

Corso di

Progettazione del Software

Anno Accademico 2003-2004

Corso di Laurea in Ingegneria Gestionale

Prof. Monica Scannapieco (A–L) & Prof. Giuseppe De Giacomo (M–Z)

Ereditarietà in Java

Unità 1

1

Ereditarietà Java

Argomenti che tratteremo in questa parte di corso:

1. Package
2. Livello di accesso
3. Derivazioni tra classi ed ereditarietà
4. Classi astratte
5. Interfacce
6. Uguaglianza superficiale e uguaglianza profonda
7. Copia superficiale e copia profonda
8. Oggetti mutabili e oggetti immutabili

2

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, una sola delle quali può essere **public**.
- 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()`).

3

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*.

4

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`).

5

Struttura dei package e dei direttori

Tutti i file relativi al package `mio_pack` devono risiedere 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` devono risiedere 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.*;`

6

Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

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

// File unita1/Esempio1.java
// uso package

import mio_package.*;    // importo mio_package.*

public class Esempio1 {
    public static void main(String[] args) {
        C c = new C();
        c.F_C();
    }
}

//Nota: posso usare C definita in mio_package, usando il "nome corto"
```

Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

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

// File unita1/Esempio2.java
// uso package

//Nota: non importo mio_package.*

public class Esempio2 {
    public static void main(String[] args) {
```

8

```
    mio_package.C c = new mio_package.C();
    c.F_C();
}

//Nota: posso usare C definita in mio_package, ma devo usare il "nome lungo"
```

Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

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

// File unita1/mio_package/mio_subpackage/D.java
package mio_package.mio_subpackage;
public class D {
    public void F_D() {
        System.out.println("Sono F_D()");
    }
}
```

9

```
// File unita1/Esempio3.java
// uso package

import mio_package.*;
// Nota: non importo mio_package.mio_subpackage.*

public class Esempio3 {
    public static void main(String[] args) {
        C c = new C();
        c.F_C();
        mio_package.mio_subpackage.D d = new mio_package.mio_subpackage.D();
        d.F_D();
    }
}
```

Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

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

// File unita1/mio_package/mio_subpackage/D.java
package mio_package.mio_subpackage;
public class D {
    public void F_D() {
        System.out.println("Sono F_D()");
    }
}
```

10

```
// File unita1/Esempio4.java
// uso package

import mio_package.mio_subpackage.*;
// Nota: non importo mio_package.*

public class Esempio4 {
    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 1: package

```
// File unita1/Esempio5.java
import java.io.*;

public class Esempio5 {
    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));
        String linea = in.readLine();
        while(linea != null) {
            System.out.println(linea);
            linea = in.readLine();
        }
        in.close();
    }
}
```

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

11

Livelli di accesso di una classe

Una classe può essere specificata con **due** livelli di accesso:

1. `public`
2. non qualificato (è il *default*)

Se una classe è dichiarata `public` allora è accessibile fuori dal package al quale appartiene, altrimenti è accessibile solo all'interno del package al quale appartiene.

12

Livelli di accesso dei campi di una classe

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).

13

Regole di visibilità tra campi

IL METODO B VEDE IL CAMPO A ?						
METODO B \	IN \CAMPO A	public	protected	non qual.	private	
-----\	-----\	-----\	-----\	-----\	-----\	
STESSA CLASSE	SI SI SI SI 1					
CLASSE STESSO PACKAGE	SI SI SI NO 2					NOTA: Decrescono i diritti
CLASSE DERIVATA PACKAGE DIVERSO	SI SI NO NO 3					
CL. NON DERIVATA PACKAGE DIVERSO	SI NO NO NO 4					V
	-----\	-----\	-----\	-----\	-----\	V

----->>
NOTA: Decrescono i diritti

14

Commento sulle regole di visibilità

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

Ricordiamo che uno dei principi 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 pubbliche.

15

Visibilità: esempio

```
// File unita1/Esempio6.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 Esempio6 {
    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 Esempio6.
    }
}
```

16

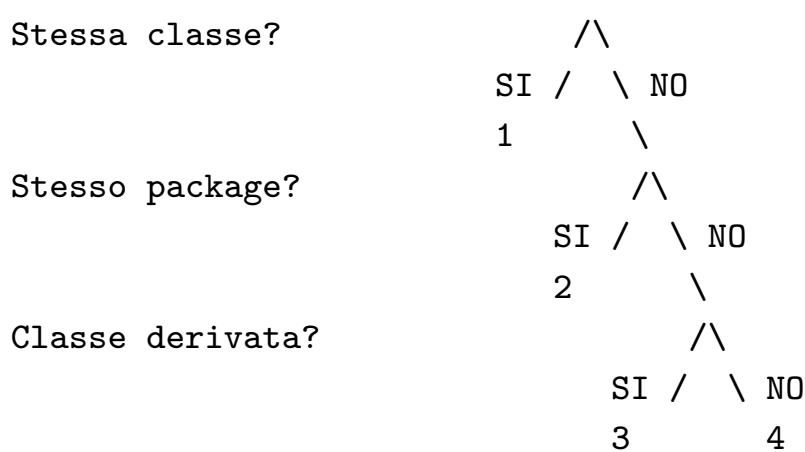
Esercizio 2: visibilità

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

17

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.

18

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; }
// ...
}
```

19

Principi fondamentali della derivazione

I quattro **principi 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.

20

Principi 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

21

Principi 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 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
```

22

Esercizio 3: 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`.

23

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.

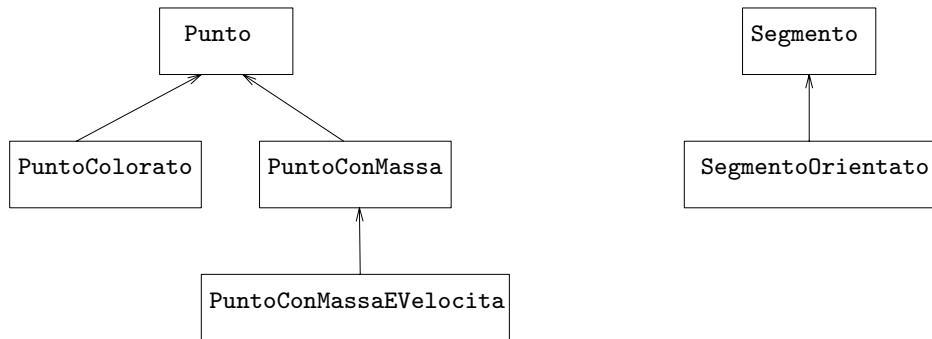
```
class B { ...  
class D extends B { ...  
class D2 extends B { ...
```

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*.

24

Gerarchie di classi: esempio



25

Esercizio 4: gerarchie di classi

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

26

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.

27

Casting

Non è possibile ai campi della classe D attraverso b. Per farlo dobbiamo prima fare un **casting**.

```
// File unita1/Esempio7.java
class B { }
class D extends B { int x_d; }

public class Esempio7 {
    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
    }
}
```

28

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 unita1/Esempio8.java
class B { }
class D extends B { int x_d; }

public class Esempio8 {
    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)
    }
}
```

29

Esercizio 5: 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);
}
```

30

Derivazione e regole di visibilità

Una classe D derivata da un'altra classe B, anche se in un package diverso, 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.

31

Costruttori di classi derivate

Al momento dell'invocazione di un costruttore della classe derivata, se il costruttore della classe derivata non contiene esplicite chiamate al costruttore della classe base (vedi dopo), viene chiamato **automaticamente** anche il costruttore senza argomenti della classe base. Ciò avviene:

- sia se la classe base ha il costruttore senza argomenti standard,
- sia se la classe base ha il costruttore senza argomenti definito esplicitamente,
- sia se la classe non ha il costruttore senza argomenti(!), in questo caso si ha un errore di compilazione.

```
class B { ... }
```

32

```
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 unita1/Esempio9.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.
```

34

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.

35

Uso di super() nei costruttori: esempio

```
// File unita1/Esempio10.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 Esempio10 {
    public static void main(String[] args) {
        D d = new D(3,4);  } }
```

36

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).

37

Esercizio 6: 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.

38

Nota: riuso di costruttori in una classe – this()

È possibile **riusare** anche i costruttori già definiti anche nell'ambito di una stessa classe. Ciò viene fatto mediante il costrutto `this()`, analogo a `super()`, che analogamente deve essere la *prima istruzione* del corpo del costruttore della classe.

```
public class Persona {  
    private String nome;  
    private String residenza;  
  
    public Persona(String n, String r) {  
        nome = n;  
        residenza = r;  
    }  
  
    public Persona(String n) {  
        this(n, null);  
    }  
}
```

39

```

    }

    public Persona() {
        this("Mario Rossi");
    }

    ...
}

```

Si noti che l'uso contemporaneo di `this()` e `super()` in uno stesso costruttore non è possibile, poiche entrambi dovrebbero essere la prima istruzione del costruttore. D'altra parte l'uso contemporaneo di entrambe nello stesso costruttore non avrebbe senso.

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()  
}
```

41

Overriding di funzioni: esempio

```
// File unita1/Esempio11.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 Esempio11 {  
    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 } }
```

42

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`).

43

Riassunto overloading e overriding

	OVERLOADING	OVERRIDING
nome della funzione	uguale	uguale
tipo restituito	qualsiasi	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

44

Esercizio 7: overriding e compatibilità

```
// File unita1/Esercizio7.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 Esercizio7 {
    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?

45

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)
```

46

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);      // INVOCÀ D.f(int)
h(bb);     // INVOCÀ B.f(int)
```

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

47

h(d): d denota un oggetto della classe D – viene invocata la funzione f(int) definita in D;

h(bb): bb denota un oggetto della classe B – viene invocata la funzione f(int) definita in B.

Esercizio 8: cosa fa questo programma?

```
// File unita1/Esercizio8.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 Esercizio8 {
    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());
    }
}
```

48

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 unita1/Esempio13.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); }
}
```

49

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 unita1/Esempio14.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
```

50

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**.

51

Sovrascrittura dei campi dati: esempio

```
// File unita1/Esempio15.java
class B { int i; }
class D extends B {
    char i;
    void stampa() {
        System.out.println(i); System.out.println(super.i);
    }
}
public class Esempio15 {
    public static void main(String[] args) {
        D d = new D();
        d.i = 'f';
        ((B)d).i = 9;
        d.stampa();
    }
}
```

52

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()**.

53

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  
}
```

54

Quando una classe va definita astratta

Una classe A si definirà come astratta quando:

- non ha senso pensare a oggetti che siano istanze di A **senza essere istanze anche di una sottoclasse (eventualmente indiretta) di A**;
- 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.

Anche se spesso si dice che una classe astratta A non ha istanze, ciò non è propriamente 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.

55

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.

56

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 ricompiere la classe Persona stessa. **Riuso ed estendibilità sarebbero compromessi!**

57

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.

58

Esercizio 9

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.

59

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 ';' } 
```

60

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()`, perché non esistono oggetti di tipo I, ma esistono solo riferimenti di tipo I.

61

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**.

62

Esempio di interfaccia e di funzione cliente

Vogliamo definire una interfaccia `Confrontabile` che offre 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 "maggiori" e "paritetico" (rispettivamente), x_1 è massimale in $\{x_1, x_2, x_3\}$ se e solo se:

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

63

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

```
// File unita1/Esempio16.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;    }  }
```

64

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).

65

Implementazione di un'interfaccia (cont.)

```
// File unita1/Esempio17.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;
    }
}
```

66

```
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`.

67

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`).

68

Esempio di uso di interfaccia (cont.)

```
// File unita1/Esempio18.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);
    }
}
```

69

Esercizio 10

Arricchire la classe Utilita con:

- una funzione `Massimale()` che, ricevuto come argomento un vettore di riferimenti a `Comparabile`, 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.

70

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() { ... }  
}
```

71

Esempio di implementazione multipla

```
// File unita1/Esempio19.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 Esempio19 {  
    public static void main(String[] args) {  
        C c = new C();  
        c.g();  
        c.h();  
        c.f();  
    }  
}
```

72

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() { ... }  
}
```

73

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() { ... }  
}
```

74

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.

75

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

*Eccetto che per campi dati **final** (cioè costanti).

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` (della prima sappiamo già fare l'overriding):

- `public String toString()`
- `public final Class getClass()`
- `public boolean equals(Object)`
- `protected Object clone()`

77

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 unita1/Esempio20.java
class B {
    private int i;
    B(int x) { i = x; }
    public String toString() { return "i: " + i; }
}

public class Esempio20 {
    public static void main(String[] args) {
        B b = new B(5);
```

78

```
        System.out.println(b);
    }
}

/* Stampa:
i: 5
Nota: se non avessimo ridefinito toString() avrebbe stampato
B@601bb1
*/
```

Esercizio 10: 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>)`

Stampa in classi derivate

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

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

80

La classe Java Class

- Esiste implicitamente un oggetto di classe `Class` per ogni classe (o interfaccia) `B` del programma, sia di libreria che definita da utente.
- Questo oggetto può essere denotato in due modi:
 - tramite letterali aventi la forma:
`... B.class ... // ha tipo Class`
 - tramite riferimenti di tipo `Class`
`Class c = ...`
- Gli oggetti di tipo `Class` sono creati dal sistema runtime in modo automatico. Si noti che `Class` non ha costruttori accessibili dai clienti.

81

La classe Java Class (cont.)

- 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.

82

La funzione `isInstance()`

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

```
... B.class.isInstance(b) ... // vale true se b e' istanza di B
```

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

```
class D extends B ...
```

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

83

Esercizio 11: cosa fa questo programma?

```
// File unita1/Esercizio11.java

class B {}
class D extends B {}

public class Esercizio11 {
    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));
    }
}
```

84

La funzione `isInstance()` (cont.)

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

```
interface I {...}
```

```
... I.class.isInstance(b) ... // vale true, se b e' istanza di
                           // una classe che implementa I
```

```
class D implements I {...}
```

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

85

Esercizio 11bis: cosa fa questo programma?

```
// File unita1/Esercizio11bis.java

interface I {}

class D implements I {}

public class Esercizio11bis {
    public static void main(String[] args) {
        I i1 = new D();
        D d1 = new D();
        System.out.println(I.class.isInstance(i1));
        System.out.println(I.class.isInstance(d1));
    }
}
```

86

L'operatore Java instanceof

Java è dotato di un operatore predefinito `instanceof` per verificare l'appartenenza di un oggetto ad una classe o la conformità di un oggetto ad una interfaccia.

In particolare le seguenti espressioni booleane si comportano in modo identico:

... `B.class.isInstance(b)` ...

... `b instanceof B` ...

Si noti che nell'ultima espressione si è usato `B` e non `B.class`. Questo perché l'operatore `instanceof` non fa uso di un oggetto della classe `Class`, ma del **nome della classe**. Ne segue che per poter applicare `instanceof` la classe a cui applicarlo deve essere nota a tempo di compilazione. Quindi la seguente istruzione non è riscrivibile utilizzando `instanceof`:

87

```
Class c = ...  
...c.isInstance(b)...
```

La funzione getClass() di Object

La classe `Object` contiene una funzione `public final Class getClass()` (che non può essere ridefinita) che restituisce la classe dell'oggetto di invocazione, cioè la classe più specifica di cui l'oggetto di invocazione è istanza.

Attraverso l'uso di `getClass()` (e di `equals()` definito per gli oggetti di tipo `Class`), possiamo, ad esempio, verificare se due oggetti appartengono alla stessa classe:

```
class B {  
    private int x;  
    public B(int n) {x=n;}  
    ...  
}  
  
B b1 = new B(10);  
...  
B b2 = new B(100);  
... b1.getClass().equals(b2.getClass()) ... // vale true
```

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!");
```

89

Uguaglianza fra oggetti

Quando confrontiamo due oggetti dobbiamo chiarire che tipo di uguaglianza vogliamo utilizzare:

- **Uguaglianza superficiale:** *verifica se due riferimenti ad un oggetto sono uguali, cioè denotano lo stesso oggetto;*
- **Uguaglianza profonda:** *verifica se le informazioni (rilevanti) contenute nei due oggetti sono uguali.*

90

Uguaglianza fra oggetti (cont.)

```
class C {  
    private int x, y;  
    public C(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}  
// ...  
C c1 = new C(4,5);  
C c2 = new C(4,5);
```

Nota: c1 e c2 ...

- ... non sono uguali superficialmente
- ... sono uguali profondamente

91

Uguaglianza superficiale

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

Ad esempio:

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

92

Uguaglianza 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**.

93

Uguaglianza fra oggetti: funzione `equals()`

La funzione `public boolean equals(Object)` definita in `Object` ha lo scopo di verificare l'uguaglianza fra oggetti.

`equals()`, come tutte le funzioni definite in `Object`, è **ereditata** da 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;  
    public C(int x, int y) {this.x = x; this.y = y;}  
}  
// ...  
C c1 = new C(4,5), c2 = new C(4,5);  
if (c1.equals(c2))  
    System.out.println("Uguali!");  
else  
    System.out.println("Diversi!");
```

94

Uguaglianza profonda: overriding di equals()

È tuttavia possibile **ridfinire** il significato della funzione `equals()`, facendone **overriding**, in modo tale da verificare l'**uguaglianza profonda** fra oggetti.

Per fare ciò dobbiamo ridefinire la funzione `equals()` come illustrato nel seguente esempio:

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

95

Analisi critica dell'overriding di equals()

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

- `public boolean equals(Object o) {`

la funzione deve avere come parametro un riferimento di tipo `Object` perchè stiamo facendo **overriding** della funzione `equals()` ereditata dalla classe `Object`.

- `if (o != null ...`

dobbiamo essere sicuri che il riferimento `o` passato alla funzione non sia `null`, altrimenti gli oggetti sono banalmente diversi, visto che l'oggetto passato alla funzione non è un oggetto;

- `... && getClass().equals(o.getClass())`

96

dobbiamo essere sicuri che `o` si riferisca ad un oggetto della stessa classe dell'oggetto di invocazione (`B`, nell'esempio), altrimenti i due oggetti sono istanze di classi diverse e quindi sono ancora una volta banalmente diversi;

- `B b = (B)o;`

se la condizione logica dell'`if` risulta vera, allora facendo un **cast** denotiamo l'oggetto passato alla funzione attraverso un riferimento del tipo dell'oggetto di invocazione (`B`, nell'esempio) invece che attraverso un riferimento generico di tipo `Object`; in questo modo potremo accedere ai campi specifici della classe di interesse (`B`, nell'esempio)

- `return (x == b.x) && (y == b.y)`

a questo punto possiamo finalmente verificare l'uguaglianza tra i singoli campi della classe

- `return false;`

non appena uno dei test di cui sopra fallisce, sappiamo che gli oggetti non sono uguali e quindi possiamo restituire `false`.

Overriding, non overloading, di equals()

Si noti che si deve fare **overriding** di `equals()` e **non overloading**. Altrimenti si possono avere risultati controintuitivi.

Cosa fa questo programma?

```
// File unita1/Esercizio12.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 Esercizio12 {
    static void stampaUguali(Object o1, Object o2) {
        if (o1.equals(o2))
```

97

```
        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);
    }
}
```

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()`.

```
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!");
```

98

Uguaglianza: classe String

In `String` la funzione `equals()` è 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!");
```

99

Esercizio 13: uguaglianza

Progettare tre classi:

Punto: vedi esercizio 1 dell'Unità 0;

Segmento: vedi esercizio 1 dell'Unità 0;

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.

100

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 z == d.z;  
        }  
        else return false;  
    }  
}
```

101

Uguaglianza profonda in classi derivate (cont.)

- `D.equals()` delega a `super.equals()` (cioè `B.equals()`) alcuni controlli (**riuso**):
 - che il parametro attuale non sia `null`;
 - 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`).

102

Esercizio 14: cosa fa questo programma?

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

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

103

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()
```

record di attivazione di F()	4	4
	a	b

104

Copia di oggetti

Quando copiamo un oggetto dobbiamo chiarire che tipo di copia vogliamo effettuare:

- **copia superficiale**: *copia dei riferimenti ad un oggetto*;
- **copia profonda**: *copia dell'oggetto stesso*.

105

Copia fra oggetti: superficiale

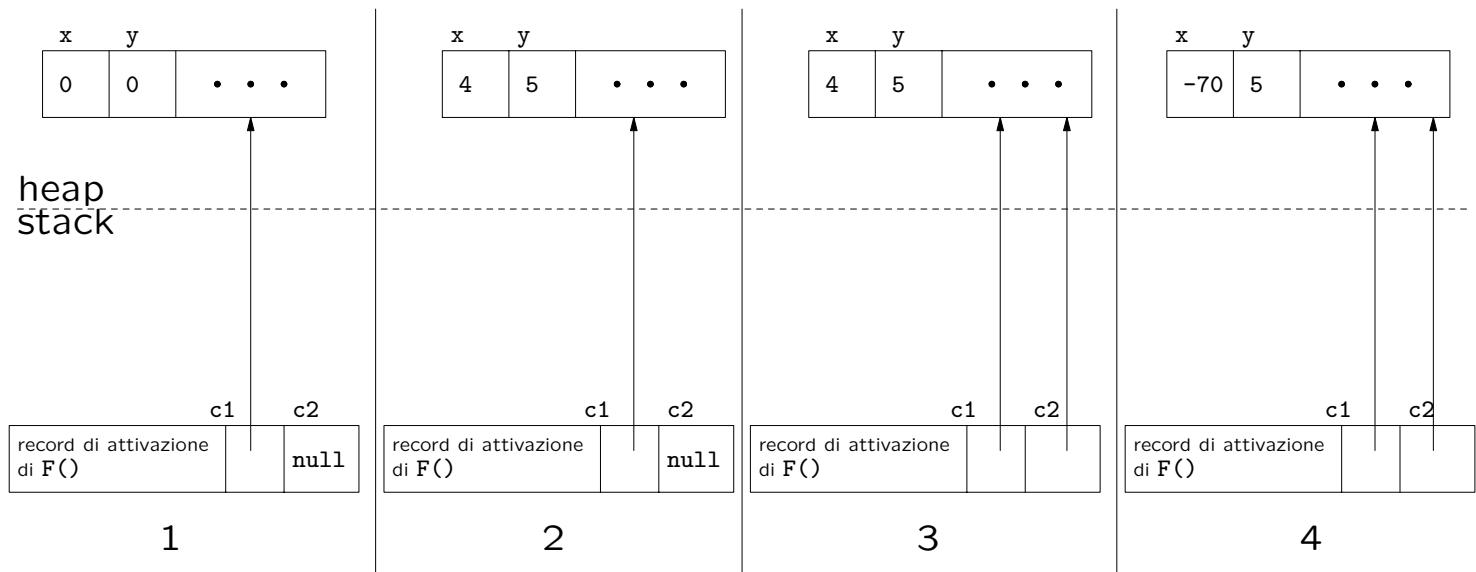
Se usiamo '=' per copiare **due oggetti**, stiamo effettuando la copia *superficiale*.

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()
```

106

Evoluzione (run-time) dello stato della memoria



107

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.

108

Copia profonda: la funzione clone()

La funzione `protected Object clone()` definita in `Object` ha lo scopo di permettere la copia profonda.

Poiché `clone()` in `Object` è `protected` essa, anche se ereditata, non è accessibile ai clienti della nostra classe.

Se lo desideriamo, possiamo **ridenominarla** (farne **overriding**), rendendola `public` e facendo in maniera tale che effettui la **copia profonda** fra oggetti, come illustrato nel esempio seguente.

109

Copia profonda: la funzione clone() (cont.)

```
class B implements Cloneable {  
    private int x, y;  
  
    public Object clone() {  
        try {  
            B b = (B)super.clone(); // Object.clone copia campo a campo  
            //eventuale copia profonda dei campi - in questo caso non necessaria  
            return b;  
        } catch (CloneNotSupportedException e) {  
            // non puo' accadere, ma va comunque gestito  
            throw new InternalError(e.toString());  
        }  
    }  
}
```

110

Analisi critica dell'overriding di clone()

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

- `class B implements Cloneable {`
per fare overriding di `clone()` è necessario dichiarare che la classe implementa l'interfaccia `Cloneable`. Questa è un'interfaccia priva di campi (non contiene dichiarazioni di funzione, né contiene costanti) che serve solo a “marcare” come “cloneable” gli oggetti della classe.
- `public Object clone() {`
nel fare l'overriding di `clone()` lo dichiariamo `public`, invece di `protected` rendendolo così accessibile ai clienti della nostra classe.
- `...super.clone()`

111

questa è l'invocazione alla funzione `clone()` definita in `Object`.

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

Si noti che questo comportamento, che di fatto corrisponde alla copia esatta della porzione di memoria dove è contenuto l'oggetto di invocazione, non è ottenibile in nessun altro modo in Java.

- `B b = (B)super.clone();`

il riferimento restituito da `super.clone()`, che è di tipo `Object`, viene convertito, mediante *casting* in un riferimento del tipo dell'oggetto di invocazione (`B`, nell'esempio), in modo da potere operare sui campi propri della classe di appartenenza (cioè `B`).

- `//eventuale copia profonda dei campi`

dopo avere fatto la copia campo a campo e avere un riferimento all'oggetto risultante del tipo desiderato, si fanno eventuali copie profonde dei campi dell'oggetto di invocazione (nell'esempio, non è necessario essendo i campi di `B` di tipo `int`).

- ```
try {
 ...
}
catch (CloneNotSupportedException e) {
 throw new InternalError(e.toString());
}
```

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`. Poiché la nostra classe implementa `Cloneable` il codice nella clausola `catch` non verrà mai eseguito.

## 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(); //si noti il casting!
System.out.println("b2.x: " + b2.x + ", b2.y: " + b2.y);
```

112

## Copia profonda: classe String

La classe `String` non fa overriding di `clone()`, quindi non possiamo fare cloni di stringhe.

Tuttavia, la classe `String` è `final`, cioè non permette di definire sottoclassi. Inoltre non ha superclassi eccetto `Object`. Con queste condizioni particolari, se vogliamo fare una copia profonda di un oggetto `String`, possiamo semplicemente utilizzare, mediante `new`, un suo costruttore, 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
```

Si noti che se la classe `String` non fosse stata `final` questo costruttore non avrebbe in nessun modo potuto garantire di generare la copia esatta (perchè non avrebbe potuto sapere la classe dell'oggetto passato come parametro a runtime).

113

## Esercizio 15: copia

Con riferimento alle tre classi Punto, Segmento e Valuta dell'esercizio 13, ridefinire il significato della funzione `clone()`, facendo in maniera tale che effettui la *copia profonda* fra oggetti.

114

## 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 profonde
 return d;
 }
 // ...
}
```

115

## 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.

116

## Copia profonda in classi derivate: esempio

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

117

## Esercizio 16: 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.

118

## Oggetti immutabili e oggetti mutabili

Nel realizzare una classe è molto importante avere presente se gli oggetti istanza della classe devono essere:

- **oggetti immutabili:** cioè, il cui stato non può cambiare nel tempo (cioè a fronte di operazioni)
- **oggetti mutabili:** cioè, il cui stato può essere modificato da alcune operazioni.

119

## Oggetti immutabili

Gli oggetti immutabili tipicamente sono usati per rappresentare “valori”. Ad esempio gli oggetti `String` sono **oggetti immutabili** ed in effetti rappresentano valori di tipo stringa (in modo analogo a come valori `int` rappresentano valori interi).

Gli oggetti immutabili sono realizzati in Java assicurandosi che tutti i metodi accessibili ai clienti (e.g., `public`) **non effettuino side-effect** sull’oggetto di invocazione.

In questo modo rendiamo impossibile la modifica dello stato dell’oggetto da parte dei clienti rendendo l’**oggetto immutabile**.

120

## Oggetti mutabili

Gli oggetti mutabili tipicamente sono usati per rappresentare “entità”, che pur non modificando la propria identità, modificano il proprio stato. Tipicamente entità del modo reale quali *persone*, *automobili*, ecc. sono rappresentate da oggetti mutabili, in quanto pur non cambiando la propria identità, cambiano stato. Un altro esempio è `StringBuffer` le cui istanze sono oggetti mutabili che mantengono una sequenza di caratteri permettendone modifiche se richiesto dal cliente.

Gli oggetti mutabili sono realizzati in Java includendo tra i metodi accessibili ai clienti (e.g., `public`) metodi che **effettuano side-effect** sull’oggetto di invocazione.

In questo modo rendiamo possibile la modifica dello stato dell’oggetto da parte dei clienti rendendo l’**oggetto mutabile**.

121

## Oggetti immutabili: uguaglianza e copia

Alcune considerazioni metodologiche:

- `equals()`. Poichè tipicamente sono usati per rappresentare “valori” l’identificatore dell’oggetto non riveste alcun ruolo quindi, è **necessario fare l’overriding di `equals()` in `Object`** in modo verifichi l’**uguaglianza profonda**.
- `clone()`. Poichè gli oggetti immutabili non possono essere modificati dal cliente, è tipicamente superfluo effettuare copie di tali oggetti (visto che possiamo utilizzare gli originali, senza rischio di modifiche). Quindi, tipicamente **non si fa overriding di `clone()` di `Object`** lasciandolo inaccessibile ai clienti.

122

## Oggetti mutabili: uguaglianza e copia

Alcune considerazioni metodologiche:

- `equals()`. Bisogna capire se **l’identificatore dell’oggetto è significativo** per classe che si sta realizzando. Se lo è, come tipicamente avviene per oggetti che corrispondono a rappresentazioni di oggetti del mondo reale, allora tipicamente **non si fa overriding di `equals()` di `Object`**, visto che va già bene per la verifica dell’uguaglianza.

Se invece **l’identificatore non è significativo**, come tipicamente avviene per oggetti che rappresentano collezioni di altri oggetti, allora **va fatto overriding di `equals()`** affinché verifichi l’uguaglianza profonda tenendo conto delle informazioni rilevanti.

- `clone()`. Valgono considerazioni analoghe, cioè bisogna distinguere i casi in cui **l’identificatore dell’oggetto è significativo** da quelli in cui non

123

lo è. Se lo è (e.g., rappresentazione di oggetti del mondo reale) allora mettere a disposizione del cliente un metodo per la copia profonda spesso non ha senso, visto che l'identificatore sarà in ogni caso diverso, quindi **non si fa overriding di** `clone()` di `Object`,

Invece nel caso in cui **l'identificatore dell'oggetto non è significativo** (e.g., oggetti che rappresentano collezioni) allora permettere la copia profonda dell' oggetto può essere molto utile per il cliente e quindi tipicamente **si fa overriding di** `clone()`.

## **Soluzioni degli esercizi dell'Unità 1 – Parte 1**

## Soluzione esercizio 1

```
// File unita1/Esercizio1.java

public class Esercizio1 {
 public static void main(String[] args) throws java.io.IOException {
 // stampa su schermo il file passato tramite linea di comando
 java.io.FileInputStream istream = new java.io.FileInputStream(args[0]);
 java.io.BufferedReader in = new java.io.BufferedReader(
 new java.io.InputStreamReader(istream));
 String linea = in.readLine();
 while(linea != null) {
 System.out.println(linea);
 linea = in.readLine();
 }
 in.close();
 }
}
```

125

## Soluzione esercizio 2

```
// File unita1/Esercizio2/Esempio2_2.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); }
}

public class Esercizio2_2 {
 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' non
 // qualificata, ma C e' nello stesso package
 //int val = c.x; // NO: il campo x e' privato in C
 }
}
```

126

## Soluzione esercizio 2

```
package package2_3;

public 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); }
}

// File unita1/Esercizio2/Esercizio2_3.java

import package2_3.*;

public class Esercizio2_3 {
 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
 }
}
```

127

## Soluzione esercizio 2

```
package package2_4;

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

// File unita1/Esercizio2/Esercizio2_4.java

import package2_4.*;

public class Esercizio2_4 {
 public static void main(String[] args) {
 C c = new C(7,12); // OK: il costruttore di C e' pubblico
 //c.stampa(); // No: la funzione stampa() di C e' non
 }
}
```

128

```

 // qualificata e C \e' in un altro package
//int val = c.x; // NO: il campo x e' privato in C
}
}

```

## Soluzione esercizio 3

```

// File unita1/Esercizio3.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 Esercizio3 {
 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
 }
}

```

## Soluzione esercizio 4

```

// File unita1/Esercizio4.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;
}

```

## Soluzione esercizio 5

```
// File unita1/Esercizio5.java
class B { }
class D extends B { int x_d; }
public class Esercizio5 {
 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)
 }
}
```

131

## Soluzione esercizio 6

```
// File unita1/Esercizio6.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;
 }
}
```

132

```
}

class PuntoConMassa extends Punto {
 protected float massa;
 public PuntoConMassa(float m) {
 // INVOCÀ 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 7

Verificare il programma su calcolatore.

## Soluzione esercizio 8

- 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'aver usato il riferimento b o il riferimento d.

134

## Soluzione esercizio 9

```
// File unita1/Esercizio9.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;
 }
}
```

135

```
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 Esercizio9 {
 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));
}
}
```

## Soluzione esercizio 10

Svolgere da soli.

## Soluzioni degli esercizi dell'Unità 1 – Parte 2

137

### Soluzione esercizio 10

```
// File unita1/Esercizio10.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) {
```

138

```

 inizio = i; fine = f;
}
public String toString() {
 return "(" + inizio + "," + fine + ")";
}
}

public class Esercizio10 {
 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 11

- 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).

## Soluzione esercizio 11bis

Lasciata allo studente

140

## Soluzione esercizio 12

- 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`.

141

## Soluzione esercizio 13 - Punto

```
// File unita1/Esercizio13Punto.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 (x == p.x) && (y == p.y) && (z == p.z);
 }
 else return false;
 }
}

public class Esercizio13Punto {
 public static void main(String[] args) {
```

142

```
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!");
}
```

## Soluzione esercizio 13 - Segmento

```
// File unita1/Esercizio13Segmento.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 inizio.equals(s.inizio) && fine.equals(s.fine);
 // inizio == s.inizio && fine == s.fine SAREBBE SBAGLIATO!
 }
 else return false;
 }
}

public class Esercizio13Segmento {
```

143

```
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!");
}
```

## Soluzione esercizio 13 - Valuta

```
// File unita1/Esercizio13Valuta.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 (unita == v.unita) && (centesimi == v.centesimi) &&
 (nome.equals(v.nome));
 // (nome == v.nome) SAREBBE SBAGLIATO!
 }
 else return false;
 }
}
```

144

```
public class Esercizio13Valuta {
 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!");
 }
}
```

## Soluzione esercizio 14

- 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 classe **più specifica** di cui l'oggetto d'invocazione è istanza.

145

## Soluzione esercizio 15 - Punto

```
// File unita1/Esercizio15Punto.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 Esercizio15Punto {
 public static void main(String[] args) {
 Punto p1 = new Punto(), p2;
```

146

```
 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);
}
```

## Soluzione esercizio 15 - Segmento

```
// File unita1/Esercizio15Segmento.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());
 }
 }
}
```

147

```
public class Esercizio15Segmento {
 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);
 }
}
```

## Soluzione esercizio 15 - Valuta

```
// File unita1/Esercizio15Valuta.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());
 }
 }
}
```

148

```
public class Esercizio15Valuta {
 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);
 }
}
```

## Soluzione esercizio 16

```
// File unita1/Esercizio16/SegmentoOrientato.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?"--->":"<---");
 }
}
```

149

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