Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

PROGETTAZIONE DEL SOFTWARE

(Ing. Gestionale)

Prof. Giuseppe De Giacomo

Anno Accademico 2002-2003

JAVA COME LINGUAGGIO OBJECT-ORIENTED

Terza parte

Generalità su Java

- Linguaggio recente (1996); precursori: Smalltalk (fine anni '70), C++ (inizio anni '80). Fondamentalmente, simile al C++, ma con alcune importanti differenze. Esiste già C#!

- Ha avuto successo perché:
  - librerie standard (in particolare, per Internet) native
  - portabilità su varie piattaforme

- In questo corso:

  Sì: metodologie di programmazione orientate agli oggetti ("da UML a Java")

  No: librerie particolari di Java (ma il linguaggio verrà usato anche in altri corsi: *Basi di dati*, ...)

Ripasso e studio di nuovi costrutti di Java

Argomenti che tratteremo in questa parte di corso:

1. Allocazione di variabili e di oggetti
2. Uguaglianza superficiale ed uguaglianza profonda
3. Copia superficiale e copia profonda
4. Passaggio di parametri
5. Package
6. Derivazione di classi, ereditarietà, classi astratte
7. Interfacce

Tipi di dato in Java

- Dobbiamo distinguere nettamente fra:

  1. variabili i cui valori sono di *tipi di dato di base (o primitivi)*, cioè `int`, `char`, `float`, `double`, `boolean`, e
  2. *oggetti*, cioè istanze delle *classi*.

- In particolare, la memoria per la loro rappresentazione viene, rispettivamente:

  1. *allocata* automaticamente, senza necessità di una esplicita richiesta mediante istruzione durante l'esecuzione del programma, ovvero
  2. *allocata* durante l'esecuzione del programma, a fronte della esecuzione di una opportuna istruzione (cioè con `new`).

Riferimenti e oggetti

Dobbiamo inoltre distinguere nettamente fra:

- *riferimenti* a oggetti, e
- oggetti veri e propri.

I primi sono di fatto assimilabili ai tipi di dato di base. Infatti, come questi, la memoria per la loro rappresentazione viene allocata automaticamente, senza necessità di una esplicita richiesta mediante istruzione durante l'esecuzione del programma

Inoltre, essi vengono inizializzati automaticamente (al valore `null`).

Un riferimento è un indirizzo di memoria

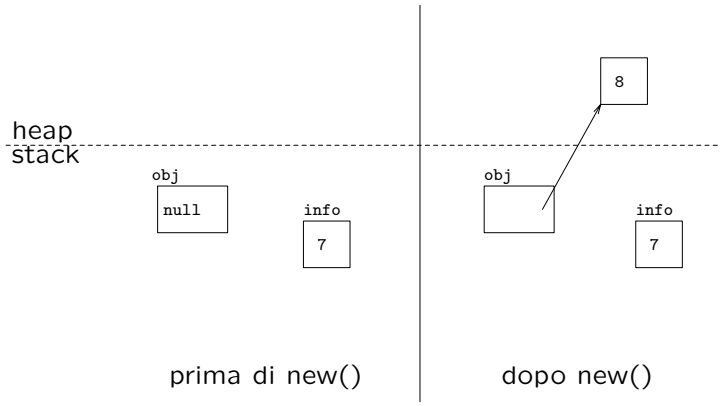
Inizializzazioni implicite per i campi delle classi

Un campo di tipo	Viene inizializzato implicitamente a	Note
`int`	0	
`float, double`	0.0	
`char`	' '	spazio
`boolean`	`false`	
`class C`	`null`	il riferimento, non l'oggetto

- Queste inizializzazioni avvengono automaticamente:
  - per i campi dati di una classe;
  - ma non per le variabili locali delle funzioni.


```
// File ParteTerza/Esempio0.java
// Evidenzia la differenza fra valore di tipo base e oggetto

public class Esempio0 {
  public static void main(String[] args) {

    int info = 7; // dichiarazione di una variabile locale di tipo base;
                  // la memoria viene allocata SENZA esplicita richiesta
                  // mediante istruzione durante l'esecuzione del programma

    Integer obj; // Integer e' una classe: dichiarazione di un riferimento
                 // la memoria DEL SOLO RIFERIMENTO viene allocata SENZA esplicita
                 // richiesta mediante istruzione durante l'esecuzione del programma
                 // Ad esempio:
                 //    System.out.println(obj);
                 //                       ^
                 // Variable obj may not have been initialized.

    obj = new Integer(8); // la memoria DELL'OGGETTO viene allocata durante l'esecuzione
                          // del programma mediante l'esecuzione dell'istruzione new
    System.out.println(obj);
  }
}
```


Evoluzione (run-time) dello stato della memoria



Allocazione della memoria

Allocazione statica: viene *decisa* a tempo di *compilazione*. Viene effettuata prendendo memoria da un'area detta *stack*.

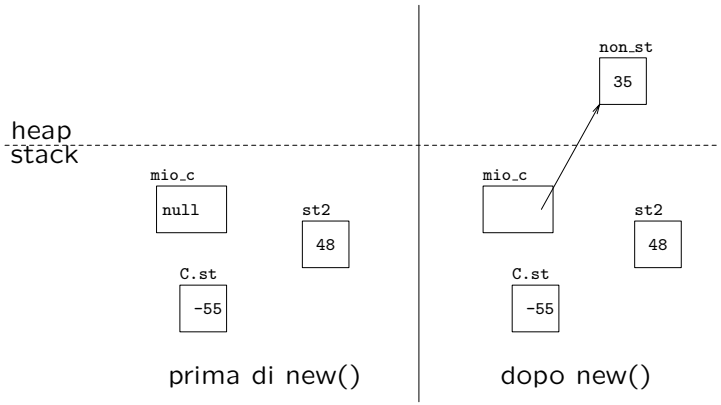
Esempi: campo dati `static` di una classe, variabile locale in una funzione, ...

Allocazione dinamica: viene *decisa* a tempo di *esecuzione*. Viene effettuata prendendo memoria da un'area detta *heap*.

Esempio: creazione di un oggetto tramite `new`.


```
// File ParteTerza/Esempio1.java
// Evidenzia la differenza fra allocazione statica e dinamica

class C {
  static int st; // allocazione STATICA
  int non_st;
}

public class Esempio1 {
  public static void main(String[] args) {
    int st2;     // allocazione STATICA
    st2 = 48;    // OK: la memoria è stata già allocata
    C.st = -55;  // OK: la memoria è stata già allocata
    C mio_c;     // allocazione STATICA (del solo riferimento)
    // mio_c.non_st = 35;  // NO: la memoria non è stata ancora allocata
    mio_c = new C();       // allocazione DINAMICA dell'oggetto
    mio_c.non_st = 35;     // OK: la memoria è stata già allocata
  }
}
```


Evoluzione (run-time) dello stato della memoria



Campi dati static

- In una classe `C` un qualsiasi campo dati `s` può essere dichiarato `static`.
- Dichiarare `s` come `static` significa che `s` è un campo **relativo alla classe** `C`, non ai singoli oggetti di `C`.
- Pertanto, per un tale campo `s` esiste **una sola locazione di memoria**, che viene allocata **prima** che venga allocato qualsiasi oggetto della classe `C`.
- Viceversa, per ogni campo non static esiste una locazione di memoria **per ogni oggetto**, che viene allocata contestualmente a `new()`.

Funzioni in Java

- esiste un solo tipo di *unità di programma*: la **funzione**
- ogni funzione appartiene ad (è *incapsulata* in) **una classe**
- esiste un'unità di programma principale (`main()`)
- le funzioni si distinguono in:
  - `static`: sono relative *alla classe*
  - non `static`: sono relative *agli oggetti della classe*

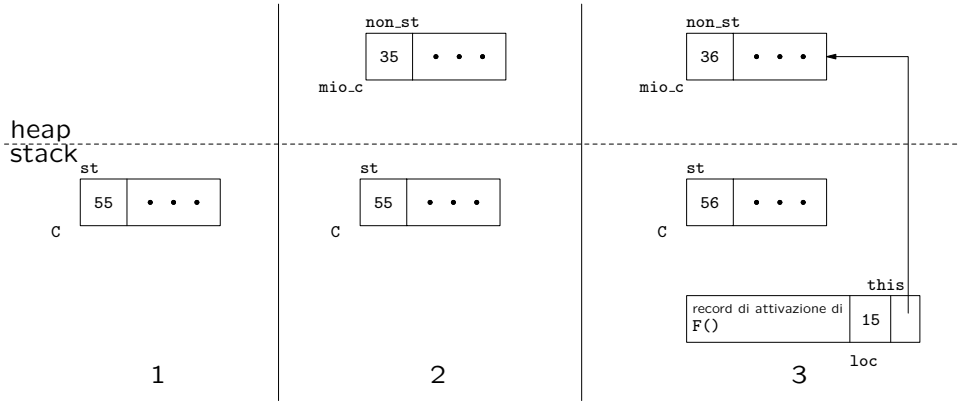

```
// File ParteTerza/Esempio2.java
// Evidenzia la differenza fra funzioni statiche e non
class C {
  int x;
  void F() { System.out.println(x); }
  static void G() { System.out.println("Funzione G()"); }
  // static void H() { System.out.println(x); }
  //                                       ^
  // Can't make a static reference to nonstatic variable x in class C.
}

public class Esempio2 {
  public static void main(String[] args) {
    C c1 = new C();
    c1.x = -4;
    c1.F(); // invocazione di funzione NON STATIC
    C.G();  // invocazione di funzione STATIC
    c1.G(); // invocazione di funzione STATIC
  }
}
```


Modello run-time dell'invocazione di funzioni

- Quando una funzione viene invocata, viene allocato **nello stack** un *record di attivazione*, che contiene le informazioni indispensabili per l'esecuzione.
- Fra queste informazioni, ci sono:
  - le variabili locali,
  - un *riferimento*, o *puntatore*, (il cui nome è `this`) all'oggetto di invocazione.
- Al termine dell'esecuzione della funzione, il record di attivazione viene deallocato.


```
// File ParteTerza/Esempio3.java
// Evidenzia il comportamento run-time di una chiamata di funzione
class C {
  static int st;
  int non_st;
  void F() {
    int loc = 15;
    non_st++;
    st++;
    System.out.println(st + "  " + non_st + "  " + loc);  }
}
public class Esempio3 {
  public static void main(String[] args) {
    C.st = 55;                                    // 1
    C mio_c;
    // mio_c.F();  // NO: la memoria non è stata ancora allocata
    mio_c = new C();
    mio_c.non_st = 35;                            // 2
    mio_c.F();     // OK: stampa 56 36 15         // 3
  } }
```

Evoluzione (run-time) dello stato della memoria



Esercizio 1: stack e heap

Progettare due classi:

`Punto`: per la rappresentazione di un punto nello spazio tridimensionale, come aggregato di tre valori di tipo reale;

`Segmento`: per la rappresentazione di un segmento nello spazio tridimensionale, come aggregato di due punti.

Esercizio 1 (cont.)

Scrivere una funzione `main()` in cui vengono creati:

- due oggetti della classe `Punto`, corrispondenti alle coordinate $\langle 1, 2, 4 \rangle$ e $\langle 2, 3, 7 \rangle$, rispettivamente;
- un oggetto della classe `Segmento`, che unisce i due punti suddetti.

Raffigurare l'evoluzione dello stato della memoria, distinguendo fra stack e heap.

Uguaglianza fra valori di un tipo base

Se vogliamo mettere a confronto due valori di un tipo base, usiamo l'*operatore di uguaglianza* '`==`'.

Ad esempio:

```
int a = 4, b = 4;
if (a == b) // verifica uguaglianza fra VALORI
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");
```

Uguaglianza fra oggetti: superficiale

Se usiamo '==' per mettere a confronto **due oggetti**, stiamo verificandone l'uguaglianza *superficiale*.

Ad esempio:

```
class C {
  int x, y;
}
// ...
    C c1 = new C(), c2 = new C();
    c1.x = 4; c1.y = 5;
    c2.x = 4; c2.y = 5;
    if (c1 == c2)
      System.out.println("Uguali!");
    else
      System.out.println("Diversi!");
```


Uguaglianza fra oggetti: superficiale (cont.)

Viene eseguito il ramo `else` (`"Diversi!"`).

Infatti, '==' effettua un confronto fra *i valori dei riferimenti*, ovvero fra i due indirizzi di memoria in cui si trovano gli oggetti.

Riassumendo, diciamo che:

1. '==' verifica l'uguaglianza *superficiale*,
2. gli oggetti `c1` e `c2` **non sono uguali superficialmente**.

Uguaglianza fra oggetti: funzione equals()

In Java esiste un'altra maniera per verificare l'uguaglianza fra oggetti, tramite la funzione `equals()`.

`equals()` **esiste implicitamente** per ogni classe (standard, o definita dal programmatore), e *se non ridefinita*, si comporta come l'operatore '=='.

Pertanto, anche nel seguente esempio viene eseguito il ramo `else` (`"Diversi!"`).

```
class C {
  int x, y;
}
// ...
    C c1 = new C(), c2 = new C();
    c1.x = 4; c1.y = 5;
    c2.x = 4; c2.y = 5;
    if (c1.equals(c2))
      System.out.println("Uguali!");
    else
      System.out.println("Diversi!");
```


Uguaglianza fra oggetti: profonda

È tuttavia possibile *ridefinire* il significato della funzione `equals()`, facendo in maniera tale che verifichi l'*uguaglianza profonda* fra oggetti.

Nel seguente esempio, ciò viene fatto per la classe `B`.

```
class B {
  int x, y;
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      B b = (B)o;
      return (b.x == x) && (b.y == y);
    }
    else return false;
  }
}
```

Uguaglianza fra oggetti: profonda (cont.)

Alcuni commenti sulla funzione `equals()` ridefinita per la classe `B`:

- `equals(Object o)` è una funzione *ereditata* dalla classe `Object`, che vogliamo ridefinire per la classe `B`.
- Tecnicamente, ciò viene effettuato mediante *overriding* di tale funzione, e comporta la definizione in `B` di una funzione con argomento (`o`) di classe `Object` (e **NON** di classe `B`) – *maggiori dettagli in seguito*.
- Il fatto che `o` sia un riferimento, di classe `Object`, ha alcune conseguenze:
  - dobbiamo essere sicuri che `o` si riferisca ad un oggetto che è stato già allocato (`o != null`);
  - dobbiamo essere sicuri che `o` si riferisca ad un oggetto della classe `B` (`getClass().equals(o.getClass())`).

Uguaglianza fra oggetti: profonda (cont.)

Alcuni commenti sulla funzione `getClass()`:

- La funzione `getClass()` è definita in `Object` e restituisce la classe dell'oggetto di invocazione (cioè, la classe più specifica di cui l'oggetto d'invocazione è istanza).
- Più precisamente, `getClass()` restituisce un oggetto della classe predefinita `Class` associato alla classe dell'oggetto di invocazione.
- Esiste un oggetto di classe `Class` per ogni classe definita nel programma.

Uguaglianza fra oggetti: profonda (cont.)

Ancora sulla funzione `equals()` ridefinita per la classe `B`:

- Se la condizione logica dell'`if` risulta vera (ovvero l'oggetto denotato da `o` esiste ed appartiene alla classe `B`), allora possiamo:
  - definire un riferimento (non un oggetto!) `b` di classe `B`;
  - assegnare a `b` l'oggetto denotato da `o`, attraverso una conversione di tipo esplicita (`B b = (B)o`);
  - verificare l'uguaglianza tra i singoli campi della classe `B` (`return (b.x == x) && (b.y == y)`).

Uguaglianza fra oggetti: profonda (cont.)

Riassumendo, se desideriamo che per una classe `B` si possa verificare l'uguaglianza profonda fra oggetti, allora:

server: il **progettista** di `B` deve effettuare l'overriding della funzione `equals()`, secondo le regole viste in precedenza;

client: il **cliente** di `B` deve effettuare il confronto fra oggetti usando `equals()` per l'uguaglianza profonda e '`==`' per quella superficiale.

```
B b1 = new B(), b2 = new B();
b1.x = 4; b1.y = 5;
b2.x = 4; b2.y = 5;
if (b1.equals(b2))
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");
```

Uguaglianza: classe String

L'unica eccezione a quanto detto è la classe predefinita `String`.

In `String` la funzione `equals()` è già ridefinita in maniera tale da realizzare l'uguaglianza profonda.

```
String s1 = new String("ciao");
String s2 = new String("ciao");

if (s1 == s2)
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");

if (s1.equals(s2))
  System.out.println("Uguali!");
else
  System.out.println("Diversi!");
```


Esercizio 2: uguaglianza

Progettare tre classi:

`Punto`: vedi esercizio 1;

`Segmento`: vedi esercizio 1;

`Valuta`: per la rappresentazione di una quantità di denaro, come aggregato di due valori di tipo intero (unità e centesimi) ed una `String` (nome della valuta).

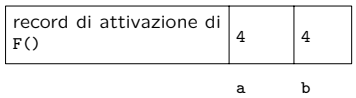
Per tali classi, ridefinire il significato della funzione `equals()`, facendo in maniera tale che verifichi l'*uguaglianza profonda* fra oggetti.

Copia di valori di un tipo base

Se vogliamo copiare un valore di un tipo base in una variabile dello stesso tipo, usiamo l'*operatore di assegnazione* '='.

Ad esempio:

```
void F() {
// ...
    int a = 4, b;
    b = a;
// ...
} // F()
```



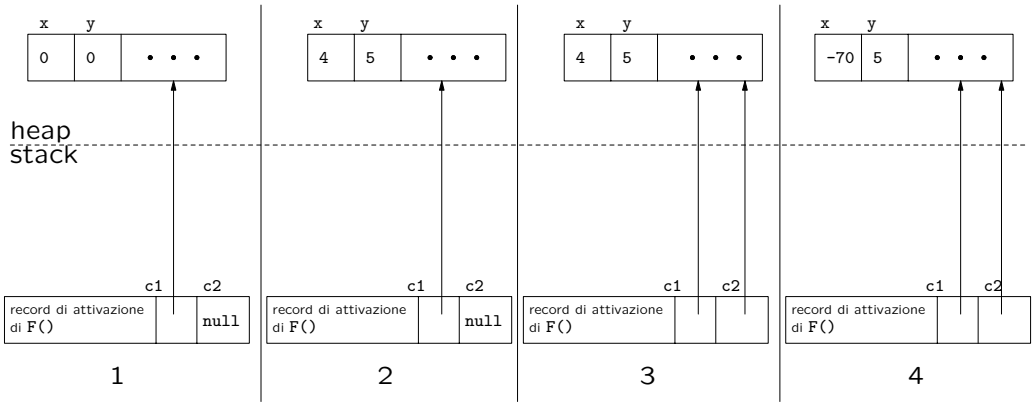
Copia fra oggetti: superficiale

Se usiamo '=' per copiare **due oggetti**, stiamo effettuando la copia *superficiale*.

Ad esempio:

```
class C {
  int x, y;
}
void F() {
// ...
    C c1 = new C(), c2;                          // 1
    c1.x = 4; c1.y = 5;                          // 2
    System.out.println("c1.x: " + c1.x + ", c1.y: " + c1.y);
    c2 = c1;    // COPIA SUPERFICIALE            // 3
    System.out.println("c2.x: " + c2.x + ", c2.y: " + c2.y);
    c2.x = -70; // SIDE-EFFECT                   // 4
    System.out.println("c1.x: " + c1.x + ", c1.y: " + c1.y);
// ...
} // F()
```

Evoluzione (run-time) dello stato della memoria



Copia fra oggetti: superficiale (cont.)

L'operatore '=' effettua una copia fra *i valori dei riferimenti*, ovvero fra i due indirizzi di memoria in cui si trovano gli oggetti.

Riassumendo, diciamo che:

1. '=' effettua la copia *superficiale*,
2. in quanto tale **non crea un nuovo oggetto**,
3. a seguito dell'assegnazione, i due riferimenti `c1` e `c2` *sono uguali superficialmente*,
4. ogni azione sul riferimento `c2` si ripercuote sull'oggetto a cui si riferisce anche `c1`.

Copia fra oggetti: clone() e copia profonda

In Java esiste un'altra maniera per copiare oggetti, la funzione `clone()`.

`clone()` **non esiste implicitamente** per una classe (*maggiori dettagli in seguito*). Se lo desideriamo, possiamo **ridefinirla**, facendo in maniera tale che effettui la *copia profonda* fra oggetti. Nel seguente esempio, ciò viene fatto per la classe `B`.

```
class B implements Cloneable {
  int x, y;
  public Object clone() {
    try {
      B t = (B)super.clone(); // Object.clone copia campo a campo
      return t;
    } catch (CloneNotSupportedException e) {
      // non puo' accadere, ma va comunque gestito
      throw new InternalError(e.toString());
    }
  }
}
```


Copia fra oggetti: copia profonda (cont.)

Alcuni commenti sulla funzione `clone()` ridefinita per la classe `B`:

- `clone()` è una funzione *ereditata* dalla classe `Object` (ma non direttamente disponibile per i clienti: *maggiori dettagli in seguito*) che vogliamo ridefinire per la classe `B`.
- Tecnicamente, ciò viene effettuato mediante *overriding* di tale funzione, e comporta la definizione in `B` di una funzione che restituisce un riferimento di classe `Object` (e **NON** di classe `B`) – *maggiori dettagli in seguito*.
- Dobbiamo inoltre dichiarare che `B` *implementa l'interfaccia* `Cloneable` – *maggiori dettagli in seguito*.

Copia fra oggetti: copia profonda (cont.)

Altri commenti sulla funzione `clone()` ridefinita per la classe `B`:

- Mediante `super.clone()` viene invocata la funzione `clone()` di `Object`. Questa funzione crea (allocandolo dinamicamente) l'*oggetto clone* ed esegue una **copia superficiale dei campi** (cioè mediante '=') dell'oggetto di invocazione, indipendentemente dalla classe a cui questo appartiene.

- Il riferimento restituito da `super.clone()`, che è di tipo `Object`, viene convertito, mediante *casting* in un riferimento di tipo `B` (`(B)super.clone()`).

- Infine, dobbiamo trattare in modo opportuno l'eccezione (*checked exception*) `CloneNotSupportedException` che `clone()` di `Object` genera se invocata su un oggetto di una classe che non implementa l'interfaccia `Cloneable`.

Copia fra oggetti: copia profonda (cont.)

Riassumendo, se desideriamo che per una classe `B` si possa effettuare la copia profonda fra oggetti, allora:

server: il **progettista** di `B` deve effettuare l'overriding della funzione `clone()`, secondo le regole viste in precedenza;

client: il **cliente** di `B` deve effettuare la copia fra oggetti usando `clone()` per la copia profonda e '=' per quella superficiale.

```
B b1 = new B();
b1.x = 10; b1.y = 20;
B b2 = (B)b1.clone();
System.out.println("b2.x: " + b2.x + ", b2.y: " + b2.y);
```


Riassunto uguaglianza e copia

	SERVER	CLIENT
Uguaglianza SUPERFICIALE	non ridefinisce `equals()`	usa '=='
Uguaglianza PROFONDA	ridefinisce `equals()`	usa `equals()`
Copia SUPERFICIALE	non ridefinisce `clone()`	usa '='
Copia PROFONDA	ridefinisce `clone()`	usa `clone()`

Copia profonda: classe String

Come per le altre classi predefinite, la funzione `clone` non esiste per la classe `String`.

Se vogliamo fare una copia profonda di un oggetto di tale classe, possiamo utilizzare, mediante `new`, un suo costruttore speciale, che accetta un argomento di tipo `String`.

```
String s1 = new String("ciao");
String s2;

s2 = new String(s1); // uso del costruttore con argomento String
                     // ora s2 si riferisce ad una copia profonda di s1
```

Esercizio 3: copia

Con riferimento alle tre classi `Punto`, `Segmento` e `Valuta` dell'esercizio 2, ridefinire il significato della funzione `clone()`, facendo in maniera tale che effettui la *copia profonda* fra oggetti.

Comunicazione fra unità di programma

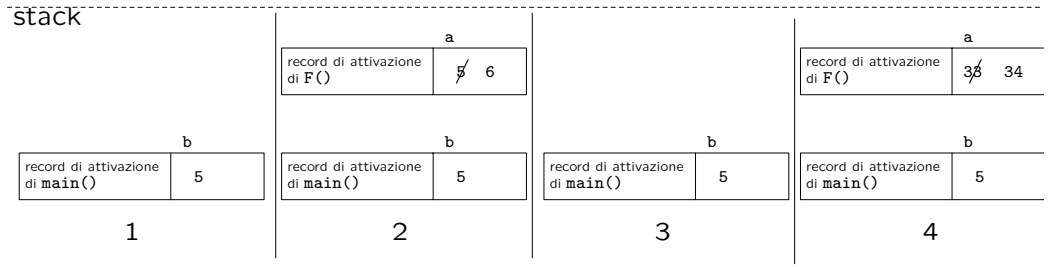
- Il passaggio di parametri ad una funzione è **solamente per valore**.
- Ciò significa che:
  1. il **parametro attuale** può essere un'espressione qualsiasi (costante, variabile, espressione non atomica);
  2. viene effettuata una **copia** del valore del parametro attuale nella locazione di memoria corrispondente al **parametro formale** che si trova nel record di attivazione della funzione chiamata;
  3. tale locazione viene ovviamente perduta al termine dell'esecuzione della funzione, quando il record di attivazione corrispondente viene deallocato.

Comunicazione fra unità di programma (cont.)

Esempio: argomento passato appartiene ad un **tipo base** (`int`).

```
public static void F(int a) {
  // a e' il PARAMETRO FORMALE
  a++;
  System.out.println("a: " + a);
}

public static void main(String[] args) {
  int b = 5;                        // 1
  F(b);                             // 2 --  b e' il PARAMETRO ATTUALE
  System.out.println("b: " + b);    // 3
  F(33);                            // 4 -- 33 e' il PARAMETRO ATTUALE
}
```


Evoluzione (run-time) dello stato della memoria



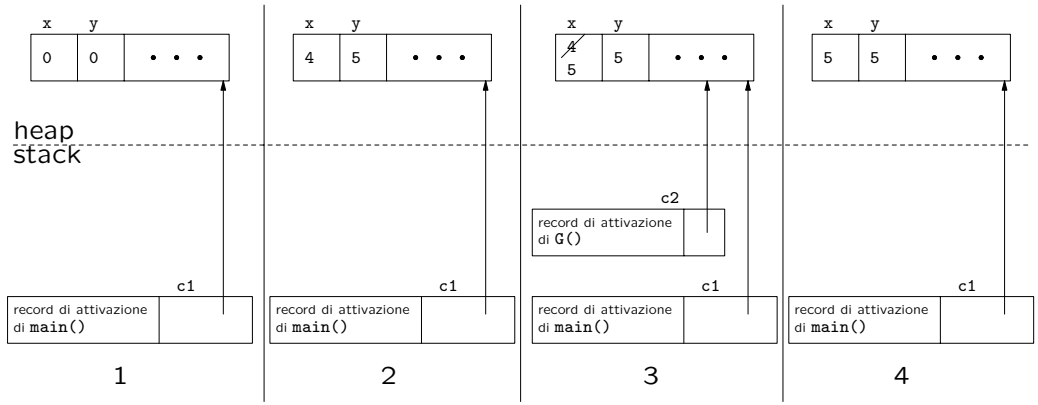
Comunicazione fra unità di programma (cont.)

Esempio: argomento passato appartiene ad una **classe** (`C`).

```
class C {
  int x, y;
}
// ...
  public static void G(C c2) {
    c2.x++;
  }

  public static void main(String[] args) {
    C c1 = new C();                                          // 1
    c1.x = 4; c1.y = 5;                                      // 2
    G(c1);    // SIDE-EFFECT                                // 3
    System.out.println("c1.x: " + c1.x + ", c1.y: " + c1.y); // 4
  }
}
```

Nota: l'oggetto cambia, **il riferimento** no!

Evoluzione (run-time) dello stato della memoria



Riassunto passaggio argomenti

	argomento TIPO BASE	argomento CLASSE
viene copiato	il valore	il riferimento, non l'oggetto
può cambiare	niente	oggetto
non può cambiare	—	riferimento

Nota: se la funzione cliente vuole essere assolutamente sicura di non alterare l'oggetto passatogli tramite riferimento, deve:

1. **farsene una copia** mediante `clone()`,
2. lavorare sulla copia.

Restituzione da parte di una funzione

Anche la restituzione da parte di una funzione è **solamente per valore**.

```
  public static C H(C c3) {
    c3.x++;
    return c3;  // RESTITUZIONE PER VALORE
  }
// ...
    System.out.println(H(c1).x);
```

Pertanto, tutte le considerazioni sul passaggio di argomenti valgono anche per la restituzione. Ad esempio:

- se il tipo restituito è un tipo base, viene fatta una copia;
- se il tipo restituito è una classe, viene fatta una copia del riferimento, **ma non dell'oggetto**.

Esercizio 4: passaggio e restituzione

Con riferimento alla classe `Segmento` vista in precedenza, scrivere le seguenti funzioni esterne ad essa, tutte con un argomento di tale classe:

1. `lunghezza()`, che restituisce un valore di tipo `double` corrispondente alla lunghezza del segmento;

2. `inizioInOrigine()`, che modifica l'argomento ponendo il punto di inizio nel punto di origine (cioè di coordinate $\langle 0, 0, 0 \rangle$);

3. `mediano()`, che restituisce un riferimento al punto (di classe `Punto`) mediano del segmento;

4. `meta()`, che restituisce un riferimento al segmento (di classe `Segmento`) i cui estremi sono, rispettivamente, l'inizio e il mediano del segmento passato come argomento.

Esercizio 5: cosa fa questo programma?

```
// File ParteTerza/Esercizio5.java
public class Esercizio5 {
  static void mistero1(int i, int j) {
    int temp = i;
    i = j;
    j = temp;
  }

  static void mistero2(Integer i, Integer j) {
    Integer temp = new Integer(i.intValue());
    i = j;
    j = temp;
  }

  public static void main(String[] args) {
    int p = 5, s = 7;
    mistero1(p,s);
    Integer o_p = new Integer(50), o_s = new Integer(70);
    mistero2(o_p,o_s);
    System.out.println("p: " + p + " s: " + s +
                       " o_p: " + o_p + " o_s: " + o_s);
  }
}
```


Esercizio 6: side-effect

Scrivere un'unità di programma che riceve, come parametri di input, due locazioni di tipo intero e scambia il loro contenuto.

Suggerimento: non utilizzare né `int` né `Integer` per rappresentare gli interi.

Struttura di un programma Java

- Una *classe* è un aggregato di *campi*, che possono essere **dati**, **funzioni**, **classi**.

- La definizione di una classe è contenuta in un *file*, e un file contiene una o più definizioni di classi.

- Un *package* è una collezione di classi.

- Un file (con tutte le classi in esso contenute) appartiene ad uno ed un solo package.

- Un *programma* è una collezione di una o più classi, appartenenti anche a diversi package. Una di queste classi deve contenere la funzione che è il punto di accesso per l'esecuzione del programma (`main()`).

Package

- Esistono nella libreria standard Java molti package (ad esempio `java.io`)
- Un **nuovo** package `mio_pack` viene dichiarato scrivendo all'inizio di un file `F.java` la dichiarazione:

```
package mio_pack;
```

- La stessa dichiarazione in un altro file `H.java`, dichiara che anche quest'ultimo appartiene allo stesso package.
- Se un file non contiene una dichiarazione di package, allora alle classi di tale file viene associato automaticamente un package di default, il cosiddetto *package senza nome*.

Uso dei package

Se, in un file `G.java`, vogliamo usare una classe `C` definita nel package `mio_pack`, possiamo usare due metodi:

1. riferirci ad essa mediante `mio_pack.C` (oppure semplicemente `C`, se essa è definita nel package senza nome);
2. scrivere all'inizio del file `G.java` una delle seguenti dichiarazioni:

```
import mio_pack.C; // semplifica il riferimento alla classe C
                   // del package mio_pack
import mio_pack.*; // semplifica il riferimento a tutte le classi
                   // del package mio_pack
```

   A questo punto, possiamo riferirci alla classe mediante `C` (senza specificare esplicitamente che appartiene a `mio_pack`).

Struttura dei package e dei direttori

Tutti i file relativi al package `mio_pack` tipicamente risiedono in un **direttorio** dal nome `mio_pack`.

È possibile definire altri package con un nome del tipo `mio_pack.mio_subpack`.

In tal caso, tutti i file relativi al package `mio_pack.mio_subpack` risiederanno in un **sottodirettorio** di `mio_pack` dal nome `mio_subpack`.

La dichiarazione `import mio_pack.*;` **non significa** che stiamo importando anche da `mio_pack.mio_subpack`.

Se desideriamo fare ciò, dobbiamo dichiararlo **esplicitamente** mediante la dichiarazione `import mio_pack.mio_subpack.*;`

Esempio uso package

```
// File ParteTerza/mio_package/C.java
package mio_package;

public class C {
  public static void F_C() {
    System.out.println("Sono F_C()");
  }
}

// File ParteTerza/mio_package/mio_subpackage/D.java
package mio_package.mio_subpackage;
public class D {
  public static void F_D() {
    System.out.println("Sono F_D()");
  }
}
```

```
// File ParteTerza/Esempio13.java
// uso package

import mio_package.mio_subpackage.*;

public class Esempio13 {
  public static void main(String[] args) {
    mio_package.C c = new mio_package.C();
    c.F_C();
    D d = new D();
    d.F_D();
  }
}
```

Esercizio 6bis: package

```
// File Esempio32.java
import java.io.*;

public class Esempio32 {
  public static void main(String[] args) throws IOException {
    // stampa su schermo il file passato tramite linea di comando
    FileInputStream istream = new FileInputStream(args[0]);
    BufferedReader in = new BufferedReader(new InputStreamReader(istream));
    boolean finito = false;
    while (!finito) {
      String linea = in.readLine();
      if (linea != null)
        System.out.println(linea);
      else
        finito = true;
    }  }  }
```

Riscrivere il programma eliminando la dichiarazione `import java.io.*;`.

Classi: qualificatori dei campi dati

Esistono tre tipi di qualificazione per i campi dati:

```
class C {
  int x;
  static int y;
  final int z = 12;
}
```

static: campo relativo alla classe, non all'oggetto;

esiste anche per campi funzione, con lo stesso significato;

Classi: qualificatori dei campi dati (cont.)

final: campo costante, deve essere inizializzato;

esiste anche per campi funzione, **ma ha diverso significato** – *maggiori dettagli in seguito*;

nessuna: campo relativo all'oggetto;

può essere inizializzato, altrimenti riceve un valore di default:

- `0` (tipi base numerici);
- `false` (tipo base `boolean`);
- `null` (riferimenti a oggetti).

Classi: sovraccarico di funzioni

È ammesso l'*overloading* (dall'inglese, sovraccarico) di funzioni.

È possibile definire nella stessa classe più funzioni **con lo stesso nome**, purché differiscano nel numero e/o nel tipo dei parametri formali.

Non è invece possibile definire due funzioni con lo stesso nome e stesso numero e tipo di argomenti ma diverso tipo di ritorno.

```
class C {
  int x;
  void F() { x++; }                       // OK
  void F(int i) { x = i; }                // OK
  void F(int i, int j) { x = i * j; }     // OK
  //  int F() { return x; }               // NO
  //      ^
  // Methods can't be redefined with a different return type
}
```


Classi: costruttori

Un costruttore è una funzione che:

- si chiama con lo stesso nome della classe;
- gestisce la nascita di un oggetto;
- viene invocata con `new`;
- (come le altre) può essere sovraccaricata.

Costruttori: esempio di definizione e uso

```
class C {
  int x, y;
  C(int p) { x = p; }
  C(int p, int s) { x = p; y = s; }
}
// ..
  public static void main(String[] args) {
    C c1 = new C(4);    // viene scelto il costruttore AD UN ARGOMENTO
    System.out.println("c1.x: " + c1.x + ", c1.y: " + c1.y);
    C c2 = new C(7,8);  // viene scelto il costruttore A DUE ARGOMENTI
    System.out.println("c2.x: " + c2.x + ", c2.y: " + c2.y);
  }
}
```


Costruttore senza argomenti

- Per le classi che **non hanno** dichiarazioni di costruttori viene invocato il cosiddetto *costruttore standard*.
- Il costruttore standard esiste per tutte le classi e inizializza a **valori di default** i valori dei campi dati.
- Il costruttore standard viene automaticamente **inibito** dal compilatore a fronte della dichiarazione di **un qualsiasi** costruttore da parte del programmatore.
- In quest'ultimo caso, **può** essere dichiarato esplicitamente un costruttore senza argomenti.

Classi: costruttore senza argomenti (esempio)

```
class C { // HA il costr. senza argomenti
  int x, y;
}

class C1 { // NON HA il costr. senza argomenti
  int x, y;
  C1(int p, int s) { x = p; y = s; }
}

class C2 { // HA il costr. senza argomenti
  int x, y;
  C2() { x = 0; y = 0; }
  C2(int p, int s) { x = p; y = s; }
}
```


Esercizio 7: costruttori

Equipaggiare le classi `Punto` e `Segmento` con opportuni costruttori.

Utilizzare i costruttori per creare:

- due oggetti della classe `Punto`, corrispondenti alle coordinate $\langle 1, 2, 4 \rangle$ e $\langle 2, 3, 7 \rangle$, rispettivamente;
- un oggetto della classe `Segmento`, che unisce i due punti suddetti.

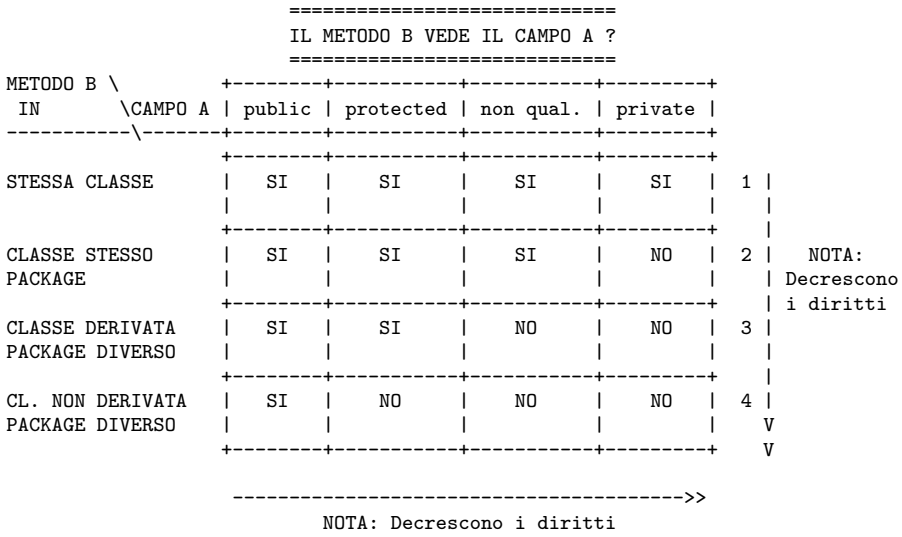
Classi: livelli di accesso

Un campo di una classe (dato, funzione o classe) può essere specificato con uno fra **quattro** livelli di accesso:

1. `public`,
2. `protected`,
3. `private`,
4. non qualificato (è il *default*, intermedio fra protetto e privato).

Anche un'intera classe `C` (ma solo se **non è interna ad altra classe**) può essere dichiarata public, ed in tale caso la classe deve essere dichiarata nel file `C.java`.

Classi: regole di visibilità



Commento sulle regole di visibilità

Le regole di visibilità vengono sfruttate per aumentare l'*information hiding*.

Ricordiamo che uno dei princìpi di buona modularizzazione è che l'information hiding deve essere **alto**.

In base a questo principio, i campi dati **non sono mai pubblici**, ma **privati** o **protetti**.

In tal modo diamo al cliente un **accesso controllato** ai campi dati, mediante le funzioni (tipicamente, *ma non sempre*, pubbliche).

Metodologia completa in seguito

Visibilità: esempio

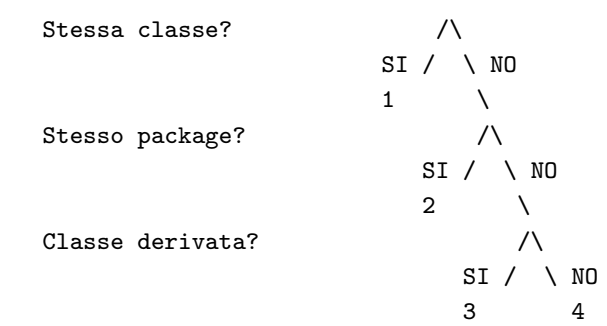
```
// File ParteTerza/Esempio14.java

class C {
  private int x, y;
  public C(int a, int b) { x = a; y = b; }
  public void stampa() { System.out.println("x: " + x + ", y: " + y); }
}

class Esempio14 {
  public static void main(String[] args) {
    C c = new C(7,12);   // OK: il costruttore di C e' pubblico
    c.stampa();          // OK: la funzione stampa() di C e' pubblica
  //    int val = c.x;   // NO: il campo x e' privato in C
  //                ^
  //Variable x in class C not accessible from class Esempio14.
  }
}
```


Esercizio 8: visibilità

Verificare, tramite opportuni frammenti di codice, la veridicità delle regole di visibilità della tabella vista in precedenza.

Regole di visibilità (cont.)

Il seguente "albero delle decisioni" fa notare che essere nello stesso package **dà più diritti** di essere una classe derivata.



Va inoltre ricordato che ogni package è "aperto", ovvero possiamo sempre dichiarare di fare parte di un package qualunque.

Derivazione fra classi

È possibile dichiarare una classe `D` come *derivata* da una classe `B`.

```
class B {               // CLASSE BASE
  int x;
  void G() { x = x * 20; }
// ...
}

class D extends B {     // CLASSE DERIVATA
  void H() { x = x * 10; }
// ...
}
```


Princìpi fondamentali della derivazione

I quattro **princìpi fondamentali** del rapporto tra classe derivata e classe base:

1. Tutte le proprietà definite per la classe base vengono **implicitamente definite** anche nella classe derivata, cioè vengono **ereditate** da quest'ultima.

   Ad esempio, implicitamente la classe derivata `D` ha:
   - un campo dati `int x;`
   - una funzione `void G()`

2. La classe derivata può avere **ulteriori proprietà** rispetto a quelle ereditate dalla classe base.

   Ad esempio, la classe `D` ha una funzione `void H()`, in più rispetto alla classe base `B`.

Princìpi fondamentali della derivazione (cont.)

3. Ogni oggetto della classe derivata **è anche** un oggetto della classe base.

   Ciò implica che è possibile usare un oggetto della classe derivata **in ogni situazione o contesto** in cui si può usare un oggetto della classe base.

   Ad esempio, i seguenti usi di un oggetto di classe `D` sono leciti.

```
  static void stampa(B bb) {
    System.out.println(bb.x);
  }
//...
D d = new D();
d.G();     // OK: uso come ogg. di invocazione di funz. definita in B
stampa(d); // OK: uso come argomento in funz. definita per B
```

La classe `D` *è* compatibile *con la classe* `B`

Princìpi fondamentali della derivazione (cont.)

4. **Non è vero che** un oggetto della classe base è anche un oggetto della classe derivata.

   Ciò implica che **non è sempre possibile** usare un oggetto della classe base laddove si può usare un oggetto della classe derivata.

```
B b = new B();
// b.H();
//    ^
// Method H() not found in class B.

// d = b;
//   ^
// Incompatible type for =. Explicit cast needed to convert B to D.

b = d;     // OK: D al posto di B
```

Esercizio 9: derivazione

Ignorando i costruttori e i livelli d'accesso ai campi, riscrivere la classe `Segmento`, equipaggiandola con una funzione (interna) `stampa()`.

Scrivere una classe `SegmentoOrientato` derivata dalla classe `Segmento`, che contiene anche l'informazione sull'orientazione del segmento (dal punto di inizio a quello di fine, o viceversa).

Verificare se:

- le funzioni esterne precedentemente definite con argomenti di classe `Segmento` (ad es. `lunghezza()`) possano essere usate anche con argomenti di classe `SegmentoOrientato`;
- sia possibile usare la funzione `stampa()` con oggetti di invocazione di classe `SegmentoOrientato`.

Gerarchie di classi

Una classe derivata può a sua volta fungere da classe base per una **successiva derivazione**.

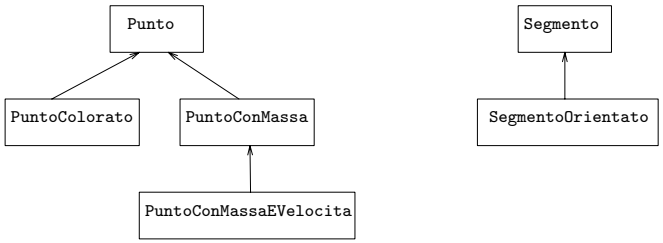
Ogni classe può avere **un numero qualsiasi** di classi derivate.

```
class B { ...
class D extends B { ...
class D2 extends B { ...
```

Una classe derivata può avere **una sola classe base**, (in Java non esiste la cosiddetta *ereditarietà multipla*).

Java supporta una sorta di ereditarietà multipla attraverso le *interfacce* – *maggiori dettagli in seguito*.

Gerarchie di classi: esempio



Esercizio 10: gerarchie di classi

Ignorando le funzioni e i livelli d'accesso dei campi, realizzare in Java la gerarchia di classi della figura precedente.

Significato dell'assegnazione

Abbiamo visto che, in base al principio 3 dell'ereditarietà, la seguente istruzione è lecita:

```
class B {               // CLASSE BASE
class D extends B {     // CLASSE DERIVATA
// ...
D d = new D();
B b = d;                // OK: D al posto di B
```

- **Non viene creato** un nuovo oggetto.
- Esiste **un solo oggetto**, di classe `D`, che viene denotato:
  - **sia** con un riferimento `d` di classe `D`,
  - **sia** con un riferimento `b` di classe `B`.

Casting

Se vogliamo accedere ai campi della classe `D` attraverso `b`, dobbiamo fare un **casting**.

```
// File ParteTerza/Esempio18.java
class B { }
class D extends B { int x_d; }

public class Esempio18 {
  public static void main(String[] args) {
    D d = new D();
    d.x_d = 10;
    B b = d;
    System.out.println(((D)b).x_d);  // CASTING
  }
}
```


Casting (cont.)

Il casting fra classi che sono nello stesso cammino in una gerarchia di derivazione è sempre *sintatticamente* corretto, ma **è responsabilità del programmatore** garantire che lo sia anche *semanticamente*.

```
// File ParteTerza/Esempio19.java
class B { }
class D extends B { int x_d; }

public class Esempio19 {
  public static void main(String[] args) {
    B b = new B();
    D d = (D)b;
    System.out.println(d.x_d); // ERRORE SEMANTICO: IL CAMPO x_d NON ESISTE
    // java.lang.ClassCastException: B
    //  at Esempio19.main(Compiled Code)
  }
}
```


Esercizio 11: casting

Con riferimento al seguente frammento di codice, scrivere una funzione `main()` che contiene un uso semanticamente corretto ed un uso semanticamente scorretto della funzione `f()`.

```
class B { }
class D extends B { int x_d; }
// ...
  static void f(B bb) {
    ((D)bb).x_d = 2000;
    System.out.println(((D)bb).x_d);
  }
```

Derivazione e regole di visibilità

Una classe `D` derivata da un'altra classe `B` ha una relazione particolare con quest'ultima:

- **non è un cliente qualsiasi** di `B`, in quanto vogliamo poter usare oggetti di `D` al posto di quelli di `B`;
- **non coincide** con la classe `B`.

Per questo motivo, è possibile che `B` voglia mettere a disposizione dei campi (ad esempio i campi dati) solo alla classe `D`, e non agli altri clienti. In tal caso, questi campi devono essere dichiarati **protetti** (e non privati).

Ciò garantisce al progettista di `D` di avere accesso a tali campi (vedi tabella delle regole di visibilità), **senza tuttavia garantire tale accesso ai clienti generici** di `B`.

Metodologia completa in seguito

Costruttori di classi derivate

Al momento dell'invocazione di un costruttore della classe derivata, viene chiamato **automaticamente** anche il costruttore senza argomenti della classe base. Ciò avviene:

- sia se la classe base ha il costruttore senza argomenti definito esplicitamente,
- sia se la classe base ha il costruttore senza argomenti standard.

```
class B { ... }
class D extends B { ... }

...
D d = new D(); // invoca il costruttore senza argomenti di B()
               // e quello di D()
```


Costruttori di classi derivate (cont.)

Il costruttore senza argomenti della classe base viene invocato:

- **anche se non definiamo** alcun costruttore per la classe derivata (che ha quindi quello standard senza argomenti),
- **prima** del costruttore della classe derivata (sia quest'ultimo definito esplicitamente oppure no).

Costruttori di classi derivate: esempio

```
// File ParteTerza/Esempio15.java

class B1 { protected int x_b1; }
class D1 extends B1 { protected int x_d1; } // OK: B1 ha cost. senza arg.

class B2 {
  protected int x_b2;
  public B2() { x_b2 = 10; }
}
class D2 extends B2 { protected int x_d2; } // OK: B2 ha cost. senza arg.

class B3 {
  protected int x_b3;
  public B3(int a) { x_b3 = a; }
}
// class D3 extends B3 { protected int x_d3; } // NO: B3 NON ha c. senza arg.
//       ^
// No constructor matching B3() found in class B3.
```

Costruttori di classi derivate: uso di super()

Se la classe base ha costruttori con argomenti, è probabile che si voglia **riusarli**, quando si realizzano le classi derivate.

È possibile invocare esplicitamente un costruttore qualsiasi della classe base **invocandolo**, nel corpo del costruttore della classe derivata.

Ciò viene fatto mediante il costrutto `super()`, che deve essere la *prima istruzione eseguibile* del corpo del costruttore della classe derivata.

Uso di super() nei costruttori: esempio

```
// File ParteTerza/Esempio16.java
class B {
  protected int x_b;
  public B(int a) {  // costruttore della classe base
    x_b = a;
  }
}
class D extends B {
  protected int x_d;
  public D(int b, int c) {
    super(b);       // RIUSO del costruttore della classe base
    x_d = c;
  }
}
class Esempio16 {
  public static void main(String[] args) {
    D d = new D(3,4);  } }
```


Costruttori di classi derivate: riassunto

Comportamento di un costruttore di una classe `D` derivata da `B`:

1. **se** ha come prima istruzione `super()`, allora viene chiamato il costruttore di `B` esplicitamente invocato;

   altrimenti viene chiamato il costruttore senza argomenti di `B`;

2. viene eseguito il corpo del costruttore.

Questo vale **anche per il costruttore standard** di `D` senza argomenti (come al solito, disponibile se e solo se in `D` non vengono definiti esplicitamente costruttori).

Esercizio 12: costruttori e gerarchie di classi

Facendo riferimento alla gerarchia di classi vista in precedenza, riprogettare le classi `Punto`, `PuntoColorato`, `PuntoConMassa` e `PuntoConMassaEVelocita` tenendo conto del livello d'accesso dei campi e con i seguenti costruttori:

`Punto`: con tre argomenti (le tre coordinate) e zero argomenti (nell'origine);

`PuntoColorato`: con quattro argomenti (coordinate e colore);

`PuntoConMassa`: con un argomento (massa); deve porre il punto nell'origine degli assi;

`PuntoConMassaEVelocita`: con due argomenti (massa e velocità); deve porre il punto nell'origine degli assi.

Derivazione e overloading

È possibile fare **overloading** di funzioni ereditate dalla classe base esattamente come lo si può fare per le altre funzioni.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(String s) { ... } // OVERLOADING DI f()
}
```

La funzione `B.f(int)` ereditata da `B` **è ancora accessibile** in `D`.

```
D d = new D();
d.f(1);       // invoca f(int) ereditata da B
d.f("prova"); // invoca f(String) definita in D
```


Overriding di funzioni

Nella classe derivata è possibile anche fare **overriding** (dall'inglese, *ridefinizione*, *sovrascrittura*) delle funzioni della classe base.

Fare overriding di una funzione `f()` della classe base `B` vuol dire definire nella classe derivata `D` una funzione con lo stesso nome, lo stesso numero e tipo di parametri della funzione `f()` definita in `B`. Si noti che **il tipo di ritorno delle due funzioni deve essere identico**.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(String s) { ... } // OVERLOADING DI f()
  public void f(int n) { ... }    // OVERRIDING DI f()
}
```


Overriding di funzioni: esempio

```
// File ParteTerza/Esempio17.java
class B {
  public void f(int i) { System.out.println(i*i); } }
class D extends B {
  public void f(String s) { // OVERLOADING DI f()
    System.out.println(s); }
  public void f(int n) {    // OVERRIDING DI f()
    System.out.println(n*n*n); }
}
public class Esempio17 {
  public static void main(String[] args) {
    B b = new B();
    b.f(5);        // stampa 25
    D d = new D();
    d.f("ciao");   // stampa ciao
    d.f(10);       // stampa 1000     } }
```


Overriding e riscrittura

Su oggetti di tipo `D` **non è più possibile invocare** `B.f(int)`.

È ancora possibile invocare `B.f(int)` **solo dall'interno della classe** `D` attraverso un campo predefinito `super` (analogo a `this`).

```
// File ParteTerza/Esempio24.java
class B {
  public int f(int i) { return i; }
}
class D extends B {
  public int f(int n) { return super.f(n*n); }
}
public class Esempio24 {
  public static void main(String[] args) {
    D d = new D();
    System.out.println(d.f(5));
  }
}
```

Riassunto overloading e overriding

	OVERLOADING	OVERRIDING
nome della funzione	uguale	uguale
tipo restituito	qualunque	uguale
numero e/o tipo argomenti	diverso	uguale
relazione con la funzione della classe base	coesiste con la funzione della classe base	cancella la funzione della classe base

Esercizio 12bis: overriding e compatibilità

```
// File Esercizio12bis.java
class B {
  protected int c;
  void stampa() { System.out.println("c: " + c); }
}

class D extends B {
  protected int e;
  void stampa() {
    super.stampa();
    System.out.println("e: " + e);
  }
}

public class Esercizio12bis {
  public static void main(String[] args) {
    B b  = new B();    b.stampa();
    B b2 = new D();    b2.stampa();
    D d  = new D();    d.stampa();
    D d2 = new B();    d2.stampa();
  }
}
```

Il programma contiene errori rilevabili dal compilatore?
Una volta eliminati tali errori, cosa stampa il programma?

Overriding di funzioni: late binding

Invocando `f(int)` su un oggetto di `D` viene invocata **sempre** `D.f(int)`, **indipendentemente** dal fatto che esso sia denotato attraverso un riferimento `d` di tipo `D` o un riferimento `b` di tipo `B`.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(int n) { ... }
}
// ...
D d = new D();
d.f(1);  // invoca D.f(int)
B b = d; // OK: classe derivata usata al posto di classe base
b.f(1);  // invoca di nuovo D.f(int)
```


Late binding (cont.)

Secondo il meccanismo del **late binding** la scelta di quale funzione invocare non viene effettuata durante la compilazione del programma, **ma durante l'esecuzione**.

```
public static void h (B b) { b.f(1); }
// ...
B bb = new B();
D d = new D();
h(d);    // INVOCA D.f(int)
h(bb);   // INVOCA B.f(int)
```

Il gestore run-time riconosce **automaticamente** il tipo dell'oggetto di invocazione:

`h(d)`: `d` è della classe `D` – viene invocata la funzione `f(int)` definita in `D`;

`h(bb)`: `bb` è della classe `B` – viene invocata la funzione `f(int)` definita in `B`.

Esercizio 13: cosa fa questo programma?

```
// File ParteTerza/Esercizio13.java
class B {
  protected int id;
  public B(int i) { id = i; }
  public boolean get() { return id < 0; }
}

class D extends B {
  protected char ch;
  public D(int i, char c) {
    super(i);
    ch = c;
  }
  public boolean get() { return ch != 'a'; }
}

public class Esercizio13 {
  public static void main(String[] args) {
    D d = new D(1,'b');
    B b = d;
    System.out.println(b.get());
    System.out.println(d.get());
  }
}
```


Overriding e livello d'accesso

Nel fare overriding di una funzione della classe base è possibile cambiare il livello di accesso alla funzione, **ma solo allargandolo**.

```
// File ParteTerza/Esempio20.java
class B {
  protected void f(int i) { System.out.println(i*i); }
  protected void g(int i) { System.out.println(i*i*i); }
}

class D extends B {
  public void f(int n) { System.out.println(n*n*n*n); }
  //  private void g(int n) {
  //                ^
  // Methods can't be overridden to be more private.
  // Method void g(int) is protected in class B.
  //    System.out.println(n*n*n*n*n); }
}
```


Impedire l'overriding: final

Qualora si voglia **bloccare l'overriding** di una funzione la si deve dichiarare `final`.

Anche una classe può essere dichiarata `final`, impedendo di derivare classi dalla stessa e rendendo implicitamente `final` tutte le funzioni della stessa.

```
// File ParteTerza/Esempio21.java
class B {
  public final void f(int i) { System.out.println(i*i); }
}
class D extends B {
// public void f(int n) { System.out.println(n*n*n*n); }
// Final methods can't be overriden. Method void f(int) is final in class B.
}
final class BB {}
// class DD extends BB {}
// Can't subclass final classes: class BB
```


Sovrascrittura dei campi dati

Se definiamo nella classe derivata una variabile con lo stesso nome e di diverso tipo di una variabile della classe base, allora:

- la variabile della classe base **esiste ancora** nella classe derivata, ma non può essere acceduta utilizzandone semplicemente il nome;
- si dice che la variabile della classe derivata **nasconde** la variabile della classe base;
- per accedere alla variabile della classe base è necessario utilizzare **un riferimento ad oggetto della classe base**.

Queste considerazioni valgono anche per campi che denotano **classi annidate**.

Sovrascrittura dei campi dati: esempio

```
// File ParteTerza/Esempio22.java
class B { int i; }
class D extends B {
  char i;
  void stampa() {
    System.out.println(i); System.out.println(super.i);
  }
}
public class Esempio22 {
  public static void main(String[] args) {
    D d = new D();
    d.i = 'f';
    ((B)d).i = 9;
    d.stampa();
  }
}
```


La classe Object

Implicitamente, **tutte le classi** (predefinite o definite da programma) sono derivate, direttamente o indirettamente, dalla classe `Object`.

Di conseguenza, tutti gli oggetti, qualunque sia la classe a cui appartengono, **sono anche implicitamente istanze** della classe predefinita `Object`.

Queste sono alcune funzioni della classe `Object` (delle prime due **sappiamo già fare** l'overriding):

- `public boolean equals(Object)`.
- `protected Object clone()`.
- `public String toString()`.

Analisi critica di equals()

```
class B {
  int x, y;
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      B b = (B)o;
      return (b.x == x) && (b.y == y);
    }
    else return false;
  }
}
```

- `equals()` "esiste implicitamente" per tutte le classi, in quanto è `public` in `Object`.
- Se la funzione avesse l'intestazione `boolean equals(B)`, faremmo overloading, **e non** overriding.
- `getClass()` è definita (`final`) in `Object`.
- Quando siamo sicuri che `o` sia di tipo `B`, allora possiamo fare il casting `B b = (B)o` (semanticamente corretto).

Esercizio 14: cosa fa questo programma?

```
// File Esercizio14.java

class B {
  private int x, y;
  public B(int a, int b) {
    x = a; y = b;
  }
  public boolean equals(B b) { // OVERLOADING, NON OVERRIDING
    if (b != null)
      return (b.x == x) && (b.y == y);
    else return false;
  }
}

public class Esercizio14 {
  static void stampaUguali(Object o1, Object o2) {
    if (o1.equals(o2))
      System.out.println("I DUE OGGETTI SONO UGUALI");
    else
      System.out.println("I DUE OGGETTI SONO DIVERSI");
  }

  public static void main(String[] args) {
    B b1 = new B(10,20);
```

```
    B b2 = new B(10,20);

    if (b1.equals(b2))
      System.out.println("I DUE OGGETTI SONO UGUALI");
    else
      System.out.println("I DUE OGGETTI SONO DIVERSI");

    stampaUguali(b1, b2);
  }
}
```

La classe Java Class

- Esiste implicitamente un oggetto di classe `Class` per ogni classe `B` (di libreria o definita da utente) del programma;
- Questo oggetto può essere denotato in due modi:
  - tramite la classe `B`, con l'espressione:

    ```
    ...  B.class ... // ha tipo Class
    ```

  - tramite oggetti della classe `B`, usando la funzione `getClass()` di `Object`, ad es.:

    ```
    B b1 = new B();
    ... b1.getClass() ... // ha tipo Class
    ```


La classe Java Class (cont.)

- L'oggetto di classe `Class` può essere usato per verificare se due oggetti appartengono alla stessa classe, ad es.:

  ```
  B b2 = new B();
  ... b1.getClass().equals(b2.getClass()) ... // vale true
  ```

- La classe `Class` ha una funzione dal significato particolare:

  ```
  boolean isInstance(Object)
  ```

  che restituisce `true` se e solo se il suo parametro attuale è un riferimento ad oggetto di una classe *compatibile per l'assegnazione* con la stessa classe dell'oggetto di invocazione.

La funzione isInstance()

- La funzione `isInstance()` può essere usata per verificare se un oggetto è istanza di una classe.

  ```
  ... B.class.isInstance(b2) ... // vale true
  ```

- Al riguardo, si ricorda che un oggetto di una classe `D` derivata da una classe `B` *è oggetto anche della classe* `B`.

  ```
  class D extends B ...

  D d1 = new D();
  ... B.class.isInstance(d1)) ... // vale true;
  ```

Esercizio 14bis: cosa fa questo programma?

```
// File Esercizio14bis.java

class B {}
class D extends B {}

public class Esercizio14bis {
  public static void main(String[] args) {
    B b1 = new B();
    D d1 = new D();
    System.out.println(B.class.isInstance(d1));
    System.out.println(D.class.isInstance(b1));
  }
}
```


Analisi critica di clone()

```
class B implements Cloneable {
  int x, y;
  public Object clone() {
    try {
      B t = (B)super.clone(); // Object.clone copia campo a campo
      return t;
    } catch (CloneNotSupportedException e) {
      // non puo' accadere, ma va comunque gestito
      throw new InternalError(e.toString());
    }
  }
}
```

- `clone()` "non esiste implicitamente" per tutte le classi, in quanto è `protected` in `Object`.
- La funzione **non può avere** l'intestazione `B clone()`: **è proibito** avere due funzioni omonime con diverso tipo di ritorno.
- Viene invocata la funzione `clone()` di `Object`, attraverso l'uso di `super.clone()`. Il casting è semanticamente corretto.

Stampa di oggetti e funzione toString()

La funzione `public String toString()` di `Object` associa una **stringa stampabile** all'oggetto di invocazione.

Se ne può fare overriding in modo opportuno nelle singole classi in modo da generare una **forma testuale** conveniente per gli oggetti della classe.

```
// File ParteTerza/Esempio23.java
class B {
  private int i;
  B(int x) { i = x; }
  public String toString() { return "i: " + i; }
}

public class Esempio23 {
  public static void main(String[] args) {
    B b = new B(5);
    System.out.println(b);
  }
}
```


Esercizio 15: overriding di toString()

Facendo riferimento alle classi `Punto` e `Segmento` viste in precedenza, ridefinire la funzione `toString()` per esse.

In particolare, vogliamo che un punto venga stampato in questo formato:

```
<1.0;2.0;4.0>
```

e che un segmento venga stampato in questo formato:

```
(<1.0;2.0;4.0>,<2.0;3.0;7.0>)
```

Uguaglianza profonda in classi derivate

Se desideriamo specializzare il comportamento dell'uguaglianza per una classe `D` derivata da `B`, si può fare overriding di `equals()` secondo il seguente schema semplificato:

```
public class D extends B {
  protected int z;
  public boolean equals(Object ogg) {
    if (super.equals(ogg)) {
      D d = (D)ogg;
      // test d'uguaglianza campi dati specifici di D
      return d.z == z;
    }
    else return false;
  }
}
```


Uguaglianza profonda in classi derivate (cont.)

- `D.equals()` delega a `super.equals()` (cioè `B.equals()`) alcuni controlli (**riuso**):
  - che l'oggetto a cui si riferisce il parametro attuale sia stato allocato;
  - che l'oggetto di invocazione ed il parametro attuale siano della stessa classe;
  - che l'oggetto di invocazione ed il parametro attuale coincidano nei campi della classe base.
- `D.equals()` si occupa solamente del controllo dei campi dati specifici di `D` (cioè di `z`).

Esercizio 16: : cosa fa questo programma?

```
class B { // ... la solita

class D extends B {
  public D(int a, int b, int c) {//...
  protected int z;
  public boolean equals(Object ogg) {
    if (super.equals(ogg)) {
      D d = (D)ogg;
      return d.z == z;
    }
    else return false;
  }
}
// ...
    D d = new D(4,5,6);
    E e = new E(4,5,6);

    if (d.equals(e))
      System.out.println("I DUE OGGETTI SONO UGUALI");
    else
      System.out.println("I DUE OGGETTI SONO DIVERSI");
```

```
class E extends B {
  public E(int a, int b, int c){//...
  protected int z;
  public boolean equals(Object ogg) {
    if (super.equals(ogg)) {
      E e = (E)ogg;
      return e.z == z;
    }
    else return false;
  }
}
```


Copia profonda in classi derivate

Quando una classe `B` ha dichiarato pubblica `clone()`, tutte le classi da essa derivate (direttamente o indirettamente) **devono** supportare la clonazione (non è più possibile "nascondere" `clone()`).

Per supportarla correttamente le classi derivate devono fare overriding di `clone()` secondo lo schema seguente.

```
public class D extends B {
  // ...
  public Object clone() {
    D d = (D)super.clone();
      // codice eventuale per campi di D che richiedono copie speciali
    return d;
  }
  // ...
}
```

Copia profonda in classi derivate (cont.)

- Una classe derivata da una classe che implementa l'interfaccia `Cloneable` (o qualsiasi altra interfaccia), implementa anch'essa tale interfaccia.

- La chiamata a `super.clone()` è **indispensabile**.

  Essa invoca la funzione `clone()` della classe base, la quale a sua volta chiama `super.clone()`, e così via fino ad arrivare a `clone()` della classe `Object` che è l'unica funzione in grado di creare (allocandolo dinamicamente) l'oggetto clone.

  Tutte le altre invocazioni di `clone()` lungo la catena di ereditarietà si occupano in modo opportuno di operare sui campi a cui hanno accesso.

  Si noti che per copiare correttamente gli eventuali campi privati è indispensabile operare sugli stessi attraverso la classe che li definisce.

Copia profonda in classi derivate: esempio

```
class B implements Cloneable {
  protected int x, y;
  public Object clone() { // ...
  // ...
}
```

```
class C implements Cloneable {
  private int w;
  public Object clone() { // ...
  // ...
}
```

```
class D extends B {
  protected int z;            // TIPO BASE
  protected C c;              // RIFERIMENTO A OGGETTO
  public Object clone() {
    D d = (D)super.clone();   // COPIA SUPERFICIALE: OK PER z, NON PER c
    d.c = (C)c.clone();       // NECESSARIO PER COPIA PROFONDA DI c
    return d;
  }
  // ...
}
```


Stampa in classi derivate

Nel fare overriding di `toString()` per una classe derivata è possibile riusare la funzione `toString()` della classe base.

```
class B implements {
  protected int x, y;
  public String toString() { // ...
  // ...
}
```

```
class D extends B {
  protected int z;
  public String toString() {
    return super.toString() + // ...
  }
  // ...
}
```


Esercizio 17: funzioni speciali in classi derivate

Scrivere una classe `SegmentoOrientato` derivata dalla classe `Segmento`, che contiene anche l'informazione sull'orientazione del segmento (dal punto di inizio a quello di fine, o viceversa).

Per questa classe vanno previsti, oltre al costruttore, l'overriding delle funzioni speciali `equals()`, `clone()` e `toString()`, sfruttando opportunamente quelle della classe base `Segmento`.

Per quanto riguarda la funzione `toString()`, si vuole che un segmento orientato venga stampato in questo formato:

```
(<1.0;2.0;4.0>,<2.0;3.0;7.0>)--->
```

se l'orientamento è dall'inizio alla fine, e nel seguente formato:

```
(<1.0;2.0;4.0>,<2.0;3.0;7.0>)<---
```

nel caso contrario.

Classi astratte

Le classi astratte sono classi particolari, nelle quali una o più funzioni possono essere solo **dichiarate** (cioè si descrive la segnatura), ma non **definite** (cioè non si specificano le istruzioni).

Esempio:

Ha certamente senso associare alla classe `Persona` una funzione che calcola la sua aliquota fiscale, ma il vero e proprio calcolo per una istanza della classe `Persona` **dipende** dalla sottoclasse di `Persona` (ad esempio: straniero, pensionato, studente, impiegato, ecc.) a cui l'istanza appartiene.

Vogliamo poter definire la classe `Persona`, magari con un insieme di campi e funzioni normali, anche se non possiamo scrivere il codice della funzione `Aliquota()`.

Classi astratte: esempio

La soluzione è definire la classe `Persona` come **classe astratta**, con la funzione `Aliquota()` astratta, e definire poi le sottoclassi di `Persona` come classi non astratte, in cui definire la funzione `Aliquota()` con il relativo codice:

```
abstract class Persona {
  abstract public int Aliquota(); // Questa e' una DICHIARAZIONE
                                  // (senza codice)
  private int eta;
  public int Eta() { return eta; }
}

class Studente extends Persona {
  public int Aliquota() { ... } // Questa e' una DEFINIZIONE
}

public class Professore extends Persona {
  public int Aliquota() { ... } // Questa e' una DEFINIZIONE
}
```


Quando una classe va definita astratta

Una classe `A` si definirà come astratta quando:

- esiste una funzione che ha senso associare ad essa, ma il cui codice non può essere specificato a livello di `A`, mentre può essere specificato a livello delle sottoclassi di `A`; si dice che tale funzione è astratta in `A`;
- non ha senso pensare a oggetti che siano istanze di `A` **senza essere istanze anche di una sottoclasse (eventualmente indiretta) di** `A`.

Anche se spesso si dice che una classe astratta `A` non ha istanze, ciò con è corretto: la classe astratta `A` non ha istanze dirette, ma ha come istanze tutti gli oggetti che sono istanze di sottoclassi di `A` non astratte.

Si noti che la classe astratta può avere funzioni non astratte e campi dati.

Uso di classi astratte

Se `A` è una classe astratta, allora:

- **Non possiamo** creare direttamente oggetti che sono istanze di `A`. Non esistono istanze dirette di `A`: gli oggetti che sono istanze di `A` lo sono indirettamente.
- **Possiamo:**
  - definire variabili o campi di altri classi (ovvero, riferimenti) di tipo `A` (durante l'esecuzione, conterranno indirizzi di oggetti di classi non astratte che sono sottoclassi di `A`),
  - usare normalmente i riferimenti (tranne che per creare nuovi oggetti), ad esempio: definire funzioni che prendono come argomento un riferimento di tipo `A`, restituire riferimenti di tipo `A`, ecc.

Vantaggi delle classi astratte

Se non ci fosse la possibilità di definire la classe `Persona` come classe astratta, dovremmo prevedere un meccanismo (per esempio un campo di tipo `String`) per distinguere istanze di `Studente` da istanze di `Professore`, e definire nella classe `Persona` la funzione `Aliquota()` così

```
class Persona {
  private String tipo;
  public int Aliquota() {
      if (tipo.equals("Studente"))
           // codice per il calcolo dell'aliquota per Studente
      else if (tipo.equals("Professore"))
           // codice per il calcolo dell'aliquota per Professore
   }
  // ....
}
```

All'aggiunta di una sottoclasse di `Persona`, si dovrebbe riscrivere e ricompilare la classe `Persona` stessa. **Riuso ed estendibilità sarebbero compromessi!**

Vantaggi delle classi astratte (cont.)

Supponiamo di dovere scrivere una funzione esterna alla classe `Persona` che, data una persona (sia essa uno studente, un professore, o altro), verifica se è tartassata dal fisco (cioè se la sua aliquota è maggiore del 50 per cento).

Se ho definito `Persona` come classe astratta posso semplicemente fare così:

```
// ....
static public boolean Tartassata(Persona p) {
     return p.Aliquota() > 50;
}
```

È importante notare che, quando la funzione verrà attivata, verrà passato come parametro attuale un riferimento ad un oggetto di una classe non astratta, in cui quindi la funzione `Aliquota()` è definita con il codice. Il late binding farà il suo gioco, e chiamerà la **funzione giusta**, cioè la funzione definita nella classe più specifica non astratta di cui l'oggetto passato è istanza.

Esercizio 18

Si definisca una classe per rappresentare soggetti fiscali. Ogni soggetto fiscale ha un nome, e di ogni soggetto fiscale deve essere possibile calcolare l'anzianità, tenendo però presente che l'anzianità si calcola in modo diverso a seconda della categoria (impiegato, pensionato o straniero) a cui appartiene il soggetto fiscale. In particolare:

- se il soggetto è un impiegato, allora l'anzianità si calcola sottraendo all'anno corrente l'anno di assunzione;
- se il soggetto è un pensionato, allora l'anzianità si calcola sottraendo all'anno corrente l'anno di pensionamento;
- se il soggetto è uno straniero, allora l'anzianità si calcola sottraendo all'anno corrente l'anno di ingresso nel paese.

Interfacce

Una interfaccia è un'astrazione per un insieme di funzioni pubbliche delle quali si definisce solo la segnatura, e non le istruzioni. Un'interfaccia viene poi implementata da una o più classi (anche astratte). Una classe che implementa un'interfaccia deve definire o dichiarare tutte le funzioni della interfaccia.

Dal punto di vista sintattico, un'interfaccia è costituita da un insieme di dichiarazioni di funzioni pubbliche (**no campi dati**, a meno che non sia `final`), la cui definizione **è necessariamente lasciata alle classi che la implementano**. Possiamo quindi pensare ad una interfaccia come ad una dichiarazione di un tipo di dato (inteso come un insieme di operatori) di cui non vogliamo specificare l'implementazione, ma che comunque può essere utilizzato da moduli software, indipendentemente appunto dall'implementazione.

Esempio: interfaccia `I` con una sola funzione `g()`

```
public interface  I {
  void g(); // implicitamente public;  e' una DICHIARAZIONE: notare ';'
}
```

Cosa si fa con un'interfaccia

Se `I` è un'interfaccia, allora **possiamo**:

- definire una o più classi che **implementano** `I`, cioè che definiscono tutte le funzioni dichiarate in `I`
- definire variabili e campi di tipo `I` (durante l'esecuzione, conterranno indirizzi di oggetti di classi che implementano `I`),
- usare i riferimenti di tipo `I`, sapendo che in esecuzione essi conterranno indirizzi di oggetti (quindi possiamo definire funzioni che prendono come argomento un riferimento di tipo `I`, restituire riferimenti di tipo `I`, ecc.);

mentre **non possiamo**:

- creare oggetti di tipo `I`, cioè non possiamo eseguire `new I()`, perchè non esistono oggetti di tipo `I`, ma esistono solo riferimenti di tipo `I`.

Utilità delle interfacce

Le funzioni di un'interfaccia costituiscono un modulo software `S` che:

- può essere utilizzato da un modulo esterno T (ad esempio una funzione `t()` che si aspetta come parametro un riferimento di tipo `S`), **indipendentemente** da come le funzioni di `S` sono implementate; in altre parole, non è necessario avere deciso l'implementazione delle funzioni di `S` per progettare e scrivere altri moduli che usano `S`;
- può essere implementato in modi alternativi e diversi tra loro (nel senso che più classi possono implementare le funzioni di `S`, anche in modo molto diverso tra loro);
- ovviamente, però, al momento di attivare un modulo `t()` che ha un argomento tipo `S`, occorre passare a `t()`, in corrispondenza di `S`, un oggetto di una classe che implementa `S`.

Tutto ciò aumenta la possibilità di **riuso**.

Esempio di interfaccia e di funzione cliente

Vogliamo definire una interfaccia `Confrontabile` che offra una operazione che verifica se un oggetto è **maggiore** di un altro, ed una operazione che verifica se un oggetto è **paritetico** ad un altro. Si noti che **nulla si dice** rispetto al criterio che stabilisce se un oggetto è maggiore di o paritetico ad un altro.

Si vuole scrivere poi una funzione che, dati tre riferimenti a `Confrontabile`, restituisca il maggiore tra i tre (o più precisamente un *massimale*, ovvero uno qualunque che non abbia tra gli altri due uno maggiore di esso).

Notiamo che, denotando con gli operatori binari infissi '$>$' e '$=$' le relazioni "maggiore" e "paritetico" (rispettivamente), x_1 è massimale in $\{x_1, x_2, x_3\}$ se e solo se:

$$(x_1 > x_2 \ \vee \ x_1 = x_2) \quad \wedge \quad (x_1 > x_3 \ \vee \ x_1 = x_3)$$

Esempio di interfaccia e di f. cliente (cont.)

```
// File Esempio29.java
interface  Confrontabile {
  boolean Maggiore(Confrontabile x);
  boolean Paritetico(Confrontabile x);
}

class Utilita {
  static public Confrontabile MaggioreTraTre(Confrontabile x1,
                                             Confrontabile x2,
                                             Confrontabile x3) {
    if ((x1.Maggiore(x2) || x1.Paritetico(x2)) &&
        (x1.Maggiore(x3) || x1.Paritetico(x3)))
      return x1;
    else if ((x2.Maggiore(x1) || x2.Paritetico(x1)) &&
             (x2.Maggiore(x3) || x1.Paritetico(x3)))
      return x2;
    else return x3;  }  }
```

Implementazione di un'interfaccia

Definiamo due classi che implementano l'interfaccia `Confrontabile`:

1. Una di queste è la classe `Edificio` (per la quale il confronto concerne l'altezza).

2. L'altra è una classe astratta `Persona` (per la quale il confronto concerne l'aliquota).

Implementazione di un'interfaccia (cont.)

```
// File Esempio30.java

class Edificio implements Confrontabile {
  protected int altezza;
  public boolean Maggiore(Confrontabile e) {
    if (e != null && getClass().equals(e.getClass()))
      return altezza > ((Edificio)e).altezza;
    else return false;
  }
  public boolean Paritetico(Confrontabile e) {
    if (e != null && getClass().equals(e.getClass()))
      return altezza == ((Edificio)e).altezza;
    else return false;
  }
}
```



```
abstract class Persona implements Confrontabile {
  protected int eta;
  abstract public int Aliquota();
  public int Eta() { return eta; }
  public boolean Maggiore(Confrontabile p) {
    if (p != null && Persona.class.isInstance(p))
      return Aliquota() > ((Persona)p).Aliquota();
    else return false;
  }
  public boolean Paritetico(Confrontabile p) {
    if (p != null && Persona.class.isInstance(p))
      return Aliquota() == ((Persona)p).Aliquota();
    else return false;
  }
}
```

Commenti sull'implementazione

Notiamo che nelle classi `Edificio` e `Persona` abbiamo usato due criteri differenti per stabilire se possiamo effettuare i confronti fra due oggetti tramite le funzioni `Maggiore()` e `Paritetico()`:

- per la classe (non astratta) `Edificio`, verifichiamo se i due oggetti siano della stessa classe (`Edificio` o derivata da essa);

- per la classe (astratta) `Persona`, verifichiamo se i due oggetti siano entrambi derivati dalla classe `Persona`.

  Ciò permette di effettuare il confronto anche fra oggetti di classi differenti, purché entrambe derivate da `Persona`.

Esempio di uso di interfaccia

A questo punto possiamo chiamare la funzione `MaggioreTraTre()`:

- sia su Persone (cioè passandole tre oggetti della classe `Persona`),
- sia su Edifici (cioè passandole tre oggetti della classe `Edificio`).

Esempio di uso di interfaccia (cont.)

```
// File Esempio31.java
class Studente extends Persona {
  public int Aliquota() { return 25; } }
class Professore extends Persona {
  public int Aliquota() { return 50; } }
class Esempio31 {
  public static void main(String[] args) {
    Studente s = new Studente();
    Professore p = new Professore();
    Professore q = new Professore();
    Edificio e1 = new Edificio();
    Edificio e2 = new Edificio();
    Edificio e3 = new Edificio();
    Persona pp = (Persona)Utilita.MaggioreTraTre(s,p,q);
    Edificio ee = (Edificio)Utilita.MaggioreTraTre(e1,e2,e3);
  }
}
```


Esercizio 19

Arricchire la classe `Utilita` con:

- una funzione `Massimale()` che, ricevuto come argomento un vettore di riferimenti a `Conparabile`, restituisca un elemento massimale fra quelli del vettore;
- una funzione `QuantiMassimali()` che, ricevuto come argomento un vettore di riferimenti a `Confrontabile`, restituisca un intero che corrisponde al numero di elementi massimali fra quelli del vettore.

Interfacce e classi che le implementano

Una classe può implementare anche più di una interfaccia (implementazione multipla), come mostrato da questo esempio:

```
public interface  I {
  void g();
}

public interface J {
  void h();
}

class C implements I,J {
  void g() { ... }
  void h() { ... }
}
```

Esempio di implementazione multipla

```
// File ParteTerza/Esempio27.java
interface  I  { void g(); }
interface  I2 { void h(); }
class B {
  void f() { System.out.println("bye!"); }
}
class C extends B implements I, I2 {
  public void g() { System.out.println("ciao!"); }
  public void h() { System.out.println("hello!"); }
}
public class Esempio27 {
  public static void main(String[] args) {
    C c = new C();
    c.g();
    c.h();
    c.f();
  }
}
```


Interfacce ed ereditarietà

L'ereditarietà si può stabilire anche tra interfacce, nel senso che una interfaccia si può definire derivata da un'altra. Se una interfaccia `J` è derivata da una interfaccia `I`, allora tutte le funzioni dichiarate in `I` sono implicitamente dichiarate anche in `J`.

Ne segue che una classe che implementa `J` deve anche definire tutte le funzioni di `I`.

```
public interface  I {
  void g();
}
public interface J extends I {
  void h();
}
class C implements J {
  void g() { ... }
  void h() { ... }
}
```


Interfacce ed ereditarietà multipla

Limitatamente alle interfacce, Java supporta l'**ereditarietà multipla**: una interfaccia può essere derivata da un qualunque numero di interfacce.

```
public interface  I {
  void g();
}
public interface J {
  void h();
}
public interface M extends I, J {
  void k();
}
class C implements M {
  void g() { ... }
  void h() { ... }
  void k() { ... }
}
```


Differenza tra interfacce e classi astratte

Interfacce e classi astratte hanno qualche similarità. Ad esempio: entrambe hanno funzioni dichiarate e non definite; non esistono istanze di interfacce, e non esistono istanze dirette di classi astratte.

Si tenga però presente che:

- Una classe astratta è comunque una classe, ed è quindi un'astrazione di un insieme di oggetti (le sue istanze). Ad esempio, la classe `Persona` è un'astrazione per l'unione delle istanze di `Studente` e `Professore`.

- Una interfaccia è un'astrazione di un insieme di funzioni. Ad esempio, è difficile pensare concettualmente ad una classe `Confrontabile` che sia superclasse di `Persona` ed `Edificio`, e che quindi metta insieme le istanze di `Persona` ed `Edificio`, solo perché ha senso confrontare tra loro (con le funzioni `Maggiore()` e `Paritetico()`) sia le persone sia gli edifici.

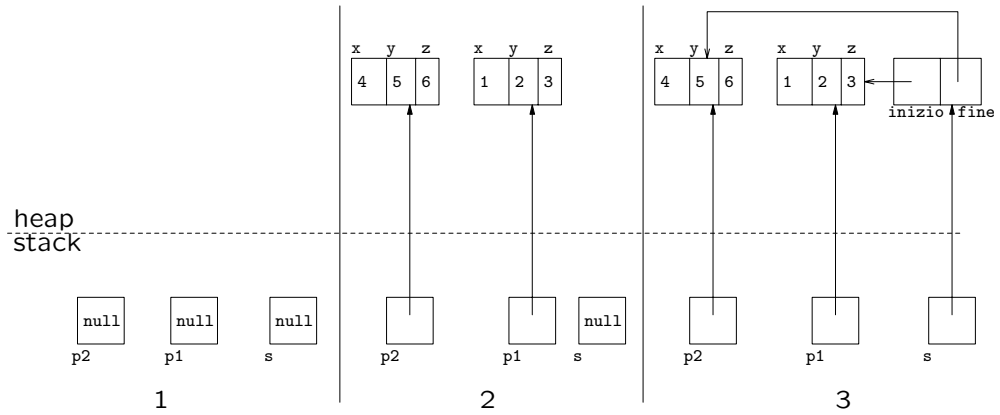
Riassunto classi, classi astratte, interfacce

	`class`	`abstract class`	`interface`
Riferimenti	SI	SI	SI
Oggetti	SI	SI (indirettamente)	NO
Campi dati	SI	SI	NO
Funzioni solo dichiarate	NO	SI	SI
Funzioni definite	SI	SI	NO
`extends (abstract) class`	0 o 1	0 o 1	0
`implements interface`	≥ 0	≥ 0	0
`extends interface`	0	0	≥ 0

Soluzioni degli esercizi della Parte 3


```
// File ParteTerza/Esercizio1.java
class Punto {
  float x, y, z;
}
class Segmento {
  Punto inizio, fine;
}
public class Esercizio1 {
  public static void main(String[] args) {
    Punto p1, p2;
    Segmento s;                       // 1
    p1 = new Punto();
    p2 = new Punto();
    p1.x = 1; p1.y = 2; p1.z = 4;
    p2.x = 2; p2.y = 3; p2.z = 7;     // 2
    s = new Segmento();
    s.inizio = p1; s.fine = p2;       // 3
  }
}
```


Esercizio 1: evoluzione dello stato della memoria



```
// File ParteTerza/Esercizio2Punto.java
// Esercizio: uguaglianza profonda

class Punto {
  float x, y, z;
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      Punto p = (Punto)o;
      return (p.x == x) && (p.y == y) && (p.z == x);
    }
    else return false;
  }
}

public class Esercizio2Punto {
  public static void main(String[] args) {
    Punto p1 = new Punto(), p2 = new Punto();
    p1.x = p1.y = p1.z = p2.x = p2.y = p2.z = 4;
```



```
    if (p1.equals(p2))
      System.out.println("Uguali!");
    else
      System.out.println("Diversi!");
  }
}
```

```
// File ParteTerza/Esercizio2Segmento.java
// Esercizio: uguaglianza profonda

class Segmento {
  Punto inizio, fine;
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      Segmento s = (Segmento)o;
      return s.inizio.equals(inizio) && s.fine.equals(fine);
      // s.inizio == inizio && s.fine == fine SAREBBE SBAGLIATO!
    }
    else return false;
  }
}

public class Esercizio2Segmento {
  public static void main(String[] args) {
    Segmento s1 = new Segmento(), s2 = new Segmento();
```



```
    s1.inizio = new Punto();
    s1.fine = new Punto();
    s2.inizio = new Punto();
    s2.fine = new Punto();
    s1.inizio.x = s1.inizio.y = s1.inizio.z =
      s2.inizio.x = s2.inizio.y = s2.inizio.z = 4;
    s1.fine.x = s1.fine.y = s1.fine.z =
      s2.fine.x = s2.fine.y = s2.fine.z = 10;
    if (s1.equals(s2))
      System.out.println("Uguali!");
    else
      System.out.println("Diversi!");
  }
}
```

```
// File ParteTerza/Esercizio2Valuta.java
// Esercizio: uguaglianza profonda

class Valuta {
  int unita, centesimi;
  String nome;
  public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
      Valuta v = (Valuta)o;
      return (v.unita == unita) && (v.centesimi == centesimi) &&
        (v.nome.equals(nome));
      // (v.nome == nome) SAREBBE SBAGLIATO!
    }
    else return false;
  }
}

public class Esercizio2Valuta {
```



```
  public static void main(String[] args) {
    Valuta v1 = new Valuta(), v2 = new Valuta();
    v1.unita = 5; v1.centesimi = 20; v1.nome = new String("euro");
    v2.unita = 5; v2.centesimi = 20; v2.nome = new String("euro");
    if (v1.equals(v2))
      System.out.println("Uguali!");
    else
      System.out.println("Diversi!");
  }
}
```

```
// File ParteTerza/Esercizio3Punto.java
// Esercizio: copia profonda

class Punto implements Cloneable {
  float x, y, z;
  public Object clone() {
    try {
      Punto p = (Punto)super.clone();
      return p;
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e.toString());
    }
  }
}

public class Esercizio3Punto {
  public static void main(String[] args) {
    Punto p1 = new Punto(), p2;
```



```
    p1.x = 1; p1.y = 2; p1.z = 3;
    p2 = (Punto)p1.clone();
    System.out.println("p2.x: " + p2.x + ", p2.y: " + p2.y +
                       ", p2.z: " + p2.z);
  }
}
```

```
// File ParteTerza/Esercizio3Segmento.java
// Esercizio: copia profonda

class Segmento implements Cloneable {
  Punto inizio, fine;
  public Object clone() {
    try {
      Segmento s = (Segmento)super.clone();
      s.inizio = (Punto)inizio.clone(); // NECESSARIO
      s.fine = (Punto)fine.clone();     // NECESSARIO
      return s;
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e.toString());
    }
  }
}

public class Esercizio3Segmento {
```



```
  public static void main(String[] args) {
    Segmento s1 = new Segmento(), s2;
    s1.inizio = new Punto();
    s1.fine = new Punto();
    s1.inizio.x = s1.inizio.y = s1.inizio.z = 4;
    s1.fine.x = s1.fine.y = s1.fine.z = 10;
    s2 = (Segmento)s1.clone();
    System.out.println("s2.inizio.x: " + s2.inizio.x + ", s2.inizio.y: " +
                       s2.inizio.y + ", s2.inizio.z: " + s2.inizio.z);
    System.out.println("s2.fine.x: " + s2.fine.x + ", s2.fine.y: " +
                       s2.fine.y + ", s2.fine.z: " + s2.fine.z);
  }
}
```

```
// File ParteTerza/Esercizio3Valuta.java
// Esercizio: copia profonda

class Valuta implements Cloneable {
  int unita, centesimi;
  String nome;
  public Object clone() {
    try {
      Valuta v = (Valuta)super.clone();
      v.nome = new String(nome); // NECESSARIO
      return v;
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e.toString());
    }
  }
}

public class Esercizio3Valuta {
```



```
  public static void main(String[] args) {
    Valuta v1 = new Valuta(), v2;
    v1.unita = 5; v1.centesimi = 20; v1.nome = new String("euro");
    v2 = (Valuta)v1.clone();
    System.out.println("v2.unita: " + v2.unita + ", v2.centesimi: " +
                       v2.centesimi + ", v2.nome: " + v2.nome);
  }
}
```

```
// File ParteTerza/Esercizio4.java
// Esercizio: passaggio e restituzione di argomenti

class Punto implements Cloneable {
  float x, y, z;
  public Object clone() {
    try {
      Punto p = (Punto)super.clone();
      return p;
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e.toString());
    }
  }
}

class Segmento implements Cloneable {
  Punto inizio, fine;
  public Object clone() {
```



```
    try {
      Segmento s = (Segmento)super.clone();
      s.inizio = (Punto)inizio.clone();
      s.fine = (Punto)fine.clone();
      return s;
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e.toString());
    }
  }
}

public class Esercizio4 {
  static double lunghezza(Segmento s) {
    return Math.sqrt( Math.pow((s.fine.x - s.inizio.x),2) +
                      Math.pow((s.fine.y - s.inizio.y),2) +
                      Math.pow((s.fine.z - s.inizio.z),2) ) ;
  }
```

```
  static void inizioInOrigine(Segmento s) {
    s.inizio.x = s.inizio.y =  s.inizio.z = 0;
  }

  static Punto mediano(Segmento s) {
    Punto med = new Punto();
    med.x = (s.fine.x + s.inizio.x) / 2;
    med.y = (s.fine.y + s.inizio.y) / 2;
    med.z = (s.fine.z + s.inizio.z) / 2;
    return med;
  }

  static Segmento meta(Segmento s) {
    Segmento met = (Segmento)s.clone();
    met.fine = mediano(s);
    return met;
  }
```

```
  public static void main(String[] args) {
    Segmento s1 = new Segmento();
    s1.inizio = new Punto();
    s1.fine = new Punto();
    s1.inizio.x = s1.inizio.y = s1.inizio.z = 4;
    s1.fine.x = s1.fine.y = s1.fine.z = 10;
    System.out.println("Lunghezza di s1: " + lunghezza(s1));
    inizioInOrigine(s1);
    System.out.println("s1.inizio.x: " + s1.inizio.x + ", s1.inizio.y: " +
                       s1.inizio.y + ", s1.inizio.z: " + s1.inizio.z);
    Punto m = mediano(s1);
    System.out.println("m.x: " + m.x + ", m.y: " + m.y + ", m.z: " + m.z);
    Segmento s2 = meta(s1);
    System.out.println("s2.inizio.x: " + s2.inizio.x + ", s2.inizio.y: " +
                       s2.inizio.y + ", s2.inizio.z: " + s2.inizio.z);
    System.out.println("s2.fine.x: " + s2.fine.x + ", s2.fine.y: " +
                       s2.fine.y + ", s2.fine.z: " + s2.fine.z);
  }
}
```

Soluzione esercizio 5

- Il programma stampa:

  ```
  p: 5 s: 7 o_p: 50 o_s: 70
  ```

- Poiché in Java il passaggio di parametri è sempre per valore, non vi è alcun effetto sui parametri attuali nelle due chiamate.

- Possiamo dire che le due funzioni `mistero1()` e `mistero2()` sono completamente inutili, in quanto:
  - non restituiscono alcun valore;
  - non alterano, tramite side-effect, i loro argomenti.


```
// File ParteTerza/Esercizio6.java
// Esercizio: side-effect
class MioIntero {
  int dato;
}
class Esercizio6 {
  static void scambia (MioIntero i, MioIntero j) {
    int temp = i.dato;
    i.dato = j.dato;
    j.dato = temp;
  }
  public static void main (String[] args) {
    MioIntero p = new MioIntero(), s = new MioIntero();
    p.dato = 5; s.dato = -4;
    System.out.println(p.dato + " " + s.dato);
    scambia(p,s);
    System.out.println(p.dato + " " + s.dato);
  }
}
```



```
// File ParteTerza/Esercizio7.java
class Punto {
  Punto(float a, float b, float c) { x = a; y = b; z = c; }
  Punto() { } // punto origine degli assi
  float x, y, z;
}
class Segmento {
  Segmento(Punto i, Punto f) { inizio = i; fine = f; }
  Punto inizio, fine;
}
public class Esercizio7 {
  public static void main(String[] args) {
    Punto p1 = new Punto(1,2,4);
    Punto p2 = new Punto(2,3,7);
    Segmento s = new Segmento(p1,p2);
  }
}
```



```
// File Esercizio8/Riga1.java
// Verifica regole di visibilita': riga 1 della tabella
package Esercizio8;

public class Riga1 {
  public int a_pub;
  private int a_pri;
  int a_nonqual;
  protected int a_pro;

  void b() {
    int x;
    x = a_pub;     // SI
    x = a_pri;     // SI
    x = a_nonqual; // SI
    x = a_pro;     // SI
  }
}
```

```
// File Esercizio8/Riga2.java
// Verifica regole di visibilita': riga 2 della tabella
package Esercizio8;

public class Riga2 {
  void b() {
    Riga1 r1 = new Riga1();
    int x;
    x = r1.a_pub;     // SI
 // x = r1.a_pri;     // NO
    x = r1.a_nonqual; // SI
    x = r1.a_pro;     // SI
  }
}

// File Esercizio8/Riga3.java
// Verifica regole di visibilita': riga 3 della tabella
```

```
 // x = r1.a_pri;     // NO
 // x = r1.a_nonqual; // NO
 // x = r1.a_pro;     // NO
  }
}
```

```
public class Riga3 extends Esercizio8.Riga1 {
  void b1() {
    int x;

    x = a_pub;     // SI
 // x = a_pri;     // NO
 // x = a_nonqual; // NO
    x = a_pro;     // SI
  }
}

// File Esercizio8/Riga4.java
// Verifica regole di visibilita': riga 4 della tabella
public class Riga4 {
  void b1() {
    Esercizio8.Riga1 r1 = new Esercizio8.Riga1();
    int x;
    x = r1.a_pub;     // SI
```

```
// File ParteTerza/Esercizio9.java

class Punto { float x, y, z; }

class Segmento {
  Punto inizio, fine;
  void stampa() {
    System.out.println("inizio.x: " + inizio.x + ", inizio.y: " +
                       inizio.y + ", inizio.z: " + inizio.z);
    System.out.println("fine.x: " + fine.x + ", fine.y: " +
                       fine.y + ", fine.z: " + fine.z);
  }
}

class SegmentoOrientato extends Segmento {
  boolean da_inizio_a_fine;
}
```

```
public class Esercizio9 {
  static double lunghezza(Segmento s) {
    return Math.sqrt( Math.pow((s.fine.x - s.inizio.x),2) +
                      Math.pow((s.fine.y - s.inizio.y),2) +
                      Math.pow((s.fine.z - s.inizio.z),2) ) ;
  }
  public static void main(String[] args) {
    SegmentoOrientato s_o = new SegmentoOrientato();
    s_o.inizio = new Punto(); s_o.fine = new Punto();
    s_o.inizio.x = s_o.inizio.y = s_o.inizio.z = 4;
    s_o.fine.x = s_o.fine.y = s_o.fine.z = 10;
    s_o.stampa();   // OK: classe derivata al posto di classe base
    System.out.println("Lunghezza di s_o: " + lunghezza(s_o));
                    // OK: classe derivata al posto di classe base
  }
}
```

```
// File ParteTerza/Esercizio10.java

class Punto { float x, y, z; }

class PuntoColorato extends Punto { char colore; }

class PuntoConMassa extends Punto { float massa; }

class PuntoConMassaEVelocita extends PuntoConMassa { float velocita; }


class Segmento { Punto inizio, fine; }

class SegmentoOrientato extends Segmento {
  boolean da_inizio_a_fine;
}
```



```
// File ParteTerza/Esercizio11.java
class B { }
class D extends B { int x_d; }
public class Esercizio11 {
  static void f(B bb) {
    ((D)bb).x_d = 2000;
    System.out.println(((D)bb).x_d);
  }
  public static void main(String[] args) {
    B b = new B();
    D d = new D(); d.x_d = 1000;
    f(d);  // OK
    f(b);  // ERRORE SEMANTICO
    // java.lang.ClassCastException: B
    // at Esercizio11.f(Compiled Code)
    // at Esercizio11.main(Compiled Code)
  }
}
```



```
// File ParteTerza/Esercizio12.java

class Punto {
  protected float x, y, z;
  public Punto() { } // punto origine degli assi
  public Punto(int a, int b, int c) {
    x = a; y = b; z = c;
  }
}

class PuntoColorato extends Punto {
  protected char colore;
  public PuntoColorato(int a, int b, int c, char col) {
    super(a,b,c);
    colore = col;
  }
}
```

```
class PuntoConMassa extends Punto {
  protected float massa;
  public PuntoConMassa(float m) {
    // INVOCA IL COSTR. SENZA ARGOMENTI DI Punto
    massa = m;
  }
}

class PuntoConMassaEVelocita extends PuntoConMassa {
  protected float velocita;
  public PuntoConMassaEVelocita(float m, float v) {
    super(m);
    velocita = v;
  }
}
```

Soluzione esercizio 13

- Il programma stampa:

```
true
true
```

- Infatti, esiste **un solo oggetto** di classe `D`. Tale oggetto viene denotato attraverso:
  - un riferimento `d` di tipo `D`, o
  - un riferimento `b` di tipo `B`.
- Per il meccanismo del late binding, se invochiamo la funzione `get()` su questo oggetto, viene sempre selezionata la funzione `D.get()`, **indipendentemente** dall'avere usato il riferimento `b` o il riferimento `d`.

Soluzione esercizio 14

- Il programma stampa:

```
I DUE OGGETTI SONO UGUALI
I DUE OGGETTI SONO DIVERSI
```

- L'uguaglianza tra gli oggetti può essere verificata attraverso la funzione `equals(B)` di `B`, che verifica l'uguaglianza dei due oggetti denotati da `b1` e `b2`, che risultano effettivamente uguali.
- Tuttavia invocando la funzione `stampaUguali()` i due oggetti risultano diversi.

  Questo effetto è dovuto al fatto che essendo i parametri formali di `stampaUguali()` di tipo `Object`, su di essi viene invocata `equals(Object)`, della quale **non si è fatto overriding** in `B`.

Soluzione esercizio 14bis

- Il programma effettua le seguenti stampe:

```
true
false
```

- Infatti:
  - `d1` è un'istanza di `B` (`D` è compatibile per l'assegnazione con `B`);
  - `b1` **non** è un'istanza di `D` (`B` **non** è compatibile per l'assegnazione con `D`).

```
// File ParteTerza/Esercizio15.java

class Punto {
  protected float x, y, z;
  public Punto() { } // punto origine degli assi
  public Punto(int a, int b, int c) {
    x = a; y = b; z = c;
  }
  public String toString() {
    return "<" + x + ";" + y + ";" + z + ">";
  }
}

class Segmento {
  protected Punto inizio, fine;
  public Segmento(Punto i, Punto f) {
    inizio = i; fine = f;
  }
```



```
  public String toString() {
    return "(" + inizio + "," + fine + ")";
  }
}

public class Esercizio15 {
  public static void main(String[] args) {
    Punto p1 = new Punto(1,2,4);
    Punto p2 = new Punto(2,3,7);
    Segmento s = new Segmento(p1,p2);
    System.out.println(p1);
    System.out.println(s);
  }
}
```

Soluzione esercizio 16

- Il programma stampa:

  `I DUE OGGETTI SONO DIVERSI`

- Infatti viene chiamata la funzione `d.equals()` che a sua volta effettua la chiamata a `super.equals(e)`, cioè a `B.equals(e)`.

  Quest'ultima restituisce `false`, in quanto `d.getClass().equals(e.getClass())` restituisce `false`.

  Il motivo per cui ciò avviene è che `getClass()` restituisce la la classe **più specifica** di cui l'oggetto d'invocazione è istanza.


```
// File Esercizio17/Segmento.java

public class SegmentoOrientato extends Segmento  {
  protected boolean da_inizio_a_fine;
  public SegmentoOrientato(Punto i, Punto f) {
    super(i,f);
  }
  public SegmentoOrientato(Punto i, Punto f, boolean verso) {
    super(i,f);
    da_inizio_a_fine = verso;
  }
  public boolean equals(Object ogg) {
    if (super.equals(ogg)) {
      SegmentoOrientato so = (SegmentoOrientato)ogg;
      return so.da_inizio_a_fine == da_inizio_a_fine;
    }
    else return false;
  }
```

```
  public Object clone() {
    SegmentoOrientato so = (SegmentoOrientato)super.clone();
    return so;
  }
  public String toString() {
    return super.toString() + (da_inizio_a_fine?"--->":"<---");
  }
}
```

```
// File ParteTerza/Esercizio18.java

abstract class SoggettoFiscale {
   public SoggettoFiscale(String s) {
      nome = s;
   }
   private String nome;
   public String Nome() { return nome; }
   abstract public int Anzianita(int anno_attuale);
}

class Impiegato extends SoggettoFiscale {
   public Impiegato(String s, int r) {
       super(s);
       anno_assunzione = r;
   }
   private int anno_assunzione;
   public int Anzianita(int anno_attuale) {
```



```
       return anno_attuale - anno_assunzione;
   }
}


class Pensionato extends SoggettoFiscale {
    public Pensionato(String s, int p) {
       super(s);
       anno_pensione = p;
    }
    private int anno_pensione;
    public int Anzianita(int anno_attuale) {
       return anno_attuale - anno_pensione;
    }
}

class Straniero extends SoggettoFiscale {
    public Straniero(String s, int g) {
```

```
        super(s);
        anno_ingresso = g;
    }
    private int anno_ingresso;
    public int Anzianita(int anno_attuale) {
        return anno_attuale - anno_ingresso;
    }
}

public class prova {
    public static boolean AltaAnzianita(SoggettoFiscale f, int anno) {
        return f.Anzianita(anno) > 10;
    }
    public static void main (String arg[]) throws IOException {
        Straniero s1 = new Straniero("Paul",1995);
        Impiegato i1 = new Impiegato("Aldo",1990);
        Pensionato p1 = new Pensionato("Giacomo",1986);
        System.out.println(s1.Nome() + " " + AltaAnzianita(s1,2002)
```

```
                + " " + i1.Nome() + " " + AltaAnzianita(i1,2002)
                + " " + p1.Nome() + " " + AltaAnzianita(p1,2002));
    }
}
```