Metodi

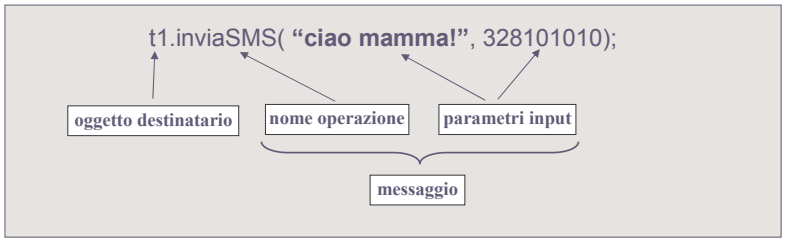
- nozione di metodo
- meccanismi per passare informazioni tra metodi
- meccanismo di invocazione dei metodi
- visibilità degli identificatori
- overloading di metodi

Metodi: considerazioni base (1)

- Ogni metodo dovrebbe eseguire un singolo e ben definito compito e il nome del metodo dovrebbe esprimere in modo chiaro quel compito
- I metodi piccoli che eseguono un compito sono più facili da verificare e correggere dei metodi di grandi dimensioni che eseguono più compiti
- Se non si riesce a scegliere un nome conciso che esprime il compito del metodo, forse il metodo pensato esegue troppi compiti. Di solito è meglio dividere tale metodo in moduli (metodi) più piccoli

Metodi: considerazioni base (2)

- I metodi sono invocati o chiamati a fronte di un messaggio inviato ad un oggetto o classe destinatari. La ricezione del messaggio da parte dell'oggetto destinatario provoca una chiamata a metodo
- Tale chiamata specifica il nome del metodo e fornisce delle informazioni (argomenti) di cui il metodo ha bisogno per completare le attività per cui è stato scritto
- Quando la chiamata di metodo è stata completata il metodo ritorna un risultato al metodo chiamante



Dichiarazione di un metodo (1)

- Esistono molti casi in cui è necessario scrivere metodi che richiedono diversi parametri per eseguire il proprio compito
- Progettiamo una applicazione che usa un metodo definito dal programmatore per determinare il più grande tra 3 numeri reali forniti in input dall'utente
- L'applicazione consta di due classi
  - MaximumFinder: fornisce il servizio ed è composta di due metodi
    - determineMaximum: ottiene tre numeri da tastiera e ne calcola il massimo richiamando il metodo maximum
    - maximum: calcola il massimo dei tre numeri ricevuti come argomenti
  - MaximumFinderTest: collauda l'applicazione usando i metodi della classe MaximumFinder

Dichiarazione di un metodo (2)

- In java esistono vari modi per restituire il controllo al punto in cui un metodo è stato chiamato:
  - Nel caso in cui il metodo non restituisce alcun risultato il controllo viene restituito semplicemente quando viene raggiunta la parentesi graffa che conclude il metodo oppure eseguendo l'istruzione

    return;
  - Nel caso in cui il metodo restituisce un risultato, l'istruzione

    return *espressione*;

    valuta *espressione* e restituisce al chiamante il risultato

Dichiarazione di un metodo (3)

```
// MaximumFinder.java: programma per il massimo di 3 double

import java.util.Scanner;  // importa la classe Scanner

public class MaximumFinder {

   public void determineMaximum( )  {
      Scanner input = new Scanner(System.in);

      System.out.print("Inserisci 3 reali separati da spazi: ");

      double number1 = input.nextDouble( );
      double number2 = input.nextDouble( );
      double number3 = input.nextDouble( );

      double result = this.maximum(number1, number2, number3);

      System.out.printf("Il massimo e' %d\n" , result);
   } //fine metodo determineMaximum

   public double maximum(double x, double y, double z ) {
      double maximumValue = x;
      if ( y > maximumValue) maximumValue = y;
      if ( z > maximumValue) maximumValue = z;
      return maximumValue;
   } //fine metodo maximum

} //fine classe MaximumFinder
```

Il metodo determineMaximum (metodo *chiamante*) invoca il metodo maximum (metodo *chiamato*) che determina il massimo di tre **double** passati come argomenti. Quando maximum termina la propria attività fornisce il risultato che viene restituito nella variabile result

Dichiarazione del metodo maximum che ritorna un valore **double** e che per essere eseguito richiede tre parametri x, y, z di tipo **double**. Quando maximum viene invocato il parametro x è inizializzato al valore dell'argomento number1 (allo stesso modo y e z sono inizializzati ai valori di number 2 e number3, rispettivamente). Quando maximum è stato eseguito e il controllo torna a determineMaximum, i parametri x, y e z non esistono più in memoria

Dichiarazione di un metodo (4)

```
// MaximumFinderTest.java: applicazione per il test di MaximumFinder

public class MaximumFinderTest
{

  public static void main( String args[] )
  {

     MaximumFinder maximumFinder = new MaximumFinder();

     maximumFinder.determineMaximum();

  } // fine del metodo main

} // fine della classe MaximumFinderTest
```

Il metodo main crea un oggetto della classe MaximumFinder denominato maximumFinder

Il metodo main chiama il metodo determineMaximum dell'oggetto maximumFinder per il calcolo del massimo di tre numeri reali

Dichiarazione di un metodo (5)

In java esistono tre situazioni inerenti la chiamata di metodi

1. Chiamare un metodo verso un oggetto della stessa classe. Ad esempio, la chiamata

   `this.maximum(number1, number2, number3)`

   nel metodo `determineMaximum()` della classe `MaximumFinder`

   Nota: è possibile omettere this poiché il compilatore riesce a determinare implicitamente l'oggetto destinatario
2. Chiamare un metodo verso un oggetto di un'altra classe. Esempio

   `maximumFinder.determineMaximum()`

   nel main della classe `MaximumFinderTest`
3. Chiamare un metodo di una classe. Esempio

   `Math.log()`

   per il calcolo del logaritmo

Passaggio argomento: *tipo primitivo*

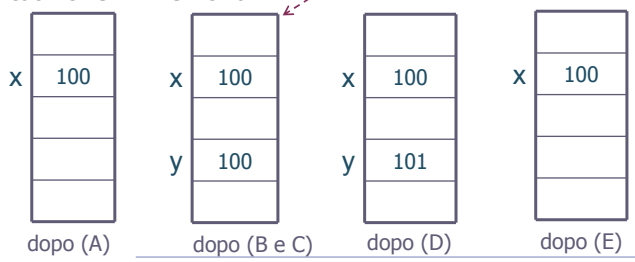
```
public class Test {

   public static void main( String args[] ) {
      int x = 100;                    // (A)
      g(x);                           // (B)
      System.out.println( x );        // (E) stampa 100
   }

   public static void g( int y ) {  // (C)
      y = y+1;                        // (D)
   }
} // fine classe Test
```

y = x

situazione in memoria:



dopo (A) dopo (B e C) dopo (D) dopo (E)

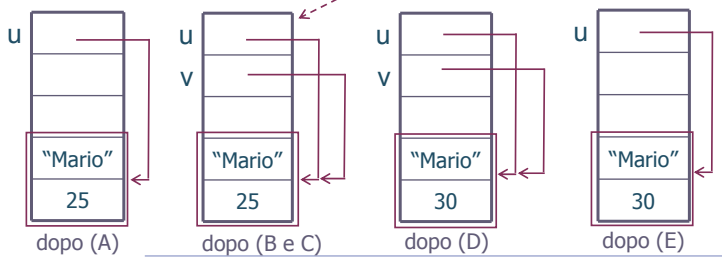
Passaggio argomento: *tipo riferimento*

```
public class Utente {
     private String nome;
     private int eta;
     Utente ( String n,  int e ) {
              nome = n;
              eta = e;
     }
     public void setEta( int e ) { eta = e; }
     public void print( ) { nome + " " + eta }
}
```

```
public class Test  {
   public static void main(String [] args) {
      Utente u = new Utente( "Mario", 25);  // (A)
      g(u);                                 // (B)
      u.print( );                           // (E) Mario 30
   }
   static void g(Utente v) {                // (C)
       v.setEta(30);                        // (D)
   }
}
```

v = u

situazione in memoria:



dopo (A) dopo (B e C) dopo (D) dopo (E)

Visibilità delle dichiarazioni (1)

- Le dichiarazioni introducono dei nomi che possono essere usati per riferirsi a delle entità (classi, variabili,…)
- La visibilità di un dichiarazione è la porzione di programma che può riferirsi ad una entità tramite il suo nome
- Le regole base della visibilità sono le seguenti
  - La visibilità di una dichiarazione di parametri è il corpo del metodo in cui la dichiarazione è presente
  - La visibilità di una dichiarazione di variabile locale parte dal punto in cui è presente la dichiarazione e arriva fino al termine del blocco
  - La visibilità di una dichiarazione di variabile locale che si trova nella sezione di inizializzazione di una istruzione **for** è il corpo dell'istruzione **for** stessa
  - La visibilità di un metodo o campo di una classe è l'intero corpo della classe

Visibilità delle dichiarazioni (2)

Esempio:

```
int a = 1;
for (int b = 0; b < 3; b++){
    int c = 1;
    for (int d = 0; d <3; d++){
        if (c < 3) c++;
    }
    System.out.print(c);
    System.out.println(b);
}
a = c; // ERROR! c is out of scope
```

abcd

abc

a

Visibilità delle dichiarazioni (3)

Visibilita.java | VisibilitaTest.java

```
1  // Visibilità degli elementi di una classe
2
3  public class Visibilita {
4     private int x = 1;  // attributo accessibile da tutti i metodi
5
6     public void begin( ) {
7
8        int x = 5;    // variabile locale al metodo
9        System.out.println( "Var. x del metodo begin(): " +  x );
10
11       useLocalVar( );
12       useAttribute( );
13       useLocalVar( );
14       useAttribute( );
15
16       System.out.printf( "\n Var. x del metodo begin(): %d\n", x );
17    }
18
19    public void useLocalVar( ) {
20       int x = 25; // inizializzata ogni volta che useLocalVariable viene chiamata
21
22       System.out.println( "Var. locale x all'inizio di useLocalVar: " + x );
23       x = x + 1;
24       System.out.println( "Var locale x alla fine di useLocalVar: " + x );
25    }
26
27    public void useAttribute() {
28       System.out.println( "Attrib. x all'inizio di useAttribute: " + x );
29       x =  x *10;   // corrisponde a this.x = this.x * 10
30       System.out.println( "Attrib. x alla fine di useAttribute: " + x );
31    }
32 }
```

Il metodo begin() crea ed inizializza la variabile locale x e chiama i metodi useLocalVar() e useAttribute()

Il metodo useLocalVar() crea e inizializza la variabile locale x durante ogni chiamata

Il metodo useAttribute() modifica l'attributo x della classe ad ogni chiamata

Visibilità delle dichiarazioni (4)

Visibilita.java | VisibilitaTest.java

```
1  // VisibilitaTest.java
2
3  public class VisibilitaTest {
4
5     public static void main( String args[] )
6     {
7
8        Visibilita v = new Visibilita();
9
10       v.begin();
11
12    }
13 }
```

Il metodo main crea un oggetto di classe Visibilita denominato **v**

Il metodo main chiama il metodo begin dell'oggetto **v** per il test della classe Scope

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Var. x del metodo begin(): 5
Var. locale x all'inizio di useLocalVar: 25
Var locale x alla fine di useLocalVar: 26
Attrib. x all'inizio di useAttribute: 1
Attrib. x alla fine di useAttribute: 10
Var. locale x all'inizio di useLocalVar: 25
Var locale x alla fine di useLocalVar: 26
Attrib. x all'inizio di useAttribute: 10
Attrib. x alla fine di useAttribute: 100

 Var. x del metodo begin(): 5
Press any key to continue...
```

Overloading (1)

- Java permette di dichiarare più metodi con lo stesso nome all'interno di una classe a patto che questi metodi abbiano insiemi di parametri diversi (in numero e/o tipo)
- Questa tecnica prende il nome di overloading o sovraccarico dei metodi
- Quando viene chiamato un metodo sovraccarico il compilatore Java seleziona quello corretto esaminando il numero e il tipo degli argomenti della chiamata
- Il sovraccarico dei metodi viene usato quando si ha la necessità di avere metodi che svolgono attività simili. Ad esempio il metodo min della classe Math è sovraccarico ed ha quattro versioni a seconda che i parametri siano double, float, int o long

Overloading (2)

```
1  // Dichiarazione di metodi sovraccarichi
2
3
4  public class MethodOverload
5  {
6     // test dei metodi square overloaded
7     public void testOverloadedMethods()
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // fine metodo testOverloadedMethods
12
13    // metodo square con argomento int
14    public int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // fine metodo square con argomento int
20
21    // metodo square con argomento double
22    public double square( double doubleValue )
23    {
24       System.out.printf( "\nCalled square with double argument: %f\n",
25          doubleValue );
26       return doubleValue * doubleValue;
27    } // fine metodo square con argomento double
28 } // fine classe MethodOverload
```

Chiama il metodo **square** con argomento **int**

Chiama il metodo **square** con argomento **double**

Dichiara il metodo **square** con parametro **int**

Dichiara il metodo **square** con parametro **double**

Overloading (3)

```
1  // MethodOverloadTest.java
2  // Applicazione per il test della classe MethodOverload
3
4  public class MethodOverloadTest
5  {
6     public static void main( String args[] )
7     {
8        MethodOverload methodOverload = new MethodOverload();
9        methodOverload.testOverloadedMethods();
10    } // fine main
11 } // fine classe MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```


Overloading (4)

```
1  // MethodOverloadError.java
2  // Metodi sovraccarichi con firme identiche
3  // causano errori di compilation, anche se i tipi di ritorno sono diversi
4
5  public class MethodOverloadError
6  {
7     // dichiarazione del metodo square con argomento int
8     public int square( int x )
9     {
10       return x * x;
11    }
12
13    // seconda dichiarazione del metodo square con argomento int
14    // causa un errore di compilazione anche se i tipi di ritorno sono diversi
15    public double square( int y )
16    {
17       return y * y;
18    }
19 } // fine classe MethodOverloadError
```

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Errore di compilazione

Overloading di costruttori

- Il costruttore di una classe specifica come un oggetto di quella classe deve essere inizializzato
- Il costruttore di una classe può essere sovraccarico al pari di qualsiasi altro metodo
- I costruttori sovraccarichi permettono di inizializzare gli oggetti di una classe in modi diversi
- Per sovraccaricare un costruttore basta fornire più dichiarazioni di costruttori dotate di diverse firme
- Mostriamo la nozione di costruttore sovraccarico attraverso la classe Punto che contiene 2 costruttori sovraccarichi che garantiscono che ogni oggetto inizi la sua esistenza sempre in uno stato coerente

La classe *Punto*

Punto.java | PuntoTest.java

```
1  // Punto.java
2  // Questa classe permette di modellare punti del piano Euclideo a coordinate
3  // intere. Ammessi solo punti a ccordinate positive
4
5  public class Punto
6  {
7     // attributi per memorizzare le coordinate
8     private int x;
9     private int y;
10
11    // costruttore di default
12    public Punto() {
13       this.x = 0;
14       this.y = 0;
15    }
16
17    // costruttore
18    public Punto( int xCoord, int yCoord ) {
19
20          this.x = (xCoord >= 0) ? xCoord : 0;
21          this.y = (yCoord >= 0) ? yCoord : 0;
22    }
23
24    public double dist( Punto p ) {
25
26        int a = (p.x - this.x) * (p.x - this.x) + (p.y - this.y) * (p.y - this.y);
27        return Math.sqrt( a );
28    }
29
30    // conversione a stringa nel formato (x,y)
31    public String toString() {
32
33       return "(" + x + "," + y + ")";
34    }
35
36 } // fine classe Punto
```

costruttore di default

costruttore generico con validazione dati

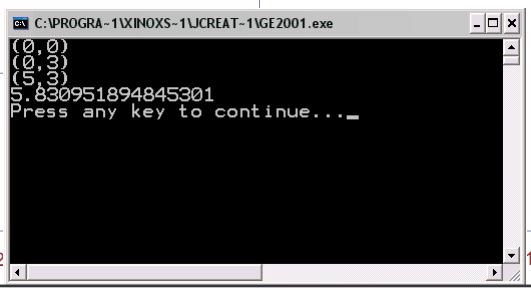
sintassi compatta per if () ... else ...

metodo statico della classe Math per il calcolo della radice quadrata

La classe *PuntoTest*

Punto.java | PuntoTest.java

```
1  // PuntoTest.java
2
3  public class PuntoTest {
4
5      public static void main( String args[] )
6      {
7
8          Punto p1 = new Punto();
9          Punto p2 = new Punto(-2,3);
10         Punto p3 = new Punto(5,3);
11
12         System.out.println( p1 );
13         System.out.println( p2 );
14         System.out.println( p3 );
15
16         System.out.println( p1.dist( p3 ) );
17
18     }
19
20 }
```



Costruttori di default con e senza argomenti

- Ogni classe deve avere almeno un costruttore
- Se non viene fornito un costruttore, il compilatore ne crea uno di default che ha zero argomenti ed inizializza le variabili di istanza assegnando dei valori sulla base dei loro tipi (0 per i tipi numerici primitivi, **false** per il tipo **boolean**, **null** per i riferimenti)
- Se viene fornito un costruttore, il compilatore non crea alcun costruttore di default
- Per specificare un'inizializzazione di default bisogna definire un costruttore senza argomenti

Garbage collection

- Ogni oggetto creato all'interno di un programma Java usa svariate risorse di sistema
- Bisogna disciplinare il modo in cui tali risorse vengono restituite al sistema stesso una volta che non sno più necessarie
- La JVM esegue automaticamente un processo chiamato garbage collection il cui scopo è quello di trovare tutti gli oggetti presenti in memoria e non più in uso e di restituire al sistema la memoria non più utilizzata da questi oggetti
- Non vi è alcuna garanzia che il garbage collector venga eseguito in un particolare momento

Membri static di una classe

- metodi `static` (o metodi di classe)
  - applicati alla classe invece che ad uno specifico oggetto della classe
  - chiamata ad un metodo `static` : `ClassName.methodName( arguments )`
  - es,. tutti i metodi della classe `Math` sono `static`
    - `Math.sqrt( 900.0 )`
- attributi `static` (o attributi di classe)
  - attributi appartenenti alla classe
  - esiste una sola copia
- `Math.PI` è un attributo statico della classe `Math`
- `main` è dichiarato `static` in modo che può essere invocato senza dover creare alcun oggetto

Esempio d'uso di file

FindMinFromFile.java

```
1  // programma per la ricerca del minimo di interi
2  // prelevati da file
3
4  import java.util.*;
5  import java.io.*;
6
7  public class FindMinFromFile {
8     public static void main( String args[] ) throws IOException {
9
10       FileReader fileIn = new FileReader("dati.txt");
11
12       Scanner input = new Scanner(fileIn);
13
14       int dato;
15       int minimo = Integer.MAX_VALUE;
16
17       while ( input.hasNextInt() ) {
18
19            dato = input.nextInt();
20            if ( dato < minimo  )
21                 minimo = dato;
22       }
23
24       String r = Integer.toString(minimo);
25       FileWriter fileOut = new FileWriter("risultato.txt");
26
27       fileOut.write("Il minimo nel file dati.txt e': ");
28       fileOut.write( r );
29       fileOut.close();
30    }
31
32 }
```