

## INTRODUZIONE

- Lo sviluppo di applicazioni complesse porta a costruire *moduli software* sempre più potenti e versatili , che possano essere *riutilizzati* in numerosi progetti
- I *linguaggi ad oggetti* (OOP, Object Oriented Programming) introducono delle strutture sintattiche adatte a scrivere programmi secondo questa filosofia

## INTRODUZIONE

- Le strutture dati sono estese alle *classi*: si raggruppano dati e funzioni per operare su tali dati (*tipo di dato astratto*)
- Gli *oggetti* sono creati a partire dalle classi
- L'*ereditarietà* permette di costruire nuove classi a partire da quelle esistenti estendendone le funzionalità
- L'*overloading* permette ad uno stesso operatore di avere specifici comportamenti su oggetti diversi
- Si può *proteggere e nascondere* l'implementazione fornendo un'interfaccia per gestire gli oggetti

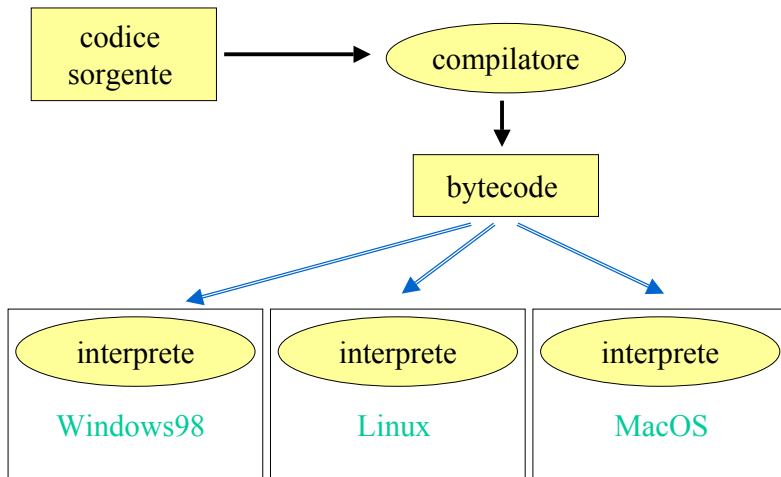
## JAVA

- Come esempio di linguaggio orientato agli oggetti si considera il *linguaggio di programmazione Java*
- Tale linguaggio è recente e ampiamente diffuso: permette agli utenti di Internet di poter utilizzare applicazioni *sicure e indipendenti dalla piattaforma*
- La tecnologia Java è stata sviluppata da un team della Sun Microsystems a partire dal 1991. Il rilascio ufficiale e l'integrazione in Internet di Java è avvenuto nel 1995.

## JAVA

- Java non è un acronimo, ma piuttosto si riferisce ad una particolare miscela di caffè...
- L'indipendenza dalla piattaforma si ottiene utilizzando sia una fase di *compilazione* sia una fase di *interpretazione*:  
il file sorgente è compilato nel formato *bytecode*, il "linguaggio macchina" di Java Virtual Machine (Java VM), in seguito la Java VM della specifica piattaforma interpreta il file bytecode e produce la funzionalità specificata dal programma

# JAVA



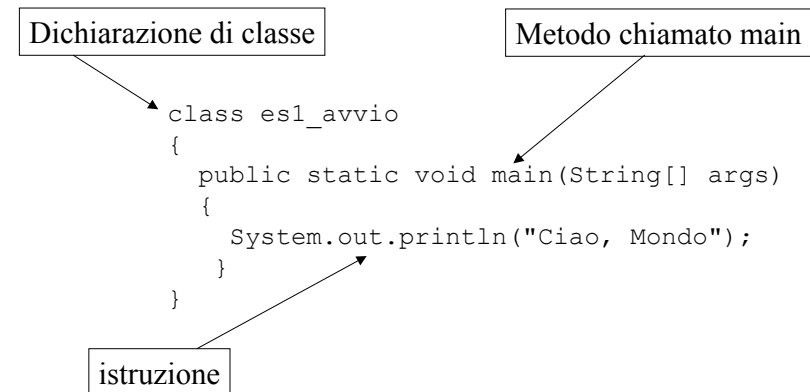
# JAVA

- L'*interprete* può essere sostituito da un *compilatore JIT* (Just In Time): modalità simile a quella degli interpreti, ma il codice tradotto viene memorizzato in modo da non ripetere la traduzione di istruzioni già eseguite da poco (migliori prestazioni, ma grande occupazione di memoria)
- La macchina virtuale Java può essere implementata in silicio, costruendo un apposito chip. Ciò non cambia la portabilità del codice: è solo un'altra implementazione della macchina virtuale Java

## CREARE IL PRIMO PROGRAMMA

- Java ha un aspetto familiare per chi conosce il C, in quanto per gli *aspetti comuni* sono stati utilizzati i costrutti di questo linguaggio
- I programmi Java sono costruiti a partire da *classi* che hanno due tipi di *membri*, detti *campi* e *metodi*: si può pensare ad una *struct del C* composta da *dati* e *funzioni* per operare sui dati
- Da una classe si possono creare *oggetti*, detti *istanze* della classe

## CREARE IL PRIMO PROGRAMMA



## CREARE IL PRIMO PROGRAMMA

- Il programma *dichiara una classe* di nome **es1\_avvio**, contenente un solo *metodo* chiamato **main** e nessun *campo*. I membri della classe sono contenuti tra parentesi graffe
- Il metodo **main** di una classe viene eseguito quando si esegue la classe come applicazione
- L'unico *parametro* di **main** è un *array di oggetti* di tipo **String** che costituiscono gli argomenti alla linea di comando del programma
- L'istruzione richiama un *metodo*, **println**, sullo *oggetto out* della *classe System*

## CREARE IL PRIMO PROGRAMMA

- Rispetto ad un programma C equivalente manca una *direttiva per il compilatore* del tipo **#include <stdio.h>**. Infatti ogni programma Java incorpora automaticamente una libreria di classi: **java.lang**
- Per usare funzionalità non presenti in **java.lang**, si devono importare altre librerie (**package**). Per esempio, gestire informazioni temporali comporta un'istruzione di questo tipo:  
**import java.util.\*;**

## CREARE IL PRIMO PROGRAMMA

- Vediamo dal punto di vista operativo come creare l'applicazione descritta:
  - si fa riferimento al prodotto *Java<sup>TM</sup> 2 SDK, Standard Edition Version 1.4.0*
  - dal prompt del DOS scrivere il codice sorgente in un file di testo con estensione `.java`  
(*notepad es1.java*)
  - Pseudo-compilare il codice sorgente nel bytecode (*javac es1.java*)
  - utilizzare l'interprete per lanciare l'applicazione (*java es1\_avvio*)

## CREARE IL PRIMO PROGRAMMA

```
B:\fondamenti1\java>notepad es1.java
B:\fondamenti1\java>javac es1.java
B:\fondamenti1\java>dir
Il volume nell'unità B è ZIP-100
Numero di serie del volume: 39BC-1102
Directory di B:\fondamenti1\java
23/08/00 09.04      <DIR>      ..
23/08/00 09.04      <DIR>      .
23/08/00 14.01      114 es1.java
23/08/00 14.13      417 es1_avvio.class
                4 File          531 byte
                18.176.000 byte disponibili
B:\fondamenti1\java>java es1_avvio
Ciao, Mondo
B:\fondamenti1\java>
```

## NOTA

- Consideriamo adesso la sintassi di Java, mettendo in evidenza solo le differenze con il C. Pertanto vedremo alcuni semplici programmi Java con una funzionalità di tipo *procedurale* tipica del C
- In seguito, avendo acquisito una certa familiarità con questo nuovo linguaggio, considereremo la programmazione orientata agli oggetti

## INSIEMI DI CARATTERI UNICODE

- Di solito il codice sorgente è scritto utilizzando il set di caratteri ASCII, invece Java è scritto in *Unicode*, un insieme di caratteri a 16 bit. Questo perché Java introduce il software internazionalizzato: i caratteri *Unicode* forniscono un insieme sufficiente per scrivere le più importanti lingue del mondo
- Gli ambienti Java correnti possono leggere file ASCII, convertendone il contenuto in *Unicode*
- Poiché pochi text editor sono in grado di utilizzare caratteri *Unicode*, Java riconosce *sequenze di escape* nella forma `\udddd`

## COMMENTI

- In Java esistono tre tipi di commenti
  - `/* commento */` come in C
  - `// commento` vengono ignorati i caratteri sino alla fine della linea
  - `/** commento */` come in C
- L'ultimo commento è detto di documentazione, perché utilizzando il tool *javadoc* si genera automaticamente la documentazione dell'applicazione in formato HTML (posizionare prima di classi e membri)

## COMMENTI

```
/** La classe es1a_avvio implementa un'applicazione
che visualizza semplicemente
"Ciao, Mondo" sullo standard output */

public class es1a_avvio
{
    public static void main(String[] args)
    {
        System.out.println("Ciao, Mondo");
    }
}
```

```
B:\fondamenti1\java>javadoc es1a.java
```

## COMMENTI

All Classes  
[es1a\\_avvio](#)

**Class** [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS NEXT CLASS  
SUMMARY: INNER | FIELD | [CONSTR](#) | [METHOD](#)      FRAMES NO FRAMES  
DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

**Class es1a\_avvio**

`java.lang.Object`  
↳ `es1a_avvio`

public class `es1a_avvio`  
extends `java.lang.Object`

La classe `es1a_avvio` implementa un'applicazione che visualizza semplicemente "Ciao, Mondo" sullo standard output

**Constructor Summary**

`es1a_avvio()`

**Method Summary**

*Annotations:* `java.lang.Object` is circled in red. A red box labeled "Automaticamente incorporato" points to it. A red box labeled "commento" points to the class description text.

## VARIABILI, ESPRESSIONI E I/O

- Consideriamo un programma che calcola il valore dell'ipotenusa di un triangolo rettangolo, ne stampa il valore e riceve i valori dei cateti alla linea di comando. Inoltre controlla che le dimensioni dei cateti siano positive.
- Un esempio di output di tale programma :

```
B:\fondamenti1\java>java es2_avvio 3 4
```

Argomenti alla linea di comando

Il triangolo rettangolo di cateti 3.0 e 4.0 ha l'ipotenusa di valore 5.0

## VARIABILI, ESPRESSIONI E I/O

```
/** La classe es2_avvio ... alla linea di comando */  
class es2_avvio  
{  
    public static void main(String[] args)  
    {  
        double c1=0,c2=0;  
        c1=Double.parseDouble (args[0]);  
        c2=Double.parseDouble (args[1]);  
        if (c1>0 && c2>0){  
            double ip=Math.sqrt (Math.pow(c1,2) + Math.pow(c2,2));  
            System.out.println("\n Il tr. rett. di cateti " +  
                c1 + " e " + c2 + " ha l'ipot. di valore " + ip);  
        }else  
            System.out.println(" \n Errore nei valori dei cat.");  
    }  
}
```

## TIPI E VARIABILI

- I tipi di dati primitivi in Java sono: **boolean**, **char** (16 bit), **byte**, **short**, **int**, **long**, **float**, **double**
- Java è stato progettato per massimizzare la portabilità, pertanto i tipi hanno lunghezze in bit predefinite (e.g. un **int** è un intero di 32 bit con segno)
- Le variabili devono essere inizializzate al momento della *dichiarazione*.
- La *dichiarazione* di una variabile può apparire in un qualunque punto del codice sorgente

## TIPI E VARIABILI

- Una variabile dichiarata in un *blocco* o in un ciclo *for* scompare al termine degli stessi

```
int c=0;
for(int i=0;i<3;i++){
    c+=1;
    System.out.println("c=" + c ); //+ " i=" + i);
```

La variabile i è disponibile solo all'interno del ciclo

Errore in fase di compilazione se inserito nel codice sorgente

## INPUT

- Gli argomenti alla linea di comando sono gestiti in modo equivalente al C.

```
c1= Double.parseDouble (args[0]);
```

- Per convertire le stringhe in valori numerici si utilizza un metodo: `Double.parseDouble ()`
- Molti tipi primitivi hanno classi che li rappresentano. Le *classi wrapper* (involucro) forniscono un ambiente per i metodi e le variabili legate al tipo

## FUNZIONI MATEMATICHE

- La *classe Math* è costituita da costanti statiche e metodi per le manipolazioni matematiche comuni. Tutte le operazioni vengono eseguite in rappresentazione *double*

```
double ip=Math.sqrt(Math.pow(c1,2) + Math.pow(c2,2));
```

- Alcuni esempi: `Math.PI`  
`Math.sin(a)`  
`Math.exp(a)`  
`Math.max(x,y)`  
`Math.abs(a)`

## PRINTLN()

- La chiamata a `println()` è più complessa , in quanto utilizza l'operatore `+` per concatenare una stringa con un'altra stringa che rappresenta la variabile `c`

```
System.out.println("c=" + c );
```

- La stampa del solo valore (`System.out.println(c );`) rappresenta il primo esempio di *overloading* di metodo in quanto `println()` può ricevere argomenti di tipo diverso: stringhe o interi

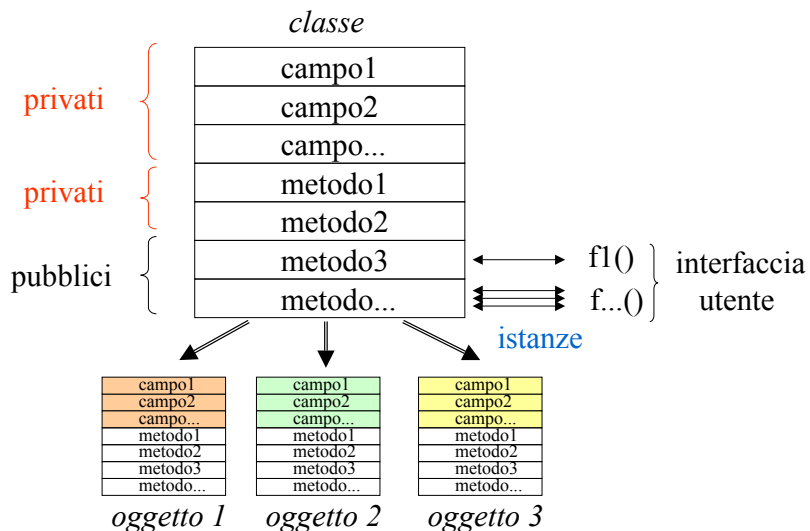
## CLASSI E OGGETTI

- In Java l'unità fondamentale dei programmi è la *classe*
- Le *classi* contengono i *metodi*, collezioni di istruzioni, che elaborano i dati contenuti nei *campi*, che costituiscono lo *stato* dell'oggetto
- Gli *oggetti*, creati (*istanziati*) dalla *classe*, hanno un *tipo*, che è la *classe* dell'oggetto
- La programmazione ad oggetti distingue nettamente la nozione di *che cosa* deve essere fatto da *come* viene fatto

## CLASSI E OGGETTI

- Il *che cosa* viene descritto mediante un insieme di metodi (e talvolta di dati disponibili pubblicamente) con le relative semantiche
- Il *come* un oggetto è realizzato viene definito dalla sua classe, mediante l'implementazione dei metodi che l'oggetto supporta. L'utente finale può non conoscere l'implementazione

## CLASSI E OGGETTI



## CLASSI E OGGETTI

- Realizziamo una classe, di nome `Point`, utile a rappresentare i punti del piano

```
public class Point{
    public double x;
    public double y;
}
```

- Una dichiarazione di classe crea un *nome di tipo*, quindi posso *dichiarare* un oggetto scrivendo

```
Point p1;
```

- La dichiarazione *non* crea un oggetto, ma solo un *riferimento* a un oggetto di tipo `Point`

## CREAZIONE DI OGGETTI

- L'oggetto cui si  *riferisce*  p1 viene creato con l'operatore **new**, specificando il tipo dell'oggetto che si vuole creare.

```
p1=new Point();
```

- Si inizializza l'oggetto creato con opportuni valori

```
p1.x=1;  
p1.y=2;
```

- Poiché a differenti oggetti corrispondono differenti istanze dei campi, ogni oggetto ha un proprio unico stato

## CREAZIONE DI OGGETTI

```
/** La classe Point implementa ... punti del piano */  
public class Point{  
    //campi  
    public double x;  
    public double y;  
    //metodi  
    public static void main(String[] args){  
        Point p1,p2;  
        p1=new Point();  
        p2=new Point();  
        p1.x=1; p1.y=2;  
        p2.x=5; p2.y=6;  
        System.out.println(p1.x + " " + p1.y);  
        System.out.println(p2.x + " " + p2.y);  
    }  
}
```

```
Z:\fabio\fondamenti1\java>java Point  
1.0 2.0  
5.0 6.0
```

## COSTRUTTORI

- Quando si crea un nuovo oggetto, gli si  *deve*  associare uno stato iniziale (si pensi agli errori che possono generare le operazioni tra variabili non inizializzate, inoltre un oggetto è più complesso di un tipo primitivo). Per questo motivo le classi hanno dei  *costruttori* .
- I costruttori sono metodi particolari che hanno lo stesso nome della classe che inizializzano

```
Point(){  
    x=0; y=0;  
}  
•Non ha un tipo di ritorno  
•Accede direttamente ai campi
```

## PROTEZIONE DATI

- Il controllo dell'accesso (per la sicurezza e la gestione futura dell'implementazione) è fornito da modificatori :  
 *public, private, protected, package*
- Il costruttore è utile anche per permettere di inizializzare dati protetti, cioè non resi accessibili direttamente
- Risulta utile spezzare il file sorgente visto in due file: nel file Point.java la classe Point e nel file Avvio.java una classe che contiene il metodo main() e utilizza la classe Point

## PROTEZIONE DATI

Point.java

```
public class Point{
    private double x;
    private double y;
    Point(){
        x=0; y=0;
    }
}
```

Avvio.java

```
class Avvio{
    public static void main(String[] args){
        Point p1,p2;
        p1=new Point();
        p1.x=1; p1.y=2;
    }
}
```

```
Z:\fabio\fondamenti1\java>javac Avvio.java
Avvio.java:8: x has private access in Point
    p1.x=1; p1.y=2;
    ^
Avvio.java:8: y has private access in Point
    p1.x=1; p1.y=2;
    ^
2 errors
```

## METODI

- Si rende necessario fornire agli oggetti (dato che possono essere protetti) la capacità di modificare il proprio stato: *metodi*
- I metodi vengono *invocati* come operazioni su oggetti, utilizzando l'operatore `.` applicato ai riferimenti *riferimento.metodo(parametri)*
- Nell'esempio si fornisce la possibilità di modificare i valori dei punti, di accedere ai valori dei singoli campi e di controllare tali valori

## METODI

```
/** La classe Point implementa ... dei punti del piano */
public class Point{
    private double x;
    private double y;

    Point(){ x=0; y=0;}

    public void Set(double a, double b){
        if (a>0 && b>0){ ←
            x=a; y=b;}
        else
            System.out.println("I valori non sono positivi");}

    public double Getx(){ return x;}

    public double Gety(){ return y;}
}
```

## METODI

```
class Avvio{

    public static void main(String[] args){
        Point p1=new Point();
        Point p2=new Point();
        p1.Set(-3,4); ←
        p2.Set(5,6);

        System.out.println("p1 "+p1.Getx()+" "+p1.Gety());
        System.out.println("p2 "+p2.Getx()+" "+p2.Gety());
    } }
```

```
Z:\fabio\fondamenti1\java>javac Avvio.java
Z:\fabio\fondamenti1\java>java Avvio
I valori non sono positivi
p1 0.0 0.0
p2 5.0 6.0
```

## METODI: THIS

- Si utilizza il riferimento **this** all'interno di metodi per invocare un metodo con riferimento all'oggetto corrente

```
public void Set(double x, double y){
    this.x=x; this.y=y;
}
```

- In questo modo si evita che i parametri avendo lo stesso nome dei campi *nascondano* i nomi dei campi

## METODI: VALORI DEI PARAMETRI

- In Java tutti i parametri sono passati ai metodi *per valore*. Questo significa che i valori delle variabili dei parametri in un metodo sono *copie* dei valori specificati al momento dell'invocazione
- Quando il parametro è un riferimento a un oggetto, tuttavia, si passa *per valore* il riferimento e non l'oggetto. In questo modo è possibile modificare l'oggetto cui il parametro si riferisce all'interno del metodo, senza intaccare il riferimento che è stato passato

## METODI: VALORI DEI PARAMETRI

```
class Avvio{
public static void main(String[] args){
    Point p1=new Point();
    p1.Set(3,4);
    System.out.println("prima " + p1.Getx());
    DividoPer2(p1.Getx());
    System.out.println("dopo " + p1.Getx());
}
public static void DividoPer2(double x){
    x/=2;
    System.out.println("in " + x);
}
}
```

```
Z:\fabio\fondamenti1\java>java Avvio
prima 3.0
in 1.5
dopo 3.0
```

## METODI: VALORI DEI PARAMETRI

```
class Avvio{
public static void main(String[] args){
    Point p1=new Point();
    p1.Set(3,4);
    System.out.println("prima " + p1.Getx());
    DividoPer2(p1);
    System.out.println("dopo " + p1.Getx());
}
public static void DividoPer2(Point arg){
    arg.Set((arg.Getx()/2),arg.Gety());
    System.out.println("in " + arg.Getx());
}
}
```

```
Z:\fabio\fondamenti1\java>java Avvio
prima 3.0
in 1.5
dopo 1.5
```

## OVERLOADING DI METODI

- In Java ogni metodo ha una *segnatura*, costituita dal nome, dal numero e dai tipi dei suoi parametri. Possono essere definiti più metodi con lo stesso nome se le rispettive segnature differiscono nel numero o nel tipo dei parametri. Questa caratteristica è detta *overloading* (sovraccaricamento) in quanto al nome di un metodo vengono associati più significati.
- Definiamo un nuovo *costruttore* che permetta l'inizializzazione dei campi al momento della creazione

## OVERLOADING DI METODI

*Point.java*

```
public class Point{
    ...
    Point(double x, double y){
        this.x=x; this.y=y;
    }
    ...
}
```

```
Z:\fabio\...>java Avvio
p1 0.0 0.0
p2 3.0 4.0
```

*Avvio.java*

```
class Avvio{
public static void main(String[] args){
    Point p1=new Point();
    System.out.println("p1 "+p1.Getx()+" "+p1.Gety());
    Point p2=new Point(3,4);
    System.out.println("p2 "+p2.Getx()+" "+p2.Gety());
}
}
```

## GARBAGE COLLECTION

- In Java è previsto un sistema di *garbage collection* (eliminazione dei rifiuti) automatico che rende superflua la necessità di eliminare esplicitamente gli oggetti
- Gli oggetti vengono creati utilizzando *new*, ma non esiste un'operazione di distruzione *delete* corrispondente
- In pratica quando un oggetto non è più referenziato lo spazio che esso occupa viene recuperato senza bisogno che il programmatore compia alcuna operazione

## STATIC

- Talvolta è utile disporre di un'unica istanza di un campo, che sia *condiviso* da tutti gli oggetti di una classe
- Ciò si ottiene dichiarando un campo statico mediante la parola riservata **static**. Questo campo viene inizializzato al momento dell'inizializzazione della classe (*I metodi statici sono pensati per compiere operazioni specifiche della classe stessa, di solito su campi statici, e non su particolari istanze della classe: e.g. Math.sqrt()*)
- Dichiariamo un nuovo campo che memorizzi il numero di punti creati e modifichiamo di conseguenza i costruttori

```
public class Point{
    ...
    protected static int n=0;
    Point(){
        x=0; y=0; n++;
    public int Number(){
        return n;}
    ...
    }}

```

```
Z:\fabio\...>java Avvio
p1 2
p2 2

```

```
class Avvio{
    public static void main(String[] args){
        Point p1=new Point();
        Point p2=new Point();
        System.out.println("p1 "+p1.Number());
        System.out.println("p2 "+p2.Number());
    }}

```

- I componenti di un array possono essere tipi primitivi o riferimenti a oggetti, compresi riferimenti ad altri array. Dichiarazione di un vettore

```
int[] vect = new int[3];

```

- Il primo elemento di un array ha indice 0.
- La lunghezza di un array è disponibile tramite il campo *length*. Il seguente codice stampa il contenuto dell'array precedente

```
for(int i=0;i<vect.length;i++)
    System.out.println(i+" : " + vect[i]);

```

ARRAY

- Dichiarando un array di un tipo di oggetti, in realtà si dichiara un array di *riferimenti* di quel tipo, quindi gli oggetti si devono creare in un ciclo successivo

```
Point[] vect = new Point[3];
//System.out.println(" " + vect[1].GetX());
for(int i=0;i<vect.length;i++)
    vect[i]= new Point(i,i);
System.out.println("array " + vect[2].GetX());

```

```
Z:\fabio\...>java Avvio
array 2.0

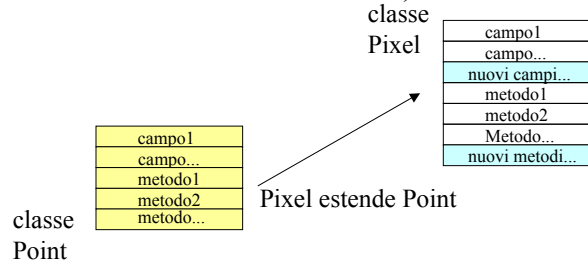
```

ESTENSIONE DELLE CLASSI

- Uno dei vantaggi più importanti della programmazione orientata agli oggetti è la possibilità di *estendere* una classe, o di *costruire sottoclassi*. Quando si estende una classe, si crea una nuova classe che *eredita* tutti i campi e metodi della classe originale. La classe su cui è basata l'estensione viene detta *superclasse*
- La parte di *contratto* (metodi e campi accessibili al di fuori della classe e il comportamento atteso) ereditata non deve essere modificato, ma solo esteso

## ESTENSIONE DELLE CLASSI

- Una classe si estende con la clausola **extends**
- Estendiamo la classe Point per rappresentare i pixel di uno schermo: abbiamo bisogno di un nuovo campo per il colore e di metodi diversi (*i campi private devono essere resi protected per essere accessibili direttamente dalle classi estese*)



## SUPER E OVERRIDING

- Per garantire un comportamento corretto, i nuovi metodi della classe estesa richiamano i metodi della superclasse con il riferimento **super**
- *Ridefinire* (overriding) un metodo significa sostituire l'implementazione di un metodo della superclasse con una nuova implementazione

## UNA CLASSE ESTESA

```
/** La classe Pixel estende la classe Point*/
public class Pixel extends Point{
    private String Color;

    Pixel(String arg){
        super();
        Color=arg;
    }

    public String Get(){
        return ("Pixel: ("+super.Getx()+", "+super.Gety()+
            +") "+Color);
    }

    public int Number(){
        System.out.println("Sono stati creati "+n+" oggetti");
        return n;
    }
}
```

## UNA CLASSE ESTESA

```
class Avvio{
    public static void main(String[] args)
    {
        Pixel p1=new Pixel("red");
        Pixel p2=new Pixel("green");
        p1.Set(3,4);

        System.out.println(p1.Get());
        System.out.println("p1 "+p1.Getx()+" "+p1.Gety());
        p1.Number();
    }
}
```

```
Z:\fabio\fondamenti1\java>java Avvio
Pixel: (3.0,4.0) red
p1 3.0 4.0
Sono stati creati 2 oggetti
```

## POLIMORFISMO

- Gli oggetti Pixel possono essere utilizzati dal codice scritto per gli oggetti Point. Per esempio, se il parametro di un metodo è di tipo Point, è possibile fornire al suo posto un oggetto Pixel.
- Questa caratteristica è detta *polimorfismo*: un oggetto di tipo Pixel può avere molte (*poli-*) forme (*-morfismo*) e può essere utilizzato sia come oggetto di tipo Pixel sia come oggetto di tipo Point

## POLIMORFISMO

```
class Avvio{
public static void main(String[] args){
    Pixel p1=new Pixel("red");
    p1.Set(3,4);
    System.out.println(p1.Get());
    DividoPer2(p1);
}
public static void DividoPer2(Point arg){
    arg.Set((arg.Getx()/2),arg.Gety()/2);
    System.out.println("in "+arg.Getx()+" "+arg.Gety());
}}
```

```
Z:\fabio\fondamenti1\java>java Avvio
Pixel: (3.0,4.0) red
in 1.5 2.0
```

## LA CLASSE OBJECT

- Le classi che non siano estensioni esplicite di altre classi estendono implicitamente la classe *Object*, ereditandone quindi i metodi.
- In altri termini ciò significa che i riferimenti a istanze della classe *Object* sono riferimenti generici che possono essere usati per oggetti di qualunque classe

## JAVADOC

**Class** [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS NEXT CLASS [FRAMES](#) [N](#)  
SUMMARY: INNER | [FIELD](#) | CONSTR | [METHOD](#) [DETAIL](#): [FIE](#)

### Class Pixel

java.lang.Object  
|  
+--Point  
|  
+--Pixel

public class Pixel  
extends Point

La classe Pixel estende la classe Point

#### Fields inherited from class Point

n

#### Method Summary

java.lang.String [Get](#) ()

#### Methods inherited from class Point

Getx, Gety, Set

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, toString, wait, wait, wait

#### Method Detail

##### Get

public java.lang.String [Get](#) ()

##### Number

public int [Number](#) ()

Overrides:  
Number in class Point