# Metodologie di Programmazione: Interfacce e classi interne

Roberto Navigli

Dipartimento di Informatica


Sapienza
Università di Roma

# Le interfacce

http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

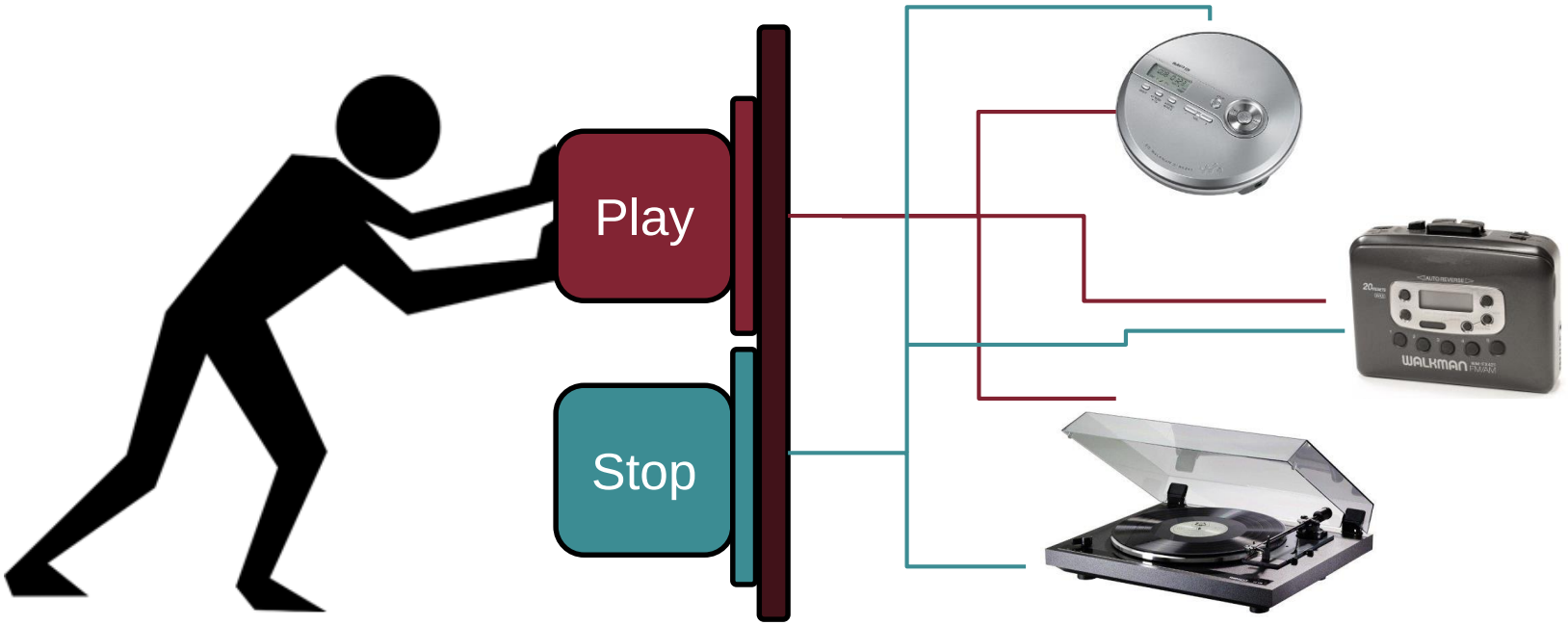
# Le interfacce in Java

- Le interfacce sono uno strumento che Java mette a disposizione per consentire a più classi di fornire e implementare un insieme di metodi comuni
- Le interfacce definiscono e standardizzano l'interazione fra oggetti tramite un insieme limitato di operazioni



# Le interfacce in Java

- Esse specificano soltanto il **comportamento** che un certo oggetto deve presentare all'esterno, cioè **cosa** quell'oggetto può fare
- L'**implementazione** di tali operazioni, cioè **come** queste vengono tradotte e realizzate, rimane invece non definito



# Le interfacce in uno slogan

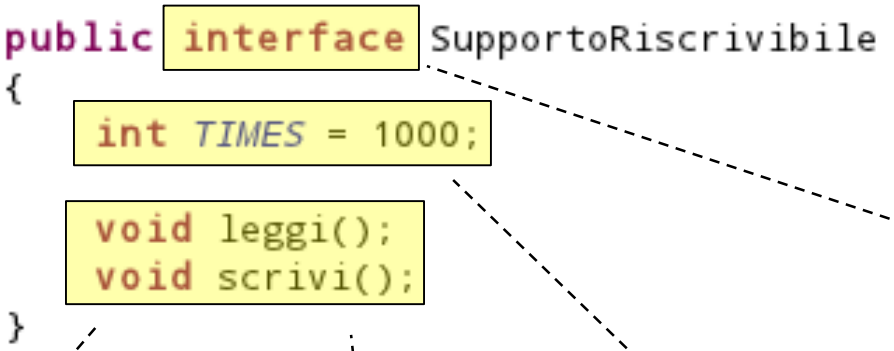
**Le interfacce sono classi astratte al 100%**


(se non definiamo metodi di default – da Java 8 in poi)

# Dichiarazione di un'interfaccia (1)

- Un'interfaccia è una classe che può contenere soltanto
  - Costanti
  - Metodi astratti
  - Java 8: Implementazione di **default** di metodi e metodi **statici** (utilità)
- Tutti i **metodi** dichiarati in un'interfaccia sono implicitamente **public abstract**
- Tutti i **campi** dichiarati in un'interfaccia sono implicitamente **public static final**
- Tranne nel caso dei metodi di default o statici, non è possibile specificare alcun dettaglio implementativo
  - non vi è alcun corpo di metodo o variabile di istanza

# Dichiarazione di un'interfaccia (2)




**Parola chiave interface:** indica la definizione di una nuova interfaccia

**Costante:** implicitamente final, public e static

**Dichiarazione di metodi:** anche se non esplicitato, sono public e abstract

**E' vietato** definire qualunque implementazione

# Implementare un'interfaccia (1)

```
public class Nastro implements SupportoRiscrivibile
{
    private Pellicola pellicola;

    @Override
    public void leggi()
    {
        attivaTestina();
        muoviTestina();
    }

    @Override
    public void scrivi()
    {
        attivaTestina();
        caricaTestina();
        muoviTestina();
        scaricaTestina();
    }

    public void attivaTestina() {}
    public void caricaTestina() {}
    public void scaricaTestina() {}
    public void muoviTestina() {}
```

- Per realizzare un'interfaccia è necessario che una classe la **implementi**, tramite la parola chiave **implements**
- Una classe che implementa una interfaccia decide di voler **esporre** pubblicamente all'esterno il comportamento descritto dall'interfaccia
- E' obbligatorio che ciascun metodo abbia esattamente la stessa intestazione che esso presenta nell'interfaccia

# Implementare un'interfaccia (2)

```
public class MemoriaUsb implements SupportoRiscrivibile
{
    private CellaMemoria[] celle;

    @Override
    public void leggi()
    {
        // Leggi la cella corretta
    }

    @Override
    public void scrivi()
    {
        // Modifica la cella corretta
    }
}
```

Anche la classe MemoriaUsb implementa l'interfaccia SupportoRiscrivibile, definendo i propri metodi leggi() e scrivi()

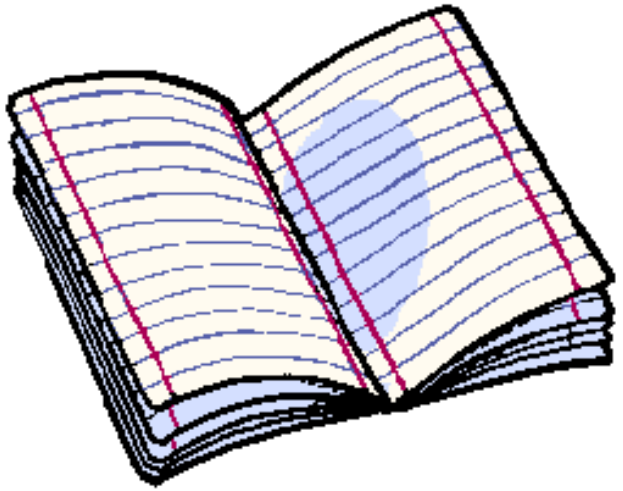
# Implementare un'interfaccia (3)

```
public class Quaderno implements SupportoRiscrivibile
{
    private Foglio[] fogli;

    @Override
    public void leggi()
    {
        // Leggi la pagina corrente
    }

    @Override
    public void scrivi()
    {
        // Scrivi sulla pagina corrente
    }
}
```

**Perfino** un quaderno può essere visto come un supporto che è possibile leggere e scrivere!



Le interfacce permettono di modellare comportamenti comuni a classi che **non sono necessariamente** in relazione gerarchica (**is-a**, **è-un**)

# Un esempio: l'interfaccia Iterabile (1)

- Ci sono molte classi di **natura diversa** che rappresentano sequenze di elementi
  - Es. array, liste, stringhe, ecc.
- Tuttavia, le sequenze hanno qualcosa in comune: è possibile iterare sui loro elementi:

```
public interface Iterabile
{
    boolean hasNext();
    Object next();
    void reset();
}
```

# Un esempio: l'interfaccia Iterabile (2)

- Ciascuna classe implementerà i metodi **a suo modo**:

```java
public class MyIntegerArray implements Iterabile
{
    private Integer[] array;
    private int k = 0;

    public MyIntegerArray(Integer[] array)
    {
        this.array = array;
    }

    @Override
    public boolean hasNext()
    {
        return k < array.length;
    }

    @Override
    public Object next()
    {
        return array[k++];
    }

    @Override
    public void reset() { k = 0; }
}
```

```java
public class MyString implements Iterabile
{
    private String s;
    private int k = 0;

    public MyString(String s)
    {
        this.s = s;
    }

    @Override
    public boolean hasNext()
    {
        return k < s.length();
    }

    @Override
    public Object next()
    {
        return s.charAt(k++);
    }

    @Override
    public void reset() { k = 0; }
}
```

# Un esempio: l'interfaccia Iterabile (2)

- Infine, testiamo le due classi:

```
public class InterfacceChePassione
{
    static public void main(String[] args)
    {
        Iterabile i1 = new MyIntegerArray(new Integer[] {10, 20, 30, 40});
        Iterabile i2 = new MyString("abcdefghi");

        for (Iterabile i : new Iterabile[] { i1, i2 })
        {
            while(i.hasNext())
                System.out.println(i.next());
        }
    }
}
```

Assegniamo un oggetto di una classe che implementa l'interfaccia a una variabile-riferimento a interfaccia

Possiamo creare un array di riferimenti a interfaccia!

Abbiamo ora un meccanismo generale per iterare su **sequenze** che implementano l'interfaccia

# L'esempio "Iterabile" non è la soluzione ideale per iterare su una collezione

- Perché?
- **Non permette** di mantenere contatori multipli sullo stesso oggetto
- Il seguente esempio **NON FUNZIONA**!

```
MyString s = new MyString("ciao");
while(s.hasNext())
{
        char c1 = s.next();
        while(s.hasNext())
        {
                char c2 = s.next();
                System.out.println(c1+"-"+c2);
        }
}
```

# La soluzione: usare le interfacce Iterable e Iterator

- Interfacce standard di Java
- java.lang.Iterable:

**Method Summary**

**Methods**

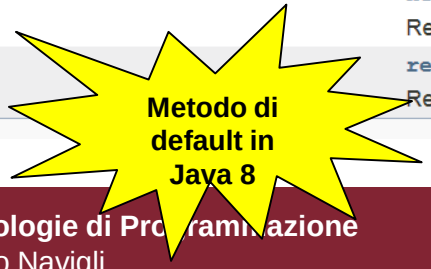
| Modifier and Type | Method and Description |
|---|---|
| `Iterator<T>` | **`iterator`**`()`<br>Returns an iterator over a set of elements of type T. |

- java.util.Iterator

**Method Summary**

**Methods**

| Modifier and Type | Method and Description |
|---|---|
| `boolean` | **`hasNext`**`()`<br>Returns `true` if the iteration has more elements. |
| `E` | **`next`**`()`<br>Returns the next element in the iteration. |
| `void` | **`remove`**`()`<br>Removes from the underlying collection the last element returned by this iterator (optional operation). |



# Implementare un'interfaccia: il contratto (1)

- Implementare un'interfaccia equivale a **firmare un contratto** con il compilatore che stabilisce l'impegno ad implementare tutti i metodi specificati dall'interfaccia o a dichiarare la classe abstract

# Implementare un'interfaccia: il contratto (2)

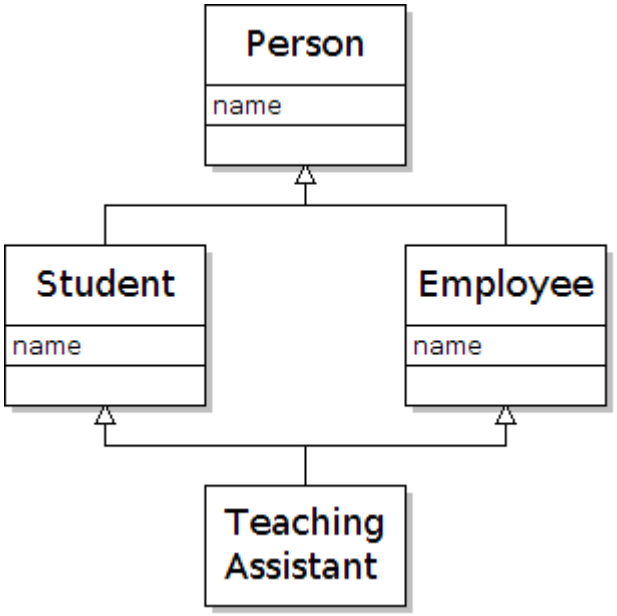
- 3 possibilità per una classe che implementa un'interfaccia:
  - fornire un'implementazione concreta di tutti i metodi, definendone il corpo
  - fornire l'implementazione concreta solo per un sottoinsieme proprio dei metodi dell'interfaccia
  - decidere di non fornire alcuna implementazione concreta
- Negli ultimi due casi, però, la classe va dichiarata abstract

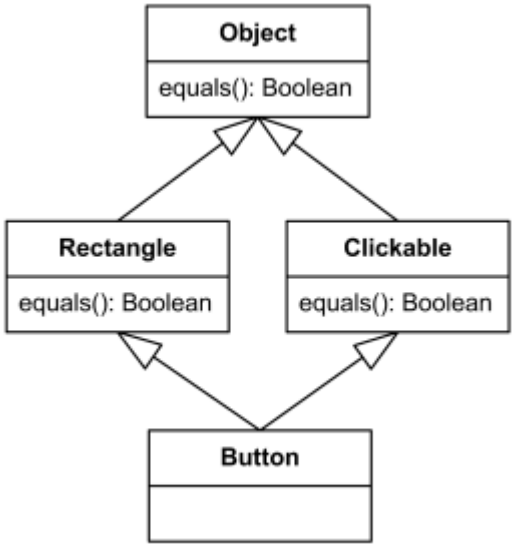
# Interfacce vs. classi astratte

- **Una domanda nasce spontanea:** Se implementando un'interfaccia **devo** dichiarare tutti i metodi in essa definiti, perché non ricorrere ad una classe astratta?

# Il problema del diamante

- Con l'ereditarietà multipla si possono creare situazioni **poco chiare** di duplicazione di metodi e campi

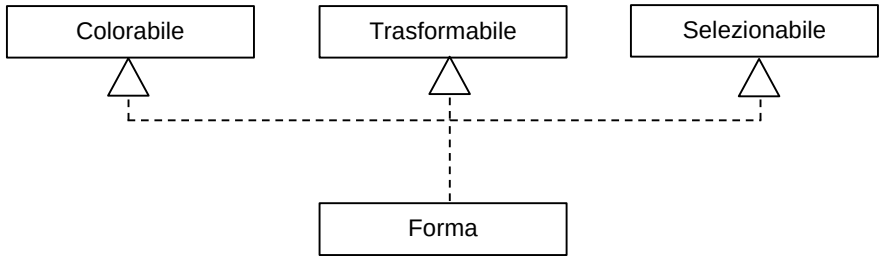


# Un altro esempio di problema del diamante


# Ereditarietà multipla (1)

- In Java non è consentito estendere più di una classe alla volta: ovvero extends può essere seguito **solo da un unico nome di classe**
- Al contrario, una classe può implementare tutte le interfacce desiderate




- E' inoltre possibile estendere **1** classe e contemporaneamente implementare **n** interfacce diverse

# Ereditarietà multipla (2)

```
public class Forma implements Colorabile, Trasformabile, Selezionabile
{

    @Override
    public void seleziona()
    {
    }

    @Override
    public void deseleziona()
    {
    }

    @Override
    public void sposta(Point point)
    {
    }

    @Override
    public void ruota(Angle angle)
    {
    }

    @Override
    public void ritaglia(Point upperLeft, Point bottoRight)
    {
    }

    @Override
    public void scala(double factor)
    {
    }

    @Override
    public void colora(Color c)
    {
    }

    @Override
    public void rimuoviColore()
    {
    }
}
```

La classe Forma implementa **3 interfacce** e definisce tutti i metodi da esse dichiarati

```
public interface Selezionabile
{
    void seleziona();
    void deseleziona();
}
```

```
public interface Colorabile
{
    void colora(Color c);
    void rimuoviColore();
}
```

```
public interface Trasformabile
{
    void sposta(Point point);
    void ruota(Angle angle);
    void ritaglia(Point upperLeft, Point bottoRight);
    void scala(double factor);
}
```

# Relazione tra interfacce e classi che le implementano

- Nel momento in cui una classe C decide di implementare un'interfaccia I, tra queste due classi si instaura una relazione di tipo **is-a**, ovvero C è di tipo I
- Comportamento simile a quello dell'ereditarietà
- Quindi anche per le interfacce valgono le regole del polimorfismo
- Ad esempio, la seguente dichiarazione è lecita:

```
SupportoRiscrivibile supporto = new Nastro();
supporto.leggi();
```

- E ci consente di usare l'oggetto della classe Nastro **come fosse di tipo SupportoRiscrivible**
  - Con conseguente **restringimento della visibilità** ai soli metodi dell'interfaccia SupportoRiscrivible

# Interfacce notevoli

| Interfaccia | Descrizione |
|---|---|
| **Comparable** | Impone un ordinamento naturale degli oggetti tramite il metodo: int compareTo(Object b), che restituisce un valore >, = o < 0 se l'oggetto è rispettivamente maggiore, uguale o minore di b |
| **Cloneable** | Le classi che implementano quest'interfaccia dichiarano al metodo clone() di Object che è legale effettuare una copia campo-a-campo delle istanze della classe<br>Il metodo clone() invocato su oggetti di classi che non implementano Cloneable solleva una CloneNotSupportedException |
| **Serializable** | Quest'interfaccia non possiede metodi o campi e serve soltanto ad identificare il fatto che l'oggetto è serializzabile, cioè memorizzabile su un certo supporto |

# Esercizio: Successioni

- Progettare tre classi che realizzano diversi tipi di successioni
  - La successione { $i^2$ } (es. 0, 1, 4, 9, 16, ecc.)
  - La successione casuale (es. -42, 2, 5, 18, 154, ecc.)
  - La successione di Fibonacci (es. 1, 1, 2, 3, 5, 8, 13, ecc.)
- Elaborare un meccanismo generale per la generazione della successione indipendentemente dall'implementazione specifica

# Esercizio: Animali

- Progettare una gerarchia di classi dei seguenti animali, ciascuno con determinate caratteristiche:
  - Uccello (vola, becca)
    - Pinguino (becca, nuota)
    - Aquila (vola, becca)
  - Pesce (nuota)
    - Pesce volante (nuota, vola)
  - Cane (salta, corre, fedele a, domestico)
  - Felino (salta, corre, fa le fusa)
    - Gatto (salta, corre, fa le fusa, domestico)
  - Uomo (salta, corre, pensa, nuota, vola?)

# Passare funzioni in input mediante le interfacce

- Le interfacce permettono il passaggio in input di funzioni con una determinata intestazione
- Ad esempio:

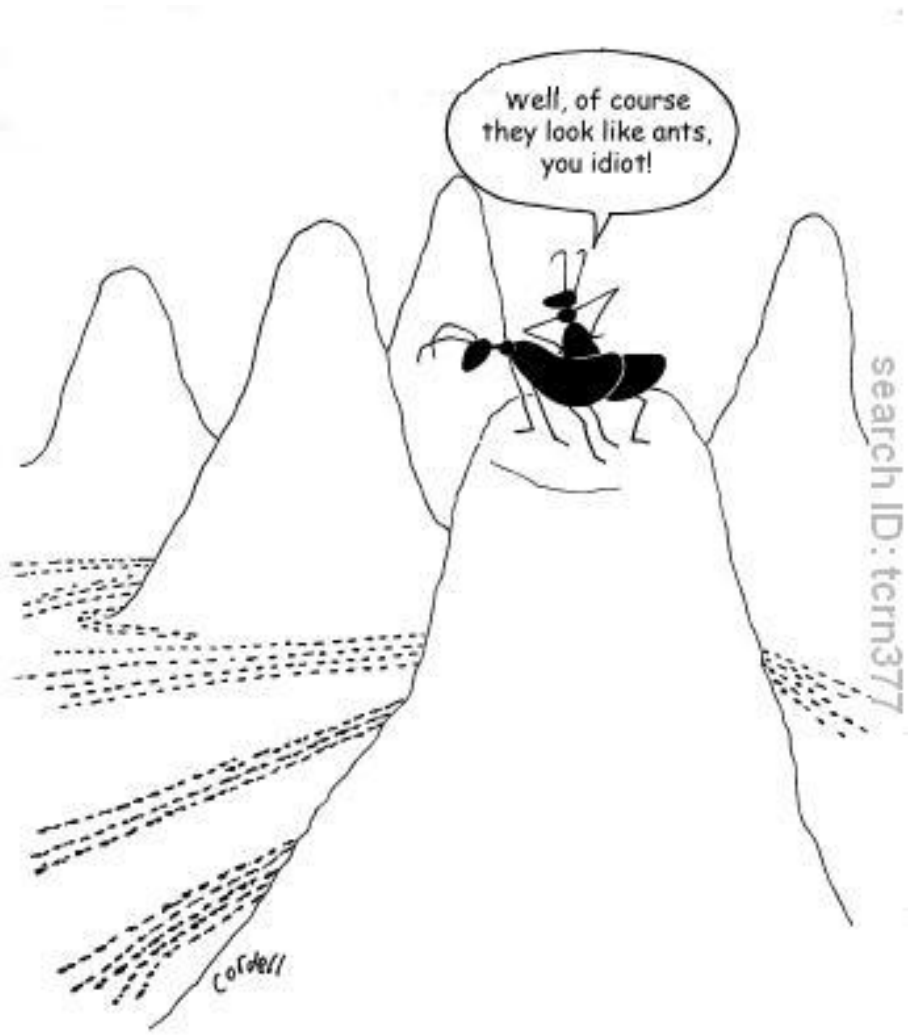
```
public interface Callable
{
    int toString(Object o);
}
```

# Le classi interne

http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

# Classi top-level

- Le classi **usate finora** vengono dette top-level, cioè esse si trovano più in alto di tutte le altre e non sono contenute in altre classi
- Questo tipo di classi richiede un file .java dedicato con lo stesso nome della classe che esso contiene



# Classi annidate (nested class)

- Java consente di scrivere classi all'interno di altre classi
- Le classi presenti all'interno sono chiamate classi annidate (nested class)
- Possono essere **static**
- Oppure **non-static**: in questo caso vengono dette **classi interne** (inner class)

# Classi interne (inner class)

- Prima di poter creare un oggetto della classe interna è necessario istanziare la classe esterna (top-level) che la contiene
- Ciascuna classe interna, infatti, ha un riferimento implicito all'oggetto della classe che la contiene
- Dalla classe interna è possibile accedere a tutte le variabili e a tutti i metodi della classe esterna
- Come tutti i membri di una classe, le classi interne possono essere dichiarate public, protected o private

# Classi interne (inner class)

- Se dalla classe interna viene usato soltanto **this** si fa riferimento ai campi e ai metodi di quella classe
- Per far riferimento alla classe esterna è necessario far precedere **this** dal nome della classe esterna e il punto

# Classi interne (inner class)

```
public class Tastiera
{
    private String tipo;
    private Tasto[] tasti;

    public class Tasto
    {
        private char c;

        public Tasto(char c)
        {
            this.c = c;
        }

        public char premi()
        {
            return c;
        }

        public String toString()
        {
            return Tastiera.this.tipo + ": " + premi();
        }
    }

    public Tastiera(String tipo, char[] caratteri)
    {
        this.tipo = tipo;
        tasti = new Tasto[caratteri.length];

        for(int i=0; i<caratteri.length; i++)
            tasti[i] = new Tasto(caratteri[i]);
    }
}
```

La classe Tasto è una inner-class (quindi non statica)

Tasto ha accesso al campo (privato!) della classe esterna

# Esercizio: Liste linkate di interi con classi interne

- Reimplementare le liste linkate di interi in modo da nascondere all'esterno l'implementazione del singolo elemento della lista

# Classi annidate statiche (static nested class)

- Se invece la classe interna è statica allora essa non richiede l'esistenza di un oggetto appartenente alla classe esterna e non ha nemmeno un riferimento implicito ad essa
  - Come con i metodi statici, non può accedere allo stato dei singoli oggetti della classe esterna
- Da un punto di vista di comportamento, una classe annidata statica è equivalente ad una classe top-level inserita all'interno di un'altra classe top-level
- Sono accessibili tramite il nome della classe esterna che le contiene, secondo la forma
  new ClasseEsterna.ClasseInnestataStatica()

# Classi innestate statiche (static nested class)

```
public class Tastiera
{
    private String tipo;
    private Tasto[] tasti;

    public static class Tasto
    {
        private char c;

        public Tasto(char c)
        {
            this.c = c;
        }

        public char premi()
        {
            return c;
        }

        public String toString()
        {
            return Tastiera.this.tipo + ": " + premi();
        }
    }

    public Tastiera(String tipo, char[] caratteri)
    {
        this.tipo = tipo;
        tasti = new Tasto[caratteri.length];

        for(int i=0; i<caratteri.length; i++)
            tasti[i] = new Tasto(caratteri[i]);
    }
}
```

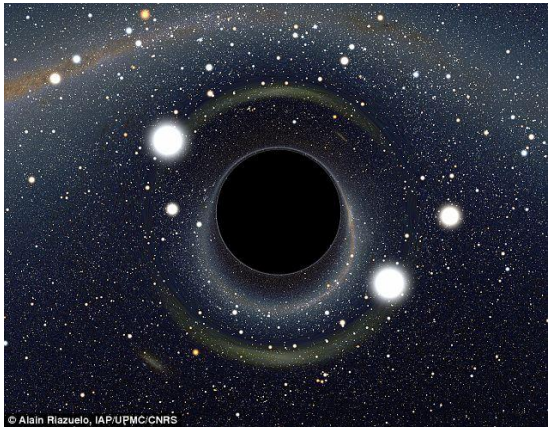
Errore in fase di compilazione: la classe static Tasto non ha accesso implicito ai membri della classe esterna

Ma la classe è visibile all'esterno

```
public class Computer
{
    public static void main(String[] args)
    {
        Tastiera.Tasto tasto = new Tastiera.Tasto('f');
        tasto.premi();
    }
}
```

# Classi annidate: perché sono utili?

- Raggruppamento logico delle classi
  - Se una classe è **utile solo ad un'altra classe**, è logico inserirla al suo interno e tenere le due classi logicamente vicine
- Incrementa l'incapsulamento
  - Una classe B annidata in A può accedere ai membri di A (anche se privati), ma B può essere nascosta al mondo esterno
- Codice più leggibile e più facile da mantenere
  - La vicinanza spaziale è un fattore decisivo



# Esercizio: Disney vs. Marvel (1)

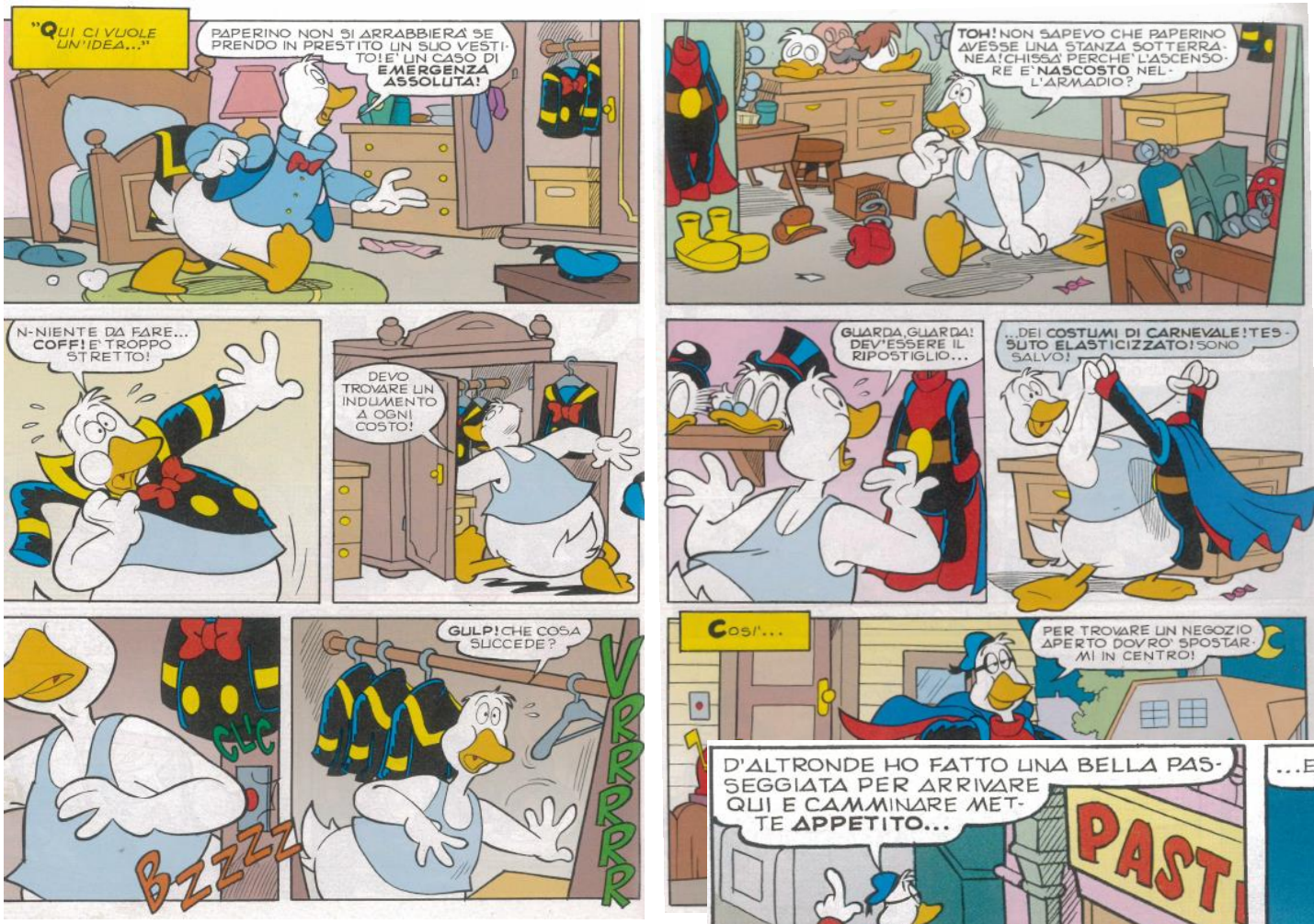
- Modellare uno scontro su campo tra personaggi Disney e personaggi Marvel
- Squadra Disney = Paperinik, Ciccionik, Superpippo
  - Paperinik, Ciccionik e Superpippo sono la **versione supereroe** dei corrispettivi personaggi (Paperino, Ciccio e Pippo)



- Squadra Marvel = Spiderman, La cosa, Magneto
  - Notare che Spiderman è la **versione supereroe** di Peter Parker, noto fotografo del Daily Bugle; La cosa e Magneto invece sono supereroi 24h



# Chi è Ciccionik? [grazie a Mauro Piva!]



# Esercizio: Disney vs. Marvel (2)

- I personaggi con una doppia vita devono esporre un'interfaccia DoppiaVita con i seguenti metodi:
  - assumiIdentitaSegreta(): consente di trasformarsi in supereroe
  - assumiIdentitaPubblica(): consente di assumere le sembianze 'umane'
- Ogni supereroe implementa l'interfaccia Supereroe con:
  - attacca(): sferra l'attacco specifico del supereroe
- Creare una partita su campo (ovvero una classe di test) in cui le due squadre si alternano in attacchi micidiali
  - Il nemico viene scelto casualmente dalla lista dei supereroi avversari

# Esercizio: Disney vs. Marvel (3)

- Hint per versione **semplificata**: Senza utilizzare classi interne, ma solo interfacce ed ereditarietà
- Hint per versione **elaborata**: Per riflettere il fatto che nessuno è a conoscenza dei superpoteri in possesso dalle controparti umane, i supereroi, laddove possibile, dovrebbero estendere le classi umane corrispondenti ed essere implementati come loro classi interne private (ad es. Paperinik estende Paperino, ma solo Paperino potrà restituire un'istanza di Paperinik tramite l'interfaccia DoppiaVita)

# Esempio: l'interfaccia Formula

Consideriamo la seguente interfaccia:

```
public interface Formula
{
        double calculate(int a);
        default double sqrt(int a) { return Math.sqrt(a); }
}
```

# Esempio:
# una classe anonima che implementa l'interfaccia

```
Formula formula = new Formula()
{
        @Override
        public double calculate(int a)
        {
                return sqrt(a * 100);
        }
};

formula.calculate(100); // 100.0
formula.sqrt(16); // 4.0
```

# Interfacce funzionali

- E' disponibile una nuova annotazione @FunctionalInterface
- L'annotazione garantisce che l'interfaccia sia dotata esattamente di un solo metodo astratto
- Ad esempio:

```
@FunctionalInterface
public interface Runnable
{
        void run();
}
```

# Espressioni lambda


- In Java 8 è possibile specificare funzioni utilizzando una notazione molto compatta, le espressioni lambda:

```
() -> { System.out.println("hello, lambda!"); }
```

- Tali espressioni creano oggetti anonimi assegnabili a riferimenti a interfacce funzionali compatibili con l'intestazione (input/output) della funzione creata

```
Runnable r = () -> { System.out.println("hello, lambda!"); };
r.run(); // stampa "hello, lambda!"
```

# Sintassi delle espressioni lambda

- Un'espressione lambda è definita come segue:

(tipo1 nome_param1, …, tipon nome_paramn) -> { codice della funzione }

- Il tipo dei parametri in input è opzionale, perché desunto dal contesto (a quale interfaccia funzionale facciamo riferimento?)
- Le parentesi tonde sono opzionali se in input abbiamo un solo parametro
- Le parentesi graffe intorno al codice della funzione sono opzionali se è costituito da una sola riga
  - Non è necessario return, se il codice è dato dall'espressione di ritorno

# Espressioni lambda equivalenti

- (int a, int b) -> { return a+b; }
- (a, b) -> { return a+b; }
- (a, b) -> return a+b;
- (a, b) -> a+b;

- (String s) -> { return s.replace(‘ ‘, ‘_’); }
- (s) -> { return s.replace(‘ ‘, ‘_’); }
- s -> { return s.replace(‘ ‘, ‘_’); }
- s -> return s.replace(‘ ‘, ‘_’);
- s -> s.replace(‘ ‘, ‘_’);

# Esempi di espressioni lambda

- (int a, int b) -> a+b
- a -> a*a
- () -> System.out.println("Hello, world!");
- s -> System.out.println(s);
- () -> 42;
- () -> 3.1428;
- (String s) -> s.length();
- s -> s.length();

# Consideriamo nuovamente il nostro esempio

```
public interface Formula
{
        double calculate(int a);
        default double sqrt(int a) { return Math.sqrt(a); }
}
```

# Esempio di implementazione di Formula prima e dopo Java 8

## In Java 7:

```
Formula formula = new Formula() {
        @Override
        public double calculate(int a)
        {
                return sqrt(a * 100);
        }
};
```

## In Java 8:

```
Formula formula = a -> Math.sqrt(a*100);
Formula formula2 = a -> a*a;
Formula formula3 = a -> a-1;
```

# Esempio: conversione da un tipo F a un tipo T

```
@FunctionalInterface
public interface Converter<F, T>
{
        T convert(F from);
}

Converter<String, Integer> converter = from -> Integer.valueOf(from);
Integer converted = converter.convert("123");
Converter<String, MyString> stringConverter = a -> new MyString(a);
MyString myString = stringConverter.convert(“123”);
```

# Single Abstract Method (SAM) type

- Le interfacce funzionali sono di tipo SAM
- A ogni metodo che accetta un'interfaccia SAM, si può passare un'espressione lambda compatibile con l'unico metodo dell'interfaccia SAM
- Analogamente per un riferimento a un'interfaccia SAM