Comandi

- Comando nullo `;`
- Comando condizionale `if`

```
class Test{
    public static void main(String[] voto){
      try{
            if ( voto.length >0 && Integer.parseInt(voto[0]) >= 18)
                    System.out.println("Promosso");
            else{
                    System.out.println("Bocciato");
                    System.out.println("...peccato!");
            }
       }catch (Exception e){
                    System.out.println("errore di input");
       }
    }
}
```


Comando switch

```
class Toomany {
    static void howMany(int k) {
        switch (k) {
           case 1: System.out.print("one ");
           case 2: System.out.print("two ");
           case 3: System.out.println("many");
        }
     }
     public static void main(String[] args) {
         howMany(3);
         howMany(2);
         howMany(1);
     }
}
```

Produce:

```
many
two many
one two many
```

- L'espressione `k` deve essere di tipo `char,byte,short,int` altrimenti vi è un errore statico

Comando switch

```
class Twomany {
        static void howMany(int k) {
                switch (k) {
                case 1: System.out.println("one"); break;
                case 2: System.out.println("two"); break;
                case 3: System.out.println("many");break;
                default: System.out.println("too many");
                }
        }
        public static void main(String[] args) {
                howMany(1);
                howMany(2);
                howMany(3);
                howMany(300);
        }
}
```

Produce:
```
one
two
many
too many
```


Comandi while...do & do...while

```
class ContaCaratteri {
    public static void main(String[] args) throws java.io.IOException {
            int num = 0;
            while (System.in.read() != -1){
                num++;
            }
            System.out.println("Input di" + num + " caratteri.");
    }
}
```

```
class ScriviFinoAlCento{
    public static void main(String[] args){
        int i = 0;
        do {
             System.out.println(++i);
        } while (i < 100);
    }
}
```

Comando for

```
class ScriviFinoAlCentoA{
    public static void main(String[] args){
        for (int i=1; i<=100; i++){
             System.out.println(i);
        }
    }
}
```

```
class ScriviFinoAlCentoB{
    public static void main(String[] args){
        for (int i=1; i<=100; ){
             System.out.println(i++);
        }
    }
}
```

```
class ScriviFinoAlCentoC{
    public static void main(String[] args){
        for (int i=1; ; ){
             System.out.println(i++);
             if (i>100) break;
        }
    }
}
```


Comandi break & continue

```
import java.io.*;

class tabella{

  int[][] tab;

  public tabella(int[][] nuova){
    this.tab = nuova;
  }

  public void inserisci(){
    int n;
    leggi:for (int i=0; i<tab.length; i++)
       for (int j=0; j<tab[0].length; j++){
          n = (i+1)*(j+1);
          if (n<5) continue;
             else if (n>10) break leggi;
             else tab[i][j] = n;
       }
  }

  public void stampa(){
    for (int i=0; i<tab.length; i++){
      for (int j=0; j<tab[0].length; j++)
        System.out.print("" + tab[i][j] + "\t");
      System.out.print("\n");
    }
 }
```

```
class Test{
   public static void main(String args[]) {
      int[][] t = new int[5][5];
      tabella p = new tabella(t);
      p.inserisci();
      p.stampa();
   }
}
```

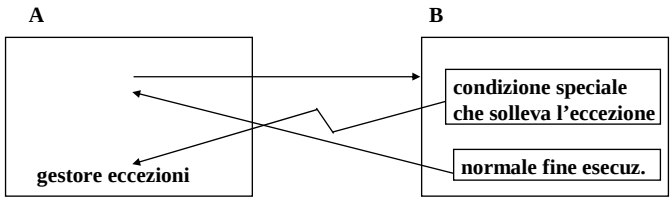
output:

```
0  0  0  0  5
0  0  6  8  10
0  6  9  0  0
0  0  0  0  0
0  0  0  0  0
```

break esce dal comando switch/ while/do/for più interno, a meno che non ci sia un'etichetta

continue passa subito all'iterazione successiva nei comandi while/do/for

Eccezioni



- try
  - contiene un blocco di comandi all'interno dei quali può essere sollevata un'eccezione
- catch
  - cattura le eccezioni specificate
- throw
  - solleva un'eccezione. Nella dichiarazione di un metodo, permette ad un'eccezione sollevata all'interno di essere gestita dal metodo chiamante

Sintassi

```
try{
     // codice che può sollevare (throw) una certa eccezione
}
catch (EccezioneDelMioTipo e){
     // codice da eseguire se l'eccezione sollevata è di tipo
     // EccezioneDelMioTipo
}
catch (Exception e){
     // codice da eseguire se viene sollevata una eccezione generica
}
```

```
try{
     // codice che può sollevare un'eccezione
}
finally{
     // codice che viene coumunque eseguito indipendentemente dal fatto
     // che l'eccezione venga gestita o meno (quindi anche nel caso in cui
     // l'eccezione non gestita venga passata al metodo chiamante)
}
```

Esempio

```
public static void main(String args[]){
     int i = 0;
     String saluti[] = {
          “Ciao”,
          “No! Volevo dire”,
          “CIAO”,
     };
     while (i<4){
          try{ System.out.println(saluti[i]);}
          catch (ArrayIndexOutOfBoundsException e){
                    System.out.println(“Risistemo l’indice”);
                    i=-1;
          }
          finally{System.out.println(“Questo lo stampo comunque!”);}
          i++;
     }
}
```

L’esecuzione:

```
Ciao
Questo lo stampo comunque!
No! Volevo dire
Questo lo stampo comunque!
CIAO
Questo lo stampo comunque!
Risistemo l’indice
Questo lo stampo comunque!
……
```


Categorie di eccezioni

- La classe `java.lang.Trowable` agisce come superclasse di tutti gli oggetti che possono essere sollevati e catturati usando il meccanismo delle eccezioni.
- Nella classe `Trowable` sono definiti i metodi per recuperare i messaggi di errore associati alle eccezioni e per stampare la traccia dello stack per capire dove l’eccezione è stata sollevata
- Ci sono due sottoclassi fondamentali di `Trowable`:
  - gli errori indicano problemi per i quali è difficile o impossibile il recupero (ad esempio `OutOfMemoryError`).
  - Le eccezioni run-time o di input-output indicano invece un problema di progettazione o implementazione (es: `ArrayIndexOutOfBoundsException`, `NegativeArraySizeException`, `ArithmeticException`, `NullPointerException`, `EOFException`, `FileNotFoundException`)
- Nuove eccezioni possono essere definite estendendo opportunamente la classe `Exception`.

Gestire o dichiarare eccezioni

- Per incoraggiare a scrivere codice robusto, Java richiede che se un'eccezione può essere sollevata durante l'esecuzione dei comandi all'interno della chiamata di un metodo, allora il metodo stesso deve determinare come tale eccezione deve essere gestita.
- Ci sono due modi per soddisfare questo requisito:
  - Avere nel corpo del metodo un comando blocco try-catch
  - Indicare nella firma del metodo che il metodo non gestirà l'eccezione, e che l'eccezione eventualmente sollevata nel corpo del metodo ne interromperà l'esecuzione e sarà immediatamente "passata" al metodo chiamante.
    Per fare questo bisogna aggiungere nella dichiarazione del metodo un throws, ad es.

    ```
    public void senzaProblemi() throws IOException
    ```


Campo di visibilità

- di un tipo `T` importato con `import T`
  - tutte le unità di compilazione in cui compare `import T`
- di un tipo introdotto con dichiarazione di classe o interfaccia
  - dichiarazioni di tutte le classi e interfacce del package
- di una variabile o metodo dichiarato o ereditato in un tipo classe o interfaccia
  - tutta la dichiarazione della classe o interfaccia
- di un parametro di un metodo o di un costruttore
  - tutto il corpo del metodo/costruttore

Esempi di campo di visibilità

```
class Test {
        int i = j;
        int j = 1;
}
```

NO

(No “forward references”)

```
class Test {
        Test() { k = 2; }
        int j = 1;
        int i = j;
        int k;
}
```

OK

```
package punti;

class Punto {
        int x, y;
        ListaPunti lista;
        Punto next;
}

class ListaPunti {
        Punto primo;
}
```

OK

(Anche in file diversi!)

Hiding

- Una variabile può essere nascosta quando ne viene dichiarata un’altra con lo stesso nome nel suo campo di visibilità.Vi si può comunque accedere utilizzando il suo nome esteso

```
class Test {
        static int x = 1;
        public static void main(String[] args) {
                int x = 0;
                System.out.print("x=" + x);
                System.out.println(", Test.x=" + Test.x);
        }
}
```

produce l’output

```
x=0, Test.x=1
```

Stesso nome, diversi contesti

```
class Point { int x, y; }

class Test {

        static Point Point(int x, int y) {
                Point p = new Point();
                p.x = x; p.y = y;
                return p;
        }

        public static void main(String[] args) {
                int Point;
                Point[] pa = new Point[2];
                for (Point = 0; Point < 2; Point++) {
                        pa[Point] = new Point();
                        pa[Point].x = Point;
                        pa[Point].y = Point;
                }
                System.out.println(pa[0].x + "," + pa[0].y);
                System.out.println(pa[1].x + "," + pa[1].y);
                Point p = Point(3, 4);
                System.out.println(p.x + "," + p.y);
        }
}
```

complila senza errori e produce l'output:

```
0,0
1,1

3,4
```


Packages

- La dichiarazione di package deve essere specificata all'inizio del file sorgente (può essere preceduta solo da spazi o commenti)
- E' consentita una sola dichiarazione di package in un codice sorgente

```
// classe Impiegato dell'ufficio Acquisti della ditta SPA

package spa.acquisti;
public class Impiegato{
    . . .
}
```

- I nomi dei packages sono ordinati gerarchicamente e separati da punti

`import`

- Il comando `import` dice al compilatore dove si trovano le classi da usare
- Deve precedere tutte le dichiarazioni delle classi

```
import spa.acquisti.*;
public class Manager extends Impiegato {
   String ruolo;
   Impiegato[] dipendenti;
}
```

- Permette di inserire nel dominio dei nomi altri packages. Il package corrente è sempre parte del dominio di nomi.

Modificatori per classi

- **`public`**
  - la classe è accessibile da qualsiasi altra classe, anche all'esterno del package
- **`(default)`**
  - la classe è accessibile solo all'interno del package

- **`final`**
  - la classe non può essere estesa (non può avere sottoclassi)
- **`abstract`**
  - la classe deve avere almeno un metodo di tipo abstract, ossia dichiarato ma non implementato;
  - è una classe "incompleta" e non può essere istanziata

Modificatori per variabili e metodi

- **`static`**
  - una variabile/metodo `static` è condiviso da tutte le istanze (oggetti) della classe: l'istanziazione di una classe non produce allocazione di memoria per questa variabile/metodo, ma solo un link
  - un metodo `static` non può essere "overridden" in uno non static. Esso esiste prima che venga creato un qualsiasi oggetto della classe (ad esempio, il metodo `main`)
- **`final`**
  - è così che si definiscono le costanti: a una variabile `final` deve essere attribuito un valore all'atto della dichiarazione, e quel valore non può essere modificato.
  - Idem nel caso dei metodi: l'introduzione di `final` impedisce l'overriding a qualsiasi sottoclasse

Modificatori di accesso per variabili/metodi

- **`public`**
  - accessibile da qualsiasi classe che accede alla classe cui la variabile appartiene, anche fuori dal package in cui tale classe è dichiarata
- **`private`**
  - è accessibile solo all'interno della classe (massima restrizione)
- **`protected`**
  - è accessibile, fuori dal package, solo dalle sottoclassi della classe cui la variabile appartiene
- **`(default)`**
  - è accessibile dentro al package in cui la classe cui la variabile appartiene è dichiarata

Criteri di accessibilità

Modificatore	Stessa classe	Stesso package	Sottoclassi	Universo
`public`	***Si***	***Si***	***Si***	***Si***
`protected`	***Si***	***Si***	***Si***	
`default`	***Si***	***Si***		
`private`	***Si***			

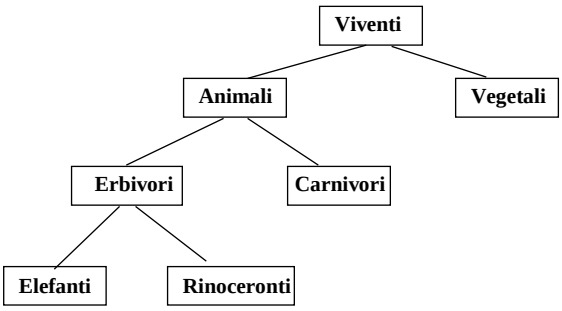
L'accesso `protected` è garantito anche alle sottoclassi che risiedono in un package diverso dalla classe che possiede il modificatore `protected`

`public - private`

```
import java.util.*;

class Impiegato{

   private String nome;
   private double paga;

   public Impiegato(String n,double s){
     nome = n;
     paga = s;
   }

   public void stampa(){
     System.out.println(nome + " " + paga);
   }

   public void aumento(double percentuale){
      paga *= 1 + percentuale / 100;
   }

}
```

```
public class Test{

   public static void main(String[] args){

      Impiegato[] staff = new Impiegato[3];

      staff[0] = new Impiegato("Ugo",1500);
      staff[1] = new Impiegato("Ada",2500);
      staff[2] = new Impiegato("Teo",3000);

      for (int i = 0; i < 3; i++)
         staff[i].aumento(5);
      for (int i = 0; i < 3; i++)
         staff[i].stampa();
   }
}
```

Ereditarietà



tutti gli animali hanno un nome, un peso, ecc
gli erbivori hanno una in più "quantità d'erba giornaliera"
gli elefanti hanno in più "lunghezza delle zanne"
i rinoceronti "lunghezza del corno"

Ereditarietà

- ogni oggetto di una sottoclasse può essere assegnato a una variabile che ha per tipo la sua superclasse
- ogni oggetto di una superclasse può essere assegnato a una variable che ha per tipo una sua sottoclasse con un casting appropriato
- in Java c'è solo *ereditarietà semplice* tra classi

```
Animali a,b;              a = dumbo                    OK
Rinoceronte pippo;        pippo = a                    NO!
Elefante dumbo;           pippo = dumbo                NO!
                          pippo = (Rinoceronte) b      OK!
```

- **Nota**: il casting puo' dare errore a run-time se il tipo di b non è un'istanza di Rinoceronte o di una sua sottoclasse

Ereditarietà

```
class Manager extends Impiegato{

  private nomeSegretaria;

  public Manager(String n, double s){
      super(n, s);
      nomeSegretaria = "";
   }

   public void aumento(double percentuale) {
      // incrementa la percentuale del 20%
      super.aumento(percentuale * 1.2);
   }

   public void setNomeSegretaria(String n){
      nomeSegretaria = n;
   }

   public String getNomeSegretaria(){
     return nomeSegretaria;
   }
}
```

```
public class Test{

   public static void main(String[] args){

      Impiegato[] staff = new Impiegato[3];

      staff[0] = new Impiegato("Ida",1500);
      staff[1] = new Impiegato("Mia",2500);
      staff[2] = new Manager("Leo",7000);

      for (int i = 0; i < 3; i++)
         staff[i].aumento(5);
      for (int i = 0; i < 3; i++)
         staff[i].stampa();
   }
}
```

Il costruttore della sottoclasse deve costruire anche i valori dei campi della superclasse, e (a meno che non accetti il costruttore di default della superclasse) deve usare esplicitamente `super`.

Ereditarietà con accesso di default

```
package points;

public class Point {
   int x, y;

   public void move(int dx, int dy) {
      x += dx; y += dy;
   }
}
```

```
package points;

public class Point3d extends Point {
   int z;
   public void move(int dx,
                    int dy, int dz) {
      x += dx; y += dy; z += dz;
   }
}
```

```
package A;
import points.Point3d;

class Point4d extends Point3d {
  int w;
  public void move(int dx, int dy,
                   int dz, int dw){
      x += dx; y += dy;
      z += dz; w += dw;
  }
}
```

NO!

```
package B;
import points.Point3d;

class Point4d extends Point3d {
  int w;
  public void move(int dx, int dy,
                   int dz, int dw) {
      super.move(dx, dy, dz);
      w += dw;
  }
}
```

OK

Ereditarietà (`public` - `protected`)

```
package points;

public class Point {
        public int x, y;
        protected int useCount = 0;
        static protected int totalUseCount = 0;

        public void move(int dx, int dy) {
                x += dx; y += dy; useCount++; totalUseCount++;
        }
}
```

```
package A;

class Test extends points.Point {
        public void moveBack(int dx, int dy) {
                x -= dx; y -= dy; useCount++; totalUseCount++;
        }
}
```

I campi x e y (public) e i campi useCount e totalUseCount (protected) sono ereditati in tutte le sottoclassi della classe Point.

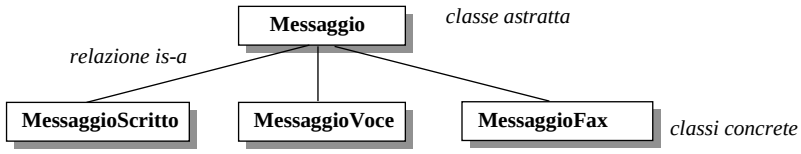
Classi astratte

- Una classe astratta è una classe che dichiara l'esistenza di un metodo ma non la sua implementazione
- Una classe può essere dichiarata astratta con il modificatore abstract

```
public abstract class Disegna{
    public abstract void disegnaPunto(int x, int y);
    public void disegnaLinea(int x1, int x2, int y1, int y2){
            // disegna usando disegnaPunto ripetutamente
    }
}
```

- Una classe astratta può contenere variabili e metodi non astratti

Astrazione: classi astratte




```
abstract class Message{

   private String sender;

   public Message(String from){
     sender = from;
   }

   public abstract void play();
   public String getSender(){
     return sender;
   }
}
```

```
class TextMessage extends Message{

   private String text;

   public TextMessage(String from, String t){
      super(from);
      text = t;
   }

   public void play(){
      System.out.println(text);
   }
}
```

Una classe astratta è una classe incompleta o che deve essere considerata incompleta

Classi astratte

- Una classe astratta non può essere istanziata
- Non si può dichiarare né `static` né `final` un metodo dichiarato `abstract`

```
abstract class Point {
        int x = 1, y = 1;
        void move(int dx, int dy) {
                x += dx;
                y += dy;
                alert();
        }
        abstract void alert();
}

abstract class ColoredPoint extends Point {
        int color;
}

class SimplePoint extends Point {
        void alert() { }
}
```

ERRORE !
```
Point p = new Point();
```

ERRORE !
```
Point p = new ColoredPoint();
```

OK !
```
Point p = new SimplePoint();
```

Interfacce

- Un'interfaccia è una collezione di definizioni di metodi (senza implementazione) e di costanti
  - Servono a catturare similarità tra classi senza forzare una relazione di sottoclasse
  - Servono a dichiarare metodi implementati da classi diverse
  - Si può rivelare l'interfaccia di un oggetto senza rivelare la sua classe (oggetti opachi)
- Dichiarando un'interfaccia, si dichiara un nuovo tipo riferimento, che può essere usato dovunque si usano i nomi dei tipi: dichiarazione di variabili, parametri di un metodo ecc.
- Per usare un'interfaccia bisogna definire una classe che la implementi, ovvero che implementi tutti i metodi in essa dichiarati

Interfacce

- Le interfacce sono utili per
  - dichiarare metodi che potranno essere implementati da una o più classi
  - determinare l'interfaccia di una classe senza rivelarne l'implementazione
  - catturare similarità tra classi che non sono legate in una gerarchia di ereditarietà
  - descrivere oggetti "function-like", che possono essere passati come parametri a metodi invocati da altri oggetti. Sono un'alternativa "safe" ai puntatori a funzioni usati in C e C++

Interfacce

- Le interfacce possono essere usate come alternativa all'ereditarietà multipla (non supportata in Java), anche se
  - Non si possono ereditare variabili da un'interfaccia
  - Non si possono ereditare implementazioni di metodi da un'interfaccia
  - La gerarchia delle interfacce è indipendente dalla gerarchia tra classi: classi che implementano la stessa interfaccia possono essere completamente scorrelate rispetto alla relazione di sottoclasse

Interfacce: esempio

```
import java.io.*;
import java.util.*
import nostri.*;

interface Ordinabile{
  boolean minore(Ordinabile a);
}

class Nazione implements Ordinabile{
  private String nome;
  Nazione (String s){
    nome = s;
  }
  public boolean minore(Ordinabile a){
    Nazione b = (Nazione) a;
    return (nome.compareTo(b.nome)<0);
  }
  public String toString(){
    return nome;
  }
}
```

```
class Sort{
  static void insertionSort(Ordinabile[] a) {
    Ordinabile temp;
    int pos;
    int n = a.length;
    for (int h = 0; h < n-1; h++) {
      pos = h;
      for (int j = h+1; j < n; j++)
        if (a[j].minore(a[pos])) pos = j;
      temp = a[pos]; a[pos] = a[h]; a[h] = temp;
    }
  }
}

class Table {
  public static void main(String[] args)
          throws IOException {
    Nazione[] T = new Nazione[10]; String s;
    DataInputStream in = new DataInputStream
            (new FileInputStream("Nazioni"));
    for (int i=0; i< T.length; i++) {
      s = Text.readString(in); T[i]=new Nazione(s)
    }
    Sort.insertionSort(T);
    for (int i=0; i< T.length; i++)
      System.out.println(T[i]);
  }
}
```

Classi interne

- Sono state aggiunte in JDK 1.1
- Permettono di inserire la dichiarazione di una classe all'interno della definizione di un'altra classe
- Sono utili per raggruppare classi che sono legate logicamente
- Il loro campo di visibilità è ristretto alla classe che le racchiude
- Il nome della classe interna deve differire da quello della classe che la racchiude
- Una classe interna può essere definita dentro ad un metodo
- Una classe interna può essere definita `abstract`

Classi "wrapper"

- Sono usate per guardare a elementi di un tipo primitivo come se fossero oggetti

Tipo Primitivo	Wrapper class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Un oggetto di questo tipo può essere costruito passando il valore al rispettivo costruttore, ad es.

```
int intero = 100;
Integer wintero = new Integer(intero);
```