Breve introduzione a Java (2)

(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.

3

Il modello implementativo di Java: macchina virtuale

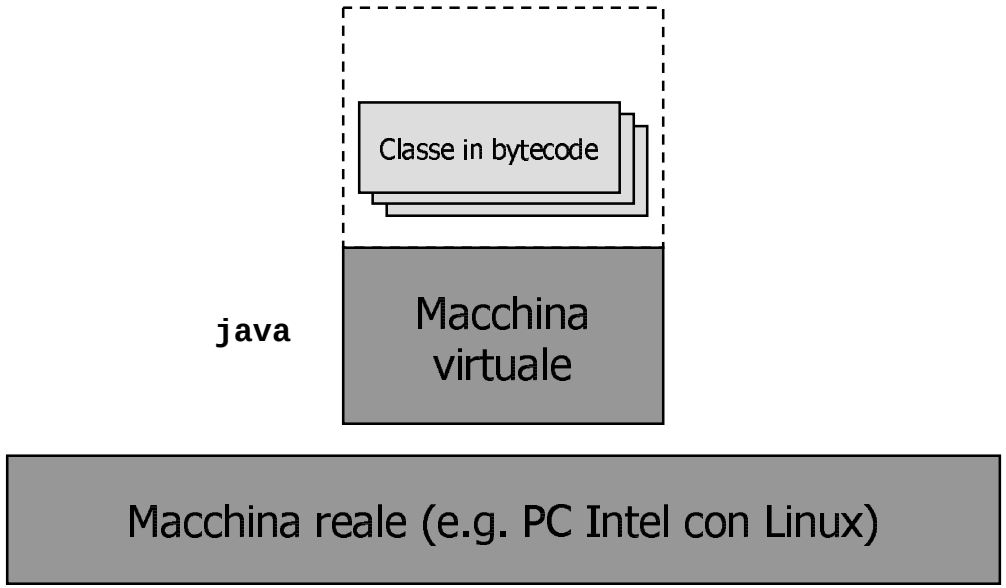
Modello implementativo

- Le classi compilate in *bytecode* sono portabili su qualsiasi macchina per cui esista una macchina virtuale Java (Java Virtual Machine – JVM)
- 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 dell'esecuzione effettiva



Bytecode (2)

metodo in Java

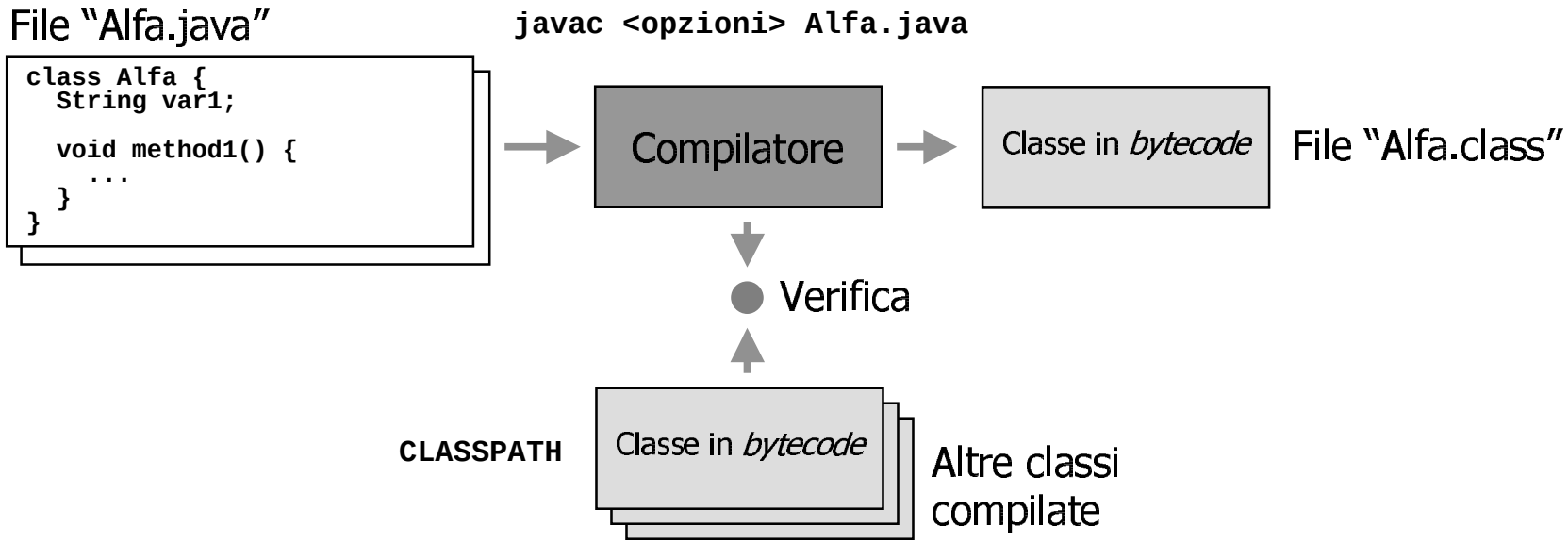
```
...
void spin() {
  int i;
  for (i = 0; i < 100; i++) {
    // Loop body is empty
  }
}
...
```

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

...
```

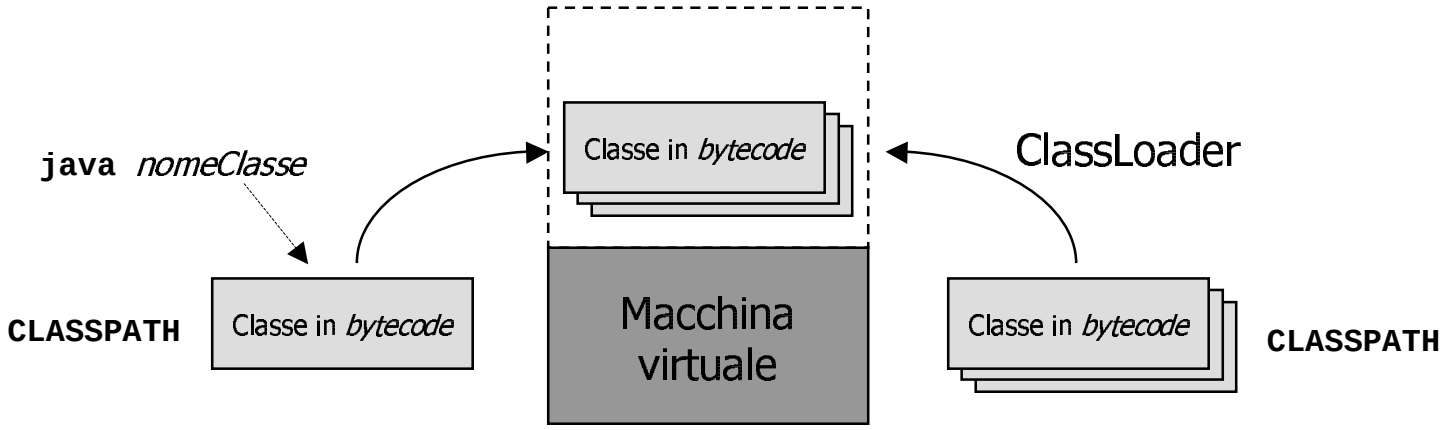
Compilazione

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



Programma

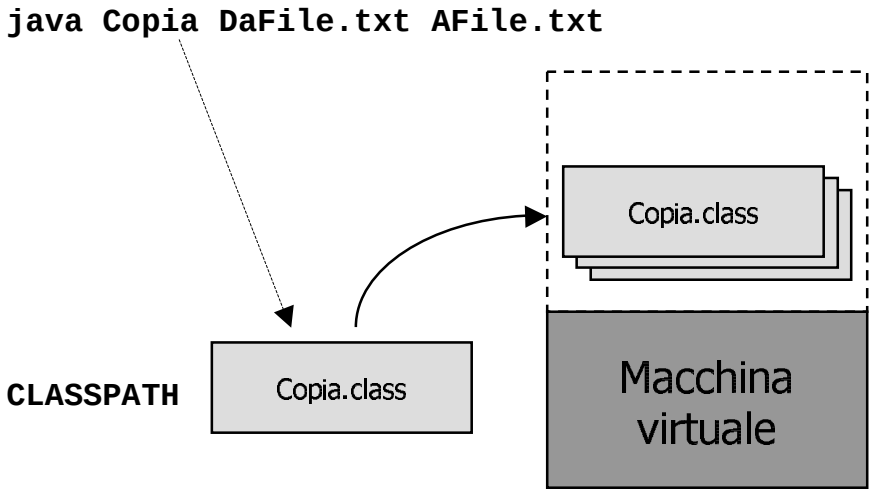
- Un programma Java è un insieme di classi
- Generalmente tale insieme è *chiuso*, ma è potenzialmente *aperto*
- La composizione di questo insieme è stabilita dalla macchina virtuale a **run-time**



Bootstrap dell'esecuzione

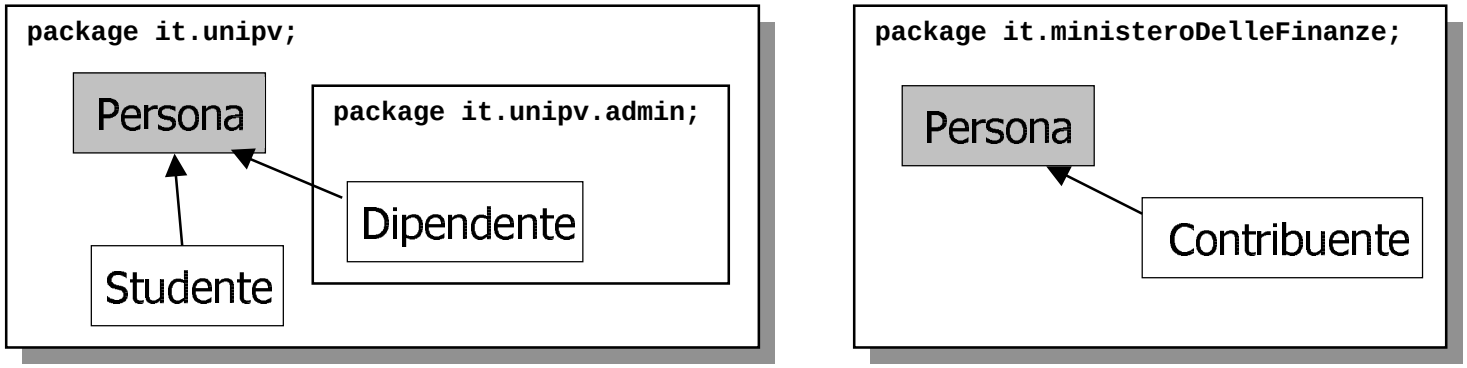
- La macchina virtuale Java esegue come prima cosa il metodo **`main`** della classe di *bootstrap*
- La strategia di caricamento delle altre classi può dipendere dall'implementazione della macchina virtuale

```
class Copia {
  ...
  static void main(String[] args) {

    if (args.length < 2) {
      System.out.println(
        "Uso: Copia <from> <to>"
      System.exit(0);
    }
    else {
      copia(args[0], args[1]);
    }

  ...
}
```



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 possone 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:
  - uso di una notazione qualificata composta da un *prefisso* e da un *simbolo*

    esempio: `it.unipv.Persona`

  - ogni file (classe o interfaccia) appartiene ad un package di riferimento
    - (il package di default è il package 'root')
  - all'inizio di ciascun file, 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

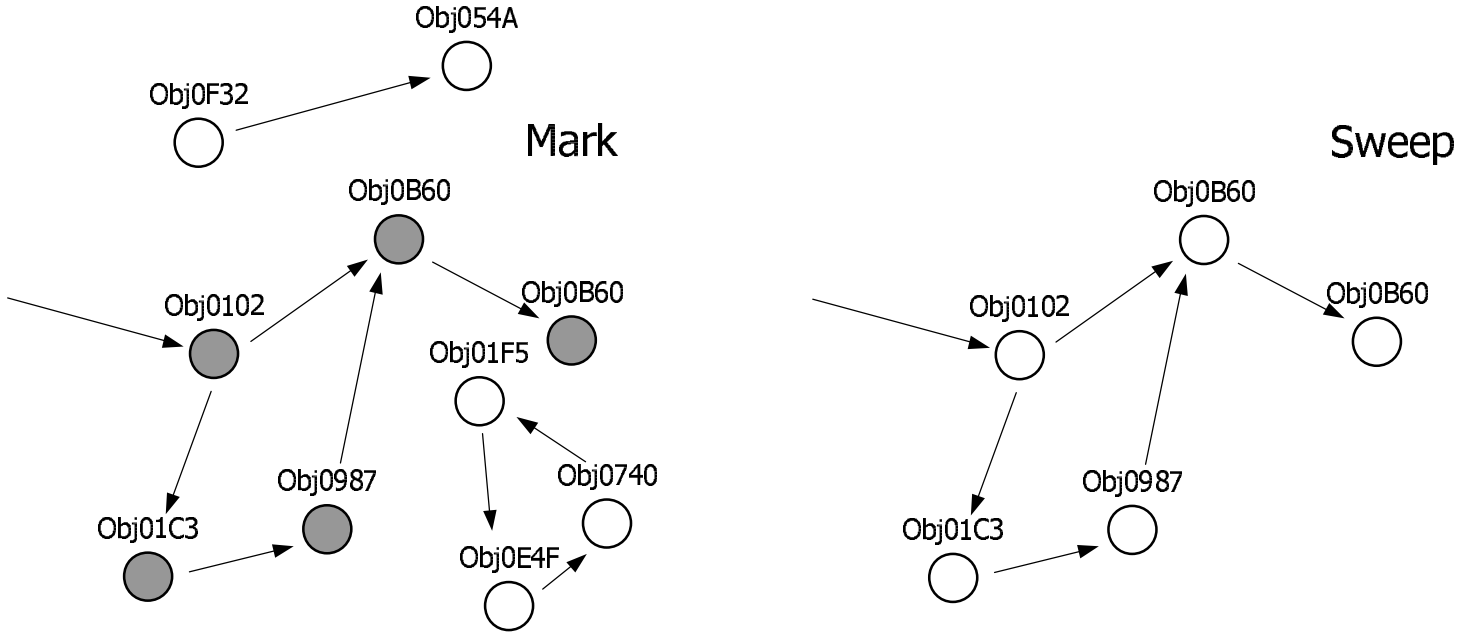
- In C la memoria dinamica deve essere gestita in modo esplicito
- Errori comuni:
  - mancata allocazione
  - mancata deallocazione
- In C++ gli oggetti sono allocati dinamicamente
- In C++ esistono delle entità standard per facilitare la gestione (*costruttori*, *distruttori*)

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*)

Garbage collection: *Mark and Sweep*

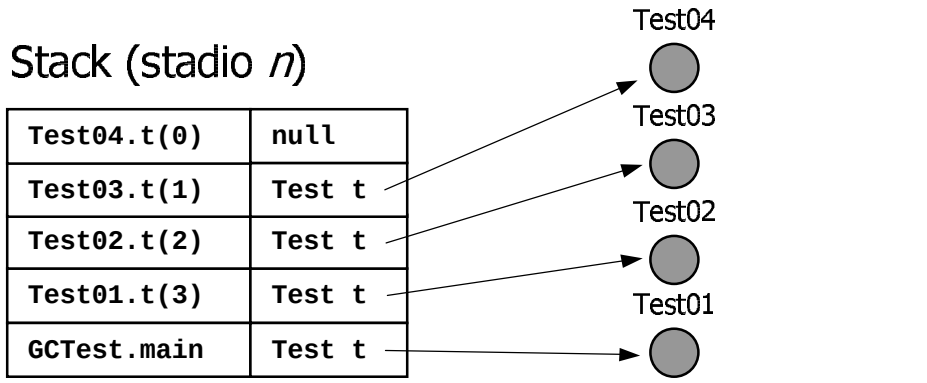
- Un algoritmo di garbage collection che prevede l'interruzione dell'esecuzione
  - in una prima fase (mark) si marcano tutti gli oggetti accessibili
  - in una seconda fase (sweep) si rimuovono gli oggetti non marcati (e si rimuove il marcatore dagli altri)
- Si usa quando gli oggetti sono molti e globalmente accessibili

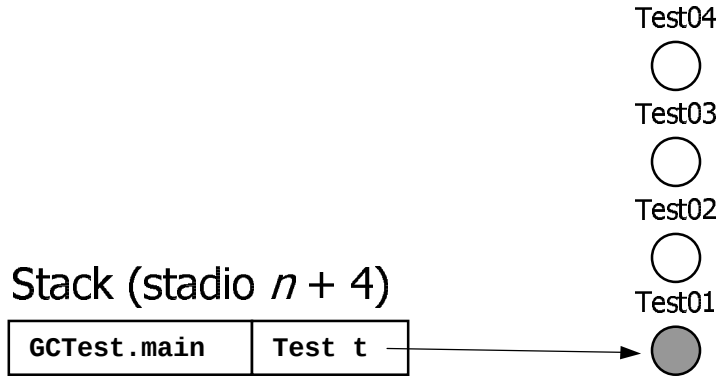


Mark and Sweep: accessibilità

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

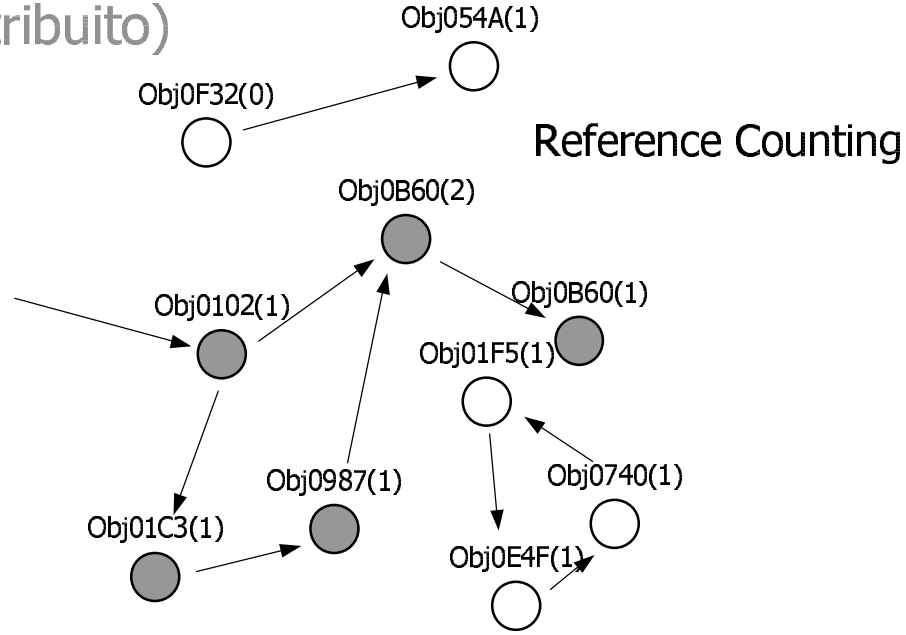
```
class Test {
  static void main(String[] args) {
    Test t = new Test();
    t.test(3);
  }
  void test(int n) {
    if (n == 0) return;
    Test t = new Test();
    t.test(n - 1);
  }
}
```





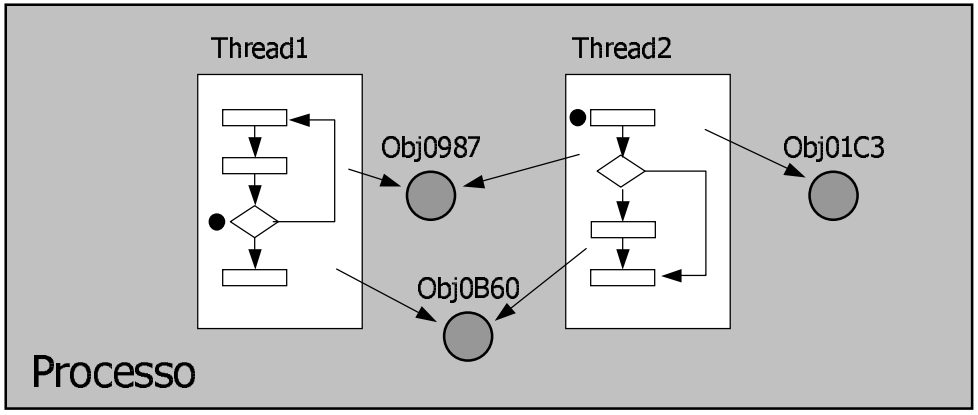
Garbage collection: *Reference Counting*

- Un algoritmo di garbage collection che non prevede l'interruzione dell'esecuzione
  - si incrementa/decrementa un contatore quando un oggetto viene referenziato/dereferenziato
  - vengono rimossi gli oggetti con contatore a zero
- Si usa quando gli oggetti sono pochi e/o non raggiungibili globalmente (e.g. sistema distribuito)



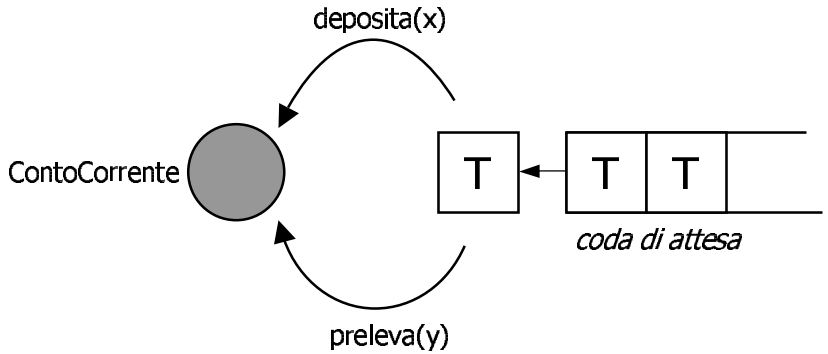
Multi-threading

- 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 (tipicamente stack e pc) ...
- ... tuttavia i *thread* di un programma condividono lo stesso spazio di indirizzamento



Sincronizzazione

- Il multi-threading 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
  - in Java è un *monitor* un oggetto che implementa almeno un metodo **`synchronized`**



Modello industriale

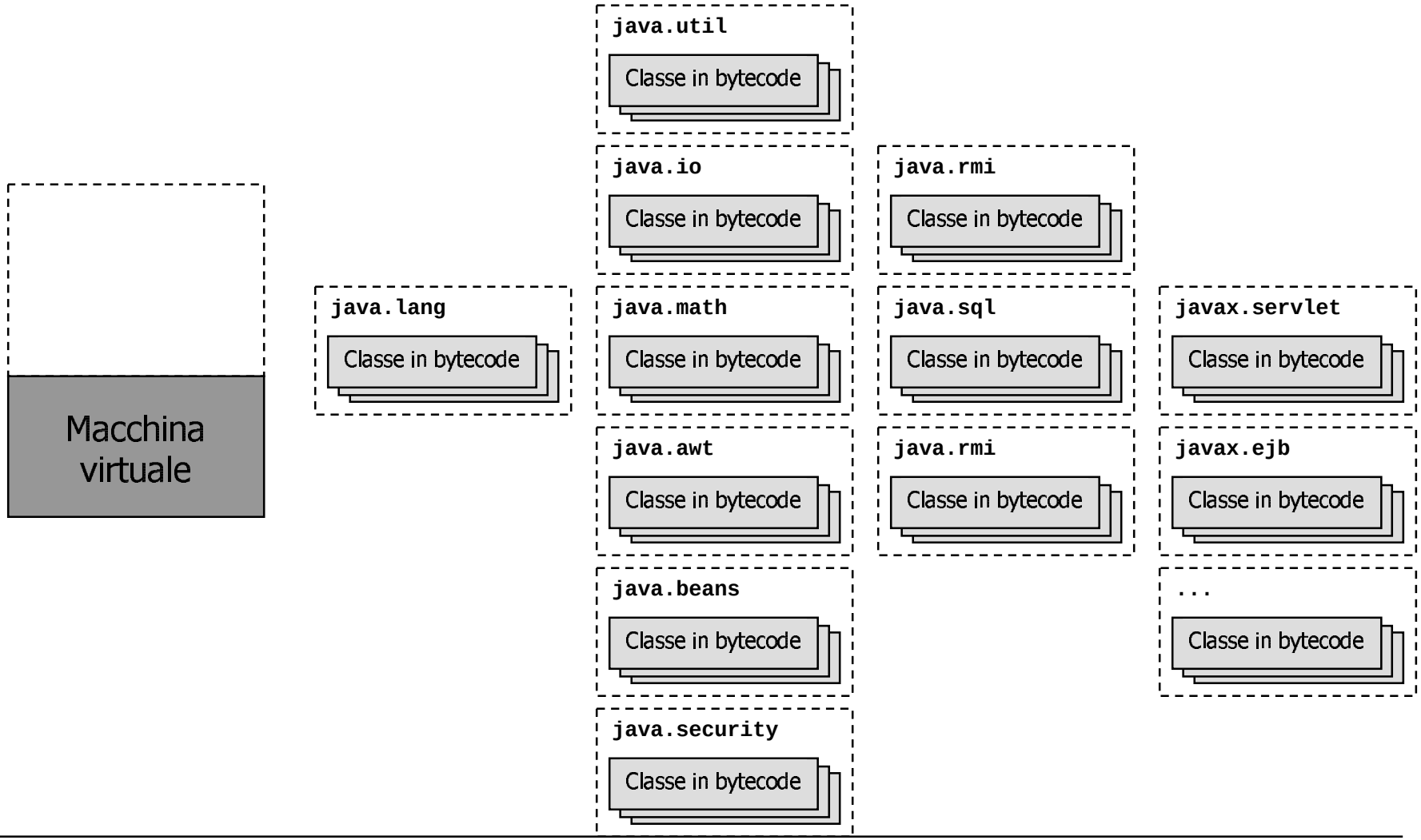
- Java **non** nasce da un progetto accademico o non-profit
- I requisiti di progetto di Java sono quindi:
  - tecnici;
  - ingegneristici;
  - di contesto (cioè di mercato).
- I requisiti tecnici sono evidenti nel progetto del modello astratto
- I requisiti ingegneristici sono evidenti nel modello implementativo 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.