Es2: lettura ciclica input (calcolo valore medio e massimo di un insieme di valori)

```
 1  import java.util.Scanner;
 2
 3  public class DataAnalyzer
 4  {
 5     public static void main(String[] args)
 6     {
 7        Scanner in = new Scanner(System.in);
 8        DataSet data = new DataSet();
 9        boolean done = false;
10        while (!done)
11        {
12           System.out.print("Enter value, Q to quit: ");
13           String input = in.next();
14           if (input.equalsIgnoreCase("Q"))
15              done = true;
16           else
17           {
18              double x = Double.parseDouble(input);
19              data.add(x);
20           }
21        }
22        System.out.println("Average = " + data.getAverage());
23        System.out.println("Maximum = " + data.getMaximum());
24     }
25  }
```

Es2: lettura ciclica input (calcolo valore medio e massimo di un insieme di valori)

```
 1  public class DataSet
 2  {
 3     private double sum;
 4     private double maximum;
 5     private int count;
 6
 7     public DataSet()
 8     {
 9        sum = 0;
10        count = 0;
11        maximum = 0;
12     }
13
14     public void add(double x)
15     {
16        sum = sum + x;
17        if (count == 0 || maximum < x) maximum = x;
18        count++;
19     }
20
21     public double getAverage()
22     {
23        if (count == 0) return 0;
24        else return sum / count;
25     }
26
27     public double getMaximum()
28     {
29        return maximum;
30     }
31  }
```

Scandire i caratteri di una stringa

- **`s.charAt(i)`** è l'(i+1)-esimo carattere della stringa `s`

```
for(int i = 0; i < s.length(); i++)
{
   char c = s.charAt(i);
   …
}
```

Esempio: un programma che conta le vocali

- `s.indexOf(c)` è l'indice della posizione in cui `c` appare per la prima volta in `s`, o -1 se `c` non appare in `s`

```
int NumVocali = 0;
String vocali = "aeiou";
for (int i = 0; i < s.length(); i++)
{
  char c = s.charAt(i);
  if (vocali.indexOf(c) >= 0)
     NumVocali++;
}
```

Usi comuni dei Loop

- Contare quante lettere maiuscole ci sono in una stringa

```
int upperCaseLetters = 0;
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    if (Character.isUpperCase(ch))
    {
        upperCaseLetters++;
    }
}
```

Usi comuni dei Loop

- Trovare la prima lettera minuscola in una stringa

```
boolean found = false;
char ch = '?';
int position = 0;
while (!found && position < str.length())
{
    ch = str.charAt(position);
    if (Character.isLowerCase(ch))
        { found = true; }
    else { position++; }
}
```

Usi comuni dei Loop

```
boolean valid = false;
double input;
while (!valid)
{
   System.out.print("Please enter a positive
         value < 100: ");
   input = in.nextDouble();
   if (0 < input && input < 100)
         { valid = true; }
   else { System.out.println("Invalid input.");
  }
}
```

Valutazioni corto-circuito

- Le condizioni composte sono valutate da sinistra a destra
- La valutazione si interrompe quando il risultato è già noto
- Esempio:
  - `atBat!=0 && hits/atBat>0.300`
- Supponiamo **`atBat`** è zero:
  - `atBat!=0` è valutato a **`false`**
- A questo punto il risultato è noto: l'intera condizione composta sarà **`false`**

- La valutazione si arresta:
  - `hits/atBat>0.300` non sarà valutata !

Valutazioni corto-circuito: altri esempi

- Se si tratta di un OR, allora se il primo operando è **`true`**, il secondo non viene valutato:
  - `person.age()>=17 || person.accompaniedByAdult()`

- Un altro esempio (dall'esempio "estremi tra oggetti")
  - ```
    if (longest==null || s.length()>longest.length())
      longest = s;
    ```
    ```

L'istruzione **break**

- Singola keyword e un punto e virgola:
  break;
- Termina l'esecuzione di un ciclo immediatamente, esempio:

```
int k=0;
while (k!=5) {
  String s = infile.next();
  if (s == null)
        break;
  process s
  k++;
}
// k==5 or s==null
```

L'istruzione **`break`**: semplicemente evitarla

- Il ciclo precedente può essere espresso come:

```
int k=0;
String s = infile.read();
while (!(k==5 || s==null)) {
    process s
    k++;
    s = infile.read();
}
```

- Con il break:
  - La condizione di terminazione non è ovvia
  - Il lettore deve costruire mentalmente una condizione di terminazione in OR
- Senza break:
  - La condizione di terminazione è ovvia e si ottiene dalla condizione del **`while`**

L'istruzione `continue`

- Singola keyword ed un punto e virgola:

```
continue;
```

- Termina l'esecuzione corrente del corpo del ciclo

- Esempio:

```
for (i=0; i<100; i++) {
    if (i%2==1)
            continue;
    if (!veryComplicatedCondition(i))
            continue;
    process i
}
```

- Vantaggio(?): riduce i livelli di nesting e le indentazioni

L'istruzione `continue`: pericoli

- Consideriamo di voler leggere qualche stringa e elaborare solo quelle che soddisfano due condizioni:

```
String s = in.next();
while (s != null) {
    if (!condition1(s))
            continue;
    if (!condition2(s))
            continue;
    process s
    s = in.next();
}
```

- Cosa è sbagliato ?

Problema

- Vogliamo costruire una classe Dado che modelli un dado
- l'interfaccia pubblica deve contenere un metodo che simuli il lancio di un dado restituendo a caso il valore di una delle sue facce
- serve un generatore di numeri casuali

Numeri casuali

- La classe Random modella un generatore di numeri casuali
- **`Random generatore = new Random();`**
  - crea un generatore di numeri casuali
- **`int n = generatore.nextInt(a);`**
  - restituisce un intero n con 0 <= n < a
- **`double x = generatore.nextDouble();`**
  - restituisce un double x con 0 <= x < 1

Esempio uso di Random

```
import java.util.Random;
public class Dado {
//costruttore che costruisce un dado
//con s facce
 public Dado(int s) {
    facce = s;
    generatore = new Random();
 }
 public int lancia() {
    return 1 +
           generatore.nextInt(facce);
 }
 private Random generatore;
 private int facce;
}
```

```
// Questo programma simula 10 lanci
    del dado
public class TestaDado {
public static void main(String[] args)
{
  Dado d = new Dado(6);
  final int LANCI = 10;
  for (int i = 1; i <= LANCI; i++) {
     int n = d.lancia();
     System.out.print(n + " ");
  }
  System.out.println();
}
}
```

La classe File

- Modella path-name di file e directory
  - Costruttore:

    `File(String pathname)`

    crea un nuova istanza di oggetto **`File`** convertendo la stringa **`pathname`** in un nome di percorso astratto

Esempio

```
File add = new File("address.txt");
```

Scrivere in un File con PrintStream

Constructor Summary
PrintStream(File file) Creates a new print stream, without automatic line flushing, with the specified file.
PrintStream(File file, String csn) Creates a new print stream, without automatic line flushing, with the specified file and charset.
PrintStream(OutputStream out) Create a new print stream.
PrintStream(OutputStream out, boolean autoFlush) Create a new print stream.
PrintStream(OutputStream out, boolean autoFlush, String encoding) Create a new print stream.
PrintStream(String fileName) Creates a new print stream, without automatic line flushing, with the specified file name.
PrintStream(String fileName, String csn) Creates a new print stream, without automatic line flushing, with the specified file name and charset.

Scrivere in un File con PrintStream

```
import java.io.*;
public class WriteAddress {
   public static void main(String a[]) throws
   Exception{
        File usFile;
        PrintStream usPS;
        usFile = new File("address.txt");
        usPS = new PrintStream(usFile);
        usPS.println("Università di Salerno");
        usPS.println("Facoltà di Scienze");
        usPS.println("Via Giovanni Paolo II, 132");
        usPS.println("84084 Fisciano(SA), Italia");
   }
}
```

Leggere da un File con Scanner

Constructor Summary
Scanner(File source) Constructs a new Scanner that produces values scanned from the specified file.
Scanner(File source, String charsetName) Constructs a new Scanner that produces values scanned from the specified file.
Scanner(InputStream source) Constructs a new Scanner that produces values scanned from the specified input stream.
Scanner(InputStream source, String charsetName) Constructs a new Scanner that produces values scanned from the specified input stream.
Scanner(Readable source) Constructs a new Scanner that produces values scanned from the specified source.
Scanner(ReadableByteChannel source) Constructs a new Scanner that produces values scanned from the specified channel.
Scanner(ReadableByteChannel source, String charsetName) Constructs a new Scanner that produces values scanned from the specified channel.
Scanner(String source) Constructs a new Scanner that produces values scanned from the specified string.

Leggere da un File con Scanner

```
import java.io.*;
import java.util.Scanner;
class ReadAddress {
   public static void main(String a[]) throws
   Exception{
        File usFile;
        Scanner sf;
        usFile = new File("address.txt");
        sf = new Scanner(usFile);
        System.out.println(sf.nextLine());
   }
}
```

Libreria di canzoni (1)

- Problema
  - Una stazione radio vuole informatizzare la propria libreria di canzoni.
  - Si è creato un file in cui sono stati inseriti degli elementi composti dai titoli e dai compositori delle canzoni.
  - Si intende dare al disk-jockey la possibilità di cercare nella libreria tutte le canzoni di un particolare artista.

Libreria di canzoni (2)

- Scenario d'esempio

  Inserisci il nome del file della libreria di canzoni: ClassicRock.lib

  File ClassicRock.lib loaded.

  Inserisci l'artista da cercare: Beatles

  Canzoni dei Beatles trovate:

  Back in the USSR

  Paperback writer

  She Loves You

  Inserisci l'artista da cercare: Mozart

  Nessuna canzone di Mozart trovata

Determinare gli oggetti primari

- Nomi: song library, song, file, entry, title, artist
- Artist e title sono parti di song, che è sussidiaria di song library
- File e entry (in un file) rappresentano solo dati da leggere
- Classe primaria: **`SongLibrary`**

```
class SongLibrary {
   ...
}
```

Determinare il comportamento desiderato

- Capacità di creare una **`SongLibrary`**
  - Costruttore
- Necessità di cercare le canzoni di un artista
  - Un metodo **`lookUp`**

Definire l'interfaccia

- Tipico codice di utilizzo

```
SongLibrary classical = new SongLibrary("classical.lib");
SongLibrary jazz = new SongLibrary("jazz.lib");
classical.lookUp("Gould");
classical.lookUp("Marsalas");
jazz.lookUp("Corea");
jazz.lookUp("Marsalas");
```

- Abbiamo bisogno della seguente interfaccia

```
class SongLibrary {
   public SongLibrary(String songFileName) {...}
   void lookUp(String artist) throws Exception {...}
   …
}
```

Definire le variabili di istanza

- Ogni volta che viene invocato **`lookUp`** crea un nuovo **`Scanner`** associato al file su disco specificato dal nome del file di canzoni (passato al costruttore).

- Questo nome deve quindi essere mantenuto in una variabile d'istanza

```
class SongLibrary {
  public SongLibrary(String songFileName) {
         this.songFileName = songFileName;
  }
  ...                          // Metodo lookup
  String songFileName; // variabile d'istanza
}
```

Implementazione del metodo lookup

```
void lookUp(String artist) throws Exception {
   Scanner in =
            new Scanner(new File(songFileName)));
   Song song = Song.read(in);  // necessità di una
                               // classe Song
   while(song != null) {
          if (artist.equals(song.getArtist()))
          System.out.println(song.getTitle());
          song = Song.read(in);
    }
 }
```

La classe Song

- L'interfaccia e le variabili d'istanza

```
class Song {
   // Metodi
   public static Song read(Scanner in) throws
   Exception {…}
   public Song(String title, String artist) {…}
   String getTitle() {…}
   String getArtist() {…}

   // Variabili d'istanza
   String title, artist;
}
```

La classe Song

- Implementazione del metodo **read**

```
public static Song read(Scanner in) throws
Exception
{
  if (!in.hasNext())
        return null;
  String title = in.next();
  String artist = in.next();
  return new Song(title, artist);
}
```

La classe Song

- Implementazione del costruttore e degli altri metodi

```
public Song(String title, String artist) {
    this.title = title;
    this.artist = artist;
}

String getTitle() {
    return this.title;
}

String getArtist() {
    return this.artist;
}
```

Gestione di valori multipli

- Il metodo `lookUp` deve scorrere il file ogni volta che viene invocato
- Per migliorare l'efficienza si può pensare di mantenere in memoria il contenuto del file
- Non si conosce a priori il numero di canzoni nel file
  - non si conosce il numero di variabili da dichiarare
- Occorre dichiarare una collezione di oggetti
  - Un gruppo di oggetti che può essere dichiarato come entità singola