Metodologie di Programmazione (M-Z)

II semestre - a.a. 2022 – 2023

Parte 6 – Polimorfismo, Liste**

a cura di Stefano Faralli*

SAPIENZA
UNIVERSITÀ DI ROMA

**I crediti sulle slide di questo corso sono riportati nell'ultima slide

Polimorfismo

Polimorfismo

- Insieme all'ereditarietà, un altro concetto cardine della programmazione orientata agli oggetti
- Polimorfismo = **molte** + **forme**

1) Una variabile di un certo tipo A può contenere un riferimento a un oggetto del tipo A o di qualsiasi sua sottoclasse

```
Animale a = new Gatto();
a = new Chihuahua();
```

2) La selezione del metodo da chiamare avviene in base all'effettivo tipo dell'oggetto riferito dalla variabile

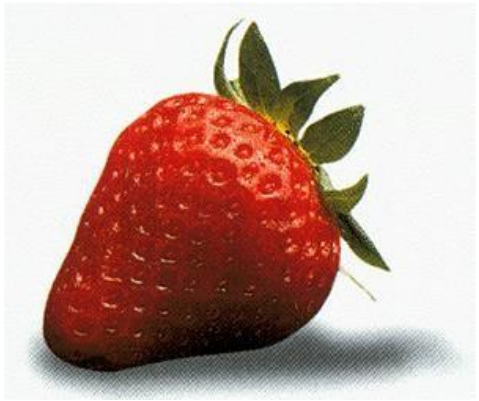
```
Animale a = new Gatto();
a.emettiVerso();
a = new Chihuahua();
a.emettiVerso();
```

Miaooo!

Bau bau!

Come nel parlare comune

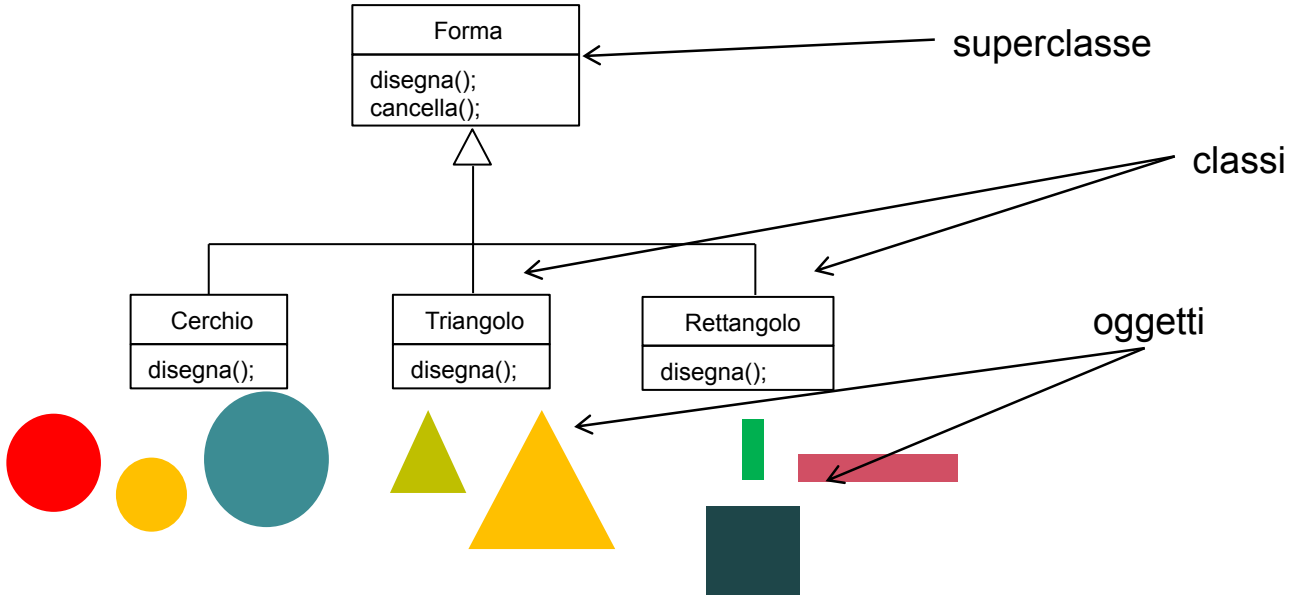
- Mangio un frutto:



- Guardo un cartone:

Motivazione

- Vogliamo utilizzare **facilmente** e **senza differenziazioni formali** oggetti diversi ma con caratteristiche comuni




```
ArrayList<Forma> forme = new ArrayList<>();
/* …codice per riempire la lista con varie forme… */
for (Forma f : forme) f.disegna();
```

Binding statico vs. dinamico

- Il binding statico consiste nell'associare un metodo con il tipo della classe di una variabile riferimento
  - Questo non permetterebbe di chiamare il metodo appropriato
- Il polimorfismo implementa invece il binding dinamico, poiché l'associazione tra una variabile riferimento e il metodo viene stabilita a tempo di esecuzione

```
Animale a = new Gatto();
a = new Chihuahua();
```

Stesso metodo ma implementazione differente

- **Senza dover conoscere il tipo esatto** (la classe) dell'oggetto su cui si invoca il metodo

Chiamare metodi della superclasse (1)

```
import java.util.Random;

public class StringaHackerata
{
    private String s;

    public StringaHackerata(String s)
    {
        this.s = s;
    }

    @Override
    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        Random random = new Random();

        for (int k = 0; k < s.length(); k++)
        {
            char c = s.charAt(k);
            if (random.nextBoolean()) c = Character.toUpperCase(c);
            else c = Character.toLowerCase(c);

            sb.append(c);
        }

        return sb.toString();
    }
}
```

Chiamare metodi della superclasse (2)

- E' sufficiente utilizzare la parola chiave super:

```
import java.util.Random;


public class StringaHackerataConStriscia extends StringaHackerata
{
    final public static int MAX_LUNGHEZZA = 10;

    public StringaHackerataConStriscia(String s)
    {
        super(s);
    }

    public String getStriscia()
    {
        Random random = new Random();
        int len = random.nextInt(MAX_LUNGHEZZA);
        StringBuffer sb = new StringBuffer();

        // -=-=-=-
        for (int k = 0; k < len; k++) sb.append(k % 2 == 0 ? '-' : '=');

        return sb.toString();
    }

    @Override
    public String toString()
    {
        String striscia = getStriscia();
        return striscia+" "+super.toString()+" "+striscia;
    }
}
```

Chiamiamo il metodo della superclasse

StringaHackerata e StringaHackerataConStriscia

- Eseguendo il metodo main:

```
public static void main(String[] args)
{
    StringaHackerata s1 = new StringaHackerata("drago della programmazione");
    StringaHackerataConStriscia s2 = new StringaHackerataConStriscia("drago della programmazione");

    System.out.println(s1);
    System.out.println(s2);
}
```

- Si ottiene:

```
DraGo DElla PROgraMMaZioNE
-=-=-=-=- dRAgO DelLA PROGRAMmaZiONe -=-=-=-=-
```

- E se scrivo…?

```
public static void main(String[] args)
{
    StringaHackerata s1 = new StringaHackerata("drago della programmazione");
    StringaHackerata s2 = new StringaHackerataConStriscia("drago della programmazione");

    System.out.println(s1);
    System.out.println(s2);
}
```

Stesso risultato grazie al **polimorfismo**!

Esempio: Modellare gli impiegati (1)

- La classe Impiegato modella il nome e il codice dell'impiegato
- Permette di aggiornare il nome dell'impiegato
- Restituisce su richiesta il codice (id) e il nome
- Definisce il metodo toString() restituendo la stringa "nome (codice)"

```java
public class Impiegato
{
    /**
     * Nome dell'impiegato
     */
    private String nome;

    /**
     * Identificativo dell'impiegato
     */
    private String id;

    public Impiegato(String nome, String id)
    {
        this.nome = nome;
        this.id = id;
    }

    /**
     * Aggiorna il nome dell'impiegato
     */
    public void setNome(String nome)
    {
        this.nome = nome;
    }

    public String getId() { return id; }
    public String getNome() { return nome; }

    @Override
    public String toString()
    {
        return nome+" ("+id+")";
    }
}
```

Esempio: Modellare gli impiegati (2)

- Modelliamo ora la classe ImpiegatoStipendiato
- Ha un suo stipendio mensile
- Sovrascrive toString() riutilizzando il toString della superclasse e aggiungendo lo stipendio

```
public class ImpiegatoStipendiato extends Impiegato
{
    private double stipendio;

    public ImpiegatoStipendiato(String nome, String id, double stipendio)
    {
        super(nome, id);
        this.stipendio = stipendio;
    }

    public double getStipendio() { return stipendio; }

    public void setStipendio(double nuovoStipendio) { stipendio = nuovoStipendio; }

    @Override
    public String toString()
    {
        return super.toString()+": "+stipendio+" euro";
    }
}
```

Esempio: Modellare gli impiegati (3)

- Modelliamo infine la classe ImpiegatoACottimo
- E' connotato dalla paga per prodotto e dal numero di prodotti lavorati

```
public class ImpiegatoACottimo extends Impiegato
{
    private double pagaPerProdotto;
    private int prodottiLavorati;

    public ImpiegatoACottimo(String nome, String id, double pagaPerProdotto, int prodottiLavorati)
    {
        super(nome, id);
        this.pagaPerProdotto = pagaPerProdotto;
        this.prodottiLavorati = prodottiLavorati;
    }

    public double getPagaPerProdotto() { return pagaPerProdotto; }
    public double getProdottiLavorati() { return prodottiLavorati; }

    public void setPagaPerProdotto(double pagaPerProdotto) { this.pagaPerProdotto = pagaPerProdotto; }
    public void setProdottiLavorati(int prodottiLavorati) { this.prodottiLavorati = prodottiLavorati; }

    @Override
    public String toString()
    {
        return super.toString()+": "+prodottiLavorati+" prodotti * "+pagaPerProdotto+" euro a prodotto";
    }
}
```

Esempio: Modellare gli impiegati (4)

- Testiamo le classi:

```
public class TestaImpiegati
{
    public static void main(String[] args)
    {
        Impiegato i1 = new ImpiegatoStipendiato("Mario", "imp1", 1500);
        Impiegato i2 = new ImpiegatoACottimo("Luigi", "imp2", 10, 5);

        System.out.println(i1);
        System.out.println(i2);
    }

}
```

- Ottenendo questo output:

```
Mario (imp1): 1500.0 euro
Luigi (imp2): 50 prodotti * 10.0 euro a prodotto
```

L'operatore instanceof

- L'operatore, applicato a un oggetto e a un nome di classe, restituisce true se l'oggetto è un tipo o un sottotipo di quella classe
- Ad esempio:

```
public class TestaImpiegati
{
    public static void main(String[] args)
    {
        Impiegato i1 = new ImpiegatoStipendiato("Mario", "imp1", 1500);
        Impiegato i2 = new ImpiegatoACottimo("Luigi", "imp2", 10, 50);

        System.out.println(i1);
        System.out.println(i2);

        System.out.println(i1 instanceof Impiegato);
        System.out.println(i1 instanceof ImpiegatoStipendiato);
        System.out.println(i1 instanceof ImpiegatoACottimo);
    }

}
```

- Stampa:
  - `true`
  - `true`
  - `false`

Conversione di tipo fra sottoclasse e superclasse

- Posso sempre convertire senza cast esplicito un sottotipo a un supertipo (upcasting)

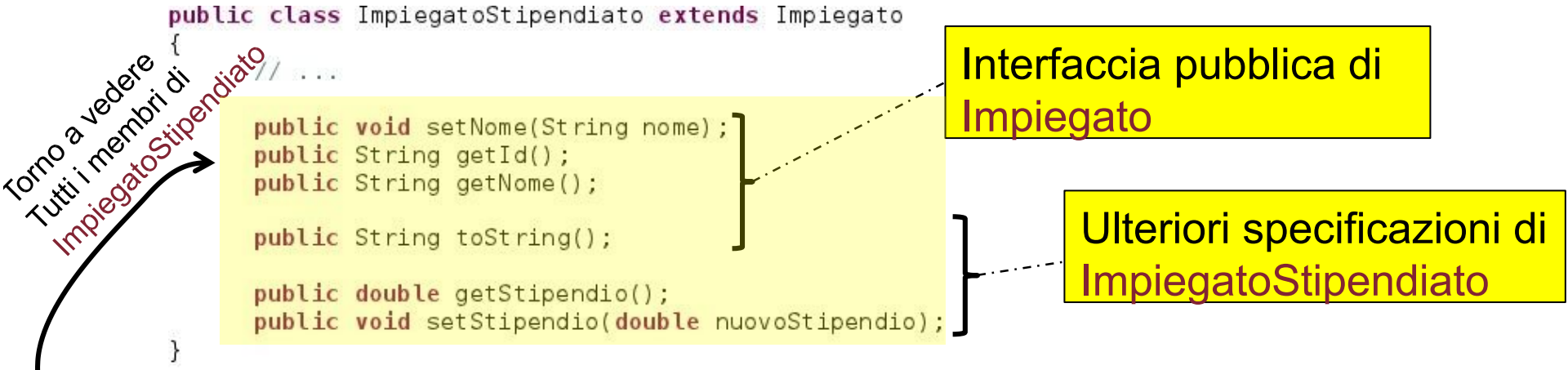
```
ImpiegatoStipendiato is1 = new ImpiegatoStipendiato("Mario", "imp1", 1500);
Impiegato i = is1;
```

- A volte può essere necessario convertire un supertipo a un sottotipo (downcasting)
  - Richiede un cast esplicito

```
ImpiegatoStipendiato is2 = (ImpiegatoStipendiato)i;
```

Che succede all'interfaccia con la conversione di tipo?

Con l'upcasting, si "restringe" **temporaneamente** l'interfaccia dell'oggetto alla superclasse:



- ImpiegatoStipendiato is = new ImpiegatoStipendiato(…);
- Impiegato i = is;
- is = (ImpiegatoStipendiato)i;

Significato di un campo riferimento di tipo astratto

- Cosa significa allora un campo (o una variabile locale) riferimento a tipo astratto?
- Supponiamo che Forma sia astratta
  - Ovviamente non potremo creare un oggetto di tipo Forma (Forma f = new Forma())
  - Ma potremo creare istanze di sottoclassi concrete di Forma e assegnarne il riferimento alla variabile di tipo Forma
  - Questo è possibile proprio grazie al polimorfismo (es. Forma f = new Cerchio())

La superclasse universale Object

- Tutte le classi in Java ereditano direttamente o indirettamente dalla classe Object
  - Tutti i suoi 11 metodi sono ereditati
- Quando si definisce una classe senza estenderne un'altra:

```
public class LaMiaClasse
{

}
```

questo è equivalente a estendere Object:

```
public class LaMiaClasse extends Object
{

}
```

I metodi principali della classe Object

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

I metodi principali della classe Object

Metodo	Descrizione
Object **clone**()	Restituisce una copia dell'oggetto
boolean **equals**(Object o)	Confronta l'oggetto con quello in input
Class<? extends Object> **getClass**()	Restituisce un oggetto di tipo Class che contiene informazioni sul tipo dell'oggetto
int **hashCode()**	Restituisce un intero associato all'oggetto (per es. ai fini della memorizzazione in strutture dati, hashtable, ecc.)
String **toString()**	Restituisce una rappresentazione di tipo String dell'oggetto (per default: tipo@codice_hash)

Sovrascrivere il metodo toString

- toString è uno dei metodi che ogni classe eredita direttamente o indirettamente dalla classe Object
- Non prende argomenti e restituisce una String
- Chiamato implicitamente quando un oggetto deve essere convertito a String (es. System.out.println(o))
- L'annotazione @Override serve a garantire che il metodo "sovrascriva" il metodo di una superclasse

```
public class Punto
{
    private int x, y, z;

    public Punto(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    @Override
    public String toString() { return "("+x+","+y+","+z+")"; }
}
```

Sovrascrive l'implementazione di toString ereditata da Object()

Sovrascrivere il metodo equals

- Il metodo equals viene invocato per confrontare il contenuto di due oggetti
- Per default, se sono "uguali", il metodo restituisce true

```
public boolean equals(Object o) { return this == o; }
```

- Tuttavia, la classe Object **non** conosce il contenuto delle sottoclassi
  - Per mantenere il "contratto" del metodo è necessario sovrascriverlo

```
public class Punto
{
	private int x, y, z;

	// ...

	@Override
	public boolean equals(Object o)
	{
		if (o == null) return false;
		if (getClass() != o.getClass()) return false;
		Punto p = (Punto)o;
		return x == p.x && y == p.y && z == p.z;
	}
}
```

In questa situazione, meglio di instanceof (che accetta anche sottoclassi)

Javadoc sulla simmetria di equals

L'uso di **getClass**() in **equals** garantisce il principio di simmetria richiesto nell'implementazione di **equals**:

equals

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

Parameters:

`obj` - the reference object with which to compare.

Returns:

`true` if this object is the same as the obj argument; `false` otherwise.

See Also:

`hashCode()`, `HashMap`

Joshua Bloch su getClass() vs. instanceof in equals

“The reason that I **favor the instanceof approach** is that, **when you use the getClass approach**, **you have the restriction that objects are only equal to other objects of the same class**, the same run time type. If you extend a class and add a couple of innocuous methods to it, then check to see whether some object of the subclass is equal to an object of the super class, even if the objects are equal in all important aspects, you will get the surprising answer that they aren't equal. In fact, this violates a strict interpretation of the **Liskov substitution principle**, and can lead to very surprising behavior. In Java, it's particularly important because most of the collections (HashMap, etc.) are based on the equals method. If you put a member of the super class in a hash table as the key and then look it up using a subclass instance, you won't find it, because they are not equal.”

Un esempio completo con instanceof e downcasting

- Implementiamo un metodo equals di confronto tra un oggetto di tipo SitoWeb e un altro oggetto qualsiasi:

```
public class SitoWeb
{
    private String url;

    public SitoWeb(String url) { this.url = url; }

    @Override
    public boolean equals(Object o)
    {
        // uso instanceof per garantire che equals
        // funzioni anche sulle sottoclassi
        // (in altre situazioni, e' piu' sensato
        // l'utilizzo di getClass())
        if (!(o instanceof SitoWeb)) return false;

        // downcasting
        return url.equals(((SitoWeb)o).url);
    }
}
```

Sovrascrivere il metodo clone (1)

- L'operatore di assegnazione = **non** effettua una copia dell'oggetto, ma solo del riferimento all'oggetto
- Per creare una copia di un oggetto è necessario richiamare clone()
- x.clone() != x **sarà sempre vero**
- clone **non richiama il costruttore della classe**
- Tuttavia l'implementazione nativa di default di Object.clone copia l'oggetto **campo per campo** (shallow copy)
  - **Ottimo** se i campi sono tutti **primitivi**
  - **Problematico** se i campi sono **riferimenti**

Sovrascrivere il metodo clone (2)

- Per implementare la copia in una propria classe è necessario sovrascrivere clone() che è

  PROTETTA (quindi visibile solo in gerarchia e nel package)

  – Se il nostro oggetto contiene riferimenti e vogliamo evitare che la copia contenga un riferimento allo stesso oggetto membro, non possiamo chiamare semplicemente (o non chiamiamo proprio) super.clone()

- E' necessario implementare l'interfaccia "segnaposto" Cloneable altrimenti Object.clone emetterà semplicemente l'eccezione CloneNotSupportedException

Clone: attenzione alla "shallow copy"!

```
public class IntVector implements Cloneable
{
    ArrayList<Integer> list = new ArrayList<>();

    public IntVector(int... values)
    {
        for (int v : values) list.add(v);
    }

    public void add(int v) { list.add(v); }

    public IntVector getCopy()
    {
        try
        {
            return (IntVector)clone();
        }
        catch(CloneNotSupportedException e) { return null; }
    }
}
```

Clone: attenzione alla "shallow copy"!

- Testiamo la classe con Junit:

```
import static org.junit.Assert.*;

public class IntVectorTest
{
    @Test
    public void testGetCopy1()
    {
        IntVector v = new IntVector(4, 8, 15, 16, 23, 42);
        IntVector v2 = v.getCopy();

        assertNotSame(v.list, v2.list);
    }

    @Test
    public void testGetCopy2()
    {
        IntVector v = new IntVector(4, 8, 15, 16, 23);
        IntVector v2 = v.getCopy();
        v.add(42);

        assertNotEquals(v.list, v2.list);
    }
}
```

I due riferimenti puntano alla stessa identica lista!

Sovrascrivere il metodo clone (3)

- Per evitare la copia dei riferimenti, è necessaria la clonazione "profonda" (deep cloning)
  - Si può usare Object.clone per la clonazione dei tipi primitivi
  - E richiamare .clone() su tutti i campi che sono riferimenti ad altri oggetti, impostando i nuovi riferimenti nell'oggetto clonato
- Con il **deep cloning**, i test Junit hanno successo!

```
public IntVector getCopy()
{
    try
    {
        IntVector v = (IntVector)clone();
        v.list = (ArrayList<Integer>)list.clone();
        return v;
    }
    catch(CloneNotSupportedException e) { return null; }
}
```

Enumerazioni e Object

- Una enumerazione ha tante istanze quante sono le costanti enumerative al suo interno
  - **Non** è possibile costruire altre istanze
- Le classi enumerative estendono la classe Enum, da cui ereditano i metodi toString e clone
  - toString() restituisce il nome della costante
  - clone() restituisce l'oggetto enumerativo stesso senza farne una copia (che non è possibile fare, visto che sono costanti…)
- Enum a sua volta estende Object, per cui il metodo equals restituisce true solo se le costanti enumerative sono identiche

Ancora sulle enumerazioni

- **Non** possono essere create nuove istanze
- **MA** possono essere costruite le istanze “costanti”
  - Si definisce un costruttore (**NON** pubblico, ma con visibilità di default)
  - Si costruisce ciascuna costante (un oggetto **separato** per ognuna)
  - Si possono definire altri metodi di accesso o modifica dei campi, ecc.

```
public enum TipoDiMoneta
{
    /**
     * Le costanti enumerative, costruite in modo appropriato
     */
    CENT(0.01), CINQUE_CENT(0.05), DIECI_CENT(0.10), VENTI_CENT(0.20), CINQUANTA_CENT(0.50), EURO(1.00), DUE_EURO(2.00);

    /**
     * Valore numerico della costante
     */
    private double valore;

    /**
     * Costruttore con visibilita' di default
     */
    TipoDiMoneta(double valore) { this.valore = valore; }

    /**
     * Metodo di accesso al valore
     */
    public double getValore() { return valore; }
}
```

Esempio: i pianeti “enumerati”

```
public enum Pianeta
{
    MERCURIO (3.303e+23, 2.4397e6),   VENERE   (4.869e+24, 6.0518e6),
    TERRA    (5.976e+24, 6.37814e6),  MARTE    (6.421e+23, 3.3972e6),
    GIOVE    (1.9e+27,   7.1492e7),   SATURNO  (5.688e+26, 6.0268e7),
    URANO    (8.686e+25, 2.5559e7),   NETTUNO  (1.024e+26, 2.4746e7);

    /**
     * Costante di gravitazione universale
     */
    public static final double G = 6.67300E-11;

    /**
     * Massa in kilogrammi
     */
    private final double massa;

    /**
     * Raggio in metri
     */
    private final double raggio;

    Pianeta(double massa, double raggio)
    {
        this.massa = massa;
        this.raggio = raggio;
    }

    private double getMassa() { return massa; }
    private double getRaggio() { return raggio; }
    public double getGravitaDiSuperficie() { return G * massa / (raggio * raggio); }
    public double getPesoDiSuperficie(double altraMassa) { return altraMassa * getGravitaDiSuperficie(); }
}
```

Metodi e classi final

- Ricordate? con la parola chiave abstract **obblighiamo** i programmatori a implementare certi metodi
- La parola chiave final ci permette di fare il contrario: **impedire** ad altri programmatori di:
  - Creare sottoclassi (se specificato di fronte a class)
  - Reimplementare (=sovrascrivere) certi metodi (di fronte all'intestazione del metodo)

Metodi final: un esempio (1)

- Supponete di creare un conto corrente “sicuro”

```
public class ContoCorrenteSicuro extends ContoCorrente
{
      public       boolean controllaPassword(String password)
      {
            // verifica la password
            // ...
      }
}
```

- Tuttavia, estendendo la classe, possiamo “gabbare” l’utente che la userà e accedere noi al conto:

```
public class ContoCorrenteSicuroFidatiDiMe extends ContoCorrenteSicuro
{
      public       boolean controllaPassword(String password)
      {
            return true;
      }
}
```

Metodi final: un esempio (2)

- Possiamo evitare questo problema specificando il metodo final

```
public class ContoCorrenteSicuro extends ContoCorrente
{
      public final boolean controllaPassword(String password)
      {
            // verifica la password
            // ...
      }
}
```

Nessuno può sovrascrivere questo metodo!

Credits

Le slide di questo corso sono il frutto di una personale rielaborazione delle slide del Prof. Navigli.

In aggiunta, le slide sono state revisionate dagli studenti borsisti della Facoltà di Ingegneria Informatica, Informatica e Statistica: Mario Marra e Paolo Straniero.