream in=null;
DataOutputStream out=null;

try {
        in = new DataInputStream( new FileInputStream(sorgente));
        out = new DataOutputStream( new FileOutputStream(destinazione));

    }
catch (FileNotFoundException e1)
          {System.out.println("File "+sorgente+" non trovato");
           return;}
catch (IOException e1)
          {System.out.println("Errore: "+e1.getMessage());
           return;}

try{
while (true)
        out.writeByte(in.readByte());
}
catch (EOFException e2) { }
catch (IOException e1) {System.out.println ("Errore: "+e1.getMessage());  }
finally {

try{ if (in != null)
        in.close(); }
 catch (IOException e1)
     {System.out.println("Errore: "+e1.getMessage());}

try{ if (out != null)
        out.close();}
 catch (IOException e1)
          {System.out.println("Errore: "+e1.getMessage());}
}
}
```

```
 public static void main(String argv[])
  {
     String nomesorgente=null;
     String nomedestinazione=null;
     BufferedReader tastiera = new BufferedReader(new InputStreamReader(System.in));

     System.out.print("nome file sorgente ");
     try {
         nomesorgente = tastiera.readLine();
                System.out.print("nome file destinazione ");
            nomedestinazione = tastiera.readLine();
                copia(nomesorgente, nomedestinazione);
         }
     catch (IOException e1)
         {System.out.println("Errore: "+e1.getMessage());}
 }
}
```

PrintStream e PrintWriter

Costruttori:

PrintWriter(Writer out);
PrintWriter(Writer out, true); // fa flush() in modo automatico dopo ogni println()

metodi:

print()	println()
System.out.print()	System.out.println()

ObjectOutputStream

Consente l'output di oggetti; utilizza l'interfaccia ObjectOutput.

Gli oggetti devono essere istanze di classi che implementano l'interfaccia Serializable
Metodi: quelli di DataOutput più:
writeObject (Object o) throws IOException;

Esempio

```
FileOutputStream s = new FileOutputStream(nomefile);
ObjectOutputStream oos = new ObjectOutputStream(s);

oos.writeInt(12345);
oos.writeObject ("oggi");
oos.writeObject (new Date());
oos.flush();
oos.close();
```

PipedOutputStream e PipedWriter

```
PipedOutputStream pout1 = new PipedOutputStream();
  PipedInputStream pin1 = new PipedInputStream(pout1);

  PipedOutputStream pout2 = new PipedOutputStream();
  PipedInputStream pin2 = new PipedInputStream(pout2);
```

14.9 File ad accesso casuale

costruttore:
public RandomAccessFile(String nomefile, String accesso) throws IOException
accesso: “r” o “rw”

metodi: quelli di DataInput e DataOutput più:

seek(long pos); int skipBytes(int n); //restituisce il numero di byte saltati

Esempi

```
import java.io.*;

//scrive un file di caratteri

public class ScriviFileChar {


  public static void main(String argv[])
   {
      String nome;
      BufferedReader tastiera = new BufferedReader(new InputStreamReader(System.in));
      PrintWriter outfile=null;
      int num;

      System.out.print("nome file da scrivere: ");
      try {
          nome = tastiera.readLine();
          outfile = new PrintWriter(new BufferedWriter(new FileWriter(nome)));
         }
      catch (IOException e1)
          {System.out.println("Errore: "+e1.getMessage());
           return;}

          for (int i=1; i<=15; i++)
              {
              outfile.println(i);
              }
         outfile.close();
    }
 }
```

```
import java.io.*;

public class ScriviFile {

        public static void main(String argv[])
        {
        String nome;
        BufferedReader tastiera = new BufferedReader(new InputStreamReader(System.in));
        DataOutputStream outfile=null; //perche' l'apertura sta in blocco try catch
        int num;

        System.out.print("nome file da scrivere: ");

        try {
        nome = tastiera.readLine();
        outfile = new DataOutputStream(new BufferedOutputStream(
                                new FileOutputStream(nome)));
        }
        catch (IOException e1) {System.out.println("Errore: "+e1.getMessage());
          return;}


        try {
                for (int i=1; i<=5; i++)
                        outfile.writeInt(i);
        }
        catch (IOException e1) { }
finally { try
              {outfile.close();}
              catch (IOException e1)
                 {System.out.println("Errore: "+e1.getMessage()); }
        }

        }
}
```