Lezione XVII
Lu 5-Nov-2007

Argomenti inattesi
(pre-condizioni)

Argomenti inattesi

- Finora abbiamo visto eccezioni lanciate da metodi di classi della libreria, ma qualsiasi metodo può lanciare eccezioni
- Spesso un metodo richiede che i suoi parametri effettivi
  - siano di un tipo ben definito
    - questo viene garantito dal compilatore
    - es: `public char charAt(int index)`
  - abbiano un valore che rispetti certi ***vincoli***, ad esempio sia un numero positivo
    - in questo caso il compilatore non aiuta...
- Come deve reagire il metodo se riceve un parametro che non rispetta i ***requisiti richiesti*** (chiamati ***precondizioni***)?

Argomenti inattesi

- Ci sono quattro modi per reagire ad argomenti inattesi
  - ***non fare niente***
    - il metodo semplicemente termina la sua esecuzione senza alcuna segnalazione d'errore. In questo caso la documentazione del metodo deve indicare chiaramente le precondizioni. La responsabilità di ottemperare alle precondizioni è del metodo chiamante.
  - ***Verificare le precondizione, ed eseguire solo se sono soddisfatte***
    - `if (precondizioni) {…}`
    - questo però si può fare solo per metodi con valore di ritorno **void**, altrimenti che cosa restituisce il metodo?
    - se restituisce un valore casuale senza segnalare un errore, chi ha invocato il metodo probabilmente andrà incontro a un errore logico

Argomenti inattesi

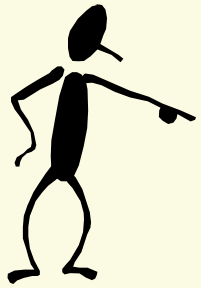
- ***terminare il programma con* System.exit(1)**
  - questo è possibile soltanto in programmi non professionali
  - La terminazione di un programma attraverso il metodo statico System.exit() non fornisce un meccanismo standard per informare dell'avvenuto
- ***lanciare un'eccezione***
- ***usare un'asserzione***

Argomenti inattesi

- Lanciare un'eccezione in risposta a un parametro che non rispetta una precondizione è la soluzione ***più corretta*** in ambito professionale
  - la libreria standard mette a disposizione tale eccezione
    - **java.lang.IllegalArgumentException**

```
/**
   ....
   @throws IllegalArgumentException
*/
public void deposit(double amount)
{
   if (amount <= 0)
      throw new IllegalArgumentException();
   balance = balance + amount;
}
```

Enunciato throw



- Sintassi:

```
throw oggettoEccezione;
```

- Scopo: lanciare un'eccezione
- Nota: di solito l'*oggettoEccezione* viene creato con **new** *ClasseEccezione*( )

```
throw new IllegalArgumentException();
```

```
throw new IllegalArgumentException
("stringa esemplificativa");
```

Le eccezioni in Java

- Quando un metodo ***lancia*** un'eccezione
  - l'esecuzione del metodo viene immediatamente interrotta
  - l'eccezione viene "propagata" al metodo chiamante la cui esecuzione viene a sua volta immediatamente interrotta
  - l'eccezione viene via via propagata fino al metodo **main()** la cui interruzione provoca l'arresto anormale del programma con la segnalazione dell'eccezione che è stata la causa di tale terminazione prematura
- Il lancio di un'eccezione è quindi un modo per terminare un programma in caso di errore
  - ***non sempre però gli errori sono così gravi...***
  - Vedremo che le eccezioni possono essere gestite

Usare un'asserzione

- Un'asserzione e' una condizione logica che dovrebbe essere vera in un punto particolare del codice
- L'asserzione verifica che la condizione sia vera
  - se la verifica ha successo, non succede niente
  - se la verifica fallisce, il programma termina con la segnalazione *AssertionError*
  - **solo se e' stata abilitata la verifica delle asserzioni**

```
public void deposit(double amount)
{
   assert amount > 0;
   balance = balance + amount;
}
```

Usare un'asserzione

- Per abilitare la verifica delle asserzioni si deve invocare l'interprete nel seguente modo:

```
$java -enableassertions MyClass
```

```
$java -ea MyClass
```

- Se la verifica delle asserzioni e' disabilitata, l'esecuzione non calcola le asserzioni e non viene appesantito.
- L'abilitazione delle asserzioni viene effettuata per i programmi in fase di sviluppo e collaudo.

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 di esemplare 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 e 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 static int lastAssignedNumber;
}
```

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

Variabili statiche

- Ora il costruttore funziona

```
public class BankAccount
{   ...
    private int accountNumber;
    private static int lastAssignedNumber = 0;
    ...
    public BankAccount()
    {   lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
    }
}
```

- Ogni metodo (o costruttore) di una classe può accedere alle variabili statiche della classe e modificarle

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 e 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 =
     3.141592653589793;
   public static final double E =
     2.718281828459045;
}
```

- 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**

Ambito di Visibilita'delle Variabili (scrittura del codice – compile time)

- Fin ora abbiamo incontrato in Java quattro diversi tipi di variabili
  - variabili ***locali*** (all'interno di un metodo)
  - variabili ***di esemplare***
  - variabili ***parametro*** (dette ***parametri formali***)
  - variabili ***statiche*** o di classe
- Vediamo ora qual è il loro ***ambito di visibilità*** ovvero quale è la ***zona*** di programma in cui ci si puo' riferire a una variabile mediante il suo nome

Variabili locali

- L'ambito di visibilità di una **variabile locale** è dal punto in cui è dichiarata fino alla fine del blocco che la contiene:
- La variabile e' visibile nei blocchi di codice annidati all'interno del blocco in cui e' definita

```
int n = 0;
while (true)
{   String inLine = in.next();
    if (inLine == null)
        break;
    n = Integer.parseInt(inLine);
   // qui n e' visibile
}
// qui inLine non e' visibile
// mentre n e' visibile
```

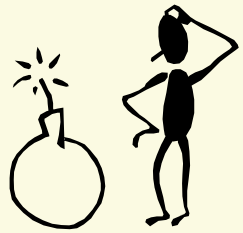
```
for (int i = 0; i < max; i++)
{…}
// qui int i non e' visibile
```

Il ciclo `for` fa eccezione

Visibilità sovrapposta

- Non si possono avere due variabili locali diverse con visibilita' sovrapposta.

```
int inLine = Integer.parseInt(line);
while (true)
{
    String inLine = in.next(); //errore!
    …
}
```



Il compilatore segnala errore

***ClassName*.java: 12: inLine is already defined in …**

```
String inLine = in.next();
```

Visibilità non sovrapposta

- Se gli ambiti di visibilità non sono sovrapposti, si possono avere variabili locali diverse con lo stesso nome

```
if (…)
{
   int k = in.nextInt();     //OK
}
else
{
   String k = in.next();     //OK
}
```

```
if (…)
{
   int k = in.nextInt();     //OK
}
else
{
   int k = 7;     //OK
}
```

Variabili di esemplare

- L'ambito di visibilità delle variabili di esemplare dipende dagli specificatori di accesso con cui sono dichiarate
- **`private`**: la variabile di esemplare è visibile in tutti e soli i metodi di esemplare della classe. Nei metodi di esemplare della classe si può accede alla variabile col solo nome: es. ***`nomeVariabile`***

```
public class BankAccount
{  public void deposit(double amount)
   {  balance = balance + amount;
   }
   public void withdraw(double amount)
   {  balance = balance - amount;
   }
   …
   private double balance;
}
```

Variabili di esemplare

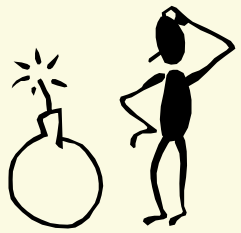
- Nell'accesso con il semplice nome nei ***metodi di esemplare***, la variabile viene attribuita al parametro implicito.

```
public class BankAccount
{  public void deposit(double amount)
   {  this.balance = this.balance + amount;
   }
   public void withdraw(double amount)
   {  this.balance = this.balance - amount;
   }
   …
   private double this.balance;
}
```

Variabili di esemplare

- Nei metodi statici (ad esempio nel metodo main()) non e' possibile accedere alle variabili di esemplare con il semplice nome
  - Non c'e' il parametro implicito!

```
public class BankAccount
{  public void deposit(double amount)
   {  this.balance = balance + amount;
   }
   public void withdraw(double amount)
   {  this.balance = balance - amount;
   }
   // …
   private double balance;

   public static void main(String[] args)
   {
      balance = 1000;
   }
}
```

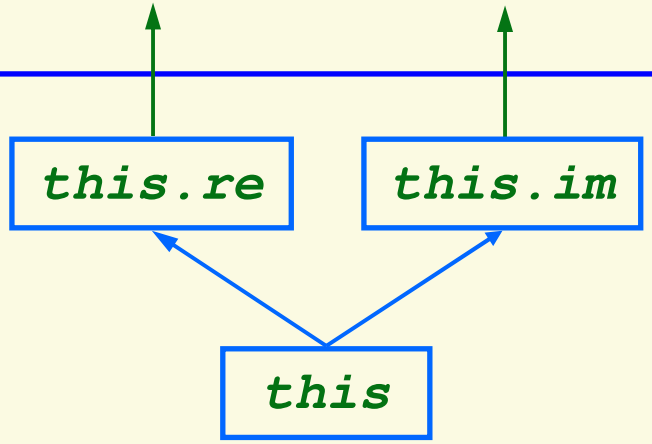


BankAccount.java: 13 non-static variable balance cannot be referenced from a static context

Variabili di esemplare

- Nei metodi della classe si può accedere direttamente alle variabili di esemplare di eventuali oggetti, diversi dal parametro implicito, con la sintassi
  - *riferimento.nomeVariabile*

```
// classe MyComplex
 public MyComplex add(MyComplex z)
 {
    return new MyComplex(re + z.re, im + z.im);
 }
```



Variabili di esemplare

- **`public`**: la variabile, oltre a essere visibile in tutti i metodi di esemplare della classe come le variabili private, è accessibile da qualsiasi metodo attraverso il costrutto ***`oggetto.nomeVariabile`*** (es. **values.length**)
- Nei metodi di esemplare della classe la variabile è accessibile usando solo il suo nome ***`nomeVariabile`***

- **`protected`**: la variabile, oltre a essere visibile in tutti i metodi della classe col suo none ***`nomeVariabile`***, è accessibile
  - dai metodi delle classi appartenenti allo stesso package
  - da tutte le sottoclassi

Uso dei metodi nelle classi

- **Un metodo di una classe, privato o pubblico, statico o non, è visibile all'interno dei metodi della classe stessa con il nome del metodo: es *nomeMotodo()***

```
public class Triangolo
{ …
  public double area() // calcola l'area
  {…}

  public double h() //calcola l'altezza sul lato a
  {
      return 2 * area() / lato_a;
  }
  private int lato_a, lato_b, lato_c;
}
```

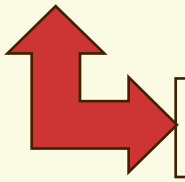
Variabili di esemplare

- Abbiamo già visto che variabili locali diverse non possono avere lo stesso nome se hanno ambiti di visibilità sovrapposti
- Gli ambiti di visibilità di variabili locali e di esemplare possono essere invece sovrapposti

```
public class Student
{  public void readNames()
   {  …
      String nome = in.next();
      …
   }
   …
   private String nome;
}
```

variabile locale

variabile di esemplare



Il compilatore non segnala errore

Mettere in ombra le variabili

- Utilizzare lo stesso nome per una variabile di esemplare e per una variabile locale può portare a un errore logico comune

```
public class Student
{
    public Student(String aName)
    {
        String name = aName; //errore!!
    }
    …
    private String name;
}
```

variabile locale

Errore logico

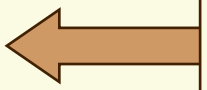
variabile di esemplare

- La variabile di esemplare **`String Name`** non viene modificata nel costruttore, bensi' viene modificata la variabile locale. Il compilatore non segnala errore!
- Il costruttore **`public Student`** non è in grado quindi di inizializzare la variabile di esemplare **`String name`**

Variabili Parametro

- Il loro ambito di visibilità coincide con il metodo a cui appartengono.
- In un metodo le variabili locali e i parametri prevalgono sulle variabili di esemplare!
- Non usare parametri di metodi e variabili di esemplare con lo stesso nome

```
public class Moneta
{    public Moneta(double valore, String nome)
     {
          this.valore = valore;
          this.nome = nome;
     }
     …
     private double valore;
     private String nome;
}
```



this permette in questo caso di risolvere l'ambiguita'

Variabili Parametro

- Conviene evitare di usare parametri di metodi e variabili di esemplare con lo stesso nome. Talvolta si ha questa tentazione con i parametri dei costruttori
- Usare invece una convenzione come la seguente

```
public class Moneta
{ public Moneta(double unValore, String unNome)
  {
     valore = unValore;
     nome = unNome;
  }
   …
  private double valore;
  private String nome;
}
```

Ambito di visibilità di una variabile

- È anche possibile dichiarare variabili di esemplare ***senza indicare uno specificatore di accesso*** (***accesso di default***)
  - sono così visibili anche all'interno di classi che si trovano ***nello stesso package*** (cioè di file sorgenti che si trovano nella stessa cartella)
  - anche le variabili **protected** hanno questa proprietà

Variabili Statiche

- Come per le variabili di esemplare, la visibilità delle variabili statiche dipende dallo specificatore di accesso:
  - `public`
  - `private`
  - `protected`
  - `accesso di default`

Lezione XVIII
Ma 6-Nov-2007

Correzione del questionario a risposte multiple

Ciclo di vita delle variabili

Ciclo di vita di una variabile (esecuzione del codice - run-time)

- Consideriamo i quattro diversi tipi di variabili
  - variabili ***locali*** (all'interno di un metodo)
  - variabili ***parametro*** (dette ***parametri formali***)
  - variabili ***di esemplare*** o di istanza
- Vediamo ora qual è il loro ***ciclo di vita***, cioè quando vengono create e fin quando continuano a occupare lo spazio in memoria riservato loro

Ciclo di vita di una variabile

- Una ***variabile locale***
  - ***viene creata*** quando viene eseguito l'enunciato in cui viene definita
  - ***viene eliminata*** quando l'esecuzione del programma esce dal blocco di enunciati in cui la variabile e' stata definita
    - se non è definita all'interno di un blocco di enunciati, viene eliminata quando l'esecuzione del programma esce dal metodo in cui la variabile viene definita

Ciclo di vita di una variabile

- Una *variabile parametro (formale)*
  - *viene creata* quando viene invocato il metodo
  - *viene eliminata* quando l'esecuzione del metodo termina
- Una *variabile statica*

  - *viene creata* quando la macchina virtuale Java carica la classe per la prima volta, *viene eliminata* quando la classe viene scaricata dalla macchina virtuale Java
  - ai fini pratici, possiamo dire che ***esiste sempre***...

Ciclo di vita di una variabile

- Una ***variabile di esemplare***
  - ***viene creata*** quando viene creato l'oggetto a cui appartiene
  - ***viene eliminata*** quando l'oggetto viene eliminato
- Un oggetto viene eliminato dalla JVM quando non esiste più alcun riferimento ad esso
  - la zona di memoria riservata all'oggetto viene "riciclata", cioè resa di nuovo libera, dal ***raccoglitore di rifiuti*** (***garbage collector***) della JVM, che controlla periodicamente se ci sono oggetti da eliminare

Ambito di visibilità di una variabile

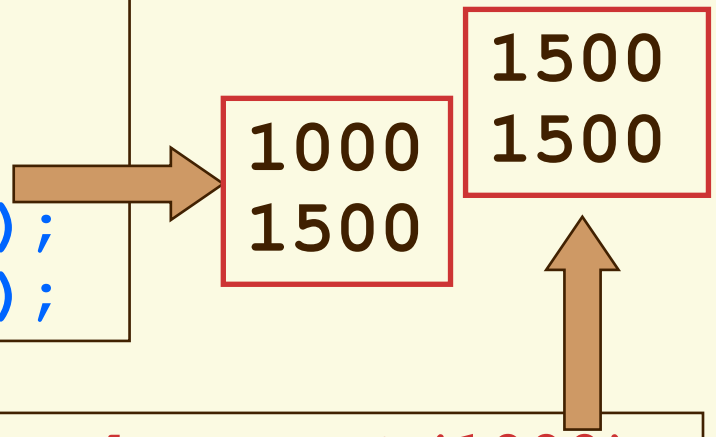
- Conoscere l'***ambito di visibilità*** e il ***ciclo di vita*** di una variabile è molto importante per capire quando e dove è possibile ***usare di nuovo il nome di una variabile che è già stato usato***
- Le regole appena viste consentono di usare, in metodi diversi della stessa classe, ***variabili locali*** o ***variabili parametro*** ***con gli stessi nomi***, senza creare alcun conflitto, perché
  - i rispettivi ambiti di visibilità non sono sovrapposti
- In questi casi, le variabili definite nuovamente non hanno alcuna relazione con le precedenti

Copiare i riferimenti agli oggetti

Copiare variabili

- Il comportamento della ***copia di variabili*** è molto diverso nel caso in cui si tratti di ***variabili oggetto*** o di ***variabili di tipi numerici*** fondamentali

```
double x1 = 1000;
double x2 = x1;
x2 = x2 + 500;
System.out.println(x1);
System.out.println(x2);
```

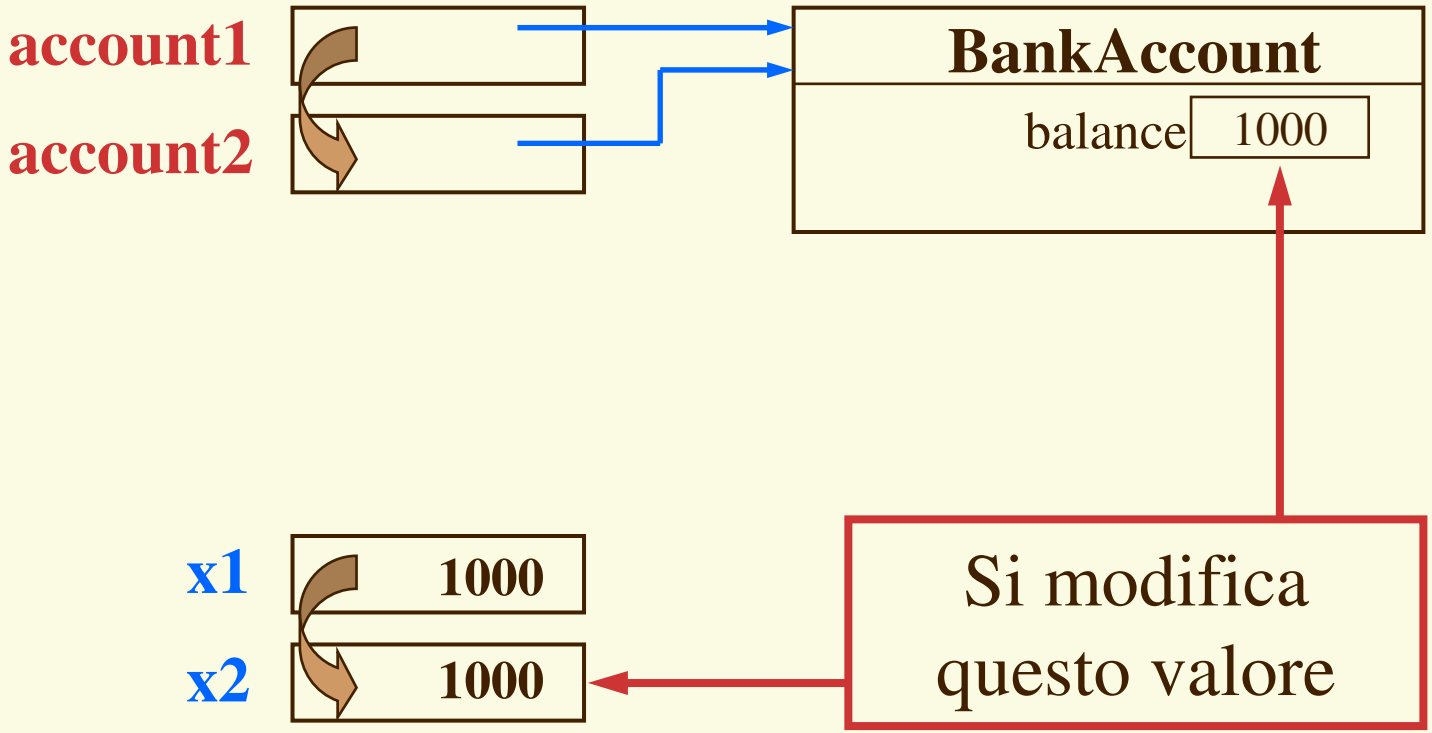



```
BankAccount account1 = new BankAccount(1000);
BankAccount account2 = account1;
account2.deposit(500);
System.out.println(account1.getBalance());
System.out.println(account2.getBalance());
```

Copiare variabili

- Le variabili di tipi numerici fondamentali indirizzano una posizione in memoria che contiene un ***valore***, che viene copiato con l'assegnazione
  - cambiando il valore di una variabile, non viene cambiato il valore dell'altra
- Le variabili oggetto indirizzano una posizione in memoria che, invece, contiene un ***riferimento ad un oggetto***, e solo tale riferimento viene copiato con l'assegnazione
  - modificando lo stato dell'oggetto, tale modifica è visibile da entrambi i riferimenti
  - ***non viene creata una copia dell'oggetto!***

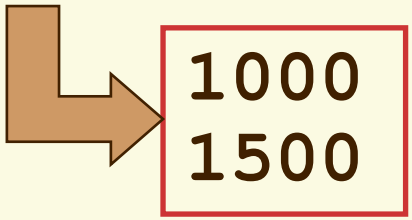
Copiare variabili



Copiare variabili

- Se si vuole ottenere anche con variabili oggetto lo stesso effetto dell'assegnazione di variabili di tipi numerici fondamentali, è necessario ***creare esplicitamente una copia dell'oggetto con lo stesso stato***, inizializzarlo adeguatamente e assegnarlo alla seconda variabile oggetto

```
BankAccount acc1 = new BankAccount(1000);
BankAccount acc2 = new BankAccount(acc1.getBalance());
acc2.deposit(500);
System.out.println(acc1.getBalance());
System.out.println(acc2.getBalance());
```



Progettazione di classi

Individuare le classi

- Una delle prime fasi di un ***progetto informatico*** da realizzare con un linguaggio di programmazione orientato agli oggetti (**OOP**) come Java consiste nell'***analisi delle specifiche del problema*** per ***individuare le classi*** con cui costruire gli oggetti

- Successivamente, si tratta di ***definire le interfacce*** pubbliche delle classi, cioè i loro metodi

- Come si fa?

Individuare le classi

- Come in molte altre attività di progettazione tipiche dell'ingegneria, ***non ci sono regole precise*** da seguire per impostare la soluzione tecnica dei problemi a partire dalle loro specifiche
  - importante l'esperienza

- Possiamo però seguire due ***regole empiriche***, cioè dettate dall'***esperienza*** anziché dalla teoria
  - ogni classe deve rappresentare un singolo concetto
  - alcuni ***sostantivi*** ("nomi") nella specifica del problema diventano le ***classi***
  - alcuni ***verbi*** nella specifica del problema diventano i ***metodi*** delle classi così individuate

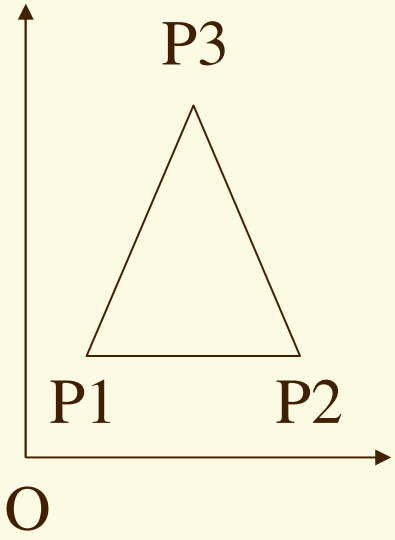
Individuare le classi

- Supponiamo di dover scrivere un programma per elaborare e raffigurare figure geometriche nel piano
- Ci soffermiamo su una figura geometrica semplice: ***il triangolo***
- Che azioni vogliamo effettuare su un triangolo nel piano?
  - traslare un triangolo
  - ruotare un triangolo
- Applichiamo la regola empirica:
  - sostantivi: triangolo ⇨ classe: MyTriangle2D
  - Verbi ⇨ metodi:
    - Traslare ⇨ traslate()
    - Ruotare rotate()

Classe TriangoloNP

- Un triangolo nel piano sia identificato da tre punti P1, P2 e P3, che rappresentano i suoi vertici
- I metodi translate() e rotate() restituiscano un triangolo traslato o ruotato, senza modificare il triangolo su cui sono invocati

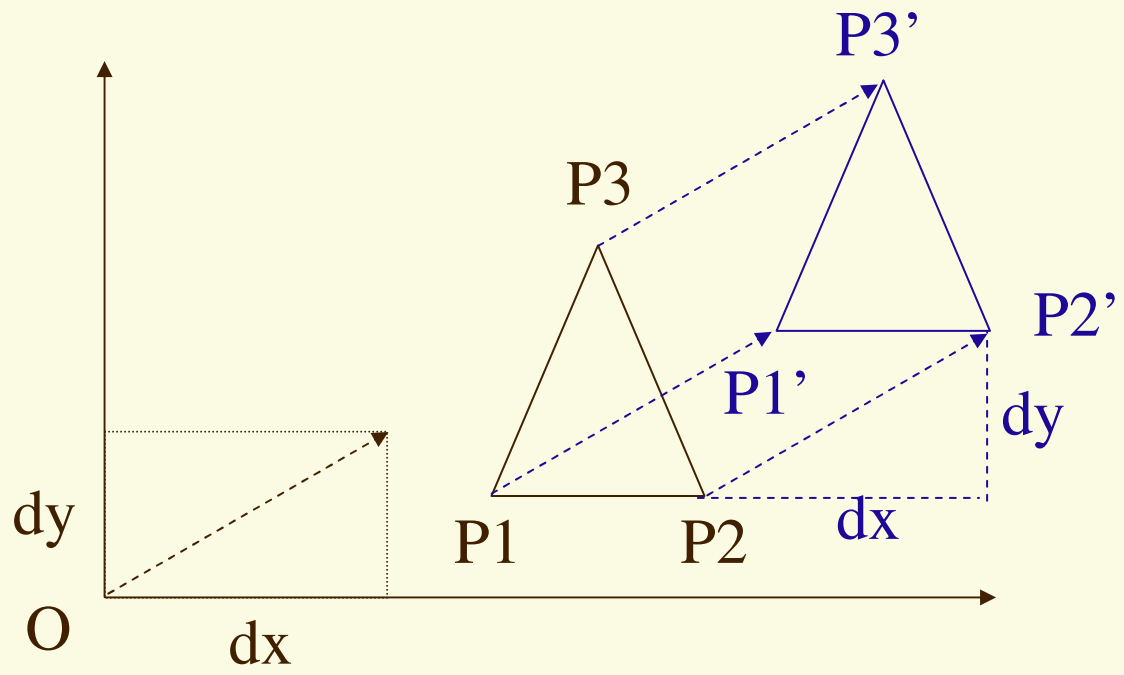
```
public class MyTriangle2D
{
  private double xP1, yP1; // punto P1
  private double xP2, yP2; // punto P2
  private double xP3, yP3; // punto P3

  public MyTriangle2D(...) {  }
  public MyTriangle2D translate(...) {  }
  public MyTriangle2D rotate(...)  {  }
}
```



Traslare un triangolo

- I punti del triangolo vengono traslati di certe quantità lungo gli assi x e y

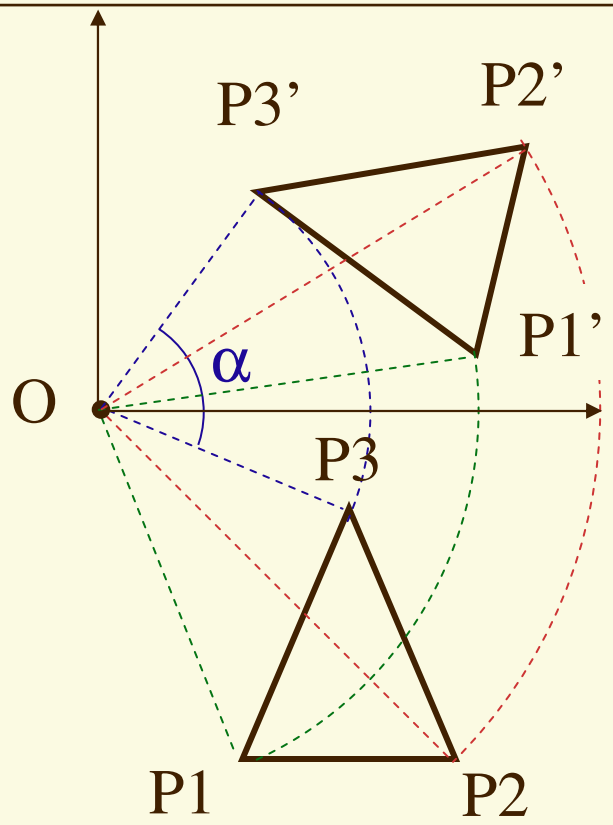
```
public MyTriangle2D translate(double dx, double dy)
{   }
```



Ruotare un triangolo

- I punti del triangolo vengono ruotati di un angolo α con centro di rotazione nell'origine degli assi

```
public MyTriangle2D rotate(double alpha)
{   }
```



Osservazione

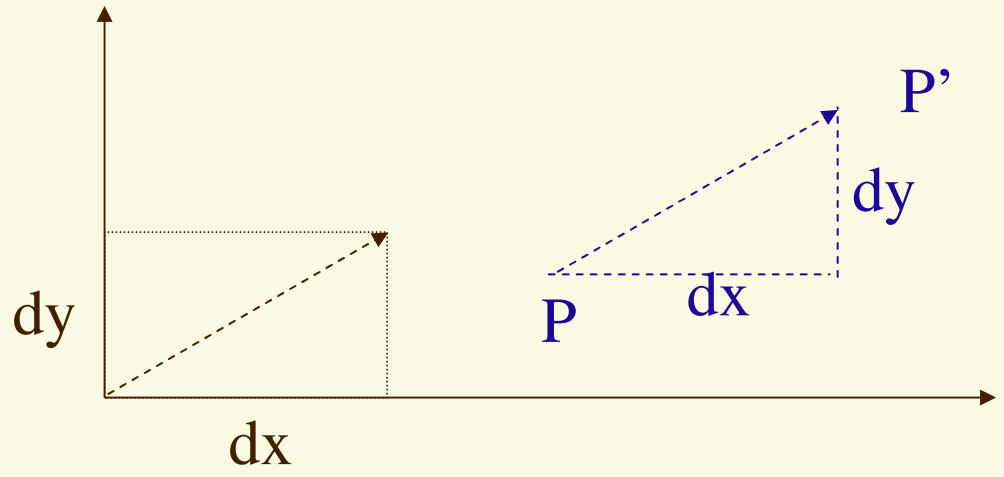
- Notiamo che:
  - un triangolo si definisce con tre vertici che sono ***punti nel piano***
  - traslare e ruotare un triangolo significa, in realtà, traslare e ruotare i suoi vertici, ovvero dei ***punti nel piano***
  - il ***punto*** rappresenta un singolo concetto ben definito
- Identifichiamo quindi il punto come una classe a se stante

```
public class MyPoint2D
{
   private double x;  // ascissa
   private double y;  // ordinata

   public MyPoint2D(double px, double py)
   {  x = px;
      y = py;
   }

   public MyPoint2D translate(...) {   }
   public MyPoint2D rotate(...) {   }
}
```

Traslare un punto

- Traslazione di un punto

```
public MyPoint2D translate(double dx, double dy)
{
    return new MyPoint2D(x + dx, y + dy);
}
```

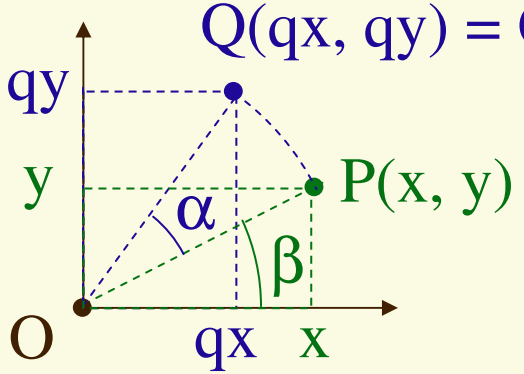


Ruotare un punto

- Rotazione di un punto di un angolo α con centro di rotazione nell'origine

```
public MyPoint2D rotate(double alpha)
{
   double qx = x * Math.cos(alpha) - y * Math.sin(alpha);
   double qy = x * Math.sin(alpha) + y * Math.cos(alpha);
   return new MyPoint2D(qx, qy);
}
```

$$Q(qx, qy) = Q(x\cos\alpha - y\sin\alpha,\ x\sin\alpha + y\cos\alpha)$$



Un po' di trigonometria!

$$\overline{QO} = \overline{PO}$$
$$x = \overline{PO}\cos\beta,\ y = \overline{PO}\sin\beta$$
$$\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$$
$$\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$$

$$qx = \overline{QO}\cos(\alpha+\beta)$$
$$qx = \overline{PO}\cos\beta\cos\alpha - \overline{PO}\sin\beta\sin\alpha$$
$$qx = x\cos\alpha - y\sin\alpha$$

$$qy = \overline{QO}\sin(\alpha+\beta)$$
$$qy = \overline{PO}\cos\beta\sin\alpha + \overline{PO}\sin\beta\cos\alpha$$
$$qy = x\sin\alpha + y\cos\alpha$$

Classe MyPoint2D

```
import java.util.Locale;
public class MyPoint2D
{  private double x;  // as
   private double y;  // ordinata

   public MyPoint2D(double px, double py)
   {  x = px;
      y = py;
   }

   public MyPoint2D translate(double dx, double dy)
   {  return new MyPoint2D(x + dx, y + dy);
   }

   public MyPoint2D rotate(double alpha)
   {  double qx = x * Math.cos(alpha) - y * Math.sin(alpha);
      double qy = x * Math.sin(alpha) + y * Math.cos(alpha);
      return new MyPoint2D(qx, qy);
   }

   public String toString()
   {  return String.format(Locale.US, "P(%.2f, %.2f)", x, y);
   }
}
```

Classe MyTriangle2D

```
public class MyTriangle2D
{ private MyPoint2D p1;
  private MyPoint2D p2;
  private MyPoint2D p3;
  public MyTriangle2D(MyPoint2D q1, MyPoint2D q2, MyPoint2D q3)
  {  p1 = q1;
     p2 = q2;
     p3 = q3;
  }
  public MyTriangle2D translate(double dx, double dy)
  {  MyPoint2D s1 = p1.translate(dx, dy);
     MyPoint2D s2 = p2.translate(dx, dy);
     MyPoint2D s3 = p3.translate(dx, dy);
     return new MyTriangle2D(s1, s2, s3);
  }
  public MyTriangle2D rotate(double alpha)
  {  MyPoint2D s1 = p1.rotate(alpha);
     MyPoint2D s2 = p2.rotate(alpha);
     MyPoint2D s3 = p3.rotate(alpha);
     return new MyTriangle2D(s1, s2, s3);
  }
  public String toString()
  {  return "triangolo [" + p1 + ", " + p2 + ", " + p3 + "]";
  }
}
```

ElaboratoreTriangoli

- Programmiamo ora una classe di prova che esegua delle semplici elaborazioni su un triangolo

- Si acquisiscano le coordinate di tre vertici da standard input
- Si costruisca un triangolo
- Si trasli il triangolo lungo gli assi x e y di una certa quantità
- Si ruoti il triangolo di un certo angolo

ElaboratoreTriangoli

```
import java.util.Scanner;
import java.util.Locale;

public class ElaboratoreTriangoli
{
   public static void main(String[] args)
   {
      final double DELTA_X = 0.0; // spostamento lungo l'asse x
      final double DELTA_Y = 0.0; // spostamento lungo l'asse y
      final double ROTATION_ANGLE = Math.PI / 2;

      Scanner in = new Scanner(System.in);
      in.useLocale(Locale.US);

      System.out.print("Vertice V1: x e y?: ");
      double x = in.nextDouble();
      double y = in.nextDouble();
      MyPoint2D p1 = new MyPoint2D(x, y);

      System.out.print("Vertice V2: x e y?: ");
      x = in.nextDouble();
      y = in.nextDouble();
      MyPoint2D p2 = new MyPoint2D(x, y);

// continua
```

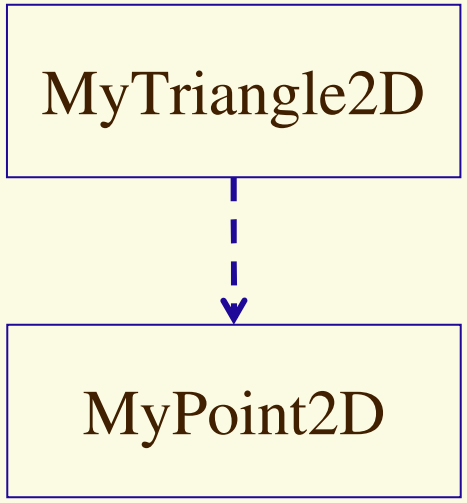
ElaboratoreTriangoli

```
        System.out.print("Vertice V3: x e y?: ");
        x = in.nextDouble();
        y = in.nextDouble();
        MyPoint2D p3 = new MyPoint2D(x, y);

        in.close();

        MyTriangle2D t = new MyTriangle2D (p1, p2, p3);

        MyTriangle2D t2 = t.translate(DELTA_X, DELTA_Y);
        t2 = t2.rotate(ROTATION_ANGLE);

        System.out.println("triangolo in ingresso: " + t);
        System.out.println("triangolo in uscita:   " + t2);
    }
}
```

Relazione di dipendenza fra le classi

- Il fatto che la classe ***MyTriangle2D*** usi oggetti di classe ***MyPoint2D*** si esprime dicendo che ***la classe MyTriangle2D dipende dalla classe MyPoint2D***
- La dipendenza fra classi viene spesso espressa in forma grafica con ***diagrammi di classi*** in notazione ***UML***
- UML: Unified Modeling Language
  - formalismo per rappresentare il rapporto fra le classi
  - ogni classe viene rappresentata con un rettangolo.
  - una linea tratteggiata indica la dipendenza di una classe da un'altra classe
  - Sulla linea c'è una freccia rivolta verso la classe nei confronto della quale esiste la dipendenza
    - la classe ***MyTriangle2D*** usa oggetti di classe ***MyPoint2D***
    - la classe ***MyPoint2D*** non contiene elementi della classe ***MyTriangle2D***

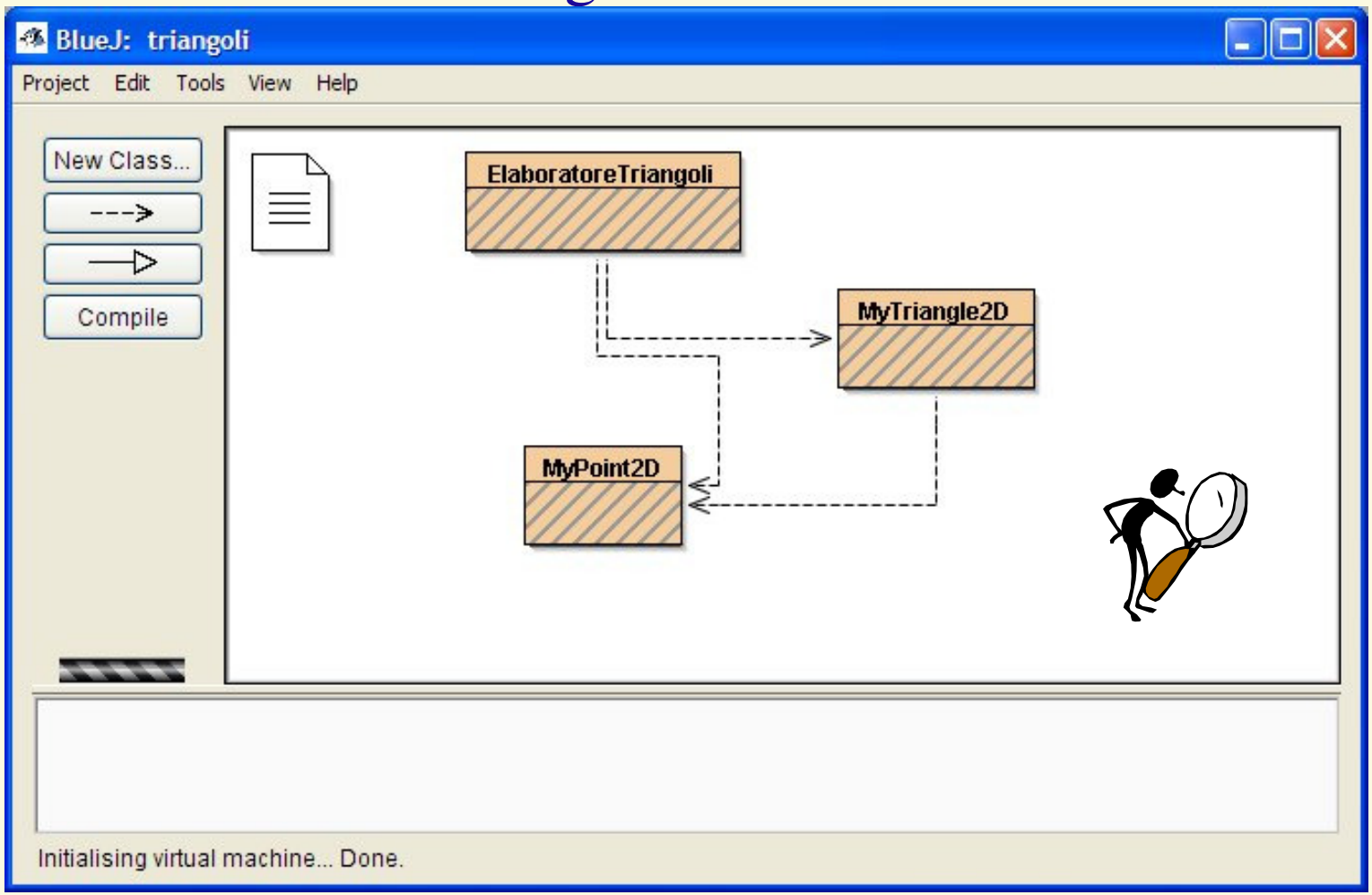
Relazione di dipendenza fra le classi



- Se ***molte classi*** di un programma dipendono l'una dall'altra, si dice che il programma ha un ***elevato accoppiamento*** fra le classi
- Se, invece, esistono ***poche*** dipendenze fra le classi, si dice che l'accoppiamento è ***basso***
- E' importante evitare accoppiamenti non necessari, perché questo semplifica la manutenzione del codice

Relazione di dipendenza fra le classi

- L'ambiente di sviluppo integrato BluJ rappresenta le classi mediante un diagramma di classi UML.



Ridurre l'accoppiamento

- Entrambe le classi ***ElaboratoreTriangoli*** e ***MyTriangle2D*** dipendono dalla classe ***MyPoint2D***
- Eliminiamo la dipendenza della classe ElaboratoreTriangoli dalla classe MyPoint2D
- Per fare questo modifichiamo la classe MyTriangle2D aggiungendo un nuovo costruttore

```
…
public MyTriangle2D( double x1, double y1,
                     double x2, double y2,
                     double x3, double y3
                   )
{  p1 = new MyPoint2D(x1, y1);
     p2 = new MyPoint2D(x2, y2);
     p3 = new MyPoint2D(x3, y3);
}
…
```

Ridurre l'accoppiamento

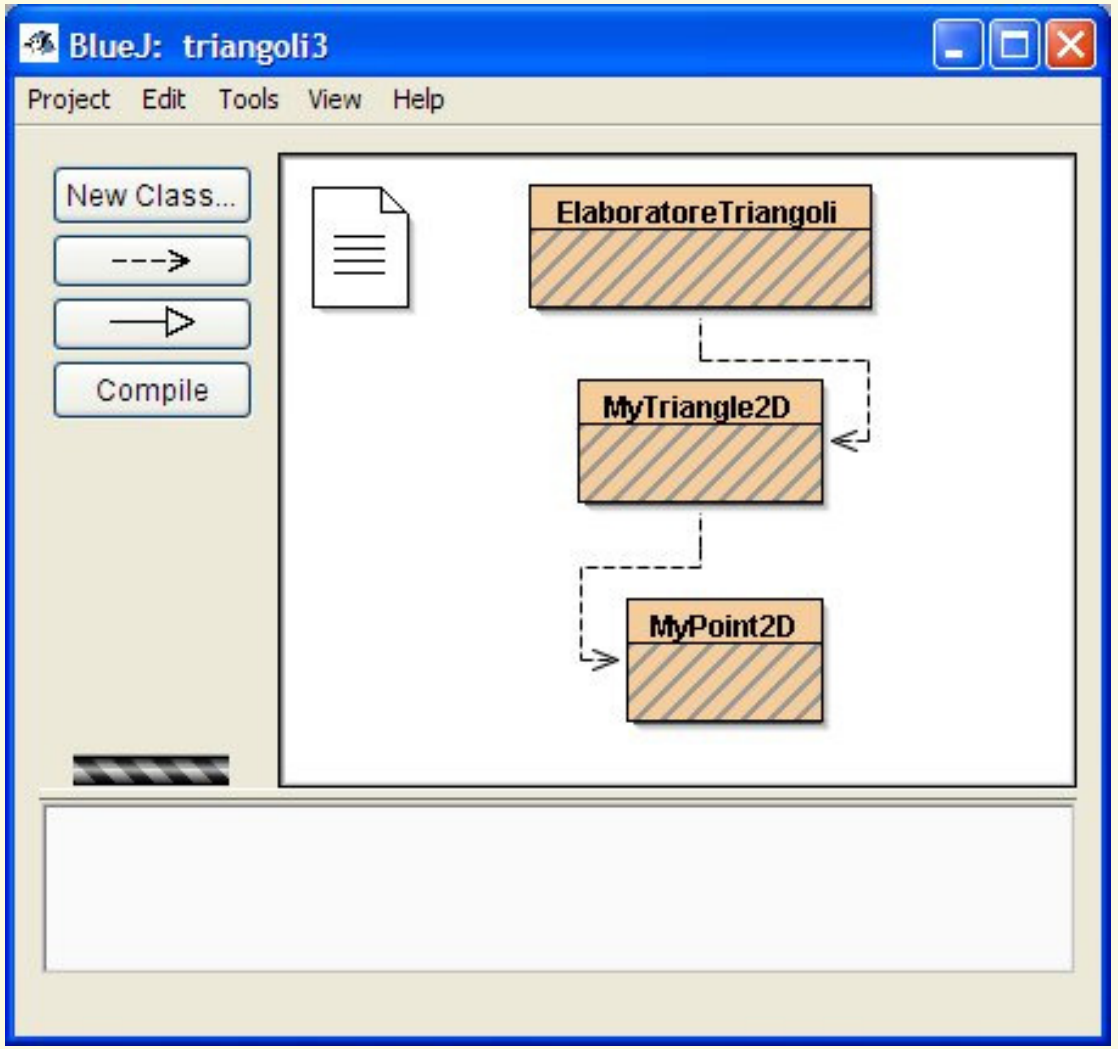
- Modifichiamo ora la classe ***ElaboratoreTriangoli*** in modo che non dipenda piu' direttamente dalla classe ***MyPoint2D***

```
Scanner in = new Scanner(System.in);
in.useLocale(Locale.US);

System.out.print("Vertice V1: x e y?: ");
double x1 = in.nextDouble();
double y1 = in.nextDouble();

System.out.print("Vertice V2: x e y?: ");
double x2 = in.nextDouble();
double y2 = in.nextDouble();

System.out.print("Vertice V3: x e y?: ");
double x3 = in.nextDouble();
double y3 = in.nextDouble();

in.close();

MyTriangle2D t = new MyTriangle2D (x1, y1, x2, y2, x3, y3);
```

Ridurre l'accoppiamento

- Il nostro programma ha ora un livello di acccoppiamento inferiore

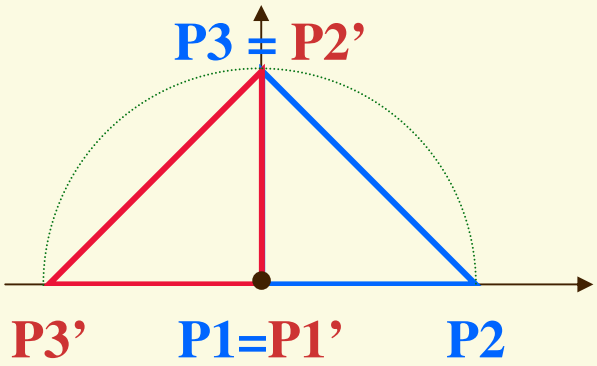
- Un'eventuale modifica della classe ***MyPoint2D*** potra' richiedere di modificare la classe ***MyTriangle2D***, ma non richiedera' la modifica della classe ElaboratoreTriangoli



Casi di prova

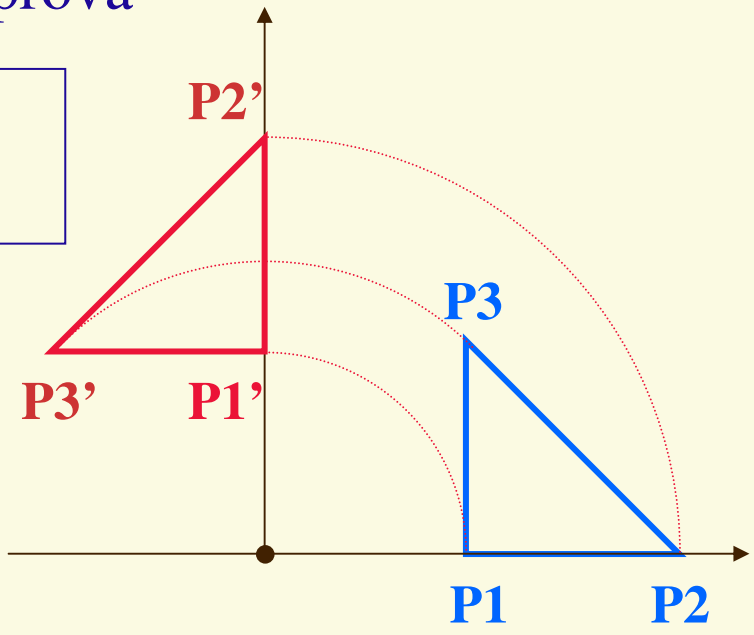
- Prepariamo alcuni semplici casi di prova

dx=0 dy=0
α=π/2




```
P1 = (0, 0)
P2 = (1, 0)
P3 = (0, 1)
```

```
P1' = (0, 0)
P2' = (0, 1)
P3' = (-1, 0)
```




```
P1 = (1, 0)
P2 = (2, 0)
P3 = (1, 1)
```

```
P1' = (0, 1)
P2' = (0, 2)
P3' = (-1, 1)
```

Lezione XIX
Me 7-Nov-2007

Introduzione all'uso di bluej

Pacchetti

Organizzare le classi in pacchetti

- Un programma java è costituito da una raccolta di classi.
- Fin ora i nostri programma erano costituiti da una o al massimo due classi, memorizzate nella stessa directory
- Quando le classi sono tante serve un meccanismo per organizzare le classi: questo meccanismo e' fornito dai pacchetti
- Un pacchetto (package) è costituito da classi correlate
- Per inserire delle classi in un pacchetto si inserisce come prima istruzione del file sorgente la seguente riga

```
package nomePacchetto;
```

Organizzare le classi in pacchetti

- Nella libreria standard il nome dei paccheti è un nome composto da vari token separati del punto
  - java.util
  - javax.swing
  - org.omg.CORBA
- Per usare una classe di un pacchetto, si importa con l'enunciato import che gia' conosciamo:

```
import nomePacchetto.nomeClasse;
```

- L'organizzazione delle classi in pacchetti permette di avere classi diverse, ma con lo stesso nome, in pacchetti diversi, e di poterle distinguere

```
import java.util.Timer;
import javax.swing.Timer;
…
java.util.Timer t = new java.util.Timer();
Javax.swing.Timer ts = new javax.swing.Timer();
```

Organizzare le classi in pacchetti

- Esiste un package speciale, chiamato **pacchetto predefinito**, che e' senza nome.
- Se non programmiamo esplicitamente un enunciato **package** in un file sorgente, le classi vengono inserite nel pacchetto predefinito
- Il pacchetto predefinito si puo' estendere sulle classi il cui codice sorgente appartiene alla stessa directory
- Il package **java.lang** e' sempre importato automaticamente
- Il nome dei package deve essere univoco, come garantirlo?
- Ad esempio invertendo i nomi dei domini web che sono univoci:
  - **ing.unipd.it → it.unipd.ing (Facolta' di Ingegneria)**
- Se non si ha un dominio, si puo' invertire il proprio indirizzo di posta elettronica (univoco!)
  - adriano.luchetta@igi.cnr.it → it.cnr.igi.adriano.luchetta
- Nella java Platform API ci sono esempi di questo tipo
  - package **org.omg.CORBA**

Organizzare le classi in pacchetti

- Se si usano solo le classi della libreria standard non ci si deve preoccupare della posizione dei file
- Un pacchetto si trova in una directory che corrisponde al nome del pacchetto
  - le parti del nome separate da punti corrispondono a directory annidate
  - it.cnr.igi.luchetta.adriano it/cnr/igi/luchetta/adriano
- Se il pacchetto e' usato con un solo programma, si annida il pacchetto all'interno della directory
  - lab5/it/cnr/igi/luchetta/adriano
- Se si vuole usare con molti programmi si crea una directory specifica
  - /usr/home/lib/it/cnr/igi/luchetta/adriano
- Si aggiunge il percorso alla variabile d'ambiente class path

linux bash

  - `$export CLASSPATH=$CLASSPATH:/usr/home/lib:.`
  - `>set CLASSPATH=%CLASSPATH%;c:\home\lib;.`

MS Windows

Consigli pratici

- Inserire ciascuna esercitazione di laboratorio in una directory separata propria come, ad esempio lab1, lab2, …
  - **`$mkdir lab4;`**
  - cosi' in esercitazioni separate potete avere classi con lo stesso nome, che possono pero' essere diverse
- Scegliete una directory di base
  - **/home/ms54637**
  - **`mkdir /home/ms54637/lab4`**
- Mettete i vostri file sorgente nella cartella
  - **/home/ms54637/lab4/Triangolo.java** ⇦ Unix (bash)
  - **/home/ms54637/lab4/ProvaTriangoloJOptionPane.java**
- Compilare i file sorgenti nella directory base
  - **cd home/ms54637**
  - **javac lab4/Triangolo.java**
  - Notare che il compilatore richiede il nome del file sorgente

Consigli pratici

- Inserire nei file sorgenti l'enunciato package
  - **package lab4;**
- Eseguire il programma nella directory base
  - **cd home/ms54637**
  - **java lab4.ProvaTriangoloJOptionPane**
- **Notare che l'interprete vuole il nome della classe, e quindi il separatore sara' il punto**
- **Se non inserite l'enunciato package, dovete eseguire all'interno della directory di lavoro**
  - **cd home/ms54637/lab4**
  - **java ProvaTriangoloJOptionPane**

Argomenti sulla riga comandi

- Quando si esegue un programma Java, è possibile fornire dei parametri dopo il nome della classe che contiene il metodo **main()**

```
$java Program 2 33 Hello
```

- Tali parametri vengono letti dall'interprete Java e trasformati in un array di stringhe che costituisce il parametro del metodo **main()**

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

```
3
33
```

Esercitazione

Uso di array

Esercitazione

- Scrivere la classe TextContainer che memorizzi un testo suddiviso in righe
- L'interfaccia pubblica della classe sia la seguente

```
public class TextContainer
{  // parte privata
     …
  public TextContainer() {…}
  public void add(String aLine){…}
  public String remove() {…}
  public boolean isEmpty() {…}
  public int replace(String find, String replacement) {…}
}
```

Esercitazione

- `public TextContainer()`: costruttore che inizializza un contenitore di testo vuoto
- `public void add(String aLine)`: aggiunge in coda al testo la riga String aLine
- `public String remove()`: restituisce, rimuovendola, la prima riga del testo
- `public boolean isEmpty()`: restituisce true se il contenitore è vuoto, false altrimenti
- `public int replace(String find, String replacement)`: ricerca nel testo la parola `String find` e la sostituisce con la parola `String replacement`. Restituisce il numero di sostituzioni effettuate. Considera una parola come una stringa delimitata da uno o più ***whitespaces***.

Progettare la classe

1. Definire l'interfaccia pubblica
   - L'interfaccia pubblica va definita con molta cura perchè fornisce gli strumenti con cui possiamo elaborare l'informazione contenuta negli oggetti istanziati con la classe
   - Il nome dei metodi deriva generalmente da verbi che descrivono le azioni effettuate dai metodi stessi
   - Nel caso della nostra esercitazione l'interfaccia pubblica è già definita
2. Definire la parte privata
   - Scegliere con cura le variabili di esemplare in cui viene memorizzata l'informazione (i dati) associata agli oggetti della classe
   - Realizzazioni diverse della stessa classe possono avere parti private differenti, ma devono avere la stessa interfaccia pubblica!!!

Progettare la classe: parte privata

- La classe memorizza un testo suddiviso in righe
- Una soluzione semplice è di realizzare la parte privata tramite un **array di riferimenti a stringa** nel quale ciascun elemento è un riferimento a una riga del testo
- L'elemento dell'array di indice zero sarà la prima riga, l'elemento di indice 1 la seconda riga, e così via
- Quanti elementi avrà il nostro array? A priori non sappiamo quante righe dobbiamo memorizzare, quindi realizzeremo un **array riempito parzialmente a ridimensionamento dinamico**

Progettare la classe: parte privata

```
/**
 contenitore di testo. Classe non ottimizzata
 @author Adriano Luchetta
 @version 28-Ott-2004
*/
import java.util.Scanner;
import java.util.NoSuchElementException;
public class TextContainer
{
    //parte privata
    private String[] line;/* riferimento ad
                             array */
    private int lineSize; /* numero di elementi
                             già inseriti
                             nell'array */
```

Commento standard javadoc

Riferimento ad array di String

array riempito solo in parte

Progettare la classe: costruttori

- Il costruttore inizializza le variabili di esemplare
- Generalmente, se non e' noto il numero di elementi da memorizzare nell'array, si inizia con **un elemento**
- Un costruttore che crea un contenitore vuoto fa al nostro caso

```
/**
   costruttore del contenitore di testo
*/
public TextContainer()
{
   line = new String[1];// un elemento!
   lineSize = 0;        // vuoto
}
```

Commento standard javadoc

Allocazione Array!

Numero di elementi già inseriti nell'array

Progettare la classe: metodo add()

```
/**
    inserisce una riga nel contenitore  di testo
    @param aLine la riga da inserire
*/
public void add(String aLine)
{  if (aLine == null) return;

   // Array dinamico: ridimensionare line[]
   if (lineSize >= line.length)
   {  String[] newLine = new String[2 *
         line.length]; //raddoppio elementi
      for (int i = 0; i < line.length; i++)
         newLine[i] = line[i];
      line = newLine;
   }
   // inserimento di String aLine
   line[lineSize] = aLine;
   // incremento del contatore
   lineSize++;
}
```

Verifica delle pre-condizioni

Array dinamico

Metodo isEmpty()

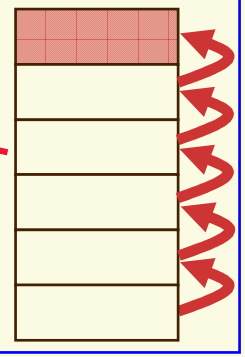
- Metodo predicativo: ritorna un valore di tipo booleano


```
/**
   verifica se il contenitore e' vuoto
   @return true se il contenitore e`
   vuoto, altrimenti false
*/
public boolean isEmpty()
{
   return lineSize == 0;//espress. logica
}
```

Metodo remove()

Commento standard javadoc

```
/**
  estrae la prima riga del testo
  @return la prima riga del testo se il
  contenitore non e' vuoto, altrimenti null
  NB: ad ogni estrazione si ridimensiona
  l'array!
  @throws NoSuchElementException
*/
public String remove()
{  if (isEmpty())
      throw new NoSuchElementException();
   //appoggio temp. dell'elemento di indice 0
   String tmpString = line[0];
   // eliminazione dell'elemento di indice 0
   for (int i = 0; i < lineSize-1; i++)
      line[i] = line[i+1];
   lineSize--; //decremento
   return tmpString;
}
```



Metodo replace()

- Il metodo deve scandire tutte le righe memorizzate per cercare la parola **`String find (`**la classe TextContainer memorizza righe, non singole parole)
- Il metodo deve essere in grado di suddividere le singole righe in parole e di individuare l'eventuale presenza della parola **`String find`**
- Nel caso trovi la parola **`String find`** deve sostituirla con **`String replacement`**, incrementare il contatore delle sostituzioni, inserire nell'array la riga modificata

Metodo replace()

Commento standard javadoc

```
/**
   sostituisce nel testo una parola con
   un'altra
   @param find la parola da sostituire
   @param replacement la parola sostitutiva
   @return il numero di sostituzioni eseguite
   @throws IllegalArgumentException
*/
public int replace(String find, String
  replacement)
{
  if (find == null || replacement == null)
     throw new IllegalArgumentException();

  int count = 0; // contatore parole
                 // sostituite
//continua
```

Verifica delle pre-condizioni

```
    for (int i = 0; i < lineSize; i++)
    {
      Scanner st = new Scanner(line[i]);
      String tmpLine = "";
      boolean found = false;
      while (st.hasNext()) // separazione in token
      {
         String token = st.next();
         if (token.equals(find))
         {
            token = replacement;
            count++; // incremento contatore delle sost.
            found = true;
         }
         tmpLine = tmpLine +  token + " ";
      }
      if (found)
         line[i] = tmpLine;
    }
    return count;
  } // fine metodo
} // fine classe
```

Ciclo for per scandire le righe del testo

Lezione XX
Gi 8-Nov-2007
Esercitazione continua

Una classe di prova

- Programmiamo una classe di prova TextContainerTester che acquisisca da standard input un testo
- Se riceve da riga di comando come argomenti due parole, le usi per ricerca/sostituzione nel testo
- Sostituisca nel testo, eventualmente, la prima parola ricevuta da riga di comando con la seconda
- Invii allo standard output il testo, eventualmente modificato
- Useremo la classe di prova per leggere un testo da file mediante re-indirizzamento dello standard input, modificare il testo e scriverlo su un nuovo file usando il re-indirizzamento dello standard output.

Una classe di prova

```
import java.util.Scanner;

public class TextContainerTester
{ public static void main(String[] args)
  {  //standard input
     Scanner in = new Scanner(System.in);

    //istanza oggetto di classe TextContainer
    TextContainer text = new TextContainer();

    //legge  e stampa il testo da standard input
    System.out.println("\n*** TESTO ORIGINALE ***");
    while(in.hasNextLine())
    { String line = in.nextLine();
      text.add(line);
      System.out.println(line);
    }
    System.out.println("\n***FINE TESTO ORIGINALE***");

// continua
```

Una classe di prova

```
    //sostituzione parola
     int n = 0;
     if (args.length > 1) // se ci sono almeno 2 arg.
     { System.out.println("\n*** TESTO MODIFICATO: ***");
       System.out.println("sostituzione di " + args[0]
         + " con " + args[1]);

       n = text.replace(args[0], args[1]);
     }

     //invio a standard output
     System.out.println();
     while(!text.isEmpty())
       System.out.println(text.remove());

     System.out.println("\n**FINE TESTO MODIFICATO**\n");
     System.out.println("n. " + n + " sost. effettuate");
   }
}
//Fine TextContainerTester
```

Provare una classe

- In alternativa a scrivere la classe di prova TextContainerTester è possibile rendere la classe TextContainer eseguibile, realizzando all'interno il metodo main()

```
import java.util.Scanner;
public class TextContainer // ora la classe e' eseguibile
{ // ... qui va il corpo della classe programmato precedentemente
  public static void main(String[] args)
  {  //standard input
     Scanner in = new Scanner(System.in);
    //istanza oggetto di classe TextContainer
    TextContainer text = new TextContainer();
    //legge  e stampa il testo da standard input
    System.out.println("\n*** TESTO ORIGINALE ***");
    while(in.hasNextLine())
    { String line = in.nextLine();
      text.add(line);
      System.out.println(line);
    }
    System.out.println("\n***FINE TESTO ORIGINALE***");
    //sostituzione parola
    int n = 0;
    if (args.length > 1) // se ci sono almeno 2 arg.
    { System.out.println("\n*** TESTO MODIFICATO: ***");
      System.out.println("sostituzione di "+args[0]+" con "+ args[1]);
      n = text.replace(args[0], args[1]);
    }
    //invio a standard output
    System.out.println();
    while(!text.isEmpty()) System.out.println(text.remove());
    System.out.println("\n**FINE TESTO MODIFICATO**\n");
    System.out.println("n. " + n + " sost. effettuate");
  }
} //Fine TextContainerTester
```

Errori Comuni
Inizializzazione di una variabile

- Le ***variabili di esemplare*** se non sono inizializzate esplicitamente, vengono ***inizializzate automaticamente*** a un valore predefinito
  - ***zero*** per le variabili di tipo numerico e carattere
  - **false** per le variabili di tipo booleano
  - **null** per le variabili oggetto

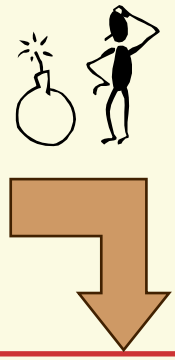
Inizializzazione di una variabile

- Le ***variabili parametro*** vengono inizializzate copiando il valore dei parametri effettivi usati nell'invocazione del metodo
- Le ***variabili locali non*** vengono inizializzate automaticamente, e il compilatore effettua un controllo semantico impedendo che vengano utilizzate prima di aver ricevuto un valore

Inizializzazione delle variabili locali

- Ecco un esempio di errore semantico segnalato dal compilatore per mancata inizializzazione di una variabile

```
int n;     // senza inizializzazione
if (in.hasNextInt())
   n = in.nextInt();
System.out.println("n = " + n);
```



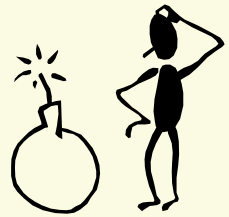
```
variable n might have not been initialized
```

```
int n = 0;                        // OK
if (in.hasNextInt())
   n = in.nextInt();
System.out.println("n = " + n);
```

Errori nell'inizializzazione di array

- Un errore frequente nella inizializzazione di un array e' il seguente

```
double[] data;
data[0] = 1.0;    /* ERRORE! data non e'
                     inizializzato */
```



```
double[] data = new double[100];
data[0] = 1.0;    /* OK */
```

Esercizio: OOP o Array paralleli?

- Scriviamo un programma che riceve in ingresso un elenco di dati che rappresentano
  - i nomi di un insieme di studenti
  - il voto della prova scritta
  - il voto della prova orale
- I dati di ciascun studente vengono inseriti in una riga separati da uno spazio
- prima il nome, poi il voto scritto, poi il voto orale
- I dati sono terminati da una riga vuota
- Il programma esegua poi un ciclo in cui ad ogni iterazione chiede il nome di uno studente e stampa la media dei suoi voti

Soluzione OOP

```
public class Student
{
  private String name;   // nome
  private double wMark; // voto prova scritta
  private double oMark; // voto prova orale

  public Student(String aName,double aWMark,
                 double aOMark)
   {  name  = aName;
      wMark = aWMark;
      oMark = aOMark;
   }

   public String getName() { return name; }
   public double getWMark(){ return wMark;}
   public double getOMark(){ return oMark;}
}
```

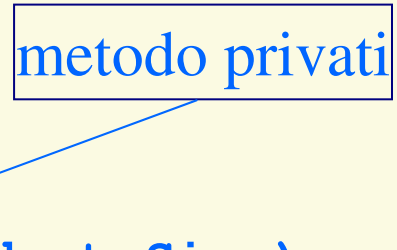
Soluzione OOP

```
import java.util.Scanner;
import java.util.Locale;
import java.util.NoSucheElementException;
public class ExaminationOOP
{ public static void main(String[] args)
  { Student[] students = new Student[10]; // array
    int studentsSize = 0;  // riepito solo in parte

    Scanner in = new Scanner(System.in);

    boolean done = false;
    while (!done)
    {  Student s = readStudent(in);
       if (s == null)
         done = true;
       else
       { if (studentsSize >= students.length)
           students = resize(students, 2 * studentsSize);

         students[studentsSize] = s;
         studentsSize++;
       }
    }
// continua
```

metodi privati

Soluzione OOP

```
    done = false;
    while (!done)
    {   System.out.print("Comando? ");
        int command = in.nextInt();
        if (command == 0)
          done = true;
        else if (command == 1)
        {
          System.out.print("nome? ");
          String name = in.next();
          printAverage(students, name, studentsSize);
        }
        else
          System.out.println("Comando errato");
    }
  } // chiude il metodo main()
// continua
```



```
 /*
   Acquisisce uno studente da flusso di dati
   ritorna null se il flusso di dati e' terminato
 */
 private static Student readStudent(Scanner input)
 { final String END_OF_DATA = "";

   String line = END_OF_DATA;
   if (input.hasNextLine())
     line = input.nextLine();

   if (line.equalsIgnoreCase(END_OF_DATA))
      return null;

   Scanner st = new Scanner(line);
   st.useLocale(Locale.US);

   String nameTmp = st.next();
   double wMarkTmp = st.nextDouble();
   double oMarkTmp = st.nextDouble();
   st.close();
   return new Student(nameTmp, wMarkTmp, oMarkTmp);
 }
// continua
```

```
/*   Stampa la media di uno studente
     @param students l'array di studenti
     @param name nome dello studente
     @param size numero degli studenti
*/
private static void printAverage(Student[] students,
   String name, int size)
{ int i = findName(students, name, size);

  /* gestione dell'errore con lancio di eccezione
  */
  if (i == -1)
     throw new NoSuchElementException("studente" +
        " non trovato");

  double avg = (students[i].getWMark()
                + students[i].getOMark()) / 2;

  System.out.println(name + " " + avg + "\n");
}
// continua
```

Soluzione OOP

```
/* Trova l'indice dell'array corrispondente a uno
   studente
     @param students l'array di studenti
     @param name nome dello studente
     @param count numero degli studenti
     @return l'indice se lo studente e' presente,
             -1 altrimenti
*/
private static int findName(Student[] students,
                            String   name,
                            int size)
{ for (int i = 0; i < size; i++)
    if (students[i].getName().equals(name))
      return i;

  return -1;
}

// continua
```

Soluzione OOP

```
  /* il solito metodo resize()… */
  private static Student[] resize(Student[] oldArray,
                                  int newLength)
  { int n = oldArray.length < newLength ?
            oldArray.length : newLength;

    Student[] newArray = new Student[n];
    for (int i = 0; i < n; i++)
      newArray[i] = oldArray[i];

    return newArray;
  }
} // fine della classe ExaminationtOOP
```

Array Paralleli

```
import java.util.Scanner;
import java.util.Locale;
import java.util.NoSucheElementException;
public class Examination
{ public static void main(String[] args)
  { final String END_OF_DATA = "";

    String line = END_OF_DATA;
    String[] names  = new String[10]; // array
    double[] wMarks = new double[10]; // riempiti
    double[] oMarks = new double[10]; // solo
    int size = 0;                     // in parte

    Scanner in = new Scanner(System.in);
    boolean done = false;
    while (!done)
    { if (in.hasNextLine()) line = in.nextLine();
      if (line.equalsIgnoreCase(END_OF_DATA))
         done = true;
      else
      { if (size >= names.length)
        {  names  = resize(names,  size * 2);
           wMarks = resize(wMarks, size * 2);
           oMarks = resize(oMarks, size * 2);
        }
// continua
```

```
        Scanner st = new Scanner(line);
        st.useLocale(Locale.US);

        names[size] = st.next();
        wMarks[size] = st.nextDouble();
        oMarks[size] = st.nextDouble();
        size++;
      }
    }

    done = false;
    while (!done)
    { System.out.print("Comando: \n- 0 termina il"
        + " programma \n- 1 calcola la media \ncomando? ");
      int command = in.nextInt();
      if (command == 0)
        done = true;
      else if (command == 1)
      { System.out.print("nome?: ");
        String name = in.next();
        printAverage(names, wMarks, oMarks, name, size);
      }
      else
        System.out.println("Comando errato");
    }
  }
// continua
```

Array Paralleli

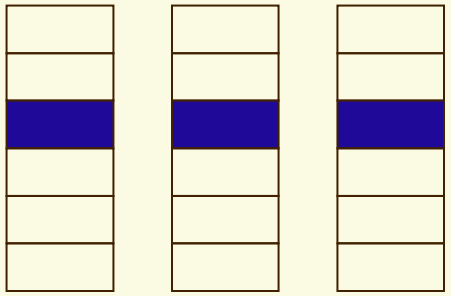
```
  private static void printAverage (String[] names,
                                    double[] wMarks,
                                    double[] oMarks,
                                    String name,
                                    int count)
  {
     int i = findName(names, name, count);
     if (i == -1)
        throw new NoSuchElementException("studente" +
         " non trovato");
     else
     { double avg = (wMarks[i] + oMarks[i]) / 2;
       System.out.println(name + " " + avg);
     }
  }
// continua
```

Array Paralleli

```
private static int findName(String[] names,
                            String name,
                            int count)
{
  for (int i = 0; i < count; i++)
    if (names[i].equals(name))
      return i;
  return -1;
}
```

```
  /* il solito metodo resize()… */
  private static String[] resize( String[] oldArray,
    int newLength)
  {
    int n = oldArray.length < newLength ?
    oldArray.length : newLength;
    String[] newArray = new String[n];
      for (int i = 0; i < n; i++)
        newArray[i] = oldArray[i];
      return newArray;
  }

  /* un altro solito metodo resize()… */
  private static double[] resize( double[] oldArray,
    int newLength)
  {
    int n = oldArray.length < newLength ?
       oldArray.length : newLength;
    double[] newArray = new double[n];
    for (int i = 0; i < n; i++)
      newArray[i] = oldArray[i];
    return newArray;
  }
} // chiude la classe Examination
```

Array paralleli



- L'esempio presentato usa una struttura dati denominata "***array paralleli***"
  - si usano ***diversi array*** per contenere i dati del problema, ma gli array sono tra loro ***fortemente correlati***
    - devono sempre contenere lo ***stesso numero di elementi***
  - ***elementi aventi lo stesso indice*** nei diversi array ***sono tra loro correlati***
    - in questo caso, rappresentano ***diverse proprietà dello stesso studente***
  - molte elaborazioni hanno bisogno di ***utilizzare tutti gli array***, che devono quindi essere passati come parametri come nel caso del metodo **printAverage()**

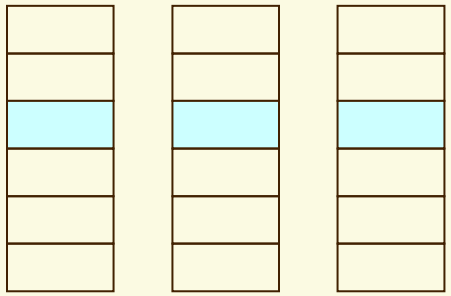
Array paralleli

- Le strutture dati di tipo “array paralleli” sono molto usate in linguaggi di programmazione **non** OOP, ma presentano ***numerosi svantaggi*** che possono essere superati in un linguaggio OOP come Java
  - le modifiche alle dimensioni di un array devono essere fatte contemporaneamente a tutti gli altri
  - i metodi che devono elaborare gli array devono avere una lunga lista di parametri espliciti
  - non è semplice scrivere metodi che devono ritornare informazioni che comprendono tutti gli array
- ***nel caso presentato, ad esempio, non è semplice scrivere un metodo che realizzi tutta la fase di input dei dati, perché tale metodo dovrebbe avere come valore di ritorno i tre array!***

Array paralleli in OOP

- Le tecniche di OOP consentono di gestire molto più efficacemente le strutture dati di tipo “array paralleli”
- Per prima cosa, si definisce una classe che contenga tutte le informazioni relative ad “una fetta” degli array, cioè raccolga tutte le informazioni presenti nei diversi array in relazione a un certo indice

```
class Student
{
  private String name;
  private double wMark;
  private double oMark;
  ...
}
```



Non usare array paralleli

- Tutte le volte in cui il problema presenta una struttura dati del tipo "array paralleli", si consiglia di ***trasformarla in un array di oggetti***
  - occorre realizzare la classe con cui costruire gli oggetti
- Risulta molto più facile scrivere il codice e, soprattutto, apportare modifiche
- Immaginiamo di introdurre un'altra informazione per gli studenti (ad esempio il numero di matricola)
  - nel caso degli array paralleli è necessario ***modificare le firme di tutti i metodi***, per introdurre il nuovo array
  - nel caso dell'array di oggetti **Student**, basta modificare la classe **Student**

Array bidimensionali (cenni)

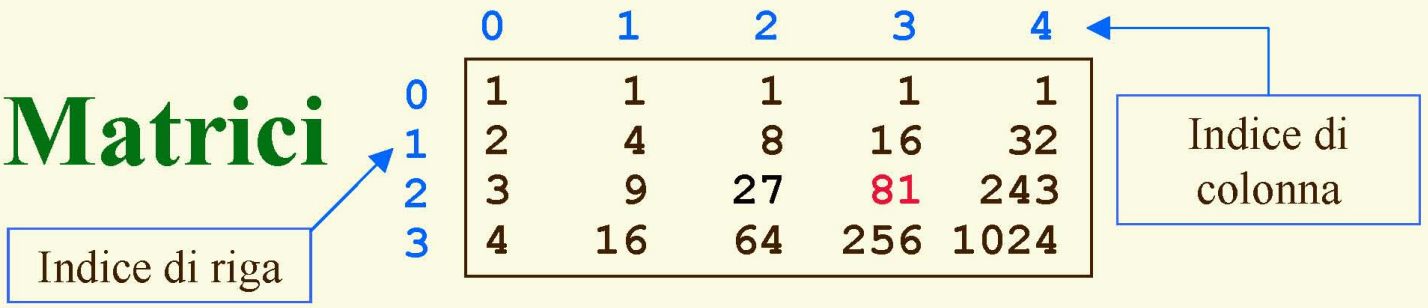
Array bidimensionali

- Problema
  - stampare una tabella con i valori delle potenze x^y, per ogni valore di **x** tra 1 e 4 e per ogni valore di **y** tra 1 e 5

```
1     1     1     1     1
2     4     8    16    32
3     9    27    81   243
4    16    64   256  1024
```

e cerchiamo di risolverlo in modo generale, scrivendo metodi che possano elaborare un'intera struttura di questo tipo.

Matrici



- Una struttura di questo tipo, con dati organizzati in righe e colonne, si dice ***matrice*** o array bidimensionale
- Un elemento a all'interno di una matrice è identificato da ***una coppia (ordinata) di indici***
  - un ***indice di riga***
  - un ***indice di colonna***
  - esempio $a_{2,3} = 81$
- In Java esistono gli array bidimensionali

Array bidimensionali in Java

- Dichiarazione di un array bidimensionale con elementi di tipo **int**

```
int[][] powers;
```

- Costruzione di array bidimensionale di **int** con 4 righe e 5 colonne

```
new int[4][5];
```

- Assegnazione di riferimento ad array bidimensionale

```
powers = new int[4][5];
```

- Accesso a un elemento di un array bidimensionale

```
powers[2][3] = 81;
```

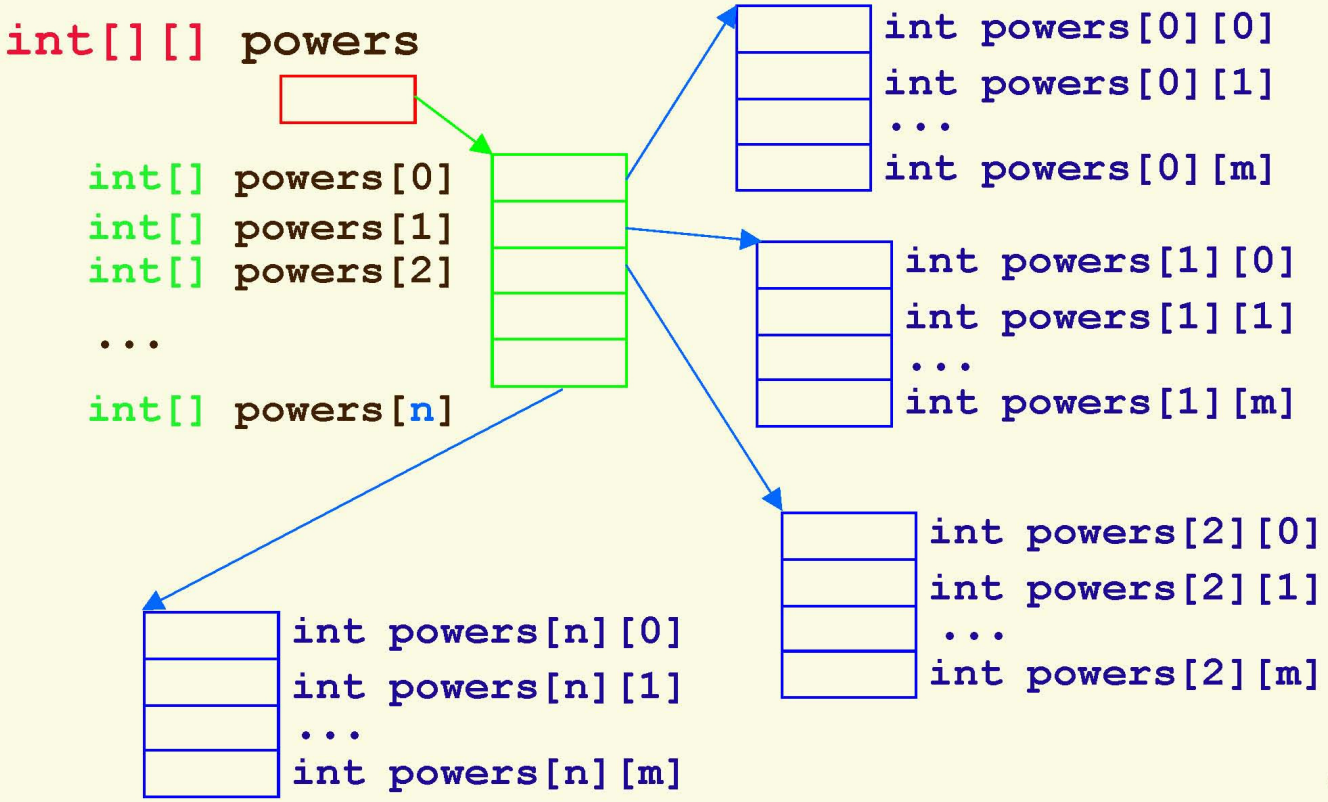
```
int n = powers[2][3];
```

Array bidimensionali in Java

- Ciascun indice deve essere
  - intero
  - maggiore o uguale a 0
  - minore della dimensione corrispondente
- Per conoscere il valore delle due dimensioni
  - il numero di righe è `powers.length;`
  - il numero di colonne è `powers[0].length;`

(perché un array bidimensionale è in realtà un array di array e ogni array rappresenta una colonna…)

Array bidimensionali



- ***int[][] powers*** in realtà è un riferimento a un array di riferimenti ad array di numeri interi interi
- ***int[] powers[k]*** è un riferimento ad un array di interi