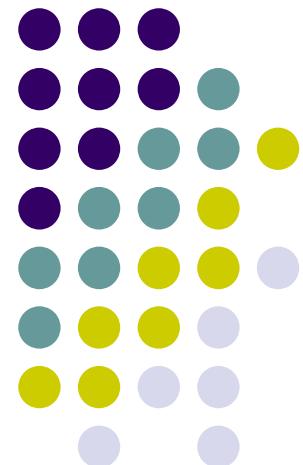


# Corso di Programmazione

*Introduzione alle Classi  
e agli Oggetti*





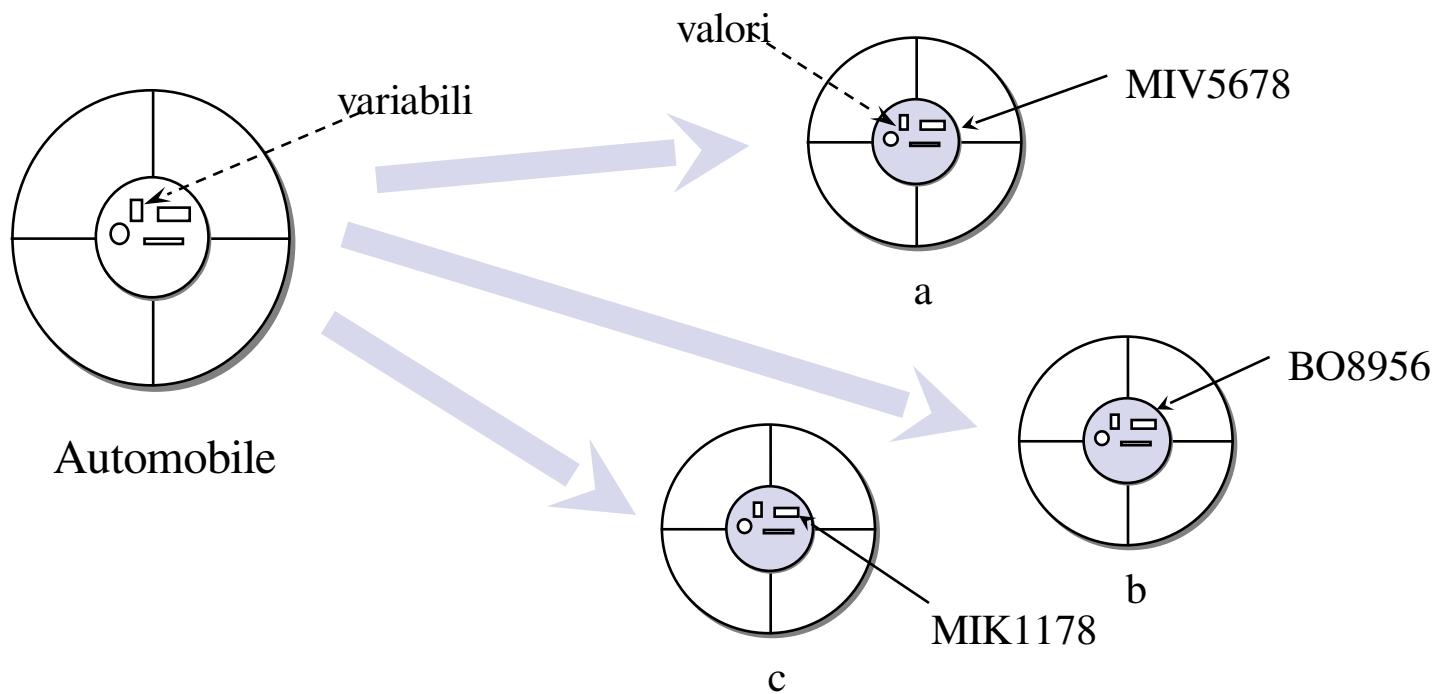
# Classi e oggetti

- Nei linguaggi a orientati agli oggetti, il costrutto classe consente di definire nuovi tipi di dato (astratti) e le relative operazioni
  - Il linguaggio fornisce i *meccanismi per l'information hiding*, cioè per «incapsulare» i dati all'interno della classe.
  - Le operazioni sono fornite sotto forma di funzioni.
  - Si possono creare istanze di una classe e si possono eseguire operazioni su di esse.
  - **Un oggetto in realtà altro non è che una istanza di una classe** e lo **stato di un oggetto** è rappresentato dai valori correnti delle variabili che costituiscono la struttura dati concreta sottostante il tipo astratto.



# L'istanziazione degli oggetti

- Un oggetto è una istanza (“esemplare”) di una classe
- Due esemplari della stessa classe sono distinguibili soltanto per il loro stato mentre il comportamento (definito dalle funzioni della classe) è sempre identico





# Dichiarazione di classe in Java

- Il linguaggio Java supporta esplicitamente la dichiarazione e la definizione di tipi astratti da parte dell'utente mediante il costrutto **class**
- le istanze di una classe vengono dette **oggetti**.
- In una dichiarazione **class** occorre specificare sia la struttura dati che le operazioni consentite su di essa.
- La dichiarazione di classe inizia con la parola riservata **public** che è un **modificatore di accesso** per consentire l'uso della classe a tutti gli utenti (fuori dal suo package).
- Ogni dichiarazione di classe deve essere salvata in un file che ha lo stesso nome della classe e termina con l'estensione **.java**

```
public class NomeClasse {  
    dichiarazioni dei dati // attributi  
    funzioni // metodi  
}
```

File NomeClasse.java



# Classi in Java e encapsulamento

- Una classe possiede, in generale, **membri pubblici e privati**.
- I **membri pubblici** sono tipicamente le operazioni (**metodi**) consentite ad un utilizzatore della classe. Esse sono *tutte e sole le* operazioni che un utente può eseguire, in maniera esplicita od implicita, sugli oggetti. Ma se necessario e opportuno possono essere pubblici anche attributi il cui accesso si vuole consentire dall'esterno.
- I **membri privati** comprendono di solito le strutture dati (**variabili di istanza**) e le operazioni che si vogliono rendere inaccessibili dall'esterno (information hiding). Le variabili di istanza rappresentano lo stato degli oggetti.
- Esempio: un tipo astratto Contatore in JAVA:

```
public class Contatore
{
    private long value; // variabile di istanza, valore corrente
    private long max; // variabile di istanza, valore massimo

    // operazione incremento
    public void Incrementa() { // corpo del metodo }
    // operazione decremento
    public void Decrementa() { // corpo del metodo }
}
```

# Classi in Java e encapsulamento



- In generale in una classe ciò che è pubblico ne costituisce l'interfaccia, accessibile a tutti
- ciò che è privato ne costituisce l'implementazione che deve essere encapsulata e non accessibile
- E' possibile poi avere membri non definiti (né pubblici né privati)
- Questi sono accessibili a tutte le classi **dello stesso package** (quindi sono privati all'esterno del package).



# UML

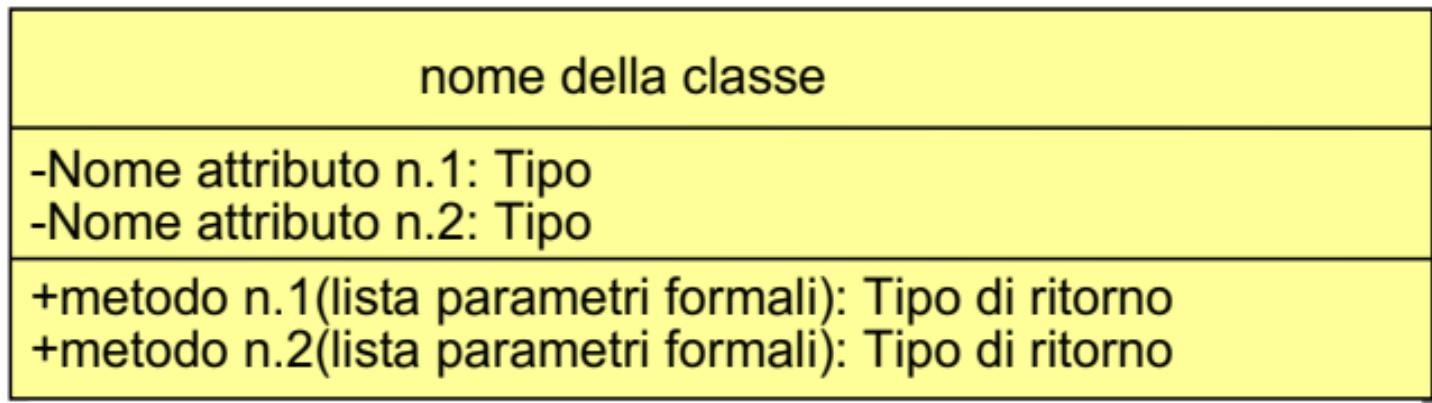
- ***Unified Modeling Language*** (UML) è uno dei linguaggi di rappresentazione grafica più utilizzati attualmente per la modellazione di sistemi orientati agli oggetti
- UML può essere utilizzato lungo tutte le fasi del ciclo di sviluppo del software, dalla specifica alla progettazione alla generazione (semi) automatica del codice
- UML è uno standard OMG, una delle sue caratteristiche più interessanti è *l'estensibilità e l'indipendenza da specifici approcci di progettazione ed analisi*
- *In questo corso utilizzeremo UML per rappresentare le classi e le relazioni tra di esse*

# Rappresentazione grafica



La notazione UML per la rappresentazione di una classe è una *metafora* nella quale sono rappresentati in tre sezioni contigue:

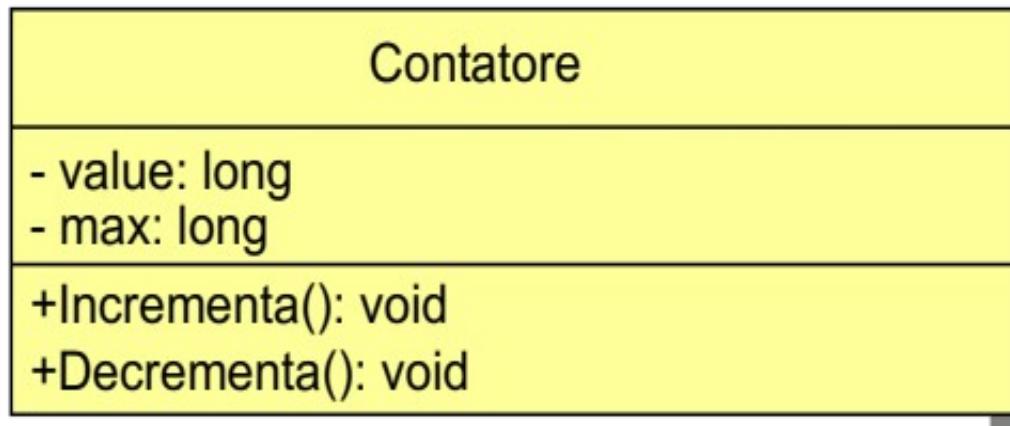
- ◆ il nome della Classe
- ◆ la struttura dati (sequenza di *variabili di istanza*)
- ◆ le operazioni(*metodi* della classe).





# Metafora per la classe Contatore

- Il simbolo + accanto ai membri di una classe significa *public*
- Il simbolo - accanto ai membri di una classe significa *private*





# Running example

- Utilizziamo la classe contatore per sviluppare un primo semplice esempio.  
L'implementazione della classe verrà estesa durante le lezioni.
- Inseriamo la classe in un **package** che chiameremo *runningexamples*
- *Chiamiamo la classe Counter e dotiamola di delle due variabili di istanza private value e max e dei due metodi pubblici per incrementare e decrementare il contatore come nella metafora UML.*



# Classe Counter – versione 1

```
package runningexamples;

public class Counter {

    private long value;
    private long max;

    public void increment() {value++;}
    public void decrement() {value--;}

}
```

File: **Counter.java**

- La classe Counter può essere utilizzata all'esterno del package a cui appartiene (è `public`)
- Gli attributi sono `private` e quindi non accessibili al di fuori della classe
- I metodi *increment* e *decrement* sono pubblici e quindi accessibili anche al di fuori della classe



# Classe di prova – versione 1

```
package runningexamples;

public class RunningExamples {

    public static void main(String[] args) {

        Counter c = new Counter();
        c.increment();
        c.decrement();
    }
}
```

File: RunningExamples.java

- Per provare la classe Counter è necessario che vengano istanziati oggetti della classe e che un metodo main sia presente *in modo che la virtual machine possa eseguire l'applicazione (la JVM invoca automaticamente solo il metodo main)*
- **Ogni classe che contiene un metodo main è un'applicazione Java**
- *la classe Counter non contiene un metodo main e quindi da sola non costituisce una applicazione Java*

# Classe di prova – versione 1



```
package runningexamples;

public class RunningExamples {

    public static void main(String[] args)
    {
        Counter c = new Counter();
        c.increment();
        c.decrement();
    }
}
```

L'istruzione new Counter() instanzia un oggetto della classe Counter

- Il main è sempre un metodo di classe (in questo caso della classe RunningExamples)
- Quindi il file sorgente si chiama RunningExamples.java
- La classe RunningExample è pubblica ed è stata inserita nello stesso package in cui si trova la classe Counter (runningexamples), in questo modo la classe RunningExamples può utilizzare senza problemi la classe Counter ed essere a sua volta utilizzata da altre classi che aggiungeremo al package.
- Il metodo main è dichiarato pubblico altrimenti non potremmo eseguirlo
- Main è dichiarato static, questo significa che può essere eseguito anche se non è stato istanziato nessun oggetto della classe alla quale appartiene (RunningExamples)

# Classe di prova – versione 1



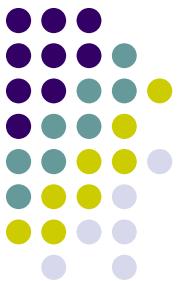
```
package runningexamples;

public class RunningExamples {

    public static void main(String[] args)
    {
        Counter c = new Counter();
        c.increment();
        c.decrement();
    }
}
```

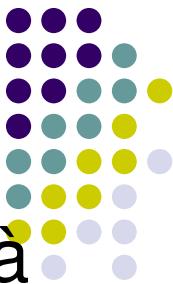
I metodi vengono chiamati su un oggetto della classe utilizzando la notazione .

- Il main è sempre un metodo di classe (in questo caso della classe RunningExamples)
- Quindi il file sorgente si chiama RunningExamples.java
- La classe RunningExample è pubblica ed è stata inserita nello stesso package in cui si trova la classe Counter (runningexamples), in questo modo la classe RunningExamples può utilizzare senza problemi la classe Counter ed essere a sua volta utilizzata da altre classi che aggiungeremo al package.
- **Il metodo main è dichiarato pubblico** (ovviamente) ed è dichiarato **static**, questo significa che può essere eseguito anche se non è stato istanziato nessun oggetto della classe alla quale appartiene (RunningExamples)

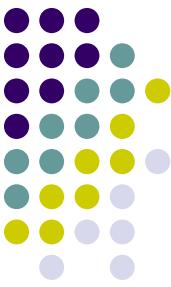


- Non possiamo eseguire
  - Increment
  - Decrement
- Senza aver prima istanziato l'oggetto
- Possiamo fare solo oggetto.increment()
- Mentre con i metodi statici possiamo invocare il metodo senza oggetto main()

# Osservazione

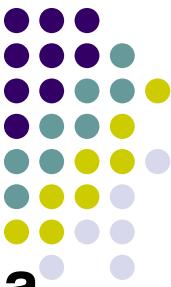


- La classe Counter nella versione attuale in realtà è...inutile!
- Non abbiamo modo di specificare lo stato iniziale degli oggetti che istanziamo. Quanto vale *max*? Quanto vale inizialmente *value*? Come facciamo ad assicurarci che *value* non superi il valore *max*? Non abbiamo neanche modo di sapere quanto vale il contatore in un certo istante visto che le variabili di istanza sono private e non possiamo accedervi.
- Quindi è necessario approfondire ulteriormente...



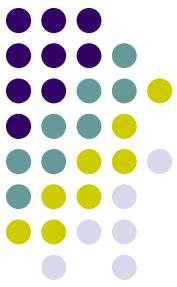
# Ciclo di vita di un oggetto

- Una classe specifica solo
  - la "struttura" condivisa da tutti gli oggetti di quel tipo e
  - le operazioni che possono essere effettuate su di essi,
- ma gli oggetti esistono in seguito ad una operazione di istanziazione che li crea e li alloca in memoria. Il ciclo di vita di un oggetto in Java comprende le seguenti "fasi":
  - ◆ creazione
  - ◆ inizializzazione
  - ◆ definizione di un riferimento
  - ◆ utilizzo
  - ◆ distruzione
- Come vedremo non tutto è a carico del programmatore... ma qualcosa sì!



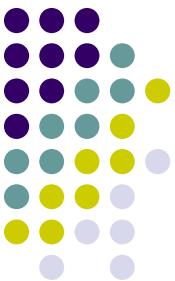
# Costruttori (1/6)

- Un oggetto viene creato utilizzando l'istruzione **new** e la chiamata ad un metodo speciale detto **costruttore che serve ad eseguire l'inizializzazione dell'oggetto**
- La chiamata al costruttore è indicata dal nome della classe seguita dalle parentesi, ad esempio:  
**new Counter()**
- Se la classe non fornisce un costruttore, viene chiamato un **costruttore di default fornito dal compilatore**, come avviene nell'esempio d'uso della prima versione della classe Counter.
- *E' opportuno che ogni classe preveda almeno un costruttore per inizializzare opportunamente lo stato degli oggetti all'atto della loro creazione*



# costruttore

- oggetto viene creato utilizzando l'istruzione **new**



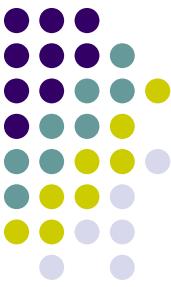
# Costruttori (2/6)

- Un *costruttore* è un metodo:
  - che ha lo stesso nome della classe
  - non restituisce alcun risultato (neanche **void**)

```
public Counter(long m) {  
    max=m;  
    value=0;  
}
```

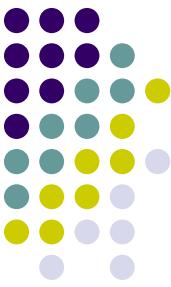
*il costruttore*

*lista dei parametri*



# Costruttori (3/6)

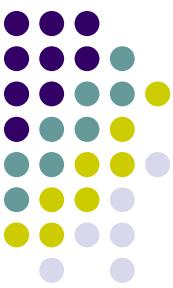
- Una classe può avere più costruttori. I costruttori possono essere:
  - senza argomenti (la lista dei parametri è vuota)
  - con argomenti (uno o più parametri)
  - di copia (per inizializzare un oggetto con lo stato di un altro oggetto già esistente), in java un caso particolare di costruttore con argomenti
- I costruttori (se più di uno) devono essere **distinguibili per overloading**
- Sovraccaricare i costruttori consente di inizializzare gli oggetti di una classe in modi diversi



# Costruttori (3/6)

- Una classe può avere più costruttori. I costruttori possono essere:
  - senza argomenti (la lista dei parametri è vuota)
  - con argomenti (uno o più parametri)
  - di copia (per inizializzare un oggetto con lo stato di un altro oggetto già esistente), in java un caso particolare di costruttore con argomenti
- I costruttori (se più di uno) devono essere **distinguibili per overloading**
- Sovraccaricare i costruttori consente di inizializzare gli oggetti di una classe in modi diversi

# Costruttori (4/6)



- **Attenzione:** nel caso in cui venga esplicitamente richiamato almeno un costruttore, il compilatore non è più in grado di utilizzare il costruttore di default!!! Quindi è necessario fornire il costruttore senza argomenti

The screenshot shows the Apache NetBeans IDE 14 interface. On the left, the Projects panel displays a project named "RunningExamples" with a source package "runningexamples" containing files "Counter.java" and "RunningExamples.java". The main workspace shows the code for "RunningExamples.java". The code is as follows:

```
package runningexamples;

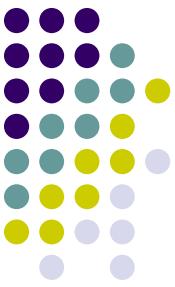
public class RunningExamples {

    public static void main(String[] args) {

        Counter c = new Counter(4);
        Counter d = new Counter();
    }
}
```

The line "Counter d = new Counter();" is highlighted in red, indicating a syntax error. The output window at the bottom shows the error message:

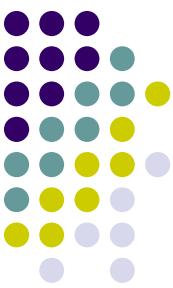
```
/Users/valeria/Documents/DIDATTICA/CORSI/Programmazione/Java/Esercizi/RunningExamples/src/runningexamples/RunningExamples.java:9: error: constructor
        Counter d = new Counter();
                           ^
               required: long
               found:   no arguments
               reason: actual and formal argument lists differ in length
1 error
```



# COSTRUTTORE

- new Counter()
- Se non ho definito un costruttore allora il compilatore provvede

# Costruttori (4/6)



- **Attenzione:** nel caso in cui venga esplicitamente richiamato almeno un costruttore, il compilatore non è più in grado di utilizzare il costruttore di default!!! Quindi è necessario fornire il costruttore senza argomenti

The screenshot shows the Apache NetBeans IDE interface with the title bar "RunningExamples - Apache NetBeans IDE 14". The left sidebar shows a project named "RunningExamples" with a source package "runningexamples" containing files "Counter.java" and "RunningExamples.java". The main editor window displays the following Java code:

```
package runningexamples;
public class RunningExamples {
    public static void main(String[] args) {
        Counter c = new Counter(4);
        Counter d = new Counter();
    }
}
```

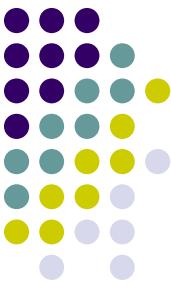
A yellow callout box with a black border and white text points to the line "Counter d = new Counter();". Inside the callout box, the text reads: "il compilatore non è più in grado di utilizzare il costruttore di default". The output panel at the bottom shows a single error message:

```
/Users/valeria/Documents/DIDATTICA/CORSI/Programmazione/Java/Esercizi/RunningExamples/src/runningexamples/RunningExamples.java:9: error: constructor Counter required: long found: no arguments reason: actual and formal argument lists differ in length
1 error
```



# COSTRUTTORE

- Se io vado a definire un costruttore con argomenti
- All'interno della classe definisco
- Public Counter(int a , int b)
- Il fatto che ci siano argomenti in input , significa che NON POSSO PIU RICHIAMARE IL COSTRUTTORE GRATUITO FORNITO DAL COMPILATORE



# COSTRUTTORE

- Public Counter(int a , int b)
- Il fatto che ci siano argomenti in input , significa che NON POSSO PIU RICHIAMARE IL COSTRUTTORE GRATUITO FORNITO DAL COMPILATORE
- Quindi in questo caso
- Dobbiamo noi definire il costruttore privo di attributi



# COSTRUTTORE

- Se definiamo un costruttore con argomenti
- DOBBIAMO DEFINIRE ANCHE IL COSTRUTTORE SENZA ARGOMENTI
- All'atto della definizione del costruttore con argomenti viene meno l'automatico del compilatore, che ci forniva il costruttore senza argomenti



# 3 tipologie di COSTRUTTORE

- senza argomenti (la lista dei parametri è vuota)
- con argomenti (uno o più parametri)
- di copia (per inizializzare un oggetto con lo stato di un altro oggetto già esistente),



# 3 tipologie di COSTRUTTORE

- senza argomenti (la lista dei parametri è vuota)
- Possiamo instanziare anche senza aver definito alcun metodo costruttore
- Il compilatore sopperisce automaticamente

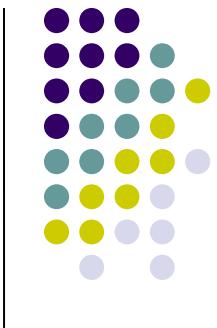


# 3 tipologie di COSTRUTTORE

- con argomenti (uno o più parametri)

Counter(33)

- In questo caso, dobbiamo definire un costruttore con argomenti, ma non solo!
- Perdiamo anche il costruttore senza argomenti creato ad hoc dal compilatore
- Dobbiamo noi definire non solo il costruttore con Argomenti, ma anche quello senza argomenti





# 3 tipologie di COSTRUTTORE

- di copia (per inizializzare un oggetto con lo stato di un altro oggetto già esistente)
- Counter c2=new Counter(c1)
- Attenzione che c1 è un oggetto dello stesso tipo di c1



# Costruttore di copia

- Il costruttore senza argomenti è un metodo senza parametri
- I costruttori con argomenti possono essere più di uno, hanno uno o più parametri, è importante che siano distinguibili per numero e tipo dei parametri a tempo di compilazione
- ***Il costruttore di copia è un costruttore con argomenti che ha come unico parametro in ingresso un riferimento ad un oggetto già esistente della stessa classe, il cui stato verrà copiato nel nuovo oggetto***

```
public Counter(Counter C) {  
    max=C.max;  
    value=C.value;  
}
```

```
Counter c = new Counter(4);  
Counter d = new Counter(c);
```

Implementazione del costruttore di copia.

Osservazione 1: C è un riferimento (passato per valore)!

Osservazione 2: Un metodo può accedere direttamente agli attributi di un oggetto della sua classe, perciò è possibile scrivere C.max e C.value

Creazione e inizializzazione di oggetti  
attraverso:

- un costruttore con un parametro di tipo long  
il costruttore di copia

Osservazione: c e d sono riferimenti che "puntano" agli oggetti.



# Costruttore di copia

- Il costruttore senza argomenti è un metodo senza parametri
- I costruttori con argomenti possono essere più di uno, hanno uno o più parametri, è importante che siano distinguibili per numero e tipo dei parametri a tempo di compilazione
- ***Il costruttore di copia è un costruttore con argomento unico parametro in ingresso un riferimento ad un oggetto esistente della stessa classe, il cui stato verrà copiato nell'oggetto***

```
public Counter(Counter C) {  
    max=C.max;  
    value=C.value;  
}
```

```
Counter c = new Counter(4);  
Counter d = new Counter(c);
```

un costruttore con  
un parametro di  
tipo long

il costruttore  
di copia

Implementazione del costruttore di copia.

Osservazione 1: C è un riferimento (passato per valore)!

Osservazione 2: Un metodo può accedere direttamente agli attributi di un oggetto della sua classe, perciò è possibile scrivere C.max e C.value perché siamo all'interno della classe

Creazione e inizializzazione di oggetti

Osservazione: c e d sono riferimenti che "puntano" agli oggetti.

# Classe Counter e classe di prova – versione 2



```
package runningexamples;

public class Counter {

    private long value;
    private long max;

    //costruttore senza argomenti
    public Counter() {max=10; value=0;}
    //costruttore con argomenti
    public Counter(long m) {max=m; value=0;}
    //costruttore di copia
    public Counter(Counter C) {max=C.max; value=C.value;}

    // metodi per la gestione del contatore
    public void increment() {value++;}
    public void decrement() {value--;}
```

```
package runningexamples;

public class RunningExamples {

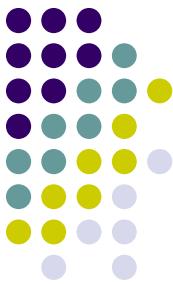
    public static void main(String[] args) {

        Counter b = new Counter();
        Counter c = new Counter(4);
        Counter d = new Counter(c);

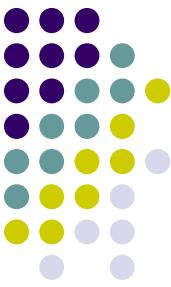
        c.increment();
        c.decrement();
    }
}
```

- Ora possiamo **creare** e **inizializzare** gli oggetti, **definire i riferimenti** agli oggetti creati e **utilizzarli** incrementando e decrementando il valore dei contatori, ma ancora non riusciamo a sapere quanto valgono i contatori e a controllare che non sforino il valore max.

# I metodi get e set (1/2)

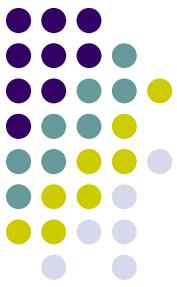


- Chi utilizza gli oggetti di una classe può dover accedere allo stato dell'oggetto in lettura o in scrittura
- Le classi perciò forniscono spesso dei metodi pubblici che consentono di ottenere il valore delle variabili di istanza private (**metodi get**) o di modificarne il valore (**metodi set**)

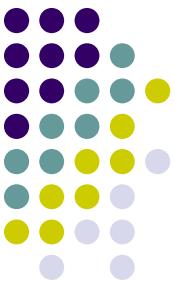


# I metodi get e set (2/2)

- Accesso in lettura delle variabili di istanza (acquisizione dello stato dell'oggetto)
  - Consentono di accedere al valore corrente dell'oggetto
  - Funzioni “get()”: **public T getVI() { return VI; }**
- Accesso in scrittura dei valori delle variabili di istanza (posizionamento dello stato dell'oggetto)
  - Consentono di modificare lo stato dell'oggetto
  - Funzioni “set(T)”: **public void setVI(T value) { VI=value; }**

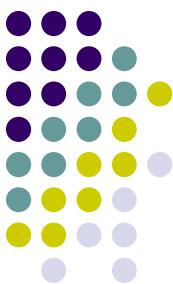


- Visualizzare un valore di un attributo
- Accedere in lettura a un attributo
- **public Tipo getAttributo()**
  - **/\*restituisce il valore dell'Attributo\*/**
  - **return attributo;}**



- Modifica il valore di un attributo
  - Accesso in scrittura all'attributo
- 
- **public void setAttributo(TipoValore  
valore)**
  - **/\*imposta attributo=valore\*/**
  - {
  - **attributo=valore**
  - }

# Classe Counter – versione 3



```
package runningexamples;

public class Counter {

    private long value;
    private long max;

    //costruttore senza argomenti
    public Counter() {max=10; value=0;}
    //costruttore con argomenti
    public Counter(long m) {max=m; value=0;}
    //costruttore di copia
    public Counter(Counter C) {max=C.max; value=C.value;}
    // getter
    public long get_value() {return value;}
    public long get_max() {return max;}
    // setter
    public void set_max(long m) {max=m;}
    // metodi per la gestione del contatore
    public void increment() {value++;}
    public void decrement() {value--;}
    public boolean overflow() {return (value+1)>=max;}
    public boolean underflow() {return (value-1)<0;}
}
```

Assegna all'attributo max il  
valore del parametro m

# Classe di prova – versione 3



```
package runningexamples;

public class RunningExamples {

    public static void main(String[] args) {

        Counter b = new Counter();
        Counter c = new Counter(4);
        Counter d = new Counter(c);

        System.out.print("valore corrente del contatore b: " + b.get_value() + "\n");
        System.out.print("valore corrente del contatore c: " + c.get_value() + "\n");
        System.out.print("valore corrente del contatore d: " + d.get_value() + "\n");
        System.out.print("soglia corrente del contatore b: " + b.get_max() + "\n");
        System.out.print("soglia corrente del contatore c: " + c.get_max() + "\n");
        System.out.print("soglia corrente del contatore d: " + d.get_max() + "\n");

        c.increment();
        System.out.print("valore corrente del contatore c dopo l'incremento: " + c.get_value() + "\n");

        c.decrement();
        System.out.print("valore corrente del contatore c dopo il decremento: " + c.get_value() + "\n");

        d.set_max(20);
        System.out.print("valore corrente della soglia massima di d: " + d.get_max() + "\n");
    }
}
```

# Classe Counter – versione 4



```
package runningexamples;

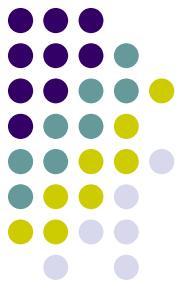
public class Counter {

    private long value;
    private long max;

    //costruttore senza argomenti
    public Counter()
        {max=10; value=0;}
    //costruttore con argomenti
    public Counter(long m)
        {max=m; value=0;}
    //costruttore di copia
    public Counter(Counter C)
        {max=C.max; value=C.value;}
    // getter
    public long get_value() {return value;}
    public long get_max() {return max;}
    // setter
    public void set_max(long m) {max=m;}
    // metodi per la gestione del contatore
    public void increment() {value++;}
    public void decrement() {value--;}
    //predicati
    public boolean overflow() {return (value+1)>=max;}
    public boolean underflow() {return (value-1)<0;}
```

- Possiamo introdurre ora dei metodi che controllino che le operazioni di incremento e decremento possano essere eseguite in sicurezza, in modo che il valore del contatore non superi mai il valore di soglia corrente e che non diventi mai minore di zero.
- Questi metodi **verificano** cioè le **pre-condizioni** per le operazioni sul contatore e **sono "predicati"** sullo stato degli oggetti, **tornano cioè un valore booleano** e **possono essere utilizzati nei passi condizionali**

# Classe di prova – versione 4



```
package runningexamples;

public class RunningExamples {
    public static void main(String[] args) {
        Counter b = new Counter();
        Counter c = new Counter(4);
        Counter d = new Counter(c);

        System.out.print("valore corrente del contatore b: " + b.get_value() + "\n");
        System.out.print("valore corrente del contatore c: " + c.get_value() + "\n");
        System.out.print("valore corrente del contatore d: " + d.get_value() + "\n");
        System.out.print("soglia corrente del contatore b: " + b.get_max() + "\n");
        System.out.print("soglia corrente del contatore c: " + c.get_max() + "\n");
        System.out.print("soglia corrente del contatore d: " + d.get_max() + "\n");

        if(!c.overflow()) {
            c.increment();
            System.out.print("valore corrente del contatore c dopo l'incremento: " + c.get_value() + "\n");
        }
        else System.out.print("pre-condizione non verificata, impossibile incrementare c! \n");
        if( !c.underflow()){
            c.decrement();
            System.out.print("valore corrente del contatore c dopo il decremento: " + c.get_value() + "\n");
        }
        else System.out.print("pre-condizione non verificata, impossibile decrementare c! \n");
        if(!c.underflow()){
            c.decrement();
            System.out.print("valore corrente del contatore c dopo il decremento: " + c.get_value() + "\n");
        }
        else System.out.print("pre-condizione non verificata, impossibile decrementare c! \n");

        d.set_max(20);
        System.out.print("valore corrente della soglia massima di d: " + d.get_max() + "\n");
    }
}
```



# Distruzione degli oggetti

- Il ciclo di vita di un **oggetto** in Java termina quando non c'è più alcun riferimento valido all'oggetto
- Un oggetto "orfano" viene deallocated in modo da ottimizzare l'uso della memoria
- La deallocazione avviene automaticamente ad opera del garbage collector, quindi la distruzione degli oggetti non è responsabilità del programmatore



# In sintesi: Ruoli di una classe

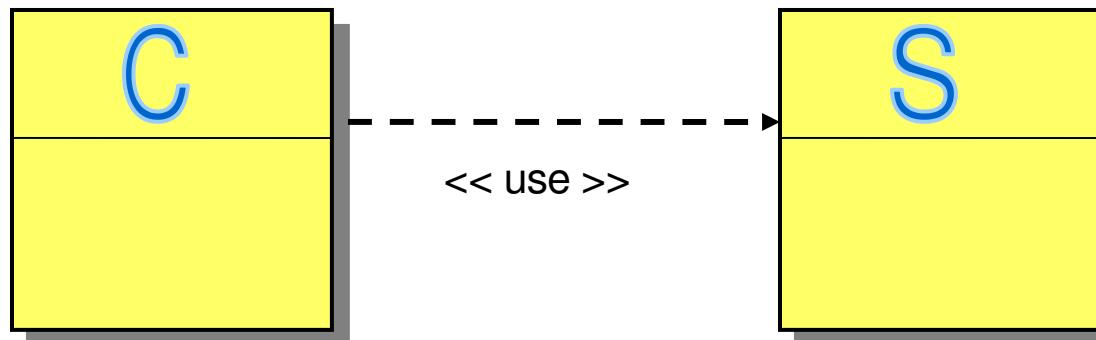
Una Classe può giocare un duplice ruolo:

**Ruolo "Utente" o "Cliente"**

cioè può utilizzare le risorse messe a disposizione da altre Classi.

**Ruolo "Servente"**

cioè essere usata da altre Classi o da un Programma Utente.

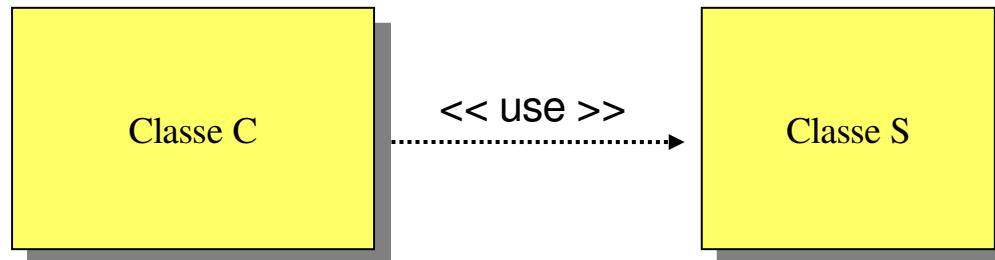


# In sintesi: uso di una classe da parte di un modulo (classe) utente (1/2)



Affinché una classe (C) possa utilizzare un'altra classe (S) *in generale* deve:

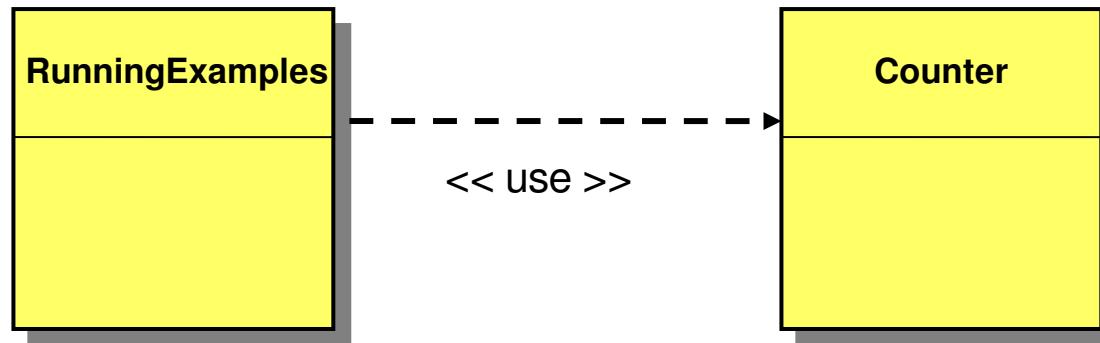
- ◆ importare il nome della classe S se necessario
- ◆ dichiarare (almeno) un riferimento alla classe S
- ◆ creare una istanza della classe S
- ◆ inizializzare il riferimento in modo che punti all'istanza creata





# In sintesi: uso di una classe da parte di un modulo (classe) utente (2/2)

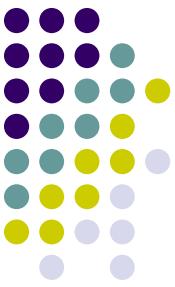
- Nel caso del nostro primo esempio:
  - RunningExamples gioca il ruolo della classe utente
  - Counter gioca il ruolo della classe servente
  - Non è stato necessario utilizzare esplicitamente la dichiarazione import perché entrambe le classi appartengono allo stesso package
  - La classe utente ha creato 3 diverse istanze (oggetti) della classe Counter e tre riferimenti (b,c,d), uno per ciascun oggetto creato inizializzandoli perché puntassero agli oggetti





# In sintesi: caratteristiche di una classe

- La classe consente di definire un nuovo tipo di dato, crearne uno o più esemplari (oggetti), inizializzarli ed applicare loro i metodi di elaborazione disponibili.
- La struttura dati e gli algoritmi sono tenuti nascosti all'interno del modulo che implementa la classe
- Lo stato dell'oggetto, cioè i valori correnti delle variabili di istanza che lo costituiscono, è encapsulato ed evolve unicamente in relazione ai metodi applicati all'oggetto.
- **L'interfaccia (pubblica) della classe è un «contratto» stipulato con l'utente, pertanto deve essere:**
  - Stabile (I prototipi dei metodi non cambiano)
  - Minimale (offro all'esterno l'insieme minimo di metodi)
  - Completa (ci sono tutte i metodi che possono servire)



# Riferimenti