



## Capitolo 6

---

# Uso della gerarchia

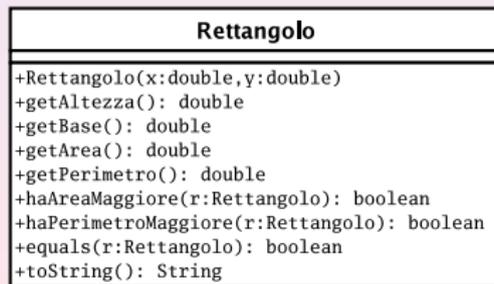
# Sommario: Uso della gerarchia

- 1 Notazione UML per le classi
- 2 Introduzione alla gerarchia
- 3 Gerarchia e tipi
  - La gerarchia di Java
  - Le classi Rettangolo e Quadrato
  - L'operatore instanceof
  - La classe Cerchio
  - La classe astratta Figura
  - Gerarchia: promozioni e cast
- 4 Le interfacce
  - L'interfaccia Iterable e il ciclo for-each
  - Tipi generici e gerarchia
  - Vincoli sui segnaposto
- 5 Tipi generici e vincoli sugli argomenti

## Rappresentazione concisa

**Rettangolo**

## Rappresentazione estesa



## 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 virgole

## Parametri

*nome\_parametro* : *tipo*

## Operazioni

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

- *lista\_dei\_parametri*

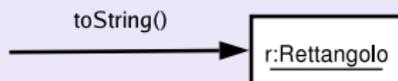
È una sequenza di dichiarazioni di parametro separate da virgole

## 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 del riferimento all'istanza



- All'interno del riquadro che rappresenta l'oggetto viene inserito il nome del riferimento all'istanza

## Sintassi

*nomeIstanza : nome\_classe*



- All'interno del riquadro che rappresenta l'oggetto viene inserito il nome del riferimento all'istanza

## Sintassi

*nomeIstanza : nome\_classe*

- Sia il nome del riferimento all'oggetto che il nome della classe sono opzionali

# Sommario: Uso della gerarchia

- 1 Notazione UML per le classi
- 2 Introduzione alla gerarchia
- 3 Gerarchia e tipi
  - La gerarchia di Java
  - Le classi Rettangolo e Quadrato
  - L'operatore instanceof
  - La classe Cerchio
  - La classe astratta Figura
  - Gerarchia: promozioni e cast
- 4 Le interfacce
  - L'interfaccia Iterable e il ciclo for-each
  - Tipi generici e gerarchia
  - Vincoli sui segnaposto
- 5 Tipi generici e vincoli sugli argomenti

- 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)
```

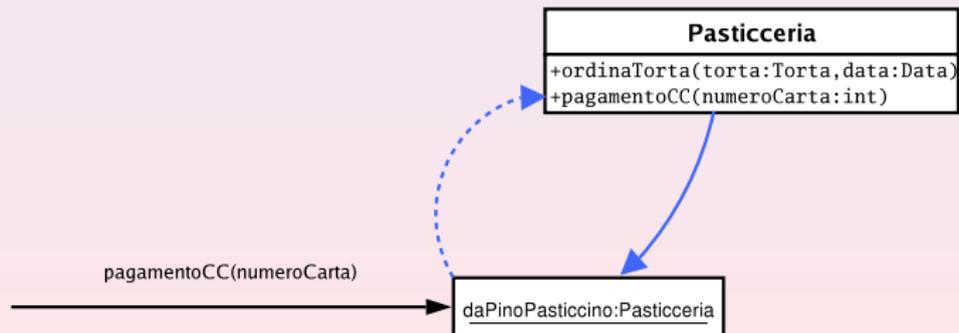
- Ogni pasticceria mette a disposizione questo metodo, Quindi ci aspettiamo che sia specificato nella classe `Pasticceria`
- Quando un oggetto riceve un messaggio che compare nel suo protocollo, risponde a tale messaggio eseguendo la sequenza di operazioni (il codice) del metodo relativo

```
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:



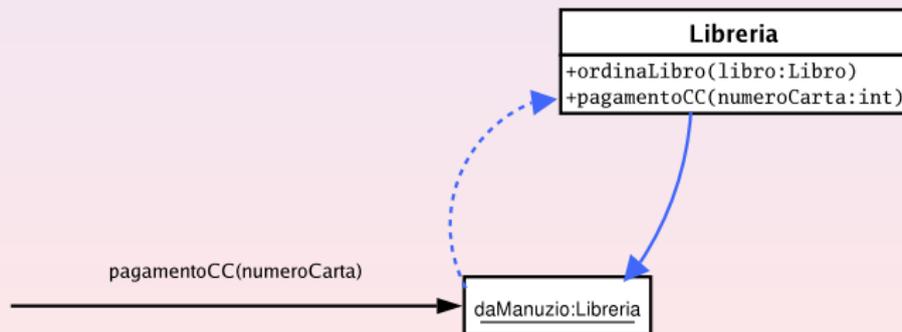
## Supponiamo ora di andare in libreria...

```
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:

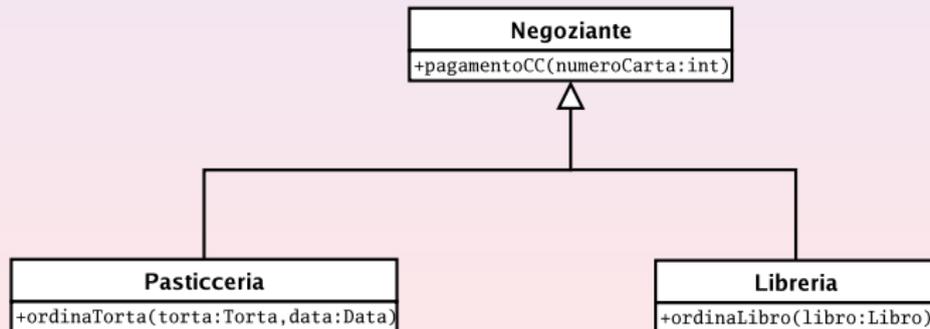


- Spesso i concetti vengono rappresentati in modo gerarchico

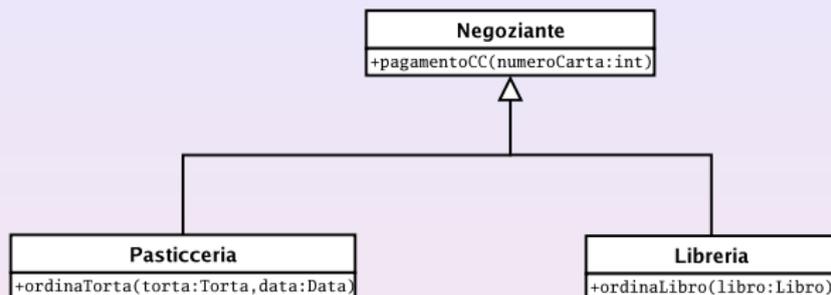
- Spesso i concetti vengono rappresentati in modo gerarchico
- Il messaggio `pagamentoCC` è proprio di una **categoria più ampia**, quella dei **Negozianti** (nuovo concetto/classe)

- 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**

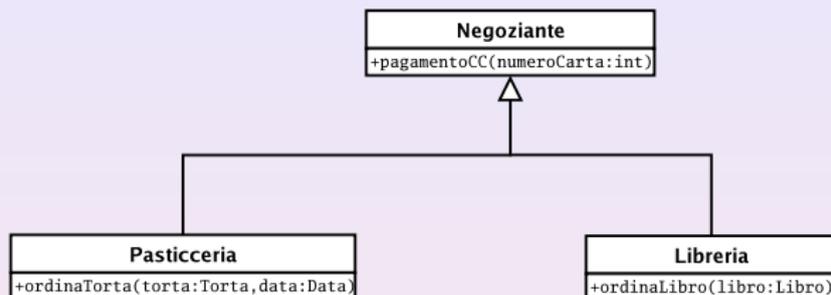


# Ricerca del metodo da eseguire



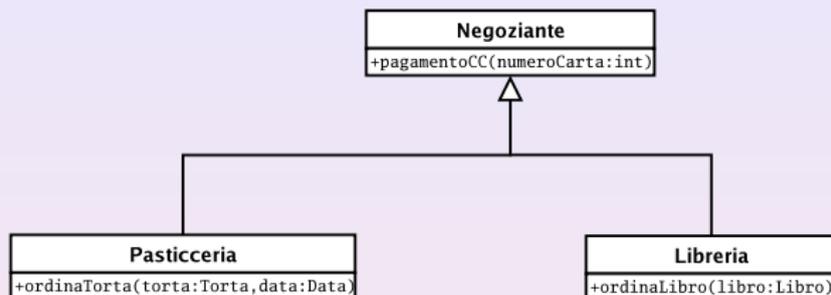
- Il metodo è definito nella classe **Negoziante** (superclasse)

# Ricerca del metodo da eseguire



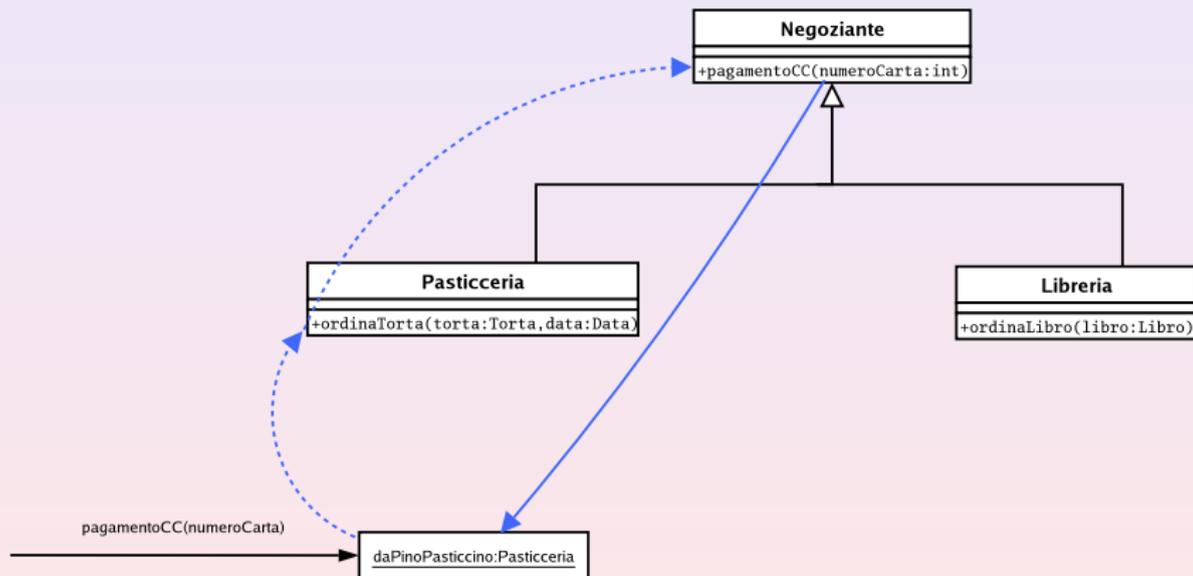
- Il metodo è definito nella classe **Negoziante** (superclasse)
- **Pasticceria** e **Libreria** (sottoclassi) lo ereditano

# Ricerca del metodo da eseguire



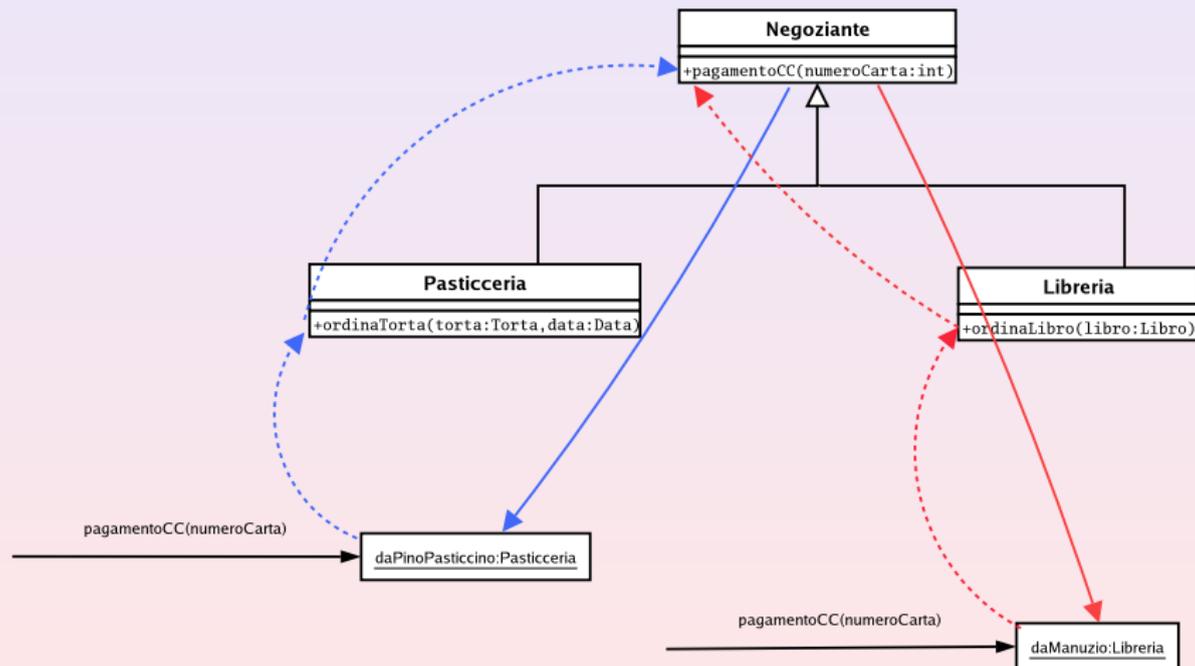
- Il metodo è definito nella classe **Negoziante** (superclasse)
- **Pasticceria** e **Libreria** (sottoclassi) lo ereditano
- La JVM deve cercare nella gerarchia il codice da eseguire in risposta ad un metodo

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



# Ricerca del metodo

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



# Sommario: Uso della gerarchia

- 1 Notazione UML per le classi
- 2 Introduzione alla gerarchia
- 3 Gerarchia e tipi
  - La gerarchia di Java
  - Le classi Rettangolo e Quadrato
  - L'operatore `instanceof`
  - La classe Cerchio
  - La classe astratta Figura
  - Gerarchia: promozioni e cast
- 4 Le interfacce
  - L'interfaccia `Iterable` e il ciclo `for-each`
  - Tipi generici e gerarchia
  - Vincoli sui segnaposto
- 5 Tipi generici e vincoli sugli argomenti

- 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**

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

## Relazione 'è un'

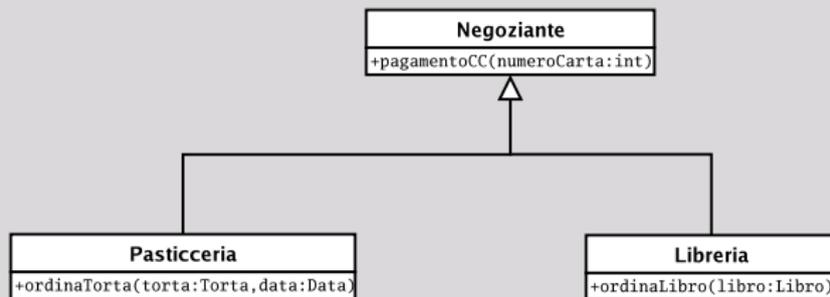
La relazione **sottoclasse/superclasse** definisce una relazione di tipo 'è un'.  
Ogni oggetto della sottoclasse è un oggetto della superclasse.

# 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();
```



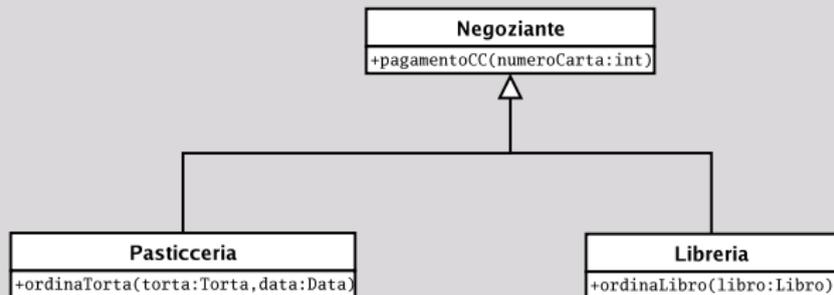
## Relazione 'è un'

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



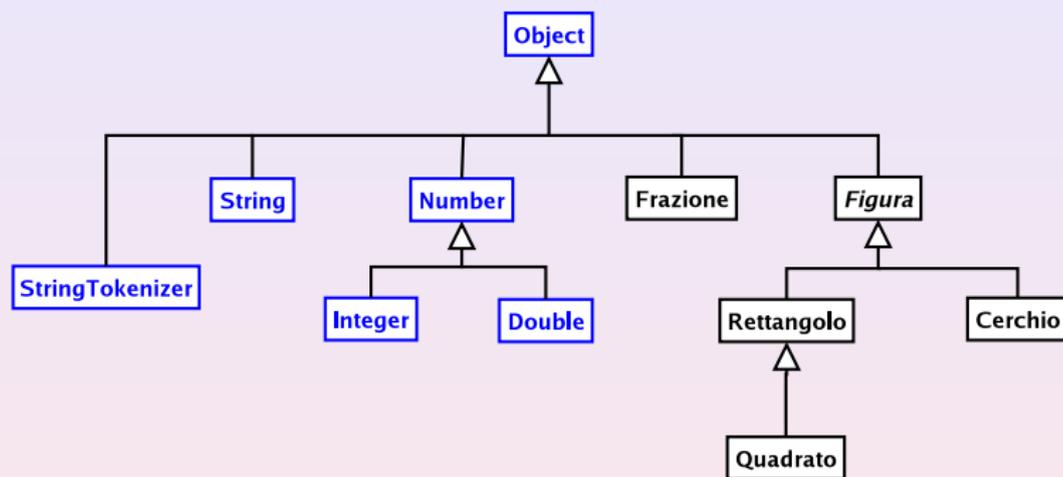
```
Negoziante n;
```

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

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

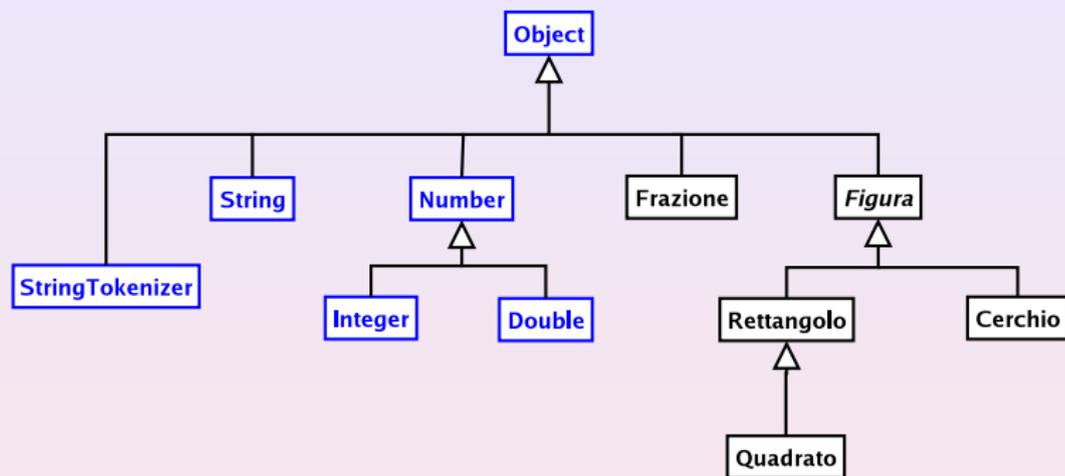
# La gerarchia di Java

È organizzata ad albero:



# La gerarchia di Java

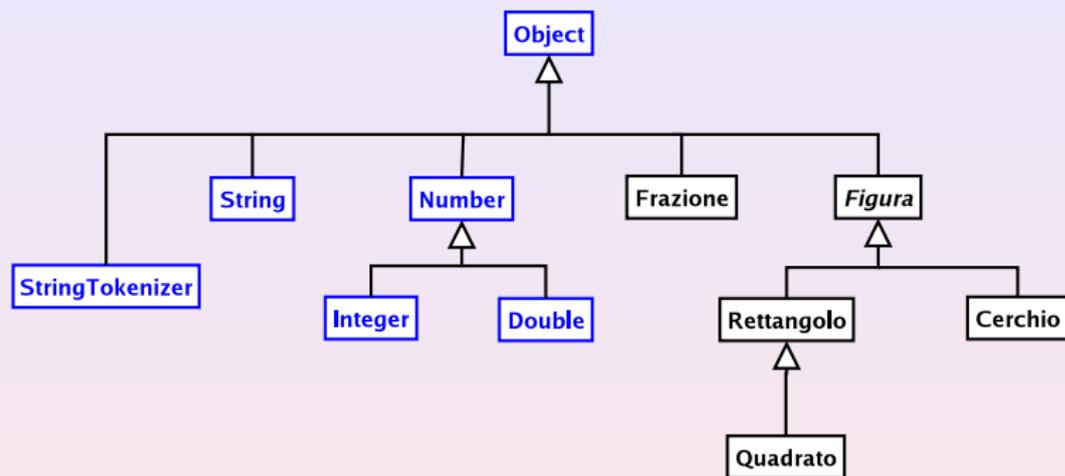
È organizzata ad albero:



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

# La gerarchia di Java

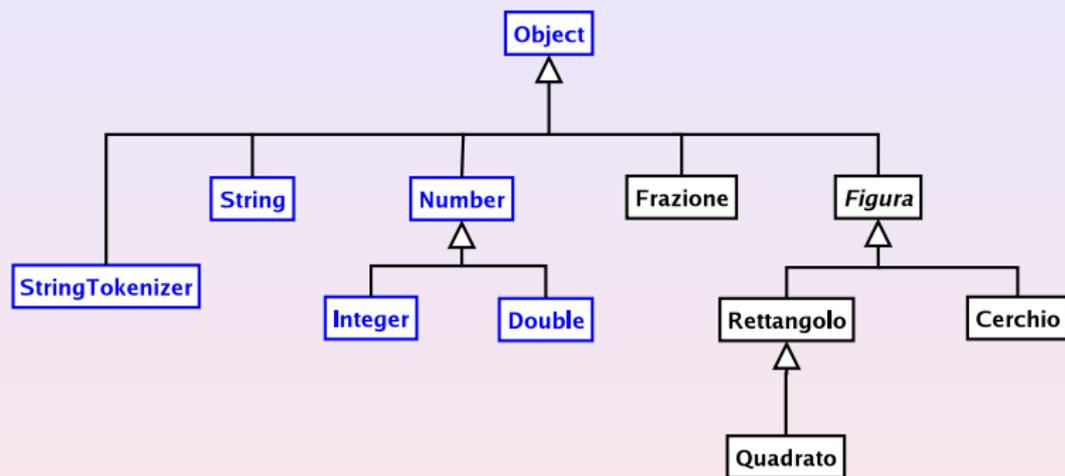
È organizzata ad albero:



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

# La gerarchia di Java

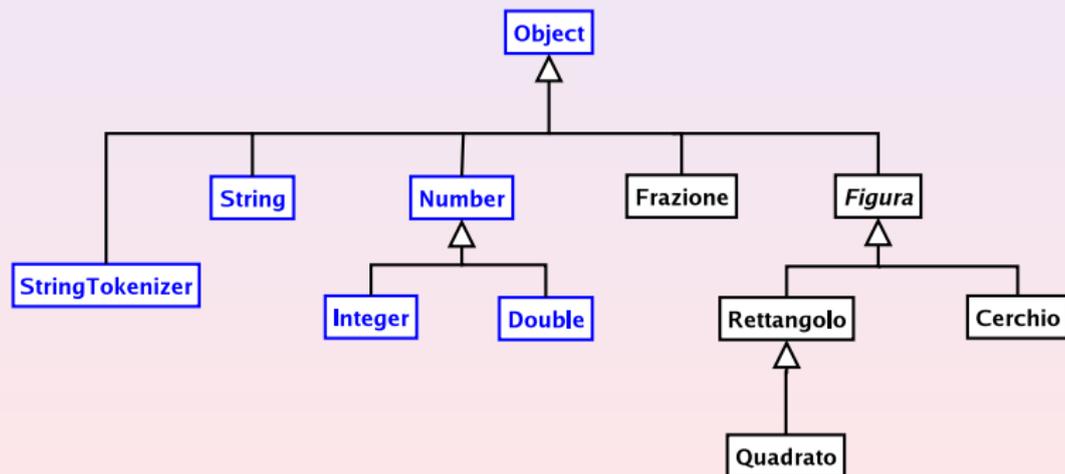
È organizzata ad albero:



- 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

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



## Contratto: Rettangolo

Le sue istanze rappresentano rettangoli.

## Contratto: 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.

# La classe Rettangolo

- `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"`

## Contratto: Quadrato

Le sue istanze rappresentano quadrati.

## Contratto: 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"`

- Geometricamente: i quadrati sono particolari rettangoli

# Quadrato e Rettangolo

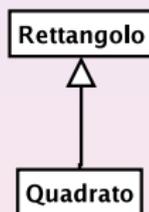
- Geometricamente: i quadrati sono particolari rettangoli
- Questa relazione è presente anche nell'implementazione

```
class Quadrato extends Rettangolo
```

# Quadrato e Rettangolo

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

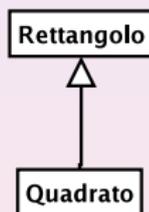
```
class Quadrato extends Rettangolo
```



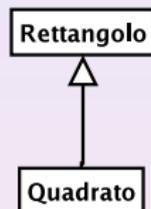
# Quadrato e Rettangolo

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

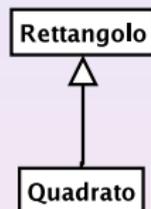
```
class Quadrato extends Rettangolo
```



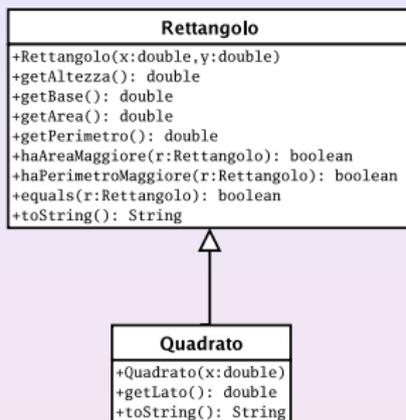
- Una classe che ne estende un'altra ne **eredita i metodi e i campi**



- **Quadrato** è una **sottoclasse** di **Rettangolo**

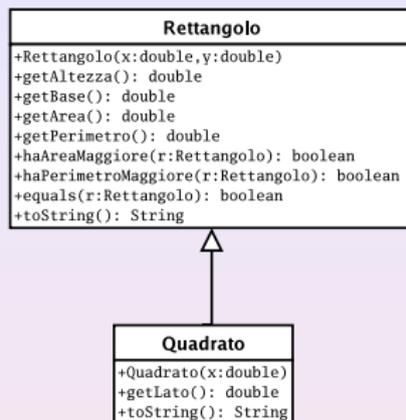


- **Quadrato** è una **sottoclasse** di **Rettangolo**
- **Rettangolo** è una **superclasse** di **Quadrato**



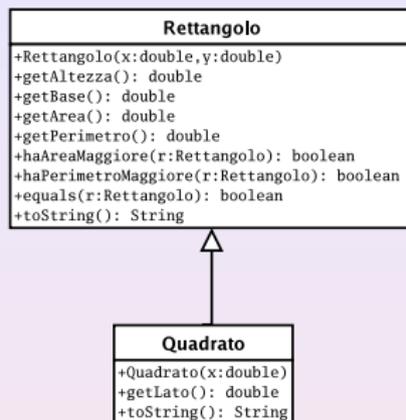
## Quadrato

- **eredita** i metodi di **Rettangolo**



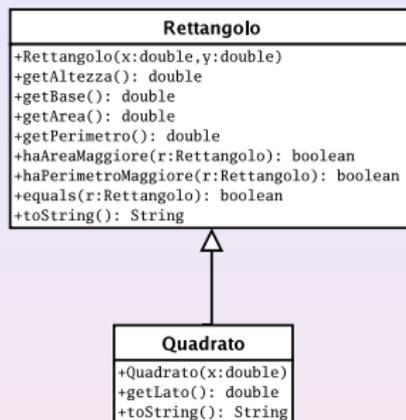
## Quadrato

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



## Quadrato

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



## 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 differenza da **Rettangolo**

- 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;
```

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

# Overriding

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

Quale metodo `toString` viene eseguito?

```
 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

```
 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 significati diversi* 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

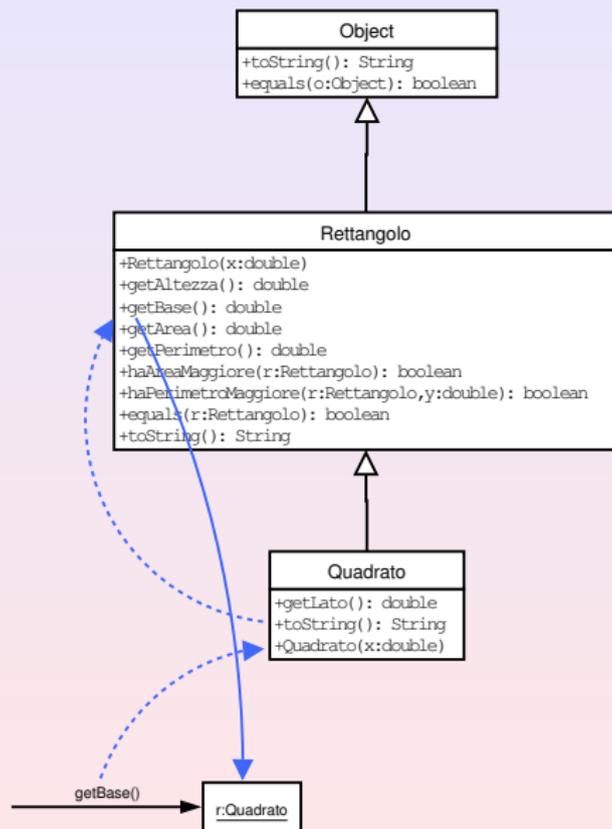
```
Rettangolo r;  
double x = in.readInt("Base?");  
double y = in.readInt("Altezza?");  
  
if (x == y)  
    r = new Quadrato(x);  
else  
    r = new Rettangolo(x,y);  
  
out.println(r.toString());
```

- Se x e y contengono 6 stampa

lato = 6.0

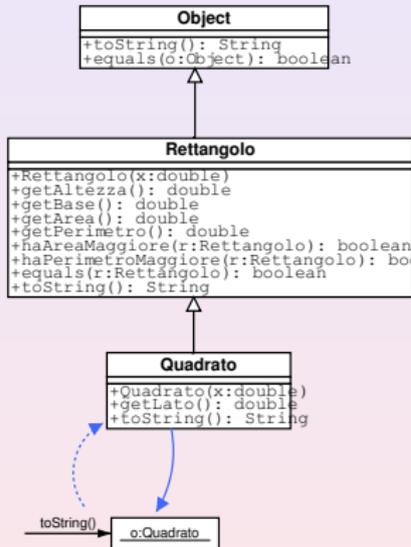
- Se x contiene 7 e y contiene 3 stampa

base = 7.0, altezza = 3.0

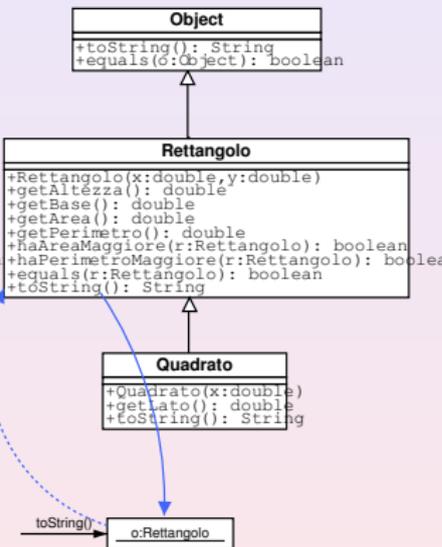


# Comportamento polimorfico

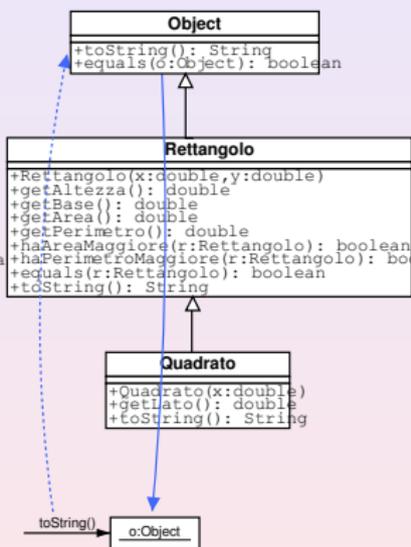
Object o = new Quadrato(3);



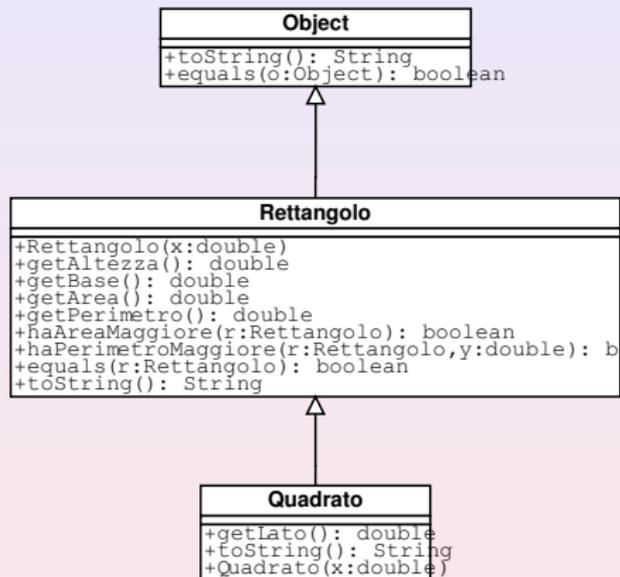
Object o = new Rettangolo(3,2);



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.

## 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.

## 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**

## 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*

## 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

## Contratto: Cerchio

Le sue istanze rappresentano cerchi.

## Contratto: 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.

# La classe Cerchio: 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"`

# 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...

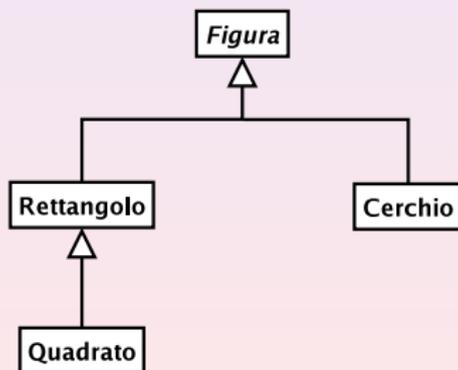
# La classe astratta Figura

- 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:

- `double getArea()`

- `double getPerimetro()`

# La classe astratta Figura

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

- `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`

# La classe astratta Figura

- Tutte le figure geometriche hanno un'area e un perimetro:
  - `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

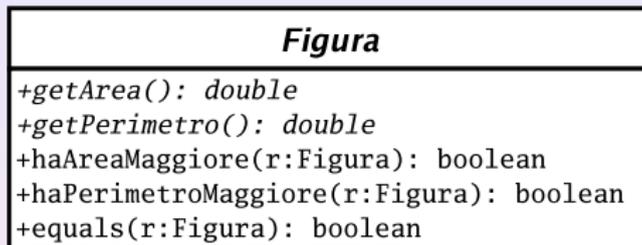
Metodi di cui è specificato il prototipo ma non l'implementazione.

- 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

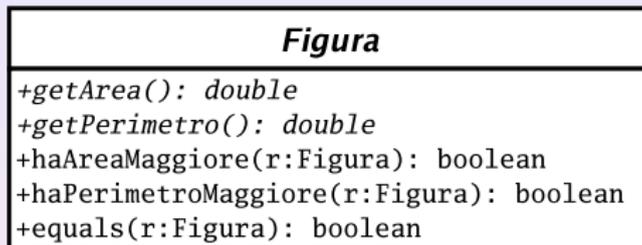
Metodi di cui è specificato il prototipo ma non l'implementazione.

- 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



- **Classi astratte**

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



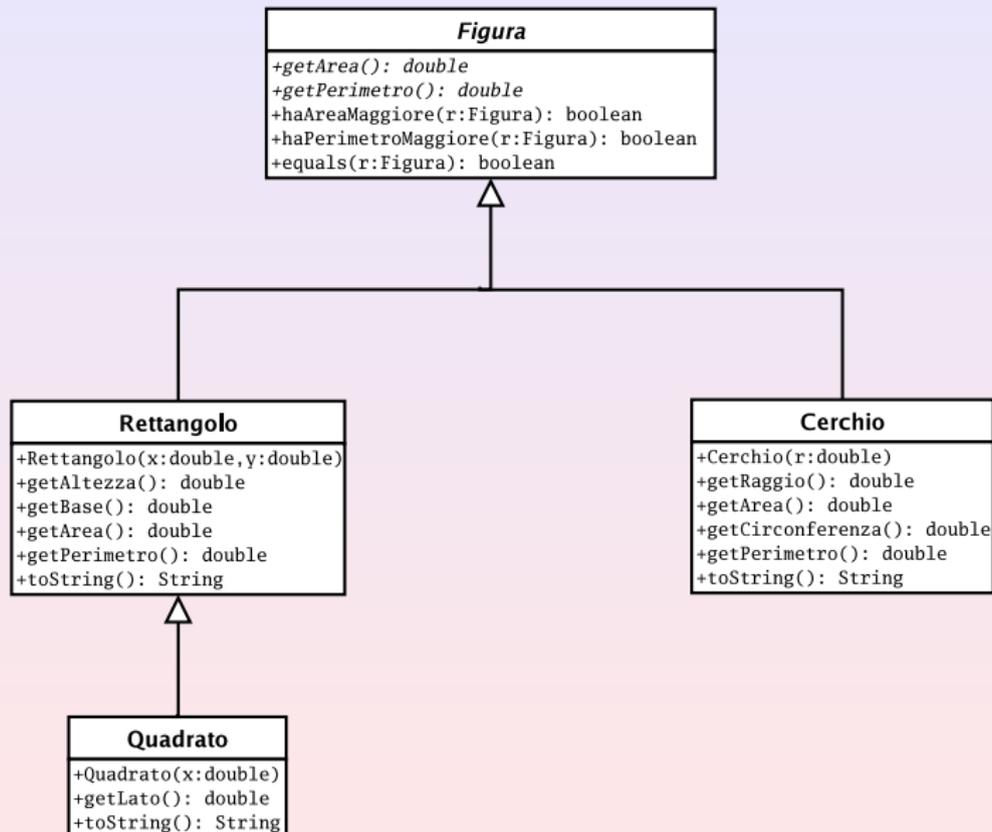
- **Classi astratte**

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

# La gerarchia di Figura



È possibile assegnare a una variabile il cui tipo sia una classe un riferimento a un [oggetto di una sottoclasse](#)

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);
```

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;
```

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  $x \neq y$

```
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());
```

# Sommario: Uso della gerarchia

- 1 Notazione UML per le classi
- 2 Introduzione alla gerarchia
- 3 Gerarchia e tipi
  - La gerarchia di Java
  - Le classi Rettangolo e Quadrato
  - L'operatore instanceof
  - La classe Cerchio
  - La classe astratta Figura
  - Gerarchia: promozioni e cast
- 4 **Le interfacce**
  - L'interfaccia Iterable e il ciclo for-each
  - Tipi generici e gerarchia
  - Vincoli sui segnaposto
- 5 Tipi generici e vincoli sugli argomenti

# SequenzaOrdinata<E>

Contratto: SequenzaOrdinata<E>

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

## Contratto: SequenzaOrdinata<E>

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

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

## Contratto: 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
```

## Esempi

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

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

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

## 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

- **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

- `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`:

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

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

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

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) Dichiarandolo nell'intestazione

# Implementare un'interfaccia

Una classe implementa un'interfaccia:

(1) Dichiarandolo 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) Dichiarandolo 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) Dichiarandolo 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 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

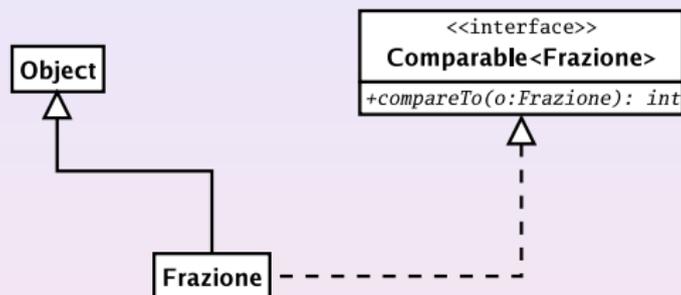
- Il nome di un'interfaccia definisce un **tipo riferimento**
- È un supertipo per tutte le classi che implementano l'interfaccia
- Valgono tutte le regole (promozioni e cast) che valgono per gli altri tipi riferimento

## Esempio

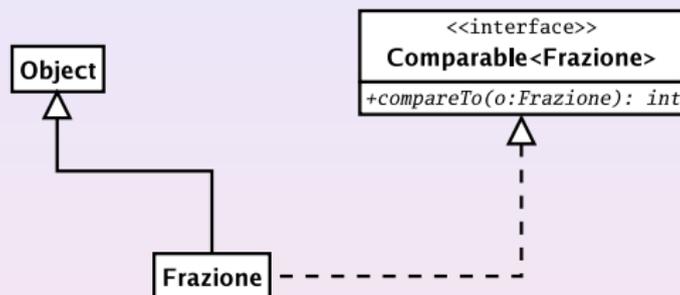
```
Comparable<Frazione> c;
```

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

# 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.

`Sequenza<E>` implementa `Iterable<E>` quindi:

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

Restituisce un oggetto di tipo `Iterator<Frazione>`

`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:

- `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:

- `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.

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

```
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-each

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

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

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:

- `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:

- `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** (cioè ogni tipo riferimento 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** (cioè ogni tipo riferimento è sottotipo di **Object**)

## Classi

Ogni classe è sottotipo di **Object**

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

# 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** (cioè ogni tipo riferimento è sottotipo di **Object**)

## Classi

Ogni classe è sottotipo di **Object**

- Direttamente: se **non estende esplicitamente alcuna classe**
- Indirettamente: se estende una classe diversa da **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** (cioè ogni tipo riferimento è sottotipo di **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** (cioè ogni tipo riferimento è sottotipo di **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**

- 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
```

# 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>();
```

# Osservazione

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());
```

# Sommario: Uso della gerarchia

- 1 Notazione UML per le classi
- 2 Introduzione alla gerarchia
- 3 Gerarchia e tipi
  - La gerarchia di Java
  - Le classi Rettangolo e Quadrato
  - L'operatore instanceof
  - La classe Cerchio
  - La classe astratta Figura
  - Gerarchia: promozioni e cast
- 4 Le interfacce
  - L'interfaccia Iterable e il ciclo for-each
  - Tipi generici e gerarchia
  - Vincoli sui segnaposto
- 5 Tipi generici e vincoli sugli argomenti

# 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: non tutti i tipi sono ordinabili

**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: non tutti i tipi sono ordinabili

**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>`