Java

inizializzazioni e pulizia

G. Prencipe
prencipe@di.unipi.it

Introduzione

- Uno dei problemi principali nella programmazione è legato alla “sicurezza”
- Due principali aspetti di questo problema sono *l’inizializzazione* e la *pulizia*
  - Molti problemi derivano da dimenticanze nell’inizializzare variabili
  - Inoltre è facile dimenticare di eliminare elementi nel programma quando non sono utilizzati più
    - Come conseguenza le risorse a disposizione, come la memoria, rischiano di esaurirsi rapidamente

Introduzione

- Per ovviare a questi problemi Java utilizza
  - *Costruttore* (come in C++)
    - Un metodo speciale che viene automaticamente invocato al momento della creazione di un oggetto
  - *Garbage collector*
    - Rilascia automaticamente la memoria quando non più utilizzata

Costruttore

- In Java l’inizializzazione degli oggetti al momento della loro creazione viene effettuata automaticamente tramite un metodo speciale detto *costruttore*
- In questo modo l’utente non deve ricordarsi di fare nulla al momento della creazione dell’oggetto

Costruttore

- Il costruttore viene invocato automaticamente al momento della creazione dell'oggetto (**new**)
- Quale nome viene dato a questo metodo speciale? Ci sono due problemi
  - Qualsiasi nome utilizzato potrebbe andare in conflitto con i nomi utilizzati per altri metodi nella classe
  - Dato che il compilatore è responsabile di invocare il costruttore, deve sempre conoscere esattamente quale metodo invocare

Costruttore

- La soluzione (adottata già nel C++) è di adottare come nome per il *costruttore lo stesso della classe*
  - In questo modo ha più senso che questo metodo venga automaticamente invocato

Costruttore -- esempio

```
class Rock {
  Rock(){ <-- Costruttore
    System.out.println("Rock");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock(); <-- Come risultato, con la new, verrà stampata 10 volte la parola Rock
    }
}
```

Costruttore -- esempio

- Quindi, quando l'oggeto **Rock** viene creato, si ha la garanzia che l'inizializzazione viene portata a termine correttamente
- Nota: il nome del costruttore deve essere *esattamente lo stesso* della classe
  - Quindi per il costruttore non si applica lo standard adottato in genere per i metodi, secondo cui la prima lettera del nome assegnato al metodo è minuscola

Costruttore

- Come ogni metodo, il costruttore può avere argomenti con cui specificare *come* creare l'oggetto
- Vediamo un esempio

Costruttore -- esempio

```
class Rock2 {
  Rock2(int i) {
    System.out.println("Rock" + i);
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
    }
}
```

Costruttore

- Il costruttore *non ha* valore di ritorno
  - È diverso da avere **void** (il metodo restituisce *nulla*)
  - Il costruttore non restituisce niente e non si ha la scelta di fargli restituire qualcosa
  - Con **new** otteniamo un riferimento a un oggetto, ma il costruttore non ha valore di ritorno

Overloading

- Gli oggetti e i metodi vengono riferiti tramite *nomi*
- Quindi è importante scegliere nomi significativi che aiutino nella leggibilità del programma
- Spesso, nell'utilizzo comune, i nomi assumono più significati
  - Es. lava macchina, lava cane, lava maglietta
  - Se volessimo mappare queste tre funzionalità diverse in un programma, dovremmo adottare tre nomi distinti

Overloading

- Questo accadeva ad esempio in C
  - Funzioni per stampare un intero o una stringa devono avere nomi diversi e unici
- In Java (e in C++) viene fornito un meccanismo che permette di assegnare a uno stesso nome più significati: *overloading*
- L'overloading si ottiene avendo metodi diversi con lo stesso nome ma che accettano parametri di tipo diverso (cioè, con segnature diverse)

Overloading

- Questo torna particolarmente utile con i costruttori
- Infatti, il nome della classe è unico, quindi senza overloading potremmo avere solo un costruttore
- Cioè, non sarebbe possibile differenziare le inizializzazioni degli oggetti a seconda dei casi

Esempio -- overloading

```
class Tree {
  int height;

Tree() {
    System.out.println(“Pianto un seme”);
    height = 0;
  }

  Tree(int i) {
    System.out.println("Creo un albero di altezza"+i);
    height = i;
  }                                  // continua
```

Esempio -- overloading

```
void info() {
    System.out.println(”L’albero è alto " + height);
  }
  void info(String s) {
    System.out.println(s + ": L’albero è alto "+ height);
  }
}                  // Fine classe Tree
```

Esempio -- overloading

```
public class Overloading { // Utilizzo della classe Tree
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("metodo con overloading");
    }
    // Overloading del costruttore:
    new Tree();
}
```

Overloading

- Per distinguere i metodi in overloading, ognuno di essi deve avere *un'unica* lista di argomenti
- Anche una semplice *differenza nell'ordine degli argomenti* è sufficiente per la distinzione
  - È preferibile non adottare questo approccio, che produce difficoltà nella lettura

Esercizio

- Provare a scrivere una classe OverloadingOrder che contiene due metodi **print** e il **main**
- Due metodi **print**
  - Il primo prende come argomenti una **String**a e un **int**
  - Il secondo un **int** e una **String**a
  - Entrambi stampano a console il valore dei due argomenti
- Il **main** invoca le due **print**

Overloading e tipi primitivi

- Un tipo primitivo può essere automaticamente promosso da un tipo "più piccolo" a uno "più grande
  - Cioè, è possibile passare un argomento più piccolo di quello atteso
  - In questo caso verrà invocato il metodo che accetta come parametro il tipo "più vicino"
- Vediamo l'esempio **PrimitiveOverloading.java**

Overloading e tipi primitivi

- Se invece l'argomento passato è "più grande" di quello atteso dal metodo, bisogna fare esplicitamente un cast, altrimenti si ottiene un errore in compilazione
  - Facendo un cast in questo modo (da un tipo "più grande" a uno "più piccolo") vengono perse informazioni
  - È per questo che il compilatore *forza* la conversione
- Vediamo l'esempio **Demotion.java**

Overloading sui tipi di ritorno

- Perché non fare overloading anche sui tipi di ritorno di un metodo?
- Non potrebbe anche questo essere un modo per distinguere due metodi con lo stesso nome?

```
void f() {}
```

e

```
int f() {}
```

- Domanda: dov'è il problema?

Overloading sui tipi di ritorno

- Il problema sta nel fatto che un metodo può essere invocato e il suo valore di ritorno ignorato
  - Si invoca il metodo solo per i suoi *effetti collaterali*
- Ad esempio, la seguente invocazione

```
f() {}
```

  a quale definizione di **f()** si riferisce?
- Quindi, niente overloading sui tipi di ritorno

Costruttori di default

- Un *costruttore di default* è un costruttore che non prende argomenti
- Se non inseriamo esplicitamente un costruttore nella definizione di una classe, allora ci penserà il compilatore, inserendone uno di default

Esempio -- costruttori di default

- In questo esempio

```
class SenzaCostr{int i;}

public class CostrDef{
        public static void main (String[] args) {
                SenzaCostr = new SenzaCostr();
        }
}
```

  La **new** provoca l'invocazione del costruttore di default

Esempio -- costruttori di default

- Se però definiamo costruttori che prendono argomenti, e poi invochiamo un costruttore di default, otterremo un errore in compilazione

```
class NoDefault {
        NoDafault(int i) {}
        NoDefault(double d) {}
}
```

- Se invochiamo new NoDefault();
  Otterremo un errore!!
  - Visto che abbiamo definito dei costruttori, definiamo tutti quelli che occorrono!!

La parola chiave **this**

- Supponiamo di essere all'interno di un metodo, e vogliamo riferire l'oggetto corrente
  - Cioè l'oggetto all'interno del quale è definito il metodo stesso
- Per fare questo si ricorre alla pqarola chiave **this**
- **this**, *che può essere usato solo all'interno di un metodo*, produce un riferimento all'oggetto per cui li metodo è invocato

La parola chiave **this**

- Ad esempio, in

```
class Albicocca {
    void raccogliere() {/* …. */}
    void sbucciare () { raccogliere(); /* …. */}
}
```

  la chiamata a **raccogliere()** in **sbucciare()** potrebbe essere fatta anche con

```
this.raccogliere()
```

  (anche se in questo caso non sarebbe necessario)

La parola chiave **this**

- **this** viene usato in tutti quei casi in cui è necessario ottenere un *riferimento esplicito* all'oggetto corrente
- Vediamo un esempio in cui viene utilizzato nel comando **return**

Esempio -- la parola chiave **this**

```
public class Leaf {
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
}
```

Dato che **this** restituisce il riferimento all'oggetto, possono essere effettuate chiamate multiple a **increment()**

Costruttori da costruttori

- Quando si hanno più costruttori all'interno di una classe, si potrebbe voler invocare un costruttore da un altro costruttore
  - Per riutilizzare il codice
- Questo si può fare con **this**
- In questo caso, l'invocazione avviene tramite **this**, che invoca il costruttore che corrisponde agli argomenti passati
- Vediamo come esempio **Flower.java**

Costruttori da costruttori

- **this** non può essere usato due volte per chiamare un costruttore dall'interno dello stesso costruttore
- La chiamata al costruttore deve essere la prima cosa che si fa o si otterrà un errore in compilazione
- Non può essere usato (per invocare un costruttore) da un metodo che non sia un costruttore
- Notare l'uso di **this.s** per riferire una delle variabili della classe

static

- Un metodo **static** non ha **this**!!
- Non è possibile invocare un metodo non-**static**o da un metodo **static**o
  - Il compilatore non sa a tempo di compilazione (quando sono creati gli oggetti **static**) se in esecuzione verranno creati gli oggetti non-**static**i
  - È chiaramente possibile fare il contrario
- I metodi **static** sono simili alle funzioni *globali* del C
  - Alcuni sostengono che non sono propriamente nella logica OO, dato che non rientrano nell'ottica *dell'invio di messaggi* agli oggetti (non è necessario crearne per invocarli)
  - Quindi un utilizzo massiccio di questi metodi non va bene

Pulizia: finalize e GC

- Associato al GC, in Java c'è un metodo speciale chiamato **finalize()**, che è possibile definire in ogni classe
- Quando il GC è pronto a rilasciare la memoria occupata da un oggetto, chiama prima **finalize()**
- Alla prossima passata del GC rilascerà la memoria dell'oggetto
- Cioè, il GC permette di effettuare delle operazioni *finali*

GC

- Bisogna tenere a mente però che
  - *Gli oggetti potrebbero non essere raccolti dal GC*
- Infatti l'esecuzione del GC è a discrezione del sistema, che magari lo chiama solo se le risorse sono molto limitate o vicino all'esaurimento
- Quindi non si può fare affidamento sul GC e **finalize()** per effettuare le *operazioni finali* (se necessarie)

GC

- Un altro punto del GC è che
  - *GC riguarda solo la memoria*
- Quindi, ogni attività collegata alla GB e al metodo **finalize()** deve riguardare solo la memoria
- Quindi se un oggetto contiene altri oggetti, **finalize()** deve rilasciare la loro memoria?
  - No!! A questo pensa il GC stesso e non è compito del programmatore

GC

- In effetti si trovano scarse applicazioni per il metodo **finalize()**
- Esiste nella possibiltà che vengano eseguite operazioni sulla memoria in stile C
  - Può accadere quando si invoca codice non-Java da codice Java
    - *Native methods*

GC

- Quindi, in generale non si può fare affidamento sul fatto che **finalize()** venga invocato in seguito alla GC, e se si vogliono effettuare operazioni di pulizia devono essere effettuate con esplicite chiamate a metodi specifici
- Esistono però delle situazioni in cui è possibile utilizzare proficuamente **finalize()** senza fare affidamento sul fatto che venga effettivamente invocato

Condizioni di terminazione

- Può essere utilizzato per controllare che l'oggetto *sia terminato* correttamente
  - Nel caso non fosse così, viene utilizzato per dare un errore
- Vediamo un esempio

Esempio

```
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");
  }
}
```

Condizione di terminazione: un oggetto **Book** deve subire il **checkIn()** prima di essere raccolto dal GC

Esempio

```
public class TerminationCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Facciamo il checkIn()
    novel.checkIn();
    // Oggetto creato male (il riferimento è perso)
    new Book(true); // Non viene fatto il checkIn()
    // Forza la garbage collection & finalization
    System.gc();
}
```

Un errore nel **main** crea un **Book** senza fare il **checkIn()**

Al momento della GC ci si accorge dell'errore grazie a **finalize()**

Inizializzazione dei membri

- Le variabili (primitive) locali ai metodi devono essere inizializzate esplicitamente, altrimenti si ottiene errore in compilazione. Con

```
void f() {
        int i;
        i++;
}
```

si ottiene errore!!

Inizializzazione dei membri

- Per le variabili (primitive) che sono membri della classe, non è necessaria l'inizializzazione esplicita
  - **boolean** false
  - **char** 0, che viene stampato come uno spazio
  - **byte**, **short**, **int**, **long** 0
  - **float**, **double** 0.0
- Per le variabili (non primitive) membri della classe che sono definite ma non inizializzate (è creato il riferimento ma non l'oggetto)
  - Automaticamente inizializzate a **null**

Inizializzazione

- Come si fa ad assegnare a una variabile un valore iniziale?
- Un modo è di assegnare il valore al momento della dichiarazione

```
int i=27;
```

- Nel caso di oggetti

```
Book = new Book();
```

  - Se l'oggetto non è stato inizializzato ma lo si usa lo stesso, si ottiene errore a tempo d'esecuzione

Inizializzazione

- Il costruttore può essere usato per effettuare le operazioni di inizializzaione
  - Questo fornisce molta flessibilità, dato che i valori di inizializzazione possono essere ottenuti a tempo d'esecuzione e poi passati al costruttore

Inizializzazione

```
class Counter {
      int i;
Counter() {i=7;}
}
```

- In questo caso, **i** viene prima inizializzata a 0 (automaticamente) e poi a 7 dal costruttore

Ordine di inizializzazione

- All'interno di una classe, l'ordine di inizializzazione è determinato dall'ordine con cui le variabili sono definite all'interno della classe
- Le variabli sono inizializzate prima che qualsiasi metodo venga chiamato
- Vediamo l'esempio **OrdineDiInizializzazione.java**

Inizializzazione static

- Il caso di variabili **static** è del tutto simile
- Bisogna ricordare solo viene allocato un unico pezzo di memoria per una variabile **static**, indipendentemente dal numero di oggetti creati

Inizializzazione di array

- Un *array* è una sequenza di oggetti o dati primitivi, tutti dello stesso tipo, e riuniti sotto lo stesso nome
- Sono definiti e utilizzati con **[]**
- Per definirne uno, possiamo scrivere

```
int[] a1;
```

o

```
int a1[];
```

Array

- Nell'esempio visto, non vengono specificate le dimensioni dell'array
- Quindi, a questo punto si ha un riferimento a un array, ma nessuno spazio allocato per esso
- Per allocare lo spazio bisogna scrivere qualche espressione di inizializzazione

```
int[] a1={1, 2, 3, 4, 5};
```

Array

- È possibile definire un array senza inizializzarlo
- Questo è utile quando si vuole assegnagli il riferimento di un array già esistente e inizializzato

```
int[] a2; a2=a1;
```

- Al solito, quello che avviene è la copia del riferimento

Esempio

```
public class Arrays {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println(
        "a1[" + i + "] = " + a1[i]);
    }
}
```

L'array è unico

I riferimenti due

Accesso agli elementi

Esempio

```
public class Arrays {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println(
        "a1[" + i + "] = " + a1[i]);
    }
}
```

Gli **array** hanno un membro intrinseco, **length**, che dice quanti elementi ha

Array

- Se si accedono elementi oltre i limiti dell'array, si ottengono errori a tempo d'esecuzione
  - **OutOfBoundException**
- L'inizializzazione può essere fatta anche con **new**

```
int[] a;
a=new int[20];
```

Oppure

```
int[] a = new int[20];
```

Esercizio

- Scrivere una classe **ArrayNew** con un **main** che crea un array di interi avente **n** elementi, con **n** un numero casuale tra 1 e 20
  - Utilizzare **nextInt()** della classe **Random**
  - Non scrivere nulla nell'array
- Scrivere poi un ciclo **for** che stampa gli elementi dell'array
- Da notare
  - Determinazione a run-time della dimensione dell'array
  - Valori di default inseriti nell'array (di tipo primitivo)

Array

- Nel caso di array di oggetti *non primitivi* l'utilizzo di **new** è obbligatorio
- In questo caso viene creato un array di riferimenti agli oggetti
  - Poi bisogna creare esplicitamente gli oggetti da collegare ai riferimenti

Esercizio

- Scrivere una classe **ArrayClassObj** con un **main** che crea un array di **Integer** avente dimensione casuale (come nell'esercizio precedente)
- Scrivere un *unico* **for** che inizializzi gli elementi nell'array (con interi casuali generati ancora con **nextInt**) e che stampi il contenuto dell'array
- Notare la conversione automatica da **Integer** a **String** al momento della stampa a console

Array

- Se ci si dimentica di creare gli oggetti nell'array, si ottiene un errore a run-time quando si accede la locazione vuota dell'array

Array -- iniz. alternative

```
Integer[] a = {
        new Integer(1);
        new Integer(2);
}

Integer[] b = new Integer[] {
        new Integer(1);
        new Integer(2);
}
```

Dimensione decisa staticamente a compilazione

Dimensione non decisa staticamente

Permette di definire array di dimensione non fissata

Esempio

- Un caso interessante è rappresentato da array di **Object**
- Vediamo l'esempio **VarArgs.java**
- Notare che nel caso di stampa di oggetti, se non è previsto un metodo che trasformi il contenuto in **String**a (chiamato tipicamente **toString()**), viene stampato il nome della classe seguito da @ e dall'indirizzo dell'oggetto

Array multidimensionali

- In Java gli array multidimensionali vengono facilmente creati
- Esempio

```
int[][] a1 = {
  {1, 2, 3, 4},
  {5, 6, 7, 8}
}
```

- Analizziamo **MultiDimArray.java**

Esercizi

1. Creare una classe con un costruttore di default che stampa un messaggio. Creare un oggetto di questa classe
2. Aggiungere un costruttore che prende una **String**a che viene stampata in aggiunta al messaggio precedente
3. Creare un array di riferimenti agli oggetti della classe creata al punto 2. Non creare gli oggetti
   1. Notare se i messaggi di inizializzazione dal costruttore vengono stampati o meno
4. Creare gli oggetti definiti nel punto 3

Esercizi

5. Creare un array di oggetti **String**a e assegnare una stringa a ogni elemento
   1. Stampare l'array con un **for**
6. Creare una classe con due metodi. Nel primo metodo invocare il secondo due volte: la prima senza l'utilizzo di **this** e la seconda con **this**
7. Creare una classe con due costruttori. Utilizzare **this** per invocare il secondo costruttore dal primo

Esercizi

8. Scrivere un metodo che crea e inizializza un array di **double** a due dimensioni
   1. La dimensione dell'array è determinata da due argomenti del metodo; i valori di inizializzazione sono in un range stabilito da altri due argomenti del metodo
   2. Creare un secondo metodo che stampa il contenuto dell'array
   3. Nel main testare i metodi scritti, creando e stampando array di diverse dimensioni

Java

inizializzazioni e pulizia

fine