IL LINGUAGGIO JAVA

- È un linguaggio ***object-oriented***, totalmente a ***oggetti***: tranne i tipi primitivi di base (int, float, …), tutto è oggetto; relazione di ***ereditarietà semplice*** fra *classi*, ***ereditarietà multipla*** delle *interfacce*
- È fortemente ispirato a C++, ma riprogettato ***senza*** il requisito della piena ***compatibilità con C*** (comunque, similitudini…)
- Ogni programma è un *insieme di classi:* ***non*** esistono ***funzioni*** definite (come in C) ***a livello esterno***; anche la funzione `main()` va definita dentro a una classe.

Riferimenti:

- `java.sun.com`
- `lia.deis.unibo.it/Misc/` (*mirror* del tutorial Java)

Esistono moltissimi libri su Java, sia su linguaggio/ architettura che su aspetti/package specifici, sia in lingua italiana che in inglese. Consigliati:

- D. Flanagan, Java in a Nutshell, Third Edition, O'Reilly, 2000.
- C. S. Horstmann, G. Cornell, Core Java 2, Fourth Edition, 1999.

JAVA: LINGUAGGIO O ARCHITETTURA?

A differenza di C++, Java viene fornito con una ***ampia gerarchia di classi standard*** di linguaggio già pronte

Caratteristiche principali di Java:

- Linguaggio ***orientato agli oggetti***
- ***Interpretato*** (*byte code* intermedio eseguito da una *macchina virtuale*)
- ***Portabile*** (linguaggio indip. implementazione della macchina virtuale, codice generato indip. piattaforma)
- ***Dinamico*** (*class loading* a tempo di esecuzione)
- ***Robusto*** (fortemente tipato, assenza di puntatori, gestione eccezioni)
- Supporto evoluto alla ***sicurezza***
  - Modello di sicurezza (`SecurityManager` e permessi)
  - Crittografia, chiavi pubbliche/private, controllo accessi…
- Supporto per il ***multi-threading***
- Supporto alla ***programmazione di rete*** (RMI, socket, class-loader distribuito, URL …)
- ***Applet*** come applicazione integrata in Web browser, con codice scaricato dalla rete a tempo di esecuzione
- ***Package grafico*** (indip. piattaforma): *AWT, Swing*
- Programmazione a ***eventi*** (molto evoluta)
- Facilità di interfacciamento con ***database****: JDBC*
- ***Performance*** (*compilatori just in time*)

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  rd()    { return val; }
}
```

- ***Dati*** (`val`) e ***operazioni*** (`reset, inc, dec, rd`) sono *riuniti in un* ***unico costrutto linguistico***
- Il campo dati **`val`** **è <u>privato</u>**, e può essere acceduto *solo da operazioni (metodi)* della *classe* (`reset,inc,dec,rd`): ogni tentativo di accesso dall’esterno viene impedito a livello di compilazione
  - I metodi **`reset(), inc(),`** etc. **sono <u>pubblici</u>**, e possono quindi essere invocati *da chiunque*
    - → dall’esterno, `val` è manipolabile solo tramite questi metodi
    - → viene garantita la proprietà di *incapsulamento.*

NOTE:

- i nomi delle classi Java sono per convenzione **maiuscoli**
- a differenza del C++, in Java i metodi vanno *definiti* dentro alla classe a cui appartengono, non fuori da essa

UN PRIMO ESEMPIO: Hello World

```
/** File Esempio0.java
* Applicazione Java da linea di comando
* che stampa la classica frase di benvenuto
*/

public class Esempio0 {

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

- main(): ***definito obbligatoriamente*** dentro a una classe **<u>pubblica</u>**, con 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)

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* (spazio dei nomi di default) 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()`

UN PRIMO ESEMPIO: Hello World (II)

```
/** File Esempio0.java
* Applicazione Java da linea di comando
* che stampa la classica frase di benvenuto
*/

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, Linux, Alpha…) *per cui sia disponibile il Java Runtime Environment* (nella stessa versione)

Distinzione fra *Java Development Kit* (**JDK** – vero e proprio ambiente di sviluppo e programmazione) e *Java Runtime Environment* (**JRE** – solo supporto a tempo di esecuzione)

UN PRIMO ESEMPIO: Hello World (III)

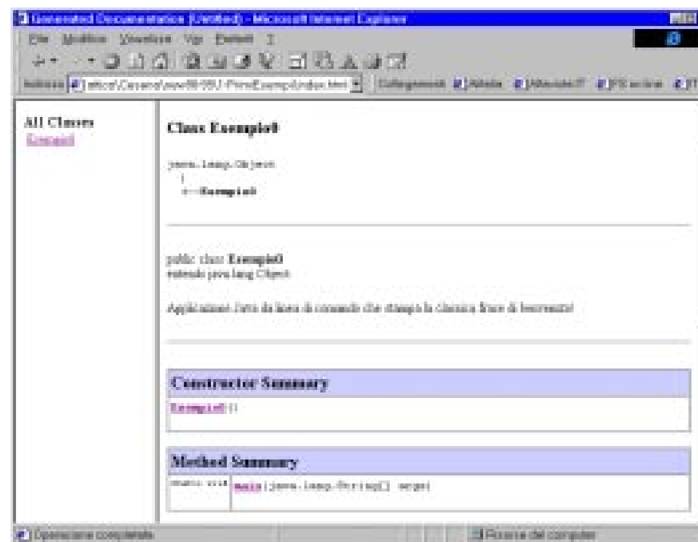
```
/** File Esempio0.java
* Applicazione Java da linea di comando
* che stampa la classica frase di benvenuto
*/

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 da compilare (limitatamente a quelle non private):

  `javadoc Esempio0.java`

Il risultato è un insieme di file HTML



ESEMPIO: Stampa degli Argomenti

```
/** File Esempio1.java
* applicazione da linea di comando
* stampa gli argomenti passati al programma
*/

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`)

ESEMPIO: Creazione e Uso di Oggetti `Counter`

```
/** File Esempio2.java
  * Prima applicazione che istanzia nuovi oggetti
  */

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 riga **`Counter c1;`** introduce un ***riferimento*** a `Counter`, cioè una sorta di puntatore ad un oggetto di quel tipo che però ***NON*** è direttamente ***manipolabile*** dall'utente e si usa *senza dereferenziazione esplicita* 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 sistema secondo politiche predefinite.

I COSTRUTTORI

Partendo dall'osservazione che molti errori di programmazione sono causati da mancate inizializzazioni di variabili ai valori iniziali previsti,

⇒ molti linguaggi a oggetti definiscono il concetto di **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
- ha un **nome fisso** (uguale al nome della classe)
- **non ha tipo di ritorno** (il suo scopo non è effettuare una operazione di calcolo, ma inizializzare un oggetto)
- **può non essere unico** (spesso vi sono più costruttori, con diverse liste di argomenti)

Se non ne viene definito nessuno, viene generato automaticamente un *costruttore di default* (senza argomenti), dal comportamento vuoto

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.
  */

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
  */

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
- `main()` in *Esempio2bis* può usare la classe Counter **senza dover includere nulla**: la piattaforma Java supporta il ***collegamento dinamico***
  Al primo uso di Counter, il file `Counter.class` verrà cercato e caricato dal ***classloader di sistema*** → adatto a *codice mobile* e *applicazioni su Internet*

IL CONCETTO DI PACKAGE

Un'applicazione ad oggetti 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 strutturazione in ***package***, *gruppi di classi* appartenenti 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;

public class Counter {
    ...
}
```

```
package pippo;

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

Per compilare (dal direttorio padre di `pippo`):

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

Per eseguire (dal direttorio padre di `pippo`):

```
java  pippo.Esempio2bis
```


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 al *direttorio corrente*.

Le classi del package di default si compilano e si richiamano senza premettere alcun nome di package.

SISTEMA DEI NOMI DI PACKAGE

Il sistema dei nomi dei package è ***strutturato***. È quindi possibile avere ***nomi "composti"*** come:

```
java.awt.print
pippo.pluto.papero
```

Conseguentemente, le classi definite all'interno di tali package avranno come ***nome assoluto***:

```
java.awt.print.Book
pippo.pluto.papero.Counter
```

PACKAGE e IMPORT

Quando si usa una classe definita in un altro package, occorre in generale indicare *il nome assoluto della classe:*

```
/* 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();
    ...  }  }
```

L'istruzione `import` consente di *importare un nome* in modo da non 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);
    ...
  }
}
```

NOTA: **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.*;
```

PACKAGE e VISIBILITÀ

Varie qualifiche di ***visibilità*** in Java:

1. **`public`**
2. **`private`**
3. **`package`**

La visibilità package:

- ***è il default per classi e metodi***
- 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 possono accedere*** a dati e metodi di queste, come se fossero privati.

NOTA: il main deve essere necessariamente `public`, e va obbligatoriamente definito in una classe `public`

<u>Attenzione:</u>

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 visibilità (scope)**

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

Riassumendo:

	tipo di visibilità		
accessibile a:	**public**	**package**	**private**
stessa classe	sì	sì	sì
classi in stesso package	sì	sì	no
differenti package	sì	no	no

QUALCHE PRECISAZIONE su Java e C

- In Java ***NON*** esiste alcun ***preprocessore***
- In Java ***NON*** esistono ***macro***: il compilatore cerca di effettuare *inline* automatico secondo strategie predefinite, 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 `->`

QUALCHE PRECISAZIONE su Java e C++

- In Java si ha ***ereditarietà semplice*** delle ***classi*** (C++ permette l'ereditarietà multipla delle classi), ed ***ereditarietà multipla*** solo delle ***interfacce*** (vedi dettaglio nei lucidi "Ereditarietà e Polimorfismo in Java")

TIPI PRIMITIVI, CLASSI e OGGETTI

In Java si distingue fra

- ***tipi primitivi*** (`boolean, char, byte, short, int, long, float, double`)
- ***classi***
- ***riferimenti a oggetti***

Non è possibile definire ***oggetti allocati staticamente***: ogni istanza deve essere allocata dinamicamente con `new`

Oggetti Java sono manipolabili ***solo tramite riferimenti***

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}

NOTE:

- 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.

RIFERIMENTI A OGGETTI

In Java, gli oggetti si manipolano ***esclusivamente tramite riferimenti***

Non è possibile definire oggetti allocati staticamente, tutto dev'essere allocato dinamicamente con `new`

Una linea di codice del tipo `c2 = c1` fa puntare i due riferimenti allo stesso oggetto, *non duplica l'oggetto riferito!*

ESEMPIO:

```
/**
* verifica il concetto di Riferimento a oggetti
*/

import java.awt.Point;

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 ripercuote anche su ciò che è riferito da `p2` (due riferimenti che referenziano lo stesso oggetto)
- Per duplicare davvero l'oggetto occorre usare un metodo specifico **`clone()`** (che è predefinito per tutte le classi)

CLONAZIONE DI OGGETTI

Per duplicare un oggetto, si ha a disposizione il metodo **`clone()`**, predefinito per tutte le classi:

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

ESEMPIO:

```
/**
* verifica il concetto di clonazione di oggetti
*/

import java.awt.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);
 }
}
```

- In questo modo, `p2` rimane inalterato al valore (100,10), mentre `p1` viene modificato in (400,10)
- Da notare ***l'operazione di cast*** per convertire il risultato di `clone()`, che è un oggetto generico (`Object`), in un oggetto della classe `Point`

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 si usa il metodo **`equals()`**, predefinito per tutte le classi Java

ESEMPIO:

```
/**
* verifica il concetto di uguaglianza fra oggetti
*/

import java.awt.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 = (Point) p1.clone();  p1.x = 400;
   System.out.println("p1==p2? " + (p1==p2));
   System.out.println("p1.equals(p2)? " +
                          p1.equals(p2));
 }
}
```

Output:

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


STRINGHE IN JAVA

Le ***stringhe Java sono oggetti***, istanze della classe `String`: un oggetto `String` rappresenta uno specifico valore (contenuto della stringa) ed è ***non modificabile***

⇒ **In Java una String non è un buffer!**

(per immagazzinare stringhe modificabili si usa `StringBuffer`)

- 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 delle Java API (Application Programming Interface) per maggiori dettagli.

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 costanti)

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
*/

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 (2)

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

NOTA:
Creare un array (con `new`) significa creare:

- N celle di un tipo primitivo, se l'array contiene elementi di tipo primitivo
- 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 opportuno

ESEMPIO:

```
/**
* Secondo esempio d'uso di array Java
*/

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
    }
}
```

TIPI PRIMITIVI e CLASSI CORRISPONDENTI

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

⇒ "**wrap classes**", una per ogni tipo primitivo

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

- a parte la lettera iniziale, la classe ha quasi sempre lo stesso nome del tipo primitivo (eccetto char e int)
- 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
  } }
```


METODI DI CLASSE (static)

I metodi definiti dentro a una classe e visti fino a questo punto (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());
  }
}
```

Ma le classi Java possono fungere anche da ***contenitori di funzioni***, anche perché in Java non è consentito di definire funzioni "esterne" come in C

I ***metodi di classe*** (**`static`**) *non sono destinati a essere invocati su istanze* della classe, ma *come* ***funzioni "stand-alone"***, che forniscono funzionalità utili (in qualche modo correlate con la classe che li implementa)

NOTA: il `main` di una classe public è il caso più evidente e frequente di metodo `static`

METODI DI CLASSE – UN ESEMPIO

Questo esempio rappresenta un caso limite in cui vengono utilizzati 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 in quanto i metodi vengono invocati da altri metodi della classe: la sintassi completa (obbligatoria per chiamate provenienti al di fuori della classe) è indicata a lato come commento

VARIABILI DI CLASSE (static)

Le ***variabili di classe*** sono associate a una classe e non ad un singolo oggetto

⇨ c'è ***una sola copia della variabile*** indipendentemente dal numero di istanze definite di quella classe

Le variabili di classe sono definite 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.

UN ESEMPIO NOTEVOLE: LA CLASSE `Math`

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

- 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`**

Generalmente, vengono utilizzate ***costanti pubbliche***, realizzate tramite variabili di classe public, rese costanti attraverso l'uso del modificatore `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;
  }
}
```

Piccole osservazioni di sintassi:

- la parola chiave **`final`** ***deve precedere il tipo*** (int o float)
- nel secondo esempio, la ***notazione 1.4142F*** per denotare una costante float è ***indispensabile***: in mancanza della F, Java interpreterebbe 1.4142 come costante double, e rifiuterebbe l'assegnamento a una variabile float

L'ISTANZA CORRENTE: `this`

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

1. per indicare un campo dati in caso di ***omonimia*** con un parametro o con una variabile locale
2. 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; }
}
```

1. 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`)
2. Il costruttore `Counter()` utilizza `this(1)` per delegare la costruzione dell'oggetto all'altro costruttore della stessa classe

NOTA: la sintassi `this()` per chiamare un altro costruttore può essere usata *solo come prima istruzione* di un costruttore

OGGETTI COMPOSTI

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 a quelli con *visibilità package* se la classe contenitore è definita nello stesso package

In fase di istanziazione:
- *prima* si costruisce l'oggetto contenitore
- *poi* si costruiscono, con `new`, gli oggetti interni

ESEMPIO:

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

public class Esempio {     // oggetto contenitore
    Counter c;             // oggetto contenuto

  public Esempio() {   //costruttore di Esempio
      c = new Counter(3);
  }

  public static void main(String args){
    Esempio e = new Esempio();
  }
}
```


AGGIORNAMENTO/MODIFICA DEI REQUISITI

Lo sviluppo di applicazioni pone di fronte spesso a problemi che richiedono ***soluzioni (classi) simili a soluzioni (classi) già esistenti***, ma ***non identiche***

Come fare per non ripetere tutto il lavoro da capo?
- copia&incolla del codice della classe esistente e cambiamenti opportuni apportati a mano
  → *proliferazione di file, procedimento compl. manuale*
- creare una nuova classe che contenga al suo interno un oggetto della classe preesistente
  → *i campi dati privati non sono direttamente manipolabili*
  → *riscrivere anche i metodi che "rimangono uguali"*

ESEMPIO: da `Counter` base a `Counter` con decremento

```
class Counter2 {
    Counter c;
    public  Counter2()      {c = new Counter(); }
    public  Counter2(int v) {c = new Counter(v);}
    public  void inc()      { c.inc(); }
    public  int  rd()       { return c.rd(); }
    public  void dec()      { c.val--; }
}
```

- il campo dati `c.val` è accessibile perché ha ***visilibità package*** e la classe `Counter2` è definita nello stesso package di `Counter`
- è comunque necessario riscrivere i metodi *solo per adattarli alla nuova classe*, ma di fatto *riscrivendoli uguali,* ***senza aggiungere nuove funzionalità***

⇨ **Occorre poter riusare le classi esistenti in modo *più flessibile***

EREDITARIETÀ

- Si vuole ***riusare*** tutto ciò che può essere riusato;
- non è utile né opportuno modificare ***codice già funzionante e corretto***, il cui sviluppo è costato in termini di tempo e risorse

Occorre disporre a livello di linguaggio di meccanismi per ***progettare alle differenze***, procedendo in modo ***incrementale***

- Obiettivo è ***NON essere obbligati a ripartire da zero*** quando serve una nuova classe, ma poterla definire ***a partire da una già esistente***

Bisognerà specificare solamente:

- cosa la nuova classe ha ***in più*** rispetto alla precedente
- sia in termini di ***dati***, sia in termini di ***operazioni***

ESEMPIO: da un Counter base al Counter con decremento

```
class Counter2 extends Counter {
   public  void dec()      { c.val--; }
}
```

- la classe `Counter2` **eredita tutti i metodi e i campi dati** di `Counter`, e può usarli come fossero definiti localmente (*sempre che la visibilità lo permetta: campi private non visibili*)
- ***in più, aggiunge dati e metodi suoi specifici*** (in questo esempio, nessun dato, solo un metodo, dec)

Quindi, una istanza di `Counter2` "contiene" al suo interno una istanza di `Counter`, più i campi specifici di `Counter2` (se ne esistono).

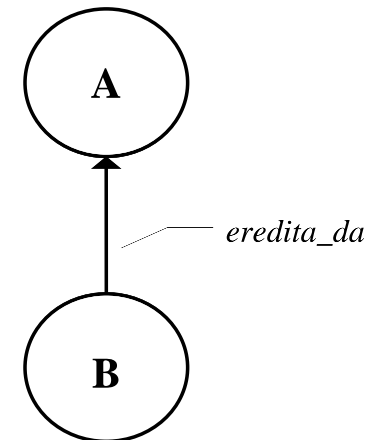
EREDITARIETÀ come RELAZIONE FRA CLASSI

Finora, l'unica relazione introdotta era ***l'istanziazione***:
- ogni ***oggetto*** è ***istanza*** di una certa ***classe***

Si aggiunge ***EREDITARIETÀ:***

Una classe B può <u>EREDITARE DA</u> una classe A (preesistente)

1) classe base (o ***superclasse***) è la classe ***da cui si eredita*** (A)
2) classe derivata (o ***sottoclasse***) è la classe ***che eredita*** (B)



LA CLASSE DERIVATA PUÒ:

- **aggiungere nuovi campi dati e nuovi metodi** a quelli ereditati dalla classe-base
- **ridefinire alcuni dei metodi** ereditati dalla classe-base (*overriding*)

LA CLASSE DERIVATA <u>NON</u> PUÒ:

- **eliminare campi dati o metodi**

VISIBILITÀ

Nella classe-base:

- ciò che è *privato* è visibile *solo ai metodi della classe stessa*
- ciò che è *di package* è visibile *solo ai metodi delle classi dello stesso package*
- ciò che è *pubblico* è *visibile a tutti*

Perciò, la classe derivata:

- ***non può vedere la parte privata*** *della classe-base*, appunto perché privata
- ***può vedere la parte di package*** *della classe-base*, purché la nuova classe sia definita nello stesso package
- ***vede, ovviamente, la parte pubblica*** *della classe-base*, in quanto visibile a tutti

È SUFFICIENTE?

Riconsideriamo il problema di definire `Counter2` (dotato di decremento) a partire da `Counter` (che ne è privo)

MEMBRI PROTETTI

Riconsideriamo il problema precedente di definire `Counter2` (dotato di decremento) a partire da `Counter`

Per realizzare l'operazione di decremento, è necessario:

- o disporre dell'***accesso diretto*** al campo `val`
- o sfruttare, se esiste, un metodo (pubblico o di package) che consenta di ***inizializzare*** il campo `val` con un valore a piacere

Nel nostro caso, non ci sono stati problemi perché **`val`** ***non era privato*** (aveva visibilità package) e **`Counter2`** è nello ***stesso package*** di **`Counter`** (quello di default)

- E se `val` fosse stato ***privato***?
- E se ***NON*** fosse stato possibile collocare `Counter2` nello ***stesso package*** di `Counter`?

Per sfruttare davvero l'ereditarietà occorre poter ***superare la rigidità*** dei tre livelli di visibilità ***private/public/package***:

visibilità specifica per le classi derivate

protected

Un campo dati o un metodo **`protected`**:

- è visibile ***alle classi derivate indipendentemente dal package*** in cui esse sono definite
- per il resto si comporta ***come la visibilità package***

REGOLE DI VISIBILITÀ

accessibile a	tipo di visibilità			
	public	**protected**	**package**	**private**
stessa classe	sì	sì	sì	sì
classi in stesso package	sì	sì	sì	no
sottoclassi in qualunque package	sì	sì	no	no
differenti package	sì	no	no	no

La qualifica `protected`:

- rende visibile un attributo a ***tutte le sottoclassi*** di quella classe, ***presenti e future***
- perciò, costituisce un ***permesso di accesso "indiscriminato"***, valido per ogni possibile sottoclasse che possa in futuro essere definita, senza possibilità di distinzione

Occorre dunque valutare molto attentamente quando sia opportuno utilizzare la clausola `protected`

COSA SI EREDITA?

La classe derivata eredita dalla classe genitore:

- ***tutti i campi dati*** (anche quelli *private* a cui la classe derivata non potrà accedere direttamente)
- ***tutti i metodi*** (anche quelli *private* che la classe derivata non potrà usare direttamente)

tranne i costruttori

PERCHÉ?

- Perché i costruttori non svolgono *funzioni*, ma costruiscono un'istanza della classe
  ➔ sono **specifici di quella particolare classe**

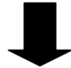
Ad esempio, costruire un `Counter` non è la stessa cosa che costruire un `Counter2` (una sottoclasse ha generalmente un numero superiore di campi dati)

Ciò non significa che la ***classe derivata*** non abbia un ***costruttore***, in quanto, come vedremo, ne viene generato ***uno di default***

COMPATIBILITÀ FRA CLASSE E SOTTOCLASSE

L'ereditarietà non è soltanto una modalità per il ***riutilizzo di codice**,* ma forza a progettare e implementare applicazioni in termini di ***una gerarchia di classificazione*** delle entità necessarie

- `Counter2` contiene la ***stessa interfaccia*** di `Counter` (i costruttori non fanno strettamente parte dell'interfaccia)
- **ogni `Counter2` è anche un `Counter`**, ovvero:
  - ogni oggetto di classe `Counter2` è *implicitamente* anche di classe `Counter`
  - ***ma non viceversa,*** perché `Counter2` è più ricco di `Counter` (sa fare cose che un `Counter` non sa fare, e può avere dei campi dati inesistenti in `Counter`)



Conseguenza (***CONFORMITÀ***):

- un'istanza di `Counter2` può sostituire un oggetto `Counter` ***in maniera trasparente*** rispetto agli oggetti che lo utilizzano
- ma *non il contrario*

ESEMPIO:

```
class Esempio1 {
   Counter c1 = new Counter(3);
   Counter2 c2 = new Counter2();
   c2.dec();      // OK: dec() esiste in Counter2
   // c1.dec();   // NO: dec() non c'è in Counter
   c1=c2;         // OK: ogni Counter2 è Counter
   // c2=c1;      // NO: c1 è un semplice Counter
}
```


COSTRUTTORI

Riprendendo la definizione di `Counter2`:

```
class Counter2 extends Counter {
   public  void dec()      { c.val--; }
}
```

si può osservare che è stato utilizzato un oggetto `Counter2 c` ***senza averne definito il costruttore***

- Java *non obbliga* a definire almeno un costruttore per ogni classe, anche se ciò è *fortemente raccomandato*
- in assenza di costruttori definiti esplicitamente, Java definisce automaticamente un ***costruttore di default***
- il costruttore di default si limita a richiamare il ***costruttore di default della superclasse***

Il codice precedente equivale cioè a:

```
class Counter2 extends Counter {
   public  void dec()      { c.val--; }
   // il costruttore di default generato da Java
   public Counter2() { super(); }
}
```

L'invocazione **`super()`** richiama invoca il costruttore di default della classe-base: qualora non esista viene sollevata un'eccezione

EREDITARIETÀ e COSTRUZIONE DI OGGETTI

Per costruire un oggetto di una classe derivata, è ***sempre necessario chiamare un costruttore della classe-base***, in quanto:

- solo il costruttore della classe base può sapere come inizializzare i campi-dati ereditati da tale classe in modo corretto
- è il solo modo per garantire l'inizializzazione di campi-dati privati (a cui la sottoclasse non può accedere direttamente)
- si evita un'inutile duplicazione di codice nella sottoclasse.

Ma *quale costruttore della classe-base* viene chiamato?

- il costruttore di default definito da Java chiama il costruttore di default della classe base
- per i costruttori definiti dall'utente occorre *specificare esplicitamente* quale costruttore della classe base vada chiamato, mediante la notazione `super(…..)`, *che deve essere la prima istruzione del corpo del costruttore*

Se manca una chiamata esplicita, il compilatore Java inserisce automaticamente una chiamata al costruttore di default della classe base (`super()`), *a meno che* non ci sia già una chiamata a un altro costruttore della stessa classe mediante la notazione `this()`

Ad esempio, l'espressione **`super(x)`** invoca il costruttore della classe-base con un argomento (eccezione se un costruttore siffatto non esiste)

Super e this

In breve:

- `this` è un riferimento all'istanza corrente
- `super` è un riferimento alla classe base

Sono espressioni lecite:

- `this.val` indica il campo val della classe di istanza corrente
- `this.f()` richiama il metodo `f()` della classe dell'istanza corrente
- `this(...)` richiama un altro costruttore della stessa classe
- `super(...)` richiama un costruttore della classe base
- `super.val` indica il campo `val` della classe base
  (notazione utile nel caso in cui la sottoclasse definisca un campo dati omonimo con uno della classe base)
- `super.f()` richiama il metodo `f()` della classe base
  (notazione utile nel caso in cui la sottoclasse definisca un metodo omonimo di uno della classe base - *overriding*)

ESEMPIO (`Counter` definisce un campo `val`):

```
class DualCounter extends Counter {
    double val;
    // val indica ora il campo val di DualCounter
    // mentre super.val indica il val di Counter
    public void inc() { super.inc(); val++; }
    // inc() indica il metodo inc() di DualCounter
    // super.inc() chiama l'inc() di Counter
}
```

COSTRUTTORI NON PUBBLICI

In assenza di costruttori definiti dall'utente, Java definisce sempre un costruttore di default per ogni classe: ***tale costruttore è pubblico***, in modo da consentire alla classe di essere normalmente istanziata

In alcune circostanze tuttavia si può desiderare di *impedire che una classe possa essere istanziata da chiunque*. In particolare, può essere necessario:

- impedire qualunque istanziazione
- consentire solo istanziazioni da parte di sottoclassi

Nel primo caso, l'utente deve definire un ***costruttore privato***, nel secondo un ***costruttore protetto***

In questo modo, si previene la generazione del costruttore pubblico automatico da parte di Java.

CLASSI FINALI (`final`)

Come si è già visto, una variabile dichiarata **`final`** diventa una costante, ossia non è più modificabile

Allo stesso modo, una classe dichiarata `final` diventa una **classe finale, ovvero da cui non è possibile derivare sottoclassi**

ESEMPIO RIASSUNTIVO: **PERSONE E STUDENTI**

```
/** la classe Persona
*/

class Persona {
  String nome;
  int anni;
  Persona() { nome = "sconosciuto"; anni = 0; }
  Persona(String n) { nome = n; anni = 0; }
  Persona(String n, int a) { nome=n; anni=a; }
  void print(){
    System.out.print("Il mio nome e' " + nome);
    System.out.println(" e ho " + anni + " anni");
  } }
```

```
/** la classe Studente che eredita da Persona
*/

class Studente extends Persona {
  int matr;
  Studente() { super(); matr = 9999; }
  Studente(String n) { super(n); matr = 8888; }
  Studente(String n, int a) { super(n,a); matr=7;}
  Studente(String n, int a, int m) {
       super(n,a); matr=m; }
  void print(){ super.print();
    System.out.println("Matricola = " + matr); }
}
```

```
public class Esempio2 {
  public static void main(String args[]){
  Studente studente;
  studente = new Studente();
  studente.print();
  studente = new Studente("Tom", 30);
  studente.print();
  studente = new Studente("Laura", 20, 1234);
  studente.print();
  } }
```

EREDITARIETÀ E COMPATIBILITÀ DI TIPO

Se la classe ***B è sottoclasse*** della classe ***A***, ogni ***oggetto di classe B è anche di classe A*** (non viceversa)

⇨ la classe B è *un sottotipo* della classe A

Ciò implica una **compatibilità (conformità) da sottotipo a tipo** (non viceversa), da cui segue la possibilità di usare un oggetto della classe derivata ***al posto di*** uno della classe-base (non viceversa)

ESEMPIO:

```
class Esempio3 {
    Persona  p = new Persona("John");
    Studente s = new Studente("Tom");
    p.print();     // stampa solo il nome
    s.print();     // stampa nome e matricola
    p=s;
    p.print();     // COSA STAMPA ???
}
```

L'assegnamento `p=s`:

- **è possibile**, perché ogni `Studente` è anche una `Persona`
- **e non comporta perdite di informazione**, perché si tratta di un assegnamento *fra riferimenti* (non fra variabili)

Problema:

- `p` è ***definito*** come riferimento a una generica `Persona`
- ***ma fa riferimento in realtà a uno*** **`Studente`**

Come viene interpretato ora `p`? Come semplice `Persona` o come `Studente`?

POLIMORFISMO

Ricordiamo che una sezione di codice si dice ***POLIMORFA*** *se è capace di operare su oggetti di tipo diverso* ***specializzando il suo comportamento in base al tipo dell'oggetto su cui opera***

In Java la possibilità di usare un oggetto della classe B al posto di uno di classe A introduce la possibilità *teorica* di avere polimorfismo. **E in pratica?**

Dipende se prevale il ***tipo formale del riferimento*** (`Persona` nell'esempio precedente) o il ***tipo effettivo dell'istanza corrente*** (`Studente` nell'esempio precedente)

In Java prevale il ***tipo effettivo dell'istanza corrente***
→ **LEGAME DINAMICO (LATE BINDING)**

ESEMPIO:

```
public class Esempio2 {
  public static void main(String args[]){
    Persona p = new Studente("Anna");
    p.print(); // STAMPA NOME E MATRICOLA !!
  }
}
```

Sebbene `p` sia un riferimento a `Persona`, poiché l'istanza corrente è uno `Studente` viene richiamato il metodo `print()` della classe `Studente`

INTERFACCE IN JAVA

Una interfaccia Java costituisce una ***pura specifica di interazione:***

- contiene solo ***dichiarazioni di metodi*** (ed eventualmente costanti)
- ***NON*** contiene ***variabili né definizioni di metodi***

Una classe Java può implementare ***una o più interfacce***
La classe ***DEVE definire*** tutti i metodi delle interfacce implementate

ESEMPIO:

```
public interface Icomplex {
    public Icomplex sum(Icomplex z);
}

public class Complex implements Icomplex {
    public Complex sum(Complex z) {…}
}
```

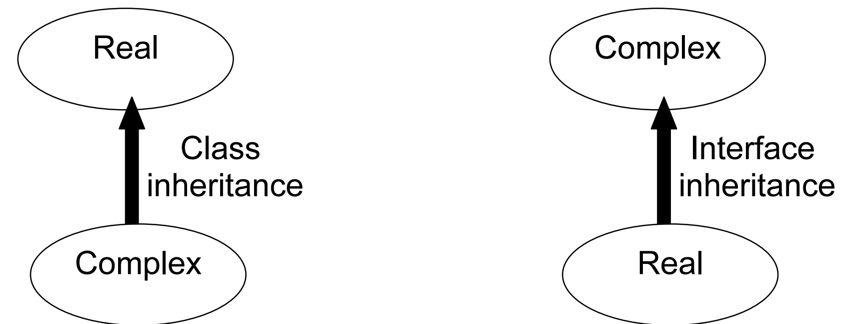
Ma a che cosa servono le interfacce?

Esempio della ***rappresentazione dei numeri reali***
Due possibilità:

classe `Complex` ***come sottoclasse di*** `Real`	***classe*** `Real` ***come sottoclasse di*** `Complex`
Approccio ***pragmatico*** (riutilizzo della classe `Real`) ma ***concettualmente scorretto***	Approccio ***concettualmente corretto*** ma che obbliga a ridefinire `Real`!!

Occorrerebbero ***due gerarchie differenziate***:

- una per la parte implementativa (riutilizzo di codice)
- una per l'interfaccia esterna (capace di mantenere la correttezza concettuale)



GERARCHIE DI INTERFACCE

Le interfacce possono dare luogo a ***gerarchie di ereditarietà*** in maniera analoga alle classi

- Gerarchia separata da quella delle classi
- Slegata da aspetti implementativi
- Esprime relazioni concettuali
- Guida la progettazione e la modellizzazione

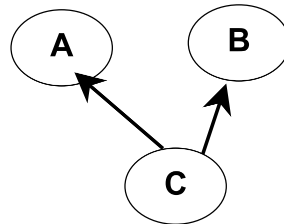
L'ereditarietà delle interfacce è però MULTIPLA

EREDITARIETÀ MULTIPLA DELLE INTERFACCE

Perché in Java c'è ***ereditarietà semplice per le classi*** (una classe può ereditare da una sola superclasse), ma ***ereditarietà multipla per le interfacce***?

L'***ereditarietà multipla fra classi*** complicherebbe notevolmente il compito della macchina virtuale a tempo di esecuzione:

- la classe C unisce sia i ***dati di A che di B***
  - che si fa con le omonimie?
  - come si distinguono?
- La classe C unisce i ***metodi delle classi A e B***
  - definizioni doppie?
  - se la cosa si ripete ricorsivamente nella gerarchia?



Al contrario, le ***interfacce contengono solo dichiarazioni*** di metodi:

- ***NON*** contengono implementazioni (nessun problema di collisioni fra ***metodi omonimi***)
- ***NON*** contengono variabili (nessun problema di collisione fra ***dati omonimi***)

INTERFACCE E PROGETTAZIONE

Le interfacce inducono un modo differente di concepire la progettazione di un'applicazione

- ***Prima*** si definiscono le interfacce che servono e si stabiliscono le ***relazioni fra interfacce***
  - La gerarchia di interfacce riflette scelte di progetto (pulizia concettuale)
- ***Poi*** si progettano le ***classi*** che implementano le interfacce definite
  - La gerarchia delle classi può riflettere le scelte implementative considerate più efficaci

CLASSI ASTRATTE IN JAVA

E le classi astratte? Perché un'altra astrazione oltre alle ***interfacce***?

Una ***classe astratta*** in Java fattorizza, dichiarandole, operazioni comuni a tutte le sue sottoclassi, ma ***NON è obbligata a definirle tutte*** tramite la specifica dell'implementazione

Non viene creata per definire istanze, ma per ***derivarne altre classi (astratte o no)***, che dovranno implementare i metodi solo dichiarati dalla classe astratta

Esempio:

```
public abstract class Animale {
public abstract String verso();
public abstract String si_muove();
public abstract String vive();
…
}

public abstract class AnimaleTerrestre
        extends Animale {
public String vive()  {       // era abstract
    return "sulla terraferma";    }
…
}
```

Diversamente dalle interfacce, le ***classi astratte sono vere e proprie classi***:

- ***ereditarietà semplice***
- ***possono definire alcuni metodi e variabili***, che vengono ereditati dalle sottoclassi, astratte o meno

NOTA:

ogni classe che dichiari ***almeno un metodo*** come `abstract` deve essere qualificata come ***classe astratta*** (altrimenti errore in compilazione)

ECCEZIONI IN JAVA

Spesso i programmi contengono ***istruzioni "critiche"***, che possono condurre a errori a tempo di esecuzione

Tipicamente, in quei casi il programmatore inserisce ***controlli*** (`if… then..`) per ***intercettare le situazioni critiche*** e cercare di ***gestirle*** in modo appropriato

Tuttavia, questo modo manuale di procedere è spesso insoddisfacente:

- non è facile prevedere ***tutte*** le situazioni che potrebbero produrre l'errore
- la gestione dell'errore viene spesso scambiata con la stampa a video di qualche messaggio

Java adotta anche in questo ambito un approccio innovativo, introducendo il concetto di ***eccezione***:

- anziché tentare di prevedere i casi che possono portare a errore, si prova a eseguire l'operazione ***in un blocco "controllato"***
- se si produce un errore, l'operazione ***solleva un'eccezione***
- l'eccezione viene ***catturata*** dal blocco entro cui l'operazione era eseguita, e può essere ***gestita*** da una routine apposita


```
try {
   // operazione critica che può sollevare eccez.
}
catch (Exception e) {
   // gestione dell'eccezione }
```

Se l'operazione può sollevare diversi tipi di eccezione (corrispondenti a diversi tipi di errore), possono essere previsti ***più blocchi*** `catch` di seguito allo stesso blocco `try`

ESEMPIO:

Una tipica operazione "critica" è la ***lettura da input***

In Java, il ***dispositivo di input standard*** è la variabile (`static`) `System.in`, di classe `InputStream` (una classe astratta, di cui `System.in` è una istanza "anomala" predefinita)

Poiché `InputStream` fornisce solo un metodo `read()` che legge singoli byte, si usa incapsulare `System.in` in un oggetto dotato di maggiori funzionalità, come ad esempio un `BufferedReader`, che fornisce anche un metodo `readLine()`:

```
import java.io.*;
class EsempioIn {
  public static void main(String args[]){
    int a = 0, b = 0;
    BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
    try { System.out.print("Primo valore: ");
      a = Integer.parseInt(in.readLine());
      System.out.print("Secondo valore:");
      b = Integer.parseInt(in.readLine());}
    catch (IOException e) {
      System.out.println("Errore in input");
    }
    System.out.println("La somma vale " + (a+b));
  } }
```

Qui, `readLine()` solleva una `IOException` in caso di errore in fase di input, mentre `Integer.parseInt()` solleva una `NumberFormatException` (non catturata) se la stringa restituita da `readLine()` non corrisponde alla sintassi di un numero intero

Una eccezione non catturata si propaga verso l'esterno, di blocco in blocco: se raggiunge il main, provoca la terminazione dell'applicazione

COS'È UN'ECCEZIONE

Una ***eccezione è un oggetto***, istanza di `java.lang.Throwable` o di una qualche sua sottoclasse.

In particolare, le due sottoclassi più comuni sono

- `java.lang.Exception`
- `java.lang.Error`

Il termine eccezione è generalmente usato in entrambi i casi

- Un `Error` indica problemi relativi al caricamento della classi o al funzionamento della macchina virtuale Java (es. *not enough memory*), e va considerato ***irrecuperabile***, perciò ***non da catturare***
- Una `Exception`, invece, indica di solito situazioni ***recuperabili*** (es. *fine file, indice di un array oltre i limiti, errori di input*)

Poiché un'eccezione è un oggetto, può contenere dati o definire metodi:

- tutte le eccezioni definiscono un metodo `getMessage()` che restituisce il messaggio d'errore associato
- alcune eccezioni definiscono dei campi, come `bytesTransferred` in `InterruptedIOException`, che forniscono ulteriori informazioni utili per meglio gestire l'evento anomalo

RILANCIARE ECCEZIONI

Java richiede che un metodo entro cui si può generare un'eccezione ***o gestisca l'eccezione***, con un costrutto `try/catch`, ***oppure dichiari di rilanciarla all'esterno*** del metodo stesso, con la clausola `throws`:

```
public int readInteger(BufferedReader in)
    throws IOException, NumberFormatException {
  return Integer.parseInt(in.readLine());
}
```

DEFINIRE E GENERARE NUOVE ECCEZIONI

Essendo un'eccezione nulla più che un normale oggetto Java (istanza di una classe opportuna), è possibile

1. ***definire nuovi tipi*** di eccezione (nuove classi)
2. ***generare eccezioni*** dall'interno di ***propri metodi***

1) è sufficiente definire una nuova classe che estenda la classe `Exception` o una delle sue sottoclassi:

```
class NumberTooBigException
    extends IllegalArgumentException {
 public NumberTooBigException() { super(); }
 public NumberTooBigException(String s){super(s);}
   }
```

Solitamente non si definiscono particolari campi dati, in quanto l'informazione fondamentale (il ***tipo dell'eccezione***) è già veicolata dalla classe stessa (e dal messaggio s)

2) è sufficiente creare l'oggetto eccezione, e poi "lanciarlo" con l'istruzione `throw`:

```
public int readInteger(BufferedReader in)
    throws IOException, NumberFormatException,
           NumberTooBigException {
    int x = Integer.parseInt(in.readLine());
    if (x>100) throw new NumberTooBigException();
    return x;  }
```

NOTA:

- non confondere **`throw`** (che è un'istruzione che lancia un'ecce-zione) con **`throws`** (usata nella dichiarazione di una classe)
- la creazione dell'oggetto `NumberTooBigException` può avvenire - e spesso avviene - entro l'istruzione `throw` stessa.