1 Lezione del 30-09-25

1.1 Metodo main

L'entrypoint di un programma in Java è il metodo `main()` (dichiarato `public` e `static`), definito all'interno della classe `Main` (dichiarata `public`).

`main()` ha solitamente come argomenti l'array di riferimenti a `String` detta `args`, che contiene gli argomenti con cui è stato chiamato il programma.

L'esempio tipico sarà quindi:

```
public class Main() {
  public static void main(String[] args) {
    // entrypoint del programma
  }
}
```

Si possono avere più di un metodo `main()` finché se ne definisce uno per ogni classe (ergo possono esistere classi non `Main` che possiedono un metodo `main()`).

Questo può essere utile a elaborare (in maniera non proprio strutturata) test intermedi sulle classi che si implementano.

Un altro caso utile può essere nel caso di applicazioni che implementano più modalità di esecuzione (magari GUI e CLI), dove ogni `main()` corrisponde ad una certa modalità.

1.2 Ereditarietà

Il Java, essendo un linguagio OOP, implementa l'**ereditarietà** delle classi: si può definire una classe come *figlia* di un altra, e in questo caso ne *eredità* campi e metodi.

Per fare questo si usa la parola chiave `extends`:

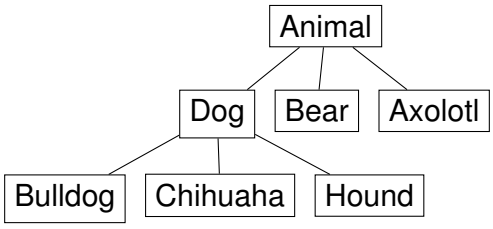
```
class Animal {
  // implementazione di Animal
}

class Dog extends Animal {
  // implementazione di Dog
}
```

Una limitazione è che una classe figlia può avere una sola classe padre (a differenza di linguaggi come il C++). Vedremo che per replicare la solita funzionalità si usano le *interfacce*.

La relazione che viene stabilita quando si usa il meccanismo dell'ereditarietà è quella dell'*"è un"*, cioè con riferimento all'esempio sopra, un `Dog` *è un* `Animal`.

La definizione della classe figlia verrà quindi fatta in base alle *differenze* con la classe padre, cioè ridefinendo i campi del padre o aggiungendone altri. I campi privati vengono ereditati, ma la classe figlia non può accedervi.

Per rappresentare le gerarchie di classi graficamente si usano diagrammi di gerarchia come il seguente:



dove la relazione *"è un"* si svolge dall'alto verso il basso.

Uno dei chiari vantaggi dell'uso dell'ereditierietà è che un riferimento alla classe padre può riferire a istanze della classe figlia. Questa si dice conversione da sottotipo a supertipo. Si può effettuare anche la conversione opposta: questo chiaramente implica che la conversione sia valida (cioè la superclasse sia specializzata in una classe del tipo che cerchiamo).

Per valutare la validità della conversione è presente l'operatore `instanceof`, che restituisce `true` quando si confronta un'istanza di classe col suo tipo.

La regola generale è quindi che non conta il tipo del riferimento, ma il tipo dell'oggetto riferito.

Nel caso di chiamate a metodi di oggetti riferiti, il tipo del riferimento determina solo l'*insieme* dei metodi che possono essere invocati. L'implementazione effettiva del metodo viene definita dal tipo reale della classe, cioè chiamando metodi specializzati in classe figlie attraverso riferimenti a classi padri, si chiamano comunque i metodi specializzati e non quelli delle classe padri.

1.2.1 Riferimento `super`

All'interno di classi figlie possiamo usare la parola chiave `super` per riferirci alla classe padre. Questo può essere utile quando si ridefiniscono metodi e si vuole chiamare la versione del metodo nella classe padre (magari per poi aggiungere altra funzionalità).

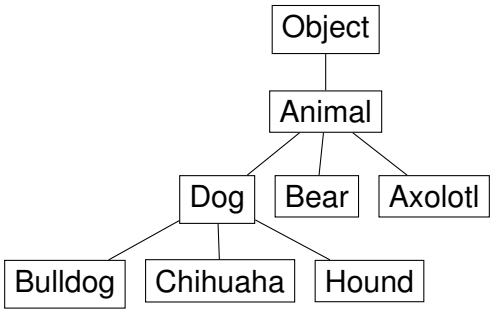
La parole chiave `super` può essere usata anche per chiamare il costruttore della classe padre (`super()`). In questo modo possiamo inizializzare la sezione di membri della classe padre quando si costruisce la classe figlia.

Se la classe padre è dotata di soli costruttori con argomenti, chiamare questo costruttore è obbligatorio.

1.2.2 Classe `Object`

Esiste una classe di sistema definita in `java.lang`, chiamata `Object`. Se una classe non indica esplicitamente di estenderne un altra, allora è implicitamente figlia della classe `Object`.

La gerarchia reale avrà quindi questo aspetto:



`Object` ha alcuni metodi, che possiamo dividere in 2 categorie:

- Metodi per la sincronizzazione fra thread (`wait()`, `notify()`, `notifyAll()`);
- Metodi di utilità generale (`equals()`, `hashcode()`, `toString()`)

1.2.3 Metodo `equals()`

Il metodo `equals()` ha la firma:

```
public boolean equals(Object o)
```

cioè prende come argomento un oggetto è confronta l'oggetto implicito (quello su cui è chiamato) con tale oggetto. A questo punto, restituisce `true` se i 2 oggetti risultano uguali, e `false` altrimenti.

Ogni classe può ridefinire il metodo `equals()` secondo il proprio criterio.

Per esempio, la classe `String` ridefinisce il metodo in modo tale da verificare se due stringhe sono uguali (composte dagli stessi caratteri).

L'implementazione di default della classe `Object` equivale invece all'operatore `==` (cioè per i riferimenti controlla se riferiscono lo stesso oggetto).

1.2.4 Metodo `getClass()`

Il metodo `getClass()` è definito per ogni oggetto `Object` e ci dà il primo sguardo al meccanismo della **riflessività**. La riflessività è il modo con cui il Java ci permette di guardare al codice sorgente del programma che stiamo sviluppando.

Il metodo `getClass()` restituisce quindi un oggetto di tipo `Class` (con la C maiuscola) che descrive la classe stessa dell'oggetto su cui è chiamata.

Possiamo usare questo metodo per capire se due classi hanno lo stesso tipo, o se una classe ha un dato tipo confrontandolo con i cosiddetti *letterali classe*, definiti come `<classe-nota>.class`.

1.2.5 Metodo `hashCode()`

Un'altro metodo definito per `Object` è `hashCode()`, che restituisce l'`hashCode` dell'oggetto a cui viene applicato. Questo è un valore che deve essere diverso per ogni oggetto (solitamente associato a dove l'oggetto è allocato).

Dobbiamo quindi impegnarci a ridefinirlo (nel caso ci torni utile) con campi che sono unici per ogni istanza di classe.

1.2.6 Metodo `toString()`

Infine, vediamo il metodo `toString()`. Questo serve semplicemente a convertire un oggetto in una stringa, e può essere ridefinito nella maniera che ha più senso (semanticamente) per il tipo di oggetto che stiamo implementando.

1.3 Polimorfismo

Il principio dell'ereditarietà, e principalmente il fatto che un riferimento di tipo superclasse può riferire ad oggetti ti tipo sottoclasse, ci permette di realizzare un'altro principio, quello del **polimorfismo**.

Supponiamo infatti che una superclasse implementi un metodo, e che più sottoclassi eredi di tale classe ridefiniscano dato metodo. Ottenendo un riferimento dei tipo superclasse a un insieme di classi figlie, potremmo chiamare il metodo ridefinito (dal riferimento di tipo superclasse) sulle sottoclassi, ottenendo ogni volta il comportamento ridefinito dalla sottoclasse.

1.4 Classi astratte

Una classe astratta non può essere istanziata, ma può essere usata come classe base nella definizione di nuove classi.

Le classi astratte sono quindi utili come modello per la definizione di classi: ci permettono di definire campi e metodi comuni a più classi figlie ma che di per sè non corrispondono a nessuna implementazione completa di classe.