Utilizzare stringhe

Utilizzare stringhe

- Consideriamo:

```
String vuota = "";
String s = null;
String p;
System.out.println("la stringa vuota
  ha lunghezza " + vuota.length());
```

- E le altre due stringhe? Cosa accade se scriviamo

`s.length()` `p.length()`

la lunghezza è 0, o si verifica un errore?

Utilizzare stringhe

- Attenzione: gli errori sono diversi:
- La stringa **`s`** è stata inizializzata ma la sua referenza è **`null`** e pertanto l'esecuzione del programma si interrompe segnalando un errore di tipo NullPointerException.
- La stringa **`p`** non è stata inizializzata e quindi non si può accedere al campo length: errore in compilazione.

Confrontare stringhe

- Abbiamo visto che per confrontare numeri interi si usa l'operatore di uguaglianza **==** .
- Abbiamo visto che anche i numeri reali si possono confrontare con lo stesso operatore **==** ma che a volte è preferibile usare un confronto del tipo Math.abs(a-b) **<** epsilon (con epsilon=10^{-n} e n compatibile con la precisione float o double), dato che i reali sono rappresentati in maniera approssimata.

Confrontare stringhe

- Ci chiediamo ora se lo stesso operatore si può usare anche con le stringhe (par. 5.2.3).

```
String s1 = "ciao";
String s2 = "ci" + "ao"; //s2 = "ciao"
String s3 = s1;
```

- Il risultato dei confronti

```
if (s1 == s2)
if (s1 == s3)
```

- Sarà vero in entrambi i casi? Ci sarà un errore in compilazione?

Confrontare stringhe

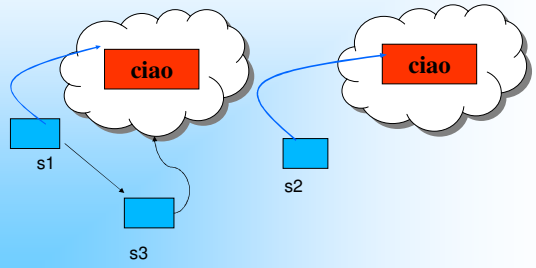
- È lecito eseguire il confronto

```
if (s1 == s2)
```

il compilatore non segnala errore; ma quale è il suo significato?

- Le variabili oggetto **`s1`** ed **`s2`** contengono ciascuna il **riferimento** al proprio oggetto: il primo costruito con l'assegnazione della costante "ciao", il secondo con la concatenazione delle due costanti "ci" e "ao".
- I due oggetti sono distinti e così pure i riferimenti.

Confrontare stringhe



Confrontare stringhe

- Invece il secondo confronto

```
if(s1 == s3)
```

- sarà sicuramente vero perché c'è stata l'assegnazione **`s3 = s1`**, e pertanto i due riferimenti sono uguali: vedono lo stesso oggetto.
- Per confrontare due stringhe e vedere se il valore contenuto sia uguale si deve usare il metodo **equals**.

Confrontare stringhe

- Possiamo quindi scrivere:

```
if (s1.equals(s2))
   System.out.prinln("le due
            stringhe sono uguali");
```

- Se invece vogliamo sapere se una stringa precede o segue un'altra stringa usiamo il metodo **compareTo**.

```
if (s1.compareTo("mondo") < 0)
      System.out.println("ciao
               precede   mondo");
```

Confrontare stringhe

- Il metodo **equals**.
- È un metodo predicativo, restituisce false se i dati dei due oggetti sono diversi e restituisce true se sono uguali.
- Il metodo **compareTo**.
- È un metodo che restituisce un valore intero:

s1. compareTo(s2)
- **>0** se s1 **segue** s2
- **=0** se s1 è **uguale** s2
- **<0** se s1 **precede** s2

Confrontare stringhe

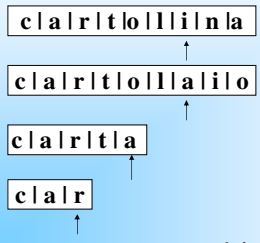
- La stringa **s1** invoca il metodo **compareTo**:

```
if(s1.compareTo(s2) > 0)
    //s1 è più grande di s2: s1 segue

if(s1.compareTo(s2) < 0)
    //s1 è più piccola di s2: s1 precede

if(s1.compareTo(s2) == 0)
    //s1 è uguale a s2
```

Confrontare stringhe

- Si effettua un **confronto lessicografico** secondo l'ordine dei caratteri UNICODE.
- Il confronto procede a partire dal primo carattere delle stringhe confrontando a due a due i caratteri nelle posizioni corrispondenti, finché una delle stringhe termina oppure due caratteri sono diversi
  - se una stringa termina, essa precede l'altra
  - se terminano entrambe, sono uguali
  - altrimenti, l'ordinamento tra le due stringhe è uguale all'ordinamento alfabetico tra i due caratteri diversi.

Confrontare stringhe



car < carta < cartolaio < cartolina

Confrontare stringhe

- Il confronto lessicografico genera un ordinamento **simile** a quello di un dizionario: infatti l'ordinamento tra caratteri in Java è diverso perché tra i caratteri non ci sono solo le lettere, ed inoltre:
  - i numeri precedono le lettere
  - tutte le lettere maiuscole precedono tutte le lettere minuscole
  - il carattere di "spazio" precede tutti gli altri caratteri

Altri metodi sulle stringhe

- Un altro metodo importante è **charAt**.
- Questo metodo ha per argomento un numero intero **n** (che deve essere compreso tra 0 e la lunghezza della stringa meno 1, altrimenti si genera un errore in esecuzione) e restituisce il carattere corrispondente alla posizione **n**.
- Esempio.

```
String s = "ciao";     //lunga 4
char car = s.charAt(3);   //car='o'
     car = s.charAt(0);   //car ='c';
```

Altri metodi sulle stringhe

- Può essere utile considerare un singolo carattere invece di una stringa di lunghezza unitaria.
- Infatti **char** è un **tipo base** mentre **String** è un **oggetto**: una variabile di tipo char occupa meno spazio in memoria di una stringa di lunghezza unitaria e le elaborazioni su variabili di tipo char sono più veloci di quelle su stringhe.
- Se si vogliono considerare tutti i caratteri di una stringa per elaborarli si utilizza un ciclo del tipo:

```
String s = "Pippo";
for (int i = 0; i < s.length(); i++){
        char car = s.charAt(i);
         // elabora car
}
```

Altri metodi sulle stringhe

- Una variabile di tipo **char** può essere argomento del metodo print (println), può essere concatenata con una stringa tramite l'operatore + e in tal caso, come con le altre variabili, verrà convertita a stringa.
- Una variabile di tipo **char** può anche essere confrontata con una **costante** di tipo **char**:

```
char car = s.charAt(1);
if(car == 'i')
    System.out.println(car +
            "e' uguale a  i ");
```

- Il singolo carattere può anche essere una "sequenza di escape": '\n'.

Altri metodi sulle stringhe

- Se invece si vuole conoscere quale è la posizione di un carattere nella stringa si usa il metodo **indexOf**.

```
String s = "ciao";
int indice = s.indexOf('i');
          //indice = 1
```

- Se il carattere non è presente il metodo restituisce -1

Scomporre stringhe

- Può essere utile individuare in una stringa delle sottostringhe individuate da spazi.
- Supponiamo di leggere una riga di testo composta da più elementi e di volerli poi considerare come valori per variabili di tipo diverso. Lo Scanner non considera spazi e tabulazioni con i quali separiamo le singole "parole" (token).
- Per separare le varie componenti si può usare la classe StringTokenizer, del pacchetto java.util

Scomporre stringhe

- Supponiamo che la linea di testo sia così composta:

  ciao 20 3.5 fine

```
Scanner in = new Scanner(System.in);
String line;
line = in.nextLine();
```

- Costruiamo un oggetto di tipo StringTokenizer:

```
StringTokenizer t =
       new StringTokenizer(line);
```

Scomporre stringhe

- Possiamo ora utilizzare sull'oggetto **`t`** i metodi:
- nextToken(): restituisce una stringa che è il token, se c'è, ed avanza al prossimo, altrimenti dà errore;
- hasMoreTokens(): restituisce true se ci sono ancora token nella stringa e false se sono finiti;
- countToken(): restituisce un intero che rappresenta il numero di token nella stringa.

Scomporre stringhe

- Esempio. Vogliamo leggere i tutti i singoli token:

```
import java.util.StringTokenizer;
…
StringTokenizer t =
        new StringTokenizer(line);
while(t.hasMoreTokens()){
   String s = t.nextToken();
   //utilizzare s
}
```

Scomporre stringhe

- Se tra i token ci sono valori che vogliamo interpretare come numeri, dobbiamo utilizzare dei metodi che trasformano la stringa in numero (analogamente a nextInt, nextDouble, …):

```
int a = Integer.parseInt(s);
double b = Double.parseDouble(s);
```

- Si tratta di metodi statici: non c'è il nome di un oggetto ma c'è il nome di una classe. Per ogni tipo base c'è una classe (involucro) corrispondente.

Scomporre stringhe

- Questi metodi possono essere utili anche per gestire un ciclo di lettura per variabili numeriche, che termina quando si inserisce una stringa con il significato di "fine dati".

```
boolean finito=false;
System.out.println("per terminare batti:
                          fine ");
while(!finito){
  String st = in.next();  //t.nextToken()
  if(st.equals("fine"))
        finito =true;
  else{ double a = Double.parseDouble(st);
        //elabora a
      }
 }
```

Metodi statici

Metodi statici

- Un **metodo statico** non ha un parametro implicito; viene chiamato anche **metodo di classe** (par. 8.6).
- I metodi che hanno parametro **implicito** (che vengono invocati da un oggetto) si chiamano anche **metodi di esemplare** o di **istanza**.
- I metodi statici si usano tutte le volte che si vogliono raggruppare istruzioni che coinvolgono numeri. Ad esempio i metodi della classe **Math** sono statici.

Metodi statici

- Un metodo statico viene definito in una classe; per utilizzarlo da un ambiente diverso dalla classe nella qual è definito, si invoca scrivendo il nome di tale classe.
- Sintassi.

```
NomeClasse.nomemetodo(…)
```

- Anche il metodo main è statico: infatti quando la JVM attiva il main, non è stato creato ancora alcun oggetto, di conseguenza il primo metodo attivato deve essere statico.

Metodi statici

- Spesso usiamo gruppi di istruzioni che risolvono problemi specifici e che non creano oggetti. Allo scopo di aumentare la leggibilità del codice e di permetterne il riutilizzo, può essere utile costruire un metodo contenente quelle istruzioni; tale metodo non è legato ad un oggetto e pertanto può essere un metodo di classe.
- Consideriamo le istruzioni per stampare un array.

Metodi statici

```
int v[] = new int [...];
 //acquisire le componenti di v
  //stampare v
for(int i=0; i<v.length; i++){
  System.out.print(v[i] + '\t');
   if((i+1)%5 == 0)
        System.out.println();
}
System.out.println();
```

Metodi statici

- Supponiamo di dover stampare nuovamente l'array perché abbiamo fatto delle modifiche, oppure di avere un altro array da stampare; invece di scrivere più volte le stesse istruzioni, è conveniente costruire un metodo per eseguire la stampa: tale metodo sarà static:

```
public static void stampa(int[] t){
   for(int i=0; i<t.length;i++){
        //istruzioni per la stampa
   }
}//fine stampa
```

Metodi statici

- Dove scriviamo il codice di questo metodo?
- Possiamo scrivere il metodo nella classe che contiene il main.
- Possiamo costruire una classe che contiene metodi per l'uso di array, in analogia con i metodi della classe System, con i quali si effettuano ad esempio copie di array (o di parte di array) in altri array.

Metodi statici

- **I caso**. Il metodo sta nella classe del main.

```
public class Usoarray{
  public static void main (String[] arg){
    int[] v = new int [100];
    int[] a = new int [20];
    //leggi v e a
    stampa(v);    //invocazione del metodo
    stampa(a);
    }//fine main
   public static void stampa(int[] t){
    //codice del metodo di stampa
    }
}//fine classe Usoarray
```

Metodi statici

- **II caso**. Costruiamo una classe AlgoritmiArray in cui inseriamo tutti i metodi per un "comodo" uso di array: lettura, stampa, raddoppio, ridimensiona, copia,….

```
public class AlgoritmiArray{
    public static void stampa (int[] t){
       …}
    public static void copia (int[] t, int[]w){
       …}
}//fine classe AlgoritmiArray
```

Per invocare il metodo da fuori della classe:

```
AlgoritmiArray.stampa(v);
```

Metodi statici

- Quando definiamo un array **v** dobbiamo scegliere la **dimensione** dell'array; per poter eseguire prove diverse con lo stesso array consideriamo una dimensione "abbastanza grande" e successivamente acquisiamo un valore **n** per il numero effettivo delle componenti che l'array deve avere in quella applicazione. Tale numero dovrà essere compatibile con la dimensione: **n ≤** oppure **<** della dimensione (a seconda dell'utilizzo oppure no della componente di indice 0).

Metodi statici

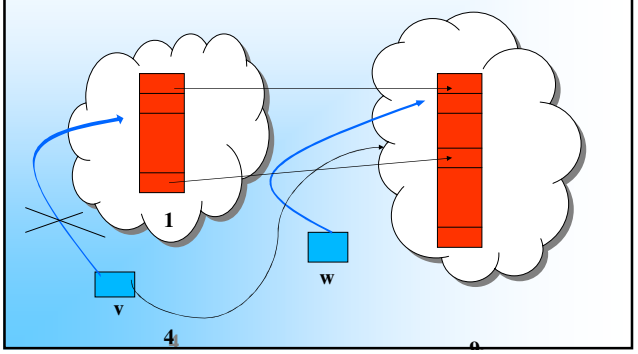
- In tale modo, se è **n < v.length**, il nostro array ha delle **componenti libere** che vanno dall'indice successivo a quello dell'ultima componente fino a v.length-1: si dice allora che l'**array** è **riempito solo in parte**.
- Può però accadere che abbiamo bisogno di utilizzare l'array per un numero di componenti maggiori di v.length: in questo caso utilizziamo la tecnica del **raddoppio** con la quale andremo a costruire un **nuovo array w** di dimensione doppia di quello iniziale, nel quale copieremo i valori del primo e del quale andremo poi a copiare la referenza nella variabile **v**.

Metodi statici

```
//raddoppio di un array
int w[] = new int [v.length*2];
//copia v in w
for (int k=0; k<v.length; k++)
     w[k]=v[k];
//copia la referenza
v=w;
```

- Attenzione: copiare la referenza si può in Java perché **v** è un riferimento ad un array di interi (di dimensione qualunque); non è possibile fare ciò in altri linguaggi.

Raddoppio di un array



Metodi statici

- Se voglio costruire un metodo per eseguire il raddoppio, dovrò fare in modo che la nuova **referenza** venga **restituita**:

```
public static int[] raddoppio(int[] t){
   //istruzioni di raddoppio
   return t;   }
```

- L'invocazione del metodo (all'interno della classe che lo contiene) sarà:

```
if(n == v.length)
        v=raddoppio(v);
```

oppure, se siamo fuori dalla classe AlgoritmiArray

```
v=AlgoritmiArray.raddoppio(v);
```

Metodi statici

- È comodo gestire un'array "tutto pieno" perché l'ultima componente ha indice **`v.length-1`**.
- Possiamo allora "buttare via " gli elementi non in uso. Vogliamo un metodo per "ridimensionare" l'array vale a dire un metodo al quale **passare** l'array e il numero di componenti inserite e che ci **restituisca** la nuova referenza.

```
if( n<v.length)
       v=ridimensiona(v,n);
```

Metodi statici

```
public static int[] ridimensiona
                (int[] t , int n){
    int[] w = new int[n];
    for(int k=0; k<n; k++)
           w[k]= t[k];
    //copia la referenza
    t=w;
    return t;
}
```

- Costruiremo anche metodi per: inserire elementi, togliere elementi, ecc.; in tale modo non ci accorgeremo che l'array è a dimensione fissa.

Metodi statici

- La classe System mette a disposizione un comodo metodo per copiare i valori di un array in un altro. Molto probabilmente tale metodo sarà più efficiente del nostro. Perché **non lo usiamo**?
- Perché **noi stiamo imparando a scrivere algoritmi** e pertanto dobbiamo **scriverli da soli**.
- Però possiamo anche sapere come lo si usa (anche se **non** lo useremo).

Il metodo System.arraycopy

- Invochiamo il metodo statico **arraycopy** della classe System (par. 7.7)

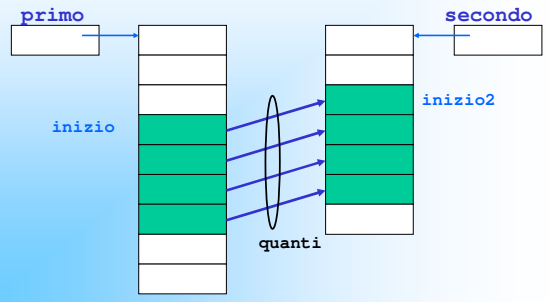
```
double[] primo = new double[10];
// inseriamo i dati nell'array
...
double[] secondo = new double[primo.length];
System.arraycopy(primo, inizio, secondo, inizio2,
                                          quanti);
```

- Il metodo System.arraycopy consente di copiare un porzione di un array in un altro array (grande **almeno** quanto la porzione che si vuol copiare). Se vogliamo copiarlo tutto:

```
System.arraycopy(primo, 0, secondo, 0,
                 primo.length);
```

Il metodo System.arraycopy

```
System.arraycopy(primo,inizio,secondo,inizio2,quanti);
```



Inserimento e rimozione di elementi in un array

Inserimento e rimozione di elementi in un array

- L'eliminazione di un elemento da un array richiede due algoritmi diversi
- **Caso I.** L'ordine tra gli elementi dell'array non è importante e l'elemento da eliminare è unico: è sufficiente copiare l'ultimo elemento dell'array nella posizione dell'elemento da eliminare; si può successivamente ridimensionare l'array, oppure tenere l'array riempito parzialmente, diminuendo di 1 il numero degli elementi.

Inserimento e rimozione di elementi in un array

```
//array tutto pieno
double[] v = {1, 2.3, 4.5, 5.6};
int indice = 1;
v[indice] = v[v.length - 1];
            //v[3]=5.6 al posto di 2.3
v = ridimensiona(v,v.length-1);

//array riempito solo in parte
int n=4;
v[indice] = v[n - 1];
n--;         //v[--n];
```

Inserimento e rimozione di elementi in un array

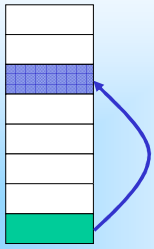
- **Caso II.** Se invece l'ordine tra gli elementi deve essere mantenuto, l'algoritmo è più complesso: **tutti** gli elementi il cui indice è maggiore dell'indice dell'elemento da rimuovere devono essere **spostati** nella posizione con indice immediatamente inferiore (**copiare il successivo sul precedente** a partire dal **primo** indice che **deve rimanere**); si può successivamente ridimensionare l'array, oppure tenere l'array riempito parzialmente, diminuendo di 1 il numero degli elementi.

Inserimento e rimozione di elementi in un array

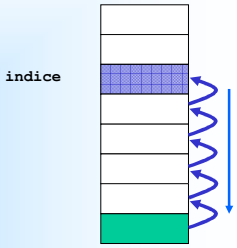
```
//array tutto pieno
  double[] v = {1, 2.3, 4.5, 5.6};
  int indice = 1;
  for(int i=indice; i<v.length-1; i++)
          v[i] = v[i+1];
  //copia v[2] su v[1]; v[3] su v[2], ...
  v = ridimensiona(v,v.length-1);

//analogamente con array riempito solo
//in parte
```

Inserimento e rimozione di elementi in un array

Senza ordinamento e unico



Con ordinamento



i trasferimenti vanno eseguiti dall'alto in basso

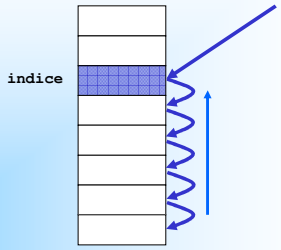
Inserimento e rimozione di elementi in un array

- Per inserire un elemento in un array si può: aggiungerlo alla fine oppure in una posizione stabilita: in questo caso bisogna "fargli spazio".
- **Tutti** gli elementi il cui indice è maggiore dell'indice della posizione voluta devono essere **spostati** nella posizione con indice immediatamente superiore: **copiare** un elemento **sul successivo** a partire **dall'ultimo elemento** dell'array.
- Infine introdurre l'elemento.

Inserimento e rimozione di elementi in un array

```
double[] v = {1, 2.3, 4.5, 5.6};
v = aumenta(v, v.length + 1);
//v.length + 1 = 5 copiare v in un array con
  //una componente in più, che rimane libera
int indice = 2;
for (int i = v.length – 1; i > indice; i--)
          v[i] = v[i – 1];
//copia v[3] su v[4], v[2] su v[3], . . .
v[indice] = 3.5;

//l'array è:  1, 2.3, 3.5, 4.5, 5.6
//analogamente con un array riempito solo in
  //parte
```

Inserimento e rimozione di elementi in un array



i trasferimenti vanno eseguiti dal basso in alto

Passaggio dei parametri

Passaggio dei parametri

- Nel metodi per array abbiamo visto che l'array era un parametro esplicito di un metodo ma era anche un valore di ritorno.
- Ricordiamo che l'array è una struttura di dati, ma che in Java un array è realizzato come oggetto e pertanto il nome dell'array (variabile oggetto) contiene la referenza all'oggetto creato e tale referenza è uno scalare (l'area di memoria assegnata dal compilatore può contenere un solo valore).

Passaggio dei parametri

- In tutti i linguaggi di programmazione si hanno sottoprogrammi (funzioni, procedure, metodi) che rappresentano specifici problemi e che possono essere chiamati (invocati) da altri sottoprogrammi.
- Tra il metodo "chiamante" e il metodo "chiamato" si ha un **passaggio di informazioni**.
- Nel metodo chiamante abbiamo:

```
cliente1.deposit(500);
v = ridimensiona(v,n);
```

Passaggio dei parametri

- Nella classe del metodo abbiamo la firma:

```
public void deposit(double amount)
public static int[] ridimensiona (int[] t,
                                  int n)
```

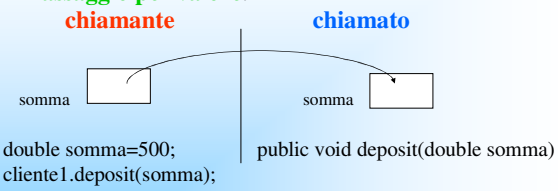
- Si usano nomi diversi per indicare i parametri espliciti di un metodo:
  - **argomenti** o **parametri effettivi**, quando ci si riferisce ai parametri del metodo nell'istruzione di chiamata che compare nel metodo **chiamante**
  - **parametri formali** quando ci si riferisce ai parametri della firma del metodo **chiamato**.
- L'argomento ha un valore che viene passato; il parametro formale assume un valore solo quando il metodo viene chiamato.

Passaggio dei parametri

- Il passaggio dei parametri avviene secondo due modalità dette:
  - per **valore**
  - per **indirizzo**
- Ci sono linguaggi, come C, Pascal, che hanno entrambi questi passaggi: si utilizza un simbolo o una parola chiave per indicare le due modalità.
- In altri linguaggi si ha un solo tipo di passaggio: in Fortran si ha solo il passaggio per indirizzo, in Java solo quello per valore (Argomenti avanzati 8.1).

Passaggio dei parametri

- **Passaggio per valore**.



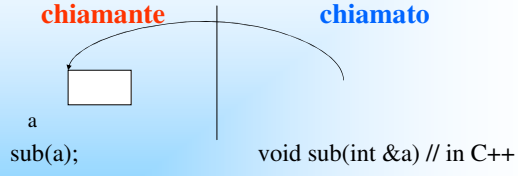
- Si tratta di **due variabili diverse**. Quando si attiva il metodo, il **valore** della variabile somma del chiamante viene "**copiato**" nella variabile somma del chiamato.

Passaggio dei parametri

- Se si effettua un passaggio per valore, la variabile del chiamato può anche assumere un nuovo valore (apparire a sinistra del simbolo di assegnazione); tale valore non apparirà nel chiamante, vale a dire: **il valore nel chiamante non viene modificato**.
  - Si dice anche che la variabile di scambio (parametro esplicito) è in ingresso.

Passaggio dei parametri

- **Passaggio per indirizzo**.



- Si tratta della **stessa variabile**. Quando si attiva il metodo, viene **passato** l'**indirizzo** di memoria della variabile del chiamante.

Passaggio dei parametri

- In tale modo l'area di memoria è visibile ad entrambi i metodi; pertanto **ogni modifica** fatta sulla variabile **a** nel chiamato si ritrova nel chiamante: la variabile ritorna modificata.
- In Java si ha solo il passaggio per valore.
- Quando passiamo un parametro esplicito, esso non torna indietro modificato.
- Se vogliamo ottenere una modifica dobbiamo restituire quel valore come tipo di ritorno del metodo.

Passaggio dei parametri

- Vediamo cosa accade con gli array.
- Il riferimento dell'array è passato per valore. Possiamo variare i valori contenuti nell'array?
- Sì infatti, attraverso il riferimento possiamo accedere ai campi dell'array.

```
//chiamante
int[] v ={1,2,3};
modifica(v);
    //chiamato
    public static void modifica(int[] t){
                t[0] = 5;}
//nel chiamante la componente 0 vale 5
```

Passaggio dei parametri

- Quando utilizziamo un metodo per eseguire il raddoppio dell'array, non possiamo definirlo void e pensare che il nuovo riferimento "ritorni" al chiamante: l'area doppia è attiva nel chiamato ma il riferimento nuovo non è visibile al chiamante.
- Dobbiamo perciò definire un metodo per il raddoppio con il tipo "riferimento ad array": è quello che abbiamo visto:

```
public static int[] raddoppio(int[]t)
```

Passaggio dei parametri

- Nella classe AlgoritmiArray vedremo dei metodi per acquisire valori per un array di interi.
- Questi metodi restituiscono la referenza del nuovo array; infatti utilizzeremo la tecnica del raddoppio, se non vogliamo decidere a priori quante sono le componenti da acquisire.
- Scriveremo anche metodi con il numero di componenti dato come parametro esplicito.

Argomenti sulla riga di comando

Argomenti sulla riga di comando

- Quando si esegue un programma Java, è possibile fornire dei parametri **dopo** il nome della classe che contiene il metodo **`main`**.
- Tali parametri vengono letti dall'interprete Java e trasformati in un array di stringhe che costituisce il parametro del metodo **`main`**.
- Supponiamo di inserire due valori per il programma di nome Prova

```
java Prova mio 50
```

Argomenti sulla riga di comando

```
public class Prova {
   public static void main
              (String[] arg) {
      System.out.println(arg.length);
      System.out.println(arg[0]);
      System.out.println(arg[1]);
      . . .
   }
}
```

- La lunghezza dell'array è 2: il primo argomento è la stringa "mio", il secondo argomento il valore 50.

Argomenti sulla riga di comando

- Questi valori inseriti possono essere dei dati di ingresso da elaborare, spesso si inserisce il nome di uno o più file che si vogliono collegare al programma oltre ai file standard System.in e System.out.
- Nel laboratorio vedremo un esempio di dati inseriti e che devono essere elaborati.
- Il parametro del main è un array di tipo String, perciò se dobbiamo considerare valori numerici dobbiamo poi trasformarli.

Argomenti sulla riga di comando

- Finora non abbiamo inserito argomenti: cosa veniva associato all'array di stringhe **`arg`**?
- Se non vengono forniti parametri sulla riga di comando, al metodo **`main`** viene passato un array di lunghezza zero.
  - Si poteva pensare che **`arg = null`**; invece il parametro fornito al metodo **`main`** non ha mai il valore **`null`**.
- Esercizio. Pensare ad un modo per verificare quanto detto.

Scelte multiple: switch

Scelte multiple: switch

- L'enunciato switch si usa al posto di una situazione in cui ci siano scelte del tipo "else if" annidate (Argomenti avanzati 5.2).
- Esempio.

```
int x, y;
if (x == 1)
      y = 1;
else if (x == 2)
      y = 4;
else if (x == 4)
      y = 16;
```

Scelte multiple: switch

- Sintassi.

```
int k;
//acquisire o assegnare un valore a k
switch(k){
   case  costante1: istruzione;
   case  costante2: istruzione;
   case  costante3: istruzione;
}
```

Scelte multiple: switch

- Semantica.
- Il valore di **k** viene **cercato** tra i valori presenti nella **case** e se viene trovato si esegue l'istruzione indicata e **tutte le successive**.
- Se si vuole che le istruzioni siano in alternativa, come avviene con l'annidamento delle if, si deve utilizzare la parola chiave **break**.

Scelte multiple: switch

```
switch(k){
   case  costante1: istruzione; break;
   case  costante2: istruzione; break;
   case  costante3: istruzione;
  }
```

- Se il valore di k non è presente, non vi è alcuna segnalazione.
- Per gestire meglio il caso "valore di k non presente", si può introdurre un "default".

Scelte multiple: switch

```
switch(k){
   case  costante1: istruzione; break;
   case  costante2: istruzione; break;
   case  costante3: istruzione; break;
  default: System.out.println("il
        valore di k non e' presente");
  }
```

- Se non si usa break, l'ordine delle **case** fa variare le operazioni da eseguire.

Scelte multiple: switch

- Esempio.

```
switch (x){
   case 1: y = 1;
   case 2: y = 4;
   case 4: y = 16;
}
        //se x = 1 y = 16

switch (x){
   case 4: y = 16;
   case 2: y = 4;
   case 1: y = 1;
}
          //se x = 1 y = 1
```

Scelte multiple: switch

- Questo enunciato è utile quando si devono scegliere metodi diversi da eseguire in alternativa, come un menu che propone delle scelte: la lettura del menu è sicuramente più chiara che non l'elenco delle varie "else if".
- Se la variabile non è intera o se i valori non sono delle costanti non è possibile utilizzare l'enunciato switch.

Ciclo do

Ciclo do

- Capita a volte di dover eseguire il corpo di un ciclo almeno una volta, per poi ripeterne l'esecuzione se è verificata una particolare condizione (Argomenti avanzati 6.1).
- Esempio: leggiamo un valore in ingresso e vogliamo che esso sia positivo: se il valore non è positivo vogliamo poter ripetere la lettura.

Ciclo do

- Sintassi.

```
do {
//iterazione
}
while(condizione);
```

- Semantica.

  L'iterazione viene eseguita la prima volta; viene valutata la condizione: se è vera si esegue nuovamente l'iterazione, se è falsa il ciclo termina.

Ciclo do

- Equivale a scrivere:

```
iterazione;
while(condizione){
   iterazione;
}
```

- oppure

```
boolean continua =true;
while(continua){
   iterazione;
   if(!condizione)
      continua=false;
}
```

Ciclo do

- Problema. Calcolare la somma

  $1 + 1./2 + 1./3 + 1./4 + \ldots + 1./n + \ldots$

  fino a quando $1./n > t$ con t valore stabilito (ad esempio $t = 10^{-9}$).

- Provare a scrivere l'algoritmo usando sia il ciclo while che il ciclo do.