# Fondamenti di informatica T-1 (A – K)
# Esercitazione 9: ereditarietà, polimorfismo, esercizi d'esame

AA 2017/2018


Tutor:

**Lorenzo Rosa**

lorenzo.rosa4@unibo.it

# Esercitazione 9

- Introduzione al calcolatore e Java
- Linguaggio Java, basi e controllo del flusso
- I metodi: concetti di base
- Stringhe ed array
- Classi e oggetti, costruttori, metodi statici, visibilità
- Eclipse, ereditarietà e polimorfismo
- Collezioni Java
- Esercizi d'esame

# Array di oggetti

E' possibile definire array di qualsiasi tipo di oggetto (purché sia lo stesso oggetto). Possiamo creare un array di interi, di stringhe, di classi Persona, di classi Contatore:

```
public class TestArrayContatore {
    public static void main(String args[]){
        Contatore[] contatori = new Contatore[3];
        contatori[0].inc();
    }
}
```

Questo codice appena **non funziona:** sta tentando di accedere ad un oggetto non ancora inizializzato (vedi Esercitazione 7, slide 17-18).

# Array di oggetti

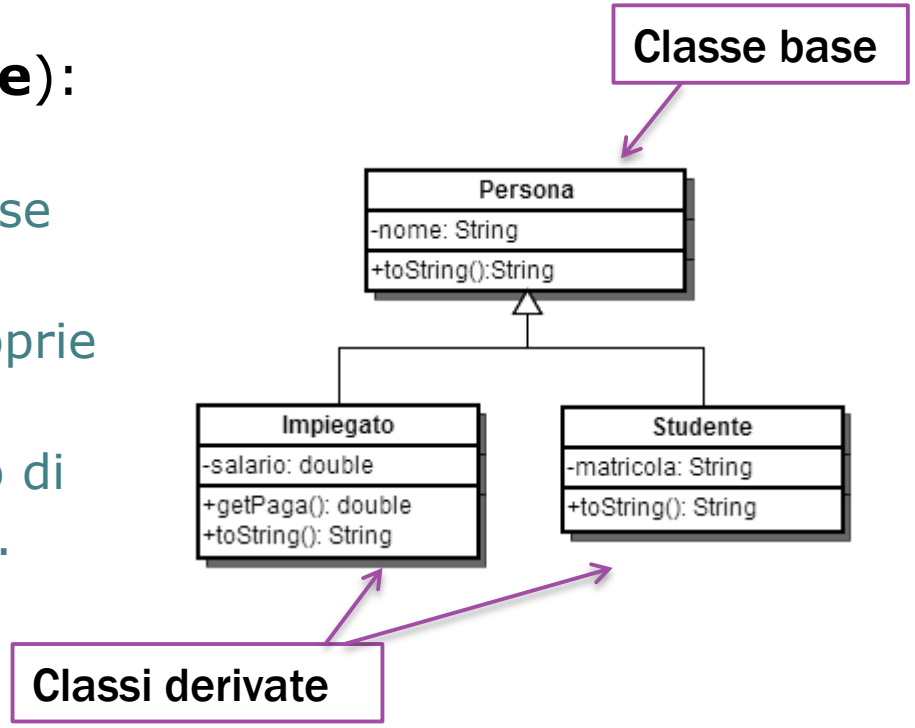
Poichè negli array di oggetti (a differenza dei tipi primitivi) i valori dell'array vengono inizializzati a *null*, è necessario **inizializzare ogni singolo oggetto con una new:**

```
Contatore[] contatori = new
Contatore[3];

for(int i=0; i<contatori.length; i++)
        contatori[i] = new Contatore();

contatori[0].inc(); //ok!
```

# Ereditarietà

- Meccanismo per definire una nuova classe (classe derivata) come specializzazione di un'altra (classe base)
  - La classe base modella un concetto generico
  - La classe derivata modella un concetto più specifico

- La classe derivata (**sottoclasse**):
  - Dispone di tutte le funzionalità (attributi e metodi) di quella base
  - Può aggiungere funzionalità proprie
  - Può ridefinirne il funzionamento di metodi esistenti (polimorfismo).



# Ereditarietà

- Processo di astrazione
  - Si introduce la superclasse che "astrae" il concetto comune condiviso dalle diverse sottoclassi
  - Le sottoclassi vengono "spogliate" delle funzionalità comuni che migrano nella superclasse
- Ogni classe definisce un tipo:
  - Un oggetto, istanza di una sottoclasse, è formalmente compatibile con il tipo della classe base
  - Il contrario non è vero!
- Ad esempio
  - Un impiegato è una persona ma una persona non è (necessariamente) un impiegato
  - Un'automobile è un veicolo ma un veicolo non è (necessariamente) un'automobile

# Esempio

```java
public class Persona {

    private String nome;
    public Persona(String nome) { this.nome = nome; }
    public String toString()    {return "Mi chiamo " + this.nome;}
}
```

```java
public class Studente extends Persona{
    private String matricola;
    public Studente(String nome, String matricola) {
        super(nome);
        this.matricola = matricola;
    }
    @Override
    public String toString() {
        return super.toString() +
        ", numero di matricola: " + this.matricola;
    }
}
```

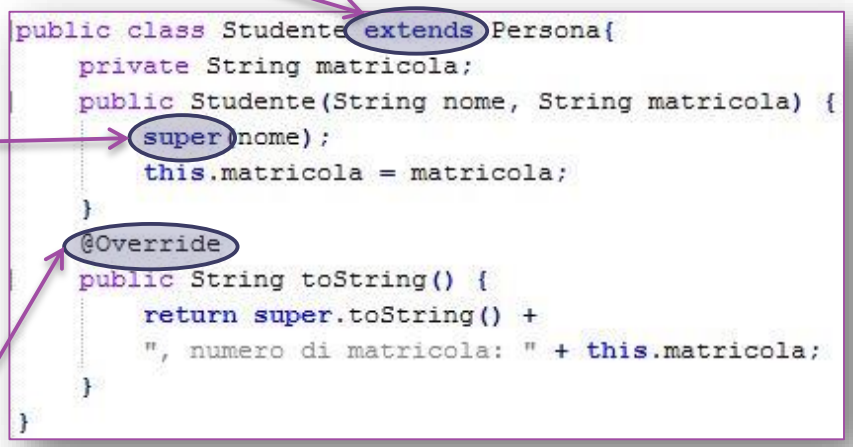
```java
public class Impiegato extends Persona {
    private double salario;
    public Impiegato(String nome, double salario) {
        super(nome);
        this.salario = salario;
    }
    public double getPaga() { return salario;}
    @Override
    public String toString() {
        return super.toString() + " e guadagno " + getPaga() + "€"; }
}
```

# Terminologia

Parola chiave ***"extends"*** :
Specifica da quale classe eredita.
Nell'esempio, Studente eredita da Persona

Parola chiave ***"super"*** :
Consente di invocare un metodo, un costruttore o un attributo della classe base purché non privati

Annotazione ***"@Override"*** :
Permette di ridefinire un metodo della superclasse a condizione che abbia stesso nome, parametri e tipo di ritorno (magari stessa semantica)



```
public class Studente extends Persona{
    private String matricola;
    public Studente(String nome, String matricola) {
        super(nome);
        this.matricola = matricola;
    }
    @Override
    public String toString() {
        return super.toString() +
        ", numero di matricola: " + this.matricola;
    }
}
```

# Il metodo *toString*

Ogni classe eredita da `Object` il metodo:

```
public String toString()
```

Esso ritorna una rappresentazione testuale che difficilmente è quella che desideriamo.

Esempio:

```
Persona p = new Persona("Giuseppe");
System.out.println(p.toString());
```

La stringa stampata in questo caso è:

```
Persona@15db9742
```

... non proprio leggibile!

# Il metodo *toString*

Per avere una stampa significativa, dobbiamo ridefinire il metodo `toString()` della nostra classe.

```
public class Persona {
        ...
       @Override
       public String toString() {
           return "Mi chiamo: "  + this.nome;
       }
}
```

Esempio:

```
Persona p = new Persona("Giuseppe");
System.out.println(p.toString());
```

La stringa stampata in questo caso è:

```
Mi chiamo Giuseppe
```

... molto meglio!

# Il metodo *equals*

- Ogni classe eredita da `Object` il metodo:

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

che indica se "un oggetto *o* è uguale a quello corrente". Così come le stringhe, anche gli altri oggetti si confrontano in questo modo (e non con "==").

```
Persona p1 = new Persona("Mario");
Persona p2 = new Persona("Mario");
bool res = p1.equals(p2); //true
```

- Problema:
  - il metodo *equals* della classe Persona prende in ingresso un generico oggetto *o,* che potrebbe non essere una persona!

```
}
```

# Il metodo *equals*

```
@Override
public boolean equals(Object obj) {
   if( obj instanceof Persona )
      return ((Persona)obj).getNome().equals(this.nome);
   else return false;
}
```

- *instanceof* è una keyword Java che serve a controllare di poter eseguire un cast. Nell'*if* controllo che obj sia davvero di classe *Persona,* così posso effettuare il *cast* senza problemi.

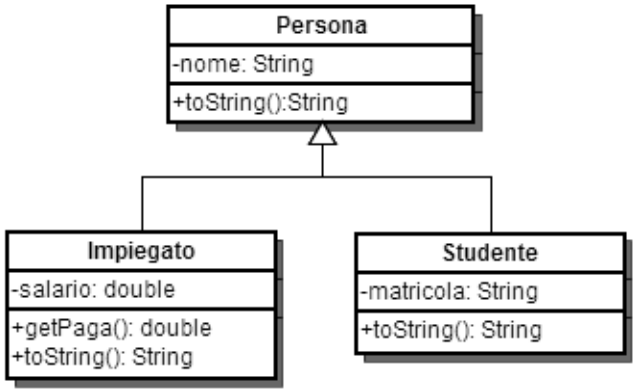
# Polimorfismo

Il **polimorfismo** permette di *invocare il metodo della sottoclasse più specifica.*

Questo comportamento si ha automaticamente.

Esempio:

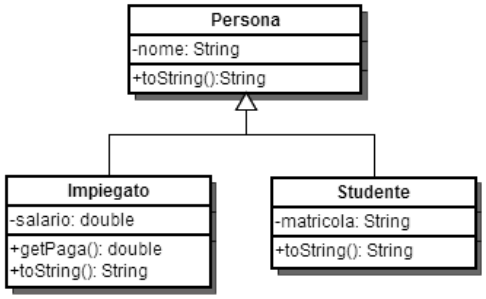
```
Persona p = new Studente("Mario",“123456”);
System.out.print( p.toString() );
```




Output:

```
Mi chiamo Mario, numero di matricola 123546
```

# Polimorfismo




Esempio

```
Persona g = new Persona("Giuseppe");
Persona m = new Studente("Mario", "123456");
Persona f = new Impiegato("Franco", 1000);

System.out.println(g);
System.out.println(m);
System.out.println(f);
```

**La `toString()` viene invocata automaticamente**

Output:

```
Mi chiamo Giuseppe
Mi chiamo Mario, numero di matricola 123456
Mi chiamo Franco, guadagno 1000€
```

# Collection

**`java.util.Collection`** è un'interfaccia, radice di una tassonomia di interfacce e di classi concrete.

Una C*ollection* rappresenta una qualunque collezione di oggetti:

- Alcune collezioni permettono l'esistenza di duplicati, altre no.
- Alcune sono collezioni ordinate, altre no. L'ordinamento viene effettuato secondo il criterio definito nella *compareTo(Object)*

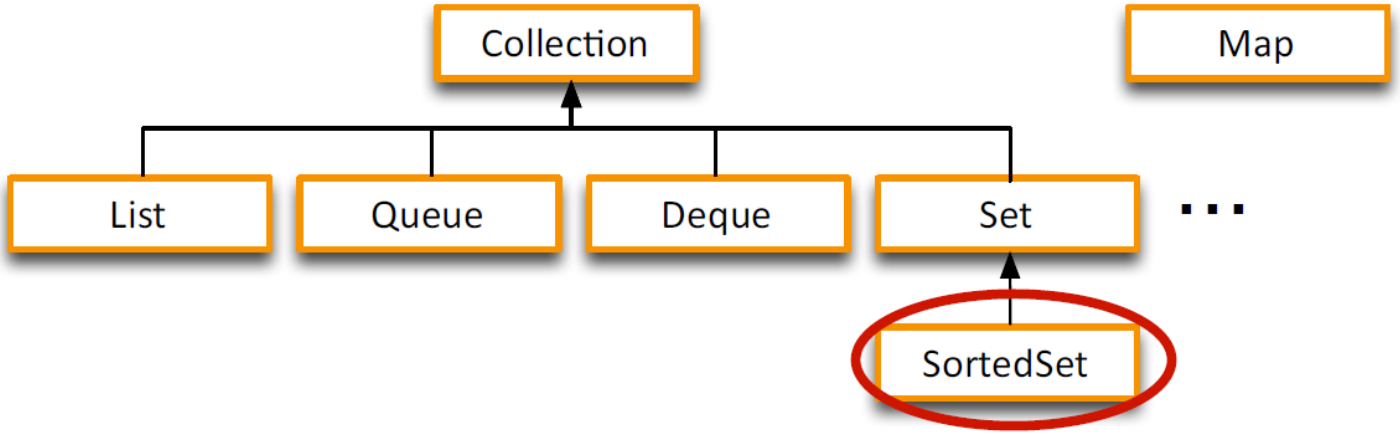
La **JDK** non prevede implementazione alcuna per questa interfaccia: prevede invece classi (come *TreeSet* ed *ArrayList*) che implementano interfacce più specifiche da essa derivate (come Set e List).

# Collection

L'interfaccia *Collection* definisce (ma non implementa) alcuni metodi base, quali :

- `boolean add (Element e)` per inserire un elemento
- `boolean remove (Element e)` per eliminare un elemento
- `int size()` restituisce il numero di elementi nella collezione
- `Bollean isEmpty()` restituisce true se è vuota

List, Queue, Set etc sono interfacce loro stesse!



# Esempio pratico: ArrayList

**ArrayList<T>** rappresenta una collezione in cui:

- posso inserire l'emento i-esimo ad un indice specifico
  - ✓ **add**(int index, E element):
    Inserts the specified element at the specified position in this list.
  - ✓ **add**(E e):
    Appends the specified element to the end of this list.
- Possono apparire duplicati.
- La classe ArrayList implementa l'interfaccia List

# Esempio pratico: ArrayList<T>

Esempio: una lista di stringhe

```
List<String> strings = new ArrayList<String>();

strings.add("Two");
strings.add("Three");
strings.add(0, "One");
strings.add(3, "One");
strings.add("Three");
strings.add(strings.size() - 1, "Two");

System.out.println(strings);
// Output: [One, Two, Three, One, Two, Three]
```

# Array VS Arraylist<T>

Array:

- Contengono solo oggetti dello stesso tipo, non possiamo avere in uno stesso Array una Stringa ed un intero
- Sono di dimensione prefissata e non estendibile.

ArrayList<T>:

- Contengono solo oggetti dello stesso tipo T, non possiamo avere in uno stesso Array una Stringa ed un intero
- La dimensione iniziale non è predeterminata: posso aggiungere o rimuovere elementi senza occuparmi della dimensione fisica, che viene scalata automaticamente.

# Esempio pratico di set: HashSet<T>

HashSet<T> rappresenta una collezione che:

- non garantisce ordinamento:
  - in un insieme non c'è la nozione di ordinamento.

- non consente duplicati: non può esistere una coppia di oggetti di tipo T, e1 ed e2, tali per cui e1.equals(e2).
  - La add() restituisce un **valore boolean**: se è falso, significa che l'elemento è già presente nell'insieme e quindi non è stato aggiunto.

- La classe HashSet<T> implementa l'interfaccia l'interfaccia Set<T>

# Esempio pratico di set: HashSet

Esempio d'utilizzo:

```
Set<String> strings = new HashSet<String>();
strings.add("One");   // true
strings.add("Two");   // true
strings.add("One");   // false: c'è gia!
strings.add("Three"); //true

System.out.println(strings.toString());
```

Output: [One, Three, Two]

# Ciclare su collezioni

Si usa, generalmente, una sintassi diversa per il *for*, che corrisponde a quella di un *foreach*:

```
List<Contatore> strings = new ArrayList<Contatore>();

...

for ( Contatore c : strings) {
      // qui "c" è una variabile
      // che rappresenta l'i-esimo contatore
      c.inc();
      System.out.println(c.getValue());
}
```

**"per ogni Contatore c nella collezione `strings`..."**

# Ciclare su collezioni

Si usa, generalmente, una sintassi diversa per il *for*, che corrisponde a quella di un *foreach*:

```
Set<String> strings = new HashSet<String>();
...

for ( String s : strings) {
      // qui "s" è una variabile
      // che contiene la stringa i-esima
      System.out.println(s);
}
```

**"per ogni Stringa c nella collezione** `strings`**..."**

# Esercizio d'esame

## Appello del 18/2/2013

Questo esercizio è simile a quello proposto in un appello passato.

Cosa manca ancora:

- Interfaccia Comparable<T>

# Esercizio 1 (1/3)

La multinazionale “D.U.M. Business” sta per avviare una massiccia campagna di vendita di frigoriferi al popolo lappone ed ha necessità di tracciare l’attività dei propri venditori. Ogni venditore è caratterizzato dal nome (lo si ipotizzi senza spazi), dal numero di giorni di servizio (almeno 1) e dal numero di frigoriferi venduti.

# Esercizio 1 (2/3)

Si scriva una classe Venditore per la "D.U.M. Business" che:

1. Possieda un opportuno costruttore con parametri.
2. Presenti opportuni metodi che permettano di accedere alle variabili di istanza dell'oggetto.
3. Possieda il metodo *aggiungiVenduti* che incrementi il numero di frigoriferi venduti di un valore passato come parametro.
4. Presenti il metodo *toString* che fornisca una descrizione del venditore.

*(... continua ...)*

# Esercizio 1 (3/3)

5. Possieda il metodo equals per stabilire l'uguaglianza con un altro oggetto Venditore (la verifica va fatta sul nome ed i giorni di servizio).
6. Definisca un metodo *compareTo* per stabilire la precedenza con un oggetto Venditore passato come parametro (in ordine crescente rispetto a numero di frigoriferi venduti). *Ovvero, tale metodo deve restituire un valore positivo, nullo o negativo* ***a seconda che*** *l'oggetto corrente abbia più, uguale o meno frigoriferi venduti di quello passato come parametro*.

*Vedremo meglio il punto 6 nella prossima esercitazione*

# Esercizio 2 (1/2)

Si scriva una classe Divisione che memorizzi le informazioni relative ad una serie di venditori della "D.U.M. Business". Per ogni divisione si memorizzi inoltre la sede operativa (es. "NordTrondelag"). La classe Divisione deve:

1. Presentare un opportuno costruttore con parametri (inizialmente, una divisione non ha venditori).
2. Possedere opportuni metodi che permettano di accedere alle variabili di istanza dell'oggetto.
3. Presentare il metodo toString che fornisca la descrizione della divisione (inclusa la descrizione di tutti i suoi venditori).

# Esercizio 2 (2/2)

4. Possedere il metodo aggiungi che, dato un oggetto Venditore, lo aggiunga a quelli gestiti dalla divisione (si noti che è possibile avere duplicati). → **che tipo di collezione?**
5. Possedere il metodo *equals* per stabilire l'uguaglianza con un altro oggetto Divisione (la verifica va fatta sulla sede operativa).
6. Possedere il metodo *totale* che restituisca il totale di frigoriferi venduti dai venditori della divisione.
7. Possedere il metodo *migliore* che restituisca il venditore con il maggior numero di frigoriferi venduti.

# Esercizio 3 (1/2)

Si scriva un'applicazione per l'agenzia "D.U.M. Business" che:

1. Crei un insieme di oggetti Divisione.
2. Crei un oggetto Divisione, lette da tastiera le informazioni necessarie.
3. Inserisca l'oggetto di cui al punto 2. all'interno dell'insieme di cui al punto 1, controllando che l'inserimento sia possibile. → **in quale situazione l'inserimento non è possible?**

# Esercizio 3 (2/2)

4. Crei un oggetto Venditore, lette da tastiera le informazioni necessarie.
5. Inserisca il venditore creato al punto 4. tra quelli associati alla divisione di cui al punto 2.
6. Stampi a video i dati del miglior venditore di tutte le divisioni. 7. Stampi a video il numero totale di frigoriferi venduti da tutte le divisioni.