# Ereditarietà e Polimorfismo

- I concetti principali
- Ereditarietà in Java
- La classe Object
- Classi astratte

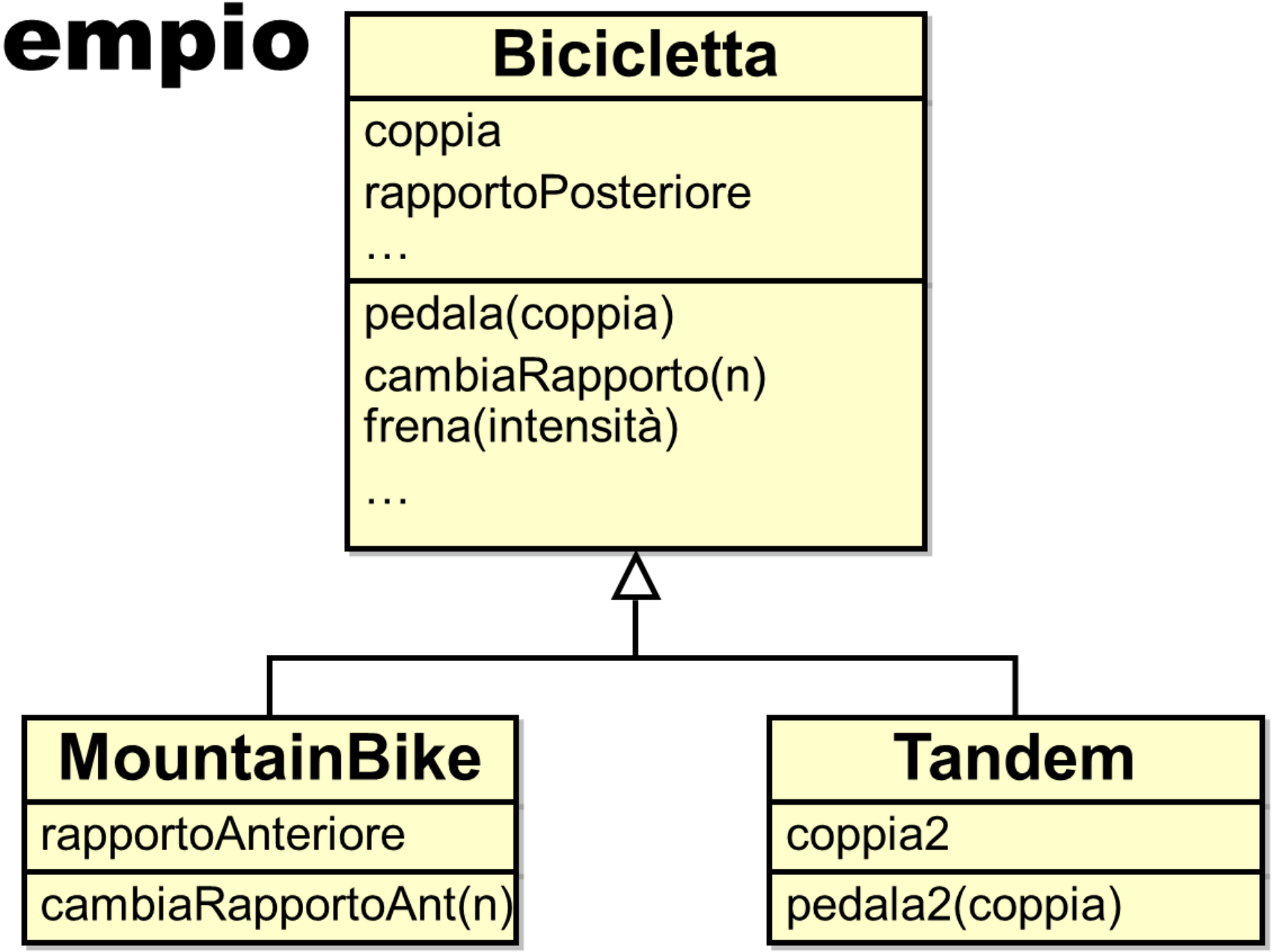
# Riusare il software

- A volte si incontrano classi con funzionalità simili
  - In quanto sottendono concetti semanticamente “vicini”
  - Una mountain bike assomiglia ad una bicicletta tradizionale
- È possibile creare classi disgiunte replicando le porzione di stato/comportamento condivise
  - L’approccio “Taglia&Incolla”, però, non è una strategia vincente
  - Difficoltà di manutenzione correttiva e perfettiva
- Meglio “specializzare” codice funzionante
  - Sostituendo il minimo necessario

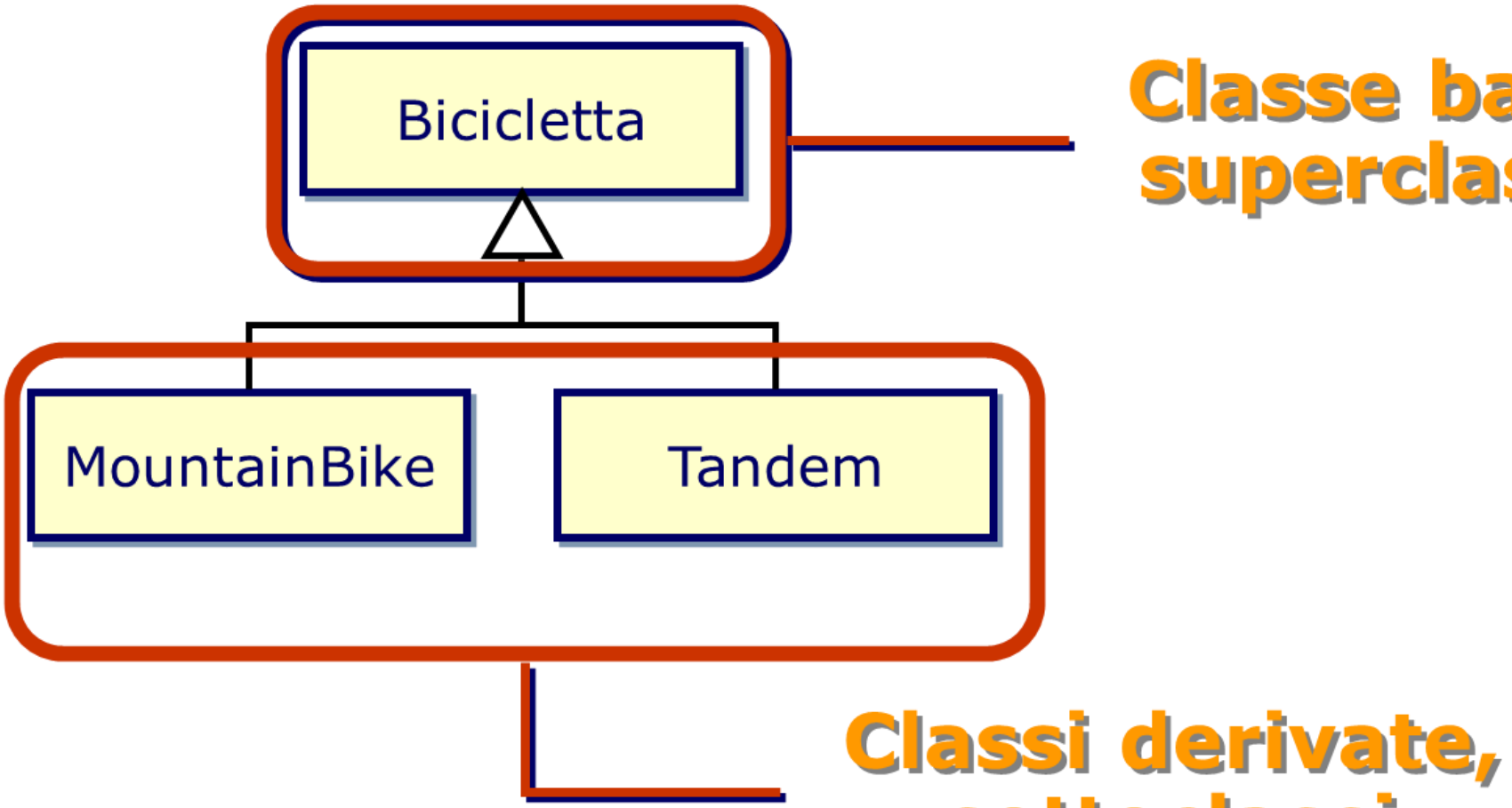
# Ereditarietà

- Meccanismo per definire una nuova classe (classe derivata) come specializzazione di un'altra (classe base)
  - La classe **base** modella un concetto **generico**
  - La classe **derivata** modella un concetto **più specifico**
- La classe derivata:
  - Dispone di tutte le funzionalità (attributi e metodi) di quella base
  - Può aggiungere funzionalità proprie
  - Può ridefinirne il funzionamento di metodi esistenti (polimorfismo)

# Esempio

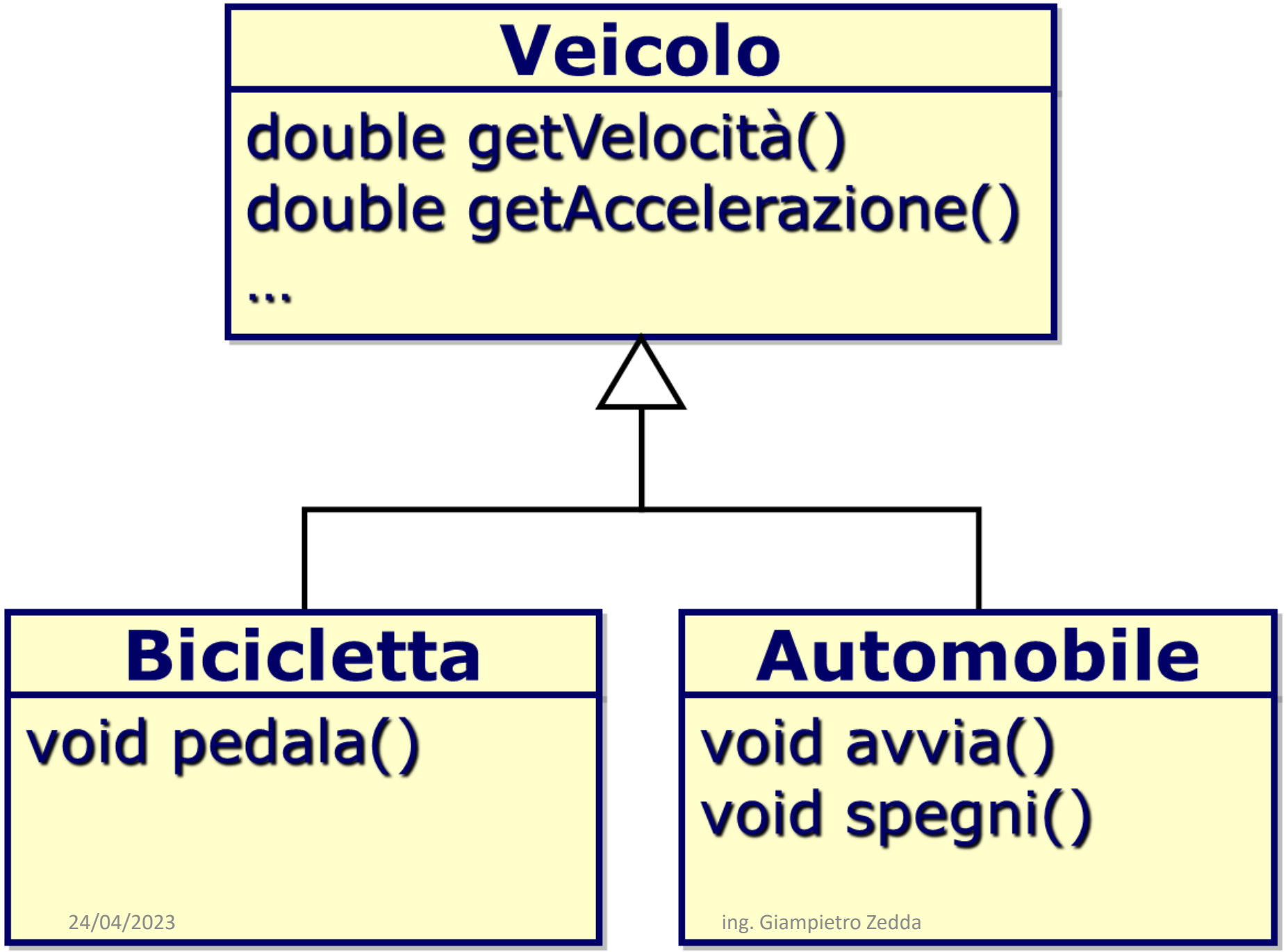

Bicicletta
coppia
rapportoPosteriore
…
pedala(coppia)
cambiaRapporto(n)
frena(intensità)
…
MountainBike
rapportoAnteriore
cambiaRapportoAnt(n)
Tandem
coppia2
pedala2(coppia)

# Terminologia



# Astrazione

- Il processo di analisi e progettazione del software di solito procede per raffinamenti successivi
  - Spesso capita che le similitudini tra classi non siano colte inizialmente
  - In una fase successiva, si coglie l'esigenza/opportunità di introdurre un concetto più generico da cui derivare classi specifiche
- Processo di astrazione
  - Si introduce la superclasse che "astrae" il concetto comune condiviso dalle diverse sottoclassi
  - Le sottoclassi vengono "spogliate" delle funzionalità comuni che migrano nella superclasse

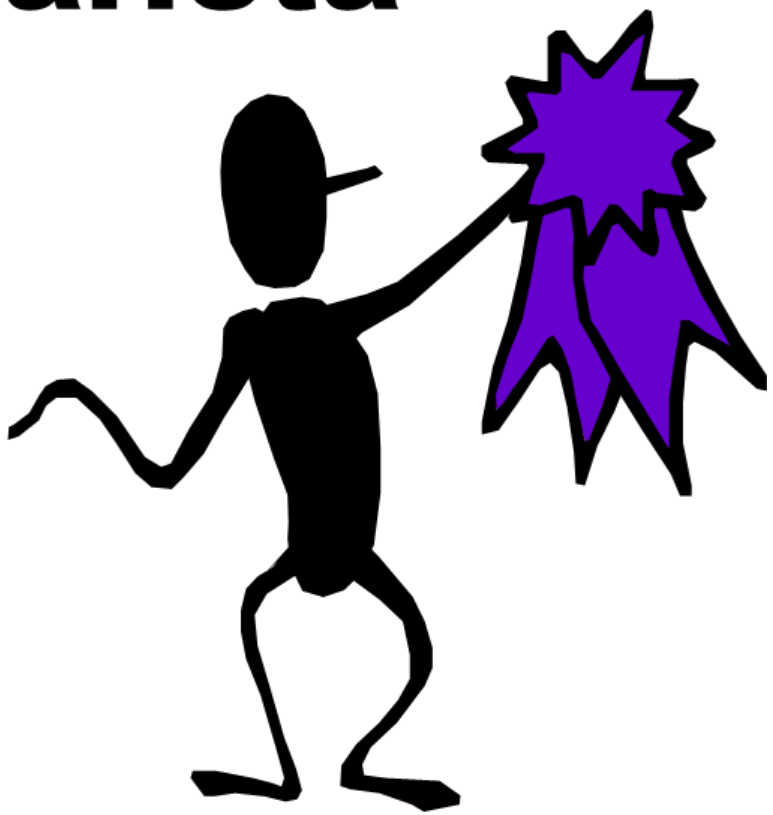

Veicolo
double getVelocità()
double getAccelerazione()
...
Bicicletta
void pedala()
Automobile
void avvia()
void spegni()

# Tipi ed ereditarietà

- Ogni classe definisce un tipo:
  - Un oggetto, istanza di una sotto-classe, è **formalmente** compatibile con il tipo della classe base
  - Il contrario non è vero!
- Esempio
  - Un'automobile è un veicolo
  - Un veicolo non è (necessariamente) un'automobile
- La compatibilità diviene effettiva se
  - I metodi ridefiniti nella sotto-classe rispettano la semantica della superclasse
- L'ereditarietà gode delle proprietà transitiva
  - Un tandem è un veicolo (poiché è una bicicletta, che a sua volta è un veicolo)

# Vantaggi dell'ereditarietà

- Evitare la duplicazione di codice
- Permettere il riuso di funzionalità
- Semplificare la costruzione di nuove classi
- Facilitare la manutenzione
- Garantire la consistenza delle interfacce



# Ereditarietà in Java

- Si definisce una classe derivata attraverso la parola chiave "**extends**"
  - Seguita dal nome della classe base
- Gli oggetti della classe derivata sono, a tutti gli effetti, estensioni della classe base
  - Anche nella loro rappresentazione in memoria

# Ereditarietà in Java

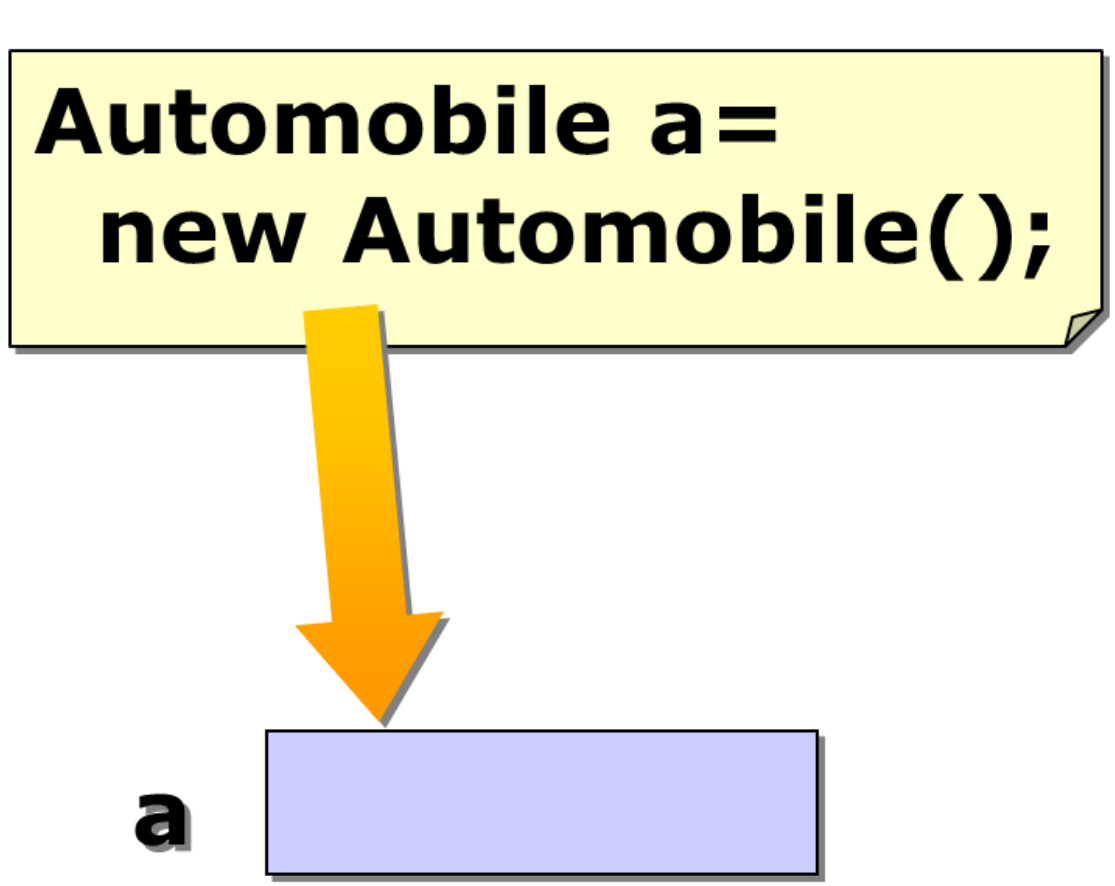
Veicolo.java

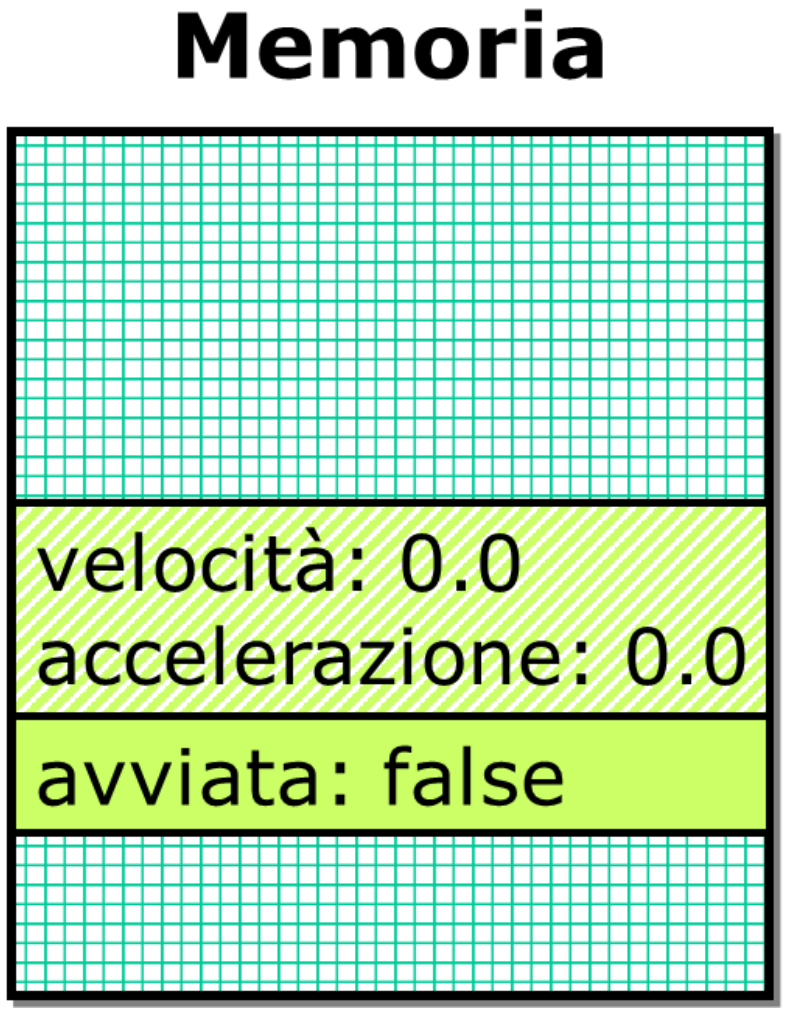
```
public class Veicolo {
    private double velocità;
    private double accelerazione;
    public double getVelocità() {…}
    public double getAccelerazione() {…}
}
```

Automobile.java

```
public class Automobile
   extends Veicolo {
       private boolean avviata;
       public void avvia() {…}
}
```

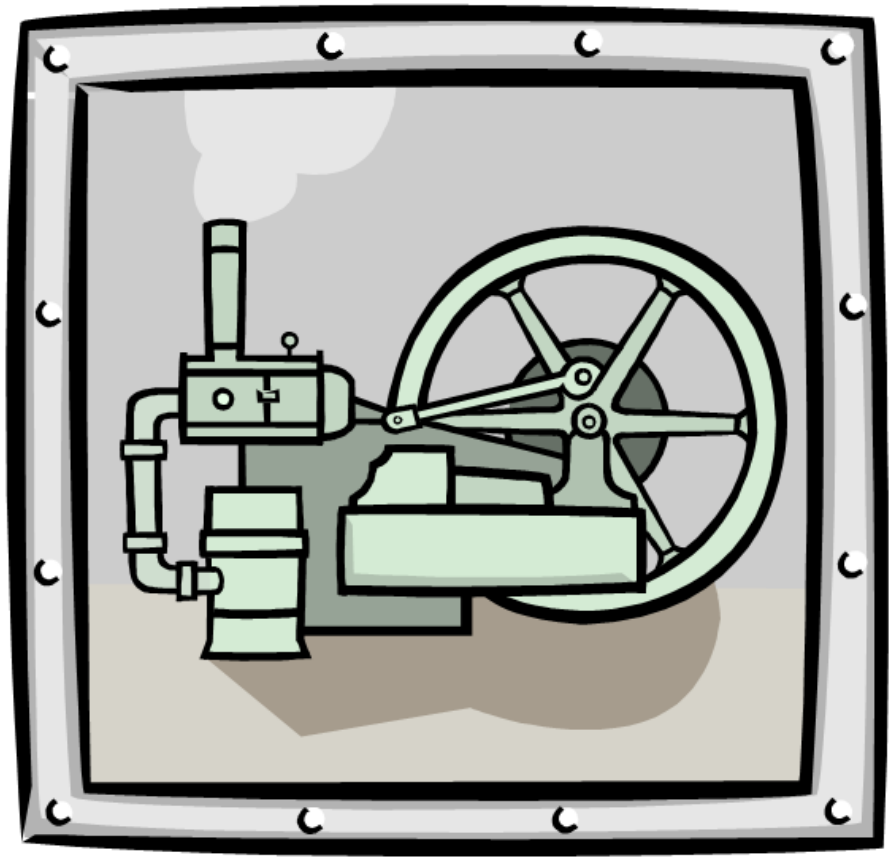
# Ereditarietà in Java



# Meccanismi

- Costruzione di oggetti di classi derivate
- Accesso alle funzionalità della superclasse
- Ri-definizione di metodi



# Costruzione

- Per realizzare un'istanza di una classe derivata, occorre – innanzi tutto – costruire l'oggetto base
  - Di solito, provvede automaticamente il compilatore, invocando – come prima operazione di ogni costruttore della classe derivata – il **costruttore anonimo** della superclasse
  - Si può effettuare in modo esplicito, attraverso il costrutto ***super(…)***
  - Eventuali ulteriori inizializzazioni possono essere effettuate solo successivamente

# Esempio

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

  Impiegato(String n) {
     nome = n;
     stipendio= 1500;
  }
}
```

```
class Funzionario
  extends Impiegato {

    Funzionario(String n) {
     super(n);
     stipendio = 2000;
  }
}
```

# Accedere alla superclasse

- L'oggetto derivato contiene **tutti i componenti** (attributi e metodi) dell'oggetto da cui deriva
  - Ma i suoi metodi **non possono** operare direttamente su quelli definiti **privati**
- La restrizione può essere allentata:
  - La super-classe può definire attributi e metodi con visibilità "**protected**"
  - Questi sono visibili alle sottoclassi

# Ridefinire i metodi

- Una sottoclasse può ridefinire metodi presenti nella superclasse
- A condizione che abbiano
  - Lo stesso **nome**
  - Gli stessi **parametri** (tipo, numero, ordine)
  - Lo stesso **tipo di ritorno**
  - (La stessa **semantica**!)
- Per le istanze della sottoclasse, il nuovo metodo **nasconde** l'originale

# Ridefinire i metodi

```
class Base {
    int m() {
        return 0;
    }
}
```

```
class Derivata
  extends Base {
    int m() {
        return 1;
    }
}
```

```
Base b= new Base();
System.out.println(b.m());
Derivata d= new Derivata();
System.out.println(d.m());
```

# Ridefinire i metodi

- A volte, una sottoclasse vuole “perfezionare” un metodo ereditato, non sostituirlo *in toto*
  - Per invocare l’implementazione presente nella super-classe, si usa il costrutto **super.<*nomeMetodo*> ( … )**

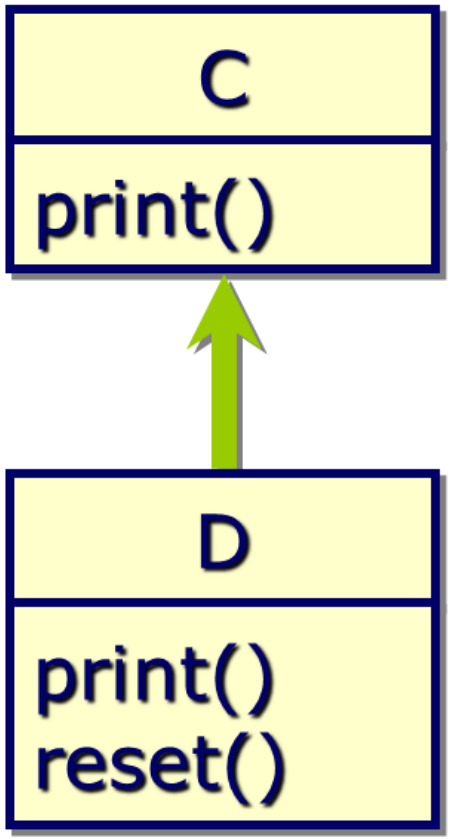
```
class Base {
      int m() {
         return 0;
     }
}
```

```
class Derivata
      extends Base {
      int m() {
          return super.m()+ 1;
     }
}
```

# Compatibilità formale

- Un'istanza di una classe derivata è formalmente compatibile con il tipo della super-classe
  - Base b = new Derivata( );
- Il tipo della variabile "b" (Base) limita le operazioni che possono essere eseguite sull'oggetto contenuto
  - Anche se questo ha una classe più specifica (Derivata), in grado di offrire un maggior numero di operazioni
  - Altrimenti viene generato un errore di compilazione

# Compatibilità formale




```
C v1= new C();
C v2= new D();
D v3= new D();
```

v1.print() ☑

v2.print() ☑

v2.reset() ☒

v3.reset() ☑

# Polimorfismo

```
class Base {
     int m() {
         return 0;
     }
}
```

```
class Derivata
  extends Base {
     int m() {
         return 1;
     }
}
```

```
Base b= new Derivata();
System.out.println(b.m());
```

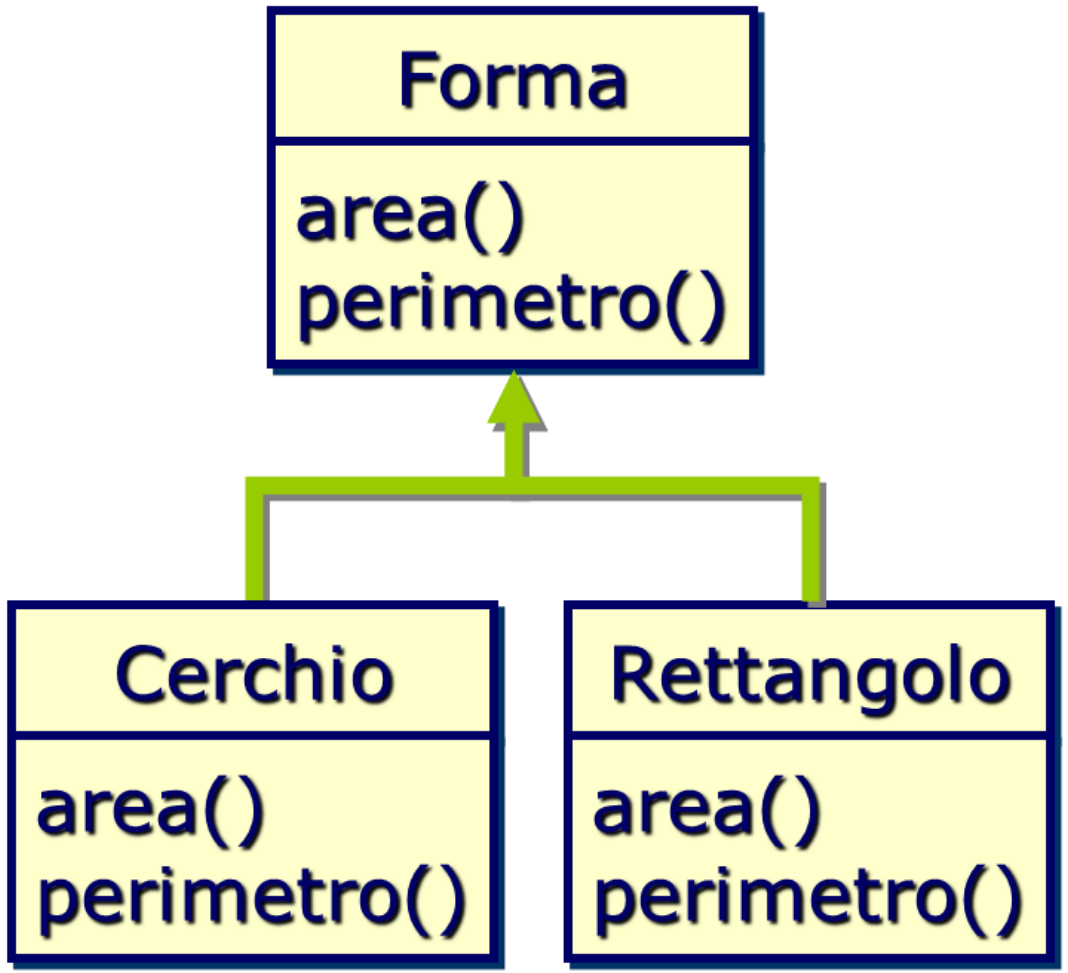
# Polimorfismo

- Java mantiene traccia della classe effettiva di un dato oggetto
  - Seleziona sempre il metodo più specifico…
  - …anche se la variabile che lo contiene appartiene ad una classe più generica!
- Una variabile generica può avere "**molte forme**"
  - Contenere oggetti di **sottoclassi differenti**
  - In caso di ridefinizione, il metodo chiamato dipende dal **tipo effettivo** dell'oggetto

# Polimorfismo

- Per sfruttare questa tecnica:
  - Si definiscono, nella super-classe, metodi con implementazione generica…
  - …sostituiti, nelle sottoclassi, da implementazioni specifiche
  - Si utilizzano variabili aventi come tipo quello della super-classe
- Meccanismo estremamente potente e versatile, alla base di molti “pattern” di programmazione

# Esempio




```
Forma f1 =
    new Cerchio();
Forma f2 = new
    Rettangolo();
double d1,d2;


d1=f1.area();
d2=f2.area();
```

# Overloading

- Consente di "**sovraccaricare**" un costruttore o un metodo di una classe con diverse varianti, in base ai parametri passati.

- Dal punto di vista dell'utilizzo della classe ciò consente di definire in maniera **dinamica** il costruttore o metodo che meglio si adatta.

# Overloading di Costruttore/Metodo

```
public class Prodotto
{
    private int id;
    // ...
    public Prodotto(int id, String desc)
    {
        // ...
    }
    public Prodotto(int id, String desc1, String desc2)
    {
        // ...
    }
    public Prodotto(int id, String desc1, String desc2, String desc3)
    {
        // ...
    }
}
```

# La classe java.lang.Object

- In Java:
  - Gerarchia di ereditarietà semplice
  - Ogni classe ha una sola super-classe
- Se non viene definita esplicitamente una super-classe, il compilatore usa la classe predefinita **Object**
  - Object non ha super-classe!

# Metodi di Object

- Object definisce un certo numero di **metodi pubblici**
  - Qualunque oggetto di qualsiasi classe li eredita
  - La loro implementazione base è spesso minimale
  - La tecnica del polimorfismo permette di ridefinirli
- public boolean **equals**(Object o)
  - Restituisce “vero” se l’oggetto confrontato è identico (ha lo stesso riferimento) a quello su cui viene invocato il metodo
  - Per funzionare correttamente, ogni sottoclasse deve fornire la propria implementazione polimorfica

# Metodi di Object

- public String **toString**()
  - Restituisce una rappresentazione stampabile dell'oggetto
  - L'implementazione base fornita indica il nome della classe seguita da un numero derivato dal riferimento all'oggetto (java.lang.Object@10878cd)
- public int **hashCode**()
  - Restituisce un valore intero legato al contenuto dell'oggetto
  - Se i dati nell'oggetto cambiano, deve restituire un valore differente
  - Oggetti "uguali" **devono** restituire lo stesso valore, oggetti diversi **possono** restituire valori diversi
  - Utilizzato per realizzare tabelle hash

# Controllare l'ereditarietà

- In alcuni casi, si vuole impedire esplicitamente l'utilizzo della tecnica del polimorfismo
  - Ad esempio, per motivi di sicurezza o per garantire il mantenimento di una data proprietà del sistema
  - Si utilizza la parola chiave "**final**"
- Un metodo "final" non può essere ridefinito da una sottoclasse
- Una classe "final" non può avere sottoclassi
- Un attributo "final" non può essere modificato
  - Non c'entra nulla con l'ereditarietà!

# Controllare l'ereditarietà

- In altri casi si vuole obbligare l'utilizzo del polimorfismo
  - Si introducono metodi privi di implementazione
  - Facendoli precedere dalla parola chiave "**abstract**"
- Se una classe contiene metodi astratti:
  - Deve essere, a sua volta, dichiarata abstract
  - Non può essere istanziata direttamente
  - Occorre definire una sottoclasse che fornisca l'implementazione dei metodi mancanti

# Classi astratte

```
abstract class
Base {
   abstract int m();
}
```

```
class Derivata
  extends Base {
     int m() {
        return 1;
     }
}
```

```
Base b= new Derivata();
System.out.println(b.m());
```