

Breve introduzione a Java

(ed alla programmazione ad oggetti)

Marco Piastra

Argomenti

1. Modello astratto e modello implementativo
(in un linguaggio di programmazione)
2. Modello astratto: rappresentazione ad oggetti
3. Modello implementativo: macchina virtuale
4. Aspetti particolari:
 - gestione implicita della memoria (*garbage collection*)
 - multi-threading;
 - programmazione distribuita.

1

Modello astratto e modello implementativo

Il progetto di un linguaggio

Da tener presente:

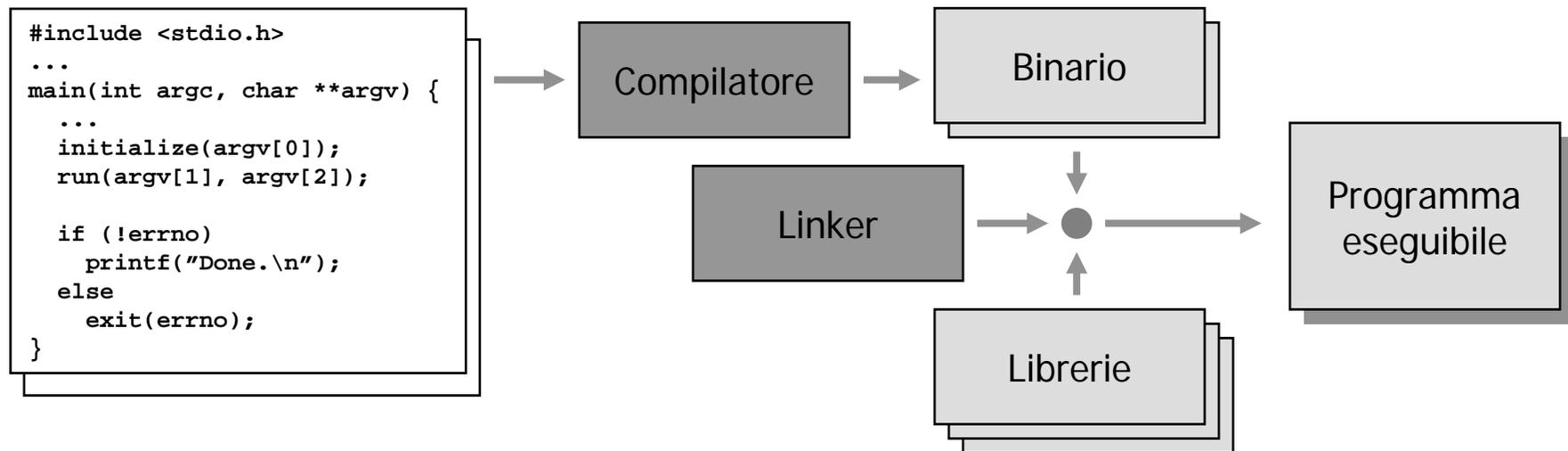
- un 'linguaggio di programmazione' è progettato in riferimento ad *modello implementativo* specifico
(linguaggio compilato, linguaggio interpretato)
- il progetto del linguaggio include un rilevante aspetto teorico, cioè un *modello astratto*
(*variabili, oggetti, classi, metodi*)
- i due aspetti sono strettamente correlati

Il progetto di un linguaggio (2)

- Scelta di un modello astratto:
 - entità di base della programmazione (e.g. variabili globali, variabili locali, array, puntatori, funzioni).
 - modo di descrivere i programmi (e. g. programmazione strutturata);
 - linguaggio (inteso come unione di una *sintassi* ed una *semantica* formale);
- Scelta di un modello implementativo:
 - tecniche di compilazione (e.g. compilazione parziale) e di linking;
 - esecuzione a run-time (e.g. gestione della memoria, gestione dello stack delle chiamate);
 - possibilità di estensione (librerie specifiche).

Creare un programma in C

Codice sorgente

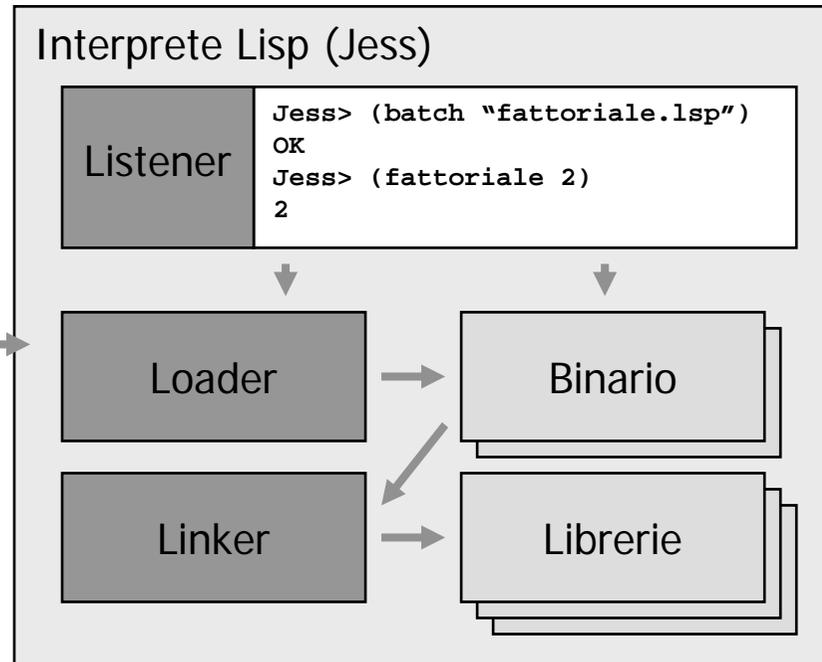


- Tipicamente:
 - si creano uno o più moduli (file) di *codice sorgente*;
 - si utilizzano *librerie* standard e/o *librerie* specifiche;
 - il *compilatore* traduce il codice sorgente in un *formato binario*;
 - il *linker* combina il prodotto della compilazione con le librerie.

Un interprete di programmi

Codice sorgente

```
;; File fattoriale.lsp
(defun fattoriale (n)
  (if (eq n 0)
      ;; then
      (return 1)
      ;; else
      (return
       (* n
         (fattoriale
          (- n 1))))))
)
```



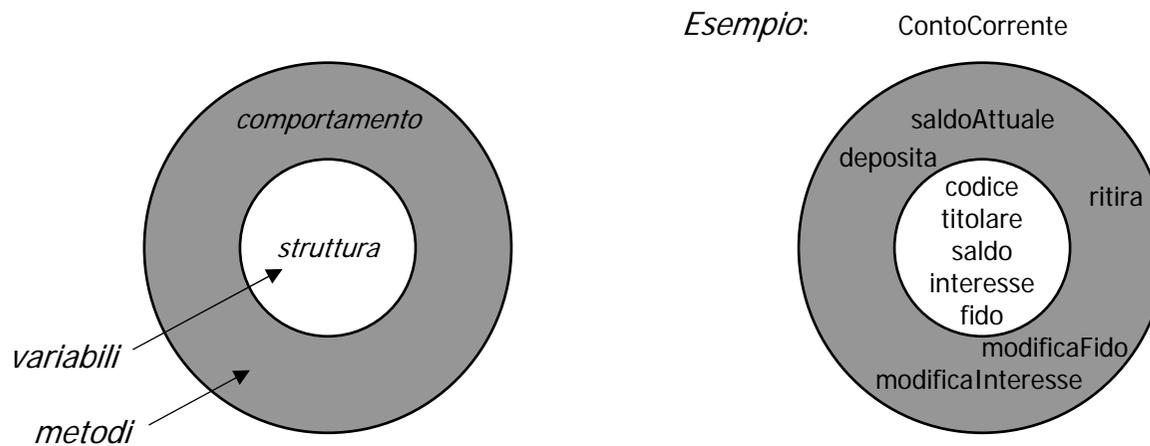
- Tipicamente:
 - non si produce un programma eseguibile indipendente
 - il codice sorgente viene tradotto in un formato binario
 - il collegamento tra 'pezzi' di programma viene fatto 'al volo'
 - le operazioni effettive vengono eseguite dall'*interprete*

2

Il modello astratto di Java: programmazione ad oggetti

Un modello ad oggetti

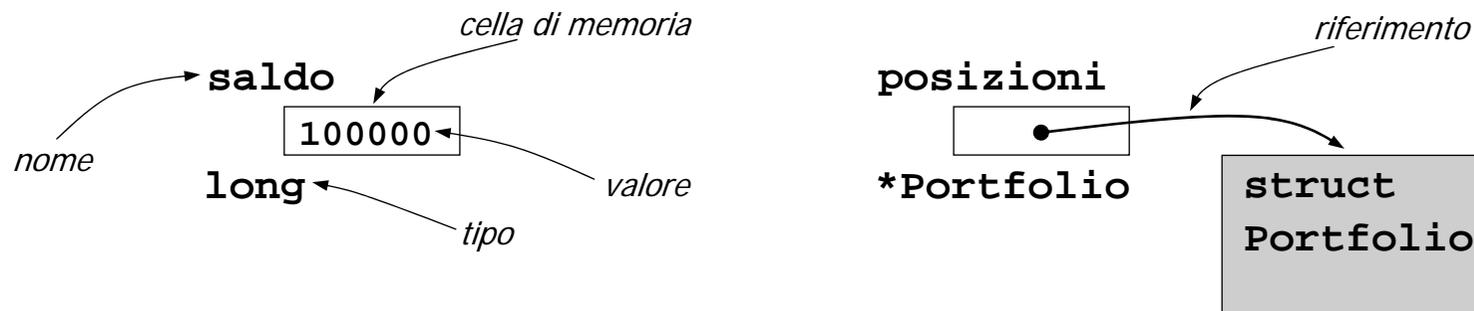
- L'idea informale:
rappresentare le entità della programmazione come aggregazioni di *variabili e metodi*



- Agli inizi, qualcuno ha suggerito l'idea di *circuiti integrati software*

Modello ad oggetti: variabili e valori

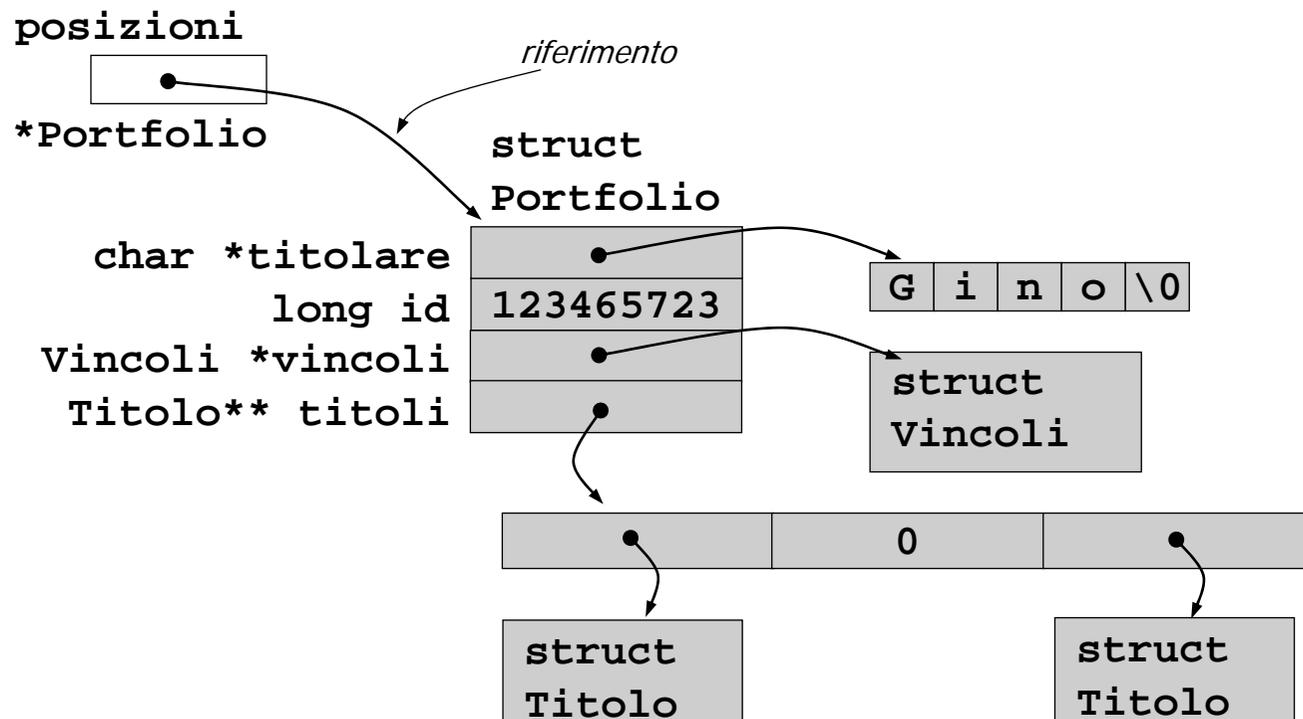
- “Una variabile è una cella di memoria”
 - (sempre ricordare)
 - la cella ha una *dimensione* prestabilita (p.es. 4 bytes)
 - di solito ogni variabile ha un *tipo* (in alcuni linguaggi no – es. Lisp/Jess)



- Vi sono due categorie principali di variabili
 - le variabili che contengono un *valore immediato* (p.es. `int`, `long`, `char`, etc.)
 - le variabili che contengono un *riferimento* (p.es. un puntatore)

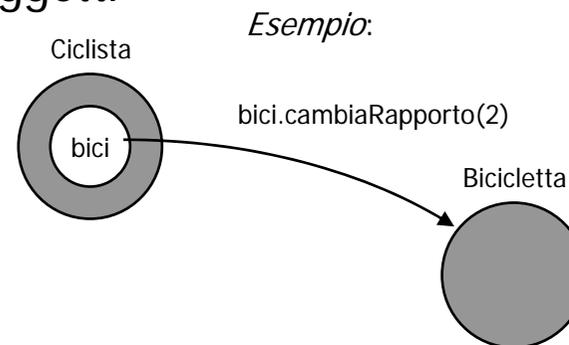
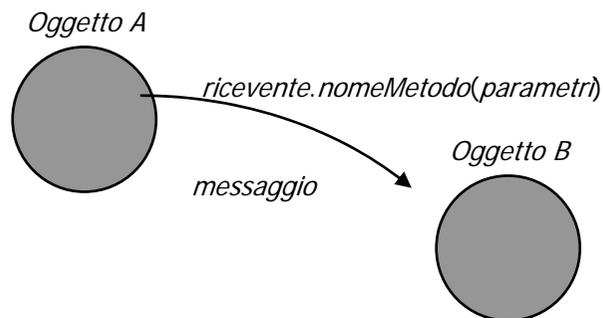
Modello ad oggetti: array e strutture

- Le variabili (le celle) possono essere aggregate in:
 - strutture: tipi eterogenei, dimensione prefissata
 - array: tipi omogenei, dimensione variabile



Scambio di messaggi

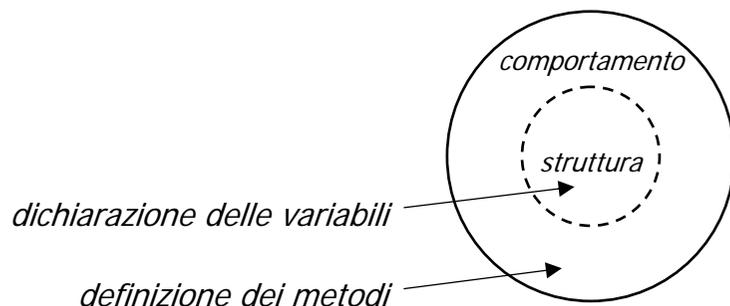
- Il flusso dell'esecuzione viene visto come un flusso di *messaggi* scambiati tra oggetti
- In generale gli oggetti 'vedono':
 - la loro propria struttura interna
 - la parte visibile (interfaccia?) degli altri oggetti



- Componenti di un *messaggio*:
 - un riferimento al ricevente;
 - il nome del metodo;
 - i parametri.

Classi

- Descrivere gli oggetti uno ad uno è poco vantaggioso. Infatti:
 - ogni oggetto richiederebbe una descrizione specifica;
 - la grande varietà ostacolerebbe la comunicazione.
- Al contrario, una *classe* è uno **schema generale** per la creazione di oggetti simili:
 - la *struttura* degli oggetti è descritta come *schema*;
 - il *comportamento* degli oggetti è definito in modo effettivo.



```
[ 'public' ] [ ( 'abstract' | 'final' ) ] 'class' class_name  
  '{'  
    // metodi e variabili  
    // sono definite qui  
  '}'
```

Classi e istanze: ContoCorrente

```

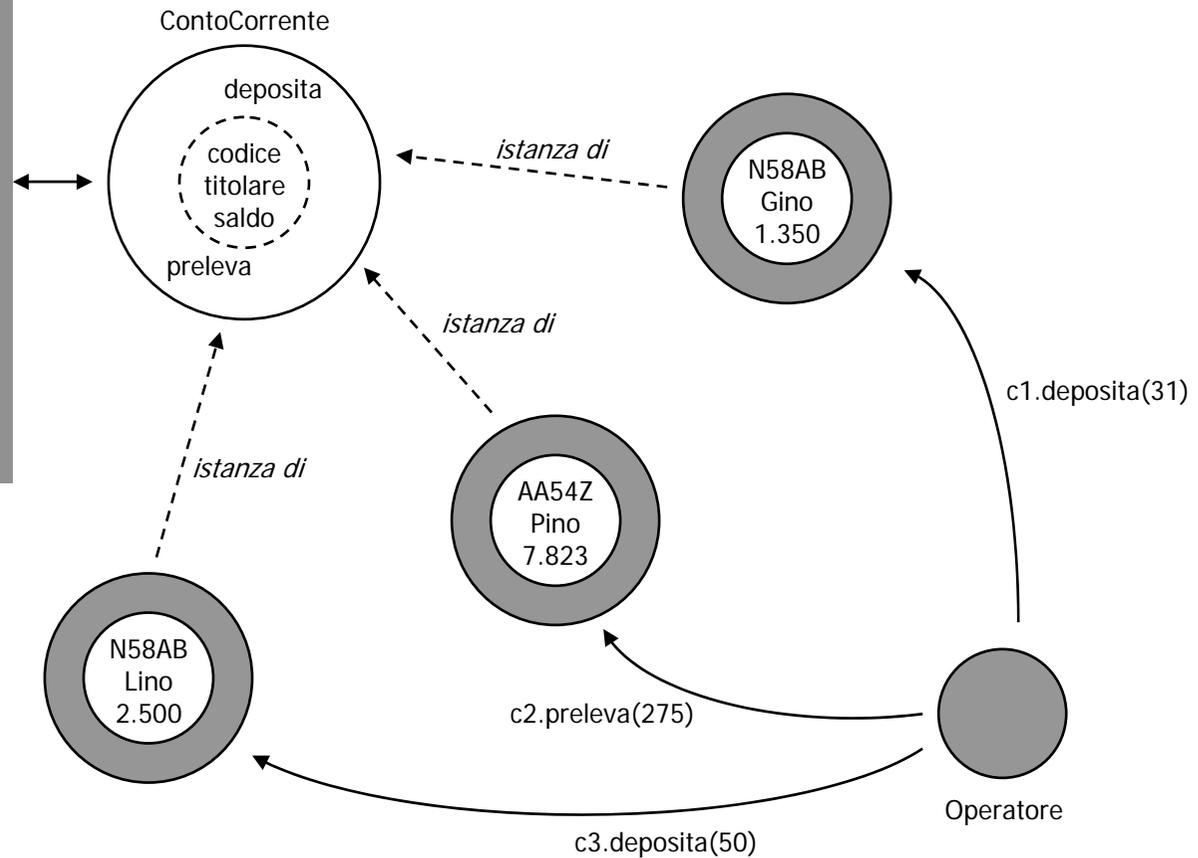
public class ContoCorrente {
    String codice;
    String titolare;
    float saldo;

    ...

    void deposita(float somma) {
        saldo += somma;
    }

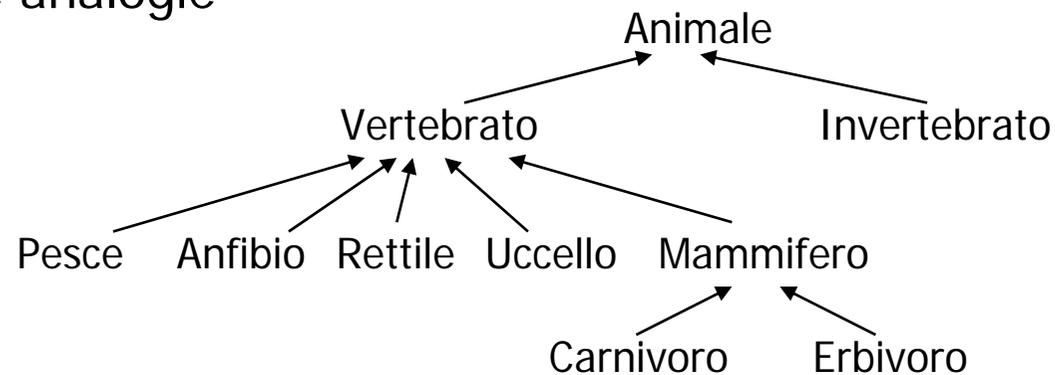
    void preleva(float somma) {
        saldo -= somma;
    }
}

```

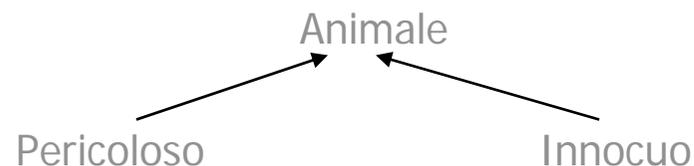


Ereditarietà

- L'idea di base è quella di classificare gli oggetti mettendo a fattor comune le analogie

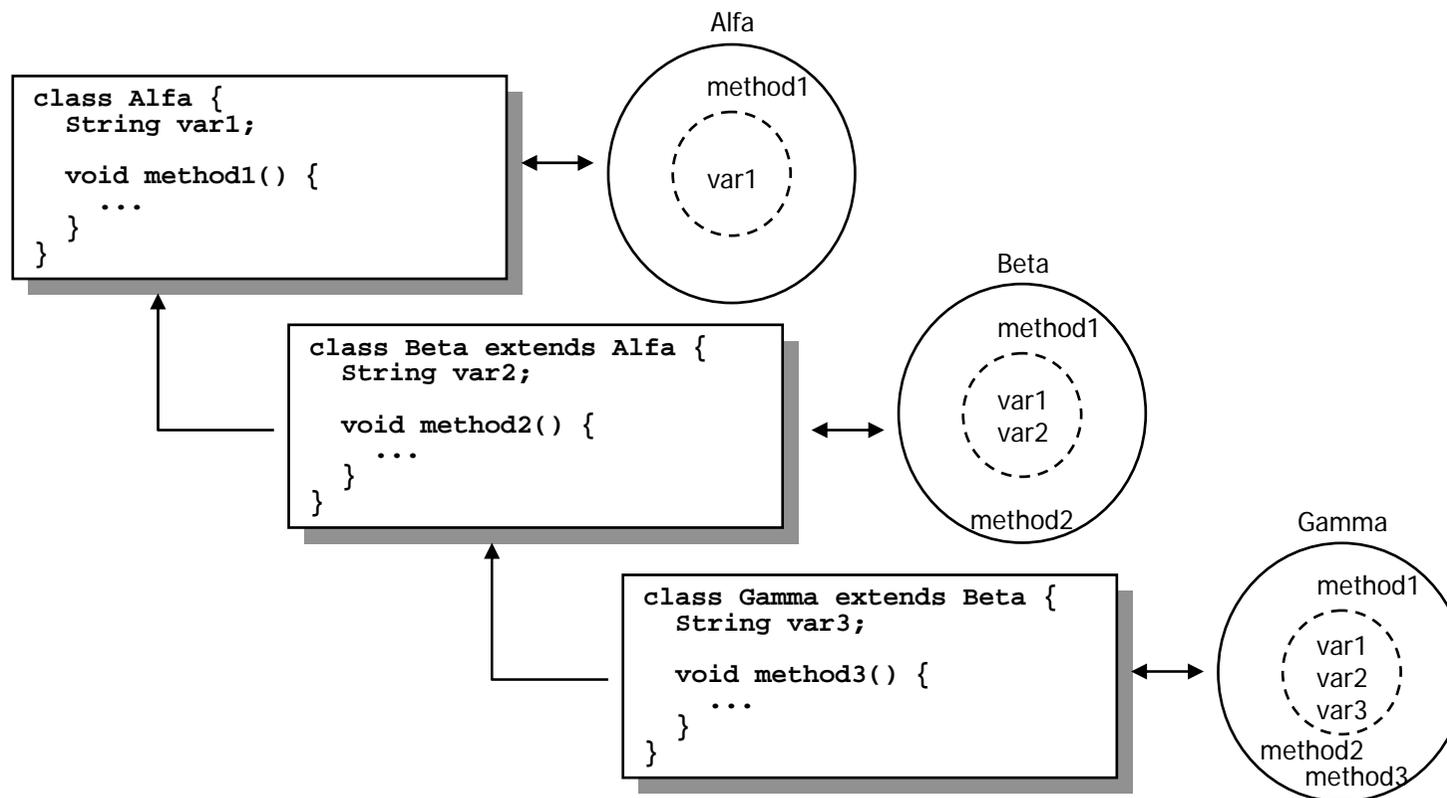


- Attenzione: la gerarchia dipende dallo *scopo*



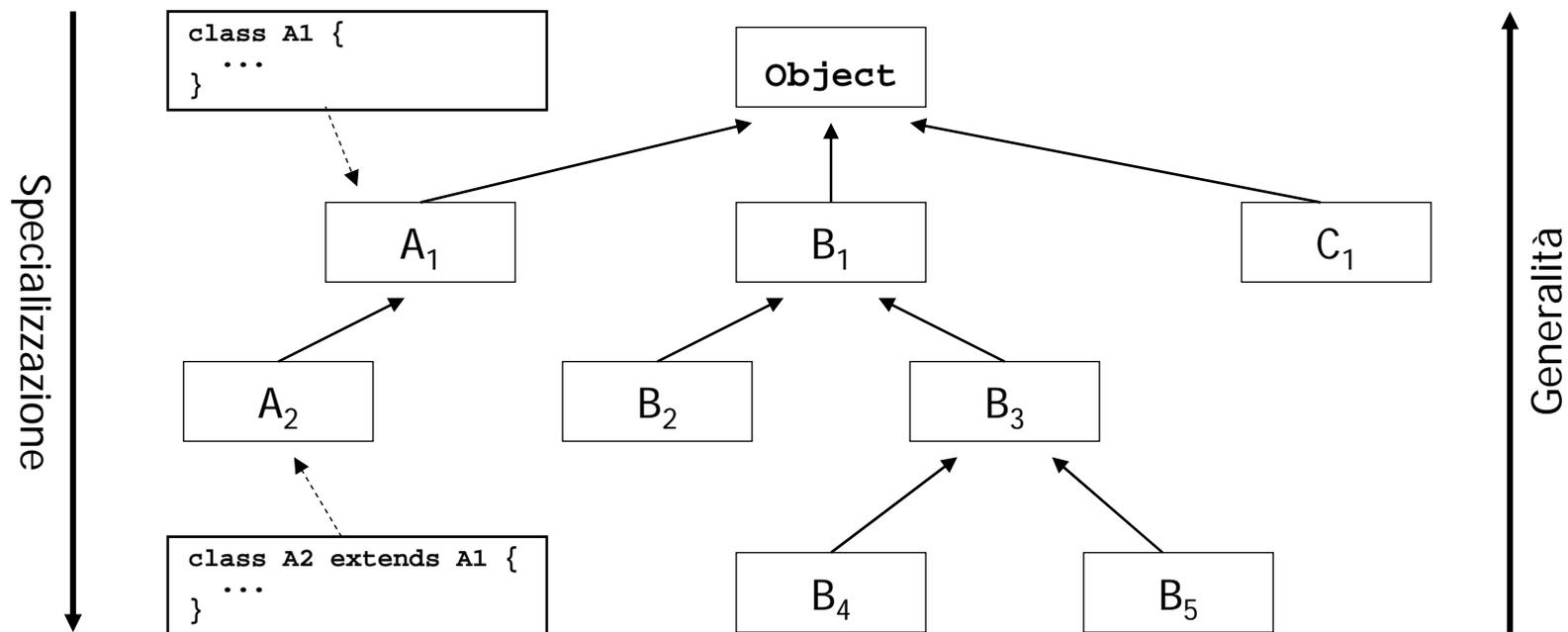
Ereditarietà in Java

- Le sottoclassi ereditano la *struttura* (intesa come *schema*) ed il *comportamento* dalle superclassi



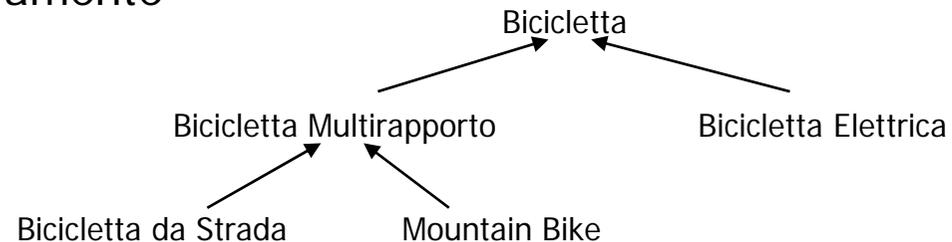
Struttura di classi

- Un programma Java è rappresentato da una *gerarchia* di classi
- La classe **Object** è la radice di tale gerarchia

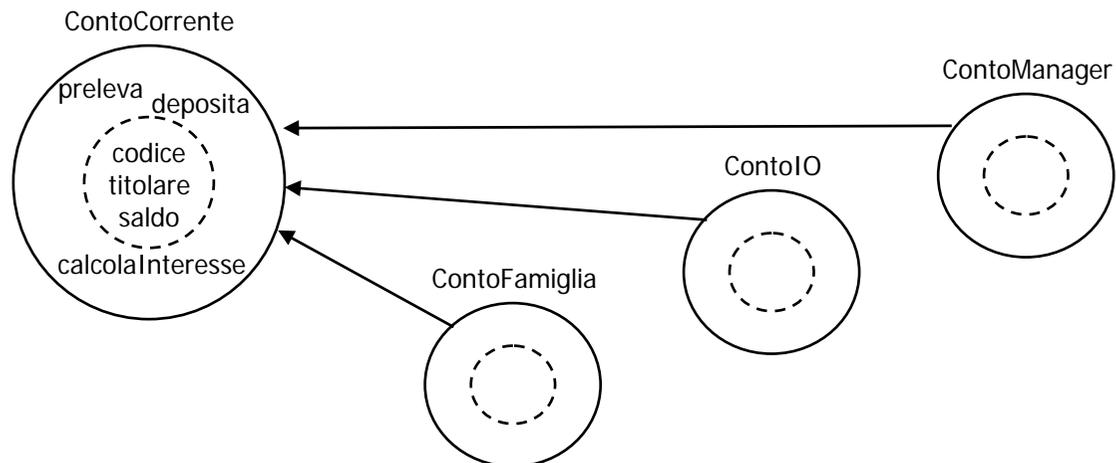


Uso di una struttura di classi

- Per *classificare*, cioè per strutturare le differenze
 - ma nella pratica, si usa raramente

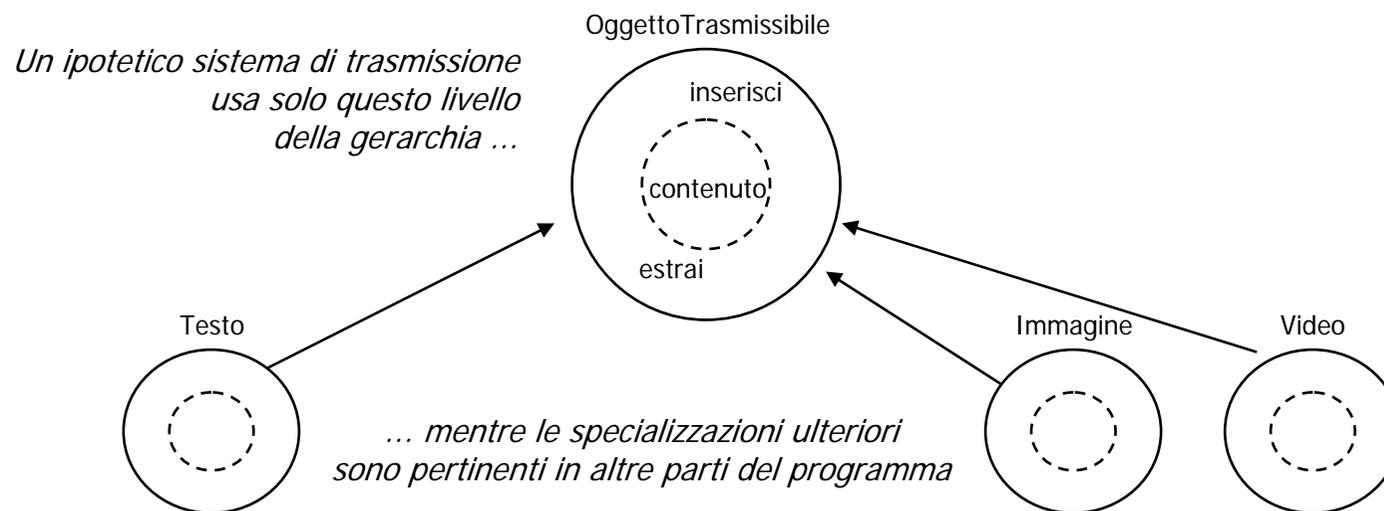


- Per *fattorizzare*, cioè per scrivere meno codice



Uso di una struttura di classi

- Per *disaccoppiare* aspetti diversi di uno stesso programma
 - e rendere lo stesso programma più facile da estendere



- a questo scopo in Java si usano spesso anche le *interfacce*

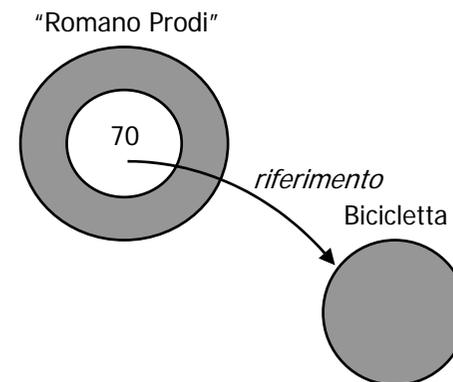
Tipo delle variabili

- In Java ogni variabile deve avere un *tipo* dichiarato
- In prima approssimazione, vi sono due categorie di tipi:
 - tipi primitivi: `int`, `float`, `byte`, `short`, `long`, `double`, `char`, `boolean`);
 - riferimenti ad oggetti (i.e. `object` o sottoclasse).

```
class Ciclista extends Persona {  
    int mediaKmAlGiorno;  
    Bicicletta bici;  
    ...  
    void allenamento() {  
        int rapporto;  
        ...  
        bici.cambiaRapporto(rapporto);  
    }  
}
```

valore → `int mediaKmAlGiorno;`

riferimento → `Bicicletta bici;`



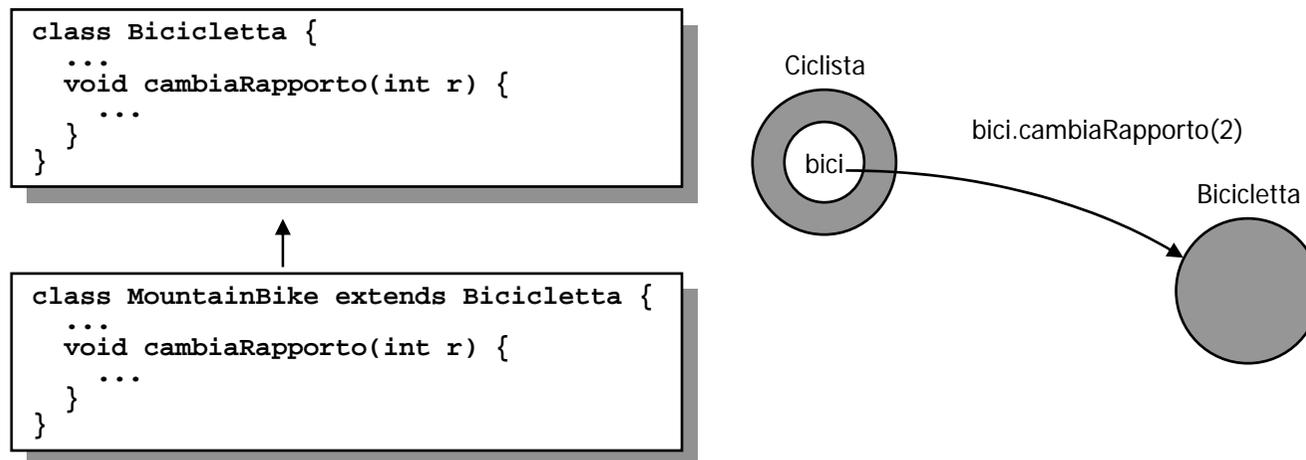
Come si scrive un metodo

- In prima approssimazione, un *metodo* Java è molto simile ad una *funzione* C:
 - dichiarazione formale: nome del metodo, tipo di valore ritornato, nome e tipo dei parametri formali;
 - gli operatori sono quelli del C (e.g. +, *, =, +=, ==, !=, ...)
 - le istruzioni di controllo sono quelle del C (i.e., if, else, switch, for, while, do)
 - l'istruzione di ritorno è `return`.

```
class Aritmetica {  
    long fattoriale(long arg) {  
        if (arg == 0) {  
            return 1L;  
        }  
        else {  
            long temp;  
  
            temp = arg * this.fattoriale(arg - 1);  
            return temp;  
        }  
    }  
}
```

Oscuramento dei metodi

- (Detto anche *overriding*)
- In una sottoclasse si può *oscurare* un metodo definito in una superclasse tramite una nuova definizione.

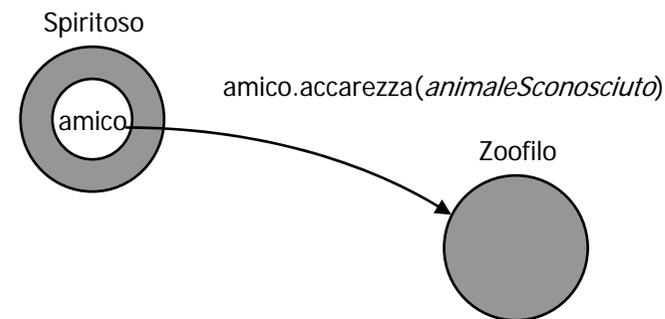


- La selezione viene effettuata (automaticamente) durante l'esecuzione in base al tipo effettivo

Specializzazione dei metodi

- (Detta anche *overloading*)
- La 'risposta' di un oggetto ad un messaggio può dipendere dal *tipo* dei parametri

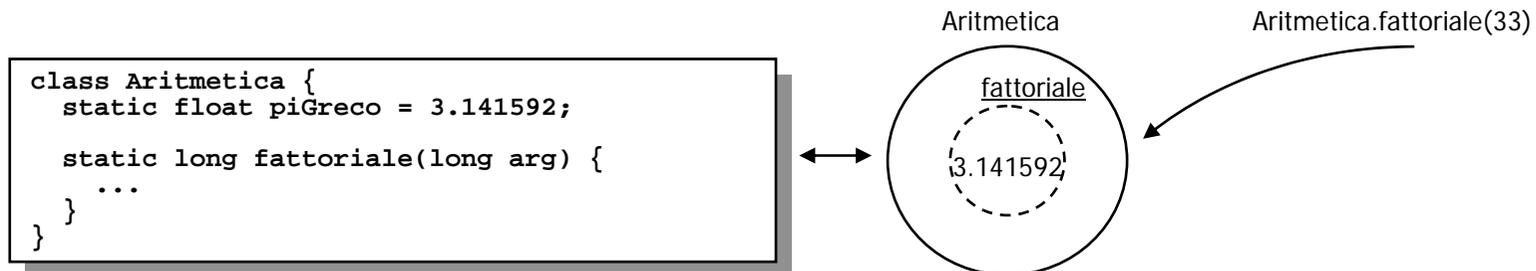
```
class Zoofilo extends Persona {  
    ...  
    void accarezza(Gattino g) {  
        affettuosamente  
    }  
    void accarezza(Tigre t) {  
        col pensiero  
    }  
    void accarezza(Object o) {  
        informati prima  
    }  
}
```



- La selezione è effettuata (automaticamente) in base al tipo dichiarato

Classi come oggetti

- In Java anche le classi sono oggetti
- Quindi anche le classi:
 - rispondono ai messaggi;
 - hanno variabili (di classe);
 - hanno metodi (di classe).
- Per definire le entità di classe si usa la parola chiave **static**

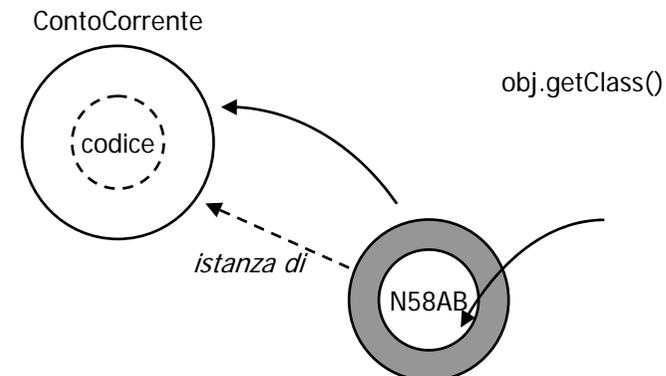


- Alle classi si può fare riferimento anche *per nome*

Nota: per chi conosce il C++

- In C++ si usa la parola chiave `static` con lo stesso significato
- Ma in C++, le classi **non** sono oggetti ma solo dichiarazioni a beneficio del compilatore
- Conseguenza notevole: in Java si può determinare a *run-time* la classe di un oggetto, in C++ no.

```
class Zoofilo extends Persona {  
    ...  
    void accarezza(Object obj) {  
        if (obj instanceof Serpente) {  
            no grazie  
        }  
        else if (obj instanceof Pesciolino) {  
            volentieri, ma come?  
        }  
        ...  
    }  
}
```



Protezioni

- L'accesso alle entità (metodi e variabili) di classe o di istanza può essere protetto
 - le verifiche di accessibilità sono effettuate durante la compilazione
- In Java vi sono quattro livelli di protezione e tre parole chiave:
 - **public** *accesso senza restrizioni*
 - **package** *solo nello stesso package*
 - **protected** *solo nella classe o nelle sottoclassi*
 - **private** *solo nella classe*

```
class ContoCorrente {
    private float saldo;
    ...
    float saldo(Persona richiedente) {
        if (richiedente.autorizzato())
            return saldo;
        else
            nega informazione
    }
}
```

```
class MountainBike
    extends Bicicletta {

    void cambiaRapporto(int r) {
        attivaProtezione();
        super.cambiaRapporto();
        disattivaProtezione();
    }

    private void attivaProtezione() {...}
    private void disattivaProtezione() {...}
}
```

Protezioni: esempio

```
class Automobile {
    // attributi di classe
    static int numeroDiRuote = 4;
    static public boolean haTarga = true;
    static private Vector archivio;

    // attributi di istanza
    public String marca;
    public String colore;
    public int annoDiImmatricolazione;
    public String targa;
    public int numero DiCavalli;
    protected Persona titolare;
    private int rapporto;

    // metodi di classe
    static protected void inserisciInArchivioAutoveicoli(Automobile a) {
        archivio.addElement(a);
    }
    ...
    // metodi di istanza
    public void cambiaRapporto(int r) {
        ...
        rapporto = r;
    }

    public int rapporto() {
        return rapporto;
    }....
    ...
}
```

Costruttori

- Un oggetto Java viene sempre creato tramite un metodo speciale detto *costruttore*
- Ogni classe ha uno o piu` costruttori
- Valgono i principi di *ereditarieta`* , di *specializzazione* e di *oscuramento*

```
class ContoCorrente {  
    ContoCorrente() {  
        generico  
    }  
    ContoCorrente(Persona richiedente) {  
        intestato  
    }  
}
```

```
class ContoAziendale  
    extends ContoCorrente {  
    ContoAziendale() {  
        // Oscuramento  
        generico  
    }  
    ContoAziendale(Azienda richiedente) {  
        // Specializzazione  
        intestato  
    }  
}
```



Pseudo-variabili

- Nei metodi di istanza, è possibile utilizzare le pseudo-variabili **this** e **super**
- **this** fa riferimento all'istanza stessa
 - si usa per evitare ambiguità

```
class Ciclista extends Persona {
    Bicicletta bici;

    boolean dimmiSeTua(Bicicletta bici) {
        return this.bici == bici;
    }
}
```

- **super** fa riferimento all'istanza stessa *come se appartenesse alla superclasse*
 - si usa per evitare ambiguità e per specializzare metodi

```
class ContoCorrente {
    float saldo;

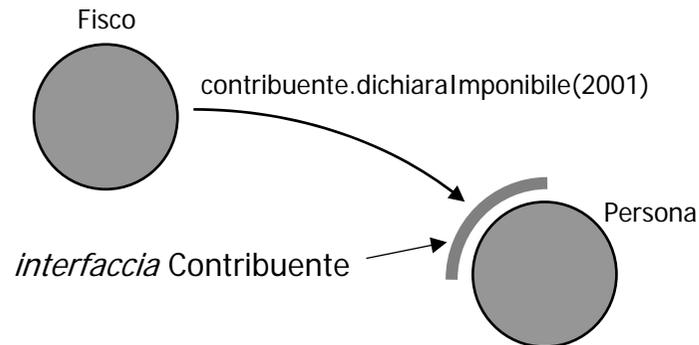
    void deposita(float somma) {
        saldo += somma;
    }
}
```

```
class ContoAziendale
    extends ContoCorrente {
    RegistroCassa registro;

    void deposita(float somma) {
        super.deposita(somma);
        registro.annotaEntrata(somma);
    }
}
```

Interfacce

- L'idea di base è quella di un *contratto* tra oggetti, una sorta di accesso limitato ma garantito



```
interface Contribuente {  
    float dichiaraImponibile(int anno);  
}
```

```
class Ingegnere extends Persona  
    implements Contribuente {  
    ...  
    float dichiaraImponibile(int anno) {  
        ...  
    }  
}
```

Uso delle interfacce

- Le interfacce contengono solo:
 - la dichiarazione di metodi (come *signature*);
 - la definizione di costanti
- Le interfacce sono un *tipo* valido per le variabili
- Le interfacce possono essere organizzate in strutture ad eredità multipla
- Una classe può implementare un numero qualsiasi di interfacce

```
class Ingegnere extends Persona
  implements Contribuente,
             MembroOrdineIngegneri,
             PersonaTipicamenteNoiosa,
             ... {
  ...
}
```

Tipi in Java

- I tipi di dati in Java sono organizzati in quattro categorie:
 - tipi primitivi (`boolean`, `byte`, `char`, `int`, `long`, `short`, `float`, `double`)
 - le classi (i.e. `Object` e tutte le sue sottoclassi)
 - le interfacce
 - gli `array` (dei tre tipi precedenti)
- Non esistono invece i tipi:
 - puntatori
 - funzioni

Array in Java

- Gli array in Java sono *oggetti* di un tipo particolare:
 - sono formati da celle contigue e di tipo omogeneo (come in C)
 - hanno una loro classe (e.g. `int[]`, `char[][]`, etc.)
 - hanno un attributo **length**
 - prevedono il controllo *run-time* degli accessi (a differenza del C)

```
class Portfolio {  
    long creaTitoli(int n) {  
        Titolo[] titoli = new Titolo[n];  
        int[] ids = new long[n];  
        for(int i = 0; i < titoli.length; i++) {  
            titoli[i] = new Titolo();  
            ids[i] = titoli[i].getId();  
        }  
    }  
}
```

Gestione delle eccezioni

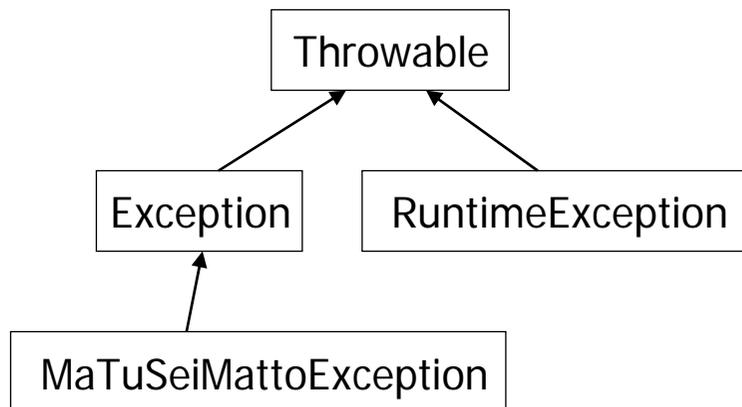
- La gestione delle eccezioni in Java è un sistema per governare il flusso dell'esecuzione a seguito di errori
 - di fatto, sostituisce l'unico uso sensato della **goto**
 - (che in Java non esiste)

```
class Spiritoso extends Persona {
    ...
    void proponi(Zoofilo amico) {
        try {
            amico.accarezza(...);
        }
        catch (MaTuSeiMattoException e) {
            // Evidentemente, non gradisce
        }
        ...
    }
}
```

```
class Zoofilo extends Persona {
    ...
    void accarezza(Object obj) {
        if (obj instanceof ScorpioneVelenoso) {
            throw new MaTuSeiMattoException();
        }
        ...
    }
}
```

Eccezioni

- Le eccezioni sono organizzate come una gerarchia di classi
- L'abbinamento **throw** / **catch** si stabilisce in base al *tipo*
- Due tipi fondamentali:
 - *checked* (sottoclassi di `Exception`)
gestione obbligatoria, controllata al momento della compilazione
 - *unchecked* (sottoclassi di `RuntimeException`)
gestione facoltativa



```
class Zoofilo extends Persona {
    ...
    void accarezza(Object obj)
        throws MaTuSeiMattoException {
        if (obj instanceof ScorpioneVelenoso) {
            throw new MaTuSeiMattoException();
        }
        ...
    }
}
```