Dispense di Informatica in preparazione al Compito Pratico di Programmazione ad Oggetti

Matteo Leggio, Mar 2023

LVL.1 - Cos'e' un Java? Si mangia?

1.1 - Cos'e' Java

Java e' un "linguaggio di programmazione orientato agli oggetti". Tutto cio' che c'e' da capire di questo e' che **ogni cosa e' un oggetto**. Di cos'e' un oggetto parleremo dopo.

1.2 - La struttura di un programma Java Semplice

Un programma Java, nella sua forma piu' basilare, e' formato da **3 strati**:

1. **La Classe**

- Deve avere lo stesso nome del file dove e' scritta
- Contiene la struttura dati del programma, come un contenitore

2. **Il Metodo Main**

- Si trova all'interno della classe
- E' il punto dove il codice inizia ad eseguire
- Deve contenere cio' che vogliamo che il programma faccia

3. **Le Istruzioni**

- Si trovano all'interno del metodo main
- Ogni istruzione verra' eseguita, a partire dall'alto verso il basso, una alla volta
- Ogni istruzione deve terminare con un punto e virgola

1.3 - Si, ma che devo scrivere?

Scriviamo un programma che stampa *"Ciao mondo"* sullo schermo. Per fare questo, useremo l'istruzione `System.out.println()`, che non fa altro che prendere una stringa e scriverla sullo schermo. Fate attenzione alle maiuscole e minuscole (capitalizzazione).

```java
// Nome file: CiaoMondo.java

// La Classe
class CiaoMondo {

  // Il Metodo Main
  // PS: IMPARA LA PRIMA RIGA A MEMORIA
  //     e' inutile per ora
  //     sapere cosa vuol dire
  public static void main(String[] args) {

    // Le Istruzioni
    System.out.println("Ciao mondo");

  }

}
```

1.4 - Cos'e' un Oggetto

Immaginiamo di dover fare un programma per gestire uno stadio di calcio. Gli oggetti del problema saranno, ad esempio, lo **stadio** stesso, il quale contiene dei **posti a sedere** (anch'essi oggetti) e due squadre che giocano. La **squadra** e' anch'essa un oggetto, che contiene dei **calciatori** e cosi' via.

Insomma, **tutto cio' che e' parte del problema e' un oggetto**.

In realta' il nome piu' corretto sarebbe Classe, ma Oggetto e' accettabile. Con la parola oggetto e' piu' corretto definire, invece di una "squadra" generica, la Juve, la Roma, l'Inter... ovvero **istanze della classe Squadra**

1.5 - Cos'e' un Attributo

Ogni oggetto puo' avere delle caratteristiche. Dato l'esempio dello stadio di calcio, l'oggetto **squadra** avra' come caratteristiche il suo **nome**, il **colore della sua maglia**, il suo **inno** e cosi' via.

Ogni caratteristica di un oggetto e' detta attributo

1.6 - Cos'e' un Metodo

Ogni oggetto puo' anche compiere delle azioni. Dato l'esempio dello stadio di calcio, l'oggetto **calciatore** potra' compiere le azioni di **passare la palla**, **calciare il pallone**, **esultare** e cosi' via.

Ogni azione che un oggetto puo' compiere e' detta metodo

1.7 - Cos'e' un Costruttore

Ogni oggetto, prima di essere utilizzato, deve essere creato. La sintassi per creare un nuovo oggetto e':

```
Oggetto nome = new Oggetto(valori)
```

Dove `Oggetto` e' la nostra classe. Notiamo che **possiamo passare dei valori all'oggetto** mentre lo creiamo. Facendo questo stiamo usando il **metodo costruttore** per impostare i valori iniziali dell'oggetto.

Il costruttore e' un metodo con lo stesso nome della classe

1.8 - Esempio pratico

Immaginiamo l'esempio dello stadio di calcio e implementiamo la classe Squadra, con i suoi metodi e attributi.

```java
// Nome file: Squadra.java
class Squadra {

  // INIZIO DEGLI ATTRIBUTI
  public String nome;
  public String inno;
  public Calciatore[] calciatori;
  // FINE DEGLI ATTRIBUTI

  // INIZIO DEI METODI
  public Squadra(String nomeDato, String innoDato, Calciatore[] calciatoriDati) {
    // COSTRUTTORE
    // Imposto gli attributi ai valori dati dall'utente
    this.nome = nomeDato;
    this.inno = innoDato;
    this.calciatori = calciatoriDati;
  }

  public void cantaInno() {
    // Scrivo sulla console l'inno della squadra
    System.out.println(this.inno);
  }

  public void cambiaCalciatore(Calciatore nuovo, int posizione) {
    // Cambia il calciatore in posizione `posizione` con il nuovo calciatore `nuovo`
    this.calciatori[posizione] = nuovo;
  }
  // FINE DEI METODI

}
```

Questa classe potra' poi essere importata ed usata cosi' in un altro file:

```java
// Nome file: Main.java
import il.mio.pacchetto.Squadra;

class Main {

  public static void main(String[] args) {
    // Per semplificare, utilizzo un array di calciatori vuoto
    Calciatore[] calciatoriDellaRoma = new Calciatore[11];

    Squadra roma = new Squadra("Roma",
                               "Roma roma roma core de sta citta'...",
                               calciatoriDellaRoma);
    roma.cantaInno();      // Output: "Roma roma roma core de sta citta'..."
  }

}
```

1.9 - Istruzioni da conoscere

- `System.out.println("ciao")` Scrive "ciao" sulla console.
- `Scanner in = new Scanner(System.in)` Crea una variabile `in` usata per chiedere valori all'utente
- `in.nextLine()` Chiede una stringa all'utente e attende che prema invio
- `in.nextInt()` Chiede un numero intero all'utente
- `in.nextDouble` Chiede un numero double all'utente. Il processo e' ripetibile per tutti i tipi base
- `"stringa 1".equals("stringa 2")` Controlla se due stringhe sono uguali
- `"stringa 1".charAt(3)` Ritorna il carattere all'indice 3 (quarto carattere) della stringa, quindi "i"
- `"stringa 1".indexOf("i")` Ritorna l'indice della prima "i" che appare nella stringa, quindi 3
- `"stringa 1".length()` Ritorna la lunghezza della stringa
- `"stringa x numero x 1".split("x")` Ritorna un aOgni oggetto puo' avere delle caratteristiche. Dato l'esempio dello stadio dirray composta dalla stringa divisa per "x", quindi ["stringa", "numero", "1"]

1.10 - Esercizio

Progetta e scrivi un programma, orientato agli oggetti, che gestisce una classe di alunni. Sviluppa le varie classi pertinenti al dominio.

LVL.2 - So' gia' scrivere una classe Java

2.1 - Modificatori di Livello d'Accesso

In Java, **se non specificato**, ogni classe, metodo e attributo e' considerato **pubblico**. Cio' significa che chiunque al di fuori della classe puo' leggere e modificare quel valore o eseguire quel metodo. Questa non e' sempre la scelta migliore.

Per ovviare a questo problema, esistono dei modificatori, **inseribili prima del tipo di un metodo o di un attributo**, capaci di modificare chi puo' accedervi. Sono:

- `public`
  - Come il default, chiunque puo' accedere, modificare e eseguire
- `private`
  - Nessuno all'esterno della classe puo' accedere, modificare e eseguire
  - Vi puo' essere bisogno di **metodi getter e setter**
- `protected`
  - Solo le classi all'interno dello stesso pacchetto possono accedere, modificare e eseguire

La sintassi e' `<modificatore> <tipo> <nome>`

2.2 - Metodi Getter e Setter

Nel caso di una classe con attributi privati, potrebbe essere necessario dare la possibilita' all'utilizzatore della classe di poter **leggere gli attributi ma non modificarli**. Questo e' possibile con l'implementazione di **metodi getter**.

I metodi getter non sono altro che **metodi pubblici che ritornano il valore di un attributo privato**. Questo rende possibile all'utilizzatore di leggere l'attributo attraverso il getter, ma non di modificarlo.

Vi sono anche casi in qui e' necessario dare la possibilita' di impostare il valore di un attributo privato. Questo e' possibile tramite i **metodi setter**.

I metodi setter sono **metodi pubblici che impostano il valore di un attributo privato e**, eventualmente, **controllano che il valore sia accettabile**. Immaginiamo di avere una classe `Orologio`, con un attributo `private int ora` che segna che ora e'. E' ovvio che `ora` potra' contenere soltanto valori tra 1 e 24. Per far rispettare questa richiesta si puo' implementare un setter che verifica la validita' del dato.

Esempio:

```
// ...

public void setOra(int ora) {
  if ((ora > 24) || (ora < 1)) {
    System.out.println("Il valore dell'ora deve essere contenuto fra 1 e 24");
    return;
  } else {
     this.ora = ora;
  }
}

// ...
```

2.3 - Lanciare Eccezioni

Il precedente esempio usa un `System.out.println()` per dire all'utilizzatore che il valore inserito e incorretto. Una soluzione piu' corretta sarebbe invece stata **lanciare un eccezione**. Questo significa **generare un errore manualmente**.

La sintassi e':

```
throw new Exception("Messaggio d'errore");
```

Sara' inoltre necessario aggiungere `throws Exception` dopo il nome e i parametri del metodo, come segue:

```
public void setOra(int ora) throws Exception {
  // ...
}
```

Se si vuole, invece, gestire un'eccezione, si possono usare le istruzioni `try/catch` come segue:

```
try {
    // Codice che potrebbe portare a un errore
} catch (Exception e) {
    // Gestisci l'errore se accade
}
```

2.4 - Array Dinamici

Come sappiamo, un normale array possiede una **dimensione ben definita**, e che **non puo' essere modificata**. Questo non e' sempre ideale. Immaginiamo un programma che richiede dei dati da un utente e li inserisce man mano in un array. Non sapendo quanti dati l'utente inserira', e' impossibile decidere una dimensione corretta dell'array.

Per ovviare a questo esistono gli array dinamici, i piu' comuni tra i quali sono gli **ArrayList**. Gli array dinamici sono un particolare tipo di array che puo' ingrandirsi o restringersi a comando.

La sintassi per crearlo e' la seguente:

```
ArrayList<String> arrayDiStringhe = new ArrayList<String>(dimensioneIniziale)
```

Questo creera' un `ArrayList` di tipo `String` con una dimensione iniziale pari a `dimensioneIniziale`

Per aggiungere e rimuovere elementi e poi possibile usare la seguente sintassi:

```
// Aggiungi
arrayDiStringhe.add("ciao!")
arrayDiStringhe.add("mondo!")
arrayDiStringhe.add(":)")

// Rimuovi
arrayDiStringhe.remove(1)     // Rimuovi l'elemento in indice 1 (seconda posizione)
```

Nota bene che, **una volta rimosso un elemento** dall'ArrayList, **tutti gli elementi che lo seguivano scaleranno indietro**, percio' quello che prima era l'elemento 7 diventera' l'elemento 6, l'elemento 6 il 5 e cosi' via.

2.5 - La liberia Math

Java **fornisce varie librerie standard**, ovvero classi e metodi gia pronti e utilizzabili. Una importante libreria tra queste e' la libreria Math, usata per **eseguire particolari calcoli matematici**, come radice quadrata, potenza, eccetera.

Si possono usare i suoi vari metodi senza bisogno di importare la libreria esplicitamente. Tra i piu' importanti spiccano:

- `Math.sqrt(x)` Calcola la radice quadrata di `x`
- `Math.random()` Ritorna un numero randomico tra 0.0 e 1.0 (con 1.0 escluso)
- `Math.pow(x, y)` Ritorna la potenza di `x` a `y`, quindi x^y
- `Math.floor(x)` Arrotonda `x` per difetto
- `Math.ceil(x)` Arrotonda `x` per eccesso

2.6 - Esercizio

Progetta e scrivi un programma orientato agli oggetti che gestisce la sveglia di un cellulare ed i suoi promemoria. Permetti all'utente di inserire un numero indefinito di promemoria e, man mano che vengono inseriti, aggiungili alla sveglia.

LVL.3 - So' programmare per la NASA

3.1 - Overloading del Costruttore

In una classe Java e' possibile **creare piu' di un costruttore**. Questo puo' essere utile in casi in cui la Classe in questione puo' essere istanziata in base a dati diversi. Questi costruttori **devono** avere degli attributi diversi (firma). Questo perche' c'e' bisogno di un modo per distinguerli.

Per esempio, immaginiamo una classe `Orologio`, che contiene un attributo `ore` e uno `minuti`. Potremmo creare due diversi costruttori, uno che accetta sia ore che minuti ed uno che accetta solamente le ore e imposta i minuti a 0. La sintassi e', in questo caso:

```
class Orologio {
  private int ore;
  private int minuti;

  // Costruttore 1
  public Orologio(int ore, int minuti) {
    this.ore = ore;
    this.minuti = minuti;
  }

  // Costruttore 2
  public Orologio(int ore) {
    this.ore = ore;
    this.minuti = 0;
  }
}
```

3.2 - Overloading e Overriding

E' possibile ampliare il concetto dell'overloading del costruttore a tutti i metodi di una classe. Questo vale sia per **metodi definiti dal programmatore**, ma anche per **metodi gia definiti**, come il `toString()`.

Ogni volta che si ridefinisce un metodo, nostro o di sistema, cambiandone la firma, si dice **overloading**.

Se invece ridefinisamo un metodo mantenendone la firma, si dice **overriding**.

Ad esempio, se si volesse rendere possibile stampare i parametri di un oggetto quando questo viene passato a `System.out.println()`, si potrebbe scrivere:

```
class Orologio {
  // ...

  public String toString() {
    return "Sono le " + ore + ":" + minuti;
  }

  // ...
}
```

3.3 - Riferimenti

Quando in Java si istanzia un Oggetto e lo si assegna ad una variabile, quello che viene assegnato alla variabile non e' la classe stessa, bensi' un **riferimento allo spazio in memoria dell'istanza**.

Questo significa che, istanziando un oggetto e assegnandolo a due variabili diverse, come segue:

```
Oggetto o1 = new Oggetto();
Oggetto o2 = o1;
```

Nonostante le variabili siano due, **essendo riferimenti allo stesso oggetto**, qualsiasi operazione eseguita sull'oggetto attraverso una delle due variabili e' ritrovabile anche nell'altra. Questo vuol dire che, se avessimo una classe `Contatore` definita cosi':

```
class Contatore {
  private int conto;

  public Contatore() {
    this.conto = 0
  }

  public int getConto() {
    return this.conto;
  }

  public void aumentaConto() {
    this.conto++;
  }
}
```

E istanziandola come prima:

```
Contatore c1 = Contatore();
Contatore c2 = c1;
```

Eseguendo `c1.aumentaConto()` e stampando `c1.getConto()` si otterrebbe il

numero 1. A quel punto, pero', stampando `c2.getConto()`, si otterrebbe di nuovo 1.

Questo perche' **sia c1 che c2 sono riferimenti alla stessa instanza**

3.4 - Esercizio

Progetta e scrivi un programma orientato agli oggetti che gestisce un calendario con degli eventi. Questi eventi devono contenere una data e una descrizione. Inoltre possono contenere una posizione, una durata o dei partecipanti. Non vi e' bisogno di una interfaccia utente, purche' vi siano abbastanza metodi da poterne creare una in seguito.