

# Polimorfismo

---

Dal materiale di  
Stefano Ferretti e Angelo Di Iorio  
[s.ferretti@unibo.it](mailto:s.ferretti@unibo.it)  
[Angelo.diiorio@unibo.it](mailto:Angelo.diiorio@unibo.it)

# Polimorfismo

- Il concetto di ereditarietà è strettamente legato a quello di **polimorfismo**
  - Dal greco *πολύμορφος*: *πολυ-* = "poli-" e *μορφή* = "forma"
- In **Informatica** il concetto di polimorfismo è principalmente distinto in:
  - **Ad hoc**
  - **Parametrico**
  - **Per inclusione**

# Polimorfismo

- **Ad hoc**: un'operazione avente **stesso nome** si può comportare in modo diverso a seconda del **tipo/numero** dei suoi parametri
  - **Overloading** di metodi: stesso nome, diversa **firma**
  - Es. **+** si comporta in modo diverso a seconda che i suoi parametri siano stringhe o numeri
- **Parametrico**: Uno o più tipi sono specificati come **parametri** anziché fissati
  - Es. un oggetto della classe **ArrayList<T>** può avere “diverse forme” a seconda del **tipo parametrico T**

# Polimorfismo

- **Per inclusione** o di **sottotipo**: ogni operazione definita su oggetti di una certa **classe A** deve funzionare correttamente per **qualsiasi** oggetto di una sua **sottoclasse B**
- Es. Un metodo `getAge( )` di una classe `Persona` deve ritornare l'età corretta per ogni istanza di `Persona` e di *ogni* sua **sottoclasse**
  - Es. `Donna`, `Uomo`, `Studente`, `Lavoratore`, ...
  - `getAge( )` può avere “*forme diverse*” a seconda della **sottoclasse** che lo sovrascrive

# Esempio

```
public class Persona {  
    private int age;  
    ...  
    int getAge() { return age; }  
}
```

```
public class Donna extends Persona {  
    ...  
    int getAge() {  
        if (age > ...) throw new WrongQuestionException();  
        return super.getAge();  
    }  
}
```

```
public class SimpaticoneDiTurno extends Uomo {  
    ...  
    int getAge() { return 18; }  
}
```

# Esempio: BankAccount

Qui costruisco due **oggetti diversi**

```
BankAccount bob = new BankAccount(123, "Bob", 345.50);  
JointBankAccount bobMary =  
    new JointBankAccount(345, "Bob", "Mary", 450.65);
```

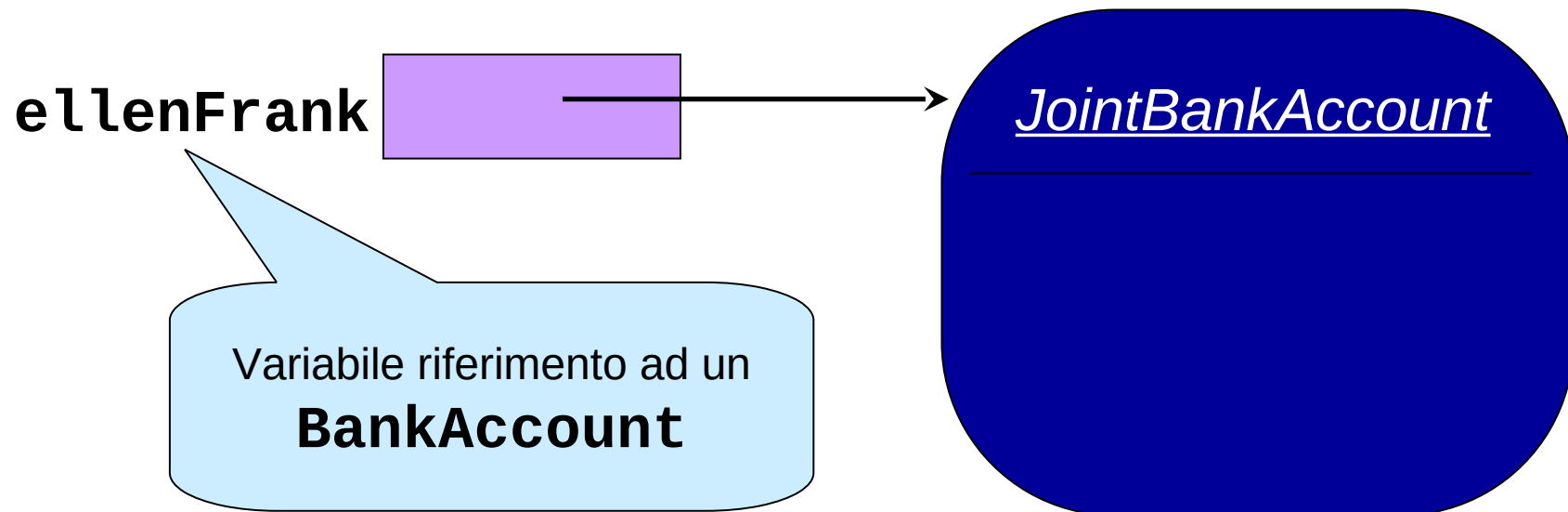
Qui uso l'ereditarietà: JointBankAccount è **un**  
(IS-A) *"tipo di"* BankAccount

```
BankAccount ellenFrank = new JointBankAccount(  
    456, "Ellen", "Frank", 3450.99);
```

ellenFrank è come se avesse 2 tipi: uno **statico**  
(BankAccount) e uno **dinamico** (JointBankAccount)

# Esempio: BankAccount

```
BankAccount ellenFrank =  
    new JointBankAccount(456, "Ellen", "Frank", 3450.99);
```



# Esempio: BankAccount

**ATTENZIONE!** Il codice seguente è **scorretto**: un oggetto di tipo **BankAccount** *non è necessariamente* di tipo **JointBankAccount**

```
JointBankAccount fred = ILLEGAL!!  
                        new BankAccount(123, "Fred", 345.50);
```

- E' possibile assegnare il riferimento di un oggetto di una **sottoclasse** ad un riferimento **superclasse** **ma non viceversa**
  - Nel nostro caso, **JointBankAccount** è un tipo di **BankAccount** ma il contrario è falso



# Esempio

```
// Questo è OK: Donna IS-A Persona  
Persona alice = new Donna("Alice Rossi");
```

```
// Questo NO: Non sempre Persona IS-A Donna  
Donna alice = new Persona("Alice Rossi");
```

```
// Questo è OK:  
Persona alice = new Persona("Alice Rossi");  
Donna alice2 = new Donna("Alice Rossi");
```

# Type casting

```
JointBankAccount fredMary =  
    new JointBankAccount(345, "Fred", "Mary", 450.65);
```

Queste istruzioni sono **legali**



```
String owner = fredMary.getName();  
String jointOwner = fredMary.getJointName();
```

# Type casting

```
BankAccount ellenFrank = new JointBankAccount(  
    456, "Ellen", "Frank", 3450.99);
```

L'istruzione seguente è **legale**

```
String name = ellenFrank.getName();
```

L'istruzione seguente **non è legale**: BankAccount **non ha** un metodo getJointName

```
String jointName = ellenFrank.getJointName();
```

In questo caso è **necessario** fare un **typecast**

```
String jointName =  
    ((JointBankAccount) ellenFrank).getJointName();
```

# Object amnesia

```
BankAccount ellenFrank =  
    new JointBankAccount(456, "Ellen", "Frank", 3450.99);
```

In questo caso l'oggetto “**dimentica**” che è un **JointBankAccount** poiché assegnato ad una variabile riferimento di tipo **BankAccount**. Questo fenomeno si chiama **object amnesia**

È necessario fare un **typecast** per “**ricordare**” che l'oggetto è in realtà di tipo **JointBankAccount**

```
String jointName =  
    ((JointBankAccount) ellenFrank).getJointName();
```

parentesi necessarie

# Demo: AccountTester

```
public class AccountTester{
    public void doTest(){
        JointBankAccount fredMary = new
            JointBankAccount(123, "Fred", "Mary", 1000);
        BankAccount ellenFrank = new
            JointBankAccount(345, "Ellen", "Frank", 1000);

        String jointName1 = fredMary.getJointName();
        String jointName2 =
            ((JointBankAccount) ellenFrank).getJointName();

        System.out.println("Joint name 1 is " + jointName1);
        System.out.println("Joint name 2 is " + jointName2);
        . . . .
    }
}
```

# Esempio

Questo si può fare perché `Point` e `Circle` sono **sottoclassi** di `Object`

```
Object p = new Point(3,4);  
Object c = new Circle(p,5);
```

Entrambi gli oggetti **dimenticano** il loro tipo originali: è necessario un **cast** se si vuole accedere ai loro metodi specifici

## Illegale

```
p.getX()  
c.getCenter()
```

## Legale

```
((Point) p).getX()  
((Circle) c).getCenter()
```

# Esempio: account transfer

- Es: dobbiamo scrivere un metodo **transfer** che prende in input 3 parametri:
  - Due conti correnti **from** e **to**
  - Una quantità **amount** da trasferire dal conto **from** verso il conto **to**
- Il metodo deve funzionare sia per parametri di tipo **BankAccount** che per parametri di tipo **JointBankAccount**
- **Soluzione**: Usare il **polimorfismo**

# Soluzione non-polimorfa

**Senza polimorfismo** avremmo bisogno di 4 metodi sovraccaricati (*overloaded*)

```
public void transfer(BankAccount from,  
    BankAccount to, double amount) {...}
```

```
public void transfer(BankAccount from,  
    JointBankAccount to, double amount) {...}
```

```
public void transfer(JointBankAccount from,  
    BankAccount to, double amount) {...}
```

```
public void transfer(JointBankAccount from,  
    JointBankAccount to, double amount) {...}
```



# Soluzione polimorfa

In realtà, col polimorfismo basta **solo un metodo**

```
public void transfer(  
    BankAccount from,  
    BankAccount to,  
    double amount) {  
    from.withdraw(amount);  
    to.deposit(amount);  
}
```



Si usa la classe  
**“più generica”**

Si può fare perchè ogni oggetto **JointBankAccount**  
"è un" oggetto **BankAccount**

# Metodi polimorfi

- In una **gerarchia** di classi possiamo avere metodi con “*forme diverse*”
  - una per ogni **sottoclasse** della gerarchia
- Ogni sottoclasse può definire una diversa **versione sovrascritta** del metodo di una classe antenata
- Questi metodi sono detti **metodi polimorfi**
  - Esempio standard: **toString()**

# Metodi polimorfi

- **NOTA:** Fare **overriding** di un metodo è diverso da fare **overloading** di un metodo
  - *Overloading = polimorfismo ad hoc*
- Es. di overloading:
  - `public void println()`
  - `public void println(String s)`
  - `public void println(int n)`

Diversi metodi della **stessa classe**, con **nome uguale** ma **firma diversa**

# Metodo polimorfo toString

- Nella gerarchia dei conti bancari ci sono **tre** versioni del metodo **toString**
  - La versione di *default* della classe **Object**.  
Viene usata se non si fa override nelle sottoclassi
  - Una per la classe **BankAccount**
  - Una per la classe **JointBankAccount**
- Il sistema Java decide a **run-time** la versione corretta da eseguire

# Demo: AccountTester2

```
public class AccountTester2{
    public void doTest() {
        BankAccount fred = new BankAccount(456, "Fred", 500);
        JointBankAccount fredMary =
            new JointBankAccount(123, "Fred", "Mary", 1000);
        BankAccount ellenFrank =
            new JointBankAccount(345, "Ellen", "Frank", 1000);
        // Quale metodo toString verrà invocato
        // nelle seguenti chiamate ?
        System.out.println(fred);
        System.out.println(fredMary);
        System.out.println(ellenFrank);
    }
    public static void main(String[] args) {
        new AccountTester2.doTest();
    }
}
```

# AccountTester2 class

Output del metodo `doTest()`

```
BankAccount[456, Fred, 500.0]  
JointBankAccount[BankAccount[123, Fred, 1000.0], Mary]  
JountBankAccount[BankAccount[345, Ellen, 1000.0],  
Frank]
```

Anche se `ellenFrank` è un riferimento alla *superclasse* (il **tipo statico** a **compile-time** è `BankAccount`), il **tipo dinamico** a **run-time** è `JointBankAccount` (viene costruito un oggetto di tale classe) quindi viene invocato il metodo `toString` della *sottoclasse*

# Final

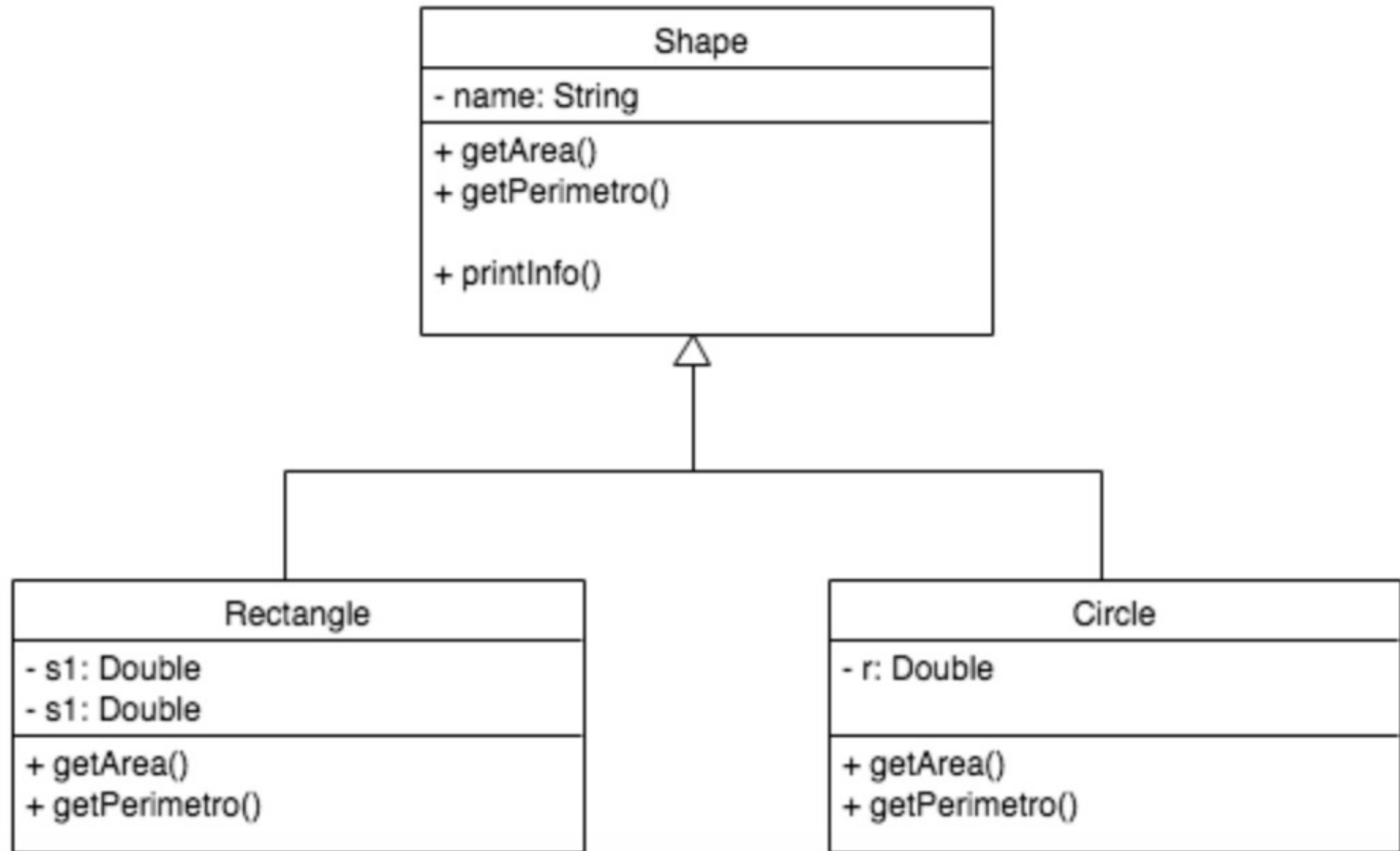
- Si può **inibire** la ridefinizione di un **metodo** con **final**

```
public final void f() {...}
```

- Anche una **classe** può essere dichiarata **final** per inibire la sua estensione

```
public final class C {...}
```

# Esempio: forme geometriche





# Classe Shape

```
public class Shape {  
    public double getArea() {  
        return 0;  
    }  
    public double getPerimeter() {  
        return 0;  
    }  
    public void printInfo(){  
        System.out.println(  
            "Perimeter: " + this.getPerimeter() +  
            " - Area: " + this.getArea());  
    }  
}
```

**NON è il modo migliore per gestire questi casi**

**Meglio usare classi astratte**  
(che vedremo più avanti)

# Classe Rectangle

```
public class Rectangle extends Shape {  
    double s1;  
    double s2;  
  
    public Rectangle(double s1, double s2) {  
        this.s1 = s1;  
        this.s2 = s2;  
    }  
  
    @Override  
    public double getArea() {  
        return s1 * s2;  
    }  
  
    @Override  
    public double getPerimeter() {  
        return (s1 + s2) * 2;  
    }  
}
```

# Classe Circle

```
public class Circle extends Shape {  
  
    double r;  
  
    public Circle(double r) {  
        this.r = r;  
    }  
  
    @Override  
    public double getArea() {  
        return r * r * 3.14;  
    }  
  
    @Override  
    public double getPerimeter() {  
        return 2 * 3.14 * r;  
    }  
  
}
```

# GeometryDemo

```
public class GeometryDemo {  
  
    public static void main(String[] args) {  
  
        Rectangle r = new Rectangle(2, 3);  
        Circle c = new Circle(1);  
        r.printlnInfo();  
        c.printlnInfo();  
    }  
}
```

Perimeter: 10.0 - Area: 6.0

Perimeter: 6.28 - Area: 3.14

# GeometryDemo

```
public class GeometryDemo {  
  
    public static void main(String[] args) {  
  
        Shape[] shapes = new Shape[2];  
        shapes[0] = new Rectangle(2, 3);  
        shapes[1] = new Circle(1);  
        for (int i = 0; i < shapes.length; i++)  
            shapes[i].printlnInfo();  
    }  
}
```

Perimeter: 10.0 - Area: 6.0

Perimeter: 6.28 - Area: 3.14

# Static methods

```
public final void printInfo(){
```

```
    this.printShapeName();
```

```
    System.out.println("Perimeter: "+getPerimeter()+" - Area: "+getArea());
```

```
}
```

```
public static void printShapeName(){
```

```
    System.out.print("I'm a Shape. ");
```

```
}
```

Shape.java

```
public static void printShapeName(){
```

```
    System.out.print("I'm a Rectangle. ");
```

```
}
```

Rectangle.java

```
public static void printShapeName(){
```

```
    System.out.print("I'm a Circle. ");
```

```
}
```


Circle.java

# GeometryDemo

```
public class GeometryDemo {  
  
    public static void main(String[] args) {  
  
        Rectangle r = new Rectangle(2, 3);  
        Circle c = new Circle(1);  
        r.printInfo();  
        c.printInfo();  
    }  
  
    Cosa stampa?  
}
```

# GeometryDemo

```
public class GeometryDemo {  
  
    public static void main(String[] args) {  
  
        Rectangle r = new Rectangle(2, 3);  
        Circle c = new Circle(1);  
        r.printInfo();  
        c.printInfo();  
    }  
  
    // Stampa questo:  
}
```



I'm a Shape. Perimeter: 10.0 - Area: 6.0  
I'm a Shape. Perimeter: 6.28 - Area: 3.14



# GeometryDemo

```
public class GeometryDemo {  
  
    public static void main(String[] args) {  
  
        Rectangle r = new Rectangle(2, 3);  
        Circle c = new Circle(1);  
        r.printlnInfo();  
        c.printlnInfo();  
    }  
  
    // ...Ma in realtà vorrei questo:  
}
```

I'm a Rectangle. Perimeter: 10.0 - Area: 6.0

I'm a Circle. Perimeter: 6.28 - Area: 3.14

# Static methods

```
public final void printInfo(){
```

```
    this.printShapeName();
```

```
    System.out.println("Perimeter: "+getPerimeter()+" - Area: "+getArea());
```

```
}
```

```
public static void printShapeName(){System.out.print("I'm a Shape. ");}
```

Shape.java

```
public static void printShapeName(){
```

```
    System.out.print("I'm a Rectangle. ");
```

```
}
```

Rectangle.java

```
public static void printShapeName(){
```

```
    System.out.print("I'm a Circle. ");
```

```
}
```

Circle.java

# Upcast e downcast

- **Upcast:** assegnare ad un oggetto della *superclasse* un oggetto della *sottoclasse*
  - Mi sposto “*verso l’alto*” nella gerarchia
  - Es. `Persona gb = new Studente("G. Bianchi");`
- **Downcast:** convertire una *superclasse* in *sottoclasse*
  - Mi sposto “*verso il basso*” nella gerarchia
  - Es. `Studente s = (Studente) x; // supponendo x di tipo Object`
- **ATTENZIONE:** Il downcast non sempre ha senso! E’ compito del programmatore fare downcasts sensati
  - ad es. nei metodi **`equals(Object x)`**
  - In questo caso può essere utile l’operatore **`instanceof`**

# Esercizio

- Definire una classe **Pet**, avente un campo privato **annoNascita** (di tipo `int`) e due classi **Dog** e **Cat** come **sottoclassi** di **Pet**
- La classe **Dog** deve contenere un metodo **getAnniUmani(x)** dove **x** è l'anno corrente, che ritorna l'età in "*anni umani*" calcolati come segue:
  - $< 1$  anno di età = 5 anni umani
  - 1 anno di età = 15 anni umani
  - 2 anni di età = 24 anni umani
  - 3+ anni di età = aggiungere 4 anni per ogni anno dopo i due anni
    - Es. 3 anni =  $24 + 4 = 28$  anni umani; 4 anni = 32 anni umani; 5 anni = 36 anni umani...
- Per **Cat** assumiamo per semplicità **anni umani = anni del gatto \* 7**
- Definire una classe **PetZoo** con un costruttore **PetZoo(x)** dove **x** è l'anno corrente e un metodo **double etaMedia(Pet[] a)** che prende in input un vettore di **Pet** e ritorna l'età media in anni umani
  - Creare una demo per **testare** `etaMedia`