UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Universita degli Studi di Salerno
Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2023-2024

Costruttori e Garbage Collector

Docente: Ing. Massimo Ficco

E-mail: *mficco@unisa.it*

Il costruttore

La maggior parte degli errori vengono fatti perché ci si dimentica di inizializzare le variabili

Per inizializzare un oggetto:

- Un metodo ad hoc ?
- Il costruttore permette di non dividere creazione ed inizializzazione dell'oggetto

Il costruttore in Java

In Java il costruttore:

- Viene creato di default se non definito
- Deve avere lo stesso nome della classe
- Non ritorna niente (neanche void)
- Può avere o non dei parametri
- Viene richiamato da new che restituisce un riferimento all'oggetto chiamato

Albero a = new Albero()

Il costruttore di default

Se definisco un costruttore senza parametri l'operatore new utilizza quello da me definito

Se non definisco un costruttore senza parametri posso usare quello di default di java che alloca semplicemente l'oggetto

Esempio 1:

```
public class Albero {
  int altezza=0;
  public Albero ()
        {
              System.out.println("Costruito un albero di
              altezza"+altezza);
        }
}
```

Esempio 2

```
public class Albero{
  int altezza=0;

  public Albero (){System.out.println("Costruito un albero di
                    altezza"+altezza;}

  public Albero (int h)
   {
     altezza=h;
     System.out.println("Costruito un albero di
          altezza"+altezza);
   }
}
```

Overload del Costruttore

```
public class ShirtTest {

  public static void main (String args[ ]) {
     Shirt  shirtFirst = new Shirt ( );
     Shirt  shirtSecond = new Shirt ('G');
     Shirt  shirtThird = new Shirt ('B', 1000);
     //...
  }
}
```

```
public class Shirt {

  //...
  public Shirt ( ) {
    colorCode = 'R';
  }

  public Shirt (char startingCode) {
    colorCode = startingCode;
  }

  public Shirt (char startingCode, int startingQuantity) {
    colorCode = startingCode;
    quantityInStock = startingQuantity;
  }
  //...
```

Il distruttore

Il distruttore

Gli oggetti vengono creati dinamicamente

Finito l'ambito di visibilità lo spazio di memoria associato deve essere deallocato

Nel caso di JAVA si occupa di questo il *"**garbage collector**"*

- Un oggetto può essere deallocato quando nessun riferimento punta più a quell'oggetto
- Non è determinabile a priori il momento in cui interverrà il *garbage collector*

Garbage collector

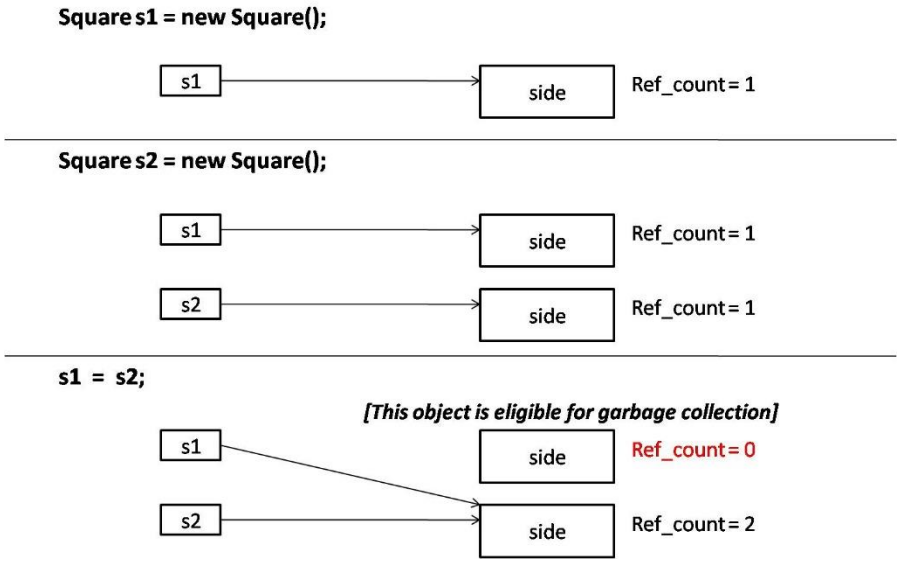
Il Garbage Collector è un thread Daemon che sta in esecuzione in background. Fondamentalmente, libera la memoria heap distruggendo gli oggetti non più raggiungibili, ovvero quelli a cui non fa più riferimento alcuna parte del programma. Per far ciò implementa una soluzione chiamata

Reference Counting:

In memoria heap ogni oggetto ha un contatore dei reference che vi fanno riferimento.

Il distruttore Esempio.1

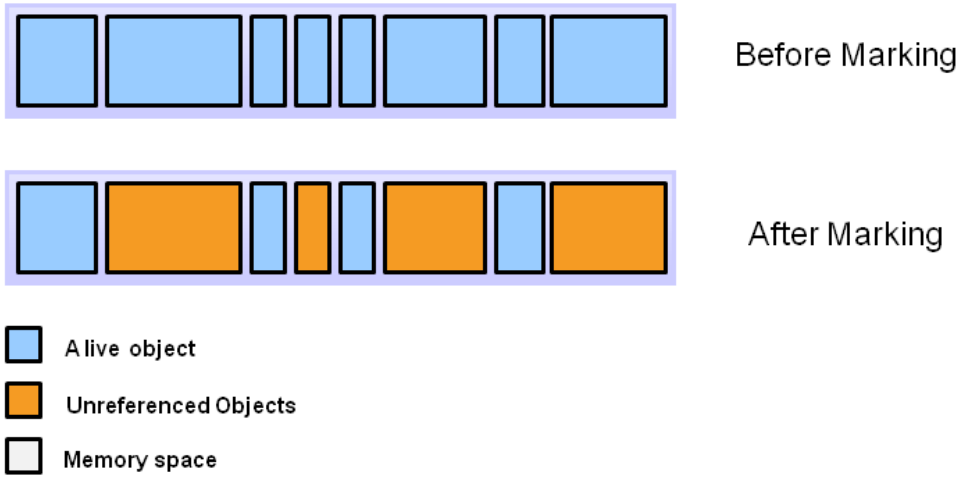
String s1=new String(“casa1”);
String s2=new String(“casa2”);
s1=s2;
s1=null; ***//s2 punta ancora all’oggetto***
s2=null; ***//oggetto può essere distrutto***



Gli oggetti nell’heap il cui ***Ref_count*** è nullo sono candidati alla deallocazione per effetto della Garbage Collection, che è un processo automatico di osservazione della memoria Heap, che identifica o meglio “marchia” gli oggetti irraggiungibili li distrugge e compatta la memoria liberata per un uso successivo più efficiente.

Garbage Collection

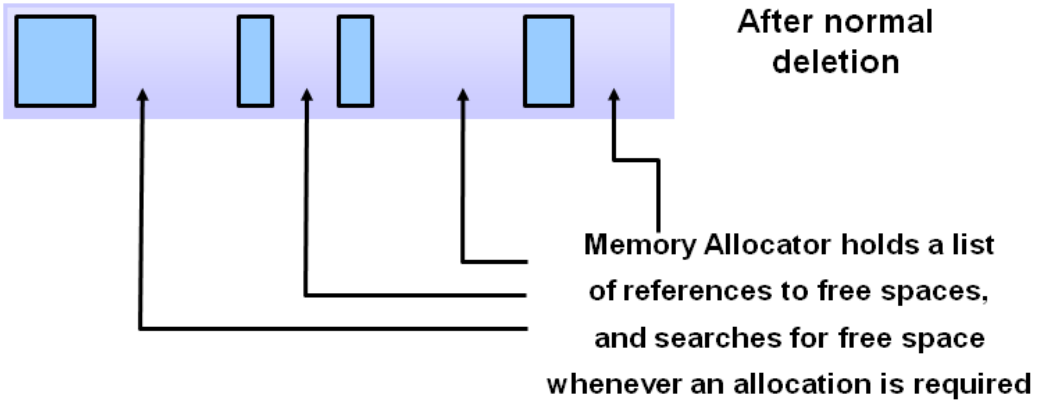
- Il **primo passo** nel processo di Garbage Collection è chiamato marking, e si occupa di identificare quali pezzi di memoria heap sono in uso e quali no, mendinate l'ispezione del Ref_counter dei vari oggetti.



- Tutti gli oggetti vengono scansionati nella fase di marcatura per effettuare questa discriminazione. Questo può essere un processo che richiede molto tempo se tutti gli oggetti in un sistema devono essere sottoposti a scansione

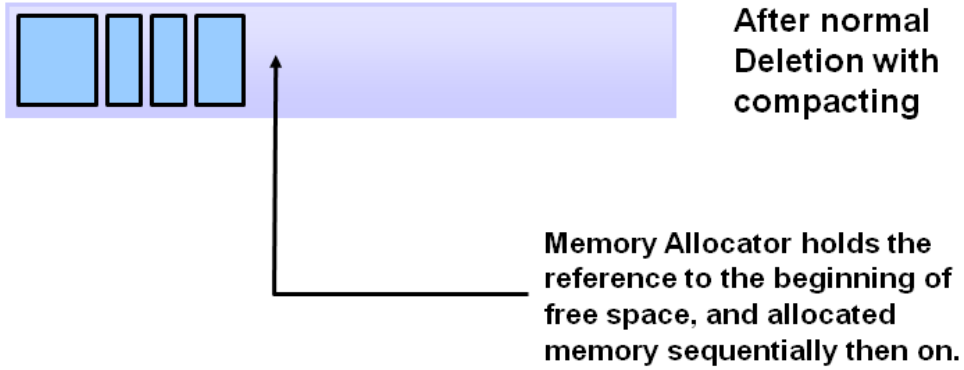
Garbage Collection

- Il secondo passo è la cancellazione normale dove vengono rimossi gli oggetti senza riferimento e lasciati gli oggetti referenziati e i puntatori allo spazio libero. **L'allocatore di memoria** contiene i riferimenti ai blocchi di spazio libero in cui è possibile allocare nuovi oggetti.



Garbage Collection

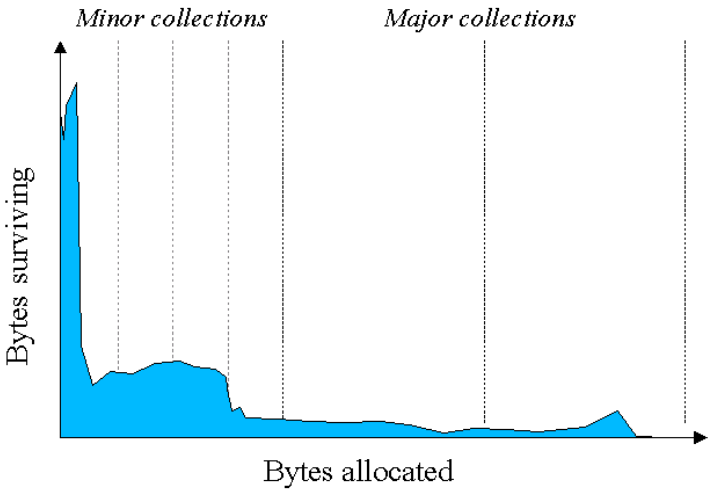
- Il secondo passo è la cancellazione normale dove vengono rimossi gli oggetti senza riferimento e lasciati gli oggetti referenziati e i puntatori allo spazio libero. **L'allocatore di memoria** contiene i riferimenti ai blocchi di spazio libero in cui è possibile allocare nuovi oggetti.

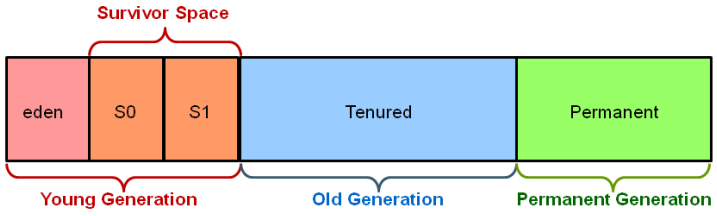


- A completare il passo 2 e per migliorare ulteriormente le prestazioni, oltre all'eliminazione di oggetti senza riferimento, è possibile compattare anche gli oggetti riferiti rimanenti. Spostando insieme gli oggetti referenziati rende le nuove allocazioni di memoria molto più semplici e veloci.

Garbage Collection

- Questo approccio causa una crescente latenza man mano che aumenta il numero di oggetti, poiché deve passare attraverso l'intero elenco di oggetti, cercando gli oggetti irraggiungibili.





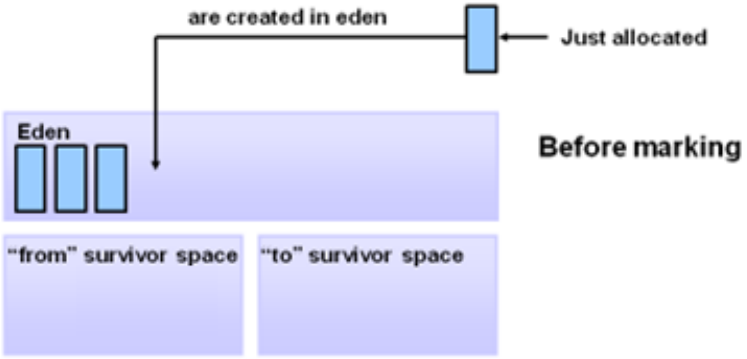
- Da analisi empiriche si è visto che la maggior parte degli oggetti ha vita breve, e ciò può sfruttato per migliorare le prestazioni della JVM creando una metodologia denominata Garbage Collection Generazionale.
  - Lo spazio di memoria heap è diviso in generazioni: Generazione giovane, Generazione vecchia e Generazione permanente, quest'ultima presente fino a Java 7.

Garbage Collection

- La Generazione Giovane contiene tutti i nuovi oggetti creati. Una volta riempito, si attiva una ***minor garbage collection*** (nota anche come GC secondario) che distrugge tutti gli oggetti non più riferiti di questa generazione.
  - Dato l'alto "tasso di mortalità" degli oggetti di questa generazione e il dover ispezionare una piccola fetta di memoria heap, tale operazione è di norma veloce.
  - Gli oggetti sopravvissuti vengono trasferiti nella Vecchia Generazione, che accoglie anche gli oggetti il cui tempo di vita ha superato una determinata soglia. Anche nella Vecchia Generazione avviene un recupero della memoria, eseguendo una ***major garbage collection***.
  - Tutte le minor Garbage Collection sono eventi "Stop the World". Ciò significa che tutti i thread dell'applicazione vengono interrotti fino al completamento dell'operazione.

Garbage Collection

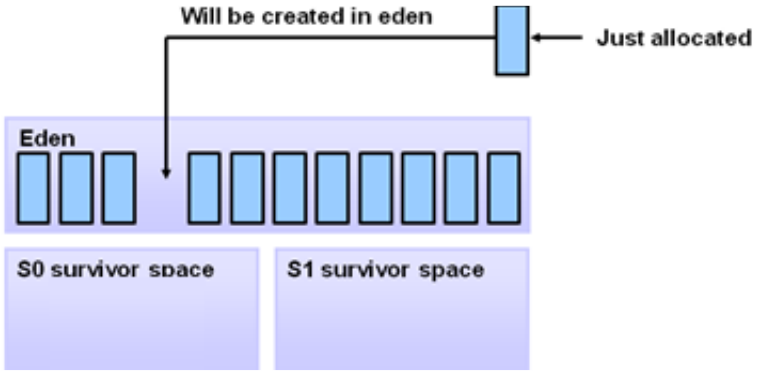
- Una major GC spesso è molto più lenta di una minor GC in quanto coinvolge tutti gli oggetti ancora “vivi”. Pertanto, il suo rate di attivazione è più basso rispetto ad una minor. Anche in questo caso si tratta di un evento "Stop the World".
  - Fino a Java7 di Java esisteva un’altra area dello heap chiamata Permanent Generation che conteneva metadati richiesti dalla JVM per descrivere le classi e i metodi usati nell’applicazione. Questa area di memoria veniva popolata dalla JVM a runtime con le classi utilizzate nell’applicazione e le classi e metodi del core di Java SE. È stata rimossa in Java 8.



- Innanzitutto, tutti i nuovi oggetti vengono assegnati allo spazio dell'eden.
- Entrambi gli spazi sopravvissuti sono vuoti.

Garbage Collection

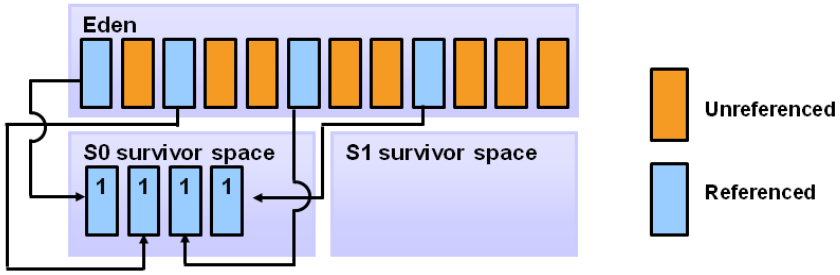
- Una major GC spesso è molto più lenta di una minor GC in quanto coinvolge tutti gli oggetti ancora “vivi”. Pertanto, il suo rate di attivazione è più basso rispetto ad una minor. Anche in questo caso si tratta di un evento "Stop the World".
  - Fino a Java7 di Java esisteva un’altra area dello heap chiamata Permanent Generation che conteneva metadati richiesti dalla JVM per descrivere le classi e i metodi usati nell’applicazione. Questa area di memoria veniva popolata dalla JVM a runtime con le classi utilizzate nell’applicazione e le classi e metodi del core di Java SE. È stata rimossa in Java 8.



- Quando lo spazio eden si riempie, viene attivata una minor GC.

Garbage Collection

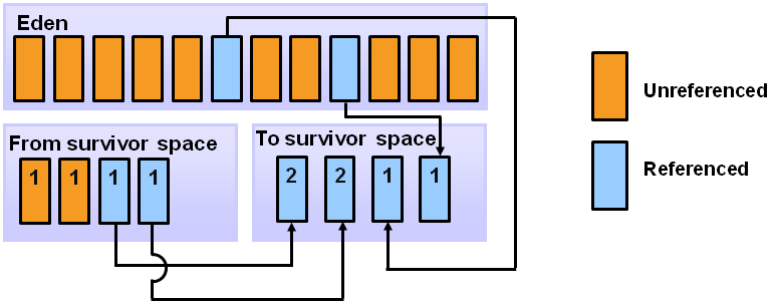
- Una major GC spesso è molto più lenta di una minor GC in quanto coinvolge tutti gli oggetti ancora “vivi”. Pertanto, il suo rate di attivazione è più basso rispetto ad una minor. Anche in questo caso si tratta di un evento "Stop the World".
  - Fino a Java7 di Java esisteva un’altra area dello heap chiamata Permanent Generation che conteneva metadati richiesti dalla JVM per descrivere le classi e i metodi usati nell’applicazione. Questa area di memoria veniva popolata dalla JVM a runtime con le classi utilizzate nell’applicazione e le classi e metodi del core di Java SE. È stata rimossa in Java 8.



- Gli oggetti referenziati vengono spostati nel primo spazio sopravvissuto.
- Gli oggetti senza riferimenti vengono eliminati quando lo spazio eden viene cancellato.

Garbage Collection

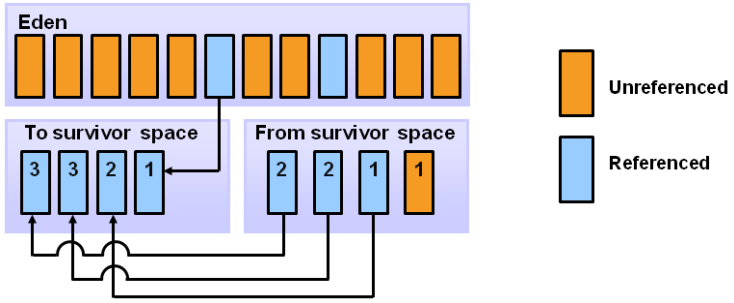
- Una major GC spesso è molto più lenta di una minor GC in quanto coinvolge tutti gli oggetti ancora “vivi”. Pertanto, il suo rate di attivazione è più basso rispetto ad una minor. Anche in questo caso si tratta di un evento "Stop the World".
  - Fino a Java7 di Java esisteva un’altra area dello heap chiamata Permanent Generation che conteneva metadati richiesti dalla JVM per descrivere le classi e i metodi usati nell’applicazione. Questa area di memoria veniva popolata dalla JVM a runtime con le classi utilizzate nell’applicazione e le classi e metodi del core di Java SE. È stata rimossa in Java 8.



- Al successivo GC minore gli oggetti referenziati in Eden e nel primo spazio sopravvissuto (S0) vengono spostati nel secondo spazio sopravvissuto (S1), con età incrementata.
- Sia S0 che eden vengono cancellati.

Garbage Collection

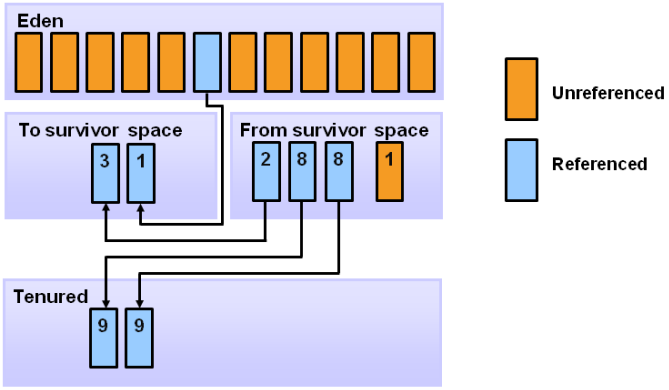
- Una major GC spesso è molto più lenta di una minor GC in quanto coinvolge tutti gli oggetti ancora “vivi”. Pertanto, il suo rate di attivazione è più basso rispetto ad una minor. Anche in questo caso si tratta di un evento "Stop the World".
  - Fino a Java7 di Java esisteva un’altra area dello heap chiamata Permanent Generation che conteneva metadati richiesti dalla JVM per descrivere le classi e i metodi usati nell’applicazione. Questa area di memoria veniva popolata dalla JVM a runtime con le classi utilizzate nell’applicazione e le classi e metodi del core di Java SE. È stata rimossa in Java 8.



- Al successivo minor GC, si ripete lo stesso processo, invertendo gli spazi dei sopravvissuti. Gli oggetti referenziati vengono spostati in S0.
- Gli oggetti sopravvissuti sono invecchiati.
- Eden e S1 sono cancellati.

Garbage Collection

- Una major GC spesso è molto più lenta di una minor GC in quanto coinvolge tutti gli oggetti ancora “vivi”. Pertanto, il suo rate di attivazione è più basso rispetto ad una minor. Anche in questo caso si tratta di un evento "Stop the World".
  - Fino a Java7 di Java esisteva un’altra area dello heap chiamata Permanent Generation che conteneva metadati richiesti dalla JVM per descrivere le classi e i metodi usati nell’applicazione. Questa area di memoria veniva popolata dalla JVM a runtime con le classi utilizzate nell’applicazione e le classi e metodi del core di Java SE. È stata rimossa in Java 8.



- Dopo un minor GC, quando gli oggetti raggiungono una certa soglia di età (8 ad es.) vengono promossi da giovane generazione a vecchia generazione.
- Alla fine, verrà eseguita una major GC sulla vecchia generazione che ripulisce e compatta quello spazio.

Garbage Collection

- **Concurrent Mark Sweep (CMS) Collector** - tenta di minimizzare le pause dovute alla Garbage Collection facendo in modo che la maggior parte della garbage collection lavori contemporaneamente ai thread dell'applicazione. Normalmente CMS non copia o compatta gli oggetti senza interrompere i thread applicativi, questo può causare una frammentazione eccessiva della memoria che può essere compensata allocando un heap più grande. CMS deve essere utilizzato per applicazioni che richiedono tempi di pausa bassi e che possono condividere risorse con il GC. Per abilitare CMS utilizzare: -XX:+UsaConcMarkSweepGC e per impostare il numero di thread utilizzare: -XX:ParallelCMSThreads=<n>
- **Garbage-First (G1) Collector** - progettato per dimensioni di heap superiori ai 4 GB, e divide l'heap in regioni che vanno da 1 a 32 Mb. Esiste una fase di marcatura globale al termine del quale, G1 sa le regioni vuote e può raccogliere prima gli oggetti irraggiungibili da queste regioni, da questo il nome Garbage-First. G1 utilizza anche un modello di previsione.

Operazioni di finalizzazione

Java non distrugge gli oggetti quando il loro ambito di visibilità è terminato. Alla fine verrà distrutto, ma non sappiamo esattamente quando. Molto probabilmente verrà distrutto molto tempo dopo.

In ogni caso, se serve effettuare delle operazioni prima che l'oggetto venga distrutto è possibile definire un metodo **finalize**:

- Esso viene richiamato appena prima l'operazione di deallocazione

Distruttore esempio .2

```
class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {    checkedOut = checkOut;  }
  void checkIn() {    checkedOut = false;  }
  public void finalize() {  if(checkedOut)   System.out.println("Error: checked out");}
}

public class TerminationCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    novel = new Book(true);
    Book poem = new Book(true);
    novel = poem;
    .......
    // Force garbage collection & finalization:
    // System.gc();
  }
} ///:~
```

Un esempio significativo !

```
public class Oggetto{
   static int n_obj=0;
   public Oggetto() {  n_obj++; }
   public void finalize() { n_obj--; }
}
```

La variabile **n_obj** conta il numero di istanze allocate per la classe oggetto !!!

In sintesi
Persistenza e visibilità

Gestione della Memoria in Java

- Nonostante i vantaggi che apportano, i finalizzatori presentano molti inconvenienti.
  - Il primo problema evidente è che non è possibile sapere quando viene eseguito un finalizzatore poiché la Garbage Collection può verificarsi in qualsiasi momento. Le risorse di sistema non sono illimitate. Pertanto, potrebbe essere possibile esaurirle prima che avvenga una pulizia.
  - I finalizzatori hanno anche un impatto sulla portabilità del programma. Poiché l'algoritmo di Garbage Collection dipende dall'implementazione della JVM, un programma può funzionare molto bene su un sistema mentre si comporta in modo diverso su un altro.
  - Il costo delle prestazioni è un altro problema significativo: la JVM deve eseguire molte più operazioni durante la distruzione di oggetti contenenti un finalizzatore non vuoto.
  - L'ultimo problema è la mancanza di gestione delle eccezioni: se un finalizzatore genera un'eccezione, il processo di finalizzazione si interrompe lasciando l'oggetto in uno stato improprio.

Persistenza e visibilità

Una variabile di un metodo è valida fino a quando non termina il metodo.

Un attributo di un oggetto è valido fino a quando è vivo l'oggetto

- L'oggetto viene creato
- **L'oggetto viene distrutto** e la memoria che impegna deallocata quando nessun riferimento punta più a quell'oggetto

Conclusione

- Il **garbage collector** deve rendersi conto di quali oggetti non sono più referenziati e liberarne lo spazio sotteso. Inoltre, deve combattere la frammentazione dovuta alle continue allocazioni e deallocazioni durante il ciclo di vita di un programma
  - Vantaggio: Maggiore produttività del programmatore – Garanzie di integrità: un programmatore non può accidentalmente (o maliziosamente) causare un crash della JVM liberando incorrettamente della memoria.
  - Svantaggio: il garbage collector implica un considerevole overhead che può inficiare le performance del programma sviluppato.