Sapienza Università di Roma, Facoltà di Ingegneria

Corso di

Fondamenti di Informatica I

Canale 1 (A-K)

Anno Accademico 2009-2010

Corso di Laurea in Ingegneria Informatica

Docente: Camil Demetrescu

Esercitatore: Andrea Ribichini

(Dispensa basata su materiale didattico del corso di Laboratorio di Programmazione tenuto negli anni scorsi)

Classi `Class` e `Object`, stampa e test di uguaglianza

Indice

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`:

- `public String toString()`
- `public final Class getClass()`
- `public boolean equals(Object)`
- `protected Object clone()` (verrà discussa in una dispensa a parte)

Stampa di oggetti e metodo `toString()`

Il metodo `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 object_class/Esempio1.java
class B {
  private int i;
  B(int x) { this.i = x; }
  public String toString() { return "i: " + this.i; }
}

public class Esempio1 {
  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
   ad esempio B@601bb1
*/
```

Stampa in classi derivate

Nel fare overriding di `toString()` per una classe derivata è possibile riusare il metodo `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 un metodo 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.

Il metodo 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 oggetto anche della classe* `B`.

```
class D extends B ...

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

Esercizio: cosa fa questo programma?

```
// File object_class/Esercizio1.java

class B {}
class D extends B {}

public class Esercizio1 {
  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));
  }
}
```

l'operatore Java instanceof

Java è dotato di un operatore predefinito `instanceof` per verificare l'appartenenza ad una classe o la conformità ad una interfaccia di un oggetto.

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)...
```

Il metodo `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, ad es.:

```
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 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: il metodo `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 redefinire 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` `o` perchè stiamo facendo **overriding** della funzione `equals()` *ereditata* dalla classe `Object`.

- `if (o != null ...`

  dobbiamo essere sicuri che il riferimento passato alla funzione `o` si riferisca non sia `null`, altrimenti gli oggetto 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 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 specificie 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`.

Uguaglianza di oggetti `Class`

Per confrontare due oggetti `Class` `a` e `b`, dobbiamo scrivere necessariamente:

```
a.getClass().equals(b.getClass())
```

oppure possiamo anche scrivere:

```
a.getClass()==b.getClass()?
```

Sebbene in generale l'uguaglianza profonda tra oggetti debba essere verificata mediante il metodo `equals`, nel caso speciale di oggetti della classe `Class`, la differenza fra due riferimenti implica anche la differenza profonda poiché non vi saranno mai due oggetti `Class` diversi come indirizzo, ma uguali internamente. Infatti, ogni oggetto `Class` viene creato internamente dalla *Java Virtual Machine* quando viene caricata la corrispondente classe in memoria, e non è possibile creare nuove istanze.

Pertanto, l'espressione `a.getClass()==b.getClass()` è equivalente a `a.getClass().equals(b.getClass())`.

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 object_class/Esercizio2.java

class B {
  private int x, y;
  public B(int a, int b) {
    this.x = a; this.y = b;
  }
  public boolean equals(B b) { // OVERLOADING, NON OVERRIDING
    if (b != null)
      return (b.x == this.x) && (b.y == this.y);
    else return false;
  }
}

public class Esercizio2 {
  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!");
```

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: 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;
  }
}
```