

Metodologie di Programmazione (M-Z)

Il semestre - a.a. 2022 – 2023

Parte 5 – Ereditarietà**

a cura di Stefano Faralli*



SAPIENZA
UNIVERSITÀ DI ROMA

*Tutti i diritti relativi al presente materiale didattico ed al suo contenuto sono riservati a Sapienza e ai suoi autori (o docenti che lo hanno prodotto). È consentito l'uso personale dello stesso da parte dello studente a fini di studio. Ne è vietata nel modo più assoluto la diffusione, duplicazione, cessione, trasmissione, distribuzione a terzi o al pubblico pena le sanzioni applicabili per legge.

**I crediti sulle slide di questo corso sono riportati nell'ultima slide

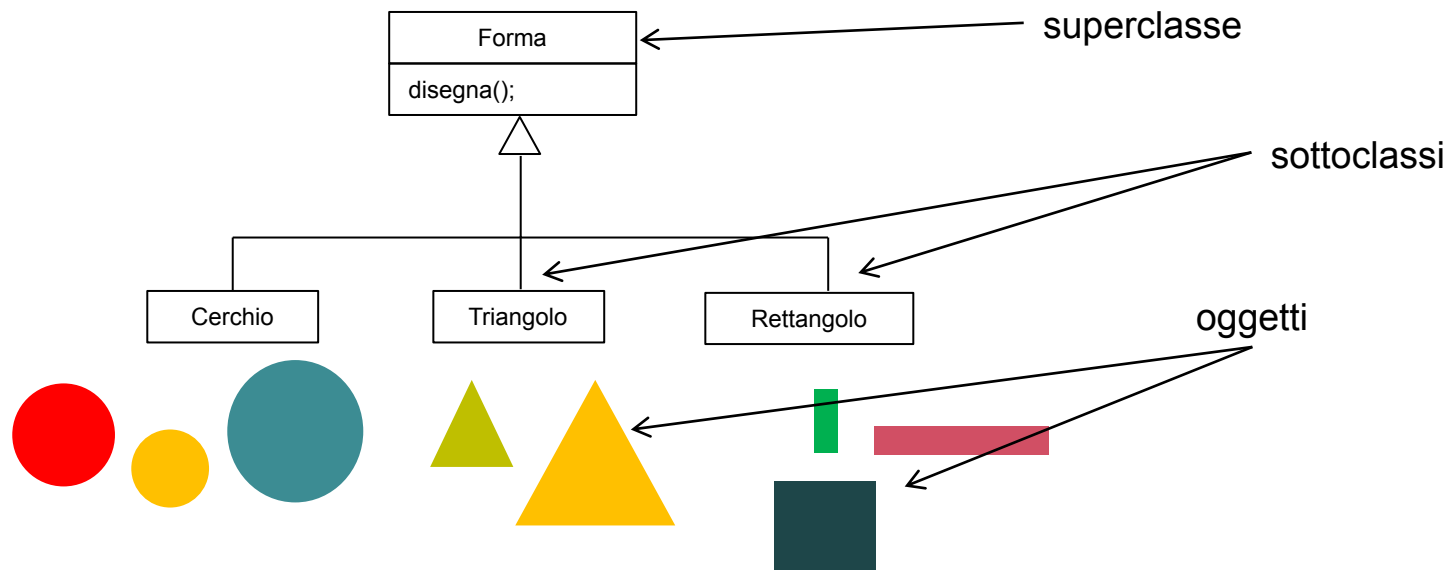
Ereditarietà

Ereditarietà

- Un **concetto cardine** della programmazione orientata agli oggetti
- Una **forma di riuso del software** in cui una classe è creata:
 - “assorbendo” i **membri di una classe esistente**
 - aggiungendo **nuove caratteristiche** o **migliorando quelle esistenti**
- **Programmazione mattone su mattone**
 - Detto anche: **non si butta via niente**
- Aumenta le probabilità che il sistema sia **implementato e mantenuto in maniera efficiente**

Molti tipi di “forma”

- Si può **progettare** una classe **Forma** che rappresenta una forma generica e poi **specializzarla estendendo** la classe



Molti tipi di “forma”

Estende la classe Forma

```
public class Forma
{
    public void disegna() { }
}
```

```
public class Triangolo extends Forma
{
    private double base;
    private double altezza;

    public Triangolo(double base, double altezza)
    {
        this.base = base;
        this.altezza = altezza;
    }

    public double getBase()
    {
        return base;
    }

    public double getAltezza()
    {
        return altezza;
    }
}
```

```
public class Cerchio extends Forma
{
    /**
     * Raggio del cerchio
     */
    private double raggio;

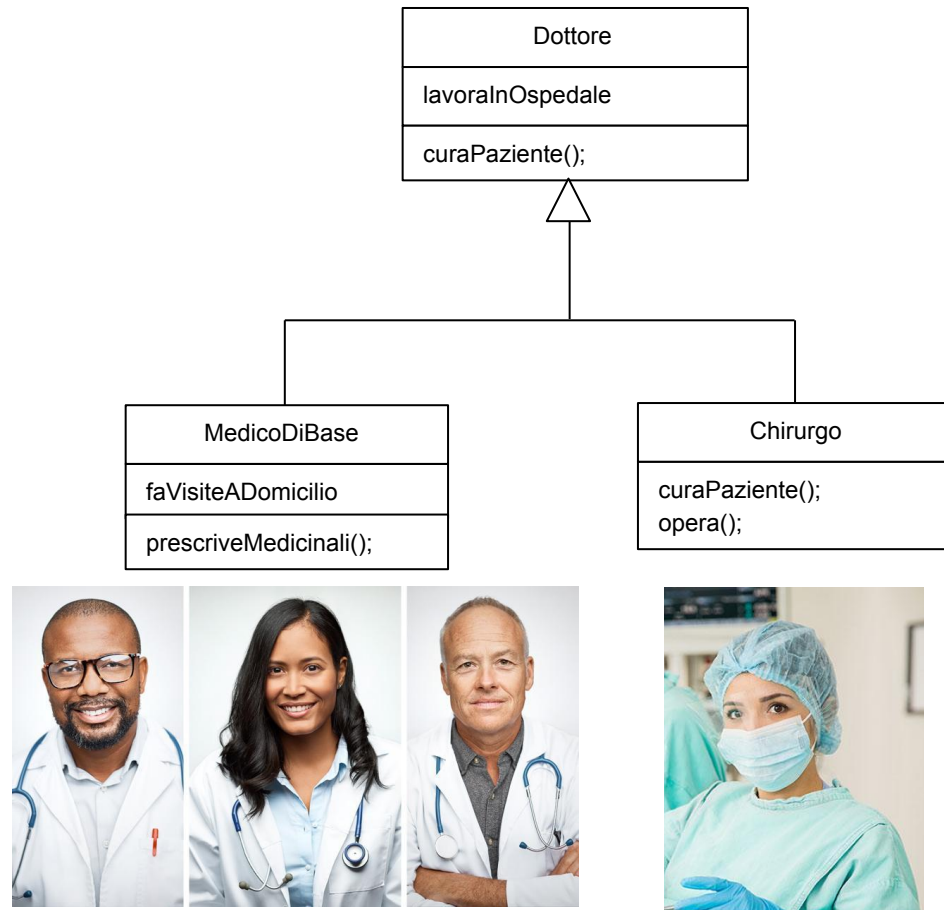
    public Cerchio(int raggio)
    {
        this.raggio = raggio;
    }

    public double getRaggio() { return raggio; }
    public double getCirconferenza() { return 2*Math.PI*raggio; }
}
```

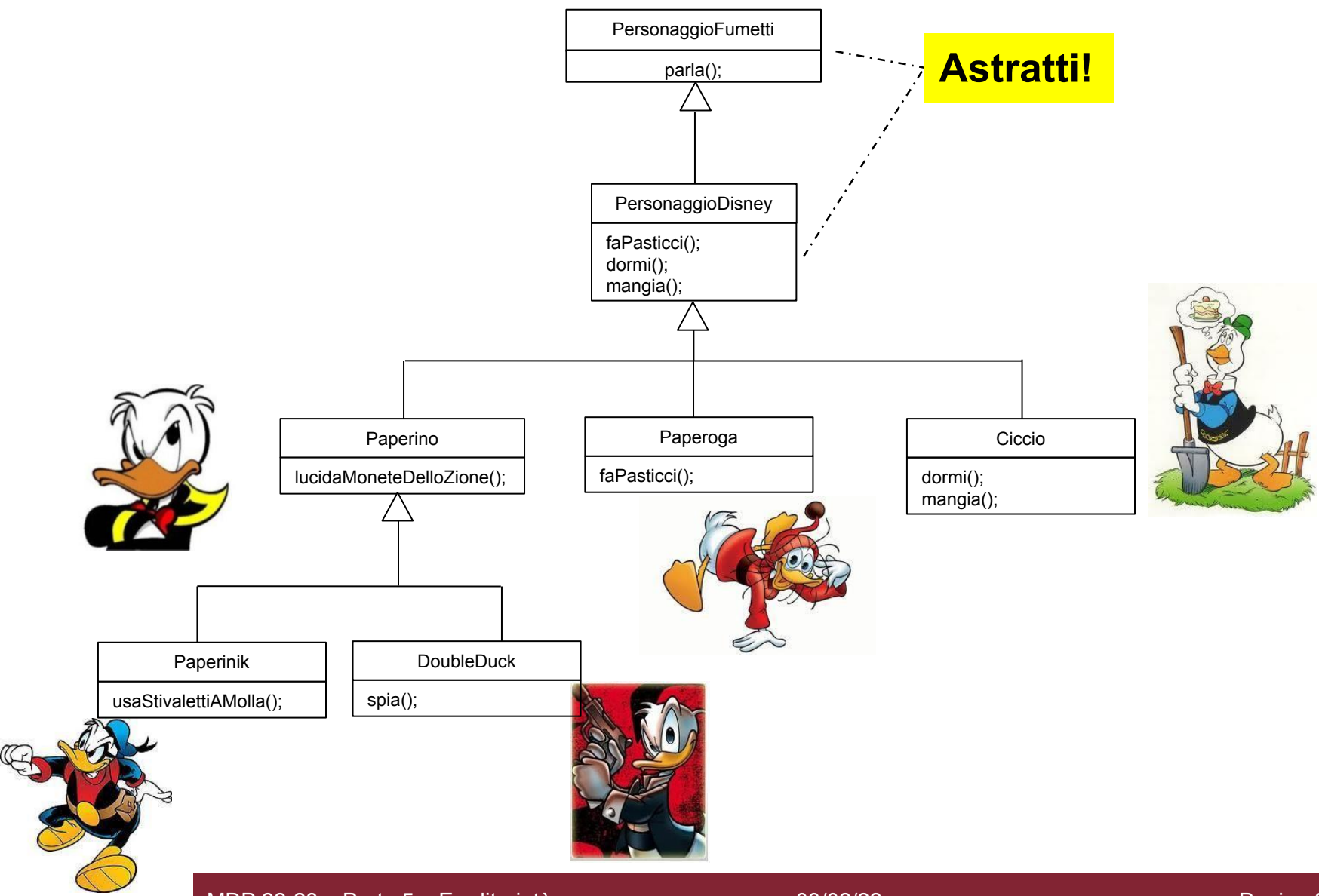
Ereditarietà: che cosa si eredita?

- Una **sottoclasse** estende la **superclasse**
- La **sottoclasse** eredita i **membri** della **superclasse**
 - **campi** e **metodi d'istanza** secondo il livello di accesso specificato
- Inoltre la **sottoclasse** può:
 - aggiungere **nuovi metodi** e **campi**
 - **ridefinire** i metodi che eredita dalla superclasse (tipicamente **NON** i campi)

Esempio: il dottore, il medico di base e il chirurgo



Esempio: paperino & company



Classi astratte

- Una classe **astratta** (definita mediante la parola chiave **abstract**) non può essere istanziata
 - Quindi **NON** possono esistere **oggetti** per quella classe

```
/**
 * Classe astratta: non e' possibile istanziarla
 */
public abstract class PersonaggioDisney
{
    /**
     * Metodo astratto senza implementazione
     */
    abstract void faPasticci();
}
```

Classi astratte

- Una classe **astratta** (definita mediante la parola chiave **abstract**) non può essere istanziata
 - Quindi **NON** possono esistere **oggetti** per quella classe

```
/**
 * Classe astratta: non e' possibile istanziarla
 */
public abstract class PersonaggioDisney
{
    /**
     * Metodo astratto senza implementazione
     */
    abstract void faPasticci();
}
```

- Tipicamente verrà **estesa** da **altre classi**, che invece potranno essere istanziate

```
public class Paperoga extends PersonaggioDisney
{
    public void faPasticci()
    {
        System.out.println("bla bla bla bla bla bla bla");
    }
}
```

Metodi astratti

- Anche i metodi possono essere definiti **astratti**
 - Esclusivamente all'interno di una **classe dichiarata astratta**
- **NON** forniscono l'implementazione per quel metodo

```
/**
 * Classe astratta: non e' possibile istanziarla
 */
public abstract class PersonaggioDisney
{
    /**
     * Metodo astratto senza implementazione
     */
    abstract void faPasticci();
}
```

Metodi astratti

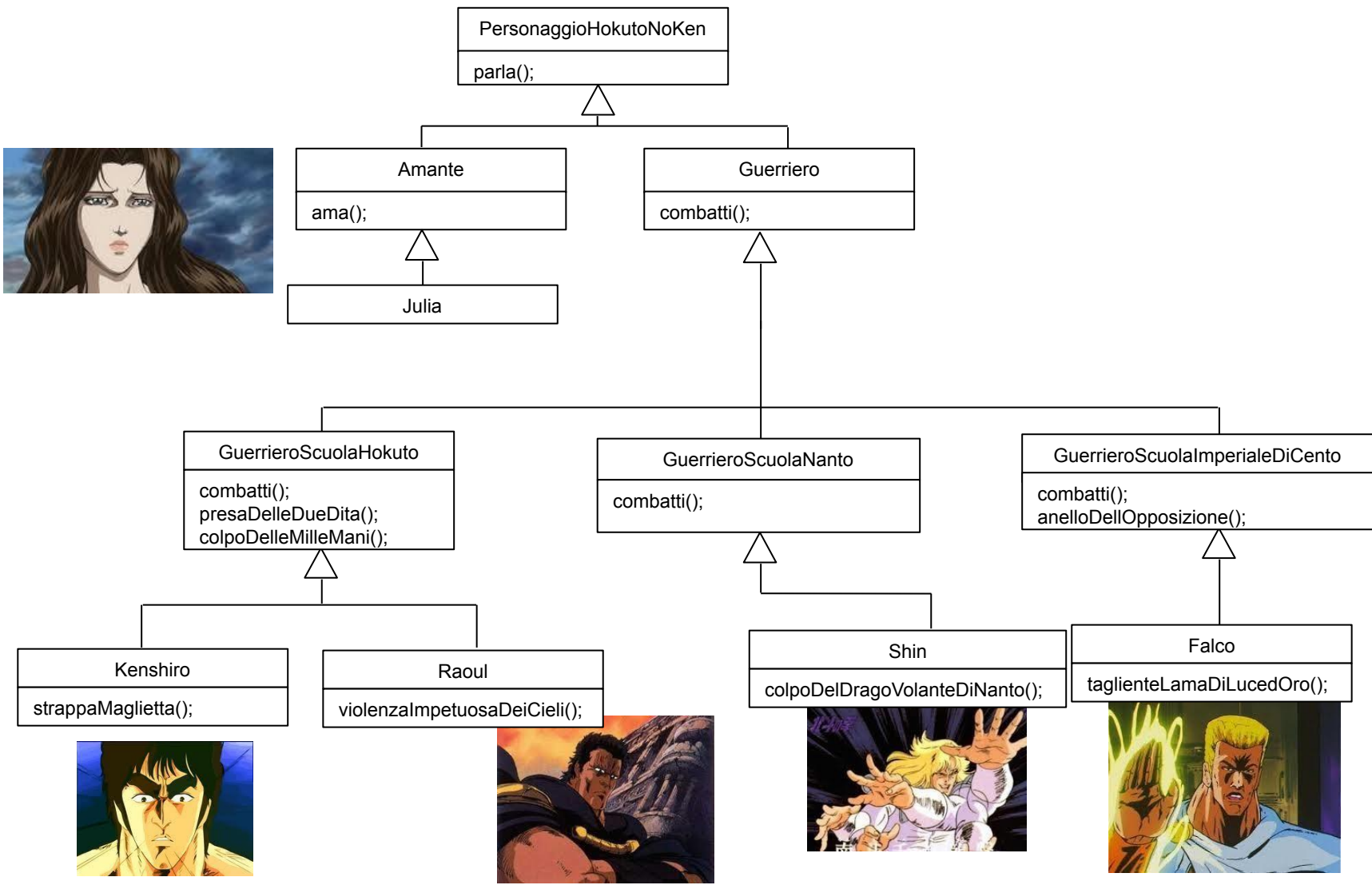
- Anche i metodi possono essere definiti **astratti**
 - Esclusivamente all'interno di una **classe dichiarata astratta**
- **NON** forniscono l'implementazione per quel metodo

```
/**
 * Classe astratta: non e' possibile istanziarla
 */
public abstract class PersonaggioDisney
{
    /**
     * Metodo astratto senza implementazione
     */
    abstract void faPasticci();
}
```

- Impongono alle sottoclassi **non astratte** di implementare il metodo

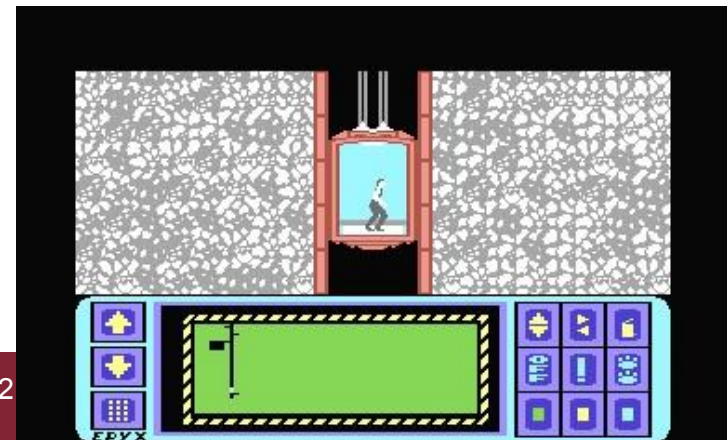
```
public class Paperoga extends PersonaggioDisney
{
    public void faPasticci()
    {
        System.out.println("bla bla bla bla bla bla");
    }
}
```

Esempio: Hokuto No Ken



Esempio: Impossible Mission

- Abbiamo tanti “oggetti” (in senso lato)
 - Piattaforme
 - Computer
 - Oggetti in cui cercare indizi
- Alcuni sono “personaggi”
 - Il giocatore
 - I robot
 - Il “bombone”
- Che cosa hanno in comune tutti?
- E che cosa li distingue?



Modellare Impossible Mission in un quarto d'ora: la classe “astratta” Entità

- Abbiamo bisogno di una classe **molto generale** (quindi **astratta**) che rappresenti oggetti mobili e immobili nel gioco:

```
package it.uniroma1.impmis;  
  
abstract public class Entita  
{  
    protected int x;  
    protected int y;  
  
    public Entita(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
}
```

Campi a **visibilità protetta**

- La visibilità protetta (**protected**) rende visibile il campo (o il metodo) a tutte le sottoclassi
 - Ma anche a **tutte le classi del package (!)**

Modellare Impossible Mission in un quarto d'ora: gli oggetti

- Modelliamo gli **oggetti immobili** in astratto:

```
package it.uniroma1.impmis;  
  
abstract public class Oggetto extends Entita  
{  
    private TesseraPuzzle tessera;  
    public Oggetto(int x, int y)  
    {  
        this(x, y, null);  
    }  
    public Oggetto(int x, int y, TesseraPuzzle tessera)  
    {  
        super(x, y);  
        this.tessera = tessera;  
    }  
    public TesseraPuzzle search() { return tessera; }  
}
```

Doppio costruttore (**overloading**)

Riuso del codice chiamando l'altro costruttore mediante la parola chiave

Richiama il costruttore della superclasse (**OBBLIGATORIO** perché ha almeno un parametro) con la parola chiave

Metodo aggiuntivo

Modellare Impossible Mission in un quarto d'ora: gli oggetti

- che possono contenere una tessera del puzzle del gioco:

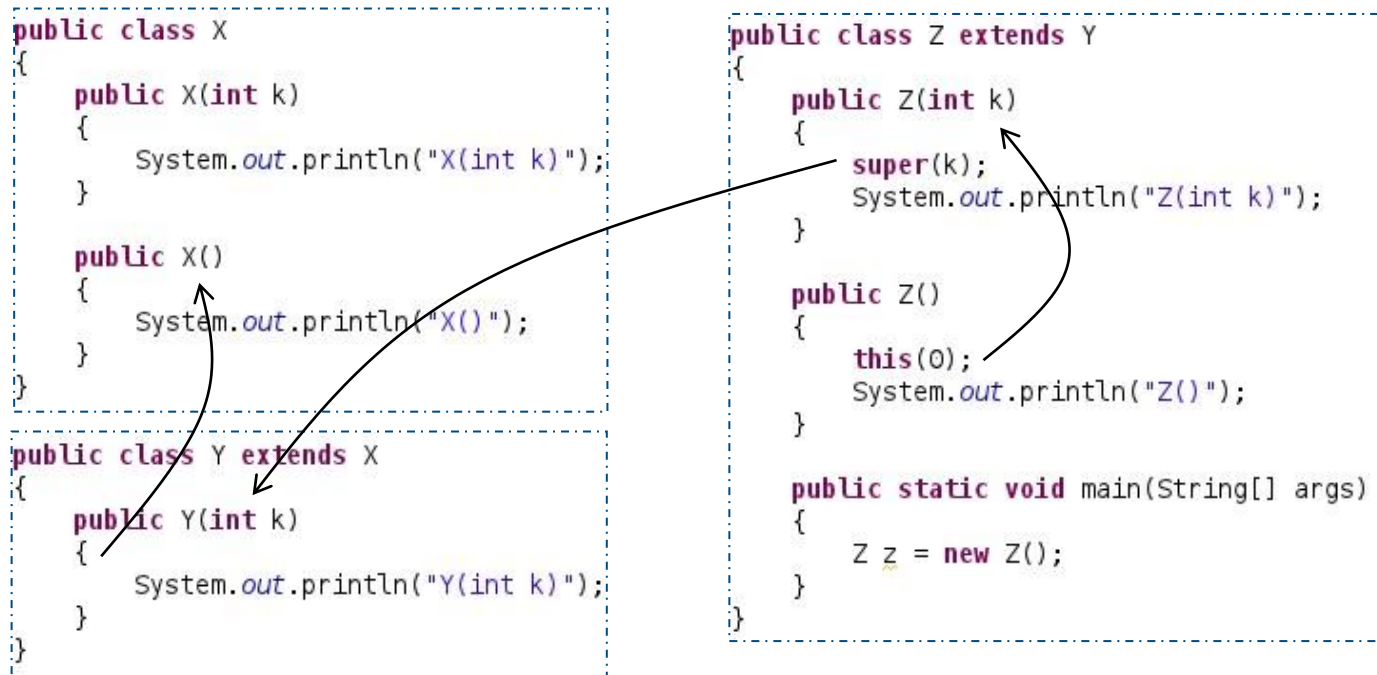
```
package it.uniroma1.impmis;  
  
public class TesseraPuzzle  
{  
    // da implementare  
}
```

this e super

- La parola chiave **this** usata come nome di metodo **obbligatoriamente** nella **prima riga del costruttore** permette di richiamare un altro costruttore della stessa classe
- La parola chiave **super** usata come nome di metodo **obbligatoriamente** nella prima riga del costruttore permette di richiamare un **costruttore della superclasse**
- Ogni sottoclasse deve **esplicitamente definire un** costruttore se la superclasse **NON fornisce un costruttore senza argomenti**
 - Questo avviene se si definisce un costruttore **con almeno un argomento** ma non si definisce il costruttore senza argomenti
- **Nota bene:** anche le classi astratte possono avere costruttori! **Perché?**

Esempio di dinamica delle chiamate a costruttori

- Definiamo una piccola gerarchia di tre classi X, Y, Z:



```
public class X
{
    public X(int k)
    {
        System.out.println("X(int k)");
    }

    public X()
    {
        System.out.println("X()");
    }
}

public class Y extends X
{
    public Y(int k)
    {
        System.out.println("Y(int k)");
    }
}

public class Z extends Y
{
    public Z(int k)
    {
        super(k);
        System.out.println("Z(int k)");
    }

    public Z()
    {
        this(0);
        System.out.println("Z()");
    }

    public static void main(String[] args)
    {
        Z z = new Z();
    }
}
```

Disegnate il diagramma delle classi in UML

Esempio di dinamica delle chiamate a costruttori

- Definiamo una piccola gerarchia di tre classi X, Y, Z:

```
public class X
{
    public X(int k)
    {
        System.out.println("X(int k)");
    }

    public X()
    {
        System.out.println("X()");
    }
}

public class Y extends X
{
    public Y(int k)
    {
        System.out.println("Y(int k)");
    }
}

public class Z extends Y
{
    public Z(int k)
    {
        super(k);
        System.out.println("Z(int k)");
    }

    public Z()
    {
        this(0);
        System.out.println("Z()");
    }

    public static void main(String[] args)
    {
        Z z = new Z();
    }
}
```

Disegnate il diagramma delle classi in UML

Esempio di dinamica delle chiamate a costruttori

- Definiamo una piccola gerarchia di tre classi X, Y, Z:

```
public class X
{
    public X(int k)
    {
        System.out.println("X(int k)");
    }

    public X()
    {
        System.out.println("X()");
    }
}

public class Y extends X
{
    public Y(int k)
    {
        System.out.println("Y(int k)");
    }
}

public class Z extends Y
{
    public Z(int k)
    {
        super(k);
        System.out.println("Z(int k)");
    }

    public Z()
    {
        this(0);
        System.out.println("Z()");
    }

    public static void main(String[] args)
    {
        Z z = new Z();
    }
}
```

Nell'eseguire l'istruzione `new Z();` abbiamo in output:

```
X()
Y(int k)
Z(int k)
Z()
```

Esempio di dinamica delle chiamate a costruttori

```
public class X  
{  
}
```

```
public class Y extends X  
{  
}
```

Costruttori di default in X e Y



```
public class X  
{  
    public X()  
    {  
    }  
}
```

```
public class Y extends X  
{  
}
```

Costruttore esplicito con zero argomenti per X e di default per Y (quello di Y chiama automaticamente quello di X)



```
public class X  
{  
}
```

```
public class Y extends X  
{  
    public Y(int k)  
    {  
    }  
}
```

Costruttore di default in X e con un argomento in Y (chiama in automatico il costruttore di default di X)



Esempio di dinamica delle chiamate a costruttori

```
public class X
{
    public X(int k)
    {
    }
}
```

```
public class Y extends X
{
}
```

Costruttore con un parametro per X e di default per Y



```
public class X
{
    public X()
    {
    }
    public X(int k)
    {
    }
}
```

```
public class Y extends X
{
}
```

Costruttori con zero e un parametro per X e di default per Y (quello di Y chiama automaticamente quello di X)



Modellare Impossible Mission in un quarto d'ora:

un esempio di oggetto e il computer

- Modelliamo un possibile oggetto immobile:

```
package it.uniroma1.impmiss;

public class Libreria extends Oggetto
{
    public Libreria(int x, int y, TesseraPuzzle tessera)
    {
        super(x, y, tessera);
    }

    public Libreria(int x, int y)
    {
        super(x, y);
    }
}
```



```
package it.uniroma1.impmiss;

abstract public class Entita
{
    protected int x;
    protected int y;

    public Entita(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
}
```

- e il computer:



```
package it.uniroma1.impmiss;

public class Computer extends Entita
{
    public Computer(int x, int y)
    {
        super(x, y);
    }

    public void login() { /* da implementare */ }
    public void logout() { /* da implementare */ }
}
```

Metodi aggiuntivi

```
package it.uniroma1.impmiss;

abstract public class Oggetto extends Entita
{
    private TesseraPuzzle tessera;

    public Oggetto(int x, int y)
    {
        this(x, y, null);
    }

    public Oggetto(int x, int y, TesseraPuzzle tessera)
    {
        super(x, y);
        this.tessera = tessera;
    }

    public TesseraPuzzle search() { return tessera; }
}
```


Modellare Impossible Mission in un quarto d'ora:

entità mobili:

- Modelliamo un generico personaggio:

```
package it.uniroma1.impmis;  
  
abstract public class Personaggio extends Entita  
{  
    public enum Direzione  
    {  
        DESTRA,  
        SINISTRA,  
        ALTO,  
        BASSO;  
    }  
  
    private String nome;  
    private int velocita;  
  
    public Personaggio(int x, int y, String nome, int velocita)  
    {  
        super(x, y);  
        this.nome = nome;  
        this.velocita = velocita;  
    }  
  
    public String getNome() { return nome; }  
    public int getVelocita() { return velocita; }  
  
    public void muoviti(Direzione d)  
    {  
        switch(d)  
        {  
            case DESTRA: x += velocita; break;  
            case SINISTRA: x -= velocita; break;  
            // in futuro: emetti eccezione!  
            default: System.out.println("Direzione non ammessa"); break;  
        }  
    }  
}
```

Enumerazione
delle direzioni

Campi aggiuntivi

Si possono fornire
implementazioni
nella classe astratta!

Metodi aggiuntivi

Modellare Impossible Mission in un quarto d'ora: il giocatore

- Modelliamo il **giocatore** (ovvero la spia):

```
package it.uniroma1.impmiss;  
  
public class Spia extends Personaggio  
{  
    public Spia(int x, int y, String nome, int velocita)  
    {  
        super(x, y, nome, velocita);  
    }  
  
    public void salta()  
    {  
        // ...  
    }  
}
```



nemico e robot

- Modelliamo un **generico nemico**:

Modellare Impossible Mission in un quarto d'ora:

nemico e robot

- Modelliamo un generico nemico:

```
package it.uniroma1.impmis;  
  
abstract public class Nemico extends Personaggio  
{  
    public Nemico(int x, int y, String nome, int velocita)  
    {  
        super(x, y, nome, velocita);  
    }  
  
    abstract public void attacca();  
}
```

Metodo astratto!

- E i robot:



```
package it.uniroma1.impmis;  
  
public class Robot extends Nemico  
{  
    public Robot(int x, int y, String nome, int velocita)  
    {  
        super(x, y, nome, velocita);  
    }  
  
    public void incenerisci() { /* fulmine elettrico */ }  
  
    public void attacca() { /* da implementare */ }  
}
```

Nella sottoclasse siamo **obbligati** a **definire** il metodo astratto

Modellare Impossible Mission in un quarto d'ora:

Il «bombone»

- Modelliamo un bombone:



```
package it.uniroma1.impmis;  
  
public class Bombone extends Nemico  
{  
    public Bombone(int x, int y, String nome, int velocita)  
    {  
        super(x, y, nome, velocita);  
    }  
  
    public void attacca()  
    {  
        // da implementare  
    }  
  
    public void muoviti(Direzione d)  
    {  
        switch(d)  
        {  
            case DESTRA: case SINISTRA: super.muoviti(d); break;  
            case ALTO: y -= getVelocita(); break;  
            case BASSO: y += getVelocita(); break;  
        }  
    }  
  
    public void muoviti(Spia p)  
    {  
        muoviti(p.x > this.x ? Direzione.DESTRA : Direzione.SINISTRA);  
        muoviti(p.y > this.y ? Direzione.ALTO : Direzione.BASSO);  
    }  
}
```

Overriding del
metodo

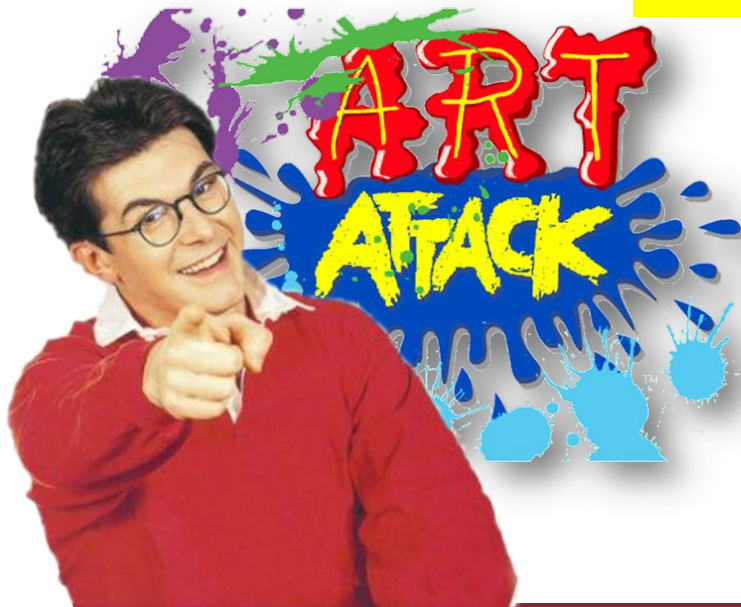
Overloading del
metodo

Esempi di riuso
del codice

Modellare Impossible Mission in un quarto d'ora:

Disegnate il diagramma delle classi risultante: Entita, Oggetto, TesseraPuzzle, Libreria, Computer, Personaggio, Spia, Nemico, Robot, Bombone

In UML una classe astratta ha il nome in corsivo e/o si usa il tag `<abstract>`



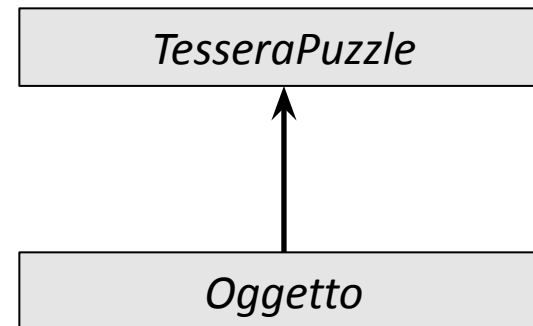
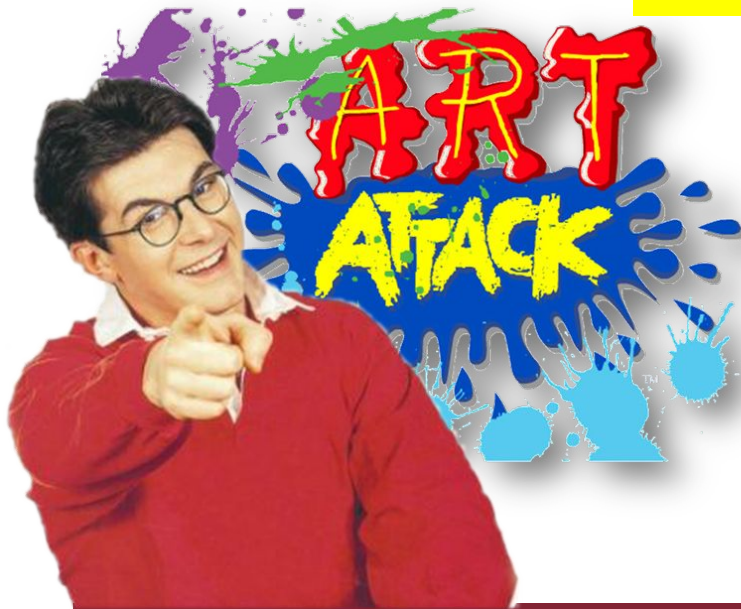
<i>NomeClasseAstratta</i>
campi
metodi

<code><abstract></code> <i>NomeClasseAstratta</i>
campi
metodi

Modellare Impossible Mission in un quarto d'ora:

Disegnate il diagramma delle classi risultante: Entita, Oggetto, TesseraPuzzle, Libreria, Computer, Personaggio, Spia, Nemico, Robot, Bombone

In UML una **dipendenza** semplice si indica con una freccia



Differenza tra Overriding e Overloading

- L'**overriding** consiste nel **ridefinire** (reimplementare) un metodo con la stessa **intestazione** (“segnatura”) presente in una superclasse
- L'**overloading** consiste nel creare un metodo con lo stesso nome, ma una **intestazione** diversa (diverso numero e/o tipo di parametri)

Mantenere il contratto

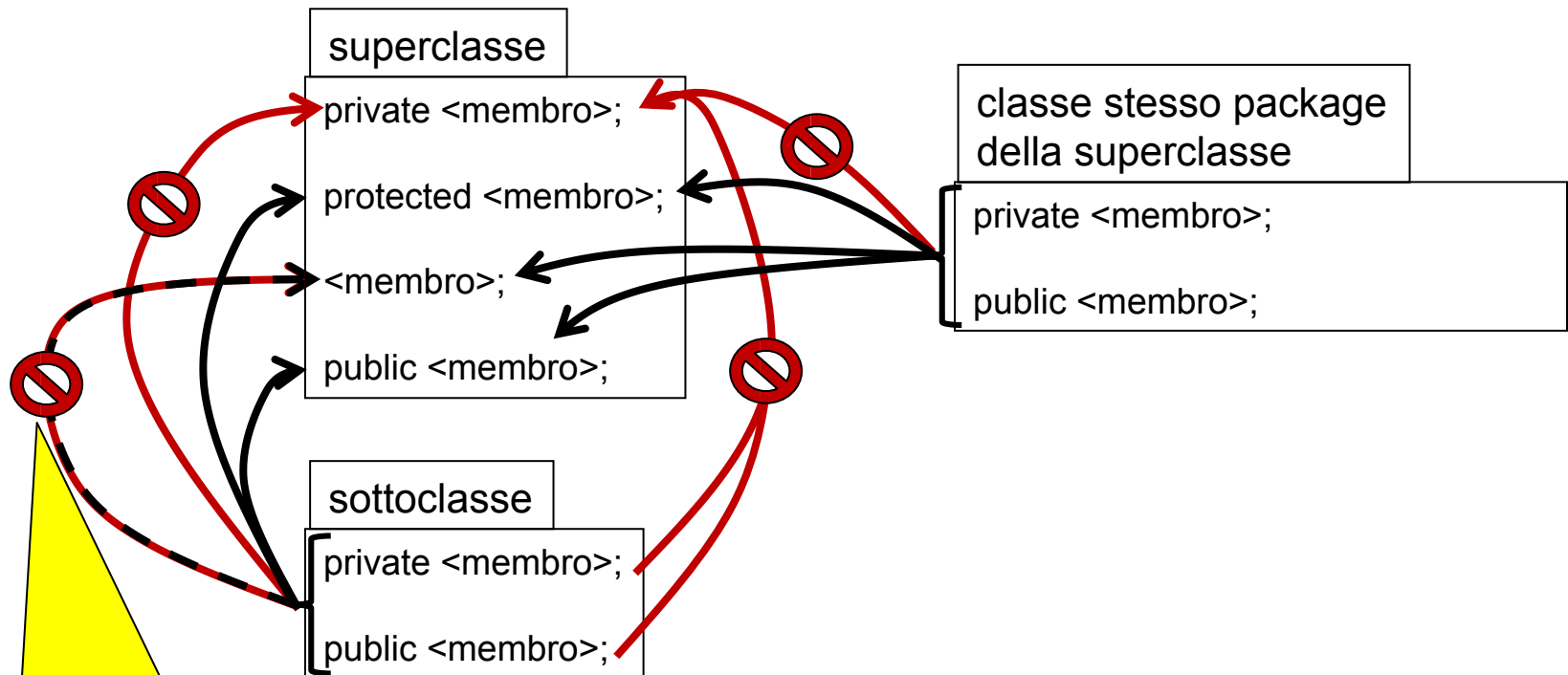
- Nell'**overriding** gli argomenti devono essere gli stessi
- I tipi di ritorno devono essere compatibili (lo stesso tipo o una sottoclasse)
- Non si può **ridurre la visibilità** (es. da public a private)
- Nell'**overloading** i tipi di ritorno possono essere diversi
 - **MA**: non si può cambiare **SOLO** il tipo di ritorno
- Si può **variare la visibilità** in qualsiasi direzione

Visibilità

Quattro possibilità per campi e metodi:

- **Private**: visibile solo all'interno della classe
- **Public**: visibile a tutti [all'interno di un modulo, vedi Java ≥ 9]
- **Default**: visibile all'interno di tutte le classi del package
- **Protected**: visibile all'interno di tutte le classi del package e delle sottoclassi (indipendentemente dal package)

Visibilità



Accessibile se la sottoclasse
è nello stesso package

Is-a contro has-a



- E' **molto** importante distinguere tra relazioni di tipo

è-un

(is-a) e relazioni di tipo **ha-un** (has-a)

- **Is-a** rappresenta l'**ereditarietà**

- Un oggetto di una sottoclasse può essere trattato come un oggetto della superclasse

- **Domanda:** la sottoclasse è-un superclasse? (es. Paperino è un PersonaggioDisney? Sì! QuiQuoQua è un Paperino? No!)

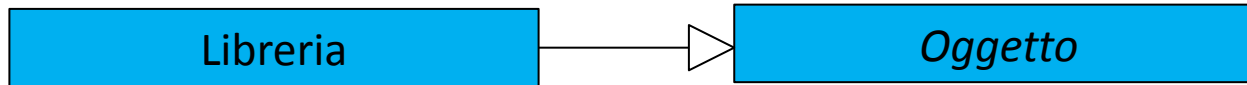
- **Has-a** rappresenta la **composizione**

- Un oggetto contiene come membri **riferimenti** ad altri oggetti

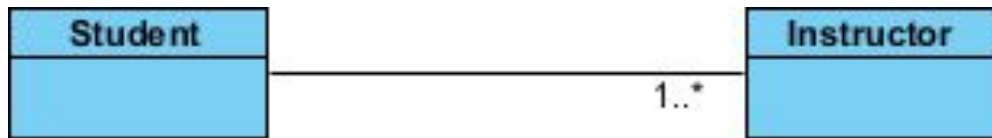
- **Domanda:** un oggetto contiene altri oggetti? (es. Bagno contiene Vasca? Sì! PersonaggioDisney contiene Paperino? No!)

In UML

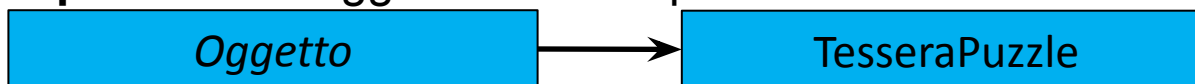
Is-A: implica una relazione gerarchica di ereditarietà



Associazione: implica una relazione generica che associa x oggetti di una classe a y oggetti di un'altra classe

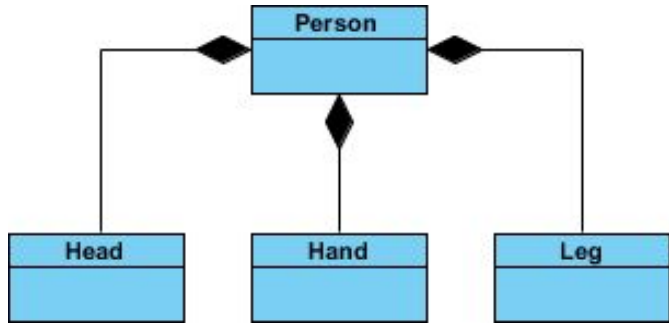


Dipendenza: Oggetto fa uso/dipende di/a TesseraPuzzle

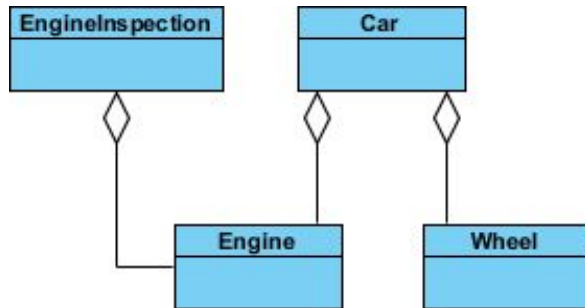


In UML

Composizione: implica una relazione dove un figlio non può esistere indipendentemente dal padre (ad es. Casa (padre) e Stanza (figlio))



Aggregazione: implica una relazione dove un figlio può esistere indipendentemente dal padre.



Esercizio: BarraDiEnergia & BarraDiEnergiaConPercentuale

- Creare una classe **BarraDiEnergia** costruita con un intero che ne determina la lunghezza massima. Inizialmente la barra è vuota. La barra è dotata di un metodo per l'incremento unitario del suo livello di riempimento e di un metodo **toString** che ne fornisca la rappresentazione sotto forma di stringa (es. se il livello è 3 su 10, la stringa sarà "OOO=====").
- Creare quindi una seconda classe **BarraDiEnergiaConPercentuale** che fornisce una rappresentazione sotto forma di stringa come BarraDiEnergia ma stampando in coda alla stringa la percentuale del livello di riempimento. Per esempio, se il livello è 3 su 10, la stringa sarà "OOO===== 30%".

Esercizio: ListaDiInteri e ListaOrdinataDiInteri

Implementare una classe **ListaDiInteri** mediante un array (con i metodi specificati in fondo alle diapositive della terza parte: “controllo e array”)

- Implementare quindi una classe **ListaOrdinataDiInteri** per creare liste di interi ordinati in modo crescente. La classe ridefinisce i seguenti 3 metodi di aggiunta:
 - **aggiungiInCoda(k)**: Aggiungi un intero in coda alla lista: l'aggiunta avviene solo se l'intero preserva l'ordine della lista.
 - **aggiungi(k, j)**: Aggiungi un intero nella posizione specificata: come sopra, l'aggiunta avviene solo se l'intero preserva l'ordine degli interi della lista
 - **aggiungi(k)**: Aggiungi un intero: l'intero viene inserito nella posizione appropriata, in modo da preservare l'ordine degli interi della lista
- L'array non deve essere ordinato con metodi di sorting, quali Arrays.sort (né vostri metodi di sorting *completo* dell'array)
- **Extra:** permettere di specificare un parametro da passare opzionalmente al costruttore di **ListaOrdinataDiInteri** per stabilire l'ordine della lista (crescente o decrescente; per default, crescente)

Esercizio: Animali

- Progettare (diagramma delle classi) ed implementare la classe **Animale** che rappresenti un generico animale
- La classe possiede i metodi **emettiVerso()** e **getNumeroDiZampe()**
- Possiede inoltre il metodo **getTaglia()** che restituisce un valore scelto tra: piccola, media e grande.
- Progettare (diagramma delle classi) ed implementare quindi le classi **Mammifero**, **Felino**, **Gatto** (taglia piccola), **Tigre** (grande), **Cane**, **Chihuahua** (piccola), **Beagle** (media), **Terranova** (grande), **Uccello**, **Corvo** (media), **Passero** (piccola), **Millepiedi** (piccola)
- Personalizzare in modo appropriato la taglia, il numero di zampe e il verso degli animali

Esercizio: Conto bancario

- Progettare la classe **ContoBancario** che rappresenti un conto con informazioni relative al denaro attualmente disponibile, il codice IBAN
- Modellare quindi una generica operazione bancaria **Operazione** che disponga di un metodo **esegui()**
- Modellare quindi i seguenti tipi di operazione:
 - **PrelevaDenaro**: preleva una specificata quantità di denaro da un dato conto
 - **SvuotaConto**: preleva tutto il denaro da un dato conto
 - **VersaDenaro**: versa del denaro sul conto specificato
 - **SituazioneConto**: stampa l'attuale saldo del conto
 - **Bonifico**: preleva del denaro da un conto e lo versa su un altro
- Specificare un metodo nella classe **ContoBancario** che restituisca l'elenco delle operazioni svolte in ordine temporale

Esercizio: Distributore automatico

- Progettare una classe **Prodotto** con un prezzo e tre tipi diversi di prodotto: **BottigliaDAcqua**, **BarraDiCioccolato**, **GommeDaMasticare**
- Progettare la classe **DistributoreAutomatico** che rappresenti un distributore automatico costruito con un intero N che determina il numero di prodotti nel distributore
- La classe prevede i seguenti metodi:
 - un metodo **carica()** che inserisce N prodotti di tipo e ordine casuale
 - un metodo **inserisciImporto()** che permette di inserire un importo nella macchinetta
 - un metodo **getProdotto()** che, dato in ingresso un numero di prodotto, restituisca il prodotto associato a quel numero e decrementi il saldo disponibile nel distributore
 - Un metodo **getSaldo()** che restituisca il saldo attuale del distributore
 - un metodo **getResto()** che restituisca il resto dovuto e azzeri il saldo

Esercizio: Espressioni matematiche

- Progettare una serie di classi che modellino le espressioni matematiche secondo la seguente definizione:
 - Una **costante** di tipo double è un'espressione
 - Una **variabile** con nome di tipo stringa e valore double è un'espressione
 - Se e_1 è un'espressione, allora $-e_1$ è un'espressione
 - Se e_1, e_2 sono espressioni, allora $e_1 \text{ op } e_2$ è un'espressione dove op può essere l'operatore $+, -, *, /, \%$
- Ogni tipo di espressione (costante, variabile, espressioni composte) deve essere modellata mediante una classe separata
- Ogni espressione dispone del metodo `getValore()` che restituisce il valore che quell'espressione possiede in quel momento
- Costruire quindi l'espressione $-(5+(3/2)-2)*x$ e calcolarne il valore quando la variabile x vale 3 e quando la variabile x vale 6
- **Suggerimenti:** la variabile può modificare il proprio valore nel tempo; servirà veramente salvare un valore nella superclasse?
- **Alternative:** progettarlo usando l'ereditarietà (meglio) e mediante enum per le espressioni binarie

Esercizio: il Gioco dell'Oca

- Progettare il Gioco dell'Oca modellando:
 - Il **Giocatore**, che mantiene l'informazione sulla posizione nel tabellone e i punti accumulati e implementa il metodo **tiraDadi()**
 - Il **Tabellone** come sequenza di caselle costruita a partire da un intero N e da un elenco di giocatori; la classe dispone dell'operazione di posizionamento dei giocatori tenendo conto dell'effetto "gioco dell'oca" in cui, arrivati alla fine, si torna indietro
 - Diversi **tipi di caselle** ciascuna con un diverso effetto a seguito del posizionamento del giocatore su quella casella:
 - una **CasellaVuota** (nessun effetto sul giocatore)
 - una **CasellaSpostaGiocatore** che sposta il giocatore di x caselle (avanti se $x > 0$ o indietro se $x < 0$)
 - una **CasellaPunti** che ha l'effetto di far guadagnare o perdere un certo numero di punti al giocatore
 - la classe **GiocoDellOca** che, dato un intero N e dati i giocatori, che inizializza un tabellone di lunghezza N e implementa il metodo **giocaUnTurno()** che fa effettuare una mossa a ognuno dei giocatori

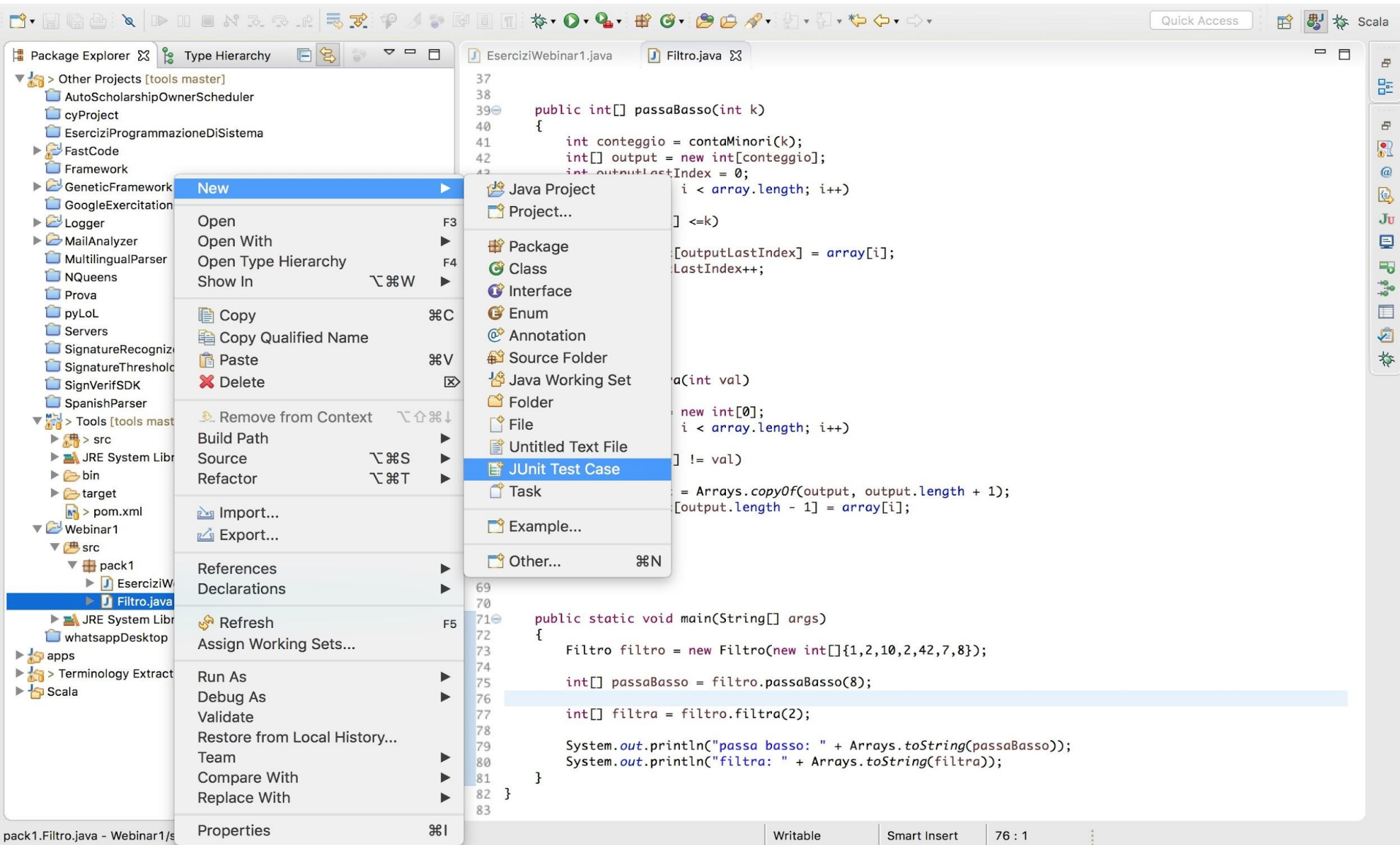
Esercizio: Tetris

- Progettare il gioco del Tetris modellando la classe **Pezzo** con le seguenti operazioni:
 - **Left** : sposta a sinistra il pezzo
 - **Right**: sposta a destra il pezzo
 - **Rotate**: ruota il pezzo in senso orario
 - **Down**: manda giù il pezzo
- Progettare anche la classe di ciascun pezzo (a forma di L, a forma di T, a serpente, a forma di I e cubo)
- Progettare infine la classe **Tetris** che somministra i pezzi, permette al giocatore di muoverli, gestisce lo spazio libero e calcola i punteggi del giocatore

Junit

Junit - Framework per fare unit test in Java.

- Utile per testare singole parti del software (ad esempio un metodo o una classe).
- Integrato in Eclipse.
- Utilizza le annotazioni:
 - **@Test** per i metodi che definiscono un test.
 - **@Before** per i metodi che devono essere eseguiti prima di ogni unit test (utile per l'inizializzazione delle variabili)
 - **@After** per i metodi che devono essere eseguiti dopo ogni unit test (utile per eliminare file creati durante il test)
 - **@BeforeClass @AfterClass** per i metodi che devono essere chiamati una sola volta prima e dopo tutti i test.



New JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder: Webinar1/src

Package: pack1

Name: TestFiltro

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test: pack1.Filtro

```

73     Filtro filtro = new Filtro(new int[] {1,2,10,2,42,7,8});
74
75     int[] passaBasso = filtro.passaBasso(8);
76
77     int[] filtra = filtro.filtra(2);
78
79     System.out.println("passa basso: " + Arrays.toString(passaBasso));
80     System.out.println("filtra: " + Arrays.toString(filtra));
81 }
82 }
83

```

New JUnit Test Case

Test Methods
Select methods for which test method stubs should be created.

Available methods:

- ☐ Filtro(int[])
- ☒ passaBasso2(int)
- ☒ passaBasso(int)
- ☒ filtra(int)
- ☐ main(String[])
- ☐ Object

3 methods selected.

☐ Create final method stubs

☐ Create tasks for generated test methods

Finish

```

73 Filtro filtro = new Filtro(new int[]{1,2,10,2,42,7,8});
74
75 int[] passaBasso = filtro.passaBasso(8);
76
77 int[] filtra = filtro.filtra(2);
78
79 System.out.println("passa basso: " + Arrays.toString(passaBasso));
80 System.out.println("filtra: " + Arrays.toString(filtra));
81 }
82 }
83

```

The screenshot shows an IDE with the Package Explorer on the left, displaying a project structure with folders like 'AutoScholarshipOwnerScheduler', 'cyProject', 'EserciziProgrammazioneDiSistema', 'FastCode', 'Framework', 'GeneticFramework', 'GoogleExercitation', 'Logger', 'MailAnalyzer', 'MultilingualParser', 'NQueens', 'Prova', 'pyLoL', 'Servers', 'SignatureRecognizer', 'SignatureThresholdLearner', 'SignVerifSDK', 'SpanishParser', 'Tools', 'Webinar1', and 'whatsappDesktop'. The 'Tools' folder is expanded, showing 'src', 'JRE System Library [JavaSE-1.8]', 'bin', 'target', 'pom.xml', and 'Webinar1'. The 'Webinar1' folder is expanded, showing 'src', 'pack1', 'EserciziWebinar1.java', and 'Filtro.java'. The 'Filtro.java' file is selected, and its content is visible in the editor.

The 'New JUnit Test Case' wizard is open, showing the 'Test Methods' tab. The wizard prompts the user to select methods for which test method stubs should be created. The 'Available methods' list is empty. The wizard also prompts the user to add JUnit 4 to the build path, with the option 'Perform the following action: Add JUnit 4 library to the build path' selected. The wizard also prompts the user to create final method stubs and create tasks for generated test methods. The wizard has buttons for '< Back', 'Next >', 'Cancel', and 'Finish'.

The code editor shows the following code:

```

73 Filtro filtro = new Filtro(new int[] {1,2,10,2,42,7,8});
74
75 int[] passaBasso = filtro.passaBasso(8);
76
77 int[] filtra = filtro.filtra(2);
78
79 System.out.println("passa basso: " + Arrays.toString(passaBasso));
80 System.out.println("filtra: " + Arrays.toString(filtra));
81 }
82 }
83

```

JUnit

```
package pack1;

import static org.junit.Assert.*;

import org.junit.Test;

public class TestFiltro
{
    @Test
    public void testPassaBasso2()
    {
        fail("Not yet implemented");
    }

    @Test
    public void testPassaBasso()
    {
        fail("Not yet implemented");
    }

    @Test
    public void testFiltro()
    {
        fail("Not yet implemented");
    }
}
```

Junit

Ogni test deve testare una singola unità della classe che stiamo controllando!!!

```
@Test
public void testPassaBasso()
{
    int k = 8;
    int[] testArray = new int[]{2, 10, 8, 4, 99, 5, 42};
    Filtro filtro = new Filtro(testArray);
    int[] outArray = filtro.passaBasso(k);
    assertEquals(outArray.length, 4);
    assertEquals(new int[]{2, 8, 4, 5}, outArray);
}
```


Junit

- Nei test vanno utilizzate le **assertion** per verificare che una certa condizione sia vera (o falsa).

method name / parameters	description
<code>assertTrue(<i>test</i>)</code> <code>assertTrue("message", <i>test</i>)</code>	Causes this test method to fail if the given boolean test is not <code>true</code> .
<code>assertFalse(<i>test</i>)</code> <code>assertFalse("message", <i>test</i>)</code>	Causes this test method to fail if the given boolean test is not <code>false</code> .
<code>assertEquals(<i>expectedValue</i>, <i>value</i>)</code> <code>assertEquals("message", <i>expectedValue</i>, <i>value</i>)</code>	Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the <code>equals</code> method to compare them.) The first of the two values is considered to be the result that you expect; the second is the actual result produced by the class under test.
<code>assertNotEquals(<i>value1</i>, <i>value2</i>)</code> <code>assertNotEquals("message", <i>value1</i>, <i>value2</i>)</code>	Causes this test method to fail if the given two values <i>are</i> equal to each other. (For objects, it uses the <code>equals</code> method to compare them.)
<code>assertNull(<i>value</i>)</code> <code>assertNull("message", <i>value</i>)</code>	Causes this test method to fail if the given value is not <code>null</code> .
<code>assertNotNull(<i>value</i>)</code> <code>assertNotNull("message", <i>value</i>)</code>	Causes this test method to fail if the given value <i>is</i> <code>null</code> .
<code>assertSame(<i>expectedValue</i>, <i>value</i>)</code> <code>assertSame("message", <i>expectedValue</i>, <i>value</i>)</code> <code>assertNotSame(<i>value1</i>, <i>value2</i>)</code> <code>assertNotSame("message", <i>value1</i>, <i>value2</i>)</code>	Identical to <code>assertEquals</code> and <code>assertNotEquals</code> respectively, except that for objects, it uses the <code>==</code> operator rather than the <code>equals</code> method to compare them. (The difference is that two objects that have the same state might be <code>equals</code> to each other, but not <code>==</code> to each other. An object is only <code>==</code> to itself.)
<code>fail()</code> <code>fail("message")</code>	Causes this test method to fail.

The screenshot displays an IDE interface with the following components:

- Package Explorer (Left):** Shows a project structure with 'Other Projects' and 'Tools'. Under 'Tools', there is a 'src' directory containing 'JRE System Library [JavaSE-1.8]', 'bin', 'target', and 'pom.xml'. A sub-project 'Webinar1' is expanded, showing 'src' with 'pack1' containing 'EserciziWebinar1.java', 'Filtro.java', and 'TestFiltro.java'.
- Main Editor:** Displays the code for 'TestFiltro.java'. The code includes package declarations, imports for JUnit, and two test methods: 'testPassaBasso()' and 'testFiltro()'. The 'testFiltro()' method is currently selected.
- Context Menu:** A right-click context menu is open over the code. It lists various actions such as 'Undo Typing', 'Revert File', 'Save', 'Open Declaration', 'Open Type Hierarchy', 'Open Call Hierarchy', 'Show in Breadcrumb', 'Quick Outline', 'Quick Type Hierarchy', 'Open With', 'Show In', 'Cut', 'Copy', 'Copy Qualified Name', 'Paste', 'Quick Fix', 'Source', 'Refactor', 'Local History', 'References', 'Declarations', 'Add to Snippets...', 'Run As', 'Debug As', 'Validate', 'Create Snippet...', 'Team', 'Compare With', 'Replace With', 'Preferences...', and 'Remove from Context'. A sub-menu for '1 JUnit Test' is also visible, showing 'Run Configurations...'.

The screenshot displays an IDE interface with the following components:

- Package Explorer (Left):** Shows a project structure with folders like 'AutoScholarshipOwnerScheduler', 'cyProject', 'EserciziProgrammazioneDiSistema', 'FastCode', 'Framework', 'GeneticFramework', 'GoogleExercitation', 'Logger', 'MailAnalyzer', 'MultilingualParser', 'NQueens', 'Prova', 'pyLoL', 'Servers', 'SignatureRecognizer', 'SignatureThresholdLearner', 'SignVerifSDK', 'SpanishParser', 'Tools', 'Webinar1', and 'Scala'.
- Code Editor (Center):** Displays the source code for `TestFiltro.java`. The code includes imports for `org.junit.Assert` and `org.junit.Test`, and a test method `testPassaBasso()` that uses `assertEquals` and `assertArrayEquals` to verify the output of the `Filtro` class.
- JUnit Console (Bottom):** Shows the execution results of the JUnit tests. It indicates that the tests finished after 0,027 seconds, with 2/2 runs, 0 errors, and 0 failures. The test results are:
 - `pack1.TestFiltro [Runner: JUnit 4] (0,001 s)`
 - `testPassaBasso (0,000 s)`
 - `testFiltro (0,000 s)`

Junit – Fallimento di un test

- Assicurarsi che il test sia scritto correttamente (chi testa i test?)
- Esultare per aver trovato un possibile bug.
- Correggere l'errore e ripetere il test.

Credits

Le slide di questo corso sono il frutto di una personale rielaborazione delle slide del Prof. Navigli.

In aggiunta, le slide sono state revisionate dagli studenti borsisti della Facoltà di Ingegneria Informatica, Informatica e Statistica: Mario Marra e Paolo Straniero.