DIETI UNINA UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Corso di Ingegneria del Software

Prof. Roberto Pietrantuono

Il linguaggio Java

::.. Sommario degli Argomenti della Lezione

- Generalità sul linguaggio Java
- Gestione della memoria
- Esempi
- Composizione ed Ereditarietà
- Polimorfismo e Classi Interne

Riferimenti:

- Bruce Eckel, “Thinking in Java” capitolo 6-7-8
- J. Cohoon e J. Davidson, “Java – Guida alla Programmazione” paragrafo 7.4 e capitolo 9
- http://java.html.it/guide/leggi/22/guida-java/
- http://www.vogella.de/articles/Eclipse/article.html

::.. PRIMO ARGOMENTO

Il linguaggio Java

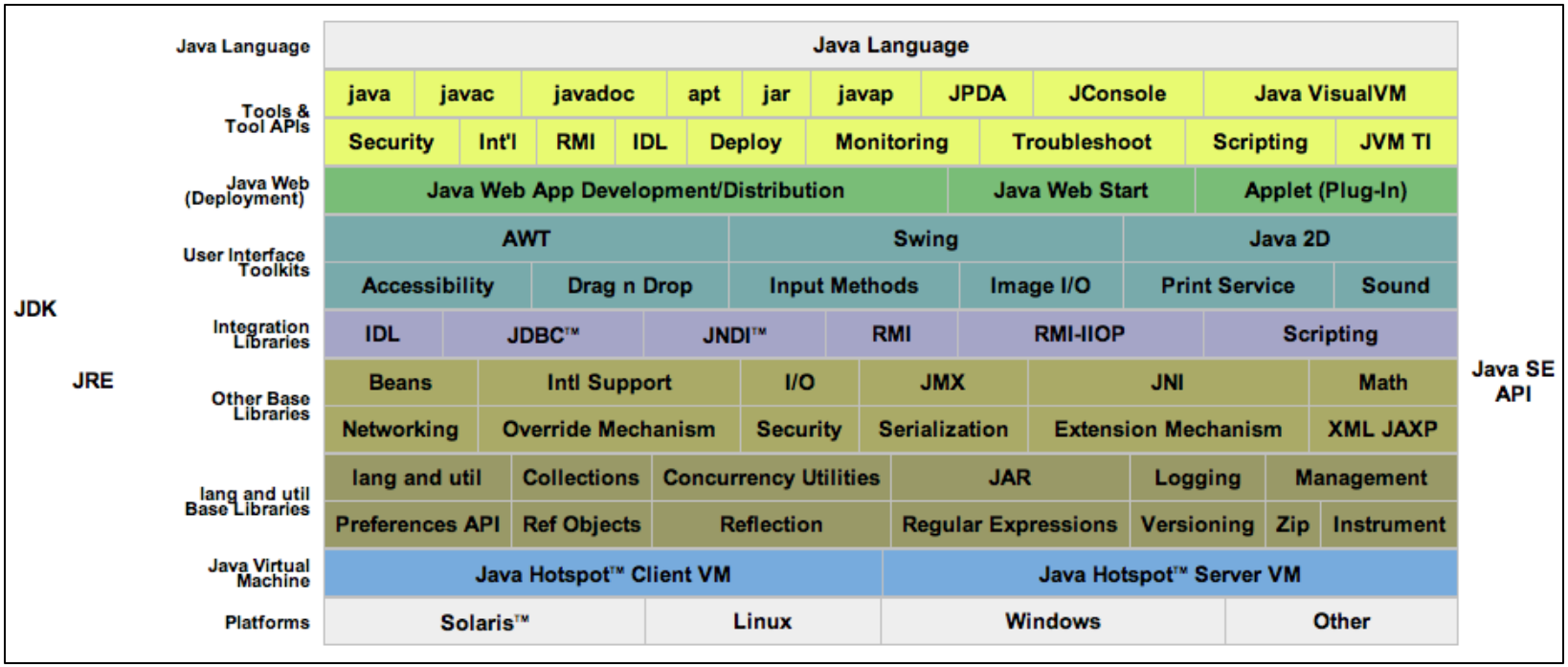
- **Generalità**
- **Gestione della memoria**
- **Esempio di programma**

::.. Il Linguaggio Java (1/3)

Java è stato creato da James Gosling ed altri ingegneri della Sun MicroSystems a partire dal 1991.

L'architettura Java si compone di quattro componenti:

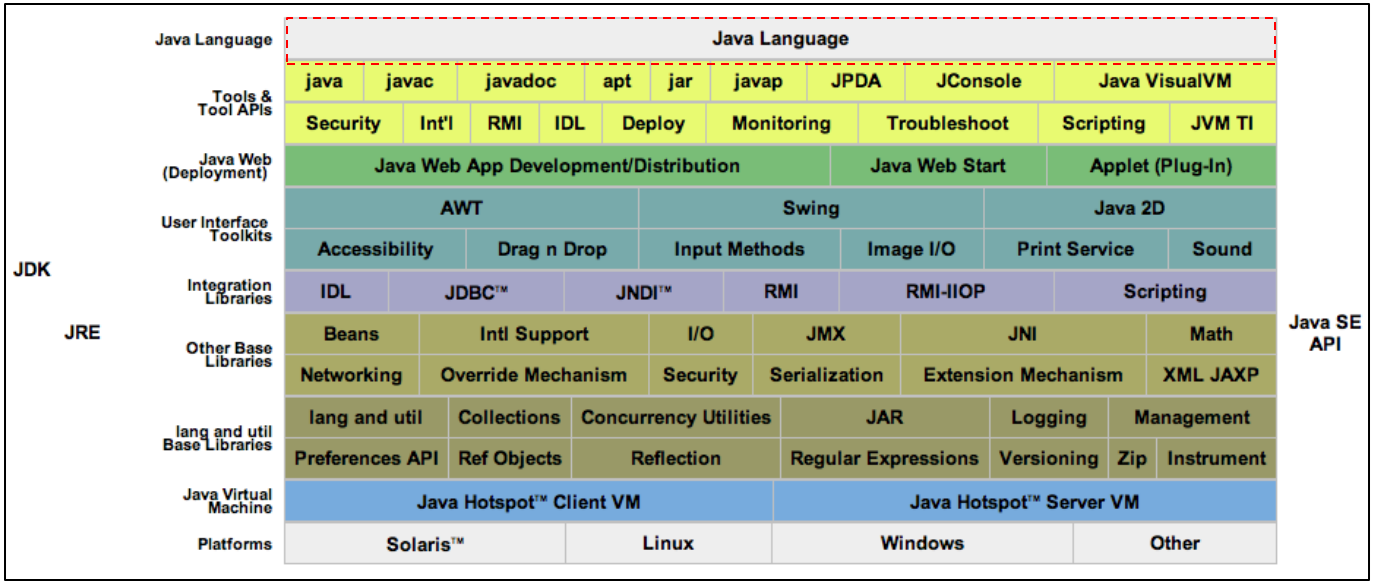
1. Il linguaggio di programmazione;
2. Application Programming Interface (API);
3. Java Virtual Machine (JMV);
4. Java tools a supporto dello sviluppo ed esecuzione di programmi.



::.. Il Linguaggio Java (1/3)

Java è un linguaggio: (i) object-oriented, (ii) multi-threaded, (iii) con gestione delle eccezioni, (iv) con *garbage collector*.

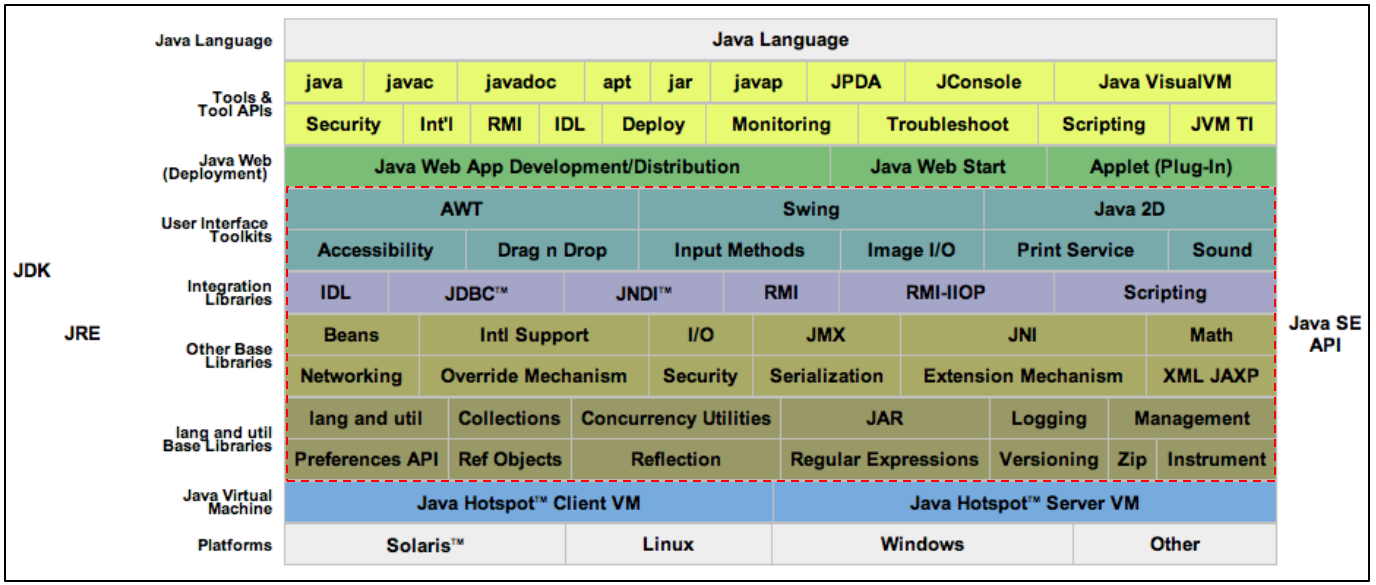
1. Il linguaggio di programmazione;
2. Application Programming Interface (API);
3. Java Virtual Machine (JMV);
4. Java tools a supporto dello sviluppo ed esecuzione di programmi.



::.. Il Linguaggio Java (1/3)

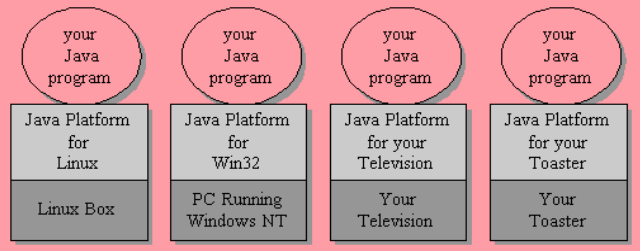
Java API è un insieme di librerie standard per poter accedere alle risorse del computer. Le JAVA API presentano una interfaccia standard, e sono implementate invocando gli opportuni metodi nativi offerti dalla macchina su cui sono eseguite.

1. Il linguaggio di programmazione;
2. Application Programming Interface (API);
3. Java Virtual Machine (JMV);
4. Java tools a supporto dello sviluppo ed esecuzione di programmi.

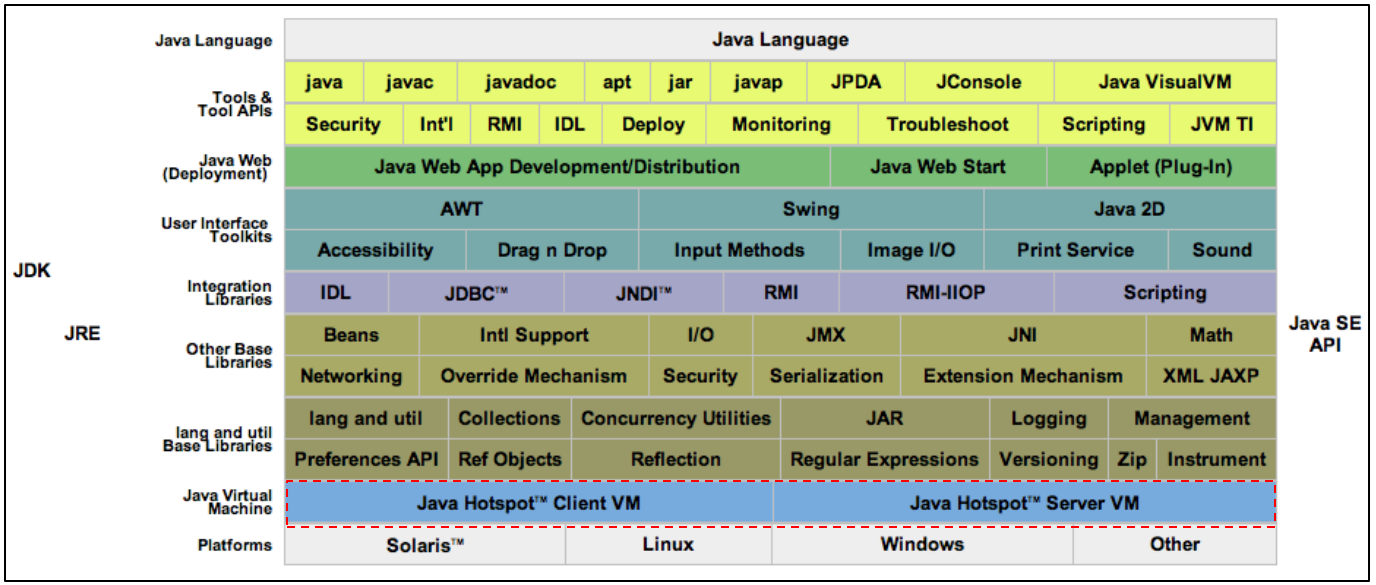


::... Il Linguaggio Java (1/3)

Il compito della JVM è di caricare i class file ed eseguirli. Viene definito un computer astratto, perché fornisce un'astrazione del sistema operativo e dell'hardware.



2. Application Programming Interface (API);
3. Java Virtual Machine (JMV);
4. Java tools a supporto dello sviluppo ed esecuzione di programmi.



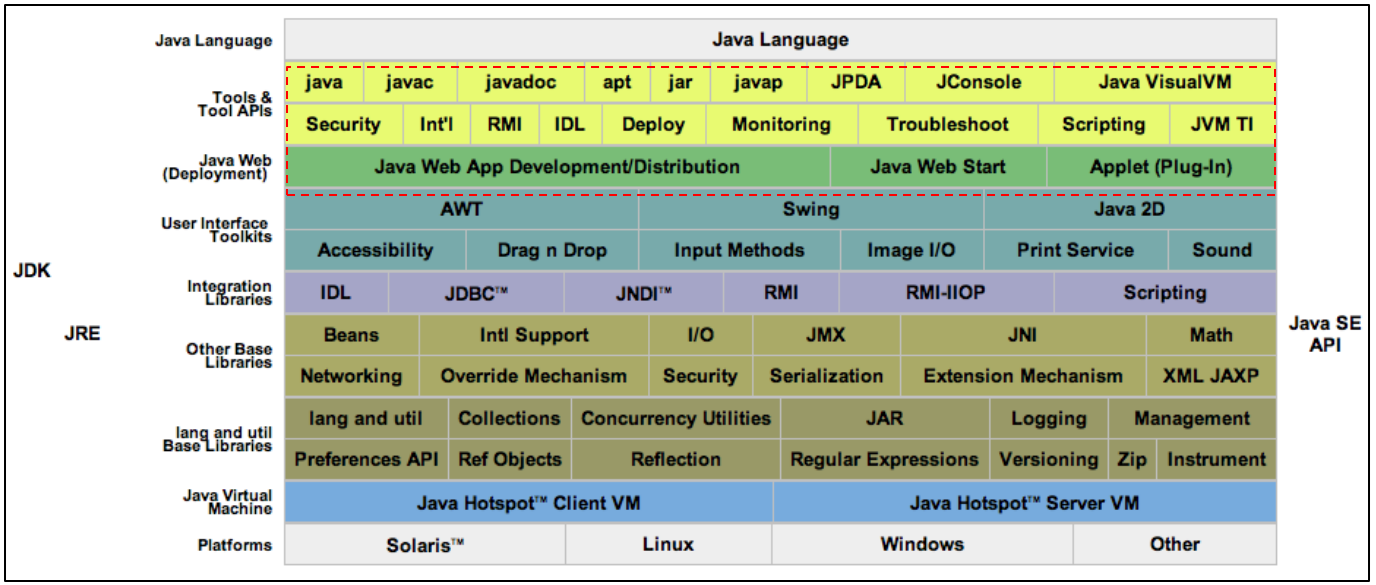
::.. Il Linguaggio Java (1/3)

Java è stato creato da James Gosling ed altri

Java dispone di un insieme di tool a supporto delle fasi di sviluppo di un programma e delle azioni necessarie per la sua esecuzione.

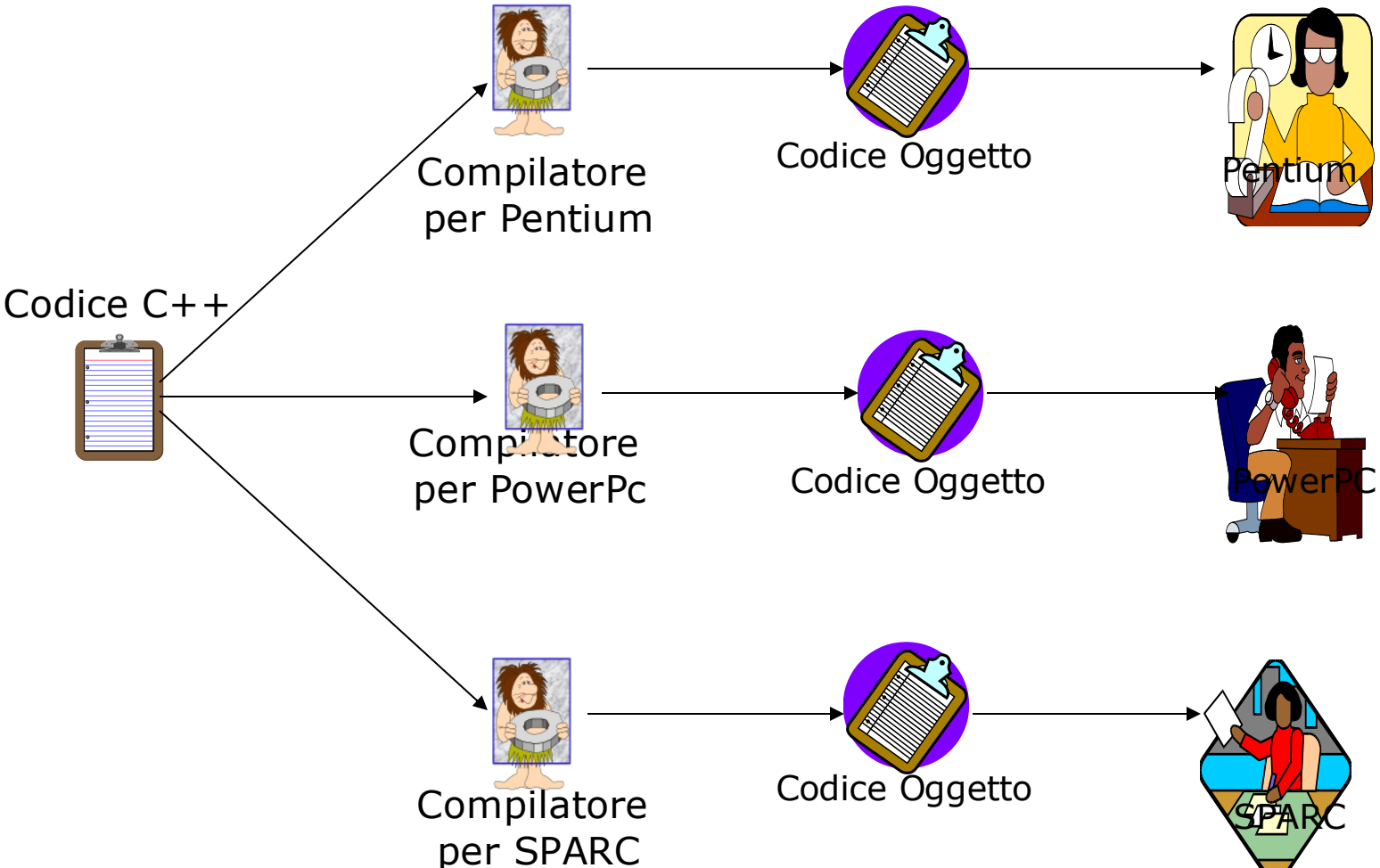
L'insieme dei tool, API e JVM viene spesso definito Java Platform.

3. Java [illegible]chine (JMV);
4. Java tools a supporto dello sviluppo ed esecuzione di programmi.



::.. Il Linguaggio Java (2/3)

L'obiettivo alla base della progettazione di Java è la portabilità: si dovrebbe essere in grado di scrivere il programma una sola volta e di poterlo eseguire dovunque ("write once, run everywhere").

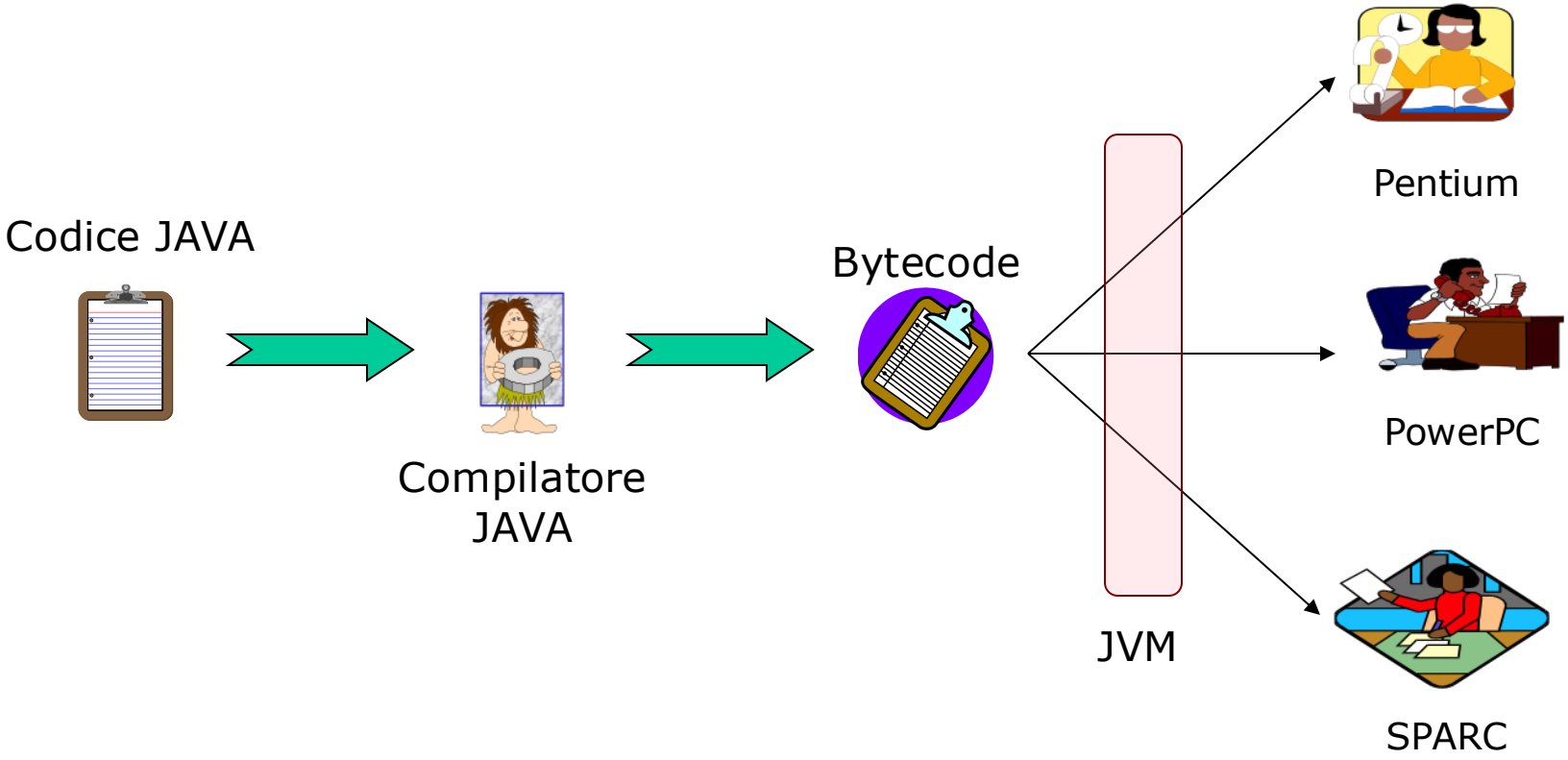


::.. Il Linguaggio Java (2/3)

L'obiettivo alla base della progettazione di Java è la portabilità: si dovrebbe essere in grado di scrivere il programma una sola volta e di poterlo eseguire dovunque ("write once, run everywhere").

::.. Il Linguaggio Java (3/3)

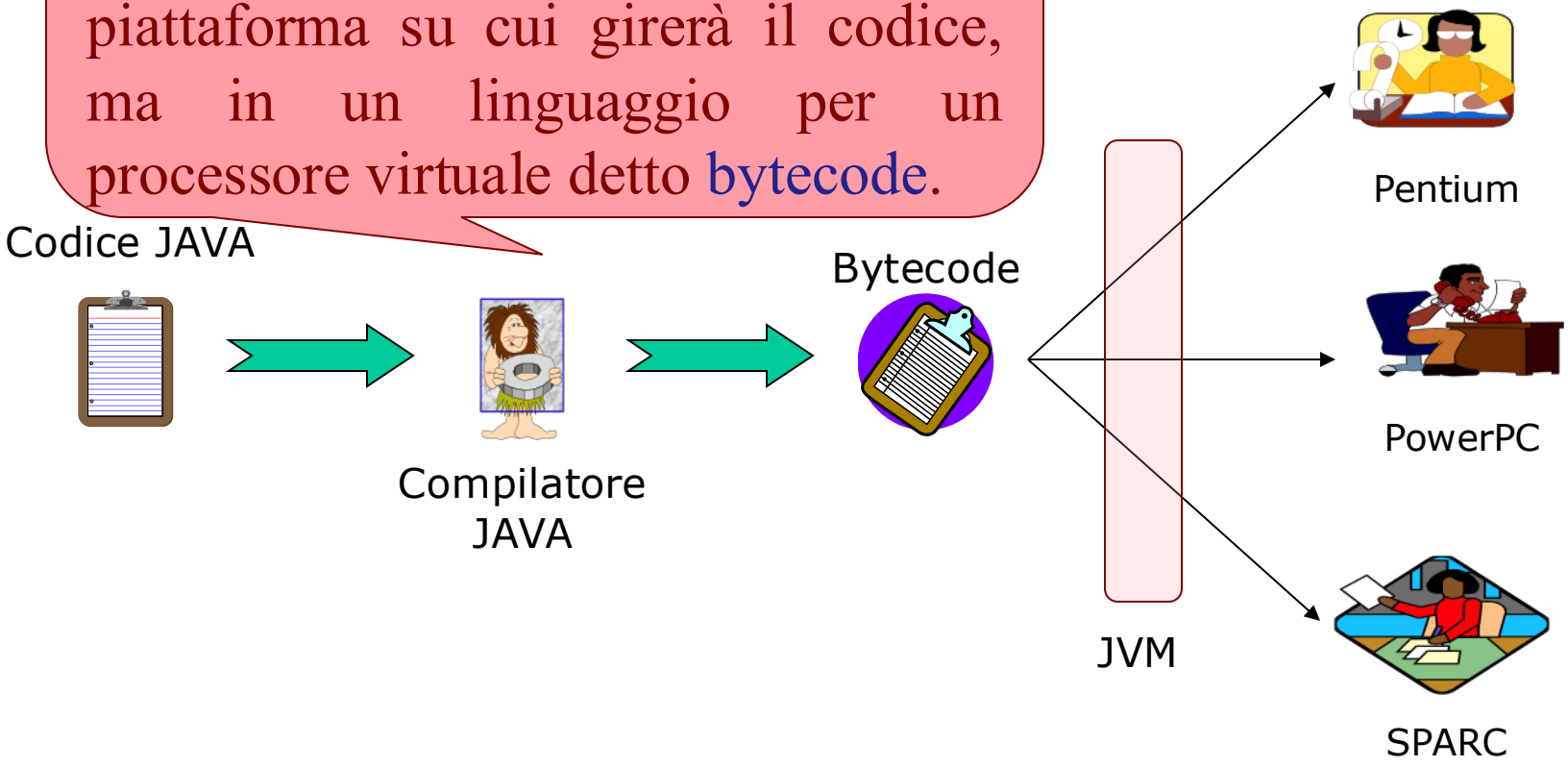
L'obiettivo alla base della progettazione di Java è la portabilità: si dovrebbe essere in grado di scrivere il programma una sola volta e di poterlo eseguire dovunque ("write once, run everywhere").



::.. Il Linguaggio Java (3/3)

L'obiettivo alla base della progettazione di Java è la port grado di scrivere il p poterlo eseguire dovu e").

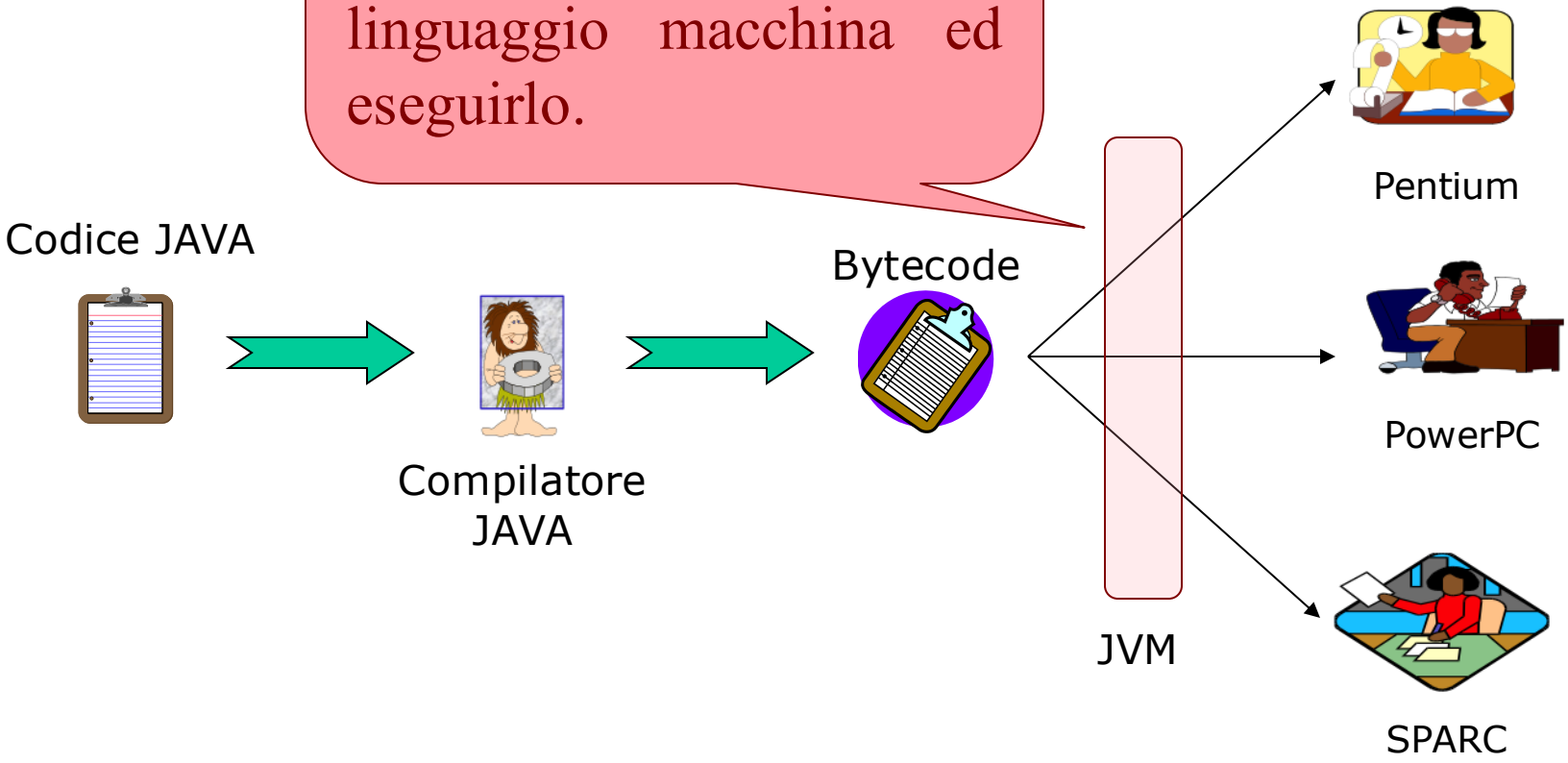
Un apposito compilatore (javac) processa il codice sorgente non in linguaggio macchina specifico della piattaforma su cui girerà il codice, ma in un linguaggio per un processore virtuale detto bytecode.



::.. Il Linguaggio Java (3/3)

L'obiettivo a… …azione di Java è la portabilità: … grado di scrivere il programma… … poterlo eseguire dovunque ("wr… …e").

Ogni JVM dispone di un interprete in grado di tradurre il bytecode in linguaggio macchina ed eseguirlo.

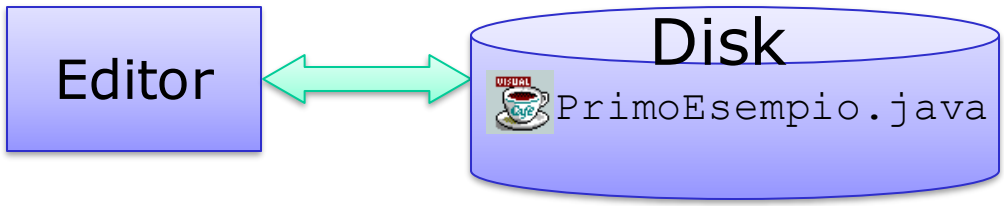


::.. Sviluppo di una applicazione Java (1/5)

La preparazione e l'esecuzione di un programma Java consiste di 5 fasi:

1) La fase di EDIT

```
C:\PRG>edit PrimoEsempio.java
```



Viene creato il file contenente il programma e, successivamente, memorizzato nel disco.

ATTENZIONE:

Il nome del file deve coincidere con quello della classe (case sensitive!)

::.. Sviluppo di una applicazione Java (1/5)

La preparazione e l'esecuzione di un programma Java consistono di 5 fasi:

1) La fase di EDIT

```
C:\PRG>edit PrimoEsempio.java
```

I programmi scritti in Java sono unicamente orientati agli oggetti, di conseguenza tutto il codice deve essere necessariamente incluso in una o più classi.

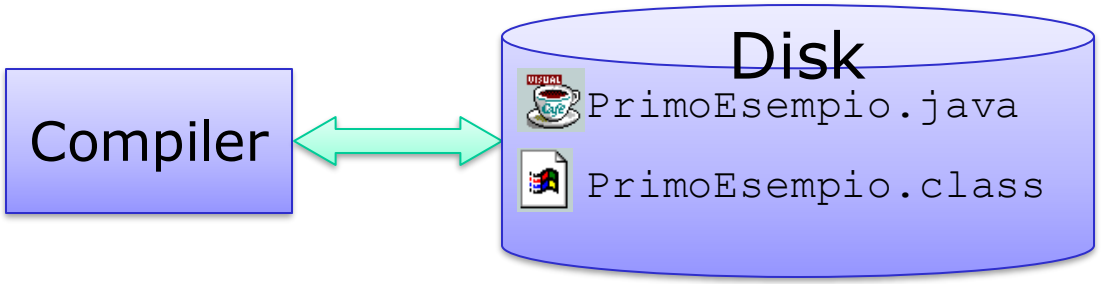
Per rendere eseguibile un'applicazione Java occorre che una classe faccia da "punto di decollo", ovvero deve contenere un metodo pubblico main(). Questo è il primo ad essere invocato durante l'esecuzione dell'applicazione.

della classe (case sensitive!)

::.. Sviluppo di una applicazione Java (2/5)

2) La fase di COMPILAZIONE

```
C:\PRG>javac PrimoEsempio.java
```

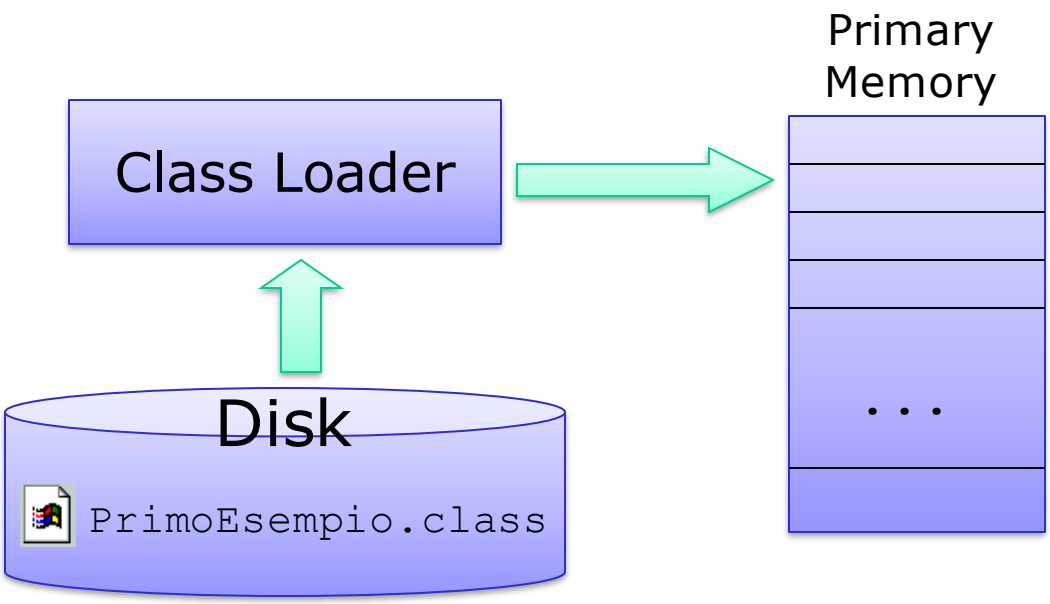


Il compilatore JAVA traduce il programma nel corrispondente bytecode, il linguaggio compreso dall'interprete java, e lo salva su disco

::.. Sviluppo di una applicazione Java (3/5)

3) La fase di CARICAMENTO

```
C:\PRG>java PrimoEsempio
```



Il Class loader carica il bytecode nella memoria principale

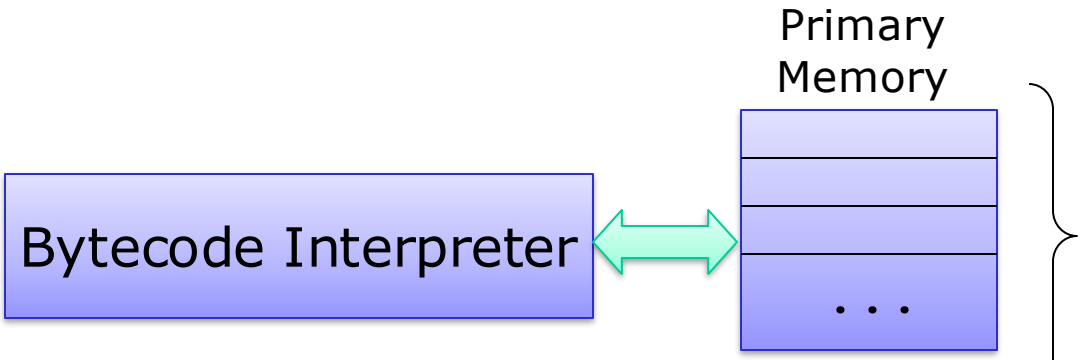
::.. Sviluppo di una applicazione Java (4/5)

4) La fase di VERIFICA del bytecode

Il "bytecode verifier" ha il compito di validare il bytecode e assicurarsi che non si violino i vincoli di sicurezza dell'ambiente JAVA

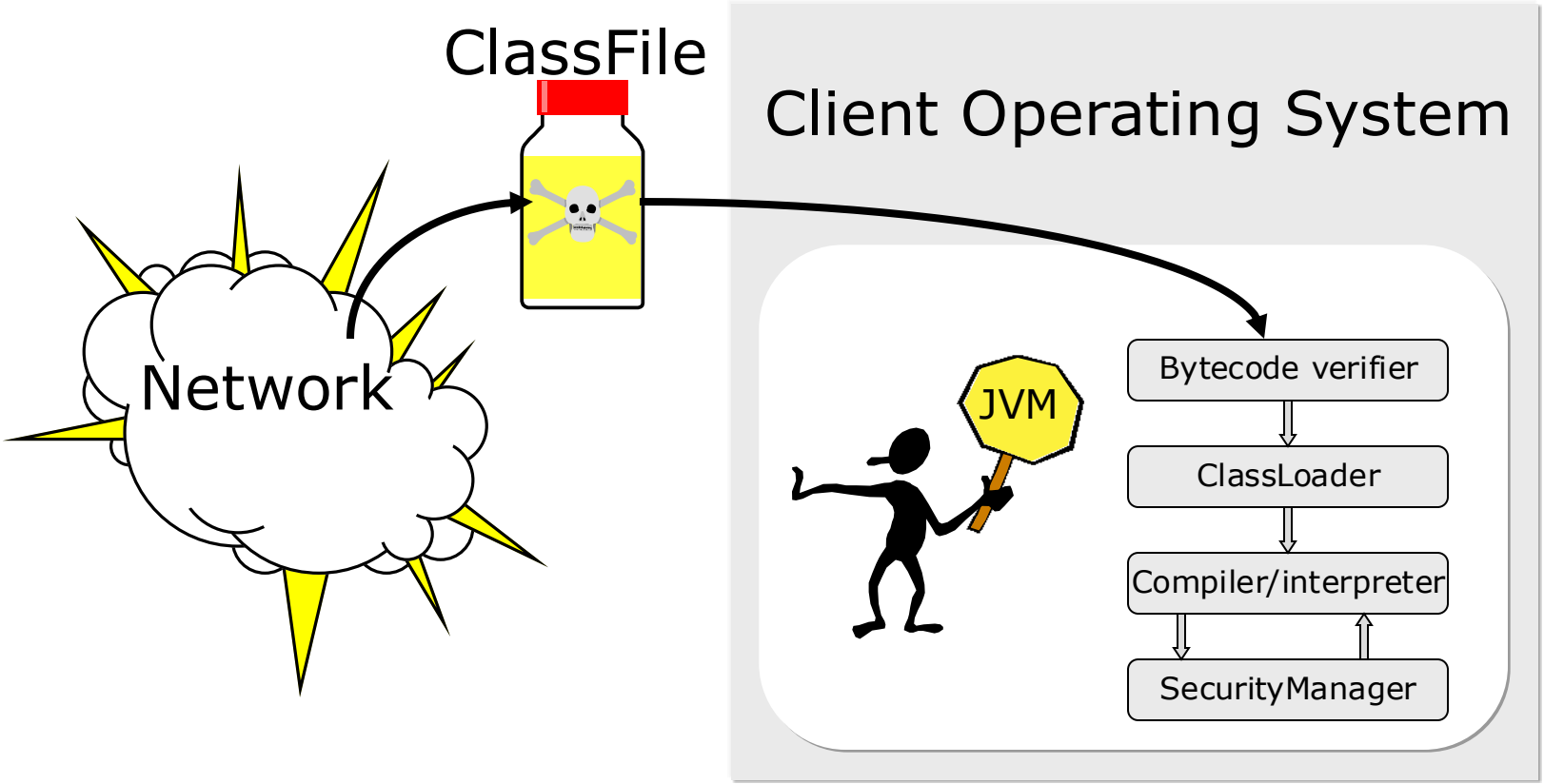
::.. Sviluppo di una applicazione Java (5/5)

5) La fase di ESECUZIONE

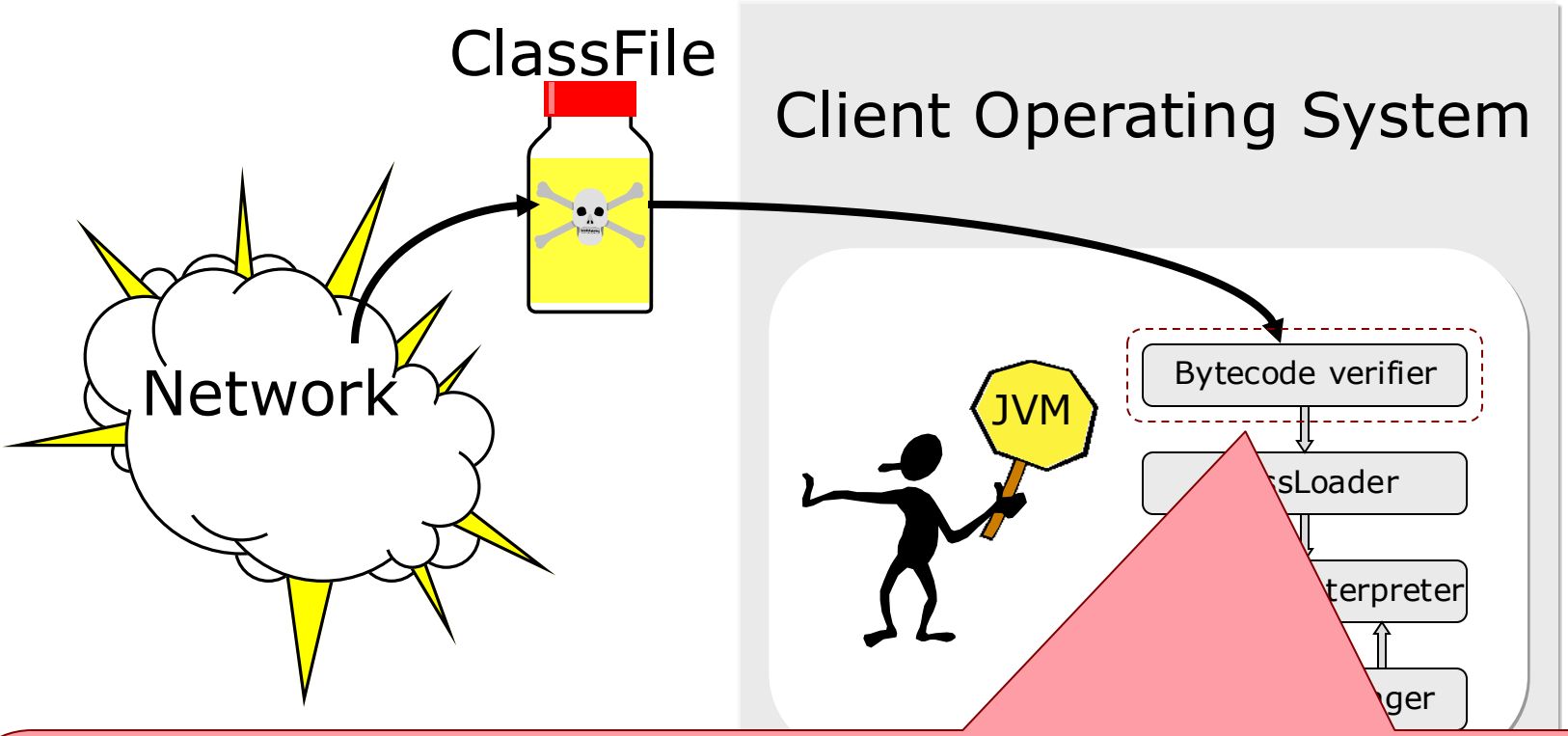


L'interprete del bytecode traduce le istruzioni del bytecode nelle istruzioni compatibili con il processore.

::.. Il modello di sicurezza

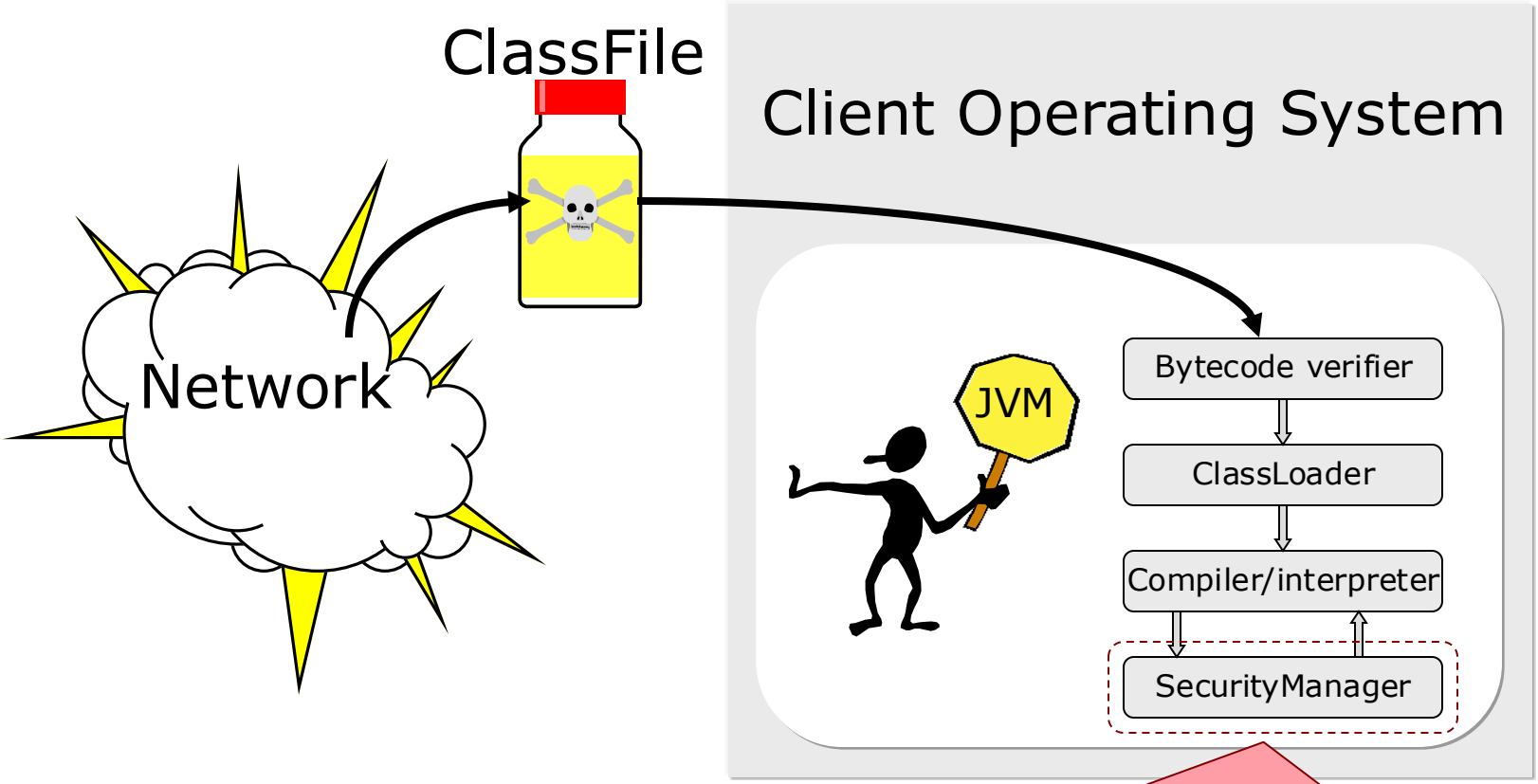


::... Il modello di sicurezza



Prima azione di controllo – Bytecode Verifier: analizza la sequenza di bytecode e controlla il riferimento ad altre classi. Ad esempio: se una classe utilizza un metodo di un'altra, controlla se questo metodo è pubblico oppure se vengono sforati i limiti di un array. Oppure controlla se una classe usata come classe base da un'altra (sotto)classe presenta il vincolo di non specializzazione.

::.. Il modello di sicurezza



Seconda azione di controllo – Security Manager: speciale classe che può essere implementata dal programmatore che disciplina l'accesso a una risorsa (ad esempio accesso a un file o a una connessione di rete).

::.. Tipi primitivi e Classi (1/5)

Java opera una distinzione netta tra classi e tipi primitivi, relativamente al modo in cui avviene l'allocazione in memoria:

- Quando si dichiara una variabile del tipo intero (int x) vengono subito allocati quattro byte.
- Se si utilizza una classe in qualità di tipo per una certa variabile (MyClass c), verrà creata subito una variabile che referenzia l'oggetto (reference) ma non verrà allocata memoria fino alla creazione vera e propria dell'istanza della classe, tramite l'utilizzo dell'operatore new.

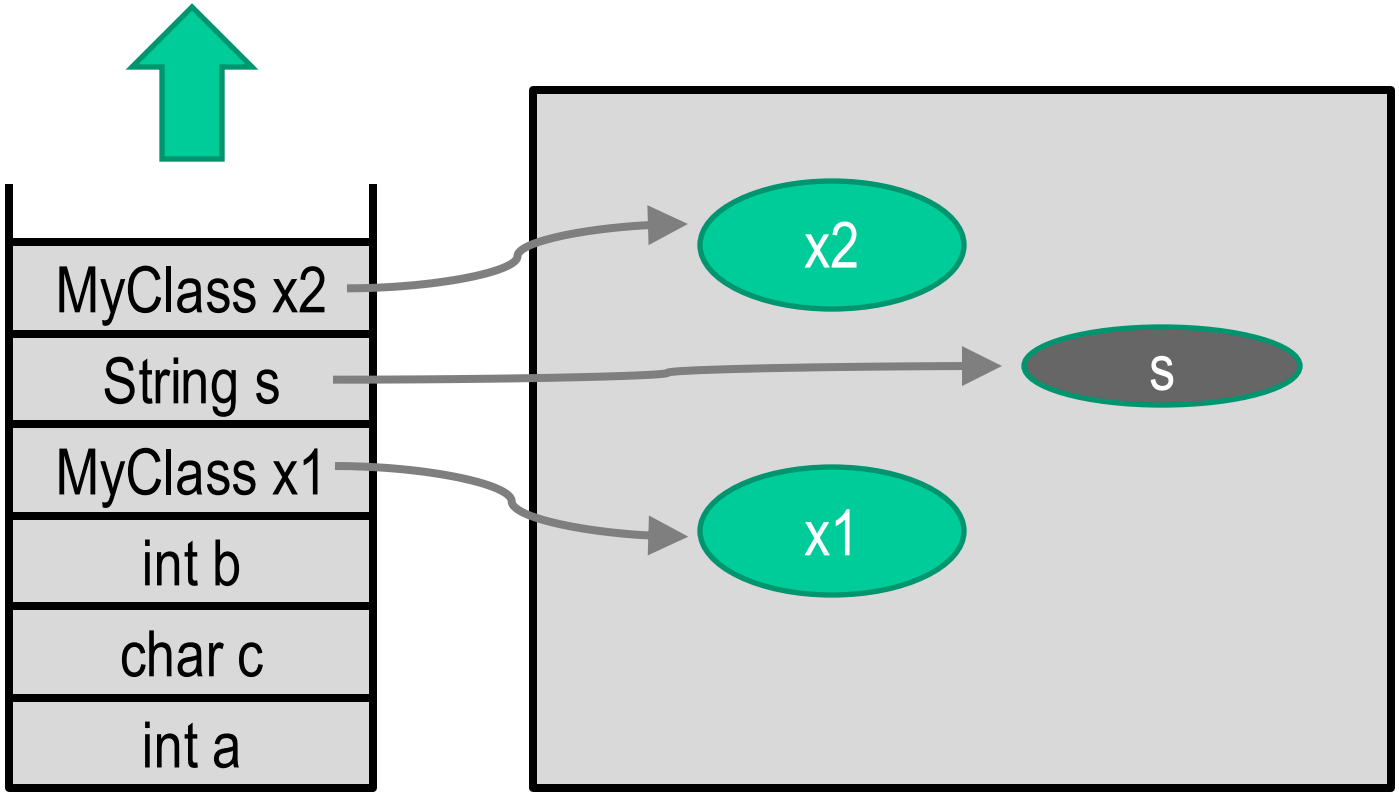
Un reference è, dunque, una variabile speciale che tiene traccia di ("punta a") istanze di tipi non primitivi. I reference possono tenere traccia soltanto di oggetti di tipo compatibile, ovvero un reference ad un oggetto di tipo MyClass non potrà tenere traccia di oggetti di diverso tipo.

::.. Tipi primitivi e Classi (2/5)

Nel linguaggio di programmazione C++, sono di comune utilizzo variabili di tipo riferimento: sono nomi alternativi dati alla variabile, ed hanno l'effetto di puntatori costanti alle variabili cui fanno riferimento; sono usati per estenderne la visibilità oltre l'ambito in cui sono state definite.
Dichiarando una variabile di tipo riferimento, non è allocato nessun oggetto, ma viene dato un nuovo nome a un oggetto già esistente. Tale variabile, quando viene usata in un'espressione, ha il valore dell'oggetto cui fa riferimento.

Java si discosta da quanto avviene in C++: i riferimenti puntano solo a oggetti, e non a variabili di tipo primitivo, e non costituiscono dei nomi alternativi. Infatti, sono degli oggetti allocati in area stack che puntano ad aree di memoria allocate nello heap.
Inoltre, il contenuto di un riferimento non è costante, ma può essere variato a piacimento dal programmatore.

::.. Tipi primitivi e Classi (3/5)



STACK
(tipi primitivi, riferimenti)

HEAP
(oggetti)

::.. Tipi primitivi e Classi (4/5)

Supponiamo di istanziare due oggetti di tipo String contenenti entrambi il valore "Pippo":

```
String s = new String("Pippo");
String s1 = new String("Pippo”).

if(s == s1) {   ...  } else {    ... }
```

Quale dei due rami del costrutto if…else viene certamente eseguito?

::.. Tipi primitivi e Classi (4/5)

Supponiamo di istanziare due oggetti di tipo String contenenti entrambi il valore "Pippo":

```
String s = new String("Pippo");
String s1 = new String("Pippo”).

if(s == s1) {   ... } else {    ... }
```

Le istruzioni che verranno eseguite saranno quelle all'interno dell'else.

::.. Tipi primitivi e Classi (4/5)

Supponiamo di istanziare due oggetti di tipo String contenenti entrambi il valore "Pippo":

```
String s = new String("Pippo");
String s1 = new String("Pippo”).

if(s == s1) {   ... } else {    ... }
```

Le istruzioni che verranno eseguite saranno quelle all'interno dell'else.

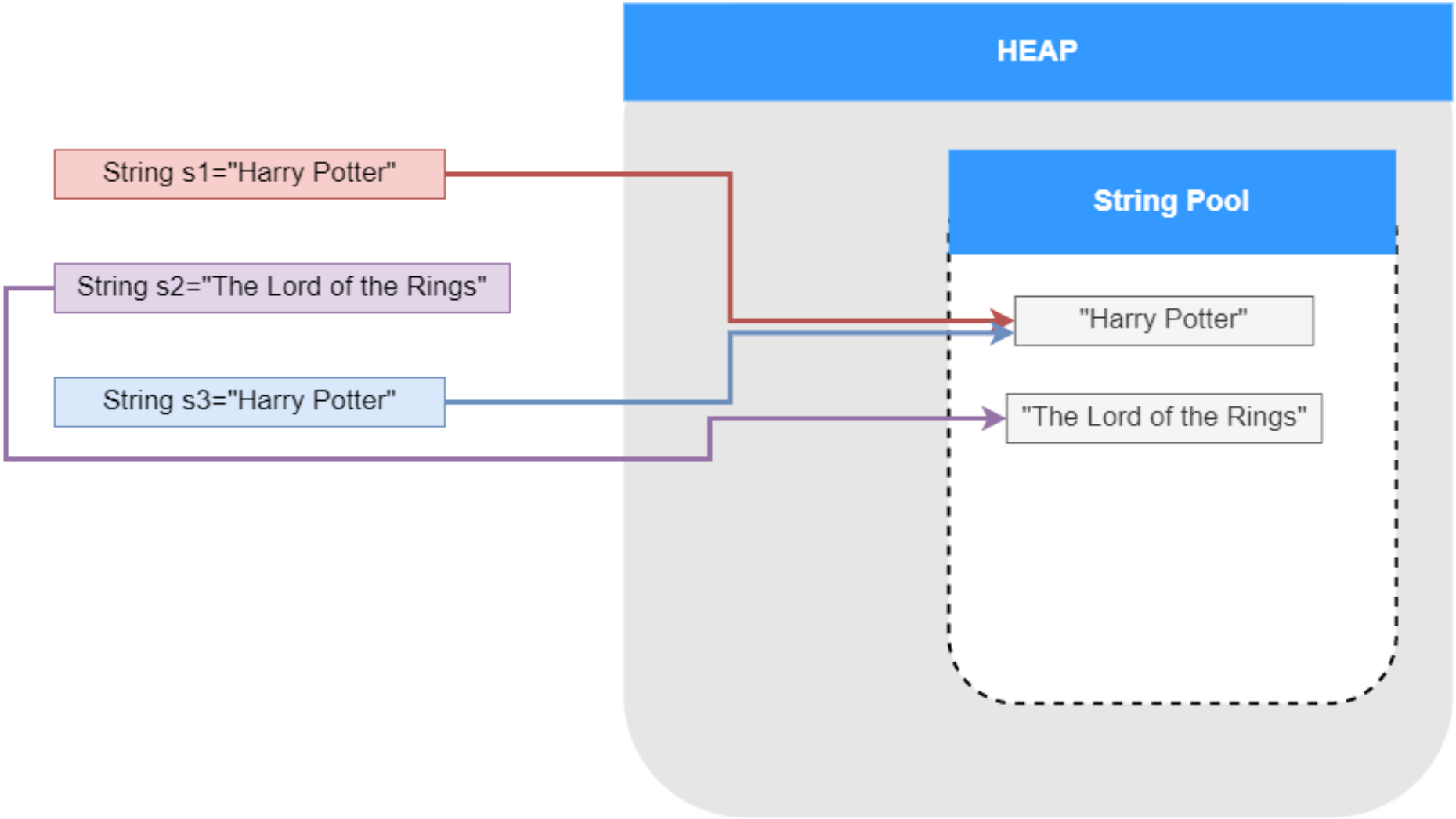
Come mai?

::.. Tipi primitivi e Classi (5/5)

La ragione è legata al fatto che l'operatore "==" esegue la comparazione tra le variabili reference che tengono traccia dei due oggetti e non tra i valori contenuti dagli oggetti stessi.
Mentre tra variabili di tipi primitivi la comparazione è normalmente effettuata tramite l'operatore "==", per la comparazione tra due istanze di oggetti della stessa classe, sarà necessario utilizzare il metodo equals().

```
String s = new String("Pippo");
String s1 = new String("Pippo");
if(s.equals(s1)){   ... } else {    ... }
```

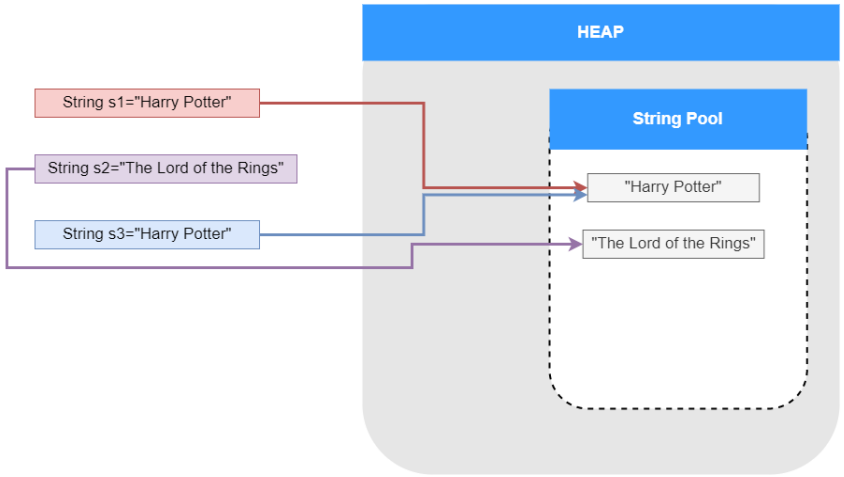
String Pool (1/3)



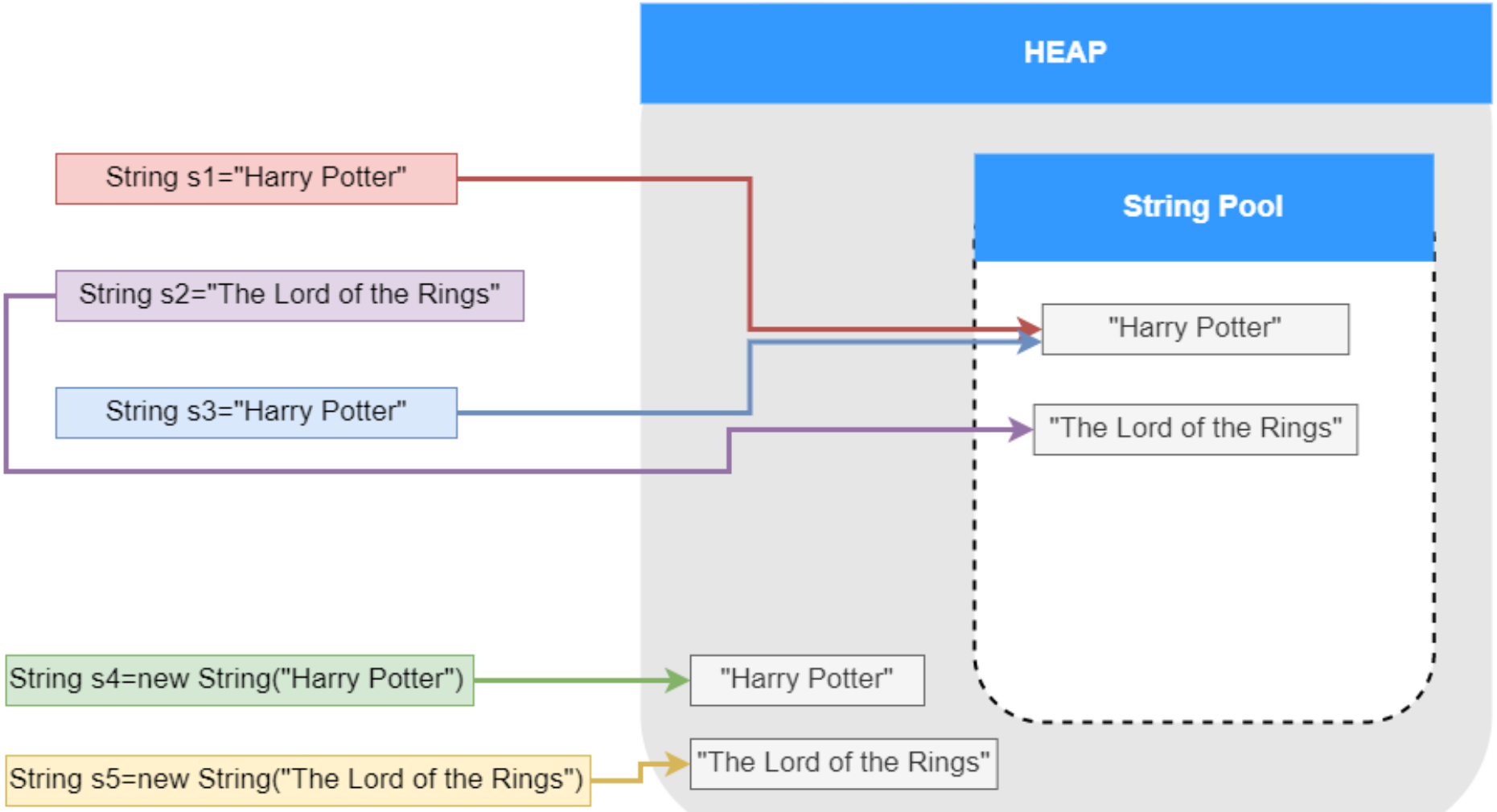
String Pool (2/3)

Java String Pool è un'area di archiviazione speciale nell'heap in cui vengono archiviate le stringhe. È implementato per migliorare le prestazioni delle operazioni sulle stringhe e per risparmiare memoria. È noto anche come String Intern Pool o String Constant Pool.

Quando viene create una nuova stringa dalla classe String, la JVM controlla prima se la stringa è già presente nello String Pool. In tal caso, la JVM restituirà un riferimento all'oggetto stringa esistente, anziché creare uno nuovo. Questo processo è noto come **string interning**.



String Pool (3/3)



::.. Gestione della memoria (1/2)

Per instanziare un oggetto dinamicamente nell'area heap, si fa ricorso all'istruzione new:

```
String str = new String("stringa");
```

Come fare se vogliamo deallocare un oggetto? Esiste in Java una istruzione di delete come in C++?

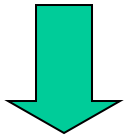
::.. Gestione della memoria (1/2)

Per istanziare un oggetto dinamicamente nell'area heap, si fa ricorso all'istruzione new:

```
String str = new String("stringa");
```

Come fare se vogliamo deallocare un oggetto? Esiste in Java una istruzione di delete come in C++?

In JAVA è stato implementato un'apposito "modulo" che, in maniera automatica, recupera la memoria degli oggetti non più utilizzati (cioè che non sono più riferiti)



garbage collector

Quindi in JAVA non esiste più il problema, tipico del C e C++, della "perdita di memoria" dovuto a...
...programmatori un po' distratti !

::.. Gestione della memoria (2/2)

Quando un oggetto non è più referenziato dal programma, lo spazio che occupa nello heap deve essere liberato, così da renderlo disponibile per nuovi oggetti.

Il *garbage collector* deve rendersi conto di quali oggetti non sono più referenziati e liberarne lo spazio sotteso. Inoltre, deve combattere la frammentazione dovuta alle continue allocazioni e deallocazioni durante il ciclo di vita di un programma

Vantaggio: Maggiore produttività del programmatore – Garanzie di integrità: un programmatore non può accidentalmente (o maliziosamente) causare un crash della JVM liberando incorrettamente della memoria.

Svantaggio: il garbage collector implica un considerevole overhead che può inficiare le performance del programma sviluppato.

::.. Gestione della memoria (2/2)

Quando un oggetto non è più referenziato dal programma, lo spazio che occupa nello heap deve essere liberato, così da renderlo disponibile per nuovi oggetti.

Il *garbage collector* deve rendersi conto di quali oggetti non sono più referenziati e liberarne lo spazio sotteso. Inoltre, deve combattere la frammentazione dovuta alle continue allocazioni e deallocazioni durante il ciclo di vita di un programma

Vantaggio: Maggiore produttività del programmatore – Garanzie di integrità: un programmatore non può accidentalmente (o maliziosamente) causare un crash della JVM liberando incorrettamente della memoria.

Svantaggio: il garbage collector implica un considerevole overhead che può inficiare le performance del programma sviluppato.

::.. Lo scambio di parametri

A differenza degli altri linguaggi, in JAVA non è possibile scegliere la modalità di scambio dei parametri con una funzione o metodo.

Esiste, invece, una regola fissa:

- **Scambio per valore: per tutti i tipi di dati primitivi (int, boolean, char…);**
- **Scambio per riferimento: per tutti gli oggetti (array, stringhe, oggetti di utente…).**

...in effetti, molti puristi amano dire che in Java viene adottato solo lo scambio per valore

::.. Esempio di programma Java

```
class Persona {
        private String nome;
        Persona (String n) {
                nome=new String(n);
        }
        public String identita() {
                return new String(nome);
        }
}
```

Persona.java

```
class Prova_Persona {

   public static void main(String argv[]) {
        Persona a;
        Persona l;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Prova_Persona.java

::.. Esempio di programma Java

```
class Persona {
        private String nome;
        Persona (String n) {
                nome=new String(n);
        }
        public String identita() {
                return new String(nome);
        }
}
```

```
class Prova_Persona {

   public static void main(String argv
        Persona arossi;
        Persona lbianchi;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Costruttore della classe, notare che non è presente alcun distruttore.

Se è necessario eseguire delle operazioni prima che l'oggetto sia deallocato dal Garbage Collector si deve implementare il metodo finalize().

::.. Esempio di programma Java

```
class Persona {
        private String nome;
        Persona (String n) {
                nome=new String(n);
        }
        public String identita() {
                return new String(nome);
        }
}
```

Persona.java

Metodo per ottenere il valore della variabile 'nome'.
Notare che si crea una copia della variabile.

```
class Prova_Persona {

   public static void main(String argv[
        Persona asergio;
        Persona lrom;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Prova_Persona.java

::.. Esempio di programma Java

```
class Persona {
        private String nome;
        Persona (String n) {
                nome=new String(n);
        }
        public String identita() {
                return new String(nome);
        }
}
```

Persona.java

Metodo principale che la JVM esegue per lanciare l'applicazione.

```
class Prova_Persona {

   public static void main(String argv[]) {
        Persona a;
        Persona l;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Prova_Persona.java

::.. Esempio di programma Java

```
class Persona {
        private String nome;
        Persona (String n) {
                nome=new String(n);
        }
        public String identita() {
                return new String(nome);
        }
}
```

```
class Prova_Persona {

   public static void main(String argv[]) {
        Persona asergio;
        Persona lrom;
        asergio=new Persona("Antonio Sergio");
        lrom= new Persona ("Luigi Romano");
        System.out.println(asergio.identita());
        System.out.println(lrom.identita());
   }
}
```

Un metodo static non richiede l'instanziazione di un oggetto della classe per il suo utilizzo.
Che significa avere una variabile static?

Prova_Persona.java

::.. Esempio di programma Java

```
class Persona {
        private String nome;
        Persona (String n) {
                nome=new String(n);
        }
        public String identita() {
                return new String(nome);
        }
}
```

```
class Prova_Persona {

   public static void main(String argv[]) {
        Persona a;
        Persona l;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Una variabile static è una variabile allocata una sola volta indipendentemente dagli oggetti della classe e accessibile da tutti gli oggetti.

Prova_Persona.java

::.. Esempio di programma Java

```
class Persona {
        private String nome;
        Persona (String n) {
                nome=new String(n);
        }
        public String identita() {
                return new String(nome);
        }
}
```

Persona.java

```
class Prova_Persona {

   public static void main(String argv[]) {
        final int = 5;
        Persona a, l;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Prova_Persona.java

Una variabile final è una variabile il cui contenuto è costante.

::.. Esempio di programma Java

```
class Persona {
        static final int i = 2;
        private String nome;
        Persona (String n) {
                nome=new String(n); }
        public String identita() {
                return new String(nome);
        }
}
```

Persona.java

Una variabile può essere allo stesso tempo static e final… che significa?

```
class Prova_Persona {

   public static void main(String argv[]) {
        final int j = 5;
        Persona asergio, lrom;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Prova_Persona.java

::... Esempio di programma Java

```
class Persona {
        static final int i = 2;
        private String nome;
        Persona (String n) {
                nome=new String(n); }
        public String identita() {
                return new String(nome);
        }
}
```

Persona.java

```
class Prova_Persona {

   public static void main(String argv
        final int j = 5;
        Persona asergio, lrom;
        a = new Persona("Antonio");
        l = new Persona ("Luigi");
        System.out.println(a.identita());
        System.out.println(l.identita());
   }
}
```

Una variabile può essere allo stesso tempo static e final… che significa?
È una variabile allocata una sola volta per tutti gli oggetti, e il cui contenuto è costante.

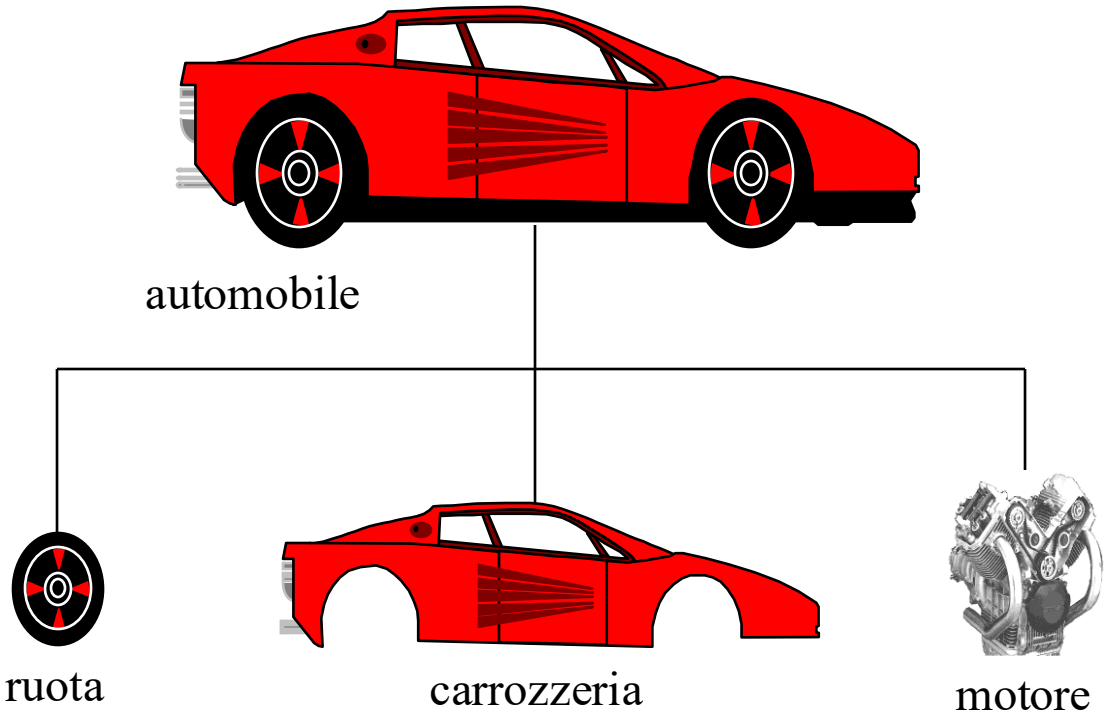
::.. SECONDO ARGOMENTO

Programmazione OO in Java

- **Composizione, Ereditarietà**
- **Visibilità in una gerarchia gen/spec**
- **Invocazione metodi di una classe base**
- **Polimorfismo e Interfacce**
- **Classi Interne**

::.. Composizione (1/4)

La composizione si realizza inserendo nella nuova classe dei riferimenti agli oggetti delle classi esistenti.



::.. Composizione (2/4)

```
class Ruota{
   private String s;
   Ruota() {
      System.out.println("Ruota()");
      s = new String("Ruota Inserita");
   }
   public String toString() { return s;
   }
}
class Motore{
   private String s;
   Motore() {
      System.out.println("Motore()");
      s = new String("Motore Montato");
   }
   public String toString() { return s;
   }
}
class Carrozzeria{
   private String s;
   Carrozzeria() {
      System.out.println("Carrozzeria()")
      ;
      s = new String("Carrozzeria
      Inserita");
   }
   public String toString() { return s;
   }
}
```

```
public class Macchina{
   private Ruota ruota1, ruota2, ruota3,
   ruota4;
   Motore motore;
   Carrozzeria carrozzeria;
   String s;
   Macchina() {
         //Istanziazione metodi
      System.out.println("Macchina()");
   }
   void print() {
      System.out.println("macchina = " + s);
      System.out.println("ruota1 = " + ruota1);
      System.out.println("ruota2 = " + ruota2);
      System.out.println("ruota3 = " + ruota3);
      System.out.println("ruota4 = " + ruota4);
      System.out.println("motore= " + motore);
      System.out.println("carrozzeria = " +
      carrozzeria);
   }
   public static void main(String[] args) {
      Macchina x = new Macchina();
      x.print();
   }
}
```

::.. Composizione (3/4)

```
class Ruota{
   private String s;
   Ruota() {
      System.out.println("Ruota()");
      s = new String("Ruota Inserita");
   }
   public String toString() { return s;
   }
}
class Motore{
   private String s;
   Motore() {
      System.out.println("Motore()");
```

```
public class Macchina{
   private Ruota ruota1, ruota2, ruota3,
   ruota4;
   Motore motore;
   Carrozzeria carrozzeria;
   String s;
   Macchina() {
      //Istanziazione metodi
   System.out.println("Macchina()");
   }
```

```
ln("macchina = " + s);
n("ruota1 = " + ruota1);
n("ruota2 = " + ruota2);
n("ruota3 = " + ruota3);
n("ruota4 = " + ruota4);
n("motore= " + motore);
n("carrozzeria = " +
```

```
main(String[] args) {
Macchina();
```

```
   x.print();
   }
}
```

```
   }
   public String toString() { return s;
   }
}
```

Il compilatore non instanzia automaticamente gli oggetti per ogni riferimento definito nella classe Macchina. In Java attributi di tipi primitivi sono automaticamente inizia-lizzati a 0, mentre i riferimenti di oggetti sono inizializzati a null.

::.. Composizione (4/4)

```
public class Macchina{
   private Ruota ruota1, ruota2, ruota3, ruota4;
   Motore motore;
   Carrozzeria carrozzeria = new Carrozzeria ();
   String s = "Macchina Assemblata";
   Macchina() {
      System.out.println("Macchina()");
      motore = new Motore();
   }
   void print() {
         ruota1 = new Ruota();
         ruota2 = new Ruota()
         ruota3 = new Ruota();
         ruota4 = new Ruota();
         System.out.println("macchina = " + s);
          System.out.println("ruota1 = " + ruota1);
          System.out.println("ruota2 = " + ruota2);
          System.out.println("ruota3 = " + ruota3);
          System.out.println("ruota4 = " + ruota4);
          System.out.println("motore= " + motore);
          System.out.println("carrozzeria = " + carrozzeria);
   }
   ....
}
```

1

I riferimenti sono inizializzati prima che il costruttore sia invocato

2

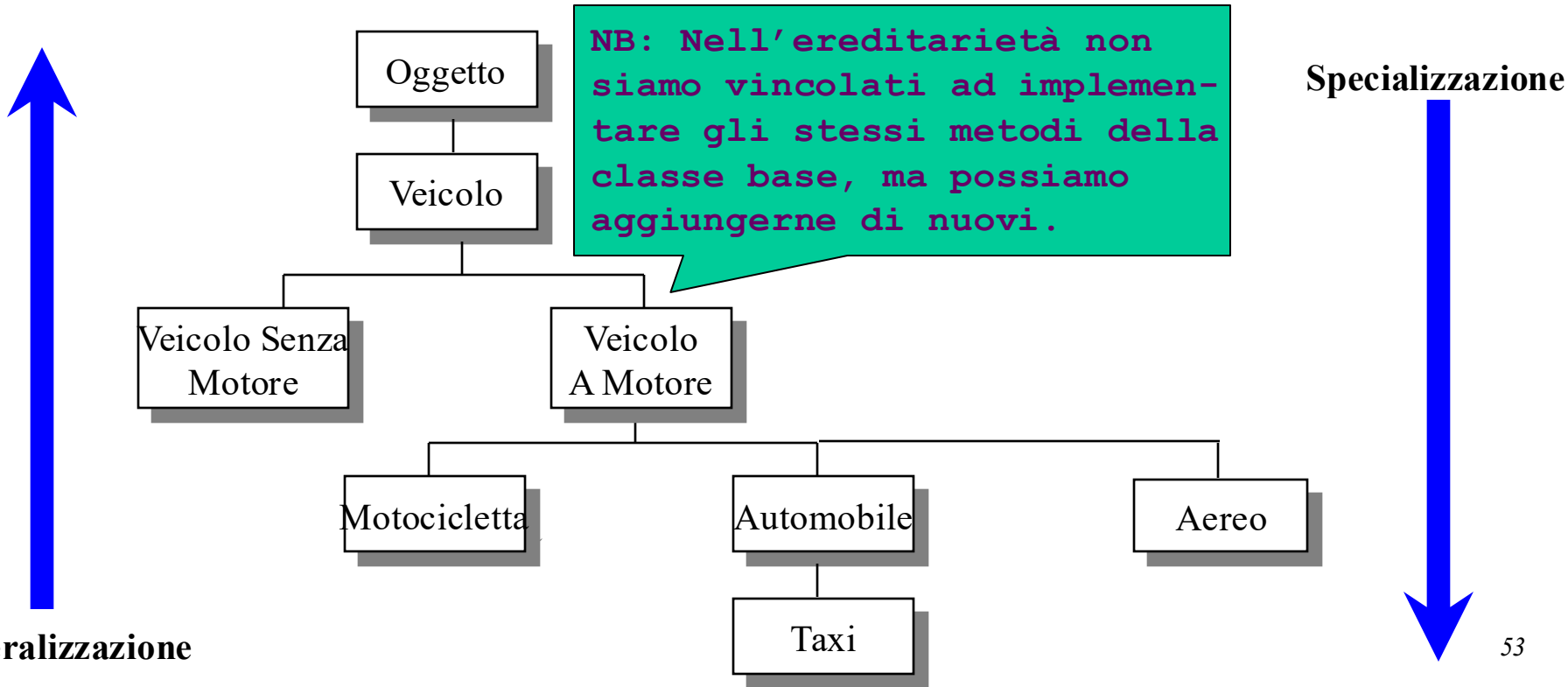
I riferimenti sono inizializzati nel costruttore

3

I riferimenti sono inizializzati prima di essere usati

::.. Ereditarietà (1/6)

L'ereditarietà consente di definire nuove classi ereditando le caratteristiche offerte da classi esistenti. In altre parole si realizzano relazioni tra classi di tipo gen-spec: una classe base, realizza un comportamento comune ad un insieme di entità, mentre le classi derivate (sottoclassi) realizzano comportamenti specializzati rispetto a quelli della classe base.



::.. Ereditarietà (2/6)

Sintassi:

```
class Veicolo : public Oggetto {…};
```

```
class Veicolo extends Oggetto {…};
```

Il legame gen-spec tra due classi è rappresentato per mezzo della parola chiave extends, secondo la seguente sintassi:

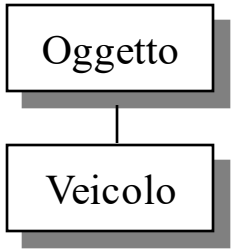
{nome_classe_derivata} extends {nome_classe_base}

In Java, a differenza del C++, non è possibile esprimere la modalità di derivazione (public, protected, private). Inoltre, esiste un qualificatore final con cui si dichiara una classe non ulteriormente derivabile.

Es. final class Ferrari [extends Macchina] {..}

::.. Ereditarietà (3/6)

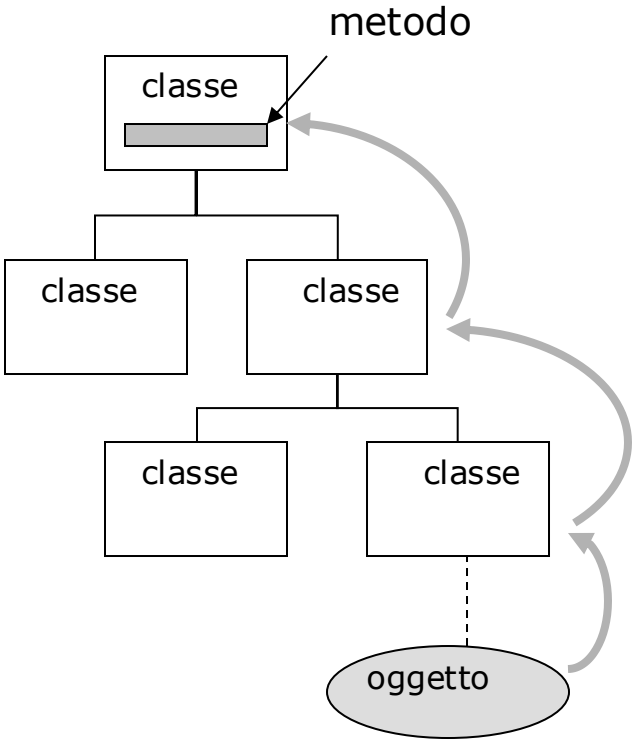
Un'altra differenza con il C++ è che Java implementa soltanto l'**ereditarietà singola**, pertanto ciascuna classe in Java può avere una sola superclasse.



Inoltre, ogni classe in Java è automaticamente e implicitamente una specializzazione della classe Object, da cui eredita un insieme di funzionalità, che può ridefinire:

- `toString()`, fornisce una rappresentazione String dell'oggetto su cui viene invocato: l'implementazione base è `nome_classe@codice_hash_oggetto`;
- `clone()`, fornisce un clone dell'oggetto corrente. Il tipo di ritorno è Object quindi va sempre effettuato un casting;
- `equals()`, nella sua versione base, verifica se due riferimenti sono uguali, non se due oggetti hanno attributi uguali.

::.. Ereditarietà (4/6)



Quando si invoca un metodo su un oggetto, l'interprete ne cerca dapprima la definizione nella classe dell'oggetto stesso; se non la trova, cerca nella superclasse e risale la gerarchia.

Nel caso esistano più metodi con lo stesso nome, tipo restituito e stessi parametri (firma) viene eseguito il metodo trovato per primo.

::.. Ereditarietà (5/6)

Consideriamo il seguente codice:

```
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
      // ...
      i.play();
    }
}

class Wind extends Instrument {
    public static void main(String[] args)
    {
      Wind flute = new Wind();
      Instrument.tune(flute);
    }
}
```

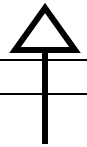
Tune ha come parametro di ingresso oggetti di tipo Instrument, come mai non ho un errore trasmettendo un tipo Wind?

Upcasting

La classe Wind estende quella Instrument, quindi un oggetto di tipo Wind è anche di tipo Instrument.

::.. Ereditarietà (6/6)

```
public class Persona {
    protected int eta;
    protected String nome;
    protected String sesso;
    public Persona(String nome, String sesso, int eta)   {
        this.nome=new String(nome);
        this.sesso=new String(sesso);
        this.eta=eta;
    }
    public void ChiSei() {
        System.out.println("Sono una persona di nome"+nome+", sesso: "+sesso+", eta: "+eta);
    }
}
```



```
public class Studente extends Persona {
   protected int esami;
   protected int matricola;
   protected String facolta;
   public Studente(String nome, String sesso, int eta, int esami,int matricola, String facolta)   {
       super(nome, sesso,eta); //chiamata esplicita al costrutt. della classe base
       this.esami=esami;
       this.matricola=matricola;
       this.facolta=new String(facolta);
   }
   public void ChiSei()              {
       super.ChiSei();
       System.out.println("In particolare sono uno studente iscritto alla facolta di "+facolta+
       ",matricola: "+matricola+", esami: "+esami);
   }
}
```

::.. Ereditarietà (6/6)

```
public class Persona {
    protected int eta;
    protected String nome;
    protected String sesso;
    public Persona(String nome, String sesso, int eta)   {
        this.nome=new String(nome);
        this.sesso=new String(sesso);
        this.eta=eta;
    }
    public void ChiSei() {
        System.out.println("Sono una persona di nome"+nome+", sesso: "+sesso+", eta: "+eta);
    }
}
```

```
public class Studente extends Persona {
   protected int esami;
   protected int matricola;
   protected String facolta;
   public Studente(String nome, String sesso, int eta, int esami, int matricola, String facolta)  {
        super(nome, sesso,eta); //chiamata costruttore classe base
        this.esami=esami;
        this.matricola=matricola;
        this.facolta=new String(facolta);
   }
   public void ChiSei()   {
        super.ChiSei();
        System.out.println("In particolare sono uno studente iscritto alla facolta di "+facolta+
        ",matricola: "+matricola+", esami: "+esami);
   }
}
```

Visibilità dei metodi/attributi

Inizializzazione classe base

Invocazione metodi della classe base

::.. Visibilità in una gerarchia gen-spec (1/2)

Che visibilità hanno i metodi/attributi di una classe base rispetto ad una classe derivata? Dipende dalla parola chiave che li precede:

Accesso privato (private)	**Visibilità solo nell'ambito della stessa classe**
Accesso di default (friendly)	**Visibilità solo nell'ambito della stesso package**
Accesso protetto (protected)	**Visibilità nelle sottoclassi e nello stesso package**
Accesso pubblico (public)	**Visibilità completa**

::.. Visibilità in una gerarchia gen-spec (2/2)

Il codice è strutturabile in insiemi di moduli (package).

- Solo all'inizio di un modulo java si può specificare l'appartenenza ad un package:

```
package mypackage;
public class MyClass { // . . .
```

- Per utilizzare MyClass dall'esterno del package si possono usare le due modalità:

```
mypackage.MyClass m = new mypackage.MyClass();
```

Oppure:

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

- i package sono strutturati in uno schema ad albero:

  Es: com.sun.media.rtp

- Senza definire uno specifico package una classe verrà inserita nel default package.

::.. Inizializzazione classe base

Quando si crea un oggetto della classe derivata, questo contiene al suo interno un sotto-oggetto della classe base.

Il sotto-oggetto va opportunamente inizializzato per mezzo della chiamata al costruttore (super(..)); Tale chiamata può essere:

- Implicita: se la classe base ha un costruttore a zero argomenti.

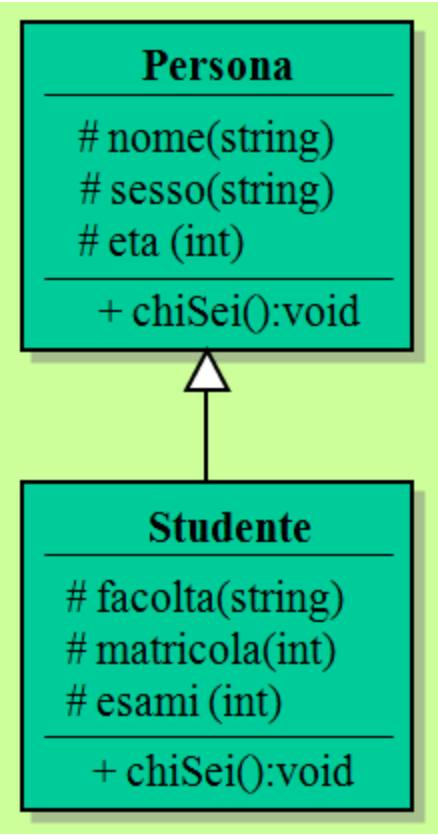
Invece deve essere:

- Esplicita: se il costruttore della classe base prevede almeno un argomento.

Quando non viene esplicitata l'invocazione al costruttore della classe base, il compilatore java automaticamente lo pone in testa al costruttore della classe derivata.

::.. Invocazione metodi della classe base

Quando si implementa una classe derivata, è possibile che un suo metodo abbia la stessa firma di un metodo della classe base, e questo viene implementato invocando il metodo della classe base ed aggiungendo altra logica applicativa.



All'interno del metodo chiSei() di studente non è possibile invocare lo stesso metodo di Persona scrivendo "chiSei()".

Per risolvere questo problema si ricorre alla parola chiave super, quindi in chiSei() di Studente troviamo correttamente:

```
super.chiSei()
```

per riferirci al chiSei() di Persona.

::.. Composizione vs Ereditarietà

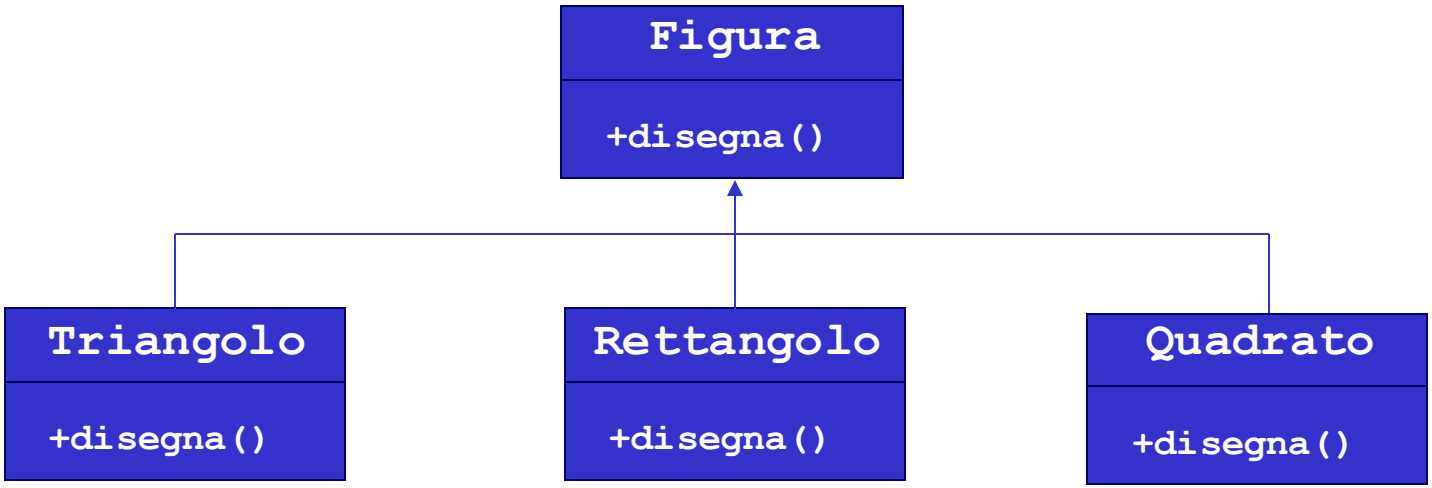
È prassi comune combinare composizione ed ereditarietà. Bisogna considerare che mentre il compilatore forza l'inizializzazione delle classi base, non lo fa per l'inizializzazione dei riferimenti delle classi componenti. Quindi bisogna ricordarsi di inizializzare sempre tali riferimenti.

Qual è la differenza tra le due tecniche e quando preferire una rispetto all'altra?

- **La composizione implementa la relazione "has-a", ovvero una classe vuole solo incorporare le funzionalità di un'altra;**
- **L'ereditarietà implementa la relazione "is-a", ovvero una classe non solo vuole incorporare le funzionalità di un'altra, ma vuole presentarsi all'utente con lo stesso vestito (interfaccia).**

::.. Polimorfismo (1/13)

Per **polimorfismo** si intende la proprietà di una entità di assumere forme diverse nel tempo.



- A è un vettore di tipo Figura, composto di N istanze di Triangolo, Rettangolo, Quadrato;
- Si consideri il seguente ciclo di istruzioni:

```
for i = 1 to N do
        A[i].disegna();
```

- L'esecuzione del ciclo richiede che sia possibile determinare dinamicamente l'implementazione della operazione disegna() da eseguire, in funzione del tipo dell'oggetto A[i].

::.. Polimorfismo (2/13)

Il collegamento tra la chiamata di un metodo con il corpo del metodo stesso prende il nome di **binding**.

Quando questa operazione avviene prima dell'esecuzione si parla di **early binding** o binding statico. Quando avviene a tempo di esecuzione, si dice **late binding** o binding dinamico.

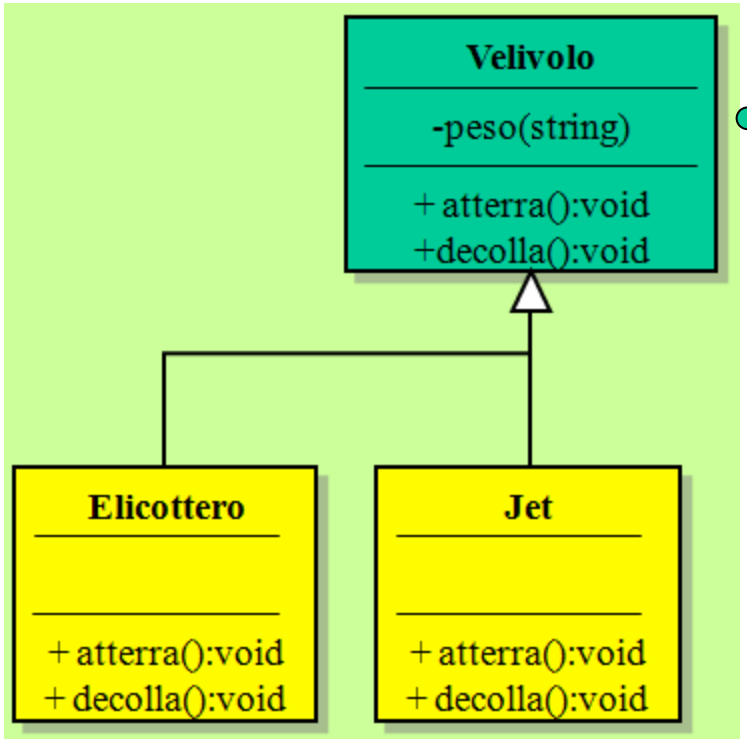
Java realizza sempre late binding a meno che al metodo non sia anteposta la parola chiave final, il cui compito è prevenire che un utente possa realizzare l'"override" di un metodo.

Vantaggio del polimorfismo:

- Supporto della proprietà di estensibilità di un sistema: si minimizza la quantità di codice che occorre modificare quando si estende il sistema, cioè si introducono nuove classi e nuove funzionalità.

::.. Polimorfismo (3/13)

Esempio di polimorfismo

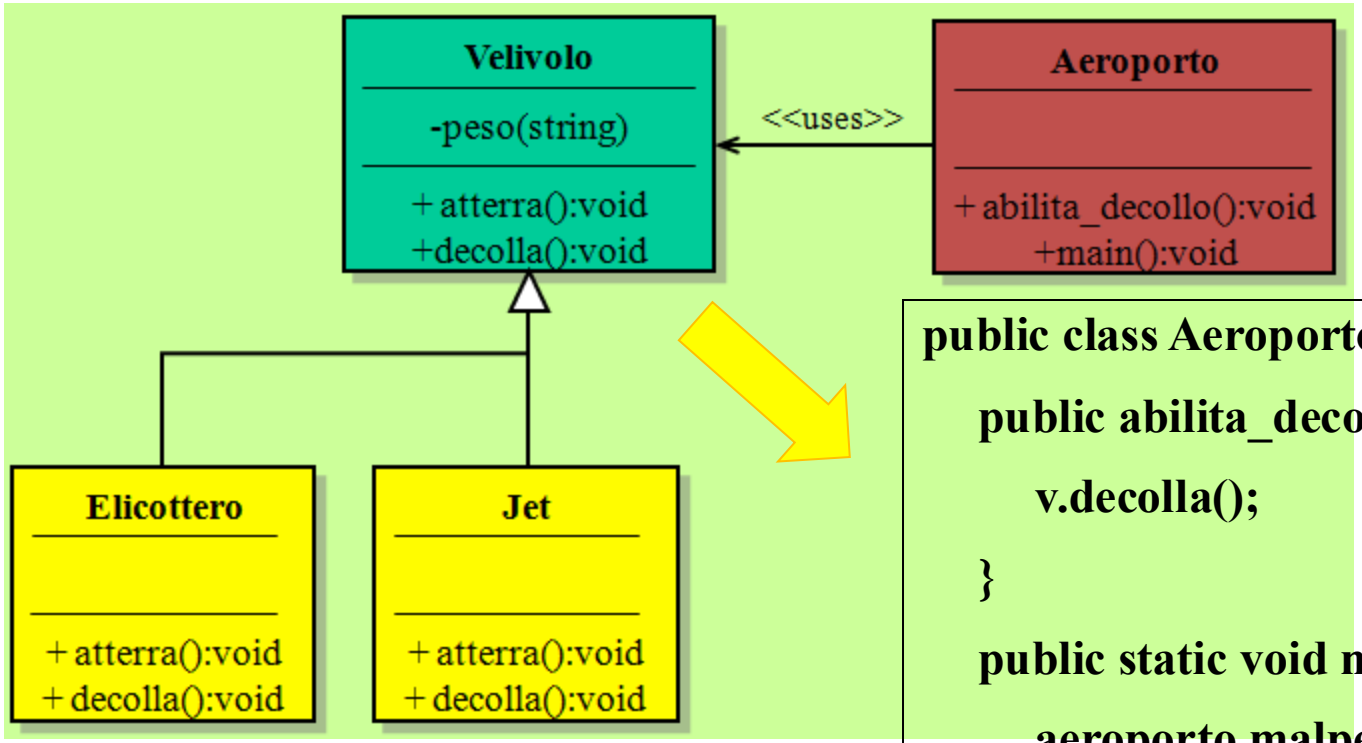


Velivolo funge da "interfaccia comune" per le classi derivate.

Le classi derivate implementano i metodi della classe base specializzandone il comportamento.

::.. Polimorfismo (4/13)

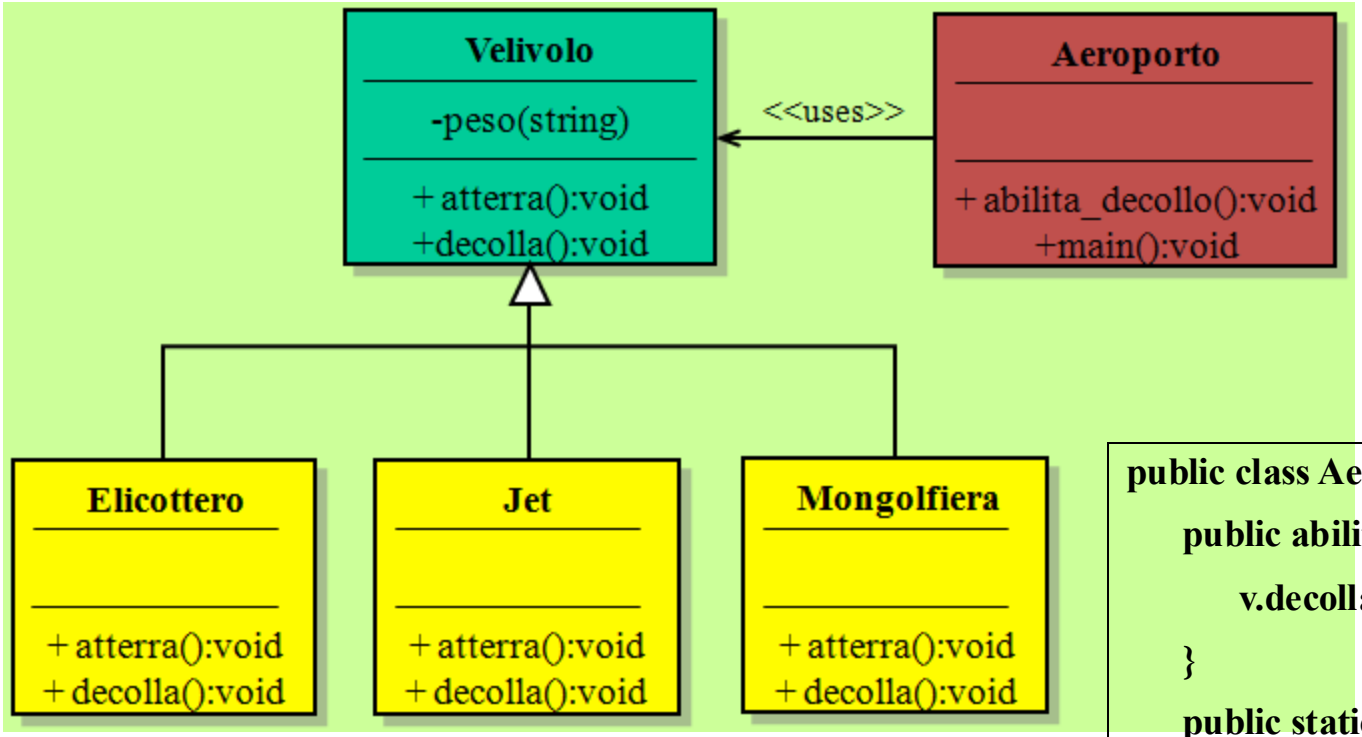
Esempio di polimorfismo




```
public class Aeroporto {
    public abilita_decollo (Velivolo v){
        v.decolla();
    }
    public static void main() {
        aeroporto malpensa=new aeroporto();
        Velivolo MI_NA=new Jet();
        malpensa.abilita_decollo(MI_NA);
     }
}
```

::.. Polimorfismo (5/13)

Aggiungendo una nuova classe alla gerarchia...



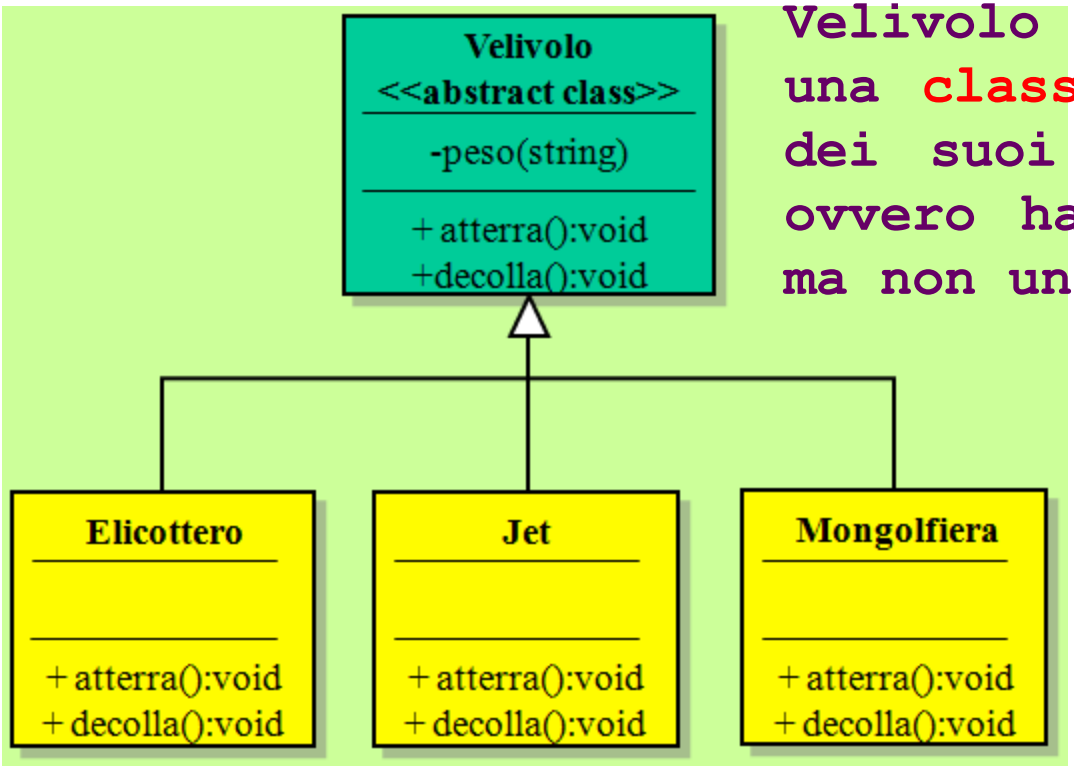
...modificando Aeroporto...

```
public class Aeroporto {
    public abilita_decollo (Velivolo v){
        v.decolla();
    }
    public static void main() {
        aeroporto malpensa=new aeroporto();
        Velivolo MI_NA=new Mongolfiera();
        malpensa.abilita_decollo(MI_NA);
    }
}
```

... il codice esegue senza errori.

::.. Polimorfismo (6/13)

La classe Velivolo può fungere solo da interfaccia per specificare come gli utenti devono utilizzare le classi derivate, ma non il loro comportamento. Pertanto, Velivolo deve essere solo un'interfaccia, ovvero un insieme di funzioni dummy che poi ogni derivata deve implementare.



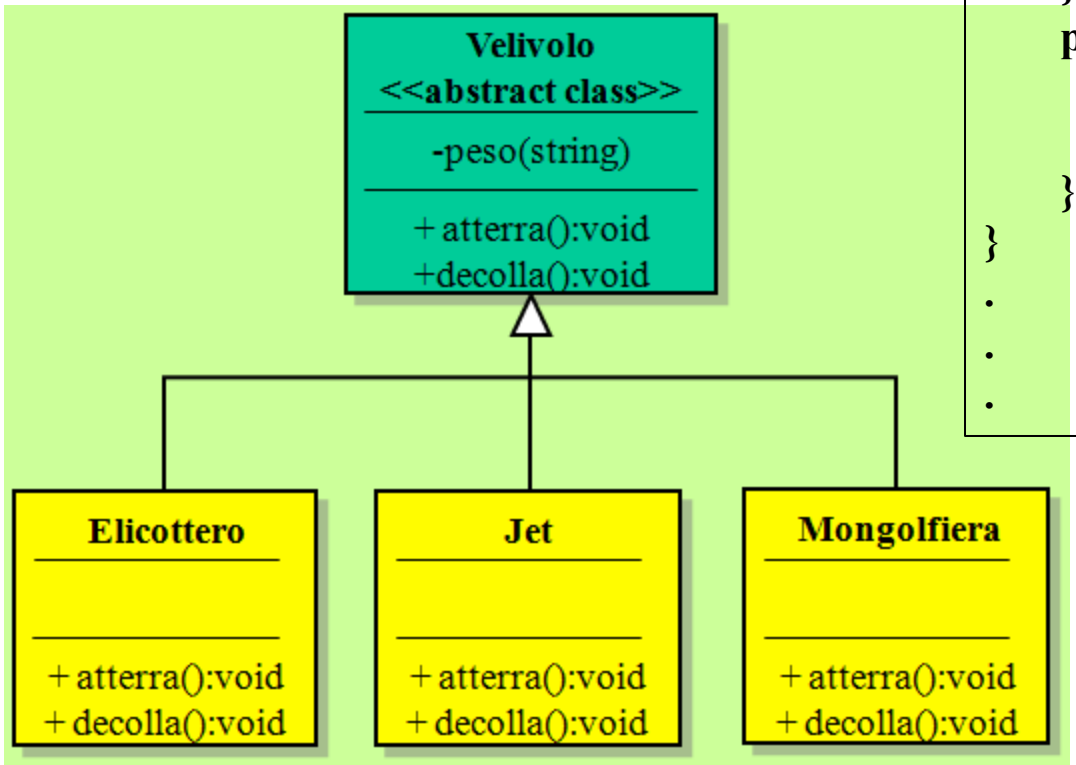
Velivolo è implementabile come una classe astratta: almeno uno dei suoi metodi sono astratti, ovvero hanno solo dichiarazioni ma non un corpo.

Non è possibile istanziare oggetti di classi astratte o con alcuni metodi astratti.

::.. Polimorfismo (7/13)

Implementato in Java...

```
abstract class Velivolo {
      string peso;
      public abstract void atterra();
      public abstract void decolla();
}
class Elicottero extends Velivolo{
      public void play() {
          System.out.println("Elicottero.play()");
      }
      public void decolla() {
          System.out.println("Elicottero.decolla()"
          );
      }
}
.
.
.
```



::.. Polimorfismo (8/13)

Le classi astratte non sono l'unica soluzione che Java mette a disposizione per realizzare interfacce comuni a un insieme di classi. "Interface" è a tutti gli effetti una classe astratta, con metodi astratti e attributi, che sono implicitamente static e final.

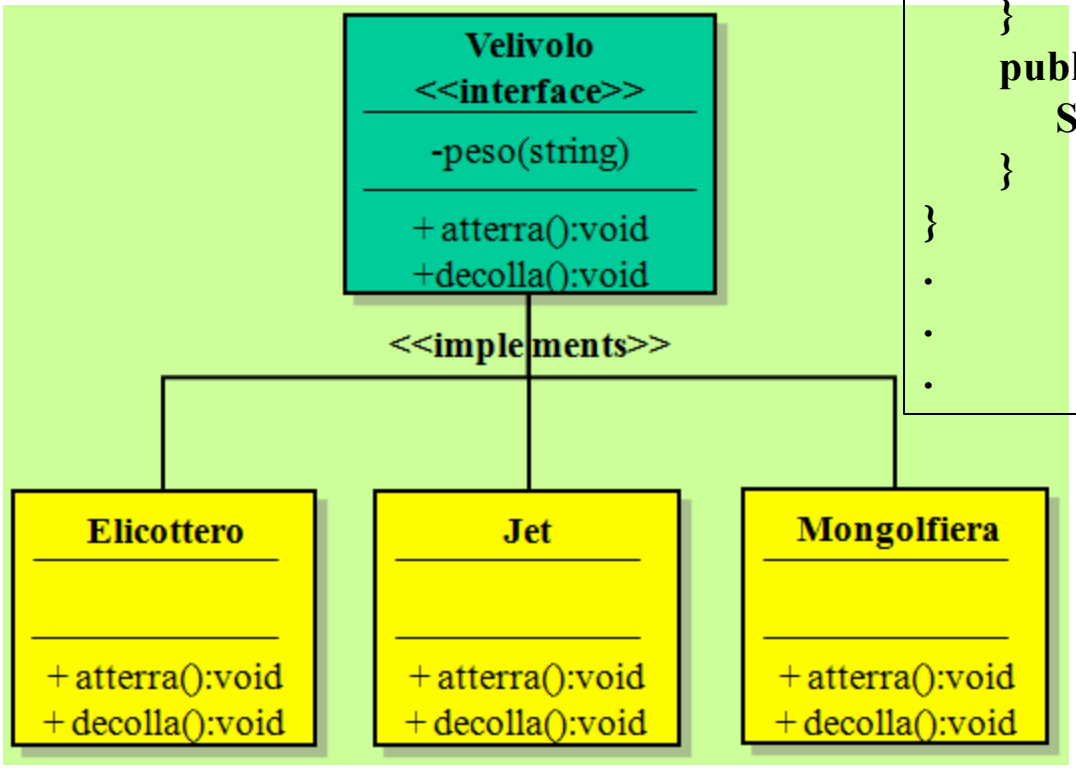
Usando delle classi astratte, il programmatore ha a disposizione solo l'ereditarietà singola (una classe eredita solo da una classe astratta), con le interfacce si può realizzare anche un'implementazione multipla (una classe può implementare più interfacce).

I metodi di un'interfaccia hanno automaticamente visibilità public, e la classe che li implementa deve dichiarare una visibilità public, altrimenti per default sono "friendly".

::.. Polimorfismo (9/13)

Implementato in Java...

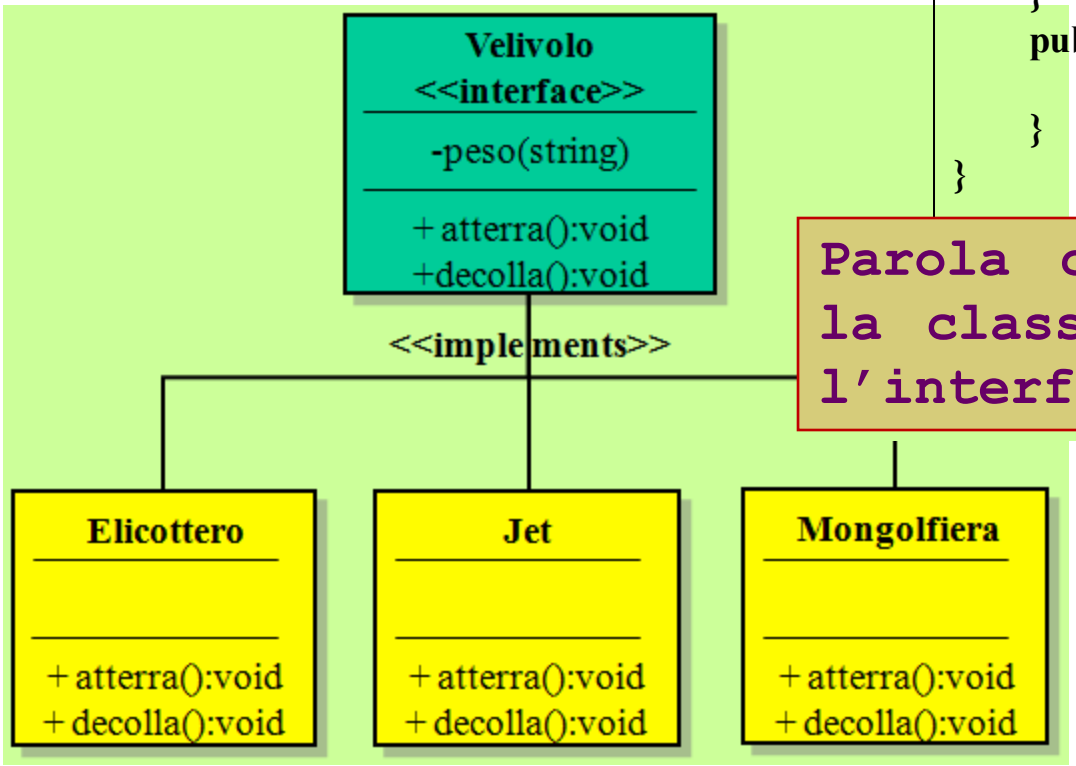
```
interface Velivolo {
        string peso;
        void atterra();
        void decolla();
}
class Elicottero implements Velivolo{
        public void play() {
            System.out.println(" Elicottero.play()");
        }
        public void decolla() {
            System.out.println(" Elicottero.decolla()");
        }
}
.
.
.
```



::.. Polimorfismo (10/13)

Implementato in Java...

```
interface Velivolo {
        string peso;
        void atterra();
        void decolla();
}
class Elicottero implements Velivolo{
        public void play() {
            System.out.println(" Elicottero.play()");
        }
        public void decolla() {
            System.out.println(" Elicottero.decolla()");
        }
}
```



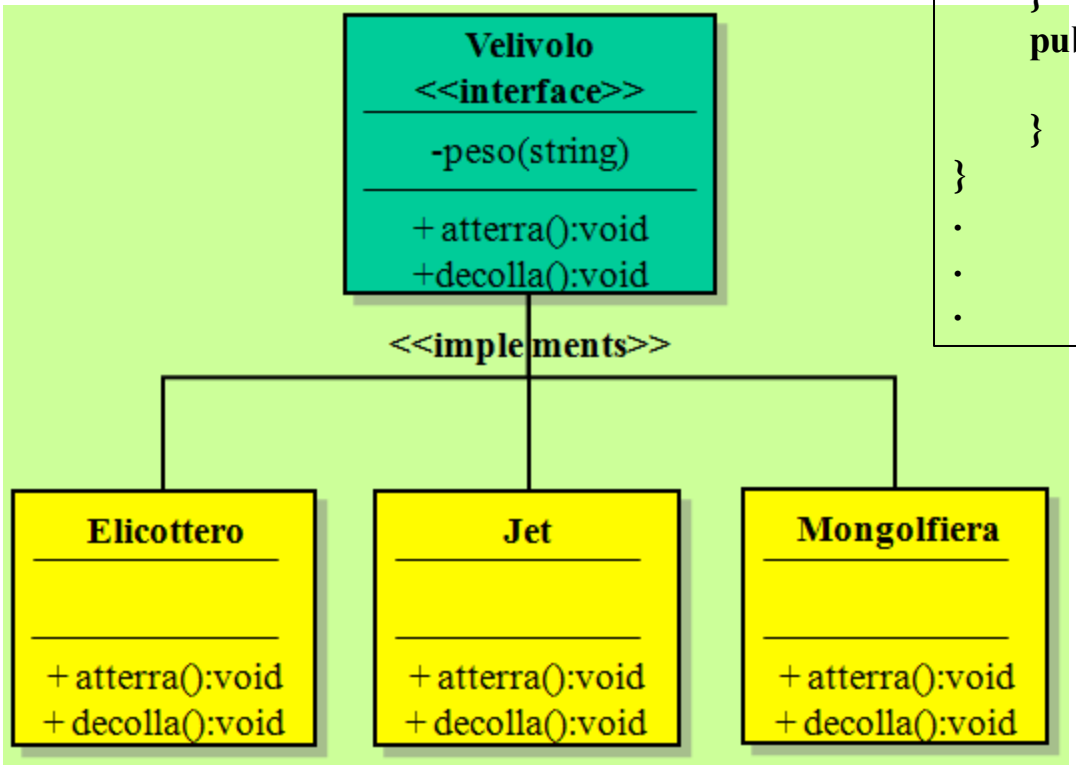
Parola chiave per indicare che la classe Elicottero implementa l'interfaccia Velivolo.

::.. Polimorfismo (11/13)

Implementato in Java...

```
interface Velivolo {
        string peso;
        void atterra();
        void decolla();
}
class Elicottero implements Velivolo{
        public void play() {
            System.out.println(" Elicottero.play()");
        }
        public void decolla() {
            System.out.println(" Elicottero.decolla()");
        }
}
.
.
.
```

Errore

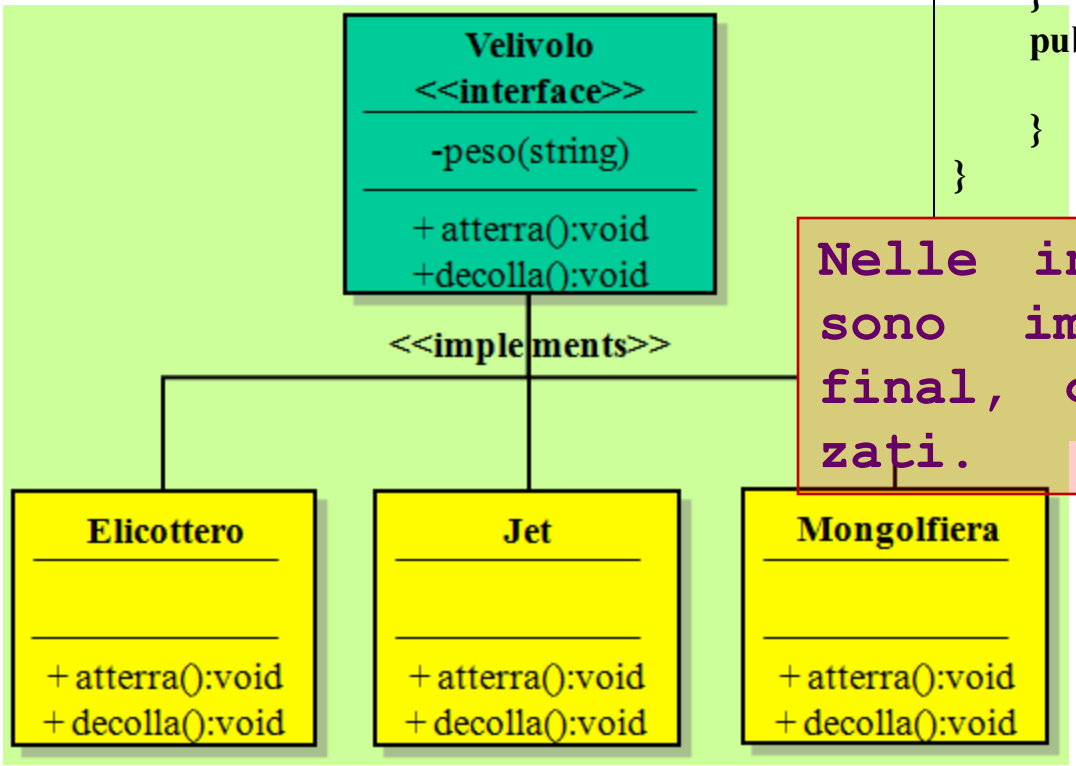


È corretto il codice così come è scritto?

::.. Polimorfismo (12/13)

Implementato in Java...

```
interface Velivolo {
      string peso = “100”;
      void atterra();
      void decolla();
}
class Elicottero implements Velivolo{
      public void play() {
          System.out.println(" Elicottero.play()");
      }
      public void decolla() {
          System.out.println(" Elicottero.decolla()");
      }
}
```



Nelle interfacce gli attributi sono implicitamente static e final, quindi vanno inizializzati.

::.. Polimorfismo (13/13)

È possibile derivare un'interfaccia a partire da un'altra interfaccia, ma questo legame di derivazione è solo a livello di specifica dei metodi, e non di implementazione (un'interfaccia non può estendere una classe con metodi concreti perché un'interfaccia deve avere tutti i suoi metodi astratti, anche se alcuni di essi sono ereditati da terze parti).

Esempio:

```
public interface Remote{...}

...

import java.rmi.Remote;

public interface ISquareRoot extends Remote{
   double calculateSquareRoot(double aNumber);
}
```

::.. Classi interne (1/6)

In Java è possibile posizionare la definizione di una classe all'interno di un'altra, realizzando quelle che prendono il nome di **classi interne**. Ciò consente di raggruppare classi che sono logicamente correlate e di controllare la loro visibilità.

```
public class Document {
    class Destination {
            private String label;
            Destination(String whereTo) {label = whereTo;}
            String readLabel() {return label;}
    }
    public void ship(String dest) {
            Destination d = new Destination(dest);
            System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
            Document doc = new Document();
            p.ship("Tanzania");
    }
}
```

::.. Classi interne (1/6)

In Java è possibile posizionare la definizione di una classe all'interno di un'altra, realizzando quello che prende il nome di **classe interna**. Ciò consente di raggruppare classi che sono logicamente correlate e di controllare la loro visibilità.

```
public class Document {
    class Destination {
            private String label;
            Destination(String whereTo) {label = whereTo;}
            String readLabel() {return label;}
    }
     public void ship(String dest) {
```

Nessuna differenza nell'uso di Destination, bisogna solo ricordarsi che i nomi sono innestati nella classe Document, quindi se si vuole far riferimento alla classe interna al di fuori di Document (o in metodi statici), la sintassi è Document.Destination.

```
    }
}
```

::.. Classi interne (2/6)

Ogni classe interna ha completa visibilità degli attributi e dei metodi della classe che la contiene.

```
public class Document {
    private String title = "Titolo";
    class Destination {
            private String label;
            Destination(String whereTo) {label = whereTo;}
            String readLabel() {return label;}
            String returnTitolo() {return Document.this.title;}
    }
    public void ship(String dest) {
            Destination d = new Destination(dest);
            System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
            Document doc = new Document();
            Document.Destination dec = doc.new Destination("Tanzania");
            String str = dec.returnTitolo();
    }
}
```

::.. Classi interne (2/6)

Ogni classe interna ha completa visibilità degli attributi e dei metodi della classe che la contiene.

```
public class Document {
    private String title = "Titolo";
    class Destination {
        private String label;
        Destination(String whereTo) {label = whereTo;}
        String readLabel() {return label;}
                                                    tle;}
    }
    publi
                                                    dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Document doc = new Document();
        Document.Destination dec = doc.new Destination("Tanzania");
        String str = dec.returnTitolo();
    }
}
```

Un'istanza di una classe interna è ottenibile solo a partire da un'istanza della classe esterna.

::.. Classi interne (2/6)

Nel corpo di un metodo di una classe interna si può fare esplicito riferimento ai membri della classe contenente mediante la notazione: NomeClasseContenente.this.nomeMembro.

```
        private String label;
        Destination(String whereTo) {label = whereTo;}
        String readLabel() {return label;}
        String returnTitolo() {return Document.this.title;}
    }
    public void ship(String dest) {
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Document doc = new Document();
        Document.Destination dec = doc.new Destination("Tanzania");
        String str = dec.returnTitolo();
    }
}
```

::.. Classi interne (2/6)

Possiamo disciplinare la visibilità della classe interna con le parole chiave public, private e protected.

```
private String title = "Titolo";
private class Destination {
        private String label;
        Destination(String whereTo) {label = whereTo;}
        String readLabel() {return label;}
        String returnTitolo() {return Document.this.title;}
}
public void ship(String dest) {
        Destination d = new Destination(dest);
}
publi
        Document doc = new Document();
        Document.Destination dec = doc.new Destination("Tanzania");
        String str = dec.returnTitolo();
}
}
```

Così da non rendere possibile istanziare un oggetto della classe interna

::.. Classi interne (3/6)

Le classi interne non possono dichiarare metodi statici. Ma è possibile che una classe interna sia static.

Un'istanza di una classe interna implicitamente mantiene un riferimento all'oggetto della classe esterna che lo ha creato. Questo non è vero per **classi interne statiche**:

- Non si ha bisogno di un oggetto della classe esterna per creare un oggetto della classe interna;
- Non è possibile accedere ad un oggetto della classe esterna da un oggetto della classe interna static;
- Può contenere metodi e attributi statici.

Normalmente non è consentito inserire codice all'interno di una interfaccia, ma una classe interna statica può essere parte di una interfaccia.

::.. Classi interne (4/6)

Una classe può essere definita all'interno di un metodo, oppure di un blocco di codice (nel ramo di un costrutto if...else): classe interna locale.

```
Enumeration myEnumerate(final Object array[]) {
    class E implements Enumeration {
            int count = 0;
            public boolean hasMoreElements(){
                    return count < array.length;
            }
            public Object nextElement() {
                    return array[count++];
            }
    }
    return new E();
}
```

Le classi interne ai metodi hanno completa visibilità di tutte le variabili locali al metodo. Non è possibile specificare esplicitamente l'ambito di visibilità delle classi interne ai metodi – esse sono private ai metodi che le includono per definizione.

::.. Classi interne (4/6)

Una classe può essere definita all'interno di un metodo, oppure di un blocco di codice (nel ramo di un costrutto if...else): classe interna locale.

```
Enumeration myEnumerate(final Object array[]) {
    class E implements Enumeration {
        int count = 0;
        public boolean hasMoreElements(){
            return count < array.length;
        }
    return new E();
}
```

Per rendere visibile all'interno della classe interna il parametro passato al metodo, questo deve essere dichiarato final, altrimenti si ha un errore a tempo di compilazione.

Le classi interne ai metodi hanno completa visibilità di tutte le variabili locali al metodo. Non è possibile specificare esplicitamente l'ambito di visibilità delle classi interne ai metodi – esse sono private ai metodi che le includono per definizione.

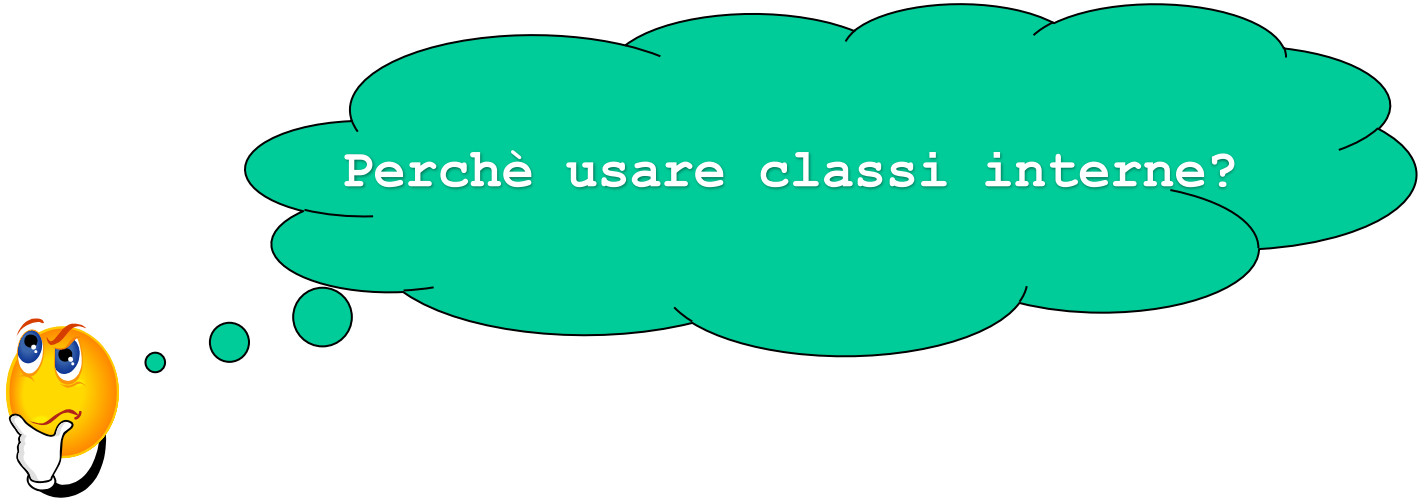
::.. Classi interne (5/6)

Una classe interna a un metodo può anche essere resa anonima al momento della creazione della sua istanza:

```
Enumeration myEnumerate(final Object array[]) {
    return new Enumeration() {
            int count = 0;
            public boolean hasMoreElements(){
                    return count < array.length;
            }
            public Object nextElement() {
                    return array[count++];
            }
    }
}
```

Le classi interne anonime sono sottoclassi delle classi specificate dopo l'operatore new. Nel caso in cui l'operatore new sia seguito dal nome di una interfaccia, la classe anonima si considera implementazione dell'interfaccia. Le classi anonime non possono avere costruttori.

::.. Classi interne (6/6)



Ogni classe interna può ereditare da un'altra classe o implementare un'interfaccia indipendentemente da quanto fatto dalla classe esterna.

Questo fornisce una ulteriore soluzione (oltre al ricorso alle interfacce) al problema dell' impossibilità della derivazione multipla presente in Java.