Javadoc

- Uno dei pregi di Java è quello di integrare la documentazione con il codice stesso
- Formato dei commenti:
  - `/* commenti */`
  - `// commenti`
  - `/** commenti documentazione */`
- Questi ultimi generano automaticamente la documentazione in formato HTML utilizzando il programma **`javadoc`** (scritto in Java)

Javadoc

- Si può inserire un commento javadoc prima di ogni dichiarazione di classe, campo, costruttore, o metodo. Si può scrivere testo normale, codice HTML, o speciali sintassi.
- Il commento ha due parti: una descrizione a parole, seguita da alcuni **block tags**.
  - I block tags sono etichette standard (opzionali) che discutono alcuni aspetti comuni.

Javadoc

- All'interno della documentazione si possono usare tag HTML nel modo: **`<code> </code>`**
- Inoltre si possono usare specifici tag per comandi di **`javadoc`**. Questi comandi iniziano con il carattere `@`
- Block tags, presenti solo nella sezione dopo la descrizione principale: **`@tag`**
- Inline tags, presenti ovunque: **`{@tag}`**

Javadoc

```
/**
 * Accede un elenco di studenti di un url e ritorna uno
 * Studente che ha come matricola il numero passato come
 * parametro. L'url deve essere un {@link URL} assoluto.
 * <p> <!-- -->
 * Il metodo ritorna sempre un valore. Se lo studente
 * non esiste o l'elenco è vuoto, ritorna null.
 * @param  url   URL assoluto dove si trova l'elenco
 * @param  matr  la matricola dello studente da trovare
 * @returns      il link allo Studente
 * @see          Studente
 */
public Studente getStudente(URL url, int matricola)
{ try {...}
  catch(MalformedUrlException) {return null;}
  catch(ElencoVuotoException) {return null;} ...
}
```

codice html

un inline tag

block tags

il metodo

Javadoc

- Una volta scritta la documentazione, questa viene generata con il programma **`javadoc`**.
- Ad esempio se abbiamo definito una classe **`Dado`** nel file **`Dado.java`** possiamo generare la documentazione con

  `javadoc Dado.java`
- Vengono generati i file di documentazione della classe a partire da un **`index.html`** che possono essere visualizzati con un browser

Esempi di block tag comuni

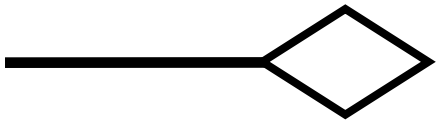
- **`@author`**
  - Specifica il nome dell'autore (si può ripetere); viene considerato solo se **`javadoc`** viene eseguito con l'opzione **`–author`**
- **`@version`**
  - Indica un numero di versione; viene considerato solo se **`javadoc`** viene eseguito con l'opzione **`–version`**
- **`@param`** (per ognuno dei parametri passati)
  - Descrive uno dei parametri passati

Esempi di block tag comuni

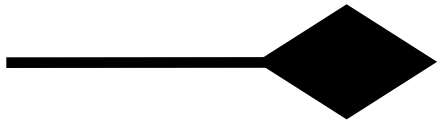
- **`@returns`**
  - Descrive il valore di ritorno restituito al chiamante
- **`@throws`** (per ogni eccezione che si può verificare)
  - Descrive il tipo di eccezione e la sua descrizione
- **`@see`**
  - Rimanda a un'altra voce di documentazione
- **`@deprecated`**
  - Indica che non andrebbe più usato

Notazione UML

- Aggregazione:
  - una classe aggrega un'altra se quest'ultima è contenuta in essa.
- Composizione:
  - è un tipo di aggregazione più forte, in cui l'oggetto contenuto non esiste al di fuori del contenitore



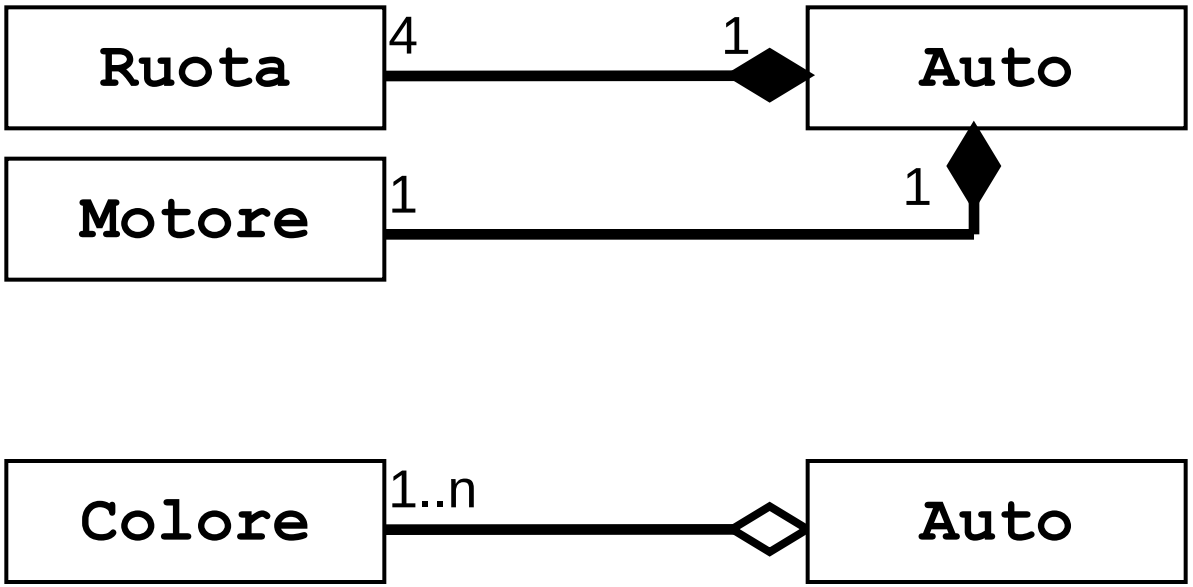
aggregazione



composizione

Notazione UML

- I numeri ai lati delle “frecce” indicano quanti elementi vengono raggruppati.



Notazione UML

- Sia l'aggregazione che la composizione vengono classificate come relazioni di tipo HA
- La dizione HA però è ambigua:
  - si ha aggregazione se la presenza dei contenuti è opzionale: il contenitore **può** contenerli
  - si ha composizione se i contenuti non esistono fuori dal contenitore
- Nella composizione, l'oggetto contenuto cessa di esistere quando si distrugge il contenitore.

Esercizio: **FarmaCell**

- Sia data la classe **Farmaco** (o **Studente**, **Calciatore**...) contenente dati strutturati

```
public class Farmaco
{ //campi
  private String nome;
  private double dose; // in mg
  ...
  //costruttori
  public Farmaco(String nome, double dose ...)
  { this.nome = nome; this.dose = dose;...}
  public Farmaco() { this(...default...); }
  ...
  //metodi
  public getNome(), setNome(), getDose() ...
}
```

Esercizio: **`FarmaCell`**

- La classe **`FarmaCell`** viene usata per una "cella" che memorizza un farmaco:

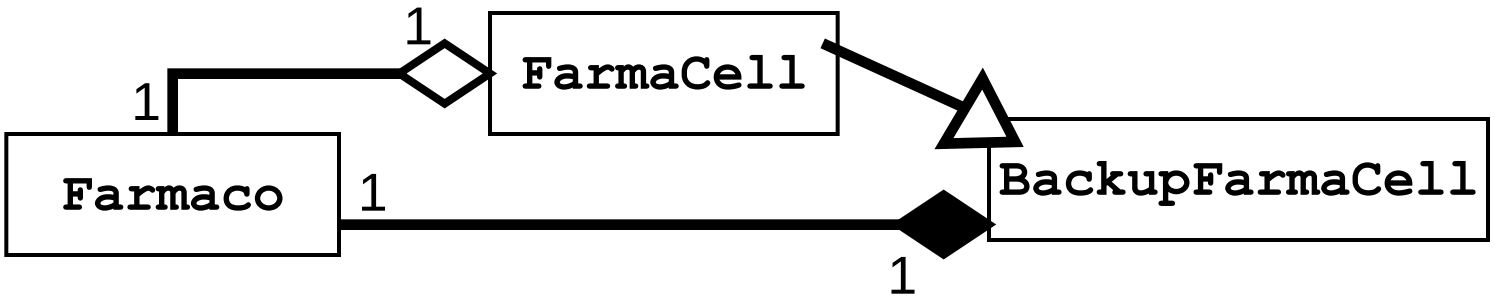
```
public class FarmaCell
{ //campi
  private Farmaco val;
  //costruttori
  public FarmaCell(Farmaco f) { val = f; }
  public FarmaCell() { this(null); }
  //metodi
  public Farmaco getVal() { return val; }
  public void setVal(Farmaco f) { val=f; }
  public void clear() { val = null; }
}
```

Esercizio: **`FarmaCell`**

- Esercizio: estendere la classe a una classe **`BackupFarmaCell`**, analogamente alla classe **`BackupCell`** vista a lezione. Quindi:
  - esiste un **campo** **`oldVal`** in più per salvare il vecchio valore tutte le volte che c'è una modifica
  - i metodi esistenti vanno sovrascritti in qualche modo per tenere conto di questa funzionalità
  - esisterà un metodo aggiuntivo **`restore()`** per ripristinare il valore di backup

Ereditarietà e composizione

- Ereditarietà = relazione “è” (is),
  Composizione = relazione “ha” (has).
  - L’ereditarietà specializza una classe a una sottoclasse più potente. La composizione crea una nuova classe partendo da elementi noti.
  - L’ereditarietà estende l’interfaccia ma di default non la varia. La composizione consente di usare vecchie funzionalità in una nuova interfaccia.



Esempio: `ClockDisplay`

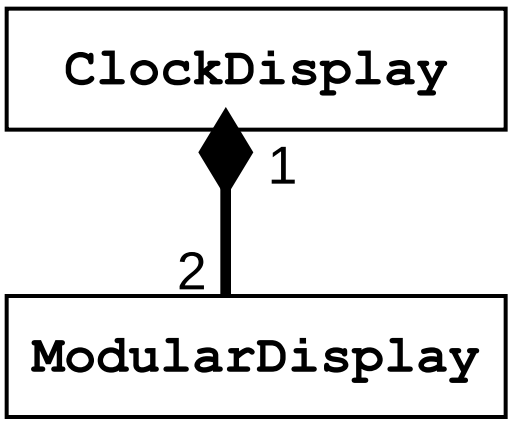
- Vogliamo creare una classe che visualizzi un numero in modalità orologio digitale.
- Ad esempio hh:mm (o versioni più complicate).
  - Visualizza le sette e mezza di sera come 19:30.
- Se usassimo un'unica classe non sarebbe modulare: i due display di ore e minuti sono analoghi. Ma uno si azzera a 24, l'altro a 60.
- Soluzione: usare una classe di ausilio, che chiameremo **`ModularDisplay`** .

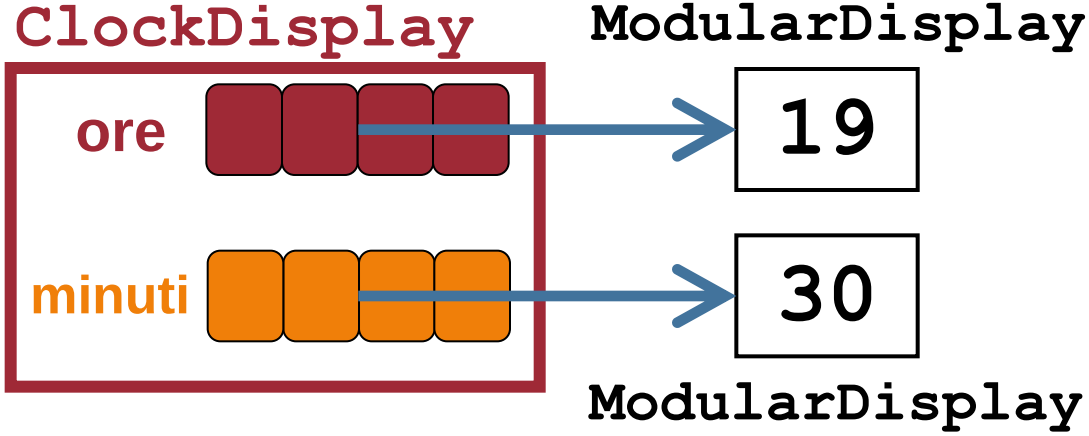
Esempio: `ClockDisplay`

- La classe **`ModularDisplay`** deve:
  - visualizzare un valore in modulo X, dove X può essere variabile (24 oppure 60)
  - fare “padding” portando tutti i valori a due cifre
- Essa deve fornire:
  - un metodo di accesso per ottenere il valore
  - un metodo di set per impostarlo
  - un metodo di set in versione incremento dal valore corrente

Esempio: `ClockDisplay`

- La classe **`ClockDisplay`** deve:
  - comprendere più campi **`ModularDisplay`**
  - visualizzare orari completi
- La struttura è quindi:





Esempio: `ClockDisplay`

- Le varie istanze di **`ModularDisplay`** usano un modulo diverso (base 24, base 60...)
  - questo valore andrà messo in un campo apposito
- Inoltre ci saranno quindi i seguenti membri:
  - campo **`valore`**, contiene il numero!
  - costruttore con opportuni parametri
  - metodo **`toString()`**, letto a 2 cifre
  - metodi get e set (quest'ultimo controllato)
  - metodi di incremento

Esempio: `ClockDisplay`

- Le istanze di **`ClockDisplay`** devono gestire due o più **`ModularDisplay`**
- **`ClockDisplay`** può ricevere un input esterno (messaggio) di aumentare di 1 il campo inferiore
  - Ci vuole un metodo che implementi l'incremento
- Non deve stampare niente a video, ma ritornare eventuali stringhe da stampare in valori di ritorno (importante per l'ereditarietà)
- Prova: visualizzare un intervallo di tempo (preso da differenza di **`currentTimeMillis`**)

Esempio: `ClockDisplay`

- Estensioni possibili:
  - considerare un display ore minuti secondi
  - considerare anche decimi di secondo: hh:mm:ss.d
  - considerare un display a 12h (notare che questa volta non si visualizza la mezza come 00:30 ma come 12:30)
  - eliminare il padding per le ore (che non serve in realtà, 06:30 dovrebbe vedersi come 6:30)
- Le ultime due modifiche richiedono refactoring.

Refactoring

- Nel progetto di classi, conviene pensare al futuro, e in particolare a chi dovrà modificarle.
- Strutture di classi semplici: una classe deve rappresentare un unico concetto (questo però non vuol dire avere un unico metodo buono)
- Poche relazioni di dipendenza: questo fa sì che le modifiche si propaghino poco.

Refactoring

- Nell'esempio di **`ClockDisplay`** però è difficile evitare il refactoring se non abbiamo considerato attentamente tutte le eventualità.
  - per il display con ore a 12: implementare un campo booleano che dice se vogliamo vedere il primo valore lecito modulo *M* come 0 o come *M*
  - per il display con una/due cifre: aggiungere un campo booleano che dica se vogliamo padding