

6 - Blocchi e cicli

Programmazione e analisi di dati
Modulo A: Programmazione in Java

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
<http://www.di.unipi.it/~milazzo>
milazzo@di.unipi.it

Corso di Laurea Magistrale in Informatica Umanistica
A.A. 2013/2014

Sommario

1 Blocchi e visibilità delle variabili

2 Comandi iterativi (cicli)

- Il comando `while`
- Il comando `for`

Blocchi (1)

Abbiamo visto che se in un ramo di un `if` devono essere eseguiti più comandi ci vuole un **blocco**

- Un **blocco** è una sequenza di comandi racchiusa tra **parentesi graffe**

```
if (saldo>=0) {  
    System.out.println("Saldo positivo");  
  
    double interesseAttivo = saldo*tassoAttivo;  
    saldo = saldo+interesseAttivo;  
}
```

Blocchi (2)

Il **corpo del metodo** `main` è un altro esempio di blocco

```
static public void main(String[] args) {  
    .....  
}
```

Un blocco può essere inserito tra gli altri comandi del programma

```
System.out.println("Prima del blocco");  
{  
    System.out.println("Dentro... ");  
    System.out.println("...al blocco");  
}  
System.out.println("Fuori del blocco");
```

Inoltre, i blocchi possono essere **annidati** (uno dentro l'altro)

```
System.out.println("Fuori dai blocchi");  
{  
    System.out.println("Nel primo blocco");  
    {  
        System.out.println("Nel secondo...");  
        System.out.println("...blocco");  
    }  
    System.out.println("Di nuovo nel primo blocco");  
}  
System.out.println("Fuori dai blocchi");
```

Variabili locali

Una proprietà importante dei blocchi è la seguente:

le variabili dichiarate in un blocco sono **locali al blocco** e, quindi, scompaiono dopo l'esecuzione del blocco

```
if (saldo>=0) {  
    System.out.println("Saldo positivo");  
  
    double interesse = saldo*tasso;  
    saldo = saldo+interesse;  
}  
  
System.out.println(interesse); //ERRORE!
```

Visibilità delle variabili (1)

Si può definire una nozione di **visibilità** (o **scope**) di una variabile

La **visibilità** (o **scope**) di una variabile è la **porzione di programma** in cui tale variabile può essere utilizzata

La visibilità di una **variabile locale** (cioè definita in un blocco) è la porzione di programma che va **dalla dichiarazione** della variabile stessa **alla fine del blocco** che la contiene

NOTA: Anche il corpo del metodo `main` è un blocco!

- Le variabili usate fino ad ora erano **locali** al metodo `main`

Visibilità delle variabili (2)

Limitare la visibilità di una variabile al blocco che la contiene consente di

- riutilizzare nomi di variabili in parti diverse del programma
- anche con tipi diversi

```
boolean leggiIntero = ....;
if (leggiIntero) {
    int val = input.nextInt();

    System.out.print("Il numero letto e'");
    System.out.println(val);
} else {
    double val = input.nextDouble();

    System.out.print("Il numero letto e'");
    System.out.println(val);
}
```

Tutto ciò sarà particolarmente utile nei cicli e nei programmi costituiti da più classi e metodi... (vedremo)

Visibilità delle variabili (3)

ATTENZIONE: Le variabili dichiarate in un blocco non possono avere il nome di variabili esterne al blocco stesso

- Il compilatore segnalerebbe un errore

```
int a=0;
{
  int a=0; // ERRORE: variabile gia' dichiarata
  int b=10;
  System.out.println(a+b);
}
```


Sommario

1 Blocchi e visibilita' delle variabili

2 Comandi iterativi (cicli)

- Il comando `while`
- Il comando `for`

Ripetere, ripetere, ripetere...

Tempo fa abbiamo visto il seguente programma *Somma*, che calcola la somma di due numeri

E se volessimo sommare un **numero arbitrario** di numeri (a scelta dell'utente)?

- Serve un comando per poter **ITERARE** (ripetere) più volte delle operazioni
- Vogliamo quindi realizzare dei **CICLI**

Il comando while (1)

Il comando while consente di ripetere un comando (o un blocco) **fintanto che** una condizione specificata è vera

```
import java.util.Scanner;

public class SommaNumeri {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.println("Quanti numeri vuoi sommare?");
        int numeri=input.nextInt();

        int cont=0; //variabile da usare come contatore
        int somma=0; //variabile da usare come accumulatore

        while (cont<numeri) { //fintanto che cont e' minore di numeri

            System.out.println("Inserisci il prossimo numero");
            int n=input.nextInt();

            somma=somma+n; //aggiorna l'accumulatore
            cont=cont+1; //incrementa il contatore
        }

        System.out.println(somma);
    }
}
```

Il comando while (2)

Il comando `while` ha la seguente **forma**:

```
while (...condizione...) ...comando....
```

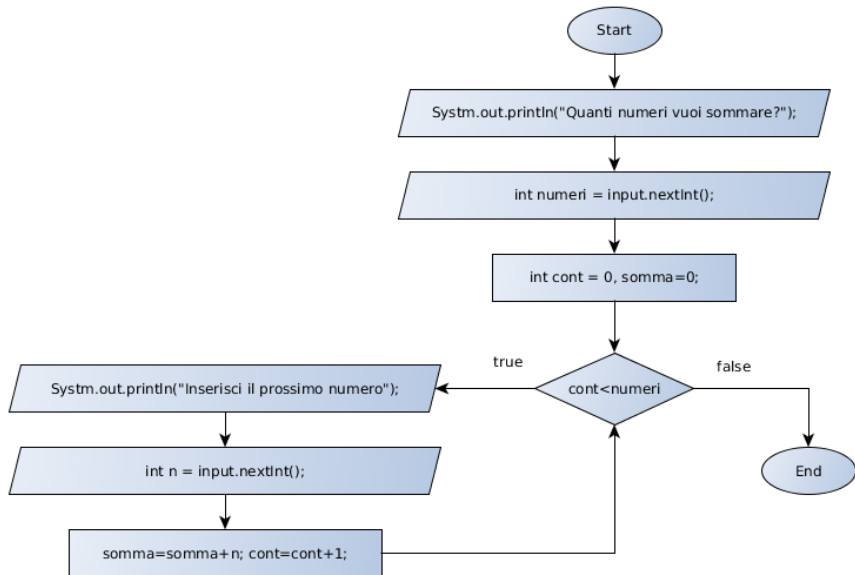
dove:

- La condizione è detta anche **guardia** del `while`
- La guardia può essere una qualunque **espressione booleana**
- Il comando (o blocco) è detto **corpo** del `while`

Semantica del comando:

1. La guardia viene valutata
2. Se la guardia è vera si esegue il corpo e si ricomincia dal punto 1
3. Se la guardia è falsa si salta il corpo e si procede con l'istruzione successiva al `while`

Il comando while (3)



Comandi di assegnamento ausiliari (1)

Nel programma `SommaNumeri` abbiamo visto i seguenti comandi:

```
somma=somma+n; //aggiorna l'accumulatore  
cont=cont+1; //incrementa il contatore
```

Quando si usano i cicli, questo tipo di assegnamenti diventano frequenti.

Per questo in Java esistono dei comandi di assegnamento ausiliari (abbreviazioni sintattiche) che semplificano un po' la scrittura:

```
somma+=n; //corrisponde a somma=somma+n  
cont++; //corrisponde a cont=cont+1
```

In maniera analoga abbiamo

```
x-=y //corrisponde a x=x-y  
x*=y //corrisponde a x=x*y  
x/=y //corrisponde a x=x/y  
i-- //corrisponde a i=i-1
```

Nota: esistono anche le forme prefisse `++i` e `--i` che hanno senso quando si usano le operazioni di assegnamento all'interno di espressioni. Questa però non è in generale una buona pratica.

Comandi di assegnamento ausiliari (2)

Modifichiamo il programma usando i comandi di assegnamento ausiliari

```
import java.util.Scanner;

public class SommaNumeri2 {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.println("Quanti numeri vuoi sommare?");
        int numeri=input.nextInt();

        int cont=0; //variabile da usare come contatore
        int somma=0; //variabile da usare come accumulatore

        while (cont<numeri) { //fintanto che cont e' minore di numeri

            System.out.println("Inserisci il prossimo numero");
            int n=input.nextInt();

            somma+=n; //aggiorna l'accumulatore (con +=)
            cont++; //incrementa il contatore (con ++)
        }

        System.out.println(somma);
    }
}
```

Esempio d'uso (1)

Negozio di Caramelle:

- abbiamo 100 caramelle da vendere al prezzo di 50 cent l'una
- i clienti arrivano uno dopo l'altro e possono comprare quante caramelle vogliono (fino ad esaurimento)

```
import java.util.Scanner;

public class CandyShop {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        int caramelle=100; //caramelle in magazzino
        final double PREZZO=0.5; //prezzo singola caramella in euro

        while (caramelle>0) { //fintanto che non ho esaurito le caramelle

            System.out.print("Sono disponibili ");
            System.out.print(caramelle);
            System.out.println(" caramelle");

            System.out.println("Quante ne vuoi comprare?");
            int num=input.nextInt();

            ....continua.....
        }
    }
}
```


Esempio d'uso (2)

```
....continua.....
```

```
if (num<0) //controlla l'input
    System.out.println("Numero errato");
else {

    if (num>caramelle) {
        num=caramelle;
        System.out.println("Hai chiesto troppe caramelle");
        System.out.print("Te ne daro' soltanto ");
        System.out.println(num);
    }

    caramelle -= num; //preleva le caramelle dal magazzino
    System.out.print("Costo: ");
    System.out.print(num*PREZZO);
    System.out.println(" euro");

}

}

// a questo punto il ciclo e' terminato
// cioe' le caramelle sono esaurite (caramelle<=0)
System.out.println("CARMELLE TERMINATE!");
}
```

I cicli do-while (1)

Quando siamo sicuri che il corpo di un ciclo deve essere eseguito almeno una volta possiamo usare un do-while

- Modifichiamo l'esempio SommaNumeri in modo che continui a sommare numeri finchè l'utente non inserisce un numero negativo

```
import java.util.Scanner;

public class SommaNumeri3 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        int somma=0; //variabile da usare come accumulatore
        int n; //variabile che memorizza i numeri inseriti
                /*NOTA: n deve essere dichiarata qui per
                essere visibile nella guardia del while*/

        do {
            System.out.println("Inserisci il prossimo numero");
            n=input.nextInt();

            if (n>=0) somma+=n; //aggiorna l'accumulatore se n>=0
        } while (n>=0); //cicla fintanto che n non e' negativo

        System.out.println(somma);
    }
}
```

I cicli do-while (2)

Il comando do-while ha la seguente **forma**:

```
do ....comando.... while (...condizione...)
```

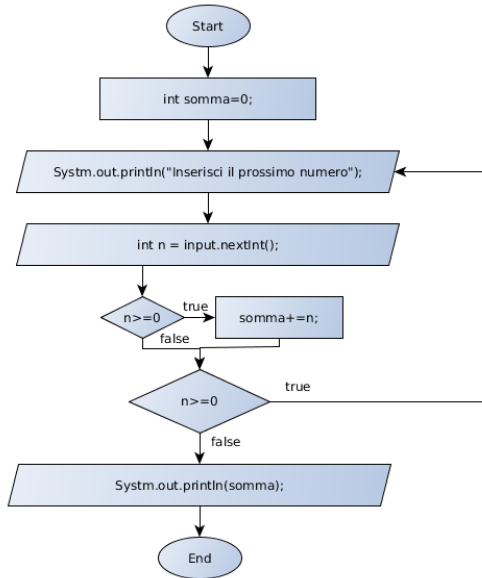
dove:

- La condizione è detta anche **guardia** del do-while
- La guardia può essere una qualunque **espressione booleana**
- Il comando (o blocco) è detto **corpo** del do-while

Semantica del comando:

1. Il corpo viene eseguito
2. La guardia viene valutata
2. Se la guardia è vera si ricomincia dal punto 1
3. Se la guardia è falsa si procede con l'istruzione successiva al while

I cicli do-while (3)



Sommario

1 Blocchi e visibilita' delle variabili

2 Comandi iterativi (cicli)

- Il comando `while`
- Il comando `for`

Quante iterazioni fa un ciclo?

Quando scriviamo un ciclo `while`, in generale potremmo non sapere **quante iterazioni** esso farà

- **Esempio:** in `CandyShop` il numero di iterazioni dipende da quante caramelle “compra” ad ogni passo l’utente
- **Iterazione indeterminata:** “corro finchè ho fiato...”

In alcuni casi, invece, il numero di iterazioni è **noto a priori**

- **Esempio:** in `SommaNumeri` si chiede all’utente quanti numeri voglia sommare, e il numero inserito dall’utente diventa esattamente il numero di iterazioni del ciclo.
- **Iterazione determinata:** “corro per 10 giri di campo...”

Quando il numero di iterazioni è noto a priori, in alternativa al `while` possiamo usare il **comando `for`**

I cicli for (1)

Il comando for ha la seguente **forma**:

```
for ( ..cmdIniz.. ; ..condiz.. ; ..cmdAgg.. ) ...corpo...
```

dove:

- cmdIniz è un comando eseguito all'**inizio del ciclo**
- **condiz** è la **guardia** del ciclo
- La guardia può essere una qualunque **espressione booleana**
- cmdAgg un comando eseguito **ad ogni iterazione** (Agg sta per aggiornamento)
- Il **corpo** può essere un singolo comando o un blocco

I cicli for (2)

Un esempietto semplice:

```
for (int i=0; i<10; i++)  
    System.out.println(i);
```

Questo programma stampa i valori da 0 a 9 uno dopo l'altro

In altre parole:

- Stampa tutti i valori di *i*, per *i* che va da 0 a 10 (escluso) aumentando ogni volta *i* di 1

I cicli for (3)

```
for ( ..cmdIniz.. ; ..condiz.. ; ..cmdAgg.. ) ...corpo...
```

Semantica del comando:

1. Viene eseguito cmdIniz
2. La guardia (condiz) viene valutata
3. Se la guardia è vera:
 - ▶ si esegue il corpo
 - ▶ si esegue cmdAgg
 - ▶ si ricomincia da 2 (ATTENZIONE: non da 1)
4. Se la guardia è falsa si salta il corpo e si procede con l'istruzione successiva al for

I cicli for (4)

Modifichiamo allora SommaNumeri usando un for

```
import java.util.Scanner;

public class SommaNumeri4 {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.println("Quanti numeri vuoi sommare?");
        int numeri=input.nextInt();

        int somma=0; //variabile da usare come accumulatore

        //per i che va da 0 a numeri (escluso)
        for (int i=0; i<numeri; i++) {

            System.out.println("Inserisci il prossimo numero");
            int n=input.nextInt();

            somma+=n; //aggiorna l'accumulatore
        }

        System.out.println(somma);
    }
}
```

I cicli for (6)

Un po' di osservazioni:

- Nel ciclo

```
for (int i=0; i<10; i++){  
    System.out.println(i);  
}
```

la variabile *i* è **locale**.

- ▶ la sua visibilità è limitata al for
 - ▶ è buona norma **non modificare** *i* nel corpo del for
 - ▶ posso riutilizzare *i* in un for successivo
- I nomi *i, j, k...* sono usati comunemente per le variabili-contatore dei cicli for
 - Si possono inserire più comandi di inizializzazione e aggiornamento usando la virgola

```
for (int i=0, j=0; i<10; i++,j+=i) {  
    System.out.println(i);  
    System.out.println(j);  
}
```

I cicli for (7)

Un ciclo for **può essere sempre tradotto** in un ciclo while equivalente

Ad esempio, l'esempietto visto prima

```
for (int i=0; i<10; i++){
    System.out.println(i);
}
```

è equivalente al seguente **blocco** (nota: i nel for e' una variabile locale)

```
{
    int i=0;
    while (i<10) {
        System.out.println(i);
        i++;
    }
}
```

ossia:

```
{
    ...cmdIniz...
    while (...condiz...) {
        ...corpo...
        ...cmdAgg...
    }
}
```

Annidamento cicli (1)

Spesso nei programmi si usano **cicli annidati**

- un ciclo all'interno del corpo di un'altro ciclo
- il ciclo interno viene ri-eseguito ad ogni iterazione del ciclo esterno
- ovviamente si possono annidare for dentro for, while dentro for, while dentro while,

```
public class TavolaPitagorica {
    public static void main(String[] args) {

        for (int i=1; i<=10; i++) {
            for (int j=1; j<=10; j++) {
                System.out.print(i*j);
                System.out.print(" ");
            }
            System.out.println(); // a capo
        }
    }
}
```

Annidamento cicli (2)

Output di TavolaPitagorica:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```