

# Introduzione al Linguaggio JAVA

Marco Gori  
Marco Maggini  
Fabrizio Santini



## Programmazione ad Oggetti

Il paradigma di Programmazione orientata agli oggetti (OOP) migliora il supporto per lo sviluppo e la gestione del software.

Linguaggi ad oggetti: [JAVA](#), [C++](#), [SmallTalk](#), [Modula-2](#)

## Concetti Base della OOP

- Tipo di dato astratto:  
**Modello + insieme di operazioni = Incapsulamento**
- Classe:
  - Descrive il comportamento globale
  - Fornisce un elenco di metodi utilizzabili con il dato astratto
  - Descrive i dettagli sulla implementazione e sulla struttura del dato, separando gli elementi della classe in:
    - Pubblici**: che possono essere utilizzati anche da altre classi
    - Privati**: che possono essere utilizzati solo nella classe di appartenenza

## Concetti Base della OOP

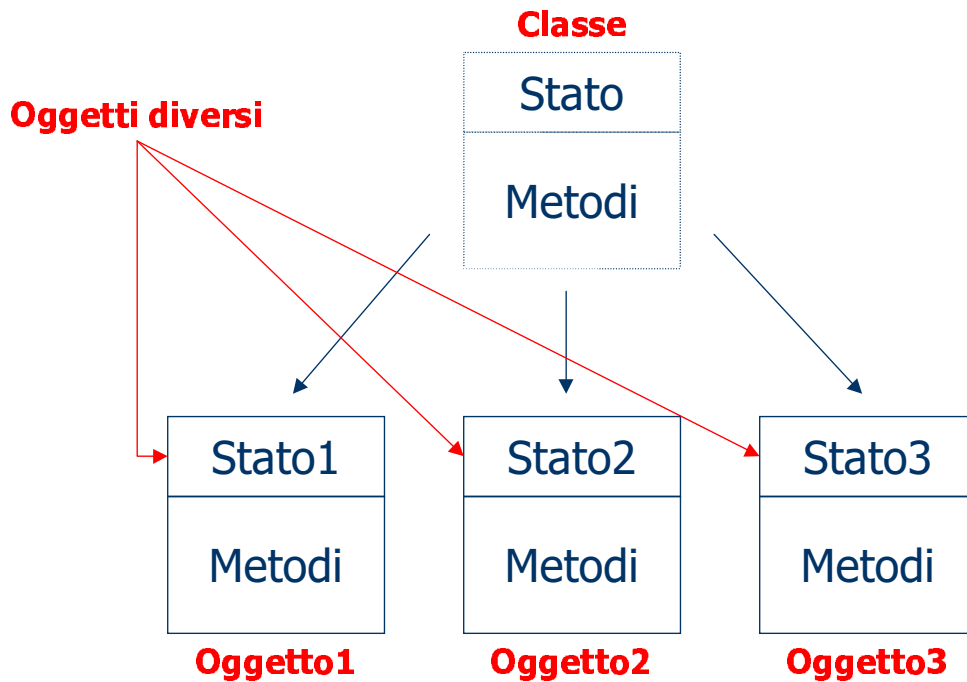
- Oggetto: **Variabile dichiarata come appartenente ad una classe**. È composta da uno **stato** e dai **metodi**.

Lo stato è rappresentato dai valori dei campi (pubblici e privati) descritti nella definizione della classe

I metodi sono procedure che possono essere applicate ad una istanza e modificare lo stato dell'oggetto stesso.

L'oggetto è una istanza fisica (che occupa memoria) di una classe. Due oggetti A e B appartenenti alla stessa classe sono due entità diverse perché hanno stati diversi.

# Concetti Base della OOP



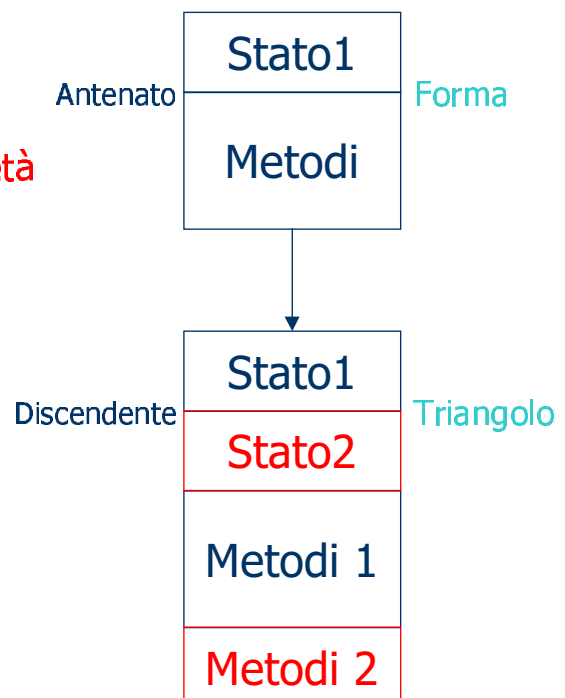
5

# Concetti Base della OOP

- Ereditarietà delle classi

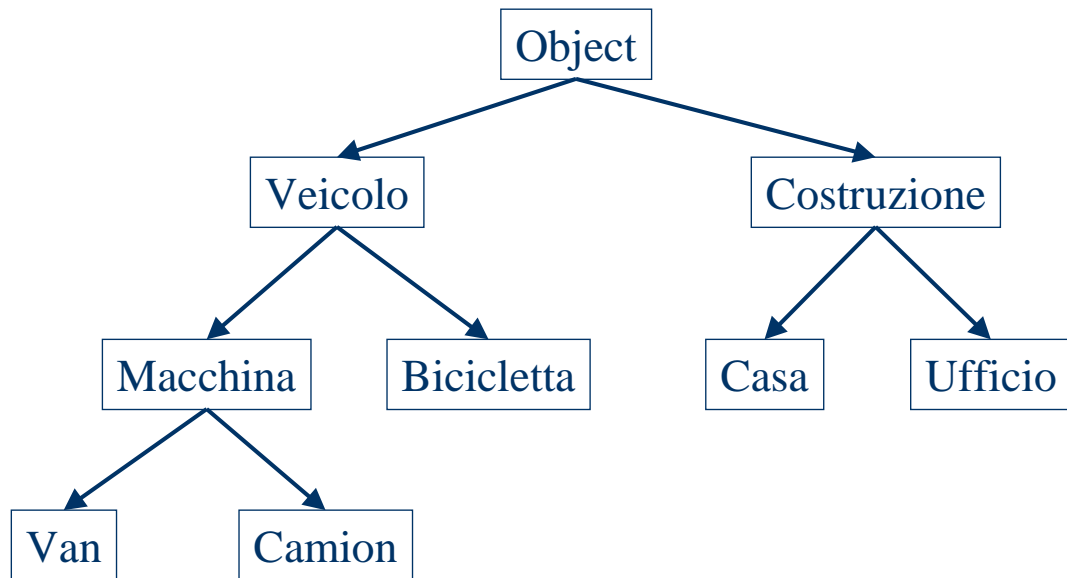
È possibile utilizzare l'**Ereditarietà** per creare nuove classi estendendo quelle già esistenti con nuove proprietà e nuovi metodi.

La ridefinizione nella classe discendente di un metodo già presente nell'antenato viene chiamato **Overriding**.



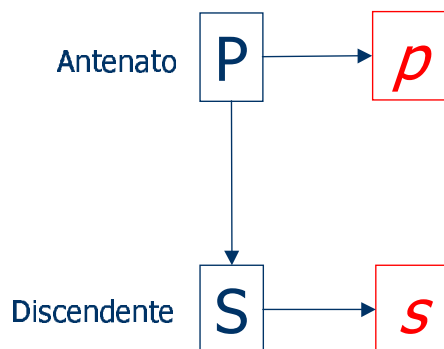
6

# Ereditarietà



7

# Ereditarietà e Polimorfismo



**Ereditarietà:** È possibile utilizzare *s* ovunque viene usato *p*

**Polimorfismo:** l'oggetto risponde a un messaggio comune a *p* ed a *s* a seconda della classe di appartenenza

8

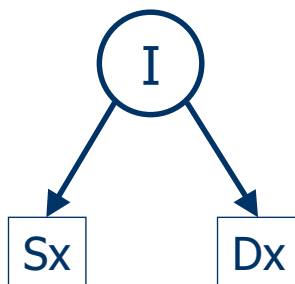
# Programmazione

- Fase di **progettazione**:
  - Definizione delle classi, discendenti e loro proprietà
  - Definizione delle interfacce delle classi
- Fase di **implementazione**:

Definizione della implementazione (algoritmi) dei metodi

Vantaggi: Riutilizzabilità del codice, facilità di manutenzione e quindi riduzione dei tempi e costi di sviluppo.

## Esempio di classe



```
class Nodo
  private
    oggetto Informazione
    oggetto Sottoalbero_Sx
    oggetto Sottoalbero_Dx
  public
    metodo Nuovo_Nodo
    metodo Get_Informazione
    metodo Get_Destro
    metodo Get_Sinistro
```

## Riutilizzo del codice

- Aggregazione: Relazione "ha-un"

È possibile includere un oggetto come elemento del nuovo oggetto:

- Semplice
- Flessibile perché permette di aggiungere nuovi oggetti a tempo di esecuzione

- Ereditarietà: Relazione "è-un" :

È possibile ereditare tutti i metodi e le proprietà della classe antenato.

11

## JAVA - la storia

- Primavera 1990 - Patrick Naughton vuole lasciare Sun, ... ma cambia idea ...
- Viene sviluppato da un piccolo team diretto da James Gosling, contemporaneamente al suo utilizzo in alcuni progetti interni alla Sun
- Originariamente si chiamava Oak (Quercia), ma prende il nome definitivo al coffee shop della Sun
- Viene lanciata FirstPerson Inc (Autunno 1992)

12

# JAVA - la storia

- Primavera 1993 - nasce NCSA Mosaic
- Giugno 1994 - Bill Joy (co-fondatore Sun) intuisce il legame con il Web ... e spinge ...
- Autunno 1994 - Naughton e Payne sviluppano Webrunner (browser Sun) per girare applets
- Primavera 1995 - Sun introduce formalmente Java e HotJava (ex-Webrunner)
- Fine 1995 - Accordo con Netscape per Java-enabled browser

13

## Caratteristiche di JAVA

- JAVA è semplice:

È un linguaggio molto simile a C++, ma più semplice. Tutto quello non strettamente necessario e le caratteristiche che potevano indurre confusione sono state eliminate:

- Templates
- overloading degli operatori
- pre-processore
- gli header file
- strutture ed unioni
- Array multi-dimensionali
- Conversione implicita dei tipi



14

## Caratteristiche di JAVA

- JAVA è un linguaggio ad alto livello, orientato agli oggetti:

Il codice viene organizzato in classi.

Java implementa il meccanismo di singola eredità, quindi ogni classe può ereditare una o più caratteristiche e comportamenti da un singolo antenato. Alla radice dell'albero di tutte le classi c'è la classe progenitrice **Object**.

La maggior parte degli elementi del linguaggio sono oggetti, anche se i tipi semplici sono implementati anche tradizionalmente.



15

## Caratteristiche di JAVA

- JAVA è indipendente dalla macchina:

Un programma JAVA viene prima compilato in un formato binario chiamato **byte-codes**, e successivamente eseguito da una macchina virtuale (**JVM**).

Questa caratteristica gli conferisce una totale indipendenza dalla piattaforma di esecuzione.



16



## Caratteristiche di JAVA

- JAVA è multi-threaded:

Il JAVA è naturalmente multi-threaded. Questo permette di implementare in maniera estremamente semplice, applicazioni complesse che richiedono l'uso di multi-programmazione.

- JAVA utilizza tecniche di Garbage Collection
- JAVA è robusto agli errori di programmazione



17

## Caratteristiche di JAVA

- JAVA funziona bene anche su piccoli sistemi
- JAVA è intrinsecamente sicuro
- JAVA è produttivo



18

## Caratteristiche di JAVA

- Heavy Compile-Time Checking
- Heavy Run-Time Checking
- Java API (e.g. Abstract Window Toolkit)

19

## Caratteristiche di JAVA

- Il Pascal, il C ed altri classici linguaggi, richiedono che la deallocazione della memoria sia gestita dal programmatore
- Il JAVA utilizza tecniche per il garbage collection automatico

20

# Caratteristiche di JAVA

- Java e problemi di sicurezza

Perché?

Problemi derivanti dagli applets: girano su hosts remoti.

Chi garantisce il loro comportamento?

La politica di Sun: open-source

21

# Caratteristiche di JAVA

## HTML e Java Applets

```
<html>
<head>
<title>Felix Applet</title>
</head>
<body>
<h1> Felix il gatto </h1>
Signori ... ecco Felix il gatto ...
<hr>
<applet code = Felix.class width=500 height =
300>
ouh ... se vedi questo allora forse tu non hai
un <i>browser abilitato Java! </i>
</applet>
<hr>
</body>
</html>
```

22

# Un semplice programma JAVA

```
/*
 * Questo programma stampa la data di esecuzione
 * del mio primo programma JAVA
 */

package MyFirstJAVA;

import java.util.*; // Importa anche la classe 'Date'

public class HelloWorld {
    public static void main(String argv[]) {

        System.out.println("Hello World!");
        System.out.println("My first JAVA program in date: ");
        System.out.println(new Date());
    }
}
```

Il modo migliore di capire la filosofia JAVA è quello di studiare subito un semplice programma.



23

## Commenti in JAVA

In Java esistono due modi per scrivere commenti nel codice:

- **Commenti su una linea**

Il compilatore ignora tutto il testo che segue i due caratteri `//` fino alla fine della riga

```
public class Prova {
    public static void main(String argv[]) {
        System.out.println("Hello!"); // stampo il messaggio
    }
}
```

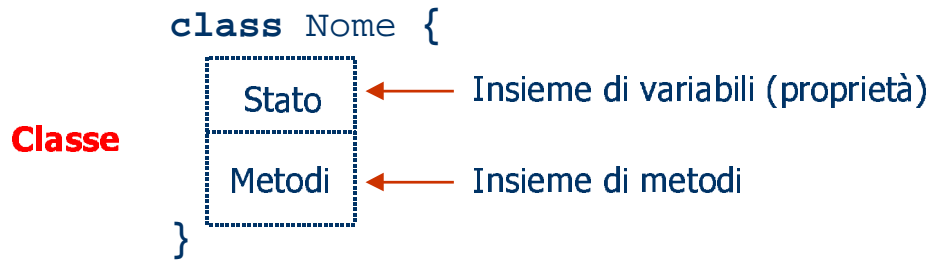
- **Commenti estesi**

Il compilatore ignora tutto il testo che è compreso fra le sequenze `/*` e `*/`

```
public class Prova {
    /* public static void main(String argv[]) {
        System.out.println("Hello!");
    } */
}
```

24

# Classi in JAVA



- Le proprietà memorizzano dei dati
- I metodi definiscono delle azioni applicabili alla classe

```
public class Conto {  
    private int valuta = 0;  
  
    public void versa(int somma) {valuta += somma;}  
    public void preleva(int somma) {valuta -= somma;}  
    public int estrattoConto() { return valuta;}  
}
```

stato

metodi

25

## JAVA: Access Specifiers

Esistono quattro vincoli diversi di accesso alle proprietà e metodi di una classe JAVA:

- Public → **Rende visibile all'esterno della classe**
- Private → **Nasconde all'esterno della classe**
- Protected → **Le classi discendenti possono accedere alle proprietà e metodi della classe progenitore.**
- "Frendly" → **Le classi discendenti possono accedere alle proprietà e metodi della classe progenitore.**

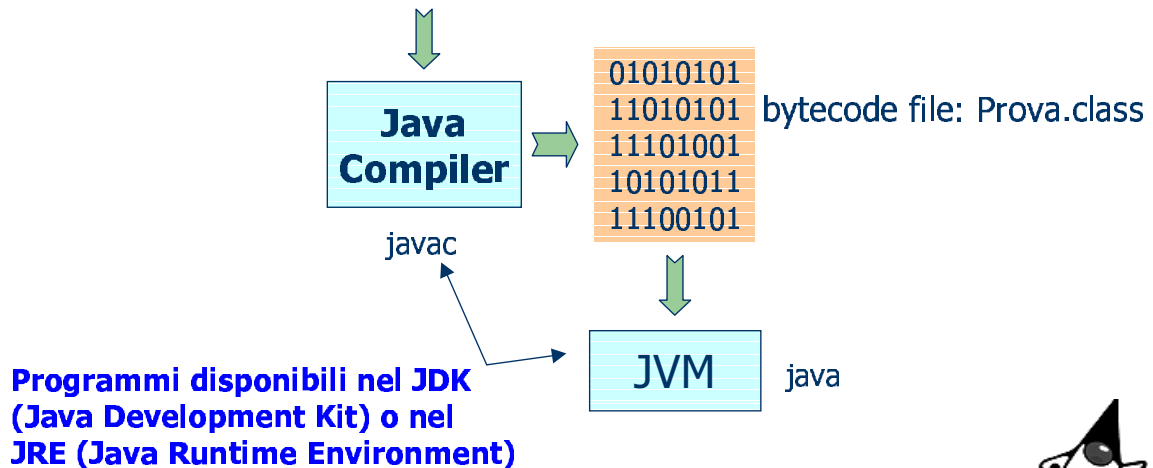
**Tutte le classi all'interno dello stesso package possono accedere alle proprietà e metodi della classe. Questo vincolo viene considerato di default se nessun altro specificatore viene indicato.**

26

# Cosa succede ad un file JAVA

File: Prova.java

```
class Prova {  
    public static void main(String argv[]) {  
        System.out.println("Hello World!");  
    }  
}
```



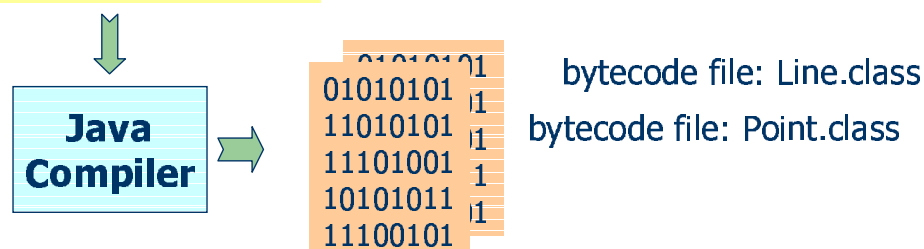
27

## Il Percorso di una classe JAVA

Il risultato della compilazione è un numero di file che corrisponde al numero di classi definite

File: Prova.java

```
public class Line {  
    int x0, y0;  
    int x1, y1;  
}  
  
class Point {  
    int x, y;  
}
```



28

## Librerie di classi in JAVA

Le classi omogenee possono essere raccolte all'interno di una libreria :

```
package MyLibrary;  
  
class Prova {  
    public static void main(String argv[]) {  
        System.out.println("Hello World!");  
    }  
}
```

che un altro utente può utilizzare in parte o completamente.

## Librerie di Classi in JAVA

È possibile importare classi esterne semplicemente indicando il percorso completo:

```
import MyLibrary.Prova;  
  
class Prova2 {  
    public static void main(String argv[]) {  
        Prova MyProva = new Prova;  
        System.out.println("Hello World!");  
    }  
}  
  
// import MyLibrary.Prova;  
  
class Prova2 {  
    public static void main(String argv[]) {  
        MyLibrary.Prova MyProva = new MyLibrary.Prova;  
        System.out.println("Hello World!");  
    }  
}
```

# Librerie di Classi in JAVA

È possibile importare anche tutte le classi di un pacchetto:

```
import MyLibrary.*;

class Prova2 {
    public static void main(String argv[]) {
        Prova MyProva = new Prova;
        System.out.println("Hello World!");
    }
}
```

In pratica lo spazio dei nomi delle librerie e delle classi JAVA assomigliano ai domini internet.

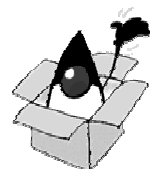
```
import Projects.Shared.MyLibrary.3D.*;
```

Il nome del pacchetto contiene anche il path relativo delle classi: `/Projects/Shared/MyLibrary/3D/`

31

# Oggetti in JAVA

- Le classi sono una descrizione teorica di ciò che potrà fare l'entità che vogliamo definire
- Per poterle utilizzare bisogna creare le istanze
- Si creano oggetti a partire da una classe utilizzando l'operatore `new`



32



# Oggetti in JAVA

Tutte le entità in JAVA sono oggetti

L'identificatore utilizzato per riferirsi ad un oggetto viene chiamato **handle**. Ovviamente un handle non è collegato necessariamente ad un oggetto.

Definizione:

- Solo Handles:  
`<nome-classe> nome-handle`
- Handle + Oggetto:  
`<nome-classe> nome-handle = oggetto;`



33

## Esempio

Definizione di una stringa

Def. di un handle:

```
String s;
```

In questo caso l'oggetto non esiste, e se provo ad accedervi si genera un run-time error.

Def. di un handle + oggetto

```
String s = "Java";
```

Valido solo per le stringhe

```
String s = new String("Java");
```

In questo caso l'handle viene definito e successivamente inizializzato con l'indirizzo dell'oggetto appena creato

**COSTRUTTORE DELL'OGGETTO**  
Questo metodo speciale indica come creare l'oggetto. Ci può essere più di un costruttore, ognuno dei quali possiede un insieme di parametri diversi (per tipo e/o numero).

34

## Dove “vivono” gli oggetti?

Lo **scope** (o campo di visibilità) è la zona in cui un oggetto JAVA vive. Questa zona è delimitata dai simboli `{` e `}`.

Non permesso perché la variabile `i` è stata già definita nello scope 1.

```
{
    int i = 5;
    {
        float w = 0.5;
        /* int i = 3; */
    }
}
```

Scope 2

Scope 1

35

## Quanto “vivono” gli oggetti?

La “vita” di tutti gli oggetti JAVA è gestita automaticamente: ci si preoccupa di creare un oggetto ma non di distruggerlo.

Questo è possibile perché esiste una entità, chiamata **Garbage Collector**, che provvede a liberare la memoria occupata da oggetti che non sono più utilizzati.

```
{
    Float W = new Float(0.5);
}
```

L'handle **W** scompare, ma non l'oggetto associato!

36

## E il main?

Se un programma JAVA è fatto con aggregazione di oggetti, allora una funzione **main** non esiste?

**In JAVA esiste il concetto di oggetto principale**

All'interno di questo oggetto esiste un metodo che viene chiamato automaticamente, all'avvio dell'oggetto stesso.

```
package MyLibrary;  
  
class Prova {  
    public static void main(String argv[]) {  
        System.out.println("Hello World!");  
    }  
}
```

37

## Tipi primitivi in JAVA

Tipi di uso frequente:

- Non vengono definiti gli handle ma le variabili vere e proprie. Questo è dovuto al costo di gestione del **new** per variabili semplici
- L'occupazione di memoria di questo tipo di variabili è indipendente dalla macchina ospite
- Per tutti i tipi primitivi esistono delle classi, chiamate **wrapper**, che permettono la creazione di oggetti del tipo associato in modo dinamica



38

# Tipi primitivi in JAVA

Tipo	Dimensione	Min	Max	Wrapper
boolean	1 bit	-	-	Boolean
char	16 bit	0	$2^{16}-1$	Char
byte	8 bit	-128	+127	Byte
short	16 bit	$-2^{15}$	$+2^{15}-1$	Short
int	32 bit	$-2^{31}$	$+2^{31}-1$	Integer
long	64 bit	$-2^{63}$	$+2^{63}-1$	Long
float	32 bit	IEEE 754		Float
double	64 bit	IEEE 754		Double
void	-	-	-	Void

39

## Esempio

```
int i = 5;
```

Variabile chiamata **i** inizializzata con il valore 5

```
Integer I;
```

Handle chiamato **I** che può puntare ad un oggetto di classe Integer

```
Integer I = new Integer(5);
```

Handle chiamato **I** a cui è stato assegnato un oggetto di classe Integer inizializzato con il valore 5.

40

# Costanti in JAVA

- Una costante è di un tipo specifico e può essere usata per inizializzare una variabile dello stesso tipo:

```
long    l = 34L;  
short   s = 0xCAFE;  
boolean b = true;  
float   f = 0.34e23F;  
double  d = 34.45D;
```

- Le costanti carattere sono rappresentate dal carattere fra apici semplici

```
char c = 'A';
```

- I caratteri UNICODE sono rappresentati col loro codice in esadecimale con il prefisso \u (es. \u 2122)
- Sono definite delle sequenze per rappresentare caratteri speciali (es. \n è l'avanzamento di riga, \b il backspace, \\ la barra \)

# Inizializzazione nelle classi


- I tipi primitivi vengono sempre inizializzati ad un valore di default (zero)
- Gli handle ad oggetti vengono inizializzati a NULL

```
class Segmento {  
  
    int x1, y1;        // Vengono inizializzati a zero  
    int x2, y2;        // idem  
  
    Integer Modulo;    // Viene inizializzato a NULL  
}
```

Anche in questo caso l'oggetto non esiste, e se provo ad accedervi si genera un run-time error.

# Inizializzazioni nelle classi

```
public class Conto {  
    private int valuta = 0;  
  
    public void versa(int somma) {valuta += somma;}  
  
    public void preleva(int somma)  
    {  
        int Spese;  
        /* int Spese = 0; Con questa modifica è meglio */  
        valuta -= (somma + Spese);  
    }  
  
    public int estrattoConto() { return valuta;}  
}
```



Se si tenta di accedere alle variabili locali dei metodi nelle classi senza una preventiva inizializzazione, viene generato un errore di compilazione

43

# Accesso alle proprietà

Si accede alle proprietà degli oggetti specificando: **<nome-classe>.nome-proprietà**

```
Segmento NuovoSegmento = new Segmento;
```

```
NuovoSegmento.x1 = 5;
```

```
NuovoSegmento.y1 = 3;
```

```
class DueSegmenti {  
    Segmento S1 = new Segmento;  
    Segmento S2 = new Segmento;  
}
```

```
DueSegmenti Spezzata = new DueSegmenti;
```

```
Spezzata.S1.x1 = 5;
```

```
Spezzata.S1.y1 = 3;
```

44

# Definizione dei metodi

I metodi indicano quali azioni è possibile compiere sugli oggetti, e sono definiti con:

- Nome
- Lista di argomenti
- Tipo del valore risultante
- Corpo

```
tipo-risultante nome_metodo (lista argomenti) {  
    /* corpo del metodo */  
}
```

- I metodi non esistono al di fuori delle classi di appartenenza
- Non si può più parlare di "parametri formali", perché chiamare un metodo significa mandare un messaggio all'oggetto in questione

45

# Esempio di metodo

Tipo del valore di ritorno

```
import java.lang.Math.*;  
  
class Segmento {  
    float x1, y1;  
    float x2, y2;  
  
    float Modulo(float Scala) {  
  
        float l1 = x2 - x1;  
        float l2 = y2 - y1;  
  
        return Math.sqrt(l1 * l1 + l2 * l2) * Scala;  
    }  
}
```

Istruzione di ritorno del metodo

Corpo del metodo

46

## Argomenti dei metodi

La lista di argomenti specifica le informazioni da trasmettere nel messaggio all'oggetto. Possono essere di due tipi:

- Tipi primitivi
- Handle di oggetti

Non restituisce nessun valore

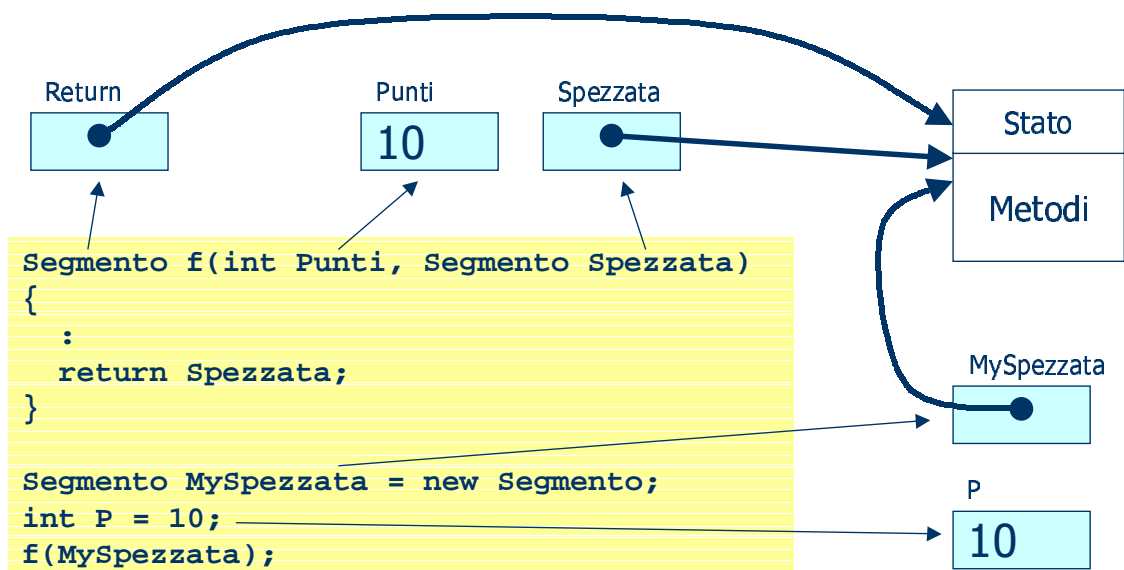
Questo è un handle

```
void Connetti(Segmento S) {  
    x2 = S.x2;  
    y2 = S.y2;  
}
```

47

## Passaggio e ritorno di oggetti

Il passaggio di variabili di tipo primitivo avviene **per copia**, mentre gli oggetti vengono passati **per riferimento**



48



# Primo operatore: Assegnazione

- L'assegnazione permette di definire il valore memorizzato in una variabile
- Il valore assegnato deve essere compatibile col tipo della variabile, altrimenti si genera un errore

```
int a, b;  
float v1, v2;  
char c = 'M';  
  
a = 4;  
b = a;  
v1 = 3.2  
v2 = a;  
b = v1;
```

**variabile ← espressione**

**b assume il valore di a (4)**

**non ammesso**

**ammesso: v2 assume il valore di a (4) convertito in float**

- Le conversioni ammesse in modo implicito fra tipi diversi sono:

byte → short → int → long → float → double



49

## Operatori comuni

### • Operatori unari

Permettono di effettuare operazioni di inversione di segno e autoincremento:

- Positivo +, Negativo -
- Pre-incremento ++a, Pre-decremento --a
- Post-incremento a++, Post-decremento a--

```
class Prova {  
  
    public static void main(String[] args) {  
  
        int A = 10;  
        System.out.println(A);           // Stampa 10  
        System.out.println(A++);         // Stampa 10  
        System.out.println(++A);         // Stampa 12  
    }  
}
```



50

# Operatori comuni

- Operatori aritmetici

- Moltiplicazione \*, Divisione /
- Modulo %
- Addizione +, Sottrazione -

- Operatori relazionali

Permettono di costruire espressioni logiche

- Uguale ==
- Diverso !=
- Maggiore >
- Maggiore e Uguale >=
- Minore <
- Minore e Uguale <=

```
(c != 'n')
```

È uguale a true se  
c **non** è il carattere 'n'

```
(b == 's')
```

È uguale a true se b è il carattere 's'



51

# Operatori comuni

- Operatori booleani

Permettono di costruire espressioni logiche:

- And &&
- Or ||
- Not !

```
( !( (c == 'n') || (c == 'N') ) )
```

c **non** è il carattere 'n'  
minuscolo o maiuscolo

```
( (b == 's') || (b == 'S') )
```

b è il carattere 's'  
minuscolo o maiuscolo

```
( !( (a < 0.5) && (a >= 0) ) )
```

a è compreso tra 0 (incluso) e 0.5 (escluso)

Non vengono mai valutate tutte i termini della condizione, ma solo fino a quando il risultato è univocamente determinato.

T      T      F      T

```
v1 && v2 && v3 && v4
```

la condizione v4 non viene  
valutata perché il risultato  
finale non può più cambiare



52

# Operatori comuni

- Operatori bit a bit

Permettono di costruire espressioni logiche:

- And &
- Or |
- Xor ^
- Not ~

```
class ProvaXor {  
  
    public static void main(String[] args) {  
  
        boolean A = 0x80; // (1000 0000)2  
        boolean B = 0xAB; // (1010 1011)2  
  
        System.out.println(A ^ B); // Stampa 0x2B  
                                    // (0010 1011)2  
    }  
}
```

53

# Operatori “meno comuni”

- Operatori di shift

Permettono di traslare a destra o a sinistra sequenze di bit:

- Shift a destra >>
- Shift a sinistra <<

```
1001001011101101 >> 2 → 0010010010111011
```

```
1001001011101101 << 3 → 0010010111011000
```

- Operatore ?

Permette di introdurre condizioni di assegnazione particolari:

```
espressione-boolean ? valore0 : valore1
```

```
double A = Math.Random();  
  
int B = (A > 0.5)? 10: 20;
```

assume il **valore0** se la condizione è true, altrimenti **valore1**

54

# Varianti di operatori

Per quasi tutti gli operatori è possibile sostituire una espressione del tipo:

```
variabile = variabile <operatore> valore
```

con una scrittura abbreviata:

```
variabile <operatore>= valore
```

esempio:

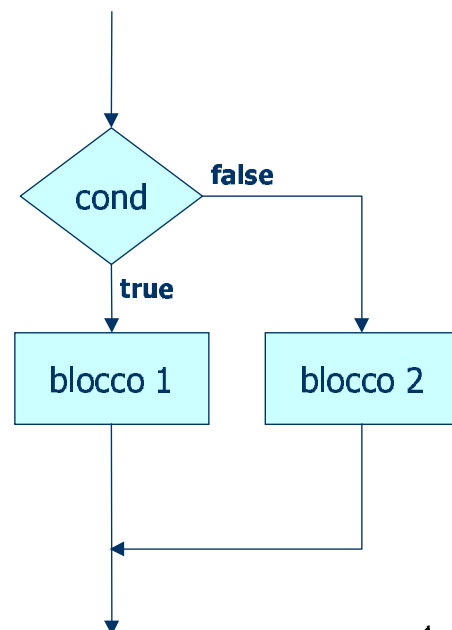
```
double A = 0.0D;  
  
// Invece di scrivere A = A + 2.0D  
// abbiamo scritto A += 2.0D  
System.out.println(A += 2.0D); // Stampa 2.0  
  
System.out.println(A /= 2.0D); // Stampa 1.0
```

55

# Controllo di flusso: If ... else ...

```
if (boolean-cond) {  
    blocco 1  
}  
else {  
    blocco 2  
}
```

```
if (vendite > media) {  
    bonus = 0.1;  
    guadagno = quota * bonus;  
}  
else {  
    guadagno = 0;  
}
```



56

# Controllo di flusso: switch

```
switch (variabile) {  
    case Valore1: blocco1  
        break;  
    case Valore2: {  
        blocco2;  
        break;  
    }  
    default: blocco default;  
}
```

```
char C = 'n';  
switch (variabile) {  
    case 'n': System.out.println("Sconto");  
        break;  
    case 's': System.out.println("Sconto no");  
        break;  
    default: System.out.println("Boh!");  
}
```

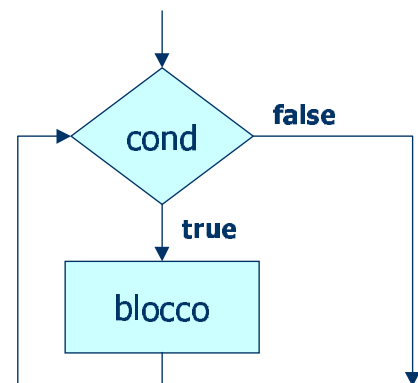
57

# Cicli indeterminati

- Cicli While

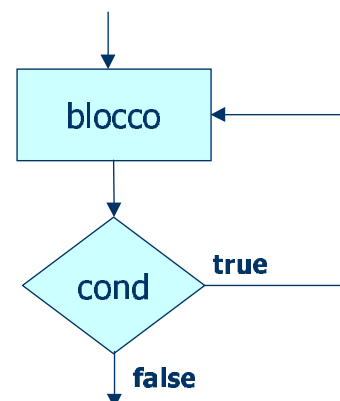
```
while (boolean-cond) {  
    blocco  
}
```

```
while (true) {  
    System.out.println("Infinito");  
}
```



- Cicli Do ... While

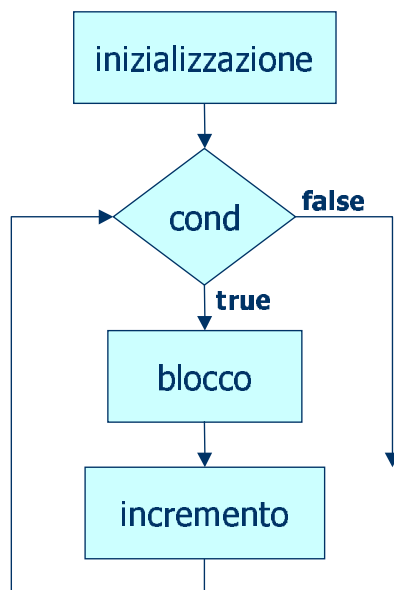
```
do {  
    blocco  
} while (boolean-cond)
```



58

## Cicli determinati

```
for (inizializzazione; boolean-cond; incremento) {  
    blocco  
}
```



```
for (int i = 0; i < 5; i++) {  
    System.out.println(i);  
}
```

```
for (;;) {  
    System.out.println("Infinito");  
}
```

- Operatore Comma

```
for (int i = 0, j = 5; i < 5; i++, j--)  
{  
    System.out.println(i);  
    System.out.println(j);  
}
```

## Break e Continue

- L'istruzione Break interrompe l'attuale istruzione di iterazione (while, do while, for).
- L'istruzione Continue interrompe la corrente iterazione e ritorna all'inizio del ciclo, iniziandone un'altra.

```
for (int i = 0, j = 10; i < 15; i++, j--) {  
    System.out.println(i);  
    if (i % 2 == 0)  
        continue;  
    if (j < 0)  
        break;  
    System.out.println(j);  
}
```



# Break e Continue

- Le istruzioni Break e Continue permettono di interrompere anche cicli diversi da quelli in esecuzione.

```
CycleI:
// Possono esserci anche commenti
// purché non ci siano altre istruzioni
for (int i = 0; i < 15; i++) {
    CycleJ:
    for (int j = 0; j < 15; j++) {
        System.out.println("i=" + i + "j=" + j);
        if (j > 4) continue CycleI;
        if (i > 10) break CycleJ;
    }
}
```

In questi casi le etichette devono essere indicate subito prima del ciclo relativo.

Output:

```
i=0 j=0
i=0 j=1
i=0 j=2
i=0 j=3
i=0 j=4
i=0 j=5
i=1 j=0
i=1 j=1
i=1 j=2
i=1 j=3
:
:
i=10 j=4
i=10 j=5
i=11 j=0
```

61

# La parola chiave STATIC

Le proprietà ed i metodi vengono normalmente utilizzati solo con oggetti costruiti con l'istruzione **new**.

Cosa possiamo fare se vogliamo:

- Una variabile comune a tutti gli oggetti di una classe: in modo tale che una modifica della variabile all'interno di un oggetto si rifletta su tutti gli altri
- Utilizzare un metodo senza allocare l'oggetto relativo

**Si dichiara l'elemento che vogliamo rendere comune o visibile come elemento di tipo STATIC**

Ovviamente un metodo dichiarato static non può utilizzare un qualsiasi elemento non-static



62

# La parola chiave STATIC

```
class Segmento {  
  
    static int NSegmenti = 0;  
    float x1, y1, x2, y2;  
  
    static void IncSegmenti() {  
        NSegmenti++;  
    }  
    public Segmento() {  
        IncSegmenti();  
        x1 = x2 = y1 = y2 = 0;  
    }  
}  
  
class MainSegmenti {  
    public static void main(String[] args) {  
        for (int i = 0; i < 10; i++) {  
            new Segmento();  
        }  
        System.out.println(Segmento.NSegmenti);  
    }  
}
```

63

## A proposito del 'main'!

```
class MainSegmenti {  
    public static void main(String[] args) {  
        :  
        :  
    }  
}
```

- il main DEVE essere **public**: perché in questo modo è possibile richiamarlo dall'esterno della classe
- il main DEVE essere **static**: perché la JVM deve poter chiamare il metodo senza allocare la classe
- il main DEVE essere **void**: perché questo metodo non restituisce nessun valore
- il main DEVE avere l'argomento **args**: perché la JVM deve poter passare i parametri inseriti nella riga di comando



64



## Il blocco STATIC

È possibile anche specificare un blocco di proprietà di tipo static:

```
class Segmento {  
  
    static {  
        int NSegmenti = 0;  
        int Nangoli = 0;  
    }  
  
    float x1, y1, x2, y2;  
  
    static void IncSegmenti() {  
        NSegmenti++;  
        Nangoli += 2;  
    }  
    :  
    :  
}
```

65

## Creare gli oggetti

Cosa succede quando creiamo un oggetto con **new**?  
l'ambiente chiama uno speciale metodo, denominato **costruttore**, che si occupa di inizializzare l'oggetto.

```
class Segmento {  
    :  
    :  
    public Segmento() {  
        IncSegmenti();  
        x1 = x2 = y1 = y2 = 0;  
    }  
}
```

Il costruttore è caratterizzato dal fatto che non ha un valore di ritorno e che il suo nome è uguale a quello della classe associata.

66

## Creare gli oggetti

Il costruttore è QUASI sempre pubblico, ma ci sono dei casi in cui è meglio renderlo privato.

Esempio: Quando è necessario costruire al massimo un certo numero di oggetti dalla stessa classe.

```
class Segmento {  
  
    static private int NSegmenti = 0;  
    float x1, y1, x2, y2;  
  
    private Segmento() {};  
    public static Segmento CreaSegmento() {  
        if (NSegmenti < 10) {  
            NSegmenti++;  
            return new Segmento();  
        }  
        else  
            return null;  
    }  
}
```

67

## Costruttori di default

Quando nessun costruttore è specificato, allora il compilatore ne crea uno automaticamente senza argomenti.

Se invece viene specificato anche solo un costruttore, allora il **costruttore di default** non viene costruito.



68

# Overloading dei metodi

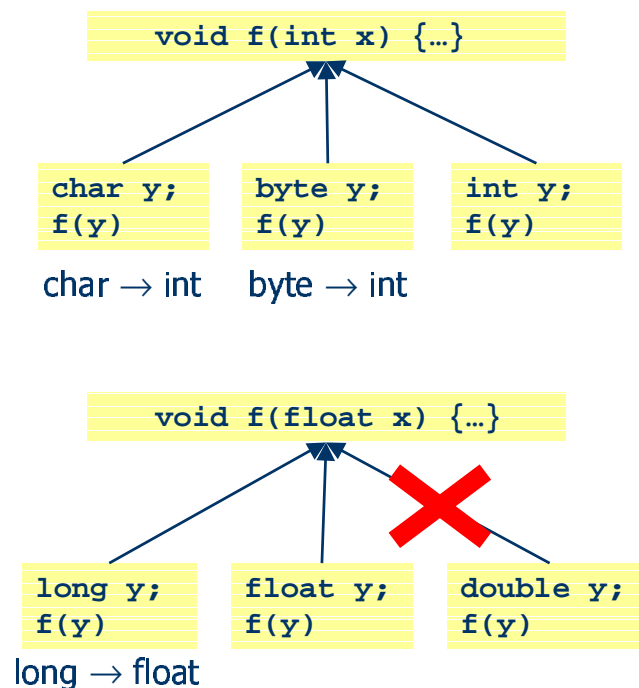
All'interno delle classi è possibile definire metodi con gli stessi nomi, ma con argomenti diversi in numero e/o tipo.

```
class Segmento {  
    :  
    :  
    public Segmento() {  
        IncSegmenti();  
        x1 = x2 = y1 = y2 = 10;  
    }  
    public Segmento(float Value) {  
        IncSegmenti();  
        x1 = x2 = y1 = y2 = Value;  
    }  
}
```

Questa prerogativa viene detta **overloading** dei metodi: è il compilatore che si preoccupa di scegliere il metodo con i parametri giusti.

69

# Overloading dei metodi



70

# Esempio di overloading

```
class MainSegmenti {  
    public static void main(String[] args) {  
  
        Segmento Primo = new Segmento(); // Senza parametri  
        for (int i = 0; i < 10; i++) {  
            new Segmento(i); // Con parametri  
        }  
        System.out.println(Segmento.NSegmenti);  
    }  
}
```

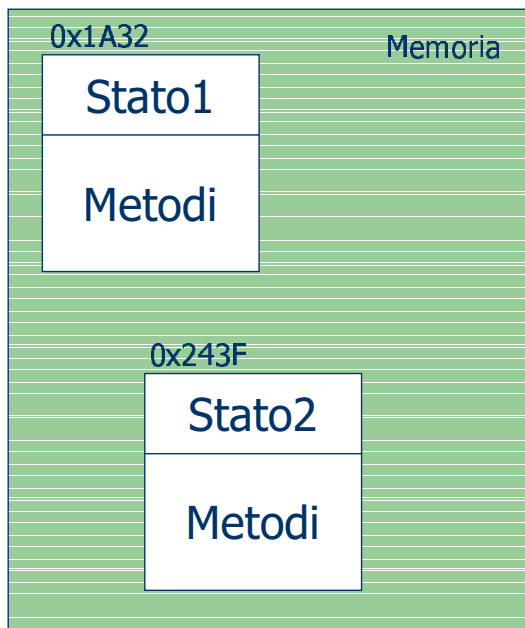
Non è possibile effettuare un overloading di metodi che hanno lo stesso numero e tipo di parametri ma restituiscono un valore diverso.

```
:  
:  
int f() { return 10; }  
float f() { return 20.0; }  
:
```

71

# Operatore 'this'

this è uno speciale operatore che restituisce l'indirizzo dell'oggetto nel quale è chiamato.



```
class Segmento {  
    :  
    :  
    public Segmento() {  
        x1 = x2 = y1 = y2 = 10;  
        System.out.println(this);  
    }  
}
```

Output:

MainSegmenti.Segmento@1A32

MainSegmenti.Segmento@243F

72

# Inizializzazioni

- Le proprietà delle classi vengono inizializzate ad un valore di default quando vengono allocate

```
class Segmento {  
    float x1, y1, x2, y2;  
}
```

queste proprietà vengono  
inizializzate a 0.0

- Le proprietà delle classi vengono inizializzate ad un valore di default PRIMA di attivare il metodo costruttore
- Le proprietà static vengono inizializzate al valore indicato solo una volta e dal primo oggetto che le utilizzerà

73

# Inizializzazioni

- Le variabili locali DEVONO essere inizializzate prima di poterle utilizzare

```
{  
    :  
    :  
    int I;  
    I++;  
}
```

viene generato un errore di  
compilazione

74

## Array in JAVA

Un **array** è un **handle** ad un vettore (o una matrice) di elementi che possono essere di tipo primitivo oppure oggetti:

```
int[] A1;  
int A2[];
```

in entrambi i casi viene generato un handle ad un vettore

```
int[] B = {1, 2, 3, 4, 5, 6};
```

in questo caso viene anche allocata la zona di memoria necessaria a contenere gli elementi indicati

Se vogliamo allocare memoria ed inizializzare così un handle:

```
int[] A1;  
int A2[];  
int[] B = {1, 2, 3, 4, 5, 6};  
  
A1 = new int[6];  
A2 = A1;
```

A1.length indica quanti elementi sono stati allocati

75

## Array in JAVA

È anche possibile creare un **array di oggetti**, cioè un vettore di handle tutti inizializzati a **null**

```
Segmento[] Spezzata;  
Spezzata = new Segmento[6];  
  
for (int i = 0; i < Spezzata.length; i++) {  
    Spezzata[i] = new Segmento();  
}
```

0:	NULL
1:	NULL
2:	NULL
3:	NULL
4:	NULL
5:	NULL

E anche possibile inizializzare direttamente un vettore di oggetti:

```
Segmento[] Spezzata = {  
    new Segmento();  
    new Segmento();  
    new Segmento();  
};
```

ma è poco utile.

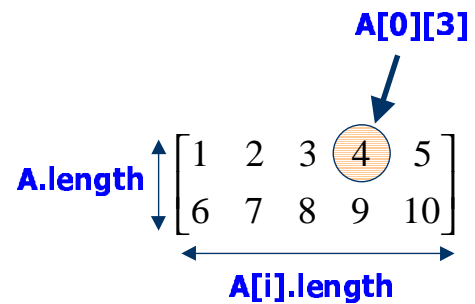
76

# Array multidimensionali

Il JAVA non pone limiti al numero di dimensioni degli array:

```
int[][] A = {  
    {1, 2, 3, 4, 5},  
    {6, 7, 8, 9, 10}  
};
```

```
int[][] B = new int[5][2];
```



```
int[][][] A = new [2][2][3];  
  
for (int x = 0; x < A.length; x++)  
    for (int y = 0; y < A[x].length; y++)  
        for (int z = 0; z < A[x][y].length; z++)  
            A[x][y][z] = 1;
```

77

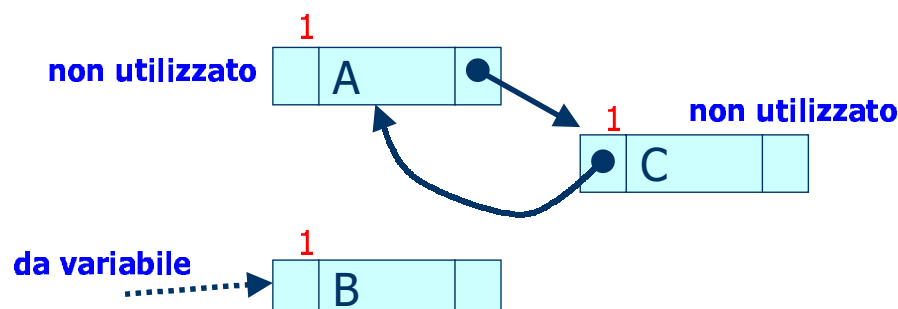
## Il Garbage Collector

È possibile attivare manualmente il Garbage Collector:

```
System.gc();
```

Come è possibile capire se un oggetto è ancora referenziato o anche **vivo** utilizzando due possibili tecniche:

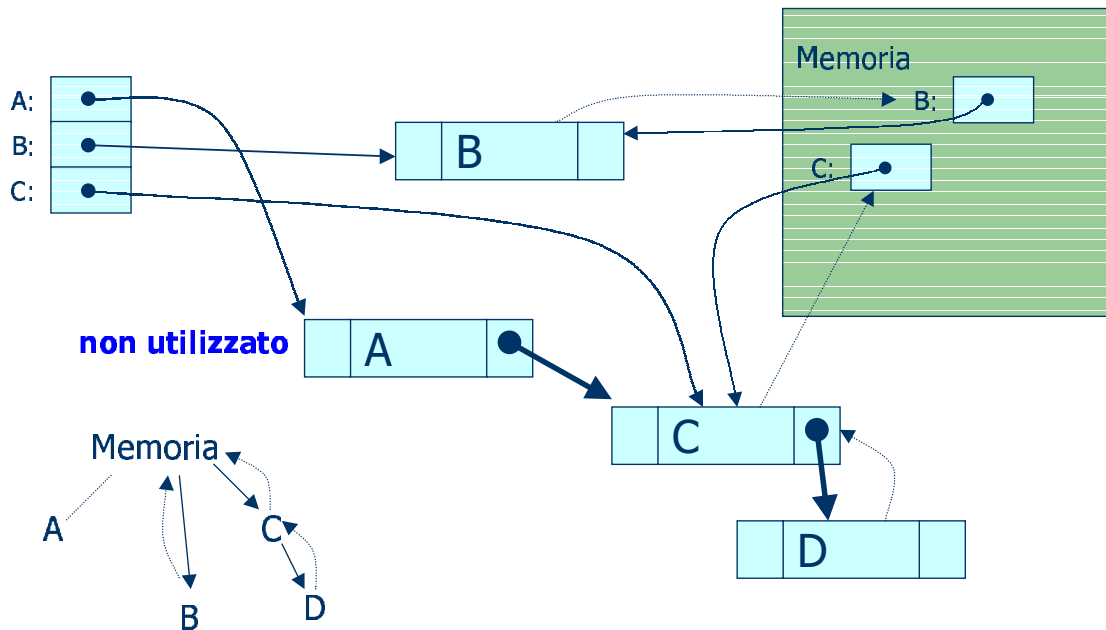
- La prima è quella di contare i riferimenti agli oggetti.  
Lo svantaggio è che molte volte gli oggetti si auto-referenziano:



78

## Il Garbage Collector

- Un'altra tecnica è quella di seguire a ritroso i riferimenti fino ad una variabile static oppure presente nello stack



79

## Strategie di Garbage Collection

Le strategie di garbage collecting sono due:

- **Stop-and-Copy**: Una volta trovati gli oggetti da rimuovere, vengono copiati in un altro heap gli oggetti ancora referenziati, e modificati tutti i riferimenti a loro collegati.
- **Mark-and-Sweep**: Una volta trovati gli oggetti da rimuovere, questi vengono eliminati senza copiare quelli ancora referenziati. In seguito l'heap può essere riorganizzato spostando solo alcuni oggetti ancora vivi, più piccoli degli spazi rilasciati.

La tendenza attuale è quella di utilizzare entrambe le tecniche (**Adaptive Garbage Collection**), ma in momenti diversi.

La tecnica Stop-and-Copy viene utilizzata solo quando l'heap è troppo frammentato, e quindi è necessaria una riorganizzazione.

80



# Ereditarietà

- Ogni classe definita in JAVA eredita dalla classe **Object** tutti i metodi
- Le classi discendenti può aggiungere ulteriori proprietà e metodi
- Le classi discendenti possono ridefinire (**overriding**) metodi e proprietà delle classi antenate

81

# Ereditarietà

```
class Homer {  
    protected int IQ = 0;  
  
    void doh(float f) {  
        System.out.println("Homer doh float: " + f);  
    }  
    void doh(double f) {  
        System.out.println("Homer doh double: " + f);  
    }  
    public String toString() { ← Overridden  
        return "My IQ is " + IQ + "!";  
    }  
}  
  
class Bart extends Homer {  
    protected int IQ = 10; ← Overridden  
    void doh(float f) { ← Overridden  
        System.out.println("Bart doh: " + f);  
    }  
}
```

82

## Ereditarietà

```
:  
Homer h = new Homer();  
h.doh(1.0f);  
  
Bart b = new Bart();  
b.doh(1L);  
b.doh(1.0f);  
b.doh(1.0D);  
System.out.println(b);
```

### Output:

```
Homer doh float: 1.0  
Bart doh: 1.0  
Bart doh: 1.0  
Homer doh double: 1.0  
My IQ is 0!
```

## Ereditarietà

Come faccio a chiamare e/o ad utilizzare metodi e proprietà della classe antenato?

```
class Homer {  
    protected int myIQ = 0;  
  
    public Homer(int IQ) { myIQ = IQ; }  
}  
  
class Bart extends Homer {  
    protected int IQ = 10;  
  
    public Bart(int IQ) {  
        super(IQ + 100);  
        super.doh(2.0D);  
    }  
}
```

# Costruzione e clausola super

Cosa succede in fase di inizializzazione?

- Vengono inizializzate le proprietà ereditate dalla classe antenato
- Viene inizializzato l'oggetto discendente

In pratica la classe antenato viene come inglobata nella classe discendente

Inoltre, se i costruttori hanno qualche parametro è necessario specificare quelli da passare al costruttore antenato utilizzando categoricamente come prima istruzione `super(...)`



85

# Overriding contro Overloading

Quando scelgo di sovraccaricare un metodo oppure di ridefinirlo?

È semplice: nel momento stesso che modifico la sua interfaccia (oltre che il suo corpo), sto effettuando un **overloading**.

Viceversa, lo stesso metodo con la medesima interfaccia ma con un corpo diverso, è un metodo **overridden**.



86

# RTTI ovvero capire le classi

L'RTTI (RunTime Identification) è il sistema che permette di capire a che classe appartiene un oggetto.

```
:
Homer h = new Homer();
Bart b = new Bart();
System.out.println(b instanceof Bart);
System.out.println(b instanceof Homer);
System.out.println(h instanceof Bart);
System.out.println(h instanceof Homer);
```

Arrows pointing to the right of the code block indicate the result of each `instanceof` check:

- Arrow from `b instanceof Bart` points to **true**
- Arrow from `b instanceof Homer` points to **true**
- Arrow from `h instanceof Bart` points to **false**
- Arrow from `h instanceof Homer` points to **true**

87

## Upcasting

È la proprietà più importante dell'ereditarietà

Il concetto è molto semplice: tutti i metodi che hanno oggetti come parametri accettano anche tutti gli oggetti discendenti.

```
class Homer {
:
    public void Speak() {
        System.out.println("DOH!");
    }
}

class Bart extends Homer {
:
    public void Speak() {
        System.out.println("Eat my socks");
    }
}
```



88

# Upcasting

```
void LetHimSpeak(Homer Him) {  
    Him.Speak();  
}  
  
:  
Homer h = new Homer();  
Bart b = new Bart();  
  
LetHimSpeak(h);  
LetHimSpeak(b);
```

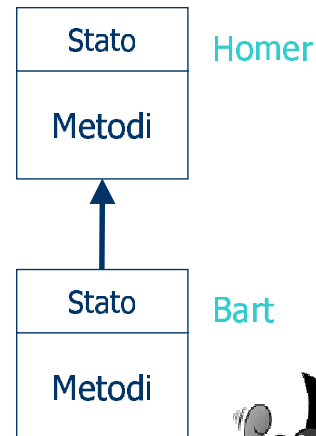
Si vede come  
**LetHimSpeak** accetti  
anche la classe **Bart**

Il termine **upcasting** deriva dalla  
direzione con cui ci si muove  
sugli alberi di ereditarietà

Output:

DOH!

Eat my socks!



89

# Downcasting

Anche qui il concetto è semplice: **se siamo sicuri** che  
un handle, pur essendo di una classe, si riferisca ad  
un oggetto discendente, allora è possibile effettuare  
la conversione.

```
class Homer {  
    :  
    public void Speak() {  
        System.out.println("DOH!");  
    }  
}  
  
class Bart extends Homer {  
    :  
    public void Speak() {  
        System.out.println("Eat my socks");  
    }  
    public void KickLisa() {  
        System.out.println("LISA!!");  
    }  
}
```

90

# Downcasting

```
void LetHimSpeak(Homer Him) {  
    if (Him instanceof Bart) {  
        Him.Speak();  
        ((Bart)Him).KickLisa();  
    }  
    else  
        Him.Speak();  
}  
  
:  
Homer h = new Homer();  
Bart b = new Bart();  
  
LetHimSpeak(h);  
LetHimSpeak(b);
```

Trasformo una  
classe in un suo  
discendente in modo  
da poter utilizzare i  
suoi metodi

## Output:

DOH!

Eat my socks!

LISA!!

91

# Aggregazione ed Ereditarietà

L'**aggregazione** permette l'allocazione di uno o più oggetti all'interno di una classe, in modo da utilizzarne i meccanismi e le caratteristiche.

Quando utilizzare l'ereditarietà o l'aggregazione?

- **L'interfaccia**: Se la classe contenitore necessita delle caratteristiche degli oggetti che definisce al suo interno, ma non vuole mostrare la loro interfaccia all'esterno, allora si parla di aggregazione
- **Upcasting**: Se il problema che affronta la classe contenitore necessita di effettuare un upcasting tra la nuova classe e la sua antenata, allora è necessaria una progettazione con eredità



92

# Polimorfismo

Il polimorfismo è un'altra caratteristica essenziale della programmazione ad oggetti. In pratica i singoli oggetti rispondono ad un metodo secondo la propria implementazione.

```
void LetHimSpeak(Homer Him) {  
    Him.Speak();  
}
```

```
:  
Homer h = new Homer();  
Bart b = new Bart();  
  
LetHimSpeak(h);  
LetHimSpeak(b);
```

Abbiamo già visto che questa chiamata si comporta in maniera diversa a seconda dell'oggetto

Chiama Speak della classe **Bart** se l'oggetto è un'istanza di quella classe

Chiama Speak della classe **Homer** se l'oggetto è un'istanza di quella classe

93

# Polimorfismo

Quale è il meccanismo alla base del polimorfismo?

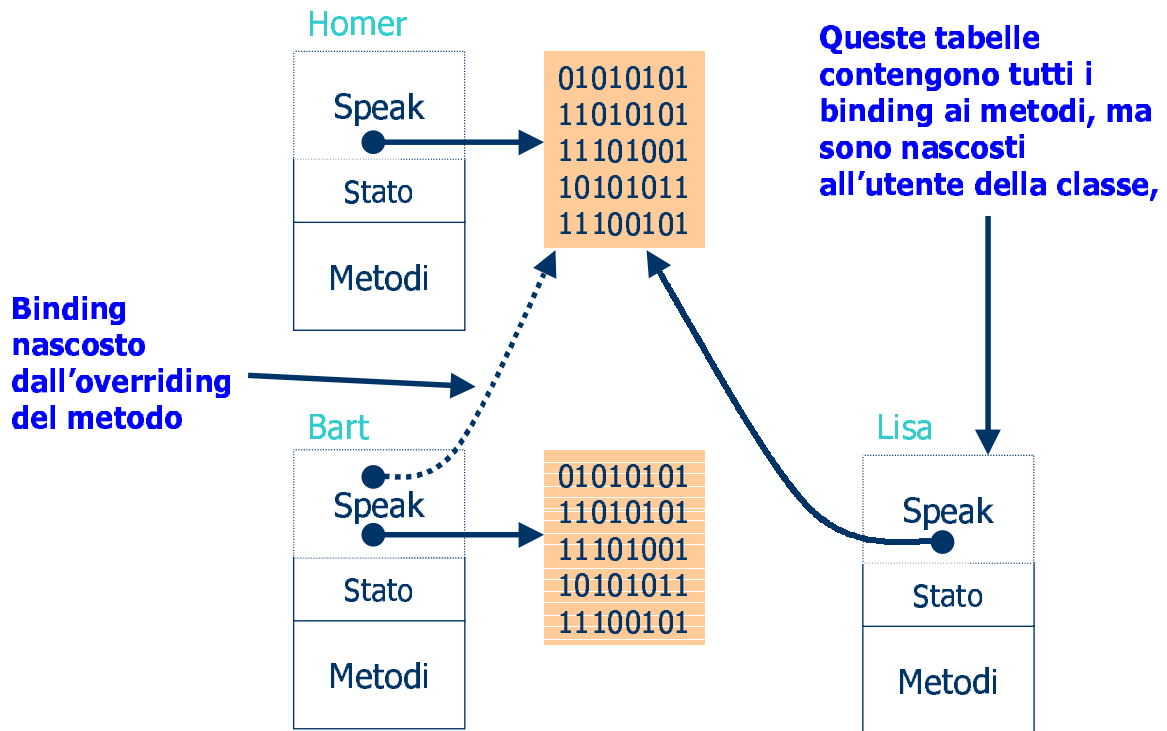
Per effettuare il binding tra chiamate di procedure e le rispettive implementazioni, i compilatori possono utilizzare due tecniche:

- **Static Binding**: Quando il compilatore conosce già l'indirizzo, e quindi è possibile effettuare il binding a tempo di compilazione.
- **Dynamic Binding**: Quando il compilatore non conosce ancora l'indirizzo, e quindi il binding viene rimandato a tempo di esecuzione.

I compilatori dei linguaggi ad oggetti utilizzano quest'ultima tecnica per risolvere il problema.

94

# Polimorfismo



95

## Il modificatore FINAL

Il modificatore **FINAL** ha un effetto diverso a seconda del contesto di utilizzo:

- **FINAL** nelle proprietà e nelle variabili locali:

Le variabili locali e le proprietà, dichiarate con **FINAL**, possono essere modificate solo alla loro definizione

```
void LetHimSpeak(Homer Him) {  
    final int C = 0;  
    :  
    C = 10;  
}
```

Questa nuova assegnazione genera un errore di compilazione

96



## Il modificatore FINAL

- FINAL nei parametri dei metodi:

I parametri dei metodi dichiarati con FINAL, non possono essere modificati

```
void LetHimSpeak(final Homer Him) {  
    :  
    Him = new Homer;  
}
```

Questa nuova assegnazione genera un errore di compilazione

- FINAL negli handle:

Gli handle dichiarati con FINAL, non possono essere modificati

```
void LetHimSpeak(Homer Him) {  
    :  
    final Homer Him2 = new Homer;  
    Him2 = new Homer;  
    Him2.IQ = 3000;  
}
```

Questa nuova assegnazione genera un errore di compilazione

Questa no!

97

## Il modificatore FINAL

- FINAL nei metodi:

I metodi dichiarati con FINAL, non possono essere ridefiniti

```
class Homer {  
    :  
    final public void Speak() {  
        System.out.println("DOH!");  
    }  
}  
  
class Bart extends Homer {  
    :  
    public void Speak() {  
        System.out.println("Eat my socks");  
    }  
}
```

Questa nuova ridefinizione genera un errore di compilazione

98

## Il modificatore FINAL

Inoltre, il corpo dei metodi dichiarati con FINAL vengono copiati ad ogni loro occorrenza, eliminando così l'overhead del passaggio di parametri, della chiamata, ecc.

Questo implica una maggiore efficienza in termini di tempo, del codice risultante, a scapito della dimensione del codice prodotto.

## Il modificatore FINAL

- FINAL nelle classi:

Le classi dichiarate con FINAL, non possono diventare antenate di altre classi

```
final class Homer {  
    :  
    public void Speak() {  
        System.out.println("DOH!");  
    }  
}  
  
class Bart extends Homer {  
    :  
    public void Speak() {  
        System.out.println("Eat my socks");  
    }  
}
```

**Questa nuova ridefinizione genera un errore di compilazione**

# Le interfacce in JAVA

Il JAVA permette di definire interfacce comuni ad intere classi ed ai rispettivi discendenti.  
I meccanismi per la definizione di interfacce in JAVA sono principalmente due, e sono caratterizzate da elementi:

- **Non sono istanziabili**: Non è possibile creare un oggetto da una interfaccia
- **Molteplicità delle interfacce**: È possibile avere classi che implementano molteplici interfacce contemporaneamente

101

# Il modificatore ABSTRACT

Il JAVA permette di definire classi i cui metodi non hanno implementazione: queste classi si chiamano **abstract classes**.

```
abstract class Simpson {  
    protected int myIQ = 0;  
  
    abstract void Speak() {}  
}  
  
class Homer extends Simpson {  
    :  
}  
  
class Bart extends Homer {  
    :  
}
```

I metodi astratti si definiscono con il modificatore **abstract**

Le classi derivate **DEVONO** implementare i metodi astratti, altrimenti sono classi astratte anch'esse e quindi devono essere definite come **abstract**

102

## Il modificatore ABSTRACT

I vantaggi nella definizione di classi abstract sono:

- Definire una classe abstract significa definire una interfaccia comune a tutte le classi derivate.
- Una classe abstract non può essere istanziata. Se ciò viene fatto il compilatore produce un errore. Questo permette al progettista della classe di costringere l'utente a definire il comportamento dei metodi astratti.

103

## Il descrittore INTERFACE

Come le **abstract classes** anche le **interface** rappresentano lo scheletro di una classe senza la rispettiva implementazione.

Le caratteristiche che le differenziano sono:

- Nelle interface è possibile definire anche proprietà, mentre nelle classi astratte questo non è possibile. Queste variabili divengono automaticamente `STATIC` and `FINAL`
- Nella interface il concetto di implementazione è assolutamente inesistente. In pratica si stabilisce un protocollo di accesso, che poi verrà utilizzato da tutte le classi che utilizzano la medesima interfaccia

104

## Esempio di interfaccia

```
interface Simpson {
    int IQ = 0;
    void Speak();
}

class Homer implements Simpson {

    public void Speak() {
        System.out.println("Homer: DOH! my IQ is " + IQ);
    }
}

class Bart implements Simpson {

    public void Speak() {
        System.out.println("Bart: Eat my socks! my IQ is " + IQ);
    }
    public void KickLisa() {
        System.out.println("Bart: I won't kick LISA anymore!");
    }
}
```

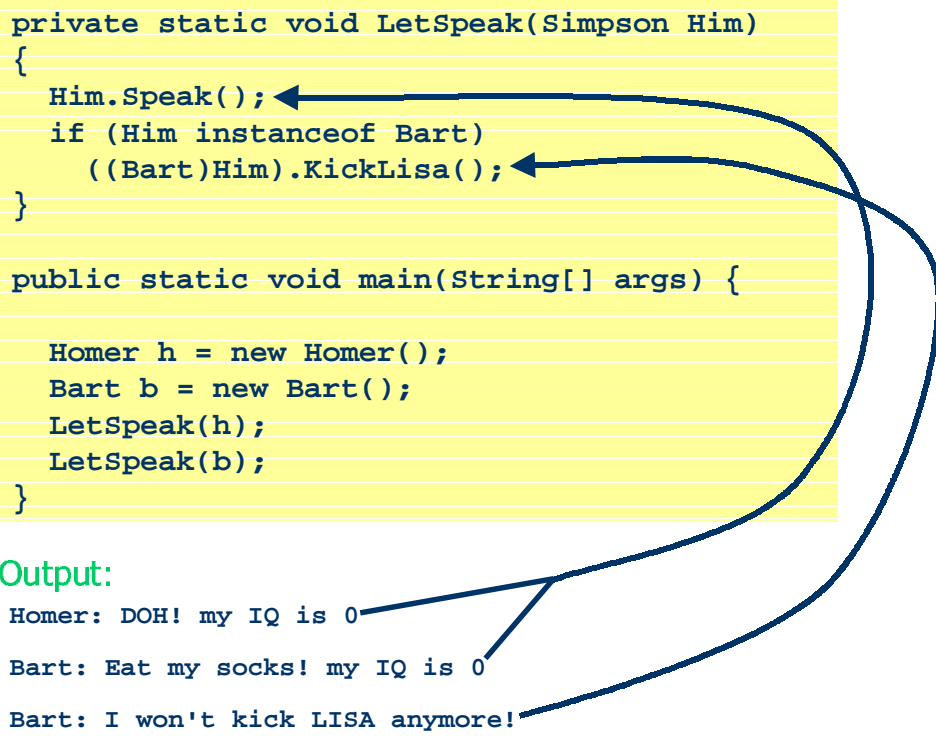
105

## Esempio di interfaccia

```
private static void LetSpeak(Simpson Him)
{
    Him.Speak();
    if (Him instanceof Bart)
        ((Bart)Him).KickLisa();
}

public static void main(String[] args) {

    Homer h = new Homer();
    Bart b = new Bart();
    LetSpeak(h);
    LetSpeak(b);
}
```



### Output:

```
Homer: DOH! my IQ is 0
Bart: Eat my socks! my IQ is 0
Bart: I won't kick LISA anymore!
```

106

## Il descrittore INTERFACE

Il vantaggio primario nella definizione di interfaces è rappresentato dal concetto ereditarietà multipla:

```
interface Simpson {
    int IQ = 0;
    void Speak();
}

interface SpringfieldPeople {
    void BecomeSane();
}

class Homer implements Simpson, SpringfieldPeople {
    public void Speak() {
        System.out.println("Homer: DOH! my IQ is " + IQ);
    }
    public void BecomeSane() {
        System.out.println("SP: Never become sane!");
    }
}
```

107

## Il descrittore INTERFACE

```
:
private static void LetSpeak(Springfield Him)
{
    Him.BecomeSane();
}

public static void main(String[] args) {

    Homer h = new Homer();
    Bart b = new Bart();
    LetSpeak((Simpson)h);
    LetSpeak((SpringfieldPeople)h);
    LetSpeak(b);
    LetSpeak((SpringfieldPeople)b);
}
```

### Output:

```
Homer: DOH! my IQ is 0
Sp: Never become sane!
Bart: Eat my socks! my IQ is 0
Bart: I won't kick LISA anymore!
```

**Questa riga produce errore, perché si sta forzando un oggetto ad essere quello che non è**

108

## Le Inner class

Le Inner class sono classi definite all'interno di altre classi.

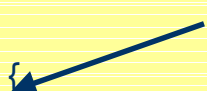
```
class Homer {  
  
    class Son {  
        public void Speak() {  
            System.out.println("Son: Eat my socks!");  
        }  
    }  
  
    Son Bart = new Son();  
  
    public void Speak() {  
        System.out.println("Homer: DOH!");  
        Bart.Speak();  
    }  
}
```

109

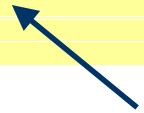
## Le Inner class

```
class Homer {  
:  
    public Son CreateSon() {  
        return new Son();  
    }  
    public static void main(String[] args) {  
        Homer Homer1 = new Homer();  
        Homer.Son Bart = Homer1.CreateSon();  
    }  
}
```

Questo è l'unico modo  
di istanziare oggetti di  
inner class



È possibile accedere alle inner  
class nei metodi static e nelle  
altre classi indicando la classe  
di appartenenza.



110

## Implementazione nascosta

Le Inner class che implementano interfacce pubbliche possono essere visibili all'esterno pur avendo l'implementazione nascosta all'interno della classe contenitore.

```
interface Simpson {  
    public void Speak();  
}  
  
class Homer {  
  
    class Son implements Simpson {  
        public void Speak() {  
            System.out.println("Son: Eat my socks!");  
        }  
    }  
    :  
    :  
}
```

111

## Inner class nei metodi

Le Inner class possono essere definite anche nei metodi, nascondendole del tutto al resto del programma.

È possibile utilizzare queste classi solo se posseggono una interfaccia pubblica o una classe base.

```
class Homer {  
  
    public void Speak() {  
        class Son {  
  
            System.out.println("Son: Eat my socks!");  
        }  
    }  
    :  
    :  
}
```

112



## Inner class anonime

```
interface Point {
    public void setXY(int x, int y)
    public int getX();
    public int getY();
}

public class Prova {
    static public Point getPoint(final int x, final int y) {
        return new Point() {
            private int X, Y;
            {X = x; Y = y; }
            public void setXY(int x, int y)
            {X = x; Y = y;}
            public int getX() { return X; }
            public int getY() { return Y; }
        }
    }
    public static void main(String[] args) {
        Point p = getPoint(0, 0);
        System.out.println("P("+p.getX()+", "+p.getY()+")");
    }
}
```

113

## Le eccezioni

Le eccezioni vengono definite come situazioni di errore non risolvibili nel contesto attuale, ma che possono esserlo in un contesto superiore.

I problemi inerenti alla gestione delle eccezioni sono fondamentalmente due:

- L'inserimento di codice di controllo ad ogni operazione soggetta ad errori critici
- L'impossibilità teorica, pratica e di stile ad effettuare **salti globali** nei linguaggio strutturati e ad oggetti

114

## Le eccezioni in JAVA

Il JAVA gestisce le eccezioni in maniera molto simile al C++, che a sua volta ha preso ispirazione dal linguaggio ADA:

In JAVA una eccezione viene inoltrata, creando con **new** un **oggetto eccezione**, al contesto superiore, che assume il controllo diretto della situazione

Esistono due zone distinte di codice: uno per la gestione della sezione protetta, l'altra per la gestione delle eccezioni.

In questo modo è possibile concentrarsi sul problema, e solo in seguito effettuare un'analisi dei possibili errori.

115

## Le eccezioni in JAVA

Il JAVA permette di utilizzare due tipi di eccezioni:

- **Eccezioni standard**: questo tipo ha due costruttori. Il primo tipo è il costruttore standard, il secondo accetta una stringa come argomento, in modo da fornire informazioni utili
- **Eccezioni definite dall'utente**: definite anche classi **throwable**, perché ereditano dalla classe `Exception` o da altre classi standard. Questo tipo di eccezioni può essere specifico per un errore e contenere molte più informazioni

```
:  
:  
if (p == null)  
    throw new NullPointerException();  
:
```

116

## Cosa è disponibile?

I progettisti JAVA hanno creato una serie di eccezioni derivate dalla classe progenitore throwable. I metodi disponibili ai discendenti sono:

- `String getMessage()`: restituisce il messaggio dell'eccezione
- `String getLocalizedMessage()`: permette di ottenere messaggi più dettagliati
- `String toString()`
- `void printStackTrace()`: permette di ottenere informazioni circa le chiamate che hanno condotto al punto di eccezione
- `void printStackTrace(PrintStream)`
- `void printStackTrace(PrintWriter)`

117

## Throwable Exceptions

<code>AcclNotFoundException,</code>	<code>MidiUnavailableException,</code>
<code>ActivationException,</code>	<code>MimeTypeParseException,</code>
<code>AlreadyBoundException,</code>	<code>NamingException,</code>
<code>ApplicationException,</code>	<code>NoninvertibleTransformException,</code>
<code>AWTException,</code>	<code>NoSuchFieldException,</code>
<code>BadLocationException,</code>	<code>NoSuchMethodException,</code>
<code>ClassNotFoundException,</code>	<code>NotBoundException,</code>
<code>CloneNotSupportedException,</code>	<code>NotOwnerException,</code>
<code>DataFormatException,</code>	<code>ParseException,</code>
<code>ExpandVetoException,</code>	<code>PrinterException,</code>
<code>FontFormatException,</code>	<code>PrivilegedActionException,</code>
<code>GeneralSecurityException,</code>	<code>PropertyVetoException,</code>
<code>IllegalAccessException,</code>	<code>RemarshalException,</code>
<code>InstantiationException,</code>	<code>RuntimeException,</code>
<code>InterruptedException,</code>	<code>ServerNotActiveException,</code>
<code>IntrospectionException,</code>	<code>SQLException,</code>
<code>InvalidMidiDataException,</code>	<code>TooManyListenersException,</code>
<code>InvocationTargetException,</code>	<code>UnsupportedAudioFileException,</code>
<code>IOException,</code>	<code>UnsupportedFlavorException,</code>
<code>LastOwnerException,</code>	<code>UnsupportedLookAndFeelException,</code>
<code>LineUnavailableException,</code>	<code>UserExceptionpublic</code>

118

# Throwable Exceptions

Esiste un numero incredibile di eccezioni ammesse. Queste sono quelle definite a partire da IOException:

```
ChangedCharSetException,  
CharConversionException,  
EOFException,  
FileNotFoundException,  
InterruptedIOException,  
MalformedURLException,  
ObjectStreamException,  
ProtocolException,  
RemoteException,  
SocketException,  
SyncFailedException,  
UnknownHostException,  
UnknownServiceException,  
UnsupportedEncodingException,  
UTFDataFormatException,  
ZipException
```

119

## Le eccezioni in JAVA

L'istruzione **throw** arresta l'esecuzione del contesto in cui l'eccezione si è verificata, passando l'oggetto appena creato ad un contesto in grado di gestirla.

```
:  
:  
if (p == null)  
    throw new NullPointerException("p = null");  
:
```

120

## Try e Catch

Se non è necessaria (o non è richiesta) la gestione di una eccezione fuori dal contesto attuale, si può tentare di risolverla all'interno di quello attivo.

```
try {  
    :  
    :  
}  
catch (ExceptionType1 e1) {  
}  
catch (ExceptionType2 e2) {  
}  
catch (ExceptionType3 e3) {  
}
```

L'argomento di una sezione catch è l'handle all'oggetto eccezione

L'ordine di scrittura delle sezioni catch coincide con l'ordine di verifica

In qualsiasi punto dello scope indicato da try, il sopraggiungere di un errore che provoca una eccezione viene gestito richiamando l'**exception handler** relativo, che può gestire la situazione.

121

## Esempio Try e Catch

```
public class ExceptionDemo {  
    class MyException extends Exception {  
        String MyMsg;  
  
        public MyException() {}  
        public MyException(String s) { super(s); }  
        public String ToString {  
            return super.toString() + " " + MyMsg; }  
    }  
    final public Print(String s) {  
        System.out.println(s);  
    }  
    void f() throws MyException {  
        Print("Inizio f()");  
        MyException E1 = new MyException("Da f()");  
        E1.MyMsg = "Big Trouble";  
        throw E1;  
        Print("Fine f()");  
    }  
}
```

122

# Esempio di Try e Catch

```
public static void main(String[] args) {  
    ExceptionDemo ED = new ExceptionDemo();  
  
    Print("Inizio main");  
    try {  
        ED.f();  
        Print("Questa riga viene eseguita?");  
    }  
    catch (MyException e) {  
        ED.Print(e.toString());  
    }  
    Print("Fine main");  
}
```

Genera l'errore

Viene gestita solo  
se l'eccezione viene  
catturata dal catch e  
tutte le sue discendenti

## Output:

```
Inizio main  
Inizio f()  
ExceptionDemo$MyException: Da f() Big Trouble  
Fine main
```

123

# Eccezioni nei metodi

Il JAVA costringe il progettista di una classe a comunicare le possibili eccezioni che i metodi relativi producono.

```
void f() throws Exception1, Exception2, Exception3 {  
    :  
    :  
}
```

Quando il compilatore si accorge che il metodo può produrre delle eccezioni, ma che queste non vengono dichiarate nella definizione del metodo, allora viene prodotto un errore di compilazione.

È anche possibile rilanciare una gestione gestita parzialmente.

```
catch (Exception e) {  
    // Gestione locale  
    throw e;  
}
```

124

## La sezione finally

La sezione **finally** permette di indicare una parte di codice che verrà eseguita sempre, qualunque sia l'esito della sezione try.

```
try {  
    :  
    :  
}  
catch (...) {  
    :  
}  
finally {  
    // Codice di clean-up  
}
```

La sezione finally viene eseguita anche se nessuna eccezione è stata gestita dalle dichiarazioni catch.

Il codice nelle sezioni finally può riguardare la chiusura di file, dei socket di comunicazione, ecc.

125

## La classe Object

La classe Object rappresenta l'antenato di tutte le classi che possono essere definite in JAVA.  
Il meccanismo di ereditarietà si attua anche quando il legame non viene dichiaratamente esplicitato.

### **clone()**

Creates and returns a copy of this object

### **boolean equals(Object obj)**

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

### **protected void finalize()**

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object

### **Class getClass()**

Returns the runtime class of an object

126

## La classe Object

**void notify()**

Wakes up a single thread that is waiting on this object's monitor

**void notifyAll()**

Wakes up all threads that are waiting on this object's monitor

**String toString()**

Returns a string representation of the object

**void wait()**

Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object

## La classe Object

**void wait(long timeout, int nanos)**

Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed

**void wait(long timeout)**

Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed



## Clonazione di oggetti

Oggetti che definiscono public il metodo clone() vengono detti **oggetti clonabili**.

Quando un oggetto è clonabile, non significa che lo sia alla maniera del senso comune.

### Esempio: Vector

Vector è una classe clonabile, ma se si tenta di clonarla senza riscrivere il metodo, quello che otteniamo è una copia i cui handle puntano agli stessi oggetti.

129

## Clonazione di oggetti

Quindi i requisiti per la clonazione sono:

- **Interfaccia cloneable**: questa interfaccia non aggiunge nessuna proprietà o metodo, ma permette di verificare se un oggetto è clonabile:

```
if (o instanceof Cloneable) {  
    :  
}
```

- **Chiamata al metodo clone di Object**: questa chiamata effettua una copia esatta dell'oggetto, ma non dei suoi oggetti aggregati. Se l'oggetto non incorpora l'interfaccia cloneable, allora viene generata una eccezione **CloneNotSupportedException**.

130

# Clonazione di oggetti

- **Clonazione degli oggetti aggregati:** questa fase richiede una chiamata del metodo clone() per ogni oggetto appartenente all'oggetto in questione.
- **Upcast del risultato:** il risultato del metodo clone() di un oggetto è sempre un handle ad un oggetto Object, quindi è necessario effettuare il cast alla classe definitiva.

## Esempio

```
class Bart implements Cloneable {  
  
    private int BartNumber = 0;  
  
    public Bart(int Number) {  
        BartNumber = Number;  
    }  
  
    public void Speak() {  
        System.out.println("Bart " + BartNumber);  
        System.out.println(": Eat my socks!");  
    }  
  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

## Esempio

```
class Homer implements Cloneable {

    static private int Homers = 0;
    private int MyNumber = 0;
    private Bart MySon;

    public Homer() {
        MyNumber = ++Homers;
        MySon = new Bart(MyNumber);
    }
    public void Speak() {
        System.out.println("Homer " + MyNumber + ": DOH!");
        MySon.Speak();
    }
    public Object clone() throws CloneNotSupportedException {
        Homer c = (Homer)super.clone();
        c.MyNumber = ++Homers;
        c.MySon = new Bart(c.MyNumber);

        return c;
    }
}
```

133

## Esempio

```
public class Prova4Class {

    public static void main(String[] args)
        throws CloneNotSupportedException
    {

        Homer h = new Homer();
        h.Speak();
        Homer h2 = (Homer)h.clone();
        h2.Speak();
    }
}
```

### Output:

```
Homer 1: DOH!
Bart 1: Eat my socks!
Homer 2: DOH!
Bart 2: Eat my socks!
```

134

## Clone(): ultime considerazioni

- **clone() dell'antenato:** Quando si definisce clone() per una classe, tutte le classi discendenti diventano clonabili. In questo modo però, se non viene definito il metodo clone() specifico per i discendenti, la clonazione potrebbe essere parziale o non corretta.
- **Il controllo delle eccezioni:** È opportuno gestire le eccezioni all'interno del metodo clone():

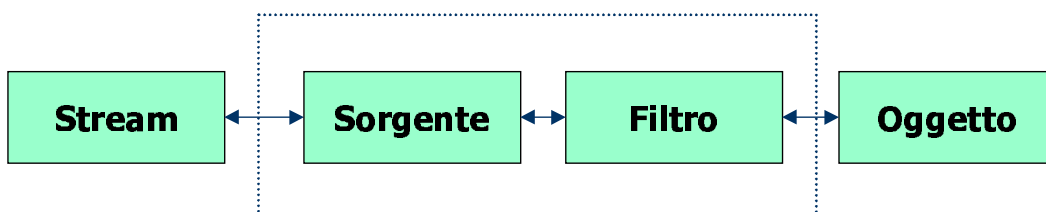
```
public Object clone() {  
    Object o = null;  
    try {  
        o = super.clone();  
    }  
    catch (CloneNotSupportedException E) {  
        E.printStackTrace();  
    }  
    return o;  
}
```

135

## JAVA 1.0 - I/O System

Il JAVA considera tutte i flussi da e verso l'esterno, come **stream di byte**. Questi possono essere di ingresso o di uscita:

- **InputStream:** Flusso di byte in ingresso. Con questa classe è possibile sia leggere il singolo byte, che un certo numero indicato.
- **OutputStream:** Flusso di byte in uscita. Anche con questa classe è possibile sia scrivere il singolo byte, che un intero array.



136

## La classe sorgente

Il JAVA divide le possibili sorgenti dei flussi in alcune classi:

- **ByteArrayInputStream**: Può essere letta come array di byte
- **StringBufferInputStream**: È possibile leggere la stringa corrispondente ai byte ricevuti
- **FileInputStream**: Può essere letta come sequenza di byte da un file
- **PipedInputStream**: Viene utilizzato per leggere sequenze di byte che provengono da un altro thread
- **SequenceInputStream**: Permette di leggere due stream in sequenza
- **Altre sorgenti**: Come per esempio i socket, ecc.

137

## Il classe filtro

Il JAVA mette a disposizione una serie di classi che aggiungono funzionalità alla classe della sorgente:

- **DataInputStream**: Viene utilizzata per leggere dati primitivi come int, float, double, ecc.
- **BufferedInputStream**: Previene la continua lettura dello stream fisico, implementando un buffer tampone
- **LineNumberInputStream**: Tiene traccia del numero di linee che è stato letto dallo stream
- **PushbackInputStream**: Mantiene un buffer di un byte, in modo tale da permettere il "push back" di byte letti

138

# Esempio

```
import java.io.*;

public static void main(String[] args) {
    try {
        DataInputStream file = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("p.txt")));

        try {
            String s;
            while ((s = file.readLine()) != null) {
                System.out.println("# " + s);
            }
        }
        finally {
            file.close();
            System.out.println("Errore di lettura");
        }
    }
    catch (IOException e) {
        System.out.println("Errore di apertura");
    }
}
```

139

## I/O su file

Le classi per la gestione dei file sono principalmente due:

- La classe File

Questa classe rappresenta un concetto molto ampio di file-system, in quanto questa classe può non indicare un solo singolo file, ma anche una directory intera, oppure un gruppo di file indicati con una stringa filtro. Con questa classe si possono ottenere tutte le informazioni utili all'accesso dei file e delle directory.

Con una stringa filtro si identifica un gruppo di file

list() restituisce un array di stringhe contenente i nomi dei file

```
:
File path = new File(".");
String[] list = path.list();
:
```

140

## I/O su file

```
:  
File f1 = new File("prova.txt");  
File f2 = new File("prova.old");  
f1.renameTo(f2);  
:
```

Permette di rinominare un file

```
:  
File f1 = new File("c:/documenti/prova");  
f1.mkdir();  
f1.mkdirs();  
:
```

Permette di creare interi alberi di directory

141

## I/O su file

- **RandomAccessFile**

Questa classe rappresenta l'insieme di record (delle stesse dimensioni) all'interno di un file. Con essa è possibile posizionarsi su un preciso record, utilizzando il metodo **seek()**.

**RandomAccessFile** è una classe completamente separata dagli stream, anche se i metodi in comune sono molti.

- **StreamTokenizer**

Questa classe permette di scomporre in token uno **InputStream**, facilitando, per esempio, l'analisi sintattica di file aventi una determinata grammatica.

142

## JAVA 1.1 - I/O System

Con il rilascio della nuova versione di JAVA sono state effettuate alcune modifiche sulla gestione degli stream. Sono state introdotte due nuove classi antenate: **Reader** e **Writer**.

Queste classi introducono funzionalità per la lettura e la scrittura dei caratteri, nonché funzionalità di lettura e scrittura Unicode.

Inoltre sono state introdotte classi che permettono la gestione di file zip, gzip e file jar.