

Metodi e attributi di classe

- Sintassi: **static** <definizione dell'attributo o metodo>
- Un attributo di classe è condiviso da tutti gli oggetti della classe
- Si può accedere a un attributo di classe (se public) senza bisogno di creare un oggetto tramite la notazione:
 - <nome classe>.<nome attributo>
- Un metodo di classe può essere invocato senza bisogno di creare un oggetto tramite la notazione:
 - <nome classe>.<nome metodo>(<par. attuali>)
- Un metodo di classe può essere comunque invocato su un oggetto della classe

Metodi e attributi di classe: vincoli

- Un metodo static può accedere ai soli attributi e metodi static
- Un metodo convenzionale può accedere liberamente a metodi e attributi static

Metodi e attributi di classe

```
public class Shape {  
    static Screen screen=new Screen(); // si noti l'inizializzazione  
    static void setScreen(Screen s) {screen=s;}  
    void show(Screen s) {setScreen(s);}  
  
    public static void main(String[] args) {  
        Shape.setScreen(new Screen()); // corretto  
        Shape.show(); // errato, show e' un metodo normale  
        Shape s1=new Shape(), s2=new Shape();  
        Screen s=new Screen();  
        s1.setScreen(s); // corretto, chiamo metodi static su oggetti  
  
        // a questo punto s2.screen==s1.screen==s  
    }  
}
```

Ancora sui costruttori

- È possibile invocare un costruttore dall'interno di un altro tramite la notazione
 - `this(<elenco di parametri attuali>);`
- Tuttavia il `this` deve essere la prima istruzione. Esempio:

```
import java.util.Calendar;
```

```
public Data(int g, int m, int a){  
    if (dataValida(g,m,a)) { // dataValida e' un metodo statico  
        giorno = g;  
        mese = m;  
        anno = a;  
    }  
    else ...  
}
```

```
public Data(int g, int m) { //giorno e mese + anno corrente  
    this(g,m,Calendar.getInstance().get(Calendar.YEAR));  
}
```

Attributi costanti

- È possibile definire attributi costanti tramite la notazione:
 - final <definizione di attributo>=<valore>

```
public class Automobile {  
    int colore;  
    final int BLU=0, GIALLO=1; // e altri  
    void dipingi(int colore) {this.colore=colore;}  
  
    public static void main(String[] args) {  
        Automobile a=new Automobile();  
        a.BLU=128; // errato  
        System.out.println("BLU="+a.BLU); // corretto  
    }  
}
```


Overloading di metodi

- All'interno di una stessa classe possono esservi più metodi con lo **stesso nome** purché si distinguano per numero e/o tipo dei parametri
 - Attenzione: Il tipo del valore restituito **non basta** a distinguere due metodi
- In Java l'intestazione di un metodo comprende il numero, il tipo e la posizione dei parametri; non include il tipo del valore restituito
 - Metodi overloaded devono avere intestazioni diverse
- Utile per definire funzioni con codice differente ma con effetti simili su tipi diversi

Esempio

```
public class Prova {  
    public int max(int a, int b, int c) {...}  
    public double max(double a, double b) {...}  
    public int max(int a, int b) {...}  
}
```

```
public static void main(String[] args){  
    Prova p = new Prova();  
  
    p.max(2,3,5);  
    p.max(2.3, 3.14);  
    p.max(2,3);  
}
```



Ogni volta verrà chiamato
il metodo “giusto”!

Reference e operatore “==”

- L'operatore di confronto == confronta i valori dei riferimenti e non gli oggetti!

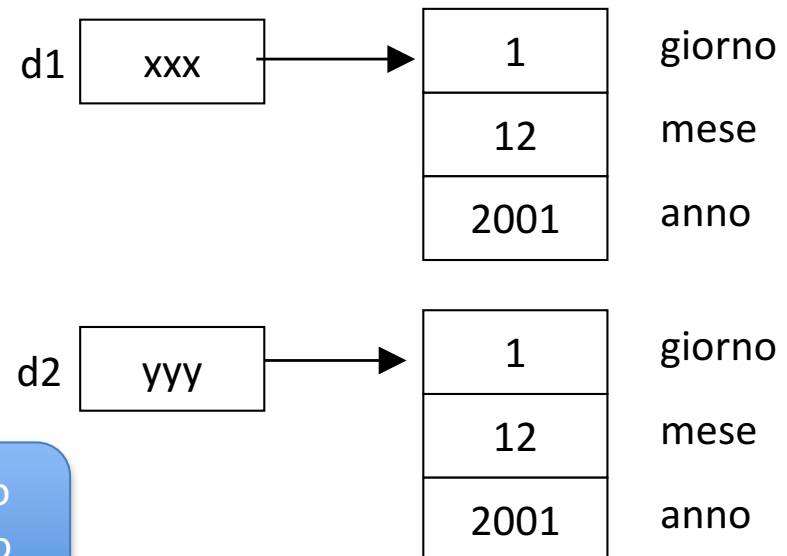
```
Data d1 = new Data(1,12,2001);
```

```
Data d2= new Data(1,12,2001);
```

```
if (d1==d2) {
```

```
...
```

```
d1=d2;
```



Falso! I riferimenti sono diversi anche se lo stato degli oggetti è lo stesso!

Il confronto d1==d2 diventa vero, abbiamo creato un alias quindi i riferimenti sono uguali

Confronto di uguaglianza

- Metodo equals() consente di verificare se due oggetti sono uguali, nel senso che hanno lo stesso valore dello stato
 - per String: contengono la stessa sequenza di caratteri
- Dice se due oggetti sono equivalenti
 - Che cosa ciò esattamente significhi **dipende dal tipo** dell'oggetto
 - ...per esempio, due insiemi sono equivalenti se contengono gli stessi elementi, indipendentemente dall'ordine di inserimento

Uso

```
String stringa1 = "Luciano";  
String stringa2 = "Giovanni";
```

```
Stringa1.equals(stringa2); //false  
String b = new String("Ciao");  
String c = new String("Ciao");  
if (b.equals(c)); //true
```

Enumerazioni

- Si possono dichiarare tipi enumerati, per modellare insiemi con cardinalità ridotta
 - `enum` Size {*SMALL*, *MEDIUM*, *LARGE*, *EXTRA_LARGE*};
 - Size s = Size.*MEDIUM*;
- Size è **una vera classe**: ha esattamente quattro istanze
 - Non se ne possono costruire altre
 - Non c'è bisogno di usare equals per confrontare i valori, basta ==
 - s può essere solo null o uno dei valori enumerati
- A una classe enumerata si possono aggiungere costruttore, metodi e attributi
 - Permettono di associare qualsiasi informazione alle costanti enumerate
 - I costruttori sono invocati solo quando vengono “costruite” le costanti
 - Non possiamo costruire oggetti generici da classi enumerazioni!

```
public enum Size {  
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");  
    private String abbreviation;  
    private Size(String abbreviation) {  
        this.abbreviation=abbreviation;  
    }  
    public String getAbbreviation(){  
        return abbreviation;  
    }  
}
```

Enumerazioni

- Tutte le classi enumerate offrono i seguenti metodi
 - static Enum valueOf(Class classname, String name)
che restituisce il nome della costante enumerata
 - String toString()
che restituisce il nome della costante

```
import java.util.Scanner;

public class ProvaString {
    public static void main (String[] args) {
        Scanner in = new Scanner(System.in);
        String str = in.next();
        Size siz = Enum.valueOf(Size.class, str);
        System.out.println(siz.toString());
    }
}
```

- Ogni classe enumerata ha un metodo che restituisce un array contenente tutti i valori della classe
 - Size[] valori = Size.values();

Altri metodi

- `ordinal()` restituisce la posizione (partendo da 0)
- `compareTo(...)` confronta l'oggetto corrente C con la variabile enumerativa passata come parametro e restituisce
 - Un valore negativo se C è minore del parametro
 - Zero se C è uguale al parametro
 - Un valore positivo se C è maggiore del parametro
- ...

Loop generalizzato

```
enum Color {Red, White, Blue}
```

```
for (Color c: Color.values()) { . . . }
```

Esempio

- Un esempio più ricco: pianeti del sistema solare, associati alla propria massa e raggio; si può calcolare il peso di un oggetto su ogni pianeta

```
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),
    PLUTO (1.27e+22, 1.137e6);

    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;}

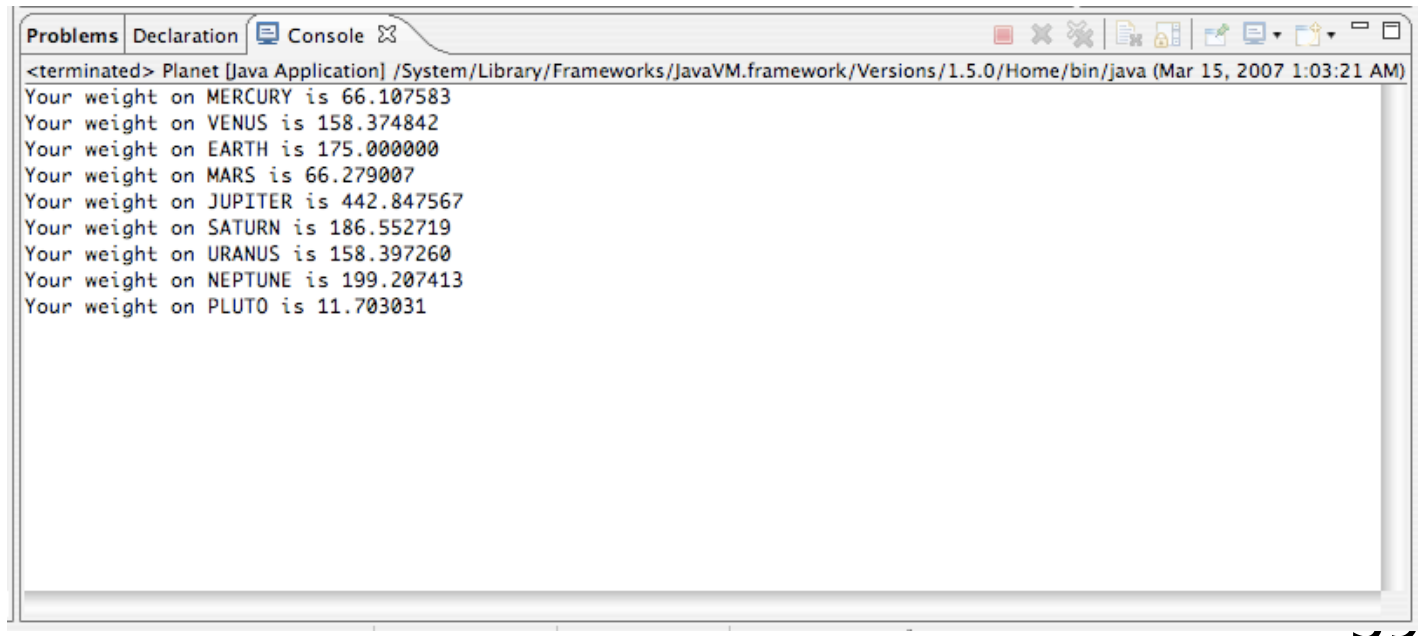
    // universal gravitational constant (m^3 kg^-1 s^-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {return G * mass / (radius * radius);}
    public double surfaceWeight(double otherMass) {return otherMass * surfaceGravity();}
}
```

Esempio

- A partire dal peso di un corpo sulla terra, calcola e stampa il peso su tutti gli altri pianeti

```
public static void main(String[] args) {  
    double earthWeight = Double.parseDouble(args[0]);  
    double mass = earthWeight/EARTH.surfaceGravity();  
    for (Planet p : Planet.values())  
        System.out.printf("Your weight on %s is %f%n", p, p.surfaceWeight(mass));  
}
```



The screenshot shows a Java IDE window with the 'Console' tab selected. The output of the program is displayed, showing the weight on various planets. The title bar of the window indicates the application is 'Planet [Java Application]' and the path is '/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin/java (Mar 15, 2007 1:03:21 AM)'.

```
<terminated> Planet [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin/java (Mar 15, 2007 1:03:21 AM)  
Your weight on MERCURY is 66.107583  
Your weight on VENUS is 158.374842  
Your weight on EARTH is 175.000000  
Your weight on MARS is 66.279007  
Your weight on JUPITER is 442.847567  
Your weight on SATURN is 186.552719  
Your weight on URANUS is 158.397260  
Your weight on NEPTUNE is 199.207413  
Your weight on PLUTO is 11.703031
```


Tipi riferimento per i tipi primitivi

- I tipi primitivi sono comodi, ma a volte si preferirebbe usarli come riferimento, per omogeneità
- Java fornisce classi predefinite
 - Integer, Character, Float, Long, Short, Double (sono in java.lang)
 - Un oggetto Integer contiene un int, ma viene inizializzato solo con i costruttori
 - Il tipo Integer è **immutabile**

Esempi

- `Integer i; // qui i vale null!`
- `i = new Integer(5); //i e' un rif. a oggetto che contiene 5`
- `Integer x = i; // sharing: x e i stesso oggetto`
- `i = y; // boxing automatico`
- `y = i; // unboxing automatico`
- `i = 3; // come sopra`

String

- Le stringhe sono **immutabili**
 - Non si possono aggiungere o togliere caratteri a una stringa, ma occorre costruirne una nuova
- Costruttori:
 - `String()`
 - `String(String s)`
- Operatore di concatenamento `+`
- Alcuni metodi pubblici:
 - `int length()` restituisce la lunghezza di una stringa
 - `char charAt(int index)` restituisce il char alla posizione index
 - il primo ha posizione 0
 - `String substring(int beginIndex)` (parte da 0)

Esempio

```
public class ProvaString {  
    public static void main (String[] args) {  
        String a = new String(); //a e' una ref a stringa vuota  
        String b = new String("Ciao"); //b e' una ref a "Ciao":  
            //abbreviazione: String b = "Ciao";  
        String c = new String(b); //Ora c e' copia di b  
        String d = b; //d e b sono alias  
        System.out.println(b + " " + c + " " + d);  
    }  
}
```

- L'assegnamento d=b è un assegnamento dei riferimenti
 - Non si copia l'oggetto!

“Catene puntate”

- Un oggetto può avere attributi che sono ancora oggetti o metodi che restituiscono oggetti: accesso a metodi e attributi avviene a “catena”

Esempi

- `System.out.println();`
 - **out** è attributo pubblico (statico) di classe **System**
 - La classe di **out** fornisce il metodo **println()**
- `String b = new String("Ciao");`
`String a = b.substring(1).substring(2);`
`System.out.println(a);` *//che oggetto e' a?*
 - Operatore “.” è associativo a sinistra:
`(b.substring(1)).substring(2);`
`(System.out).println();`

...ma è tutto qui?

Ereditarietà

- È possibile stabilire una relazione “sottoclasse_di” (\subseteq) fra le classi di un programma Java
 - Relazione d'ordine parziale (riflessiva e transitiva)
 - Public class B **extends** A {...}
- A **classe base**, o antenato, o padre, o superclasse, ...
- B **classe derivata**, o discendente, o figlio, o erede, o sottoclasse, ...

Relazione di ereditarietà

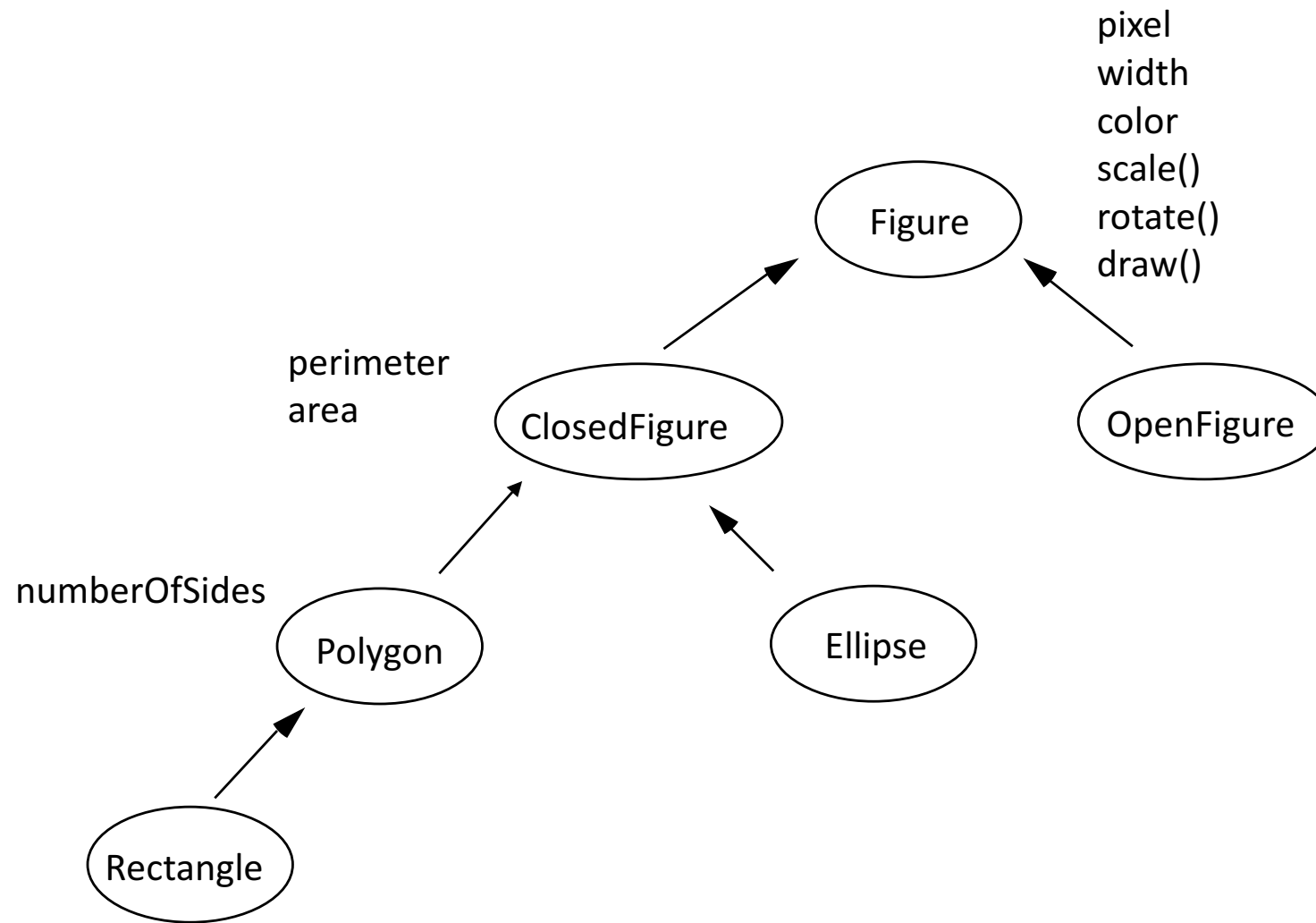
- La sottoclasse **eredita** tutta l'implementazione (attributi e metodi) della superclasse
 - Gli attributi e metodi della superclasse sono implicitamente definiti anche nella sottoclasse (ma con alcune differenze che vedremo fra poco)
- Una sottoclasse può **aggiungere** nuovi attributi e metodi ma anche **ridefinire** i metodi delle sue superclassi
 - Lasciando invariato numero e tipo dei parametri
 - Anche detto **overriding**

Un semplice esempio

```
public class Automobile {
    private String modello;
    private boolean accesa;
    public Automobile(String modello) {
        this.modello=modello;
        this.accesa = false;
    }
    public void accendi() {accesa=true;}
    public boolean puoPartire() {return accesa;}
}

public class AutomobileElettrica extends Automobile {
    private boolean batterieCariche;
    public void ricarica() {batterieCariche=true;}
    ...
}
}
```

Gerarchia a più livelli



Overriding

- Una sottoclasse può ridefinire l'implementazione di un metodo
 - L'intestazione del metodo **non deve cambiare**
 - L'intestazione **non include il tipo restituito**, che quindi **può cambiare** a patto che **generalizzi** il tipo del metodo ridefinito
 - Anche detta regola della **covarianza**...

```
public class AutomobileElettrica extends Automobile{
    private boolean batterieCariche;
    public void ricarica() {batterieCariche = true;}
    ...
}

    public void accendi() {// OVERRIDING
        // accensione auto elettrica e' diversa
        // da quella di auto a benzina ... la
        // reimplementiamo
    }
}
```

Pseudo variabile super

- All'interno di un metodo della sottoclasse ci si può riferire ai metodi della superclasse:
 - super.<nome metodo>(<lista par. attuali>)

```
public class AutomobileElettrica extends Automobile{  
  
    private boolean batterieCariche;  
    public void ricarica() {batterieCariche=true;}  
  
    public void accendi() {// OVERRIDING  
        if(batterieCariche) super.accendi();  
        else System.out.println("Batterie scariche");  
    }  
}
```

Costruttori

- I costruttori **non sono ereditati** perché occorre inizializzare anche i nuovi attributi
 - Per inizializzare gli attributi private ereditati, all'interno di un costruttore è possibile richiamare il costruttore della superclasse tramite:
 - `super(<lista di par. attuali>)`
posta come prima istruzione del costruttore
- Se il programmatore non chiama esplicitamente un costruttore della superclasse, il compilatore inserisce automaticamente il codice che invoca il costruttore di default della superclasse
 - ...che potrebbe non esistere!

```
public AutomobileElettrica(String modello) {  
    super(modello); //qui inizializza modello e accesa  
    batterieCariche=false;  
}
```

Object

- In mancanza di un'indicazione differente, una classe Java estende la classe Object
- La classe Object fornisce alcuni metodi di default tra i quali:
 - `public boolean equals(Object);`
 - `public String toString();`
 - `public Object clone();`