Lezione 14

giovedì 4 novembre 2021 14:31

Classi wrapper

classi che corrispondono ai tipi primitivi

boolean	→	Boolean
byte	→	Byte
short	→	Short
int	→	Integer
long	→	:
float		;
double		
char		

Utili per rappresentare valori di tipo intero, booleano, etc
non come tipi primitivi ma come istanze di classi

Esempio:

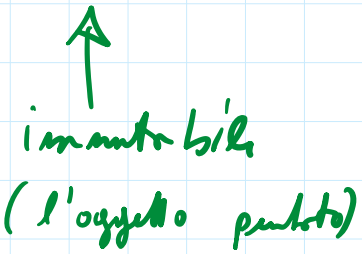
```
mappa.insert(chiave, valore)
```

riferimenti

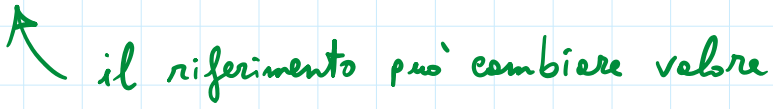
Se nella mia applicazione la chiave è un valore intero
posso passare un Integer ma non un int

Altra funzione delle classi wrapper:
raccolgono metodi di utilità relativi al tipo primitivo corrispondente

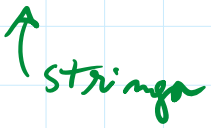
```
Integer i1 = new Integer(10);
```




```
i1 = new Integer(20);
```




```
Integer i2 = new Integer("30");
```



Gli oggetti possono essere creati anche attraverso il metodo statico valueOf()

```
Integer i4 = Integer.valueOf(50);
```

```
Integer i4 = Integer.valueOf(50);
Integer i5 = Integer.valueOf("60");
```

metodo statico della classe Integer

può restituire un riferimento a un oggetto con lo stesso valore creato in precedenza (caching)

```
Integer i6 = new Integer(70);
int x = i6.intValue();
```

restituisce il valore dell'Integer come un int

x vale 70

costanti:

```
Integer.MAX_VALUE
Integer.MIN_VALUE
```

estremi dell'intervallo di rappresentabilità degli int

Integer implementa l'interfaccia Comparable

Autoboxing / unboxing

Conversione automatica da tipo primitivo a classe wrapper e viceversa

```
Integer i7 = 80;
```

è come scrivere

```
Integer i7 = Integer.valueOf(80);
```

Autoboxing

- assegnamento da tipo primitivo a tipo wrapper
- passaggio di parametri: l'argomento formale è di tipo wrapper e l'argomento attuale è un tipo primitivo

Esempio:

```
void met(Integer x){
        ≡
}
```

```
int z = 10;
met(z);
```

Unboxing

- assegnamento
- chiamata di un metodo

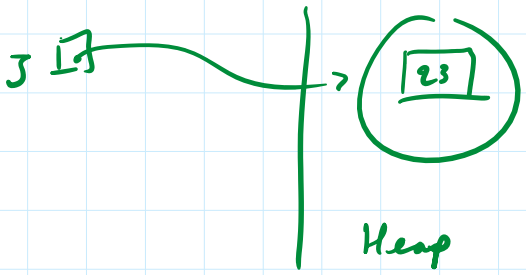
boxing

```
Integer j = 23;      // Integer j = Integer.valueOf(23);
int w = j;
```

unboxing

è come sopra

```
int w = j.intValue();
```



Un esempio particolare…

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    System.out.println("Faccio partire un altro thread");
    Altro t1 = new Altro();
    t1.start();
    Scanner sc  = new Scanner(System.in);
    String l = sc.next();
    System.out.println("Metto fine a true");
    t1.fine = true;
    System.out.println("Leggo ancora da tastiera");
    l = sc.next();
```

```
  }
}


public class Altro extends Thread {
  boolean fine = false;
  public void run(){
    while(!fine) {
      for(int i=0; i<1000000; i++);
    }
    System.out.print("ESCO");
  }
}
```

PA1920_memorymodel

Java Memory Model

Ogni valore condiviso tra thread deve sempre essere acceduto in mutua esclusione (synchronized) per evitare interferenze.

L'acquisizione di un lock ha un costo e in alcuni casi potrebbe non essere necessario.

Il linguaggio garantisce che le operazioni di lettura /scrittura su variabili (eccetto double e long) è atomica.

Se un thread esegue x = 5 e un altro esegue x=-100 il valore finale di x sarà 5 o -100, non un mix delle due.

Quindi se c'è un thread che modifica una variabile e altri che la leggono non ci sono problemi di interferenza.

Questo però non assicura che un thread legga il valore più recente di una variabile (valore scritto da un altro thread), in assenza di meccanismi di sincronizzazione.

Diversi fattori influiscono su quando una variabile modificata da un thread appare come tale ad altri thread.

Java Memory Model: insieme di regole che stabiliscono l'odinamento degli accessi alla memoria e quando le modifiche sono visibili in modo garantito.

Un thread esegue istruzioni. L'ordine in cui sono scritte è il program order.

I valori delle variabili lette da un thread sono determinati dal memory model.

Il memory model definisce l'insieme di valori che possono essere restituiti a un thread quando legge una variabile.

Il programmatore è abituato a pensare che l'unico valore possibile sia l'ultimo valore scritto.

In assenza di sincronizzazione può non essere così e i valori possibili risultano essere più di uno.

Supponiamo che un thread esegua il seguente frammento di codice:

```
x = 5.0;       // (*)
for(;;){
    display.show(x);
    Thread.sleep(100);
}
```

e un altro thread cambi il valore di x:

```
void f() {
    x = Math.random();
}
```

Se quando si esegue * il metodo f() non è mai stato chiamato l'unico valore possibile di x è 5.0.

Ogni volta che f() viene chiamato l'insieme dei valori possibili per x si amplia:
5.0, 0.32, 0.78, 0.91, 0.54, …

Secondo il memory model uno qualunque di tali valori può essere restituito da una read x, in assenza di sincronizzazione.

Da un punto di vista pratico: il compilatore vede che in * non ci sono modifiche a x e sostituisce 5.0 a x.

Questo è coerente con il Memory Model perché 5.0 è uno dei valori dell'insieme.

Bisogna restrigere l'insieme di valori possibili, secondo il memory model, all'ultimo valore scritto in f().

Il linguaggio prevede che certe "azioni" siano caratterizzate da una relazione "happens-before" ("avviene prima"):

- il rilascio di un lock su un oggetto "avviene prima" di ogni successiva acquisizione del lock sullo stesso oggetto

- la scrittura su una variabile **volatile** "avviene prima" di ogni successiva lettura della stessa variabile

- la chiamata a **start()** "avviene prima" delle azioni nel thread che è stato fatto partire

- le operazioni in un thread "avvengono prima" della **join()** su tale thread.

volatile non garantisce atomicità su operazioni "multiple".

In effetti le cose sono anche più complicate di così: in assenza di sincronizzazione un thread può vedere le operazioni fatte da un altro thread "al contrario"

Supponiamo di avere inizialmente

A==B==0

Thread 1	Thread 2
1) r2 = A; 2) B = 1;	3) r1 = B; 4) A = 2;

Sembra che avere r2 == 2 e r1 == 1 sia impossibile.

Intuitivamente 1) e 3) dovrebbero venire prima in una possibile esecuzione
Se viene prima 1) non dovrebbe essere possibile vedere il risultato della scrittura 4)
Se viene prima 3) non dovrebbe essere possibile vedere il risultato della scrittura 2)
Dovremmo avere infatti
   4) prima di 1)
   1) prima di 2)
   2) prima di 3)
   3) prima di 4)
Assurdo

Ai compilatori è permesso riordinare le istruzioni di un thread a patto che il riordinamento non influsca sul comportamento del singolo thread

Se 1) scambiata di ordine con 2): possibile per il singolo thread perché non ha impatto su di esso

A questo punto però possiamo avere r2==2 e r1==1

Il malfunzionamento è dovuto a incorretta sincronizzazione

Riordinamenti introdotti da
- compilatore JIT
- processore
- l'architettura della memoria può far sembrare che ci siano dei riordinamenti

program order
+
code order
+
execution order
+
ordine con cui vengono viste le modifiche