Linguaggi di Programmazione I – Lezione 14

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

29 maggio 2008

Complementi

`String` **e** `StringBuffer`

Garbage collection

Questionario

LP1 – Lezione 14 2 / 18

`String` e `StringBuffer` 3 / 18

`String` (1)

- Le stringhe sono oggetti che possono essere creati come segue:

```
String s1 = new String();
s = "abcdef";
```

oppure così:

```
String s = new String("abcdef");
```

oppure ancora così:

```
String s = "abcdef";
```

- Vedremo tra poco quali sono le (sottili) differenze tra questi modi.
- La cosa importante da capire è che le stringhe sono oggetti **immutabili**.

LP1 – Lezione 14 4 / 18

`String` (2)

- Immutabilità significa che, una volta assegnato all'oggetto un valore, esso è fissato per sempre (capiremo tra poco anche perché deve essere così).
- Attenzione: immutabili sono gli oggetti `String`, non i loro riferimenti. Questi ultimi possono cambiare valore.

LP1 – Lezione 14 5 / 18

Esempi

```
String s = "Marcello";
String s2 = s;
s = s.concat(" Sette");
System.out.println(s);   // cosa stampa?
System.out.println(s2);  // e qui?
```

```
String x = "Marcello";
x.concat(" Sette");
System.out.println(x);   // cosa stampa?
```

```
String s1 = "A";
String s2 = s1 + "B";
s1.concat("C");
s2.concat(s1);
s1 += "D";                    // Quanti oggetti in gioco?
System.out.println(s1 + s2); // Cosa stampa?
```

```
String s1 = "abc";
String s2 = s1 + "";
String s3 = "abc";
System.out.println(s1 == s2);  // false!
System.out.println(s1 == s3);  // true!
```


`String` (3)

- Per motivi di efficienza, poichè nelle applicazioni i litterali `String` occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la *String constant pool*.
- Quando il compilatore incontra un litterale `String`, esso controlla che non sia già presente nel pool. Se è presente, allora il riferimento al nuovo litterale è diretto alla stringa esistente (la stringa esistente ha semplicemente un ulteriore riferimento), altrimenti lo aggiunge al pool.
- Ora si capisce perché gli oggetti `String` devono essere immutabili. Se ci sono molti riferimenti, in punti diversi del codice, allo stesso litterale, sarebbe deleterio permettere la modifica del litterale usando uno solo di tali riferimenti.
- Qual è quindi la (sottile) differenza tra i due enunciati?

```
String s = "abcdef";
```

```
String s = new String("abcdef");
```

`StringBuffer` (1)

- Se proprio si deve fare un uso intensivo di manipolazione di stringhe, allora è opportuno usare gli oggetti `StringBuffer`: essi sono un po' come gli oggetti `String`, ma non sono immutabili.
- Per esempio:

```
StringBuffer s = new StringBuffer("Marcello");
s.append(" Sette");
System.out.println(s); // cosa stampa?
```

- Attenzione:

```
StringBuffer s = "abc";
  // Illegale: String e StringBuffer
  // non sono auto-convertibili!

StringBuffer s = (StringBuffer) "abc";
  // Illegale: neanche con cast!
```


Note importanti

- ATTENZIONE: mentre la classe `String` sovrappone il metodo `equals` in modo da controllare l'uguaglianza del contenuto dei due oggetti (quello corrente e quello ricevuto come parametro), la classe `StringBuffer` non lo sovrappone ed usa quello ereditato da `Object` che funziona essenzialmente come l'operatore == (cioè compara i riferimenti).
- Le classi `String` e `StringBuffer` sono `final`. Esse non possono essere specializzate in modo da sovrapporre i loro metodi: l'invocazione virtuale di metodi sarebbe un grave problema di sicurezza.

Garbage collection 10 / 18

Ancora?

```
public class TestGC {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Ciao");
    System.out.println(sb);
    // l'oggetto riferito da sb non e' ancora
    // eleggibile per GC
    sb = null;
    // ora e' eleggibile.
  }
}
```

Ma questo non è l'unico modo di rendere eleggibile un oggetto!

Ancora??

```
public class TestGC {
  public static void main(String[] args) {
    StringBuffer s1 = new StringBuffer("Ciao");
    StringBuffer s2 = new StringBuffer("Addio");
    System.out.println(s1);
    // l'oggetto riferito da s1 non e' ancora
    // eleggibile per GC
    s1 = s2;
    // ora e' eleggibile.
  }
}
```

Ma questo non è l'ultimo modo di rendere eleggibile un oggetto!

Ancora???

```
 1. import java.util.Date;
 2. public class TestGC {
 3.   public static void main(String[] args) {
 4.     Date d = getDate();
 5.     System.out.println(d);
 6.   }
 7.
 8.   public static Date getDate() {
 9.     Date d2 = new Date();
10.     String now = d2.toString();
11.     System.out.println(now);
12.     return d2;
13.   }
14. }
```

Tracciare il codice con riferimento alla eleggibilità per GC degli oggetti.
Ma questo non è l'ultimo modo di rendere eleggibile un oggetto!

Mo' basta!

```
public class Isola {
  Isola i;

  public static void main(String[] args) {
    Isola i1 = new Isola();
    Isola i2 = new Isola();
    Isola i3 = new Isola();

    i1.i = i2;
    i2.i = i3;
    i3.i = i1;

    i1 = null;
    i2 = null;
    i3 = null;

    // qui quali oggetti sono eleggibili?
  }
}
```

D 1

Data la stringa costruita mediante `s = new String("xyzzy")`, quali delle seguenti invocazioni di metodi modificherà la stringa?

A. `s.append("aaa");`
B. `s.trim();`
C. `s.substring(3);`
D. `s.replace('z', 'a');`
E. `s.concat(s);`
F. Nessuna delle precedenti.

D 2

Qual è l'output del seguente brano di codice?

```
1. String s1 = "abc" + "def";
2. String s2 = new String(s1);
3. if (s1 == s2)
4.   System.out.println("A");
5. if (s1.equals(s2))
6. System.out.println("B");
```

A. `AB`
B. `A`
C. `B`
D. Nessun output.

D 3

Quanti oggetti sono prodotti nel seguente frammento di codice?

```
1. StringBuffer sbuf = new StringBuffer("abcde");
2. sbuf.insert(3, "xyz");
```

A. 1
B. 2
C. 3
D. 4
E. 5