File e flussi (capitolo 11)

Leggere/scrivere file di testo

Gestione di file in Java

- Finora abbiamo visto programmi Java che interagiscono con l'utente soltanto tramite i **flussi di ingresso e di uscita standard**
- Ci chiediamo: è possibile leggere e scrivere file **all'interno** di un programma Java?
- Ci interessa soprattutto affrontare il problema della ***gestione di file di testo*** (file contenenti caratteri)
  - esistono anche i ***file binari***, che contengono semplicemente configurazioni di bit che rappresentano qualsiasi tipo di dati
- La gestione dei file avviene interagendo con il sistema operativo mediante classi del pacchetto **java.io** della libreria standard

Leggere un file di testo

- Il modo più semplice:
  - Creare un oggetto "**lettore di file**" (**FileReader**);
  - Creare un oggetto **Scanner**, che già conosciamo
  - **Collegare** l'oggetto **Scanner** al lettore di file invece che all'input standard

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
```

- In questo modo possiamo usare i consueti metodi di **Scanner** (**next**, **nextLine**, ecc.) per leggere i dati contenuti nel file

La classe FileReader

- 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"
- ***Attenzione***: se il file non esiste, viene lanciata l'eccezione **FileNotFoundException** a gestione obbligatoria

Leggere file con FileReader

- 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
   //... elaborazione di c
} // il metodo puo` lanciare IOException, da gestire!!
```

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

Leggere file con Scanner

- È più comodo "**avvolgere**" l'oggetto **FileReader** in un oggetto di tipo **Scanner**
  - Si può leggere **una riga alla volta** usando il metodo **nextLine** di **Scanner**

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
while(in.hasNextLine())
{   String line = in.nextLine();
    //... elaborazione della stringa
} // il costruttore FileReader lancia IOException, da gestire!!
```

  - Quando il file finisce, il metodo **hasNextLine** di **Scanner** restituisce un valore false

Chiudere file in lettura

- 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

Scrivere su un file di testo

- Il modo più semplice:
  - Creare un oggetto "scrittore" **PrintWriter**
  - **Collegare** l'oggetto **PrintWriter** ad un file

```
PrintWriter out = new PrintWriter("output.txt");
```

  - In questo modo possiamo usare i metodi **print**, **println**, ecc. per leggere i dati contenuti nel file
- ***Attenzione***: se *output.txt* non esiste viene **creato**. Ma se esiste già viene **svuotato** prima di essere scritto
  - 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);
```


Scrivere su un file di testo

- ***Attenzione***: bisogna sempre **chiudere** un oggetto **PrintWriter** dopo avere terminato di usarlo

```
PrintWriter out = new PrintWriter("output.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**

Rilevare la fine dell'input

- Molti problemi di elaborazione richiedono la ***lettura di una sequenza di dati in ingresso***
  - ad esempio, calcolare la somma di numeri in virgola mobile, ogni numero inserito su una riga diversa
- Altrettanto spesso il programmatore non sa ***quanti saranno*** i dati forniti in ingresso dall'utente
- **Nuovo Problema**: leggere una sequenza di dati in ingresso ***finché i dati non sono finiti***
  - In particolare questo succede quando leggiamo dati da un file
- **Possibile soluzione**:
  - **Scanner** possiede i metodi **hasNext**, **hasNextLine**, ecc., che restituiscono false se l'input è **terminato**

Rilevare la fine dell'input

- Usiamo il metodo predicativo **hasNextLine** come condizione di uscita dal ciclo di input

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
// ma anche Scanner in = new Scanner(System.in);

while (in.hasNextLine())
{
  String line = in.nextLine();
      ... // elabora line
}
```

Esempio: LineNumberer.java

```
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LineNumberer
{
   public static void main(String[] args)
   {
      Scanner console = new Scanner(System.in);
      System.out.print("Input file: ");
      String inputFileName = console.next();
      System.out.print("Output file: ");
      String outputFileName = console.next();

      try
      {
                                                  //(continua)
```


Esempio: LineNumberer.java

```
         FileReader reader = new FileReader(inputFileName);
         Scanner in = new Scanner(reader);
         PrintWriter out = new PrintWriter(outputFileName);
         int lineNumber = 1;

         while (in.hasNextLine())
         {
            String line = in.nextLine();
            out.println("/* " + lineNumber + " */ " + line);
            lineNumber++;
         }
         out.close();
      }
      catch (IOException e) //gestione obbligatoria...
      {
         System.out.println("Error processing file:" + e);
      }
   }
}
```


È tutto chiaro? …

1. Cosa succede se al programma LineNumberer viene fornito lo stesso nome per file in ingresso e in uscita?
2. Cosa succede se al programma LineNumberer viene fornito un nome di file di ingresso inesistente?

Rilevare la fine dell'input da tastiera

- Dopo aver iniziato l'esecuzione di un programma, si introducono da tastiera i dati che si vogliono elaborare
- Al termine dell'elenco, occorre ***comunicare al sistema operativo*** che l'input da console, destinato al programma in esecuzione, è terminato
  - in una finestra DOS/Windows bisogna digitare **Ctrl+Z**
  - in una ***shell*** di Unix bisogna digitare **Ctrl+D**
- Il flusso **System.in** di Java non leggerà questi caratteri speciali, ma riceverà dal sistema operativo la segnalazione che l'input è terminato, e **hasNextLine** restituirà un valore **false**

Esempio: calcolare la somma di numeri

```
import java.util.Scanner;

public class SumTester
{  public static void main(String[] args)
   {   Scanner in = new Scanner(System.in);
       double sum = 0;
       boolean done = false;
       while (!done)
       {
         String line;
         /* attenzione a questa condizione: stiamo usando la
            valutazione pigra e stiamo assegnando un nuovo
            valore a line */
         if (!in.hasNextLine() ||
             (line = in.nextLine()).equalsIgnoreCase("Q"))
            done = true;
         else
            sum = sum + Double.parseDouble(line);
       }
       System.out.println("Somma: " + sum);
   }
}
```


Ancora elaborazione dell'input

- Scomposizione di stringhe in "token"
- redirezione, piping
- Il flusso di errore standard

Scomposizione di stringhe

Scomposizione di stringhe

- Tutti gli esempi visti fino ad ora prevedevano l'inserimento dei ***dati in ingresso uno per riga***, ma spesso è più comodo o più naturale per l'utente inserire **più dati per riga**
  - ad esempio, cognome dello studente e voto
- Dato che **nextLine** legge un'intera riga, bisogna imparare ad estrarre le sottostringhe relative ai singoli dati che compongono la riga
  - non si può usare **substring**, perché in generale non sono note la lunghezza e la posizione di inizio dei singoli dati nella riga

Scomposizione di stringhe

- Per la scomposizione di stringhe in sottostringhe delimitate da spazi, è di nuovo molto utile la classe **Scanner**
  - una ***sottostringa con caratteristiche sintattiche ben definite*** (ad esempio, delimitata da spazi…) si chiama ***token***
  - **Scanner** considera come delimitatori di sottostringhe gli spazi, i caratteri da tabulazione e i caratteri di "andata a capo"

Scomposizione di stringhe

- Per scomporre una stringa in token usando **Scanner**, innanzitutto bisogna creare un oggetto della classe fornendo **la stringa** come parametro al costruttore

```
Scanner in = new Scanner(System.in);
String line = in.nextLine();
Scanner t = new Scanner(line);
```

- Successive invocazioni del metodo **next** restituiscono successive sottostringhe, fin quando l'invocazione di **hasNext** restituisce **true**

```
while (t.hasNext())
{  String token = t.next();
   // elabora token
}
```


Esempio: contare parole di un testo

```
import java.util.Scanner;

public class WordCounter
{
  public static void main(String[] args)
  {
    Scanner in = new Scanner(System.in);
    int count = 0;
    boolean done = false;
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        Scanner t = new Scanner(line);
        while (t.hasNext())
        {
           t.next(); // non devo elaborare
           count++;
        }
    }
    System.out.println(count + " parole");
  }
}
```

Riassunto: elaborare input in java

- Principali classi e metodi utilizzabili
  - **Scanner** e **nextLine()** per leggere intere righe di input
    - **FileReader** per gestire input da file
  - **Scanner** e **next** per scomporre le righe in parole
  - **Integer.parseInt()** e **Double.parseDouble()** per convertire stringhe in numeri
- Situazioni da gestire
  - **Terminazione di input**: metodo **hasNextLine()** di **Scanner**, e/o uso di caratteri sentinella (ad es. Q)
  - **IOException** (da gestire **obbligatoriamente**) se lavoriamo con file
  - **NumberFormatException** (gestione opzionale) lanciate da **Integer.parseInt()** e **Double.parseDouble()**

Reindirizzamento di input e output, canalizzazioni ("pipes")

Reindirizzamento di input e output

- Usando i programmi scritti finora si inseriscono dei dati da tastiera, che al termine non vengono memorizzati
  - ***per elaborare una serie di stringhe bisogna inserirle tutte, ma non ne rimane traccia!***
- Una soluzione "logica" sarebbe che ***il programma leggesse le stringhe da un file***
  - questo si può fare con il **reindirizzamento dell'input standard**, consentito da quasi tutti i sistemi operativi

Reindirizzamento di input e output

```
//classe che ripete a pappagallo l'input inserito, un token a riga
import java.util.Scanner;
public class Pappagaller
{   public static void main(String[] args)
    {   Scanner in = new Scanner(System.in);
        while(in.hasNext())
           System.out.println(in.next());
    }
}
```

- Il reindirizzamento dell'input standard, sia in sistemi Unix che nei sistemi MS Windows, si indica con il carattere **<** seguito dal **nome del file da cui ricevere l'input**

```
java Pappagaller < testo.txt
```

  - Il file **testo.txt** viene ***collegato*** all'input standard
- Il programma non ha bisogno di alcuna istruzione particolare, semplicemente **System.in** non sarà più collegato alla tastiera ma al file specificato

Reindirizzamento di input e output

- A volte è comodo anche il reindirizzamento dell'**output**
  - ad esempio, quando il programma produce molte righe di output, che altrimenti scorrono velocemente sullo schermo senza poter essere lette

```
java Pappagaller > output.txt
```

- I due reindirizzamenti possono anche essere **combinati**

```
java Pappagaller < testo.txt > output.txt
```


Canalizzazioni ("pipes")

- Supponiamo di dovere ulteriormente elaborare l'output prodotto da un programma
- Ad esempio, una elaborazione molto comune consiste nell'***ordinare*** le parole
  - questa elaborazione è cosi` comune che moltii sistemi operativi hanno un programma **sort**, che riceve da standard input un insieme di stringhe (una per riga) e le stampa a standard output ordinate lessicograficamente (una per riga)
  - Per ottenere le parole di **testo.txt** una per riga e ordinate, abbiamo bisogno di un ***file temporaneo*** (ad es. **temp.txt**) che ***serve solo a memorizzare il risultato intermedio***, prodotto dal primo programma e utilizzato dal secondo

```
java Pappagaller < testo.txt > temp.txt
sort < temp.txt > testoOrdinato.txt
```

Canalizzazioni ("pipes")

- Per ottenere le parole di **testo.txt** una per riga e ordinate, abbiamo bisogno di un ***file temporaneo***

```
java Pappagaller < testo.txt > temp.txt
sort < temp.txt > testoOrdinato.txt
```

- Il file temporaneo **temp.txt** ***serve soltanto per memorizzare il risultato intermedio***, prodotto dal primo programma e utilizzato dal secondo
- Questa situazione è talmente comune che quasi tutti i sistemi operativi offrono un'alternativa

Canalizzazioni ("pipes")

- Anziché utilizzare un file temporaneo per memorizzare l'output prodotto da un programma che deve servire da input per un altro programma, si può usare una ***canalizzazione*** ("***pipe***")

```
java Pappagaller < testo.txt | sort > testoOrdinato.txt
```

- La canalizzazione può anche ***prolungarsi***
  - Ad esempio, vogliamo sapere quante parole ci sono nel testo, senza contare le ripetizioni
  - Usiamo **sort** con l'opzione **-uf** (non ripete parole uguali, e non considera differenze tra maiuscole e minuscole)
  - Canalizziamo il risultato sul programma **wc** (word count)

```
java Pappagaller < testo.txt | sort | wc
```


Il flusso di errore standard

Il flusso di errore standard

- Abbiamo visto che un programma Java ha sempre due flussi ad esso collegati
  - il flusso di ingresso standard, **System.in**
  - il flusso di uscita standard, **System.out**

  che vengono forniti dal sistema operativo
- In realtà esiste un altro flusso, chiamato ***flusso di errore standard*** o ***standard error***, rappresentato dall'oggetto **System.err**
  - **System.err** è di tipo **PrintStream** come **System.out**

Il flusso di errore standard

- La differenza tra **System.out** e **System.err** è solo convenzionale
  - si usa **System.out** per comunicare all'utente i risultati dell'elaborazione o qualunque altro messaggio che sia previsto dal corretto e normale funzionamento del programma
  - si usa **System.err** per comunicare all'utente eventuali condizioni di errore (fatali o non fatali) che si siano verificate durante il funzionamento del programma

Il flusso di errore standard

- In condizioni normali (cioè senza redirezione) lo **standard error** finisce sullo schermo insieme allo **standard output**
- In genere il sistema operativo consente di effettuare la redirezione dello standard error in modo **indipendente** dallo standard output
  - in Windows è possibile redirigere i due flussi verso due file distinti

```
C:\> java HelloTester > out.txt 2> err.txt
```

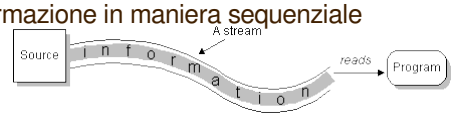
  - in Unix è (solitamente) possibile redirigere i due flussi verso due file distinti (la sintassi dipende dalla shell)

Approfondimento per gli interessati

Gestione di input/output nel linguaggio Java standard

Flussi di informazione

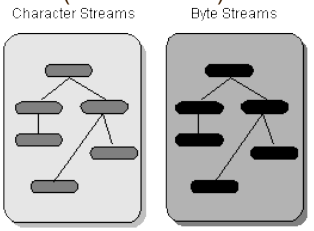
- Per ricevere informazione dall'esterno un programma
  - Apre un flusso su una sorgente di informazione (che può essere un file, la memoria, l'input standard…)
  - Legge l'informazione in maniera sequenziale



- Per spedire informazione verso l'esterno un programma
  - Apre un flusso su una destinazione (che può essere un file, la memoria, l'output standard…)
  - Scrive l'informazione in maniera sequenziale

Formato binario e di testo

- Java gestisce l'input usando due categorie di oggetti
  - Ovvero due distinte **gerarchie di classi**
- ***Flussi di input/output***
  - Le classi *astratte* **InputStream**, **OutputStream** e le loro sottoclassi gestiscono sequenze di byte (*formato binario*)
- ***Lettori e scrittori***
  - Le classi *astratte* **Reader**, **Writer**, e le loro sottoclassi gestiscono sequenze di caratteri (*formato testo*)
  - È possibile ***trasformare un flusso di input in un lettore***
  - E un ***flusso di output in uno scrittore***

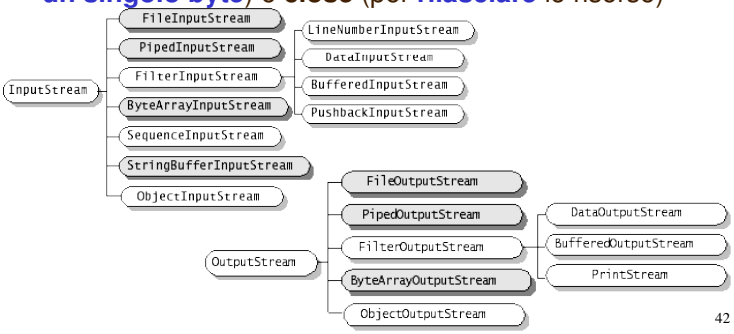


Formato testo e formato binario

- **Formato testo**:
  - I dati sono rappresentati come sequenze di **caratteri**
- **Formato binario**:
  - I dati sono rappresentati come sequenze di **byte**
- **Esempio**: il numero intero **12345**
  - In formato testo viene memorizzato come sequenza dei cinque caratteri **'1' '2' '3' '4' '5'**
  - In formato binario viene memorizzato come sequenze dei quattro byte **0 0 48 57**
    - $12345 = 2^{13} + 2^{12} + 2^5 + 2^4 + 2^3 + 2^0 =$
      $= (2^5 + 2^4) \times 2^8 + (2^5 + 2^4 + 2^3 + 2^0) = 48 \times 256 + 57$

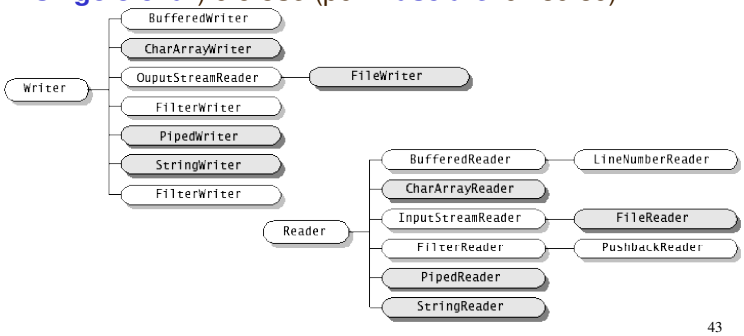
Formato binario

- Tutti i flussi di input hanno metodi **read** (per **leggere un singolo byte**) e close (per **rilasciare** le risorse)
- Tutti i flussi di output hanno metodi **write** (per **scrivere un singolo byte**) e **close** (per **rilasciare** le risorse)



Formato di testo

- Tutti i lettori hanno metodi **read** (per **leggere un singolo char**) e **close** (per **rilasciare** le risorse)
- Tutti gli scrittori hanno metodi **write** (per **scrivere un singolo char**) e **close** (per **rilasciare** le risorse)



Gestione dell'input standard

- Per usare l'oggetto **System.in** (di tipo InputStream), senza ricorrere a **Scanner**, dobbiamo risolvere tre problemi
  - **System.in** consente di leggere ***byte***, mentre noi abbiamo bisogno di leggere ***caratteri*** e ***righe*** di input
    - questo problema si risolve creando un oggetto della classe **BufferedReader** e invocando il suo metodo **readLine**
  - se il metodo **readLine** trova un ***errore*** durante la lettura dell'input, ***genera una eccezione***
    - dobbiamo dire al compilatore come ***gestire*** tale eccezione
  - il metodo **readLine** restituisce sempre una stringa
    - per leggere numeri dobbiamo convertire la stringa
    - sappiamo già fare…

Gestione dell'input standard

- Per ***trasformare un flusso di input in un lettore di caratteri*** si usa la classe **InputStreamReader**

```
InputStreamReader reader =
   new InputStreamReader(System.in);
```

- Oggetti di tipo **InputStreamReader** leggono caratteri
  - La classe ha solo il metodo **read** che legge ***un carattere alla volta*** (e restituisce **-1** se l'input è terminato)
  - Noi vogliamo leggere ***un'intera riga*** di input, fino al carattere **Invio**
  - Questo ***si può fare***, leggendo un carattere alla volta e poi componendo la stringa totale, ma ***è scomodo***

Leggere caratteri invece di byte

- La classe **BufferedReader** ***trasforma un lettore a caratteri singoli in un lettore con buffer*** ("*bufferizzato*"), che può leggere una riga per volta
  - Si dice che l'oggetto di tipo **BufferedReader** **avvolge** l'oggetto di tipo **InputStreamReader** (*wrapping*)

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader buffer = new BufferedReader(reader);
```

- **BufferedReader** ha un comodo metodo **readLine**
  - **readLine** restituisce **null** ***se i dati in ingresso sono terminati***

```
System.out.println("Inserire il nome");
String firstName = buffer.readLine();
```


Leggere caratteri invece di byte

- Riassumendo:

```
BufferedReader buffer = new BufferedReader(
                        new InputStreamReader(System.in));
System.out.println("Inserire il nome");
String firstName = buffer.readLine();
```

- L'oggetto di tipo **InputStreamReader** viene utilizzato soltanto per costruire l'oggetto di tipo **BufferedReader**, quindi può venire passato direttamente ***senza memorizzarlo in una variabile***

Esempio

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class MakePassword3
{  public static void main(String[] args)
   {  try {
         BufferedReader c = new BufferedReader(
            new InputStreamReader(System.in));
         System.out.println("Inserire il nome");
         String firstName = c.readLine();
         System.out.println("Inserire il cognome");
         String lastName = c.readLine();
         System.out.println("Inserire l'età");
         int age = Integer.parseInt(c.readLine());
                                          //(continua)
```

Esempio

```
                                                    //(continua)
            String initials = firstName.substring(0, 1)
               + lastName.substring(0, 1);
            String pw = initials.toLowerCase() + age;
            System.out.println("La password è " + pw);
        }
        catch (IOException e)
        {  System.out.println(e);
           System.exit(1);
        }
    }
}
```


Gestione di file nel linguaggio Java standard

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
  - se il file non esiste, viene lanciata l'eccezione **FileNotFoundException**, che deve essere obbligatoriamente gestita

Lettura di file di testo

- Con l'oggetto di tipo **FileReader** si può invocare il metodo **read**
  - che restituisce **un carattere** ad 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 si può tornare indietro e rileggere caratteri già letti
  - bisogna creare un nuovo oggetto di tipo **FileReader**

Lettura con buffer

- In alternativa, si può costruire un'esemplare di **BufferedReader** con cui leggere righe di testo da file
- **Problema**: leggere una sequenza di dati in ingresso ***finché i dati non sono finiti***
  - Il metodo **readLine** di **BufferedReader** restituisce **null** ***se i dati in ingresso sono terminati***

```
FileReader reader = new FileReader("file.txt");
BufferedReader bf = new BufferedReader(reader);
while(true)
{  String s = bf.readLine();
   if (s == null) break;
   ...  // elabora s
}
// il metodo lancia IOException, da gestire
```


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();
```

- Anche questo metodo 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

Scrittura di file di testo

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

```
FileWriter writer = new FileWriter("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***

Scrittura di file di testo

- L'oggetto di tipo **FileWriter** non ha i comodi metodi **print**/**println**
  - è utile creare un oggetto di tipo **PrintWriter** che **avvolge** l'esemplare di **FileWriter**, aggiungendo la possibilità di invocare **print**/**println** con qualsiasi argomento

```
FileWriter writer = new FileWriter("file.txt");
PrintWriter pw = new PrintWriter(writer);
pw.println("Ciao");
...
```


Scrittura di file di testo

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

```
FileWriter writer = new FileWriter("file.txt");
...
writer.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***

Array (capitolo 7)

Nota: la classe ArrayList trattata nel capitolo 8 del libro di testo non fa parte del nostro programma

Problema

- Scrivere un programma che
  - legge dallo standard input una sequenza di dieci numeri in virgola mobile, uno per riga
  - chiede all'utente un numero intero **index** e visualizza il numero che nella sequenza occupava la posizione indicata da **index**
- Occorre ***memorizzare tutti i valori della sequenza***
- Potremmo usare dieci variabili diverse per memorizzare i valori, selezionati poi con una lunga sequenza di alternative, ***ma se i valori dovessero essere mille?***

Memorizzare una serie di valori

- Lo strumento messo a disposizione dal linguaggio Java per memorizzare una sequenza di dati si chiama ***array*** (che significa "sequenza ordinata")
  - la struttura ***array*** esiste in quasi tutti i linguaggi di programmazione
- Un array in Java è ***un oggetto*** che realizza ***una raccolta di dati che siano tutti dello stesso tipo***
- Potremo avere quindi array di numeri interi, array di numeri in virgola mobile, array di stringhe, array di conti bancari...

Costruire un array

- Come ogni ***oggetto***, un array deve essere ***costruito*** con l'operatore **new**, dichiarando il ***tipo di dati*** che potrà contenere

```
new double[10];
```

- Il tipo di dati di un array può essere qualsiasi tipo di dati valido in Java
  - uno dei tipi di dati fondamentali o una classe

  e nella costruzione deve essere seguito da ***una coppia di parentesi quadre*** che contiene la ***dimensione*** dell'array, cioè il numero di elementi che potrà contenere

Riferimento ad un array

- Come succede con la costruzione di ogni oggetto, l'operatore **new** restituisce un ***riferimento*** all'array appena creato, che può essere memorizzato in una ***variabile oggetto*** dello stesso tipo

```
double[] values = new double[10];
```

  - **Attenzione**: nella definizione della variabile oggetto devono essere presenti le parentesi quadre, ma non deve essere indicata la dimensione dell'array; la variabile potrà riferirsi solo ad array di quel tipo, ma di qualunque dimensione

```
// si può fare in due passi
double[] values;
values = new double[10];
```


Utilizzare un array

- Al momento della costruzione, tutti gli elementi dell'array vengono inizializzati ad un valore, seguendo ***le stesse regole viste per le variabili di esemplare***
- Per ***accedere*** ad un elemento dell'array si usa

```
double[] values = new double[10];
double oneValue = values[3];
```

- La stessa sintassi si usa per ***modificare*** un elemento dell'array

```
double[] values = new double[10];
values[5] = 3.4;
```


Utilizzare un array

```
double[] values = new double[10];
double oneValue = values[3];
values[5] = 3.4;
```

- Il numero utilizzato per accedere ad un particolare elemento dell'array si chiama ***indice***
- L'indice può assumere un valore compreso tra **0** (***incluso***) e la **dimensione** dell'array (***esclusa***), cioè segue le stesse convenzioni viste per le posizioni dei caratteri in una stringa
  - il primo elemento ha indice 0
  - l'ultimo elemento ha indice (***dimensione*** - 1)

Utilizzare un array

- L'indice di un elemento di un array può, in generale, essere un'espressione con valore intero

```
double[] values = new double[10];
int a = 4;
values[a + 2] = 3.2; // modifica il
                     // settimo elemento
```

- Cosa succede se si accede ad un elemento dell'array con un indice sbagliato (maggiore o uguale alla dimensione, o negativo) ?
  - l'ambiente di esecuzione genera un'eccezione di tipo **ArrayIndexOutOfBoundsException**

La dimensione di un array

- Un array è un oggetto un po' strano…
  - non ha metodi pubblici, né statici né di esemplare
- L'unico elemento pubblico di un oggetto di tipo array è la sua dimensione, a cui si accede attraverso la sua variabile pubblica di esemplare **length** (attenzione, non è un metodo!)

```
double[] values = new double[10];
int a = values.length; // a vale 10
```

- Una variabile pubblica di esemplare sembrerebbe una violazione dell'incapsulamento...

La dimensione di un array

```
double[] values = new double[10];
values.length = 15; // ERRORE IN COMPILAZIONE
```

- In realtà, **length** è una variabile pubblica ma è dichiarata **final**, quindi ***non può essere modificata***, può soltanto essere ispezionata
- Questo paradigma è, in generale, considerato accettabile nell'OOP
- L'alternativa sarebbe stata fornire un metodo pubblico per accedere alla variabile privata
  - la soluzione scelta è meno elegante ma fornisce lo stesso livello di protezione dell'informazione ed è più veloce in esecuzione

Soluzione del problema iniziale

```
public class SelectValue
{  public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      double[] values = new double[10];
      for (int i = 0; i < values.length; i++)
         values[i] = in.nextDouble();
      System.out.println("Inserisci un numero:");
      int index = in.nextInt();
      if (index < 0 || index >= values.length)
         System.out.println("Valore errato");
      else
         System.out.println(values[index]);
   }
}
```


Costruzione di un array

- Sintassi: `new NomeTipo[lunghezza]`
- Scopo: costruire un array per contenere dati del tipo ***NomeTipo***; la ***lunghezza*** indica il numero di dati che saranno contenuti nell'array
- Nota: ***NomeTipo*** può essere uno dei tipi fondamentali di Java o il nome di una classe
- Nota: i singoli elementi dell'array vengono inizializzati con le stesse regole delle variabili di esemplare
  - **0** (zero) per variabili numeriche e caratteri
  - **false** per variabili booleane
  - **null** per variabili oggetto

Riferimento ad un array

- Sintassi:

```
NomeTipo[] nomeRiferimento;
```

- Scopo: definire la variabile ***nomeRiferimento*** come variabile oggetto che potrà contenere un riferimento ad un array di dati di tipo ***NomeTipo***
- Nota: le parentesi quadre **[ ]** sono necessarie e ***non*** devono contenere l'indicazione della dimensione dell'array

Accesso ad un elemento di un array

- Sintassi:

```
riferimentoArray[indice]
```

- Scopo: accedere all'elemento in posizione ***indice*** all'interno dell'array a cui ***riferimentoArray*** si riferisce, per conoscerne il valore o modificarlo
- Nota: il primo elemento dell'array ha indice 0, l'ultimo elemento ha indice (***dimensione*** - 1)
- Nota: se l'***indice*** non rispetta i vincoli, viene lanciata l'eccezione **ArrayIndexOutOfBoundsException**

Errori di limiti negli array

- Uno degli errori più comuni con gli array è l'utilizzo di un ***indice che non rispetta i vincoli***
  - il caso più comune è l'uso di un indice uguale alla dimensione dell'array, che è il primo indice non valido…

```
double[] values = new double[10];
values[10] = 2; // ERRORE IN ESECUZIONE
```

- Come abbiamo visto, l'ambiente runtime (cioè l'interprete Java) segnala questo errore con un'eccezione che arresta il programma

Inizializzazione di un array

- Quando si assegnano i valori agli elementi di un array si può procedere così

```
int[] primes = new int[3];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
```

ma se si conoscono tutti gli elementi da inserire si può usare questa sintassi (***migliore***)

```
int[] primes = { 2, 3, 5};
```

oppure (***accettabile, ma meno chiara***)

```
int[] primes = new int[] { 2, 3, 5};
```


Passare un array come parametro

- Spesso si scrivono metodi che ricevono array come parametri espliciti

```
public static double sum(double[] values)
{  if (values == null)
      throw new IllegalArgumentException();
   if (values.length == 0)
      return 0;
   double sum = 0;
   for (int i = 0; i < values.length; i++)
      sum = sum + values[i];
   return sum;
}
```


Usare array come valori di ritorno

- Un metodo può anche usare un array come **valore di ritorno**
  - Questo metodo restituisce un array contenente i dati dell'array **oldArray** e con lunghezza **newLength**

```
public static int[] resize(int[] oldArray, int newLength)
{  if (newLength < 0 || oldArray == null)
      throw new IllegalArgumentException();
   int[] newArray = new int[newLength];
   int count = oldArray.length;
   if (newLength < count)
      count = newLength;
   for (int i = 0; i < count; i++)
      newArray[i] = oldArray[i];
   return newArray;
}
```

```
int[] values = {1, 7, 4};
values = resize(values, 5);
values[4] = 9;
```


È tutto chiaro? …

1. Quali valori sono presenti nell'array dopo l'esecuzione delle istruzioni seguenti?

```
double[] data = new double[10]
for (int i = 0; i < data.length; i++)
   data[i] = i * i;
```

2. I seguenti enunciati sono corretti? Se sì, cosa visualizzano?

a)
```
double[] a = new double[10];
System.out.println(a[0]);
```

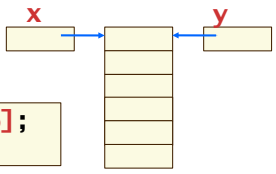
b)
```
double[] b = new double[10];
System.out.println(b[10]);
```

c)
```
double[] c;
System.out.println(c[0]);
```


Copiare array

Copiare un array

- Ricordando che una variabile che si riferisce ad un array è una variabile oggetto
  - contiene un riferimento all'oggetto array
- copiando il contenuto della variabile in un'altra ***non si copia l'array***, ma si ottiene un altro riferimento allo ***stesso oggetto array***




```
double[] x = new double[6];
double[] y = x;
```


Copiare un array

- Se si vuole ottenere ***una copia dell'array***, bisogna
  - ***creare un nuovo array dello stesso tipo e con la stessa dimensione***
  - ***copiare ogni elemento del primo array nel corrispondente elemento del secondo array***

```
double[] values = new double[10];
// inseriamo i dati nell'array
...
double[] otherValues = new double[values.length];
for (int i = 0; i < values.length; i++)
   otherValues[i] = values[i];
```


Copiare un array

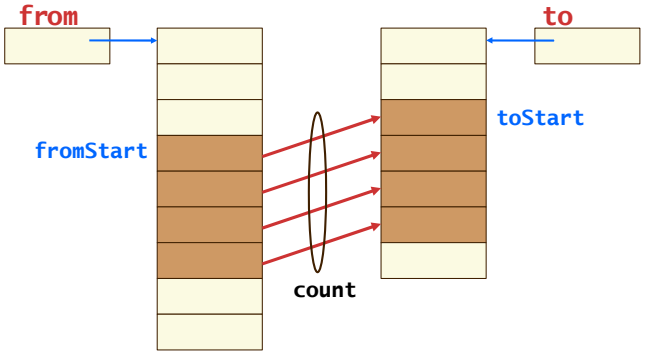
- Invece di usare un ciclo, è possibile (e ***più efficiente***) invocare il metodo statico **arraycopy** della classe **System** (nel pacchetto **java.lang**)

```
double[] values = new double[10];
// inseriamo i dati nell'array
...
double[] otherValues = new double[values.length];
System.arraycopy(values, 0, otherValues, 0,
                 values.length);
```

- Il metodo **System.arraycopy** consente di copiare un porzione di un array in un altro array (grande almeno quanto la porzione che si vuol copiare)

System.arraycopy

```
System.arraycopy(from,fromStart,to,toStart,count);
```



Copiare un array

- È anche possibile usare il metodo **clone**

```
double[] otherValues = (double[]) values.clone();
```

- **Attenzione**: il metodo **clone** restituisce un riferimento di tipo **Object**
  - È necessario effettuare un **cast** per ottenere un riferimento del tipo desiderato
    - In questo caso **double[]**

Array riempiti solo in parte (cfr. argomenti avanzati 8.4)

Array riempiti solo in parte

- Riprendiamo un problema già visto, rendendolo un po' più complesso
- Scrivere un programma che
  - legge da standard input una sequenza di numeri int, uno per riga, **finché i dati non sono finiti**
  - chiede all'utente un numero intero **index** e visualizza il numero che nella sequenza occupava la posizione indicata da **index**
- La differenza rispetto al caso precedente è che ora **non sappiamo quanti saranno i dati** introdotti dall'utente

Array riempiti solo in parte

- **Problema**: se creiamo un array che contenga i numeri double, è necessario **indicarne la dimensione**, che è una sua proprietà **final**
  - gli array in Java **non possono crescere**!
- **Soluzione**: costruire un array di dimensioni **sufficientemente grandi** da poter accogliere una sequenza di dati di lunghezza "**ragionevole**", cioè tipica per il problema in esame
- **Nuovo Problema**: al termine dell'inserimento dei dati, in generale non tutto l'array conterrà dati validi
  - è necessario **tenere traccia** di quale sia **l'ultimo indice** nell'array che contiene **dati validi**

Array riempiti solo in parte

```
import java.util.Scanner;

public class SelectValue2
{  public static void main(String[] args)
   {
      final int ARRAY_LENGTH = 1000;
      int[] values = new int[ARRAY_LENGTH];
      Scanner in = new Scanner(System.in);
      int valuesSize = 0;
      boolean done = false;
      while (!done)
      {  String s = in.next();
         if (s.equalsIgnoreCase("Q"))
            done = true;
         else
         { values[valuesSize] = Integer.parseInt(s);
           valuesSize++;
         } // valuesSize è l'indice del primo dato non valido
      }
      System.out.println("Inserisci un numero:");
      int index = Integer.parseInt(in.next());
      if (index < 0 || index >= valuesSize)
         System.out.println("Valore errato");
      else
         System.out.println(values[index]);
   }
}
```

Array riempiti solo in parte

- **values.length** è il numero di valori ***memorizzabili***, **valuesSize** è il numero di valori ***memorizzati***
- Questa soluzione ha però ancora una debolezza
  - Se la ***previsione*** del programmatore sul numero massimo di dati inseriti dall'utente è sbagliata, il programma si arresta con un'eccezione di tipo **ArrayIndexOutOfBoundsException**
- Ci sono due possibili soluzioni
  1. **impedire l'inserimento** di troppi dati, segnalando l'errore all'utente
  2. **ingrandire l'array** quando ce n'è bisogno

Array riempiti solo in parte

- **Soluzione 1**: impedire l'inserimento di troppi dati

```
...
final int ARRAY_LENGTH = 1000;
int[] values = new int[ARRAY_LENGTH];
Scanner in = new Scanner(System.in);
int valuesSize = 0;
done = false;
while(!done)
{  String s = in.nextLine();
   if (s.equalsIgnoreCase("Q"))
      done = true;
   else if (valuesSize == values.length)
   {  System.out.println("Troppi dati");
      done = true;
   }
   else
   {  values[valuesSize] = Integer.parseInt(s);
      valuesSize++;
   }
}
...
```


Array riempiti solo in parte

- **Soluzione 2**: cambiare dimensione all'array
  - si parla di ***array dinamico***
  - Ma è ***impossibile*** modificare la dimensione di un array...

```
...
final int ARRAY_LENGTH = 1000;
int[] values = new int[ARRAY_LENGTH];
Scanner in = new Scanner(System.in);
int valuesSize = 0;
done = false;
while(!done)
{  String s = in.nextLine();
   if (s.equalsIgnoreCase("Q"))
      done = true;
   else
   {   if (valuesSize == values.length)
         values = resize(values, valuesSize*2);
       values[valuesSize] = Integer.parseInt(s);
       valuesSize++;  }
}
...
```

Il metodo statico resize

- Restituisce un array di lunghezza **newLength** e contenente i dati dell'array **oldArray**
  - Crea un **nuovo** array più grande di quello "pieno" e copia in esso il contenuto del vecchio array
  - Useremo questo metodo **molto spesso**!

```
public static int[] resize(int[] oldArray, int newLength)
{  if (newLength < 0 || oldArray == null)
      throw new IllegalArgumentException();
   int[] newArray = new int[newLength];
   int count = oldArray.length;
   if (newLength < count)
      count = newLength;
   for (int i = 0; i < count; i++)
      newArray[i] = oldArray[i];
   return newArray;
}
```

```
int[] values = {1, 3, 7};
values = resize(values, 5);
```


Semplici algoritmi su array

La classe ArrayAlgs

- Costruiremo una classe **ArrayAlgs**
  - Sarà una "**classe di utilità**" (come la classe **Math**) che contiene una collezione di **metodi statici** che realizzano algoritmi per l'elaborazione di array
    - Per ora trattiamo array di numeri **interi**
    - Più avanti tratteremo array di **oggetti generici**

```
public class ArrayAlgs
{ ...
   public static int[] resize(int[] oldArray, int newLength)
   {  ... }

   public static  ...
}
```

```
//in un'altra classe i metodi verranno invocati cosi`
v = ArrayAlgs.resize(v,2*v.length);
```


Generare array di numeri casuali

- La classe **Math** ha il metodo **random( )** per generare sequenze di numeri pseudo-casuali
  - Una invocazione del metodo restituisce un numero **reale** pseudo-casuale nell'intervallo **[0, 1)**

```
double x =Math.random();
```

  - Per ottenere numeri **interi** casuali nell'intervallo **[a, b]**...

```
int n = (int)(a + (1+b-a)*Math.random());
```

- Usando **random** scriviamo nella classe **ArrayAlgs** un metodo che genera array di numeri interi casuali

```
public static int[] randomIntArray(int length, int n)
{
  int[] a = new int[length];
  for (int i = 0; i < a.length; i++)
    // a[i] e` un num intero casuale tra 0 e n-1 inclusi
    a[i] = (int) (n * Math.random());
  return a;
}
```


Convertire array in stringhe

- Se cerchiamo di stampare un array sullo standard output non riusciamo a visualizzarne il contenuto ...

```
int[] a = {1,2,3};
System.out.println(a);
```

→ `[I@10b62c9`

- Scriviamo nella classe **ArrayAlgs** un metodo che **crea una stringa** contenente gli elementi di un array

```
public static String printArray(int[] v, int vSize)
{    String s = "[";
     for (int i = 0; i<vSize; i++)
         s = s + v[i] + " ";
     s = s + "\b]";
     return s;
}
```

```
int[] a = {1,2,3};
int aSize = a.length;
System.out.println(
       ArrayAlgs.printArray(a,aSize));
```

→ `[1 2 3]`

Eliminazione/inserimento di elementi in un array

Eliminare un elemento di un array

- **Primo algoritmo**: se **l'ordine** tra gli elementi dell'array **non** è importante (cioè se l'array realizza il concetto astratto di insieme), è sufficiente
  - **copiare l'ultimo elemento** dell'array nella posizione dell'elemento da eliminare
  - **ridimensionare** l'array (oppure usare la tecnica degli array riempiti soltanto in parte)

```
public static void remove(int[] v, int vSize, int index)
{
    v[index] = v[vSize - 1];
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
ArrayAlgs.remove(a,aSize,1);
aSize--;
```

a diventa [1,5,3,4]

Eliminare un elemento di un array

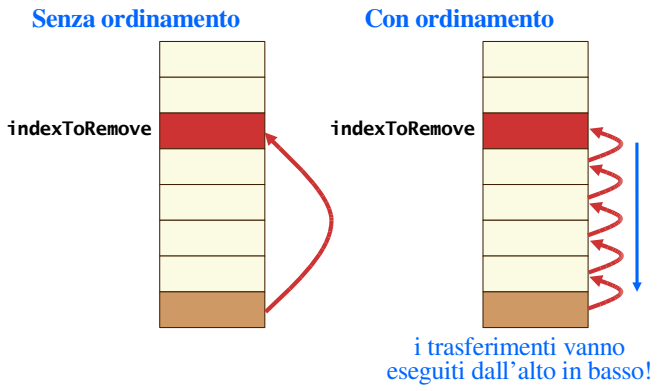
- ***Secondo algoritmo*** se **l'ordine** tra gli elementi dell'array deve essere mantenuto allora l'algoritmo è più complesso. Bisogna
  - **Spostare tutti gli elementi** dell'array successivi all'elemento da rimuovere nella posizione con indice immediatamente inferiore
  - **ridimensionare** l'array (oppure usare la tecnica degli array riempiti soltanto in parte)

```
public static void removeSorted(int[] v, int vSize, int index)
{    for (int i=index; i<vSize-1; i++)
        v[i] = v[i + 1];
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
ArrayAlgs.removeSorted(a,aSize,1);
aSize--;
```

a diventa [1,3,4,5]

Eliminare un elemento di un array



Inserire un elemento in un array

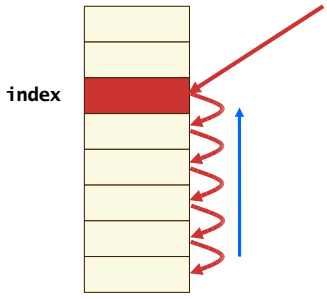
- **Algoritmo**: per inserire l'elemento nella posizione voluta, se non è la prima posizione libera, bisogna "**fargli spazio**". È necessario
  - **ridimensionare** l'array (oppure usare la tecnica degli array riempiti soltanto in parte)
  - **Spostare tutti gli elementi** dell'array successivi alla posizione di inserimento nella posizione con indice immediatamente superiore (a partire dall'ultimo)

```
public static int[] insert(int[] v,int vSize, int index,int val)
{    if (vSize == v.length)  v = resize(v, 2*v.length);
     for (int i = vSize; i > index; i--)
         v[i] = v[i - 1];
     v[index] = value;
     return v;
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
a = ArrayAlgs.insert(a,aSize,2,7);
aSize++;
```

a diventa [1,2,7,3,4,5]

Inserire un elemento in un array



Trovare un valore in un array

- **Algoritmo**: la strategia più semplice è chiamata **ricerca lineare**. Bisogna
  - **scorrere** gli elementi dell'array **finchè** l'elemento cercato non viene trovato oppure si raggiunge la fine dell'array
  - Nel caso in cui il valore cercato compaia più volte, questo algoritmo trova **soltanto** la prima occorrenza del valore e non le successive

```
public static int linearSearch(int[] v, int vSize, int value)
{
    for (int i = 0; i < vSize; i++)
        if (v[i] == value) return i; // trovato valore
    return -1;  // valore non trovato
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
int i = ArrayAlgs.linearSearch(a,aSize,4);
```

i vale 3

Trovare il valore massimo

- **Algoritmo**: è necessario esaminare **l'intero array**. Bisogna
  - **Inizializzare** il valore candidato con il primo elemento
  - **Confrontare** il candidato con gli elementi rimanenti
  - **Aggiornare** il valore candidato se viene trovato un valore maggiore

```
public static int findMax(int[] v, int vSize)
{   int max = v[0];
    for (int i = 1; i < vSize; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
int max = ArrayAlgs.findMax(a, aSize);
```

max vale 5

Trovare il valore minimo

- **Algoritmo** (del tutto analogo a quello per la ricerca del massimo): è necessario esaminare **l'intero array**. Bisogna
  - **Inizializzare** il valore candidato con il primo elemento
  - **Confrontare** il candidato con gli elementi rimanenti
  - **Aggiornare** il valore candidato se viene trovato un valore minore

```
public static int findMin(int[] v, int vSize)
{   int min = v[0];
    for (int i = 1; i < vSize; i++)
       if (v[i] < min)
           min = v[i];
    return min;
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
int min = ArrayAlgs.findMin(a, aSize);
```

min vale 1

Array bidimensionali

Array bidimensionali

- Rivediamo un problema già esaminato
  - 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 più generale, scrivendo metodi che possano ***elaborare un'intera struttura di questo tipo***

Matrici

- Una struttura di questo tipo, con dati organizzati in righe e colonne, si dice ***matrice*** o array bidimensionale

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

- Un elemento all'interno di una matrice è identificato da ***una coppia (ordinata) di indici***
  - un ***indice di riga***
  - un ***indice di colonna***
- In Java esistono gli **array bidimensionali**

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 ad un elemento di un array bidimensionale

```
powers[2][3] = 2;
```

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 riga…)


```
import java.util.Scanner;

/**
    Programma che visualizza una tabella con i valori
    delle potenze "x alla y", con x e y che variano
    indipendentemente tra 1 ed un valore massimo
    assegnato dall'utente.
    I dati relativi a ciascun valore di x compaiono
    su una riga, con y crescente da sinistra
    a destra e x crescente dall'alto in basso.
*/
public class TableOfPowers
{  public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      System.out.println(
         "Calcolo dei valori di x alla y");
      System.out.println("Valore massimo di x:");
      int maxX = in.nextInt();
      System.out.println("Valore massimo di y:");
      int maxY = in.nextInt();
      int maxValue =
            (int) Math.round(Math.pow(maxX, maxY));
      int columnWidth =
         1 + Integer.toString(maxValue).length();
      int[][] powers = generatePowers(maxX, maxY);
      printPowers(powers, columnWidth);
   }
                              //continua
```



```
                                          //continua
   /**
       Genera un array bidimensionale con i
       valori delle potenze di x alla y.
   */
   private static int[][] generatePowers(int x,
                                         int y)
   {  int[][] powers = new int[x][y];
      for (int i = 0; i < x; i++)
         for (int j = 0; j < y; j++)
            powers[i][j] =
               (int)Math.round(Math.pow(i + 1, j + 1));
      return powers;
   }
                                          //continua
```

- Notare l'utilizzo di metodi **private** per la scomposizione di un problema in sottoproblemi più semplici
  - in genere non serve preoccuparsi di pre-condizioni perché il metodo viene invocato da chi l'ha scritto


```
                                          //continua
   /**
       Visualizza un array bidimensionale di
       numeri interi con colonne di larghezza
       fissa e valori allineati a destra.
   */
   private static void printPowers(int[][] v,
                                   int width)
   {  for (int i = 0; i < v.length; i++)
      {  for (int j = 0; j < v[i].length; j++)
         {  String s = Integer.toString(v[i][j]);
            while (s.length() < width)
               s = " " + s;
            System.out.print(s);
         }
         System.out.println();
      }
   }
}
```


Argomenti sulla riga dei comandi

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
```

- Tali 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
```

Argomenti sulla riga di comandi

- Uso tipico degli argomenti sulla riga di comandi
  - Specificare **opzioni** e **nomi di file** da leggere/scrivere

```
java LineNumberer -c HelloWorld.java HelloWorld.txt
```

- Per convenzione le stringhe che iniziano con un **trattino** sono considerate opzioni

```
for (int i = 0; i< args.length; i++)
{ String a = args[i];
  if (a.startsWith("-")) // è un'opzione
  {
    if (a.equals("-c")) useCommentDelimiters = true;
  }
  else if (inputFileName == null) inputFileName = a;
  else if (outputFileName == null) outputFileName = a;
}
```


Esercizio: Array paralleli (cfr. Consigli per la Qualità 8.2)

Array paralleli

- Scriviamo un programma che riceve in ingresso un elenco di dati che rappresentano
  - i **cognomi** di un insieme di studenti
  - il **voto** della prova scritta
  - il **voto** della prova orale
- I dati di uno studente vengono inseriti in una riga separati da uno spazio
  - prima il cognome, poi il voto scritto, poi il voto orale
- I dati sono terminati da una **riga vuota**

Array paralleli

- Ora aggiungiamo le seguenti funzionalità
  - il programma chiede all'utente di inserire un comando per identificare l'elaborazione da svolgere
    - **Q** significa "**termina il programma**"
    - **S** significa "**visualizza la media dei voti di uno studente**"
  - Nel caso **S** il programma
    - chiede all'utente di **inserire il cognome** di uno studente
    - **Stampa il cognome** dello studente seguito dalla **media dei suoi voti**

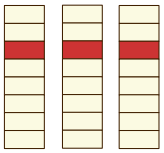

```
import java.util.StringTokenizer;
import java.util.Scanner;

public class StudentManager
{  public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      String[] names = new String[10];
      double[] wMarks = new double[10];
      double[] oMarks = new double[10];
      int count = 0; // array riempiti solo in parte
      boolean done = false;
      while (!done)
      {  String input = in.nextLine();
         if (input.length() == 0) done=true;
         else
         {
           StringTokenizer t = new StringTokenizer(input);
           if (count == names.length)
           {  names = resizeString(names, count * 2);
              wMarks = resizeDouble(wMarks, count * 2);
              oMarks = resizeDouble(oMarks, count * 2);
           }
           names[count] = t.nextToken();
           wMarks[count] = Double.parseDouble(t.nextToken());
           oMarks[count] = Double.parseDouble(t.nextToken());
           count++;
         }
      }
                                                    //continua
```



```
      done = false;                                 //continua
      while (!done)
      {
        System.out.println("Comando? (Q per uscire, S per vedere)");
        String command = in.nextLine();
        if (command.equalsIgnoreCase("Q"))
            done = true;
        else if (command.equalsIgnoreCase("S"))
        {
            System.out.println("Cognome?");
            String name = in.nextLine();
            printAverage(names, wMarks, oMarks,name, count); //NOTA:
            //non abbiamo gestito l'eccezione lanciata da printAverage
        }
        else
        {  System.out.println("Comando errato");
        }
      }
   }

   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);
      }
   }                                                //continua
```

```
                                                          //continua
   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;
   }

   private static String[] resizeString(String[] oldv, int newLength)
   {  if (newLength < 0 || oldv == null)
        throw new IllegalArgumentException();
      String[] newv = new String[newLength];
      int count = oldv.length;
      if (newLength < count) count = newLength;
      for (int i = 0; i < count; i++)
            newv[i] = oldv[i];
      return newv;
   }

   private static double[] resizeDouble(double[] oldv, int newLength)
   {  if (newLength < 0 || oldv == null)
        throw new IllegalArgumentException();
      double[] newv = new double[newLength];
      int count = oldv.length;
      if (newLength < count)  count = newLength;
      for (int i = 0; i < count; i++)
           newv[i] = oldv[i];
      return newv;
   }
}
```

Array paralleli



- L'esempio presentato usa una struttura dati denominata "***array paralleli***"
  - si usano ***diversi array*** per contenere i dati del problema, ma questi 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 operazioni hanno bisogno di ***usare tutti gli array***, che devono quindi essere passati come parametri
    - come nel caso di **printAverage**

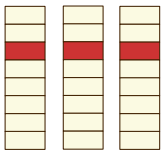
Array paralleli

- Array paralleli sono molto usate in linguaggi di programmazione **non OO**, ma presentano ***numerosi svantaggi*** che possono essere superati in 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!***

Array paralleli in OOP

- Le tecniche di ***OOP*** consentono di gestire molto più efficacemente le strutture dati di tipo "array paralleli"
  - Definire una classe che contenga tutte le informazioni relative ad "una fetta" degli array, cioè raccolga tutte le informazioni presenti nei diversi array in relazione ad un certo indice
  - Costruire un array di oggetti di questa classe

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

   ...
}
```



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
- È molto più facile scrivere il codice e, soprattutto, modificarlo
- Immaginiamo di introdurre un'altra caratteristica 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**