Giorno 5
OOP: Java

1 Java

1.1 Struttura di un programma Java

Un programma Java è *un insieme di classi*.

- In linea di principio: ogni classe in un file diverso
- Una delle classi dovrà avere un metodo *main* da cui parte l'esecuzione del programma.

1.1.1 Hello World

```
public class HelloWorld {
   public static void main (String [] args) {
       System.out.println("Hello World !");
   }
}
```

1.1.2 Nucleo imperativo

Tutto uguale a C, tranne il fatto che non ci sono puntatori, gli array hanno una sintassi diversa:

```
int[] numeri;      // attenzione alle parentesi:
                   // "int numeri[]" e' sbagliato
// inizializzazione
numeri = new int[10];  // dimensione 10 e valori di default (0)
numeri = {5,3,2,6};    // dimensione 4 e valori 5,3,2 e 6

// anche insieme
int[] numeri = new int[10
```

e il tipo `String` è in realtà una speciale classe di libreria; la concatenazione si fa con +, come se fosse un tipo primitivo.

1.2 Classi

```
class Name {
   public ...
   private ...
   public static ...
}
```

- `public`: l'attributo/metodo si può accedere anche dall'esterno della classe
- `private`: l'attributo/metodo non si può accedere dall'esterno
- `static`: un metodo statico può essere chiamato senza istanziare la classe.
- ... **final, abstract, transient, synchronized, volatile**

1.2.1 Incapsulamento

I modificatori `public` e `private` consentono di realizzare sistemi di *incapsulamento* (i.e. la rappresentazione dell'oggetto rimane nascosta/privata, l'accesso dall'esterno è consentito solo attraverso un certo numero di metodi pubblici)

Si possono definire anche *metodi privati*, utilizzabili all'interno della classe (metodi ausiliari).

1.3 Interfacce

Un'interfaccia contiene le intestazioni dei membri pubblici di una classe e non contiene costruttori:

```
public interface BankAccount {                          // interface, non class
   public double getSaldo();
   public void versa(double somma);
   public boolean preleva(double somma);
}
```

Per implementare un'interfaccia in una classe si scrive:

```
public class ContoCorrente implements BankAccount {    // implements!
   private double saldo;
   public ContoCorrente(double saldoIniziale) { ... }
   public void versa(double somma) { ... }
   public double getSaldo() { return saldo; }
   public boolean preleva(double somma) {
       if (saldo>=somma) { ... }
       else return false;
   }
}
```

Nessun costruttore! Se ne usa uno di default e senza parametri che inizializza le variabili come da dichiarazione, oppure (se la dichiarazione non prevede un assegnamento) a valori standard.

1.3.1 A che serve implements?

Usando `BankAccount` come tipo, abbiamo appena definito la relazione di sottotipo:

`ContoCorrente <: BankAccount`

1.3.2 implementare più interfacce

Una classe può implementare più interfacce:

```
public class ContoFlessibile implements BankAccount,DepositAccount {...}
```

Si avrà:

`ContoFlessibile <: BankAccount` `ContoFlessibile <: DepositAccount`

1.4 Tipo apparente e tipo effettivo

- Il Tipo Apparente (o **statico**) è il tipo usato dal compilatore per fare i controlli;
- Il Tipo Effettivo (o **dinamico**) è il tipo che l'oggetto avrà a runtime.

E.g. Dichiaro due variabili di tipo `BankAccount`, istanze di due diversi sottotipi di BankAccount:

```
BankAccount conto1 = new ContoCorrente(1000);
BankAccount conto2 = new ContoLimitato(200,10);
```

Il **tipo apparente** di entrambe è `BankAccount`, mentre i **tipi effettivi** sono `ContoCorrente` e `ContoLimitato`.

1.4.1 Cast/Coercion

- Il cast da sottotipo a supertipo (**upcast**) è implicito
- Il cast da supertipo a sottotipo (**downcast**) deve essere esplicito (stessa sintassi di C).

In OCaml il downcast non è possibile.

1.5 Membri statici e d'istanza

- I membri (variabili e metodi) **d'istanza** sono quelli che codificano lo stato di **un singolo oggetto** (un'**istanza** della classe)
- I membri **statici** codificano operazioni **di classe** (non operano sullo stato dei singoli oggetti).

1.5.1 Esempio di utilizzo

Se vogliamo assegnare un numero univoco ad ogni oggetto di una classe, si utilizzano una **variabile d'istanza** per mantenere il numero del singolo oggetto ed una **variabile statica** per mantenere il conto dei numeri assegnati (e.g. mantenere il numero di oggetti istanziati, se assegnamento lineare)

1.6 Modello della memoria

Identifichiamo tre aree nella JVM:

- **Ambiente delle classi - Workspace**, che contiene il codice dei metodi e le variabili statiche
- **Stack**, contiene i RdA dei metodi con le variabili locali
- **Heap**, contiene gli oggetti (raggiungibili tramite **riferimenti**), con le loro variabili d'istanza

1.6.1 Descrizione del funzionamento

- Inizialmente le classi sono caricate nell'ambiente delle classi
- Poi il RdA del metodo main è caricato nello stack
- Gli oggetti creati con **new** dal metodo `main` (e poi quelli creati dai metodi degli oggetti chiamati da `main`, ecc.) sono caricati nello heap, e mantenuti per riferimento dai RdA
- I RdA dei metodi chiamati dagli oggetti vengono caricati nello stack.