



Programmazione orientata agli oggetti Il modello di Java

Caratteristiche di Java

- Java è generalmente considerato un linguaggio orientato agli oggetti “puro”, aderente quindi ai concetti esposti in precedenza
- Tuttavia il suo modello si discosta per alcuni aspetti dalla visione “classica”
- Si tratta perlopiù di estensioni: nel corso del tempo il modello OOP si è arricchito di concetti nuovi e spesso molto importanti
- C'è anche qualche compromesso legato a motivi di efficienza
- La sintassi di Java è derivata da quella del C, la più diffusa, con alcune differenze significative che evidenzieremo nell'esposizione del linguaggio

Tipi primitivi

- Nel modello classico un linguaggio ad oggetti comprende solo oggetti e classi
- In Java, principalmente per motivi di efficienza, esistono anche i tipi primitivi (o predefiniti) che hanno un ruolo simile a quello che hanno in C
- Sono cioè strutture dati slegate da qualunque comportamento
- La definizione dei tipi primitivi è però più precisa e non ci sono ambiguità:
 - Non esistono tipi la cui dimensione dipende dalla piattaforma (come int in C)
 - Interi, caratteri e booleani sono tipi ben distinti fra loro

Tipi primitivi: caratteri

- I caratteri in Java seguono lo standard UNICODE e hanno una rappresentazione su 2 byte
- La codifica UNICODE è uno standard internazionale che consente di rappresentare anche gli alfabeti non latini e i simboli di scritture non alfabetiche (cinese)
- La codifica UNICODE coincide con quella ASCII per i primi 127 caratteri e con ANSI / ASCII per primi 255 caratteri
- Le costanti char oltre che con la notazione 'A' possono essere rappresentate anche in forma '\u2122' (carattere UNICODE con il codice 2122)
- Non esiste alcuna identità fra numeri interi e caratteri: sono due tipi completamente diversi fra di loro

Tipi primitivi: interi

- Tutti i tipi interi in Java sono con segno: non esistono tipi senza segno
- byte (1 byte) -128...+127
- short (2 byte) -32768...+32767
- int (4 byte) -2.147.483.648 ...2.147.483.647
- long (8 byte) $-9 \times 10^{18} \dots +9 \times 10^{18}$
- NB: le costanti long terminano con la lettera L.
Per esempio 345L

Tipi primitivi: reali

- I reali in Java si basano sullo standard IEEE-754, adottato praticamente da tutti i linguaggi di programmazione.
- Abbiamo due tipi a virgola mobile:
 - float (4 byte) $- 10^{45} \dots + 10^{38}$
(6-7 cifre significative)
 - double (8 byte) $- 10^{328} \dots + 10^{308}$
(14-15 cifre significative)

Tipi primitivi: booleani

- I booleani costituiscono un tipo autonomo totalmente separato dagli interi: non si convertono boolean in interi e viceversa.
- `boolean` (1 bit) `false` e `true`
- La conseguenza più importante è che tutte le espressioni relazionali e logiche danno come risultato un `boolean`, non più un `int`
- `false` non vale 0 e `true` non vale 1

Variabili e costanti

- In Java le variabili in Java vengono dichiarate come in C e possono essere inizializzate:

```
int n,m;  
float f = 5;  
boolean b = false;
```

- Anche il loro uso è lo stesso:

```
n = 4;  
m = n * 3 + 5;  
b = m > 6;
```

- L'unica differenza rispetto al C è che possono essere definite ovunque, non solo all'inizio di un blocco
- Esiste anche la possibilità di dichiarare delle costanti anteponendo alla dichiarazione la parola chiave `final`

```
final int n = 8;  
final boolean b = false;
```

Ancora su Java e C

- Per quanto riguarda le strutture di controllo: if, while, for, switch, la sintassi è esattamente la stessa del C
- Anche gli operatori e le loro regole di priorità sono le stesse
- L'unica cosa da ricordare bene è che in Java non c'è identità fra booleani e interi e quindi
 - Gli operatori di relazione (< > == e !=) danno come risultato un boolean
 - Le strutture di controllo richiedono espressioni con risultati di tipo boolean
- Un'altra piccola differenza è la possibilità di usare la coppia di caratteri // per i commenti di una sola riga
- `a = 5; /* commento classico in stile C */`
`a = 5; // commento monoriga (arriva fino a fine riga)`

Classi in Java: metodi statici

- Nel modello “classico” le classi hanno due funzioni:
 - Definire una struttura e un comportamento
 - Creare gli oggetti (istanze)
- Sono quindi delle matrici, degli “stampini”, che consentono di creare istanze fatte nello stesso modo ma con identità distinte.
- Svolgono nel contempo il ruolo di tipo e di strumenti di costruzione
- In Java le classi hanno anche un'altra capacità: possono fornire servizi indipendenti dalla creazione di istanze
- E' infatti possibile definire dei metodi (individuati dalla parola chiave static) che possono essere invocati anche se non esiste alcuna istanza

Classi in Java: visibilità

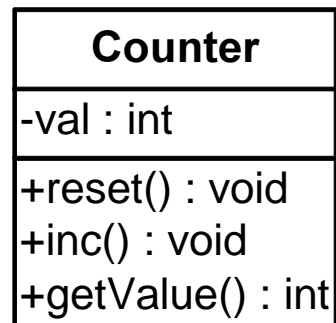
- Nel modello “classico”, in virtù dell’ incapsulamento abbiamo i seguenti comportamenti:
 - Tutti i campi (variabili) sono invisibili all’esterno
 - Tutti i metodi sono visibili all’esterno
- Java introduce un meccanismo molto più flessibile: è possibile stabilire il livello di visibilità di ogni metodo e di ogni variabile usando le parole chiave private e public.
- Questa estensione consente di avere due comportamenti nuovi:
 - Metodi non visibili all’esterno (privati): molto utili per nascondere dettagli implementativi e migliorare l’incapsulamento
 - Variabili visibili all’esterno (pubbliche): pericolose e da evitare perché “rompono” l’incapsulamento

Esempio di classe: Counter - Descrizione

- Proviamo a costruire una semplice classe che rappresenta un contatore monodirezionale
- Come accade sempre nella programmazione ad oggetti si parte definendo il comportamento
- La nostra classe si chiamerà Counter (per convenzione i nomi di classe sono sempre in maiuscolo) e sarà in grado di:
 - Azzerare il valore del contatore: reset()
 - Incrementare il valore: inc()
 - Restituire il valore corrente: getValue()
- A livello implementativo useremo una variabile intera (val) per memorizzare il valore corrente del contatore
- In termini OOP: la classe Counter avrà uno stato costituito dalla variabile val e un comportamento definito dai tre metodi: reset(), inc() e getValue()

Esempio di classe: Counter - Modello

- Vediamo la rappresentazione UML (modello) della nostra classe
- I caratteri + e – davanti ai metodi e ai campi indicano la visibilità:
 - - sta per private
 - + sta per public



Esempio di classe: Counter - Implementazione

```
public class Counter
{
    private int val;
    public void reset()
    { val = 0; }
    public void inc()
    { val++; }
    public int getValue()
    { return val; }
}
```

- Anche le classi hanno una visibilità, nel nostro caso public
- Il campo val è definito correttamente come privato, mentre i metodi sono pubblici
- La sintassi per la definizione dei metodi è uguale a quella del C e lo stesso vale per la dichiarazione della variabile val.
- La variabile val può essere liberamente utilizzata dai metodi della classe

Classi di sistema

- In C esiste una collezione di funzioni standard messe a disposizione dal sistema che prende il nome di libreria di sistema
- In Java, come nella maggior parte dei linguaggi ad oggetti, esiste invece una collezione di classi di sistema
- Abitualmente ci si riferisce all'insieme delle classi di sistema con il nome di framework
- Il framework di Java è molto ricco e comprende centinaia di classi che possono essere utilizzate per scrivere le proprie applicazioni
- Praticamente ogni applicazione Java fa uso di una o più classi di sistema

Programmi in Java

- Un programma Java è costituito da un insieme di classi
- Per convenzione deve esistere almeno una classe, definita come pubblica, che implementi un metodo static (cioè un metodo che può essere invocato anche in assenza di istanze) chiamato main()
- Questo metodo ha lo stesso ruolo della funzione main nei programmi C
- Il fatto che il metodo sia static consente al sistema di invocarlo alla partenza del programma, quando non esiste ancora nessuna istanza.
- Il più semplice programma Java è costituito quindi da una sola classe con le caratteristiche sopra descritte

Esempio 1 Hello world!

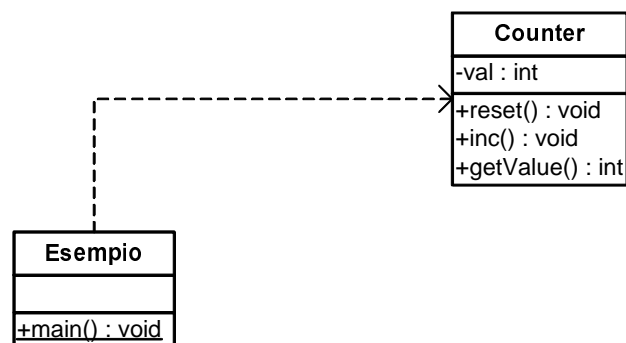
- Scriviamo il “mitico” programma Hello World in Java

```
public class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello world!");
    }
}
```

- Per il momento ignoriamo il parametro di main() (vedremo più avanti il suo significato)
- System.out.println() è un metodo messo a disposizione da una delle classi di sistema (System.out) e consente di scrivere sul video.

Esempio 2: descrizione e modello

- Proviamo a scrivere un’applicazione più complessa che fa uso di due classi: la classe che definisce il metodo main() e la classe Counter definita in precedenza.
- Nel metodo main() creeremo un’istanza di Counter e invocheremo alcuni metodi su di essa
- Il diagramma sottostante rappresenta la struttura dell’applicazione (la linea di collegamento ci dice che Esempio “usa” Counter)



Esempio 2: implementazione

```
public class Counter
{
    private int val;
    public void reset() { val = 0; }
    public void inc(){ val++; }
    public int getValue() { return val;}
}

...

public class Esempio
{
    public static void main(String[] args)
    {
        int n;
        Counter c1;
        c1 = new Counter();
        c1.inc();
        n = c1.getValue();
        System.out.println(n);
    }
}
```

Passo 1: dichiarazione del riferimento

- Java, come tutti i linguaggi ad oggetti, consente di dichiarare variabili che hanno come tipo una classe
`Counter c1;`
- Queste variabili sono riferimenti ad oggetti (in qualche modo sono dei puntatori).

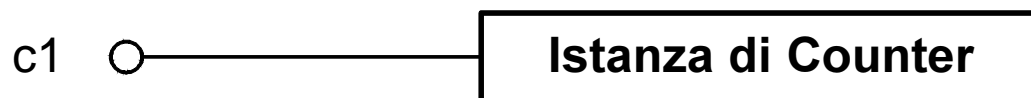
💣 Attenzione!

- La dichiarazione della variabile non implica la creazione dell'oggetto:
- la variabile `c1` è a questo punto è solo un riferimento vuoto (un puntatore che non punta da nessuna parte)

`c1` ○ —————

Passo 2: creazione dell'oggetto

- Per creare l'oggetto devo utilizzare un'istruzione apposita che fa uso della parola chiave `new`
`c1 = new Counter();`
- A questo punto, e non prima, ho a disposizione un oggetto (un'istanza di `Counter`) e `c1` è un riferimento a questa istanza



Passo 3: uso dell'oggetto

- A questo punto abbiamo un oggetto ed un riferimento a questo oggetto (la variabile `c1`)
- Possiamo utilizzare il riferimento per invocare i metodi pubblici dell'oggetto utilizzando la cosiddetta "notazione puntata":

`<nome variabile>.<nome metodo>`

- Per esempio:
`c1.inc();`
`n = c1.getValue();`
- Dal momento che `c1` è di "tipo" `Counter`, il compilatore, basandosi sulla dichiarazione della classe `Counter` può determinare quali sono i metodi invocabili (quelli dichiarati come pubblici)

Passo 4: distruzione dell'oggetto

- Non occorre distruggere manualmente l'oggetto, in Java esiste un componente del sistema, chiamato garbage collector che distrugge automaticamente gli oggetti quando non servono più
- Come fa il garbage collector a capire quando un oggetto non serve più? Un oggetto non serve più quando non esistono più riferimenti ad esso
- `c1` è una variabile locale del metodo `main()`: quando il metodo `main()` termina `c1` non esiste più
- Quindi non esistono più riferimenti all'oggetto che abbiamo creato e il garbage collector può distruggerlo



Esempio 3 - Interazione con l'utente

- Quasi tutti i programmi hanno bisogno di interagire con l'utente
- Nel caso più semplice (applicazioni non grafiche) l'interazione avviene attraverso il video e la tastiera
- Comprendere in modo completo come Java gestisce la comunicazione con il video e soprattutto con la tastiera richiede concetti avanzati che vedremo più avanti
- Per il momento ci accontentiamo di imparare una "ricetta pronta" per gestire queste problematiche

Esempio 3: Il programma EsempioInt

- Prendiamo un altro semplice esempio basato sulla classe Counter
- Il nostro programma crea un'istanza di Counter e consente all'utente di agire su questo contatore digitando comandi da tastiera
- Ad ogni comando il programma mostra a video il nuovo valore assunto dal contatore e richiede un altro comando
- Premendo + il contatore viene incrementato
- Premendo a il contatore viene azzerato
- Premendo * il programma termina
- * **Attenzione:** per trasmettere i comandi al programma bisogna premere il tasto Invio

Esempio 3 - Prima parte

- In rosso sono evidenziate le parti “misteriose” da prendere così come sono

```
import java.io.*;

public class EsempioInt
{
    public static void main(String[] args)
        throws IOException
    {
        // Creiamo un'oggetto tastiera facile da usare --
        BufferedReader kbd =
            new BufferedReader(new InputStreamReader(System.in));

        Counter c = new Counter();
        String st = "";
        ...
    }
}
```

Esempio 3 - Seconda parte

- Usando un ciclo while il programma esce solo quando diamo il comando *

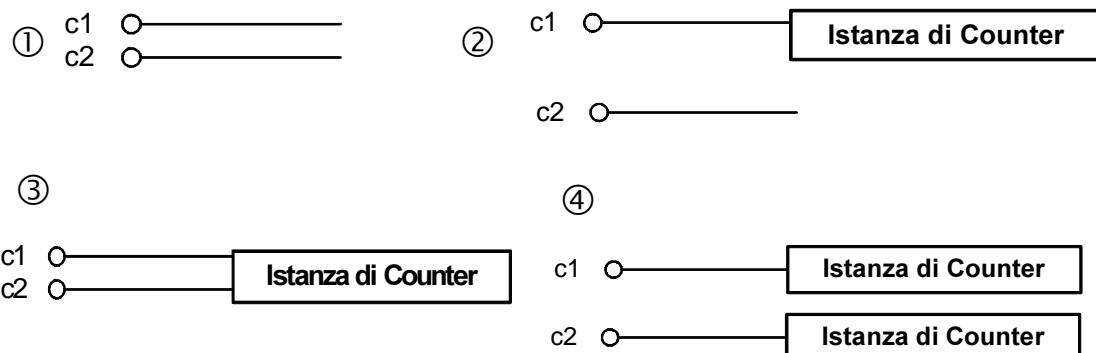
```
...
while (!st.equals("*"))
{
    System.out.println("Valore contatore: "+c.getValue());
    System.out.println("+ per incrementare");
    System.out.println("a per azzerare");
    System.out.println("* per terminare");
    st = kbd.readLine();
    if (st.equals("+"))
        c.inc();
    else if (st.equals("a"))
        c.reset();
}
}
```

Riferimenti e oggetti

- Consideramo questa serie di istruzioni:

1. Counter c1,c2;
2. c1 = new Counter();
3. c2 = c1;
4. c2 = new Counter();

- Ecco l'effetto su variabili e gestione della memoria:



💣 **Attenzione!**

- L'assegnamento `c2=c1` non crea una copia dell'oggetto!
- Si hanno invece due riferimenti allo stesso oggetto!
- Questo perché le variabili che hanno come tipo una classe sono riferimenti agli oggetti: non contengono un valore ma l'indirizzo dell'oggetto
- Quindi un assegnamento copia l'indirizzo
- L'effetto è che abbiamo due variabili che contengono lo stesso indirizzo e quindi "puntano" allo stesso oggetto

- Ma allora come si fa a copiare un oggetto?
- Innanzitutto aggiungiamo alla classe un metodo che copia esplicitamente lo stato (nel nostro caso `val`)

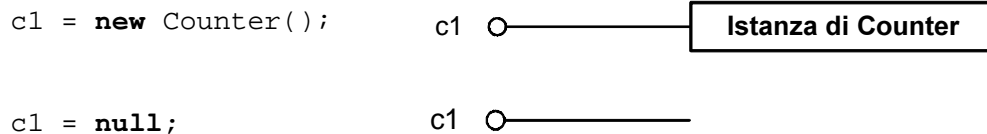
```
public class Counter
{
    private int val;
    public void reset() { val = 0; }
    public void inc(){ val++; }
    public int getValue() { return val;}
    public void copy(Counter x) { val = x.val;}
}
```

- Poi si procede così:

```
Counter c1, c2; /* Dichiariamo due variabili */
c1 = new Counter(); /* creiamo due istanze */
c2 = new Counter();
c2.copy(c1); /* copiamo lo stato di c1 in c2 */
```

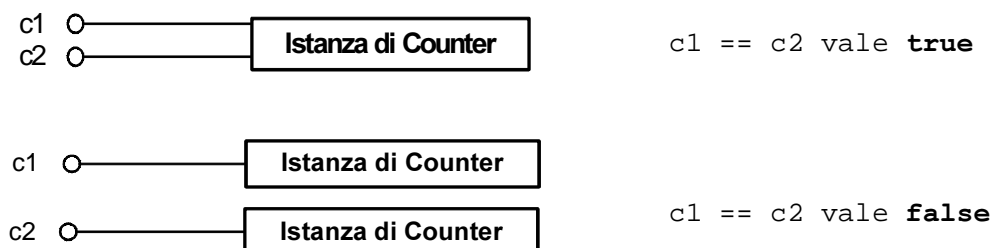
Riferimenti e oggetti – Valore null

- A differenza dei puntatori del C non è possibile fare “pasticci” con i riferimenti agli oggetti
- Con un riferimento possiamo fare solo 4 cose:
 1. Dichiararlo: `Counter c1, c2;`
 2. Assegnargli un oggetto appena creato:
`c1 = new Counter();`
 3. Assegnargli un altro riferimento: `c2 = c1;`
 4. Assegnargli il valore null: `c1 = null;`
- Assegnando il valore null (è una parola chiave di Java) il puntatore non punta più da nessuna parte



Riferimenti e oggetti – Uguaglianza fra riferimenti

- Che significato può avere un'espressione come:
`c1 == c2`
- `c1` e `c2` sono due riferimenti e quindi sono uguali se l'indirizzo che contengono è uguale
- Due riferimenti sono uguali se puntano allo stesso oggetto e non se gli oggetti a cui puntano sono uguali
- Quindi:



Riferimenti e oggetti – Uguaglianza fra oggetti

- Come si fa se invece si vuole verificare se due oggetti sono uguali, cioè hanno lo stesso stato?
- Si fa come per la copia: si aggiunge a Counter il metodo equals()

```
public class Counter
{
    private int val;
    public void reset() { val = 0; }
    public void inc(){ val++; }
    public int getValue() { return val; }
    public void copy(Counter x) { val = x.val };
    public boolean equals(Counter x){return val == x.val;}
}
```

- Poi si procede così:

```
Counter c1, c2; boolean b1,b2;
c1 = new Counter(); c1.inc();
c2 = new Counter();
b1 = c1.equals(c2); /* b1 vale false */
c1.copy(c2);
b2 = c1.equals(c2); /* b2 vale true */
```

Costruttori

- Riprendiamo in esame la creazione di un'istanza
`c1 = new Counter();`
- Cosa ci fanno le parentesi dopo il nome della classe?
Sembra quasi che stiamo invocando una funzione!
- In effetti è proprio così: ogni classe in Java ha almeno un metodo costruttore, che ha lo stesso nome della classe
- Il compito del costruttore è quello inizializzare la nuova istanza: assegnare un valore iniziale alle variabili, creare altri oggetti ecc.
- Quando usiamo l'operatore `new`, il sistema crea una nuova istanza e invoca su di essa il costruttore
- Un costruttore è un metodo che viene chiamato automaticamente dal sistema ogni volta che si crea un nuovo oggetto

Caratteristiche dei costruttori in Java

- Un costruttore ha lo stesso nome della classe
- Non ha tipo di ritorno, nemmeno void
- Ha una visibilità, comunemente public
- Vediamo la definizione del classe Counter con la definizione del costruttore:

```
public class Counter
{
    private int val;
    public Counter() { val = 0; }
    public void reset() { val = 0; }
    public void inc(){ val++; }
    public int getValue() { return val;}
}
```

Costruttori multipli - 1

- Java ammette l'esistenza di più costruttori che hanno lo stesso nome (quello della classe) ma si differenziano per il numero e il tipo dei parametri
- I costruttori con i parametri permettono di inizializzare un'istanza con valori passati dall'esterno

```
public class Counter
{
    private int val;
    public Counter() { val = 0; }
    public Counter(int n) { val = n; }
    public void reset() { val = 0; }
    public void inc(){ val++; }
    public int getValue() { return val;}
}
```

- In questo caso abbiamo la possibilità di creare un contatore con un valore iniziale prefissato

Costruttori multipli - 2

- **Vediamo come si può operare in presenza di più di un costruttore:**

```
Counter c1, c2;  
c1 = new Counter();  
c2 = new Counter(5);
```

- **Nel primo caso viene invocato il costruttore senza parametri: il valore del contatore viene inizializzato con il valore 0**
- **Nel secondo caso viene invocato il costruttore che prevede un parametro e il valore iniziale del contatore è quindi 5**
- **Il costruttore privo di parametri viene chiamato costruttore di default**
- **Ogni classe ha un costruttore di default: se non viene definito esplicitamente il sistema ne crea automaticamente uno vuoto**

Overloading - 1

- **Nel caso dei costruttori abbiamo visto che Java consente di avere due metodi con lo stesso nome ma con un numero parametri diverso o con tipi diversi**
- **Questa caratteristica non è limitata ai costruttori ma è comune a tutti i metodi e si chiama overloading (sovraccarico)**
- **Due metodi con lo stesso nome ma parametri diversi di dicono overloaded**
- **Attenzione: non è sufficiente che sia diverso il valore di ritorno, deve essere diverso almeno uno dei parametri**

Overloading - 2

- **Riprendiamo la nostra classe Counter e definiamo una seconda versione del metodo inc():**

```
public class Counter
{
    private int val;
    public Counter() { val = 0; }
    public void reset() { val = 0; }
    public void inc(){ val++; }
    public void inc(int n){ val = val + n; }
    public int getValue() { return val; }
}
```

- **Vediamo un esempio di uso:**

```
Counter c1;
c1 = new Counter();
c1.inc(); /* Viene invocata la prima versione */
c1.inc(3); /* Viene invocata la seconda versione */
```

Passaggio dei parametri - 1

- **Come il C, Java passa i parametri per valore: all'interno di un metodo si lavora su una copia**
- **Finché parliamo di tipi primitivi non ci sono particolarità da notare**
- **Per quanto riguarda invece i riferimenti agli oggetti la cosa richiede un po' di attenzione:**
 - **Passando un riferimento come parametro questo viene ricopiato**
 - **Ma la copia punta all'oggetto originale!**
- **In poche parole: passare per valore un riferimento significa passare per riferimento l'oggetto puntato!**

Passaggio dei parametri - 2

- **Quindi:**
 - **Un parametro di tipo primitivo viene copiato, e la funzione riceve la copia**
 - **Un riferimento viene copiato, la funzione riceve la copia, ma con ciò accede all'oggetto originale!**
- **Ovvero:**
 - **I tipi primitivi vengono passati sempre per valore**
 - **Gli oggetti vengono passati sempre per riferimento**