Università degli Studi di Roma “La Sapienza”
Facoltà di Ingegneria – Corso di Laurea in Ingegneria Gestionale

Corso di Progettazione del Software

Proff. Toni Mancini e Monica Scannapieco
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”

S.JOO.5 – Java: copia profonda

versione del February 2, 2008

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

Copia di oggetti

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

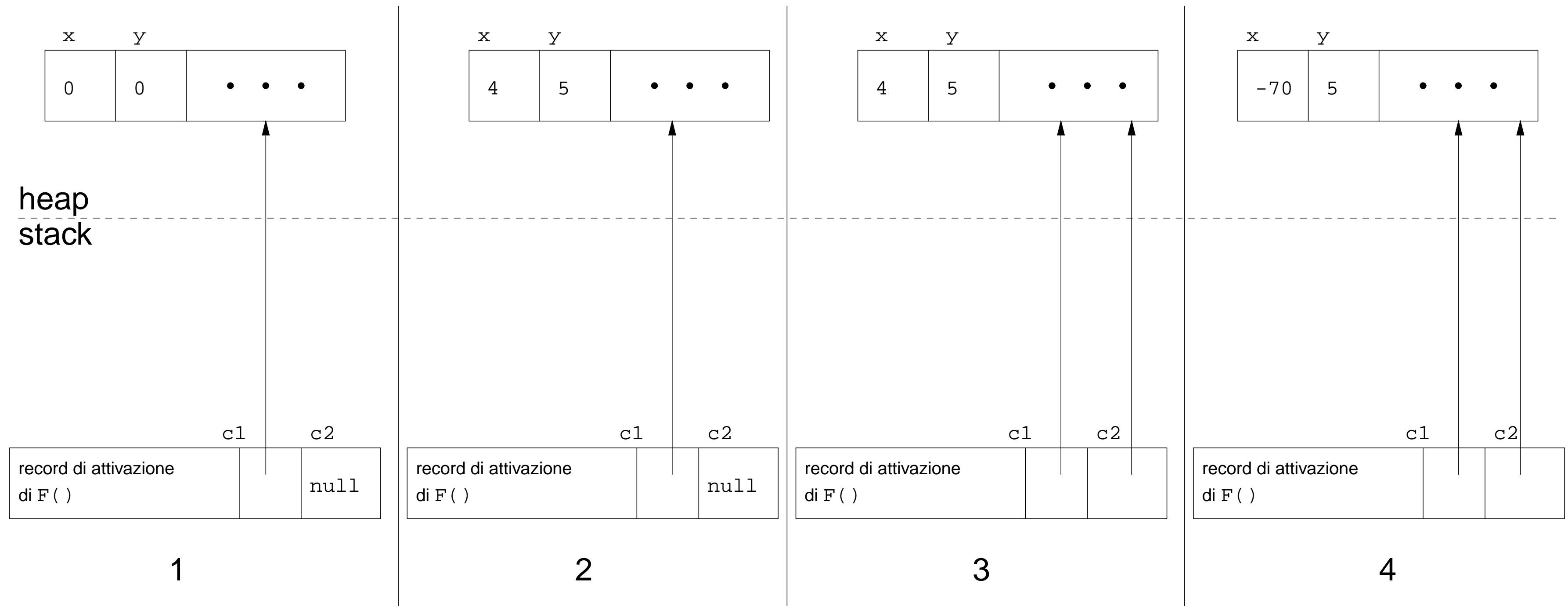
- **copia superficiale**: copia dei riferimenti ad un oggetto;
- **copia profonda**: copia dell'oggetto stesso.

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(); C c2 = null;                   // 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()
```

Evoluz. (run-time) 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: 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.

Analisi dell'overriding di clone() (cont.)

- `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 prof. 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 deriv.: 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: funz. speciali in classi deriv.

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()`.