



**Agent and Object Technology Lab**  
Dipartimento di Ingegneria dell'Informazione  
Università degli Studi di Parma



Ingegneria del software A

Ereditarietà e Polimorfismo

**Prof. Agostino Poggi**

- ◆ Tra le classi è possibile definire una relazione di **sottoclasse** (sottoinsieme)
  - Classi: Animale, Felino e Gatto
  - Gatto è una sottoclasse (classe derivata) di Felino, Felino è una sottoclasse (classe derivata) di Animale
  - Animale è la **classe base** di Felino e di Gatto
- ◆ Quindi si possono usare delle classi esistenti come base per creare nuove classi

- ♦ La generalizzazione è una relazione di tipo **is-a**
- ♦ È sempre possibile usare un oggetto di una sottoclasse al posto di un oggetto della classe base
- ♦ Se `Square` è una sottoclasse di `Rectangle`, allora possiamo scrivere:

```
list.addElement(new Rectangle(5, 7));  
...  
list.addElement(new Square(6));  
list.addElement(new Square(200));  
  
SelectionSorter sorter = new SelectionSorter();  
sorter.sort(list);
```

8/Main.java

- ♦ Una classe derivata (sottoclasse) viene costruita partendo dalla classe base

```
public class Square extends Rectangle {  
    // caratteristiche aggiuntive  
}
```

- ♦ L'ereditarietà tra le classi Java è singola
  - Solo una classe base (diretta) per ogni classe derivata
- ♦ Tutte le classi sono derivate (implicitamente) dalla classe **Object**

```
Object obj = new Rectangle(4, 3);  
obj.equals(new Rectangle(5, 5));
```

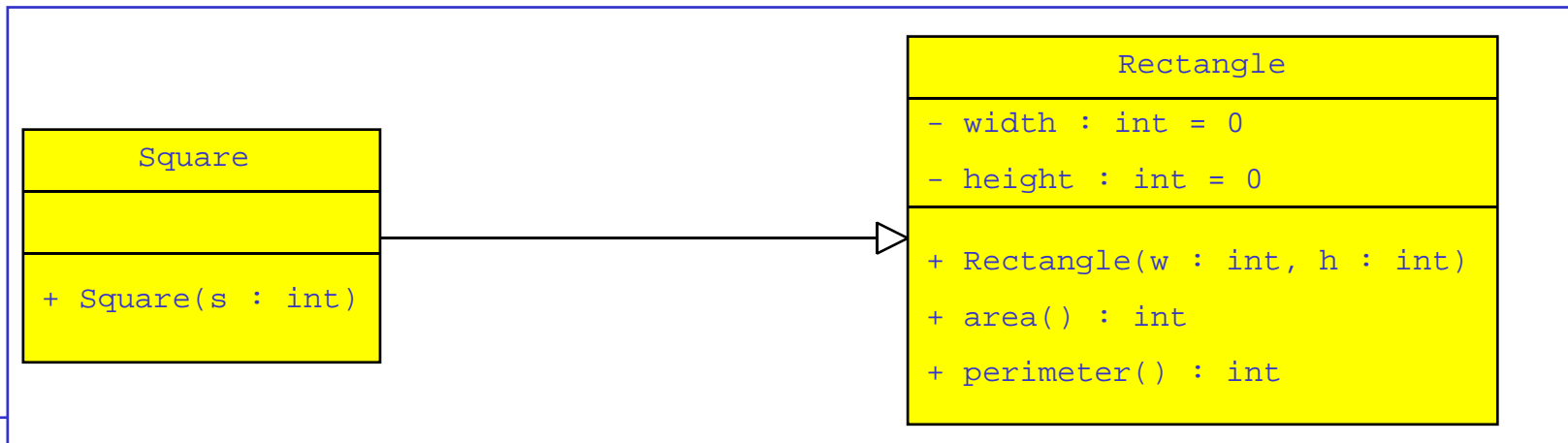
◆ Una sottoclasse

- Eredita tutte le caratteristiche pubbliche della classe base
- Ma non può accedere alle caratteristiche private della classe base
- Può dichiarare nuove caratteristiche che non sono visibili dalle classi base

◆ La classe base

- Può definire delle caratteristiche protette a cui solo lei, le sue sottoclassi e le classi del suo package possono accedere

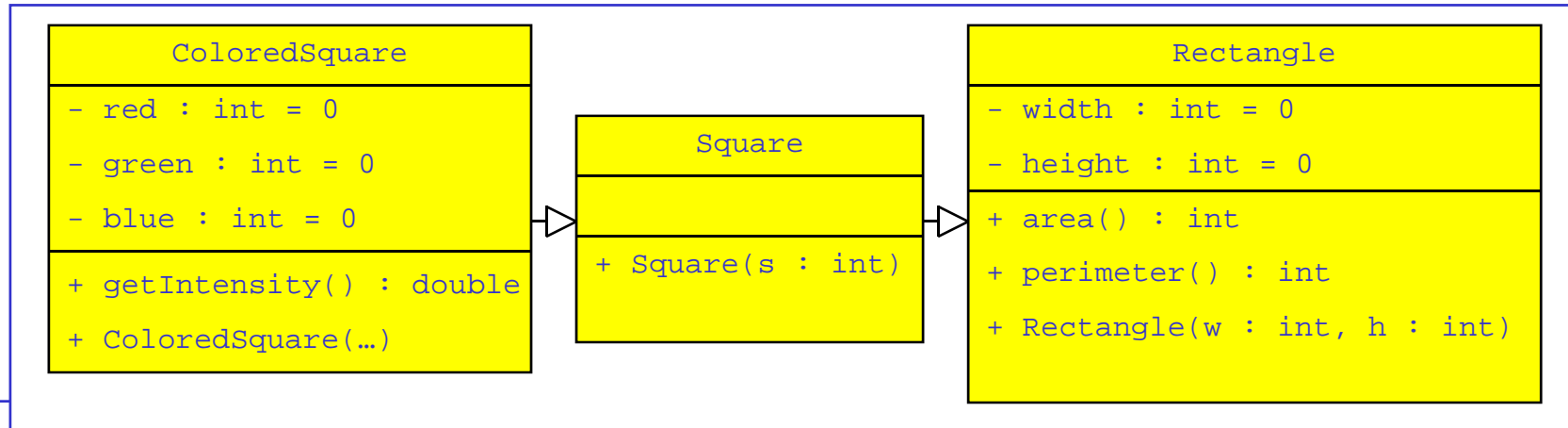
- ♦ I costruttori non vengono ereditati
  - Non contengono il codice di costruzione della classe base
- ♦ È possibile accedere a un costruttore della classe base mediante `super(...)`
  - È necessario costruire le caratteristiche della classe base prima di passare a quelle della classe derivata
  - Quindi `super(...)` deve essere chiamato all'inizio del nuovo costruttore
- ♦ Se la chiamata `super(...)` non viene eseguita, allora viene eseguito il costruttore senza parametri della classe base



```
public class Square extends Rectangle {
    public Square(final int s) {
        super(s, s);
    }
}
```

9/Square.java

## Classe ColoredSquare



```

public class ColoredSquare extends Square {
    private int red = 0, green = 0, blue = 0;

    public double getIntensity() {
        return (red / 255.0 + green / 255.0 + blue / 255.0) / 3;
    }

    public ColoredSquare(int s, int r, int g, int b) {
        super(s);
        red = r; green = g; blue = b;
    }
}

```

10/Square.java



- ◆ Un metodo della classe base può essere ridefinito nelle classi derivate (metodo polimorfo)

```
public class Square extends Rectangle {  
    public int area() { return getWidth() * getWidth(); }  
    public int perimeter() { return 4 * getWidth(); }  
}
```

```
Rectangle r1 = new Rectangle(3, 4);
```

```
Rectangle r2 = new Square(6);
```

```
r1.area(); // invoca il metodo di Rectangle
```

```
r2.area(); // invoca il metodo di Square
```

11/Square.java

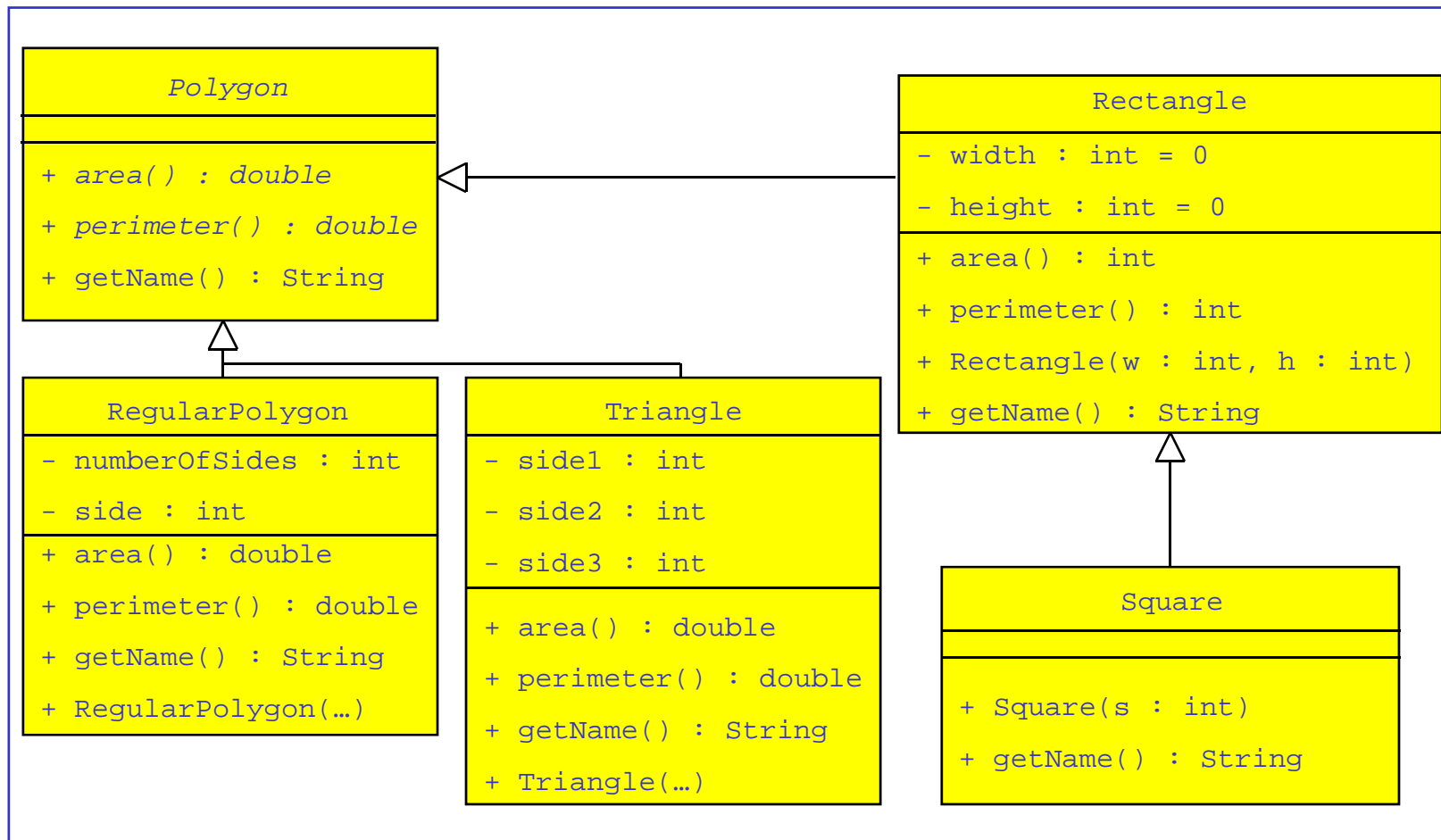
- ◆ **null**, punta all'oggetto nullo
  - Una reference punta ad un oggetto valido o a `null`
- ◆ **this**, punta all'oggetto corrente
  - Nella classe `Rectangle`, è possibile chiamare `this.area()` o `this.perimeter()`
- ◆ **super**, punta all'oggetto corrente e consente di accedere alle caratteristiche della classe base
  - Nella classe `Square` è possibile utilizzare il metodo `area()` di `Rectangle` mediante `super.area()`

- ◆ Gli attributi e i metodi protetti sono accessibili solo dalla classe e dalle derivate
  - Non sono visibili all'esterno dell'oggetto, come se fossero privati
  - Consentono di rendere visibili alcune parti di una classe alle sue sottoclassi e alle classi dello suo package
- ◆ Si dovrebbe usare, ma con cautela, solo i metodi protetti quando
  - Non conviene rendere pubblici questi metodi perché non offrono servizi significativi (e.g., il metodo `swap()` in `SelectionSorter`)
  - Ma si vogliono riutilizzare nelle sottoclassi
- ◆ Gli attributi e i metodi protetti sono identificati dalla parola chiave `protected`

- ◆ Un attributo non modificabile può essere costante (definito a tempo di compilazione) o assegnato una e una sola volta a runtime
- ◆ Un metodo non modificabile non può essere modificato da una sottoclasse
- ◆ Una classe non modificabile non può essere modificata e quindi estesa da un'altra classe

- ◆ Gli attributi e i metodi non modificabili possono essere di classe e di istanza
- ◆ La ragione per definire attributi, metodi e classi non modificabili è in certi casi l'efficienza e in altri casi una scelta progettuale
- ◆ Gli attributi, i metodi e le classi non modificabili sono identificati dalla parola chiave `final`

- ◆ Se una classe che dichiara dei metodi senza implementarli si dice classe **astratta**
- ◆ Questi metodi sono implementati dalle sue sottoclassi perché
  - La classe base è solo un'unità di riuso e le informazioni per realizzare i metodi non sufficienti nella classe base
  - Le sottoclassi possono implementare questi metodi in modo differente, ma è conveniente che forniscano una interfaccia comune
- ◆ Una classe astratta non può essere istanziata



```
public abstract class Polygon {
    public abstract double area();
    public abstract double perimeter();
    public String getName() { return "Polygon"; }
}

public class Triangle extends Polygon {
    private int side1 = 0, side2 = 0, side3 = 0;
    public double area() {
        double s = perimeter() / 2;
        double a2 = s * (s - side1) * (s - side2) * (s - side3);
        return Math.sqrt(a2);
    }
    public double perimeter() { return side1 + side2 + side3; }
    public String getName() { return "Triangle"; }
}
```

12/Polygon.java, Triangle.java



- ◆ Strutture dati ed algoritmi possono essere definite usando
  - La classe base `Polygon` anziché di `Rectangle`
  - La classe `ListOfPolygons` visto che la classe `SelectionSorter` può sfruttare il fatto che tutti i poligoni hanno un area
- ◆ Se le strutture dati e gli algoritmi sono realizzati in funzione della classe più generale
  - Allora si massimizzano il riuso e la flessibilità del codice

```
public class ListOfPolygons {
    private int size = 0;
    private Polygon[] elements;

    public void addElement(Polygon r) {
        if (size < elements.length)
            elements[size++] = r; }

    public void setElement(int i, Polygon r) {
        if (i < size)
            elements[i] = r;    }

    public Polygon getElement(int i) {
        return elements[i];    }

    public int getSize() {
        return size; }

    public ListOfPolygons() {
        elements = new Polygon[100]; }

}
```

13/ListOfPolygons.java

```
public class SelectionSorter {
    public void sort(ListOfPolygons list) {
        for(int i = 0; i < list.getSize(); i++)
            for(int j = i + 1; j < list.getSize(); j++) {
                Polygon left  = list.getElement(i);
                Polygon right = list.getElement(j);

                if(left.area() > right.area()) swap(list, i, j);
            }
    }

    private void swap(ListOfPolygons list, int i, int j) {
        Polygon t = list.getElement(i);

        list.setElement(i, list.getElement(j));
        list.setElement(j, t);
    }
}
```

14/SelectionSorter.java

```
public class Main {  
    public static void main(String[] args) {  
        // Creazione della lista dei poligoni.  
        ListOfPolygons list = new ListOfPolygons();  
  
        // Creazione dei poligoni e aggiunta alla lista.  
        list.addElement(new Rectangle(5, 7));  
        list.addElement(new Square(6));  
        list.addElement(new Triangle(3, 4, 5));  
        list.addElement(new RegularPolygon(5, 7));  
  
        // Creazione di un oggetto ordinatore  
        SelectionSorter sorter = new SelectionSorter();  
  
        // Ordinamento della lista.  
        sorter.sort(list);  
  
        // Stampa delle aree degli elementi della lista.  
        Printer printer = new Printer();  
        printer.print(list);  
    }  
}
```

15/Main.java

- ◆ Un quadrato è contemporaneamente
  - Un rettangolo
  - Un poligono regolare
- ◆ La classe `Square` dovrebbe estendere le due classi `Rectangle` e `RegularPolygon`
  - Quale stato usare?
  - Quale metodi chiamare?
- ◆ Ma Java non permette l'ereditarietà multipla tra le classi

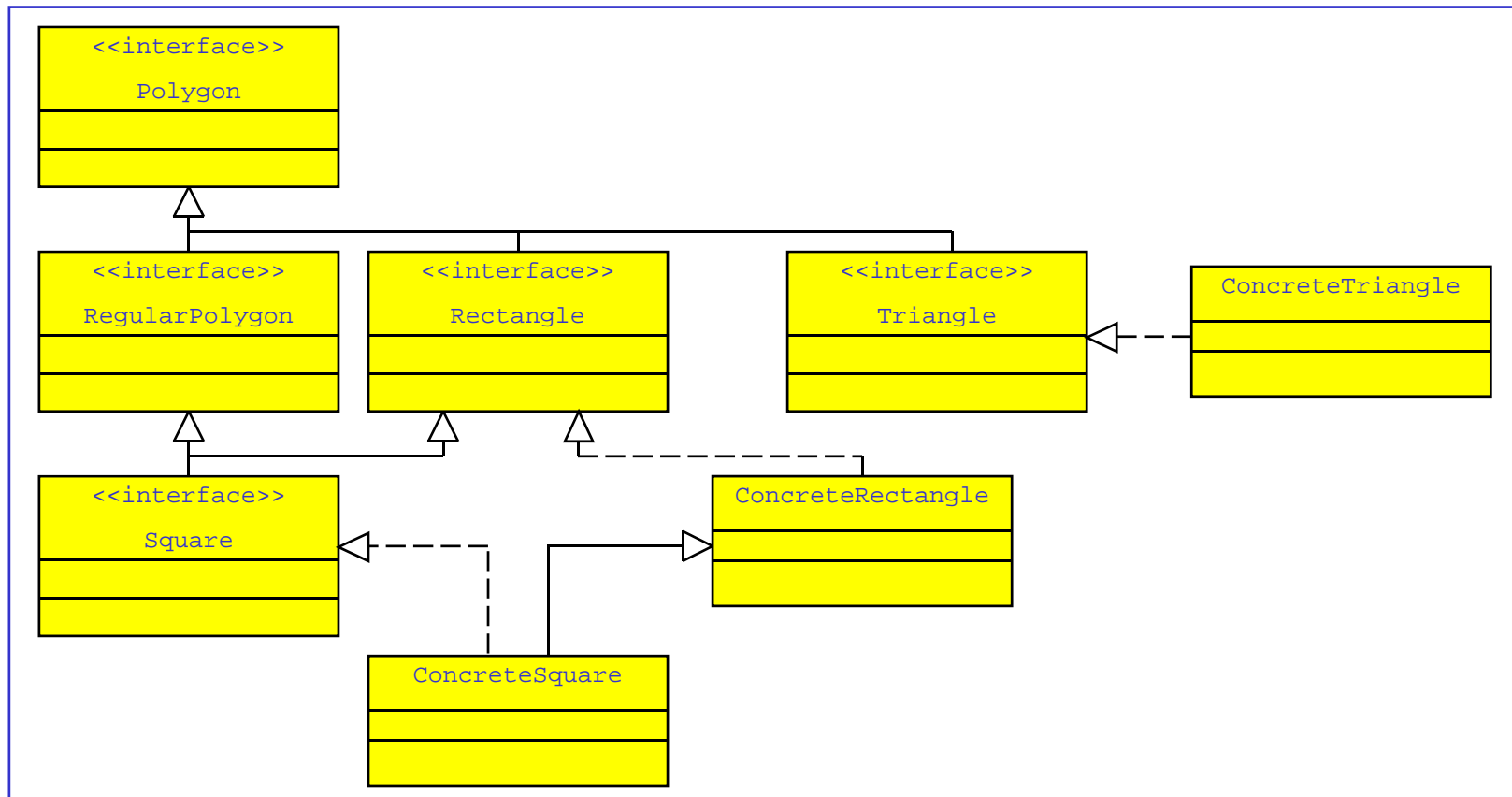
- ♦ L'interfaccia di un oggetto è definita dall'insieme dei metodi pubblici della sua classe
  - L'interfaccia indica cosa un oggetto sa fare e non come lo fa
  - I corpi dei metodi, cioè come i servizi vengono implementati, non sono parte dell'interfaccia
- ♦ L'interfaccia della classe `Rectangle` è definita da:  
`double area()`  
`double perimeter()`
- ♦ Un'interfaccia Java può anche definire degli attributi statici, ma non dei metodi statici

- ◆ Java permette
  - La definizione dell'interfaccia di una oggetto in modo separato dall'implementazione
  - L'ereditarietà multipla tra le interfacce
- ◆ L'uso delle interfacce migliora
  - La pulizia del modello e l'aderenza alla realtà modellata
  - La riusabilità del codice
- ◆ Tuttavia, l'uso delle interfacce dovrebbe essere limitato al caso in cui
  - Se ne prevedono diverse implementazioni
  - Bisogna simulare l'ereditarietà multipla

```
public interface Polygon {  
    public double area();  
    public double perimeter();  
    public String getName();  
}  
  
public interface RegularPolygon extends Polygon {}  
  
public class ConcreteRegularPolygon implements RegularPolygon {  
    public double area()      { ... }  
    public double perimeter() { return numberOfSides*side; }  
    public String getName()   { return "Regular polygon"; }  
    public ConcreteRegularPolygon(int n, int s) { ... }  
    private int numberOfSides = 0;  
    private int side = 0;  
}
```

16/Polygon.java, RegularPolygon.java, ConcreteRegularPolygon.java

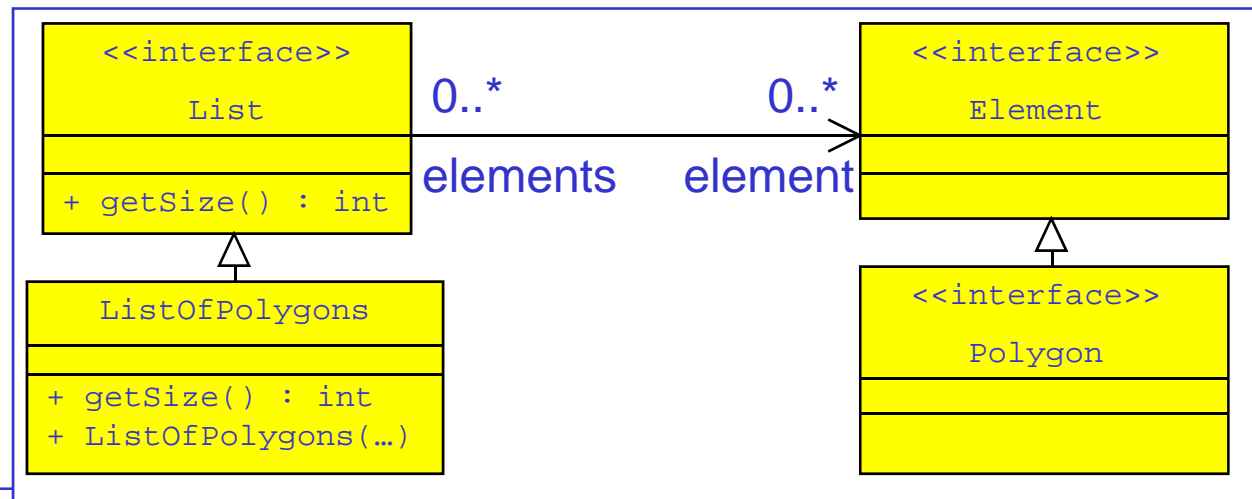




- ◆ L'ereditarietà ha due utilizzi principali
  - Modella il problema (o la soluzione), cosa molto importante nella fase di analisi
  - Massimizza il riuso, cosa molto importante nelle fasi di progettazione e implementazione
- ◆ I due utilizzi sono legati perché la prima bozza di un progetto è il modello che analizza il dominio del problema (o della soluzione)

- ◆ Estendendo una classe
  - Si riutilizza il suo codice
  - Si possono sfruttare le parti protette
- ◆ Costruendo sistemi che vengono realizzati mediante classi che implementano ben determinate interfacce
  - Le classi di un sistema possono essere facilmente utilizzate in altri sistemi
  - Oggi, questo meccanismo di riuso è il più apprezzato

- ♦ Il servizio che abbiamo visto nelle trasparenze precedenti, ordina i poligoni in base alla dimensione della loro area
- ♦ Per massimizzarne la riusabilità bisogna poter permettere almeno criteri di ordinamento differenti
- ♦ Il riuso non viene dalla possibilità di riutilizzare parti dell'algoritmo di ordinamento, ma dalla possibilità di utilizzare algoritmi diversi



```

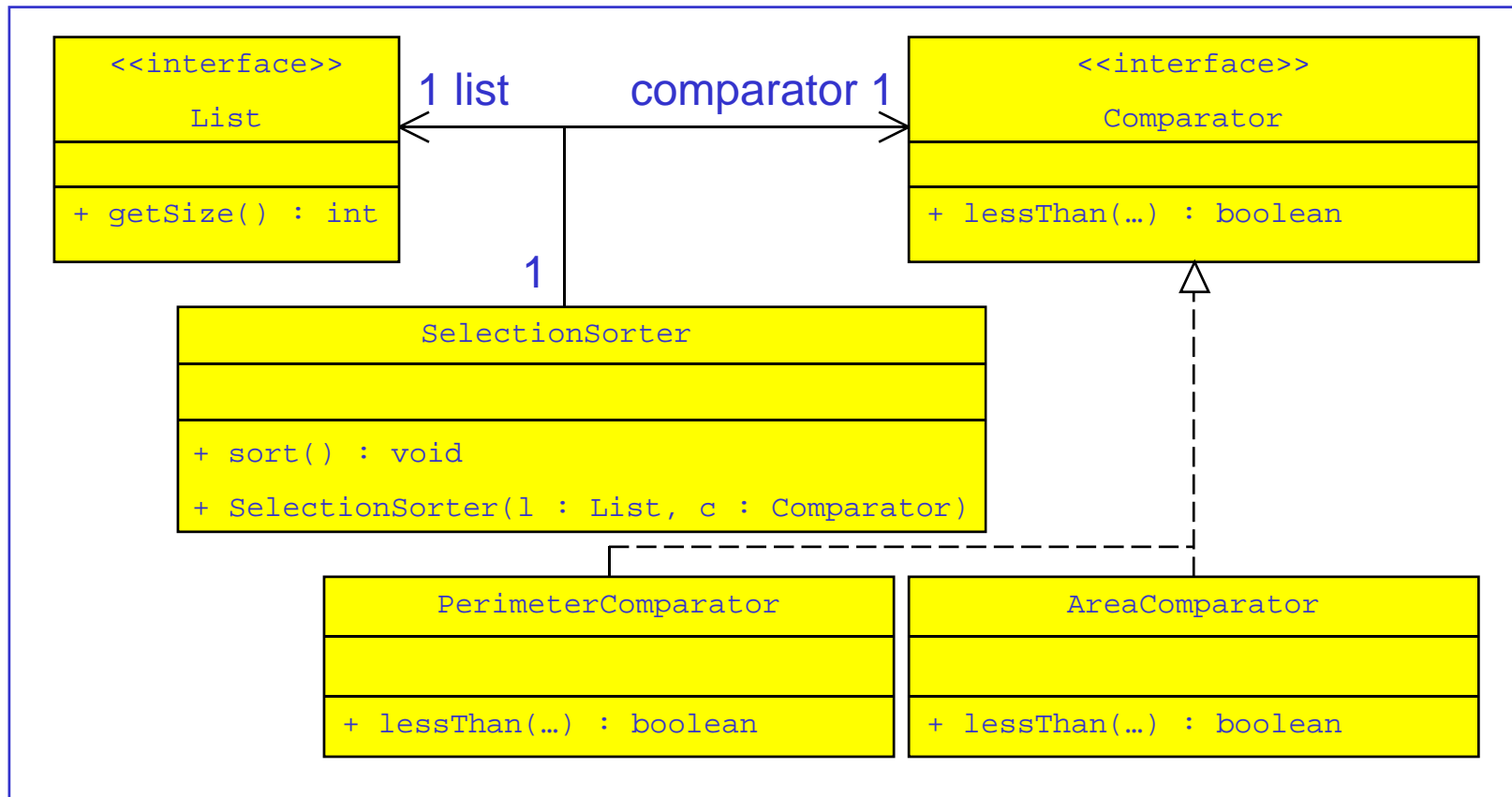
public interface List {
    public void    addElement(Element e);
    public void    setElement(int i, Element r);
    public Element getElement(int i);
    public int    getSize();
}

Public class ListOfPolygons implements List { ... }

public interface Element {}

public interface Polygon extends Element { ... }
  
```

17/List.java, ListOfPolygons.java, Element.java, Polygon.java



```
public class SelectionSorter {
    private Comparator comparator = null;
    private List list = null;

    public SelectionSorter(List l, Comparator c) {
        list = l;
        comparator = c;
    }

    public void sort() {
        for(int i = 0; i < list.getSize(); i++)
            for(int j = i + 1; j < list.getSize(); j++) {
                Element left = list.getElement(i);
                Element right = list.getElement(j);
                if(!comparator.lessThan(left, right)) swap(i, j);
            }
    }

    private void swap(int i, int j) {
        Element t = list.getElement(i);
        list.setElement(i, list.getElement(j));
        list.setElement(j, t);
    }
}
```

18/SelectionSorter.java

```
public interface Comparator {
    public boolean lessThan(Element e1, Element e2);
}

public class AreaComparator implements Comparator {
    public boolean lessThan(Element e1, Element e2) {
        Polygon p1 = (Polygon)e1;
        Polygon p2 = (Polygon)e2;
        return p1.area() < p2.area();
    }
}

public class PerimeterComparator implements Comparator {
    public boolean lessThan(Element e1, Element e2) {
        Polygon p1 = (Polygon)e1;
        Polygon p2 = (Polygon)e2;
        return p1.perimeter() < p2.perimeter();
    }
}
```

19/Comparator.java, AreaComparator.java, PerimeterComparator.java



- ♦ Java permette l'assegnazione di un oggetto di una classe,  $D$ , a una variabile il cui tipo è la classe base,  $B$ , della classe  $D$  (upcast)

```
B b = new D(); // upcast
```

- ♦ Inoltre, Java permette l'assegnazione del valore di una variabile di tipo  $B$ , ma il cui valore è un oggetto di tipo  $D$ , a una variabile di tipo  $D$  (downcast)

```
D d = (D)b; // downcast
```

- ◆ Quando il numero di classi aumenta, conviene strutturare il progetto in package
  - Un package è un insieme di classi che sono logicamente viste come un unico gruppo
  - Collezione di classi coese (e interdipendenti)
- ◆ È possibile strutturare i package come alberi
  - Un sottopackage isola un gruppo di funzionalità del suo package padre
- ◆ C'è una corrispondenza uno a uno tra i package e le directory dove sono memorizzate i file sorgenti
  - Le classi di un package sono tutti nella stessa cartella e questa cartella si deve chiamare come il package

- ♦ Il nome del package va dichiarato all'inizio del file di ogni sua classe
- ♦ Il nome del package deve essere scelto in modo da evitare conflitti con altri package
- ♦ Per usare una classe di un altro package, bisogna esplicitamente importare questa classe

```
package it.unipr.aotlab.isa.impl;  
  
import java.util.ArrayList;  
  
import it.unipr.aotlab.isa.model.*;  
  
...
```

- ♦ Solitamente un progetto è strutturato in cartelle:

```
src/ o src/java  
classes/  
images/  
lib/
```

- ♦ All'interno della cartella `src` c'è una cartella per ogni package

```
src/editor  
src/editor/model  
src/editor/view
```

- ♦ La directory `classes` rispecchia la struttura di `src`

	<b>attributo</b>	<b>metodo</b>	<b>classe</b>
<b>private</b>	nessuno	nessuno	non permessa
<b>protected</b>	sottoclassi e package	sottoclassi e package	non permessa
<b>public</b>	tutti	tutti	tutti
	package	package	package

- ♦ Una classe interna (inner class) è una classe la cui dichiarazione si trova all' interno di un' altra classe detta (classe ospitante)

```
class Impiegato {  
    String nomeDip;  
    double stipendio;  
  
    private class SalarityPropriety implements Propriety {  
        public String getNome() {  
            return nome; }  
        public String getVaue() {  
            String val="";  
            return val + stipendio; }  
    }  
    ...  
}
```

20/Impiegato.java

- ◆ Non possono avere il nome uguale a quello di una delle classi pubbliche
- ◆ Possono essere private, cioè invisibili all'esterno della classe ospite o protetta (visibili solo alle classi derivate dalla classe ospitante)
- ◆ Possono accedere a tutti i metodi e i campi della classe ospitante, mentre la classe ospitanti possono vedere solo la loro parte non privata
- ◆ Una loro istanza non può esistere se non esiste un oggetto della classe ospitante (non possono avere campi statici)

- ♦ Una classe locale è una classe definita all'interno di un metodo
- ♦ Una classe locale non solo non è visibile all'esterno della classe ospitante, ma neanche dagli altri metodi della classe ospite
- ♦ Nel caso in cui il tipo definito dalla classe locale non è necessario per accedere alle sue funzionalità (i.e., la classe è l'implementazione di una interfaccia) allora possiamo creare una classe senza nome



```
class Impiegato {  
    String nomeDip;  
    double stipendio;  
  
    public Propriety getSalarityPropriety() {  
        class SalarityPropriety() {  
            public String getNome() {  
                return nome; }  
            public String getVaue() {  
                String val="";  
                return val + stipendio; }  
        }  
        return new SalarityPropriety();  
    }  
    ...  
}
```

21/Impiegato.java

```
class Impiegato {
    String nomeDip;
    double stipendio;

    public Propriety getSalarityPropriety() {
        return new Propriety() {
            public String getNome() {
                return nome;
            }
            public String getVaue() {
                String val="";
                return val + stipendio;
            }
        };
    }
    ...
}
```

22/Impiegato.java

- ♦ Una libreria è un insieme di package che implementano funzionalità comuni
  - Tutte le classi con funzionalità matematiche avanzate
  - Tutte le classi necessarie a gestire l'input/output
  - Tutte le classi dell'interfaccia grafica
- ♦ Solitamente le librerie sono distribuite come file di tipo Java Archive (JAR)
  - È uno ZIP dei bytecode delle classi della libreria
  - Può essere gestito come un file ZIP o mediante il comando jar
    - jar cf lib.jar classes/\*.class image

- ◆ Per compilare un progetto è necessario includere
  - Tutte le cartelle dei sorgenti
  - Tutte le librerie esterne
- ◆ La linea di comando diventa

```
javac -d classes -cp classes;lib\lib1.jar; f1.java ... fn.java
```

```
javac -d classes -cp classes;lib\lib1.jar; *.java
```

```
javac -d classes -cp classes;lib\lib1.jar; @listfile.txt
```

- ◆ Per eseguire il programma è necessario includere
  - Tutte le directory contenenti i bytecode delle classi coinvolte nel programma
  - Tutti i file JAR delle librerie utilizzate
  - Tutte le directory contenenti delle risorse utilizzabili dal programma
- ◆ La linea di comando diventa

```
java -cp classes;lib\lib1.jar;images Main
```