Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

Laboratorio di Programmazione

Anno Accademico 2004-2005

Corso di Laurea in Ingegneria Informatica

Prof. Giuseppe De Giacomo (A–L) & Prof. Paolo Liberatore (M–Z)

Ereditarietà in Java

Unità 1 – Parte 2

Ereditarietà Java

Argomenti che tratteremo in questa parte di corso:

1. ...
2. La classe `Object`
3. Uguaglianza superficiale e uguaglianza profonda
4. Copia superficiale e copia profonda
5. Oggetti mutabili e oggetti immutabili

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()`

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);
```

```
    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() + // ...
  }
  // ...
}
```


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.

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.

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 anchexs oggetto della classe* `B`.

```
class D extends B ...

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

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));
  }
}
```


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;
```

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));
  }
}
```


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`:

```
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!");
```


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.*

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

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!");
```

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**.

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!");
```

Uguaglianza profonda: overriding di equals()

È tuttavia possibile **ridefinire** 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;
  }

}
```


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())`

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 ugiali e quindi possiamo restituire `false`.

Overriding, non overloding, 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))
```



```
      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!");
```


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!");
```

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.

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;
  }
}
```

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`).

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");
```

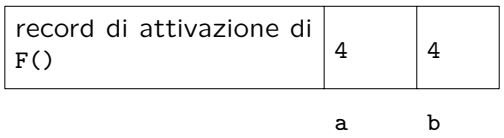
```
class E extends B {
  protected int z;
  public E(int a, int b, int c){//...
  public boolean equals(Object ogg) {
    if (super.equals(ogg)) {
      E e = (E)ogg;
      return z == e.z;
    }
    else return false;
  }
}
```

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 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*.

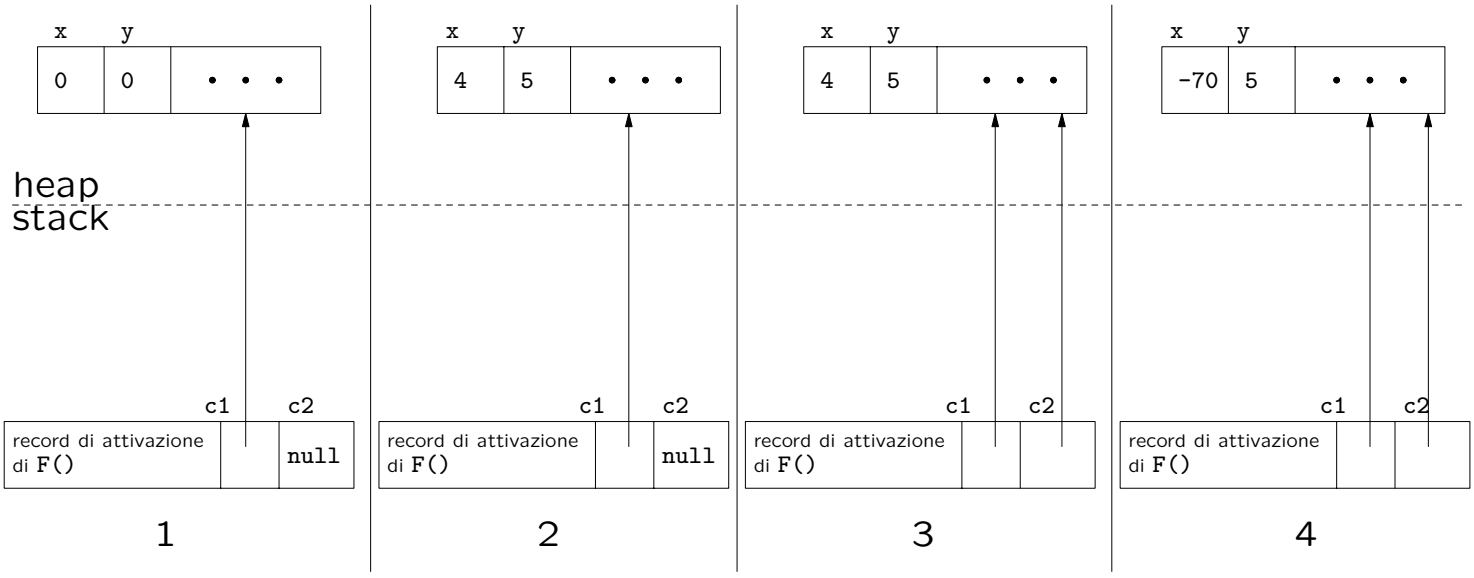
Copia fra oggetti: superficiale

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

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());
    }
  }

}
```


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, nè 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()`

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);
```


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).

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.

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;
  }
  // ...
}
```

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;
  }
  // ...
}
```

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.

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.

Oggetti immutabili

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

Gli oggetti immutabili sono relizzati 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**.

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 relizzati 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**.

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.

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 cosiderazioni analoghe, cioè bisogna distinguere i casi in cui **l'identificatore dell'oggetto è significativo** da quelli in cui non

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 2

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) {
```



```
    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

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`.

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) {
```

```
    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 {
```

```
  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;
  }
}
```

```
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.

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;
```

```
    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());
    }
  }
}
```

```
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());
    }
  }
}
```

```
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?"--->":"<---");
  }
}
```