La classe java.lang.Object

- In Java:
  - Gerarchia di ereditarietà semplice
  - Ogni classe ha una sola super-classe

- Se non viene definita esplicitamente una super-classe, il compilatore usa la classe predefinita **Object**
  - Object non ha super-classe!

Metodi di Object

- Object definisce un certo numero di **metodi pubblici**
  - Qualunque oggetto di qualsiasi classe li eredita
  - La loro implementazione base è spesso minimale
  - La tecnica del polimorfismo permette di ridefinirli

- public boolean **equals**(Object o)
  - Restituisce “vero” se l’oggetto confrontato è identico (ha lo stesso contenuto) a quello su cui viene invocato il metodo
  - Per funzionare correttamente, ogni sottoclasse deve fornire la propria implementazione polimorfica

Metodi di Object

- public String **toString**()
  - Restituisce una rappresentazione in forma di stringa dell'oggetto
  - L'implementazione base fornita indica il nome della classe seguita dal riferimento relativo all'oggetto

- public int **hashCode**()
  - Restituisce un valore intero legato al contenuto dell'oggetto
  - Se i dati nell'oggetto cambiano, deve restituire un valore differente
  - Oggetti "uguali" **devono** restituire lo stesso valore, oggetti diversi **possono** restituire valori diversi
  - Utilizzato per realizzare tabelle hash

Controllare l'ereditarietà

- In alcuni casi, si vuole impedire esplicitamente l'utilizzo della tecnica del polimorfismo
  - Ad esempio, per motivi di sicurezza o per garantire il mantenimento di una data proprietà del sistema
  - Si utilizza la parola chiave "**final**"

- Un metodo "final" non può essere ridefinito da una sottoclasse
- Una classe "final" non può avere sottoclassi
- Un attributo "final" non può essere modificato
  - Non c'entra nulla con l'ereditarietà!

Controllare l'ereditarietà

- In altri casi si vuole obbligare l'utilizzo del polimorfismo
  - Si introducono metodi privi di implementazione
  - Facendoli precedere dalla parola chiave "**abstract**"

- Una classe che contiene metodi astratti
  - Deve essere, a sua volta, dichiarata **abstract**
  - Non può essere istanziata direttamente
  - Occorre definire una sottoclasse che fornisca l'implementazione dei metodi mancanti

Classi astratte

```
abstract class
Base {
    abstract int m();
}
```

```
class Derivata
  extends Base {
        int m() {
          return 1;
      }
}
```

```
Base b= new Derivata();
System.out.println(b.m());
```

Interfacce

- Una classe astratta può contenere metodi non astratti
  - A beneficio delle proprie sottoclassi
- In alcuni casi, si vogliono definire metodi astratti senza vincolare la gerarchia di ereditarietà delle classi che li implementeranno
- Si utilizzano le **interfacce**:
  - Insiemi di metodi astratti e costanti (attributi ***static final***)
  - Pubblici per definizione
- Una classe può **implementare** un'interfaccia
  - Fornendo il codice relativo a tutti i metodi dichiarati nell'interfaccia

Esempio

```
public interface Comparable {
        public int  compareTo(Object o);
}
```

```
public class Rettangolo extends Forma
        implements Comparable {
            public int  compareTo(Object o) {
          //codice relativo…
            }
            //altri attributi e metodi…
}
```

Interfacce e tipi

- Analogamente alle classi, ogni interfaccia definisce un **tipo**
  - Un oggetto che implementa una data interfaccia ha come tipo **anche** il tipo dell'interfaccia
  - Un oggetto può implementare **molte** interfacce
  - Di conseguenza può avere molti tipi

- Si può verificare se un oggetto ha un dato tipo con l'operatore "**instanceof**"
  - if (myObject instanceof Comparable) …

Interfacce vuote

- Alcune interfacce **non hanno** metodi
  - Servono solo come "**marcatori**" o indicatori di tipo
  - Indicano che gli oggetti delle classi che le implementano godono di qualche **proprietà**

Classi e metodi astratti

- Una **classe astratta** è una classe che viene definita solo per stabilire una interfaccia comune per tutte le sue sottoclassi

- Non viene fornita l'implementazione completa per quella classe

- Si definisce solo l'interfaccia o parte dell'implementazione

- Quindi non ha senso creare oggetti di una classe astratta
  E infatti non si possono creare oggetti di una classe astratta altrimenti il compilatore si lamenta

Classi e metodi astratti

- All'interno della classe astratta è possibile definire metodi senza darne una implementazione

  Essi sono detti **metodi astratti**

- La loro sintassi è **abstract void f()**

- Se una classe contiene uno o più metodi astratti, anch'essa deve essere qualificata come abstract

Classi e metodi astratti

- Se si scrive una sottoclasse della classe astratta e se ne vogliono creare oggetti, si devono fornire implementazioni dei metodi astratti della superclasse

- Altrimenti, anche la sottoclasse diviene astratta

  In questo caso il compilatore forza l'inserimento della parola chiave abstract

- È possibile creare una classe astratta senza inserire metodi astratti

  È utile se vogliamo comunque evitare la creazione di oggetti di quella classe

Esempio di classe astratta

Costruiamo la **classe astratta Solido** che contiene come variabile il pesoSpecifico e tre metodi: uno che calcola la superficie del solido, uno che calcola il volume e uno che calcola il peso.

Quindi costruiamo la **classe Cubo** e la **classe Sfera** estensione della classe astratta Solido.

Esempio di classe astratta Solido

```
public abstract class Solido {

protected double pesoSpecifico;

private double peso(){
return (volume() * pesoSpecifico);}

public abstract double volume();

public abstract double superficie();
}
```

Sottoclasse concreta Sfera

```
public class Sfera extends Solido{

private double raggio;

public Sfera(double r, double ps){
raggio=r;
pesoSpecifico=ps;}

public double volume(){
return (4/3 * Math.PI * Math.pow(raggio,3));}

public double superficie(){
return (4 * Math.PI * Math.pow(raggio,2));}}
```

Sottoclasse concreta Cubo

```
public class Cubo extends Solido{

private double lato;

public Cubo(double l, double ps){
lato=l;
pesoSpecifico=ps;}

public double volume(){
return (Math.pow(lato,3));}

public double superficie(){
return (6 * lato * lato);} }
```

Esempio di interfaccia

Costruiamo l'interfaccia **Figura Piana** che contiene due metodi: il metodo che calcola il perimetro e il metodo che calcola l'area.

Quindi costruiamo la **classe Quadrato** e la **classe Rettangolo** che implementano l'interfaccia Figura Piana.

Esempio di interfaccia Figura Piana

```
public interface FiguraPiana {

public double Perimetro();
public double Area();

}
```

Classe Quadrato che implementa Figura Piana

```
public class Quadrato implements FiguraPiana{
    private double lato;

    public Quadrato(double l){
    lato=l;    }

public double Perimetro(){
return lato+lato;}

public double Area(){
return lato*lato;    }
}
```

Classe Rettangolo che implementa Figura Piana

```
public class Rettangolo implements FiguraPiana{
    private double  base;
    private double  altezza;

    public Rettangolo(double b, double a){
    base=b;
    altezza=a;    }

public double Perimetro(){
return (base * 2)+(altezza * 2);}

public double Area(){
return base * altezza;}
}
```

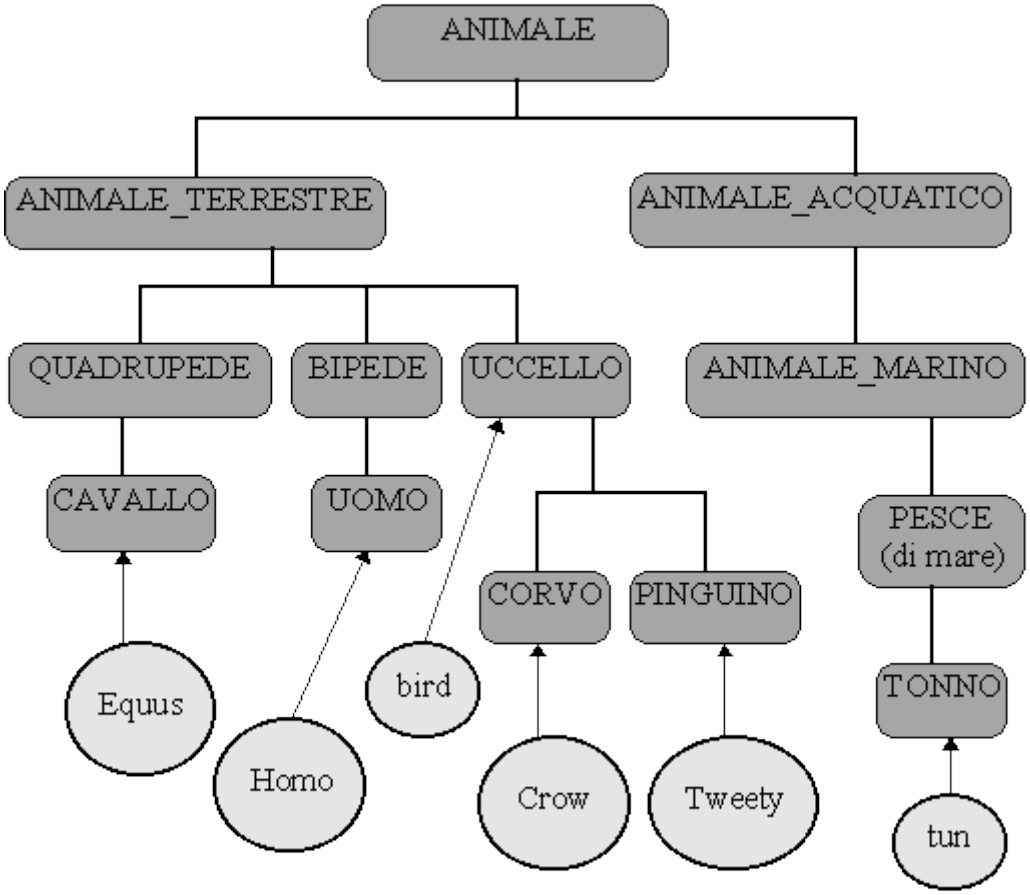
Esercizi su classi astratte

Esercizio 6

Realizzare una classe astratta per le Figure piane e due sottoclassi, la sottoclasse Quadrato e la sottoclasse Rettangolo.

Esercizio 7

Sviluppare un modello di classi animali seguendo lo schema sottostante:



Si richiede nella classe astratta Animali,

- due variabili di tipo String: nome e verso
- Un costruttore
- Un metodo astratto *si_muove* che restituisce in una stringa dove si muove l'animale
- Un metodo astratto *vive* che restituisce in una stringa dove vive l'animale
- Un metodo astratto *chi_sei* che restituisce in una stringa chi è l'animale
- Un metodo astratto *mostra* che stampa a video System.out.println(nome + ", " + chi_sei() + ", " + verso + ", si muove " + si_muove() + " e vive " + vive());