# Intelligenza Artificiale

Breve introduzione a Java
(ed alla rappresentazione
ad oggetti)

Marco Piastra



## Argomenti

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





## 1

Modello astratto e modello
implementativo
nei linguaggi di programmazione





## Il progetto di un linguaggio

Da tener presente:

- un 'linguaggio di programmazione' è progettato in riferimento ad *modello implementativo* specifico;
- il progetto del linguaggio include un rilevante aspetto teorico, cioè un *modello astratto* (e.g. il concetto di *variabile*);
- 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).

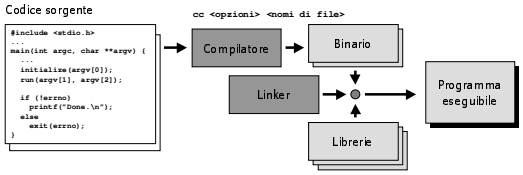




## Origini di Java e C (e C++)

- I due linguaggi hanno origini diverse
- Il C (ed il C++) derivano dagli studi sui linguaggi per la realizzazione di sistemi operativi (e.g. Unix) e software di base (e.g. database server)
- Java deriva (in gran parte) dai linguaggi sviluppati per l'intelligenza artificiale
- Infatti, ad esempio:
  - anche se la *sintassi* assomiglia a quella del C++;
  - il *modello ad oggetti* e l'idea della *macchina virtuale* e del *bytecode* provengono dallo Smalltalk-80;
  - l'idea dei *package* proviene dal Common Lisp.

## Creare un programma in C

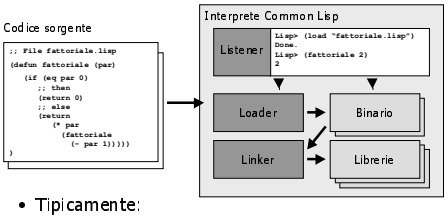



- Tipicamente:
  - si creano uno o più moduli (file) di *codice sorgente*;
  - si utilizzano *librerie* standard e/o *librerie* specifiche;
  - il *compilatore* traduce il codice sorgente in un *formato binario*;
  - il *linker* combina il prodotto della compilazione con le librerie.





## Un interprete di programmi




- 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*.





## Differenze di paradigma

- In C:
  - la sintassi è concepita per favorire i controlli di correttezza;
  - le entità principali sono *statiche* (e.g. variabili globali, array, funzioni) e si usano tecniche esplicite per la gestione della memoria dinamica (i.e. `malloc` e `free`);
  - un programma compilato ed eseguibile è di fatto *immutabile*.
- In un linguaggio interpretato (e.g. Common Lisp):
  - la sintassi è concepita per facilitare la scrittura di codice;
  - le entità principali sono *dinamiche*, e.g. *liste* di elementi;
  - la composizione di un programma è definita dallo *stato* dell'*interprete* al momento dell'esecuzione.





## 2

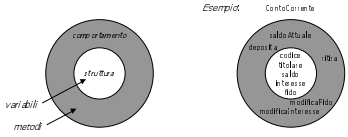
## Il modello astratto di Java: rappresentazione ad oggetti





## Un modello ad oggetti

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



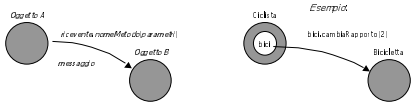

- Agli inizi, qualcuno ha suggerito l'idea di *circuiti integrati software*





## Scambio di messaggi

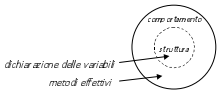
- Il flusso dell'esecuzione viene visto come un flusso di *messaggi*




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

## Classi

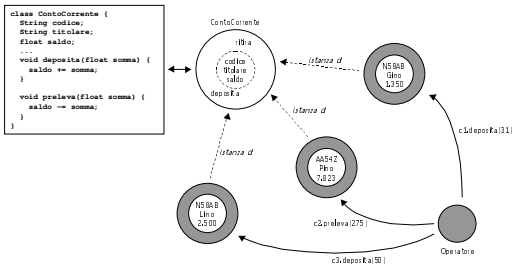
- Descrivere gli oggetti uno ad uno è poco vantaggioso. Infatti:
  - ogni oggetto richiederebbe una descrizione specifica;
  - la grande varietà ostacolerebbe la comunicazione.
- Al contrario, una *classe* è uno **schema generale** per la creazione di oggetti simili:
  - la *struttura* degli oggetti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








## Classi e istanze: ContoCorrente

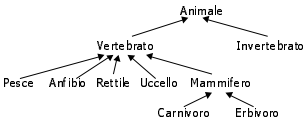







## Ereditarietà

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



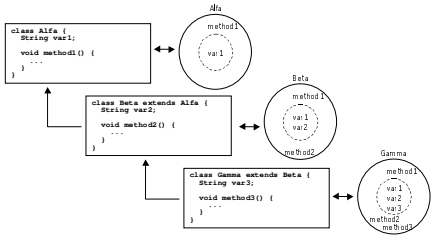

- Attenzione: la gerarchia dipende dallo *scopo*





## Ereditarietà in Java

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

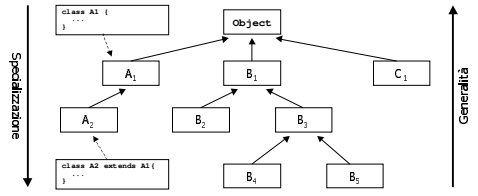







## Struttura di classi

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



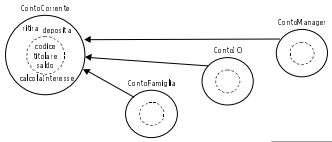





## Uso di una struttura di classi

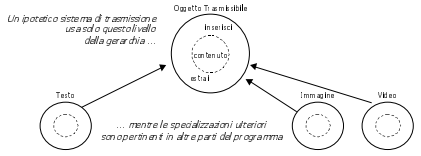
- Per *classificare*, ovvero strutturare le differenze
  - nella pratica, si usa raramente

- Per *fattorizzare*, ovvero 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 si usano spesso anche le *interfacce*



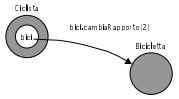


## Oscuramento dei metodi

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

```
class Bicicletta {
  ...
  void cambiaRapporto(int r) {
    ...
  }
}
```

```
class MountainBike extends Bicicletta {
  ...
  void cambiaRapporto(int r) {
    ...
  }
}
```




- La compatibilità con la *segnatura* definita nella superclasse viene preservata (*polimorfismo*)

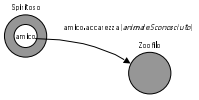




## Specializzazione dei metodi

- (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 pericolo
  }
  void accarezza(Object o) {
    informati prima
  }
}
```



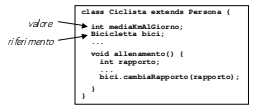

- La selezione è automatica in base alla *segnatura*

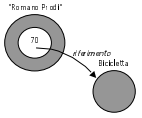




## Tipo delle variabili

- 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).








## 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;
    }
  }
}
```



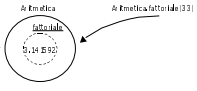


## Classi come oggetti

- 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`

```
class Aritmetica {
  static float piGreco = 3.141592;

  static long fattoriale(long arg) {
    ...
  }
}
```

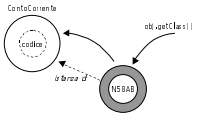



- Alle classi si può fare riferimento *per nome*

## 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?
    }
    ...
  }
}
```




Marco Pistira



## Protezioni

- L'accesso alle entità di classe o di istanza può essere protetto
- In Java vi sono quattro livelli di protezione e tre parole chiave:
  - **`public`** *accesso senza restrizioni*
  - **`protected`** *solo nello stesso package o nelle sottoclassi*
  - *package* *solo nello stesso package*
  - **`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() {...}
}
```

Marco Pistira



## 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;
  }....
  ...
}
```

Marco Pistira



## 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
  }
}
```

Marco Pistira



## 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;
  }
}
```

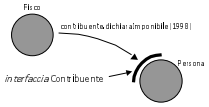
```
class ContoAziendale {
  RegistroCassa registro;
  void deposita(float somma) {
    super.deposita(somma);
    registro.annotaEntrata(somma);
  }
}
```

Marco Pistira



## Interfacce

- L'idea di base è quella di un *contratto* tra oggetti, una sorta di visibilità limitata




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

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

Marco Pistira

## 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 (**`char`**, **`boolean`**, **`byte`**, **`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





## 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
    }
    ...
  }
}
```

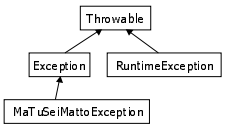
```
class Zoofilo extends Persona {
  ...
  void accarezza(Object obj) {
    if (obj instanceof ScorpioneVelenoso) {
      throw new MaTuSeiMattoException();
    }
    ...
  }
}
```





## Eccezioni

- Le eccezioni sono organizzate come una gerarchia di classi;
- L'abbinamento **`throw`** / **`catch`** si stabilisce in base al *tipo*;
- Esistono eccezioni *checked* (i.e. che vengono controllate dal compilatore) ed eccezioni *uncheked*.




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





## 3

Il modello implementativo di Java: macchina virtuale



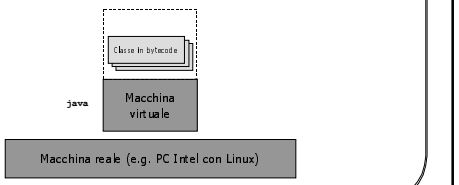


## Modello implementativo

- Le classi compilate in *bytecode* sono portabili su qualsiasi macchina per cui esista una macchina virtuale Java
- Il programma finale viene assemblato a run-time (i.e. *linking* tra le classi) e dipende quindi dall'ambiente di esecuzione
- Quindi:
  - si possono costruire programmi indipendenti dalla piattaforma hardware e software (purchè esista una macchina virtuale)
  - le classi compilate in *bytecode* assomigliano a componenti software ricombinabili a piacere

## *Bytecode*

- **non** è un codice direttamente eseguibile
- ma è un codice (binario) per una macchina virtuale (cioè un *interprete*) che si incarica della esecuzione effettiva








## *Bytecode* (2)

metodo in Java

```
...
void spin() {
  int i;
  for (i = 0; i < 100; i++) {
    // Loop body is empty
  }
}
...
```

bytecode compilato (in forma simbolica)

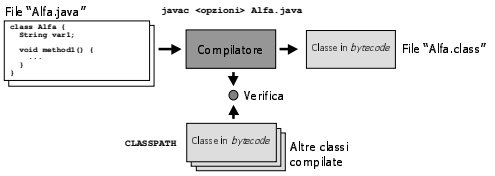
```
...
0    iconst_0          // Push int constant 0
1    istore_1          // Store into local variable 1 (i=0)
2    goto 8            // First time through don't increment
5    iinc 1 1          // Increment local variable 1 by 1 (i++)
8    iload_1           // Push local variable 1 (i)
9    bipush 100        // Push int constant 100
11   if_icmplt 5       // Compare and loop if less than (i < 100)
14   return            // Return void when done
...
```





## Compilazione

- In Java la compilazione è:
  - parziale (ciascuna classe viene compilata separatamente)
  - con verifica (di correttezza sintattica e di tipo)

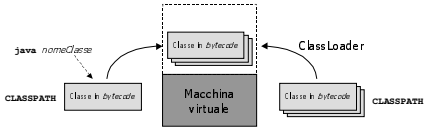







## Programma

- Un programma Java è un insieme di classi
- Generalmente tale insieme è *chiuso*, ma è potenzialmente *aperto*
- La composizione di questo insieme è stabilita dalla macchina virtuale a **run-time**

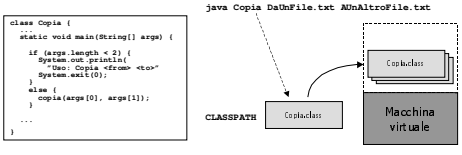







## Bootstrap dell'esecuzione

- La macchina virtuale Java esegue come prima cosa il metodo `main` della classe di *bootstrap*

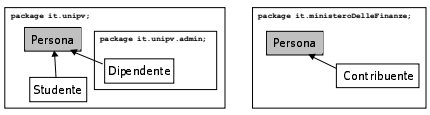







## Package

- L'idea originale (Common Lisp) è quella di un sistema per organizzare i *simboli*
- nomi comuni come ad esempio "Persona" sono facilmente usati in più programmi
- i package possono essere utilizzati per *contestualizzare* il nome di una classe o interfaccia



## Package (2)

- L'idea originale (Common Lisp) è di un sistema di organizzazione dei simboli:
  - si usa quindi una notazione qualificata composta da un *prefisso* e da un *simbolo*

    esempio: `it.unipv.Persona`
  - la pseudo-istruzione `package` definisce il package di riferimento
  - la pseudo-istruzione `import` serve ad abbreviare i riferimenti simbolici

```
package it.unipv.esercitazioniAI;
import it.unipv.Persona;

class StudenteDelCorsoAI extends Persona {
  ...
}
```





## Packages & directories

- In Java il sistema dei package è anche un modo per organizzare le classi (sia .java che .class):
  - la struttura del prefisso viene trasformata in percorso nel *file system* :

    `it.unipv.Persona`

    diventa

    `$CLASSPATH/it/unipv/Persona` (.java o .class)
  - i *classloader* della macchina virtuale (i.e. `java`) e del compilatore (i.e `javac`) usano le stesse convenzioni.





## 4

Altri aspetti particolari di Java:
*garbage collection*, *multithreading*, programmazione distribuita





## Allocazione dinamica della memoria

- In C la memoria dinamica deve essere gestita in modo esplicito
- Errori comuni:
  - mancata allocazione
  - mancata deallocazione
- In C++ gli oggetti sono allocati dinamicamente
- In C++ esistono delle entità standard per facilitare la gestione (*costruttori*, *distruttori*)





## Allocazione dinamica in Java

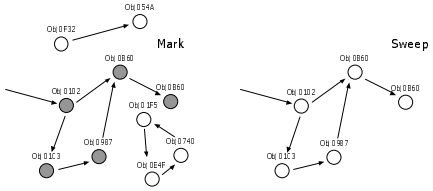
- In Java **tutte** le entità sono allocate dinamicamente (e.g. non esistono *variabili globali*)
- Tuttavia, solo l'allocazione della memoria (intesa come creazione di oggetti) è gestita esplicitamente
- La deallocazione è gestita dalla macchina virtuale (*garbage collection*)





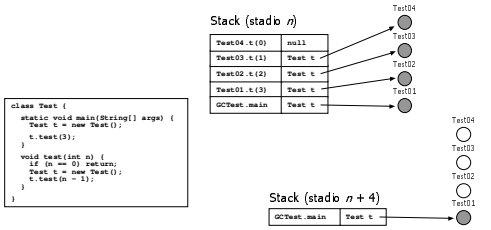
## Mark and Sweep

- Un algoritmo di garbage collection che prevede l'interruzione dell'esecuzione
- Si usa quando gli oggetti sono molti e globalmente accessibili



## Mark and Sweep (2)

- Il principale punto di partenza per la fase di mark è lo *stack*

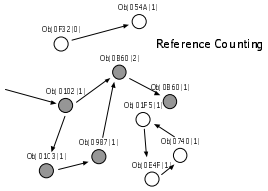







## Reference Counting

- Un algoritmo di garbage collection che non prevede l'interruzione dell'esecuzione
- Si usa quando gli oggetti sono pochi e/o non raggiungibili globalmente (e.g. sistema distribuito)

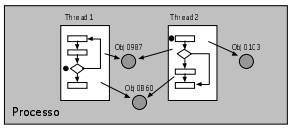







## Multithreading

- Un *thread* è un flusso di controllo sequenziale in un programma
  - un *thread* viene talvolta anche chiamato *lightweight process*
- Ogni *thread* possiede delle risorse private (e.g. stack e pc) ...
- ... tuttavia i *thread* di un programma condividono lo stesso spazio di indirizzamento

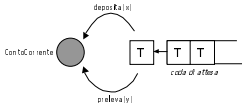







## Sincronizzazione

- Il multithreading non è utilizzabile in pratica senza la possibilità di *sincronizzare* i thread
- In Java ciascun oggetto può comportarsi come un *monitor*:
  - un *monitor* è un insieme di procedure relative ad una singola entità (i.e. un oggetto Java);
  - un *monitor* implementa il principio di mutua esclusione: solo un thread alla volta può eseguire una procedura.

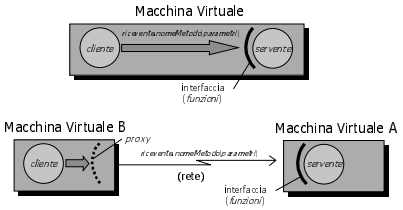







## Interfacce remote

- Il principio di base è quello di inviare messaggi a distanza

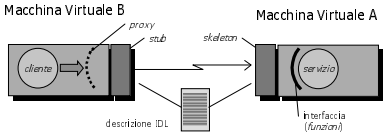






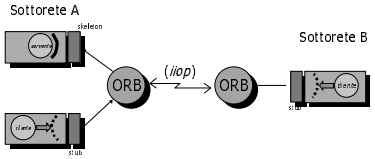

## Proxy & Skeleton

- L'effetto viene ottenuto usando due componenti software (solitamente generate automaticamente):
  - che simulano la presenza del servente in B
  - e trasmettono le chiamate remote in A



## CORBA

- Common Object Request Broker Architecture
- La comunicazione tra applicazioni non è diretta ma è mediata

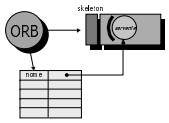







## Naming Service

- Il Naming Service serve a pubblicare oggetti *serventi* a beneficio dei potenziali *clienti*
  - gli oggetti servente vengono registrati in base ad un nome simbolico
  - i clienti possono 'agganciare' i *serventi* usando questo nome
  - il Naming Service è solitamente realizzato da un processo separato








## Modello industriale

- Java **non** nasce da un progetto accademico o non-profit
- I requisiti di progetto di Java sono quindi:
  - tecnici;
  - ingegneristici;
  - di contesto (cioè di mercato).
- I requisiti tecnici sono evidenti nel progetto del modello astratto
- I requisiti ingegneristici sono evidenti nel modello implementativo e nell'idea di piattaforma
- (Per capire i requisiti di mercato, si pensi a chi comanda il gioco)





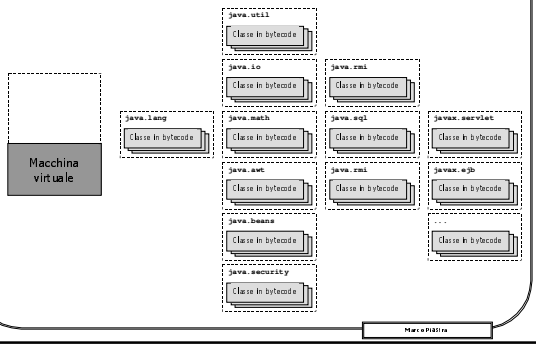
## L'idea di piattaforma

- Uno dei grossi problemi della costruzione software dal punto di vista ingegneristico è la grande quantità di aspetti dipendenti da una piattaforma specifica (e.g. Windows NT, Linux)
- In questo senso, la storia dei linguaggi di programmazione registra numerosi insuccessi:
  - il Pascal fu definito senza specificare le funzioni di I/O;
  - il C (ed il C++) hanno un insieme di librerie standard limitato (e.g. non c'è grafica o networking);
  - il modello astratto di Common Lisp e Prolog è poco adatto alla gestione grafica interattiva (e.g. multithreading).





## Java come piattaforma








## Per Ulteriori Approfondimenti

- Tutorial on-line
  - http://www.javasoft.com/docs/books/tutorial
  - Consigli:
    - » saltare (in prima lettura) la parte sulle applet
    - » seguire (come minimo): Learning the Java Language, Essential Java Classes, Collections.
    - » consigliati: Creating a GUI, Java Beans
- Libri
  - Arnold, K., Gosling, J., *The Java Programming Language - Second Edition*, Addison-Wesley, 1998.
  - Bishop, J., *Java Gently - Second Edition*, Addison-Wesley, 1998.