Classi, Ereditarietà e Polimorfismo

Lezione 7

Scopo della Lezione

- Presentare il concetto di classe ed oggetto come istanza della classe;
- Affrontare il concetto di ereditarietà tra classi e le sue ricadute (polimorfismo, late binding, etc.);

Filosofia di Progettazione OO: Decomposizione del Problema

Dividere il problema in sottoproblemi semplifica la soluzione.

- Applichiamo il principio di "Dividi e Conquista" ripetutamente finché i sottoproblemi non sono semplici da risolvere.

Nella progettazione OO, ogni oggetto è dedito alla risoluzione di un sottoproblema.

Classi e Oggetti: Progettazione

Quale sottoproblema deve risolvere un oggetto (istanza di una certa classe)?

Quali informazioni gli serviranno per adempiere al suo compito?

Quali azioni (metodi) dovrà effettuere per adempiere al suo compito?

Quale interfaccia mostrerà agli altri oggetti?

Quali informazioni dovrà nascondere agli altri oggetti?

Classe Rectangle

Nome della Classe: **Rectangle**.

Task: rappresentare l'omonima figura geometrica.

Informazioni necessarie (variabili d'istanza).

- **height**: memorizza l'altezza del rettangolo.
- **width**: memorizza la larghezza del rettangolo.

Operazioni necessarie (metodi pubblici).

- **Rectangle()**: permette di inizializzare la larghezza e l'altezza di un rettangolo.
- **calculateArea()**: calcola l'area del rettangolo.
- **calculatePerimeter()**: calcola il perimetro del rettangolo.
- **getHeigth()**, **getWidth()**: ritornano rispettivamente lunghezza e larghezza del rettangolo

Variabili di Istanza e Metodi Pubblici

Due nuovi concetti: variabile d'istanza e metodi pubblici.

- Le variabile d'istanza sono locazioni di memoria in cui vengono memorizzati i dati necessari agli oggetti per adempiere al proprio compito.

Sono caratteristiche dell'oggetto e non della classe.

- Un metodo pubblico è una porzione di codice usata per computare un sottoproblema.

Rappresentano il modo con cui è possibile interagire con gli oggetti.

Dati, Metodi e Algoritmi

Quale tipo di dato rappresenta le informazioni necessarie al rettangolo?

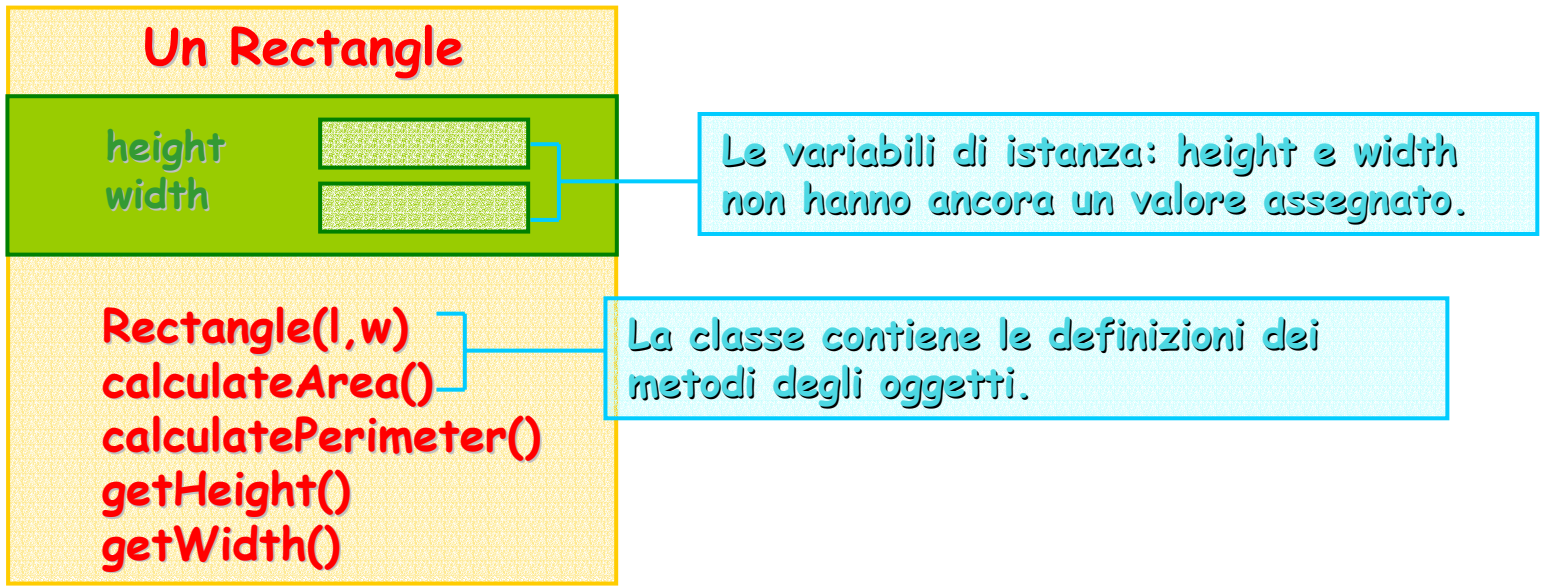
In che modo i metodi espleteranno il proprio compito?

- Quale problema il metodo risolve?
- Quali informazioni servono al metodo?
- Che risultato produrrà?
- Che algoritmo il metodo dovrà usare?

Un algoritmo è una descrizione passo-passo della soluzione di un problema.

La Classe Rectangle

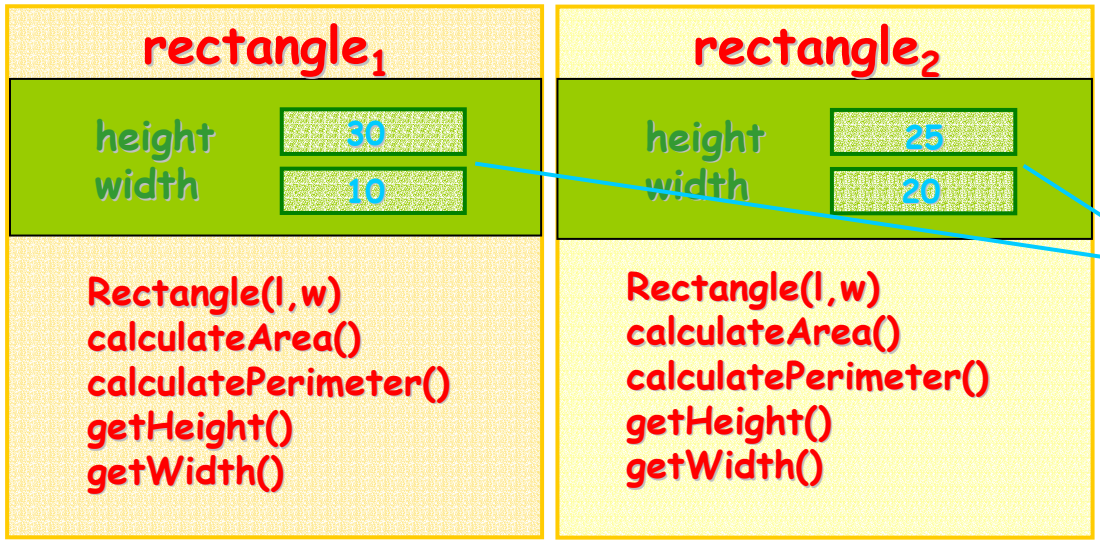
La classe è un template. Descrive la forma degli oggetti ma non il loro contenuto.



Istanziare la Classe Rectangle

Creazione (o istanziazione) di due istanze della classe Rectangle:

```
Rectangle rectangle1 = new Rectangle(30,10);
Rectangle rectangle2 = new Rectangle(25, 20);
```



Interagire con i Rettangoli

Tramite le chiamate di metodo noi possiamo chiedere ad ogni rettangolo di dirci qual è la sua area:

```
System.out.println("Area rectangle1: " + rectangle1.calculateArea());
System.out.println("Area rectangle2: " + rectangle2.calculateArea());
```

Riferimenti agli Oggetti

Chiamate di Metodo

output:

```
Area rectangle1: 300
Area rectangle2: 500
```

La Classe RectangleUser


```
import prog.io.*;
public class RectangleUser {
   public static void main(String argv[]) {
      ConsoleOutputManager video = new ConsoleOutputManager();
      Rectangle rectangle1 = new Rectangle(30,10);
      Rectangle rectangle2 = new Rectangle(25,20);
      video.println("r1 area " + rectangle1.calculateArea());
      video.println("r2 area " + rectangle2.calculateArea());
    } // main()
} // RectangleUser

[21:21]cazzola@ulik:esercizi>java RectangleUser
r1 area 300.0
r2 area 500.0
```

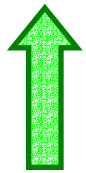
Creazione Oggetti

Usare gli Oggetti

Ereditarietà: Relazione

L'ereditarietà permette di specializzare una classe.

```
public class Rectangle extends Object { ... }
```



Un quadrato è un rettangolo la cui larghezza è pari alla propria altezza.

```
public class Square extends Rectangle { ... }
```

Superclasse

```
public class Cube extends Square { ... }
```

Sottoclasse

Un Cube è (is-a) un Square che è un (is-a) Rectangle che è (is-a) un Object

Usare la Classe Square


```
import prog.io.*;

public class TestSquare {
  public static void main(String argv[]) {
    ConsoleOutputManager video = new ConsoleOutputManager();
    Square square = new Square ( 100 );
    video.println( "L'area del quadrato è "+square.calculateArea() );
  }
} // TestSquare
```

Crea un nuovo quadrato di lato 100.

Il metodo ereditato calculateArea() può essre usato come se fosse stato definito in Square.

```
[21:46]cazzola@ulik:esercizi>java TestSquare
L'area del quadrato è 10000.0
```

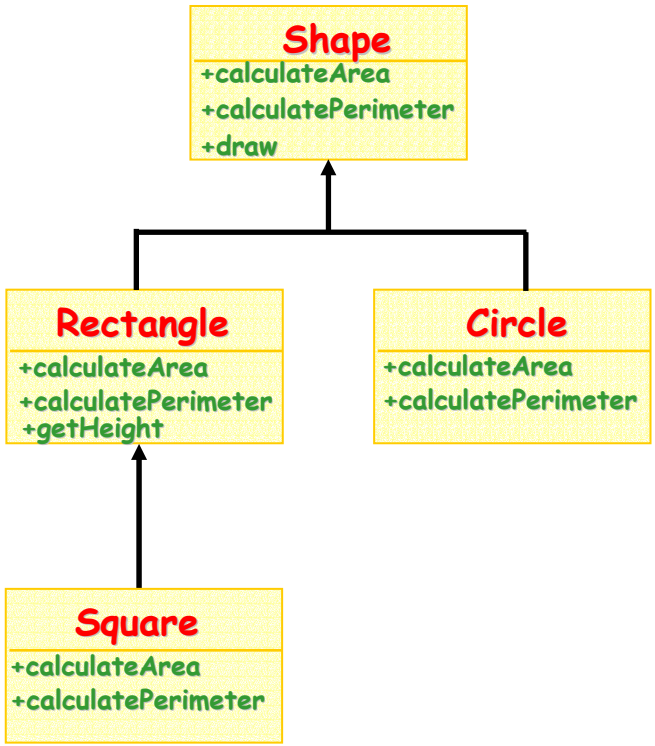
Classi e Metodi Astratti

Qualsiasi classe che contiene un metodo astratto, cioè il prototipo di un metodo non implementato, è definita astratta ed è introdotta dalla keyword **abstract**.

Una classe astratta non può essere istanziata. **DEVE** essere specializzata.

Una sottoclasse di una classe astratta può essere istanziata solo se implementa **TUTTI** i metodi astratti definiti nella super classe. Se implementa solo alcuni metodi è anch'essa astratta.

Gerarchia di Figure



Shape è una classe astratta, possiamo prevedere che una forma abbia un'area ed un perimetro ma di certo non siamo in grado di calcolarle.

calculateArea() e **calculatePerimeter()** sono metodi astratti che verranno implementati dalle sottoclassi (**Circle** e **Rectangle**) e ridefiniti nella classe **Square**.

getHeight() è un metodo che non può appartenere a **Shape** perché non è detto che una forma abbia un lato (ad es. il cerchio non ha lati) ma può essere usato anche da **Square** senza modifiche.

Polimorfismo ed Estendibilità

Un metodo polimorfo ha comportamento diverso se chiamato da oggetti di tipo diverso.

Es. **calculateArea()** attivato da un'instanza di Circle o da un'instanza si Rectangle.

Estendibilità: la funzionalità definita nella super classe è applicata anche alla sottoclasse.

Es. **getHeight()** può essere attivato da un'istanza di Square anche se la classe non definisce quel metodo.

Polimorfismo e Late Binding


```
import prog.io.*;

public class LateBinding {
  public static void main(String argv[]) {
    Rectangle r = new Rectangle(6,7); Square s = new Square(7);
    r.draw(); s.draw();
    r = s; // Ok!!!! un quadrato è anche un rettangolo
    s = r; // No!!!! un rettangolo non è un quadrato
    r.draw();
  }
}

[17:15]cazzola@ulik:esercizi>javac LateBinding.java
LateBinding.java:9: incompatible types
found   : Rectangle
required: Square                    ← errore di tipo
    s = r; // No!!!! un rettangolo non è un quadrato
      ^
1 error                             ← commentando riga 8 invece si ha
[17:16]cazzola@ulik:esercizi/lucidi>java LateBinding
I'm a Rectangle! My sides are: 6.0,7.0
I'm a Square! My side is: 7.0
I'm a Square! My side is: 7.0
```

Array Polimorfo di Figure


```
import prog.io.*;

public class PolymorphicArray {
  public static void main(String argv[]) {
    Shape a[] = {new Square(7), new Circle(3.14), new Rectangle(6,7),
                 new Square(5), new Circle(0.7), new Rectangle(7,2),
                 new Square(2)};
    for (int i=0;i<a.length;i++) a[i].draw();
  }
}
```

draw() è un metodo astratto di **Shape**, viene ridefinito in ogni sua sottoclasse. A seconda della ridefinizione stampa un messaggio relativo alla classe di appartenenza.

Array Polimorfo di Figure


```
[11:34]cazzola@ulik:esercizi>java PolymorphicArray
I'm a Square! My side is: 7.0
I'm a Circle! My ray is: 3.14
I'm a Rectangle! My sides are: 6.0,7.0
I'm a Square! My side is: 5.0
I'm a Circle! My ray is: 0.7
I'm a Rectangle! My sides are: 7.0,2.0
I'm a Square! My side is: 2.0
```

← draw() è attivato su uno Square

← draw() è attivato su un Circle

↑ draw() è attivato su un Rectangle

Nota il tipo dinamico è quello che determina quale versione di **draw()** verrà attivata.

es. Shape a = **new** Circle(3.14);

Shape è il tipo statico di a (definito durante la compilazione) **Circle** è il tipo dinamico di a (definito durante l'esecuzione).

L'importanza delle Classi Astratte!


```
import prog.io.*;

public class PolymorphismWithError {
  public static void main(String argv[]) {
    Shape a = new Square(7);
    a.getHeight();
  }
}
```

```
[11:56]cazzola@ulik:esercizi>javac PolymorphismWithError.java
PolymorphismWithError.java:7: cannot resolve symbol
symbol  : method getHeight ()
location: class Shape
    a.getHeight();   ← getHeight() non è definito in Shape ma in Rectangle
      ^
1 error
```

L'importanza delle Classi Astratte!


```
import prog.io.*;

public class PolymorphismWithoutErrors {
  public static void main(String argv[]) {
    ConsoleOutputManager video = new ConsoleOutputManager();
    Shape a = new Square(7);
    video.println("Ora funziona: "+((Square)a).getHeight());
    a.draw();                                  ↑ dopo il cast getHeight() è Ok
        ↖ draw() invece è definito in Shape quindi non serve il cast!
  }
}
```

```
[12:29]cazzola@ulik:esercizi>javac PolymorphismWithoutErrors.java
[12:29]cazzola@ulik:esercizi>java PolymorphismWithoutErrors
Ora funziona: 7.0
I'm a Square! My side is: 7.0
```

Es. Calcolo Aree

Considerando l'array polimorfo dell'esercizio precedente scrivere la classe **CompareAreas** che:

- calcola l'area di ogni singola figura presente nell'array usando il metodo **calculateArea()**, e
- determina quale figura ha l'area più grande e quale l'area più piccola

Es. Calcolo Aree


```
import prog.io.*;

public class CompareAreas {
  public static void main(String argv[]) {
    ConsoleOutputManager video = new ConsoleOutputManager();
    Shape a[] = {new Square(7), new Circle(3.14), new Rectangle(6,7), new Square(5),
                 new Circle(0.7), new Rectangle(7,2), new Square(2)};
    double max = a[0].calculateArea(), min = max;
    int maxPosition = 0, minPosition = 0;
    for (int i=1;i<a.length;i++) {
      double aux = a[i].calculateArea();
      if (aux > max) { max = aux; maxPosition = i; }
      else if (aux < min) { min = aux; minPosition = i; }
    }
    video.println("The shape in position "+maxPosition+" has the largest area: "+max);
    video.println("The shape in position "+minPosition+" has the smallest area: "+min);
  }
}
[14:27]cazzola@ulik:esercizi>java CompareAreas
The shape in position 0 has the largest area: 49.0
The shape in position 4 has the smallest area: 1.5393804002589986
```

Interfacce

Le interfacce sono usate durante l'implementazione delle classi:

Es. `class Rectangle implements Comparable {…}`

Specificano un insieme di metodi pubblici che la classe in questione DEVE fornire;

L'uso delle interfacce permette di ovviare all'assenza dell'ereditarietà multipla:

Es. `class Rectangle implements Comparable, Clonable {…}`

Interfacce vs Classi Astratte

Le interfacce sono più astratte delle classi astratte.

Le interfacce non possono contenere:

- metodi statici;
- attributi;
- implementazioni di metodi;

mentre le classi astratte sì.

Le classi astratte possono contenere del codice le interfacce no.

Comparable

L'interfaccia **Comparable** impone un ordinamento totale sulle istanze della classe che la implementa.

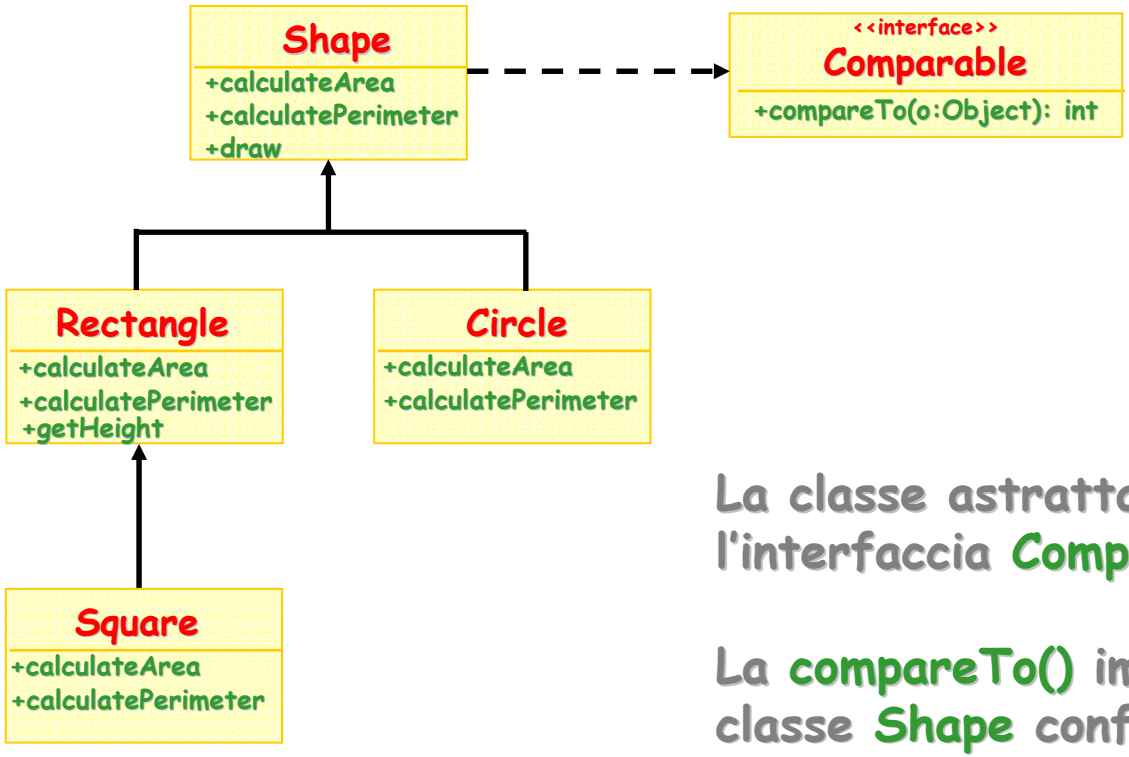
Le classi che vogliono implementarla DEVONO implementare il metodo:

```
public int compareTo(Object o)
```

Già visto per la classe **String**.

compareTo() confronta due oggetti: o_1 (chiamante) e o_2 (parametro) ritornando un valore negativo se $o_1 < o_2$, un valore positivo se $o_1 > o_2$, zero altrimenti.

Gerarchia di Figure Estesa



La classe astratta **Shape** implementa l'interfaccia **Comparable**.

La **compareTo()** implementata dalla classe **Shape** confronta le aree di due figure e determina qual è la più ampia.

Rectangle, **Circle** e **Square** ereditano la **compareTo()** implementata da **Shape**.

Es. Ordinamento Polimorfo

Scrivere la Classe **SortPolymorphicArray** che prende un array di **Shape** e le dispone in ordine crescente in base alle aree.

Suggerimenti:

- sfruttare l'interfaccia **Comparable** implementata da **Shape**;
- utilizzare i metodi della classe **java.util.Arrays**.

java.util.Arrays

La classe **Arrays** fornisce un insieme di metodi utili per la manipolazione di array:

- `public static Object binarySearch(Object a[], Object val);`
- `public static void sort(Object a[]);`
- `public static void fill(Object a[], Object val);`

I metodi di **Arrays** usano:

- il polimorfismo per gli elementi dell'array su cui operano;
- il fatto che le componenti dell'array implementino l'interfaccia **Comparable**.

Esattamente come accade per il nostro array di **Shape**.

SortPolymorphicArray


```
import prog.io.*;

public class SortPolymorphicArray {
  public static void main(String argv[]) {
    ConsoleOutputManager video = new ConsoleOutputManager();
    Shape a[] = {new Square(7), new Circle(3.14), new Rectangle(6,7),
                 new Square(5), new Circle(0.7), new Rectangle(7,2),
                 new Square(2)};
    java.util.Arrays.sort(a);      // ordina un array generico i cui
                                   // elementi implementano Comparable
    for(int i=0;i<a.length;i++) {
       a[i].draw();
       video.println("My area is: "+a[i].calculateArea());
    }
  }
}
```

SortPolymorphicArray


```
[22:20]cazzola@mjolnir:esercizi>java SortPolymorphicArray
I'm a Circle! My ray is: 0.7
My area is: 1.5393804002589986
I'm a Square! My side is: 2.0
My area is: 4.0
I'm a Rectangle! My sides are: 7.0,2.0
My area is: 14.0
I'm a Square! My side is: 5.0
My area is: 25.0
I'm a Circle! My ray is: 3.14
My area is: 30.97484692733928
I'm a Rectangle! My sides are: 6.0,7.0
My area is: 42.0
I'm a Square! My side is: 7.0
My area is: 49.0
```