Lezione

- Ereditarietà ….CONTINUA

Classi astratte

- è una classe che contiene la dichiarazione di alcuni metodi di cui viene specificata l'intestazione, ma **non** il corpo (metodi astratti)
- Pertanto non può essere istanziata
- A che serve??
  - A definire il comportamento comune a tutte le classi della gerarchia
  - A dichiarare (senza implementare) le funzionalità che sono specifiche delle classi che si estendono

Classi astratte

- Una classe incompleta che contiene metodi astratti
- È un segnaposto nella gerarchia delle classi, modella un concetto generico
  - Il modificatore **`abstract`** nell'intestazione della classe

- Una classe **`abstract`** diventa incompleta per definizione
- La classe **`abstract`** non può essere instanziata ma deve essere specializzata

Classi astratte

- Una classe deve essere dichiarata astratta se:
  - Contiene la dichiarazione di un metodo astratto
  - Eredita un metodo astratto dalla superclasse e non ne fornisce l'implementazione
  - Dichiara di implementare un'interfaccia, ma non fornisce l'implementazione di tutti i metodi

Classi astratte

- Il discendente di una classe astratta deve sovrascrivere i metodi astratti della superclasse da cui discende o sarà considerata anche lei astratta

- Un metodo astratto non può essere definito **`final`**, **`static`** o **`private`**
  - Perché dovrà essere sovrascritto (final)
  - Perchè non ha ancora una definizione (static)
  - Perchè non potrebbe essere ereditato (private)

Classe astratte – sintesi

- Una classe astratta è definita con il modificatore **`abstract`**
- Può contenere metodi implementati e costruttori.
- Se ha almeno un metodo astratto allora deve essere dichiarata astratta.
- Non è possibile istanziarla. Spesso dichiara una variabile del tipo classe astratta, in cui poi si archvivia il riferimento a oggetti delle sottoclassi
- Le sottoclassi devono fornire implementazione per tutti i metodi astratti o anche loro saranno considerate astratte.
- I metodi astratti non possono essere dichiarati con i modificatori **`private`**, **`static`** o **`final`**.

Quando sono appropriate

- Non si devono istanziare oggetti della classe. La dichiarazione **`abstract`** garantisce che non verrà mai istanziata.
- I dati comuni possono essere raccolti in una superclasse, anche senza metodi.
  - Qui si usa una gerarchia di classi in cui una classe serve per scambiare dati ma non ci sono metodi comuni
- Iniziare una gerarchia di classi che devono essere specializzate ulteriormente
  - Qui si usa la classe astratta per contenere metodi la cui specializzazione è rinviata alle classi specializzate
  - Definire il comportamento fondamentale della classe senza darne l'implementazione

Classe astratta: esempio

- è una classe che rappresenta forme geometriche nel piano cartesiano

```
abstract class Forma
{  protected String colore;

   public Forma(String col)
   {  this.colore= col;
   }
   public String get colore()
   {  return colore;
   }
   abstract Point center();
   abstract double area();

}
```

La classe Circle


```
public class Circle extends Forma{
  private Point center;
  private double raggio;

   public Circle(Point center,double raggio)
   {  this.center= center;
      this.radius= radius;}

   public double area()
   {  return Math.PI*radius*radius;
   }
   public Point center()
   {  return center;
   }
}
```

La classe Quadrato


```
public class Square extends Forma{
  private Point corner;//northWest
  private double side;

  public Square(Point corner,double side)
   {  this.corner = corner;
      this.side = side;}

  public double area()
   {  return side*side;}

   public Point center()
   { Point c=new Piont (corner);
     c.translate(side/2, -side/2); return c;}
 public double lato()
   {  return side;}
}
```

La classe TestForma

- è una classe che rappresenta forme geometriche nel piano cartesiano

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

  Square q=new Square(“5”,”blue”);
   System.out.println(q.area());
   System.out.println(q.colore());
   System.out.println(q.lato());

Forma q=new Square(“10”,”blue”);
   System.out.println(q.area());
   System.out.println(q.colore());
   System.out.println(q.lato()); //ERRORE
}
```

NOTA

- Forma è stata dichiarata abstract perché contiene almeno un metodo abstract
- Non può essere istanziata

Una variabile di tipo Forma:

- Può memorizzare il riferimento a un Quadrato od un Cerchio
- Può invocare un metodo definito dalla classe Forma:
  - Anche se trattasi di un metodo astratto (in tal caso ci si riferisce al riferimento memorizzato dalla variabile)

Interfaccia

- È un'unità di programmazione definita da un certo numero di metodi d'istanza e pubblici che sono implicitamente astratti
- È una classe a tutti gli effetti
- La classe che implementa un'interfaccia DEVE implementare tutti i metodi d'istanza definiti nell'interfaccia

Java Interface

- Possiamo dire che una **`interface`** specifica un tipo in termini dei servizi, ovvero dei metodi, che questi può offrire
- Una **`interface`** non specifica i dettagli implementativi dei vari servizi, specifica solamente in che modo i servizi possono essere invocati (nome, parametri, tipo restituito)
- In definitiva una **`interface`** consiste in una specifica delle segnature (e dei tipi restituiti) dai metodi che il tipo può offrire

Java interface

- Esempio:

```
public interface Strumento {
    public void produciSuono();
}
```

- L'interface **`Strumento`** definisce il tipo di oggetti che possono offrire il metodo **`produciSuono()`**

Java interface

- Nelle interface specifichiamo solo le segnature (e il tipo restituito) dei metodi che un tipo può offrire
- In una **`interface`** non c'è nessun dettaglio relativo alla implementazione
  - Niente variabili
  - Niente costruttori
  - Niente corpo dei metodi
- Le **`interface`** non si possono istanziare
- Ma una classe può implementare una (o più) interface
- Una classe che implementa una **`interface`** garantisce che le sue istanze rispettano il tipo specificato nella **`interface`**

Java interface

```
public class Tamburo implements Strumento {
        public void produciSuono() {
    System.out.println("bum-bum-bum");
                        }
                    }
```

- La parola chiave **`implements`** serve a specificare che la classe **`Tamburo`** implementa l'interfaccia **`Strumento`**
- Questo significa che gli oggetti **`Tamburo`** sono in grado di offrire i metodi del tipo **`Strumento`**

Java interface

- Una classe che implementa una **`interface`** può avere altri metodi (oltre a quelli della **`interface`**) specifici della classe

```
public class Chitarra implements Strumento {
        private int[] corde;
        public Chitarra(){
         corde = new int[6];
        }
    public void produciSuono() {
System.out.println("dlen-dlen-dlen");
        }
public int accorda(int corda, int val) {
        return corde[corda] += val;
        }
}
```

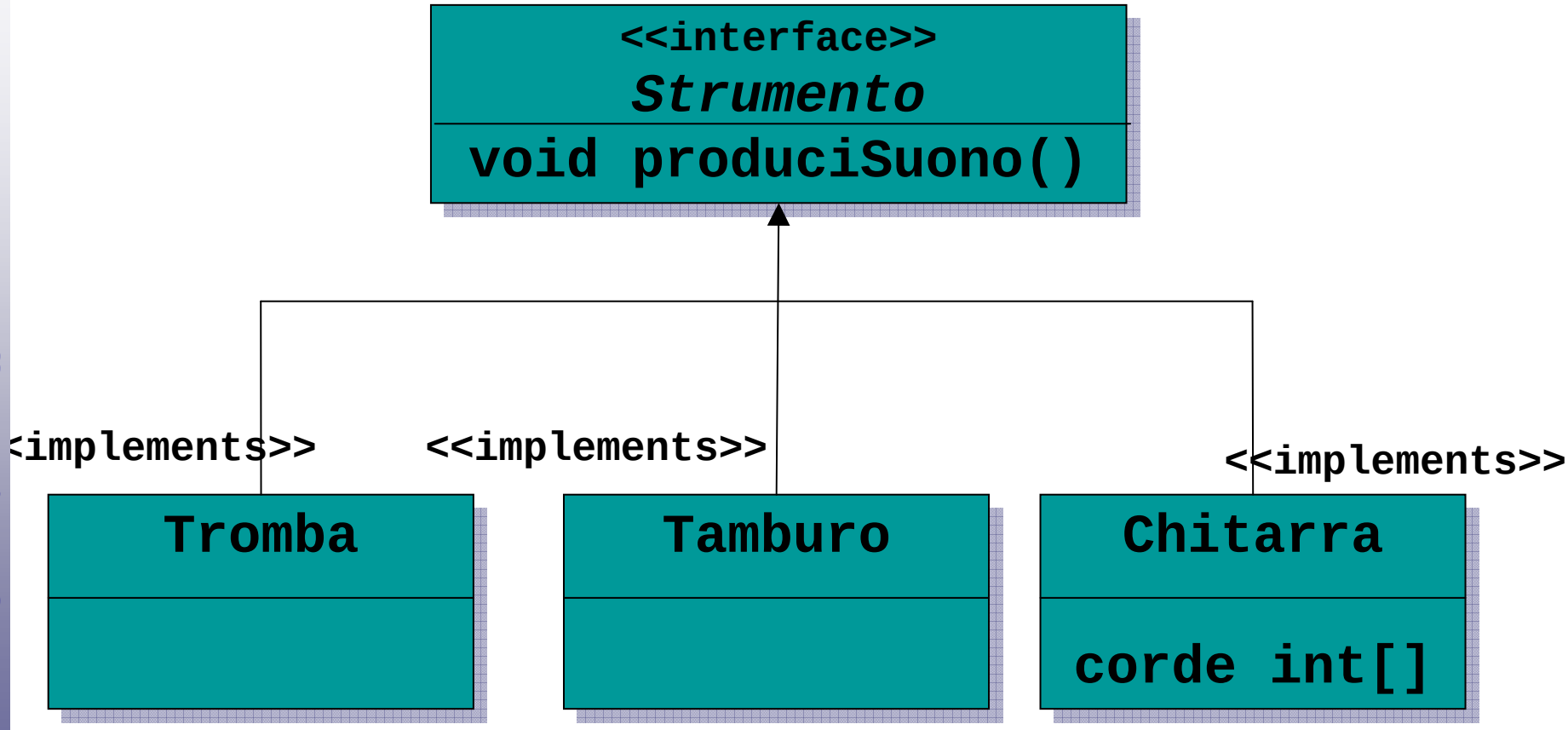
Esempio

```
public class Tamburo implements Strumento {
        public void produciSuono() {
        System.out.println("bum-bum-bum");
                }
        }
```

```
public class Tromba implements Strumento {
        public void produciSuono() {
        System.out.println("pe-pe-re-pe-pe");
                }
        }
```

```
public class Chitarra implements Strumento {
        private int[] corde;
         public Chitarra(){
         corde = new int[6];
                }
      public void produciSuono() {
    System.out.println("dlen-dlen-dlen");
                }
  public int accorda(int corda, int val) {
        return corde[corda] += val;
                }
        }
```

Diagramma delle Classi



Tipi, sottotipi, supertipi

- Abbiamo detto che una **`interface`** definisce un tipo
- Se la classe **`C`** implementa una **`interface I`** diciamo che:
  - **`C`** è un *sottotipo* di **`I`**

  e che
  - **`I`** è un *supertipo* di **`C`**
- Ad esempio **`Tamburo`** è un sottotipo di **`Strumento`**
- E **`Strumento`** è un supertipo di **`Tamburo`**

Interfaccia esempio

L'interfaccia **Comparable** è definita nel package **java.lang** definisce il confronto tra oggetti

```
public interface Comparable{

      int compareTo(Object o);

}
```

- L'interfaccia **Comparable** è definita nel pacchetto **java.lang**, per cui **non** deve essere importata né deve essere definita
  - la classe **String**, ad esempio, realizza **Comparable**

Realizzare una proprietà astratta

```
public interface Comparable
{  int compareTo(Object other);
}
```

- La definizione di un'interfaccia è simile alla definizione di una classe
  - si usa la parola chiave **interface** al posto di **class**
- Un'interfaccia può essere vista come ***una classe ridotta***, perché
  - non può avere costruttori
  - non può avere variabili di esemplare
  - ***contiene soltanto le firme di uno o più metodi*** non statici, ma non può definirne il codice
    - i metodi sono ***implicitamente*** **public**

Realizzare una proprietà astratta

- Se una classe dichiara di realizzare concretamente un comportamento astratto definito in un'interfaccia, deve ***implementare*** l'interfaccia
  - oltre alla dichiarazione, **DEVE** ***definire i metodi specificati nell'interfaccia, con la stessa firma***

```
public class BankAccount implements Comparable
{  ...
   public int compareTo(Object other)
   {  // notare che viene sempre ricevuto un Object
      BankAccount acct = (BankAccount)other;
      if (balance < acct.balance) return -1;
      if (balance > acct.balance) return 1;
      return 0;
   }
}
```

Realizzare una proprietà astratta


```
public class BankAccount implements Comparable
{  ...
   public int compareTo( BankAccount acc)
   {  // notare che viene ricevuto un BankAccount
      if (balance < acc.balance) return -1;
      if (balance > acc.balance) return 1;
      return 0;
   }
public int compareTo(Object other)
   {  // notare che viene sempre ricevuto un Object
return compareTo((BankAccount)other);
   }
}
```

Realizzare una proprietà astratta

- Non è possibile costruire oggetti da un'interfaccia

```
new Comparable(); // ERRORE DI SINTASSI
```

- È invece possibile definire riferimenti ad oggetti che realizzano un'interfaccia

```
Comparable c = new BankAccount(10);
```

- Queste conversioni tra un oggetto ed un riferimento ad una delle interfacce che sono realizzate dalla sua classe sono automatiche
  - ***come se l'interfaccia fosse una superclasse***

NOTA

- Un'interfaccia può essere implementata da più classi
- ***Una classe estende sempre una sola altra classe, mentre può realizzare più interfacce*** elencandole dopo la parola implements (separate da virgola) ***(eredità multipla in C++)***
- Un'interfaccia può essere definita come estensione di un'altra interfaccia
- Una classe può estendere un'altra classe e contemporaneamente implementare più interfacce

Interface, ruoli e riuso

- Consideriamo un problema noto che si presta naturalmente ad un comportamento polimorfo degli oggetti interessati: l'ordinamento
- Supponiamo di avere una classe che modella un "orario", espresso in ore e minuti
- Supponiamo di avere una collezione (per semplicità un array) di oggetti orario
- Supponiamo di voler ordinare questa collezione

La classe Orario

```
public class Orario {
   private int ore;
   private int minuti;

   public Orario(int ore, int minuti) {
     this.ore = ore;
     this.minuti = minuti;
   }

   public int getOre() {
     return this.ore;
   }

   public int getMinuti() {
     return this.minuti;
   }

    public boolean minoreDi(Orario o) {
          if (this.getOre() > o.getOre())
          return false;
          if (this.getOre() == o.getOre())
          return (this.getMinuti() < o.getMinuti());
          return true;
    }

   public String toString() {
     return this.getOre()+":"+this.getMinuti();
   }
```

Interface, ruoli e riuso

- Per ordinare la collezione creiamo una opportuna classe che offre questa funzionalità attraverso il metodo **`ordina(Orario[])`**
- Scriviamo il codice
  (usiamo un qualsiasi algoritmo di ordinamento, ad esempio selectionSort)
- vedi classe **`OrdinatoreOrari`**

La classe OrdinatoreOrari

```
public class OrdinatoreOrari {

 public static void ordina(Orario[] lista) {
   int imin;
   for (int ord=0; ord<lista.length-1; ord++) {
    imin = ord;
    for (int i=ord+1; i<lista.length; i++)
      if (lista[i].minoreDi(lista[imin])) {
          Orario temp=lista[i];
          lista[i]=lista[imin];
          lista[imin]=temp;
      }
   }
 }
}
```

Interface, ruoli e riuso

- Osserviamo bene il codice di **`OrdinatoreOrari`**
- Affinché gli oggetti dell'array possano essere ordinati, l'unica proprietà che questi oggetti devono avere è che possiedano un metodo **`minoreDi(Orario)`**
- In altri termini l'ordinamento funziona su oggetti che sappiano interpretare il ruolo di poter di essere confrontati
- Questo ruolo lo possiamo esplicitare in una opportuna interface

Interface, ruoli e riuso

- Creiamo l'interface **`Comparabile`**: gli oggetti delle classi che la implementano sono in grado di essere confrontati tramite il metodo **`minoreDi(Comparabile)`**

L'interface Comparabile


```
public interface Comparabile {

  public boolean minoreDi(Comparabile
  c);

}
```

Interface, ruoli e riuso

- Possiamo ora generalizzare la nostra classe **`Ordinatore`** (e il relativo algoritmo di ordinamento) affinché funzioni su tutte le classi che sappiano interpretare il ruolo **`Comparabile`**

La classe Ordinatore

```
public class Ordinatore {


 public static void ordina(Comparabile[] lista){
   int imin;
   for (int ord=0; ord<lista.length-1; ord++){
    imin = ord;
    for (int i=ord+1; i<lista.length; i++)
      if (lista[i].minoreDi(lista[imin])){
          Orario temp=lista[i];
          lista[i]=lista[imin];
          lista[imin]=temp;
      }
   }
```

La classe Orario (rivista)

```
ublic class Orario implements Comparabile {
  private int ore;
  private int minuti;
  public Orario(int ore, int minuti) {
    this.ore = ore;
    this.minuti = minuti;
  }
  public int getOre() {  return this.ore;  }

  public int getMinuti() { return this.minuti; }

  public boolean minoreDi(Comparabile c) {
    Orario o;
    o = (Orario)c;
    if (this.getOre() > o.getOre())       return false;
    if (this.getOre() == o.getOre())    return (this.getMinuti() < o.getMinuti());
    return true;
  }

  public String toString() { return this.getOre()+":"+this.getMinuti(); }
}
```

Interface, ruoli e riuso

- Per rispettare l'inteface, il metodo **`minoreDi()`** deve prendere come parametro un oggetto **`Comparabile`**

```
public boolean minoreDi(Comparabile c)
```

- Quando però scriviamo il codice, dobbiamo poter usare i metodi specifici della classe **`Orario`** (altrimenti non potremmo implementare il metodo!)
- Il compilatore non ce lo permette: il tipo statico del parametro è **`Comparabile`**

Downcasting

- Quello che facciamo è allora una forzatura sul tipo del parametro
- In particolare forziamo il sottotipo
- Questa operazione viene chiamata *downcasting* (in opposizione ad upcasting)

Downcasting

- Quando si forza il downcasting, la macchina virtuale effettua un controllo a tempo dinamico per verificare che l'operazione sia possibile
  - Ovvero verifica che l'oggetto appartenga al sottotipo a cui si sta forzando il cast
- In caso contrario il programma abortisce sollevando una eccezione **`java.lang.ClassCastException`**

Esercizio

- Scrivere in una classe di test un metodo con le istruzioni per:
  - Definire e creare un array di 5 oggetti Orario
  - Creare 5 oggetti orario, che rappresentino i seguenti orari: 12:30, 21:40, 9:20, 4:00, 1:35
  - Mettere i 5 oggetti creati negli elementi dell'array
  - Stampare l'array
  - Ordinare l'array
  - Stampare l'array e verificare che sia ordinato correttamente

Esercizio

- Scrivere una classe **`Data`**, che contenga i campi giorno, mese, anno (rappresentati con tre int), un costruttore con tre parametri, e i metodi accessori
- La classe Data deve implementare l'interfaccia **`Comparabile`**, descritta in precedenza (vedi codice di **`Orario`**)
- Scrivere un metodo che crea un array di oggetti **`Data`** e lo ordina usando il metodo **`Ordinatore.selectionSort`**()
- Scrivere una classe di test per verificare che, dopo l'invocazione del metodo **`Ordinatore.selectionSort`**() l'array sia effettivamente ordinato

Esercizio

- Introdurre nell'interfaccia **`Comparabile`** un nuovo metodo

  `int compara(Comparabile c)`

  restituisce un valore negativo, pari a **`0`**, positivo, se l'oggetto su cui è chiamato il metodo è rispettivamente minore, uguale, maggiore del valore del parametro.

- Nella classe **`Ordinatore`**, scrivere il codice del metodo

  `public static int`

  `ricercaBinaria(Comparabile`[] v, **`Comparabile ricercato`**)

  che implementa l'algoritmo di ricerca binaria (cfr Fondamenti II); questo metodo restituisce un intero il cui valore corrisponde alla posizione dell'elemento **`ricercato`** nell'array v oppure a **`-1`** se tale elemento non è presente

- Scrivere una classe di test per verificare se il metodo funziona correttamente

L'interfaccia Comparable

```
public interface Comparable
{  int compareTo(Object other);
}
```

- Come può l'interfaccia **Comparable** risolvere il nostro problema di ***definire un metodo di ordinamento valido per tutte le classi***?
  - basta definire un metodo di ordinamento che ordini un array di riferimenti ad oggetti che realizzano l'interfaccia **Comparable**, indipendentemente dal tipo

Variabili statiche

Problema

- Vogliamo modificare **BankAccount** in modo che
  - il suo stato contenga anche un ***numero di conto***

```
public class BankAccount
{  ...
   private int accountNumber;
}
```

  - il numero di conto sia assegnato dal costruttore
    - ogni conto deve avere un numero diverso
    - i numeri assegnati devono essere progressivi, iniziando da 1

Soluzione

- **Prima idea** (che non funziona…)

  usiamo una variabile per memorizzare l'ultimo numero di conto assegnato

```
public class BankAccount
{  ...
   private int accountNumber;
   private int lastAssignedNumber;
   ...
   public BankAccount()
   {  lastAssignedNumber++;
      accountNumber = lastAssignedNumber;
   }
}
```

Soluzione

```
public class BankAccount
{  ...
   private int accountNumber;
   private int lastAssignedNumber;
   ...
   public BankAccount()
   {  lastAssignedNumber++;
      accountNumber = lastAssignedNumber;
   }
}
```

- Questo costruttore non funziona perché la variabile **lastAssignedNumber** è una ***variabile di esemplare***, ne esiste una copia per ogni oggetto ed il risultato è che tutti i conti creati hanno il numero di conto uguale a 1

Variabili statiche

- Ci servirebbe una ***variabile condivisa da tutti gli oggetti della classe***
  - una variabile con questa semantica si ottiene con la dichiarazione **static**

```
public class BankAccount
{  ...
   private int static lastAssignedNumber;
}
```

- Una variabile **static** (***variabile di classe***) è condivisa da tutti gli oggetti della classe e ne esiste un'unica copia

Soluzione

```
public class BankAccount
{  ...
   private int accountNumber;
   private int lastAssignedNumber;
   ...
   public BankAccount()
   {  lastAssignedNumber++;
      accountNumber = lastAssignedNumber;
   }
}
```

- Questo costruttore non funziona perché la variabile **lastAssignedNumber** è una ***variabile di esemplare***, ne esiste una copia per ogni oggetto ed il risultato è che tutti i conti creati hanno il numero di conto uguale a 1

Variabili statiche

- Osserviamo che le variabili statiche non possono (da un punto di vista logico) essere inizializzate nei costruttori, perché il loro valore verrebbe inizializzato di nuovo ogni volta che si costruisce un oggetto, perdendo il vantaggio di avere una variabile condivisa!
- Bisogna inizializzarle mentre si dichiarano

```
private static int lastAssignedNumber = 0;
```

Variabili statiche

- Nella programmazione ad oggetti, ***l'utilizzo di variabili statiche deve essere limitato***, perché
  - metodi che leggono variabili statiche ed agiscono di conseguenza, hanno un ***comportamento che non dipende soltanto dai loro parametri*** (implicito ed espliciti)
- In ogni caso, le variabili statiche devono essere **private**, per evitare accessi indesiderati

Variabili statiche

- È invece pratica comune (senza controindicazioni) usare ***costanti*** statiche, come nella classe **Math**

```
public class Math
{  ...
   public static final double PI =
                3.14159265358979323846;
}
```

- Tali costanti sono di norma **public** e per ottenere il loro valore si usa il nome della classe seguito dal punto e dal nome della costante, **Math.PI**