**Corso di base JAVA**

*Mauro Donadeo*
*mail: mauro.donadeo@gmail.com*

# Metodi e Variabili statiche, Gestione File, Array


HT Lab

# Metodi statici

Esistono classi che non servono a creare oggetti ma contengono **metodi statici** e **costanti**

- Queste si chiamano solitamente **classi di unità**
- La classe **Math** è un esempio di questo tipo di classi

# Metodi statici

Esistono classi che non servono a creare oggetti ma contengono **metodi statici** e **costanti**

- Queste si chiamano solitamente **classi di unità**
- La classe **Math** è un esempio di questo tipo di classi

```
1 public class Financial{
2     public static double percentOf(double p, double a){
3         return (p / 100) * a;
4     }
5 // qui si possono aggiungere altri metodi finanziari
6 }
```

Non è necessario **creare oggetti** di tipo **Financial** per usare i metodi della classe

**double tax = Financial.percentOf(taxRate,total);**

# Variabili statiche

Vogliamo modificare **BankAccount** in modo che:

- il suo stato contenga anche **numero di conto**

```
1 public class BankAccount
2 { ...
3     private int accountNumber;
4 }
```

- il numero di conto sia assegnato dal costruttore:
  - ogni conto deve avere un numero diverso
  - i numeri assegnati devono essere progressivi, iniziano da 1.

# Soluzione

## Prima idea

Usiamo una variabile per memorizzare l'ultimo numero di conto assegnato

```
1  public class BankAccount
2  { ...
3      private int accountNumber;
4      private int lastAssignedNumber;
5  ...
6      public BankAccount()
7      {
8          lastAssignedNumber++;
9          accountNumber = lastAssignedNumber;
10     }
11 }
```

## Il costruttore non funziona perché?

Questo costruttore non funziona perché **lastAssignedNumber** è una **variabile di esemplare**:

- ne esiste una copia per ogni oggetto;
- il risultato è che tutti i conti creati hanno un numero di conto uguale a 1

# Variabili statiche

Ci serve una **variabile condivisa da tutti gli oggetti della classe**

- una variabile con questa semantica si ottiene con la dichiarazione **static**

```
private static int lastAssignedNumber;
```

- Una vatiabile **static** **(variabile di classe)** è condivisa da tutti gli oggetti della classe;
- Ne esiste **un'unica copia** indipendentemente da quanti oggetti siano creati.

```
1  public class BankAccount{
2      ...
3      private int accountNumber;
4      private static int lastAssignedNumber = 0;
5      ...
6      public BankAccount(){
7          lastAssignedNumber++;
8          accountNumber = lastAssignedNumber;
9      }
10 }
```

Ogni metodo (o costruttore) di una classe può **accedere** alle variabili statiche della classe **modificarle**

- Le variabili statiche **non** possono (da un punto di vista logico) essere inizializzate nei costruttori:
  - Il loro valore verrebbe inizializzato nuovamente **ogni volta che si costruisce un oggetto**, perdendo il vantaggio di avere una variabile condivisa.
- Bisogna inizializzarle quando queste si dichiarano;
- Questo può valere anche per le variabili di esemplare, anziché usare un costruttore:
  - **non** è una buona pratica di programmazione.

È invece pratica comune (senza controindicazioni) usare **costanti** statiche, come la classe **Math**.

```
1 public class Math
2 { ...
3     public static final double PI
          =3.14159265358979323846;
4 }
```

Tali costanti sono di norma **public** e per ottenere il loro valore si usa il nome della classe seguito dal punto e dal nome della costante, **Math.PI**

Sappiamo che in Java esistono quattro diversi tipi di variabili:

- variabili **locali** (all'interno di un metodo)
- variabili **parametro** (dette **parametri formali**)
- variabili **di esemplare** (o di istanza)
- variabili **statiche** o di classe

Hanno in comune il fatto di contenere valori appartenenti ad un tipo ben preciso. Differiscono per quanto riguarda il loro **ciclo di vita**

- cioè nell'intervallo di tempo in cui, dopo essere state create, continuano ad occupare lo spazio in memoria riservato loro.

## Variabile locale

- **viene creata** quando viene eseguito l'enunciato in cui è definita;
- **viene eliminata** quando l'esecuzione del programma esce dal **blocco di enunciati** in cui la variabile è definita

## Variabile parametro (formale)

- **viene creata** quando viene invocato il metodo
- **viene eliminata** quando l'esecuzione del metodo termina

## Variabile statica

- **viene creata** quando la macchina virtuale Java carica la classe per la prima volta
- **viene eliminata** quando l'esecuzione del metodo termina
- a fini pratici possiamo dire che **esiste sempre**

## Variabile di esemplare

- **viene creata** quando viene creato l'oggetto a cui appartiene
- **viene eliminata** quando l'oggetto viene eliminato

- Per evitare conflitti, dobbiamo conoscere l'**amito di visibilità** di ogni tipo di variabile
  - Ovvero la **porzione** del programma all'interno della quale si può accedere ad essa;
- **Esempio:** due variabili locali con lo stesso nome. Funziona perché gli ambiti di visibilità sono **sono disgiunti**

```
1 public class RectangleTester{
2     public static double area(Rectangle rect){
3         double r = rect.getWidth() * rect.getHeight();
4         return r; }
5     public static void main(String[] args){
6         Rectangle r = new Rectangle(5, 10, 20, 30);
7         double a = area(r);
8         System.out.println(r); }
9 }
```

Anche qui gli ambiti di visibilità sono **disgiunti**

```
1 if (x >= 0){
2     double r = Math.sqrt(x);
3     ... } // la visibilita' di r termina qui
4 else{
5     Rectangle r = new Rectangle(5, 10, 20, 30);
6     // OK, questa e' un'altra variabile r
7 ...
8 }
```

Invece l'ambito di visibilità di una variabile **non** può contenere la definizione di un'altra variabile locale con lo stesso nome:

```
1 Rectangle r = new Rectangle(5, 10, 20, 30);
2 if (x >= 0)
3 {
4     double r = Math.sqrt(x);
5     // Errore: non si puo' dichiarare un'altra var. r
          qui
```

## Visibilità di membri di classe

- Membri **private** hanno visibilità di classe
  - Qualsiasi metodo di una classe può accedere a variabili e metodi della stessa classe
- Membri **public** hanno visibilità al di fuori della classe
  - A patto di renderne **qualificato** il nome, ovvero:
    - Specificare il nome della classe per membri static: **Math.PI, Math.sqrt(x)**
  - Specificare l'oggetto per i membri **non static**
- Non è necessario qualificare i membri appartenenti ad una stessa classe.

# Array

## Problema

- **Scrivere un programma che legge dallo standard input una sequenza di dieci numeri in virgola mobile, uno per riga**
- chiedere all'utente un numero intero **index** e visualizzare il numero che nella sequenza occupava la posizione indicata da **index**.

- Occorre **memorizzare tutti i valori della sequenza**

## Problema

- **Scrivere un programma che legge dallo standard input una sequenza di dieci numeri in virgola mobile, uno per riga**
- chiedere all'utente un numero intero **index** e visualizzare il numero che nella sequenza occupava la posizione indicata da **index**.

- Occorre **memorizzare tutti i valori della sequenza**
- Potremmo usare dici 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 un 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, mi di qualunque dimensione.

# 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 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;
```

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

# Leggere/scrivere file di testo

# 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 **Scanner** per leggere i dati 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 numero intero a ogni invocazione