Marco Faella

Clonazione di oggetti

v.1.2

Lezione n. 8bis

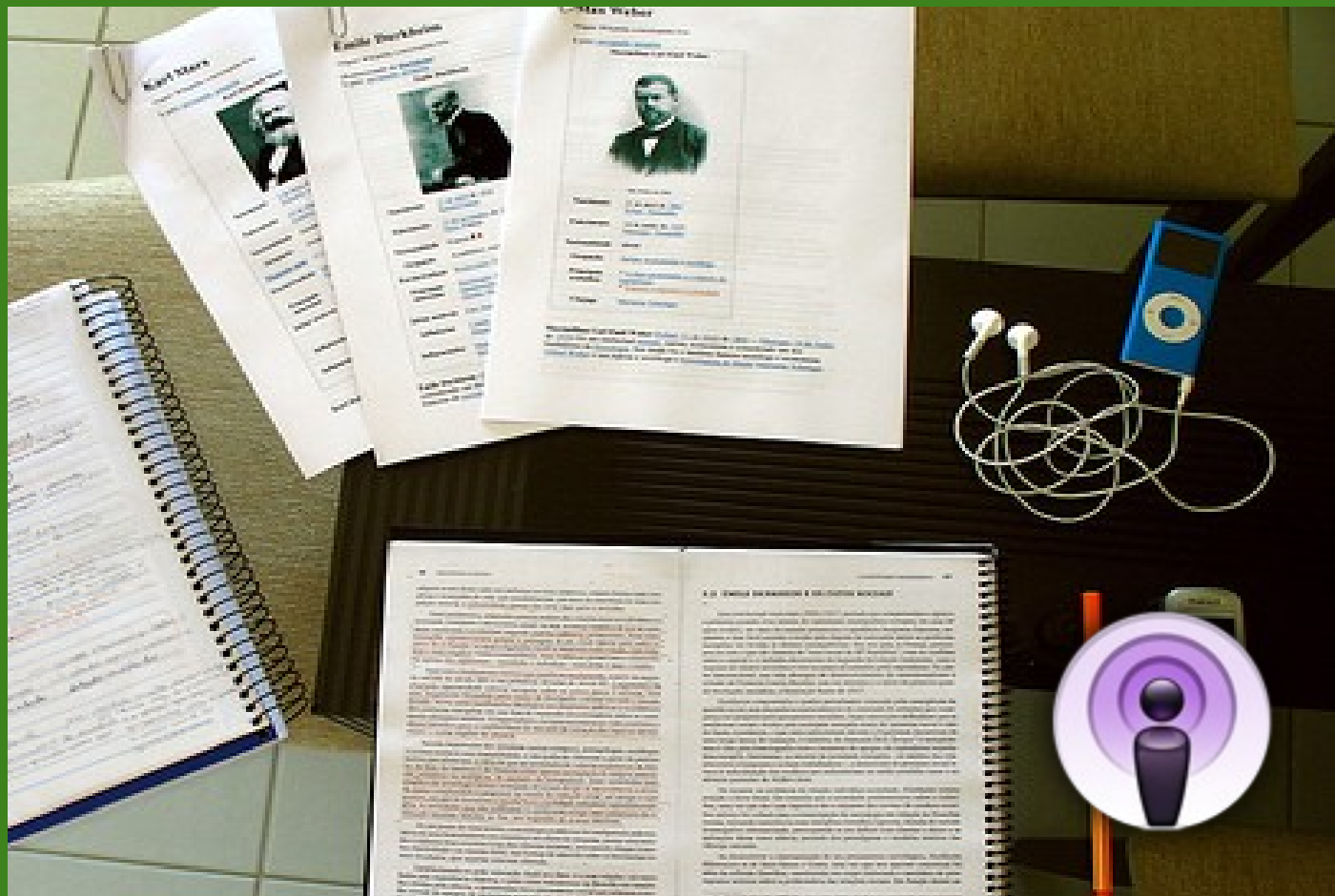
Parole chiave:
Java

Corso di Laurea:
Informatica

Insegnamento:
Linguaggi di Programmazione II

Email Docente:
faella.didattica@gmail.com

A.A. 2009-2010



Clonazione di oggetti

- Per “clonazione” si intende la creazione di un oggetto uguale ad uno esistente, ma **indipendente** da esso
- Ovvero, solitamente si desidera che le modifiche eventualmente apportate al clone non abbiano effetto sull'oggetto clonato

- Il linguaggio Java suggerisce un modo standard di clonare gli oggetti: il metodo clone della classe Object

```
protected Object clone() throws CloneNotSupportedException
```

- Tale metodo effettua una cosiddetta “copia superficiale” dell'oggetto sul quale viene chiamato
  - cioè, ogni campo dell'oggetto in questione viene copiato nel nuovo oggetto tramite una semplice assegnazione
  - quindi, nel caso ci sia un campo x di tipo riferimento (o array), nel clone il campo x farà riferimento allo stesso oggetto (o array) cui faceva riferimento nell'oggetto clonato
  - il più delle volte, questo comportamento **non è quello desiderato**, in quanto contrasta con il principio che il clone abbia vita indipendente dall'originale
  - a tale scopo, è utile ridefinire il metodo clone nelle proprie classi, realizzando una copia “profonda” anziché “superficiale”
  - la copia profonda consiste nel clonare ricorsivamente gli oggetti puntati dai propri campi

Visibilità di clone

- Si dice che un membro protected è visibile nel pacchetto e nelle sottoclassi
- In realtà, la regola di visibilità è più complicata
  - un membro protected di una classe A è visibile in una classe B se:
    1) B si trova nello stesso pacchetto di A, oppure
    2) B estende (anche indirettamente) A, e l'accesso al membro in questione avviene tramite un riferimento di tipo dichiarato B o suo sottotipo

- Questo fa sì che una sottoclasse non possa accedere ai membri protected degli oggetti della sua superclasse
- Ovvero, le funzionalità protected sono disponibili solo per gli oggetti su cui la sottoclasse ha effettivamente responsabilità, cioè per le sue istanze

- Nota: un membro statico protected è accessibile nel pacchetto e in tutte le sottoclassi, senza la restrizione di cui sopra

Visibilità di clone

- Consideriamo una classe A, che non ridefinisce il metodo clone, e un'altra classe Test
  - Supponiamo che la classe Test voglia clonare un oggetto di tipo A
  - Naturalmente, Test estende Object
  - Tuttavia, Test non può chiamare il metodo clone (di Object) su un oggetto di tipo A
  - Test potrebbe chiamare il metodo clone di Object solo su oggetti di tipo Test, o suoi sottotipi

- Quindi, perché una classe sia effettivamente clonabile, essa deve **ridefinire** il metodo clone, rendendolo **pubblico**

Clausola “throws” di clone

- L'eccezione CloneNotSupportedException (abbreviando, CNSE), che clone dichiara di lanciare, è verificata
- Come vedremo, clone lancia questa eccezione se l'oggetto this non implementa l'interfaccia Cloneable
- Il fatto che l'eccezione sia verificata serve ad attirare l'attenzione del programmatore sulla delicatezza della clonazione
- L'eccezione verificata, insieme alla corrispondente interfaccia e alla visibilità protected, fa si che attivare correttamente la clonazione richieda una serie di passi che vanno accuratamente programmati

- Se, nel ridefinire il metodo clone, eliminiamo dalla nuova versione la dichiarazione “throws CNSE”, per le regole dell'overriding nessuna futura sottoclasse potrà ridefinire il metodo clone in modo da lanciare quell'eccezione
  - Così facendo, impediamo alle nostre future sottoclassi di disabilitare la clonazione
- D'altra parte, mantenere la clausola “throws” comporta un fastidio per i client, che devono gestire l'eccezione anche se essa non verrà lanciata
- Sta al progettista decidere di volta in volta quale aspetto far prevalere

L'interfaccia Cloneable

- Il metodo clone di Object controlla preliminarmente che l'oggetto su cui è stato chiamato implementi l'interfaccia Cloneable
  - in caso negativo, viene lanciata l'eccezione verificata CloneNotSupportedException

- L'interfaccia Cloneable è un'interfaccia "di tag" (*tag interface*), ovvero è completamente vuota
- Le interfacce di tag servono solo a specificare che una certa classe gode di una certa proprietà
- In questo caso, essa serve a indicare che la classe supporta la clonazione

- Altre due interfacce di tag sono *RandomAccess* e *Serializable*

Copia superficiale e copia profonda

- Consideriamo una classe Employee, con campi **nome** (Stringa), **salario** (int), **data di assunzione** (java.util.Date) e **capoufficio** (un altro Employee)
- Come va clonato un oggetto Employee?
  - in realtà, la risposta dipende dal contesto applicativo
  - delineiamo comunque una proposta plausibile di clonazione

- il campo **salario** non presenta alcuna difficoltà
  - essendo di un tipo primitivo, è sufficiente copiarlo tramite assegnazione
  - quindi: copia **superficiale**

- il campo **nome** è di tipo riferimento
  - siamo tentati di effettuare una copia profonda
  - tuttavia, gli oggetti String sono immutabili
  - non c'è alcun rischio nel condividere lo stesso oggetto String tra clone e oggetto originale
  - quindi: copia **superficiale**

Copia superficiale e copia profonda

- il campo **"data di assunzione"** punta ad un oggetto di tipo java.util.Date
  - uni sguardo alla documentazione di quella classe ci dice che i suoi oggetti sono mutabili
  - cioè, la classe offre dei metodi modificatori per cambiare una data
  - per essere indipendente dall'originale, l'oggetto clonato dovrà avere una copia della data di assunzione dell'originale
  - quindi: copia **profonda**

- il campo **capoufficio** punta ad un altro Employee
  - supponiamo che gli Employee siano mutabili
  - siamo tentati di effettuare una copia profonda
  - tuttavia, il capoufficio dell'oggetto clonato dovrebbe essere effettivamente la stessa persona che è capoufficio dell'originale
  - inoltre, sarebbe assurdo che la classe Employee offrisse metodi per modificare i dati del proprio capoufficio
  - più probabilmente, essa offre un metodo per cambiare integralmente il proprio capoufficio, assegnando l'impiegato ad un altro supervisore
  - per questi motivi, e in mancanza di ulteriori informazioni sul contesto, è maggiormente indicata la copia **superficiale**

Copia superficiale e copia profonda

- In definitiva, otteniamo la seguente implementazione

```
class Employee implements Cloneable {
...
   public Employee clone() throws CloneNotSupportedException {
     Employee e = (Employee) super.clone();
     e.hireDate = (Date) hireDate.clone();
     return e;
   }
}
```

- Notiamo che, da Java 1.5, è possibile specializzare il tipo restituito da clone, passando da Object a Employee
- Iniziamo chiamando il metodo clone di Object, che effettua una preliminare copia superficiale
- Il cast è necessario e sicuro, perché sappiamo che this è di tipo Employee (o eventualmente sua sottoclasse)
- Poi, procediamo con la copia profonda per quei campi che, in base all'analisi precedente, lo richiedano
- In questo caso, l'unico campo che richiede copia profonda è la data di assunzione
- Un rapido controllo al manuale della libreria standard ci informa che la classe java.util.Date è clonabile, cioè implementa Cloneable ed è dotata di un metodo clone pubblico
    - tuttavia, il metodo clone di Date restituisce un semplice Object, e quindi necessita di un cast

Clonazione ed ereditarietà

- Il metodo clone **non dovrebbe utilizzare un costruttore** per creare il nuovo oggetto da restituire
- Infatti, in presenza di sottoclassi, questo porta alla creazione di un clone di tipo errato

- Ad esempio, supponiamo che Employee abbia la seguente versione di clone

```
public Employee clone() throws CloneNotSupportedException {
   Employee e = new Employee(salary, name, (Date) hireDate.clone(), boss);
   return e;
}
```

- Consideriamo la classe **Manager**, sottoclasse di **Employee**
- Il metodo clone di Manager non può utilizzare quello di Employee

```
public Manager clone() throws CloneNotSupportedException {
   Manager m = (Manager) super.clone(); // ERRORE a run-time!
   ...
   return m;
}
```

- Il metodo qui sopra solleva ClassCastException perché l'oggetto restituito da super.clone() è di tipo effettivo Employee
- Se invece il metodo clone di Employee fosse stato implementato come nella slide precedente, questo approccio avrebbe funzionato

Controindicazioni di clone

- Come si è visto, una corretta implementazione di clone presenta parecchi aspetti delicati

- Inoltre, la clonazione tramite clone non è compatibile con i campi **final** che richiedano una copia **profonda**
- Infatti, se l'implementazione di clone richiama quella presente nella classe Object, quei campi si troveranno copiati superficialmente, ed essendo final non sarà possibile correggerli in una copia profonda

- Alcuni esperti suggeriscono di non utilizzare affatto il metodo clone
- Piuttosto, si suggerisce di clonare gli oggetti tramite un *costruttore di copia* o un *metodo fabbrica*

- Per approfondire l'argomento, si consulti il testo:

Effective Java, di J. Bloch, Item 11

“I think clone is deeply broken”, J. Bloch