



Programmazione orientata agli oggetti

Classi astratte e interfacce

Classi astratte

- Java ci consente di definire classi in cui uno o più metodi non sono implementati, ma solo **dichiarati**
- Questi metodi sono detti **astratti** e vengono marcati con la parola chiave **abstract**
- Non hanno un corpo tra parentesi graffe ma solo la dichiarazione terminata con ;
- 💣 **Attenzione:** un metodo vuoto ({}) e un metodo astratto sono due cose diverse
- Una classe che ha **almeno un metodo astratto** si dice **classe astratta**
- Le classi **non astratte** si dicono **concrete**
- Una classe astratta **deve** essere marcata a sua volta con la parola chiave **abstract**

Utilità delle classi astratte

- L'aspetto più importante è che **non è possibile creare istanze di una classe astratta**
- Dal momento che una classe astratta non può generare istanze a che cosa serve?
- **Serve come superclasse comune per un insieme di sottoclassi concrete**
- Queste sottoclassi, in virtù del **subtyping**, sono in qualche misura **compatibili** e **intercambiabili** fra di loro
- Infatti sono **tutte sostituibili con la superclasse**: sulle istanze di ognuna di esse possiamo invocare i metodi ereditati dalla classe astratta
- Ciascuna li potrà implementare diversamente però

Esempio - 1

- Scriviamo la **classe astratta** Shape che definisce una generica figura geometrica di cui possiamo calcolare area e perimetro

```
public abstract class Shape
{
    public abstract double area();
    public abstract double perimeter();
}
```

<i>Shape</i>
<i>+area() : double</i> <i>+perimeter() : double</i>

- A lato vediamo la rappresentazione UML: metodi e classi astratte sono in **corsivo**
- Definiamo quindi due **classi concrete**, Circle e Rectangle che discendono da Shape e forniscono un'implementazione dei metodi astratti di Shape

Esempio - 2

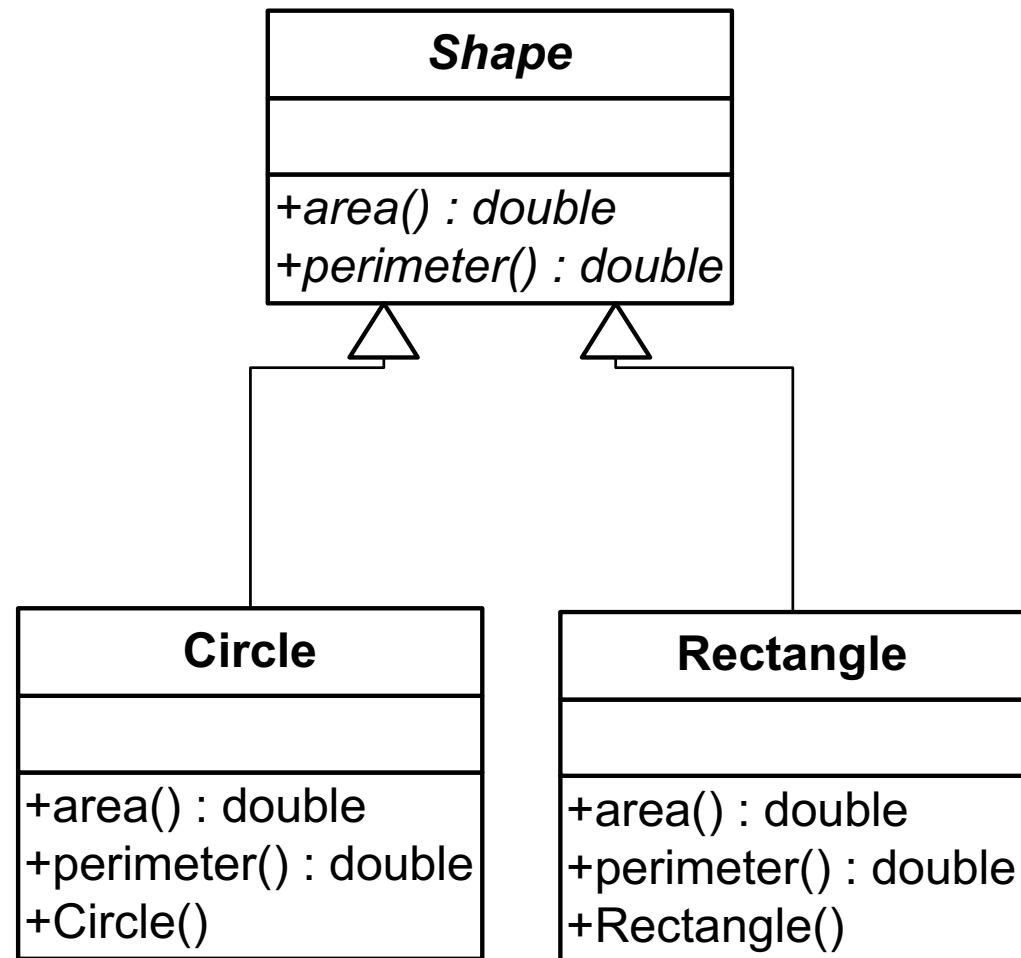
- Vediamo l'implementazione di Circle e di Rectangle:

```
public class Circle extends Shape
{
    protected double r;
    public Circle(double r) { this.r = r; }
    public double area() { return Math.PI * r * r; }
    public double perimeter() { return 2 * r * Math.PI; }
    public double getRadius() { return r }
}
```

```
public class Rectangle extends Shape
{
    protected double w,h;
    public Rectangle(double w, double h)
    { this.w = w; this.h = h; }
    public double area() { return w * h; }
    public double perimeter() { return 2 * (w + h); }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
}
```

Esempio – Diagramma UML

- Ecco il diagramma delle classi:



Esempio - 3

- Vediamo infine la classe **EempioShape**:

```
public class EempioShape
{
    public static void main(String args[])
    {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Circle(2.5);
        shapes[1] = new Rectangle(1.2, 3.0);
        shapes[2] = new Rectangle(5.5, 3.8);
        double totalArea = 0;
        for (int i=0; i<shapes.length; i++)
            totalArea=totalArea+shapes[i].area();
        System.out.println(totalArea);
    }
}
```

- Grazie all'uso della classe astratta abbiamo potuto costruire un array che contiene indifferentemente cerchi e rettangoli
- Abbiamo poi calcolato l'area totale trattando uniformemente cerchi e rettangoli

Esempio – 1bis

- Scriviamo la **classe astratta** Shape che definisce una generica figura geometrica di cui possiamo calcolare area e perimetro

```
public abstract class Shape
{
    public abstract double area();
    public abstract double perimeter();
    public String toString()
        {return "geometrical shape"}
}
```

Shape
+area() : double +perimeter() : double

Esempio - 2

```
public class Circle extends Shape
{
    protected double r;
    public Circle(double r) { this.r = r; }
    public double area() { return Math.PI * r * r; }
    public double perimeter() { return 2 * r * Math.PI; }
    public double getRadius() { return r }
    public toString()      System.out.println("'circle'');
```

■ }

```
public class Rectangle extends Shape
{
    protected double w,h;
    public Rectangle(double w, double h)
    { this.w = w; this.h = h; }
    public double area() { return w * h; }
    public double perimeter() { return 2 * (w + h); }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
    public toString()      System.out.println("'rectangle'');
```

Esempio - 3

- Vediamo infine la classe **EempioShape**:

```
public class EempioShape
{
    public static void main(String args[]) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Circle(2.5);
        shapes[1] = new Rectangle(1.2, 3.0);
        shapes[2] = new Rectangle(5.5, 3.8);
        for (int i=0; i<shapes.length; i++)
            shapes[i].toString();
        double totalArea = 0;
        for (int i=0; i<shapes.length; i++)
            totalArea=totalArea+shapes[i].area();
        System.out.println(totalArea);
    }
}
```

- Grazie all'uso della classe astratta abbiamo potuto costruire un array che contiene indifferentemente cerchi e rettangoli
- Abbiamo poi calcolato l'area totale trattando uniformemente cerchi e rettangoli

Limiti delle classi astratte

- **Le classi in Java sono a ereditarietà singola (una sola classe base)**
- Se vogliamo estendere in nostro lavoro implementando forme geometriche che possono anche essere disegnate sullo schermo ...
- Potremmo definire una classe astratta DrawableShape da cui far discendere DrawableCircle e DrawableRectangle.
- Conflitto (clashing di nomi) su quale metodo di Circle e Rectangle usare per calcolare area e perimetro
- DrawableCircle non può discendere contemporaneamente da Circle e da DrawableShape
- **Non possiamo scrivere**
`class DrawableCircle extends Circle, DrawableShape`

Interfacce

Interfacce

- Fortunatamente Java ci mette a disposizione uno strumento per risolvere questo problema: le **interfacce**
- Possiamo definire l'**interfaccia** Drawable in questo modo:

```
public interface Drawable
{
    public void setColor(int c);
    public void setPosition(double x, double y);
    public void draw();
}
```

«interface» Drawable
+setColor() +setPosition() +draw()

- La definizione di un'interfaccia è molto simile a quella di una classe astratta, è un'elenco di metodi senza implementazione
- A differenza di una classe astratta: **tutti** i metodi sono privi di implementazione
- A lato la rappresentazione UML

Uso delle interfacce

- Possiamo scrivere DrawableRectangle così:

```
public class DrawableRectangle
    extends Rectangle implements Drawable
{
    protected int c;
    protected double x, y;
    public DrawableRectangle(double w, double h)
    { super(w,h); }
    public void setColor(int c) { this.c = c; }
    public void setPosition(double x, double y)
    { this.x = x; this.y = y; }
    public void draw()
    { System.out.println("Rettangolo, posizione"+x +
        " " +y+" colore "+c);}
}
```

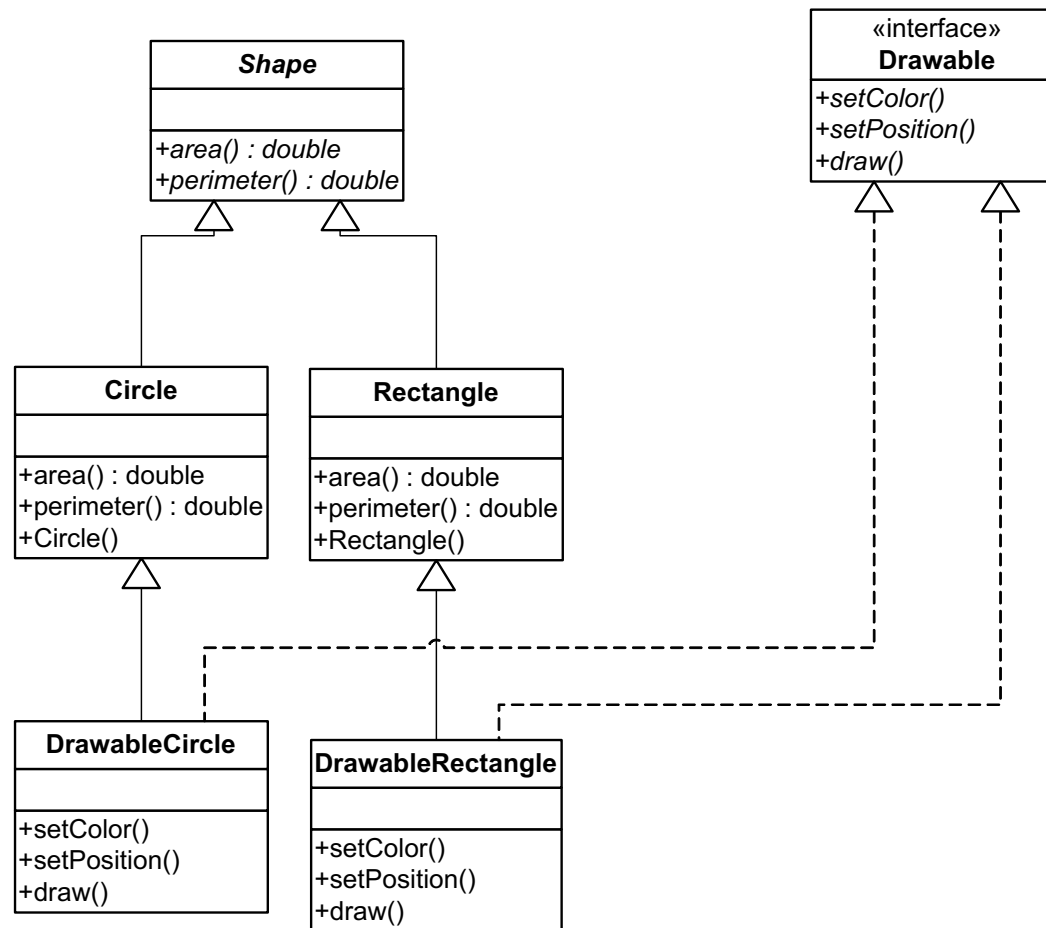
Uso delle interfacce

- In maniera del tutto simile possiamo definire **DrawableCircle** che sarà dichiarata come:

```
public class DrawableCircle
    extends Circle implements Drawable
{
    protected int c;
    protected double x, y;
    public DrawableCircle(double r)
    { super(r); }
    public void setColor(int c) { this.c = c; }
    public void setPosition(double x, double y)
    { this.x = x; this.y = y; }
    public void draw()
    { System.out.println("Cerchio, posizione"+x +
        " " +y+" colore "+c);}
}
```

Diagramma UML

- Il diagramma UML complessivo è riportato qui sotto
- Le relazioni **implements** sono rappresentate in modo simile a extends, ma con una **riga tratteggiata**



Precisazioni sulle interfacce

- Un'interfaccia è una collezione dichiarazioni di metodi, simile ad una classe astratta
- Una classe oltre a discendere da una superclasse, specificata con la parola chiave **extends**, può implementare una o più interfacce usando la parola chiave **implements**
- **Attenzione:** se una classe dichiara che implementa una interfaccia deve obbligatoriamente fornire un'implementazione di **tutti i metodi** dell'interfaccia
- Non può implementare solo alcuni metodi!
- In Java è possibile definire variabili (riferimenti) che hanno come tipo un'interfaccia:
Drawable d;
- A cosa servono?

Interfacce e subtyping

- Java prevede una forma estesa di **subtyping**
- Nella definizione classica il subtyping ci permette di utilizzare una classe derivata al posto della classe base
- Quindi ci permette di scrivere

```
Shape s;  
s = new Circle(5.7);
```
- Il subtyping in Java ci permette anche di utilizzare al posto di un'interfaccia qualunque classe la implementi
- Possiamo quindi scrivere, usando una variabile che ha come tipo l'interfaccia Drawable

```
Drawable d;  
d = new DrawableCircle(6.5);
```

Esempio

- Vediamo la classe **EsempioDrawable**, simile a **EsempioShape**:

```
public class EsempioDrawable
{
    public static void main(String args[])
    {
        Drawable[] drawables = new Drawable[3];
        drawables[0] = new DrawableCircle(2.5);
        drawables[1] = new DrawableRectangle(1.2, 3.0);
        drawables[2] = new DrawableRectangle(5.5, 3.8);
        for (int i=0; i<drawables.length; i++)
        {
            drawables[i].setColor(i);
            drawables[i].setPosition(i*10.0, i*20.0);
            drawables[i].draw();
        }
    }
}
```

- Grazie all'uso dell'interfaccia abbiamo potuto costruire un array che contiene indifferentemente istanze di classi diverse che implementano l'interfaccia **Drawable** e disegnarle tutte insieme con un solo ciclo **for**

Ancora sulle interfacce

- Un altro aspetto interessante è la possibilità di definire una classe che **implementa Drawable**, ma **non discende** da Shape
- Esempio: testo disegnato ad una data posizione:

```
public class DrawableText implements Drawable
{
    protected int c;
    protected double x, y;
    protected String s;
    public DrawableText(String s) { this.s = s }
    public void setColor(int c) { this.c = c; }
    public void setPosition(double x, double y)
    { this.x = x; this.y = y; }
    public void draw()
    { System.out.println("Testo, posizione"+x +
        " " +y+" colore "+c);}
}
```

- Si ha quindi una forma di compatibilità e di sostituibilità tra classi indipendente dalla catena di ereditarietà

Esempio

- Potremmo riscrivere EsempioDrawable così:

```
public class EsempioDrawable
{
    public static void main(String args[])
    {
        Drawable[] drawables = new Drawable[3];
        drawables[0] = new DrawableCircle(2.5);
        drawables[1] = new DrawableRectangle(1.2, 3.0);
        drawables[2] = new DrawableText("Ciao");
        for (int i=0; i<drawables.length; i++)
        {
            drawables[i].setColor(i);
            drawables[i].setPosition(i*10.0,i*20.0);
            drawables[i].draw();
        }
    }
}
```

- Abbiamo trattato l'istanza di DrawableText in modo del tutto uniforme a DrawableRectangle e DrawableCircle

Precisazioni

- A differenza di `extends` dopo la clausola **implements** possiamo aggiungere un numero qualsiasi di nomi di interfacce

```
public class DrawableRectangle
    extends Rectangle
    implements Drawable, Sizeable, Draggable
```

- **Una classe può implementare più interfacce**
- Talvolta si dice che **un'interfaccia è un contratto tra chi la implementa e chi la usa**
- La classe che implementa un'interfaccia, essendo obbligata ad implementarne tutti i metodi, garantisce la fornitura di un servizio
- Chi usa un'interfaccia ha la **garanzia** che il contratto di **servizio è effettivamente realizzato**: non può accadere che un metodo non possa essere chiamato.