Istruzioni di Controllo

Emilio Di Giacomo e Walter Didimo

Limite delle istruzioni viste

- L'insieme delle istruzioni che abbiamo visto fino ad ora consiste per lo più di:
  - dichiarazioni e assegnazioni di variabili
  - invocazione di metodi
  - restituzione di valori (return)

- Con queste istruzioni non sono molte le cose che possiamo fare

Istruzioni di controllo

- In questa lezione introduciamo le istruzioni di controllo; esse consentono di:
  - confrontare dei dati e stabilire comportamenti diversi in funzione dell'esito del confronto (istruzioni condizionali)
  - ripetere una stessa sequenza di istruzioni più volte, fintanto che certe condizioni permangono (istruzioni iterative)

Istruzioni condizionali

- Un'istruzione condizionale serve a verificare se una determinata condizione è vera o falsa
  - la condizione deve essere un predicato, ossia un'espressione di tipo boolean
- In funzione dell'esito, l'istruzione condizionale stabilisce le prossime istruzioni da eseguire

Istruzioni condizionali

- In Java esistono tre istruzioni condizionali:
  - if
  - if-else
  - switch-case
- Esiste inoltre un operatore simile all'if-else
  - operatore condizionale (o operatore ternario)

L’istruzione if: sintassi

- La sintassi dell’istruzione *if* è la seguente

  if (⟨*condizione*⟩)

  ⟨*istruzione*⟩
- ⟨*condizione*⟩ è un’espressione di tipo *boolean* detta condizione dell’*if*
- ⟨*istruzione*⟩ è una qualunque istruzione Java detta corpo dell’*if*

L’istruzione if: semantica

if (⟨*condizione*⟩)

⟨*istruzione*⟩

- la ⟨*condizione*⟩ viene valutata
- se il suo valore è *true* viene eseguito il corpo dell’*if*
- se il valore della condizione è *false* il corpo non viene eseguito

Istruzione if: esempio

- Vogliamo scrivere un metodo statico che presi due numeri interi *a* e *b* come parametri, restituisce il minore dei due
- Se i numeri sono uguali, restituirà indifferentemente il valore di uno dei due

Istruzione if: esempio

```
public static int minoreTra(int a, int b){
    int min = a;
    if (b<a)
        min = b;
    return min;
}
```

- La variabile *min* viene inizializzata con il valore di *a*
- Se *b* è più piccolo di *a*, il valore di *min* viene cambiato in *b*
- Altrimenti non accade nulla e *min* resta pari ad *a*

Istruzioni multiple nel corpo dell'if

- La sintassi dell'*if* impone che il corpo sia costituito da una sola istruzione
- È possibile eseguire più istruzioni nel corpo dell'*if*?
- Sì, bisogna racchiuderle tra parentesi graffe

Esempio

- Supponiamo di voler scrivere un metodo statico che prenda come parametri due stringhe *str1* e *str2* e che, se *str1* è un prefisso di *str2*, restituisca la sottostringa ottenuta “rimuovendo” *str1* da *str2*
- Nel caso in cui *str1* non sia un prefisso di *str2*, allora il metodo restituirà *str2* inalterata.

Esempio

- Esempio 1

  str1=“gatto”

  str2=“gattopardo”

  risultato: *“pardo”*

- Esempio 2

  str1=“sole”

  str2=“mattino”

  risultato: *“mattino”*

Esempio

```
public static String tagliaDa(String str1, String
str2){
    String str = str2;
    if (str2.indexOf(str1)==0){
          int pos = str1.length();
          str = str2.substring(pos);
    }
    return str;
}
```

- Il corpo dell'*if* è costituito da due istruzioni

Blocchi di istruzioni

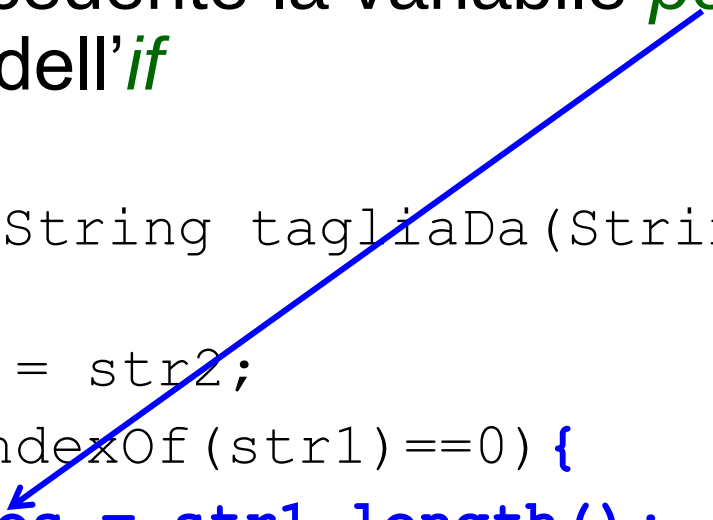
- Le due istruzioni racchiuse tra graffe dell'esempio precedente costituiscono un blocco di istruzioni
- Un blocco di istruzioni è una istruzione:
  - può essere utilizzato in qualunque punto ci si aspetta una istruzione
  - ad esempio nel corpo dell'istruzione *if*
- È possibile annidare blocchi cioè definire un blocco in un altro blocco

Blocchi e variabili

- Le variabili definite in un blocco sono visibili solo in quel blocco e in tutti quelli in esso annidati
- Non è possibile definire una variabile in un blocco se ne esiste una con lo stesso nome visibile in quel blocco

Blocchi e variabili

- Nel codice precedente la variabile *pos* è visibile solo nel corpo dell'*if*



```
public static String tagliaDa(String str1,
String str2){
    String str = str2;
    if (str2.indexOf(str1)==0){
        int pos = str1.length();
        str = str2.substring(pos);
    }
    return str;
}
```

Blocchi annidati e variabili

```
public static void met(){
   int a = 10;
       {// inizio blocco 1
         int b = 5;
         a += b;
             {// inizio blocco 2
               int c = 2;
               b += c;
               a += c;
             }// fine blocco 2
         System.out.println(b);
       }// fine blocco 1
   System.out.println(a);
}
```

Blocchi annidati e variabili

```
public static void met(){
   int a = 10;
       {// inizio blocco 1
         int b = 5;
         a += b;
             {// inizio blocco 2
               int c = 2;
               b += c;
               a += c;
             }// fine blocco 2
         System.out.println(b);
       }// fine blocco 1
   System.out.println(a);
}
```

blocco 1

Blocchi annidati e variabili

```
public static void met(){
   int a = 10;
       {// inizio blocco 1
         int b = 5;
         a += b;
             {// inizio blocco 2
               int c = 2;
               b += c;
               a += c;
             }// fine blocco 2
         System.out.println(b);
       }// fine blocco 1
   System.out.println(a);
}
```

blocco 2

Blocchi annidati e variabili

```
public static void met(){
   int a = 10;
       {// inizio blocco 1
         int b = 5;
         a += b;
             {// inizio blocco 2
               int c = 2;
               b += c;
               a += c;
             }// fine blocco 2
         System.out.println(b);
       }// fine blocco 1
   System.out.println(a);
}
```

visibilità di *a*

Blocchi annidati e variabili

```
public static void met(){
   int a = 10;
       {// inizio blocco 1
         int b = 5;
         a += b;
             {// inizio blocco 2
               int c = 2;
               b += c;
               a += c;
             }// fine blocco 2
         System.out.println(b);
       }// fine blocco 1
   System.out.println(a);
}
```

visibilità di *b*

Blocchi annidati e variabili

```
public static void met(){
   int a = 10;
       {// inizio blocco 1
         int b = 5;
         a += b;
             {// inizio blocco 2
               int c = 2;
               b += c;
               a += c;
             }// fine blocco 2
         System.out.println(b);
       }// fine blocco 1
   System.out.println(a);
}
```

visibilità di *c*

If annidati

- Poiché l'*if* è una istruzione Java al pari di tutte le altre, è possibile che il corpo di una istruzione *if* sia un'altra istruzione *if*
- Si parla in questo caso di *if* annidati
- Esempio: modifichiamo il metodo *tagliaDa(...)* in modo da rimuovere *str1* da *str2* solo nel caso in cui *str1* non coincida con *str2*

If annidati: esempio

```
public static String tagliaDa(String str1, String
   str2){
   String str = str2;
   if (!str2.equals(str1))
       if (str2.indexOf(str1)==0){
             int pos = str1.length();
             str = str2.substring(pos);
       }
   return str;
}
```

- La seconda istruzione *if* verrà eseguita solo se la condizione della prima è vera

If annidati: esempio

```
public static String tagliaDa(String str1, String
   str2){
   String str = str2;
   if (!str2.equals(str1))
      if (str2.indexOf(str1)==0){
            int pos = str1.length();
            str = str2.substring(pos);
      }
   return str;
}
```

- Nota: il secondo if non è racchiuso tra graffe
- Questa è infatti un'unica istruzione

L'istruzione if-else: sintassi

- La sintassi dell'istruzione *if-else* è la seguente

  if (⟨*condizione*⟩)

  ⟨*istruzione 1*⟩

  else

  ⟨*istruzione 2*⟩
- ⟨*condizione*⟩ è un'espressione di tipo *boolean* detta condizione dell'*if-else*
- ⟨*istruzione 1*⟩ è una qualunque istruzione Java (anche un blocco) detta corpo dell'*if*
- ⟨*istruzione 2*⟩ è una qualunque istruzione Java (anche un blocco) detta corpo dell'*else*

L’istruzione if-else: semantica

if (⟨*condizione*⟩)

⟨*istruzione 1*⟩

else

⟨*istruzione 2*⟩

- la ⟨*condizione*⟩ viene valutata
- se il suo valore è *true* viene eseguito il corpo dell’*if*
- se il valore della condizione è *false* viene eseguito il corpo dell’*else*

Istruzione if-else: esempio

- Vogliamo riscrivere il metodo *minoreTra* già visto utilizzando l'istruzione *if-else*
- Ricordiamo che il metodo *minoreTra* prende due numeri interi *a* e *b* come parametri e restituisce il minore dei due. Se i numeri sono uguali, restituisce indifferentemente il valore di uno dei due

Istruzione if-else: esempio

```
public static int minoreTra(int a, int b){
    int min;
    if (b<a)
          min = b;
    else
          min = a;
    return min;
}
```

- Se *b* è più piccolo di *a*, il valore di *b* viene assegnato a *min*
- Altrimenti a *min* viene assegnato il valore di *a*

MinoreTra: codice alternativo

```
public static int minoreTra(int a, int b){
    if (b<a)
        return b;
    else
        return a;
}
```

- Se *b* è più piccolo di *a* viene restituito il valore di *b*
- Altrimenti viene restituito il valore di *a*

Commenti

- Un metodo con *return* multipli può essere più difficile da comprendere e più soggetto ad errori
- In base ai principi della programmazione strutturata ogni metodo dovrebbe avere un solo punto di ingresso e un solo punto di uscita
- È quindi preferibile avere un'unica istruzione *return* al termine del metodo.

Cascate di if-else

- Scriviamo un metodo che ricevuto un intero *n* come parametro:
  - decide se il numero è pari o dispari
  - se è pari stampa il risultato della divisione di *n* per 2
  - se è dispari verifica se *n* è divisibile per 3
  - se lo è stampa il risultato della divisione di *n* per 3
  - se non lo è verifica se *n* è divisibile per 5
  - se lo è stampa il risultato della divisione di *n* per 5
  - se non lo è stampa un messaggio opportuno

Cascate di if-else

```
public static void divisibilePer235(int n){
  if (n%2==0){
    System.out.println("Num. pari");
    System.out.println("Quoziente "+n+"/2: "+n/2);
  }
  else
    if (n%3==0){
      System.out.println("Num. dispari divis. per 3");
      System.out.println("Quoziente "+n+"/3: "+n/3);
    }
    else
      if (n%5==0){
        System.out.println("Num. dispari divis. per 5 ma non per 3");
        System.out.println("Quoziente "+n+"/5: "+n/5);
      }
      else
        System.out.println("Num. dispari non divis. per 3 o per 5");
}
```

Indentazione alternativa

```
public static void divisibilePer235(int n){
  if (n%2==0){
    System.out.println("Num. pari");
    System.out.println("Quoziente "+n+"/2: "+n/2);
  } else if (n%3==0){
    System.out.println("Num. dispari divis. per 3");
    System.out.println("Quoziente "+n+"/3: "+n/3);
  } else if (n%5==0){
    System.out.println("Num. dispari divis. per 5 ma non per 3");
    System.out.println("Quoziente "+n+"/5: "+n/5);
  } else
    System.out.println("Num. dispari non divis. per 3 o per 5");
}
```

Altre istruzioni condizionali

- Java mette a disposizione un'altra istruzione condizionale:
  - switch-case
- e un operatore simile all'if-else
  - operatore condizionale (o operatore ternario)

- Per i dettagli su tali istruzioni si rimanda al libro di testo

Istruzioni iterative

- le istruzioni iterative consentono di ripetere più volte l'esecuzione di una sequenza di istruzioni all'interno di un metodo
- un'istruzione iterativa è anche chiamata ciclo
- Java mette a disposizione tre diversi tipi di istruzioni iterative:
  - while
  - do-while
  - for

L’istruzione while

- Supponiamo di voler scrivere un metodo che, ricevuto un intero *n* come parametro, stampi tutti i numeri pari da *0* a *n*

Un’idea per la soluzione

- Generiamo tutti i numeri da *0* a *n* e per ognuno controlliamo se è pari. Partiamo da *i=0* e ripetiamo i seguenti passi:
  1. controlla se *i<n+1*: se no termina, altrimenti prosegui
  2. controlla se *i* è pari: se sì stampa *i*
  3. incrementa *i* di un’unità
  4. torna al passo 1
- Questa strategia può essere realizzata con l’istruzione *while*

L'istruzione while

- La sintassi dell'istruzione *while* è la seguente

  while (⟨*condizione*⟩)

  ⟨*istruzione*⟩

- ⟨*condizione*⟩ è un'espressione di tipo *boolean* detta condizione del *while*
- ⟨*istruzione*⟩ è una qualunque istruzione Java (anche un blocco) detta corpo del *while*

L’istruzione while: semantica

while (⟨*condizione*⟩)

⟨*istruzione*⟩

- la ⟨*condizione*⟩ viene valutata; se è falsa il corpo non viene eseguito e si passa oltre
- se invece la ⟨*condizione*⟩ è vera si esegue il corpo del *while*
- al termine dell’esecuzione del corpo si torna a valutare la ⟨*condizione*⟩ e si itera il comportamento precedente

Esempio

```
public static void stampaPariDa0A(int n){
   int i = 0;
   while (i<n+1){
       if (i%2==0)
             System.out.println(i);
       i++;
   }
}
```

- La variabile *i* viene inizializzata con il valore *0* e viene incrementata di una unità alla fine di ogni iterazione del ciclo *while*
- Si continua a ripetere il ciclo finché il valore di *i* si mantiene non superiore a *n*
- Quando *i* assume il valore *n+1*, il ciclo termina.

L’istruzione while: commenti

- È necessario fare in modo che la condizione del *while* prima o poi diventi falsa
- Se così non fosse, il corpo del ciclo verrebbe ripetuto all’infinito (loop infinito)
- Le istruzioni del corpo del *while* devono modificare la condizione
  - nell’esempio precedente l’istruzione *i*++ modifica il valore di *i* e fa sì che *i* diventa ad un certo punto pari a *n+1*

L’istruzione while: commenti

- Il corpo del *while* potrebbe anche non essere eseguito mai
- Ciò accade se la condizione è falsa la prima volta che viene valutata

Esempio: codice alternativo

```
public static void stampaPariDa0A(int n){
   int i = 0;
   while (i<n+1){
        System.out.println(i);
        i+=2;
   }
}
```

- La variabile *i* viene inizializzata con il valore *0* e viene incrementata di due unità ad ogni iterazione del ciclo *while*
- In questo modo *i* assume soltanto i valori pari
- Si continua a ripetere il ciclo finché il valore di *i* si mantiene non superiore a *n*
- Quando *i* assume un valore superiore a *n*, il ciclo termina.

L’istruzione do-while: sintassi

- L’istruzione *do-while* è simile all’istruzione *while*
- La sintassi dell’istruzione *do-while* è la seguente

  do

  ⟨*istruzione*⟩

  while (⟨*condizione*⟩)
- ⟨*condizione*⟩ è un’espressione di tipo *boolean* detta condizione del *do-while*
- ⟨*istruzione*⟩ è una qualunque istruzione Java (anche un blocco) detta corpo del *do-while*

L’istruzione do-while: semantica

do

⟨*istruzione*⟩

while (⟨*condizione*⟩)

- viene eseguito il corpo una prima volta
- viene poi valutata la ⟨*condizione*⟩
- se è falsa l’esecuzione del *do-while* termina
- se invece la ⟨*condizione*⟩ è vera si torna ad eseguire il corpo e si itera il comportamento precedente

L’istruzione do-while: commenti

- Così come per il *while*, il corpo del *do-while* viene eseguito fintanto che la condizione rimane vera
- La differenza è che, nel caso del *do-while*, il corpo viene eseguito almeno una volta

Esempio

```
public static void stampaPariDa0A(int n){
   int i = 0;
   do{
       System.out.println(i);
       i += 2;
   }
   while (i<n+1);
}
```

- Poiché il primo valore (cioè *0*) è pari, la prima iterazione deve essere eseguita

L’istruzione for: sintassi

- L’istruzione *for* è abbastanza diversa dalle due precedenti
- La sintassi dell’istruzione *for* è la seguente

  for (⟨*inizializzazione*⟩;⟨*condizione*⟩; ⟨*aggiornamento*⟩)

  ⟨*istruzione*⟩
- ⟨*inizializzazione*⟩ è un’espressione di qualunque tipo (anche l’invocazione di un metodo *void*):
  - quasi sempre è un’istruzione di assegnazione
- ⟨*condizione*⟩ è un’espressione di tipo *boolean*

L’istruzione for: sintassi

- L’istruzione *for* è abbastanza diversa dalle due precedenti
- La sintassi dell’istruzione *for* è la seguente

  for (⟨*inizializzazione*⟩;⟨*condizione*⟩; ⟨*aggiornamento*⟩)

  ⟨*istruzione*⟩
- ⟨*aggiornamento*⟩ è un’espressione di qualunque tipo (anche l’invocazione di un metodo *void*):
  - quasi sempre è un incremento o decremento di una variabile
- ⟨*istruzione*⟩ è una qualunque istruzione Java (anche un blocco)

L’istruzione for: sintassi

- ⟨*inizializzazione*⟩, ⟨*condizione*⟩ e ⟨*aggiornamento*⟩ sono dette rispettivamente inizializzazione, condizione e aggiornamento del *for*
- ⟨*istruzione*⟩ è detta, come al solito, corpo del *for*

L’istruzione for: semantica

- Quando la JVM incontra un’istruzione *for*, esegue le seguenti azioni:
  1. esegue l’inizializzazione
  2. valuta la condizione; se essa è falsa l’esecuzione del *for* termina
  3. se la condizione è vera viene eseguito il corpo
  4. una volta eseguito il corpo viene eseguito l’aggiornamento e si torna al punto 2

L’istruzione for: commenti

- In base a quanto detto, l’inizializzazione viene eseguita una ed una sola volta in ogni caso, prima di qualunque iterazione
- dopo di ciò ogni iterazione consisterà di:
  - verifica condizione
  - esecuzione del corpo
  - aggiornamento

Esempio

```
public static void stampaPariDa0A(int n){
   int i;
   for(i = 0; i<n+1; i += 2)
       System.out.println(i);
}
```

- La variabile *i* viene inizializzata a *0* prima di qualunque iterazione
- Ad ogni iterazione viene valutato se *i* è minore o uguale a *n*
- in caso positivo *i* viene stampata e incrementata di due

Esempio: commenti

- Il codice precedente è molto compatto
  - in particolare, poiché l'incremento della variabile *i* avviene con l'istruzione di aggiornamento, il corpo è costituito dalla sola istruzione di stampa
- L'istruzione *for* precedente può essere letta come: "*Per i che va da 0 a n, stampa il valore di i e poi incrementalo di 2 unità*"
- In effetti l'istruzione *for* viene tipicamente utilizzata per far assumere ad una variabile un numero discreto di valori in un intervallo fissato

Esempio: versione alternativa

```
public static void stampaPariDa0A(int n){
   for(int i = 0; i<n+1; i += 2)
       System.out.println(i);
}
```

- In questo caso la variabile *i* viene definita all'interno del *for* (nell'inizializzazione)
- Così facendo la variabile *i* è visibile solo all'interno del ciclo *for*
  - nelle tre parti tra parentesi e nel corpo

Istruzione for: commenti

- Ciascuna delle tre parti del *for* è opzionale e può essere omessa
- Nelle parti di inizializzazione e aggiornamento è possibile scrivere un elenco di istruzioni separate da virgole
- Per i dettagli su entrambi gli aspetti si rimanda al libro di testo

Confronto tra le istruzioni iterative

- Abbiamo visto tre diversi tipi di istruzioni iterative...
- ...ed un problema che poteva essere risolto con tutte e tre
- Ci sono situazioni in cui è necessario l'uso di una specifica istruzione iterativa?

Equivalenza di istruzioni iterative

- L'istruzione *for* e l'istruzione *while* sono equivalenti

for (⟨*iniz.*⟩; ⟨*cond.*⟩; ⟨*aggior.*⟩)
    ⟨*istruzione*⟩

⟺

⟨*iniz.*⟩
while(⟨*cond.*⟩){
    ⟨*istruzione*⟩
    ⟨*aggior.*⟩
}

Equivalenza di istruzioni iterative

- Il *do-while* è meno flessibile delle altre due in quanto esegue il corpo almeno una volta
- Usando le istruzioni condizionali, è possibile superare tale vincolo

```
while (⟨condizione⟩)
    ⟨istruzione⟩
```

⇒

```
if (⟨condizione⟩){
    do
      ⟨istruzione⟩
    while(⟨condizione⟩)
}
```

```
do
   ⟨istruzione⟩
while (⟨condizione⟩)
```

⇒

```
⟨istruzione⟩
while(⟨condizione⟩)
    ⟨istruzione⟩
```

Equivalenza di istruzioni iterative

- In definitiva le tre istruzioni iterative sono equivalenti
- In linea di principio se ne potrebbe utilizzare una soltanto
- Tuttavia ci sono casi in cui usare un tipo di istruzione piuttosto che un’altra rende la scrittura del codice più semplice ed immediata
- La scelta del tipo di istruzione da utilizzare è questione di esperienza e gusto personale

Alcune indicazioni

- Vista la minor flessibilità del *do-while*, esso è meno usato delle altre due
  - si usa in quei casi in cui si vuole sfruttare la sua caratteristica di eseguire il corpo almeno una volta
- L'istruzione *for* viene preferita quando il numero di iterazioni da eseguire è noto (anche in forma parametrica) al momento della scrittura del codice
  - Ad esempio per far assumere ad una variabile una serie di valori in un intervallo noto (come nell'esempio visto)

Alcune indicazioni

- L'istruzione *while* viene preferita quando non è noto il numero di iterazioni all'atto della scrittura del codice
  - ad esempio il ciclo potrebbe essere ripetuto finché l'utente non immette un valore

Le istruzioni break e continue

- All'interno di un ciclo è possibile utilizzare le istruzioni *break* e *continue* per alterare la normale esecuzione del ciclo
- L'istruzione *break* causa la terminazione del ciclo più interno in cui si trova
- L'istruzione *continue* interrompe l'iterazione corrente e:
  - nel caso del *while* torna a valutare la condizione
  - nel caso del *for* esegue l'aggiornamento e poi torna a valutare la condizione

Istruzione break: esempio

```
InputWindow in = new InputWindow();
while (true){
   String str = in.readString("Inserisci una parola");
   if (str.equals("FINE"))
       break;
   else
       System.out.println("lunghezza="+str.length());
}
```

- Il codice chiede all'utente di inserire ripetutamente una parola; se questa è FINE termina altrimenti la stampa
- Si noti che in questo caso il *break* interrompe un ciclo la cui condizione è sempre vera

Istruzione continue: esempio

```
InputWindow in = new InputWindow();
int soglia = in.readInt("Inserisci un valore");
for (int i = 0; i<=100; i++){
   if (i<soglia)
        continue;
   System.out.println(i);
}
```

- Il codice scandisce tutti i numeri da zero a 100 e stampa tutti quelli sotto la soglia inserita dall'utente

Break e continue: commenti

- Le istruzioni *break* e *continue* permettono di semplificare la scrittura del codice
- Tuttavia non sono indispensabili:
  - qualunque codice usi le istruzioni *break* e *continue* può essere riscritto senza di esse
- L'uso di tali istruzioni tende a destrutturare il codice rendendolo poco leggibile
- Per questo motivo si tende a farne un uso limitato o nullo

Esempi di uso delle istruzioni di controllo

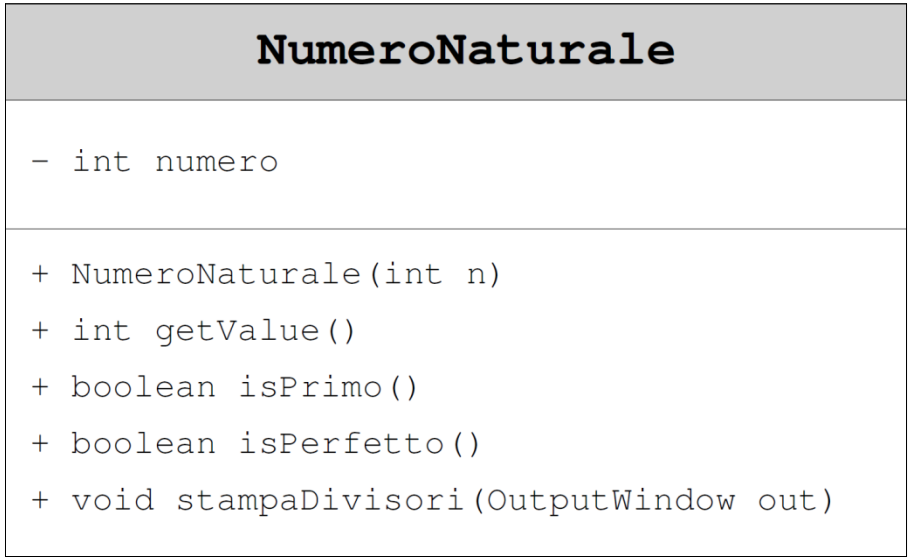
Es. 1: la classe NumeroNaturale

- Vogliamo realizzare una classe di nome *NumeroNaturale* in base alle seguenti specifiche:
  - un oggetto di tipo *NumeroNaturale* rappresenta un singolo numero intero non negativo;
  - un oggetto *NumeroNaturale* viene creato attraverso un costruttore che prende come parametro il numero rappresentato sotto forma di valore di tipo int;
  - la classe *NumeroNaturale* dispone di alcuni metodi pubblici di istanza, che consentono di verificare se il numero rappresentato gode di certe proprietà o di effettuare specifiche elaborazioni su tale numero.

Es. 1: la classe NumeroNaturale

- I metodi della classe *NumeroNaturale* sono i seguenti
  - Il metodo *getValue()* restituisce il numero naturale rappresentato dall'oggetto
  - Il metodo *isPrimo()* verifica se il numero rappresentato dall'oggetto è primo oppure no; se lo è restituisce *true*, altrimenti restituisce *false*.
  - Il metodo *isPerfetto()* restituisce *true* se il numero rappresentato dall'oggetto è un numero perfetto e *false* in caso contrario.
  - Il metodo *stampaDivisori(OutputWindow out)* stampa nella finestra grafica *out* i divisori interi del numero rappresentato dall'oggetto.

Es. 1: la classe NumeroNaturale



la classe NumeroNaturale

```
import fond.io.OutputWindow;

public class NumeroNaturale{
  private int numero;

  /* crea un oggetto Numero che
     rappresenta il numero n
  */
  public NumeroNaturale(int n){
    this.numero = n;
  }

  /* restituisce il numero rappresentato
     da questo oggetto
  */
  public int getValue(){
    return this.numero;
  }
  ...
}
```

Il metodo isPrimo

- Ricordiamo che *N* è primo se *N*>1 e non ha divisori interi oltre a 1 e a se stesso
- Algoritmo 1: scandiamo tutti i numeri nell'intervallo (1,*N*) e verifichiamo se uno di essi divide *N*
- Algoritmo 2: scandiamo tutti i numeri nell'intervallo (1,$\sqrt{N}$] e verifichiamo se uno di essi divide *N*
- In entrambi gli algoritmi appena trovo un divisore posso interrompere la scansione

Il metodo isPrimo

```
/* restituisce true se il numero rappresentato
   da questo oggetto e' un numero primo
*/
public boolean isPrimo(){
  boolean primo;
  if (this.numero<2)
    primo = false;
  else{
    primo = true;
    int i = 2;
    while (primo && i<=Math.sqrt(this.numero)){
      if (this.numero%i==0)
        primo = false;
      i++;
    }
  }
  return primo;
}
```

Il metodo isPerfetto

- Ricordiamo che *N* è perfetto se è uguale alla somma dei suoi divisori interi (compreso 1 ed escluso *N*)
  - I primi numeri perfetti sono 6, 28, 496, …
- Algoritmo:
  - scandiamo tutti i numeri da 1 fino ad *N*/2 alla ricerca di divisori interi di *N*
  - ogni volta che viene trovato un nuovo divisore, esso viene sommato a quelli già incontrati in precedenza
  - al termine della ricerca, si concluderà che *N* è un numero perfetto se e solo se la somma dei divisori trovati varrà *N*

Il metodo isPerfetto

```
/* restituisce true se il numero rappresentato
   da questo oggetto e' un numero perfetto
*/
public boolean isPerfetto(){
 boolean perfetto;
 int somma = 1;
 int i = 2;
 while (somma<=this.numero && i<=this.numero/2){
      if (this.numero%i == 0)
              somma += i;
      i++;
 }
 if (somma==this.numero)
      perfetto = true;
 else
      perfetto = false;
 return perfetto;
}
```

Nota: se *somma* diviene maggiore di *this.numero* il ciclo si interrompe

Il metodo stampaDivisori

- Algoritmo:
  - scandiamo tutti i numeri da 1 fino ad *N*/2
  - per ogni numero considerato testiamo se è un divisore di *N*
  - in caso affermativo stampiamo il numero
- Nota: la ricerca non può interrompersi prima di aver considerato tutti i possibili divisori

Il metodo stampaDivisori

```
/* stampa nella finestra grafica out
   tutti i divisori interi del numero
   rappresentato da questo oggetto
*/
public void stampaDivisori(OutputWindow out){
  for (int i = 1; i<=this.numero/2; i++)
    if (this.numero%i==0)
      out.writeln(i);
}
```

la classe di test

- Vediamo adesso una classe di test per la classe *NumeroNaturale*

```
import fond.io.*;

public class TestNumeroNaturale{
  public static void main(String[] args){
    InputWindow in = new InputWindow();
    OutputWindow out = new OutputWindow();
    int n;
    // inserimento numero
    do
      n = in.readInt("Inserisci un numero naturale");
    while (n<0);

     ...
  }
}
```

Nota l'uso del *do-while* per ripetere la lettura nel caso venga inserito un numero negativo

la classe di test

```
import fond.io.*;

public class TestNumeroNaturale{
  public static void main(String[] args){
    ...
    // test del costruttore
    NumeroNaturale num = new NumeroNaturale(n);

    // test dei metodi isPrimo e getValue
    if (num.isPrimo())
      out.writeln(num.getValue()+" e' primo");
    else
      out.writeln(num.getValue()+" non e' primo");

    // test del metodo stampaDivisori
    out.writeln("Divisori del numero "+n+":");
    num.stampaDivisori(out);

    // test del metodo isPerfetto
    out.write("Numeri perfetti fino a "+n+": ");
    for (int i = 0; i<=n; i++){
      if (new NumeroNaturale(i).isPerfetto())
       out.write(i+" ");
    }
  }
}
```

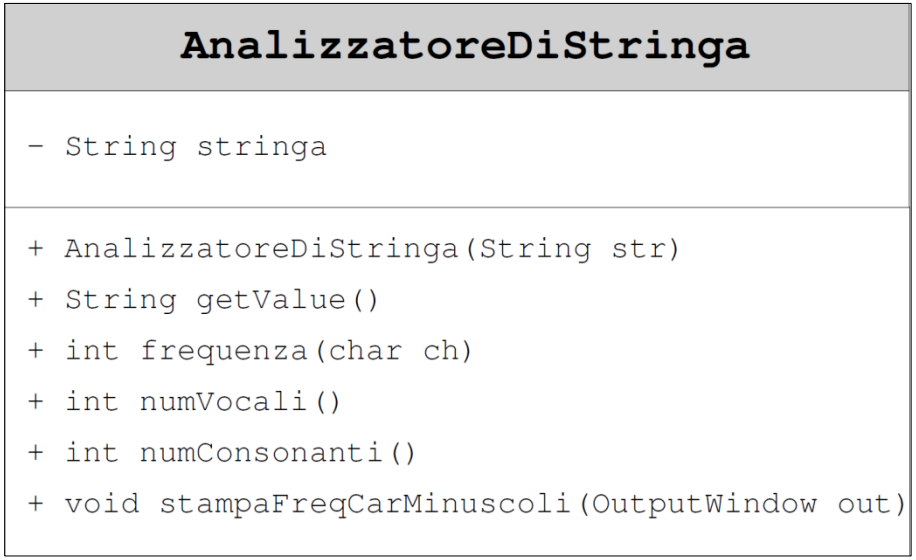
Es. 2: AnalizzatoreDiStringa

- Vogliamo ora realizzare una classe di nome *AnalizzatoreDiStringa*, tale che una sua istanza rappresenti una qualunque stringa di caratteri
- La classe sarà dotata di metodi di istanza che permetteranno di estrarre alcune statistiche relative alla stringa rappresentata

Es. 2: AnalizzatoreDiStringa

- Metodi della classe *AnalizzatoreDiStringa*:
  - Il metodo *getValue*() restituisce la stringa rappresentata dall'oggetto
  - Il metodo *frequenza*(*char ch*) restituisce il numero di occorrenze del carattere *ch* nella stringa rappresentata dall'oggetto
  - Il metodo *numVocali*() restituisce il numero di vocali presenti nella stringa rappresentata dall'oggetto
  - Il metodo *numConsonanti()* restituisce il numero di consonanti presenti nella stringa rappresentata dall'oggetto
  - Il metodo *stampaFreqCarMinuscoli(OutputWindow out*) stampa nella finestra grafica out la frequenza di ogni carattere minuscolo presente nella stringa rappresentata dall'oggetto

Es. 2: AnalizzatoreDiStringa



La classe AnalizzatoreDiStringa

```
import fond.io.*;

public class AnalizzatoreDiStringa{
  private String stringa;

  /* crea un oggetto che rappresenta
     la stringa str
  */
  public AnalizzatoreDiStringa(String str){
    this.stringa = str;
  }

  /* restituisce la stringa rappresentata
     da questo oggetto
  */
  public String getValue(){
    return this.stringa;
  }

  ...
}
```

Il metodo frequenza

- Algoritmo:
  - scandiamo tutti i caratteri della stringa
  - ogni carattere considerato viene confrontato con il carattere cercato
  - se il carattere corrente è uguale a quello cercato incrementiamo di uno un contatore inizialmente posto a zero

Il metodo frequenza

```
/* restituisce la frequenza del carattere ch
   all'interno della stringa rappresentata
   da questo oggetto
*/
public int frequenza(char ch){
  int f = 0;
  for (int i = 0; i<this.stringa.length(); i++){
      char chTemp = this.stringa.charAt(i);
      if (chTemp==ch)
            f++;
  }
  return f;
}
```

Il metodo numVocali

- Algoritmo:
  - scandiamo tutti i caratteri della stringa
  - se il carattere corrente è una vocale incrementiamo di uno un contatore inizialmente posto a zero
- Per evitare di dover confrontare il carattere corrente sia con le vocali maiuscole che con le minuscole utilizziamo una stringa *temp* ottenuta da quella di partenza utilizzando il metodo *toUpperCase*

Il metodo numVocali

```
/* restituisce il numero di vocali della
   stringa rappresentata da questo oggetto
*/
public int numVocali(){
  int vocali = 0;
  String temp = this.stringa.toUpperCase();
  for (int i = 0; i<temp.length(); i++){
    char chTemp = temp.charAt(i);
    if ((chTemp=='A') ||
        (chTemp=='E') ||
        (chTemp=='I') ||
        (chTemp=='O') ||
        (chTemp=='U'))
          vocali++;
  }
  return vocali;
}
```

Il metodo numConsonanti

- Il metodo *numConsonanti* è analogo al metodo *numVocali*
- Poiché però le consonanti sono più delle vocali sarebbe scomodo elencarle tutte come abbiamo fatto per le vocali

- Nota: non è possibile ottenere il numero di consonanti sottraendo il numero di vocali al numero complessivo di caratteri:
  - esistono caratteri che non sono né vocali né consonanti

Il metodo numConsonanti

```
 /* restituisce il numero di consonanti della
    stringa rappresentata da questo oggetto
*/
public int numConsonanti(){
   int consonanti = 0;
   String temp = this.stringa.toUpperCase();
   for (int i = 0; i<temp.length(); i++){
     char chTemp = temp.charAt(i);
     if ((chTemp>='B' && chTemp<='D') ||
         (chTemp>='F' && chTemp<='H') ||
         (chTemp>='J' && chTemp<='N') ||
         (chTemp>='P' && chTemp<='T') ||
         (chTemp>='V' && chTemp<='Z'))
           consonanti++;
    }
    return consonanti;
}
```

Il metodo stampaFreqCarMinuscoli

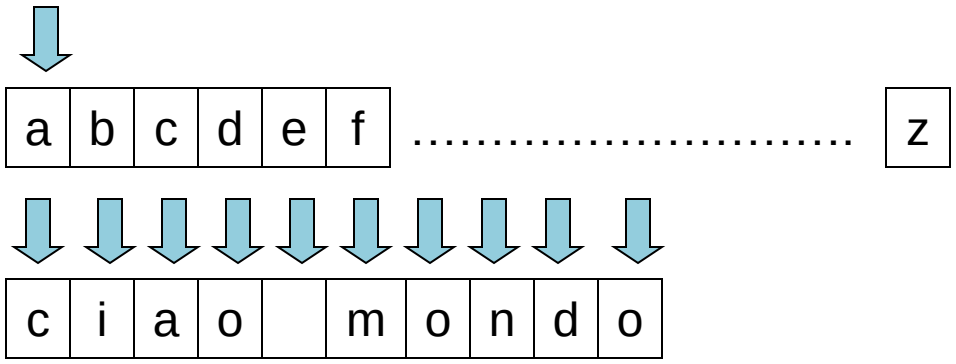
- La strategia è la stessa vista per il metodo *frequenza* applicata a tutti i possibili caratteri minuscoli dell'alfabeto
- Scandiamo tutti i caratteri minuscoli dell'alfabeto
- per ognuno di essi applichiamo la strategia descritta per il metodo *frequenza*

Il metodo stampaFreqCarMinuscoli

```
public void stampaFreqCarMinuscoli(OutputWindow out){
  for (char ch = 'a'; ch<='z'; ch++){
    int f = 0; // frequenza del carattere ch
    for (int i = 0; i<this.stringa.length(); i++){
      char chTemp = this.stringa.charAt(i);
      if (chTemp==ch)
        f++;
    }
    if (f>0)
      out.writeln("frequenza di "+ch+": "+f);
  }
}
```

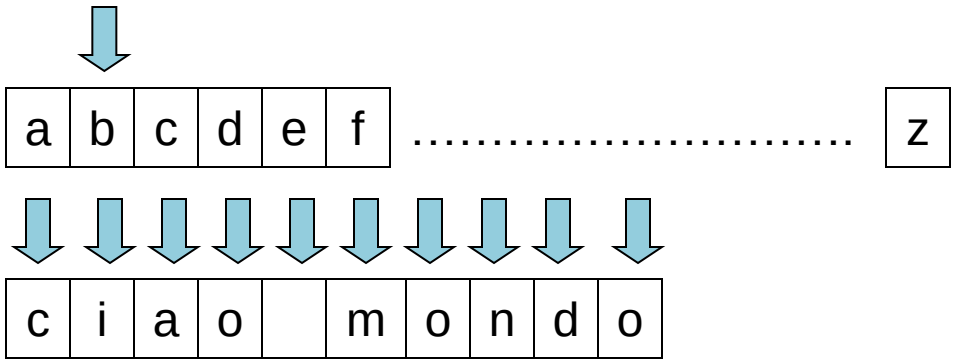
Il metodo stampaFreqCarMinuscoli

- Si noti l'uso di due cicli *for* annidati:
  - il primo scandisce tutti i caratteri da *'a'* a *'z'*
  - per ogni valore fissato di *ch* si scandisce l'intera stringa (secondo ciclo *for*) in maniera analoga a quanto fatto nel metodo *frequenza*



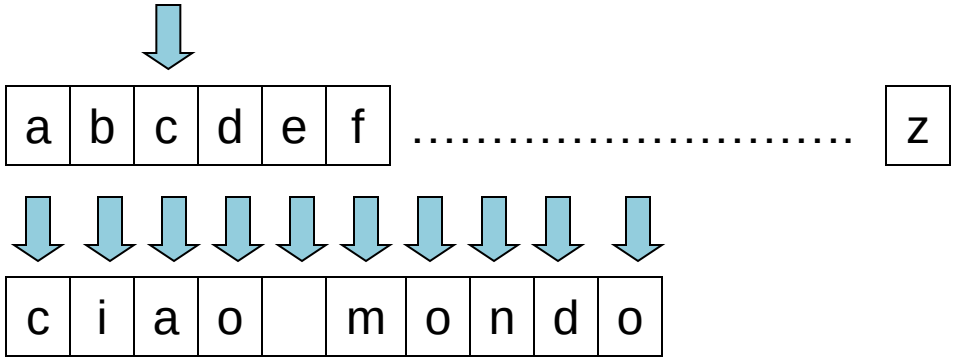
Il metodo stampaFreqCarMinuscoli

- Si noti l'uso di due cicli *for* annidati:
  - il primo scandisce tutti i caratteri da *'a'* a *'z'*
  - per ogni valore fissato di *ch* si scandisce l'intera stringa (secondo ciclo *for*) in maniera analoga a quanto fatto nel metodo *frequenza*



Il metodo stampaFreqCarMinuscoli

- Si noti l'uso di due cicli *for* annidati:
  - il primo scandisce tutti i caratteri da *'a'* a *'z'*
  - per ogni valore fissato di *ch* si scandisce l'intera stringa (secondo ciclo *for*) in maniera analoga a quanto fatto nel metodo *frequenza*



Il metodo stampaFreqCarMinuscoli

- Si noti l'uso di due cicli *for* annidati:
  - il primo scandisce tutti i caratteri da *'a'* a *'z'*
  - per ogni valore fissato di *ch* si scandisce l'intera stringa (secondo ciclo *for*) in maniera analoga a quanto fatto nel metodo *frequenza*

- È possibile implementare il metodo *stampaFreqCarMinuscoli* in maniera più compatta riutilizzando il metodo *frequenza*

Il metodo stampaFreqCarMinuscoli

```
/* visualizza nella finestra grafica out
    la frequenza di ogni carattere minuscolo
    presente nella stringa
 */
 public void stampaFreqCarMinuscoli(OutputWindow out){
   for (char ch = 'a'; ch<='z'; ch++){
     int f = this.frequenza(ch);
     if (f>0)
       out.writeln("frequenza di "+ch+": "+f);
   }
 }
```