Costruire le Classi, e la Gestione dei File

Lezione 8

Scopo della Lezione

- Imparare a costruire delle classi su esempi;
- Introduzione delle classi per la gestione dei file;

File

Nome fuorviante! – rappresenta il *percorso* di un file o una directory (che possono non esistere!)

Consiste di una sequenza di nomi separati da **File.separator** o **File.separatorChar** (es: "/" o "\")

Quando rappresenta una directory i metodi **list()** e **list(FilenameFilter)** restituiscono la lista dei files contenuti come **String[]**

Permette di conoscere le proprietà del file:

- canRead()/canWrite()
- exists()
- isDirectory()/isFile()/isHidden()
- …

File: Lista dei File

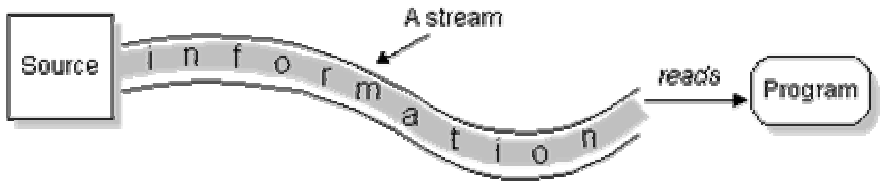

```
import java.io.File;
import java.util.Date;
import prog.io.*;

class Ls {
  public static void main(String[] args) {
    ConsoleOutputManager video = new ConsoleOutputManager();
    File f1 = new File(args[0]), f;
    String[] ls = f1.list();
    if (ls != null)
      for (int i = 0; i < ls.length; i++) {
        f = new File(f1, ls[i]);
        video.print(f.isDirectory() ? "d" : "-");
        video.print((f.canRead() ? "r" : "-") + (f.canWrite() ? "w" : "-"));
        video.print("\t"+f.length()+"\t");
        video.print(new Date(f.lastModified()));
        video.print('\t');
        video.println(f.getName());
      }
  }
}
```

File: Lista dei File

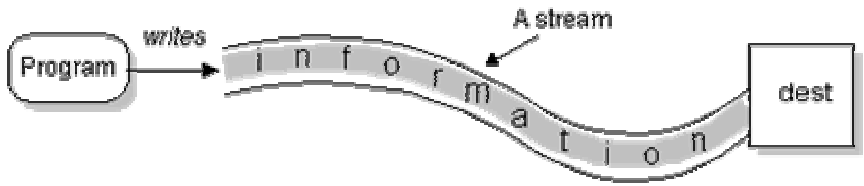

```
[16:02]cazzola@ulik:esercizi>java Ls ShapeHierarchy
-rw     945     Thu Nov 27 22:20:56 CET 2003    Rectangle.class
drw     568     Thu Nov 27 22:21:58 CET 2003    package-doc
-rw     751     Thu Nov 27 22:20:56 CET 2003    Circle.class
-rw     429     Thu Nov 27 22:20:56 CET 2003    Shape.class
-rw     597     Thu Nov 27 22:20:56 CET 2003    Square.class
-rw     2549    Thu Nov 27 22:23:12 CET 2003    ShapeHierarchy.jar
-rw     1017    Thu Nov 27 22:17:14 CET 2003    Rectangle.java
-rw     514     Thu Nov 27 22:17:14 CET 2003    Circle.java
-rw     358     Thu Nov 27 22:17:14 CET 2003    Square.java
-rw     977     Thu Nov 27 22:17:14 CET 2003    Shape.java
```

Streams

Input streams (legge da: file, memoria, socket, etc.)

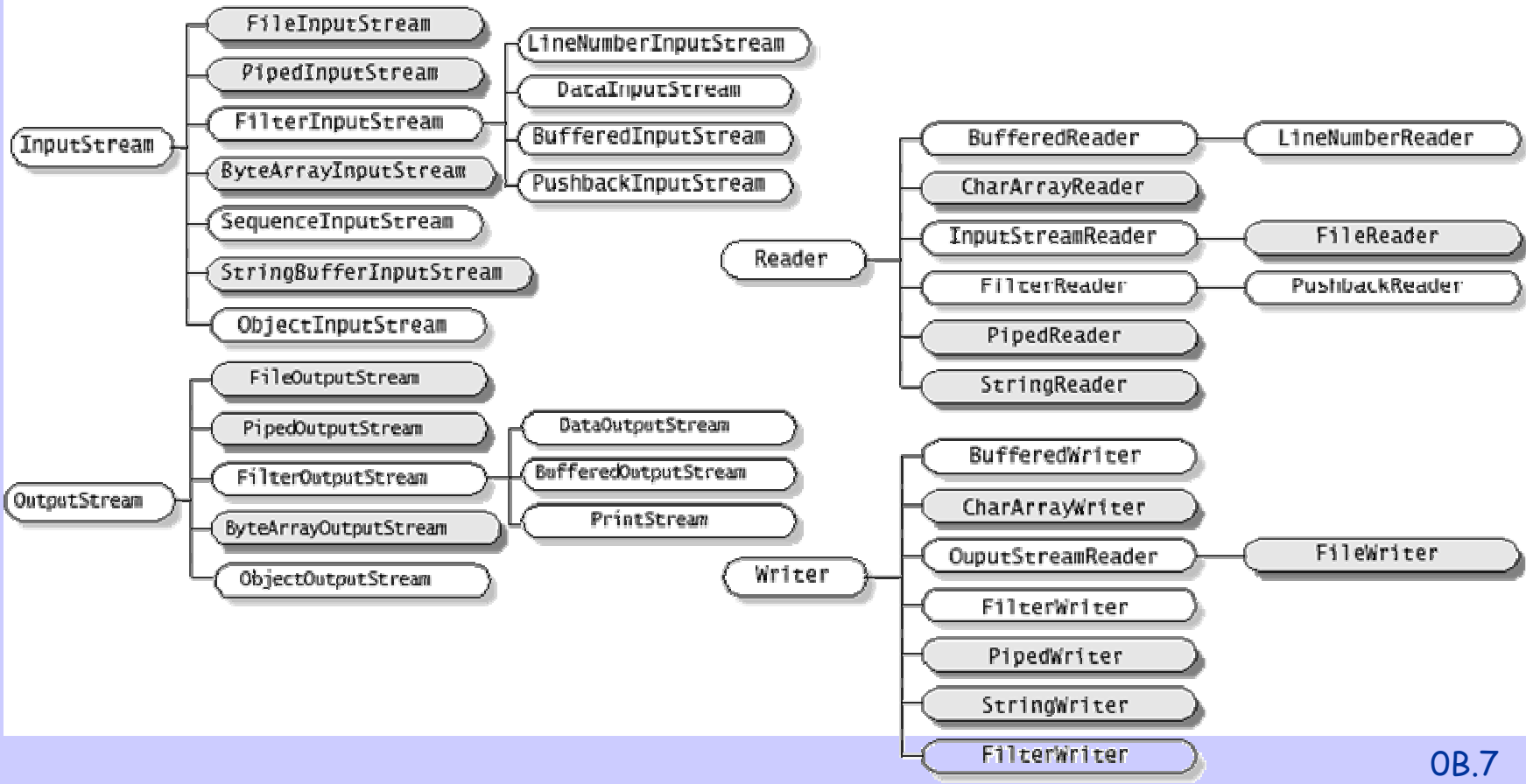


Output streams (scrive su: file, memoria, socket, etc.)



Byte e Character Streams

Input/Output-Streams: I/O orientato ai byte
Reader/Writer: I/O orientato ai caratteri (a 16 bit!)



Funzionalità delle Classi Astratte

InputStream:

- int read()
- int read(byte cbuf[])
- int read(byte cbuf[], int off, int len)
- void close()

OutputStream

- void write(byte[] b)
- void write(byte[] b, int off, int len)
- void write(int b)
- void close()
- void flush()

Reader:

- int read()
- int read(char cbuf[])
- int read(char cbuf[], int off, int len)
- void close()

Writer

- void write(char[] cbuf)
- void write(char[] cbuf, int off, int len)
- void write(int c)
- void write(String str)
- void write(String str, int off, int len)
- close()
- flush()

byte stream → char stream:
OutputStreamWriter(OutputStream out)
InputStreamReader(InputStream in)

Gestione di uno stream

Lo stream viene aperto automaticamente quando viene creato l'oggetto;

Si leggono/scrivono i dati;

Lo stream viene chiuso esplicitamente chiamando il metodo **close()** o implicitamente dal *GC*;

Quasi tutti i metodi/costruttori possono lanciare eccezioni di Input/Output: IOException

Es.: Copia File

Scrivere la classe **CopyBytes** che copia un file in un altro.

```
import java.io.*;

public class CopyBytes {
  public static void main(String[] args) throws IOException {
     FileInputStream in = new FileInputStream(args[0]);
     FileOutputStream out = new FileOutputStream(args[1]);
     int c;
     while ((c = in.read()) != -1) out.write(c);
     in.close();
     out.close();
   }
}
```

Es.: Copia File


```
[17:32]cazzola@ulik:esercizi>echo "Nel mezzo del cammin di
    nostra vita mi ritrovai in una selva oscura che la diritta via
    era smarrita e quanto a dir qual era è cosa dura esta selva
    selvaggia ed aspra forte che nel pensier rinnova la paura." >
    input
[17:34]cazzola@ulik:esercizi>java CopyBytes input output
[17:34]cazzola@ulik:esercizi>cat output
Nel mezzo del cammin di nostra vita mi ritrovai in una selva
    oscura che la diritta via era smarrita e quanto a dir qual
    era è cosa dura esta selva selvaggia ed aspra forte che nel
    pensier rinnova la paura.
```

Filter Streams

Sottoclassi di FilterInputStream e FilterOutputStream.

Incapsulano un altro stream e ne "filtrano" i dati

- DataInputStream e DataOutputStream
- BufferedInputStream e BufferedOutputStream
- PushbackInputStream
- PrintStream

Es.

```
DataOutputStream out =
    new DataOutputStream(new FileOutputStream("file.txt"));
out.writeDouble(3.14);
out.writeChar('\t');
out.writeInt(42);
```

Standard Streams

Tre campi di System:

- InputStream in
- PrintStream out
- PrintStream err

Es: System.out.println("ciao!");

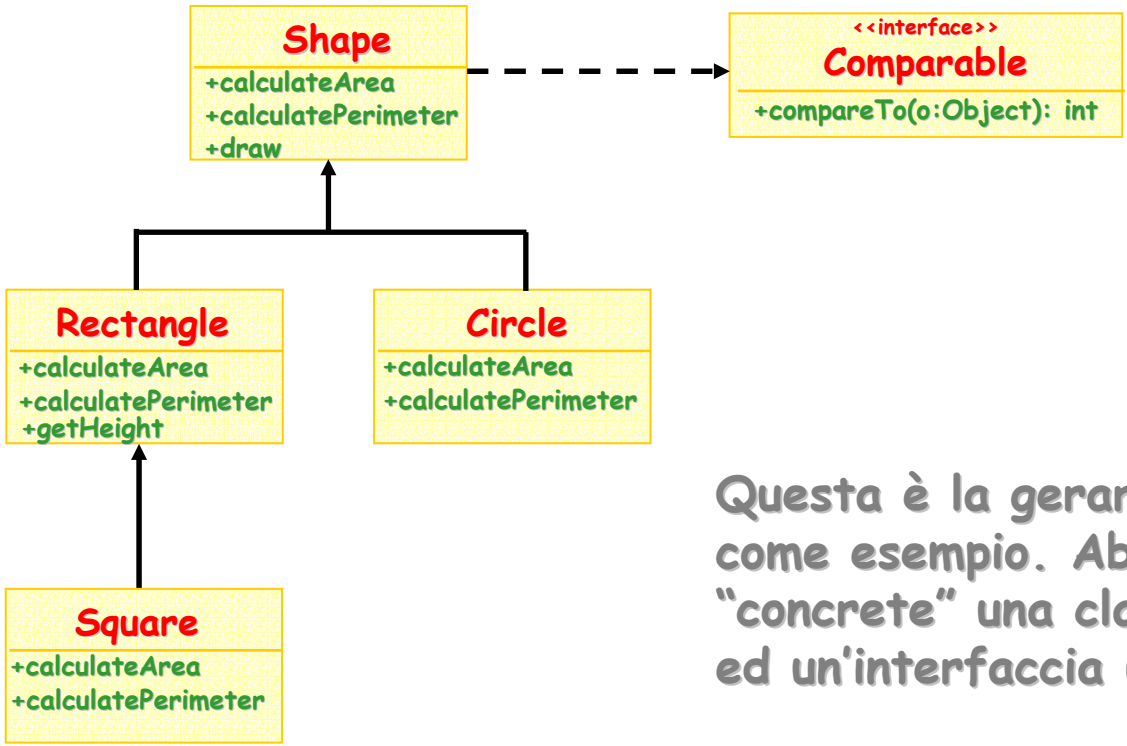
Modificabili tramite tre metodi (setIn, setOut, setErr) della classe System

Esempio: Lettura da Standard Input

```
BufferedReader input = new BufferedReader(new
   InputStreamReader(System.in)) ;

Integer objI = Integer.valueOf(input.readLine()) ;
int i = Integer.parseInt(input.readLine()) ;

System.out.println("objI="+objI+", i="+i) ;
System.out.println(objI.getClass()) ;
```

… metodi analoghi per gli altri tipi primitivi (e relativi Wrapper)

Gerarchia di Figure Estesa



Questa è la gerarchia delle classi usata come esempio. Abbiamo tre classi "concrete" una classe astratta (**Shape**) ed un'interfaccia (Comparable).

Nel seguito entremo nel merito di come queste classi debbano essere scritte allo scopo di imparare a scriverne altre.

Classi in Java

Esempio di definizione di Classe:

```
public class Rectangle extends Shape { // Header della Classe
    // Corpo della classe
}
```

In Generale:

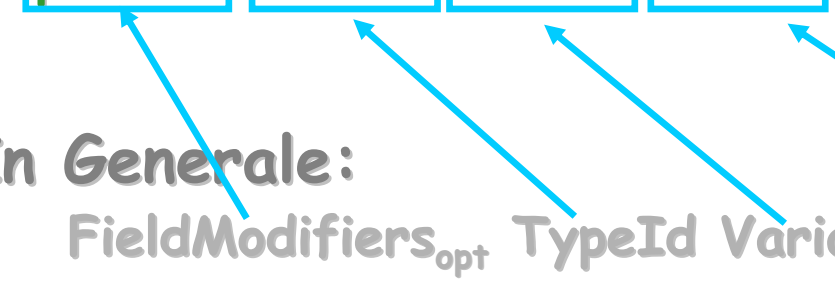
$[\text{ClassModifiers}]_{opt}$ **class** ClassName
[extends ParentName$]_{opt}$ **[implements** InterfaceList$]_{opt}$

- La keyword **extends** definisce le regole di ereditarietà;
- La keyword **implements** identifica le interfacce da implementare;
- I ClassModifiers indicano le modalità di accesso (**public**, **private**, etc.) o le caratteristiche (es. **abstract**) che identificano la class.

Attributi o Variabili d'Istanza

Esempi:

```
private double height = 0.0;    // variabili d'istanza
private double width  = 0.0;
```



In Generale:

$FieldModifiers_{opt}$ TypeId VariableId $Initializer_{opt}$

Gli attributi hanno visibilità a livello di classe, quindi possono essere usati ovunque all'interno della classe.

Gli attributi possono essere:

- public,
- private,
- protected

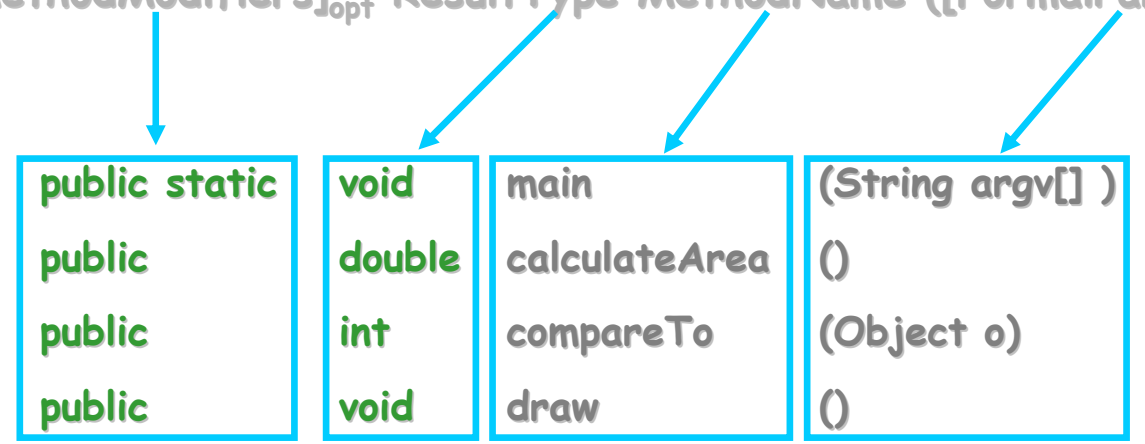
Definizione di Metodo

Esempio:

```
public double calculateArea() {
        return height * width;
    } // calculateArea
```

L'header del metodo

$[MethodModifiers]_{opt}$ ResultType MethodName $([FormalParameterList]_{opt})$



Classe Rectangle


```
public class Rectangle extends Shape {
    private double height;   // variabili d'istanza
    private double width;

    public Rectangle(double h, double w) { // costruttore
        height = h; width = w;     ↖ Il costruttore non ha un tipo di ritorno esplicito
    } // costruttore di Rectangle

    public double getHeight() {return height;}
    public double getWidth() {return width;}
    public void setHeight(double h) {height = h;}
    public void setWidth(double w) {width = w;}
    public void draw(){
      System.out.println("I'm a Rectangle! My sides are: "+height+","+width);
    }
    public double calculateArea() {
        return height * width;
    } // calculateArea
    public double calculatePerimeter() {
        return 2 * (height+width);
    }                          ↖ height e width sono variabili private eppure le posso usare
} // classe Rectangle
```

Classe Rectangle (Alternativa)


```
public class Rectangle extends Shape {
    private double height;    // variabili d'istanza
    private double area;  ← cambio la rappresentazione interna, ma …

    public Rectangle(double h, double w) { // costruttore
        height = h; area = h*w;     ↖ … mantengo la stessa interfaccia.
    } // costruttore di Rectangle

    public double getHeight() {return height;}
    public double getWidth() {return area/height;}
    public void setHeight(double h) {area /= height; height = h; area *= height;}
    public void setWidth(double w) {area = height*w;}
    public void draw(){
      System.out.println("I'm a Rectangle! My sides are: "+height+","+this.getWidth());
    }
    public double calculateArea() {
        return area;
    } // calculateArea
    public double calculatePerimeter() {
        return 2 * (height+this.getWidth() );
    }                              ↖this rappresenta l'oggetto su cui il metodo è stato invocato.
} // classe Rectangle
```

Accesso Pubblico/Privato

Le variabili d'istanza solitamente sono dichiarate private. Questo le rende inaccessibili dagli altri oggetti.

Dei metodi pubblici forniranno un limitato e controllato accesso agli attributi privati (es. getWidth() e setWidth()).

I metodi pubblici di un oggetto costituiscono la sua interfaccia, cioè quella parte accessibile agli altri oggetti. Da non confondere con le interface.

Accesso Pubblico/Privato

Nota: l'uso di attributi pubblici può condurre ad uno stato inconsistente.

Es.: consideriamo la seconda implementazione di **Rectangle** con **area** e **height** definiti come attributi pubblici.

In questo caso è lecito scrivere:

```
r.height = 25;
```

Ma senza aggiornare di conseguenza l'area, avremo che il metodo **calculateArea()** mi ritornerà un volore incosistente.

La consistenza viene mantenuta usando **setHeight()**.

Progettazione dei Metodi

- I metodi pubblici svolgono il ruolo di interfaccia della classe.
  - Se un metodo è pensato per passare informazioni ad un altro oggetto, allora **DEVE** essere dichiarato pubblico.
- I metodi definiti in una classe hanno visibilità a livello di classe. Indipendentemente dai diritti d'accesso possono essere usati ovunque all'interno della classe.
- I metodi che non ritornano dei valori devono essere dichiarati con tipo di ritorno **void**.

Parametri Attuali e Formali

I parametri attuali rappresentano i valori passati ai metodi al momento dell'invocazione.

```
Rectangle r1 = new Rectangle(3,7);
r1.setHeight(3*r1.getWidth());
```

Parametri Attuali: sono espressioni che coinvolgono variabili, costanti ed altre espressioni.

Il tipo ed il numero dei parametri attuali deve corrispondere al tipo ed al numero dei parametri formali usati nella definizione del metodo invocato.

```
r1.setHeight("ciao")
```

Errore **setHeight()** richiede un argomento di tipo **double**;

Regole di Accesso

I package contengono classi che contengono membri (metodi e attributi).

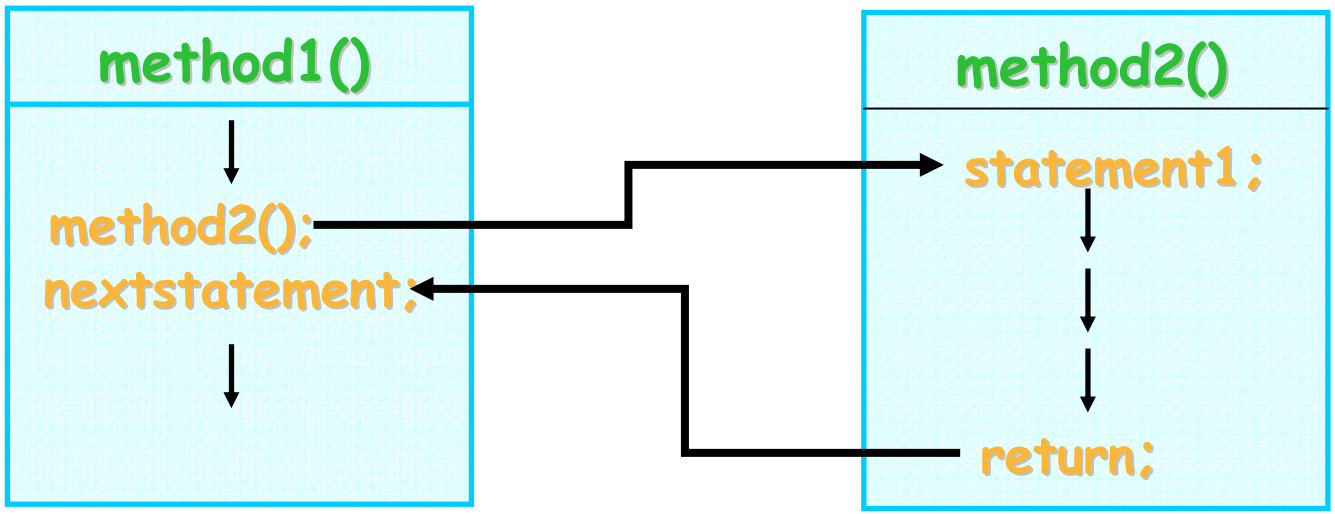
Gli accessi sono determinati dall'elemento più esterno a quello più interno. Se non è specificato nessun accesso, il valore di default è utilizzato.

Tipo di Entità	Dichiarazione	Regole di Accesso
Package	N/A	L'accesso è determinato dal sistema.
Classe	public	Accessibile se il package che la contiene è accessibile.
	default	Accessibile solo entro il package che la contiene.
Membro (campo o metodo) di una classe accessibile	public	Accessibile a tutti gli altri oggetti.
	protected	Accessibile alle classi nel package ed alle sottoclassi.
	private	Accessibile solo entro la classe.
	default	Accessibile solo entro il package.

Flusso di Esecuzione

La chiamata di un metodo trasferisce il flusso di controllo alla prima istruzione del metodo chiamato.

L'istruzione **return** o la fine del corpo del metodo invocato ritorna il controllo all'istruzione chiamante.



Progettazione OO: Principi

Encapsulation: La classe **Rectangle** incapsula uno stato e le azioni che manipolano tale stato.

Information Hiding: Lo stato delle istanze di **Rectangle** è privato.

Interfaccia: I metodi pubblici, **getHeight()**, **setHeight()**, …, e **calculatePerimeter()**, limitano le possibilità di utilizzo delle istanze di **Rectangle**.

Estensibilità: Le funzionalità di Rectangle possono essere estese facilmente (vedi classe **Square**).

Classe Square


```
public class Square extends Rectangle {
  public Square (double side) {
    super(side, side); // costruttore del padre
  }
  public void draw(){
    System.out.println("I'm a Square! My side is: "+getHeight());
  }
} // Square
```

↖ riferisce la classe padre e permette di invocarne i metodi

← metodo di Rectangle ridefinito

↑ invocazione di un metodo ereditato

Square estende le funzionalità di **Rectangle**. Tutti i metodi egli attributi definiti ed implementati in **Rectangle**, se non ridefiniti, sono ereditati tali e quali dalla classe **Square**.

Quando un metodo con lo stesso prototipo è presente sia nella classe figlia che nella classe padre il metodo della classe figlia RIDEFINISCE il metodo della classe padre (es. **draw()**).

Classe Circle


```
public class Circle extends Shape {
    private double ray;    // variabili d'istanza

    public Circle(double r) { // costruttore
        ray = r;
    }

    public void draw(){System.out.println("I'm a Circle! My ray is: "+ray);}
    public double calculateArea() {
      return Math.PI*ray*ray;
    } // calculateArea
    public double calculatePerimeter() {
      return 2*Math.PI*ray;
    }

}
```

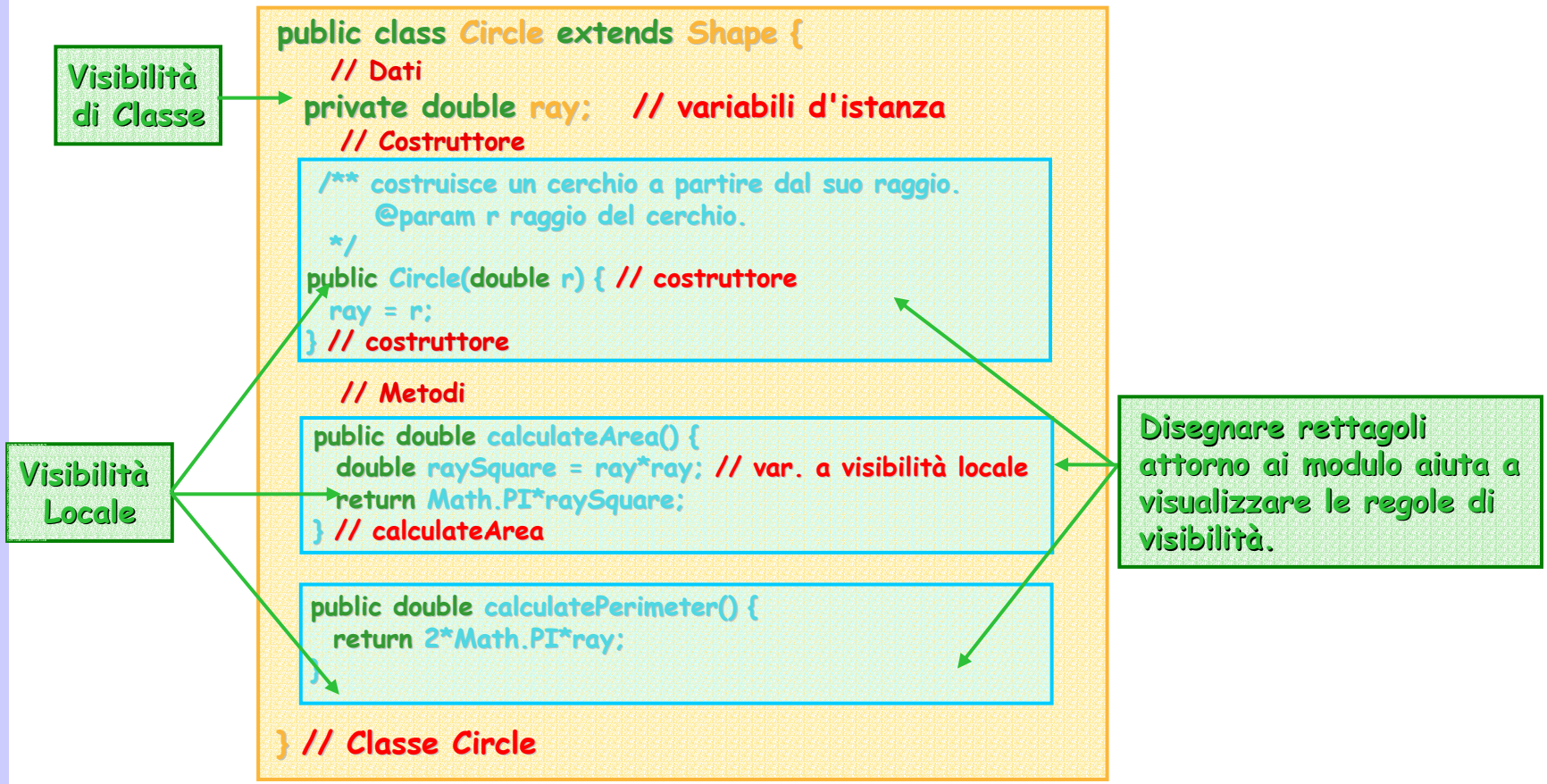
Regole di Visibilità

La visibiltà di un identificatore specifica dove, quell'identificatore, può essere usato all'interno del programma.

- **Visibilità locale**: la visibilità di un parametro è limitata al metodo in cui è dichiarato;
- **Visibilità di classe**: una variabile d'istanza può essere usata ovunque all'interno della classe.

Disegnare rettangoli attorno ai moduli del programma è un buon modo per visualizzare le regole di visibilità.

Regole di Visibilità (Continua)



Costruttori

I costruttori sono particolari metodi utilizzati per creare oggetti istanza di una classe.

- Non sono ereditati dalle sottoclassi.
- Sono usati per inizializzare le variabili di istanza.
- Non ritornano un valore.

Esempio:

```
/** costruisce un quadrato a partire dalla lunghezza del suo lato.
    @param side la lunghezza del lato del quadrato.
 */
public Square (double side) {
  super(side, side); // costruttore del padre
}
```

super è una keyword speciale che identifica la classe padre permette di invocarne i metodi (in questo caso il costruttore).

Costruttori di Default

Se non viene definito alcun costruttore, Java fornisce un costruttore di default.

Se la classe è pubblica, il costruttore di default è anch'esso pubblico.

Invocare il costruttore di default per costruire un'istanza di **Rectangle**:

```
Rectangle r = new Rectangle();
```

equivale ad invocare un costruttore definito come:

```
public Rectangle() {}
```

Interface

Un interfaccia contiene solo metodi astratti che devono essere implementati nella classe che accetta il contratto.

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

L'implementazione deve avere la stessa segnatura presente nell'interfaccia.

```
public int compareTo(Object o) {
  Shape s = (Shape)o;
  if (calculateArea() < s.calculateArea()) return -1;
  else if (calculateArea() > s.calculateArea()) return 1;
  else return 0;
}
```

Classe Astratta: Shape


```
public abstract class Shape implements Comparable {
  public abstract double calculateArea();
  public abstract double calculatePerimeter();
  public abstract void draw();
          ↖ abstract introduce un metodo senza implementazione
          ↙ contratto stipulato con Comparable
  public int compareTo(Object o) {
     Shape s = (Shape)o; ← il cast serve per poter accedere …
                ↙ ai metodi di Shape non definiti in Object
     if (calculateArea() < s.calculateArea()) return -1;
     else if (calculateArea() > s.calculateArea()) return 1;
     else return 0;
  }
}
```