Parte terza: strutture di controllo del flusso, array e stringhe.

Ci addentriamo in una parte essenziale del linguaggio Java. In questa parte prenderemo in esame come gestire l'input/output (in una applicazione a interfaccia a caratteri), le eccezioni, come utilizzare le strutture di controllo del flusso, gli array e le stringhe, con molti metodi.

1. Gestione dell' I/O

Per iniziare, consideriamo un esempio: *dobbiamo chiedere all'utente il suo nome*. Come abbiamo visto nelle parti precedenti, per scrivere a video abbiamo usato i metodi `print()` e `println()` dell'oggetto `System.out`. In Java esistono anche altri due oggetti: `System.err` e `System.in`. In particolare, l'oggetto `System.in` gestisce il flusso di dati proveniente dai dispositivi di input. L'oggetto `System.in`, però, non supporta dei metodi diretti per leggere da tastiera. Utilizzeremo, quindi, `System.in` con altri oggetti che supportano tali metodi. Cominciamo dunque ad utilizzare gli oggetti `InputStreamReader` e `BufferedReader` del package java.io (che andrà importato). Scriveremo quindi:

```
InputStreamReader streamin = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(streamin);
```

Abbiamo usato, in questo esempio, anche l'operatore **new**. Vedremo più dettagliatamente questo operatore nelle prossime parti. Ci occorre, per il momento, sapere solamente che questo operatore crea un'istanza di una classe.

Il metodo che ci occorre per leggere i dati è `readLine()`, della classe `BufferedReader`. Dato che, nell'esempio, chiediamo all'utente di inserire il suo nome, dobbiamo utilizzare una variabile stringa per conservare in memoria il risultato della lettura. Scriveremo quindi:

```
String nome;
nome = br.readLine();
```

Quando l'utente premerà INVIO l'input verrà letto. Segue il codice del programma appena visto. Una precisazione: il seguente programma non funziona. Questo perché occorre gestire le eccezioni.

```
import java.io.*;

class LeggiTastiera
{
      public static void main(String[] args)
      {
            String nome;
            InputStreamReader streamin = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(streamin);

            System.out.print("Inserisci il tuo nome: ");
            nome = br.readLine();
            System.out.println("Il tuo nome: " + nome);
      }
```

```
}
```

Vediamo adesso come gestire le eccezioni.

2. La gestione delle eccezioni

Per prima cosa, chiariamo che cos'è un'eccezione. Un'eccezione è un errore che si verifica all'esecuzione di un programma. La gestione delle eccezioni semplifica quindi la gestione degli errori. Tutte le eccezioni in Java sono rappresentate da classi. Tutte le classi di eccezione derivano da una classe: la classe **Throwable**. Gli errori causati dall'attività del programma sono rappresentati dalle sottoclassi di **Exception**, una sottoclasse diretta di Throwable. Per gestire le eccezioni si utilizzano le seguenti cinque parole chiave: **try, catch, finally, throws** e **throw**. L'utilizzo è molto semplice: le dichiarazioni del programma che si desiderano controllare sono contenute nel blocco try. Se all'interno di tale blocco si verifica un'eccezione, essa viene *lanciata*. Il codice può catturare l'eccezione usando catch. È inoltre possibile lanciare manualmente un'eccezione, usando throw. Un codice che deve essere necessariamente eseguito all'uscita di un blocco try viene inserito in un blocco finally. In alcuni casi, un'eccezione lanciata fuori da un metodo deve essere specificata come tale dalla clausola throws.
Vediamo direttamente come gestire tali eccezioni. Abbiamo detto che l'esempio della lettura dell'input riportato sopra non funziona, a causa di un'eccezione che non viene gestita. Proviamo a gestirla. Se proviamo a compilare il codice scritto sopra, il compilatore ci darà il seguente messaggio di errore:

```
...\LeggiTastiera.java:12: unreported exception java.io.IOException; must be
caught or declared to be thrown
            nome = br.readLine();
                               ^
1 error
```

Il compilatore ci avverte che non abbiamo gestito un'eccezione, precisamente IOException. Proviamo quindi a modificare il codice, inserendo try e catch:

```
import java.io.*;

class LeggiTastiera
{
      public static void main(String[] args)
      {
            String nome;
            InputStreamReader streamin = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(streamin);

            System.out.print("Inserisci il tuo nome: ");

      try
      {
            nome = br.readLine();
            System.out.println("Il tuo nome: " + nome);
      }
      catch(IOException e)
            {
```

```
            System.out.println("Catturata eccezione I/O");
            }
      }
}
```

Se proviamo a ricompilare, ci accorgeremo che il codice adesso è corretto, ed il compilatore non segnalerà più alcun errore. Java, inoltre, permette di associare più dichiarazioni catch ad un blocco try, a condizione che ogni catch catturi un tipo diverso di eccezione. Ad esempio:

```
try
{
// codice da controllare
}
catch(Ecc1)
{
// gestione della prima eccezione
}
catch(Ecc2)
{
// gestione della seconda eccezione
}
```

Abbiamo visto come catturare un'eccezione. Adesso vedremo come lanciare un'eccezione. Come abbiamo già detto, per lanciare un'eccezione si utilizza la dichiarazione throw. La sua forma generale è:

```
throw <obj_eccezione>;
```

Throw non lancia dei tipi, ma degli oggetti. <obj_eccezione> deve essere un oggetto di una classe di eccezione derivata da Throwable. Proviamo a lanciare manualmente l'eccezione IOException:

```
import java.io.*;

class LanciaEccezione
{
      public static void main(String[] args)
      {
            try
            {
            System.out.println("Lancia l'eccezione...");
            throw new IOException();
            }
            catch(IOException e)
            {
            System.out.println("Eccezione catturata.");
            }
            System.out.println("Fuori dal blocco try/catch");
      }
}
```

Ancora una volta abbiamo utilizzato l'operatore new nella dichiarazione throw. Rammentiamo ancora che throw lancia un oggetto, quindi occorre creare un oggetto da lanciare. Non basta quindi lanciare un tipo.

Vediamo in dettaglio anche la classe Throwable. Per ereditarietà, dato che tutte le eccezioni sono sottoclassi di Throwable, tutte supportano i metodi definiti da Throwable. Vediamo rapidamente i metodi della classe Throwable:

`String getLocalizedMessage()`	Restituisce una descrizione dell'eccezione
`String getMessage()`	Restituisce una descrizione dell'eccezione
`void printStackTrace()`	Visualizza la traccia della pila
`void printStackTrace(PrintStream s)`	Inviano la traccia della pila al flusso specificato
`void printStackTrace(PrintWriter s)`	
`String toString()`	Restituisce una stringa contenente una descrizione dell'eccezione

Proviamo utilizzando in particolare il metodo printStackTrace():

```
import java.io.*;

class LanciaEccezione
{
      public static void main(String[] args)
      {
            try
            {
            throw new IOException();
            }
            catch(IOException e)
            {
            System.out.println("Messaggio standard: " + e);
            System.out.println("Traccia pila: ");
            e.printStackTrace();
            }
    }
}
```

Il risultato sarà:

```
Messaggio standard: java.io.IOException
Traccia pila:
java.io.IOException
        at LanciaEccezione.main(LanciaEccezione.java:9)
```

Non spaventiamoci se non conosciamo cosa sono i PrintStream, i PrintWriter e gli Stack, tutto verrà chiarito al momento giusto.

Vediamo adesso come utilizzare finally. Per specificare un blocco di codice da eseguire quando si esce da un blocco try/catch, occorre includere un blocco finally alla fine di una sequenza try/catch. Il codice della clausola opzionale finally viene eseguito qualunque cosa accada, ovvero con o senza eccezioni. Ecco un esempio di finally:

```
import java.io.*;

class LanciaEccezione
{
      public static void main(String[] args)
      {
            try
            {
            throw new IOException();
            }
```

```
            catch(IOException e)
            {
            System.out.println("Messaggio del blocco catch");
            }
            finally
            {
            System.out.println("Messaggio del blocco finally");
            }
    }
}
```

L'output prodotto è il seguente:

```
Messaggio del blocco catch
Messaggio del blocco finally
```

Quindi, come mostra l'output, in qualunque modo termini il blocco try, il blocco finally viene eseguito.

L'ultima parole chiave che prenderemo in esame sarà throws. Usiamo la clausola throws se un metodo genera un'eccezione che non gestisce. Vediamo subito come usare throws. Nell'esempio della lettura dell'input da testiera abbiamo usato try/catch per gestire le eccezioni. Possiamo riscrivere il codice del primo esempio nel seguente modo, utilizzando throws:

```
import java.io.*;

class LeggiTastiera
{
      public static void main(String[] args) throws IOException
      {
            String nome;
            InputStreamReader streamin = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(streamin);

            System.out.print("Inserisci il tuo nome: ");
            nome = br.readLine();
            System.out.println("Il tuo nome: " + nome);
      }
}
```

In seguito apprenderemo anche come creare delle proprie classi di eccezioni. Di seguito è riportata la lista delle eccezioni incorporate in Java.

Eccezioni

`ArithmeticException`	Errore aritmetico (ad esempio la divisione per 0)
`ArrayIndexOutOfBoundsException`	Indice di array esterno ai confini
`ArrayStoreException`	Assegnazione a un elemento di array di un tipo incompatibile
`ClassCastException`	Cast non valido

`IllegalArgumentException`	Argomento non valido usato per invocare un metodo
`IllegalMonitorStateException`	Operazione di controllo non valida
`IllegalStateException`	L'ambiente o l'applicazione è in uno stato non corretto
`IllegalThreadStateException`	L'operazione richiesta non è compatibile con lo stato corrente del Thread
`IndexOutOfBoundsException`	Un tipo di indice è esterno ai confini
`NegativeArraySizeException`	Array creato con dimensioni negative
`NullPointerException`	Uso non valido di un riferimento nullo
`NumberFormatException`	Conversione non valida di una stringa in formato numerico
`SecurityException`	Tentativo di violare la sicurezza
`StringIndexOutOfBoundsException`	Tentativo di indicizzare fuori dai confini di una stringa
`UnsupportedOperationException`	Si è incontrata un'operazione non supportata

`ClassNotFoundException`	Classe non trovata
`CloneNotSupportedException`	Tentativo di clonare un oggetto che non implementa l'interfaccia Cloneable
`IllegalAccessException`	Accesso ad una classe negato
`InstantiationException`	Tentativo di creare un oggetto di una classe astratta o un'interfaccia
`InterruptedException`	Un Thread è stato interrotto da un altro Thread
`NoSuchFieldException`	Un campo richiesto non esiste
`NoSuchMethodException`	Un metodo richiesto non esiste

Vedremo ora come utilizzare le strutture di controllo.

3. Le strutture di controllo

Istruzioni condizionali: le istruzioni condizionali permettono di operare delle scelte tra due o più alternative. Per far ciò vengono utilizzate le espressioni booleane. Per esempio:

```
(numero >= 10 && numero <= 20)
```

L'espressione vista sopra restituisce il valore booleano true se il numero è compreso fra 10 e 20, altrimenti restituirà il valore booleano false. Un errore molto frequente è quello di confondere l'operatore di assegnamento (un solo uguale) con quello di confronto (due uguali).

Il costrutto if…else: L'istruzione condizionale più importante è il costrutto **if…else**. La sua sintassi è:

```
if(<condizione>)
     <istruzioni>
[else
     <istruzioni_altrimenti>]
```

<condizione> può assumere solo i valori true o false
<istruzioni> è un blocco di istruzioni che viene eseguito quando <condizione> risulta vera
<istruzioni_altrimenti> è un blocco di istruzioni che è eseguito quando <condizione> risulta falsa

In alcuni casi è possibile utilizzare **if** senza **else**. Consideriamo il seguente esempio: *dobbiamo controllare la variabile 'ang'. Essa viene automaticamente incrementata. Dobbiamo impedire che il suo valore superi 360*. Come possiamo fare? La soluzione è semplice.

```
if(ang == 360)
  ang = 0;
```

Consideriamo un esempio completo: *dati in input due numeri, trovare il massimo*.

```
import java.io.*;

class Massimo
{
      public static void main(String[] args)
      {
      InputStreamReader streamin = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(streamin);
      int n1, n2;

            try
          {
            System.out.println("Inserisci il primo numero:");
            n1 = Integer.parseInt(br.readLine());
            System.out.println("Inserisci il secondo numero");
            n2 = Integer.parseInt(br.readLine());

               if(n1 > n2)
                  System.out.println("n1 > n2 (" + n1 + " > " + n2 + ")");
               else
                  System.out.println("n1 < n2 (" + n1 + " < " + n2 + ")");
          }
            catch(Exception e)
          {
            System.out.println("ECCEZIONE!");
            System.out.println(e);
            System.out.println("Traccia pila:");
            e.printStackTrace();
          }
```

```
      }
}
```

Nel caso occorra effettuare una serie di controlli, si può utilizzare **else** in questo modo:

```
if(<condizione>)
   <istruzioni>
[else if(<condizione>)
   <istruzioni>]
else
   <istruzioni>
```

Il funzionamento è il seguente. Viene valutata l'espressione iniziale: se questa è vera viene eseguito il relativo blocco di istruzioni, altrimenti viene valutata la condizione posta accanto ad ogni else e, se l'espressione risulta vera, viene eseguito il relativo blocco di istruzioni, altrimenti viene eseguito il blocco di istruzioni dell'else finale. Per comprendere meglio questo concetto, riprendiamo l'esempio visto sopra, modificandolo un po'. Il problema è il seguente: *dato in input un numero, verificarne il valore. Se il valore del numero è pari a 5, oppure a 10, oppure a 20, segnalarlo all'utente. In qualsiasi altro caso, segnalare all'utente che ha inserito un numero diverso da 5, 10 e 20.*

```
import java.io.*;

class Confronta
{
      public static void main(String[] args)
      {
      InputStreamReader streamin = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(streamin);
      int n;

            try
           {
            System.out.println("Inserisci il numero:");
            n = Integer.parseInt(br.readLine());

               if(n == 5)
                  System.out.println("Il valore del numero è pari a 5");
               else if(n == 10)
                  System.out.println("Il valore del numero è pari a 10");
               else if(n == 20)
                  System.out.println("Il valore del numero è pari a 20");
               else
                  System.out.println("Il valore del numero è diverso da 5, 10 e
20 (numero inserito = " + n + ")");
          }
            catch(Exception e)
          {
            System.out.println("ECCEZIONE!");
            System.out.println(e);
            System.out.println("Traccia pila:");
            e.printStackTrace();
          }
      }
}
```

Il costrutto switch: la forma dell'istruzione switch è:

```
switch(<espressione>)
{
      case <etichetta>:
      {
            <istruzioni>
            [break;]
      }
      [case <etichetta>:
      {
            <istruzioni>
            [break;]
      }]
      [default:
            <istruzioni>]
}
```

dove il valore <espressione> viene messo a confronto con i diversi valori di <etichetta>. Quando viene trovata corrispondenza, viene eseguito il relativo blocco di istruzioni. Il blocco di istruzione di default viene eseguito solamente se non viene trovata corrispondenza con tutti i valori precedentemente dichiarati nei case. Notiamo che abbiamo inserito anche l'istruzione break. Questa istruzione serve per interrompere il ciclo di switch; se questa istruzione mancasse il programma continuerebbe a confrontare l'espressione con il successivo valore di <etichetta>. L'istruzione break è opzionale. Vediamo un esempio: *dato in input un numero, verificarne il valore. Se il valore del numero è pari a 5, oppure a 10, oppure a 20, segnalarlo all'utente. In qualsiasi altro caso, segnalare all'utente che ha inserito un numero diverso da 5, 10 e 20.* Il problema è lo stesso visto sopra. Stavolta, però, lo risolveremo senza utilizzare il costrutto if…else ma con il costrutto switch.

```
import java.io.*;

class Confronta
{
      public static void main(String[] args)
      {
      InputStreamReader streamin = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(streamin);
      int n;

            try
          {
            System.out.println("Inserisci il numero:");
            n = Integer.parseInt(br.readLine());

                switch(n)
                {
                    case 5:
                    {
                        System.out.println("Il valore del numero è pari a 5");
                        break;
                    }
                    case 10:
                    {
```

```
                        System.out.println("Il valore del numero è pari a 10");
                        break;
                    }
                    case 20:
                    {
                        System.out.println("Il valore del numero è pari a 20");
                        break;
                    }
                    default:
                        System.out.println("Il valore del numero è diverso da 5,
10 e 20. (numero inserito = " + n + ")");
                }
            }
              catch(Exception e)
            {
              System.out.println("ECCEZIONE!");
              System.out.println(e);
              System.out.println("Traccia pila:");
              e.printStackTrace();
            }
      }
}
```

Le istruzioni iterative: le istruzioni iterative permettono di ripetere l'esecuzione di un blocco di istruzioni. Le istruzioni iterative sono: **while**, **do**…**while**, **for**.

Costrutto while: la sintassi del costrutto while è la seguente:

```
while(<condizione>)
{
      <istruzioni>
}
```

viene verificata <condizione>: se risulta vera vengono eseguite <istruzioni>. Quando viene eseguita l'ultima istruzione del blocco, viene verificata nuovamente la condizione. Se risulta vera viene rieseguito il blocco. Il ciclo termina quando <condizione> risulta falsa. Vediamo un esempio: *scrivere i primi 10 numeri positivi maggiori di zero*.

```
class DieciPositivi
{
      public static void main(String[] args)
      {
         int i = 1;

         while(i <= 10)
            {
                  System.out.println(i + "\n");
                  i++;
```

```
            }
      }
}
```

Costrutto do…while: il costrutto do…while esegue il blocco di istruzioni almeno una volta. La sua sintassi è:

```
do
{
      <istruzioni>
}
while(<condizione>)
```

Il blocco delle istruzioni viene eseguito finché <condizione> risulta vera. Vediamo un altro esempio: *continuare a leggere da input un numero finché l'utente non ne inserisce uno positivo e diverso da 0*. Per risolvere questo problema, utilizziamo il costrutto do…while:

```
import java.io.*;

class DoWhile
{
      public static void main(String[] args)
      {
         InputStreamReader streamin = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(streamin);
         int n = 0;

             try
            {
               System.out.println("Inserisci un numero positivo diverso da 0");
                  do
                  {
                     n = Integer.parseInt(br.readLine());
                  }
                  while(n <= 0);

                  System.out.println("Numero inserito: " + n);
            }
            catch(Exception e)
            {
            System.out.println("ECCEZIONE!");
            System.out.println(e);
            System.out.println("Traccia pila:");
            e.printStackTrace();
            }
      }
}
```

Il programma continuerà a chiedere di inserire il numero finché non ne verrà inserito uno positivo.

Costrutto for: un altro costrutto iterativo molto utilizzati è il costrutto for. La sua sintassi è la seguente:

```
for(<espressione> ; <condizione> ; <iterazione>)
{
```

```
      <istruzioni>
}
```

viene eseguita <espressione> e si verifica <condizione>. Se risulta vera viene eseguito il blocco delle istruzioni. A ogni ciclo avviene <iterazione> e viene verificata nuovamente <condizione>. L'esecuzione termina quando <condizione> diventa falsa. Per comprendere meglio il costrutto for, consideriamo un esempio: *scrivere i primi 10 numeri positivi*. Il codice è il seguente:

```
class DieciPositivi
{
      public static void main(String[] args)
      {
            for(int i = 1; i <= 10; i++)
                  {
              System.out.println(i + "\n");
                  }
      }
}
```

Istruzioni break e continue: queste istruzioni vengono utilizzate per ottimizzare i cicli for e while e il costrutto if…else. L'istruzione break permette di uscire immediatamente da un ciclo, ignorando qualunque codice rimanente nel corpo del ciclo e la verifica condizionale del ciclo. L'istruzione continue forza l'iterazione successiva del ciclo ad avere luogo, ignorando il codice tra esso e l'espressione condizionale che controlla il ciclo. Vediamo con degli esempi come utilizzare queste istruzioni.

Bloccare un ciclo infinito dopo n iterazioni definite dall'utente. Per creare un ciclo infinito è sufficiente usare for lasciando vuota l'espressione condizionale. Per terminarlo useremo l'istruzione break.

```
import java.io.*;

class IstruzioneBreak
{
      public static void main(String[] args) throws Exception
      {
         InputStreamReader streamin = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(streamin);
         int i, n;

         i = 0;

         System.out.println("Specificare il numero di iterazioni da eseguire:");

         n = Integer.parseInt(br.readLine());

            for(;;)
               {
              System.out.println("Iterazione...");
              i++;
                 if(i == n)
                    break;
               }

         System.out.println("Iterazione avvenuta per " + i + " volte");
      }
}
```

Passiamo ora ad un altro problema. *Visualizzare solo i numeri pari da 0 a 100.* Per risolvere questo tipo di problema useremo l'istruzione continue.

```
class IstruzioneContinue
{
      public static void main(String[] args)
      {
         int i;

         for(i = 0; i <= 100; i++)
            {
               if((i % 2) != 0)
                  continue;

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

Nell'esempio vengono visualizzati solo i numeri pari. Infatti un numero dispari farà iterare prima il ciclo, ignorando la chiamata a println().

4. Gli Array

Che cosa sono gli array? La definizione di array è la seguente: un array è un insieme ordinato di variabili dello stesso tipo, a cui si fa riferimento tramite un nome comune e ad un indice. Il tipo di un elemento può essere un tipo primitivo o un oggetto: l'importante è che tutti gli elementi siano dello stesso tipo. Otterremo quindi array di stringhe, di caratteri, di numeri interi ecc., ma non avremo un *array denso*, cioè un array contenente tipi diversi di dati.
Per dichiarare un array si utilizzano le seguenti sintassi:

```
<tipo> <nome>[];
```

oppure

```
<tipo>[] <nome>;
```

Se l'array è già stato dichiarato, per dichiarare un nuovo oggetto array si utilizza la seguente sintassi:

```
<nome> = new <tipo>[<num_elementi>];
```

Se l'array non è già stato dichiarato, per dichiarare e creare un nuovo oggetto array si utilizza la sintassi seguente:

```
<tipo>[] <nome> = new <tipo>[<num_elementi>];
```

oppure

```
<tipo> <nome>[] = new <tipo>[<num_elementi>];
```

Per accedere ad un elemento dell'array, si utilizza l'indice che indica la posizione dell'elemento all'interno dell'array.

```
<nome>[<indice_elemento>];
```

Una precisazione: <indice_elemento> inizia da 0 e non da 1. Il primo elemento è quindi contrassegnato da indice 0, il secondo da 1 ecc.

Poiché gli array vengono implementati come oggetti, ogni array ha associata una variabile di istanza **length**, con il numero di elementi che può contenere l'array. Per utilizzare questa variabile utilizziamo la seguente sintassi:

```
<nome>.length;
```

Vediamo alcuni esempi di array:

```
int numbers[] = new int[5];
```

Questa dichiarazione funziona come una dichiarazione di oggetto. La variabile numbers contiene un riferimento alla memoria associata dall'operatore new. Questa memoria è capace di contenere 5 elementi di tipo int. Come per gli oggetti è possibile dividere in due la precedente dichiarazione. Per esempio:

```
int numbers[];
numbers = new int[5];
```

In tal caso, quando è stato creato, numbers è **null**, perché non ha riferimenti ad un oggetto fisico. Solo dopo che la seconda dichiarazione viene eseguita, numbers si concatena ad un array.
Riferendoci all'array numbers, per inserire gli elementi scriveremo:

```
numbers[0] = 150;
numbers[1] = 0;
numbers[2] = 112;
numbers[3] = 1;
numbers[4] = 47;
```

Un metodo alternativo è il seguente:

```
int numbers[] = {150, 0, 112, 1, 47};
```

In questo secondo modo viene creato un array di dimensione pari al numero di elementi inseriti. Tutti gli elementi devono essere del tipo dichiarato. Proviamo ad accedere ad uno degli elementi.

```
int n = numbers[2];
```

La variabile n avrà valore 112. Proviamo ad utilizzare il membro length per verificare il numero di elementi che l'array numbers contiene:

```
int l = numbers.length;
```

Array multidimensionali: la forma più semplice dell'array multidimensionale è l'array a due dimensioni. In Java un array multidimensionale è un array di array. Vediamo come dichiarare un array a due dimensioni.
Per dichiarare un array di tipo intero a due dimensioni di dimensioni 3, 5 si scrive:

```
int m[][] = new int[3][5];
```

Java ammette inoltre array con più di due dimensioni. La seguente dichiarazione, ad esempio, crea un array multidimensionale di tipo intero di di 5 x 10 x 10:

```
int m[][][] = new int[5][10][10];
```

Per inizializzare un array multidimensionale è possibile racchiudere l'elenco inzializzatore di ogni dimensione tra parentesi graffe. L'esempio seguente inzializza un array con i numeri da 1 a 5 ed i loro rispettivi doppi.

```
int doubles[][] = {
                  {1, 2},
                  {2, 4},
                  {3, 6},
                  {4, 8},
                  {5, 10}
                  };
```

Riprenderemo gli array molto dettagliatamente più avanti nel corso, quando considereremo gli algoritmi di ordinamento dei vettori e le varie operazioni che possono essere effettuate sui vettori. Considereremo ora molto superficialmente le stringhe, che verranno riprese molto dettagliatamente nella quinta parte.

5. Stringhe

Come già detto, rivedremo molto dettagliatamente le stringhe nella quinta parte, utilizzando anche le classi StringBuffer e StringTokenizer. Per adesso ci limitiamo a descrivere superficialmente alcuni metodi della classe String.

Costruttori:

```
String();
String(String s);
```

Questi metodi costruttori creano rispettivamente una stringa vuota e una stringa copia della stringa che viene passata come parametro.

Metodi:

`boolean equals(String s)`	Restituisce true se la stringa che lo invoca contiene la stessa sequenza di caratteri di *s*.
`int length()`	Ottiene la lunghezza di una stringa.
`char charAt(int indice)`	Ottiene il carattere all'indice specificato da *indice*
`int compareTo(String s)`	Restituisce un numero: • Minore di zero se la stringa che lo invoca è minore di *s* • Maggiore a zero se la stringa che lo invoca è maggiore di *s* • Zero se le stringhe sono uguali
`int indexOf(String s)`	Cerca la stringa che invoca per la sottostringa specificata da *s*. Restituisce l'indice della prima corrispondenza o –1 se non viene trovata corrispondenza.
`int lastIndexOf(String s)`	Cerca la stringa per la sottostringa specificata da *s*. Restituisce l'indice dell'ultima corrispondenza o –1 se non viene trovata corrispondenza.

L'esempio che segue utilizza alcuni dei metodi descritti sopra.

```
import java.io.*;

class StringDemo
{
      public static void main(String[] args)
      {
         InputStreamReader streamin = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(streamin);
         String s1, s2;
         int res;

             try
             {
                  System.out.println("Inserisci la prima stringa:");
                  s1 = br.readLine();
                  System.out.println("Inserisci la seconda stringa:");
                  s2 = br.readLine();


          System.out.println("Lunghezza della prima stringa: " + s1.length());
          System.out.println("Lunghezza della seconda stringa: " + s2.length());

            if(s1.equals(s2))
                  System.out.println("Le stringhe inserite sono uguali.");
            else
                   System.out.println("Le stringhe inserite sono diverse.");

         System.out.println("Utilizzando compareTo():");

            res = s1.compareTo(s2);

            if(res == 0)
```

```
                    System.out.println("Le due stringhe sono uguali");
              else
              if(res < 0)
                    System.out.println("Prima stringa minore della seconda");
              else
                    System.out.println("Prima stringa maggiore della seconda");
               }
               catch(Exception e)
               {
                    System.out.println("ECCEZIONE!");
                    System.out.println(e);
                    e.printStackTrace();
               }
        }
}
```

Nelle parti successive analizzeremo nuovamente le stringhe, ma molto approfonditamente. Si conclude così questa terza parte. Nella prossima analizzeremo la programmazione ad oggetti.