Reti e Laboratorio
Modulo Laboratorio 3
AA. 2024-2025

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 1

JAVA: ripasso concetti base
classi, oggetti,
ereditarietà, polimorfismo statico e dinamico
20/9/2024

INFORMAZIONI GENERALI

- Corso A (matricole pari)
- Docente: Laura Ricci (laura.ricci@unipi.it),
- 6 CFU Reti (Prof. Paganelli), 3 CFU Laboratorio III (Prof. Laura Ricci)
- Orario Modulo Laboratorio
  - Venerdì 16.00 – 18.00 – AULA E, BYOD (Bring Your Own Device)
- Orario ricevimento
  - Venerdì 14:00 – 16:00 - Dipartimento Informatica, oppure online su Teams per appuntamento
- Materiale su Moodle e su Teams
  - slides
  - forum
  - assignments

LABORATORIO: INFORMAZIONI GENERALI

- ogni lezione (a parte le prime due lezioni di ripasso) sarà divisa in due parti
  - parte teorica: presentazione dei concetti di base
  - parte pratica
    - assegnamento di esercizi da svolgere a casa
    - verifica esercizi assegnati nelle lezioni precedenti
- consegna facoltativa degli `assignment`

MODALITA' DI ESAME

- l'esame si distingue in due prove:
  - prova di Reti
  - prova di Laboratorio
- non ci sono vincoli di precedenza tra la prova di Reti e quella di Laboratorio.
- Il voto di ciascuna prova ha validità per l'A.A. 2024/25 (entro l'appello straordinario di ottobre/novembre 2025 compreso per chi ha i requisiti per partecipare all'appello).
- voto finale: media pesata dei voti ottenuti nelle due prove e arrotondamento all'intero più vicino
- ad esempio:

Reti	Lab	Totale (2/3*reti +1/3*Lab)	VOTO
26	21	24.33…	24
26	28	26,66	27
30	18	26,00	26

MODALITA' DI ESAME

- tutte le prove d'esame prevedono obbligatoriamente l'iscrizione sul SISTEMA DI ISCRIZIONE DI ATENEO
  - chi non si iscrive entro i termini non può partecipare alla prova di esame
  - attenzione alle scadenze!!!
  - non è possibile partecipare all'esame se non si è iscritti
- chi ha già sostenuto una delle due prove parziali (i.e. prova di Reti o prova di Laboratorio) mantiene il voto parziale per l'A.A. 2024/25 (entro l'appello straordinario di ottobre/novembre 2025 compreso per chi ha i requisiti per partecipare all'appello)

PROVA DI LABORATORIO

- lo studente deve consegnare un progetto, da svolgere secondo le specifiche consegnate durante il corso.
- le specifiche del progetto sono valide fino all'appello straordinario di novembre 2025 (a questo appello può accedere solo chi ha i requisiti, controllare in segreteria didattica).
- la prova consiste in un colloquio orale che include la discussione del progetto (se sufficiente) e verifica dell'apprendimento dei concetti e contenuti presentati a lezione.
- il progetto deve essere svolto individualmente

SOSTENERE L'ESAME CON IL VECCHIO REGOLAMENTO

- reti di calcolatori e laboratorio (12 CFU)
- laboratorio (6 CFU)
  - programma A.A. 2021/22
  - progetto + discussione su contenuti programma
  - specifiche del progetto ≠ progetto nuovo regolamento
  - le specifiche del progetto sono valide fino all'appello di settembre 2025 (incluso l'appello straordinario di novembre 2025 per chi ha i requisiti per accedere)
- chi ha già sostenuto una delle due prove parziali (i.e. prova di Reti o prova di Laboratorio) mantiene il voto parziale per l'AA 2024/25 (entro l'appello straordinario di ottobre/novembre 2025 compreso per chi ha i requisiti per partecipare all'appello)

INFORMAZIONI UTILI

- prerequisiti
  - laboratorio 2, Paradigmi di Programmazione
  - dal modulo teorico di Reti: conoscenza protocollo TCP/IP
- linguaggio di programmazione di riferimento: anche se l'ultima release è la 20, facciamo riferimento a JAVA 8
  - concorrenza:
    - costrutti base,
    - JAVA.UTIL.CONCURRENT
    - concurrent collections
  - JAVA.NET
- ambiente di sviluppo di riferimento: Eclipse, ma vanno bene anche altri IDE per JAVA

INFORMAZIONI UTILI

- Materiale Didattico:
  - slide delle lezioni
  - testi consigliati (non obbligatori) per la parte relativa ai threads
    - *Bruce Eckel*, Thinking in JAVA - Volume 3 - Concorrenza e Interfacce Grafiche
    - *B. Goetz*, JAVA Concurrency in Practice, 2006
  - testi consigliati (non obbligatori) per la parte relativa alla programmazione di rete
    - *Dario Maggiorini*, Introduzione alla Programmazione Client Server, Pearson
    - *Esmond Pitt*, Fundamental Networking in JAVA
- Materiale di Consultazione:
  - *Harold*, JAVA Network Programming 3nd edition O'Reilly 2004.
  - *K.Calvert, M.Donhaoo*, TCP/IP Sockets in JAVA, Practical Guide for Programmers
  - Costrutti di base: Horstmann , Concetti di Informatica e Fondamenti di Java 2

PROGRAMMA PRELIMINARE DEL CORSO

JAVA: ripasso dei concetti base

- Classi, oggetti, ereditarietà, polimorfismo, gestione delle eccezioni, principali collezioni

Threads

- meccanismi di gestione di pools di threads, Callable
- richiami su primitive di sincronizzazione (lock)
- concurrent collections

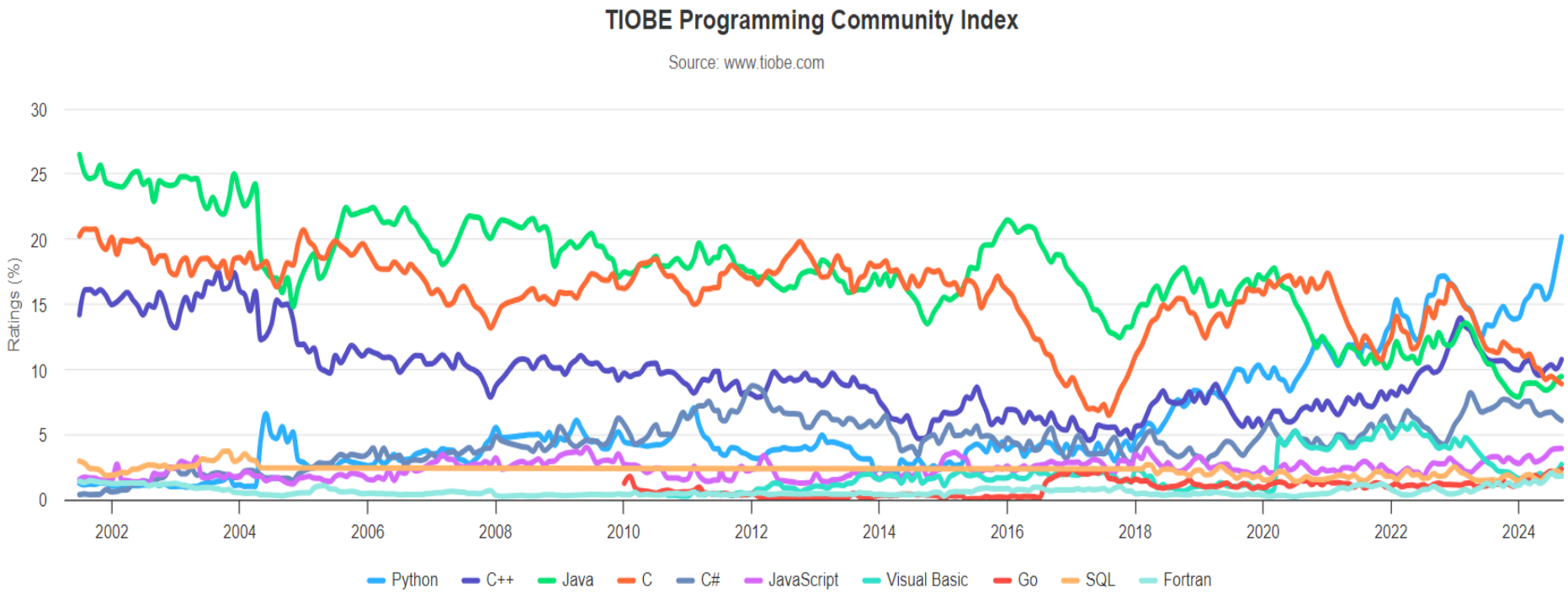
Stream-based IO

- tipi di streams, composizione di streams
- meccanismi di serializzazione: `JSON` libreria `GSON`

Programmazione di rete

- connection oriented Sockets, socket lato client e lato server
- connectionless sockets: UDP

PERCHE' JAVA?: L'INDICE TIOBE DEI LINGUAGGI



- misura la popolarità dei linguaggi di programmazione in funzione del numero di ricerche contenenti il nome del linguaggio
- tra le altre fonti `StackOverflow`
- JAVA uno dei top-4 linguaggi

JAVA: CARATTERISTICHE

- object-oriented
  - un modello chiaro e semplice per gli oggetti
- robusto
  - controlli a tempo di compilazione (strong typing) e a run-time (exception handling), gestione automatica della memoria
- multithreaded
- interpretato
  - compilato in bytecode (rappresentazione intermedia), poi interpretato
- portabile
  - JVM: ambiente di esecuzione indipendente dalla piattaforma
- distribuito
  - gestione del protocollo TCP/IP
- sicuro
  - ambiente di esecuzione ben definito e confinato

CHE TIPO DI APPLICAZIONI DI RETE?

- applicazioni client server
  - web browsers
  - SSH
  - email
  - social networks
  - teleconferences (`skype`, `Zoom`,...)
  - program development environments: GIT
  - collaborative work: Overleaf
  - multiplayer games: War of Warcraft
- applicazioni peer-to-peer
  - `P2P File sharing`: `Bittorrent`
  - blockchain: cryptocurrencies (`Bitcoin`), `NFT`,...
- scopo del corso: fornire agli studenti gli strumenti per sviluppare una semplice applicazione di rete, utilizzando costrutti ad alto livello offerti da JAVA per multithreading e networking

NETWORK APPLICATIONS

- due o più processi (non thread!) in esecuzione su hosts diversi, distribuiti geograficamente sulla rete comunicano e cooperano per realizzare una funzionalità globale
- ogni processo può essere strutturati utilizzando
  - multithreading
  - multiplexing dei canali
- i processi comunicano sulla rete: per comunicare si utilizzano protocolli, ovvero insieme di regole che i partners devono seguire per comunicare correttamente.

- in questo corso utilizzeremo i protocolli di livello trasporto:
  - connection-oriented: `TCP, Trasmission Control Protocol`
  - connectionless: `UDP, User Datagram Protocol`

INIZIAMO DAL MAIN....

```
public class MyProgram{
        public static void main(String[] args) {
                System.out.println("My First Java Program.");
        }
}
```

- salvare il programma nel file `MYProgram.java`
- a linea di comando, in una shell:

```
 > dir
MyProgram.java
> javac MyProgram.java
> dir
MyProgram.java, MyProgram.class
> java MyProgram
First Java program.
```

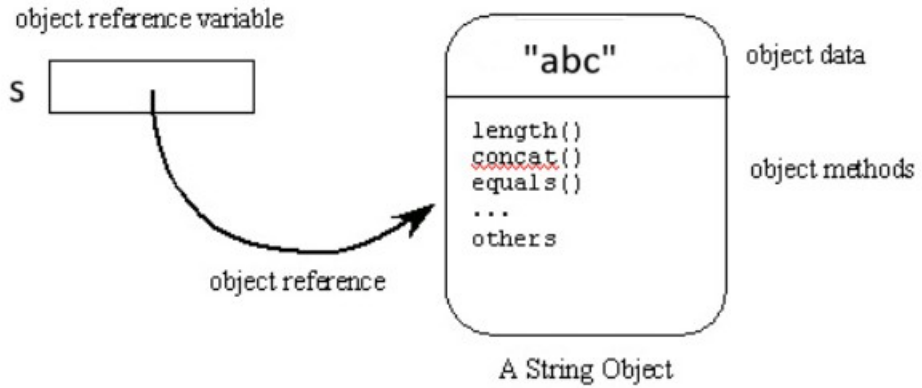
- in generale si usa un `IDE`, come `Eclipse`

GLI OGGETTI

- ogni cosa (...quasi) è un oggetto in `Java`
- `String s;`
  - dichiarazione di una variabile s che riferisce un oggetto (predefinito) di tipo `String`
  - il valore di `s` è `null`, non riferisce alcun oggetto
  - creazione di un oggetto di tipo `String` con valore iniziale "`abc`"

```
String s = new String("abc")
```

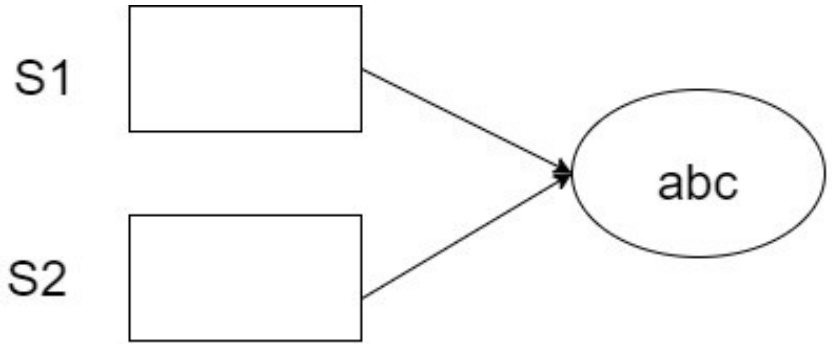
  - dopo la creazione s contiene l'indirizzo (riferimento) al nuovo oggetto, che contiene lo stato dell'oggetto e i metodi definiti su di esso



DUE VARIABILI, UN SOLO OGGETTO...

```
String s1 = new String("abc");
String s2;
s2=s1;
```

- l'assegnamento copia l'indirizzo, non il valore
- ora `s1` ed `s2` riferiscono lo stesso oggetto.



S1==S2 **true**

which also implies

S1.equals(S2) **true**

se eseguo l'assegnamento

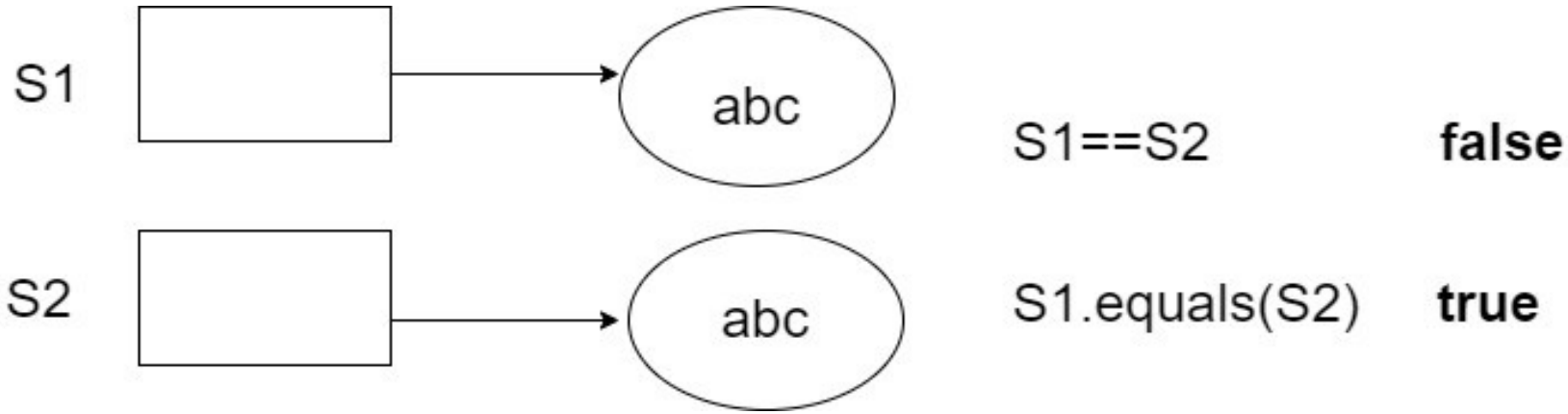
```
s1 = null
```

`s2` punta ancora all'oggetto creato

DUE VARIABILI, DUE OGGETTI...

```
String s1 = new String("abc");
String s2 = new String("abc");
```

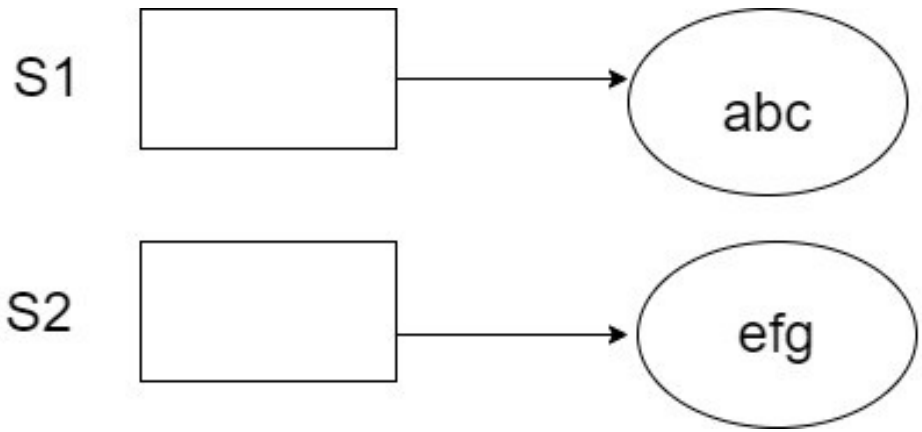
- `s2=s1` hanno lo stesso valore
  - ma non sono lo stesso oggetto...
- puntano a due oggetti diversi che possono essere cambiati in modo indipendente uno dall'altro



UNA VARIABILE, DUE OGGETTI...

```
String s1 = new String("abc");

§String s2 = new String("efg");
```



S1==S2 **false**

S1.equals(S2) **false**

UNA VARIABILE, DUE OGGETTI...

```
String s1 = new String("abc");

s1 = new String("efg");
```

- l'oggetto "abc" non è più raggiungibile da alcuna variabile
- In Java, quando un oggetto non è più raggiungibile da alcuna variabile, viene prima o poi rimosso dal garbage collector
- garbage collector è parte integrante del run-time environment

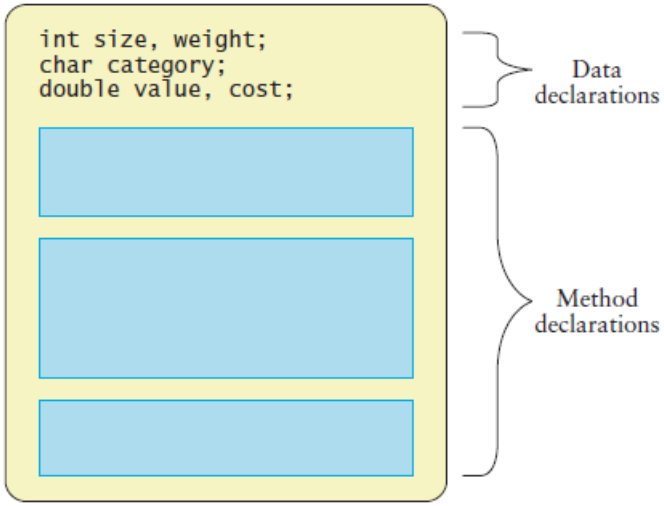
JAVA CLASSES

- classe è come un "cookie cutter", classe= cutter, cookies=istanze della classe
- "blueprint" che descrive una classe di oggetti dello stesso tipo incapsula
  - definizione dei dati (attributi)
  - operazioni su questi dati (metodi) comuni a tutti gli oggetti di quel tipo

DEFINIZIONE DI CLASSI

Syntassi

```
[AccessControlModifier] class ClassName {
                        type instance-variable-1;
                         …
                        type instance-variable-n;
                        type method-name-1(parameter-list) { … }
                        type method-name-2(parameter-list) { … }
                         …
                         type method-name-m(parameter-list) { … }
}
```



[AccessControlModifier]

- public
- private
- protected

COSTRUTTORI

- sintatticamente simili a un metodo, ma
  - il nome deve coincidere con il nome della classe
  - nessun valore di ritorno
- chiamati immediatamente dopo che l'oggetto è stato creato, ma prima del completamento dell'operatore new, danno valore alle variabili di istanza
- non possono essere ereditati (vedere prossime slides)
- Se non viene dichiarato alcun costruttore, la JVM inserisce un costruttore di default, senza parametri
  - inizializza le variabili di istanza con il corrispondente valore di default
  - automaticamente invocato quando la classe non definisce alcun costruttore,

CLASSI IN JAVA: COSTRUTTORI

```
class Box {
   double width;
   double height;
   double depth;

   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
   double volume() { return width * height * depth; }
}
```

OVERLOADING DI METODI E COSTRUTTORI

- possibile avere in una classe due o più costruttori/metodi con lo stesso nome
- stesso nome per metodi che realizzano la stessa funzionalità su dati diversi
- devono però avere firme differenziate, cioè differire per
  - numero dei parametri
  - tipo dei parametri
  - entrambi
  - non conta return type

```
public class Dog {
      public void bark() {
            System.out.println("baubau");
     }
      public void bark(int num) {
         for (int i=0; i<num; i++)
            System.out.println("baubau");
     }
  }
```

OVELOADING DI COSTRUTTORI

```
public class Circle {
   // Private instance variables
   private double radius;
   private String color;
   // Constructors (overloaded)
   // Constructs a Circle instance with default radius and color
   public Circle() {                  // 1st Constructor (default constructor)
      radius = 1.0;
      color = "red";
   }
   // Constructs a Circle instance with the given radius and default color
   public Circle(double r) {          // 2nd Constructor
      radius = r;
      color = "red";
   }
   // Constructs a Circle instance with the given radius and color
   public Circle(double r, String c) {// 3rd Constructor
      radius = r;
      color = c; }
```

OVELOADING DI COSTRUTTORI

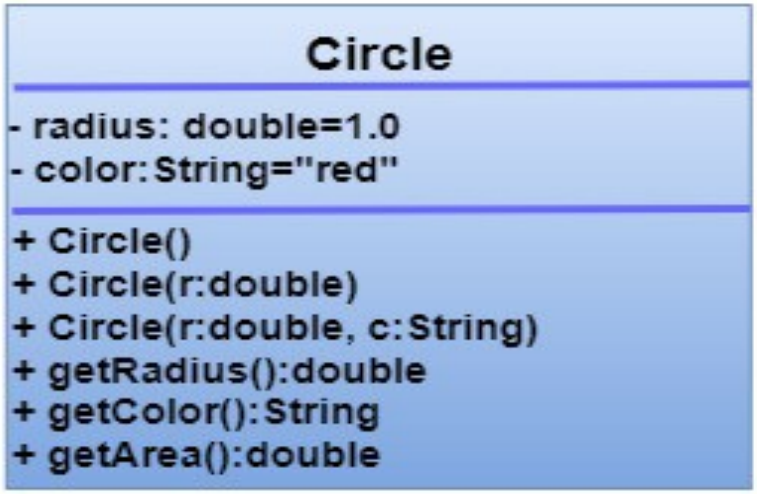
```
    // Public methods
    // Returns the radius
    public double getRadius() {  // getter for radius
       return radius;
    }
    // Returns the color
    public String getColor() {   // getter for color
       return color;
    }
    // Returns the area of this circle
    public double getArea() { // getter for Area
       return radius * radius * Math.PI;
    }
}
```

OVELOADING DI COSTRUTTORI

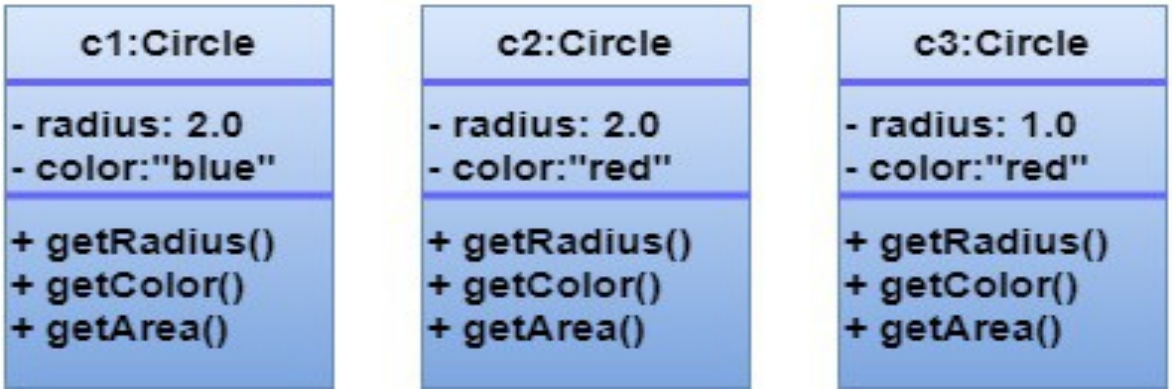
```
public class TestCircle {
  public static void main(String[] args) {   // Program entry point
     Circle c1 = new Circle(2.0, "blue");  // Use 3rd constructor
     System.out.println("The radius is: " + c1.getRadius());
     // use dot operator to invoke member methods
     //The radius is: 2.0
     System.out.println("The color is: " + c1.getColor());
     //The color is: blue
     System.out.printf("The area is: %.2f%n", c1.getArea());
     //The area is: 12.57
     Circle c2 = new Circle(2.0);  // Use 2nd constructor
     //The radius is: 2.0, the color is: red, The area is: 12.57
     Circle c3 = new Circle();  // Use 1st constructor
     //The radius is: 1.0 The color is: red The area is: 3.14 } }
```

CLASSE E ISTANZE DELLA CLASSE

Definizione della classe



Istanze della classe



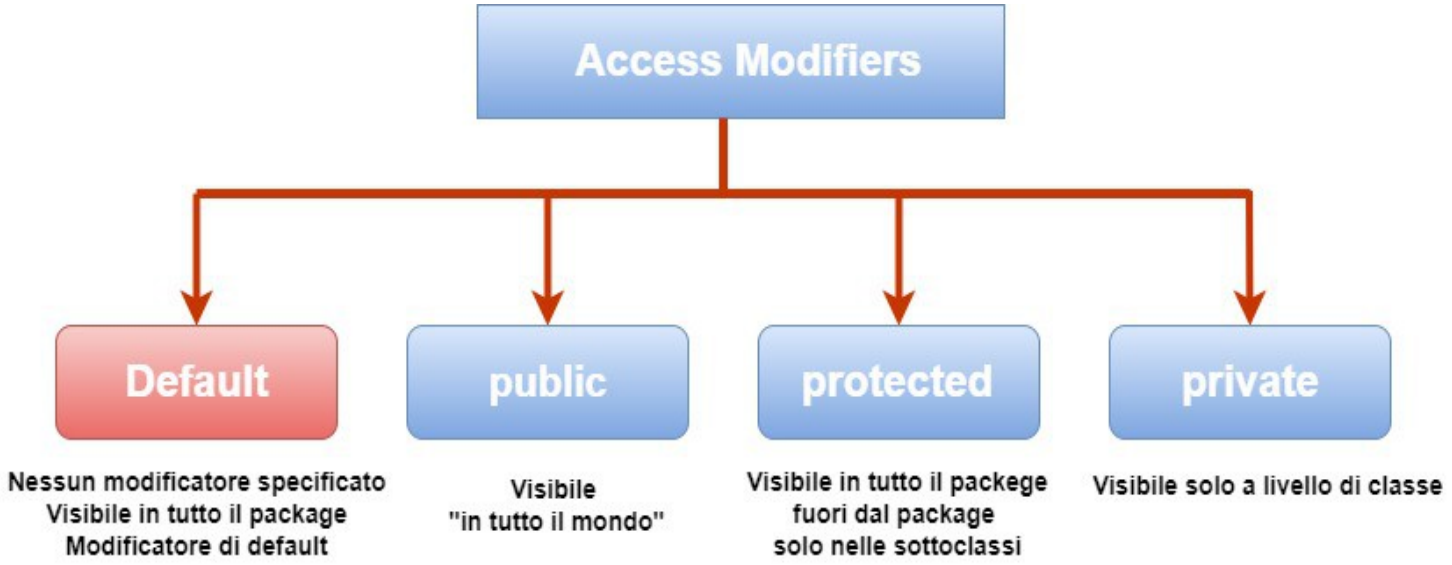
IL RIFERIMENTO "THIS"

- in ogni metodo di una istanza di una classe, oppure nel costruttore è sempre disponibile il costruttore `this, detto parametro implicito`
- riferimento all'oggetto che ha invocato il metodo
- può essere utile per distinguere gli attributi dell'oggetto dalle variabili di istanza

```
class Car{
    String color;
     …..
  void paint (String color) {
       this.color = color;
 }
}
```

MODIFICATORI DI ACCESSO

- utilizzati per controllare la visibilità di una classe, di una interfaccia, di una variabile, di un metodo o di un costruttore di una classe



- information hiding e encapsulation: per il momento
  - variabili di classe dichiarate private, accedute con metodi di accesso (getter) dichiarati pubblici
  - classi pubbliche
- in notazione UML, membri pubblici denotati con "+", privati con "-"

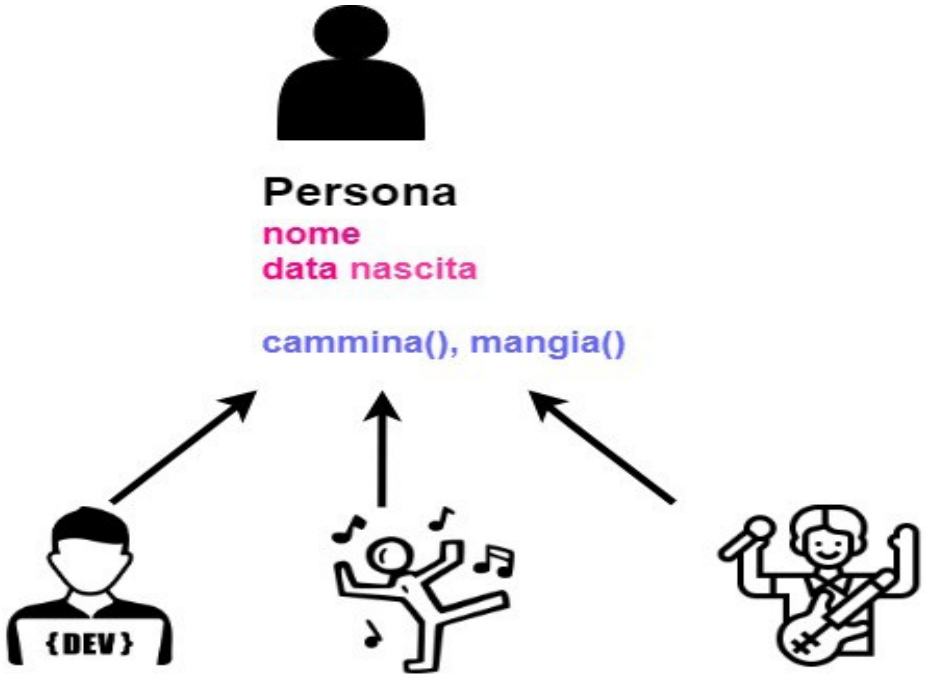
ALTRI MODIFICATORI

- `final keyword`
  - non access modifier
  - applicabile a una variabile, ad un metodo o ad una classe
- `final variable`
  - per creare valori costanti, deve essere inizializzata
- `final method`
  - non può essere sovrascritto (vedere overriding in slide successive)
- `final class`: non può essere estesa

ALTRI MODIFICATORI

- `static keyword`
  - variabile
    - appartiene alla classe e non ad una specifica istanza della classe
    - può essere utilizzata per riferire proprietà comuni a tutti gli oggetti istanze di quella classe
    - occupa un solo slot di memoria
  - method (class methods):
    - parte della definizione della classe e non di una sua istanza
    - riferito con il nome della classe, non richiede la creazione di una istanza dell'oggetto

EREDITARIETA': CONCETTO GENERALE



Programmatore
nome
data nascita
nome della azienda

cammina(), mangia()
programma()

Ballerina
nome
data nascita
nome della gruppodanza

cammina(), mangia()
danza()

Cantante
nome
data nascita
nome della band

cammina(), mangia()
canta()

EREDITARIETA' IN JAVA

- per definire una classe come estensione di un'altra si deve usare la keyword `extends`

- ogni classe ha al massimo una super classe; non esiste eredità multipla in `Java`

```
public class Ballerina extends Persona {
  .....
  }
```

  - una nuova classe viene derivata a partire dalla vecchia classe
    - la classe esistente è la super classe
    - la classe derivata è detta sotto classe
  - la sottoclasse è una versione specializzata della super classe
    - può accedere a tutti i membri non privati della superclasse
    - meccanismo di overriding: può ridefinire l' implementazione di alcuni metodi della super classe

EREDITARIETA' E COSTRUTTORI

- ogni oggetto "contiene" un'istanza della sua super classe, e questa istanza deve essere inizializzata
- Java inserisce automaticamente una chiamata al costruttore senza parametri (o al costruttore di default) della classe padre
- questa chiamata viene inserita come prima istruzione in ogni costruttore delle sottoclasse
- l'esecuzione dei costruttori avviene quindi top-down nella gerarchie della ereditarietà
- in questo modo, quando un metodo della sottoclasse è eseguito (incluso il costruttore), la super-classe risulta completamente inizializzata

COSTRUTTORI ED EREDITARIETA'

```
class A {
     A() {
           System.out.println("Inside A's constructor.");
         }
  }
 class B extends A {
     B() {
           System.out.println("Inside B's constructor.");
         }
 }
class C extends B {
     C() {
           System.out.println("Inside C's constructor.");
     }
}
public class CallingCons {
        public static void main(String args[]) {
        C c = new C();
        }
}
```

```
Stampa

Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

IL RIFERIMENTO SUPER

- parola chiave riservata che può essere utilizzata per riferire la classe padre
- permette l'accesso ai membri della classe padre, se non `private`

```
class Parent {
      int parentVar = 10;
 }

class Child extends Parent {

 int childVar = 20;

 void display() {
     System.out.println("Child Variable: " + childVar);
     System.out.println("Parent Variable: " + super.parentVar);
     }
 }
```

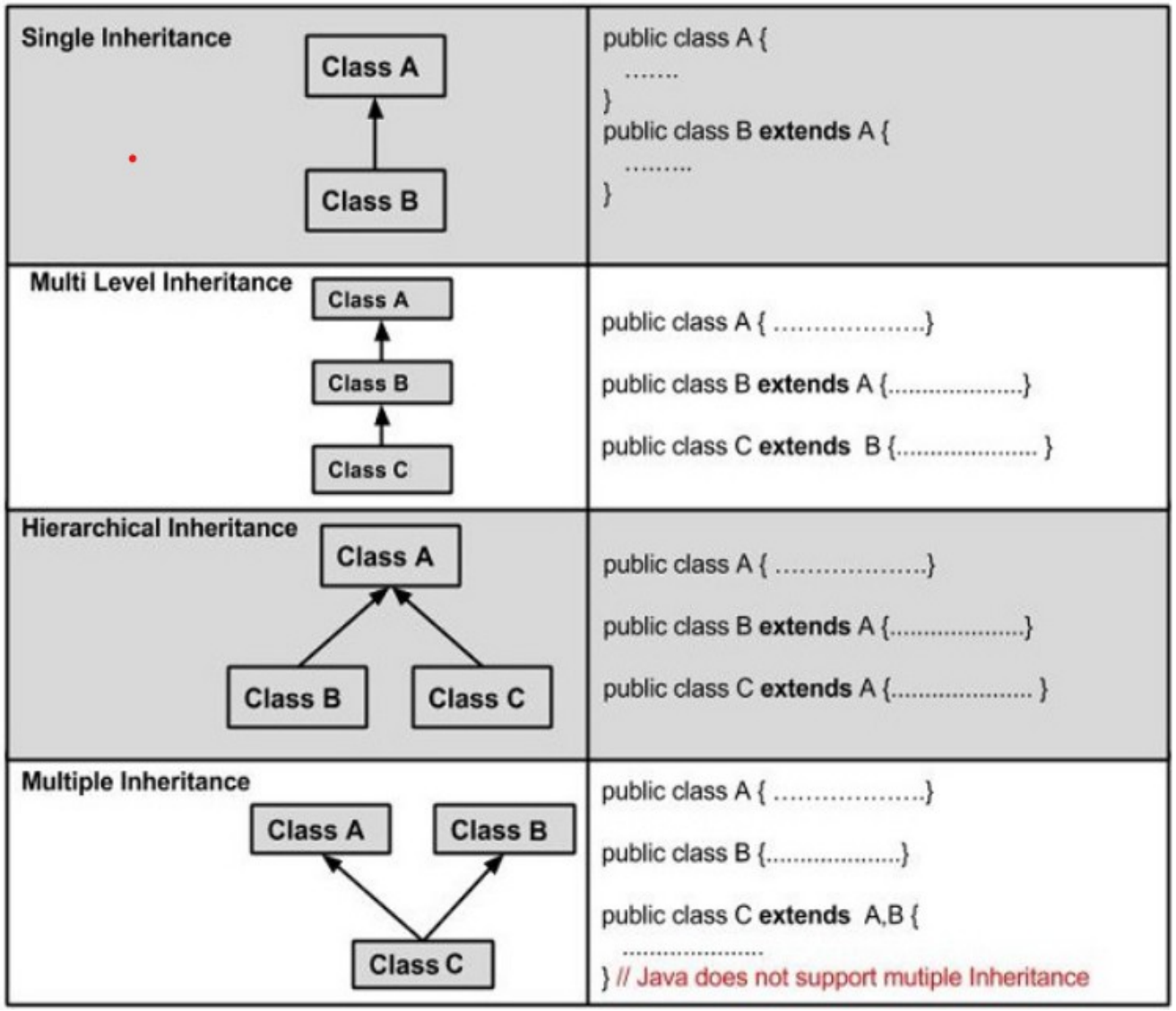
IL RIFERIMENTO SUPER: USO IN COSTRUTTORI

- utile quando una super classe definisce esplicitamente costruttori con argomenti e non esiste un costruttore senza argomenti nella super classe

```
class Employee{
   private String name;
   private double wage;
   Employee (String n, double w){
       name=n;
       wage=w; } }
class Manager extends Employee{
   private int unit;
   Manager(String n, double w, int u) {
         super(n,w);
         unit=u; } }
```

- attenzione: il costruttori di default non è significativo se viene inserito un costruttore esplicitamente!

TIPI DI EREDITARIETA'



CLASSI ASTRATTE

```
package Solidi;
public class Sfera {
   private double raggio ;
   private double pesoSpecifico ;
   public Sfera ( double raggio , double ps ) {
      this.raggio = raggio ;
      pesoSpecifico = ps ;
   }
   public double volume () {
      return 4/3 * Math . PI * Math.pow ( raggio ,3);
   }
   public double superficie () {
      return 4 * Math . PI * raggio * raggio ;
   }
   public double peso () {
     return pesoSpecifico * volume ();
   }
}
```

CLASSI ASTRATTE

```
package Solidi;
public class Cubo {
   private double lato ;
   private double pesoSpecifico ;
   public Cubo (double lato, double ps ) {
      this.lato = lato ;
      pesoSpecifico = ps ;
   }
   public double volume () {
      return Math.pow ( lato ,3);
   }
   public double superficie () {
      return 6* lato * lato ;
    }
   public double peso () {
      return pesoSpecifico * volume ();
    }
}
```

CLASSI ASTRATTE

- possiamo usare l'ereditarietà: le classi `Sfera` e `Cubo` hanno diverse cose in comune
- potremmo creare una classe `Solido` che raggruppa in membri comuni di `Sfera` e `Cubo`, quella classe dovrebbe contenere:
  - la variabile `pesoSpecifico`
  - il metodo `peso()` identico nelle due classi
  - i metodi `volume()` e `superficie()`, perchè:
    - tutti i solidi hanno un volume e una superficie
    - il metodo `peso()` invoca `volume()`
- ma come si calcolano superficie e volume di un solido generico?
  - ogni solido ha le sue formule...

CLASSI ASTRATTE

- soluzione: definire i `volume()` e `superficie()` come metodi astratti
  - un metodo astratto è un metodo che prevede solo una intestazione, ma che non è implementato
  - una classe che contiene (almeno) un metodo astratto è una classe astratta

```
public abstract class Solido { // classe astratta
     private double pesoSpecifico ;
     public Solido ( double ps ){ pesoSpecifico = ps ; }
     public double peso () {
         return volume () * pesoSpecifico ;
     }
     public abstract double volume (); // metodo astratto
     public abstract double superficie (); // metodo astratto
   }
```

- notare che nel metodo `peso()` si può richiamare `volume()` anche se è un metodo astratto

CLASSI ASTRATTE

- una classe astratta
  - viene definita mediante il modificatore `abstract`
  - non può essere usata per istanziare oggetti, perchè la classe non è completa
  - può solo essere estesa da un'altra classe che ne definisce i metodi astratti o da un'altra classe astratta
  - può prevedere costruttori che saranno richiamati dalle sottoclassi

CLASSI ASTRATTE

```
public class Sfera extends Solido {
    private double raggio ;
    //peso specifico è efinito nella superclasse
    public Sfera (double raggio , double ps ) {
         super ( ps );
         this.raggio = raggio ;
     }
    public double volume () {
         return 4/3 * Math.PI * Math . pow ( raggio ,3);
     }
     public double superficie () {
        return 4 * Math.PI * raggio * raggio ;
     }
     //peso è definito nella superclasse
     public String toString () {
        return " Sfera ("+ raggio +")";
     }
}
```

CLASSI ASTRATTE

```
public class Cubo extends Solido {
  private double lato ;
  // pesoSpecifico e' definito nella superclasse
  public Cubo ( double lato , double ps ) {
    super ( ps );
    this.lato = lato ;
  }
  public double volume () {
    return Math.pow ( lato ,3);
   }
  public double superficie () {
    return 6* lato * lato ;
   }
   //peso è definito nella superclasse
  public String toString (){
    return " Cubo ["+ lato +"]";}}
```

CLASSI ASTRATTE

```
package Solidi;
import java . util . Scanner ;
public class UsaSolido {
  public static void main ( String args []) {
    Scanner input = new Scanner ( System . in );
    System.out.println (" Vuoi creare una sfera o un cubo (s/c)?");
    char scelta = input . nextLine (). charAt (0);
     if ( scelta == 's' || scelta == 'c') {
         System . out . print (" Peso Specifico ? ");
         double ps = input . nextDouble ();
         Solido sol ; // variabile di tipo Solido
         if ( scelta == 's') {
             System . out . print (" Raggio ? ");
             double raggio = input . nextDouble ();
             sol = new Sfera ( raggio , ps );
} else {//...continua pagina successiva
```

CLASSI ASTRATTE

```
} else {
      System . out . print (" Lato ? ");
      double lato = input . nextDouble ();
      sol = new Cubo ( lato , ps );
}
System . out . println ( "Ho creato un solido " + sol +
                    " con volume " + sol . volume () +
                    " e peso " + sol . peso () );
}
}
```

INTERFACCE

- abbiamo visto che le classi astratte includono alcuni metodi astratti
- estremizzando l'uso dei metodi astratti potremmo avere una classe astratta dotata solo di metodi astratti
- un'interfaccia si definisce in maniera simile a una classe astratta, ma:
  - è costituita da soli metodi astratti (è completamente non definita)
  - non usa il modificatore `abstract`
  - usa la parola chiave `interface` al posto di `class`
- inoltre, una classe che implementa i metodi dell'interfaccia deve usare la parola chiave `implements` invece che `extends`

INTERFACCE

- quale è il vantaggio di una interfaccia rispetto a una classe astratta?
  - una classe può estendere una sola classe (astratta o meno)
  - una classe può implementare tante interfacce

OVERRIDING

- consente a una sottoclasse o classe figlia di fornire un'implementazione specifica di un metodo che è già fornito da una delle sue superclassi nella gerarchia della ereditarietà
- un metodo nella sottoclasse sovrascrive il metodo nella superclasse.
  - deve avere lo stesso nome, gli stessi parametri e lo stesso tipo di ritorno (o sottotipo) di un metodo nella sua superclasse,
- la versione del metodo che viene eseguita viene determinata dall'oggetto che invoca quel metodo

OVERRIDING: UN ESEMPIO

```
package MoveVehicle


public class Vehicle
    {
     public void move()
        {
        System.out.println("la mia velocità dipende da chi sono");
       }
    }
```

OVERRIDING: UN ESEMPIO

```
package MoveVehicle;

    class Bike extends Vehicle
      {
        public void move()
       {
           System.out.println("sono meno veloce di una macchina");
        }
     }
```

OVERRIDING: UN ESEMPIO

```
package MoveVehicle;


   class Car extends Vehicle
     {
      public void move()
        {
          System.out.println("sono più veloce della bici");
        }
     }
```

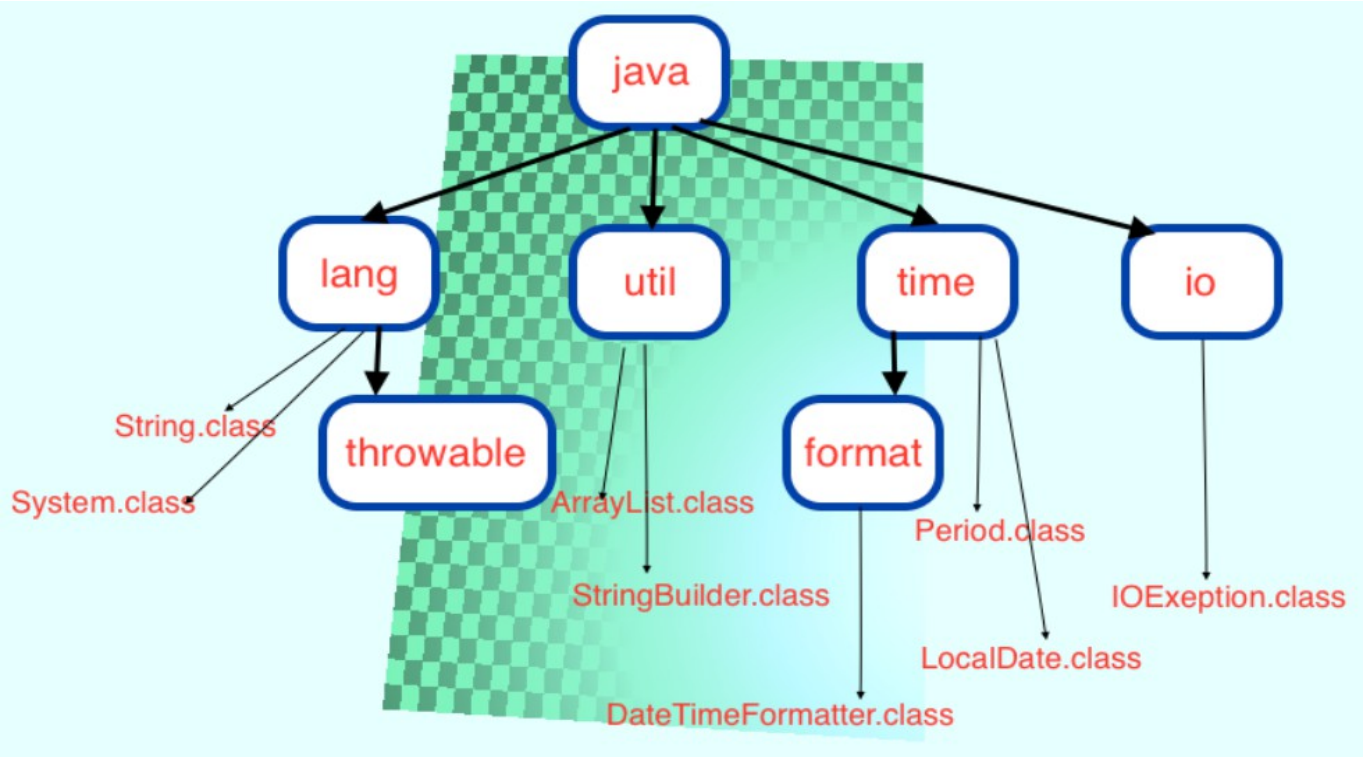
OVERRIDING: UN ESEMPIO

```
package MoveVehicle;
    public class OverrideTester {
       public static void main(String[] args)
          {
            Vehicle v1 = new Bike();
            v1.move();
            Vehicle v2 = new Car();
            v2.move();
            Vehicle v3 = new Vehicle();
            v3.move();
          }
    }
```

```
Stampa:
sono meno veloce di una macchina
sono più veloce della bici
la mia velocità dipende da chi sono
```

PACKAGES

- consentono di raggruppare classi
- la libreria standard di `Java` (`Java API`) è organizzata in packages



- spesso è utile raggruppare anche le proprie classi in packages

PACKAGES

- per specificare in quale package debba essere inclusa una classe occorre usare la direttiva package all'inizio del file sorgente della classe

```
package server ;
     public class serverimplementation {
     .....
 }
```

- i file delle varie classi vengono salvati in directory diverse che corrispondono ai diversi package
- generate automaticamenti dall'IDE (esempio Eclipse)
- possibile definire anche una struttura gerarchica dei package

LA DIRETTIVA IMPORT

- quando si vuole utilizzare una classe che appartiene ad un package diverso da quello corrente bisogna utilizzare la direttiva `import`

```
package myPackage;
public class MyClass
     { public void printNames(String s)
                { System.out.println(s);}
     }

import myPackage.MyClass;
public class PrintName
{public static void main(String args[])
    { String name = "LaboratorioDiReti";
      MyClass obj = new MyClass();
      obj.printNames(name);
     } }
```

ASSIGNMENT N.1

- considerare un'azienda nella cui organizzazione sono coinvolte diverse persone, con i seguenti diversi ruoli
  - impiegati livello 1: hanno uno stipendio mensile base
  - impiegati livello 2: ottengono un bonus da aggiungere allo stipendio mensile base degli impiegati di primo livello
  - lavoratori a ore: vengono pagati una cifra standard per ogni ora lavorata
  - volontari: non percepiscono alcuna paga
- tutte le persone coinvolte nella organizzazione sono caratterizzate dal nome, l'indirizzo ed il numero di telefono.
- per tutti gli impiegati di livello 1 e di livello 2 livello e per i lavoratori a ore viene registrato anche il codice fiscale e un campo che riporta lo stipendio base per gli impiegati di livello 1 e di livello 2 e la paga oraria per i lavoratori a ore

ASSIGNMENT N.1

- si scriva un programma JAVA che
  - crei uno staff contenente un numero prefissato (dato in input) di lavoratori per ognuno dei tipi precedenti, impostando, in fase di creazione, lo stipendio mensile standard per gli impiegati di livello 1 e 2, il bonus per quelli di livello 1 e la cifra oraria per i lavoratori a ore
  - quindi calcoli e stampi la paga per tutte le persone coinvolte nell'azienda, stampando per i volontari la stringa "`Grazie!`", invece dello stipendio
  - si deve considerare il valore dello stipendio mensile, calcolato in base ai valori registrati in fase di creazione, e alle ore lavorate per i lavoratori a ore
- il programma
  - deve essere strutturato in un insieme di classi organizzate gerarchicamente
  - deve utilizzare il polimorfismo per la definizione del metodo che calcola lo stipendio di ogni membro dell'organizzazione