Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria

CORSO DI

RETI DI CALCOLATORI
Linguaggio Java: Introduzione e Concetto di Classe

Prof. Franco Zambonelli

Lucidi realizzati in collaborazione con
Ing. Enrico Denti - Univ. Bologna

Anno Accademico 2001-2002

IL CONCETTO DI CLASSE

Una classe descrive le proprietà di un insieme di “oggetti” aventi:

- la medesima *struttura interna*
- lo stesso *protocollo di accesso* (insieme di operazioni)
- lo stesso *comportamento*

Riunisce le proprietà di:

- ***componente software***
  → può essere dotato di suoi propri *dati* e *operazioni*
- ***tipi di dato astratto (ADT)***
  → funge da “stampo” per *creare* nuove *istanze* di oggetti
- ***modulo e centro di servizi***
  → riunisce dati e relative operazioni (servizi)
  → fornisce meccanismi di protezione

I LINGUAGGI AD OGGETTI

Forniscono come astrazione base la possibilità di definire classi, di mettere in relazione le classi tramite ereditarietà, di istanziare oggetti a partire dalla classe.

- Linguaggi ad Oggetti "puri": la classe e gli oggetti sono le uniche, o quasi, astrazioni del linguaggio (Smalltalk, Java)

- Linguaggi ad Oggetti "non puri": aggiungono l'astrazione di classe e oggetto ai costrutti tradizionali della programmazione (C++, Object Pascal, OO COBOL)

IL LINGUAGGIO JAVA

- È un linguaggio a oggetti puro: tranne i tipi primitivi di base (`int`, `float`, …), tutto il resto in una applicazione sono oggetto appartenenti a classe
- È fortemente ispirato al C++, ma riprogettato *senza il requisito della piena compatibilità col C* (a cui però assomiglia…)
- Un programma è un *insieme di classi:* non esistono funzioni definite (come in C) a livello esterno: **anche il `main` è definito dentro a una classe!**

JAVA: LINGUAGGIO O ARCHITETTURA?

A differenza del C++, Java viene fornito con una notevole gerarchia di classi standard già pronte

E' un'architettura già pronta per l'uso!

- **indipendente dalla piattaforma**
- Package grafico (indipendente dalla piattaforma!): AWT
- Programmazione a eventi (molto evoluta!)
- Supporto di rete: URL, connessioni via Socket, …
- Supporto per il multi-threading
- Concetto di APPLET = piccola (?) applicazione da eseguirsi dentro un browser Internet
- Supporto per sicurezza
- Modello di sicurezza "sandbox" per le applet scaricate via rete
- Cifratura, crittografia, chiavi pubbliche/private…
- Connessione e interfacciamento con database

JAVA: STORIA

- Nasce nel 1994 da un progetto (fallito!) per la **network television**;
- Si diffonde soprattutto grazie alle applet, piccole (originariamente) applicazioni eseguite dentro ad un browser e scaricate dinamicamente dal WWW server
- Si afferma negli ultimi anni come linguaggio generale per la programmazione

Molto orientato al "**Network Computing**":

- Interazione con oggetti remoti (RMI)
- Interazione con Basi di Dati distribuite (JDBC)
- Interazione con sistemi CORBA
- Forte interconnessione con WWW (Applet, CGI, JavaScript).

Futuro promettente (si ritorna alle origini del progetto) per **sistemi embedded**:

- Java Card
- Java Rings
- Java Operating Systems per PDA

Portabilitá ottenuta attraverso una via intermedia tra interpretazione e compilazione

- I programmi (le classi) Java sono compilate in un linguaggio macchina astratto (*Java bytecode*)

IL COSTRUTTO "`class`" IN JAVA

Definizione di una classe:

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

Si potranno poi istanziare oggetti delle classe Counter

```
Counter c1, c2;
```

E chiedere loro servizi:

```
c1.inc();
if (c2.getValue() == 3) {……};
```

- ***Dati*** (`val`) e ***operazioni*** (`reset`, `inc`, `dec`, `rd`) s**ono *riuniti in un unico costrutto linguistico***
- **Il campo dati `val` è <u>privato</u>**, e può essere acceduto *solo dalle operazioni (metodi) della classe* (`reset`, `inc`, `dec`, `rd`): ogni tentativo di acceso dall'esterno verrà individuato e respinto dal compilatore!

  ```
  c2.val = 3; /* ERRORE */
  ```

- **I metodi `reset()`**, **`getValue()`**, etc. **sono <u>pubblici</u>**, e possono quindi essere invocati *da chiunque*
  → dall'esterno, `val` è manipolabile solo tramite questi metodi
  → si garantisce l'*incapsulamento.*
  → modo di invocazione **`oggetto.metodo`**

In genere: tutti i **dati** devono avere **visibilitá privata** e i **metodi** (ma non necessariamente tutti) devono avere **visibilitá pubblica**.

Programmi in JAVA

- A parte un piccolo insieme di tipi primitivi predefiniti, un programma Java è definito da un insieme di classi e, durante l'esecuzione, dagli oggetti che da tali classi verranno istanziati
  - Le classi sono **entitá statiche**, che sono definite prima della esecuzione. Non esiste nulla che non sia definito all'interno di qualche classe;
  - Gli oggetti sono **entitá dinamiche** che assumono esistenza a partire dalle classi durante l'esecuzione.
- Ci deve essere una sola classe funzione pubblica di nome main, all'interno di una classe con la seguente interfaccia:

```
public static void main(String args[])
```

- Si dichiara la lista degli argomenti anche se non ce ne sono
- Il main deve ***necessariamente*** essere definito dentro a una classe **pubblica e ha *obbligatoriamente* la forma**

```
public static void main(String args[]){...}
```

- A differenza del C, non c'è valore di ritorno, mentre gli argomenti dalla linea di comando sono oggetti `String` (il primo *non* è il nome del programma come in C)

CONVENZIONI:

- I nomi delle classi sono per convenzione maiuscoli
- I nomi di metodi, variabili e oggetti per convenzione sono minuscoli

NOTA: Una funzione senza parametri viene definita senza la parola void nella lista dei parametri. Esempio:

```
public  void inc()   { val++; }
```

Hello World (I)

```
/** File Esempio0.java
* Applicazione Java da linea di comando
* che stampa la classica frase di benvenuto
@author Enrico Denti
@version 1.0, 5/4/98
*/

public class Esempio0 {

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

Esiste una **corrispondenza ben precisa** fra *nomi delle classi* e *nomi di file*:

- in un file sorgente ci può essere un'unica classe **pubblica,** che deve avere **lo stesso nome del file** (case-sensitive!)

Ogni file sorgente importa automaticamente la libreria fondamentale `java.lang`:

- `System` è una classe di sistema, che rappresenta "il sistema sottostante", qualunque esso sia
- `out` è un oggetto (static) della classe `System`, e rappresenta il dispositivo standard di uscita
- sull'oggetto `out` è possibile invocare il metodo `println()`

Hello World (II)

```
/** File Esempio0.java
* Applicazione Java da linea di comando
* che stampa la classica frase di benvenuto
@author Enrico Denti
@version 1.0, 5/4/98
*/
// questo e' commento su una sola linea stile C++
/* questo e' il commento stile C normale */
public class Esempio0 {
    public static void main(String args[]){
        System.out.println("Hello World!");
    }
}
```

- Per compilare:

  `javac Esempio0.java` *(produce `Esempio0.class`)*

- Per eseguire:

  `java Esempio0`

- Il formato compilato .class (*bytecode Java*) è ***portabile*** e ***indipendente dalla piattaforma***
- una classe compilata su un sistema funzionerà *su qualunque altro sistema* (Mac, Unix, Windows…) *per cui sia disponibile il Java Runtime Environment* (nella stessa versione)
- il formato compilato di Java (**bytecode**) viene interpretato *dalla Java Virtual Machine* tramite l'interprete Java
- Sun rende di pubblico dominio il compilatore che si lancia **javac** e la macchina che si lancia con il comando **java**. *NON FORNISCE AMBIENTI DI SVILUPPO GRAFICI INTEGRATI (BISOGNA COMPRARLI!)*

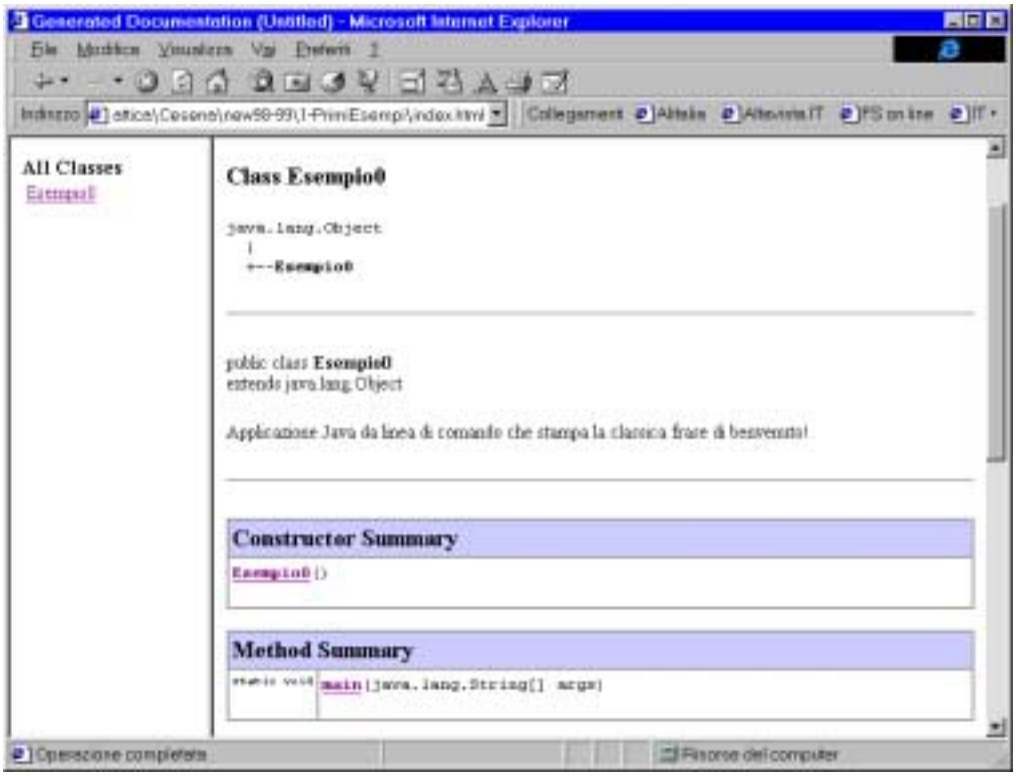
Hello World (III)

```
/** File Esempio0.java
* Applicazione Java da linea di comando
* che stampa la classica frase di benvenuto
@author Enrico Denti
@version 1.0, 5/4/98  */
/* i commenti che iniziano con /** servono per
generare i file di documentazione */
public class Esempio0 {
    public static void main(String args[]){
        System.out.println("Hello World!");
}}
```

Java consente di generare automaticamente la documentazione delle classi contenute nel file (limitatamente a quelle non private):

```
javadoc Esempio0.java
```

Il risultato è un insieme di file HTML (il formato esatto dipende dalla versione)



GLI ARGOMENTI

```
/** File Esempio1.java
* applicazione da linea di comando
* stampa gli argomenti passati al programma
@author Enrico Denti
@version 1.0, 9/11/97
*/

public class Esempio1 {

  public static void main(String args[]){
    if (args.length == 0)
      System.out.println("Occorrono argomenti");
    else
      for(int i=0; i<args.length; i++)
        System.out.println("arg" + (i+1) +
                           ": " + args[i]);
  }
}
```

- `String` è una classe predefinita: le stringhe Java sono oggetti, non pezzi di memoria assegnata a un puntatore come in C
- L'operatore `+` è predefinito sulle stringhe, e serve a concatenarle
- args è un vettore (di stringhe): come tutti i vettori Java, è un oggetto, il cui campo (pubblico) **`length`** indica il numero di elementi (numerati da 0 a `length`-1)

UNA RIFLESSIONE

Finora, gli unici componenti del nostro sistema sono classi, cioè oggetti definiti staticamente, la cui esistenza è nota a priori. I sistemi reali richiedono però una maggiore dinamicità.

Creazione e uso di oggetti Counter

```
/** File Esempio2.java
  * Prima applicazione che istanzia nuovi oggetti
  @author Enrico Denti
  @version 1.0, 5/4/98 */

class Counter {
    private int val;
    public void reset() { val = 0; }
    public void inc()   { val++; }
    public void dec()   { val--; }
    public int  rd()    { return val; }
}
public class Esempio2 { // contiene solo il main
  public static void main(String args[]){
    Counter c1, c2;
    c1 = new Counter(); c2 = new Counter();
    c1.reset(); c2.reset();
    c1.inc(); c1.inc(); c2.inc();
    System.out.println("c1 vale " + c1.rd());
    System.out.println("c2 vale " + c2.rd());
}
```

- Compilando questo file, si ottengono *due* file .class, uno per ogni classe definita (Esempio2.class e Counter.class)
- La frase **`Counter c1;`** introduce un ***riferimento*** a `Counter`, cioè una sorta di puntatore che però non è direttamente manipolabile dall'utente e si usa *senza doverlo dereferenziare esplicitamente* come in C o C++ (vedi `*p` o `p->inc()`)
- Il `Counter` vero e proprio viene creato dinamicamente all'atto della `new`, ed è visibile *fino alla fine del blocco* in cui è definito. La deallocazione è automatica e gestita dal **garbage collector** di Java secondo le sue politiche.
- uso delle istanze mediante uno stile a "invio di messaggi"

I COSTRUTTORI

Molti errori nel software sono causati da mancate inizializzazioni di variabili ai valori iniziali previsti. Per ovviare a ciò, molti linguaggi a oggetti definiscono il concetto di **metodo costruttore**, che **automatizza *l'inizializzazione*** delle istanze.

Il costruttore:

- **non viene mai chiamato esplicitamente dall'utente**, ma solo automaticamente dal sistema all'atto della creazione di una nuova istanza. Si possono quindi passare parametri quando di crea un nuovo oggetto, corrispondenti ai parametri del metodo costruttore
- ha un **nome fisso** (uguale al nome dellc classe)
- **non ha tipo di ritorno** (il suo scopo non è calcolare qualcosa, ma inizializzare un oggetto)
- **può non essere unico**. Spesso vi sono più costruttori, con diverse liste di argomenti: il sistema riconosce automaticamente quale invocare sulla base dei parametri passati all'atto della creazione.

Se non ne viene definito nessuno, si genera automaticamente un *costruttore di default* (senza argomenti), che non fa nulla.

ESEMPIO (ridefinizione della classe `counter`)

```
class Counter {
    private int val;
    public Counter()      { val=1; }  // default
    public Counter(int x) { val=x; }
    public void reset() { val = 0; }
    public void inc()   { val++; }
    public void dec()   { val--; }
    public int  rd()    { return val; }
}
```

L'ESEMPIO PRECEDENTE… ristrutturato

```
/** FILE Counter.java
  * Questa classe definisce il concetto di
     contatore avanti/indietro.
  @author Enrico Denti
  @version 1.0, 8/3/99  */
public class Counter {
 private int val;
 public Counter()    { val=1; }  // default
 public Counter(int x) { val=x; }
 public void reset() { val = 0; }
 public void inc()   { val++; }
 public void dec()   { val--; }
 public int  rd()    { return val; }
}
```

```
/** FILE Esempio2bis.java
  * Prima applicazione che istanzia nuovi oggetti
  @author Enrico Denti
  @version 1.0, 8/3/99  */
public class Esempio2bis {
  public static void main(String args[]){
    Counter c1, c2;
    c1 = new Counter(); c2 = new Counter(5);
    c1.inc(); c1.inc(); c2.inc();
    System.out.println("c1 vale " + c1.rd());
    System.out.println("c2 vale " + c2.rd());
}}
```

- La classe `Counter`, resa *pubblica*, è ora in un file separato nello stesso direttorio
- Il main può usare la classe `Counter` **senza dover includere nulla**: Java supporta il ***collegamento dinamico!***
  Al primo uso di `Counter`, il file `Counter.class` verrà cercato e caricato → adatto a *codice mobile* e *applicazioni su Internet*

NON ESISTE IL CONCETTO DI LINKING STATICO!!!!

L'ISTANZA CORRENTE: THIS

La parola chiave `this` costituisce un riferimento all'istanza corrente, e può servire:

- per indicare un campo dati in caso di omonimia con un parametro o con una variabile locale
- in un costruttore, per richiamare un altro costruttore della stessa classe.

ESEMPIO (ridefinizione della classe `Counter`)

```
class Counter {
    private int val;
    public Counter()      { this(1); }
    public Counter(int val) { this.val=val; }
    public void reset() { val = 0; }
    public void inc()   { val++; }
    public void dec()   { val--; }
    public int  rd()    { return val; }
}
```

- Il costruttore `Counter(int val)` utilizza `this` per distinguere il parametro `val` dal campo dati di nome `val` (che viene espresso con la notazione `this.val`)
- Il costruttore `Counter()` utilizza `this(1)` per delegare la costruzione dell'oggetto all'altro costruttore della stessa classe
- NB: la sintassi `this()` per chiamare un altro costruttore puó essere usata *solo come prima istruzione* di un costruttore.

OGGETTI COMPOSTI

Da quanto detto finora dovrebbe risultare chiaro che **un oggetto può contenere (riferimenti a) altri oggetti**

- l'oggetto "contenitore" può ***usarli***...
- ..ma ***non può accedere*** ai loro ***dati privati***
- può però accedere ai dati *pubblici* e ai suoi metodi

In fase di costruzione:

- *prima* si costruisce l'oggetto contenitore
- *poi* si costruiscono, con `new`, gli oggetti interni

ESEMPIO

```
class Counter {
    int val;
    public  Counter()       { val=1; }
    public  void inc()      { val++; }
    public  int  rd()       { return val; }
}

public class DoppioCounter {
// oggetto contenitore

    Counter c;
    // oggetto contenuto

  public DoppioCounter() {c = new Counter();}

  public void inc(){c.inc(); c.inc();}

    public int rd() {return c.rd();}
  }

}
```

TIPI PRIMITIVI, CLASSI, OGGETTI

In Java si distingue fra

- **tipi primitivi**: boolean, char, byte, short, int, long, float, double
- **Classi**
- **riferimenti a oggetti.**

TIPI PRIMITIVI

Tipo	Descrizione	Default	Size	Range
Boolean	`true` o `false`	`False`	1 bit	-
Char	carattere UNICODE	`\u0000`	16 bit	`\u0000` - `\uFFFF`
Byte	intero con segno	0	8 bit	-128 - 127
Short	intero con segno	0	16 bit	-32768 - 32767
Int	intero con segno	0	32 bit	-2^{31} - $2^{31}-1$
Long	intero con segno	0	64 bit	-2^{63} - $2^{63}-1$
Float	IEEE-754 float	0.0	32 bit	-10^{-45} - 10^{38}
Double	IEEE-754 double	0.0	64 bit	-10^{-324} - 10^{308}

- i caratteri UNICODE sono considerati senza segno, e offrono ampio supporto all'internazionalizzazione, e sono ASCII compatibili (sui primi 256);
  valgono inoltre i classici "escape alla C": `'\n'`, `'\''`, `'\t'`
- la nuova notazione `'\u0040'` consente di esprimere un carattere UNICODE tramite 4 cifre esadecimali
- in Java *non sono ammessi* tipi come `long int`, `short int`, etc: i nomi corretti sono semplicemente `long`, `short`, etc.

I tipi primitivi si manipolano come **valori**, con la stessa semantica del C e di altri linguaggi. Quindi, quando si passano alle funzioni vengono passati per ***copia***.

RIFERIMENTI A OGGETTI

In Java, a differenza dei tipi primitivi gli oggetti si manipolano *esclusivamente tramite riferimenti*. Non è possibile definire istanze di oggetti (variabili) allocate staticamente, tutto dev'essere allocato dinamicamente con `new`.

A tutti gli effetti i riferimenti sono come puntatori, indirizzi di memoria, ma *sono dereferenziati automaticamente* (non come in C, ***p**), e non rendono visibile né manipolabile l'indirizzo.

Una frase del tipo c2 = c1 fa puntare i due riferimenti allo stesso oggetto, *non duplica l'oggetto stesso!*

ESEMPIO

```
/**
* verifica il concetto di Riferimento a oggetti
@author Enrico Denti
@version 1.0, 5/4/98
*/
class Point {public int x; public int y;  …}

class Esempio3 { // contiene solo il main
 public static void main(String args[]){
   Point p1 = new Point(100,10), p2 = p1;
   System.out.println("p1: " + p1.x + "," + p1.y);
   System.out.println("p2: " + p2.x + "," + p2.y);
   p1.x = 400;
   System.out.println("p1: " + p1.x + "," + p1.y);
   System.out.println("p2: " + p2.x + "," + p2.y);
 }
}
```

- Ogni modifica a `p1` si ritrova anche in `p2`, a testimonianza che i due sono solo riferimenti che referenziano lo stesso oggetto.

RIFERIMENTI A OGGETTI e PASSAGGIO PARAMETRI

I parametri passati nelle invocazioni di metodi sono passati per *copia*. Non esiste il passaggio di parametri per riferimento.
QUINDI: chiamando un metodo a cui passiamo un riferimento ad oggetti, stiamo passando in verità la copia di un riferimento.

ESEMPIO

```
// funzione di una qualche classe…
void swap (Point p1, Point p2)
{ Point pTemp;

   pTemp = p1;
   p1 = p2;
   p2 = pTemp;
}
```

- Questa funzione non fa assolutamente nel chiamante, perché abbiamo semplicemente fatto lo *scambio sulle copie dei riferimenti*, ma i riferimenti originali rimangono invariati.

ESEMPIO 2

```
// funzione di una qualche classe…
void swap_correct (Point p1, Point p2)
{ Point pTemp;
   pTemp.x = p1.x; pTemp.y = p1.y
   p1.x = p2.x; p1.y = p2.y
   p2.x = pTemp.x; p2.y = pTemp.y;
}
```

- Qui abbiamo modificato i valori degli oggetti riferiti, che sono gli stessi del chiamante, anche se i riferimenti sono una copia. Il chiamante VEDE le modifiche.

CLONAZIONE DI OGGETTI

Per duplicare un oggetto occorre definire un metodo specifico `clone()`, che provveda a duplicare l'oggetto

```
p2 = p1.clone();
```

ESEMPIO

```
class Point {…}

class Esempio3bis {
 public static void main(String args[]){
   Point p1 = new Point(100,10), p2=p1;
   System.out.println("p1: " + p1.x + "," + p1.y);
   System.out.println("p2: " + p2.x + "," + p2.y);
   p2 = (Point) p1.clone();
   p1.x = 400;
   System.out.println("p1: " + p1.x + "," + p1.y);
   System.out.println("p2: " + p2.x + "," + p2.y);
 } }
```

Il metodo clone() sarà qualcosa del genere:

```
public Object clone()
{Point p2 = new Point();
p2.x = x;
p2.y = y;
return p2;}
```

- In questo modo, `p2` rimane inalterato al valore (100,10), mentre `p1` viene modificato in (400,10).
- In veritá il metodo `clone`() è predefinito per tutti gli oggetti, ma è molto complicato da usare ed è troppo facile usarlo in modo scorretto. Meglio ridefinirlo completamente!!
  ATTENZIONE: deve ritornare un **Object** !!!!

UGUAGLIANZA DI OGGETTI

Poiché le variabili Java sono *riferimenti*, **un test di uguaglianza del tipo c1==c2 verifica se due riferimenti puntano allo stesso oggetto**, ***non se due oggetti hanno lo stesso valore!***

Per controllare se due oggetti hanno lo stesso valore di usa il metodo **`equals()`**, pure predefinito per tutte le classi:

ESEMPIO

```
/**
* verifica il concetto di uguaglianza fra oggetti
@author Enrico Denti
*/
class Point {…}

class Esempio4 {
 public static void main(String args[]){
   Point p1 = new Point(100,10), p2=p1;
   System.out.println("p1==p2? ” + (p1==p2));
   System.out.println("p1.equals(p2)? ” +
                          p1.equals(p2));
   p2 = p1.clone();  p1.x = 400;
   System.out.println("p1==p2? ” + (p1==p2));
   System.out.println("p1.equals(p2)? ” +
                          p1.equals(p2));
 } }
```

Risposta:

```
p1==p2? true
p1.equals(p2)? true
p1==p2? False
p1.equals(p2)? false
```

TIPI PRIMITIVI e CLASSI

Spesso è necessario trattare i tipi primitivi come oggetti (ad esempio, per passarli per riferimento a una funzione, ma non solo).

A questo servono le "**wrap classes**", una per ogni tipo primitivo, che in pratica "incapsulano"un tipo primitivo:

Tipo	Classe corrispondente
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

- a parte l'iniziale, la classe ha quasi sempre lo stesso nome del tipo primitivo, tranne `char` / `Character` e `int` / `Integer`
- le classi forniscono metodi per convertire dal tipo primitivo a un oggetto della classe, e viceversa

```
class Esempio6 {
  public static void main(String args[]){
    int x = 35;        // tipo primitivo
    Integer ix = new Integer(x);
    // conversione int → Integer per costruzione
    System.out.println("ix = " + ix);
    // conversione implicita Integer → String
    x = ix.intValue()*4;
    // conversione esplicita Integer → int
    System.out.println("x = " + x);
    // conversione implicita int → String
  }}
```

STRINGHE IN JAVA

In Java, *le stringhe sono oggetti*, istanze della classe `String`: una `String` rappresenta uno specifico valore, ed è quindi un oggetto *non modificabile*.

In Java una `String` non è un buffer modificabile!!

Per immagazzinare stringhe modificabili si usa `StringBuffer`.

Le stringhe possono essere concatenate con **l'operatore +**

La concatenazione di costanti stringa *è fatta a compile-time*, quindi non introduce inefficienze.

- Quando si scrivono costanti stringa fra virgolette (es. `"ciao"`), viene creato automaticamente un oggetto `String` inizializzato a tale valore.
- Una costante stringa non può eccedere la riga: dovendo scrivere costanti più lunghe, è opportuno spezzarle in parti più corte e concatenarle con l'operatore +.

Attenzione:
la selezione di un carattere si fa tramite il metodo `charAt()`, non con l'operatore [] come in C o C++!

```
String s = "Nel mezzo del cammin ";
char ch = s.charAt(3);
```

Esistono decine di operatori predefiniti su `String`: si veda la documentazione per maggiori dettagli.

STRINGHE IN JAVA: LA LIBRERIA

Ecco alcune delle operazioni disponibili per gli oggetti di tipo stringa.

```
char charAt(int index)
```

Ritorna il carattere a quella specifica locazione

```
boolean endsWith(String suffix)
```

Ritorna vero se la stringa termina con quel suffisso

```
boolean beginsWith(String prefix)
```

Ritorna vero se la stringa comincia con quel prefisso

```
boolean equals (Object other)
```

Ritorna vero se la stringa e' uguale a other

```
String substring (int beginindex)
String substring (int beginindex, int endindex)
```

Ritorna la sottostringa compresa tra in carattere di indice beginindex e a) la fine b) il carattere di indice endindex

---

```
int lenght()
```

Ritorna la lunghezza della stringa

```
String toLowerCase()
String toUpperCase()
```

Torna una stringa tutta in minuscolo/maiuscolo

ARRAY IN JAVA

In Java, *gli array sono oggetti*: non esiste un vero nome di classe, in quanto sono identificati dall'operatore **`[]`**.

Definire un array Java significa definire un *riferimento*:
l'array vero e proprio va creato con `new`
(a meno che non sia specificato tramite una costante)

La lunghezza dell'array è indicata dal campo (pubblico e non modificabile) `length` dell'array stesso.

ESEMPIO

```
/**
* Primo esempio d'uso di array in Java
@author Enrico Denti
@version 1.0, 5/4/98 */

class Esempio5 {
    public static void main(String args[]){
    int[] v1 = {8,12,3,4};   // vettore di 4 int
    int[] v2; // riferimento a un vettore di int
    v2 = new int[5]; // creazione vett. di 5 int
    // (crea 5 celle int, inizializzate a 0)

    for(int i=1; i<=v2.length; i++) v2[i-1] = i*i;
    for(int i=0; i<v1.length; i++)
          System.out.print(v1[i] + '\t');
    System.out.println("");
    for(int i=0; i<v2.length; i++)
          System.out.print(v2[i] + '\t');
    }
}
```

ARRAY IN JAVA (II)

Attenzione:

Creare un array (con new) significa creare

- N celle di un tipo primitivo, se l'array è di tipi primitivi
- **N *riferimenti* a oggetti (tutti `null`) se l'array è di oggetti**

Nel secondo caso, quindi, *i singoli oggetti dovranno essere creati esplicitamente con `new`* se del caso.

ESEMPIO

```
/**
* Secondo esempio d'uso di array Java
@author Enrico Denti
@version 1.0, 5/4/98 */

class Esempio5bis {
    public static void main(String args[]){

    String v1[]; // RIFERIMENTO al futuro vettore
    // si può anche scrivere  String[] v1;

    v1 = new String[2];
    // creazione vettore di due (rif. a) String
    // NB: le due String però ancora non esistono!

    v1[0] = "Ciao mondo";  // una String costante
    v1[1] = new String(v1[0]);
    // una String creata din. uguale alla prima
    }
}
```

METODI DI CLASSE

Finora, i metodi definiti dentro a una classe (a parte il `main`…) erano destinati a essere invocati su un'istanza di quella classe:

```
class Counter {
 private int val;
 public Counter()    { val=1; }
 public Counter(int x) { val=x; }
 public void reset() { val = 0; }
 public void inc()   { val++; }
 public void dec()   { val--; }
 public int  rd()    { return val; }
}
```

```
public class Esempio2bis {
  public static void main(String args[]){
    Counter c1, c2;
    c1 = new Counter(); c2 = new Counter(5);
    c1.inc(); c1.inc(); c2.dec();
    System.out.println("c1 vale " + c1.rd());
    System.out.println("c2 vale " + c2.rd());
  }
}
```

Una applicazione Java, però, consiste di *istanze* e ***classi***.

Dato che Java non consente di definire funzioni "esterne" come in C, le classi Java fungono anche da *contenitori* di funzioni.

Tali metodi *non sono destinati a essere invocati su istanze* della classe, ma *come semplici funzioni "stand-alone"*, che forniscono funzionalità utili (e in qualche modo correlate con la classe)

Tali metodi si dicono *metodi di classe* (`static`)

Il `main` è il caso più eclatante di metodo `static`.

METODI DI CLASSE – ESEMPIO

Questo esempio, come caso limite, usa solo metodi di classe:

```
class Esempio7 {

  static int[] creaIntervallo(int inf, int sup){
    // restituisce un (rif. a) un array di int
    int[] v = new int[sup-inf+1];
    for (int i=0; i<v.length; i++) v[i]=inf++;
    return v;
  }

  static void print(int v[]){
    for(int i=0; i<v.length; i++)
        System.out.println(v[i]);
  }
  static void increment(int v[]){
    for(int i=0; i<v.length; i++) v[i]++;
  }
  public static void main(String args[]){
    int[] v1 = creaIntervallo(3,8);
    print(v1);          // Esempio7.print(v1);
    increment(v1);      // Esempio7.increment(v1);
    print(v1);          // Esempio7.print(v1);
  }
}
```

Nelle chiamate:

- il nome della classe può qui essere omesso se i metodi vengono invocati da altri metodi della classe
- bisogna mettere il nome della classe, secondo le regole dettate da visibilità e package, per chiamare un metodo di classe al di fuori della classe stessa (vedi commento nel codice)

METODI DI CLASSE – CAUTELE

I metodi di classe *non hanno nulla a che fare* con i metodi di istanza (cioè quelli non-`static`), né con i campi dati di istanza definiti nella classe (che esistono solo nelle istanze).

In particolare, ***un metodo di classe non può né richiamare metodi né manipolare dati di istanza*** *senza aver prima creato una istanza*:

```
class Esempio7bis {
  int x;                    // campo dati di istanza
  void inc() { x++; }       // metodo di istanza

  public static void main(String args[]){
    inc(); // NO!! ERRATO!! Cosa incrementerebbe??
    Esempio7bis istanza = new Esempio7bis();
    // questa è un'istanza!
    istanza.inc();          // OK
    istanza.x--;            // OK
  }
}
```

VARIABILI DI CLASSE

I metodi di classe *possono invece agire sulle variabili di classe*, definite a loro volte come `static`:

```
class Esempio7ter {
  static int x = 8;       // variabile di classe

  public static void main(String args[]){
    x--;                  // OK (visib. package)
    // Sintassi completa:  Esempio7ter.x--;
  }
}
```

Come per i metodi di classe, anche per le variabili di classe il nome completo comprende il nome della classe: qui lo si può omettere in quanto l'uso di `x` è effettuato localmente alla classe stessa.

In definitiva:

i metodi static possono agire solo su variabili static

UN ESEMPIO NOTEVOLE: LA CLASSE Math

Math è una classe di sistema (definita in `java.lang`) che comprende *solo metodi e dati statici*: in pratica, è la libreria matematica.

- variabili di classe: `E` e `PI`
- metodi di classe: `abs()`, `asin()`, `acos()`, `atan()`, `min()`, `max()`, `exp()`, `log()`, `pow()`, `sin()`, `cos()`, `tan()`, `sqrt()`, etc.

COSTANTI

Ogni variabile Java può essere considerata una costante, se etichettata dalla parola chiave `final`.

Tipicamente, una costante pubblica è una variabile di classe pubblica resa costante grazie a `final`:

```
class Esempio7quater {
  public static final int mymax = 8;
  // costante pubblica (di classe)
}
```

È possibile definire anche costanti "a uso interno", sia sotto forma di variabili di classe non pubbliche (visibili e utilizzabili in tutta la classe), sia di variabili locali a un metodo (visibili e utilizzabili solo entro quel metodo):

```
class Esempio7quater {
  static final int mymax = 8;
  // costante a uso interno (di classe)
  void metodo(){
    final float radiceDi2 = 1.4142;
  }
}
```

NOTE:

- la parola chiave `final` deve *precedere* il tipo (`int` o `float`)

IL CONCETTO DI PACKAGE

Una applicazione è quasi sempre composta di molte classi, e averle tutte insieme nella stessa cartella (directory) non è pratico.

Per questo, Java introduce il concetto di ***package*** come *gruppo di classi* appartenenti concettualmente allo stesso progetto.

Caratteristiche principali:

- Un package **può comprendere molte classi, anche definite in file separati**
- **Esiste una corrispondenza biunivoca** fra ***nome del package*** e ***posizione nel file system*** delle classi del package:
  un package di nome `pippo` presuppone che tutte le sue classi si trovino in una cartella (directory) di nome `pippo`.

ESEMPIO

```
package pippo; //dichiaro che la classe di questo
               // file appartiene al package pippo

public class Counter {...}
```

```
package pippo; // come prima, dichiarazione di
               // appartenenza al package

public class Esempio2bis {
  public static void main(String args[]){
...}
```

COMPILAZIONE ED ESECUZIONE

Per compilare una classe che appartiene a un package bisogna compilare *obbligatoriamente* dalla cartella di livello superiore, altrimenti non si generano le informazioni relative alla posizione nel file system):

```
javac  pippo/Counter.java  pippo/Esempio2bis.java
```

Anche per eseguire una classe che appartiene a un certo package bisogna andare *obbligatoriamente* dalla cartella di livello superiore e invocare la classe con il suo nome "completo":

```
java  pippo.Esempio2bis
```

Notare il punto al posto della sbarra!!!

PACKAGE DI DEFAULT

Se una classe non dichiara esplicitamente a quale package appartiene, viene implicitamente assegnata al ***package di default***, che fa riferimento per convenzione alle classi del *direttorio corrente*. Le classi del package di default si compilano e si richiamano senza premettere alcun nome di package.

Questo è l'approccio seguito finora per tutti gli esempi, ed è spesso utilizzato per test e piccole applicazioni.

SISTEMA DEI NOMI DI PACKAGE

Il sistema dei nomi dei package è strutturato e gerarchico, come il file system. È quindi possibile avere nomi “composti” come

```
java.awt.print
```

corrisponde al package contenuto in *java/awt/print*

```
pippo.pluto.papero
```

corrisponde al package contenuto in *pippo/pluto/papero*

Conseguentemente, le classi definite all’interno di tali package avranno come *nome composto* un nome come

```
java.awt.print.Book
```

```
pippo.pluto.papero.Counter
```

Il punto di partenza da dove cercare le classi è stabilito dalla variabile di ambiente **CLASSPATH**

Esempio:

```
CLASSPATH=C:/user/java; C:/franco/prove
```

Si cercano le classi a partire da quei due direttori

Quindi la classe: `C:/user/java/ccc/ddd/prova.class`

Si trova indicandola col nome composto: `ccc.ddd.prova`

Quindi la classe: `C:/franco/prove/files/f1.class`

Si trova indicandola col nome composto `files.f1`

PACKAGE e IMPORT

Quando si usa una classe definita in un altro package, occorre in generale indicare *il nome composto della classe* (in dipendenza dei valori della variabile CLASSPATH).

```
/* classe definita nel package di default */

public class Esempio2ter {
  public static void main(String args[]){
    pippo.Counter c1, c2;
    c1 = new pippo.Counter();
    c2 = new pippo.Counter(5);
    c1.inc(); c1.inc(); c2.inc();
    ...
} }
```

Questo è scomodo se la classe è usata spesso.

L'istruzione `import` consente di *importare un nome* in modo da poterlo usare senza dover ripetere il nome del package:

```
import pippo.Counter;

public class Esempio2ter {
  public static void main(String args[]){
    Counter c1, c2;
    c1 = new Counter(); c2 = new Counter(5);
    ...  }
}
```

NB: l'istruzione `import` non è una `#include`, **non include nulla**, evita solo di dover riscrivere lunghi nomi di classi.

Per importare tutte le classi di un package in un sol colpo:

```
import pippo.*;
```

NB: le classi importate devono naturalmente trovarsi nelle directory imposte dal nome del package.

PACKAGE e VISIBILITÀ

Oltre a pubblico / privato, in Java esiste una terza qualifica di visibilità: la **visibilità package**.

La visibilità package:

- è il default per classi e metodi (quando non si specifica altro)
- significa che dati e metodi sono *accessibili solo per le altre classi dello stesso package* (in qualunque file siano definite)
- eventuali altre classi definite in altri package *non potranno accedere* a dati e metodi di queste, come se fossero privati.

Ricorda: il main deve essere necessariamente pubblico, e va obbligatoriamente definito in una classe pubblica!

Attenzione:

A differenza del C (o del C++), in Java non è possibile definire classi visibili *solo in un file*: **il file è solo un contenitore fisico, non delimita un ambito di visilibità (scope).**

Se si ha una necessità del genere, la via maestra è *definire un apposito package* un cui inserire tali classi.

QUALCHE PRECISAZIONE su Java e C

- In Java non esiste alcun preprocessore
- In Java non esistono macro: il compilatore è efficiente e effettua l'inline automatico quando opportuno e possibile
- Java non distingue fra *dichiarazione* e *definizione* di funzioni, quindi non c'è necessità di file header, prototipi, etc.
- in Java *non ci sono* `struct`, `union`, `enum`, `typedef` (esistono solo classi!), né campi di bit , né gli operatori `&`, `*` e `->`