# Lezione XVII
# Lu 24-Ott-2005

1

# Array bidimensionali
## cenni

2

## Array bidimensionali

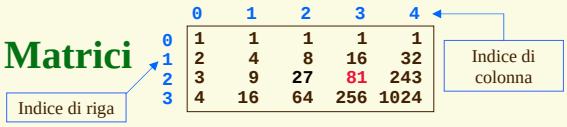
- Problema
  - stampare una tabella con i valori delle potenze $x^y$, per ogni valore di **x** tra 1 e 4 e per ogni valore di **y** tra 1 e 5

```
1    1    1    1    1
2    4    8   16   32
3    9   27   81  243
4   16   64  256 1024
```

e cerchiamo di risolverlo in modo generale, scrivendo metodi che possano ***elaborare un'intera struttura di questo tipo***

3

## Matrici




- Una struttura di questo tipo, con dati organizzati in righe e colonne, si dice ***matrice*** o array bidimensionale
- Un elemento a all'interno di una matrice è identificato da ***una coppia (ordinata) di indici***
  - un ***indice di riga***
  - un ***indice di colonna***
  - esempio $a_{2,3} = 81$
- In Java esistono gli array bidimensionali

4

## Array bidimensionali in Java

- Dichiarazione di un array bidimensionale con elementi di tipo **int**

```
int[][] powers;
```

- Costruzione di array bidimensionale di **int** con 4 righe e 5 colonne

```
new int[4][5];
```

- Assegnazione di riferimento ad array bidimensionale

```
powers = new int[4][5];
```

- Accesso a un elemento di un array bidimensionale

```
powers[2][3] = 81;
int n = powers[2][3];
```

5

## Array bidimensionali in Java

- Ciascun indice deve essere
  - intero
  - maggiore o uguale a 0
  - minore della dimensione corrispondente
- Per conoscere il valore delle due dimensioni
  - il numero di righe è

```
powers.length;
```

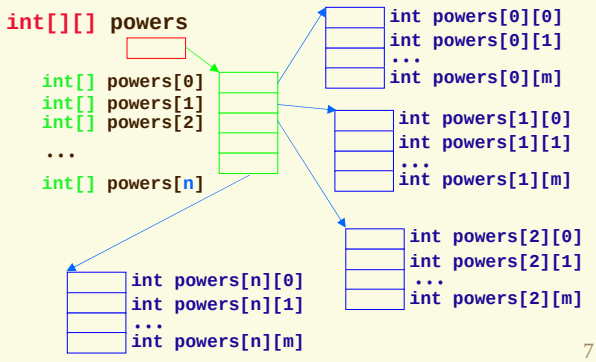
  - il numero di colonne è

```
powers[0].length;
```

(perché un array bidimensionale è in realtà un array di array e ogni array rappresenta una colonna…)

6

## Array bidimensionali




7

## Pacchetti

8

## Organizzare le classi in pacchetti

- Un programma java e' costituito da una raccolta di classi.
- Fin ora i nostri programma erano costituiti da una o al massimo due classi, memorizzate nella stessa directory
- Quando le classi sono tante serve un meccanismo per organizzare le classi: questo meccanismo e' fornito dai pacchetti
- Un pacchetto (package) e' costituito da classi correlate
- Per inserire delle classi in un pacchetto si inserisce come prima istruzione del file sorgente la seguente riga

```
package nomePacchetto;
```

9

## Organizzare le classi in pacchetti

- Nella libreria standard il nome dei paccheti e' un nome composto da vari token separati del punto
  - java.util
  - javax.swing
  - org.omg.CORBA
- Per usare una classe di un pacchetto, si importa con l'enunciato import che gia' conosciamo:

```
import nomePacchetto.nomeClasse;
```

- L'organizzazione delle classi in pacchetti permette di avere classi diverse, ma con lo stesso nome, in pacchetti diversi, e di poterle distinguere

```
import java.util.Timer;
import javax.swing.Timer;
…
java.util.Timer t = new java.util.Timer();
Javax.swing.Timer ts = new javax.swing.Timer();
```

10

## Organizzare le classi in pacchetti

- Esiste un pacchetto speciale, chiamato **pacchetto predefinito**, che e' senza nome
- Se non inseriamo un enunciato `package` in un file sorgente, le classi vengono inserite nel pacchetto predefinito
- Il pacchetto predfinito si puo' estendere sulle classi appartenenti alla stessa directory
- Il pacchetto **java.lang** e' sempre importato automaticamente
- I nomi dei pacchetti devono essere univoci. Come garantirlo?
- Ad esempio invertendo i nomi dei domini
  - **it.unipd.ing (facolta' di Ingegneria)**
- Se non si ha un dominio, si puo' invertire il proprio indirizzo di posta elettronica
  - adriano.luchetta @igi.cnr.it -> it.cnr.igi.luchetta.adriano

11

## Come vengono localizzate le classi

- Se si usano solo le classi della libreria standard non ci si deve preoccupare della posizione dei file
- Un pacchetto si trova in una directory che corrisponde al nome del pacchetto
  - Le parti del nome separate da punti corrispondono a directory annidate
  - it.cnr.igi.luchetta.adriano it/cnr/igi/luchetta/adriano
- Se il pacchetto e' usato con un solo programma, si annida il pacchetto all'interno della directory
  - lab4/it/cnr/igi/luchetta/adriano
- Se si vuole usare con molti programmi si crea una directory specifica
  - /usr/home/**lib/it/cnr/igi/luchetta/adriano**
- Si aggiunge il percorso alla variabile d'ambiente class path
  - $export CLASSPATH=/usr/home/lib: ⇦ Unix (bash)
  - >set CLASSPATH=c:\home\lib; ⇦ Windows

12

## Consigli pratici

- Inserire ciascuna esercitazione di laboratorio in una directory separata propria come, ad esempio lab1, lab2, …
  - `$mkdir lab4;`
  - cosi' in esercitazioni separate potete avere classi con lo stesso nome, che possono pero' essere diverse
- Scegliete una directory di base
  - **/home/ms54637**
  - `mkdir /home/ms54637/lab4`
- Mettete i vostri file sorgente nella cartella
  - **/home/ms54637/lab3/Triangolo.java** ⇦ Unix (bash)
  - **/home/ms54637/lab3/ProvaTriangoloJOptionPane.java**
- Compilare i file sorgenti nella directory base
  - **cd home/ms54637**
  - **javac lab3/Triangolo.java**
  - Notare che il compilatore richiede il nome del file sorgente

13

## Consigli pratici

- Inserire nei file sorgenti l'enunciato package
  - **package lab3;**
- Eseguire il programma nella directory base
  - **cd home/ms54637**
  - **java lab3.ProvaTriangoloJOptionPane**
- **Notare che l'interprete vuole il nome della classe, e quindi il separatore sara' il punto**
- **Se non inserite l'enunciato package, dovete eseguire all'interno della directory di lavoro**
  - **cd home/ms54637/lab3**
  - **java ProvaTriangoloJOptionPane**

14

# Eccezioni

15

## Lanciare eccezioni

- Sintassi:

```
throw oggettoEccezione;
```

- Scopo: lanciare un'eccezione
- Nota: di solito l'*oggettoEccezione* viene creato con **new** *ClasseEccezione*( )

```
public void deposit(double amount)
{
   if (amount <= 0)
      throw new IllegalArgumentException();
   balance = balance + amount;
}
```

16

## Le eccezioni in Java

- Quando un metodo ***lancia*** un'eccezione
  - l'esecuzione del metodo viene immediatamente interrotta
  - l'eccezione viene "propagata" al metodo chiamante la cui esecuzione viene a sua volta immediatamente interrotta
  - l'eccezione viene via via propagata fino al metodo **main()** la cui interruzione provoca l'arresto anormale del programma con la segnalazione dell'eccezione che è stata la causa di tale terminazione prematura
- Il lancio di un'eccezione è quindi un modo per terminare un programma in caso di errore
  - ***non sempre però gli errori sono così gravi...***

17

## Gestire le eccezioni di input

- Nell'esempio di conversione di stringhe in numeri supponiamo che la stringa sia stata introdotta dall'utente
  - se la stringa non contiene un numero valido viene generata un'eccezione **NumberFormatException**
  - sarebbe interessante poter ***gestire*** tale eccezione segnalando l'errore all'utente e chiedendo di inserire nuovamente il dato numerico ***anziché terminare*** prematuramente il programma
- Possiamo ***intercettare*** l'eccezione e gestirla con il costrutto sintattico **try/catch**

18

## Conversione di stringhe in numeri

- La conversione corretta si ottiene invocando il metodo **statico** **parseInt()** della classe **Integer**

```
String ageString = "36";
int age = Integer.parseInt(ageString);
// age contiene il numero 36
```

- La conversione di un ***numero in virgola mobile*** si ottiene analogamente invocando il metodo **statico** **parseDouble()** della classe **Double**

```
String numberString = "34.3";
double number =
    Double.parseDouble(numberString);
// number contiene il numero 34.3
```

19

## Eccezioni nei formati numerici

```
int n = 0;
boolean done = false;
while (!done)
{
   try
   {
      String line = in.next();
      n = Integer.parseInt(line);
      // l'assegnazione seguente viene
      // eseguita soltanto se NON viene
      // lanciata l'eccezione
      done = true;
   }
   catch (NumberFormatException e)
   {
      System.out.println("Riprova");
      // done rimane false
   }
}
```

| done | !done |
|---|---|
| false | true |
| true | false |

## Gestire le eccezioni

- L'enunciato che contiene l'invocazione del metodo che può generare l'eccezione deve essere racchiuso all'interno di una coppia di parentesi graffe precedute dalla parola chiave **try**

```
try
{
   ...
   n = Integer.parseInt(line);
   ...
}
```

- Bisogna poi sapere ***di che tipo è*** l'eccezione generata dal metodo
  - ***ogni eccezione è un esemplare di una classe specifica***
    - ✓nel nostro caso **NumberFormatException**

21

## Gestire le eccezioni

- Il ***blocco try*** è seguito da una ***clausola catch*** definita in modo simile a ***un metodo che riceve un solo parametro*** del ***tipo dell'eccezione*** che si vuole gestire

```
catch (NumberFormatException e)
{
  System.out.println("Riprova");
}
```

- Nel blocco ***catch*** si trova ***il codice che deve essere eseguito nel caso in cui si verifichi l'eccezione***
  - l'esecuzione del blocco **try** viene interrotta nel punto in cui si verifica l'eccezione e non viene più ripresa

22

## Blocco try

- Sintassi:

```
try
{
  enunciatiCheGeneranoUnaEccezione
}
catch (ClasseEccezione oggettoEccezione)
{
  enunciatiEseguitiInCasoDiEccezione
}
```

- Scopo:

  eseguire enunciati che generano una eccezione
  - ***se si verifica l'eccezione*** di tipo ***ClasseEccezione*** eseguire gli enunciati contenuti nella ***clausola catch***
  - ***altrimenti*** ignorare la ***clausola catch***

23

## Blocco try

- Se gli enunciati nel blocco try generano piu' di una eccezione

```
try
{  enunciatiCheGeneranoPiu'Eccezioni
}
catch (ClasseEccezione1 oggettoEccezione1)
{  enunciatiEseguitiInCasoDiEccezione1
}
catch (ClasseEccezione2 oggettoEccezione2)
{  enunciatiEseguitiInCasoDiEccezione2
}
…
```

24

# Lezione XVIII
# Ma 25-Ott-2005

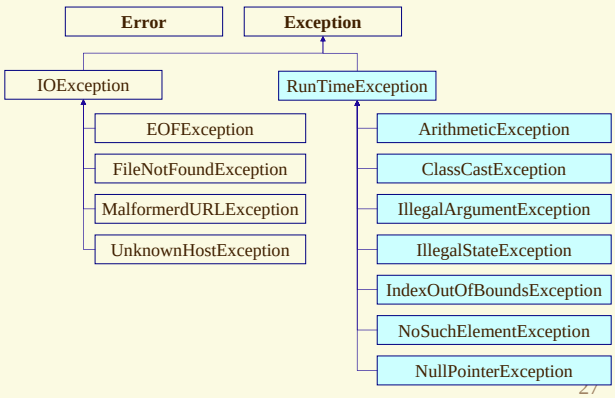
25

## Diversi tipi di eccezioni

- In Java esistono diversi tipi di eccezioni (cioè diverse **classi** di cui le eccezioni sono esemplari)
  - eccezioni di tipo **Error**
  - eccezioni di tipo **Exception**
    - ✓ un sottoinsieme sono di tipo **RuntimeException**
- La gestione delle eccezioni di tipo **Error** e di tipo **RuntimeException** è ***facoltativa***
  - se non vengono gestite e vengono lanciate, provocano la terminazione del programma

26

## Diversi tipi di eccezioni




27

## Diversi tipi di eccezioni

- Ad esempio non e' obbligatorio gestire l'eccezione `NumberFormatException` che appartiene all'insieme delle RuntimeException
- Viene lanciata fra l'altro dai metodi `Integer.parseInt()` `Double.parseDouble()`

```
…
int n = Integer.parseInt(line);
…
```

**Il compilatore non segnala errore se l'eccezione non e' gestita**

- Durante l'esecuzione del programma, se viene generata l'eccezione, questa si propaga fino al metodo main() e provoca la terminazione del programma

```
Exception in thread main java.lang.NumberFormatException:
at java.lang.Integer.parseInt(….)
```

28

## Diversi tipi di eccezioni

- E' invece obbligatorio gestire le eccezioni che sorgono nella gestione dell'input/output, come `java.io.IOException` o `java.io.FileNotFoundException` che appartengono all'insieme delle eccezioni Exception
- La classe `Scanner` può essere usata anche per leggere file (lo vedremo prossimamente); in questo caso il parametro esplicito e' un oggetto di classe `FileReader`
- Il costruttore `FileReader()` se non trova il file lancia l'eccezione controllata (a gestione obbligatoria) `java.io.FileNotFoundException`

```
…
String filename = …;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
…
```

**Il compilatore segnala errore se l'eccezione non e' gestita**

29

## Eccezioni controllate

- Possiamo gestire questa situazione in due modi
  - `try / catch`

```
try
{
   String filename = …;
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
}
catch (FileNotFondException e)
{
…
}
…
```

- Spesso pero' il metodo potrebbe non avere il contesto per gestire l'eccezione (il programmatore non sa come reagire all'interno del metodo)

30

## Eccezioni Controllate

- In alternativa alla gestione con l'enunciato `try / catch,` si può avvisare il compilatore che si e' consapevoli che l'eventuale insorgenza di un'eccezione causera' la terminazione del metodo
- Si lascia quindi la gestione al metodo chiamante

```
public void read(String filename)
   throws FileNotFoundException
{
   FileReader reader = new FileReader(filename);
   Scaner in = new Scanner(reader);
   …
}
```

- Se il metodo può lanciare più eccezioni, si pongono nella firma dopo la clausola `throws`, separandole con una virgola

```
public void read(String filename)
   throws FileNotFoundException,
          ClassNotFoundException
{
   …
}
```

31

## Gestire le eccezioni di input

```
// NON FUNZIONA!
import java.io.FileNotFoundException;
Import java.io.FileReader;
Import java.util.Scanner;
public class ReadInput
{  public static void main(String[] args)
   { String filename = …;
     FileReader reader = new FileReader(filename);
     Scanner in = new Scanner(reader);
     …
   }
}
```

**ReadInput.java:10:**
**unreported exception java.io.FileNotFoundException;**
**must be caught or declared to be thrown**

32

## Avvisare il compilatore: `throws`

- Un'alternativa alla gestione delle eccezioni con il costrutto `try/catch` e' quella di avvisare il compilatore che si e' consapevoli dell'eccezione e che semplicemente si vuole che il metodo termini in caso di eccezione.
- Per dichiarare che un metodo dovrebbe essere terminato quando accade un'eccezione controllata al suo interno si contrassegna il metodo con il marcatore `throws`

```
// FUNZIONA!
import java.io.FileNotFoundException;
Import java.io.FileReader;
Import java.util.Scanner;
public class ReadInput
{  public static void main(String[] args)
      throws FileNotFoundException
   { String filename = …;
     FileReader reader = new FileReader(filename);
     Scanner in = new Scanner(reader);
     …
   }
}
```

## Avvisare il compilatore: `throws`

- La clausola `throws` segnala al chiamante del metodo che potra' trovarsi di fronte a un'eccezione del tipo dichiarato nell'intestazione del metodo.
  - Ad esempio FileNotFoundException nel caso di un metodo che usi il costruttore `FileReader()` della classe `FileReader`
- Il metodo chiamante dovra' prendere una decisione:
  - o gestire l'eccezione (`try/catch`)
  - o dire al proprio metodo chiamante che può essere generata un'eccezione (`throws`)
- Spesso non siamo in grado di gestire un'eccezione in un metodo, perché non possiamo reagire in modo adeguato o non sappiamo che cosa fare (inviare un messaggio allo standard output e terminare il programma potrebbe non essere adeguato)
- In questi casi è meglio lasciare la gestione dell'eccezione al metodo chiamante mediante la clausola `throws`

34

## Lanciare eccezioni: Enunciato throw

```
throw oggettoEccezione;
```

```
public void deposit(double amount)
{
   if (amount <= 0)
      throw new IllegalArgumentException();
   balance = balance + amount;
}
```

- Ora sappiamo
  - programmare nei nostri metodi il lancio di eccezioni appartenenti alla libreria standard (throw new …)
  - gestire eccezioni (try/catch)
  - lasciare al chiamante la gestione di eccezioni sollevate nei nostri metodi (throws nell'intestazione del metodo)
- Non sappiamo ancora costruire eccezioni solo nostre (ad esempio se vogliamo lanciare l'eccezione **LaNostraEccezioneException** come facciamo?)

35

## Argomenti sulla riga comandi

- Quando si esegue un programma Java, è possibile fornire dei parametri dopo il nome della classe che contiene il metodo **main()**

```
$java Program 2 33 Hello
```

- Questi parametri vengono letti dall'interprete Java e trasformati in un array di stringhe che costituisce il parametro del metodo **main()**

```
public class Program {
   public static void main(String[] args)
{
      System.out.println(args.length);
      System.out.println(args[1]);
   }
}
```

```
3
33
```

36

# Esercitazione
## Uso di array

37

## Esercitazione

- Scrivere la classe TextContainer che memorizzi un testo suddiviso in righe
- L'interfaccia pubblica della classe sia la seguente

```
public class TextContainer
{  // parte privata
    …
   public TextContainer() {…}
   public void add(String aLine){…}
   public String remove() {…}
   public boolean isEmpty() {…}
   public int replace(String find,
      String replacement) {…}
}
```

## Esercitazione

- **`public TextContainer():`** costruttore che inizializza un contenitore di testo vuoto
- **`public void add(String aLine):`** aggiunge in coda al testo la riga String aLine
- **`public String remove():`** restituisce, rimuovendola, la prima riga del testo
- **`public boolean isEmpty():`** restituisce true se il contenitore è vuoto, false altrimenti
- **`public int replace(String find, String replacement):`** ricerca nel testo la parola **`String find`** e la sostituisce con la parola **`String replacement`**. Restituisce il numero di sostituzioni effettuate. Considera una parola come una stringa delimitata da uno o più spazi (o TAB)
- **`public String toString(): descrizione testuale`**

39

## Progettare la classe

1. Definire l'interfaccia pubblica
   - L'interfaccia pubblica va definita con molta cura perchè fornisce gli strumenti con cui possiamo elaborare l'informazione contenuta negli oggetti istanziati con la classe
   - Il nome dei metodi deriva generalmente da verbi che descrivono le azioni effettuate dai metodi stessi
   - Nel caso della nostra esercitazione l'interfaccia pubblica è già definita
2. Definire la parte privata
   - Scegliere con cura le variabili di esemplare in cui viene memorizzata l'informazione (i dati) associata agli oggetti della classe
   - Realizzazioni diverse della stessa classe possono avere parti private differenti, ma devono avere la stessa interfaccia pubblica!!!

40

## Progettare la classe: parte privata

- La classe memorizza un testo suddiviso in righe
- Una soluzione semplice è di realizzare la parte privata tramite un **array di riferimenti a stringa** nel quale ciascun elemento è un riferimento a una riga del testo
- L'elemento dell'array di indice zero sarà la prima riga, l'elemento di indice 1 la seconda riga, e così via
- Quanti elementi avrà il nostro array? A priori non sappiamo quante righe dobbiamo memorizzare, quindi realizzeremo un **array riempito parzialmente a ridimensionamento dinamico**

41

## Progettare la classe: parte privata

```
/**
contenitore di testo. Classe non ottimizzata
@author Adriano Luchetta
@version 28-Ott-2003
*/
import java.util.Scanner;
public class TextContainer
{
   //parte privata
   private String[] line;/* riferimento ad
                            array */
   private int lineSize; /* numero di elementi
                            già inseriti
                            nell'array */
```

Commento standard javadoc

Riferimento ad array di String

array riempito solo in parte

42

## Progettare la classe: costruttori

- Il costruttore inizializza le variabili di esemplare
- Generalmente, se non e' noto il numero di elementi da memorizzare nell'array, si inizia con **un elemento**
- Un costruttore che crea un contenitore vuoto fa al nostro caso

```
/**
costruttore del contenitore di testo
*/
public TextContainer()
{
   line = new String[1];// un elemento!
   lineSize = 0;        // vuoto
}
```

Commento standard javadoc

Allocazione Array!

Numero di elementi già inseriti nell'array

43

## Progettare la classe: metodo add()

```
/**
inserisce una riga nel contenitore  di testo
@param aLine la riga da inserire
*/
public void add(String aLine)
{  // Array dinamico: ridimensionare line[]
   if (lineSize >= line.length)
   {  String[] newLine = new String[2 *
         line.length]; //raddoppio elementi
      for (int i = 0; i < line.length; i++)
         newLine[i] = line[i];
      line = newLine;
   }
   // inserimento di String aLine
   line[lineSize] = aLine;
   // incremento del contatore
   lineSize++;
}
```

Commento standard javadoc

Array dinamico

## Metodo isEmpty()

- Metodo predicativo: significa che ritorna un valore boolean

```
/**
verifica se il contenitore e' vuoto
@return true se il contenitore e' vuoto,
altrimenti false
*/
public boolean isEmpty()
{
   return lineSize == 0;//espress. logica
}
```

Commento standard javadoc

45

## Metodo remove()

```
/**
  estrae la prima riga del testo
  @return la prima riga del testo se il
  contenitore non e' vuoto, altrimenti null
  NB: ad ogni estrazione si ridimensiona
  l'array!
*/
public String remove()
{  if (isEmpty())
      return null;
   //appoggio temp. dell'elemento di indice 0
   String tmpString = line[0];
   // eliminazione dell'elemento di indice 0
   for (int i = 0; i < lineSize-1; i++)
      line[i] = line[i+1];
   lineSize--; //decremento
   return tmpString;
}
```

Commento standard javadoc

## Metodo replace()

- Il metodo deve scandire tutte le righe memorizzate per cercare la parola **`String find`** (la classe TextContainer memorizza righe, non singole parole)
- Il metodo deve essere in grado di suddividere le singole righe in parole e di individuare l'eventuale presenza della parola **`String find`**
- Nel caso trovi la parola **`String find`** deve sostituirla con **`String replacement`**, incrementare il contatore delle sostituzioni, inserire nell'array la riga modificata

47

## Metodo replace()

```
/**
sostituisce nel testo una parola con un'altra
@param find la parola da sostituire
@param replacement la parola sostitutiva
@return il numero di sostituzioni eseguite
@throws IllegalArgumentException se uno dei
 parametri e' null
*/
public int replace(String find, String
  replacement)
{   if (find == null || replacement == null)
       throw new IllegalArgumentException();
    int count = 0; // contatore parole
                   //    sostituite
//continua
```

Commento standard javadoc

48

```
   for (int i = 0; i < lineSize; i++)
   {
      Scanner st = new Scanner(line[i]);
      String tmpLine = "";
      boolean found = false;
      while (st.hasNext()) // separazione in token
      {
         String token = st.next();
         if (token.equals(find))
         {
            token = replacement;
            count++; // incremento contatore delle sost.
            found = true;
         }
         tmpLine = tmpLine +  token + " ";
      }
      if (found)
         line[i] = tmpLine;
   }
   return count;
} // fine metodo
} // fine classe
```

Ciclo for per scandire le righe del testo

## Metodo toString()

Commento standard javadoc

```
/**
   sostituisce la descrizione testuale
   @return descrizione testuale
*/
public String toString()
{
    String ret = "";
    for (int i = 0; i < lineSize; i++)
       ret = ret + line[i] + "\n";

    return ret;
}
```

50

## Una classe di prova

- Programmiamo una classe di prova TextContainerTester che acquisisca da standard input un testo
- Se riceve da riga di comando come argomenti due parole, le usi per ricerca/sostituzione nel testo
- Sostituisca nel testo, eventualmente, la prima parola ricevuta da riga di comando con la seconda
- Invii allo standard output il testo, eventualmente modificato
- Useremo la classe di prova per leggere un testo da file mediante re-indirizzamento dello standard input, modificare il testo e scriverlo su un nuovo file usando il re-indirizzamento dello standard output.

- In alternativa a scrivere la classe di prova TextContainerTester è possibile rendere la classe TextContainer eseguibile, realizzando all'interno il metodo main()

51

## Una classe di prova

```
import java.util.Scanner;
public class TextContainerTester
{
   public static void main(String[] args)
   { //standard input
     Scanner in = new Scanner(System.in);

     //istanza oggetto di classe TextContainer
     TextContainer text = new TextContainer();

    //legge il testo da standard input
     while(in.hasNextLine())
     {
         String str = in.nextLine();
         text.add(str);
     }

     System.out.prinln(text);

// continua
```

52

## Una classe di prova

```
//continuazione

     //sostituzione parola
     int n = 0;
     if (args.length > 1) // se ci sono almeno 2 arg.
         n = text.replace(args[0], args[1]);

     //invio a standard output
     while(!text.isEmpty())
       System.out.println(text.remove());

     System.out.println("\nn. " + n +
        " sostituzioni effettuate");
   }
} //Fine TextContainerTester
```

53

## Provare una classe

- In alternativa alla classe di prova, si puo' rendere la classe eseguibile programmando il metodo main() nella classe

```
import …; // import necessari
public class TextContainer // ora la classe e' eseguibile
{ …// qui va il corpo della classe programmato precedentemente
   public static void main(String[] args)
   { //standard input
     Scanner in = new Scanner(System.in);
     //istanza oggetto di classe TextContainer
     TextContainer text = new TextContainer();
     //legge il testo da standard input
     while(in.hasNextLine())
        text.add(in.nextLine());
     //sostituzione parola
     int n = 0;
     if (args.length > 1) // se ci sono almeno 2 arg.
         n = text.replace(args[0], args[1]);
     //invio a standard output
     while(!text.isEmpty())
       System.out.println(text.remove());
     System.out.println("\nn. " + n + " sostituzioni"
                        + "effettuate");
   }
} //Fine TextContainer
```

# Lezione XIX
# Me 26-Ott-2005

# Gestione di file in java

55

## Gestione di file in Java

- Finora abbiamo visto programmi Java che interagiscono con l'utente soltanto tramite i flussi standard di ingresso e di uscita (System.in, .out)
  - ciascuno di tali flussi può essere collegato a un file con un comando di sistema operativo (re-indirizzamento)
- Ci chiediamo: è possibile leggere e scrivere file in un programma Java?
  - con la re-direzione di input/output, ad esempio, non possiamo leggere da due file o scrivere su due file...

56

## Gestione di file in Java

- Limitiamoci inizialmente ad affrontare il problema della ***gestione di file di testo*** (file contenenti caratteri)
- In un file di testo i numeri vengono memorizzati come stringhe di caratteri numerici decimali e punti: 123.45 diventa "123.45"
  - esistono anche i ***file binari***, che contengono semplicemente configurazioni di bit (byte) che rappresentano qualsiasi tipo di dati. Nei file binari i numeri sono memorizzati con la loro rappresentazione numerica. Es.: l'intero 32 diventa la sequenza di 4 byte 00000000 00000000 000000000 00100000
- La gestione dei file avviene interagendo con il sistema operativo mediante classi del pacchetto **java.io** della libreria standard

57

## Gestione di file in Java

- Per leggere/scrivere file di testo si usano oggetti istanziati con le classi
  - **java.io.FileReader**
  - **java.io.FileWriter**
- In generale le classi **reader** e **writer** sono in grado di gestire flussi di caratteri
- Per leggere/scrivere file binari si usano oggetti delle classi
  - **java.io.FileInputStream**
  - **java.io.FileOutputStream**
- In generale le classi con suffisso ***Stream*** sono in grado di gestire flussi di byte

58

## Lettura di file di testo

- Prima di ***leggere caratteri*** da un file (esistente) occorre ***aprire*** il file in ***lettura***
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileReader**

```
FileReader reader = new FileReader("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di stringa: "`file.txt`"
  - se il file non esiste, viene lanciata l'eccezione **FileNotFoundException** a gestione obbligatoria

59

## Lettura di file di testo

- Con l'oggetto di tipo **FileReader** si può invocare il metodo **read()** che restituisce un intero a ogni invocazione, iniziando dal primo carattere del file e procedendo fino alla fine del file stesso

```
FileReader reader = new FileReader("file.txt");
while(true)
{  int x = reader.read(); // read restituisce un
   if (x == -1) break;    // intero che vale -1
   char c = (char) x;     // se il file è finito
   // elabora c
} // il metodo lancia IOException, da gestire
```

- Non è possibile tornare indietro e rileggere caratteri già letti
  - bisogna creare un nuovo oggetto di tipo **FileReader**

60

## Lettura con buffer

- In alternativa, si può costruire un'esemplare della classe **Scanner**, con il quale leggere righe di testo dal file

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
while(in.hasNextLine())
{  String line = in.nextLine();
   …  // elabora
}
// FileReader lancia FileNotFoundException,
// da gestire
```

- Quando il file finisce, il metodo hasNext() della classe BufferedReader ritorna **false**!

61

## Lettura di file di testo

- Al termine della lettura del file (che non necessariamente deve procedere fino alla fine…) occorre *chiudere* il file

```
FileReader reader = new FileReader("file.txt");
...
reader.close();
```

- Il metodo close() lancia **IOException**, da gestire obbligatoriamente
- Se il file non viene chiuso non si ha un errore, ma una potenziale situazione di instabilità per il sistema operativo

62

## Scrittura di file di testo

- Per scrivere caratteri in un *file di testo* occorre *aprire* il file in scrittura
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **PrintWriter**

Java 1.5

```
PrintWriter writer = new PrintWriter("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di stringa e può lanciare l'eccezione **IOException**, che deve essere gestita
    - ✓ *se il file non esiste, viene creato*
    - ✓ *se il file esiste, il suo contenuto viene sovrascritto con i nuovi contenuti*
    - ✓ *e se vogliamo aggiungere in coda al file senza cancellare il testo gia' contenuto?*

boolean append

```
FileWriter writer = new
    FileWriter("file.txt“, true);
PrintWriter out = new PrintWriter(writer);
```

63

## Scrittura di file di testo

- Al termine della scrittura del file occorre *chiudere* il file

```
PrintWriter out = new PrintWriter("file.txt");
  ...
  out.close();
```

- Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ***Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto***

64

## Esempio:Copiare un file di testo

```
/**
   Copia1.java copia un file di testo una riga alla volta
   I nomi dei file sono passati come parametri dalla
   riga di comando
*/
import java.io.IOException;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Scanner;
public class Copia1
{ public static void main (String[] args) throws
     IOException
  {
    FileReader reader = new FileReader(args[0]);
    Scanner in = new Scanner(reader);
    PrintWriter out = new PrintWriter(args[1]);

    while (in.hasNextLine())
      out.println(in.nextLine());
    out.close(); //chiusura scrittore
    in. close(); //chiusura lettore
  }
}
```

65

## Copiare a caratteri un file di testo

```
/** Copia2.java -- copia un file di testo un carattere
    alla volta
    I nomi dei file sono passati cone argomenti nella
    riga di comando
    Fare attenzione a non cercare di copiare un file
    binario con questo programma
 */
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class Copia2
{ public static void main (String[] arg) throws
     IOException
  {  FileReader in = new FileReader(arg[0]);
     FileWriter out = new FileWriter(arg[1]);
     int c = 0;
     while ((c = in.read()) != -1)
       out.write(c);
     out.close(); //chiusura scritture
     in.close();  //chiusura lettore
  }
}
```

66

## Esempio:numerare le righe

```
/**
  NumeratoreRighe.java crea la copia di un file di testo
  numerandone le righe. I nomi dei file sono passati
  come parametri dalla riga di comando
*/
import java.io.IOException;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Scanner;
public class NumeraRighe
{ public static void main (String[] args) throws
     IOException
  {
    FileReader reader = new FileReader(args[0]);
    Scanner in = new Scanner(reader);
    PrintWriter out = new PrintWriter(args[1]);
    int numRiga = 0;
    while (in.hasNextLine())
    { numRiga++;
      out.println(numRiga + " " + in.nextLine());
    }
    out.close(); //chiusura scrittore
    in. close(); //chiusura lettore
  }
}
```

67

## Lettura di file binari

- Prima di ***leggere byte*** da un file (esistente) occorre ***aprire*** il file in ***lettura***
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo `FileInputStream`

```
FileInputStream inputStream = new
   FileInputStream("file.dat");
```

  - il costruttore necessita del nome del file sotto forma di stringa
  - se il file non esiste, viene lanciata l'eccezione **FileNotFoundException** che deve essere gestita

68

## Lettura di file binari

- Con l'oggetto di tipo `FileInputStream` si può invocare il metodo **read()** che restituisce un ***intero*** (fra 0 e 255 o -1) a ogni invocazione, iniziando dal primo byte del file e procedendo fino alla fine del file stesso

```
FileInputStream in = new
   FileInputStream("file.txt");
while(true)
{  int x = in.read();      // read restituisce un
   if (x == -1) break;     // intero che vale -1
   byte b = (byte) x;      // se il file è finito
   // elabora b
} // il metodo lancia IOException, da gestire
```

- Non è possibile tornare indietro e rileggere caratteri già letti
  - bisogna creare un nuovo oggetto di tipo **FileInputStream**

69

## Scrittura di file binari

- Prima di scrivere byte in un *file binario* occorre ***aprire*** il file in scrittura
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileOutputStream**

```
FileOutputStream outputStream = new
   FileOutputStream("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di stringa e può lanciare l'eccezione **IOException**, che deve essere gestita
    - ✓ ***se il file non esiste, viene creato***
    - ✓ ***se il file esiste, il suo contenuto viene sovrascritto con i nuovi contenuti***
    - ✓ ***e se vogliamo aggiungere in coda al file senza cancellare il testo gia' contenuto?***

```
FileOutputStream outputStream = new
   FileOutputStream("file.txt", true);
```

`boolean append`

70

## Scrittura di file binario

- L'oggetto di tipo **FileOutputStream** ha il metodo write() che scrive un byte alla volta
- Al termine della scrittura del file occorre ***chiudere*** il file

```
FileOutputStream outStream = new
   FileOutputStream("file.txt");
...
outStream.close();
```

- Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ***Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto***

71

## Gestione di file in Java

- Usando le classi
  - ***FileReader*** e ***FileWriter*** del pacchetto **java.io** è possibile manipolare, all'interno di un programma Java, più ***file di testo*** in lettura e/o più file in scrittura
  - ***FileInputStream*** e ***FileOutputStream*** del pacchetto **java.io** è possibile manipolare, all'interno di un programma Java, più ***file binari*** in lettura e/o più file in scrittura
  - I metodi read() e write() di queste classi sono gli unici metodi basilari di lettura/scrittura su file della libreria standard
  - Per elaborare linee di testo (o addirittura interi oggetti in binario) i flussi e i lettori devono essere combinati con altre classi, ad esempio Scanner o PrintWriter

72

## Copiare un file binario

```
/**
 Copia3.java - copia un file binario un byte alla volta
*/
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class Copia3
{
  public static void main (String[] arg) throws
    IOException
  {
    FileInputStream in = new FileInputStream(arg[0]);
    FileOutputStream out = new
        FileOutputStream(arg[1]);
    int c = 0;
    while ((c = in.read()) != -1)
       out.write(c);
    out.close(); //chiusura scrittore
    in.close();  //chiusura lettore
  }
}
```

73

## Fine riga e EOF

74

## Che cosa è il *fineriga*?

- Un file di testo è un file composto da caratteri (nel nostro caso da caratteri di un byte secondo la codifica ASCII)
- Il file è diviso in *righe* cioè in sequenze di caratteri terminati dalla *stringa fineriga*
- Il ***fineriga*** dipende dal sistema operativo
  - "\r\n" Windows
  - "\n" Unix – Linux
  - "\r" MAC OS
- L'uso di un `Scanner` e del metodo `nextLine()` in ingresso, di un `PrintWriter` e del metodo `println()` in uscita, consentono di ignorare il problema della dipendenza del ***fineriga*** dall'ambiente in cui un'applicazione è utilizzata
- Se si elabora un file di testo un carattere alla volta è necessario gestire le differenze fra i diversi ambienti, in particolare il fatto che in due casi la stringa fineriga è di un carattere e nell'altro è composta da due caratteri

75

## Che cosa è l'*End Of File* (EOF)

- La fine di un file (EOF) corrisponde alla fine dei dati che compongono il file
- L'EOF non è un carattere contenuto nel file!
- La condizione di EOF viene segnalata quando si cerca di leggere un byte o un carattere dopo la fine fisica del file

76

## Errori tipici con i metodi e con i costruttori

77

## Invocare un metodo senza oggetto

```
public class Program
{  public static void main(String[] args)
   {  BankAccount account = new BankAccount();
      deposit(500); // ERRORE
   }
}
```

- In questo caso il compilatore non è in grado di compilare la classe **Program**, perché non sa a quale oggetto applicare il metodo **deposit**, che non è un metodo statico e quindi richiede un riferimento a un oggetto da usare come parametro implicito

78

## Invocare un metodo senza oggetto

- Ci sono, però, casi in cui è ammesso (ed è ***pratica molto comune***!) invocare metodi non statici senza specificare un riferimento ad oggetto
- Questo accade quando si invoca ***un metodo non statico*** di una classe da ***un altro metodo non statico della stessa classe***
  - viene usato il parametro implicito **this**

```
public void withdraw(double amount)
{  balance = getBalance() - amount;
}
```

```
this.getBalance()
```

79

## Tentare di *ricostruire* un oggetto

- A volte viene la tentazione di ***invocare un costruttore su un oggetto già costruito*** con l'obiettivo di ***riportarlo alle condizioni iniziali***

```
BankAccount account = new BankAccount();
account.deposit(500);
account.BankAccount(); // NON FUNZIONA!
```

- Il messaggio d'errore è un po' strano… il compilatore cerca un ***metodo*** **BankAccount**, non un ***costruttore***, e naturalmente non lo trova!

```
cannot resolve symbol
symbol   : method BankAccount ( )
location : class BankAccount
```

80

## Tentare di *ricostruire* un oggetto

- Un costruttore crea un ***nuovo oggetto*** con prefissate condizioni iniziali
  - un costruttore può essere invocato ***soltanto*** con l'operatore **new**
- La soluzione è semplice
  - ***assegnare un nuovo oggetto alla variabile oggetto che contiene l'oggetto che si vuole "ricostruire"***

```
BankAccount account = new BankAccount();
account.deposit(500);
account = new BankAccount();
```

81

## Metodi statici

- Metodi statici: non operano su variabili di esemplare, ma solo sui parametri espliciti ed, eventualmente, su variabili statiche
- L'invocazione del metodo avviene

  Con la sintassi `NomeClasse.nomeMetodo();`

  per i metodi pubblici invocati al di fuori della classe in cui sono definiti

  `Math.exp(2.5);`

  Con la sintassi `nomeMetodo();`

  all'interno della classe dove sono definiti.

82

## Metodi di esemplare (non-statici)

- Metodi non-statici o di esemplare: operano su variabili di esemplare della classe, oltre a operare sui parametri espliciti e variabili statiche
- L'invocazione del metodo avviene
  - con l'istanza di un oggetto della classe in cui sono definiti

    `oggetto.nomeMetodo();`

    `System.out.println();`
  - il riferimento all'oggetto con cui viene effettuata l'invocazione diventa internamente il parametro implicito **this**
- Possono essere invocati all'interno della classe dove sono definiti con la sintassi: `nomeMetodo();`
- In questo caso operano sul parametro implicito e' **this**.

```
public Complex div(Complex z)
{ return this.mult(z.inv()); }
// this puo essere omesso
```

83

## Metodi di esemplare e statici

- La classe DynamicStringArray1 seguente usa il metodo di esemplare

  `private String[] resize(int newLength)`

  per ridimensionare un array definito come variabile di esemplare:
  - Il riferimento all'array da ridimensionare e' memorizzato nella variabile d'istanza line che e' visibile nel metodo
  - Il metodo ritorna il riferimento al nuovo array
  - Il metodo add() illustra come si invoca il metodo di istanza resize()
- La classe DynamicStringArray2 seguente usa il metodo statico

```
private static String[] resize(String[] oldArray,
                               int newLength)
```

  per ridimensionare un array:
  - Il riferimento all'array da ridimensionare e' passato come parametro esplicito del metodo
  - Il metodo ritorna il riferimento al nuovo array
  - Il metodo add() illustra come si invoca il metodo di istanza resize()

84

```
public class DynamicStringArray1
{ private String[] line; // variabile di istanza
  private int lineSize;  // variabile di istanza
  // costruttore
  public DynamicStringArray1()
  { line = new String[1];
    lineSize = 0;
  }
  // usa un metodo di esemplare per ridimensionare
  public void add(String aLine)
  { if (lineSize >= line.length)
      line = resize(2 * line.length);
    line[lineSize] = aLine;
    lineSize++;
  }
  // ridimensiona l'array: metodo d'esemplare
  private String[] resize(int newLength)
  { String[] newArray = new String[newLength];
    int n = line.length;
    if (newLength < n)
      n = newLength;
    for (int i = 0; i < n; i++)
      newArray[i] = line[i];
    return newArray;
  }
} /* esempio di classe che usa un metodo
     di esemplare privato */
```

85

```
public class DynamicStringArray2
{ private String[] line; // variabile di istanza
  private int lineSize;  // variabile di istanza
  // costruttore
  public DynamicStringArray2()
  { line = new String[1];
    lineSize = 0;
  }
  // usa un metodo statico per ridimensionare
  public void add(String aLine)
  { if (lineSize == line.length)
      line = resize(line, 2 * line.length);
    line[lineSize] = aLine;
    lineSize++;
  }
  // ridimensiona l'array: metodo statico
  private static String[] resize(String[] oldArray,
                                 int newLength)
  { String[] newArray = new String[newLength];
    int n = oldArray.length;
    if (newLength < n)
      n = newLength;
    for (int i = 0; i < n; i++)
      newArray[i] = oldArray[i];
    return newArray;
  }
} /* esempio di classe che usa un metodo
     di esemplare privato */
```

86

# Lezione XX
# Gi 27-Nov-2005

# Array paralleli

87

## Array paralleli

- Scriviamo un programma che riceve in ingresso un elenco di dati che rappresentano
  - i nomi di un insieme di studenti
  - il voto della prova scritta
  - il voto della prova orale
- I dati di ciascun studente sono inseriti in una riga separati da uno spazio
- prima il nome, poi il voto scritto, poi il voto orale
- I dati sono terminati da una riga vuota
- Aggiungiamo le seguenti funzionalità
  - il programma chiede all'utente di inserire un comando per identificare l'elaborazione da svolgere
    - ✓ 0 significa "termina il programma"
    - ✓ 1 significa "visualizza la media dei voti di uno studente"
- il programma chiede all'utente di inserire il cognome di uno studente, se necessario (cioè nel caso 1)

88

## Soluzione OOP

```
class Student
{
   private String name;       // nome
   private double wMark;      // voto prova scritta
   private double oMark;      // voto prova orale

   public Student(String aName, double aWMark,
                  double aOMark)
   {
      name = aName;
      wMark = aWMark;
      oMark = aOMark;
   }

   public String getName() { return name; }
   public double getWMark(){ return wMark;}
   public double getOMark(){ return oMark;}
}
```

89

## Soluzione OOP

```
import java.util.Scanner;
import java.util.Locale;
public class ExaminationOOP
{ public static void main(String[] args)
  {
    Student[] students = new Student[10];
    int studentsSize = 0;

    Scanner in = new Scanner(System.in);

    while (true)
    { Student s = readStudent(in);

      if (s == null) break;

      if (studentsSize >= students.length)
         students = resize(students, 2 * studentsSize);

      students[studentsSize] = s;
      studentsSize++;
    }
```

Metodo privato

90

## Soluzione OOP

```
    while (true)
    { System.out.println("Comando?");
      int command = in.nextInt();

      if (command == 0) break;

      if (command == 1)
      { System.out.println("nome?");
        String name = in.next();

        printAverage(students, name, studentsSize);
      }
      else
        System.out.println("Comando errato");
    }
  }
```

Metodo privato

91

## Soluzione OOP

```
/* Acquisisce uno studente da flusso di dati
   ritorna null se il flusso di dati e' finito */
private static Student readStudent(Scanner input)
{ final String END_OF_DATA = "";

  String line = END_OF_DATA;
  if (input.hasNextLine())
     line = input.nextLine();

  if (line.equalsIgnoreCase(END_OF_DATA))
     return null;

  Scanner st = new Scanner(line);
  st.useLocale(Locale.US);

  String nameTmp = st.next();
  double wMarkTmp = st.nextDouble();
  double oMarkTmp = st.nextDouble();
  return new Student(nameTmp, wMarkTmp, oMarkTmp);
}
```

92

## Soluzione OOP

```
/* Stampa la media di uno studente
   @param students l'array di studenti
   @param name nome dello studente
   @param size numero degli studenti
*/
 private static void printAverage(Student[] students,
                     String name, int size)
 {
    int i = findName(students, name, size);
    if (i == -1)
      throw new IllegalArgumentException();
    else
    { double avg = (students[i].getWMark()
                    + students[i].getOMark()) / 2;
      System.out.println(name + " " + avg);
    }
 }
```

93

## Soluzione OOP

```
/* Trova l'indice dell'array corrispondente a uno
   studente
   @param students l'array di studenti
   @param name nome dello studente
   @param count numero degli studenti
   @return l'indice se lo studente e' presente,
           -1 altrimenti
*/
private static int findName(Student[] students,
                            String name, int count)
 {
     for (int i = 0; i < count; i++)
     if (students[i].getName().equals(name))
       return i;

   return -1;
 }
```

94

## Soluzione OOP

```
  /* il solito metodo resize()… */

  private static Student[] resize(Student[] oldArray,
                                  int newLength)
  {  int n = oldArray.length < newLength ?
                oldArray.length : newLength;

     Student[] newArray = new Student[n];
     for (int i = 0; i < n; i++)
       newArray[i] = oldArray[i];

     return newArray;
  }
} // fine della classe Examinationt
```

95

## Array paralleli

```
import java.util.*;
public class Examination
{ public static void main(String[] args)
  {String line = END_OF_DATA;
    String[] names = new String[10]; // array paralleli
    double[] wMarks = new double[10];
    double[] oMarks = new double[10];
    int size = 0; // array riempiti solo in parte
    Scanner in = new Scanner(System.in);
    while (true)
    {
      if (in.hasNextLine())  line = in.nextLine();
      if (line.equalsIgnoreCase(END_OF_DATA)) break;

      if (size >= names.length)
      {  names =  resize(names,  size * 2);
         wMarks = resize(wMarks, size * 2);
         oMarks = resize(oMarks, size * 2);
      }
      Scanner st = new Scanner(line);
      st.useLocale(Locale.US);
      names[size] = st.next();
      wMarks[size] = st.nextDouble();
      oMarks[size] = st.nextDouble();
      size++;
    }
```

96

## Array paralleli

```
while (true)
{
  System.out.print("Comando: \n- 0 termina il"
    + " programma \n- 1 calcola la media \ncomando? ");

  int command = in.nextInt();

  if (command == 0)
     break;

  if (command == 1)
  {
    System.out.print("nome?: ");
    String name = in.next();
    printAverage(names, wMarks, oMarks, name, size);
  }
  else
    System.out.println("Comando errato");
 }
}
```

97

## Array paralleli

```
private static void printAverage (String[] names,
                                  double[] wMarks,
                                  double[] oMarks,
                                  String name,
                                  int count)
{
  int i = findName(names, name, count);

  if (i == -1)
    throw new IllegalArgumentException();
  else
  { double avg = (wMarks[i] + oMarks[i]) / 2;
    System.out.println(name + " " + avg);
  }
}

private static int findName(String[] names,
                            String name, int count)
{ for (int i = 0; i < count; i++)
  if (names[i].equals(name))
    return i;
  return -1;
}
```

98

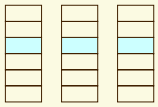
## Array paralleli

```
/* il solito metodo resize()… */
 private static String[] resize(String[] oldArray,
                                int newLength)
 {
   int n = oldArray.length < newLength ?
             oldArray.length : newLength;
   String[] newArray = new String[n];
   for (int i = 0; i < n; i++)
     newArray[i] = oldArray[i];
   return newArray;
 }

 /* un altro solito metodo resize()… */
 private static double[] resize(double[] oldArray,
                                int newLength)
 {
   int n = oldArray.length < newLength ?
             oldArray.length : newLength;
   double[] newArray = new double[n];
   for (int i = 0; i < n; i++)
     newArray[i] = oldArray[i];
   return newArray;
 }
}
```

99

## Array paralleli



- L'esempio presentato usa una struttura dati denominata "***array paralleli***"
  - si usano ***diversi array*** per contenere i dati del problema, ma gli array sono tra loro ***fortemente correlati***
    - ✓devono sempre contenere lo ***stesso numero di elementi***
  - ***elementi aventi lo stesso indice*** nei diversi array ***sono tra loro correlati***
    - ✓in questo caso, rappresentano ***diverse proprietà dello stesso studente***
  - molte elaborazioni hanno bisogno di ***utilizzare tutti gli array***, che devono quindi essere passati come parametri come nel caso del metodo **printAverage()**

100

## Array paralleli

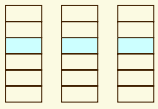
- Le strutture dati di tipo "array paralleli" sono molto usate in linguaggi di programmazione **non OOP**, ma presentano ***numerosi svantaggi*** che possono essere superati in un linguaggio OOP come Java
  - le modifiche alle dimensioni di un array devono essere fatte contemporaneamente a tutti gli altri
  - i metodi che devono elaborare gli array devono avere una lunga lista di parametri espliciti
  - non è semplice scrivere metodi che devono ritornare informazioni che comprendono tutti gli array
- ***nel caso presentato, ad esempio, non è semplice scrivere un metodo che realizzi tutta la fase di input dei dati, perché tale metodo dovrebbe avere come valore di ritorno i tre array!***

101

## Array paralleli in OOP

- Le tecniche di OOP consentono di gestire molto più efficacemente le strutture dati di tipo "array paralleli"
- Si definisce una classe che contiene tutte le informazioni relative a "una fetta" degli array, cioè raccolga tutte le informazioni presenti nei diversi array in relazione a un certo indice

```
class Student
{
  private String name;
  private double wMark;
  private double oMark;
}
```



102

## Non usare array paralleli

- Tutte le volte in cui il problema presenta una struttura dati del tipo "array paralleli", si consiglia di ***trasformarla in un array di oggetti***
  - occorre realizzare la classe con cui costruire gli oggetti
- Risulta molto più facile scrivere il codice e, soprattutto, apportare modifiche
- Immaginiamo di introdurre un'altra informazione per gli studenti (ad esempio il numero di matricola)
  - nel caso degli array paralleli è necessario ***modificare le firme di tutti i metodi***, per introdurre il nuovo array
  - nel caso dell'array di oggetti **Student**, basta modificare la classe **Student**

103

## Ricorsione

104

## Il calcolo del fattoriale

- La funzione fattoriale, molto usata nel calcolo combinatorio, è così definita

$f: \mathcal{N} \to \mathcal{N}$

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

dove **n** è un numero intero non negativo

105

## Il calcolo del fattoriale

- Vediamo di capire cosa significa…

  0! = 1

  1! = 1(1-1)! = 1·0! = 1·1 = 1

  2! = 2(2-1)! = 2·1! = 2·1 = 2

  3! = 3(3-1)! = 3·2! = 3·2·1 = 6

  4! = 4(4-1)! = 4·3! = 4·3·2·1 = 24

  5! = 5(5-1)! = 5·4! = 5·4·3·2·1 = 120
- Quindi, per ogni **n** intero positivo, il fattoriale di **n** è il prodotto dei primi **n** numeri interi positivi

106

## Il calcolo del fattoriale

- Scriviamo un metodo statico per calcolare il fattoriale

```
public static int factorial(int n)
{
   if (n < 0)
      throw new IllegalArgumentException();

   if (n == 0)
      return 1;

   int prod = 1;
   for (int i = 2; i <= n; i++)
       prod = prod * i;

   return prod;
}
```

107

## Il calcolo del fattoriale

- Fin qui, nulla di nuovo… però abbiamo dovuto fare un'analisi matematica della definizione per scrivere l'algoritmo
- Realizzando direttamente la definizione, sarebbe stato più naturale scrivere

```
public static int factorial(int n)
{  if (n < 0)
      throw new IllegalArgumentException();

   if (n == 0)
      return 1;

   return n * factorial(n - 1);
}
```

108

## Il calcolo del fattoriale

```
public static int factorial(int n)
{  if (n < 0)
      throw new IllegalArgumentException();
   if (n == 0)
      return 1;

   return n * factorial(n - 1);
}
```

- Si potrebbe pensare: "Non è possibile ***invocare un metodo*** mentre si esegue ***il metodo stesso***!"
- Invece, come è facile verificare scrivendo un programma che usi **factorial()**, questo è lecito in Java, così come è lecito in quasi tutti i linguaggi di programmazione

109

## La ricorsione

- ***Invocare un metodo*** mentre si esegue ***lo stesso metodo*** è un paradigma di programmazione che si chiama

  **ricorsione**

  e un metodo che ne faccia uso si chiama

  **metodo ricorsivo**
- La ricorsione è uno strumento molto potente per realizzare alcuni algoritmi, ma è anche fonte di molti errori di difficile diagnosi

110

## La ricorsione

- Per capire come utilizzare correttamente ***la ricorsione***, vediamo innanzitutto ***come funziona***
- Quando un metodo ricorsivo invoca se stesso, ***la macchina virtuale Java esegue le stesse azioni che vengono eseguite quando viene invocato un metodo qualsiasi***
  - sospende l'esecuzione del metodo invocante
  - esegue il metodo invocato fino alla sua terminazione
  - riprende l'esecuzione del metodo invocante dal punto in cui era stata sospesa

111

## La ricorsione

- Vediamo la sequenza usata per calcolare 3!

  factorial(3) invoca factorial(2)
  factorial(2) invoca factorial(1)
  factorial(1) invoca factorial(0)
  factorial(0) restituisce 1
  factorial(1) restituisce 1
  factorial(2) restituisce 2
  factorial(3) restituisce 6
- Si crea una lista di metodi "in attesa", che prima si allunga e poi si accorcia fino ad estinguersi

112

## La ricorsione: caso base

- **Prima regola**
  - il metodo ricorsivo ***deve fornire la soluzione del problema in almeno un caso particolare, senza ricorrere a una chiamata ricorsiva***
  - tale caso si chiama ***caso base*** della ricorsione
  - nel nostro esempio, il caso base è

    ```
    if (n == 0)
       return 1;
    ```
  - a volte ci sono più casi base, non è necessario che sia unico

113

## La ricorsione: passo ricorsivo

- **Seconda regola**
  - il metodo ricorsivo ***deve effettuare la chiamata ricorsiva dopo aver semplificato il problema***
  - nel nostro esempio, per il calcolo del fattoriale di **n** si invoca la funzione ricorsivamente per conoscere il fattoriale di **n-1**, cioè per risolvere un problema più semplice

    ```
    if (n > 0)
       return n * factorial(n - 1);
    ```
  - il concetto di "problema più semplice" varia di volta in volta: in generale, ***bisogna tendere a un caso base***

114

## La ricorsione: un algoritmo?

- Le regole appena viste sono fondamentali per poter dimostrare che la soluzione ricorsiva di un problema sia un algoritmo
  - in particolare, che arrivi a conclusione in un numero **finito** di passi
- Si potrebbe pensare che le chiamate ricorsive si possano succedere una dopo l'altra, all'infinito

115

## La ricorsione: un algoritmo?

- Invece, se
  - a ogni invocazione il problema diventa sempre più semplice e si avvicina al caso base
  - la soluzione del caso base non richiede ricorsione

allora certamente la soluzione viene calcolata in un numero finito di passi, per quanto complesso possa essere il problema

116

## Ricorsione infinita

- Non tutti i metodi ricorsivi realizzano algoritmi
  - ***se manca il caso base***, il metodo ricorsivo continua a invocare se stesso all'infinito
  - ***se il problema non viene semplificato a ogni invocazione ricorsiva***, il metodo ricorsivo continua ad invocare se stesso all'infinito
- Dato che la lista dei metodi "in attesa" si allunga indefinitamente, l'ambiente runtime esaurisce la memoria disponibile per tenere traccia di questa lista e il programma termina con un errore

117

## Ricorsione infinita

- Provare a eseguire un programma che presenta ricorsione infinita

```
public class InfiniteRecursion
{  public static void main(String[] args)
   {  main(args);
   }
}
```

- Il programma terminerà con la segnalazione dell'eccezione **java.lang.StackOverflowError**
  - il ***runtime stack*** (o java stack) è la struttura dati all'interno dell'interprete Java che gestisce le invocazioni in attesa
  - ***overflow*** significa ***trabocco***

118

## La ricorsione in coda

- Esistono diversi tipi di ricorsione
- Il modo visto fino a ora si chiama **ricorsione in coda** (***tail recursion***)
  - il metodo ricorsivo esegue ***una sola invocazione ricorsiva*** e tale invocazione è l'***ultima azione*** del metodo

```
public void tail(...)
{  ...
   tail(...);
}
```

119

## Eliminazione della ricorsione in coda

- La ricorsione in coda può sempre essere agevolmente ***eliminata***, trasformando il metodo ricorsivo in un metodo che usa un *ciclo*

```
public static int factorial(int n)
{  if (n == 0) return 1;
   else return n * factorial(n - 1);
}
```

```
public static int factorial(int n)
{  int k = n;
   int p = 1;
   while (k > 0)
   {  p = p * k;
      k--;
   }
   return p;
}
```

120

## La ricorsione in coda

- Allora, a cosa serve la ricorsione in coda?
- Non è necessaria, però in alcuni casi rende il codice più leggibile
- È utile quando la soluzione del problema è esplicitamente ricorsiva (es. fattoriale)
- In ogni caso, la ricorsione in coda è ***meno efficiente*** del ciclo equivalente, perché il sistema deve gestire le invocazioni sospese

121

## La ricorsione non in coda

- Se la ricorsione non è in coda, non è facile eliminarla (cioè scrivere codice non ricorsivo equivalente), però si può dimostrare che ciò è sempre possibile
  - deve essere così, perché il processore esegue istruzioni in sequenza e non può tenere istruzioni in attesa… quindi l'interprete deve farsi carico di eliminare la ricorsione (usando il runtime stack)

122

## La ricorsione multipla

- Si parla di ricorsione multipla quando un metodo invoca se stesso più volte durante la sua esecuzione
  - la ricorsione multipla è ancora più difficile da eliminare, ma è sempre possibile
- Esempio: il calcolo dei numeri di Fibonacci

$$\text{Fib}(n) = \begin{cases} n & \text{se } 0 \le n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{se } n \ge 2 \end{cases}$$

123

## I numeri di Fibonacci

```
public static int fib(int n)
{  if (n < 0)
      throw new IllegalArgumentException();
   else if (n < 2)
      return n;
   else
      return fib(n-2) + fib(n-1);
}
```

- Il metodo `fib()` realizza una ricorsione multipla

124

## La ricorsione multipla

- La ricorsione multipla va usata con molta attenzione, perché può portare a programmi molto inefficienti
- Eseguendo il calcolo dei numeri di Fibonacci di ordine crescente...
  - … si nota che il tempo di elaborazione cresce MOLTO rapidamente… quasi 3 milioni di invocazioni per calcolare fib(31) !!!!
  - più avanti quantificheremo questo problema

125

## Esempio di Ricorsione: Torri di Hanoi

126

## Torri di Hanoi: regole

- Il rompicapo è costituito da tre pile di dischi ("torri") allineate
  - all'inizio tutti i dischi si trovano sulla pila di sinistra
  - alla fine tutti i dischi si devono trovare sulla pila di destra
- I dischi sono tutti di dimensioni diverse e quando si trovano su una pila devono rispettare la seguente regola
  - ***nessun disco può avere sopra di sé dischi più grandi***

127

## Torri di Hanoi: regole

Situazione iniziale Situazione finale

128

## Torri di Hanoi: regole

- Per risolvere il rompicapo bisogna ***spostare un disco alla volta***
  - un disco può essere rimosso dalla cima della torre e inserito in cima a un'altra torre
    - ✓non può essere "parcheggiato" all'esterno…
  - in ogni momento deve essere rispettata la regola vista in precedenza
    - ✓***nessun disco può avere sopra di sé dischi più grandi***

129

## Algoritmo di soluzione

- Per il rompicapo delle Torri di Hanoi è noto un algoritmo di soluzione ricorsivo
- Il problema generale consiste nello spostare **n** dischi da una torre a un'altra, usando la terza torre come deposito temporaneo
- Per spostare **n** dischi da una torre all'altra si suppone di saper spostare **n-1** dischi da una torre all'altra, come sempre si fa nella ricorsione
- Per descrivere l'algoritmo identifichiamo le torri con i numeri interi 1, 2 e 3

130

## Algoritmo ricorsivo

- Per spostare **n** dischi dalla torre 1 alla torre 3
  1. gli **n-1** dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)
  2. il disco rimasto nella torre 1 viene portato nella torre 3
  3. gli **n-1** dischi in cima alla torre 2 vengono spostati sulla torre 3, usando la torre 1 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 1 rimane vuota)
- Il **caso base** si ha quando **n** vale 1 e l'algoritmo usa il solo passo 2, che non ha chiamate ricorsive

131

## Algoritmo ricorsivo

- Si può dimostrare che il numero di mosse necessarie per risolvere il rompicapo con l'algoritmo proposto è

$$2^n - 1$$

- con n pari al numero di dischi

132

## Soluzione delle Torri di Hanoi

```
public class Hanoi
{ public static void main(String[] args)
  {
    int n = Integer.parseInt(args[0]);
    solveHanoi(1, 3, 2, n);
  }

  private static void solveHanoi(int from,
                    int to, int temp, int dim)
  {
     if (dim <= 0)   // caso base
       return;

     solveHanoi(from, temp, to, dim - 1);
     System.out.println(
           "Sposta da " + from + " a " + to);
     solveHanoi(temp, to, from, dim - 1);
  }
}
```

133

## Torri di Hanoi: n=2

```
main() chiama solveHanoi(1,3,2,2)
solveHanoi(1,3,2,2) chiama solveHanoi(1,2,3,1)
   solveHanoi(1,2,3,1) chiama solveHanoi(1,3,2,0)
   solveHanoi(1,2,3,1) stampa: Sposta da 1 a 2
   solveHanoi(1,2,3,1) chiama solveHanoi(3,2,1,0)
solveHanoi(1,3,2,2) stampa: Sposta da 1 a 3
solveHanoi(1,3,2,2) chiama solveHanoi(2,3,1,1)
   solveHanoi(2,3,1,1) chiama solveHanoi(2,1,3,0)
   solveHanoi(2,3,1,1) stampa: Sposta da 2 a 3
   solveHanoi(2,3,1,1) chiama solveHanoi(1,3,2,0)
```

134

## Quando finirà il mondo?

135

## Torri di Hanoi: la leggenda

- Una leggenda narra che alcuni monaci buddisti in un tempio dell'Estremo Oriente siano da sempre impegnati nella soluzione del rompicapo, spostando fisicamente i loro **64** dischi da una torre all'altra, consapevoli che quando avranno terminato il mondo finirà
- Sono necessarie $2^{64}-1$ mosse, che sono circa 16 miliardi di miliardi di mosse… cioè circa $1.6\times10^{18}$
- Supponendo che i monaci facciamo una mossa ogni minuto, essi fanno circa 500000 mosse all'anno, quindi il mondo finirà tra circa 30 mila miliardi di anni…
- Un processore a 1GHz che fa una mossa a ogni intervallo di clock (un miliardo di mosse al secondo…) impiega 16 miliardi di secondi, che sono circa 500 anni...

136