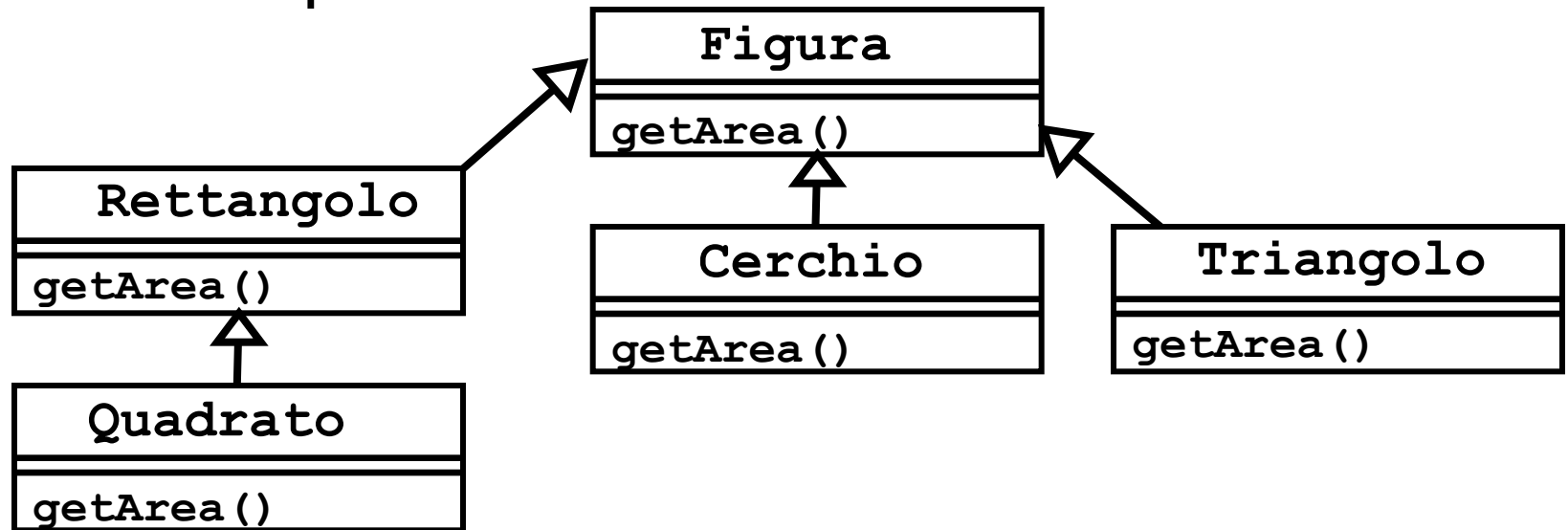


Classi astratte

- Nella gerarchia delle classi, potrebbe essere utile prevedere un “supertipo” generale per le classi che si usano.
- Ad esempio:



Classi astratte

- La classe **Figura** contiene un metodo **getArea()**, che dev'essere però sovrascritto da tutte le sue sottoclassi.
 - ▣ In effetti la classe **Figura** non ha un “suo” metodo per calcolare l'area.
 - ▣ Specificando che **Figura** contiene un metodo **getArea()** vogliamo semplicemente dire che una sottoclasse di **Figura** deve per forza avere un suo metodo **getArea()**.

Classi astratte

- Per svolgere la funzione di **getArea()** potremmo usare un metodo vuoto.
 - ▣ Ma così un programmatore distratto può scordarsi di sovrascriverlo, creando un errore logico.
- Java ci mette invece a disposizione un costrutto più valido: i **metodi astratti**.
- Un metodo astratto è solo dichiarato, ma non implementato. Quindi serve da “segnaposto” per ricordarci che se scriviamo delle classi estese, va inserito quel metodo.

Classi astratte

- Un metodo astratto è un metodo la cui implementazione è rimandata alle sottoclassi.
 - ▣ Dichiarandolo tale, il programmatore dice: manca qualcosa nell'implementazione della classe; e questo qualcosa non lo si può ereditare.
 - ▣ È quindi diverso da un metodo “che non fa nulla”. Quest'ultimo ha un'implementazione specifica, che determina il comportamento “non fare nulla”, il quale si può (anche erroneamente) ereditare.
- Invece un metodo astratto **non ha** implementazione.

Classi astratte

- Una classe che ha almeno un metodo astratto è una classe astratta.
 - ▣ Quindi, la sua implementazione non è completa.
 - ▣ Può comunque servire da base per sottoclassi.
 - ▣ Però una classe può anche essere astratta anche senza avere metodi astratti: vuol dire che riteniamo incompleta la sua implementazione.
- Metodi e classi astratte si denotano con il modificatore (parola chiave) **abstract**.

Classi astratte

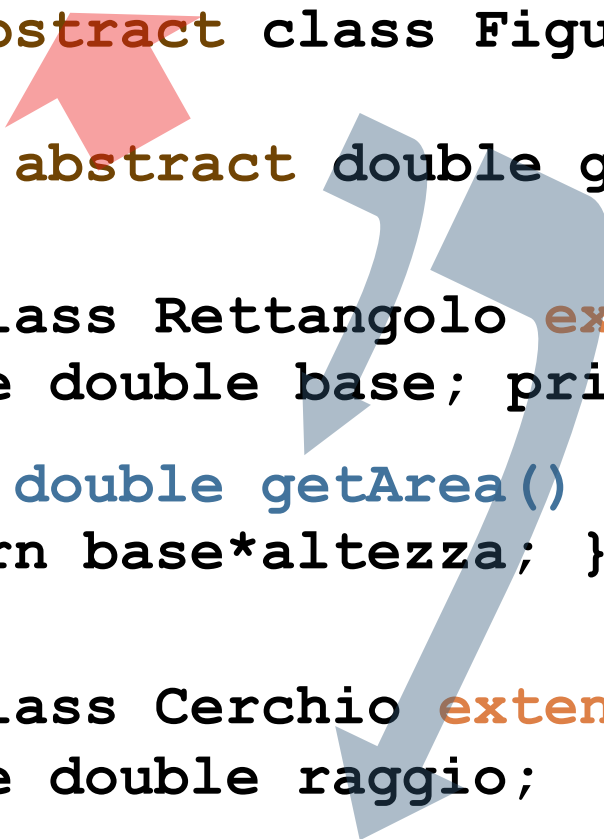
- Attenzione: se una classe ha uno o più metodi **abstract**, dev'essere **abstract** lei stessa.
- Un metodo astratto **deve** essere riscritto nelle sottoclassi (al contrario di un metodo **final** che **non si può** riscrivere).
- In realtà non è obbligatorio riscrivere il metodo, possiamo sempre lasciarlo stare.
 - ▣ In tal caso però, il metodo rimane astratto e quindi la sottoclasse rimarrà astratta.

Classi astratte

```
public abstract class Figura
{
    ...
    public abstract double getArea();
}

public class Rettangolo extends Figura
{
    private double base; private double altezza;
    ...
    public double getArea()
    { return base*altezza; }
}

public class Cerchio extends Figura
{
    private double raggio;
    ...
    public double getArea()
    { return raggio*raggio*Math.PI; }
}
```



Classi astratte

```
public abstract class Pietanza
{
    ...
    public abstract void mangia();
}

public abstract class PCarne extends Pietanza
{
    final int[] posata = {Tool.FORCHETTA,
        Tool.COLTELLO};
    ...
    public abstract void mangia();
}

public abstract class PPesce extends Pietanza
{
    final int[] posata = {Tool.FORCHETTA,
        Tool.COLTELLO_DA_PESCE};
    ...
}
```

Classi astratte

- Visto che le classi astratte non hanno un'implementazione completa, non possiamo istanziarne oggetti: sarebbero incompleti!
 - ▣ Talvolta, le classi di cui possiamo costruire oggetti sono anche dette **classi concrete**.

```
public abstract class Figura { ... }  
public class Cerchio extends Figura { ... }  
Figura f = new Figura(); // NO: astratta!  
Cerchio c = new Cerchio(); // OK: concreta
```

Classi astratte

- Tuttavia, nulla vieta di dichiarare un riferimento alla classe astratta, e metterlo in una variabile.
 - ▣ Questo riferimento può anche essere usato per un oggetto, naturalmente però l'oggetto deve essere un'istanza di una classe concreta.

```
public abstract class Figura { ... }  
public class Cerchio extends Figura { ... }  
Figura f; // OK: solo riferimento!  
f = null; // OK: riferimento lecito  
f = new Cerchio(); // OK: istanza concreta
```

Classi astratte

- Una classe astratta **deve** avere sempre **almeno** un costruttore.
 - ▣ Visto che la sua motivazione è quella di essere “estesa in seguito” ci vuole qualcosa per costruirne la parte che si eredita.
- Il costruttore non potrà però essere invocato con **new** (visto che non si possono creare oggetti istanza di questa classe astratta).
- Verrà perciò invocato univocamente dentro la sottoclasse (come **super ()**).

Classi astratte

- Una classe astratta può comunque possedere:
 - ▣ variabili istanza
 - ▣ metodi concreti (ovvero, non astratti)

```
public abstract class Figura
{ //campi
  protected String color;
  //costruttori
  public Figura(String x) {color = x;}
  //un metodo concreto: viene ereditato
  public String getColor() {return color;}
  //un metodo astratto: andrà sovrascritto
  public abstract double getArea();
}
```

Classi astratte

```
public class Cerchio extends Figura
{ //nuovi campi (oltre a color, ereditato)
  private double raggio;
  //costruttori
  public Cerchio(double r, String color)
  { super(color);
    this.raggio = r;
  }
  //un metodo concreto nuovo, oltre getColor
  public double getRaggio() {return raggio;}
  //sovrascriamo il metodo getArea
  public double getArea()
  { return raggio*raggio*Math.PI; }
}
```

Classi astratte

```
public class Rettangolo extends Figura
{ //nuovi campi (oltre a color, ereditato)
  private double base; private double altezza;
  //costruttori
  public Rettangolo(double b, double a, String col)
  { super(col); this.base = b; this.altezza = a; }
  //due metodi concreti nuovi, oltre getColor
  public double getBase() {return base;}
  public double getAltezza() {return altezza;}
  //sovrascriviamo il metodo getArea
  public double getArea()
  { return base*altezza; }
}
```

- Provare per esercizio a scrivere Quadrato.

Classi astratte

- Avendo dichiarato il metodo astratto **getArea()**, è possibile invocarlo anche su oggetti che hanno come tipo statico quello della classe base **Figura**.
 - ▣ Ovviamente il loro tipo dinamico sarà diverso, visto che non esistono oggetti istanza di **Figura**, che è una classe astratta.
- Ulteriore vantaggio rispetto a delegare anche la dichiarazione alle sottoclassi (non conviene fare affidamento sugli sviluppatori futuri)

Classi astratte

```
class Prova
{ public static void main(String[] args)
  { Cerchio c = new Cerchio(4,"blu");
    Figura f = new Cerchio(10,"rosso");

    String s = c.getColor(); //vale: blu
    double x = c.getArea(); //vale: 50.265
    double y = c.getRaggio(); //vale: 4.00

    s = f.getColor(); //vale: rosso
    x = f.getArea(); //vale: 314.159
    y = f.getRaggio(); //NO!!
  }
}
```

f non ha getRaggio()
nel tipo statico: errore
in fase di compilazione

f ha un getArea() sia
nel tipo statico che nel tipo
dinamico (quello che usa!)

Classi astratte

- Una sottoclasse può anche fare overriding di un metodo concreto della classe base, ridefinendolo come **metodo astratto**.
- In questo modo, il metodo in questione non è più visibile in tutto il sotto-albero a partire da quel punto.
- Serve a dire che ulteriori sotto-sottoclassi non devono ereditare quel metodo (perché non è più considerato valido).

Interfacce

- Java fornisce un ulteriore strumento, simile alle classi astratte, per implementare variazioni di classi e sfruttare il polimorfismo.
- Un'interfaccia è un'entità di programmazione che funge da schema (scheletro, prototipo) di una classe.
- È ancora più ridotta di una classe astratta, infatti possiede solo metodi pubblici e astratti; inoltre, non può avere variabili istanza.

Interfacce

- Un'interfaccia è diversa da una classe astratta sotto due aspetti generali.
 - ▣ **Teorico**: una classe combina **progetto** e **implementazione** degli oggetti (ciò vale anche per una classe astratta, la cui implementazione è tuttavia solo parziale); l'interfaccia è puro progetto
 - ▣ **Pratico**: una classe può essere estesa, garantendo così l'ereditarietà **singola** di Java. Grazie alle interfacce si può invece avere una sorta di ereditarietà **multipla**.

Interfacce

- Dentro a un'interfaccia si possono definire:
 - ▣ campi, ma solo se sono delle costanti; anche se non viene esplicitato, Java considererà eventuali campi di un'interfaccia come **static** e **final**;
 - ▣ metodi, che sono sempre **public** e **abstract**; questo anche se non sono esplicitati come tali
- Tutti i metodi di un'interfaccia devono essere **abstract**, quindi non hanno implementazione. La definizione di quest'ultima è lasciata alle classi che **implementano** l'interfaccia.

Interfacce

- Una classe **implementa** un'interfaccia se ne definisce tutti i metodi d'istanza.
 - ▣ Può comunque definirne solo alcuni: in tal caso la classe sarà astratta.
- Analogamente alle classi astratte, si può usare un'interfaccia per definire variabili riferimento.
 - ▣ Queste devono riferire oggetti di classi che implementano l'interfaccia.

Interfacce

- Una classe può estendere solo un'altra classe ma può implementare **più di un'interfaccia**.
- Così ci si avvicina all'ereditarietà multipla.
 - ▣ Infatti il problema principale è come comportarsi nel caso di conflitto tra due classi base “genitrici”. Il problema non si pone se una sola può essere una classe vera e propria, e le altre unità da cui provengono i metodi sono semplici interfacce.
- L'implementazione di un'interfaccia si dichiara con la parola chiave **implements**.

Interfacce

- La sintassi è ad esempio:

```
class X implements I1, I2, I3 { ... }
```

se la classe `x` implementa le tre interfacce `I1`, `I2` e `I3` (separate da virgole).

- Da notare però che in questo caso bisogna poi implementare tutti i metodi delle tre interfacce.
- Se c'è anche `extends`, va prima di `implements`.

```
class X extends B implements I { ... }
```

Interfacce

- Quando si estende o si implementa, si eredita sempre il tipo, ossia si può usare una variabile di tipo classe base (interfaccia implementata) per riferire un'istanza della classe estesa.
- La differenza è nei membri ereditati:
 - ▣ tutti, per le classi concrete
 - ▣ solo alcuni dei metodi, per le classi astratte (gli altri vanno specificati, o la classe rimarrà astratta)
 - ▣ nessuno, per le interfacce (a parte le costanti)

Interfacce

- Un'interfaccia promette certi comportamenti.
- Una classe che la implementa soddisfa queste promesse, cioè fornisce effettivamente i comportamenti promessi dall'interfaccia.
- La stessa interfaccia può essere implementata da classi differenti, anche se non sono in relazione diretta nella gerarchia.

Interfacce

- I metodi dell'interfaccia devono essere implementati con la stessa intestazione indicata dall'interfaccia. Quindi:
 - ▣ il prototipo è lo stesso
 - ▣ anche la firma è la stessa
 - ▣ anche i modificatori di visibilità sono gli stessi (pertanto i metodi saranno sempre **public**)
- Sono invece ininfluenti i **nomi** dei parametri: si possono cambiare con termini più espressivi.

Interfacce

- Il nome di un'interfaccia è anche un tipo.
 - ▣ Però non esistono istanze dell'interfaccia, come per le classi astratte
 - ▣ Il tipo interfaccia può essere un **tipo statico** di una variabile riferimento, ma non il **tipo dinamico** dell'oggetto che essa referencia, che dev'essere invece di una classe che implementa l'interfaccia.

Interfacce

- Le interfacce non hanno mai costruttori
 - ▣ Non servono: non devono essere istanziate e non hanno campi ereditabili da inizializzare
- Le interfacce possono a loro volta essere discendenti nella scala gerarchica ereditaria.
 - ▣ Un'interfaccia può estenderne altre (anche più d'una a differenza delle classi): a tale scopo si usa sempre la parola chiave **extends**

```
public interface I0 extends I1, I2, ...  
{ ... }
```

Interfacce marker

- Un'interfaccia può anche degenerare al caso di **interfaccia marker** se:
 - ▣ non contiene metodi e non contiene costanti
- Ha il solo scopo di “segnare” una possibile caratteristica di classe, un legame relazionale con la gerarchia delle classi.
 - ▣ In questo modo il contratto degenera alla sola documentazione, dove verrà descritto cosa ci si aspetta che faccia una classe implementatrice

L'interfaccia Comparable

- Alcuni algoritmi generali (visite di alberi, ordinamento di vettori,...) potrebbero avere bisogno di confrontare oggetti.
 - ▣ Come visto per **String**, sarebbe utile avere una relazione d'ordine **compareTo()** che confronta due oggetti **a** e **b** invocando **a.compareTo(b)**
- Non possiamo però metterla in **Object**.
 - ▣ Non sappiamo come fare i confronti, e la erediterebbero tutti: potremmo metterla in una unità apposita, che chiameremo **Comparable**.

L'interfaccia Comparable

- Sembra semplice: se vogliamo oggetti con una relazione d'ordine, dichiariamo la loro classe come sottoclasse di **Comparable**.
- Unico suo metodo dev'essere **compareTo()**:
 - ▣ deve ritornare un **int** <0 , $=0$, >0 a seconda che sia **$a < b$** , **$a == b$** , **$a > b$** .
- E se gli oggetti hanno già una superclasse?
 - ▣ Non si può ereditare da più superclassi.
 - ▣ Ma **Comparable** deve avere solo **compareTo...**

L'interfaccia Comparable

- Soluzione: **Comparable** è un'interfaccia
 - ▣ È contenuta in `java.lang`
- Il suo unico metodo `compareTo()` va ridefinito in ogni classe che la implementa.
 - ▣ L'interfaccia **Comparable** promette un metodo `public int compareTo(Object x)`. Non può scriverlo: non sa confrontare il generico parametro implicito con `x` (che è un `Object`)
 - ▣ Sarà l'implementazione, a dire (oltre a qual è la relazione d'ordine) come fare cast dell'oggetto `x`.

L'interfaccia Comparable

- All'interno del package `java.lang`, l'interfaccia `Comparable` è definita come:

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



unico metodo: senza bisogno di dirlo esplicitamente, è per forza **public** e **abstract** come ogni metodo di un'interfaccia

L'interfaccia Comparable

- Ad esempio, la classe `String`, come si è visto, implementa `Comparable`. Anche la classe `Integer` (classe involucro degli interi) lo fa.
 - ▣ In tal caso, il parametro di `compareTo()` va rispettivamente “castato” a `String` e `int`
- Visto che il parametro passato è di tipo `Object`, potremmo passare qualunque cosa.
 - ▣ Ma se il parametro e l'oggetto `this` non sono confrontabili, il metodo `compareTo()` solleverà un'eccezione di tipo `ClassCastException`.

L'interfaccia Comparable

- Ogni classe su cui ha senso definire una **relazione d'ordine totale** dovrebbe implementare l'interfaccia **Comparable**.
 - ▣ Per farlo, occorre trovare un criterio opportuno per ritornare un valore negativo, nullo, positivo a seconda che la relazione sia $<$, $=$, $>$.
- Il metodo **compareTo()** allora è detto **metodo di confronto naturale** della classe, e l'ordinamento che ne risulta è detto **ordinamento naturale** della classe.

compareTo() e equals()

- L'ordinamento naturale di una classe è detto **consistente con equals()** se e solo se
$$o1.compareTo(o2) == 0$$
ha lo stesso valore booleano di
$$o1.equals(o2)$$
per ogni coppia $o1, o2$ di oggetti della classe.
- Questa identità non è obbligatoria, ma è **fortemente raccomandata** per evitare strani comportamenti di alcuni metodi.

L'interfaccia Comparable

- L'interfaccia `Comparable` usa un ordinamento “naturale”. A volte l'ordinamento non è tanto naturale, oppure esistono più ordinamenti.
 - ▣ Un elenco di studenti può essere ordinato per matricola, nome, cognome, media dei voti... e in ordine crescente/decrescente
- C'è un'altra interfaccia `Comparator` con un solo metodo `int compare(Object a, Object b)`
- Stesse regole di `compareTo()` ma con due parametri espliciti: così è uno schema del **metodo di confronto**, non dell'oggetto confrontabile