

## Capitolo 6

### Uso della gerarchia



# Sommario:

Notazione UML per le classi

Introduzione alla gerarchia

Gerarchia e tipi

La gerarchia di Java

La classe Rettangolo

La classe Quadrato

L'operatore instanceof

La classe Cerchio

La classe astratta Figura

Gerarchia: promozioni e cast

Le interfacce

L'interfaccia Iterable e il ciclo for-each

Tipi generici e gerarchia

Vincoli sui segnaposto

Tipi generici e vincoli sugli argomenti

# UML - rappresentazione delle classi

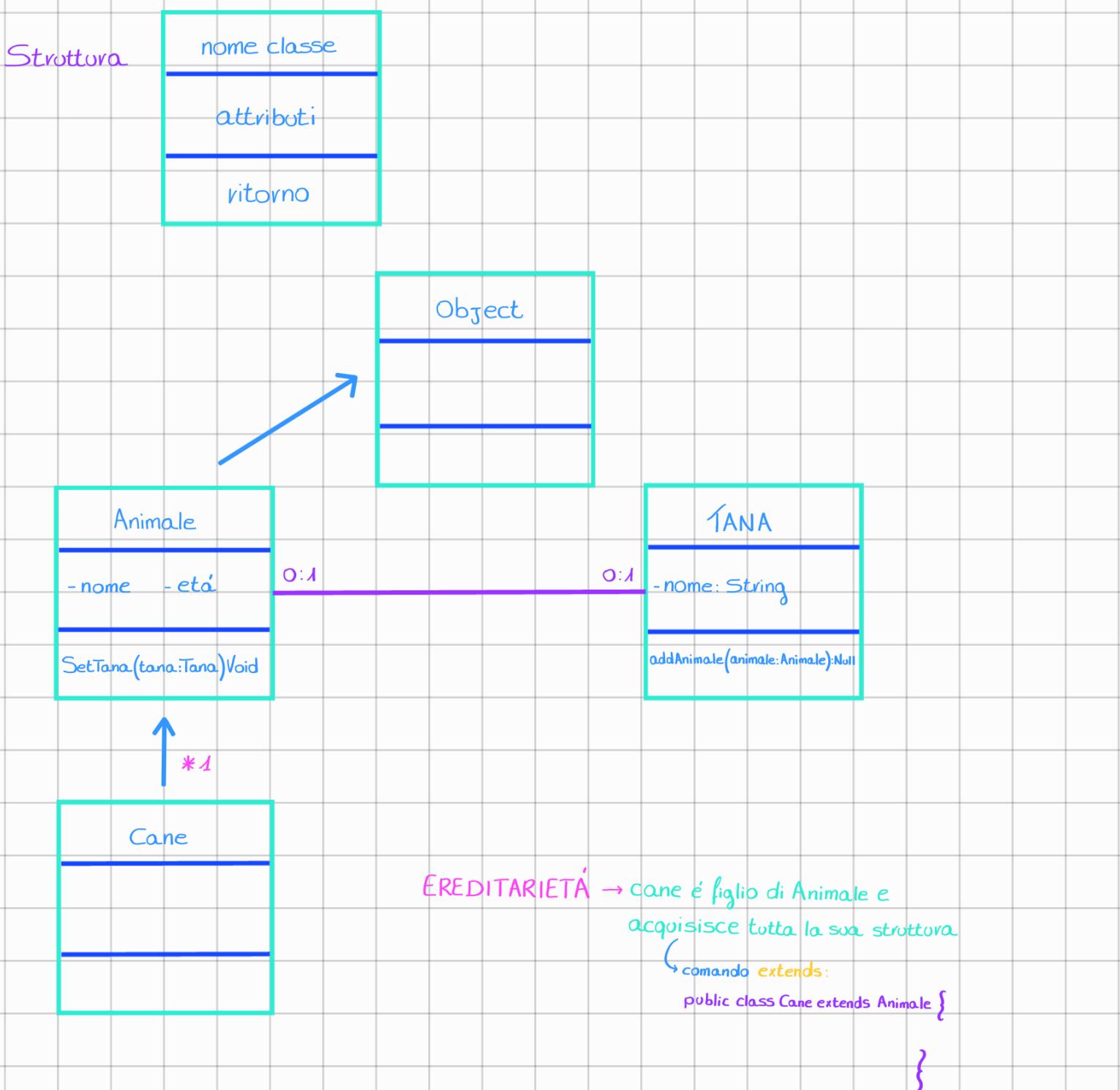
## Rappresentazione concisa

```
Rettangolo
```

## Rappresentazione estesa

```
Rettangolo
+Rettangolo(x:double,y:double)
+getAltezza(): double
+getBase(): double
+getArea(): double
+getPerimetro(): double
+haAreaMaggiore(r:Rettangolo): boolean
+haPerimetroMaggiore(r:Rettangolo): boolean
+equals(r:Rettangolo): boolean
+toString(): String
```

UML. ci sono vari diagrammi, è più precisamente una CLASSE di diagrammi (useremo Class Diagram)



\*1 Ereditarietà → Cane è figlio di Animale e acquisisce tutta la sua struttura

COMANDO: extends

`public class Cane extends Animale`

assume tutti gli attributi di Animale

Ereditarietà: Relazione particolare, cioè Cane è un animale, è un sottoinsieme.

Altre relazioni: Due insiemi hanno una relazione, comunicano, senza che una sia un sottoinsieme dell'altra.

## Operazioni

*nome\_operazione* (*lista\_dei\_parametri*) : *tipo\_restituito*

- ▶ *lista\_dei\_parametri*

È una sequenza di dichiarazioni di parametro separate da virgole

# Operazioni e parametri

## Operazioni

*nome\_operazione* (*lista\_dei\_parametri*) : *tipo\_restituito*

- ▶ *lista\_dei\_parametri*

È una sequenza di dichiarazioni di parametro separate da virgolette

## Parametri

*nome\_parametro* : *tipo*

# Operazioni e parametri

## Operazioni

*nome\_operazione* (*lista\_dei\_parametri*) : *tipo\_restituito*

- ▶ *lista\_dei\_parametri*

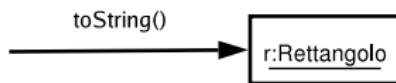
È una sequenza di dichiarazioni di parametro separate da virgole

## Parametri

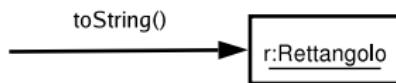
*nome\_parametro* : *tipo*

- ▶ +

Indica che la visibilità dell'operazione è pubblica (**public**)



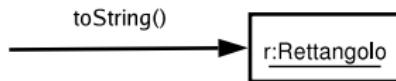
- ▶ All'interno del riquadro che rappresenta l'oggetto viene inserito il nome dell'istanza



- ▶ All'interno del riquadro che rappresenta l'oggetto viene inserito il nome dell'istanza

## Sintassi

*nomelistanza : nome\_classe*



- ▶ All'interno del riquadro che rappresenta l'oggetto viene inserito il nome dell'istanza

## Sintassi

*nomelistanza : nome\_classe*

- ▶ Sia il nome dell'oggetto che il nome della classe sono opzionali

- ▶ Supponiamo di voler pagare la torta tramite la carta di credito:

`pagamentoCC(numeroCarta)`

- ▶ Supponiamo di voler pagare la torta tramite la carta di credito:  
`pagamentoCC(numeroCarta)`
- ▶ Ogni pasticceria mette a disposizione questo metodo, quindi ci aspettiamo che sia specificato nella classe `Pasticceria`

- ▶ Supponiamo di voler pagare la torta tramite la carta di credito:  
`pagamentoCC(numeroCarta)`
- ▶ Ogni pasticceria mette a disposizione questo metodo, quindi ci aspettiamo che sia specificato nella classe **Pasticceria**
- ▶ Quando un oggetto riceve un messaggio che compare nel suo protocollo, risponde a tale messaggio eseguendo la sequenza di operazioni (il codice) del metodo relativo

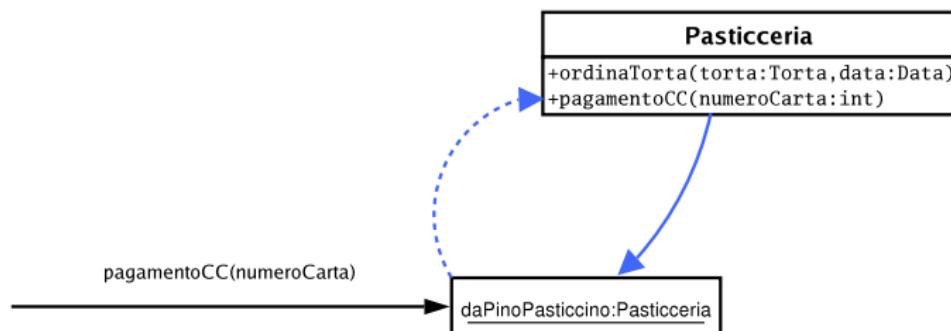
## Esecuzione del metodo

```
Pasticceria daPinoPasticcino = new Pasticceria();  
...  
daPinoPasticcino.pagamentoCC(numeroCarta);
```

# Esecuzione del metodo

```
Pasticceria daPinoPasticcino = new Pasticceria();  
...  
daPinoPasticcino.pagamentoCC(numeroCarta);
```

La JVM cerca nella classe **Pasticceria** il metodo da eseguire:



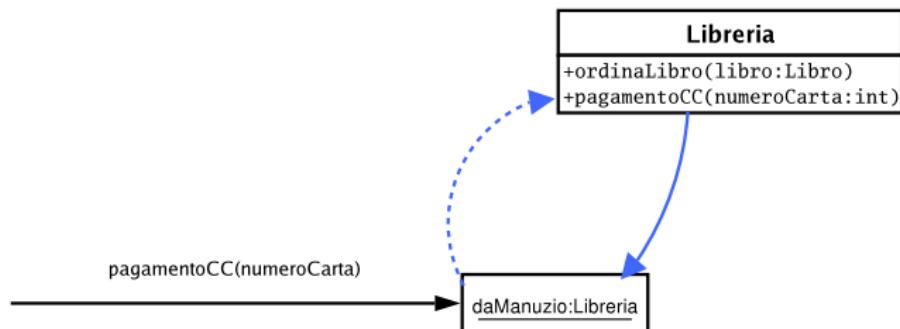
## Supponiamo ora di andare in libreria...

```
Libreria daManuzio = new Libreria();
...
daManuzio.pagamentoCC(numeroCarta);
```

# Supponiamo ora di andare in libreria...

```
Libreria daManuzio = new Libreria();  
...  
daManuzio.pagamentoCC(numeroCarta);
```

La JVM cerca nella classe **Libreria** il metodo da eseguire:

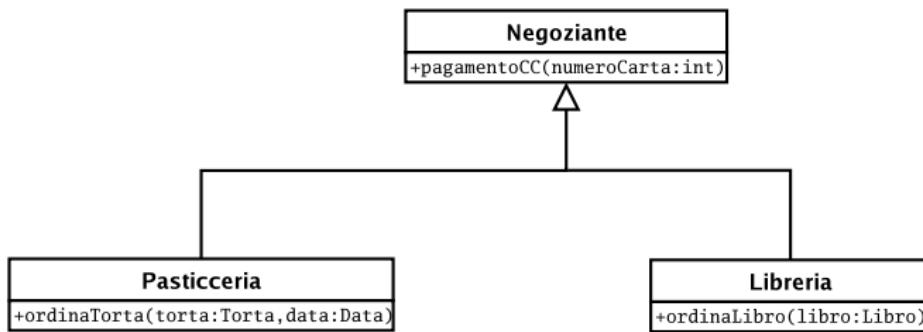


- ▶ Spesso i concetti vengono rappresentati in modo gerarchico

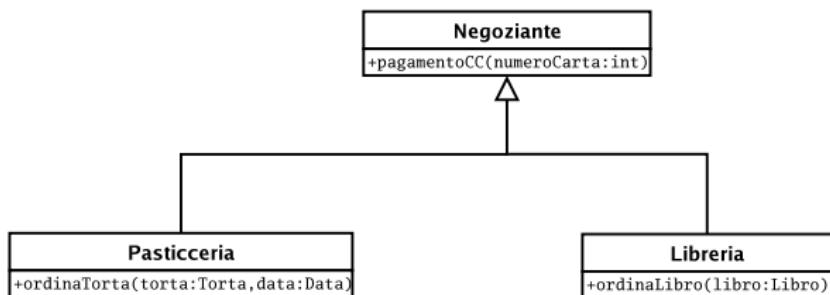
- ▶ Spesso i concetti vengono rappresentati in modo gerarchico
- ▶ Il messaggio `pagamentoCC` è proprio di una **categoria più ampia**, quella dei **Negozianti** (nuovo concetto/classe)

- ▶ Spesso i concetti vengono rappresentati in modo gerarchico
- ▶ Il messaggio `pagamentoCC` è proprio di una **categoria più ampia**, quella dei **Negozianti** (nuovo concetto/classe)
- ▶ Le proprietà ed i comportamenti di una categoria di oggetti vengono **ereditate** dalle **sotto-categorie**

- ▶ Spesso i concetti vengono rappresentati in modo gerarchico
- ▶ Il messaggio `pagamentoCC` è proprio di una **categoria più ampia**, quella dei **Negozianti** (nuovo concetto/classe)
- ▶ Le proprietà ed i comportamenti di una categoria di oggetti vengono **ereditate** dalle **sotto-categorie**

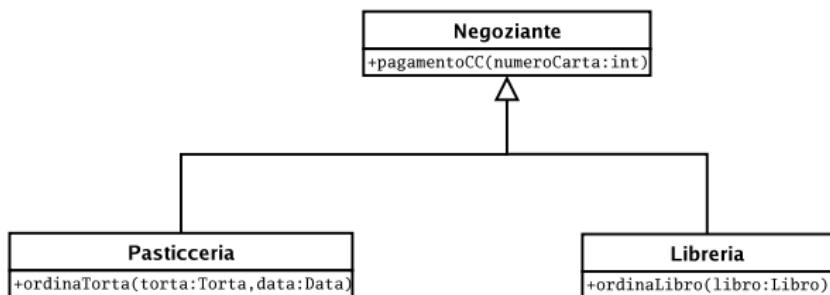


# Ricerca del metodo da eseguire



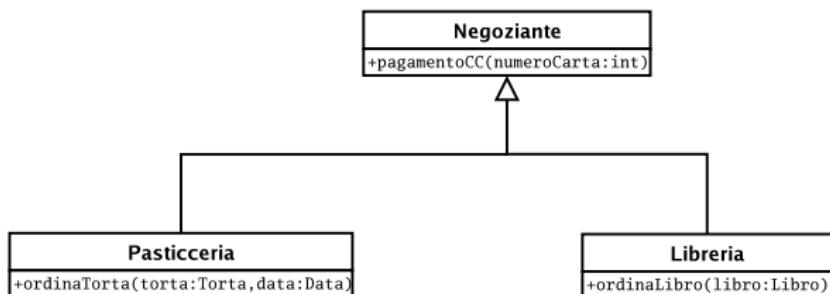
- ▶ Il metodo è definito nella classe **Negoziante** (**superclasse**)

# Ricerca del metodo da eseguire



- ▶ Il metodo è definito nella classe **Negoziante** (**superclasse**)
- ▶ **Pasticceria** e **Libreria** (**sottoclassi**) lo ereditano

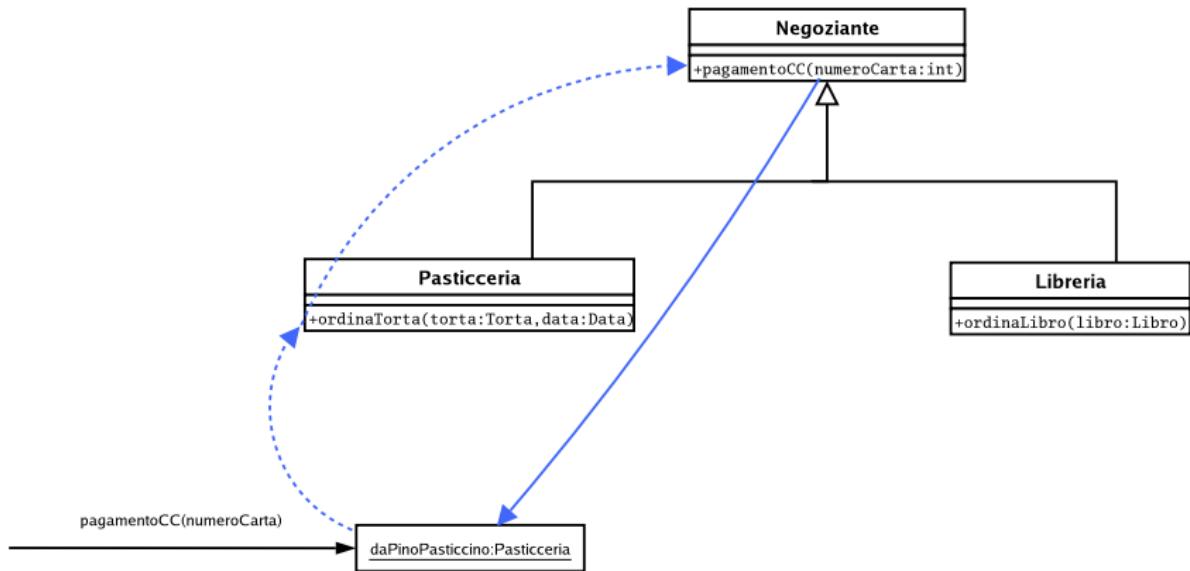
# Ricerca del metodo da eseguire



- ▶ Il metodo è definito nella classe **Negoziante** (**superclasse**)
- ▶ **Pasticceria** e **Libreria** (**sottoclassi**) lo ereditano
- ▶ La JVM deve **cercare nella gerarchia** il codice da eseguire in risposta ad un metodo

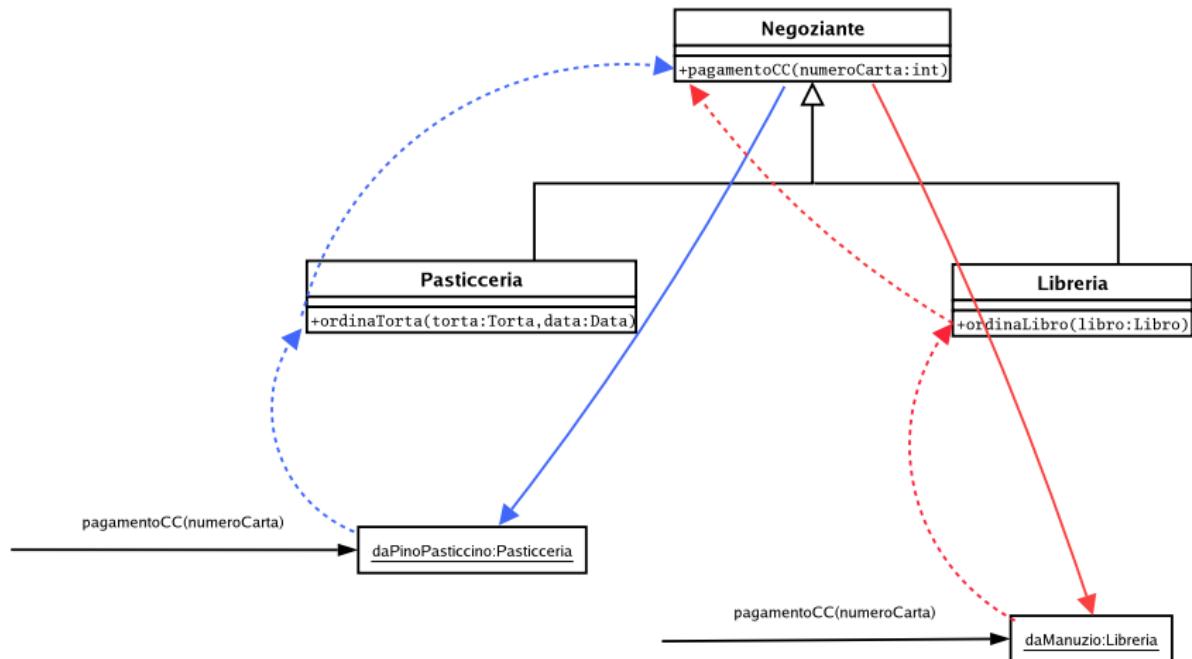
# Ricerca del metodo

```
daPinoPasticcino.pagamentoCC(numeroCarta);
```



# Ricerca del metodo

```
daPinoPasticcino.pagamentoCC(numeroCarta);  
daManuzio.pagamentoCC(numeroCarta);
```



- ▶ Una **classe** definisce un **tipo** i cui valori sono tutte le istanze possibili della classe

- ▶ Una **classe** definisce un **tipo** i cui valori sono tutte le istanze possibili della classe
- ▶ Possiamo pensare che il tipo di un oggetto è determinato dall'insieme dei messaggi a cui può rispondere

- ▶ Una **classe** definisce un **tipo** i cui valori sono tutte le istanze possibili della classe
- ▶ Possiamo pensare che il tipo di un oggetto è determinato dall'insieme dei messaggi a cui può rispondere
- ▶ A un oggetto di **tipo A** possiamo inviare tutti i messaggi specificati nella **classe A**

- ▶ Una **classe** definisce un **tipo** i cui valori sono tutte le istanze possibili della classe
- ▶ Possiamo pensare che il tipo di un oggetto è determinato dall'insieme dei messaggi a cui può rispondere
- ▶ A un oggetto di **tipo A** possiamo inviare tutti i messaggi specificati nella **classe A**

A `daPinoPasticcino` possiamo inviare tutti i messaggi specificati nella classe `Pasticceria`.

- ▶ In presenza di ereditarietà tutti i metodi delle superclassi vengono ereditati dalle sottoclassi

- ▶ In presenza di ereditarietà tutti i metodi delle superclassi vengono ereditati dalle sottoclassi

A `daPinoPasticcino` possiamo inviare anche tutti i messaggi definiti nella classe `Negoziante`

- ▶ In presenza di ereditarietà tutti i metodi delle superclassi vengono ereditati dalle sottoclassi

A `daPinoPasticcino` possiamo inviare anche tutti i messaggi definiti nella classe `Negoziante`

- ▶ Il tipo determinato dalla sottoclasse è un **sottotipo** del tipo determinato dalla superclasse.

- ▶ In presenza di ereditarietà tutti i metodi delle superclassi vengono ereditati dalle sottoclassi

A `daPinoPasticcino` possiamo inviare anche tutti i messaggi definiti nella classe `Negoziante`

- ▶ Il tipo determinato dalla sottoclasse è un **sottotipo** del tipo determinato dalla superclasse.

Tutti gli oggetti della `sottoclasse` possono essere trattati anche come oggetti della `superclasse`

## Relazione 'è un'

La relazione **sottoclasse/superclasse** definisce una relazione di tipo '**è un**'.

## Relazione 'è un'

La relazione **sottoclasse/superclasse** definisce una relazione di tipo '**è un**'.

Ogni oggetto della sottoclasse **è un** oggetto della superclasse.

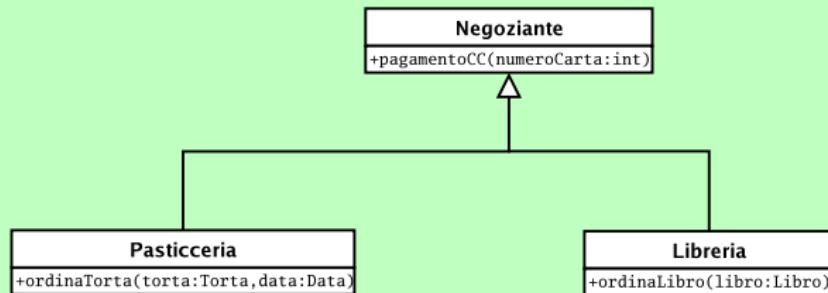
# Relazione 'è un'

La relazione **sottoclasse/superclasse** definisce una relazione di tipo '**è un**'.

Ogni oggetto della sottoclasse **è un** oggetto della superclasse.

daPinoPasticcino è *un* Negoziante

```
Pasticceria daPinoPasticcino = new Pasticceria();
```



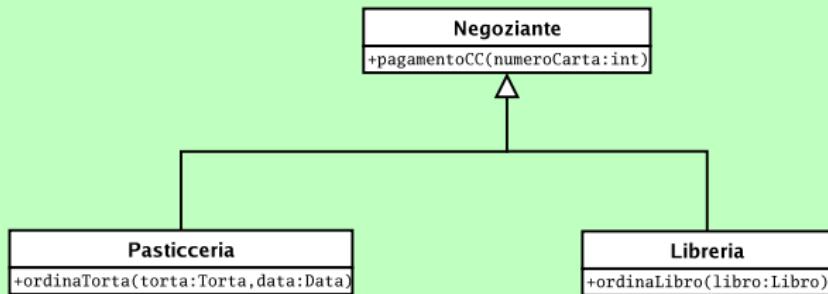
## Relazione 'è un'

Possiamo assegnare ad una variabile del tipo della **superclasse** un oggetto della **sottoclasse**.

# Relazione 'è un'

Possiamo assegnare ad una variabile del tipo della **superclasse** un oggetto della **sottoclasse**.

## Esempio



```
Negoziante n;
```

```
n = new Pasticceria();
```

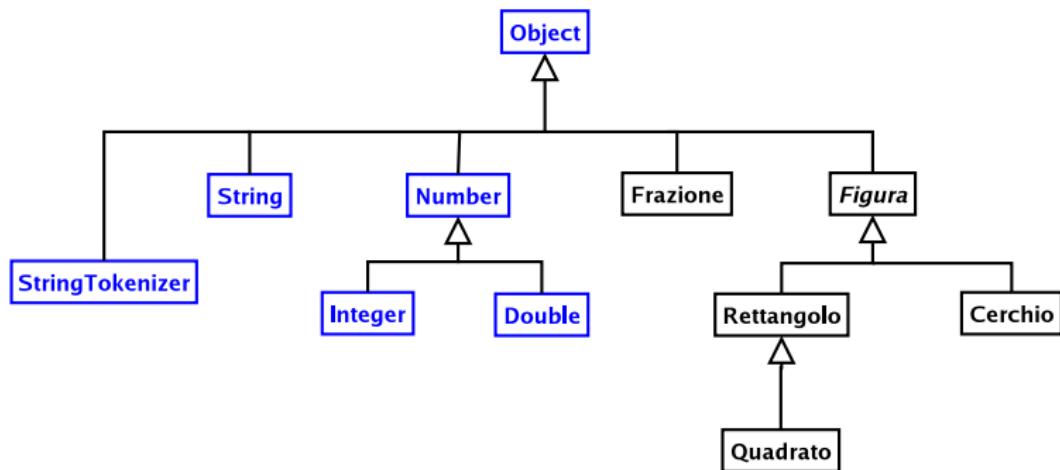
```
n = new Libreria();
```

Comando "Super": Sintassi Super (attributi)

Anche se gli attributi sono PRIVATI, li posso prendere comunque.

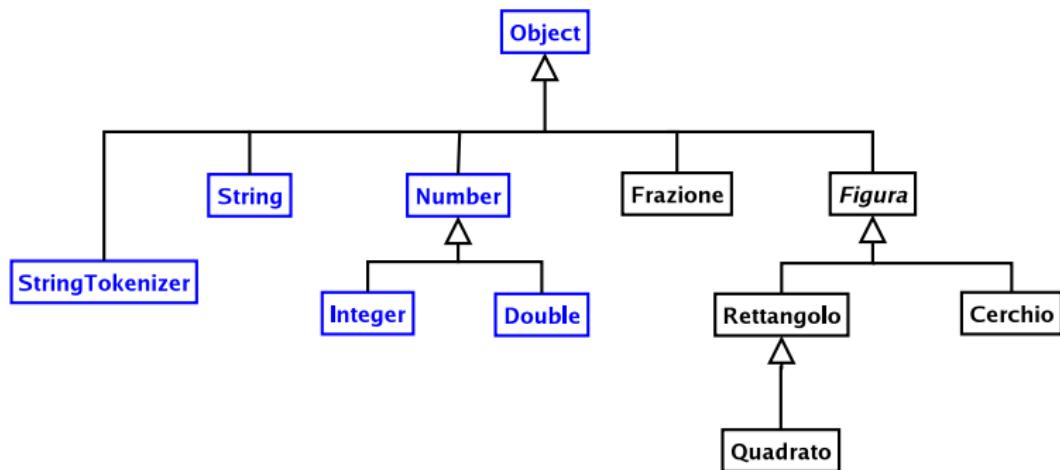
# La gerarchia di Java

È organizzata ad albero:



# La gerarchia di Java

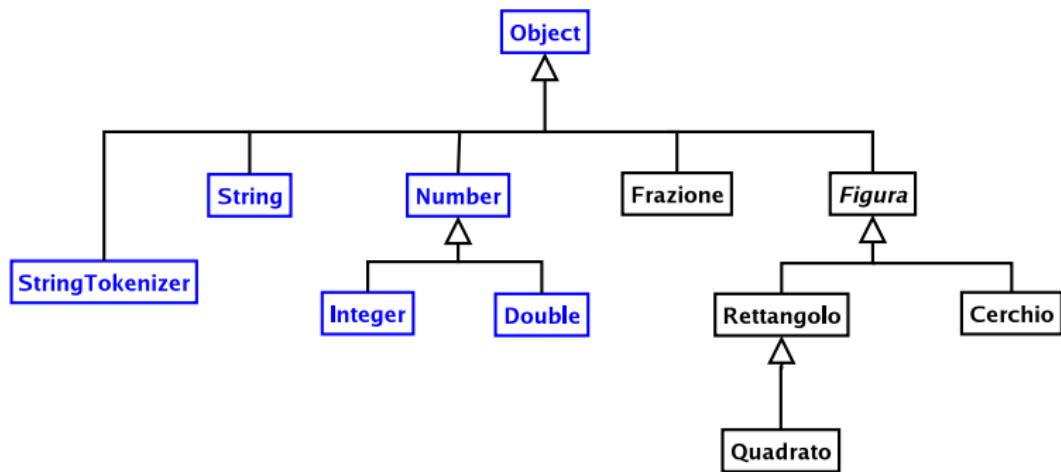
È organizzata ad albero:



- ▶ Ogni classe estende **al più** una classe (la sua superclasse diretta)

# La gerarchia di Java

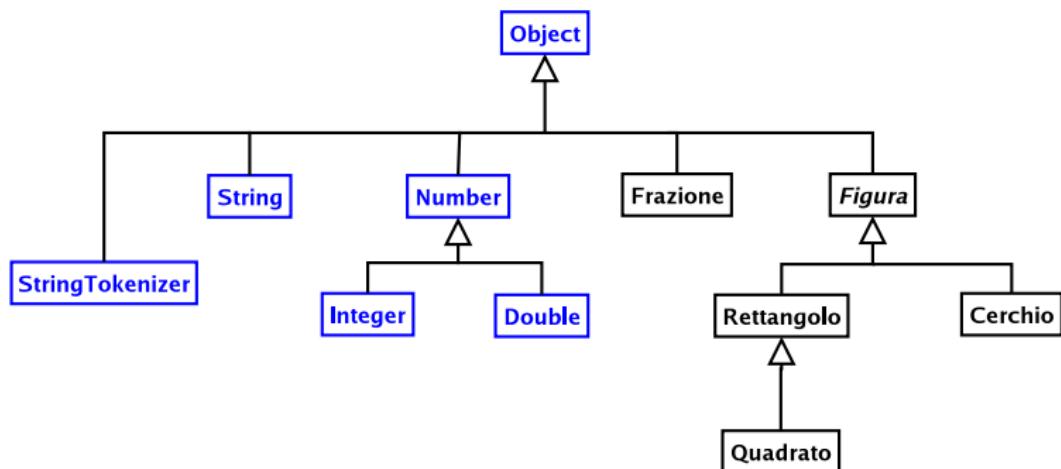
È organizzata ad albero:



- ▶ Ogni classe estende **al più** una classe (la sua superclasse diretta)
- ▶ Ogni classe può essere estesa da **più** sottoclassi

# La gerarchia di Java

È organizzata ad albero:



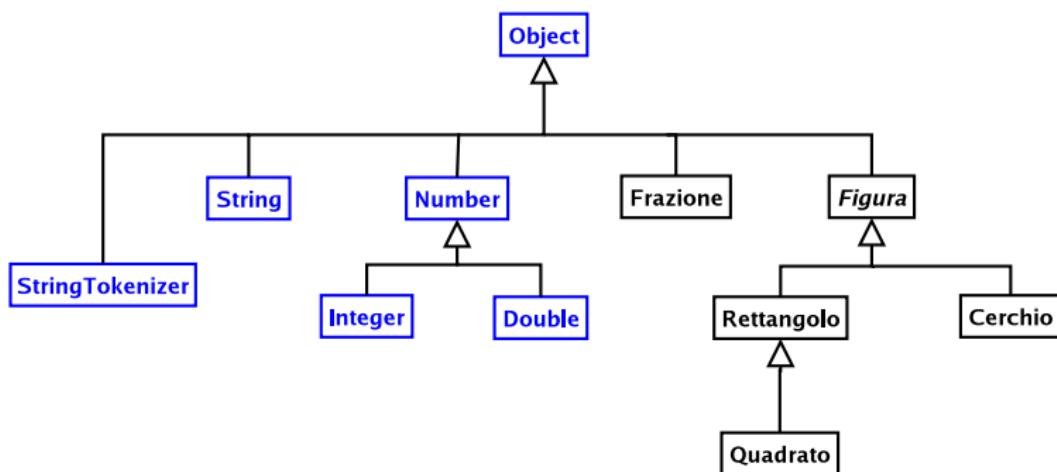
JAVA NON SUPPORTA EREDITARIETÀ MULTIPLA

- ▶ Ogni classe estende **al più** una classe (la sua superclasse diretta)
- ▶ Ogni classe può essere estesa da **più** sottoclassi
- ▶ Ha una radice: la classe **Object** (`java.lang`)

# La gerarchia di Java: Object

## Metodi

- ▶ `public String toString()`
- ▶ `public boolean equals(Object o)`
- ▶ ...



## Rettangolo

Le sue istanze rappresentano rettangoli.

## Rettangolo

Le sue istanze rappresentano rettangoli.

## Costruttori

► `public Rettangolo(double x, double y)`

Costruisce un oggetto che rappresenta un rettangolo, la cui base e la cui altezza hanno le lunghezze fornite, rispettivamente, tramite il primo e il secondo argomento.

## Metodi

- ▶ public double getArea()
- ▶ public double getPerimetro()
- ▶ public boolean equals(Rettangolo r)
- ▶ public boolean haAreaMaggiore(Rettangolo r)
- ▶ public boolean haPerimetroMaggiore(Rettangolo r)
- ▶ public double getBase()
- ▶ public double getAltezza()
- ▶ public String toString()  
"base = 3.4, altezza = 1.0"

# ProvaRettangolo

```
...
//lettura dati
out.println("Inserire i dati del rettangolo:");
double b = in.readDouble("base? ");
double a = in.readDouble("altezza? ");
```

# ProvaRettangolo

```
...
//lettura dati
out.println("Inserire i dati del rettangolo:");
double b = in.readDouble("base? ");
double a = in.readDouble("altezza? ");

//costruzione dell'oggetto
Rettangolo r = new Rettangolo(b, a);
```

# ProvaRettangolo

```
...
//lettura dati
out.println("Inserire i dati del rettangolo:");
double b = in.readDouble("base? ");
double a = in.readDouble("altezza? ");

//costruzione dell'oggetto
Rettangolo r = new Rettangolo(b, a);

//comunicazione del risultato
out.print("Rettangolo letto: ");
out.println(r.toString());
out.println("L'area e' " + r.getArea());
out.println("Il perimetro e' " + r.getPerimetro());
...
...
```

# ProvaRettangolo

```
...
//lettura dati
out.println("Inserire i dati del rettangolo:");
double b = in.readDouble("base? ");
double a = in.readDouble("altezza? ");

//costruzione dell'oggetto
Rettangolo r = new Rettangolo(b, a);

//comunicazione del risultato
out.print("Rettangolo letto: ");
out.println(r.toString());
out.println("L'area e' " + r.getArea());
out.println("Il perimetro e' " + r.getPerimetro());
...
...
```

Cosa succede se l'applicazione riceve base o altezza negative?

# ProvaRettangolo

```
...
//lettura dati
double b, a;
out.println("Inserire i dati del rettangolo:");

while ((b = in.readDouble("base? ")) < 0)
    out.println("Attenzione: la base di un rettangolo " +
        "non puo' essere negativa!");
```

# ProvaRettangolo

```
...
//lettura dati
double b, a;
out.println("Inserire i dati del rettangolo:");

while ((b = in.readDouble("base? ")) < 0)
    out.println("Attenzione: la base di un rettangolo " +
                "non puo' essere negativa!");

while ((a = in.readDouble("altezza? ")) < 0)
    out.println("Attenzione: l'altezza di un rettangolo " +
                "non puo' essere negativa!");
```

# ProvaRettangolo

```
...
//lettura dati
double b, a;
out.println("Inserire i dati del rettangolo:");

while ((b = in.readDouble("base? ")) < 0)
    out.println("Attenzione: la base di un rettangolo " +
                "non puo' essere negativa!");

while ((a = in.readDouble("altezza? ")) < 0)
    out.println("Attenzione: l'altezza di un rettangolo " +
                "non puo' essere negativa!");

//costruzione dell'oggetto
Rettangolo r = new Rettangolo(b, a);

//comunicazione del risultato
...
```

## Esempio

- ▶ Si scriva un'applicazione per determinare il rettangolo con area maggiore in una sequenza di rettangoli fornita da tastiera.

## Esempio

- ▶ Si scriva un'applicazione per determinare il rettangolo con area maggiore in una sequenza di rettangoli fornita da tastiera.
- ▶ Se vi sono più rettangoli con la stessa area, maggiore di quella degli altri, l'applicazione comunicherà i dati di uno qualunque di essi.

# Esempio

- ▶ Si scriva un'applicazione per determinare il rettangolo con area maggiore in una sequenza di rettangoli fornita da tastiera.
- ▶ Se vi sono più rettangoli con la stessa area, maggiore di quella degli altri, l'applicazione comunicherà i dati di uno qualunque di essi.

## Schema

```
do {  
    leggi i dati di un rettangolo  
    se il rettangolo letto ha area maggiore dei precedenti  
        memorizzane il riferimento  
} while (l'utente vuole inserire un altro rettangolo)  
comunica i dati relativi al rettangolo di area maggiore
```

## Sottoprogramma per la lettura di un rettangolo

```
private static Rettangolo leggiRettangolo(ConsoleInputManager in,
                                         ConsoleOutputManager out) {
    //leggi i dati del rettangolo
    out.println("Inserisci i dati di un rettangolo:");
    double x = in.readDouble(" - base? ");
    double y = in.readDouble(" - altezza? ");
    while (x < 0 || y < 0) {
        out.println("I dati inseriti non rappresentano un rettangolo.");
        out.println("Inserisci i dati di un rettangolo:");
        x = in.readDouble(" - base? ");
        y = in.readDouble(" - altezza? ");
    }
    Rettangolo r = new Rettangolo(x, y);
    out.println("Rettangolo:");
    out.println(" " + r.toString());
    out.println(" area = " + r.getArea() +
               ", perimetro = " + r.getPerimetro());
    out.println();
    return r;
}
```

## RettangoloAreaMax

```
...
Rettangolo rAreaMax = null;
boolean continuare;

do {
    //leggi i dati di un rettangolo
    Rettangolo r = leggiRettangolo(in, out);

    //confronta il rettangolo con quello di area maggiore
    if (rAreaMax == null || r.haAreaMaggiori(rAreaMax))
        rAreaMax = r;

    continuare = in.readSiNo("Vuoi inserire i dati di un altro " +
                            "rettangolo? (s/n) ");
} while (continuare);

//comunica le caratteristiche del rettangolo di area maggiore
out.println("Rettangolo di area maggiore: ");
out.println(" " + rAreaMax.toString());
out.println(" area = " + rAreaMax.getArea() +
            ", perimetro = " + rAreaMax.getPerimetro());
...
...
```

# La classe Quadrato

## Quadrato

Le sue istanze rappresentano quadrati.

# La classe Quadrato

## Quadrato

Le sue istanze rappresentano quadrati.

## Costruttori

- ▶ **public Quadrato(double x)**

Costruisce un oggetto che rappresenta un quadrato il cui lato ha la lunghezza fornita tramite il parametro.

## Metodi

- ▶ public double getArea()
- ▶ public double getPerimetro()
- ▶ public boolean equals(Quadrato q)
- ▶ public boolean haAreaMaggiore(Quadrato q)
- ▶ public boolean haPerimetroMaggiore(Quadrato q)
- ▶ public double getLato()
- ▶ public String toString()  
    "lato = 5.1"

## Esempio: estensione di RettangoloAreaMax

- ▶ Si scriva un'applicazione per determinare la figura con area maggiore in una sequenza di *rettangoli e quadrati* fornita da tastiera.

## Esempio: estensione di RettangoloAreaMax

- ▶ Si scriva un'applicazione per determinare la figura con area maggiore in una sequenza di *rettangoli e quadrati* fornita da tastiera.

```
... rAreaMax = null;  
boolean continuare;  
...  
  
do {  
    //legge i dati di una figura (rettangolo o quadrato)  
    ... r = ...lettura della figura...  
  
    //confronta la figura con quella di area maggiore  
    if (rAreaMax == null || r.haAreaMaggiori(rAreaMax))  
        rAreaMax = r;  
    ...  
} while (continuare);  
  
...comunicazione dei risultati
```

## Esempio: estensione di RettangoloAreaMax

- ▶ Si scriva un'applicazione per determinare la figura con area maggiore in una sequenza di *rettangoli e quadrati* fornita da tastiera.

```
... rAreaMax = null;  
boolean continuare;  
...  
  
do {  
    //legge i dati di una figura (rettangolo o quadrato)  
    ... r = ...lettura della figura...  
  
    //confronta la figura con quella di area maggiore  
    if (rAreaMax == null || r.haAreaMaggiori(rAreaMax))  
        rAreaMax = r;  
    ...  
} while (continuare);  
  
...comunicazione dei risultati
```

- ▶ Di che tipo definiamo *r* e *rAreaMax*?

# Quadrato e Rettangolo

- Geometricamente: i quadrati sono particolari rettangoli

# Quadrato e Rettangolo

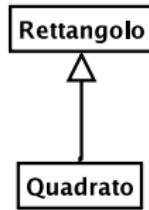
- ▶ Geometricamente: i quadrati sono particolari rettangoli
- ▶ Questa relazione è presente anche nell'implementazione

```
class Quadrato extends Rettangolo
```

# Quadrato e Rettangolo

- ▶ Geometricamente: i quadrati sono particolari rettangoli
- ▶ Questa relazione è presente anche nell'implementazione

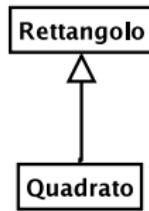
```
class Quadrato extends Rettangolo
```



# Quadrato e Rettangolo

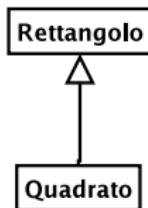
- ▶ Geometricamente: i quadrati sono particolari rettangoli
- ▶ Questa relazione è presente anche nell'implementazione

```
class Quadrato extends Rettangolo
```



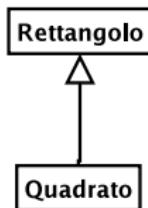
- ▶ Una classe che ne estende un'altra ne **eredita i metodi e i campi**

# Terminologia e notazione UML



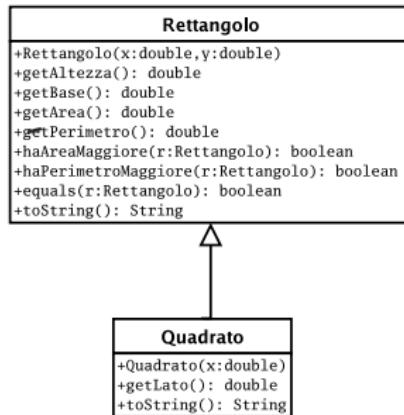
- **Quadrato** è una **sottoclass**e di **Rettangolo**

# Terminologia e notazione UML



- ▶ Quadrato è una sottoclasse di Rettangolo
- ▶ Rettangolo è una superclasse di Quadrato

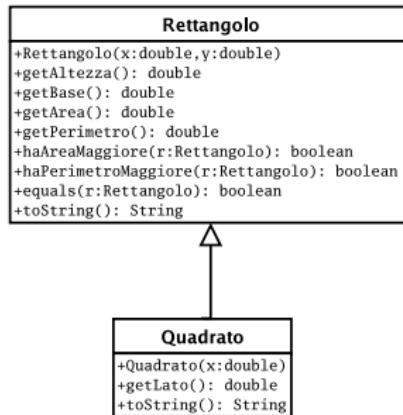
# Ereditarietà



## Quadrato

- eredita i metodi di **Rettangolo**

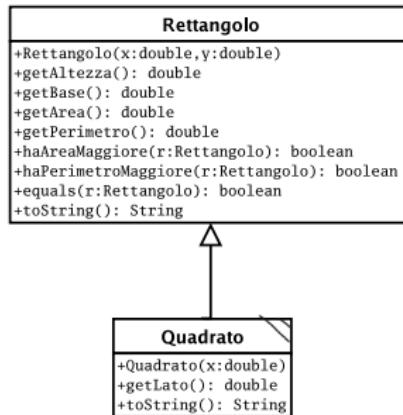
# Ereditarietà



## Quadrato

- ▶ eredita i metodi di **Rettangolo**
- ▶ ha dei metodi in più rispetto a **Rettangolo**

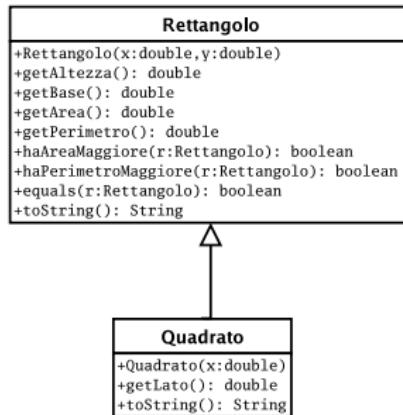
# Ereditarietà



## Quadrato

- ▶ eredita i metodi di **Rettangolo**
- ▶ ha dei metodi in più rispetto a **Rettangolo**
- ▶ ridefinisce alcuni metodi di **Rettangolo**

# Ereditarietà



## Quadrato

- ▶ eredita i metodi di **Rettangolo**
- ▶ ha dei metodi in più rispetto a **Rettangolo**
- ▶ ridefinisce alcuni metodi di **Rettangolo**

## Riutilizzo del codice

Chi ha implementato **Quadrato** ha scritto solo ciò che la differenzia da **Rettangolo**

## Relazione fra i tipi riferimento

- ▶ Il tipo della **superclasse** è un tipo più ampio di quello della **sottoclasse**

## Relazione fra i tipi riferimento

- ▶ Il tipo della **superclasse** è un tipo più ampio di quello della **sottoclasse**
- ▶ I riferimenti a oggetti della sottoclasse possono essere **promossi** al tipo della superclasse

## Relazione fra i tipi riferimento

- ▶ Il tipo della **superclasse** è un tipo più ampio di quello della **sottoclasse**
- ▶ I riferimenti a oggetti della sottoclasse possono essere **promossi** al tipo della superclasse

```
Rettangolo r;  
Quadrato q = new Quadrato(6);  
r = q;
```

# Relazione fra i tipi riferimento

- ▶ Il tipo della **superclasse** è un tipo più ampio di quello della **sottoclasse**
- ▶ I riferimenti a oggetti della sottoclasse possono essere **promossi** al tipo della superclasse

```
Rettangolo r;  
Quadrato q = new Quadrato(6);  
r = q;
```

```
Rettangolo r = new Quadrato(6);
```

# Overriding

```
Rettangolo r = new Quadrato(6);  
out.println(r.toString());
```

Quale metodo `toString` viene eseguito?

# Overriding

```
Rettangolo r = new Quadrato(6);  
out.println(r.toString());
```

Quale metodo `toString` viene eseguito?

- ▶ Il metodo da eseguire viene scelto dalla `JVM` in fase di esecuzione

**OVERIDING:** Poniamo il caso che andiamo a dichiarare un nuovo metodo che richiamiamo nel main, ma supponiamo di dichiarare nella classe figlio lo stesso metodo ma che fa cose differenti. Per gli oggetti della classe figlio, quando richiamiamo il metodo viene eseguito quello dichiarato nella classe figlio.

```
Rettangolo r = new Quadrato(6);  
out.println(r.toString());
```

Quale metodo `toString` viene eseguito?

- ▶ Il metodo da eseguire viene scelto dalla **JVM** in fase di esecuzione
- ▶ Dipende dal **tipo dell'oggetto** e non dal tipo del riferimento

# Overriding

```
Rettangolo r = new Quadrato(6);  
out.println(r.toString());
```

Quale metodo `toString` viene eseguito?

- ▶ Il metodo da eseguire viene scelto dalla **JVM** in fase di esecuzione
- ▶ Dipende dal **tipo dell'oggetto** e non dal tipo del riferimento

## Polimorfismo

La stessa chiamata può *assumere più forme* a seconda del tipo dell'oggetto a cui viene rivolta.

## Esempio

```
Rettangolo r;  
double x = in.readInt("Base?");  
double y = in.readInt("Altezza?");  
  
if (x == y)  
    r = new Quadrato(x);  
else  
    r = new Rettangolo(x,y);  
  
out.println(r.toString());
```

- ▶ Se x e y contengono 6 stampa

lato = 6.0

## Esempio

```
Rettangolo r;  
double x = in.readInt("Base?");  
double y = in.readInt("Altezza?");  
  
if (x == y)  
    r = new Quadrato(x);  
else  
    r = new Rettangolo(x,y);  
  
out.println(r.toString());
```

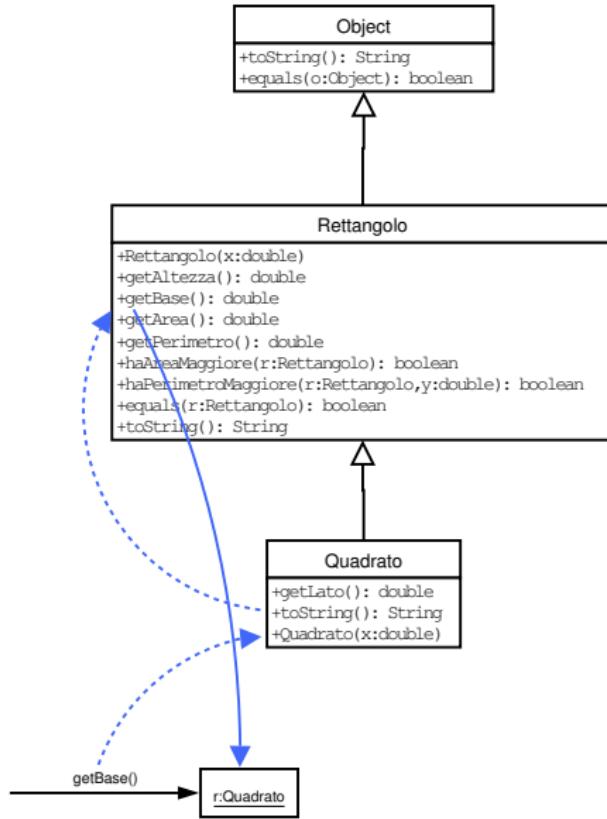
- ▶ Se x e y contengono 6 stampa

lato = 6.0

- ▶ Se x contiene 7 e y contiene 3 stampa

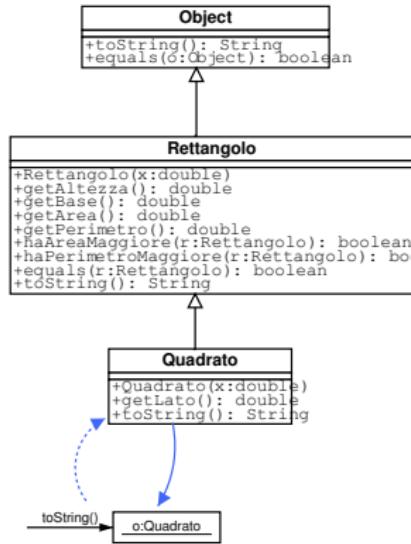
base = 7.0, altezza = 3.0

# Ricerca del metodo

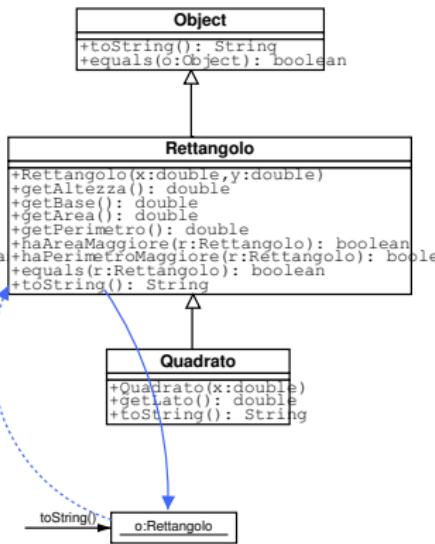


# Comportamento polimorfico

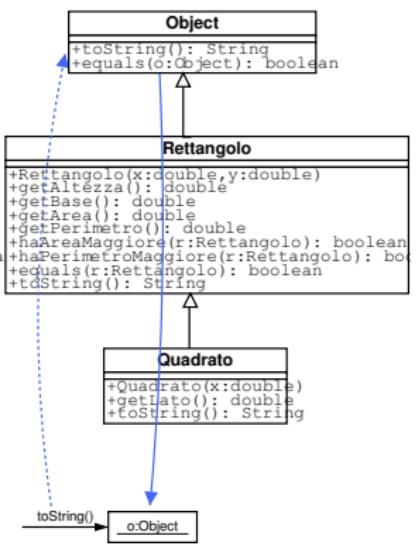
```
Object o = new Quadrato(3);
```



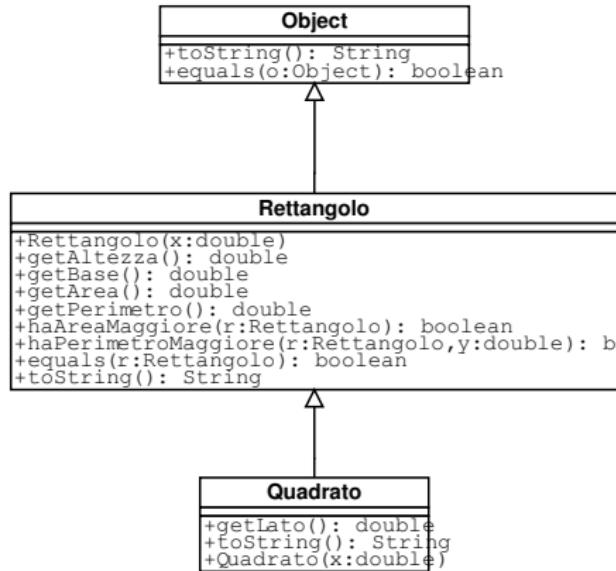
```
Object o = new Rettangolo(3,2);
```



```
Object o = new Object();
```



# Compilazione ed esecuzione



```
Object o;
...
o.getBase()
```

Non viene accettata dal compilatore.

## Fase di compilazione

Si verifica l'esistenza, **per il tipo del riferimento utilizzato**, di un metodo che soddisfi la chiamata.

# Compilazione ed esecuzione

## Fase di compilazione

Si verifica l'esistenza, **per il tipo del riferimento utilizzato**, di un metodo che soddisfi la chiamata.

## Fase di esecuzione

Viene selezionato il metodo da eseguire, sulla base del **tipo effettivo dell'oggetto** (e non del tipo del riferimento).

## Fase di compilazione

Si verifica l'esistenza, **per il tipo del riferimento utilizzato**, di un metodo che soddisfi la chiamata.

## Fase di esecuzione

Viene selezionato il metodo da eseguire, sulla base del **tipo effettivo dell'oggetto** (e non del tipo del riferimento).

- ▶ La ricerca avviene a partire dalla **classe dell'oggetto**, risalendo nella gerarchia.

# Compilazione ed esecuzione

## Fase di compilazione

Si verifica l'esistenza, **per il tipo del riferimento utilizzato**, di un metodo che soddisfi la chiamata.

## Fase di esecuzione

Viene selezionato il metodo da eseguire, sulla base del **tipo effettivo dell'oggetto** (e non del tipo del riferimento).

- ▶ La ricerca avviene a partire dalla **classe dell'oggetto**, risalendo nella gerarchia.
- ▶ Poiché il compilatore ha controllato l'esistenza di un tale metodo per il **tipo del riferimento**, prima o poi il metodo selezionato sarà trovato (al massimo risalendo nella gerarchia fino al tipo del riferimento).

# L'operatore instanceof

## Sintassi

*espressione\_riferimento* instanceof *tipo\_riferimento*

# L'operatore instanceof

## Sintassi

*espressione\_riferimento instanceof tipo\_riferimento*

- ▶ È un'espressione:  
**tipo: boolean**

# L'operatore instanceof

## Sintassi

*espressione\_riferimento* instanceof *tipo\_riferimento*

- ▶ È un'espressione:

**tipo:** boolean

**valore:** true se e solo se *espressione\_riferimento* si riferisce a un oggetto che è un'istanza di *tipo\_riferimento*

# L'operatore instanceof

## Sintassi

*espressione\_riferimento instanceof tipo\_riferimento*

- È un'espressione:

**tipo:** boolean

**valore:** true se e solo se *espressione\_riferimento* si riferisce a un oggetto che è un'istanza di *tipo\_riferimento*

Ricordando che un'istanza di una classe è istanza anche di tutte le sue superclassi

**COMANDO "InstanceOff":** Serve per far capire al compilatore di quale tipo è un oggetto nel caso ricada sull'utente e quindi il compilatore non lo sa ancora.

## Esempio

```
Rettangolo r;
double x = in.readInt("Base?");
double y = in.readInt("Altezza?");

if (x == y)
    r = new Quadrato(x);
else
    r = new Rettangolo(x,y);

if (r instanceof Quadrato)
    out.print("Quadrato: ");
else
    out.print("Rettangolo: ");

out.println(r.toString());
```

## FiguraAreaMax: il metodo leggiRettangolo (1)

```
private static Rettangolo leggiRettangolo(ConsoleInputManager in,
                                         ConsoleOutputManager out) {
    //leggi i dati del rettangolo
    out.println("Inserisci i dati di un rettangolo:");
    double x = in.readDouble(" - base? ");
    double y = in.readDouble(" - altezza? ");

    while (x < 0 || y < 0) {
        out.println("I dati inseriti non rappresentano un rettangolo.");
        out.println("Inserisci i dati di un rettangolo:");
        x = in.readDouble(" - base? ");
        y = in.readDouble(" - altezza? ");
    }
    ...
}
```

## FiguraAreaMax: il metodo leggiRettangolo (2)

```
private static Rettangolo leggiRettangolo(ConsoleInputManager in,
                                         ConsoleOutputManager out) {
    //leggi i dati del rettangolo
    out.println("Inserisci i dati di un rettangolo:");
    double x = in.readDouble(" - base? ");
    double y = in.readDouble(" - altezza? ");

    ...

    Rettangolo r;
    if (x == y) {
        r = new Quadrato(x);
        out.println("La figura Ã“ un quadrato:");
    } else {
        r = new Rettangolo(x, y);
        out.println("La figura Ã“ un rettangolo:");
    }
    out.println(" " + r.toString());
    out.println(" area = " + r.getArea() +
               ", perimetro = " + r.getPerimetro());
    out.println();
    return r;
}
```

## FiguraAreaMax (1)

```
...
Rettangolo rAreaMax = null;
boolean continuare;

do {
    //leggi i dati di una figura (rettangolo o quadrato)
    Rettangolo r = leggiRettangolo(in, out);

    //confronta la figura con quella di area maggiore
    if (rAreaMax == null || r.haAreaMaggiori(rAreaMax))
        rAreaMax = r;

    continuare = in.readSiNo("Vuoi inserire i dati di un'altra " +
                            "figura? (s/n) ");
} while (continuare);

...
```

## FiguraAreaMax (2)

```
...
Rettangolo rAreaMax = null;
boolean continuare;

...
//comunica le caratteristiche della figura di area maggiore
out.print("La figura di area maggiore Ã" un ");
if (rAreaMax instanceof Quadrato)
    out.println("quadrato:  ");
else
    out.println("rettangolo:  ");
out.println(" " + rAreaMax.toString());
out.println(" area = " + rAreaMax.getArea() +
            ", perimetro = " + rAreaMax.getPerimetro());
...
...
```

## Cerchio

Le sue istanze rappresentano cerchi.

## Cerchio

Le sue istanze rappresentano cerchi.

## Costruttore

- ▶ **public Cerchio(double r)**

Costruisce un oggetto che rappresenta il cerchio il cui raggio è specificato dall'argomento.

## Metodi

- ▶ public double getRaggio()
- ▶ public double getArea()
- ▶ public double getCirconferenza()
- ▶ public double getPerimetro()
- ▶ public boolean equals(Cerchio c)
- ▶ public boolean haAreaMaggiore(Cerchio c)
- ▶ public boolean haPerimetroMaggiore(Cerchio c)
- ▶ public String toString()  
    "raggio = 3.1"

## FiguraAreaMax: schema

- ▶ Si scriva un'applicazione per determinare la figura con area maggiore in una sequenza di *rettangoli, quadrati e cerchi* fornita da tastiera.

## FiguraAreaMax: schema

- ▶ Si scriva un'applicazione per determinare la figura con area maggiore in una sequenza di *rettangoli, quadrati e cerchi* fornita da tastiera.

```
... r, rAreaMax = null;  
boolean continuare;  
...  
  
do {  
    //legge i dati di una figura (rettangolo, quadrato o cerchio)  
    ... r = ...lettura della figura...  
  
    //confronta la figura con quella di area maggiore  
    if (rAreaMax == null || r.haAreaMaggiore(rAreaMax))  
        rAreaMax = r;  
    ...  
} while (continuare);  
  
...comunicazione dei risultati
```

## FiguraAreaMax: schema

- ▶ Si scriva un'applicazione per determinare la figura con area maggiore in una sequenza di *rettangoli, quadrati e cerchi* fornita da tastiera.

```
... r, rAreaMax = null;  
boolean continuare;  
...  
  
do {  
    //legge i dati di una figura (rettangolo, quadrato o cerchio)  
    ... r = ...lettura della figura...  
  
    //confronta la figura con quella di area maggiore  
    if (rAreaMax == null || r.haAreaMaggiori(rAreaMax))  
        rAreaMax = r;  
    ...  
} while (continuare);  
  
...comunicazione dei risultati
```

- ▶ Di che tipo definiamo r e rAreaMax?

# La classe astratta Figura

- ▶ Rettangoli, quadrati e cerchi sono particolari **figure geometriche**

# La classe astratta Figura

- ▶ Rettangoli, quadrati e cerchi sono particolari **figure geometriche**
- ▶ **Rettangolo**, **Quadrato** e **Cerchio**, sono state progettate tenendone conto...

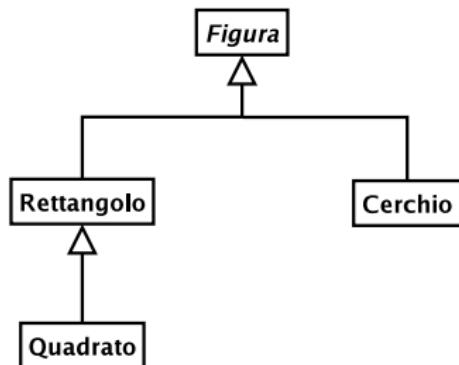
# La classe astratta Figura

- ▶ Rettangoli, quadrati e cerchi sono particolari **figure geometriche**
- ▶ **Rettangolo**, **Quadrato** e **Cerchio**, sono state progettate tenendone conto...  
... sono sottoclassi della classe **Figura**

# La classe astratta Figura

- ▶ Rettangoli, quadrati e cerchi sono particolari **figure geometriche**
- ▶ **Rettangolo**, **Quadrato** e **Cerchio**, sono state progettate tenendone conto...  
... sono sottoclassi della classe **Figura**

```
class Rettangolo extends Figura  
class Cerchio extends Figura  
class Quadrato extends Rettangolo
```



## La classe astratta Figura

- ▶ Tutte le figure geometriche hanno un area e un perimetro:

# La classe astratta Figura

- ▶ Tutte le figure geometriche hanno un area e un perimetro:

## Metodi

- ▶ `double getArea()`
- ▶ `double getPerimetro()`

# La classe astratta Figura

- ▶ Tutte le figure geometriche hanno un area e un perimetro:

## Metodi

- ▶ `double getArea()`
  - ▶ `double getPerimetro()`
- 
- ▶ Il procedimento per il calcolo dell'area e del perimetro dipende dal tipo concreto di figura, quindi non siamo in grado di definirli nella classe **Figura**

- ▶ Tutte le figure geometriche hanno un area e un perimetro:

## Metodi

- ▶ `double getArea()`
- ▶ `double getPerimetro()`

- ▶ Il procedimento per il calcolo dell'area e del perimetro dipende dal tipo concreto di figura, quindi non siamo in grado di definirli nella classe **Figura**
- ▶ Sono definiti come **metodi astratti** (**abstract**)

## Metodi astratti

Metodi di cui è specificato il prototipo ma non l'implementazione.

## Metodi astratti

Metodi di cui è specificato il prototipo ma non l'implementazione.

- ▶ Una classe contenente **metodi astratti** dev'essere dichiarata **astratta** (**abstract**)

## Metodi astratti

Metodi di cui è specificato il prototipo ma non l'implementazione.

- ▶ Una classe contenente **metodi astratti** dev'essere dichiarata **astratta** (**abstract**)
- ▶ Una classe astratta **non può essere istanziata**, ma può essere estesa

## Metodi astratti

Metodi di cui è specificato il prototipo ma non l'implementazione.

- ▶ Una classe contenente **metodi astratti** dev'essere dichiarata **astratta** (**abstract**)
- ▶ Una classe astratta **non può essere istanziata**, ma può essere estesa
- ▶ Le sottoclassi di una classe astratta devono **fornire l'implementazione di tutti i metodi astratti** (salvo che siano anch'esse astratte)

## Metodi astratti

Metodi di cui è specificato il prototipo ma non l'implementazione.

- ▶ Una classe contenente **metodi astratti** dev'essere dichiarata **astratta** (**abstract**)
- ▶ Una classe astratta **non può essere istanziata**, ma può essere estesa
- ▶ Le sottoclassi di una classe astratta devono **fornire l'implementazione di tutti i metodi astratti** (salvo che siano anch'esse astratte)
- ▶ Una classe astratta può contenere anche metodi implementati.

## Metodi astratti

Metodi di cui è specificato il prototipo ma non l'implementazione.

- ▶ Una classe contenente **metodi astratti** dev'essere dichiarata **astratta** (**abstract**)
- ▶ Una classe astratta **non può essere istanziata**, ma può essere estesa
- ▶ Le sottoclassi di una classe astratta devono **fornire l'implementazione di tutti i metodi astratti** (salvo che siano anch'esse astratte)
- ▶ Una classe astratta può contenere anche metodi implementati.
- ▶ **Classi concrete:** le classi che non sono astratte

**COMANDO "Abstract":** Public Abstract Void NomeMetodoGenerale() } }

- Mi obbliga a dover dichiarare in che modo avviene il metodo astratto dichiarato.
- Tutti i metodi della classe madre sono anche della classe figlia ma non so quale sarà nello specifico => Abstract (so che c'è ma non come si sviluppa)

<i>Figura</i>
+ <i>getArea()</i> : double
+ <i>getPerimetro()</i> : double
+ <i>haAreaMaggiore(r:Figura)</i> : boolean
+ <i>haPerimetroMaggiore(r:Figura)</i> : boolean
+ <i>equals(r:Figura)</i> : boolean

## ► Classi astratte

Descritte come le classi concrete ma il nome è indicato in italico.

<i>Figura</i>
+ <i>getArea()</i> : double
+ <i>getPerimetro()</i> : double
+ <i>haAreaMaggiore(r:Figura)</i> : boolean
+ <i>haPerimetroMaggiore(r:Figura)</i> : boolean
+ <i>equals(r:Figura)</i> : boolean

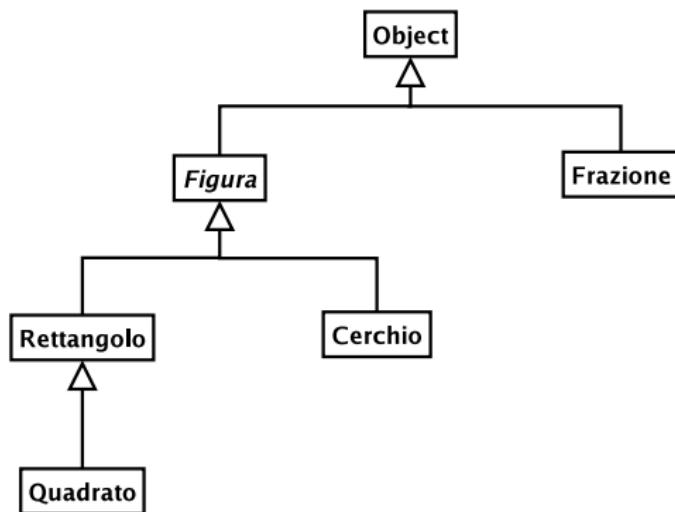
## ► Classi astratte

Descritte come le classi concrete ma il nome è indicato in italico.

## ► Metodi astratti

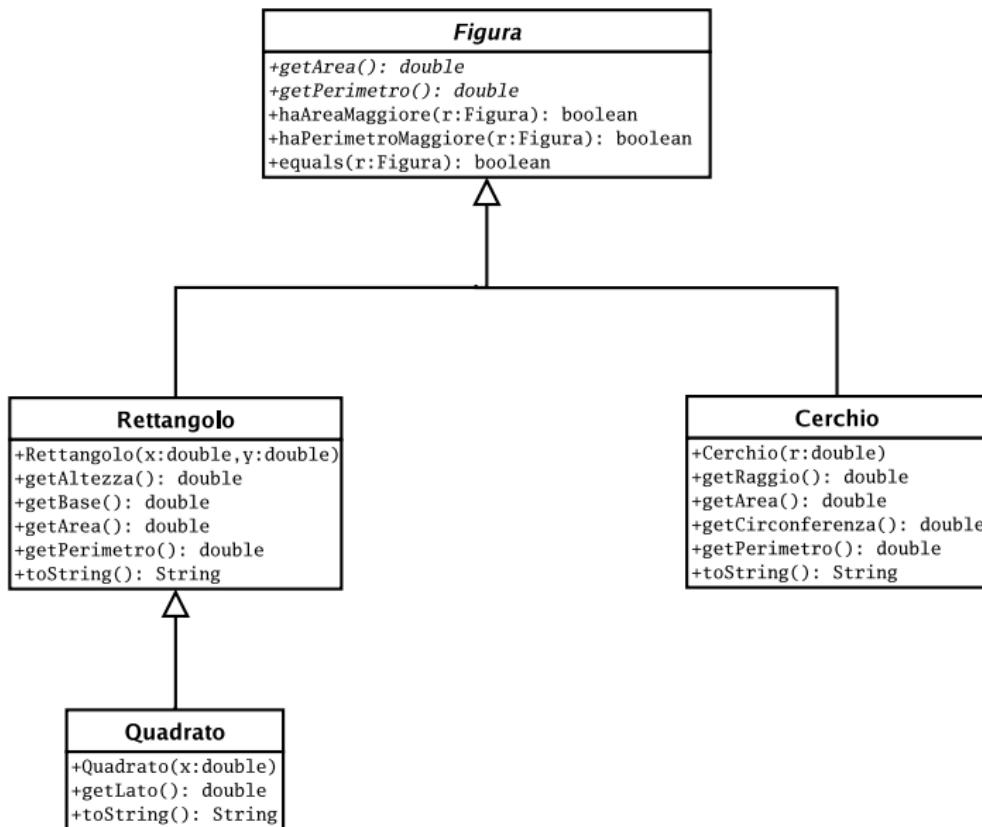
Descritti come i metodi concreti ma il prototipo è indicato in italico

# Classi astratte e gerarchia



- ▶ Le classi astratte forniscono (come quelle concrete) un **supertipo comune** i cui valori sono tutte le possibili istanze di tutte le sottoclassi

# La gerarchia di Figura



È possibile assegnare a una variabile il cui tipo sia una classe un riferimento a un oggetto di una sottoclasse

# Gerarchia e uso dei riferimenti

È possibile assegnare a una variabile il cui tipo sia una classe un riferimento a un oggetto di una sottoclasse

## Esempi

```
Object o = new String("pippo");
o = new Frazione(1,2);
```

# Gerarchia e uso dei riferimenti

È possibile assegnare a una variabile il cui tipo sia una classe un riferimento a un oggetto di una sottoclasse

## Esempi

```
Object o = new String("pippo");  
o = new Frazione(1,2);
```

```
Rettangolo r;  
Quadrato q;  
...  
r = q;
```

# Gerarchia e uso dei riferimenti

Non è possibile assegnare (direttamente) a una variabile del tipo della sottoclasse un **riferimento della superclasse**

# Gerarchia e uso dei riferimenti

Non è possibile assegnare (direttamente) a una variabile del tipo della sottoclasse un **riferimento della superclasse**

## Esempi

```
Object o = new String("pippo");  
  
String s = o; //errore in compilazione
```

# Gerarchia e uso dei riferimenti

Non è possibile assegnare (direttamente) a una variabile del tipo della sottoclasse un **riferimento della superclasse**

## Esempi

```
Object o = new String("pippo");  
  
String s = o; //errore in compilazione
```

```
Rettangolo r;  
Quadrato q;  
...  
q = r; //errore in compilazione
```

## Cast di variabili riferimento

È possibile assegnare a una variabile del tipo della sottoclasse un riferimento della superclasse **mediante l'uso del cast**

# Cast di variabili riferimento

È possibile assegnare a una variabile del tipo della sottoclasse un riferimento della superclasse **mediante l'uso del cast**

## Esempi

```
Object o = new String("pippo");  
  
String s = (String)o;
```

# Cast di variabili riferimento

È possibile assegnare a una variabile del tipo della sottoclasse un riferimento della superclasse **mediante l'uso del cast**

## Esempi

```
Object o = new String("pippo");  
  
String s = (String)o;
```

```
Rettangolo r;  
Quadrato q;  
...  
q = (Quadrato) r;
```

# Cast di variabili riferimento

È possibile assegnare a una variabile del tipo della sottoclasse un riferimento della superclasse **mediante l'uso del cast**

## Esempi

```
Object o = new String("pippo");  
  
String s = (String)o;
```

```
Rettangolo r;  
Quadrato q;  
...  
q = (Quadrato) r;
```

Se l'oggetto riferito da **r** non è  
un'istanza di **Quadrato** si ha un  
**errore in esecuzione**

## Esempio

```
Rettangolo r;  
double x = in.readInt("Base?");  
double y = in.readInt("Altezza?");  
  
if (x == y)  
    r = new Quadrato(x);  
else  
    r = new Rettangolo(x,y);  
  
Quadrato q = (Quadrato)r;  
out.println("Lato = " + q.getLato());
```

## Esempio

```
Rettangolo r;  
double x = in.readInt("Base?");  
double y = in.readInt("Altezza?");  
  
if (x == y)  
    r = new Quadrato(x);  
else  
    r = new Rettangolo(x,y);  
  
Quadrato q = (Quadrato)r;  
out.println("Lato = " + q.getLato());
```

Esecuzione: se x != y

Exception in thread "main" java.lang.ClassCastException:  
 prog.utili.Rettangolo  
...

## Controllare con instanceof

È dunque opportuno effettuare un controllo usando `instanceof`, prima di effettuare il cast.

```
Rettangolo r;
double x = in.readInt("Base?");
double y = in.readInt("Altezza?");

if (x == y)
    r = new Quadrato(x);
else
    r = new Rettangolo(x,y);

if (r instanceof Quadrato)
    out.println("Lato = " + ((Quadrato)r).getLato());
```

## SequenzaOrdinata<E>

Le sue istanze rappresentano sequenze ordinate di oggetti di tipo E.

## SequenzaOrdinata<E>

Le sue istanze rappresentano sequenze ordinate di oggetti di tipo E.

- ▶ Per utilizzarla dobbiamo instanziare il **tipo parametro** con un **tipo riferimento**

## SequenzaOrdinata<E>

Le sue istanze rappresentano sequenze ordinate di oggetti di tipo E.

- ▶ Per utilizzarla dobbiamo instanziare il **tipo parametro** con un **tipo riferimento**
- ▶ Per costruire sequenze ordinate è **necessario** che il tipo degli elementi sia **“ordinabile”**, cioè che sia definita una relazione d’ordine totale tra i suoi elementi

## Esempi

```
SequenzaOrdinata<String>    //lecita  
SequenzaOrdinata<Frazione> //lecita  
SequenzaOrdinata<Integer>   //lecita
```

```
SequenzaOrdinata<Rettangolo> //non lecita
```

# SequenzaOrdinata<E>

## Esempi

```
SequenzaOrdinata<String>    //lecita  
SequenzaOrdinata<Frazione> //lecita  
SequenzaOrdinata<Integer>   //lecita
```

```
SequenzaOrdinata<Rettangolo> //non lecita
```

type parameter prog.utili.Rettangolo is not within its bound

# SequenzaOrdinata<E>

## Esempi

```
SequenzaOrdinata<String>    //lecita  
SequenzaOrdinata<Frazione> //lecita  
SequenzaOrdinata<Integer>   //lecita
```

```
SequenzaOrdinata<Rettangolo> //non lecita
```

type parameter prog.utili.Rettangolo is not within its bound

String, Frazione e Integer implementano l'*interfaccia Comparable<T>*

Questo dice al compilatore che i relativi oggetti sono ordinabili

# L'interfaccia Comparable<T>

- ▶ **Interfaccia**

Specifica il prototipo di alcuni metodi senza fornire alcuna implementazione

# L'interfaccia Comparable<T>

- ▶ **Interfaccia**

Specifica il prototipo di alcuni metodi senza fornire alcuna implementazione

- ▶ **Comparable<T>**

Specifica un unico metodo il cui scopo è quello di definire un ordine totale sugli oggetti del tipo che la implementano

# L'interfaccia Comparable<T>

- ▶ **Interfaccia**

Specifica il prototipo di alcuni metodi senza fornire alcuna implementazione

- ▶ **Comparable<T>**

Specifica un unico metodo il cui scopo è quello di definire un ordine totale sugli oggetti del tipo che la implementano

## Metodi

- ▶ **public int compareTo(T o)**

Confronta l'oggetto che esegue il metodo con quello specificato come argomento, e restituisce un intero negativo, zero, o un intero positivo, a seconda che l'oggetto che esegue il metodo sia minore, uguale o maggiore di quello specificato come argomento.

Se `a` e `o` sono riferimenti a oggetti di tipo `T`:

$$a.\text{compareTo}(o) = \begin{cases} \text{intero negativo} & \text{se } a \text{ è minore di } o \\ 0 & \text{se } a.\text{equals}(o) \\ \text{intero positivo} & \text{se } a \text{ è maggiore di } o \end{cases}$$

Se `a` e `o` sono riferimenti a oggetti di tipo `T`:

$$a.\text{compareTo}(o) = \begin{cases} \text{intero negativo} & \text{se } a \text{ è minore di } o \\ 0 & \text{se } a.\text{equals}(o) \\ \text{intero positivo} & \text{se } a \text{ è maggiore di } o \end{cases}$$

L'ordinamento rispetto al quale viene effettuato il confronto dipende dagli oggetti:

- ▶ `String` ordinamento lessicografico
- ▶ `Integer`, `Double`..., ordinamento usuale
- ▶ `Data`, `Orario`, ordinamento cronologico

# Implementare un'interfaccia

Una classe implementa un'interfaccia:

- (1) Dichiarendolo nell'intestazione

# Implementare un'interfaccia

Una classe implementa un'interfaccia:

- (1) Dichiarendolo nell'intestazione

```
public class Frazione implements Comparable<Frazione>
public class String implements Comparable<String>
public class Integer implements Comparable<Integer>
```

# Implementare un'interfaccia

Una classe implementa un'interfaccia:

- (1) Dichiarendolo nell'intestazione

```
public class Frazione implements Comparable<Frazione>
public class String implements Comparable<String>
public class Integer implements Comparable<Integer>
```

- (2) Fornendo l'implementazione del metodo `compareTo(T o)`, tenendo conto del tipo parametro utilizzato.

# Implementare un'interfaccia

Una classe implementa un'interfaccia:

- (1) Dichiarendolo nell'intestazione

```
public class Frazione implements Comparable<Frazione>
public class String implements Comparable<String>
public class Integer implements Comparable<Integer>
```

- (2) Fornendo l'implementazione del metodo `compareTo(T o)`, tenendo conto del tipo parametro utilizzato.

```
public int compareTo(Frazione o)
public int compareTo(String o)
public int compareTo(Integer o)
```

INTERFACCIA: - Costrutto che mi permette di definire certi comportamenti.

Describe un'insieme di metodi.

L'unica differenza con una classe, è che l'interfaccia non ha un costruttore, non ha senso che abbia attributi.

Utilità? Mi interessa solo i metodi, interessante quando devo solo mostrare delle operazioni/comportamenti degli utenti e non ho bisogno di importare dati e quindi dichiarare e manipolare attributi.

Public Class Passero extends Animale implements Volatile



## Interfaccia Java

Parte di codice che specifica dei comportamenti senza fornirne l'implementazione.

- ▶ Specificano solo prototipi e contratti di metodi (**metodi astratti**) o costanti

## Interfaccia Java

Parte di codice che specifica dei comportamenti senza fornirne l'implementazione.

- ▶ Specificano solo prototipi e contratti di metodi (**metodi astratti**) o costanti
- ▶ Un'interfaccia “**promette**” uno o più metodi

## Interfaccia Java

Parte di codice che specifica dei comportamenti senza fornirne l'implementazione.

- ▶ Specificano solo prototipi e contratti di metodi (**metodi astratti**) o costanti
- ▶ Un'interfaccia “**promette**” uno o più metodi
- ▶ Una classe che implementa l'interfaccia soddisfa la promessa

## Interfaccia Java

Parte di codice che specifica dei comportamenti senza fornirne l'implementazione.

- ▶ Specificano solo prototipi e contratti di metodi (**metodi astratti**) o costanti
- ▶ Un'interfaccia “**promette**” uno o più metodi
- ▶ Una classe che implementa l'interfaccia soddisfa la promessa
- ▶ Se una classe dichiara di implementare l'interfaccia, ma non fornisce l'implementazione di tutti i suoi metodi allora deve essere **astratta**

- ▶ Il nome di un'interfaccia definisce un **tipo riferimento**

- ▶ Il nome di un'interfaccia definisce un **tipo riferimento**
- ▶ È un supertipo per tutte le classi che implementano l'interfaccia

- ▶ Il nome di un'interfaccia definisce un **tipo riferimento**
- ▶ È un supertipo per tutte le classi che implementano l'interfaccia
- ▶ Valgono tutte le regole (promozioni e cast) che valgono per gli altri tipi riferimento

- ▶ Il nome di un'interfaccia definisce un **tipo riferimento**
- ▶ È un supertipo per tutte le classi che implementano l'interfaccia
- ▶ Valgono tutte le regole (promozioni e cast) che valgono per gli altri tipi riferimento

## Esempio

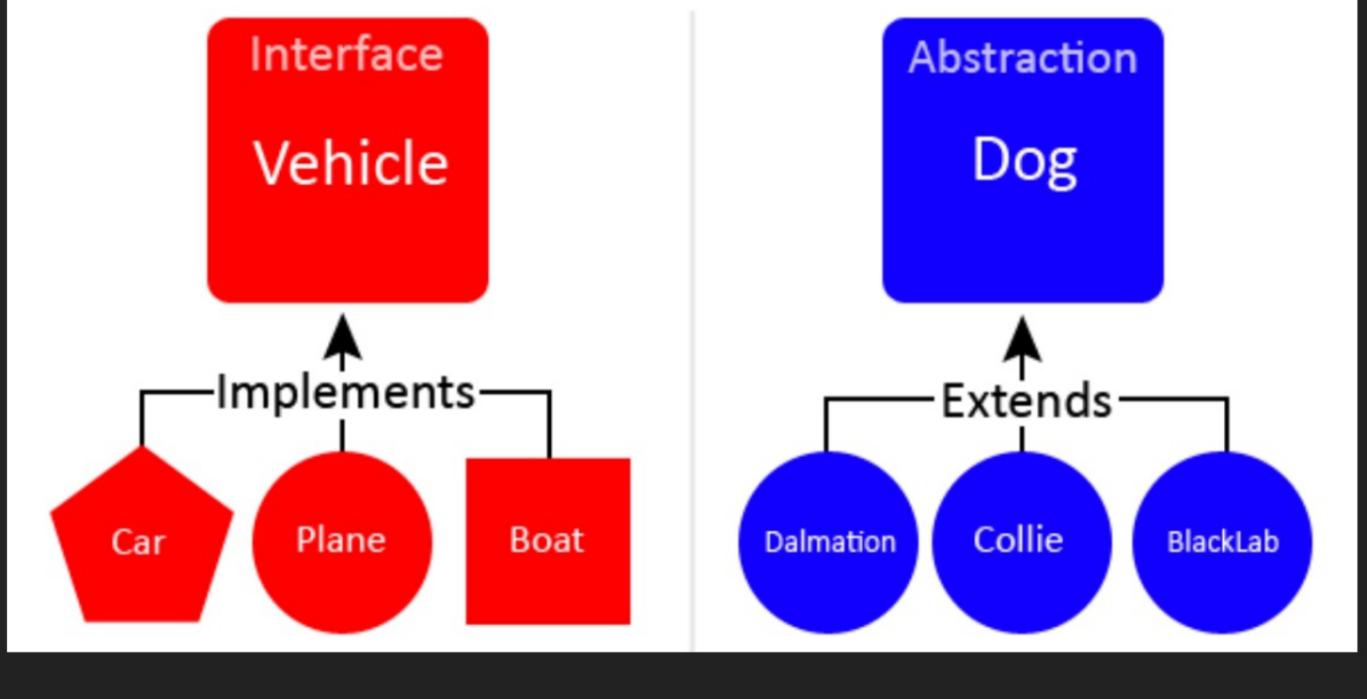
```
Comparable<Frazione> c;
```

```
c = new Frazione(3,4);
```

## Differenze tra classi astratte e interfacce

Entrambe permettono di definire un concetto senza dover conoscere i dettagli di una classe, dettagli che saranno implementati dai figli.

### Interfaces vs. Abstract Classes

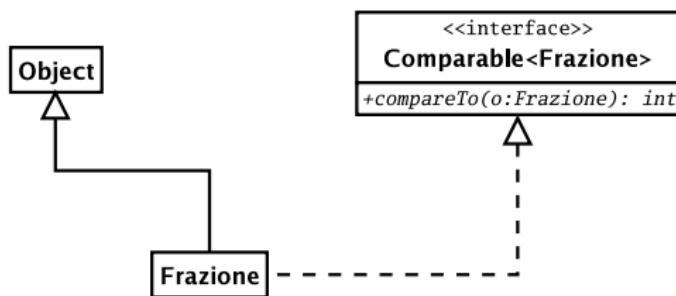


## Differenze sintattiche principali:

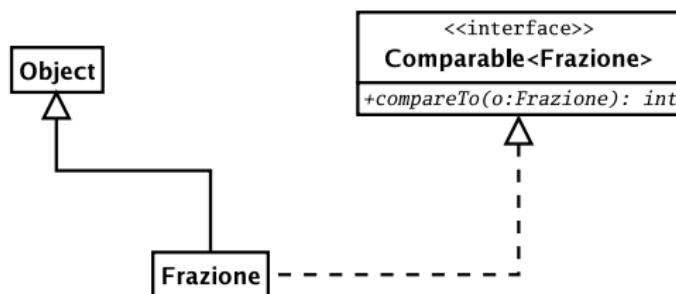
- Una classe astratta, oltre a metodi astratti, può contenere dati e metodi non astratti. Una interfaccia no.
- Una classe astratta può avere un costruttore. Una interfaccia no.
- In una interfaccia si possono definire solo costanti.
- In una interfaccia i metodi la visibilità sarà public in automatico (c'è anche la possibilità di usare quella di default ma i figli saranno "costretti" a fare l'override dei metodi con public, anche se sono nello stesso package).

Oltre le differenze sintattiche, si usa una classe astratta o una interfaccia in base al tipo di legame che si vuole instaurare con le classi derivate. Se si vuole un legame molto forte con le classi derivate si usa una classe astratta, in questo modo tutti i figli avranno un comportamento comune (ricordiamoci che una classe può estendere solo una classe). Le interfacce, invece, servono per definire un comportamento più generico con i figli, potendo questi essere di vario genere e natura (ciò è dovuto al fatto che un figlio può implementare tante interfacce ma solo una classe, astratta o meno).

# Rappresentazione UML



- ▶ Stessa notazione utilizzata per le classi con lo **stereotipo <><interface>>**



- ▶ Stessa notazione utilizzata per le classi con lo **stereotipo <<interface>>**
- ▶ La relazione fra un'interfaccia e una classe che la implementa è indicata da una **relazione di realizzazione**

# L'interfaccia generica Iterable<E>

- ▶ Ogni classe che implementa **Iterable** rappresenta una collezione di dati che può essere “iterata”, cioè scandita, un elemento alla volta, mediante l’uso di un oggetto che viene detto **iteratore**

# L'interfaccia generica Iterable<E>

- ▶ Ogni classe che implementa **Iterable** rappresenta una collezione di dati che può essere “iterata”, cioè scandita, un elemento alla volta, mediante l’uso di un oggetto che viene detto **iteratore**
- ▶ Un iteratore è un elenco degli elementi presenti nella collezione

# L'interfaccia generica Iterable<E>

- ▶ Ogni classe che implementa **Iterable** rappresenta una collezione di dati che può essere “iterata”, cioè scandita, un elemento alla volta, mediante l’uso di un oggetto che viene detto **iteratore**
- ▶ Un iteratore è un elenco degli elementi presenti nella collezione

L'interfaccia generica Iterable<E> prevede un solo metodo:

## Metodi

- ▶ `public Iterator<E> iterator()`

Restituisce un iteratore degli oggetti presenti nella collezione che esegue il metodo.

## Esempi

Sequenza<E> implementa Iterable<E> quindi:

```
Sequenza<Frazione> seq = new Sequenza<Frazione>();  
...seq.iterator()...
```

Restituisce un'oggetto di tipo Iterator<Frazione>

# Esempi

Sequenza<E> implementa Iterable<E> quindi:

```
Sequenza<Frazione> seq = new Sequenza<Frazione>();  
...seq.iterator()...
```

Restituisce un'oggetto di tipo Iterator<Frazione>

```
Sequenza<String> seq = new Sequenza<String>();  
...seq.iterator()...
```

Restituisce un'oggetto di tipo Iterator<String>

Un iteratore può essere utilizzato mediante i metodi:

## Metodi

- ▶ **public E next()**

Restituisce il prossimo elemento dell'iteratore, eliminandolo dall'iteratore (non dalla struttura per la quale l'iteratore è stato costruito). Se l'iteratore è vuoto, si verifica un errore in fase di esecuzione.

Un iteratore può essere utilizzato mediante i metodi:

## Metodi

- ▶ `public E next()`

Restituisce il prossimo elemento dell'iteratore, eliminandolo dall'iteratore (non dalla struttura per la quale l'iteratore è stato costruito). Se l'iteratore è vuoto, si verifica un errore in fase di esecuzione.

- ▶ `public boolean hasNext()`

Restituisce `true` se l'iteratore contiene degli elementi e `false` in caso contrario.

## Scandire un elenco

```
Sequenza<Frazione> frazioni = new Sequenza<Frazione>();
```

# Scandire un elenco

```
Sequenza<Frazione> frazioni = new Sequenza<Frazione>();
```

Mediante un ciclo while

```
Iterator<Frazione> elenco = frazioni.iterator();

while (elenco.hasNext())
    out.println(elenco.next());
```

# Scandire un elenco

```
Sequenza<Frazione> frazioni = new Sequenza<Frazione>();
```

## Mediante un ciclo while

```
Iterator<Frazione> elenco = frazioni.iterator();

while (elenco.hasNext())
    out.println(elenco.next());
```

## Mediante un ciclo for

```
for (Iterator<Frazione> elenco = frazioni.iterator();
     elenco.hasNext();)
    out.println(elenco.next());
```

# Scandire un elenco

```
Sequenza<Frazione> frazioni = new Sequenza<Frazione>();
```

## Mediante un ciclo while

```
Iterator<Frazione> elenco = frazioni.iterator();

while (elenco.hasNext())
    out.println(elenco.next());
```

## Mediante un ciclo for

```
for (Iterator<Frazione> elenco = frazioni.iterator();
        elenco.hasNext();)
    out.println(elenco.next());
```

## Mediante un ciclo for-each

```
for (Frazione f : frazioni)
    out.println(f);
```

## Iterable<E> e ciclo for-each

Il ciclo **for-each** può essere utilizzato per scandire qualunque oggetto che implementi l'interfaccia Iterable<E>.

## Iterable<E> e ciclo for-each

Il ciclo **for-each** può essere utilizzato per scandire qualunque oggetto che implementi l'interfaccia Iterable<E>.

Se A è una classe che implementa Iterable<E>

## Iterable<E> e ciclo for-each

Il ciclo **for-each** può essere utilizzato per scandire qualunque oggetto che implementi l'interfaccia Iterable<E>.

Se A è una classe che implementa Iterable<E>

```
for (E o: A)
```

```
...
```

## L'interfaccia generica Iterable<E>

Iterable<E> è un **interfaccia generica** che definisce i prototipi dei metodi:

# L'interfaccia generica Iterable<E>

Iterable<E> è un **interfaccia generica** che definisce i prototipi dei metodi:

## Metodi

- ▶ `public E next()`

Restituisce il prossimo elemento dell'iteratore, eliminandolo dall'iteratore (non dalla struttura per la quale l'iteratore è stato costruito). Se l'iteratore è vuoto, si verifica un errore in fase di esecuzione.

# L'interfaccia generica Iterable<E>

Iterable<E> è un **interfaccia generica** che definisce i prototipi dei metodi:

## Metodi

- ▶ `public E next()`

Restituisce il prossimo elemento dell'iteratore, eliminandolo dall'iteratore (non dalla struttura per la quale l'iteratore è stato costruito). Se l'iteratore è vuoto, si verifica un errore in fase di esecuzione.

- ▶ `public boolean hasNext()`

Restituisce **true** se l'iteratore contiene degli elementi e **false** in caso contrario.

# Gerarchia dei tipi: riassunto

- ▶ **Tipi riferimento:** classi, array, interfacce

# Gerarchia dei tipi: riassunto

- ▶ **Tipi riferimento:** classi, array, interfacce
- ▶ Tutti i tipi riferimento si trovano all'interno della **gerarchia dei tipi riferimento**

## Gerarchia dei tipi: riassunto

- ▶ **Tipi riferimento:** classi, array, interfacce
- ▶ Tutti i tipi riferimento si trovano all'interno della **gerarchia dei tipi riferimento**
- ▶ In cima alla gerarchia: **Object**

- ▶ **Tipi riferimento:** classi, array, interfacce
- ▶ Tutti i tipi riferimento si trovano all'interno della **gerarchia dei tipi riferimento**
- ▶ In cima alla gerarchia: **Object**

## Classi

Ogni classe è sottotipo di **Object**

- Direttamente: se **non estende esplicitamente alcuna classe**

- ▶ **Tipi riferimento:** classi, array, interfacce
- ▶ Tutti i tipi riferimento si trovano all'interno della **gerarchia dei tipi riferimento**
- ▶ In cima alla gerarchia: **Object**

## Classi

Ogni classe è sottotipo di **Object**

- Direttamente: se **non estende esplicitamente alcuna classe**
- Indirettamente: se estende una classe diversa da **Object**

- ▶ **Tipi riferimento:** classi, array, interfacce
- ▶ Tutti i tipi riferimento si trovano all'interno della **gerarchia dei tipi riferimento**
- ▶ In cima alla gerarchia: **Object**

## Classi

Ogni classe è sottotipo di **Object**

- Direttamente: se **non estende esplicitamente alcuna classe**
- Indirettamente: se estende una classe diversa da **Object**

## Array

Ogni array è un oggetto: il tipo di un array è sottotipo di **Object**

# Gerarchia dei tipi: riassunto

- ▶ **Tipi riferimento:** classi, array, interfacce
- ▶ Tutti i tipi riferimento si trovano all'interno della **gerarchia dei tipi riferimento**
- ▶ In cima alla gerarchia: **Object**

## Classi

Ogni classe è sottotipo di **Object**

- Direttamente: se **non estende esplicitamente alcuna classe**
- Indirettamente: se estende una classe diversa da **Object**

## Array

Ogni array è un oggetto: il tipo di un array è sottotipo di **Object**

## Interfacce

Il tipo definito da un'interfaccia è sottotipo di **Object**

# Gerarchia dei tipi: relazioni

Le relazioni all'interno della gerarchia sono stabilite da:

- (1) Estensione delle classi ([superclasse/sottoclasse](#))

```
class B extends A
```

# Gerarchia dei tipi: relazioni

Le relazioni all'interno della gerarchia sono stabilite da:

- (1) Estensione delle classi ([superclasse/sottoclasse](#))

class B extends A



B sottotipo di A

# Gerarchia dei tipi: relazioni

Le relazioni all'interno della gerarchia sono stabilite da:

- (1) Estensione delle classi ([superclasse/sottoclasse](#))

```
class B extends A
```



B sottotipo di A

- (2) Implementazione di interfacce

```
class B implements I
```

# Gerarchia dei tipi: relazioni

Le relazioni all'interno della gerarchia sono stabilite da:

- (1) Estensione delle classi ([superclasse/sottoclasse](#))

`class B extends A`



**B sottotipo di A**

- (2) Implementazione di interfacce

`class B implements I`



**B sottotipo di I**

# Gerarchia dei tipi: relazioni

Le relazioni all'interno della gerarchia sono stabilite da:

- (1) Estensione delle classi ([superclasse/sottoclasse](#))

class B extends A



B sottotipo di A

- (2) Implementazione di interfacce

class B implements I



B sottotipo di I

- (3) Relazioni indotte sugli array dai tipi base

B sottotipo di A

# Gerarchia dei tipi: relazioni

Le relazioni all'interno della gerarchia sono stabilite da:

- (1) Estensione delle classi (**superclasse/sottoclasse**)

`class B extends A` → **B sottotipo di A**

- (2) Implementazione di interfacce

`class B implements I` → **B sottotipo di I**

- (3) Relazioni indotte sugli array dai tipi base

**B sottotipo di A** → **B[] sottotipo di A[]**

## S extends T

Il tipo riferimento **S estende** il tipo riferimento **T** se e solo se **S** è un qualunque sottotipo di **T** (compreso **T** stesso).

## S extends T

Il tipo riferimento **S** estende il tipo riferimento **T** se e solo se **S** è un qualunque sottotipo di **T** (compreso **T** stesso).

## T super S

**T** è supertipo di **S** se e solo se:

- ▶ **S**, **T** classi e **S** è una sottoclasse di **T** (diretta o indiretta).

## S extends T

Il tipo riferimento **S** estende il tipo riferimento **T** se e solo se **S** è un qualunque sottotipo di **T** (compreso **T** stesso).

## T super S

**T** è supertipo di **S** se e solo se:

- ▶ **S**, **T** classi e **S** è una sottoclasse di **T** (diretta o indiretta).
- ▶ **S** classe, **T** interfaccia e **S** implementa (direttamente o indirettamente) **T**

## S extends T

Il tipo riferimento **S** estende il tipo riferimento **T** se e solo se **S** è un qualunque sottotipo di **T** (compreso **T** stesso).

## T super S

**T** è supertipo di **S** se e solo se:

- ▶ **S**, **T** classi e **S** è una sottoclasse di **T** (diretta o indiretta).
- ▶ **S** classe, **T** interfaccia e **S** implementa (direttamente o indirettamente) **T**
- ▶ **S**, **T** interfacce e **S** estende l'interfaccia (direttamente o indirettamente) l'interfaccia **T**

# Tipi generici e gerarchia

- ▶ Come tutti i tipi riferimento, anche i tipi generici e i relativi tipi parametrizzati si collocano all'interno della gerarchia dei tipi

# Tipi generici e gerarchia

- ▶ Come tutti i tipi riferimento, anche i tipi generici e i relativi tipi parametrizzati si collocano all'interno della gerarchia dei tipi
- ▶ Le relazioni fra i tipi parametro **non inducono** relazioni sui relativi tipi parametrizzati

# Tipi generici e gerarchia

- ▶ Come tutti i tipi riferimento, anche i tipi generici e i relativi tipi parametrizzati si collocano all'interno della gerarchia dei tipi
- ▶ Le relazioni fra i tipi parametro **non inducono** relazioni sui relativi tipi parametrizzati

**B sottotipo di A** non implica **Gen<B> sottotipo di Gen<A>**

## Esempio

```
Sequenza<Rettangolo> sr;  
Sequenza<Quadrato> sq;  
...  
sq = sr; //non e' lecito  
sr = sq; //non e' lecito
```

- ▶ **Sequenza<Object>**  
non è supertipo di **Sequenza<Quadrato>**, **Sequenza<Rettangolo>**

- ▶ **Sequenza<Object>**

non è supertipo di **Sequenza<Quadrato>**, **Sequenza<Rettangolo>**

Come facciamo a disporre di un tipo che possa essere utilizzato per riferirsi a sequenze di qualsiasi tipo?

- ▶ **Sequenza<Object>**  
non è supertipo di **Sequenza<Quadrato>**, **Sequenza<Rettangolo>**

Come facciamo a disporre di un tipo che possa essere utilizzato per riferirsi a sequenze di qualsiasi tipo?

- ▶ Per ogni tipo generico, esiste un supertipo comune a tutti i suoi tipi parametrizzati, indicato utilizzando il simbolo speciale **?**

# Tipi generici e gerarchia

- ▶ **Sequenza<Object>**  
non è supertipo di **Sequenza<Quadrato>, Sequenza<Rettangolo>**

Come facciamo a disporre di un tipo che possa essere utilizzato per riferirsi a sequenze di qualsiasi tipo?

- ▶ Per ogni tipo generico, esiste un supertipo comune a tutti i suoi tipi parametrizzati, indicato utilizzando il simbolo speciale **?**
- ▶ **Sequenza<?>**  
è supertipo di **Sequenza<Quadrato>, Sequenza<Rettangolo>, Sequenza<String>...**

## Segnaposto (wildcard)

- ▶ ? funge da **segnaposto** per un tipo che non è noto al momento della compilazione

## Segnaposto (wildcard)

- ▶ ? funge da **segnaposto** per un tipo che non è noto al momento della compilazione
- ▶ Ogni sequenza è un caso particolare di **Sequenza<?>**, dove il segnaposto viene sostituito con un tipo effettivo

# Segnaposto (wildcard)

- ▶ ? funge da **segnaposto** per un tipo che non è noto al momento della compilazione
- ▶ Ogni sequenza è un caso particolare di **Sequenza<?>**, dove il segnaposto viene sostituito con un tipo effettivo
- ▶ **Sequenza<?>** è supertipo di tutti i tipi parametrizzati ottenibili da sequenza

```
Sequenza<?> s;  
...  
s = new Sequenza<Quadrati>();  
...  
s = new Sequenza<Rettangoli>();  
...  
s = new Sequenza<String>();
```

Sequenza<E> fornisce il metodo public boolean add(E o)

Sequenza<E> fornisce il metodo public boolean add(E o)

```
Sequenza<?> s = sq;  
...  
s.add(x) //non e' permessa qualunque sia il tipo di x
```

Sequenza<E> fornisce il metodo `public boolean add(E o)`

```
Sequenza<?> s = sq;  
...  
s.add(x) //non e' permessa qualunque sia il tipo di x
```

- ▶ Il compilatore non può garantire che il tipo di `x` sia compatibile con il tipo argomento della sequenza.

Sequenza<E> fornisce il metodo `public boolean add(E o)`

```
Sequenza<?> s = sq;  
...  
s.add(x) //non e' permessa qualunque sia il tipo di x
```

- ▶ Il compilatore non può garantire che il tipo di `x` sia compatibile con il tipo argomento della sequenza.
- ▶ Infatti il metodo è invocato tramite il riferimento `s`, **il cui tipo argomento per il compilatore è sconosciuto**

Sequenza<E> fornisce il metodo `public boolean add(E o)`

```
Sequenza<?> s = sq;  
...  
s.add(x) //non e' permessa qualunque sia il tipo di x
```

- ▶ Il compilatore non può garantire che il tipo di `x` sia compatibile con il tipo argomento della sequenza.
- ▶ Infatti il metodo è invocato tramite il riferimento `s`, **il cui tipo argomento per il compilatore è sconosciuto**
- ▶ È invece permessa

```
Sequenza<?> s;  
...  
s.add(null)
```

in quanto il letterale `null` è assegnabile a **qualunque** tipo riferimento.

È possibile limitare l'insieme dei tipi sostituibili al segnaposto a una parte della gerarchia

► **? extends T**

Il tipo sconosciuto può essere un qualunque sottotipo del tipo T indicato (compreso T stesso).

È possibile limitare l'insieme dei tipi sostituibili al segnaposto a una parte della gerarchia

► ? **extends T**

Il tipo sconosciuto può essere un qualunque sottotipo del tipo T indicato (compreso T stesso).

► ? **super T**

Il tipo sconosciuto può essere un qualunque supertipo del tipo T indicato (compreso T stesso).

## Esempio

```
//definizione variabile utilizzata, dopo la lettura, per
//riferirsi alla sequenza
Sequenza<? extends Figura> seq = null;

//lettura della sequenza
switch (scelta) {
case 'c':
    Sequenza<Cerchio> sc = new Sequenza<Cerchio>();
    ...legge una sequenza di cerchi e li memorizza in sc...

case 'q':
    Sequenza<Quadrato> sq = new Sequenza<Quadrato>();
    ...legge una sequenza di quadrati e li memorizza in sc...
}

//stampa delle aree
for (Figura f : seq)
    out.println(f.getArea());
```

# Tipi generici e vincoli sugli argomenti

```
class Sequenza<E>
```

- ▶ E indica la possibilità di fornire come argomento un qualunque tipo

# Tipi generici e vincoli sugli argomenti

```
class Sequenza<E>
```

- ▶ E indica la possibilità di fornire come argomento un qualunque tipo

```
class SequenzaOrdinata<E>
```

- ▶ E indicherebbe la possibilità di fornire come argomento un qualunque tipo

# Tipi generici e vincoli sugli argomenti

```
class Sequenza<E>
```

- ▶ E indica la possibilità di fornire come argomento un qualunque tipo

```
class SequenzaOrdinata<E>
```

- ▶ E indicherebbe la possibilità di fornire come argomento un qualunque tipo
- ▶ Non va bene!!

E deve essere un tipo che implementa l'interfaccia  
`Comparable<E>`

# Tipi generici e vincoli sugli argomenti

```
class Sequenza<E>
```

- ▶ E indica la possibilità di fornire come argomento un qualunque tipo

```
class SequenzaOrdinata<E>
```

- ▶ E indicherebbe la possibilità di fornire come argomento un qualunque tipo
- ▶ Non va bene!!

E deve essere un tipo che implementa l'interfaccia  
`Comparable<E>`

Questo viene specificato mediante un **vincolo sul tipo argomento E**

## Vincolo sull'argomento di SequenzaOrdinata

```
class SequenzaOrdinata<E extends Comparable<E>>
```

E può essere un qualunque tipo riferimento sottotipo di Comparable<E>

# Vincolo sull'argomento di SequenzaOrdinata

```
class SequenzaOrdinata<E extends Comparable<E>>
```

E può essere un qualunque tipo riferimento sottotipo di Comparable<E>

- Il compilatore ha tutte le informazioni che servono per verificare che il tipo utilizzato come parametro abbia le caratteristiche richieste.

Esempio

```
SequenzaOrdinata<String>
```

Va bene:

String implementa Comparable<String>



String è sottotipo di Comparable<String>

## Vincolo sull'argomento di SequenzaOrdinata

In realtà l'intestazione della classe **SequenzaOrdinata** è più complicata.

```
public class SequenzaOrdinata<E extends Comparable<? super E>>
```

**Tipo argomento:** un qualunque sottotipo di **Comparable<? super E>**