

JAVA - Classi e oggetti

Metodi Avanzati di Programmazione
Laurea Triennale in Informatica
Università degli Studi di Bari Aldo Moro
Docente: Pierpaolo Basile

Sommario

- Dichiarazione di una classe
- Dichiarazione di variabili e metodi
- Costruttori
- Utilizzare i metodi
- Interfacce ed ereditarietà
- Classi astratte

Dichiarazione di una classe...

```
class MyClass {
```

```
    // attributi
```

Stato della classe



```
    // costruttori
```

Creazione (inizializzazione) di un oggetto di questa classe

```
    // metodi
```

Funzionalità della classe

```
}
```

...dichiarazione di una classe

```
<visibilità> class <nome> extends <superclass>  
implements <interface1>, <interface2> ... {  
//body  
}
```

- visibilità: *public/private/protected*: la modalità con cui la classe è visibile alle altre classi
- superclass: la classe da cui eredità
- interface: se implementa o meno delle interfacce

Variabili all'interno di una classe

- Field/attributi: variabili visibili in tutta la classe
- Locali: visibili solo nel metodo in cui sono utilizzate
- Parametri: passate quando si chiama un metodo

Dichiarazione degli attributi

```
public class Bicycle {  
    private int gear = 1;  
    private int speed;  
}
```

valore (se non specificato assumono il valore di default)

nome

tipo

Visibilità

- private: visibili solo all'interno della classe
- public: visibili alle altre classi
- protected: visibili al package o sottoclassi

Dichiarazione dei metodi

```
<visibilità> <tipo> <nome> ( parametri, ... ) {  
    //body  
}
```

Il tipo può essere **void** che indica nessun valore
(il metodo non restituisce alcun valore)

public: visibile alle altre classi
private: visibile solo all'interno della classe
protected: visibili al package o sottoclassi

```
public int sum(int a, int b) {  
    return a+b;  
}
```

Nome dei metodi

- In minuscolo
- Dovrebbero iniziare con un verbo
 - I metodi indicano una funzione
- Se composti da più parole, utilizzare la maiuscola per identificare le parole
 - Es. getGear

Overloading dei metodi

- Una classe può avere più metodi con lo stesso nome purché abbiano una lista differente di parametri
- I metodi con lo stesso nome **devono restituire lo stesso tipo valore**

```
public int sum(int a, int b) {  
    return a+b;  
}
```

```
public int sum(int a, int b, int c) {  
    return a+b+c;  
}
```

Parametri

- Ogni metodo può avere **0** o N parametri
- Un parametro può essere di qualsiasi tipo
 - un tipi primitivo
 - oggetti di altre classi (array, String, ...)
- I parametri devono avere nomi differenti
 - Non potete dichiarare variabili locali con lo stesso nome di un parametro
 - Un parametro può avere lo stesso nome di un attributo della classe
 - in questo caso rende **NON** visibile l'attributo al metodo almeno che non utilizzate *this*

Costruttori...

- Sono dei metodi che servono ad inizializzare gli oggetti di una classe
- Hanno lo stesso nome della classe
 - non restituiscono alcun valore

```
public Bicycle(int startSpeed, int startGear) {  
    gear = startGear;  
    speed = startSpeed;  
}
```

Per creare un oggetto di tipo Bicycle devo chiamare il suo costruttore con *new*

```
Bicycle myBike = new Bicycle(0, 3);
```

...Costruttori...

- Una classe può avere più di un costruttore

```
public Bicycle(int startSpeed, int startGear) {  
    gear = startGear;  
    speed = startSpeed;  
}
```

```
public Bicycle() {  
    gear = 1;  
    speed = 0;  
}
```

Non si possono dichiarare due costruttori che hanno lo stesso numero e tipo di parametri!!!

...Costruttori

- Una classe può non avere costruttori
 - in questo caso eredita il costruttore senza parametri della sua superclasse
 - se la superclasse non ha un costruttore senza parametri eredita quello di Object
- Anche i costruttori possono essere private/public/protected in base alla loro visibilità all'interno o all'esterno della classe

Oggetti

- Gli oggetti vengono creati utilizzando l'istruzione ***new*** e invocando un costruttore della classe a cui deve appartenere l'oggetto

```
Bicycle myBike = new Bicycle(0, 3);
```

- E' possibile accedere agli attributi e ai metodi dell'oggetto attraverso il .

```
myBike.gear  
myBike.getSpeed()
```

L'istruzione *this*

- ***this*** permette di accedere a costruttori e attributi della classe

```
public class Bicycle {  
    private int gear;  
    private int speed;  
    public Bicycle(int startSpeed, int startGear) {  
        this.gear = startGear;  
        this.speed = startSpeed;  
    }  
  
    public Bicycle() {  
        this(3, 0);  
    }  
}
```

L'istruzione *this*

- ***this*** permette di accedere a costruttori e attributi della classe

```
public class Bicycle {  
    private int gear;  
    private int speed;  
    public Bicycle(int startSpeed, int startGear) {  
        this.gear = startGear;  
        this.speed = startSpeed;  
    }  
  
    public dummy(int g) {  
        int gear = 3;  
        this.gear = g;  
    }  
}
```


Ricapitolando...una calcolatrice

```
public class Calculator {
```

dichiarazione della classe



```
    private double memory; //memoria della calcolatrice
```

```
    public Calculator() {  
    }
```

costruttori



```
    public Calculator(double memory) {  
        this.memory = memory;  
    }
```

Ricapitolando...una calcolatrice

```
public double getMemory() {  
    return memory;  
}
```

```
public void setMemory(double memory) {  
    this.memory = memory;  
}
```

```
public void resetMemory() {  
    this.setMemory(0);  
}
```

metodi

Ricapitolando...una calcolatrice

```
public double sum(double a, double b) {  
    return a + b;  
}  
public double diff(double a, double b) {  
    return a - b;  
}  
public double mul(double a, double b) {  
    return a * b;  
}  
public double div(double a, double b) {  
    return a/b;  
}  
public double sqrt(double n) {  
    return Math.sqrt(n);  
}  
}
```

metodi

← Stiamo invocando un metodo statico di Math

Utilizziamo la calcolatrice

```
public class TestCalculator {  
    public static void main(String[] args) {  
        Calculator calc1=new Calculator();  
        System.out.println(calc1.getMemory());  
        double a = calc1.diff(3, 1);  
        calc1.setMemory(a);  
        System.out.println(calc1.sqrt(49)+calc1.getMemory());  
  
        Calculator calc2=new Calculator(100);  
        System.out.println(calc2.getMemory());  
        System.out.println(calc2.sum(3, 2));  
    }  
}
```

Utilizziamo la calcolatrice

```
public class TestCalculator {  
    public static void main(String[] args) {  
        Calculator calc1=new Calculator();  
        System.out.println(calc1.getMemory());  
        double a = calc1.diff(3, 1);  
        calc1.setMemory(a);  
        System.out.println(calc1.sqrt(49)+calc1.getMemory());  
  
        Calculator calc2=new Calculator(100);  
        System.out.println(calc2.getMemory());  
        System.out.println(calc2.sum(3, 2));  
    }  
}
```

Sono due istanze diverse di Calculator

Utilizziamo la calcolatrice

```
public class TestCalculator {  
    public static void main(String[] args) {  
        Calculator calc1=new Calculator();  
        System.out.println(calc1.getMemory());  
        double a = calc1.diff(3, 1);  
        calc1.setMemory(a);  
        System.out.println(calc1.sqrt(49)+calc1.getMemory());  
  
        Calculator calc2=new Calculator(100);  
        System.out.println(calc2.getMemory());  
        System.out.println(calc2.sum(3, 2));  
    }  
}
```

Invoco i costruttori di Calculator



Utilizziamo la calcolatrice

```
public class TestCalculator {  
    public static void main(String[] args) {  
        Calculator calc1=new Calculator();  
        System.out.println(calc1.getMemory());  
        double a = calc1.diff(3, 1);  
        calc1.setMemory(a);  
        System.out.println(calc1.sqrt(49)+calc1.getMemory());  
  
        Calculator calc2=new Calculator(100);  
        System.out.println(calc2.getMemory());  
        System.out.println(calc2.sum(3, 2));  
    }  
}
```

Invoco i metodi di Calculator

Dettagli sulla visibilità

Modificatore	Class	Package	Subclass	World
public	X	X	X	X
private	X			
protected	X	X	X	
nessuno	X	X		

Numero arbitrario di argomenti

- E' possibile definire in un metodo un numero arbitrario di argomenti dello stesso tipo

```
public void print(String... s) {  
    for (String item:s) {  
        System.out.println(item);  
    }  
}
```

s è visto come un array

```
print("Pippo", "Topolino", "Pluto")
```

Posso invocare *print* passando tanti oggetti di tipo String

static

- Con il modificatore *static* posso definire variabili (attributi) e metodi a livello di classe
 - Una variabile *static* è la stessa per tutte le istanze di quella classe
 - I metodi *static* possono essere chiamati senza creare un'istanza della classe `Math.sqrt(49);`

static...esempio

```
public class Person {  
    static int numbOfPersons=0;  
    private String name;  
    private String surname;  
    public Person(String name, String surname) {  
        this.name=name;  
        this.surname=surname;  
        ++numbOfPersons;  
    }  
}
```

static...esempio

```
Person p1=new Person("Pippo", "Rossi");  
Person p2=new Person("Topolino", "Bianchi");  
System.out.println(Person.numbOfPersons);
```



Accedo al valore della variabile
static della classe Person

Costanti

- E' possibile definire degli attributi costanti
 - non posso cambiare valore
 - si utilizza il modificatore *final*

`private final int A = 3;`

Visibile solo nella classe

`public final int X = 4;`

Visibile ad altre classi

`public static final double PI=3.14`

Visibile ad altre classi in modo static (senza dover creare un'istanza della classe)

Classi innestate...

- E' possibile definire una classe all'interno di un'altra classe

```
class OuterClass {  
    ...  
    class NestedClass {  
        ...  
    }  
}
```

...Classi innestate

```
class OuterClass {
```

```
...
```

```
static class StaticNestedClass {
```

```
...
```

```
}
```

```
class InnerClass {
```

```
...
```

```
}
```

```
}
```

Non ha accesso alle risorse di OuterClass

Ha accesso alle risorse di OuterClass anche se dichiarate *private*

Quando utilizzare le classi innestate?

1. Quando la inner class è utile solo alla outer class
2. Una classe B deve accedere a risorse private di A, quindi incapsuliamo B dentro A in modo da mantenere private le risorse di A
3. Rendere più leggibile il codice

Tipi enumerativi...

- Permettono di definire dei tipi che possono assumere solo un set predefinito di costanti

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY  
}
```

Quando si crea un tipo enum JAVA crea automaticamente una classe di tipo enum che mette a disposizione dei metodi e permette di definire anche dei costruttori e nuovi metodi

...Tipi enumerativi...

- `Day.values()` restituisce tutti i valori che può assumere `Day`

```
for (Day d : Day.values()) {  
    System.out.println(d);  
}
```

...Tipi enumerativi

```
public enum Planet {  
    MERCURY (3.303e+23, 2.4397e6),  
    VENUS  (4.869e+24, 6.0518e6),  
    EARTH  (5.976e+24, 6.37814e6),  
    MARS   (6.421e+23, 3.3972e6),  
    JUPITER (1.9e+27, 7.1492e7),  
    SATURN  (5.688e+26, 6.0268e7),  
    URANUS  (8.686e+25, 2.5559e7),  
    NEPTUNE (1.024e+26, 2.4746e7);  
  
    private final double mass; // in kilograms  
    private final double radius; // in meters  
    Planet(double mass, double radius) {  
        this.mass = mass;  
        this.radius = radius;  
    }  
    public double mass() { return mass; }  
    public double radius() { return radius; }  
}
```

Definisco i valori che può assumere questo enum chiamando il costruttore

Definisco un costruttore per questo tipo enum

Definisco dei metodi

INTERFACCE ED EREDITARIETÀ

Interfacce

- Le interfacce permettono di definire il funzionamento di una classe
 - definiscono quali metodi deve avere una classe per **aderire** all'interfaccia (contratto)
- Contengono solo la descrizione dei metodi e non la loro implementazione
 - possono contenere costanti
- Permettono di definire delle classi che hanno stesse funzionalità ma ognuna le implementa in maniera differente

Dichiarazione interfacce

```
public interface <name> extends <interface1>,  
<interface2>, <interface3> {
```

```
// constant declarations
```

```
// method signatures
```

```
void methodA(int i, double x);
```

```
int methodB(String s);
```

```
}
```

Solo dichiarazione del
metodo, nessuna
implementazione

Implementazione interfacce

```
public class <name> implements <interface> {
```

```
    //implementazione di tutti i metodi definiti in  
    <interface>
```

```
}
```

Esempio...la VAT (IVA)

```
public interface Vat {  
    public double computeVat(double b);  
}
```

```
public class ItalianVat implements Vat {  
    public double computeVat(double b) {  
        return b * 0.22;  
    }  
}
```

```
public class GermanyVat implements Vat {  
    public double computeVat(double b) {  
        return b * 0.19;  
    }  
}
```


Esempio...la VAT (IVA)

```
Vat italianVat=new ItalianVat();
```

```
System.out.println(italianVat.computeVat(10));
```

```
Vat germanyVat=new GermanyVat();
```

```
System.out.println(germanyVat.computeVat(10));
```

Esempio...la VAT (IVA)

Vat italianVat=new ItalianVat();

System.out.println(italianVat.computeVat(10);

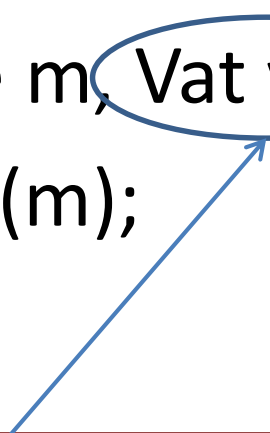
Vat germanyVat=new GermanyVat();

System.out.println(germanyVat.computeVat(10);

Vat è un tipo di oggetto valido può essere utilizzato anche come tipo di parametro in un metodo

Esempio...la VAT (IVA)

```
public class VatCalculator {  
    public double addVat(double m, Vat vat) {  
        return m + vat.computeVat(m);  
    }  
}
```



L'interfaccia Vat è utilizzata come tipo di un parametro

Ereditarietà

- In JAVA le classi possono ereditare da altre classi, ciò implica
 - la classe eredita tutti gli attributi (*public* e *protected*) della superclasse
 - la classe eredita tutti i metodi (*public* e *protected*) della superclasse
- Tutte le classi in JAVA ereditano dalla classe Object

La calcolatrice

```
public class Calculator {  
    private double memory;  
    public Calculator() {  
    }  
    public Calculator(double memory) {  
        this.memory = memory;  
    }  
    public double getMemory() {  
        return memory;  
    }  
    public void setMemory(double memory) {  
        this.memory = memory;  
    }  
    public void resetMemory() {  
        this.setMemory(0);  
    }  
}
```

La calcolatrice

```
public double sum(double a, double b) {  
    return a + b;  
}  
public double diff(double a, double b) {  
    return a - b;  
}  
public double mul(double a, double b) {  
    return a * b;  
}  
public double div(double a, double b) {  
    return a/b;  
}  
}
```

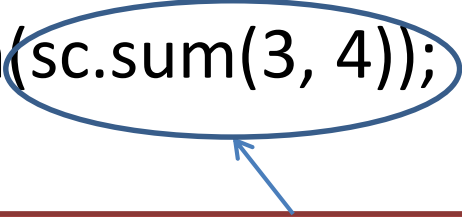
La Calcolatrice scientifica

```
public class ScientificCalculator extends Calculator {  
    public double tan(double x) {  
        return Math.tan(x);  
    }  
    public double sin(double x) {  
        return Math.sin(x);  
    }  
}
```

ScientificCalculator estende Calculator. Quindi sarà di tipo Calculator ed erediterà tutti gli attributi e i metodi pubblici di Calculator

La calcolatrice scientifica

```
ScientificCalculator sc=new ScientificCalculator();  
System.out.println(sc.sin(4));  
System.out.println(sc.sum(3, 4));
```



ScientificCalculator eredità il metodo pubblico *sum* della classe padre Calculator

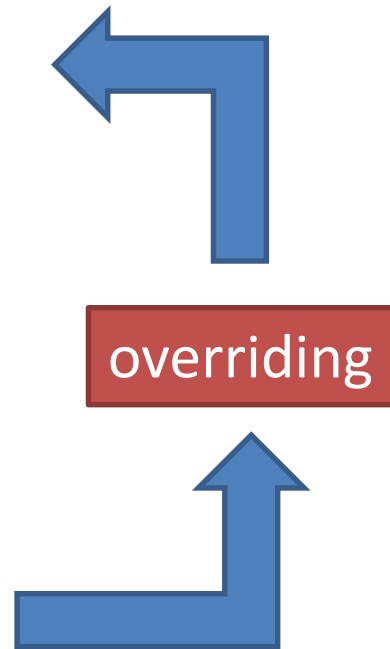
Overriding...

- Una classe può ridefinire un metodo della sua superclasse (classe padre)
 - in questo caso il metodo della superclasse viene nascosto e viene invocato quello della classe
 - questo fenomeno si chiama overriding

...Overriding

```
public class ClassA {  
    public void printMe() {  
        System.out.println("lo sono A");  
    }  
    public void sayHello() {  
        System.out.println("Hello!");  
    }  
}
```

```
public class ClassB extends ClassA {  
    public void printMe() {  
        System.out.println("lo sono B");  
    }  
}
```



Polimorfismo

- Le sotto classi possono riscrivere alcuni comportamenti della superclasse pur avendo ancora delle caratteristiche in comune
- La capacità di ogni sottoclasse di poter ridefinire i comportamenti della superclasse è detta polimorfismo

Polimorfismo

```
ClassA a=new ClassA();
```

```
ClassA b=new ClassB();
```

```
a.sayHello();
```

```
b.sayHello();
```

```
a.printMe();
```

```
b.printMe();
```

ClassB estende ClassA quindi è di tipo ClassA

In questo caso viene chiamato il printMe ridefinito da ClassB anche se b è di tipo ClassA

Accesso alla superclasse

- E' possibile accedere ad attributi e metodi della superclasse attraverso *super*
- Invocare costruttori della classe padre
 - `super()`
 - `super(lista parametri)`
- Invocare metodi della classe padre
 - `super.methodSuper(...);`

Super...esempio

```
public class ClassA {  
    public void printMe() {  
        System.out.println("Io sono A");  
    }  
    public void sayHello() {  
        System.out.println("Hello!");  
    }  
}
```

```
public class ClassB extends ClassA {  
    public void printMe() {  
        super.printMe();  
        System.out.println("Io sono B");  
    }  
}
```

LE CLASSI ASTRATTE

Le classi astratte

- Le classi astratte hanno dei metodi astratti, ovvero **non** fornisco un'implementazione per alcuni metodi
 - N.B.: le classi astratte possono avere anche dei metodi implementati
- Non è possibile istanziare oggetti di classi astratte
- Le classi astratte possono essere ereditate e in quel caso la sottoclasse deve fornire un'implementazione dei metodi astratti

Classi astratte

```
public abstract class GraphicObject {  
    // declare fields  
    // declare nonabstract methods  
    abstract void draw();  
}
```

Classi astratte vs. interfacce

- Le classi astratte sono simili alle interfacce
 - Non possono essere istanziate
 - Contengono metodi senza implementazione
- Tuttavia nelle classi astratte
 - È possibile dichiarare attributi che non siano static e final
 - È possibile dichiarare metodi con un'implementazione
- Nelle interfacce tutti gli attributi devono essere static e final e tutti i metodi public
- **Ricorda che:** è possibile estendere una sola classe (anche se astratta), ma è possibile implementare numerose interfacce

Classi astratte vs. interfacce

- Utilizza le classi estratte se:
 - vuoi condividere del codice (metodi) tra un insieme di classi che sono tra di loro strettamente correlate
 - ti aspetti che le classi che andranno ad estendere la classe astratta hanno molti metodi o attributi in comune
 - vuoi utilizzare attributi non static o final. Questo ti permette di avere dei metodi che modificano gli attributi degli oggetti ai quali appartengono

Classi astratte vs. interfacce

- Utilizza le interfacce se:
 - le classi che devono implementare l'interfaccia non sono strettamente correlate
 - vuoi specificare il comportamento di una particolare struttura dati senza entrare nei dettagli implementativi
 - hai bisogno di ricorrere all'ereditarietà multipla

Classe astratta: esempio

```
abstract class GraphicObject {  
    int x, y;  
    ...  
    void moveTo(int newX, int newY) {  
        ...  
    }  
    abstract void draw();  
    abstract void resize();  
}
```

Classe astratta: esempio

```
class Circle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}  
class Rectangle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

Classi astratte + interfacce

- Una classe astratta può implementare anche una o più interfacce
 - Può fornire un'implementazione per alcuni metodi delle interfacce che implementa
 - Può NON fornire un'implementazione per alcuni metodi delle interfacce che implementa
 - Le sotto classi dovranno fornire le implementazioni

Classi astratte + interfacce: esempio

```
public interface Y {  
    // methods declaration  
}
```

```
abstract class X implements Y {  
    // implements all but one method of Y  
}
```

```
class XX extends X {  
    // implements the remaining method in Y  
}
```


Esercizio 1

- Sia data la seguente interfaccia:

```
public interface Figura {  
    // restituisce l'area della figura  
    double area();  
}
```

- Scrivere la classe astratta FiguraComp che implementa sia l'interfaccia Figura che Comparable¹.

```
int compareTo(Object o)
```

Compares this object with the specified object for order.

Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

¹ Comparable è un'interfaccia presente in Java nel package java.lang

Esercizio 1

- La classe `FiguraComp` deve contenere
 - due variabili d'istanza `dim1` e `dim2`, di tipo `double` e dichiarate `protected`;
 - un costruttore con due parametri `double`, che inizializza le variabili d'istanza;
 - il metodo `public int compareTo(Object o)` che realizza il metodo astratto dell'interfaccia `Comparable`, confrontando due oggetti di tipo `FiguraComp` in base all'area: un oggetto precede un altro se ha area minore;
 - il metodo `equals`, definito in modo consistente con `compareTo` (se `compareTo==0` allora devono essere uguali i due oggetti)
- Si noti che la classe `FiguraComp` deve essere astratta, perché non fornisce un'implementazione del metodo dell'interfaccia `Figura`.

Esercizio 1

- Scrivere quindi le classi concrete Rettangolo e Triangolo che estendono FiguraComp e la classe Quadrato che estende Rettangolo. Ricordarsi di definire opportunamente anche il metodo toString()
- Per testare le classi, realizzare una classe TestFigure con un *main*. In particolare, verificare che con questa organizzazione delle classi è possibile confrontare oggetti di classi diverse (Triangolo e Rettangolo) con compareTo e con equals poiché questi metodi sono definiti in una superclasse (astratta) comune.

THE END

