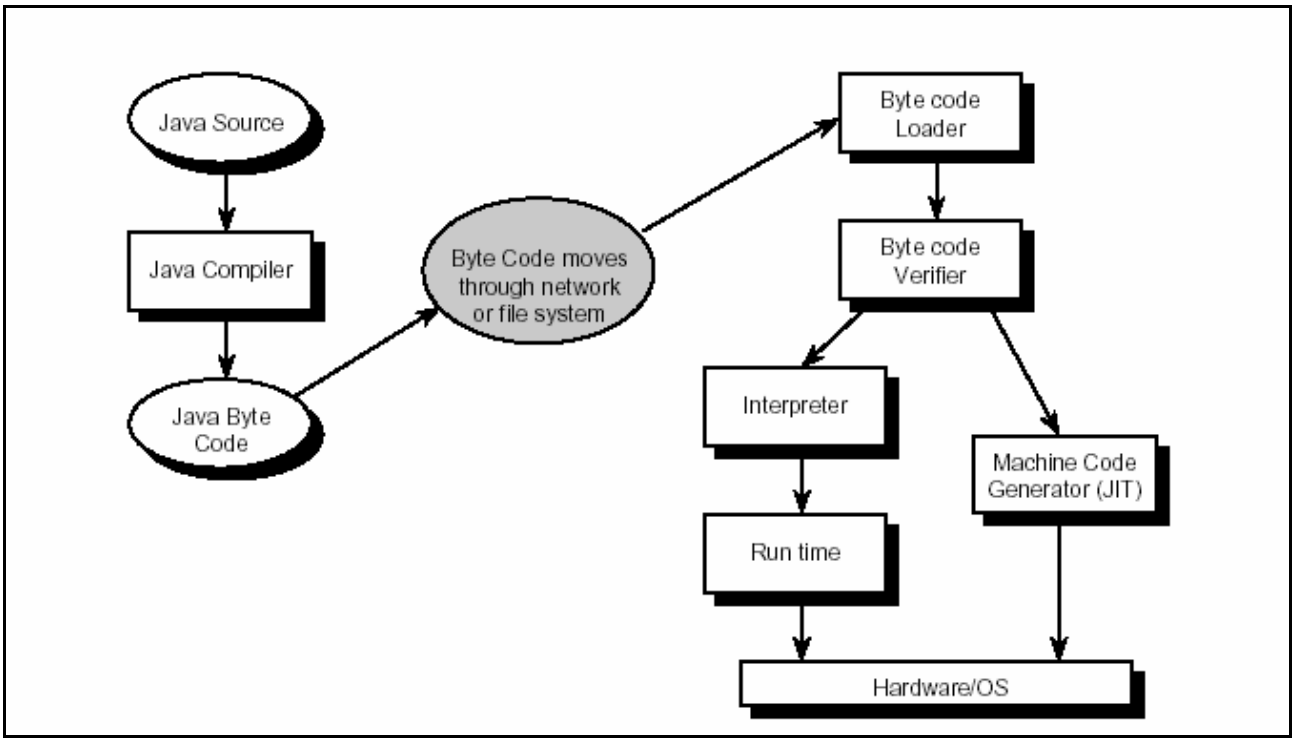
INGEGNERIA DEL SOFTWARE

JAVA

Avvertenza: gli appunti si basano sul corso di Ingegneria del Software tenuto dal prof. Picco della facoltà di Ingegneria del Politecnico di Milano (che ringrazio per aver acconsentito alla pubblicazione). Essendo stati integrati da me con appunti presi a lezione, il suddetto docente non ha alcuna responsabilità su eventuali errori, che vi sarei grato mi segnalaste in modo da poterli correggere.

e-mail: webmaster@morpheusweb.it
web: http://www.morpheusweb.it

L’AMBIENTE JAVA

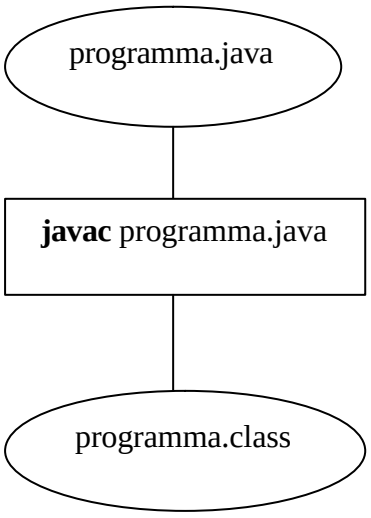


Java è in parte compilato ed in parte interpretato.

All’inizio il sorgente viene compilato del **COMPILATORE** tramite il comando:

javac programma.class,

che genera un file chiamato **programma.class** contenente il ***byte code***. E’ la traduzione del programma in un linguaggio intermedio



Il byte code può essere interpretato su qualsiasi piattaforma che supporta Java tramite il comando:

java programma

La caratteristica è più importante per sistemi distribuiti, che tipicamente hanno macchine eterogenee.

Java supporta inoltre la **mobilità del codice**, posso cioè rilocare il codice di un'applicazione mentre questa è in esecuzione.

Il byte code arriva al **CLASS LOADER** (o **code loader**) tramite la rete oppure il file system.
Il class loader si occupa di risolvere il nome di ogni classe, occorre sapere com'è fatta ogni classe per poter allocare la memoria.
Il class loader quindi reperisce il codice della classe.

> Nota:
> Il class loader di Java è riprogrammabile tramite delle API (application program interface) mediante cui ci può ridefinire il comportamento del class loader.

Di default il class loader riceve la classe da caricare. Cerca il codice sul disco fisso (o nel nodo di rete in caso di applet) in un insieme di directory definite in una variabile di ambiente (tipo il PATH di dos e windows) detta **CLASSPATH**. Se la classe viene trovata, viene caricata in memoria, altrimenti viene sollevata un'eccezione.

Il byte code viene poi verificato (controllando che sia compatibile con il sorgente) dal **BYTE CODE VERIFIER** poiché viaggiando in rete potrebbe essere manomesso.

Ci sono infine due modalità:

INTERPRETAZIONE: ogni istruzione del byte code è verificata ed eseguita

ESECUZIONE JUST IN TIME: il file .class è tradotto da codice intermedio a nativo (dipendente dalla macchina). Non è compilato tutto il programma ma ogni singola classe man mano che serve (per diluire durante l'esecuzione i costi in termini di tempo dovuti alla traduzione)

NOZIONI PRELIMINARI

IL PRIMO PROGRAMMA

```
public class HelloWorld {
public static void main(String args[]){
      System.out.println(“Hello world!”);
      }
}
```

Il main non è una funzione, è un metodo. Quando eseguo un programma java, devo lanciare il programma che contiene il metodo main.

> ***Nota:***
> *Per il debug, si consiglia di mettere un metodo main in ogni classe così da poter fare il testing delle unità.*

```
public class HelloWorld {
```

specifico il metodo di accesso alla classe

```
public static void main(String args[]){
```

può esserci o meno davanti ad un metodo a attributo, indica che sono metodi o attributi **di classe**.

Per gli attributi: ne esiste uno per ogni classe, non uno per ogni oggetto (potrebbe essere ad esempio usato per implementare le variabili globali), un esempio classico è dato dall’implementazione di un contatore di oggetti allocati.

I metodi static possono essere usati per modificare attributi static oppure anche per realizzare delle librerie di funzioni.

COMPILAZIONE ED ESECUZIONE

```
c:\classes>javac HelloWorld.java
c:\classes>java HelloWorld
HelloWorld!
c:\classes>
```

LA STRUTTURA DI UN PROGRAMMA JAVA

Un programma Java è organizzato come un insieme di classi
Classi diverse possono essere raggruppate all’interno della stessa “Compilation unit” che hanno un estensione **.java**

Il programma principale è rappresentato da un metodo speciale (main) della classe il cui nome coincide con il nome del programma

JAVA - IL PRIMO PROGRAMMA GRAFICO

```
import java.awt.*;
package examples,
class HelloWorldWindow {
public static void main(String args[]) {
Frame f=new Frame("HelloWorldWindow");
f.add(new Label("HelloWorld!",Label.CENTER),"Center");
f.pack();
f.setVisible(true);
}
}
```

import: importa caratteristiche particolari di un particolare package.

package examples: il programma è dichiarato nel package examples. Il file deve essere nella directory examples. lo compilo li e lo eseguo (oppure in una dir esterna, ma riferendomi al nome completo della classe che è: examples.HelloWorld)

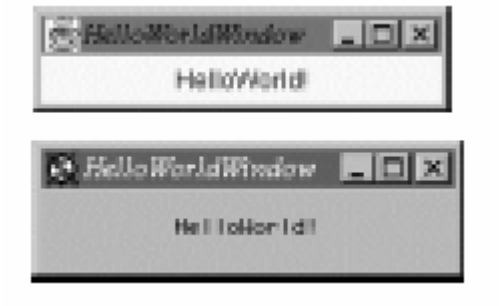
Un package definisce uno spazio dei nomi; consente di avere un meccanismo per strutturare lo spazio dei nomi (posso ad esempio definire due classi con lo stesso nome in package diversi). Vincola inoltre il modo in cui i file sono memorizzati nel file system.

I package possono essere annidati: ad esempio

myProg

- examples
- lectures

accederei ad una classe di examples con la notazione puntata: **myProg.examples.NomeClasse**



COMPILIAZIONE ED ESECUZIONE

```
c:\classes>cd examples
c:\classes\examples>javac HelloWorldWindow.java
c:\classes\examples>cd ..
c:\classes>java examples.HelloWorldWindow
c:\classes>
```

LA STRUTTURA DI UN PROGRAMMA JAVA

- **Package**
  - Ogni package contiene una o più compilation unit
  - Il package introduce un nuovo ambito di visibilità dei nomi
- **Compilation unit**
  - Ogni compilation unit contiene una o più classi o interfacce delle quali una sola pubblica
- **Classi e interfacce**
- Relazioni con il file system
  - package ⇔ directory
  - compilation unit ⇔ file

IL LINGUAGGIO JAVA IN DETTAGLIO

- Programmazione “in the small”
  - Tipi primitivi
  - Dichiarazione di variabili
  - Strutture di controllo: selezione condizionale, cicli
- sono identiche a quelle del C: if, switch, while, for...
  - Array
- Programmazione “in the large”
  - Classi
  - Interfacce
  - Packages

TIPI PRIMITIVI E VARIABILI

Tipi numerici:

- byte: 8 bit
- short: 16 bit
- int: 32 bit
- long: 64 bit
- float: 32 bit
- double: 64 bit

Altri tipi:

- boolean: true o false
- char: 16 bit, carattere Unicode

I tipi primitivi sono allocati nello stack (sempre), mentre in C++ li potevo allocare nello heap mediante i puntatori.

Per allocare i tipi primitivi nello heap si possono usare i wrapper, che mi permettono di avvolgere all'interno di una classe un tipo primitivo.

DICHIARAZIONE DI VARIABILI

```
byte un_byte;
int a, b=3, c;
char c='h', car;
boolean trovato=false;
```

I TIPI DI RIFERIMENTO

- Tipi array
- Tipi definiti dall'utente
  - Classi
  - Interfacce

I tipi oggetto sono sempre allocati nello heap (mentre in C++ se li definivo come tipi normali, venivano allocati sullo stack e venivano allocati nello heap solo con la *new*)

TIPI ARRAY

Anche gli array sono considerati tipi riferimento e sono allocati nello heap.

Dato un tipo T (predefinito o definito dall'utente) un array di T è definito come: **`T[]`**

Similmente sono dichiarati gli array multidimensionali: **`T[][] T[][][]`** …

Esempi:
int[] *//array di interi*
float[][] *//matrice di float*
Persona[] *//array di elementi riferimento*

TIPI ARRAY: DICHIARAZIONE E INIZIALIZZAZIONE

Dichiarazione:
int[] ai1, ai2;
float[] af1;
double ad[];
Persona[][] ap;

Inizializzazione:
int[] ai={1,2,3};
double[][] ad={{1.2, 2.5}, {1.0, 1.5}}

Posso anche inizializzarli al volo.

TIPI ARRAY: ALLOCAZIONE DI MEMORIA

In mancanza di inizializzazione, la dichiarazione di un array non alloca spazio per gli elementi dell'array.

L'allocazione si realizza dinamicamente tramite l'operatore:

`new <tipo> [<dimensione>]`

La dimensione dell'array deve essere nota.
Esiste una classe **Vector** che consente di avere più oggetti dello stesso tipo, senza dover dichiararne il numero.

Esempi:
int[] i=new int[10], j={10,11,12};
float[][] f=new float[10][10];
Persona[] p=new Persona[30];

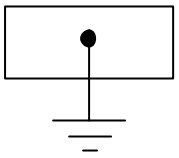
Se gli elementi non sono di un tipo primitivo, l'operatore "new" alloca solo lo spazio per i riferimenti. In C++ se dichiaro una variabile senza inizializzarla, a questa non viene assegnato nessun valore, in Java viene assegnato un valore di default che per i riferimenti è "null".

ARRAY DI OGGETTI: DEFINIZIONE

```
Person[] person;
```

crea lo spazio per il riferimento, l'array vero e proprio non esiste

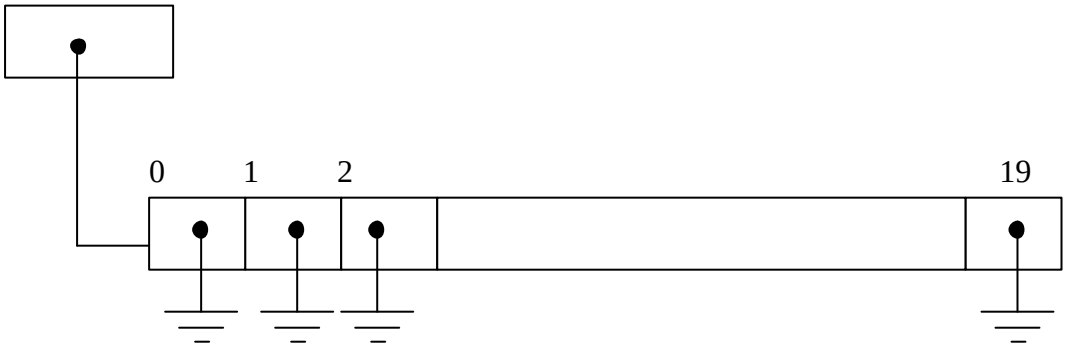
person



```
Person[] person;
person = new Person[20];
```

Ora l'array è stato creato ma i diversi oggetti di tipo Person non esistono ancora.

person

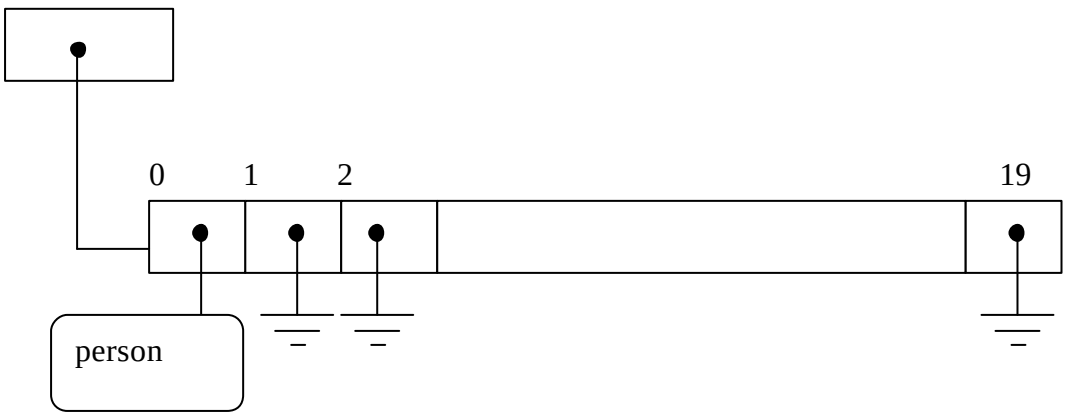


L'allocazione crea 20 celle di memoria logiche tutte contenenti un riferimento inizializzato a null.

```
Person[] person;
person = new Person[20];
person[0] = new Person()
```

Un oggetto di tipo persona è stato creato e tale oggetto è stato inserito in posizione 0.

person



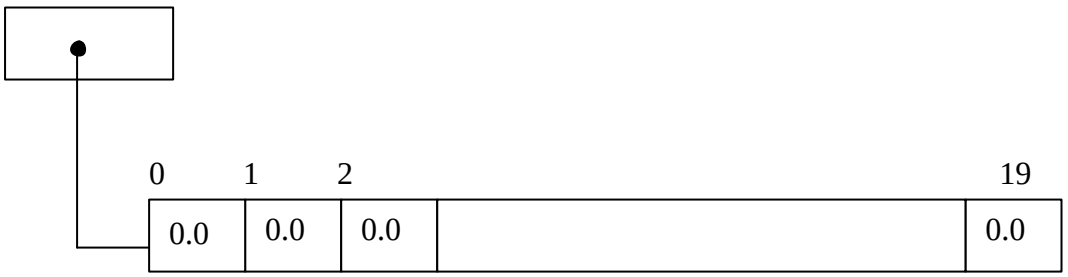
ARRAY DI OGGETTI VS. ARRAY DI TIPI BASE

L'istruzione:

```
float f[] = new float[20];
```

crea un oggetto di tipo array di float e alloca spazio per 20 float

f

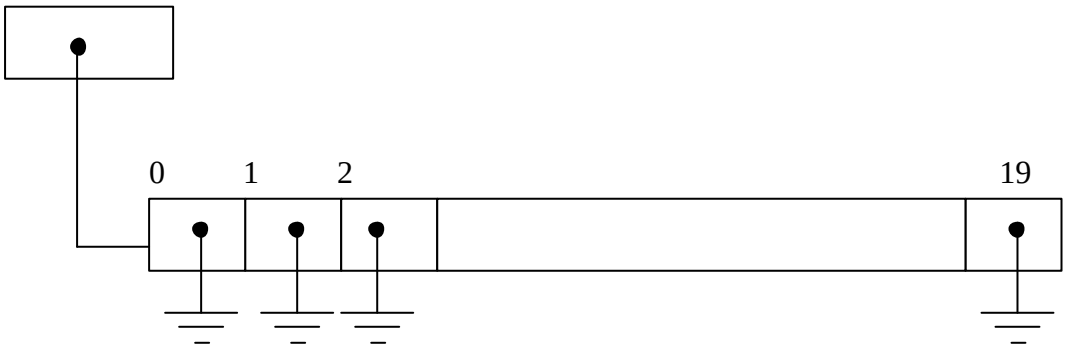


L'istruzione:

```
Person p[] = new Person[20];
```

crea un oggetto di tipo array di Person e alloca spazio per 20 riferimenti a oggetti di tipo Person

person



Non viene allocato spazio per gli oggetti veri e propri

Diverso dal C++, in cui potrei scrivere **`Person p[20];`** poiché l'array è allocato staticamente ed ogni cella di memoria contiene lo spazio per un oggetto di tipo persona. Se faccio allocazione dinamica con la new è come Java.

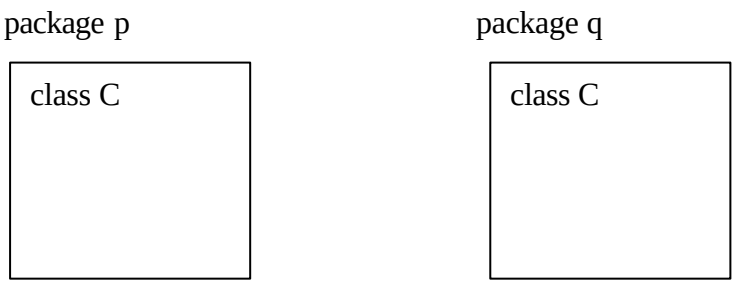
DEFINIZIONE DI UNA NUOVA CLASSE

Una nuova classe viene definita nel seguente modo:

```
class <nome classe> {
      <lista di definizioni di attributi e metodi>
}
```

Esempio:
class Automobile {...
}

CLASSI DEFINITE A LIVELLO PUBLIC O PACKAGE



Le due classi C sono distinte. Il package permette di restringere la visibilità della classe.

```
public class C
```

C è visibile a tutte le classi che hanno importato il package dove c'è C

```
class C
```

C è visibile solo dentro il package in cui è definita.

In un package ci deve essere almeno una classe public che serva da interfaccia con l'esterno.
Se non definiamo nessun package, le classi sono messe in un package di default e sono tutte tra di loro visibili.

DEFINIZIONE DI ATTRIBUTI

Gli attributi costituiscono lo "stato" degli oggetti appartenenti ad una classe
Gli attributi si definiscono usando la stessa sintassi della dichiarazione di variabili

Esempio

```
class Automobile {
      String colore, marca, modello;
      int cilindrata, numPorte;
      boolean accesa;
}
```

DEFINIZIONE DI METODI

I metodi definiscono il comportamento degli oggetti appartenenti ad una classe

Ogni metodo è definito come segue:

```
<tipo val. rit.> <nome.>([<dic. par. formali>]){
     <corpo>
     }
```

Il tipo void viene utilizzato per metodi che non ritornano alcun valore

Esempio:

```
class Automobile {
     String colore, marca, modello;
     int cilindrata, numPorte;
     boolean accesa;
     void accendi() {accesa=true;}
     boolean puoPartire() {return accesa;}
     void dipingi(String col) {colore=col;}
     void trasforma(String ma, String mo) {
          marca=ma;
          modello=mo;
     }
}
```

CLASSI E OGGETTI

Le **istanze** di una classe si chiamano **oggetti**

Ogni variabile il cui tipo sia una classe (o un interfaccia) contiene un **riferimento** ad un oggetto

Ad ogni variabile di tipo riferimento può essere assegnato il riferimento **null:** Automobile a=null;

Regola per il passaggio parametri:

- I parametri il cui tipo sia uno dei **tipi semplici** sono passati per **copia**
- I parametri il cui tipo sia un tipo **riferimento** (classi, interfacce e array) sono passati per **riferimento** (ovvero per copia del riferimento)

- Quindi, gli oggetti sono sempre passati per riferimento

ESEMPIO

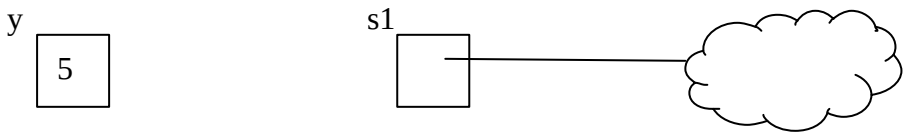
Abbiamo una classe che definisce un metodo

```
class NomeClasse{
      …
      m (int x, Stack s) {
            …
      }
      …
}
```

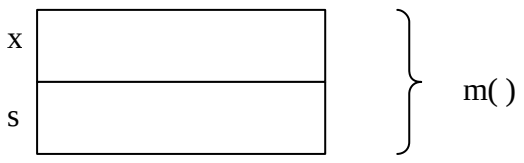
Invochiamo:

```
NomeClasse o = new NomeClasse();
int 5;
Stack s1 = new Stack(20);
o.m(y, s1);
```

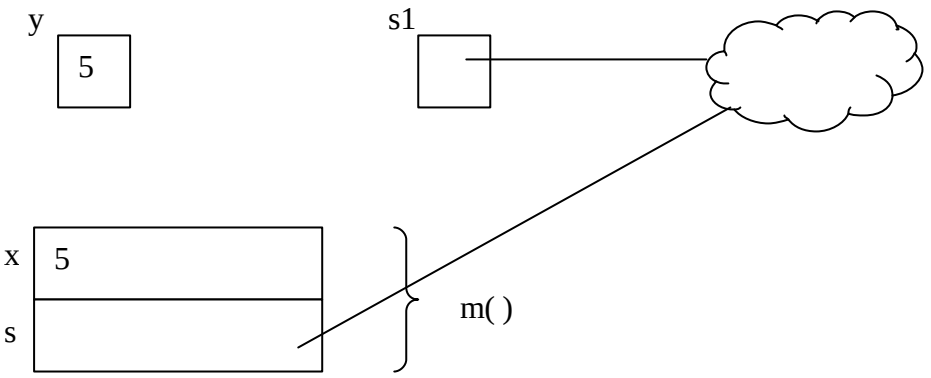
Vediamo cosa succede



Nell'AR di m, davo avere due celle, una per x ed una per s



Il 5 viene copiato nel parametro formale x, mentre per s copiamo il valore del puntatore.



s viene passato per riferimento.

ACCESSO AD ATTRIBUTI E METODI DI UN OGGETTO

L'accesso ad attributi e metodi di un oggetto per il quale si abbia un riferimento si effettua tramite la "notazione punto" (inglese: **dot-notation**)

Esempio:

```
Automobile a;
....
if(a!=null){
      a.accendi();
      a.dipingi("Blu");
}
```

ACCESSO AD ATTRIBUTI E METODI LOCALI

Nella definizione di un metodo ci si riferisce ad attributi e metodi dell'oggetto sul quale il metodo sia stato invocato direttamente (senza notazione punto)

Esempio

```
class Automobile {
      String colore;
      void dipingi(String col) {colore=col;}
      ...
}
```

VISIBILITÀ DEI NOMI

Regola generale:

le variabili dichiarate all'interno di un blocco mascherano le variabili **omonime** dichiarate nei blocchi più esterni

In particolare:

Le variabili locali ad un metodo (o i parametri formali) possono mascherare gli attributi della classe

Soluzione:

La pseudo-variabile **this** contiene un riferimento all'oggetto corrente e può essere utilizzata per aggirare eventuali mascheramenti

Esempio

```
class Automobile {
String colore;
.....
void dipingi(String colore) {
     String colore;
     colore = colore;          //variabile locale
     this.colore = colore;     //attributo
     }
}
```

OVERLOADING DI METODI

All'interno di una stessa classe possono esservi più metodi con lo stesso nome purché si distinguano per numero e/o tipo dei parametri

Attenzione: Il tipo del valore di ritorno non basta a distinguere due metodi

Esempio:

```
class C {

     int f() {...}

     int f(int x) {...}
          // corretto
     void f(int x) {...}
          // errato

}
```

CREAZIONE E DISTRUZIONE DEGLI OGGETTI

Nuovi oggetti sono costruiti usando l'operatore new

```
Automobile a=new Automobile();
```

La creazione di un oggetto comporta l'invocazione di un particolare metodo dell'oggetto chiamato costruttore.

Il **costruttore** svolge due operazioni fondamentali:
l'allocazione della memoria necessaria a contenere l'oggetto
l'inizializzazione dello spazio allocato

A differenza del C++, in Java non è necessario deallocare esplicitamente gli oggetti. Di ciò si occupa il **garbage collector**.

CLASSI E COSTRUTTORI

Nella definizione di una classe è possibile specificare uno o più (vedi overloading) costruttori

- Un costruttore è un metodo particolare che ha **lo stesso nome della classe**
- Per i costruttori non viene indicato il tipo del risultato

Se non si definisce nessun costruttore, il compilatore fornisce il **costruttore di default** (senza parametri), che svolge le seguenti funzioni:

- Alloca lo spazio per gli attributi di tipo primitivo
- Alloca lo spazio per i riferimenti agli attributi di tipo definito dall'utente
- Inizializza a **null** tutti i riferimenti

NOTA:
Se in una classe definiamo un costruttore con parametri, nelle classi figlie non posso utilizzare il costruttore di default.

COSTRUTTORI: ESEMPIO

```
class Automobile {
      String colore, marca, modello;
      int cilindrata, numPorte;
      boolean accesa;

Automobile() {
      colore=marca=modello=null;
      cilindrata=numPorte=0;
      }

Automobile(String colore, String marca, String modello) {
      this.colore=colore; this.marca=marca;
      this.modello=modello;
```

```
      cilindrata=numPorte=0;
      }

void accendi() {accesa=true;}
boolean puoPartire() {return accesa;}
void dipingi(String col) {this.colore=col;}
}
```

ANCORA SUI COSTRUTTORI

A volte è comodo fattorizzare il codice dei costruttori

È possibile invocare un costruttore dall'interno di un altro tramite la notazione:
`this(<elenco di parametri attuali>);`

Esempio:

```
class Persona {
      String nome;
      int eta;

      Persona(String nome) {this.nome=nome; eta=0;}

      Persona(String nome, int eta) {
            this(nome);
            this.eta=eta;
      }
}
```

Per evitare di riscrivere il codice.

`this(nome);` //richiama il costruttore che gestisce il nome (il costruttore associato all'oggetto corrente)

METODI E ATTRIBUTI DI CLASSE

Sintassi:

`static <definizione dell'attributo o metodo>`

Un attributo di classe è condiviso da tutti gli oggetti della classe

Si può accedere ad un attributo di classe senza bisogno di creare un oggetto tramite la notazione:

`<nome classe>.<nome attributo>`

Poiché sono statici, i metodi di classe può essere invocato senza bisogno di creare un oggetto tramite la notazione:

`<nome classe>.<nome metodo>(<par. attuali>)`

METODI E ATTRIBUTI DI CLASSE: VINCOLI

Un metodo static può accedere ai soli attributi e metodi static

Un metodo convenzionale può accedere liberamente a metodi ed attributi static

METODI E ATTRIBUTI DI CLASSE: ESEMPIO

```
class Shape {
      static Screen screen = new Screen();  // si noti l'inizializzazione
      static void setScreen(Screen s) {screen=s;}
      void show(Screen s) {setScreen(s); ...}
}
```

Lo schermo è static perché è uguale per tutte le figure che istanzio.

setScreen setta lo schermo (a livello di classe)

show permette di visualizzare la figura (è di istanza)

```
Shape.setScreen(new Screen());
```
// corretto, infatti setScreen è static

```
Shape.show();
```
/* errato, show è un metodo normale e non può essere chiamato su una classe. Dobbiamo prima creare un oggetto di tipo Shape e poi chiamare il metodo per l'oggetto */

```
Shape s1=new Shape(), s2=new Shape();
Screen s=new Screen();
s1.setScreen(s);
```
/* corretto, si possono chiamare metodi static su oggetti in questo punto s2.screen==s1.screen==s */

setScreen che è di classe, si può chiamare anche da istanza.

Volendo è anche possibile dichiarare dei blocchi static

```
static{
...
}
```

ATTRIBUTI COSTANTI

Spesso **static** è affiancato da **final** che consente di definire attributi costanti.

Lo si può fare tramite la notazione:

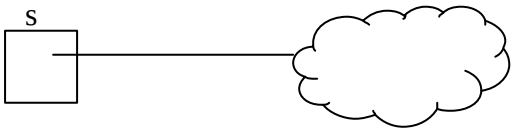
```
final <definizione di attributo> = <valore>
```

Esempio

```
class Automobile {
      int colore;

      final int BLU=0, GIALLO=1,...;

      void dipingi(int colore) {this.colore=colore;}
      }

...

Automobile a=new Automobile();

a.BLU=128;                          /* errato non si può assegnare un valore ad un
                                       attributo costante */

System.out.println("BLU="+a.BLU);    // corretto
```

Se invece di avere un tipo primitivo abbiamo un oggetto:

```
final Stack s = new Stack();
```



Non possiamo cambiare il riferimento ma possiamo cambiare lo stato.

```
s = new Stack();      //Errore, non posso cambiare il riferimento
s.pop();              //OK, posso cambiare lo stato
```

Possiamo anche fare:

```
final Stack s;
s = new Stack();      //ma da qui in poi non posso più fare assegnazioni ad s
```

In genere final è combinata con static.

```
static final int BLU=0, GIALLO=1,...;
```

Stiamo così definendo degli attributi final validi per tutta la classe.

LA GERARCHIA DI EREDITARIETÀ

Le classi di un sistema OO sono legate in una gerarchia di ereditarietà

Data una classe C, una sottoclasse di C si definisce tramite la notazione:

```
class K extends C {
      <definiz. di attr. e metodi>
}
```

La sottoclasse eredita tutti gli attributi ed i metodi della sopraclasse (è public di default)

Java supporta solo ereditarietà semplice

LA CLASSE OBJECT

In mancanza di una differente indicazione, una classe Java estende la classe Object
La classe Object fornisce alcuni metodi tra i quali vale la pena citare i seguenti:

- **`public String toString();`**

  Ha come output una stringa che è la rappresentazione dell'oggetto. L'implementazione di default stampa il nome della classe e l'identificatore dell'oggetto. Possiamo ridefinirla come vogliamo. Tipicamente è usata per fare debugging.

- **`protected void finalize();`**

  Consente di fare alcune operazioni prima che l'oggetto venga rimosso. E' simile al distruttore C++ anche se qui non abbiamo modo di richiamarlo esplicitamente.

- **`public boolean equals(Object);`**

  Consente di confrontare due oggetti.

  ```
  int x = 2;
  int y = 3;
  String s1 = "Tizio";
  String s2 = "Tizio";

  x = = y          //ritorna false
  s1 = = s2        //ritorna false, perché i riferimenti non sono gli stessi
  ```

  equals verifica il contenuto degli oggetti

  `s1.equals(s2)` //ritorna true o false a seconda del codice associato ad s1 o s2.

  Per la classe String, il metodo è ridefinito in modo da tornare true se le stringhe sono uguali.

OVERRIDING

Una sottoclasse può aggiungere nuovi attributi e metodi ma anche ridefinire i metodi delle sue sopraclassi. Valgono le sesse regole del C++, ma in Java ho **sempre dispatching dinamico**.

Esempio:

```
class AutomobileElettrica extends Automobile {

      boolean batterieCariche;

      void ricarica() {batterieCariche=true;}

      void accendi() {

            if(batterieCariche)
                  accesa=true;
            else
                  accesa=false;
      }
}
```

LA PSEUDO VARIABILE SUPER

All'interno di un metodo che ridefinisce un metodo della sopraclasse diretta ci si può riferire al metodo che si sta ridefinendo tramite la notazione:

```
super.<nome metodo>(<lista par. attuali>)
```

Esempio:

```
class AutomobileElettrica extends Automobile {
      ...
      void accendi() {

            if(batterieCariche)
                  super.accendi();
            else
                  System.out.println("Batterie scariche");
      }
}
```

Accendi era definita nella super classe.

Posso usarlo in due modi:

- `super.metodo();`
- `super();` che è il costruttore della superclasse (**necessario perché i costruttori non vengono ereditati**)

VINCOLO: `super()` deve per forza essere nella prima linea di un eventuale costruttore.

EREDITARIETÀ E COSTRUTTORI: RIASSUNTO

I costruttori non vengono ereditati. All'interno di un costruttore è possibile richiamare il costruttore della sopraclasse tramite la notazione:

```
super(<lista di par. attuali>)
```

posta come prima istruzione del costruttore. Se il programmatore non chiama esplicitamente un costruttore della sopraclasse, il compilatore inserisce automaticamente il codice che invoca il costruttore di default della sopraclasse

INFORMATION HIDING IN JAVA

Attributi e metodi di una classe possono essere:

- **public**

  sono visibili a tutti, vengono ereditati

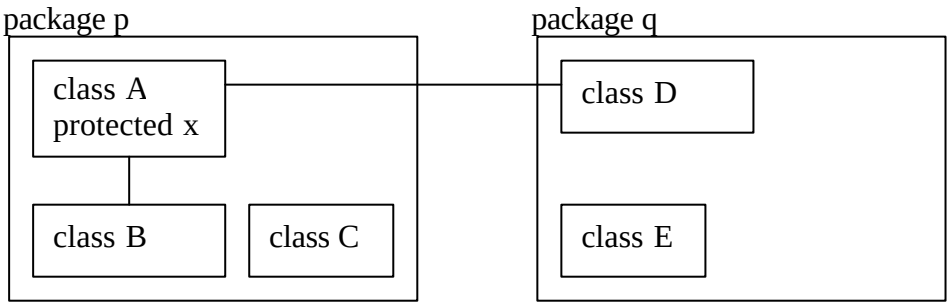
- **package**

  come identificatore non mettiamo niente. Visibile a tutte le classi definite nello stesso package della classe

- **protected**

  sono visibili alle sottoclassi ed alle classi dello stesso package, vengono ereditati

  Esempio:



  B può usare x (eredita ed è nello stesso package)
  D può usare x (eredita)
  C può usare x (è nello stesso package)

- **private**

  sono visibili solo all'interno della stessa classe, non sono visibili nelle sottoclassi

CLASSI E METODI ASTRATTI

Un metodo astratto è un metodo per il quale non viene specificata alcuna implementazione.

Una classe è astratta se contiene almeno un metodo astratto.

Non è possibile creare istanze di una classe astratta.

Le classi astratte sono molto utili per introdurre della astrazioni di alto livello.

In C++ si faceva mettendo "virtual … = 0". In Java con **abstract**.

CLASSI E METODI ASTRATTI: ESEMPIO

```
abstract class Shape {
      static Screen screen;
      Shape(Screen s) {screen=s;}
      abstract void show();
}
class Circle extends Shape {
      void show() {
      ...
      }
}
...
```

E' più leggibile del C++

Non posso creare istanze di classi astratte

```
Shape s = new Shape();          // errato
```

Lo posso fare invece per Circle.

```
Circle c = new Circle();        // corretto
```

CLASSI E METODI FINAL

Sono un metodo per vincolare l'ereditarietà.

Se vogliamo impedire che sia possibile creare sottoclassi di una certa classe la definiremo final

Esempio:

```
final class C {...}

class C1 extends C          // errato
```

Similmente, se vogliamo impedire l'overriding di un metodo dobbiamo definirlo final

Esempio:

```
class C { final void f() {...} }

class C1 extends C {        // corretto

     void f() {...}         // errato
}
```

EREDITARIETÀ ED ARRAY

Se X è una sopraclasse di Y allora l'array X[] è "sopra-array" dell'array Y[]. Lo stesso vale per gli array multidimensionali.

Tale scelta non è type safe

Esempio:

Consideriamo una funzione f() che accetta come parametro un oggetto di tipo X[]

```
void f(X[] ax) {

     ax[0] = new X();
     }
...
```

Il frammento di codice assegna al primo elemento un oggetto di tipo X creato al volo.

Se abbiamo un invocazione del tipo:
`f(new X[10]);`
Non ci sono problemi, perché scatta il polimorfismo.

Se invece consideriamo:

```
f(new Y[10]);
```

Passando come parametro un sottotipo (posso farlo per le regole del polimorfismo)

Quando facciamo:

```
ax[0] = new X();
```

Ha come tipo dinamico Y e come tipo statico X

Vi assegno un oggetto il cui tipo statico e dinamico sono X

Questo porterebbe ad un assegnamento del tipo Y = X che non è type safe.

INTERFACCE

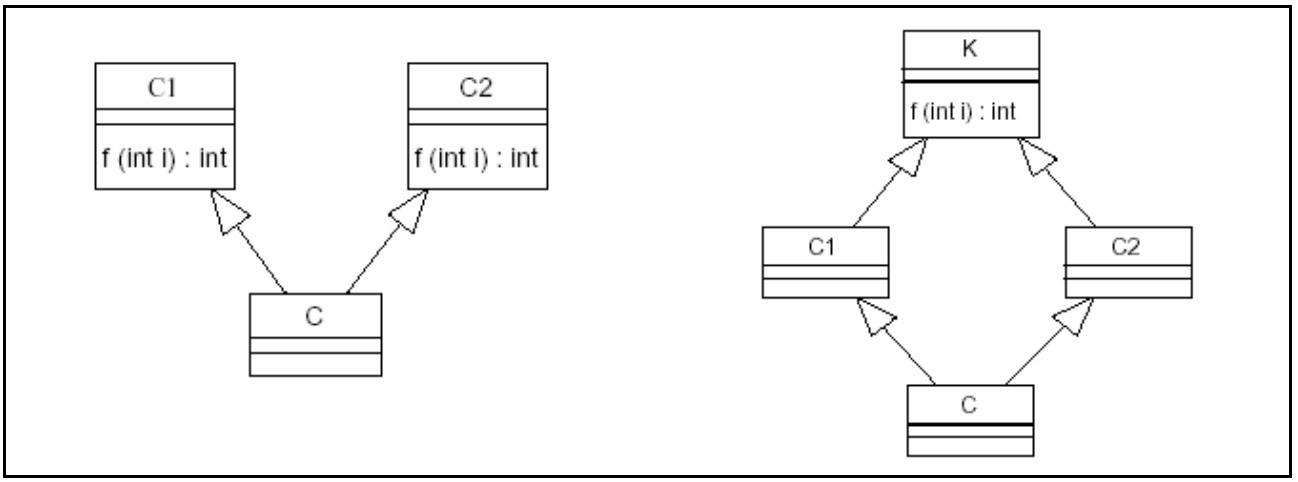
I LIMITI DELL'EREDITARIETÀ SEMPLICE

L'ereditarietà semplice non permette la descrizione di numerose situazioni reali

> ***Esempio:***
> *Supponiamo di avere una classe Giocattolo ed una classe Automobile. In assenza di ereditarietà multipla non posso definire la classe AutomobileGiocattolo*

I PROBLEMI DELL'EREDITARIETÀ MULTIPLA

In presenza di ereditarietà multipla è possibile ereditare due o più metodi con la stessa "signature" da più sopraclassi. Ciò determina un conflitto tra implementazioni diverse



LA SOLUZIONE DI JAVA: LE INTERFACCE

Distinguere tra una gerarchia di ereditarietà (semplice) ed una gerarchia di specializzazione (di tipi) (multipla) introducendo il costrutto delle interfacce

In tal modo è possibile distinguere tra l'uso dell'ereditarietà al fine di riutilizzare il codice e l'uso al fine di descrivere una gerarchia di tipi

Una **interfaccia** è una classe priva di attributi non costanti ed i cui metodi sono tutti pubblici ed astratti

Sintassi:

```
interface <nome> {
     <lista di definizione di metodi privi di corpo>
}
```

INTERFACCE ED EREDITARIETÀ

Una interfaccia può ereditare da una o più interfacce

Sintassi:

```
interface <nome> extends <nome1>,..,<nomen> {...}
```

La gerarchia di ereditarietà tra interfacce definisce una gerarchia di tipi

LA GERARCHIA DI IMPLEMENTAZIONE

Una classe può implementare una o più interfacce

se la classe non è astratta deve fornire una implementazione per tutti i metodi presenti nelle interfacce che implementa

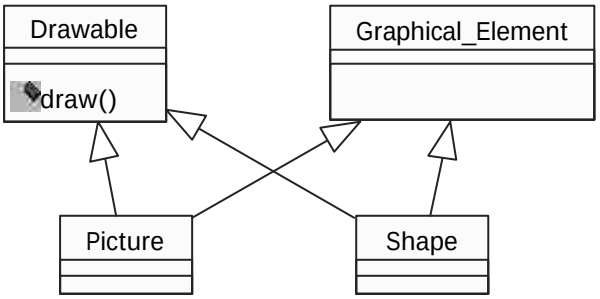
altrimenti la classe è astratta

Sintassi:

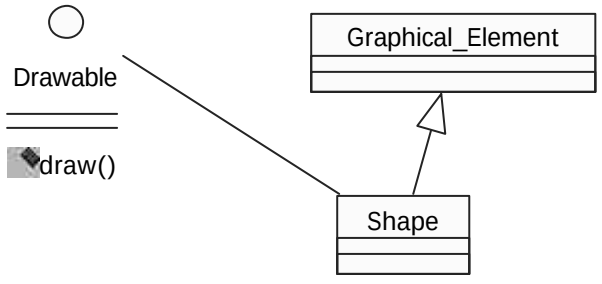
```
class <nome> implements <nome1>,..,<nomen> {...}
```

ESEMPIO

Possiamo volere una funzione di disegno che valga indipendentemente dalle figure geometriche. Definiamo un nuovo tipo **Drawable**.



Così da poter usare draw() sia per Picture che per Shape. In Java non si può fare.
Si usano le interfacce.



Nota: Shape deve implementare tutti i metodi dell'interfaccia drawable.

POLIMORFISMO

Polimorfismo è la capacità per un elemento sintattico di riferirsi a elementi di diverso tipo
In Java una variabile di un tipo riferimento T può riferirsi ad un qualsiasi oggetto il cui tipo sia T o un sottotipo di T
Similmente un parametro formale di un tipo riferimento T può riferirsi a parametri attuali il cui tipo sia T o un sottotipo di T

POLIMORFISMO: ESEMPIO

```
class Automobile {...}
class AutomobileElettrica extends Automobile {...}
class Parcheggio {
      private Automobile buf[];
      private int nAuto;
      public Parcheggio(int dim) {buf=new Automobile[dim];}
      public void aggiungi(Automobile a) {buf[nAuto++]=a;}
}
...
AutomobileElettrica b=new AutomobileElettrica();
Automobile a=new Automobile();
Parcheggio p=new Parcheggio(100);
p.aggiungi(a);
p.aggiungi(b);
```

POLIMORFISMO ED INTERFACCE

Una interfaccia può essere utilizzata come tipo di una variabile

Una siffatta variabile potrà riferirsi ad un qualsiasi oggetto che implementi l'interfaccia (polimorfismo)

Esempio:

```
interface OggettoCheSpara {
      spara();
}
class Pistola implements OggettoCheSpara {...}
OggettoCheSpara o=new Pistola();
o.spara();
```

POLIMORFISMO: TIPO STATICO E TIPO DINAMICO

In presenza di polimorfismo distinguiamo tra il tipo statico ed il tipo dinamico di una variabile (o di un parametro formale)

La regola del polimorfismo prima enunciata obbliga il tipo dinamico ad essere un sottotipo (proprio o meno) del tipo statico

POLIMORFISMO E BINDING DINAMICO

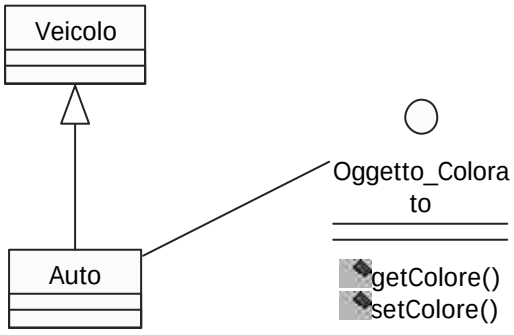
In Java, a fronte della invocazione **x.f(x1,...,xn)**, l'implementazione scelta per il metodo f dipende dal tipo dinamico di x e non dal suo tipo statico

ESEMPIO

```
class Persona {
private String nome;
public Persona(String nome) {this.nome=nome;}
public void chiSei() {
      System.out.println("Ciao, io sono "+nome);
      }
}
class Uomo extends Persona {
public Uomo(String nome) {super(nome);}
public void chiSei()
      {super.chiSei(); System.out.println("sono un maschio");}
}
...
Persona p=new Uomo("Giovanni"); p.chiSei();
```

OUTPUT: Ciao, io sono Giovanni sono un maschio

ESEMPIO FINALE E CONCLUSIONI



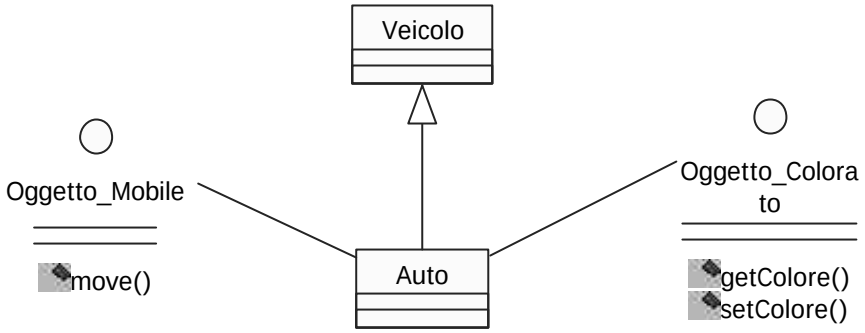
Voglio vedere l'auto sia come un veicolo che come un oggetto colorato.

Eredito dalla classe che rappresenta meglio l'oggetto, ed estendo dall'altra di cui mi interessano i metodi.

```
interface Oggetto_Colorato {

      Colore getColore();
      void setColore(Colore c);
}

class Auto extends Veicolo implements Oggetto_Colorato{

      public Colore getColore(){...};
      public void setColore(Colore c){...};
}
```

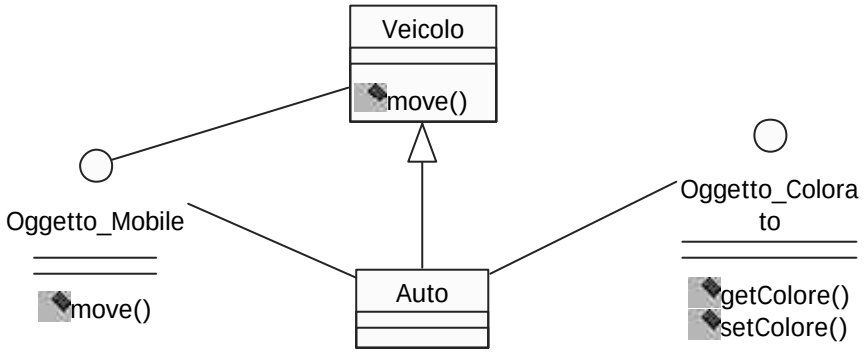
In Auto **posso** ridefinire alcuni dei metodi di Veicolo (non è obbligatorio, se non lo faccio eredito il metodo originale). Invece **devo** fornire un implementazione dei metodi di Oggetto_Colorato specificando l'identificatore d'accesso. Se la classe la dichiaro come abstract, posso anche non implementare il codice dei metodi anche se devo comunque dichiararli.

Posso implementare un interfaccia sia in maniera **diretta** che **indiretta**.




```
class Auto extends Veicolo implements Oggetto_Colorato, Oggetto_Mobile{
```

Se anche la classe Veicolo implementa Oggetto_Mobile, posso definire move() in Veicolo e non c'è bisogno di definirlo in Auto.



Posso fare cose tipo:

```
Oggetto_Colorato oc = new Auto();
```

perché valgono le regole del polimorfismo

L'interfaccia è un modo per operare sullo stato di un oggetto attraverso "pannelli di controllo" diversi.

Lo stesso Oggetto concreto lo posso trattare in modi diversi a seconda di come uso il polimorfismo.

Differenza con le classi astratte:

Le classi astratte sono una definizione parziale della classe, alcuni metodi sono definiti. Voglio vincolare una parte del codice ma non tutto (sono utili per definire le strutture interne dei tipi).

PROBLEMA

In interfacce diverse posso definire metodi con lo stesso nome, una classe che implementa due interfacce che hanno un metodo con lo stesso nome farà una sola implementazione del metodo e può accadere che quello che implemento dei due non sia quello che mi serve.

TAGGING INTERFACES

Sono usate senza dichiarare nessun metodo.

Ad esempio in java.io.*, è dichiarata un'interfaccia **serializable** che è vuota.

To serialize in inglese vuol dire "appiattire" (ad esempio per salvare in memoria di massa devo poter appiattire la struttura di un oggetto).

Un'interfaccia di questo tipo serve solo ad "etichettare" degli oggetti. Il run-time support sa che può appiattire una classe che implementa serializable.

CONSTANTS

Nelle interfacce oltre ai metodi posso dichiarare anche delle costanti che vengono semplicemente ereditate (e se le definiamo static sono a livello di classe).

E' spesso usato per mettere tutte le costanti in un unico file.

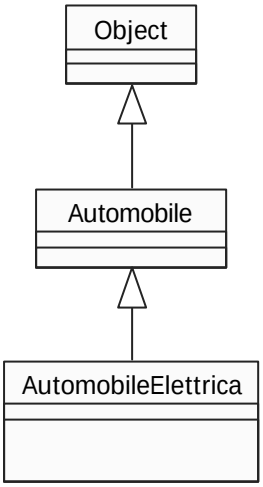
```
interface Constants {

      static final int UNO = 1;
      static finao int DUE = 2;
      ...
}
```

e poi le posso usare semplicemente dicendo che la classe implementa Constants.

CONVERSIONI FORZATE TRA TIPI RIFERIMENTO: CASTING

È possibile forzare esplicitamente la conversione da un tipo riferimento T ad un sottotipo T1 purché il tipo dinamico dell'espressione che convertiamo sia un sottotipo di T1

Esempio:




```
Object o = new AutomobileElettrica();
```

ad o, viene assegnato un tipo AutomobileElettrica

```
Automobile a = o;                  // errato, Object non è un sottotipo di Automobile

Automobile a = (Automobile) o;     // corretto (casting)
```

Viene generata una conversione di tipo, che però è passibile di errori a run-time perché se quando faccio l'assegnamento, l'oggetto non è un AutomobileElettrica, ho un errore dovuto al fatto che ho forzato la conversione.

Posso evitare questo tipo di errori mediante:

```
if (o instanceof Automobile) {
    Automobile a = (Automobile) o;
    ...
}
```

interroga il tipo dinamico

Il cast è fondamentale per implementare i contenitori generici.

Esempio:

```
Vector v;
```

V[i]; non posso usarlo (è per gli array)

si fa con:

```
v.elementAt(i);
```

la cui signature è: Object elementAt(int i)

e ritorna un Object.

Se ho un Vector i cui elementi sono di tipo Persona, non posso scrivere:

```
v.elementAt(i).getName();
```

poiché v.elementAt(i) è un Object e non un oggetto di tipoPersona.

Devo fare il casting:

```
Persono p = (Persona) v.elementAt(i);
p.getName();
```

PACKAGE ED INFORMATION HIDING

Package e visibilità degli attributi e dei metodi

- Attributi e metodi di una classe C per i quali non sia dichiarato alcun tipo di visibilità sono visibili solo all'interno di classi che appartengano allo stesso package di C
- Tale costrutto gioca un ruolo analogo alla dichiarazione di friendship in C++

Package ed importazione di classi

- Un package contiene un insieme di classi pubbliche ed un insieme di classi private
- Solo le classi pubbliche si possono importare in altri package

Package e compilation unit

- Ogni compilation unit contiene classi appartenenti allo stesso package
- Una compilation unit contiene una sola classe pubblica (per default la prima) ed eventualmente altre classi private

IL PACKAGE JAVA.LANG

Il package java.lang contiene classi di uso molto frequente (String, Object, ecc.)

Non è necessario importare le classi appartenenti al package java.lang prima di utilizzarle

GESTIONE DEGLI ERRORI

Spesso, si tende a scrivere programmi assumendo che tutto vada a buon fine In pratica, vi sono spesso situazioni impreviste che devono essere gestiti dal programma

La mancata gestione di tali situazioni solitamente determina la terminazione improvvisa del programma.

Le situazioni anomale possono essere *dipendenti dall'ambiente* (esempi: disco pieno, connessione di rete non disponibile) oppure *dal programma* (ad esempio una pop() su una pila vuota)

Tradizionalmente, gli errori vengono gestiti:

- Terminando il programma nel metodo in cui si verifica l'errore
  - Spesso, una scelta drastica …
  - … che peraltro spetta al chiamante (che sa come sta usando la routine), e non al chiamato
- Usando valori di ritorno convenzionali per segnalare errori al chiamante
  - Causano confusione se gli errori sono dello stesso tipo del valore di ritorno
  - In ogni caso impossibile per i costruttori
  - Non forniscono informazione riguardo alla natura dell'errore
- Uso di una funzione di gestione degli errori
  - Centralizza la gestione degli errori, che invece spetterebbe al chiamante
  - Diminuisce la leggibilità del programma

LE ECCEZIONI IN JAVA

Un'eccezione è un evento imprevisto, anomalo e indesiderato che si verifica durante l'esecuzione di un programma. I linguaggi di programmazione moderni dedicano costrutti apposito alla gestione delle eccezioni, che permettono di superare i limiti della gestione tradizionale degli errori.

La gestione delle eccezioni presenta i seguenti vantaggi:

- Separare la gestione degli errori dal codice applicativo
- Consentire una propagazione controllata degli errori
- Raggruppare o differenziare gli errori

Una eccezione può essere catturata e gestita attraverso il costrutto:

```
try {…}
catch(ClasseEccezione e) {…}
```

Più clausole catch possono seguire lo stesso blocco try

ESEMPIO

```
leggiFile {
      apri file;
      determina dimensione file;
      alloca memoria;
      trasferisci file in memoria;
      chiudi file;
}
```

E' senza le gestione degli errori.

Vediamo un modo per gestire gli errori

```
tipoCodiceErr leggiFile {
      inizializza codiceErr = 0;
      apri file;
      if (fileAperto) {
            determina dimensione  file;
            if (ottenutaLunghezza)     {
                  alloca memoria;
                  if (ottenutaMemoria) {
                        trasferisci file in   memoria;
                        if (trasferimentoFallito)  {
                              codiceErr = -1;
                        }
                  } else {
                        codiceErr = -2;
                  }
            } else {
                  codiceErr = -3;
            }
            chiudi file;
            if (fileNonChiuso && codiceErr == 0) {
                  codiceErr = -4;
            } else {
            codiceErr = codiceErr and -4;
            }
      } else {
            codiceErr = -6;
      }
      return codiceErr;
}
```

Devo cambiare la signature della funzione, e si complica il codice che non è più molto leggibile.
Non si riesce a distinguere il codice dell'applicazione da quello per la gestione degli errori.

Vediamo adesso con la gestione delle eccezioni:

```
leggiFile {
     try {
          apri file;
          determina dimensione file;
          alloca memoria;
          trasferisci file in memoria;
          chiudi file;
     } catch (aperturaFileFallita) {
          ...
     } catch (determinaDimFallito) {
          ...
     } catch (allocazioneMemFallita) {
          ...
     } catch (letturaFallita) {
          ...
     } catch (chiusuraFileFallita) {
          ...
     }
}
```

Racchiudo il codice applicativo in un blocco **try**. Se qualcosa va storto è eseguito il blocco **catch** relativo all'errore.

ESEMPIO: PROPAGAZIONE DEGLI ERRORI

```
metodo1 {
     chiama metodo2;
}
metodo2 {
     chiama metodo3;
}
metodo3 {
     chiama leggiFile;
}
```

Se va storto leggiFile non ha senso che sia leggiFile a decidere cosa fare, devo propagare all'indietro l'informazione relativa all'errore.

Con la gestione tradizionale i metodi devono verificare se c'è stato un errore per dare al chiamante la possibilità di gestirlo.

```
metodo1 {
     tipoCodiceErr errore;
     errore = chiama metodo2;
     if (errore)
          gestisciErrore;
     else procedi;
} tipoCodiceErr metodo2 {
     tipoCodiceErr errore;
     errore = chiama metodo3;
     if (errore)
          gestisciErrore;
```

```
      else procedi;
} tipoCodiceErr metodo3 {
      tipoCodiceErr errore;
      errore = chiama leggiFile;
      if (errore)
            return errore;
      else procedi;
}
```

Con la gestione delle eccezioni:

```
metodo1 {
      try {                        //cattura le eccezioni propagate
            chiama metodo2;
      } catch(exception) {
            gestisciErrore;
      }

metodo2 throws exception {         //fa solo da passamano
      chiama metodo3;
}
metodo3 throws exception {
      chiama leggiFile;
}
```

Dichiaro nei metodi cosa posso propagare e poi è il sistema a gestire tutto.

THROW

Un'eccezione è rappresentata da un oggetto (contiene informazioni circa la causa dell'eccezione)

Un'eccezione viene sollevata mediante il costrutto **throw**, seguito dall'oggetto che rappresenta l'eccezione.

Un metodo deve obbligatoriamente dichiarare se solleva eccezioni. (ciò viene fatto nell'interfaccia del metodo, mediante il costrutto **throws)**

ESEMPIO

```
public Object pop() throws EmptyStackException {

      if (!isEmpty())

            return stack[--ptr];

      else throw new EmptyStackException();
}
```

Un metodo deve dichiarare tutte le eccezioni che può sollevare.

COME DELIMITARE E GESTIRE L'ECCEZIONE

Il segmento di codice che può sollevare l'eccezione viene delimitato da un blocco **try**

Esso è sempre seguito da uno o più blocchi **catch** (gestori di eccezioni), che contengono il codice di gestione dell'eccezione.

- I blocchi **catch** vengono chiamati *exception handlers* (gestori di eccezione)
- Handlers diversi sono necessari se il codice in questione può sollevare eccezioni diverse

ESEMPIO

```
...

Stack s = new Stack

try {
     Object o = s.pop();
} catch(EmptyStackException e) {
     System.out.println("Stack vuoto!");
     System.exit(1);
}
```

LA PROPAGAZIONE DEGLI ERRORI

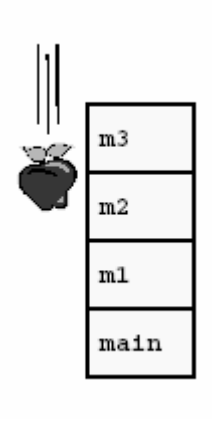
L'esecuzione di *un'istruzione throw solleva un'eccezione,* e dà inizio al seguente processo:

- viene terminata l'esecuzione del blocco di codice che contiene l'istruzione throw
- l'eccezione argomento della throw viene propagata lungo la catena dinamica delle chiamate, verificando di volta in volta se esiste nel chiamante un blocco try/catch con un gestore appropriato per l'eccezione:
  - se sì, l'esecuzione riprende dal codice contenuto nella catch così determinata (si dice che il gestore ha "catturato" l'eccezione);
  - altrimenti, la propagazione continua al chiamante del chiamante
- Se non viene trovato nessun blocco try/catch compatibile, l'esecuzione termina

ESEMPIO

```
public class Test {
     public static void main(String[] args) {
          try {
               m1();
          } catch (TestException e) { … }
     }
```

```
    void m1() throws TestException {
        m2();
    }

    void m2() throws TestException {
        m3();
    }

    void m3() throws TestException {

        … throw new TestException();
    }
}
```



Dichiaro m1, m2 ed m3 che propagano TestException. L'unico che può sollevarla è m3 (devo definire a parte la classe TestException)

Il main invoca m1 che invoca m2 che invoca m3. L'eccezione man mano che scende o viene catturata oppure passa laa'AR successivo (man mano gli AR vengono eliminati) finché si trova il blocco try/catch compatibile con l'eccezione sollevata.

L'errore viene propagato da m3 fino (eventualmente) al main

GESTIRE L'ERRORE O PROPAGARLO?

- Java richiede che ogni metodo debba:
  - gestire le eccezioni che possono essere sollevate all'interno del proprio corpo, definendo un blocco try seguito da uno o più blocchi catch; oppure,
  - propagare tali eccezioni, dichiarando ciò mediante una clausola throws nell'interfaccia del metodo
- Dal secondo requisito, è evidente come la dichiarazione delle eccezioni sia parte integrante dell'interfaccia di ogni metodo
  - Ciò è necessario affinché gli utilizzatori del metodo siano consapevoli delle eccezioni che possono essere sollevate al suo interno

LA CLAUSOLA FINALLY

Opzionalmente, alla fine di un blocco try/catch può apparire una clausola finally

Essa viene eseguita comunque sia che si verifichi un'eccezione (e quindi venga eseguito uno dei blocchi catch), oppure no (e quindi venga eseguito tutto il codice nel blocco try)

Serve a evitare duplicazione di codice e a garantire che alcune operazioni, che non verrebbero eseguite in caso di un'eccezione per cui non è presente il relativo gestore, vengano comunque effettuate. Tipicamente usata per operazioni di "pulizia" finale durante I/O

Esempio:

```
try {
      FileInputStream f = new FileInputStream(filename);
      … usa f …
} catch(IOException ioe) {
      …
      gestisci l'errore di I/O
      …
} finally { f.close(); }
```

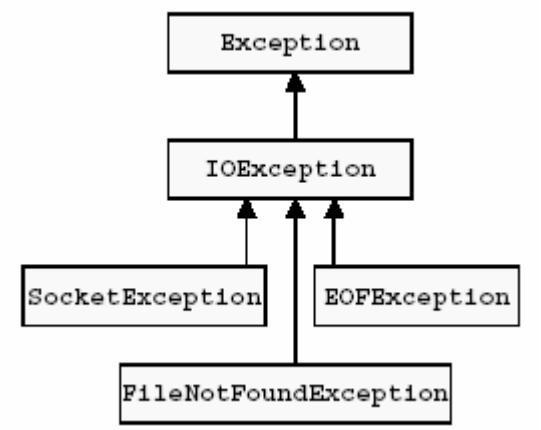
Apro un file e lo uso. Posso generare una eccezione IO oppure no. Indipendente dal fatto che sia andato a buon fine voglio chiudere il file. Senza finally dovrei metterlo in due punti duplicando il codice.

ECCEZIONI ED EREDITARIETA'

Spesso, le eccezioni possono essere raggruppate in categorie

Es., eccezioni relative all'I/O, al display, ecc.

Poiché le eccezioni sono rappresentate come oggetti, l'ereditarietà fornisce un meccanismo naturale per rappresentare tale raggruppamento Inoltre, l'ereditarietà consente di creare di eccezioni definite dal programmatore, come sottoclassi di **Exception**



Serve anche come documentazione, in più posso decidere di gestire un'eccezione al livello di dettaglio che desidero.

SPECIFICARE PIU' GESTORI

Un blocco try/catch può contenere più di un gestore di eccezioni (cioè più di un blocco catch)

L'ereditarietà applicata alle eccezioni fa sì che sia possibile che vi siano più gestori compatibili con l'eccezione sollevata

Esempio:

```
try {
     FileOutputStream f = new FileOutputStream("f.dat");
} catch (FileNotFoundException fnfe){ … }
} catch (IOException ioe){ … }
} catch (SecurityException se){ … }
} catch (Exception e){ … }
```

Viene eseguito il primo blocco di catch compatibile con l'eccezione sollevata (le sottoclassi vanno dichiarate prima delle rispettive superclassi (altrimenti non verrebbero mai selezionate)

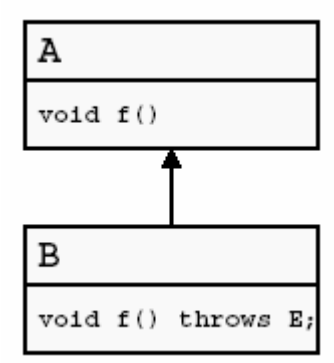
Nell'esempio se non trovo il file ho tre gestori compatibili con l'eccezione: FileNotFoundException IOException, Exception) ma viene sollevata FileNotFoundException che è la prima ad essere trovata.

ECCEZIONI E POLIMORFISMO

Un metodo m definito in una sottoclasse B che ridefinisce un metodo m della superclasse A può sollevare soltanto le eccezioni E1, E2, …, En dichiarate nella firma di m in C, oppure sottoclassi di E1, E2, …, En

- **La specifica delle eccezioni può solo "restringersi"** procedendo verso le sottoclassi, ma non "allargarsi" (al contrario delle normali regole di ridefinizione)
- Questo vincolo è necessario, per preservare la semantica del polimorfismo
- Non vale per i costruttori

Infatti se avessi:



potrei chiamare:

```
A a = new B();          //il tipo dinamico di a è B
a.f();                  //legale perché f() è dichiarata nel tipo statico e soprascritto in B
```

Però a.f() potrebbe sollevare un eccezione anche se non dichiarata in A.

THROWABLE E SOTTOCLASSI

Gli oggetti che possono essere "sollevati" da una throw devono avere come tipo una sottoclasse di **Throwable**

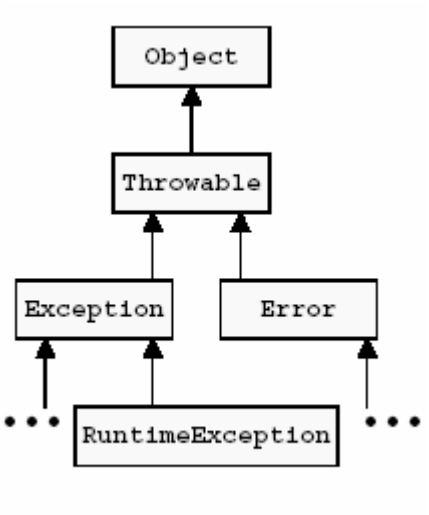
Tra tali sottoclassi, in Java si distingue fra Error ed Exception:

- Le sottoclassi di **Error** rappresentano errori "gravi", che non possono essere gestiti dai programmi; è inusuale che un programma sollevi un Error (Es. VirtualMachineError, LinkageError)

- Le sottoclassi di **Exception** rappresentano le comuni eccezioni, di cui abbiamo trattato finora (Es. IOException)

Per queste non c'è il vincolo "catch or specify"

RUNTIME EXCEPTION



Fra le sottoclassi di Exception, le sottoclassi di RuntimeException definiscono una famiglia di eccezioni cosiddette runtime, con regole proprie

Tali classi rappresentano eccezioni che avvengono nell'interprete Java (virtual machine) ma che non sono così gravi da essere considerati Error

Un esempio classico è NullPointerException, che si verifica quando un metodo cerca di accedere a un membro di un oggetto attraverso un riferimento il cui valore è null(es. obj = null; obj.m();)

Tali eccezioni si possono verificare praticamente ovunque: forzare il requisito di gestire oppure dichiarare tali eccezioni porterebbe alla stesura di codice illeggibile

Per questo motivo, non è necessario gestire o dichiarare questo tipo di eccezioni (anche se è possibile farlo, quando necessario e/o ragionevole)

Non è consigliabile definire tutte le eccezioni come RuntimeException!

FORNIRE INFORMAZIONI CIRCA L'ECCEZIONE

La classe che descrive un'eccezione può possedere attributi e metodi che vengono usati per fornire informazioni aggiuntive al chiamante (poiché può dare informazione strutturata)

ESEMPIO:

```
public class DataIllegaleException extends Exception {
      int   giorno,
            mese,
            anno;
      DataIllegale(int g, int m, int a) {
            giorno = g;
            mese = m;
            anno = a;
      }
}

public class Data {
      private int     giorno,
                      mese,
                      anno;
      private boolean corretta(int g,int m,int a) { ... }
      public Data(int g, int m, int a) throws DataIllegale {
            if(!corretta(g,m,a))
                  throw new DataIllegale(g,m,a);
                  giorno = g;
                  mese = m;
                  anno = a;
      }
}
```

Ho una classe Data che ha un test che vede se la data è corretta altrimenti viene sollevata l'eccezione chiamando un costruttore con i parametri opportuni.

ECCEZIONI E DEBUGGING

Se ho un errore il programma si pianta e stampa a video il tipo di eccezione che si è verificata a dove.

La gestione delle eccezioni è utile in fase di debugging, in quanto dà informazioni sul verificarsi di un'anomalia

In seguito al verificarsi di un'eccezione viene stampata una stack trace, che contiene l'indicazione delle procedure attive nella catena dinamica, e dei numeri di linea in cui l'eccezione è stata propagata.

ESEMPIO

```
public class Test {
      public static void main(String[] args) {
            try {
                  m1();
            } catch (TestException e) { … }
      }

      void m1() throws TestException {
            m2();
      }

      void m2() throws TestException {
            m3();
      }

      void m3() throws TestException {

            … throw new TestException();
      }
}
```

Potrebbe generare il seguente errore:

```
Exception in thread "main" java.lang.NullPointerException
            at Test.m3(TestException.java:8)
            at Test.m2(TestException.java:7)
            at Test.m1(TestException.java:6)
            at Test.main(TestException.java:3)
```

con la sintassi **classe.metodo**

Così so qual è il punto in cui è stato generato l'errore.

ERRORE FATALE:
per questo motivo è consigliabile non inserire mai un'istruzione di questo tipo:

```
try { …
} catch (Exception e){}
```

senza niente

in quanto essa di fatto "spegne" il meccanismo delle eccezioni e non dà modo al programmatore di accorgersi di un malfunzionamento.

Tipicamente come minimo in fase di debugging se non voglio gestire l'eccezione devo mettere:

```
try { …
} catch (Exception e){
      e.printStackTrace();          //stampa un messaggio simile a quello sopra
      system.exit(1);
}
```

SUGGERIMENTI PRATICI

- Quando si definiscono le classi di un programma, è spesso utile lasciare la gestione delle eccezioni a chi usa le classi, anziché gestirle internamente
  - Infatti, è spesso chi usa la classe che deve decidere come reagire a un evento anomalo, nel contesto specifico dell'applicazione
- Quando possibile, conviene sempre riusare un'eccezione già fornita dal linguaggio, se questa esiste
  - Onde evitare la proliferazione di eccezioni, e per aumentare la comprensibilità e leggibilità del programma

ESEMPIO FINALE

Assumiamo che il metodo a lato sia all'interno del codice che definisce uno stack, rappresentato internamente da un oggetto di classe Vector, e la cui dimensione è nota attraverso il campo size.

```
public void scriviStack() {
      PrintWriter out = null;
      try {
            System.out.println("Ingresso nel blocco try");
            out = new PrintWriter(new FileWriter("out.txt");
            for (int i=0; i < size; i++)
                  out.println("Valore " + i + "=" + v.elementAt[i];
      } catch(ArrayIndexOutOfBoundsException e) {
            System.err.println("ArrayOutOfBoundException!");
      } catch(IOException e) {
            System.err.println("IOException!");
      } finally {
            if (out != null) {
                  System.out.println("Chiudo PrintWriter");
                  out.close();
            } else {
                  System.out.println("PrintWriter non è aperto");
            }
      }
}
```

Possono darsi tre casi:

- Viene sollevata IOException

  Ingresso nel blocco try IOException!
  PrintWriter non è aperto

- Viene sollevata ArrayOutOfBoundsException

  Ingresso nel blocco try

ArrayOutOfBoundsException!
Chiudo PrintWriter

- Il blocco try termina normalmente

Ingresso nel blocco try
Chiudo PrintWriter

LA PROGRAMMAZIONE CONCORRENTE

Nei linguaggi orientati agli oggetti, il concetto di oggetto consente di dividere un programma in unità di codice indipendenti. Tuttavia, in molte applicazioni è spesso necessario o conveniente suddividere un programma in *flussi di esecuzione indipendenti.*

Con il termine "programmazione concorrente" ci riferiamo alla possibilità di implementare dei programmi che contengano più flussi di esecuzione paralleli

Si distingue tra **processi** e **thread** (o "processi leggeri")

- Processi diversi eseguono concorrentemente senza condividere dati
- Thread diversi eseguono concorrentemente condividendo dati

PROCESSI E THREAD

La programmazione concorrente viene tradizionalmente supportata da **processi** concorrenti, a livello di sistema operativo (multitasking)

- Un processo esegue il codice di un intero programma
- Processi diversi hanno spazi di indirizzamento diversi: essi eseguono concorrentemente senza condividere dati attraverso la memoria

La gestione dei processi concorrenti è dispendiosa

- Creare un processo per ogni attività parallela è costoso
- Lo stesso vale per la comunicazione fra processi, poiché non sfrutta la memoria condivisa

Quindi, sono stati introdotti i thread (o lightweight process, processi leggeri) nei sistemi operativi multithreaded

Thread diversi eseguono all'interno dello stesso spazio di indirizzamento (cioè all'interno dello stesso processo) e possono quindi comunicare condividendo memoria all'interno di esso

Processi e thread consentono quindi di avere attività parallele con diverse granularità

PREEMPTION VS COOPERAZIONE

Sistemi diversi usano modelli di concorrenza diversi:

- i sistemi più semplici usano un modello cooperativo, in cui i processi o thread di volta in volta usano e poi rilasciano esplicitamente la CPU, secondo le loro politiche

La realizzazione del supporto runtime è più semplice, ma la gestione della concorrenza è lasciata al programmatore: errori nella cooperazione possono bloccare il sistema

- I sistemi più moderni usano un modello di tipo **preemptive**, in cui l'esecuzione di un processo o thread può essere interrotta dal supporto runtime, per permettere ad altre attività di usare la CPU

  Spesso si usa un meccanismo di *time slicing*, in cui ogni processo o thread può usare la CPU per un determinato quanto di tempo, scaduto il quale il controllo passa ad un altro

LA PROGRAMMAZIONE CONCORRENTE IN JAVA

Java supporta sia la gestione dei processi che dei thread

- Per la prima, si appoggia ai meccanismi del sistema operativo in cui esegue; per la seconda, viene supportata direttamente a livello del linguaggio e dell'interprete

- Usa un modello preemptive, in cui un thread a priorità più elevata interrompe quello in esecuzione

- Non necessariamente usa un meccanismo di time slicing, adottato tuttavia da alcune implementazioni

In particolare, Java fornisce:

- una classe **Thread** con metodi per attivare, terminare, sospendere un thread, e cambiarne la priorità

- costrutti per specificare che l'accesso a un oggetto condiviso tra più thread deve avvenire in mutua esclusione

- meccanismi di sincronizzazione per coordinare l'accesso a oggetti condivisi

CREAZIONE DI NUOVI THREAD

Un nuovo thread può essere creato istanziando un oggetto di una sottoclasse della classe di libreria Thread

Tale sottoclasse deve ridefinire il metodo run() della classe Thread inserendovi il codice che si vuole eseguire concorrentemente

Il thread così creato si fa partire invocando il metodo start()

THREAD: ESEMPIO

t1 e t2 sono creati con la new specificando il costruttore

```
public class ProvaThread {
public static void main(String[] args) {
      MyThread t1,t2;
      t1=new MyThread("primo thread");
      t2=new MyThread("secondo thread");
      t1.start();
      t2.start();
      }
}
```

la classe MyThread ha un costruttore che prende una stringa

```
class MyThread extends Thread {
private String message;
public MyThread(String m) {message=m;}
public void run() {
      for(int r=0; r<20; r++) System.out.println(message);
      }
}
```

run() è un metodo esportato da Thread ed è ridefinito (contiene il codice che fa partire il thread)
run() può invocare metodi di altri oggetti

La creazione di un thread non fa partire le attività concorrenti, dobbiamo farle partire con la start()

CREAZIONE DI NUOVI THREAD: UN PROBLEMA

Per creare un nuovo thread di esecuzione bisogna creare nuove sottoclassi di Thread, una per ogni nuova attività che si vuole eseguire

- Ciò può portare alla creazione di un gran numero di classi che differiscono unicamente per il solo metodo run()

- In un certo senso, è “scorretto” usare questa tecnica, in quanto si dovrebbe creare una sottoclasse solo quando quest’ultima modifica qualcuna delle funzionalità di base della superclasse (il che non è in questo caso)

Inoltre, questo metodo non consente di utilizzare classi che già ereditano da un’altra sottoclasse, diversa da Thread

CREAZIONE DI NUOVI THREAD: UN METODO ALTERNATIVO

Il secondo modo sfrutta l'interfaccia **Runnable**, la quale fornisce il metodo run(), e uno dei tanti costruttori di Thread, il quale permette di costruire un oggetto Thread a partire da un Runnable

```
public class MyThread implementsRunnable
{
      private String message;

      public MyThread(String m) {message = m;}

      public void run()
      {
            for(int r=0; r<20; r++)
            System.out.println(message);
      }
}

public class ProvaThread
{
      public static void main(String[] args)
      {
            MyThread r1, r2;
            Thread t1, t2;

            r1 = new MyThread("primo thread");
            r2 = new MyThread("secondo thread");
            t1 = new Thread(r1);
            t2 = new Thread(r2);

            t1.start();
            t2.start();
      }
}
```

Runnable ha un solo metodo run() ed ogni classe che implementa l'interfaccia deve ridefinirlo.

Cambia il modo in cui uso ed inserisco il thread.

MyThread r1, r2; → le istanze di MyTread
Thread t1, t2; → attività concorrenti vere e proprie

Prima creo due istanze di r1 ed r2

poi creo t1 e t2 vuote inizializzate con due oggetti che implementano runnable e specificano il comportamento applicativo del thread

poi invoco start() sugli oggetti di tipo thread

Posso farlo in un solo passo:

```
t1 = new Thread(new MyThread("primo thread"));
```

NON DETERMINISMO

```
public class TestThreads {
  public static void main(String[] args) {
    Thread[] threads = new Thread[Integer.parseInt(args[0])];
    for (int i = 0; i < threads.length; i++) {
      Runnable r = new MsgThread (i);
      threads[i] = new Thread(r);
      threads[i].start();
    }
    System.out.println("Sono il thread main...vado a dormire.");
    try {
      Thread.sleep(5000);
    } catch(InterruptedException e) { System.exit(0); }
    System.out.println("OK. La festa e' finita.");
    for (int i = 0; i < threads.length; i++) threads[i].stop();
  }
}
class MsgThread implements Runnable {
  final int myNumber;
  static final String msg = "Sono il thread n. ";
  MsgThread(int myNumber) { this.myNumber = myNumber; }
  public void run() {
    while(true) {
      System.out.println(msg + myNumber);
      try {
        Thread.sleep((long) (1000));
      } catch(InterruptedException e) {System.exit(0); }
    }
  }
}
```

- All'interno della classe TreadTest ho un array di thread il cui numero lo passo attraverso la command line.
- Per tutti gli elementi dell'array viene creato un thread.
- Dopo è stampato un messaggio dal main e poi un operazione sul thread che lo fa addormentare per s secondi. Dopo il main si risveglia e stampa un altro messaggio.
- final int myNumber; → è una costante che può avere valori diversi in ciascuno degli oggetti
- static final String msg = "Sono il thread n. "; → definita a livello di classe e non di oggetto
- MsgThread(int myNumber) { this.myNumber = myNumber; } → in myNumber è passato l'indice dell'array
- run() stampa il messaggio e si addormenta per 1 secondo.
- C'è un blocco try perché il metodo può sollevare un'eccezione

L'esecuzione dei vari thread procede senza un ordine predefinito
Lo stesso codice, eseguito su una computer diverso potrebbe dare un output diverso
L'ordine dipende da come vengono schedulati i processi, dalla caratteristiche del processore, …

Si dice che i thread eseguono in maniera non-deterministica

Il non-determinismo è una caratteristica fondamentale della concorrenza ed è ciò che la rende complicata da gestire

NON DETERMINISMO E RISORSE CONDIVISE

Il non determinismo ha un impatto sulle risorse condivise.

Esempio:

Siano x=500, y=400, z=100 tre variabili di un programma concorrente, e si supponga l'esistenza di un vincolo tale per cui la loro somma s deve rimanere costante e pari a 1000

s potrebbe rappresentare la somma complessiva di denaro presente nei conti bancari il cui ammontare è rappresentato da x, y, e z

Nel sistema esistono due thread: a causa del non-determinismo non è possibile stabilire a priori la sequenza definita dall'intrecciarsi delle loro istruzioni

Thread 1	Thread 2
read(x)	
	read(y)
	y = y - 100
	read(z)
	z = z + 100
	write(y)
	write(z)
read(y)	
read(z)	
s = x + y + z	

In questo caso, l'esecuzione termina correttamente con s = 1000

Thread 1	Thread 2
read(x)	
	read(y)
read(y)	
	y = y - 100
	read(z)
	z = z + 100
	write(y)
	write(z)
read(z)	
s = x + y + z	

In questo caso, invece, il valore di y viene letto troppo presto, durante la sua modifica, e quindi la somma ritorna un valore errato di s = 1100

Vanno prese contromisure per le **sezioni critiche**

MONITOR

Ad ogni oggetto Java, cioè a ogni istanza di **Object** o di una sua sottoclasse, è implicitamente associato un **lock**, cioè una variabile che determina se l'oggetto è "libero" oppure è già acceduto in maniera esclusiva da un altro thread

È possibile controllare l'accesso concorrente di più thread a uno stesso oggetto dichiarando uno o più metodi dell'oggetto come **synchronized**

Si dice anche che a un oggetto contenente uno o più metodi synchronized è associato un monitor (Hoare), che governa l'acquisizione e il rilascio del lock

ACCESSO AL MONITOR

Quando unthread `t` tenta di accedere ad un oggetto `o` attraverso un metodo `o.m` dichiarato come `synchronized`:

- se nessun thread sta eseguendo **o.m**, **t**"acquisisce il monitor" sull'oggetto **o**, ed inizia ad eseguire il metodo **m**

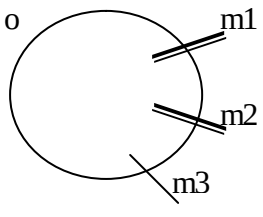
- da questo momento in avanti, l'accesso ad **o** diventa esclusivo per il thread **t**: se altri thread cercano concorrentemente di accedere ad **o** attraverso metodi **synchronized**, questi thread vengono sospesi fino a quando **t** non "rilascia il monitor"

- il monitor viene rilasciato da **t** quando quest'ultimo termina l'esecuzione del metodo **synchronized o.m**

- quando ciò avviene, tutti i thread sospesi su **o** vengono risvegliati, e competono per acquisire il monitor: uno solo di essi, scelto in maniera non-deterministica, riuscirà ad ottenerlo

- se invece all'atto dell'invocazione di **o.m**da parte di **t** un altro thread **t'** sta già eseguendo **o.m**, **t** si blocca fino a quando **t'**non rilascia il monitor

Quando un metodo synchronized viene invocato da un altro metodo synchronized appartenente al medesimo oggetto, il thread chiamante non deve competere per il monitor, in quanto quest'ultimo è già stato acquisito durante l'invocazione del primo metodo (reentrant lock)
L'accesso mutuamente esclusivo vale solo per i metodi dichiarati synchronized: l'accesso attraverso gli altri metodi non è mutuamente esclusivo, cioè può avvenire anche mentre un thread ha acquisito il monitor
È possibile dichiarare come synchronized anche metodi static: ogni classe ha infatti associato un suo monitor, distinto da quello degli oggetti istanziati a partire da essa

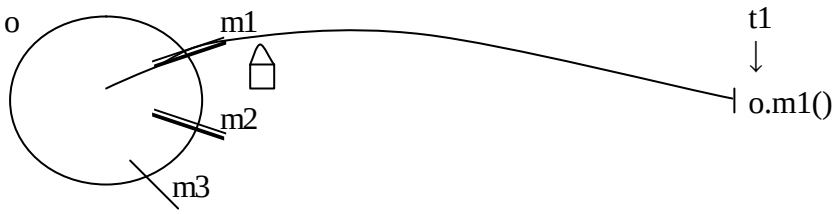


Abbiamo un oggetto **o** che definisce tre metodi. m1 ed m2 sono **synchronized**

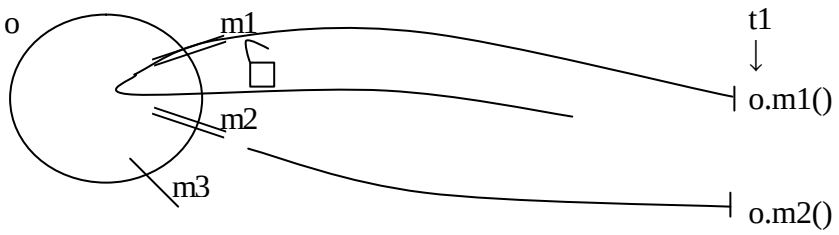
Supponiamo che esistano t1 e t2 in esecuzione

t1 contiene un invocazione a o.m1()

m1() blocca l'accesso all'oggetto usando il lock

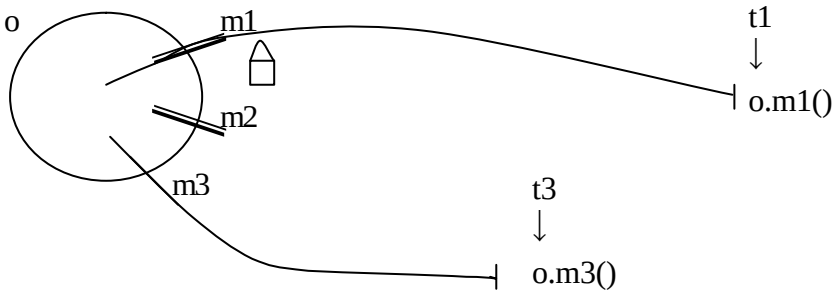


Se t2 cerca di eseguire o.m1() oppure o.m2(), si sospende fino a quando l'oggetto o non viene rilasciato dell'oggetto, cioè quando m1 termina l'esecuzione.



Nessun metodo synchronized è eseguibile se c'è un lock.

Se abbiamo t3 che contiene una chiamata a m3() non synchronized, la chiamata ha sempre successo anche se non c'è il lock



Se dentro m1() ho un invocazione ad m2(), non ho problemi perché ho già ottenuto il lock.

L'uso dei metodi synchronized è costoso ed andrebbero usati sono in caso di necessità (tipicamente in tutti i casi in cui faccio operazioni critiche), posso non usarli quando faccio operazioni di lettura.

BLOCCO SYNCHRONIZED

Talvolta risulta necessario controllare l'accesso concorrente a porzioni di codice con una granularità più fine del metodo

Più e grande la porzione di codice sincronizzata, minore il parallelismo

In questi casi, è possibile impiegare il blocco synchronized

```
synchronized(obj) {
     … codice critico …
}
```

La semantica del blocco synchronized è simile a quella precedentemente descritta, con la differenza che il monitor viene acquisito sull'oggetto obj specificato come parametro

I blocchi synchronized sono più generali dei metodi synchronized: i secondi possono essere rappresentati in maniera equivalente utilizzando i primi

```
void synchronized m() {
     … codice critico …
}
```

equivale a:

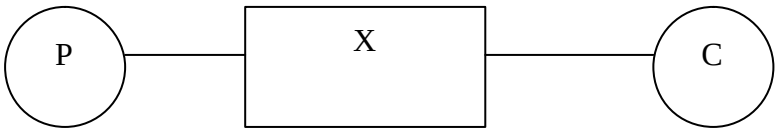
```
void m() {
     synchronized(this) {
          … codice critico …
     }
}
```

SINCRONIZZARE ATTIVITA' CONCORRENTI

Garantire un accesso mutuamente esclusivo allo stato dell'oggetto spesso non basta: è necessario fornire ai thread la possibilità di coordinare le loro operazioni

Esempio classico: produttore-consumatore

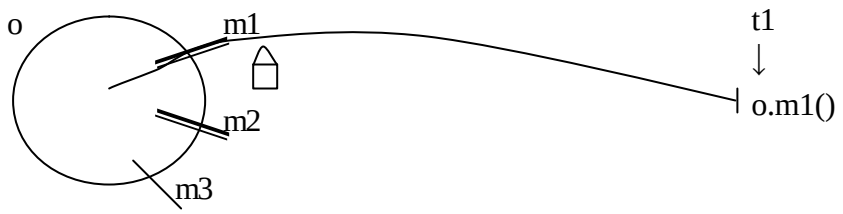
- Due thread, un produttore P e un consumatore C, condividono una risorsa X (ad esempio, per semplicità, una variabile contenente un intero)
- P "produce" un nuovo valore di X, in momenti a priori non noti
- Analogamente, C "consuma" il valore di X, leggendolo
- Se P e C non sono sincronizzati, è possibile che vi siano delle inconsistenze tra i dati prodotti e quelli consumati:
    - Se P è più veloce di C, un valore prodotto potrebbe sovrascrivere un valore in X che non è ancora stato letto da C
    - Se C è più veloce di P, uno stesso valore in X potrebbe essere letto più volte



SOSPENDERE UN THREAD

Quando un thread invoca il metodo **wait** su un oggetto, il thread si sospende e rilascia il monitor associato a tale oggetto

I thread che erano sospesi in attesa di poter accedere al blocco synchronized vengono risvegliati

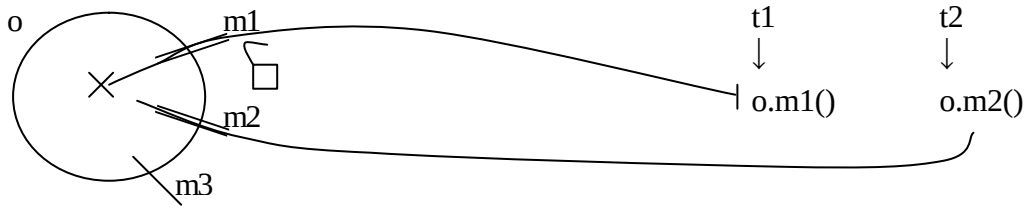


Se in m1() ho una wait:

```
m1() {
     ...
     if(condizione)
          wait();
     ...
}
```

ho come effetto quello di rimuovere il lock.

Da notare che m1() non è terminata, ma in attesa di essere risvegliata.



A questo punto o.m2() va in esecuzione.

Nel produttore-consumatore:

Se P verifica che C non ha ancora letto il valore di X, P si può sospendere lasciando il controllo a C, che può quindi prima o poi effettuare la lettura;

Analogamente, se C verifica che P non ha ancora prodotto un nuovo valore, si può sospendere

RISVEGLIARE UN THREAD

I thread sospesi da una wait possono essere risvegliati da una **notify** o una **notifyAll**

- **notify** risveglia un thread scelto non-deterministicamente fra tutti quelli sospesi, mentre notifyAll risveglia tutti i thread

- **notifyAll** è più costosa, ma più sicura: infatti, può darsi che il thread selezionato da notify non possa procedere e venga immediatamente ri-sospeso, bloccando l'intero programma

- È disponibile anche una **wait(long millisec)**, che permette di associare un timeout alla wait, scaduto il quale il thread viene automaticamente risvegliato

Nel produttore-consumatore:

P, sospeso in attesa di poter produrre un nuovo dato, viene svegliato da C non appena quest'ultimo legge il valore di X (e analogamente per il consumatore)

PRODUTTORE-CONSUMATORE IN JAVA

Produttore e consumatore condividono un buffer rappresentato dalla classe buffer che contiene un intero privato **sec** che rappresenta il dato prodotto e contiene un flag **available** per dire se esiste un dato disponibile per il consumatore.

Il *consumatore* verifica se available è **true** e legge il dato, altrimenti si sospende ed aspetta di essere risvegliato dal produttore dopo che questo ha prodotto il dato.

Funziona tramite i metodi synchronized della classe buffer.

```
class Producer extends Thread {
  private Buffer buf;
  private int number;
  public Producer(Buffer b)
   { buf = b; }
  public void run() {
    for (int i = 0; i < 10; i++)
      buf.put(i);
      System.out.println(
        "Producer put: " + i);
    }
  }
}
```

```
class Consumer extends Thread {
  private Buffer buf;
  public Consumer(Buffer b)
   { buf = b; }
  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      value = buf.get();
      System.out.println(
        "Consumer got: " + value);
    }
  }
}
```

```
class Buffer {
 private int seq;
 private boolean available = false;
 public synchronized int get() {
  while (!available) {wait();}
  available = false;
  notifyAll();
  return seq;
 }
 public synchronized void put(int value){
  while (available) {wait();}
  seq = value;
  available = true;
  notifyAll();
 }
}
```

```
public class ProducerConsumerTest {
  public static void main(String[] args) {
    Buffer b = new Buffer();
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);
    p.start();
    c.start();
  }
}
```

La classe **ProducerConsumerTest** crea il buffer, il produttore ed il consumatore e poi avvia i due thread.

Producer e Consumer hanno un buffer provvisorio **buf**.

Producer ha **number** che è il dato.
run() inserisce un elemento il cui valore è quello dell'indice del ciclo for, e stampa il messaggio "*producer put: …*" seguito dal numero dell'elemento.

```
buf.put(i);
System.out.println( "Producer put: " + i);
```

Consumer non ha l'intero da prelevare che è messo come variabile locale nel run()
run() fa una get() dal buffer

```
value = buf.get();
```

I metodi con cui produttore e consumatore inseriscono e prelevano i dati sono **put()** e **get()** della classe Buffer.

put() prende come elemento il valore che deve essere inserito e poi c'è un ciclo finchè esiste un dato nel buffer, il codice della put() sospende il chiamante ed il controllo passa al consumatore.

```
while (available) {wait();}
seq = value;
available = true;
notifyAll();
```

Viene invocato **get()**, il consumatore è risvegliato (se non ci sono dati aspetta), setta il valore di available a false, con la notifyAll() risveglia tutti i thread e stampa il valore.

```
while (!available) {wait();}
available = false;
notifyAll();
return seq;
```

notifyAll() risveglia i processi sospesi con una wait.
Tutti competono per eseguire, viene eseguito un thread e gli altri tornano nello stato di sospeso.
Il risveglio avviene quando il metodo synchronized termina la sua esecuzione.

Vediamo cosa succede in esecuzione:

Senza la sincronizzazione:

Producer put: 0
Producer put: 1
Producer put: 2
Producer put: 3
Producer put: 4
Producer put: 5
Producer put: 6
Producer put: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Consumer got: 7
Producer put: 8
Producer put: 9

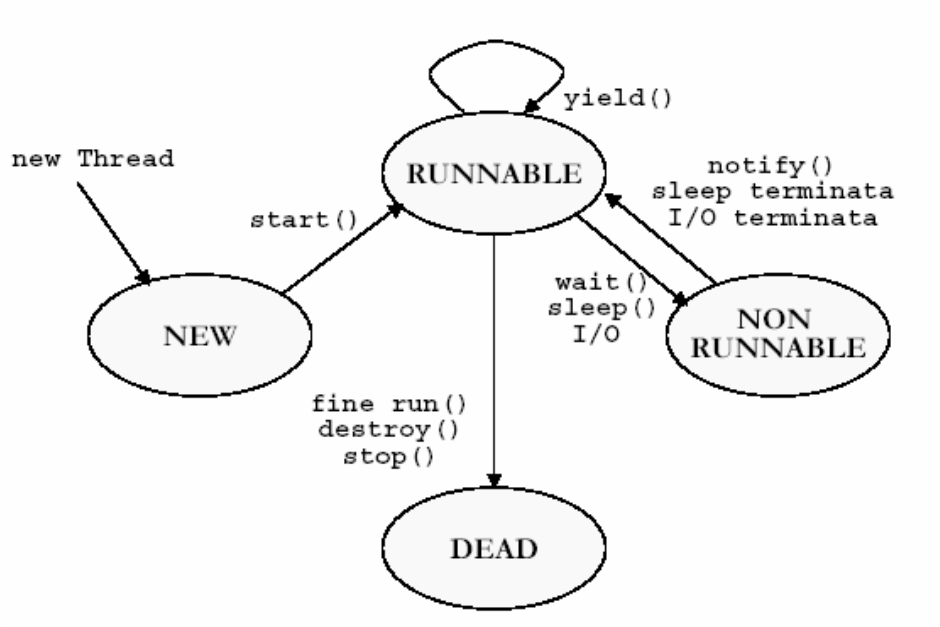
Il consumatore perde quasi tutti i dati.

Con la sincronizzazione:

Producer put: 0
Producer put: 1
Consumer got: 0
Producer put: 2
Consumer got: 1
Producer put: 3
Consumer got: 2
Producer put: 4
Consumer got: 3
Producer put: 5
Consumer got: 4
Producer put: 6
Consumer got: 5
Producer put: 7
Consumer got: 6
Producer put: 8
Consumer got: 7
Producer put: 9
Consumer got: 8
Consumer got: 9

nessun dato perso (Si noti come la stampa dei messaggi sia fuori ordine: essa infatti non è all'interno della sezione sincronizzata)

CICLO DI VITA DI UN THREAD



Quando un thread è creato, va in uno stato NEW

Il thread può essere attivato con una **start()** diventando RUNNABLE (potenzialmente può essere eseguito.

Da runnable, posso passare a NON RUNNABLE (sospeso) mediante **wait()** oppure **sleep()**

Passa allo stato di DEAD quando finisce oppure con metodi di **destroy()** o **stop()**

Da notare **yield()**, usata per passare esplicitamente il controllo ad un altro thread.
Essa rilascia i monitor correntemente posseduti dal thread. È necessario il suo uso nelle implementazioni della Java virtual machine che non hanno preemptive time slicing.

SPIN LOCK

Una pessima tecnica per risolvere problemi simili al produttore consumatore è quella di usare, al posto di

```
while (!available) { wait(); }
```

istruzioni del tipo

```
while (!available) { yield(); }
```

Questo tipo di istruzioni vengono chiamate spin lock (o busy-wait, o spin loop) in quanto il thread non viene mai sospeso, ma continua a testare il valore della variabile available ogni volta che viene selezionato per essere eseguito, finché tale variabile non diventa vera.

Questa soluzione ha due forti svantaggi:

- Può sprecare tempo di CPU per un tempo indefinito: infatti, la versione basata sulla wait controlla la variabile solo quando viene notificata che lo stato del sistema è cambiato, mentre questa lo fa continuamente

- Non garantisce che altri thread possano accedere alla variabile condivisa: infatti, se il thread che esegue la yield ha priorità più alta degli altri, continuerà ad eseguire da solo, mentre la wait di fatto rimuove il thread dallo stato di Runnable

SAFETY E LIVENESS

Le proprietà di correttezza di un sistema concorrente sono di due tipi:

- **safety**: il sistema non entra in stati indesiderati
  Es. available $\Rightarrow$ buffer pieno, $\neg$available $\Rightarrow$ buffer vuoto

- **liveness**: il sistema prima o poi entra negli stati desiderati
  Es. prima o poi un valore prodotto dal produttore viene letto dal consumatore

  Safety: bad things never happen
  Liveness: good things eventually happen

La correttezza del sistema complessivo dipende da ambedue le proprietà

A seconda degli scenari applicativi, una delle due può essere più importante dell'altra (es. impianti nucleari e interfacce utente)

DEADLOCK E STARVATION

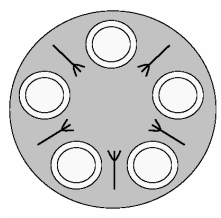
La liveness di un sistema può venire a mancare in molti modi: deadlock e starvation sono fra i più rilevanti

- **Deadlock**: l'intero sistema è bloccato, in quanto ciascuna delle attività concorrenti è in attesa che altre rilascino alcune delle risorse condivise necessarie alla computazione

- **Starvation**: un attività concorrente, benché abilitata ad eseguire, non riesce a farlo perché le risorse necessarie sono ottenute soltanto dalle altre attività

Evitare situazioni di deadlock è completamente a carico del programmatore.

Esempio: i cinque filosofi



Cinque filosofi si trovano a cena. Ciascuno ha bisogno di due forchette per mangiare. Ogni filosofo alternativamente mangia o pensa Un caso di *deadlock* si verifica se ogni filosofo prende sempre la forchetta alla sua sinistra, e attende che quella di destra si liberi. Un caso di *starvation* si verifica se i vicini di un filosofo F sono più rapidi di F e riescono sempre a prendere ambedue le forchette prima di lui.

TERMINARE UN THREAD

Un thread termina “naturalmente” la sua esecuzione quando raggiunge la fine del metodo run()
Tuttavia, è talvolta necessario terminarlo esplicitamente (in particolare quando il thread esegue all’interno di un ciclo infinito).
Java fornisce il metodo **stop()** che, invocato su un oggetto Thread, ne causa la terminazione

Tuttavia, tale metodo è deprecato dalla versione 1.2 del linguaggio, in quanto insicuro
Infatti, termina l’esecuzione del thread indipendentemente dal punto dell’esecuzione in cui si trova; pertanto, non dà modo al thread di rilasciare risorse condivise, e quindi è potenzialmente causa di deadlock.
Per analoghi motivi, anche le primitive **suspend** e **resume**, che sospendono e risvegliano un thread, sono deprecate, e la primitiva **destroy** non è mai stata implementata

TERMINARE UN THREAD IN MAINERA SICURA

Per evitare i problemi causati da **stop()**, si può strutturare il codice in maniera opportuna.

esempio:

```
...
private Thread t;
public void start() {
  t = new Thread(this);
  t.start();
}
public void stop() {
  t.stop(); // NON SICURO!
}
public void run() {
  Thread thisThread =
    Thread.currentThread();
  while (true) {
    try { thisThread.sleep(1000); }
    catch(InterruptedException e){ ... }
    faiQualcosa ();
  }
}
```

In maniera sicura:

```
...
private Thread t;
public void start() {
  t = new Thread(this);
  t.start();
}
public void stop() {
  t = null;
}
public void run() {
  Thread thisThread =
    Thread.currentThread();
  while (t == thisThread) {
    try { thisThread.sleep(1000); }
    catch(InterruptedException e){ ... }
    faiQualcosa ();
  }
}
```

INTERROMPERE UN THREAD

La tecnica precedente non funziona quando il thread è sospeso, ad esempio su una **wait**, una **sleep**, o una **join**.
In questi casi, il metodo **interrupt()** fornisce un modo sicuro per terminare il thread.
L'invocazione di tale metodo su un oggetto thread fa sì che venga sollevata un'eccezione **InterruptedException** consentendo così al thread di gestire l'interruzione in maniera dipendente dal suo stato attuale

Nel caso precedente:

```
public void stop() {
     Thread t2 = t;
     t = null;
     t2.interrupt();
}
```

Questa tecnica fa sì che il flag di terminazione sia settato nel caso in cui il thread stia eseguendo, e una **InterruptedException** sia propagata nel caso in cui il thread sia all'interno della **sleep**.

ALTRE PRIMITIVE PER LA GESTIONE DEI THREAD

JOIN

Il metodo **join** consente al thread chiamante di sospendersi in attesa della terminazione del thread corrispondente all'oggetto su cui il metodo è invocato

```
Thread t = new Thread(aRunnable);
t.start();
try {
     t.join();
} catch(InterruptedException e) { … }
```

CURRENTTHREAD

Il metodo (di classe) **currentThread** ritorna un riferimento al thread attualmente in esecuzione (cioè al thread chiamante, che ha effettuato l'invocazione)

```
Thread myself = Thread.currentThread
```

Si noti che se il thread è stato creato usando un Runnable, il thread così ritornato è diverso dall'oggetto Runnable

GESTIONE DELLA PRIORITA’

I thread all’interno di un singolo processo eseguono tutti condividendo la stessa CPU

Tuttavia, non tutte le attività concorrenti hanno la stessa importanza: in Java, esistono costrutti appositi per specificare la priorità associata a un thread (e quindi garantire ad alcuni thread un accesso più frequente alla CPU)

Un thread ha per default la stessa priorità del thread che l’ha creato; tale priorità può essere modificata con il metodo **setPriority**

Lo scheduler dell’interprete Java selezionerà per primi fra i thread nello stato Runnable quelli con più alta priorità

Nota:
Gli algoritmi di scheduling dipendono dall’implementazione della JVM

GRUPPI DI THREAD

Talvolta è utile raggruppare un insieme di thread e manipolarli tutti in una volta come se fossero un unico thread, anziché separatamente

Ad esempio, può essere utile cambiare la priorità di tutti in un colpo solo.

Questa funzionalità viene fornita dalla classe **ThreadGroup**.

Normalmente, i thread sono inseriti in un gruppo di default: è possibile creare un thread in un altro gruppo usando gli opportuni costruttori

Esempio:

```
Thread t = new Thread(group,runnable)
```

La classe ThreadGroup contiene metodi che consentono di:

- gestire la collezione di thread, es. aggiungendo o togliendone alcuni dal gruppo, enumerarli, verificare quanti sono attivi
- cambiare la priorità dei thread nel gruppo
- terminare l’esecuzione dei thread nel gruppo (anche se i metodi forniti, stop, suspend, resume, soffrono dei problemi precedentemente menzionati per i thread)

CONSIDERAZIONI FINALI

I **thread** hanno un costo, sia in termini di prestazioni che in termini di complessità: vanno utilizzati solo quando ciò è reso necessario dalle esigenze dell'applicazione

Ogniqualvolta si pensa di definire un nuovo thread, conviene chiedersi se la funzionalità da esso fornita può essere ottenuta attraverso un nuovo oggetto acceduto attraverso lo stesso flusso di controllo

La **sincronizzazione** ha a sua volta un costo

Per alcune implementazioni, invocare un metodo synchronized costa 4 volte tanto un'invocazione normale

Tuttavia, la sincronizzazione ha un impatto fondamentale sulla correttezza del sistema

La parola chiave synchronized non va sparsa ovunque; al contrario, si deve cercare di delimitare il più possibile le sezioni critiche, in modo da massimizzare il parallelismo del sistema