
PROGRAMMAZIONE 2

3. Primi passi in Java

Why Java?

- Java offre moltissime cose utili
 - Usato a livello industriale
 - Librerie vastissime
 - Complicato ma necessariamente complicato
- Obiettivo di Programmazione II
 - Presentare le caratteristiche essenziali della programmazione Object-Oriented
 - Illustrare come le tecniche OO aiutano nella soluzione di problemi
 - Sperimentare con Java

Tipi di Astrazione: Java

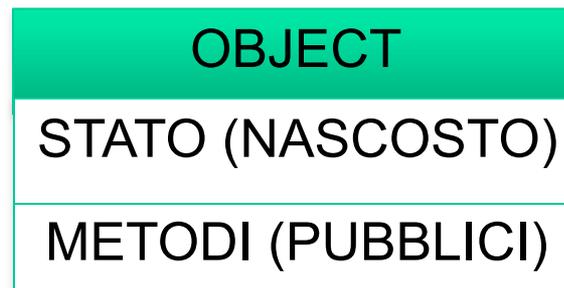
- **astrazione procedurale**
 - si aggiungono nuove operazioni
- **astrazione di dati**
 - si aggiungono nuovi tipi di dato
- **gerarchie di tipo**
 - permette di astrarre da specifici tipi di dato a famiglie di tipi correlati
- **iterazione astratta**
 - permette di iterare su elementi di un insieme, senza sapere come questi sono ottenuti

Astrazione tramite **specifica**

- Aggiunge nuovi tipi di dato e relative operazioni
- In questo caso la **specifica** descrive le relazioni fra le operazioni
- Si realizza con **opportune annotazioni inserite nel codice**
- L'implementazione del tipo di dato deve essere nascosta per garantire la correttezza

Oggetti e classi

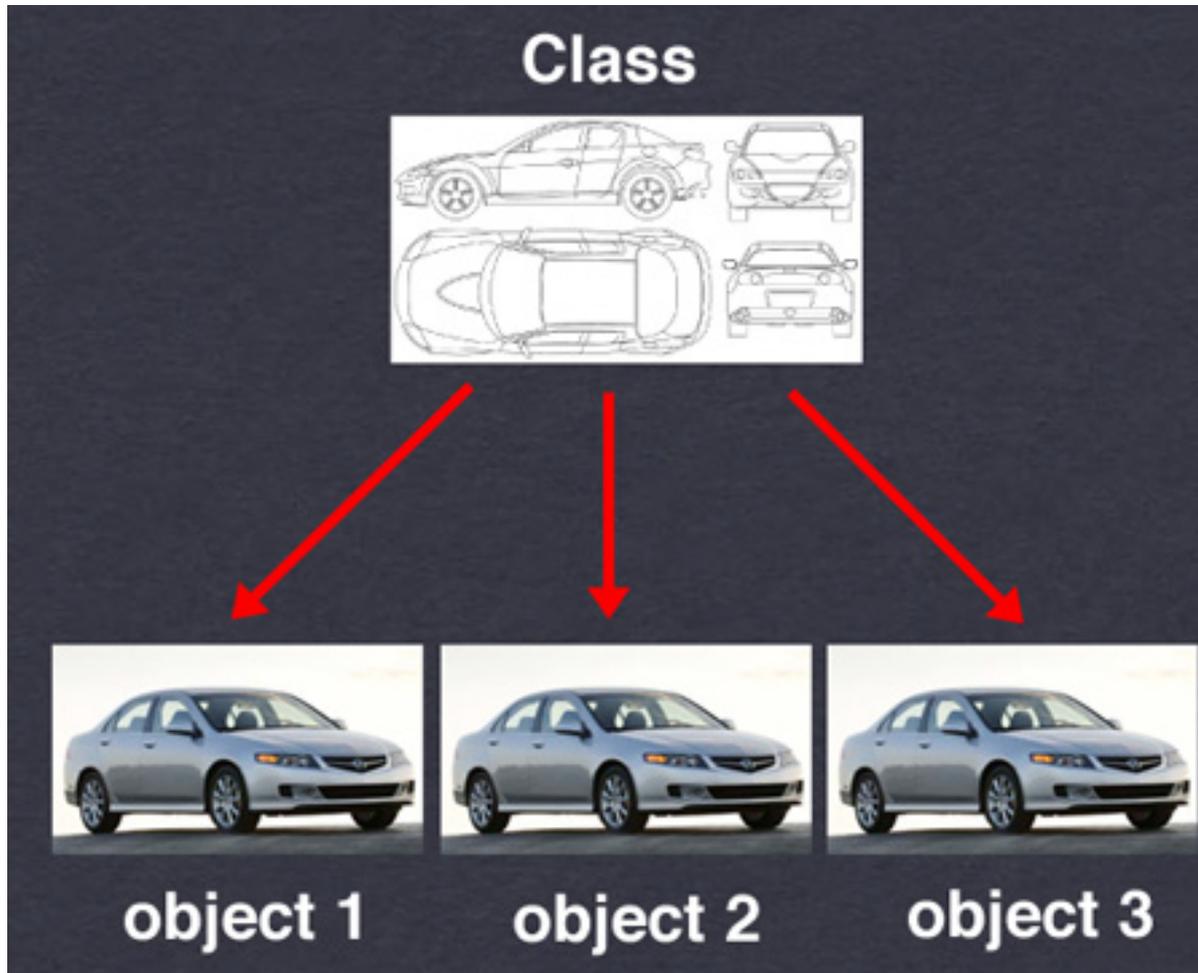
- **Oggetto**: insieme strutturato di *variabili di istanza* (stato) e *metodi* (operazioni)
- **Classe**: *modello (template)* per la creazione di oggetti



Oggetti e classi

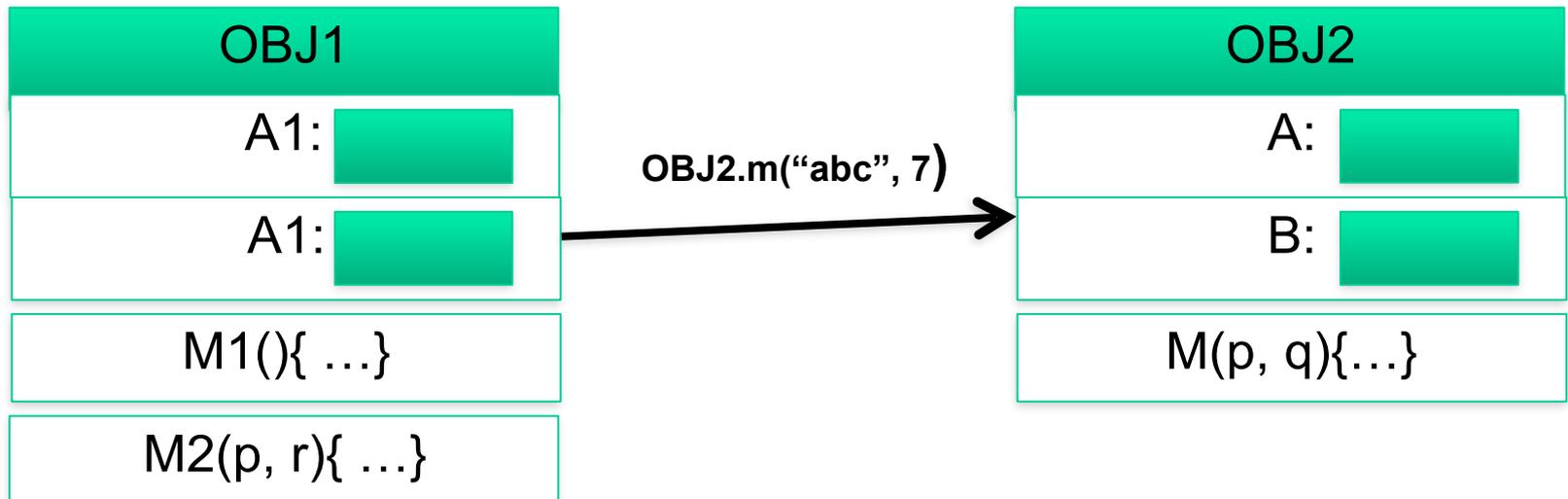
- La definizione di una **classe** specifica
 - **Tipo e valori iniziali** dello stato locale degli oggetti (**le variabili di istanza**)
 - **Insieme delle operazioni** che possono essere eseguite (**metodi**)
 - **Costruttori** (uno o più): codice che deve essere eseguito al momento della creazione di un oggetto
- Ogni oggetto è una **istanza** di una **classe** e può (opzionalmente) implementare una **interfaccia**

Oggetti e classi

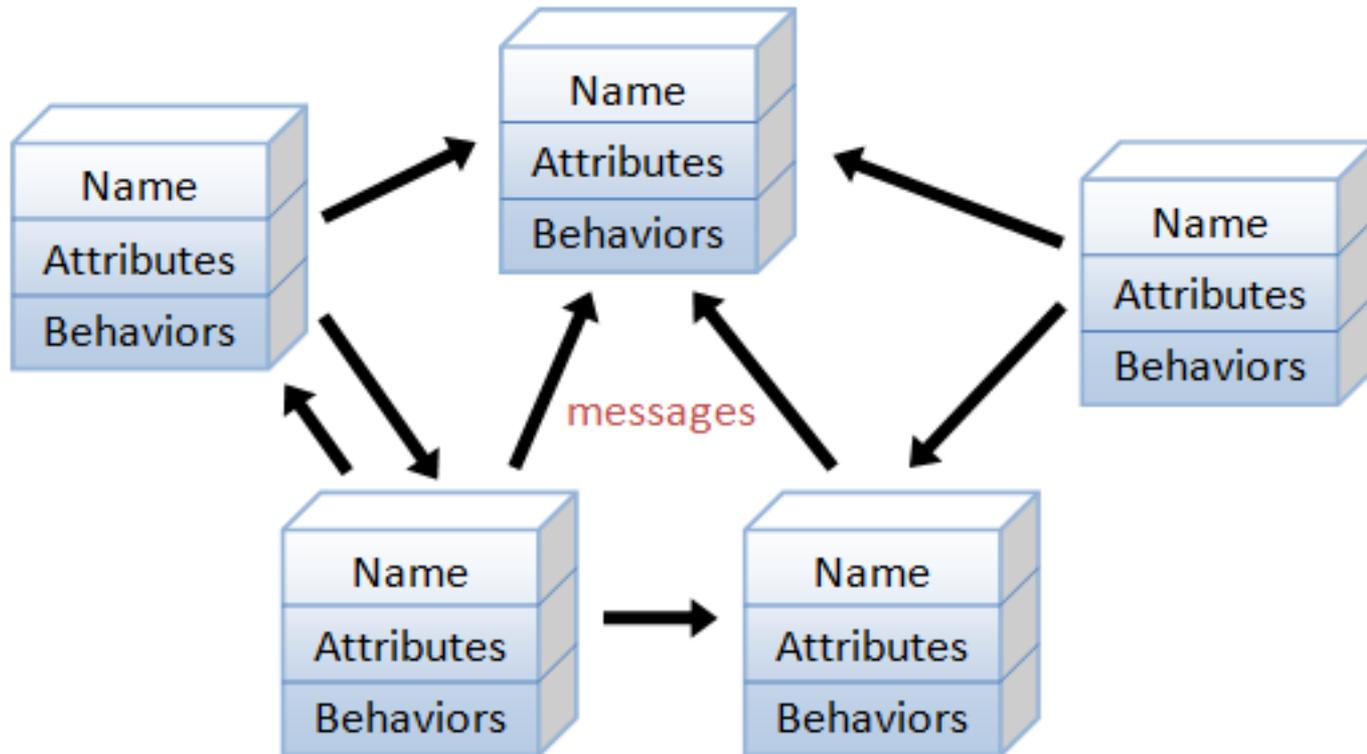


Il paradigma a oggetti

- Sistema software = insieme di **oggetti cooperanti**
- Oggetti sono caratterizzati da uno **stato** e da un insieme di **funzionalità**
- Oggetti cooperano scambiandosi dei **messaggi**



Oggetti e classi



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Un primo esempio

```
public class Counter {  
    // nome della classe  
  
    private int cnt; // lo stato locale  
  
    // metodo costruttore  
    public Counter ( ) { cnt = 0; }  
    // metodo  
    public int inc ( ) { cnt++; return cnt; }  
    // metodo  
    public int dec ( ) { cnt--; return cnt; }  
}
```

DICHIARAZIONE DI CLASSE

public = visibile fuori
dell'oggetto

private = visibile solo
all'interno dell'oggetto

Esecuzione di Java

Un programma Java è mandato in esecuzione invocando un metodo speciale di una opportuna classe chiamato **main**

```
public class First {  
    public static void main(String[ ] args) {  
        Counter c = new Counter( );  
        System.out.println(c.inc( ));  
        System.out.println(c.dec( ));  
    }  
}
```

Compilare ed eseguire

```
prompt$ javac Counter.java
```

Viene creato il bytecode Counter.class

```
prompt$ javac First.java
```

Viene creato il bytecode First.class

```
prompt$ java First
```

```
1
```

```
0
```

```
prompt$
```

Cosa è il **Java bytecode**?

- È il linguaggio della **Java Virtual Machine**
- Load & store (e.g. `aload_0`, `istore`)
- Arithmetic & logic (e.g. `ladd`, `fcmpl`)
- Object creation & manipulation (`new`, `putfield`)
- Operand stack management (e.g. `swap`, `dup2`)
- Control transfer (e.g. `ifeq`, `goto`)
- Method invocation & return (e.g. `invokespecial`, `areturn`)
- Visualizzabile con `javap` !!
 - E anche editabile: ad esempio l'appena rilasciato [BCEL](#)

Creare oggetti

Dichiarare una variabile di tipo **Counter**

Invocare il costruttore per **creare** l'oggetto di tipo **Counter**

```
Counter c;  
c = new Counter( );
```

Counter è il tipo!!!

Soluzione alternativa: fare tutto in un passo!!

```
Counter c = new Counter( );
```

Costruttori con parametri

```
public class Counter {  
    // nome della classe  
  
    private int cnt; // lo stato locale  
  
    // metodo costruttore  
    public Counter (int v0) {cnt = v0; }  
    // metodo  
    public int inc ( )  
        { cnt++; return cnt; }  
    // metodo  
    public int dec ( )  
        { cnt--; return cnt; }  
}
```

DICHIARAZIONE
DI CLASSE

Costruttori con parametri

```
public class First {  
    public static void main(String[ ] args) {  
        Counter c = new Counter(25);  
        System.out.println(c.inc( ));  
        System.out.println(c.dec( ));  
    }  
}
```

quale è il valore dello stato locale?

Strutture mutabili

Ogni variabile di oggetto in Java denota una entità mutabile

```
Counter c;  
c = new Counter(5);  
c = new Counter(10);  
int x=c.inc( );
```

quale è il valore dello stato locale?

Assegnamento

- » Cosa succede quando assegnamo il valore ad una variabile che è di tipo riferimento?

```
c = new Counter(5);  
d = new Counter(10);  
c = d;
```

L'oggetto denotato da **d** viene assegnato a **c** di conseguenza le due variabili si riferiscono allo stesso oggetto (aliasing)!!!

Il valore **NULL**

Il valore **NULL** è generico e può essere assegnato a qualunque variabile di tipo riferimento

Restituisce un oggetto di tipo **Counter** o **NULL** se non lo trova



```
Counter c = cercaContatore( );  
if (c== null) System.out.println("contatore non trovato");
```

Attenzione: come in C
= **singolo**: assegnamento
== **doppio**: test di uguaglianza

Nello **heap**...

- Gli oggetti Java sono memorizzati nello **heap**
- Nello **heap** vengono allocate
 - variabili di istanza, quando si crea un oggetto
 - variabili statiche (o di classe), quando è caricata una classe
- Le variabili allocate nello **heap** sono inizializzate dal sistema
 - con **0** (zero), per le variabili di tipi numerici
 - con **false**, per le variabili di tipo **boolean**
 - con **null**, per le variabili di tipo riferimento
- Le variabili dichiarate localmente in metodi/costruttori non vengono inizializzate per default: bisogna assegnare loro un valore prima di leggerle

Stato locale

- **Modificatori:** meccanismo per controllare l'accesso allo stato locale dell'oggetto
 - **public:** visibile/accessibile da ogni parte del programma
 - **private:** visibile/accessibile solo all'interno della classe
- **Design Pattern (suggerimento grossolano)**
 - **Tutte le variabili di istanza: private**
 - **Costruttori e metodi: public**

Frammento Imperativo

- Il “frammento imperativo” di Java richiama da vicino la sintassi del C
 - `int x = 3;` //dichiara x e lo inizializza al valore 3
 - `int y;` //dichiara y e gli viene assegnato il valore di default 0
 - `y=x+3;` //assegna a y il valore di x incrementato di 3

Alcuni comandi...

- **Condizionali**

- `if (cond) stmt1;`
- `if (cond) { stmt1; stmt2; }`
- `if (cond) { stmt1; stmt2; } else { stmt3; stmt4; }`

- **Iterativi**

- `while (exp) { stmt1; stmt2; }`
- `do { stmt1; stmt2; } until (exp);`
- `for (init; term; inc) { stmt1; stmt2; }`

Tipi primitivi

- **int** // *standard integers*
- **byte, short, long** // *other flavors of integers*
- **char, float** // *unicode characters*
- **double** // *floating-point numbers*
- **boolean** // *true and false*
- **String** *non è un tipo primitivo (ma quasi...)*

Interfacce in Java

Tipi in Java

- Java è un linguaggio **fortemente tipato** (ogni entità ha un tipo)
- Le classi definiscono il tipo degli oggetti
- Java prevede un ulteriore meccanismo per associare il tipo agli oggetti: **le interfacce**

Interfacce in Java

- Una **interfaccia** definisce il *tipo degli oggetti in modo astratto*
- Non viene presentato nessun dettaglio dell'implementazione
- Non hanno costruttore e quindi neanche oggetti
- **Interfaccia=contratto d'uso dell'oggetto**
- **I sotto-tipi definiscono le implementazioni del tipo dato astratto**

Esempio

Nome del tipo

```
public interface Displaceable {  
    public int getX ( );  
    public int getY ( );  
    public void move(int dx,int dy);  
}
```

Dichiarazione
dei tipi dei metodi

Una implementazione...

```
public class Point
    implements Displaceable {
    private int x, y;

    public Point (int x0, int y0) {
        x = x0;
        y = y0;
    }
    public int getX( ) { return x; }
    public int getY(){return Y;}
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Devono essere implementati tutti i metodi dell'interfaccia

Point
ha solo i
metodi
dell'inter
faccia

Un'altra

```
public class ColorPoint
    implements Displaceable {
    private Point p;    \\ Delega all'oggetto Point
    private Color c;

    public ColorPoint (int x0, int y0, Color c0) {
        p=new Point (x0,y0); c=c0;    }
    public int getX() {return p.getX();}
    public int getY( ) {return p.getX(); }
    public void move(int dx, int dy) {
        p.move(dx,dy);}
    public Color getColor( ) {return c; }
}
```

Oggetti che implementano la stessa interfaccia possono avere stato locale differente e un numero maggiore di metodi di quelli previsti dal contratto

Tipi e interfacce

- Dichiarare variabili che hanno il tipo di una interfaccia

```
Displaceble d;  
d = new Point(1, 2);  
int x=d.getX();
```

- Assegnare una implementazione

```
d = new ColorPoint(1, 2, new Color("red"));  
int x=d.getX();
```

ColorPoint e **Point** sono sottotipi di **Displaceble**
!!!!

Sotto-tipi

- La situazione descritta illustra il fenomeno del **subtyping (sotto-tipo)**: un tipo A è un sotto-tipo di B se un oggetto di tipo A in grado di soddisfare tutti gli obblighi che potrebbero essere richiesti dall'interfaccia (o una classe) B
- Intuitivamente, un oggetto di tipo A può fare qualsiasi cosa che può fare (un oggetto di tipo) B
- Maggiori dettagli in seguito (quando **ColorPoint** potrà essere sotto-classe di **Point!**)

Interfacce multiple

```
public interface Area {  
    public double    getArea( );  
}
```

Interfacce Multiple

```
public class Circle implements Displaceable, Area {
    private Point center;
    private int rad;

    public Circle(int x0, int y0, int r0) {
        rad = r0; center = new Point(x0, y0);
    }

    public double getArea ( ) {
        return Math.PI * rad * rad;
    }

    public int getRadius( ) { return rad; }
    public getX( ) { return center.getX( ); }
    public getY( ) { return center.getY( ); }
    public move(int dx, int dy){center.move(dx, dy);}
}
```

Esempi d'uso

```
Circle c = new Circle(10,10,5);  
Displaceable d = c;  
Area a = c;
```

```
Rectangle r =new Rectangle (10,10,5,20);  
d = r;  
a = r;
```

Circle e' sottotipo di **Displaceable, Area!!!!**

Le stringhe in Java

Java String

- Le **stringhe (sequenze di caratteri)** in Java sono una classe predefinita
 - "3" + " " + "Volte 3" equivale a "3 Volte 3"
 - Il "+" è anche l'operatore di concatenazione di stringhe
- Le stringhe sono **oggetti immutabili** (a là OCaml)

Uguaglianza

- Java ha due operatori per testare l'uguaglianza
 - `o1 == o2` restituisce true se le variabili o1 e o2 denotano lo stesso riferimento (*pointer equality*)
 - `o1.equals(o2)` restituisce true se le variabili o1 e o2 denotano due oggetti identici (*deep equality*)
- Esempio
 - `new String("test").equals("test")` --> true
 - `new String("test") == "test"` --> false
 - `new String("test") == new String("test")` --> false

Un quesito più standard...

```
String str1 = new String("Java");  
String str2 = new String("Java");
```

```
str1.equals(str2) // true: stesso contenuto  
str1==str2 // false: oggetti differenti
```

Variabili e metodi statici

Variabili e metodi statici

- **Java** ha sia espressioni che comandi!
- Le espressioni restituiscono valori e i comandi operano via *side effects*
- Inoltre si possono dichiarare nelle classi *variabili statiche e metodi statici*
- **Variabili e metodi statici** appartengono alle classi

Metodi statici

```
public class Max {  
    public static int max (int x, int y) {  
        if (x > y) return x;  
        else return y;  
    }  
    public static int max3 (int x, int y, int z)  
        return max(max(x, y), z) ;  
}  
}
```

simile alla
definizione di una
funzione

```
public class Test {  
    public static void main (String[ ] args) {  
        System.out.println(Max.max(3, 4));  
    }  
}
```

A cosa servono?

- » Intuitivamente appartengono alla classe e non agli oggetti
- » I **metodi statici** non hanno accesso alle **variabili d'istanza degli oggetti della classe** (se esistenti)
- » Le **variabili statiche** servono per memorizzare valori comuni a tutti gli oggetti della classe
- » Possono essere modificate dai **metodi statici e anche dai metodi d'istanza della classe**
- » I **metodi statici** servono per realizzare funzioni e operazioni **indipendenti dai oggetti**

Esempio

```
public class Prova {  
    private int c;  
    .....  
    public static int add (int x) {  
        c=c+x;  
    }  
}
```

Un errore non si può modificare c