

ORACLE FORNISCE 3 STACK TECNOLOGICI UTILI ALL' IMPLEMENTAZIONE DELLE APPLICAZIONI

- APPLICAZIONI STAND ALONE (DESKTOP)  
APPLICAZIONI CHE VENGONO ESEGUITE LOCALMENTE SU UN DISPOSITIVO  
**JAVA SE (STANDARD EDITION)**

APPLICAZIONI WEB (ENTERPRISE)  
APPLICAZIONI CHE PREVEDONO L'USO DEL PROTOCOLLO HTTP E QUINDI ADERISCONO AL MODELLO  
INFRASTRUTTURALE CLIENT SERVER (DEVICE DIVERSI DA SMARTPHONE)  
**JAVA EE (JAVA ENTERPRISE EDITION)**

APPLICAZIONI MOBILE  
APPLICAZIONI CHE PREVEDONO L'USO DEL PROTOCOLLO HTTP E QUINDI ADERISCONO AL MODELLO  
INFRASTRUTTURALE CLIENT SERVER (SMARTPHONE)  
**JAVA ME (JAVA MOBILE EDITION)**

STACK TECNOLOGICO = INSIEME DI API (APPLICATION PROGRAM INTERFACE)

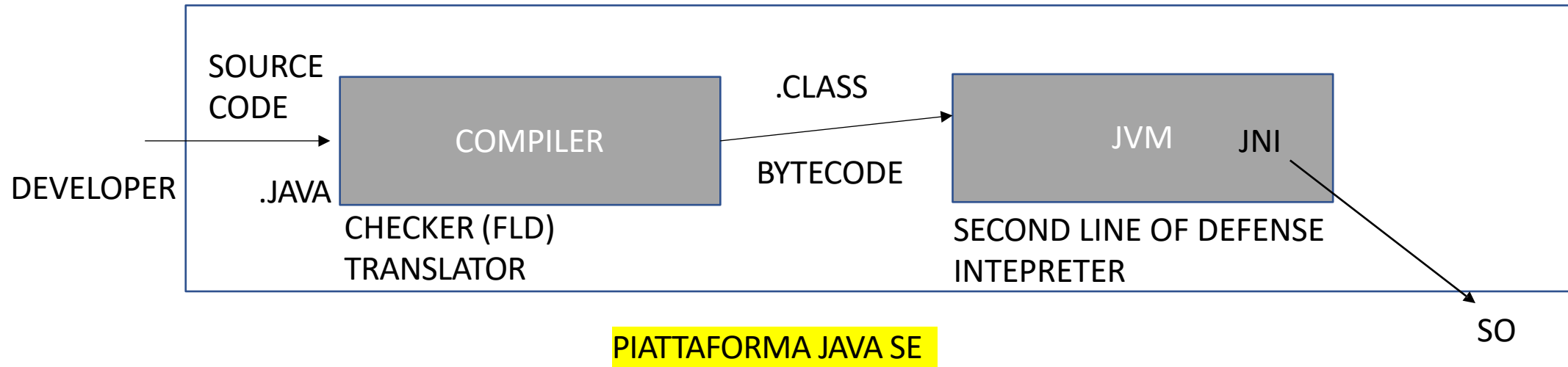
API--→>>> COMPONENTI APPLICATIVE READY (BUILT-IN)

JAVA E' UN LINGUAGGIO CON LE SEGUENTI CARATTERISTICHE:

- COMPILATO
- INTERPRETATO
- FORTEMENTE TIPIZZATO (STRONGLY TYPED) -->>> ad ogni variabile dichiarata/inizializzata occorre assegnare un tipo in fase di scrittura del codice; ad ogni metodo dichiarato deve essere associato un tipo di ritorno in fase di scrittura del codice
- ADERENTE (COMPLIANT) a DUE PARADIGMI : OOP e FUNZIONALE

JAVA SE = INSIEME DI API FORNITE DA ORACLE AL FINE DI IMPLEMENTARE APPLICAZIONI DESKTOP

LA PIATTAFORMA NECESSARIA AD ESEGUIRE APPLICAZIONI SE E' COMPOSTA DA COMPILER E JVM



PARADIGMA = FILOSOFIA DI PENSIERO APPLICATIVA = UNA METODOLOGIA DI APPROCCIO DI SCRITTURA DEL CODICE BASATA SU PRINCIPI E CONCETTI

OOP

CONCETTI : ENTITA', OGGETTO

PRINCIPI : SEPARATION OF CONCERNS, CODE REUSE, INHERITANCE, POLYMORPHISM, HAS-A, INFORMATION HIDING (ENCAPSULATION)

FUNZIONALE

CONCETTI : FUNZIONE

PRINCIPIO : AD UNA VARIABILE PUO' ESSERE ASSEGNATA LA DICHIARAZIONE DI UNA FUNZIONE, COSI' COME NELLA LISTA DEI PARAMETRI DI UNA FUNZIONE POSSO PASSARE COME ARGOMENTO UNA FUNZIONE STESSA

## OOP

ENTITA' = INSIEME DI PROPRIETA' E COMPORTAMENTI

OGGETTO = UNA DELLE N POSSIBILI RAPPRESENTAZIONI  
DI UNA ENTITA'

## JAVA

CLASSE = INSIEME STATICO DI VARIABILI E METODI =  
PROTOTIPO (BLUEPRINT) DI UN OGGETTO  
OGNI CLASSE JAVA STABILISCE LA STRUTTURA DI TUTTI GLI  
OGGETTI CHE VERRANNO CREATI A PARTIRE DALLA CLASSE

OGGETTO = UNA DELLE N POSSIBILI ISTANZE DI UNA CLASSE

ASSIOMI JAVA -→>>> REGOLE STABILITE DAL LINGUAGGIO E SONO INCONTROVERBILI

ASSIOMA -→>>>> In ogni Applicazione Java Stand Alone è necessario implementare una Classe all'interno della quale è presente un metodo di nome main

Il metodo main rappresenta l'entry point per il COMPILER

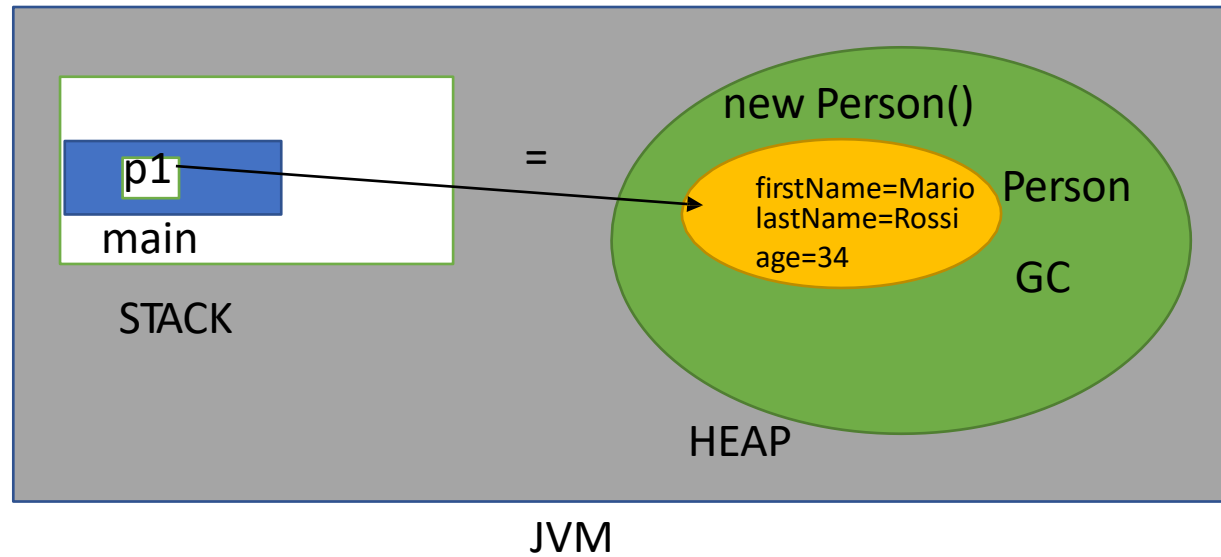
ASSIOMA -→>> Ogni Classe HA AL SUO INTERNO un metodo «SPECIALE» di nome Costruttore (tale metodo ha lo stesso nome della classe)-→>> Costruttore Implicito, Costruttore vuoto, Costruttore di default

JVM = STACK (MEMORIA STATICA) + HEAP (MEMORIA DINAMICA) + JNI (INTERPRETER)

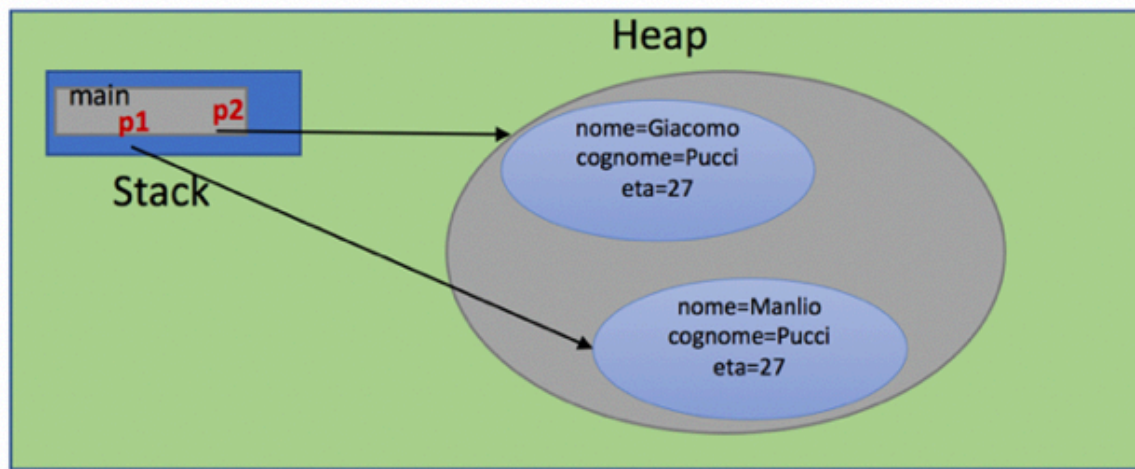
STACK ---→>>> GESTISCE IL CICLO DI VITA DEI METODI

HEAP -→>>> CICLO DI VITA DEGLI OGGETTI

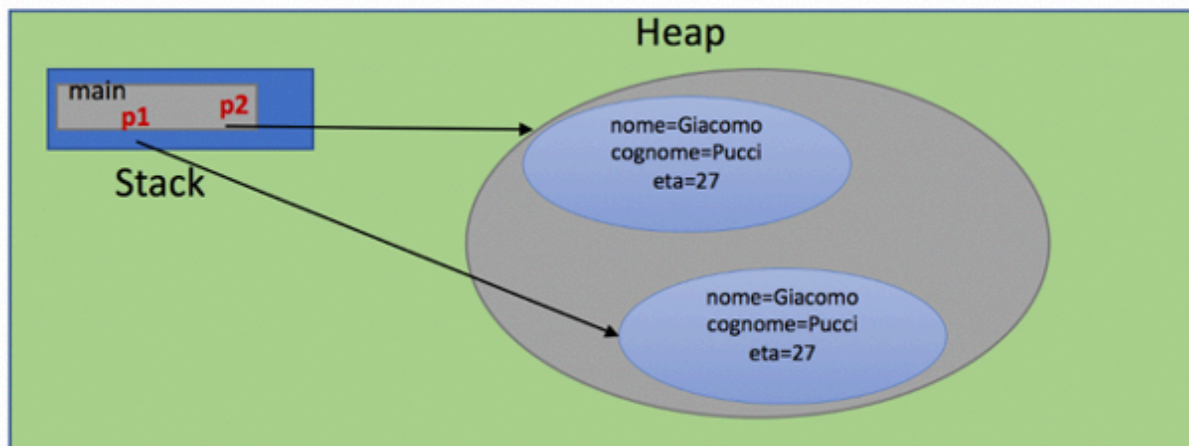
JNI--→>>TRADUCE IL BYTECODE PER ESSERE COMPRENSIBILE DAL SO



Una reference rappresenta un controllo remoto sull'Oggetto nel senso in cui è in grado di modificarne lo stato a runtime  
stato di un oggetto = contenuto dell'Oggetto = insieme delle variabili di istanza e dei relativi valori



**ESEMPIO DI OGGETTI DIVERSI**

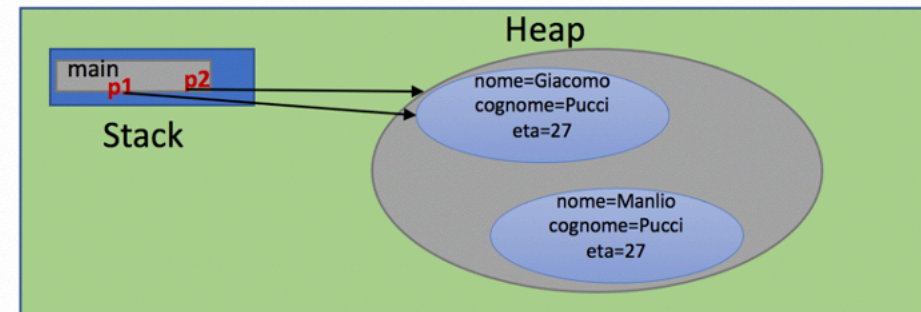


**ESEMPIO DI OGGETTI UGUALI**



LA REFERENCE p1 viene assegnata a p2 referenzia lo stesso Oggetto referenziato da p1  
l'Oggetto precedentemente referenziato da p2 è deferenziato per cui **eligible for destruction**

```
ApplicationMain.java X
1 package com.example;
2
3 public class ApplicationMain {
4
5     public static void main(String args[]) {
6
7         Persona p1 = new Persona("Giacomo","Pucci",27);
8
9         Persona p2 = new Persona("Manlio","Pucci",27);
10
11         p1=p2;
12
13         System.out.println(p1==p2);
14
15
16
17     }
18 }
19
20
21
22 }
23
Problems Javadoc Declaration Console X
<terminated> ApplicationMain [Java Application] /Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/bin/java (20 nov 2023, 17:16:28) [pid: 47993]
true
```



IN JAVA ESISTE IL CONCETTO DI UGUAGLIANZA TRA OGGETTI

DUE OGGETTI SONO UGUALI FRA DI LORO SE HANNO ESATTAMENTE LO STESSO CONTENUTO (STATO) PUR OCCUPANDO DUE AREE DI MEMORIA DIFFERENTI ALL'INTERNO DELL'HEAP

IN JAVA ESISTE IL CONCETTO DI UGUAGLIANZA TRA REFERENCE

DUE REFERENCE SONO UGUALI TRA DI LORO SE REFERENZIANO ESATTAMENTE LO STESSO OGGETTO, OVVERO SE SI RIFERISCONO ALLO STESSO INDIRIZZO DI MEMORIA OCCUPATO NELL'HEAP DALL'OGGETTO

UN OGGETTO JAVA DIVENTA ELIGIBLE FOR DESTRUCTION PER IL GC (APPLICAZIONI JAVA STAND ALONE)

1. SE E' STATA ASSEGNATA A NULL LA REFERENCE CHE LO REFERENZIAVA
2. SE LA REFERENCE CHE LO REFERENZIAVA E' STATA ASSEGNATA AD UN ALTRO OGGETTO
3. INDIPENDENTEMENTE DA TUTTO ALLA FINE DELL'ESECUZIONE DEL METODO main

UN LIVELLO DI VISIBILITA' JAVA PUO' ESSERE APPLICATO A VARIABILI, METODO E CLASSI E DETERMINA IN QUALI PUNTI DELL'APPLICAZIONE LE VARIABILI, I METODI O LE CLASSI POSSANO ESSERE USATE

IN JAVA ESISTONO 4 LIVELLI DI VISIBILITA' (MODIFICATORI DI ACCESSO)

- PRIVATE (VISIBILITA' LIMITATA ALL'INTERNO DELLA CLASSE)  
Per associare livello di visibilità PRIVATE occorre utilizzare la parola chiave private
- PACKAGE O DEFAULT (VISIVILITA' LIMITATA ALL'INTERNO DI TUTTE LE CLASSI APPARTENENTI ALLO STESSO PACKAGE)  
Per associare livello di visibilità PACKAGE o DEFAULT non occorre utilizzare alcuna parola chiave
- PROTECTED (VISIVILITA' LIMITATA ALL'INTERNO DI TUTTE LE CLASSI APPARTENENTI ALLO STESSO PACKAGE E ALL'INTERNO DI EVENTUALI SOTTOCLASSI PUR NON APPARTENENTI ALLO STESSO PACKAGE)  
Per associare livello di visibilità PROTECTED occorre utilizzare la parola chiave protected
- PUBLIC (VISIVILITA' IN TUTTA L'APPLICAZIONE)

## OOP

PRINCIPIO DI INFORMATION HIDING = E' PREFERIBILE NASCONDERE LE PROPRIETA' DICHIARATE DA UNA ENTITA' ALLE ALTRE ENTITA' (ENTITA' TERZE)

## JAVA

PRINCIPIO DI INFORMATION HIDING = E' PREFERIBILE NASCONDERE LE VARIABILI DI ISTANZA DICHIARATE DA UNA CLASSE ALLE ALTRE CLASSI (CLASSI TERZE)

JAVA OFFRE 2 MODALITA' PER L'IMPLEMENTAZIONE DEL PRINCIPIO DI INFORMATION HIDING:

1. DICHIARARE LE VARIABILI DI ISTANZA DI UNA CLASSE CON LIVELLO DI VISIBILITA' private e DICHIARARE i metodi di set e get (che usano le variabili di istanza) con livello di visibilità public
2. DICHIARARE LE VARIABILI DI ISTANZA DI UNA CLASSE CON LIVELLO DI VISIBILITA' private e IMPLEMENTARE UN COSTRUTTORE CUSTOM

Ogni metodo (concreto) Java deve essere caratterizzato da una firma (signature) ed un corpo (body)

La firma deve contenere:

- Livello di visibilità
- Tipo di ritorno del metodo
- Nome del metodo
- Lista dei parametri (gli elementi di input al metodo)

Il corpo di un metodo (concreto) è rappresentato dalle righe di codice inserite tra { e } che effettivamente verranno eseguite all'invocazione del metodo!!!!

ESISTE DIFFERENZA TRA LA DICHIARAZIONE DI UN METODO E LA SUA INVOCAZIONE

LA DICHIARAZIONE DI UN METODO RAPPRESENTA LA «PROMESSA» DELLE RIGHE DI CODICE CHE VERRANNO ESEGUITE IN FASE DI INVOCAZIONE

Ogni volta che viene invocato un metodo viene ALLOCATO uno spazio fisico dedicato al metodo all'interno dello STACK che permane per tutto il tempo di esecuzione del metodo stesso e viene DEALLOCATO al termine dell'esecuzione del metodo

Un Costruttore è un metodo speciale che ha lo stesso nome della Classe e non ha alcun tipo di ritorno

Ogni Classe Java ha un Costruttore di default (implicito)/vuoto

E' possibile implementare uno o più Costruttori Custom; in tal caso il Costruttore di default viene «perso» e se vogliamo utilizzarlo dobbiamo riscriverlo esplicitamente; tutti i Costruttori (sia quello di default sia quello Custom servono per costruire Oggetti)

## JAVA BEAN

Classe contenente solo:

- variabili di istanza
- metodi di set e get (setters e getters)
- almeno il Costruttore Vuoto (Implicito)

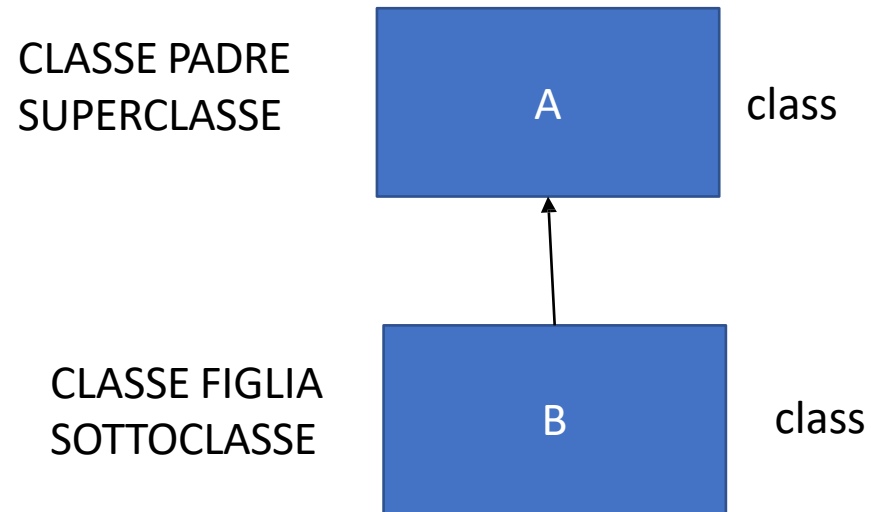
UN JAVA BEAN E' UNA CLASSE PRIVA DI METODI DI BUSINESS LOGIC

BUSINESS LOGIC = CODICE APPLICATIVO CHE SVOLGE OPERAZIONI DI:

- calcolo
- conversione dati
- controllo

## INHERITANCE

IN JAVA A PARTIRE DA UNA COMPONENTE APPLICATIVA E' POSSIBILE CREARE UNA O PIU' COMPONENTI APPLICATIVE FIGLIE



Una classe B figlia di una classe A eredita dalla Classe Padre tutti i suoi membri, ovvero tutte le variabili di istanza e tutti i metodi

Dal punto di vista sintattico una Classe per «rendersi» figlia di un'altra Classe deve usare la parola chiave `extends`

Esiste un assioma Java secondo il quale ogni Classe è figlia della Classe `Object` (API)

```
public class A {
```

```
    method1
```

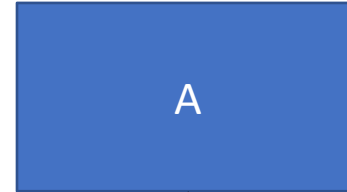
```
    method2
```

```
}
```

```
public class B extends A {
```

```
}
```

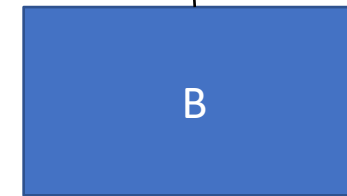
method1  
method2



class

IS-A

method1  
method2  
method3



class

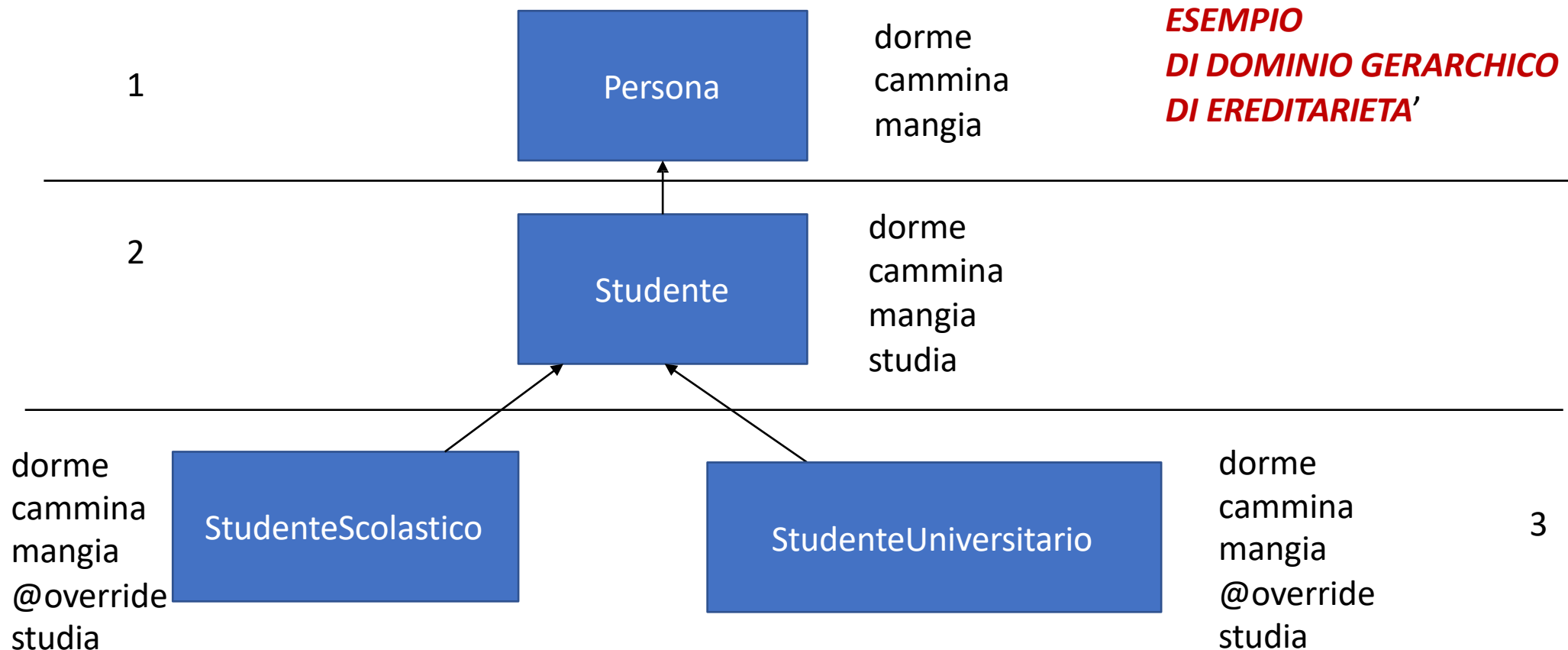
UNA CLASSE FIGLIA EREDITA DALLA CLASSE PADRE TUTTI I MEMBRI MA PUO' ANCHE :

- AGGIUNGERE NUOVI MEMBRI RISPETTO ALLA CLASSE PADRE
- ESEGUIRE L'OVERRIDE DI UNO O PIU' MEMBRI DELLA CLASSE PADRE

ESEGUIRE L'OVERRIDE DI UN METODO IN UNA CLASSE FIGLIA VUOL DIRE RISCRIVERE IL METODO DICHIARATO DAL PADRE MANTENENDONE INTATTA LA FIRMA E CAMBIANDONE IL CORPO



Si definisce **DOMINIO GERARCHICO DI EREDITARIETA'** L'INSIEME DELLE COMPONENTI APPLICATIVE FRA LE QUALI VIENE STABILITA UNA RELAZIONE DI PARENTELA  
UN DOMINIO GERARCHICO DI EREDITARIETA' PUO' ESSERE COMPOSTO DA N LIVELLI, AL MINIMO 2  
UN LIVELLO PUO' CONTENERE UNA O PIU' COMPONENTI APPLICATIVE FIGLIE DI UNA COMPONENTE APPLICATIVA COLLOCATA A LIVELLO SUPERIORE  
IN JAVA VIGE UN PRINCIPIO CHIAMATO PRINCIPIO DI TRANSITIVITA' DELL'EREDITARIETA' SECONDO IL QUALE OGNI COMPONENTE APPLICATIVA DI UN LIVELLO EREDITA TUTTI I MEMBRI DELLE COMPONENTI APPLICATIVE COLLOCATE NEI LIVELLI SUPERIORI



Una ANNOTATION è uno strumento sintattico rappresentato dal simbolo @seguito da una parola chiave

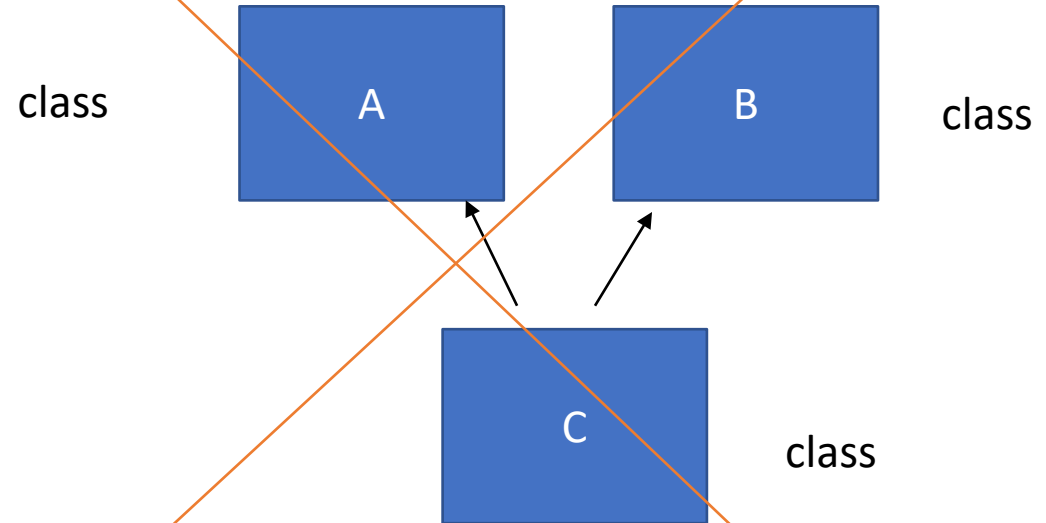
IN JAVA esistono due tipi di annotation:

- ANNOTATION di tipo commento (sono utili per chi legge il codice e non vengono interpretate in alcun modo a runtime)
- ANNOTATION di tipo metadato (sono delle vere e proprie istruzioni per il runtime environment che in base a tali istruzioni esegue un comportamento specifico)

@Override è una annotation di tipo commento (serve per notificare a chi legge il codice che stiamo effettuando l'override di un metodo, ma non è necessaria per la JVM)

IN JAVA UNA CLASSE PADRE PUO' AVERE PIU' CLASSI FIGLIE **MA NON E' VERO IL CONTRARIO**  
OVVERO UNA CLASSE FIGLIA NON PUO' AVERE PIU' CLASSI PADRE CONTEMPORANEAMENTE  
**IN JAVA NON ESISTE EREDITARIETA' MULTIPLA FRA CLASSI**

```
public class C extends A,B {  
}  
}
```



## IN JAVA ESISTONO 3 TIPOLOGIE DI COMPONENTI APPLICATIVE

- CLASSE CONCRETA (una classe che può contenere solo metodi concreti, ovvero metodi che hanno sia una firma sia un corpo)

Dal punto di vista sintattico per creare una classe concreta basta usare la parola chiave `class` senza altre parole chiave

- CLASSE ASTRATTA (una classe che può contenere sia metodi astratti sia metodi concreti; un metodo astratto è un metodo caratterizzato da firma senza corpo)

Dal punto di vista sintattico per creare una classe astratta occorre utilizzare la parola chiave `abstract` prima di `class`

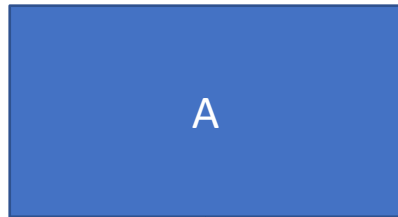
Una classe astratta non può essere MAI istanziata, ovvero non è possibile generare Oggetti a partire da una classe astratta. Una classe astratta può avere dei figli.

- INTERFACCIA (una componente applicativa che può contenere solo metodi astratti)

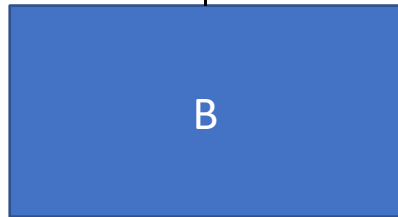
Dal punto di vista sintattico una Interfaccia si deve creare utilizzando la parola chiave `interface`

Una Interfaccia non può essere MAI istanziata, ovvero non è possibile generare Oggetti a partire da una Interfaccia

CLASSE CONCRETA



extends

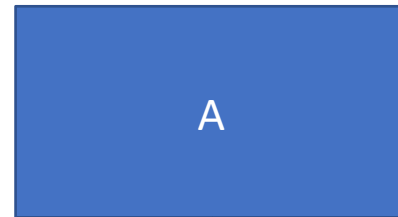


CLASSE CONCRETA

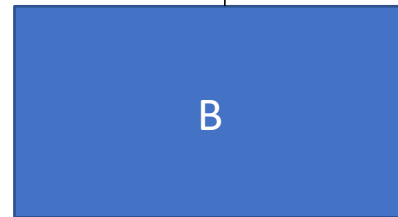
LA CLASSE FIGLIA PUO':

- mantenere intatti i metodi del padre
- fare l'override dei metodi del padre
- aggiungere metodi rispetto al padre

CLASSE ASTRATTA



extends



CLASSE CONCRETA

LA CLASSE FIGLIA PUO':

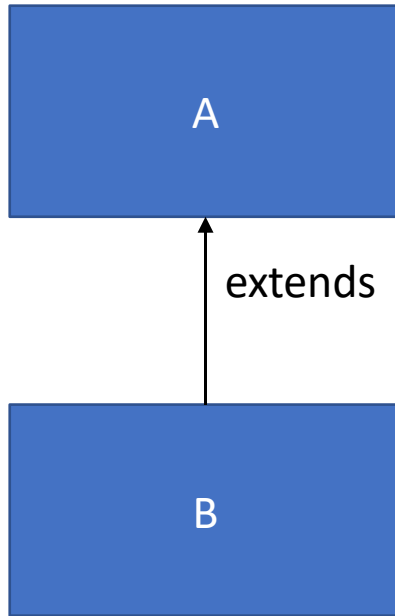
- mantenere intatti i metodi del padre se i metodi del padre sono concreti
- oppure farne l'override se lo desidera (senza esserne obbligata)
- aggiungere metodi rispetto al padre

LA CLASSE FIGLIA DEVE:

fare l'override di tutti i metodi astratti della Classe Padre

***CASISTICHE DI EREDITARIETA'(1)***

CLASSE ASTRATTA

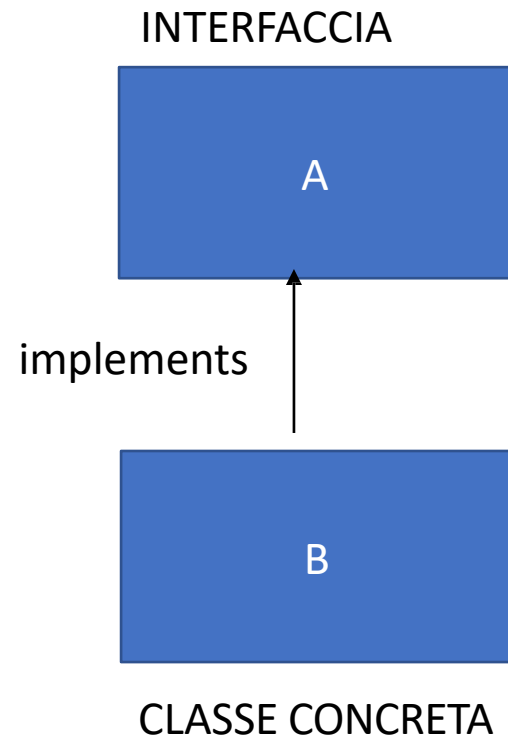


## ***CASISTICHE DI EREDITARIETA'(2)***

CLASSE ASTRATTA

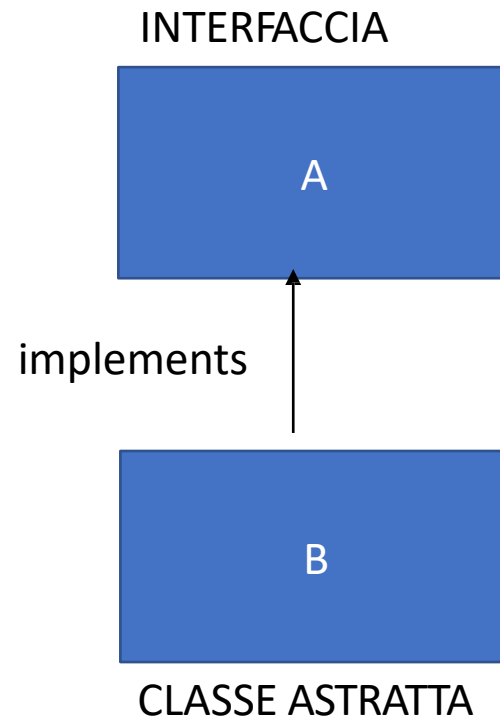
LA CLASSE FIGLIA PUO':

- mantenere intatti i metodi del padre (sia quelli concreti sia quelli astratti) oppure farne l'override se lo desidera (senza esserne obbligata)
- aggiungere metodi rispetto al padre



LA CLASSE FIGLIA DEVE:  
fare l'override di tutti i metodi del padre

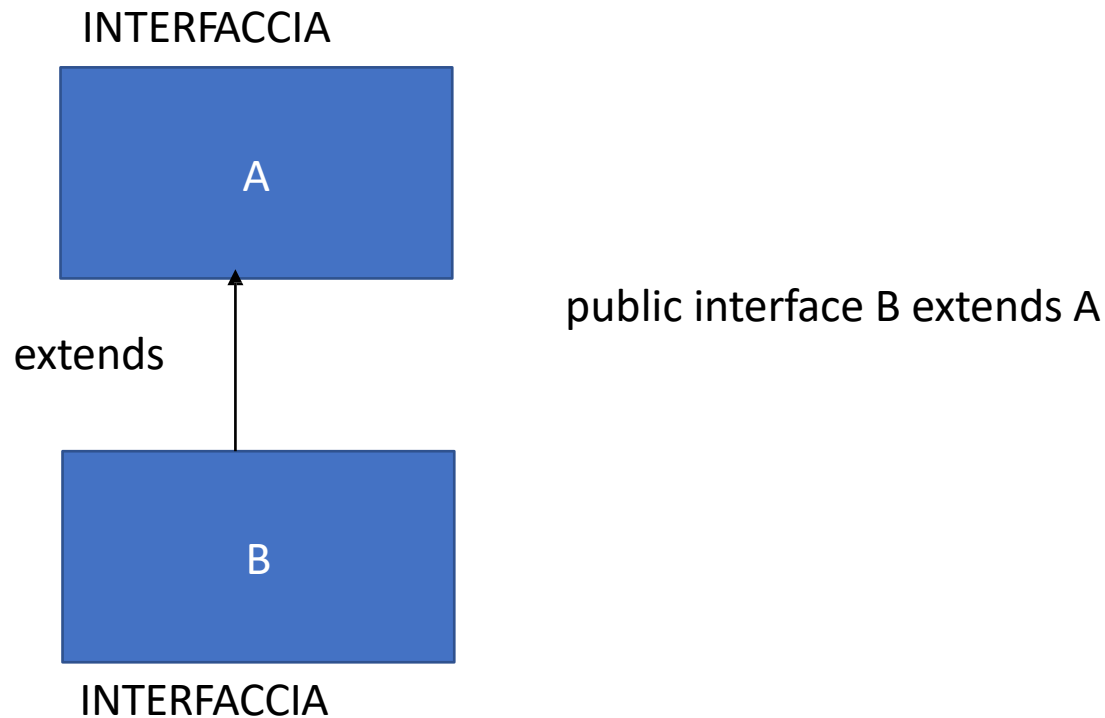
LA CLASSE FIGLIA PUO':  
aggiungere metodi rispetto al padre (solo concreti)



### ***CASISTICHE DI EREDITARIETA'(3)***

LA CLASSE FIGLIA PUO':

- fare l'override dei metodi del padre (senza esserne obbligata)
- aggiungere metodi rispetto al padre (sia astratti sia concreti)



### ***CASISTICHE DI EREDITARIETA'(4)***

L'INTERFACCIA FIGLIA PUO':  
aggiungere metodi rispetto al padre (solo astratti)

L'INTERFACCIA FIGLIA NON PUO' FARE l'override di nessun metodo dell'Intefaccia Padre

L' INTERFACCIA viene chiamata anche:

CONTRATTO (rappresenta un Contratto tra le Classi Figlie che intendono implementare in maniera diversa le funzionalità Comuni concordate tra le Classi stesse)

PROMESSA (rappresenta delle funzionalità che non vengono immediatamente concretizzate)

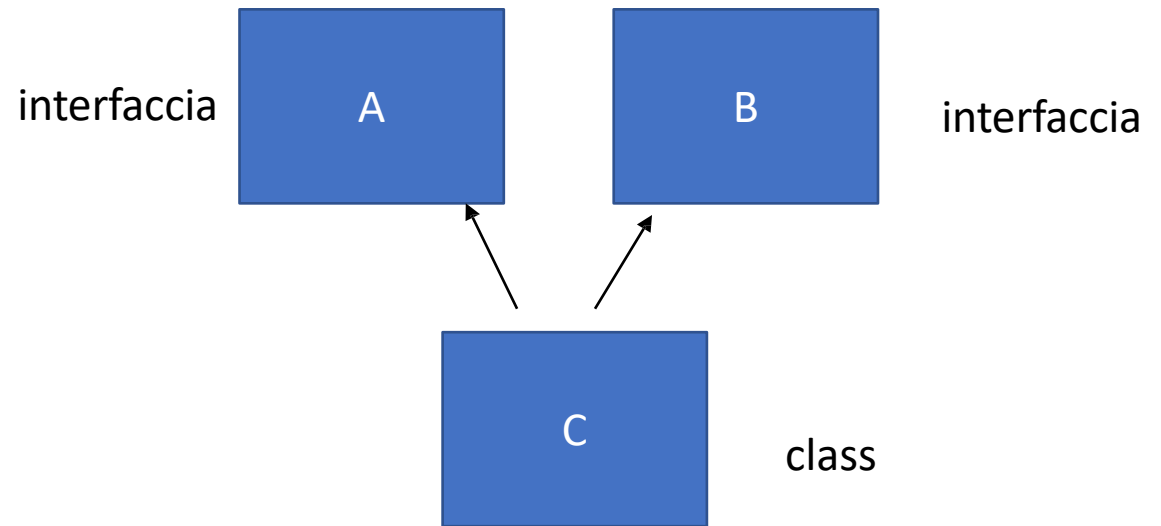


IN JAVA' PIU' INTERFACCE POSSONO ESSERE IMPLEMENTATE CONTEMPORANEAMENTE DA UNA CLASSE

**IN JAVA ESISTE EREDITARIETA' MULTIPLA FRA CLASSI E INTERFACCE**

**IN JAVA, DUNQUE, UNA CLASSE PUO' AVERE PIU' PADRI SE I PADRI SONO DELLE INTERFACCE**

```
public class C implements A,B {  
  
}
```



DIFATTI L'INTERFACCIA COME COMPONENTE APPLICATIVA

E' INTERESSANTE IN QUANTO «SUPERA» UN LIMITE DELLE CLASSI

CONSENTENDO EREDITARIETA' MULTIPLA (TRA CLASSI E CLASSI NON E' POSSIBILE)

ABSTRACT  
CLASS

Azienda

produce  
assume  
fattura

## ***ARCHITETTURA APPLICAZIONE Inheritance\_Abstract\_Class***

CONCRETE  
CLASS

AziendaMeccanica

CONCRETE  
CLASS

AziendaInformatica

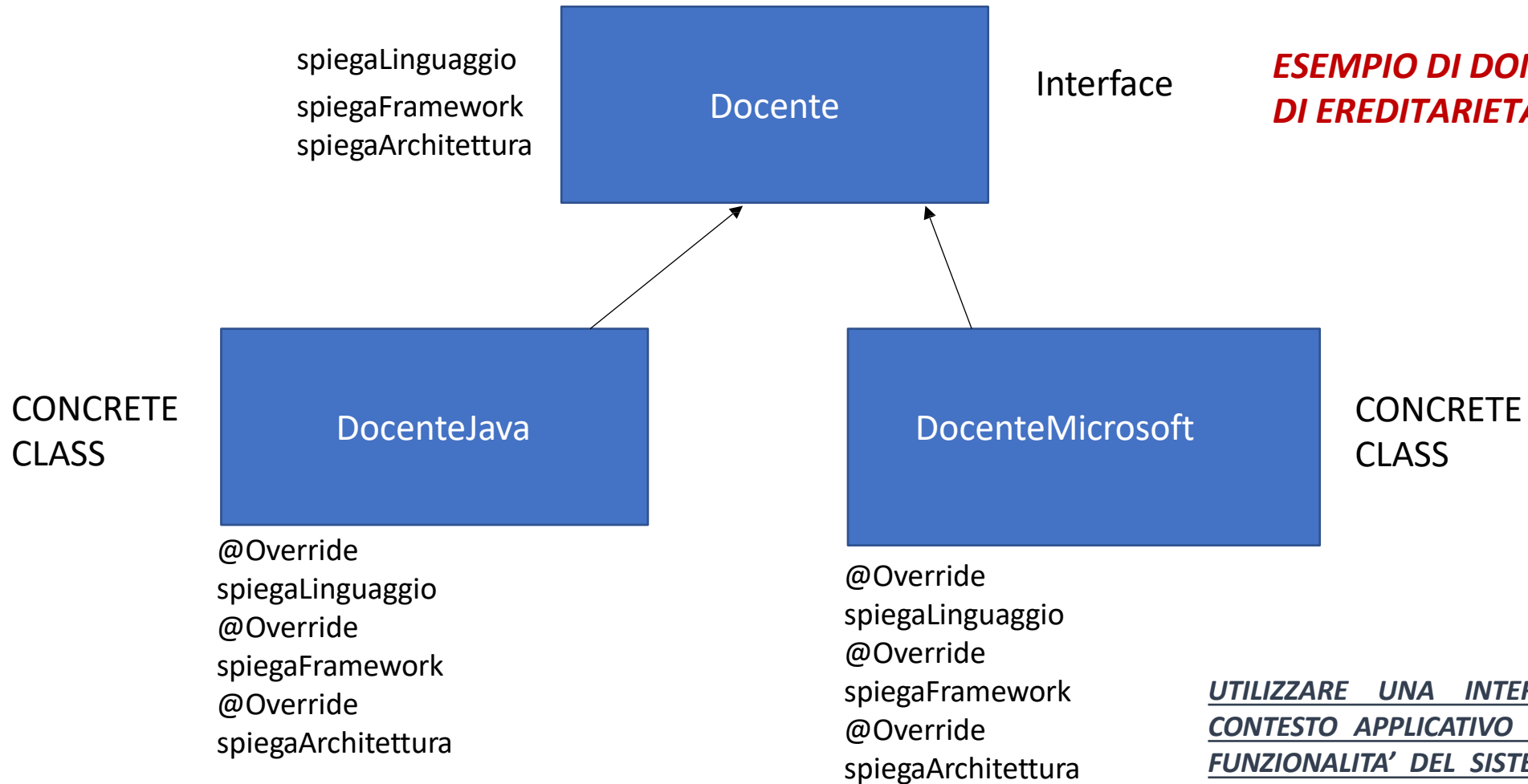
@Override

produce  
assume  
fattura

@Override

produce  
assume  
fattura

UTILIZZARE UNA CLASSE ASTRATTA ALL'INTERNO DI UN CONTESTO APPLICATIVO HA SENSO QUANDO ALCUNE FUNZIONALITA' SONO COMUNI ALLE COMPONENTI FIGLIE MA OGNI COMPONENTE FIGLIA LE IMPLEMENTERA' IN MANIERA DIFFERENTE E ALTRE FUNZIONALITA' SONO COMUNI MA IMPLEMENTATE DALLE COMPONENTI FIGLIE IN MANIERA ASSOLUTAMENTE IDENTICA



***ESEMPIO DI DOMINIO GERARCHICO  
DI EREDITARIETA' CON INTERFACCIA***

UTILIZZARE UNA INTERFACCIA ALL'INTERNO DI UN CONTESTO APPLICATIVO HA SENSO QUANDO TUTTE LE FUNZIONALITA' DEL SISTEMA SONO COMUNI AI FIGLI E TUTTI I FIGLI INTENDONO IMPLEMENTARLE IN MANIERA DIFFERENTE. IN TAL SENSO UNA INTERFACCIA PUO' ESSERE CONSIDERATA COME UN INDICE APPLICATIVO O COME UN CONTRATTO TRA I FIGLI CHE SI ACCORDANO SULL'USO DI METODI COMUNI CHE VENGONO ESPOSTI DALL'INTERFACCIA E POI CONCRETIZZATI IN MANIERA DIFFERENTE DA OGNI FIGLIO

In Java esistono due tipi di POLIMORFISMO:

- POLIMORFISMO DI OGGETTI
- POLIMORFISMO DI METODI (Overloading)

In Java reference e Oggetto referenziato possono avere lo stesso tipo

In Java una reference di tipo Padre può referenziare un Oggetto di tipo Figlio (POLIMORFISMO DI OGGETTI)

Cos'è il Polimorfismo di Oggetti (JAVA)?

Il Polimorfismo di Oggetti rappresenta la possibilità di far assumere ad un Padre la forma di uno dei suoi figli(a runtime) limitatamente ai metodi in comune

Modalità di implementazione del Polimorfismo di Oggetti

In Java è possibile dichiarare una reference di tipo Padre che referencia un Oggetto di tipo figlio (POLIMORFISMO DI OGGETTI)

In Java un metodo può assumere più forme all'interno della stessa Classe

All'interno di una stessa Classe è possibile dichiarare più metodi che hanno lo stesso nome purchè differiscano almeno per lista dei parametri ed eventualmente per tipo di ritorno

POLIMORFISMO DI METODI/OVERLOADING

PATTERN = UNA SOLUZIONE AD UN PROBLEMA RICORRENTE O UN INSIEME DI LINEE GUIDA PER LA REALIZZAZIONE DI UN OBIETTIVO IMPLEMENTATIVO

PATTERN **LOOSE COUPLING** (DEBOLE ACCOPPIAMENTO) = E' PREFERIBILE DISACCOPIARE UNA APPLICAZIONE INVOCANTE DA UN'APPLICAZIONE INVOCATA, OVVERO FARE IN MODO CHE L'APPLICAZIONE INVOCANTE POSSA COLLOQUIARE CON L'APPLICAZIONE INVOCATA SENZA CONOSCERNE I DETTAGLI IMPLEMENTATIVI

L'APPLICAZIONE DEL POLIMORFISMO DI OGGETTI JAVA CON REFERENCE DI TIPO INTERFACCIA PADRE E OGGETTO DI TIPO CLASSE CONCRETA FIGLIA RENDE POSSIBILE L'IMPLEMENTAZIONE INTRINSECA DEL PATTERN LOOSE COUPLING

In Java esistono due tipi

- Tipo primitivo

byte 8 bit

short 16 bit

int 32 bit

long 64 bit

float 32 bit

double 64 bit

char

boolean

- Tipo Object

In Java esistono 8 CLASSI API denominate CLASSI WRAPPER (o BRIDGE)

Byte

Short

Integer

Long

Float

Double

Character

Boolean

Tali Classi possono essere utilizzate per effettuare conversioni dati

int x = 67;

Integer y = new Integer(x);

String z = y.toString();

y

67



LA VERSIONE 5 DI JAVA HA INTRODOTTTO IL MECCANISMO DI **AUTOBOXING**

AUTOBOXING = CONVERSIONE IMPLICITA DI SOTTOTIPI PRIMITIVI IN CORRISPONDENTI WRAPPER

L'AUTOBOXING VIENE ESEGUITO BEHIND THE SCENES (DIETRO LE QUINTE) DALLA JVM

COME FUNZIONA L'AUTOBOXING?

L'ASSEGNAZIONE DIRETTA DI UNA VARIABILE DI TIPO WRAPPER AD UN VALORE (COME SE FOSSE UN SOTTOTIPO PRIMITIVO) COMPORTA UN'AZIONE DELLA JVM CHE ISTANZIA LA CLASSE WRAPPER ALLOCANDO UN OGGETTO CHE INVOLUCRA IL VALORE

Integer d = 8;

**ESEMPIO DI AUTOBOXING**

Integer d = new Integer(8)

A black arrow points from the text 'Integer d = 8;' to the text 'Integer d = new Integer(8)'. The arrow starts at the end of the first line and points diagonally down and to the right towards the start of the second line.

IN JAVA esistono 2 relazioni possibili tra componenti applicative

IS-A-→>> RELAZIONE UNIDIREZIONALE FRA FIGLIO E PADRE (IL FIGLIO E' TUTTO CIO' CHE E' IL PADRE MA PUO' ANCHE ESSERE PIU' DEL PADRE, OVVERO SPECIALIZZARE IL PADRE AGGIUNGENDO DELLE FUNZIONALITA'. IL CONTRARIO NON E' VERO. IL FIGLIO NON E' UN PADRE, NEL SENSO CHE IL PADRE NON CONOSCE I COMPORTAMENTI DIFFERENZIATI CHE IL FIGLIO ASSUME (COME AD ESEMPIO GLI OVERRIDE E L'AGGIUNTA DI METODI))

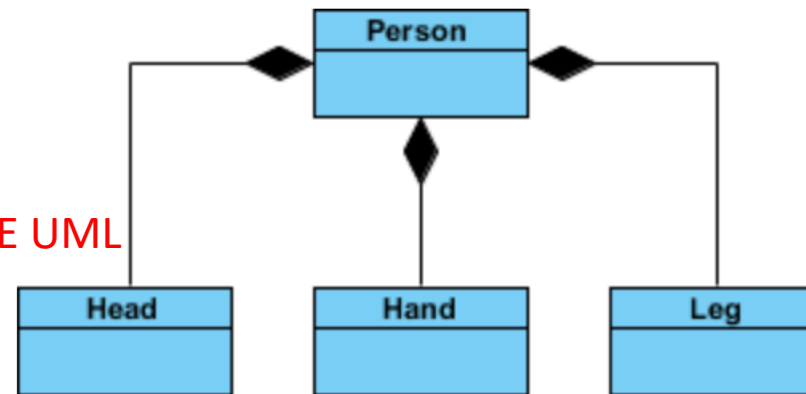
HAS-A --→> SI TRADUCE NELLA DICHIARAZIONE, ALL'INTERNO DI UNA CLASSE, DI UNA VARIABILE DEL TIPO DI UN'ALTRA CLASSE. TALE RELAZIONE ESULA DALL'EREDITARIETA'

N.B.: COSA SI PUO' PASSARE IN INPUT AD UN METODO JAVA

Alla lista dei parametri di un metodo Java può essere passato in input:

- Un elemento di tipo Primitivo
- Una reference ad un Object
- Una funzione

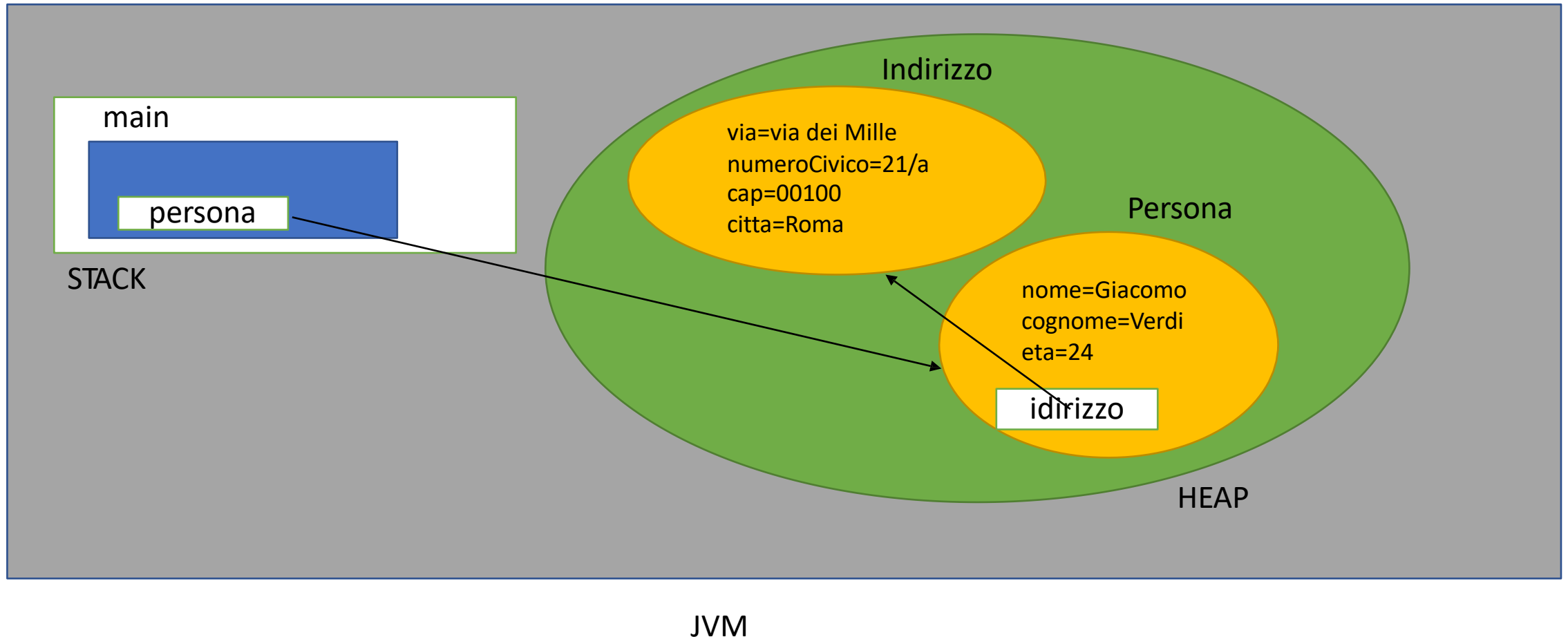
ESEMPIO DI RELAZIONE HAS-A  
RAPPRESENTATA TRAMITE NOTAZIONE UML





Una reference Java può essere collocata in:  
stack(se dichiariamo e inizializziamo una reference come variabile locale)  
heap(se viene passata come argomento ad un costruttore)

### ESEMPIO DI RELAZIONE HAS-A A RUNTIME



STRUTTURA DATI = STRUMENTO APPLICATIVO CHE CONSENTE DI ARCHIVIARE PIU' INFORMAZIONI CONTEMPORANEAMENTE

JAVA METTE A DISPOSIZIONE DUE TIPOLOGIE DI STRUTTURE DATI:

- STRUTTURA DATI STATICA (LA CUI DIMENSIONE DEVE ESSERE FISSATA IN FASE DI INIZIALIZZAZIONE E NON PUO' ESSERE MAI VARIATA)  
*Array*
- STRUTTURA DATI DINAMICA (LA CUI DIMENSIONE NON DEVE ESSERE NECESSARIAMENTE FISSATA IN FASE DI INIZIALIZZAZIONE E PUO' ESSERE VARIATA)  
*ArrayList, HashMap, HashSet*

UN ARRAY IN JAVA E' UNA STRUTTURA DATI CON LE SEGUENTI CARATTERISTICHE:

- STATICA (DIMENSIONE DA FISSARE INIZIALMENTE SENZA POSSIBILITA' DI VARIAZIONE SUCCESSIVA)
- PUO' CONTENERE ELEMENTI DI TIPO PRIMITIVO E REFERENCE AD OGGETTI
- OMOGENEA (PUO' CONTENERE SOLO ELEMENTI DELLO STESSO TIPO)
- INDICIZZATA E ORDINATA PER INDICE
- AMMETTE DUPLICATI

UN' ARRAYLIST IN JAVA E' UNA STRUTTURA DATI CON LE SEGUENTI CARATTERISTICHE:

- DINAMICA (NON OCCORRE NECESSARIAMENTE FISSARE LA DIMENSIONE IN FASE DI DICHIARAZIONE E LA DIMENSIONE PUO' ESSERE ACCRESCIUTA O DIMINUITA DINAMICAMENTE)
- PUO' CONTENERE SOLO REFERENCE AD OGGETTI
- SIA OMOGENEA CHE ETEREOGENEA (PUO' CONTENERE ELEMENTI SIA DELLO STESSO TIPO SIA DI DIVERSO TIPO)
- INDICIZZATA E ORDINATA PER INDICE
- AMMETTE DUPLICATI

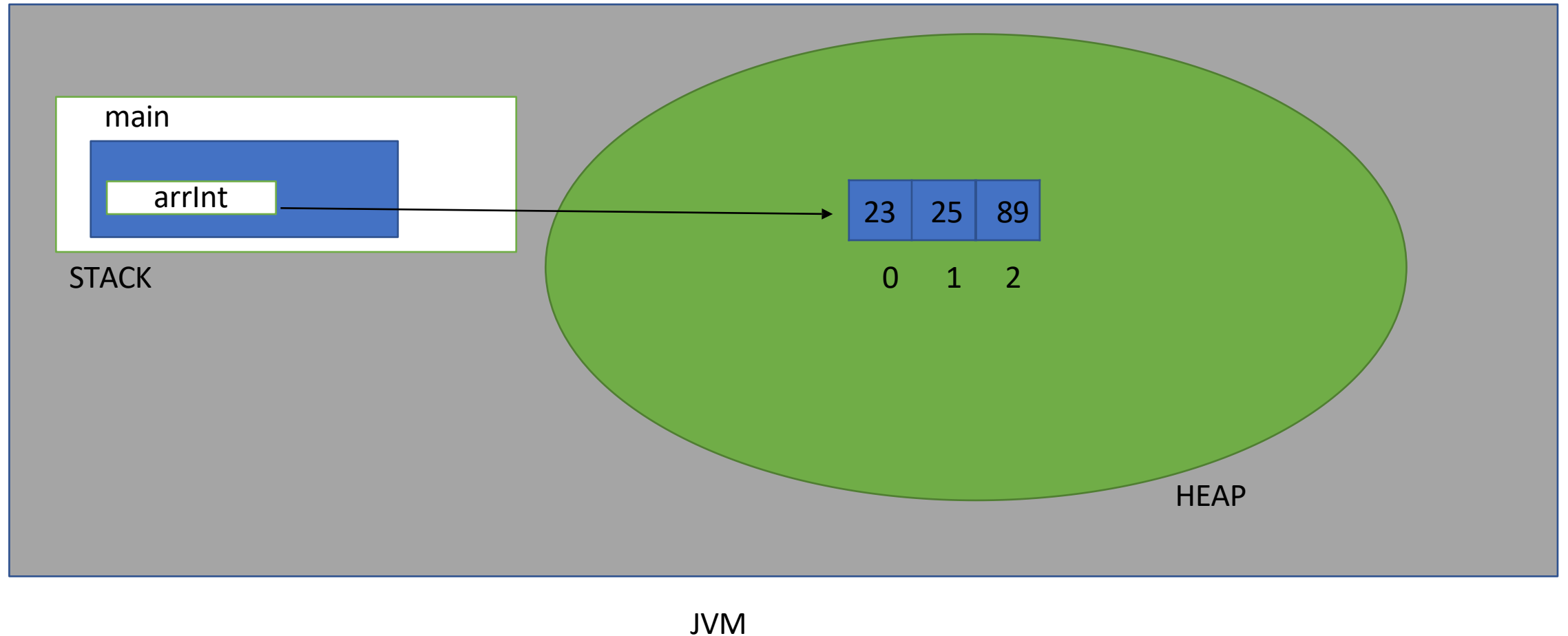
UN' HASHSET IN JAVA E' UNA STRUTTURA DATI CON LE SEGUENTI CARATTERISTICHE:

- DINAMICA (NON OCCORRE NECESSARIAMENTE FISSARE LA DIMENSIONE IN FASE DI DICHIARAZIONE E LA DIMENSIONE PUO' ESSERE ACCRESCIUTA O DIMINUITA DINAMICAMENTE)
- PUO' CONTENERE SOLO REFERENCE AD OGGETTI
- SIA OMOGENEA CHE ETEREOGENEA (PUO' CONTENERE ELEMENTI SIA DELLO STESSO TIPO SIA DI DIVERSO TIPO)
- NON INDICIZZATA NE' ORDINATA PER INDICE
- NON AMMETTE DUPLICATI (INVOCA I METODI HASHCODE ED EQUALS CHE DEVONO ESSERE OVERRIDATI PER CONTROLLARE OGGETTI UGUALI)

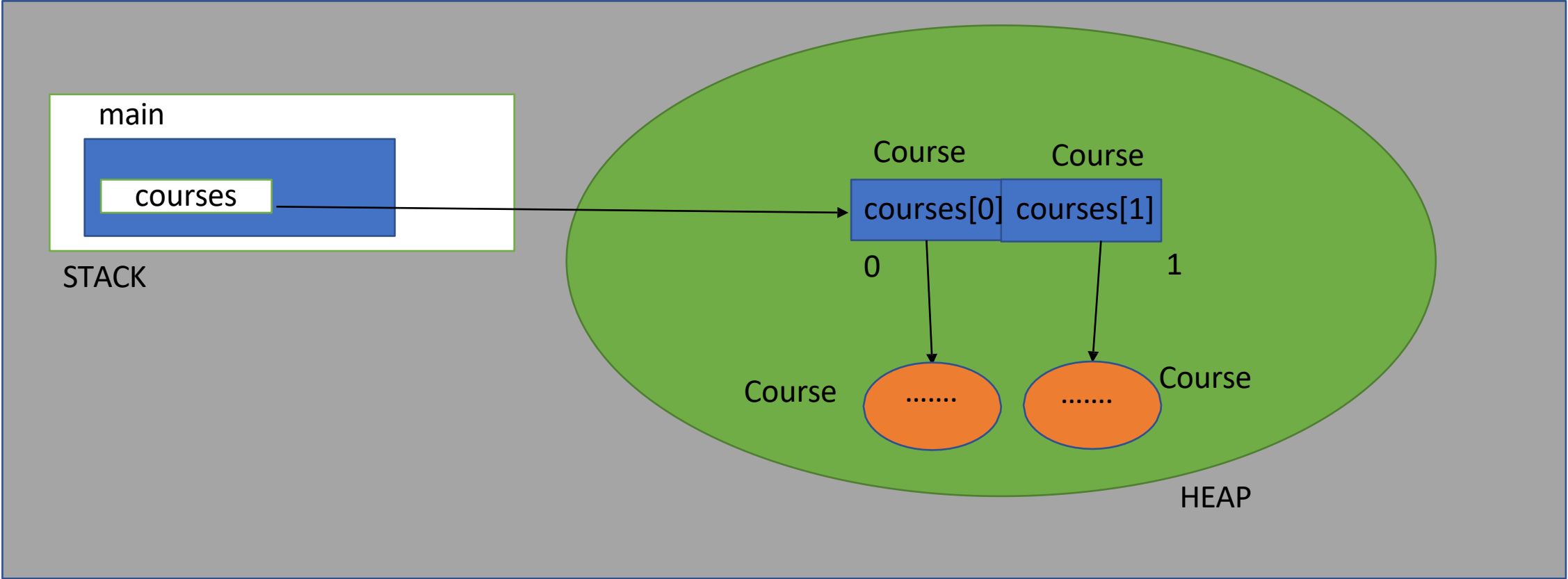
UN' HASHMAP IN JAVA E' UNA STRUTTURA DATI CON LE SEGUENTI CARATTERISTICHE:

- DINAMICA (NON OCCORRE NECESSARIAMENTE FISSARE LA DIMENSIONE IN FASE DI DICHIARAZIONE E LA DIMENSIONE PUO' ESSERE ACCRESCIUTA O DIMINUITA DINAMICAMENTE)
- PUO' CONTENERE SOLO REFERENCE AD OGGETTI
- SIA OMOGENEA CHE ETEREOGENEA (PUO' CONTENERE ELEMENTI SIA DELLO STESSO TIPO SIA DI DIVERSO TIPO)
- INDICIZZATA MA NON ORDINATA (L'INDICE VIENE APPLICATO DAL PROGRAMMATORE TRAMITE LA CHIAVE)
- AMMETTE DUPLICATI

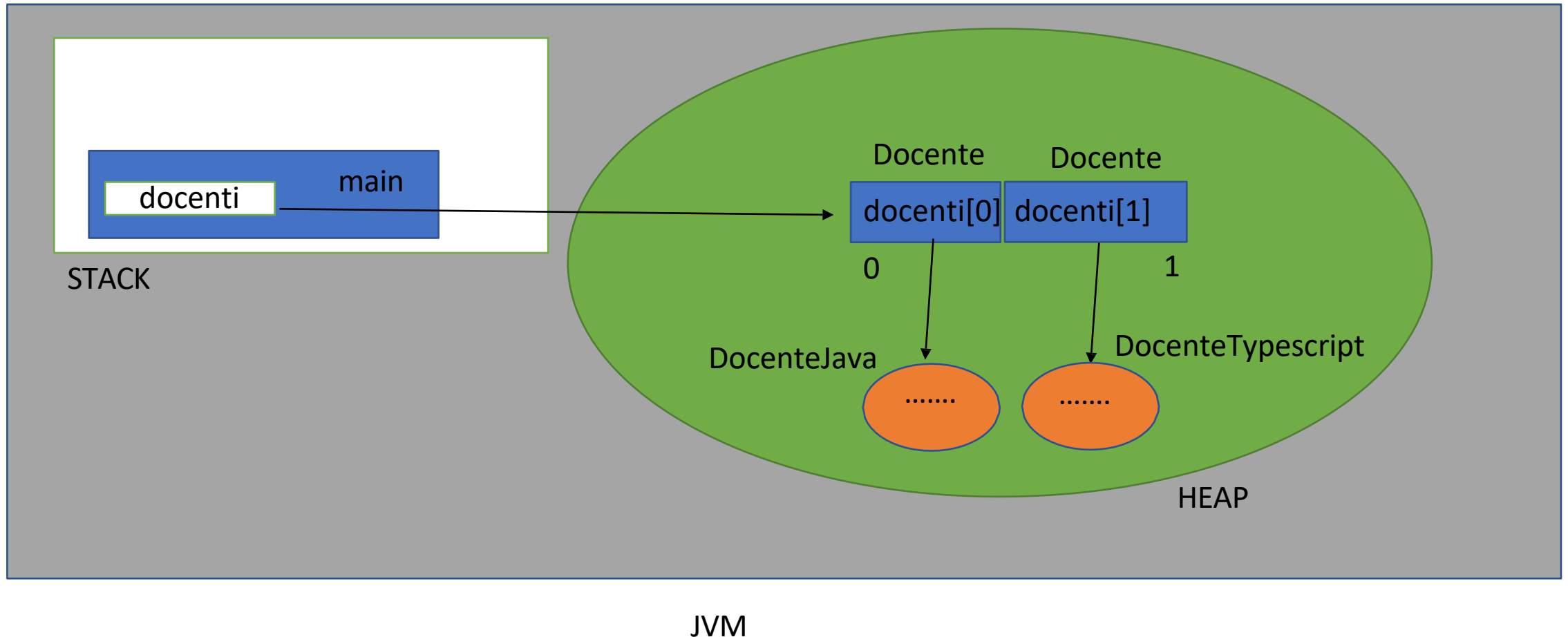
Fotografia a Runtime di un Array contenente elementi di tipo int



Fotografia a Runtime di un Array contenente reference di tipo Course

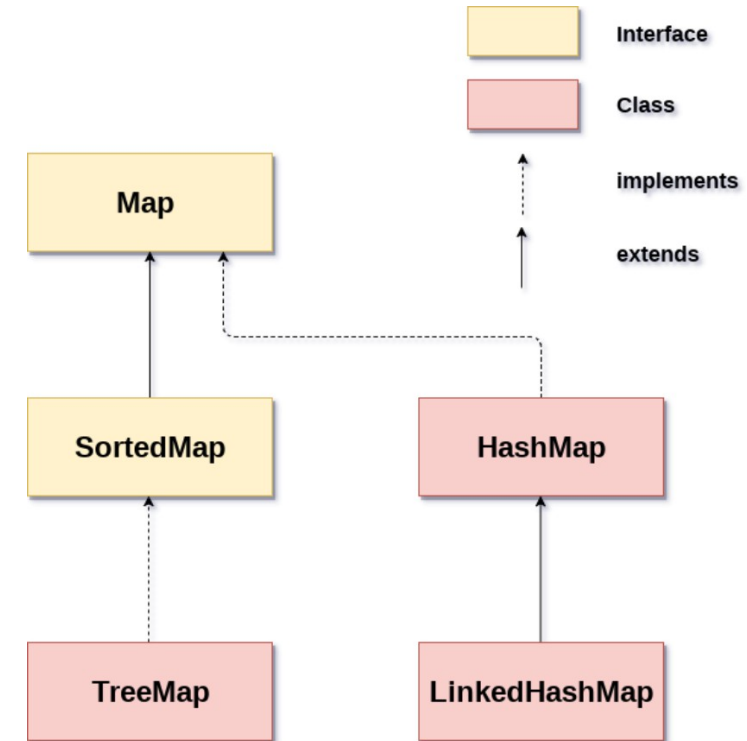
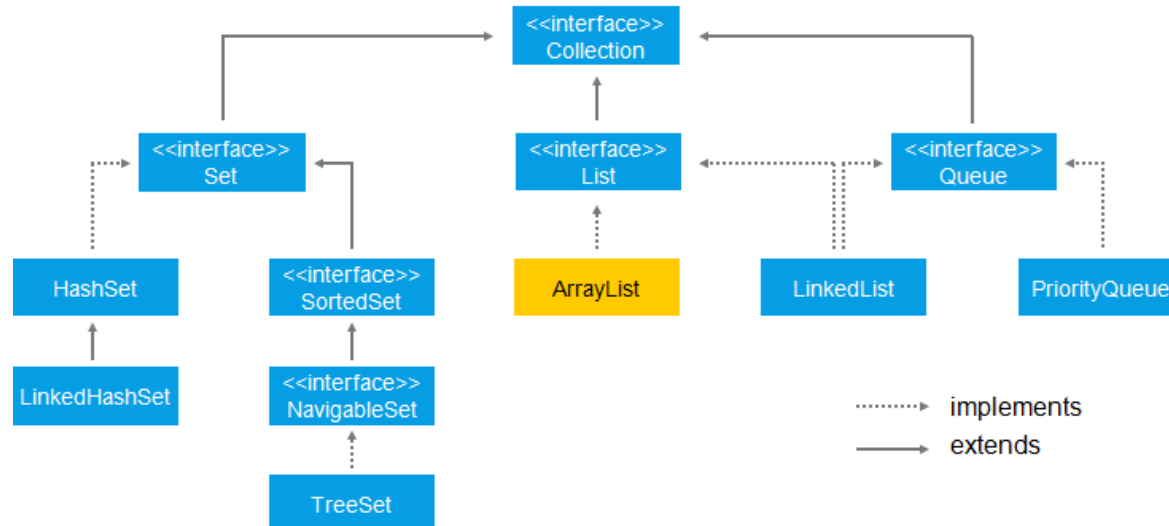


GRAZIE AL POLIMORFISMO E' POSSIBILE LAVORARE CON OGGETTI DI DIVERSO TIPO REFERENZIATI DA REFERENCEDELLO STESSO TIPO (PADRE), RENDENDO DI FATTO POLIMORFICA UNA STRUTTURA DATI CHE NON LO E' DI DEFAULT, SENZA VIOLARNE I VINCOLI

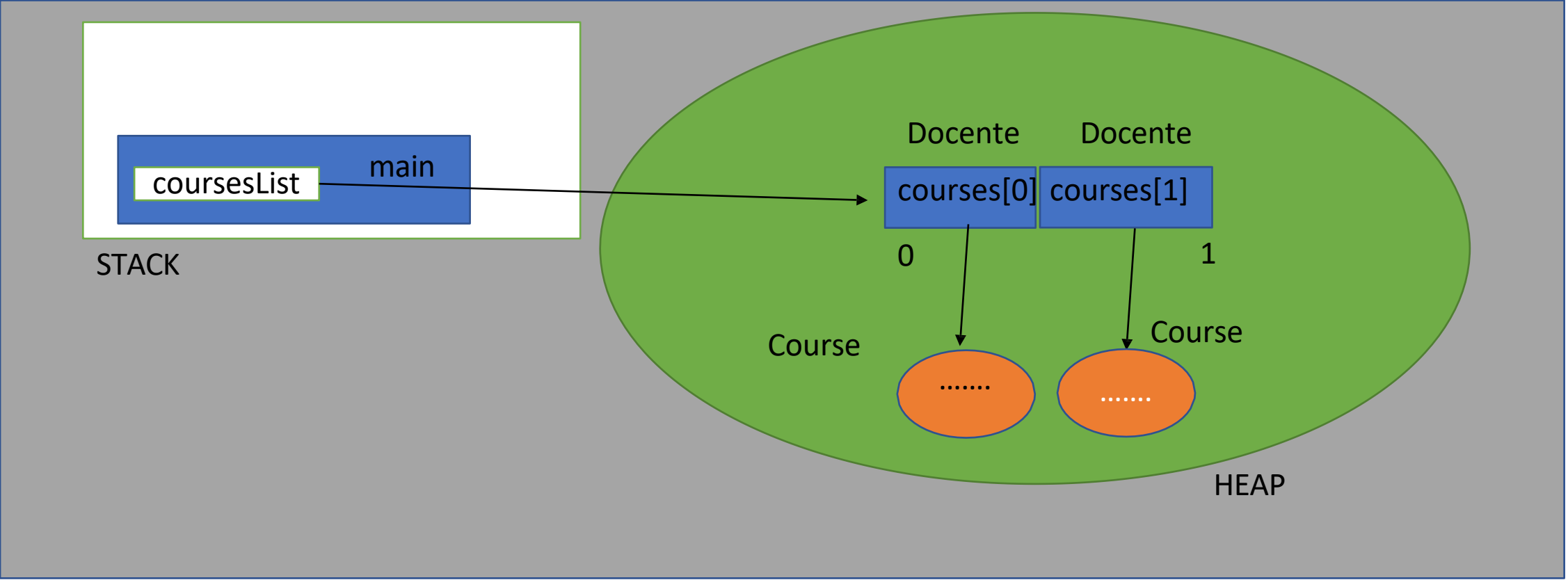


Alcune Strutture Dati Dinamiche Java (come ad esempio ArrayList ed HashMap) vengono denominate anche COLLECTIONS. Per dichiarare ed inizializzare tali strutture dati dinamiche occorre utilizzare API figlie di una INTERFACCIA chiamata Collection (API). HashMap è anch'essa una Struttura Dati Dinamica ma non rientra nella gerarchia di ereditarietà delle Collection, per cui formalmente non è una Collection.

## Collection Interface Hierarchy



Fotografia a Runtime di un Array polimorfico



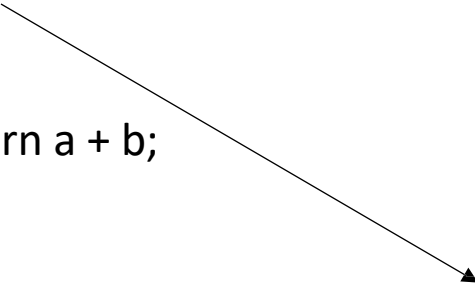


SECONDO IL PARADIGMA FUNZIONALE AD UNA PROPRIETA' (VARIABILE) E' POSSIBILE ASSEGNARE LA DICHIARAZIONE DI UNA FUNZIONE LA VARIABILE ASSUMERA' SUCCESSIVAMENTE LA FORMA DELLA FUNZIONE STESSA E SI POTRA' INVOCARE LA FUNZIONE TRAMITE IL NOME DELLA VARIABILE

```
let x = function somma(a,b){
```

```
  return a + b;
```

```
}
```



x(7,9)

ANONYMOUS FUNCTION

```
let somma = (a,b) => return a +b;
```



somma(7,9)

SECONDO IL PARADIGMA FUNZIONALE E' POSSIBILE PASSARE IN INPUT AD UN METODO UNA FUNZIONE  
SECONDO IL PARADIGMA FUNZIONALE E' POSSIBILE PASSARE IN INPUT AD UNA FUNZIONE UN'ALTRA FUNZIONE

```
daysList.forEach( (day) -> System.out.println(day) );
```

FUNZIONE ANONIMA



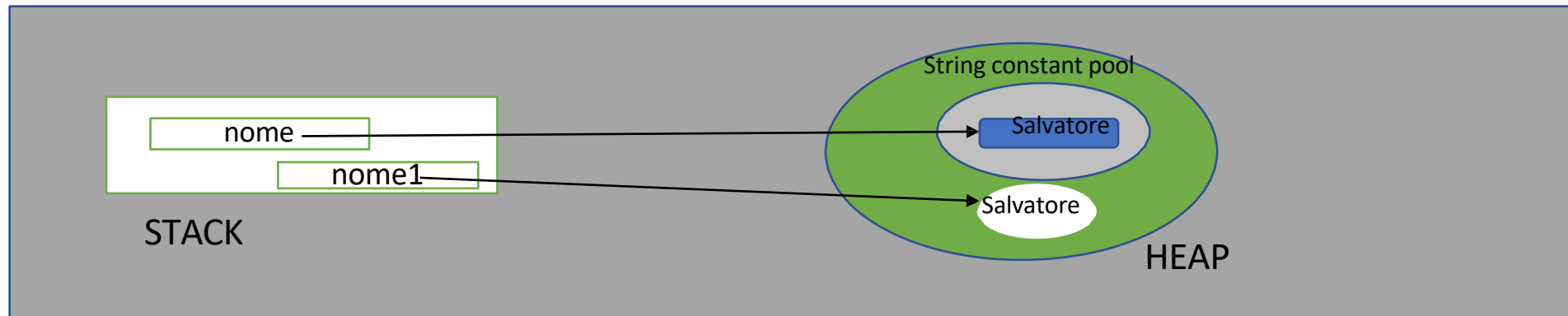
metodo

String è l'unico Oggetto Java che può essere utilizzato in due differenti vesti:

- Literal (equiparazione di String ad un tipo primitivo; è possibile eseguire un'assegnazione diretta)
- Classic Object

literal

```
String nome = «Salvatore»;
```

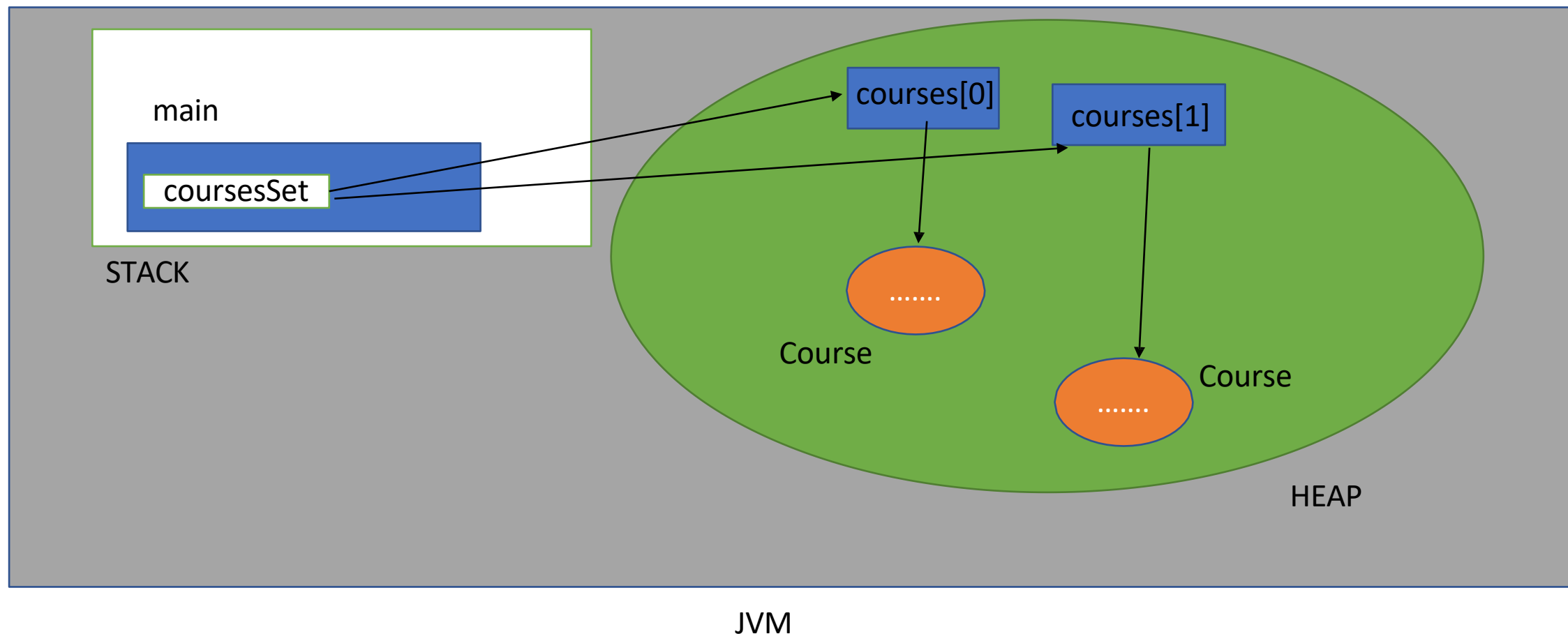


Classic Object

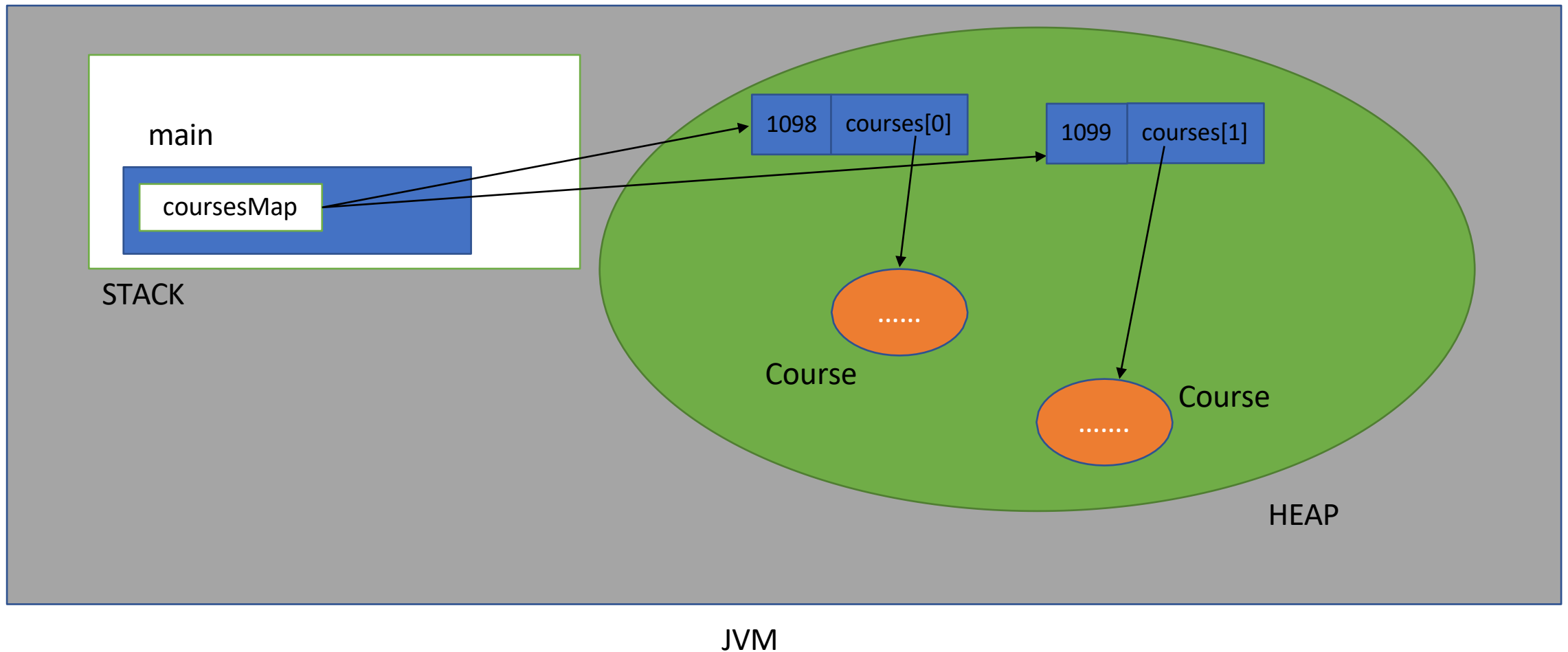
```
String nome1 = new String(«Salvatore»);
```

UTILIZZARE String come literal è conveniente per le performance in quanto le locazioni di memoria dello String constant Pool occupano meno spazio rispetto agli Oggetti tradizionali

Fotografia a Runtime di un'HashSet di reference ad Oggetti di tipo Course



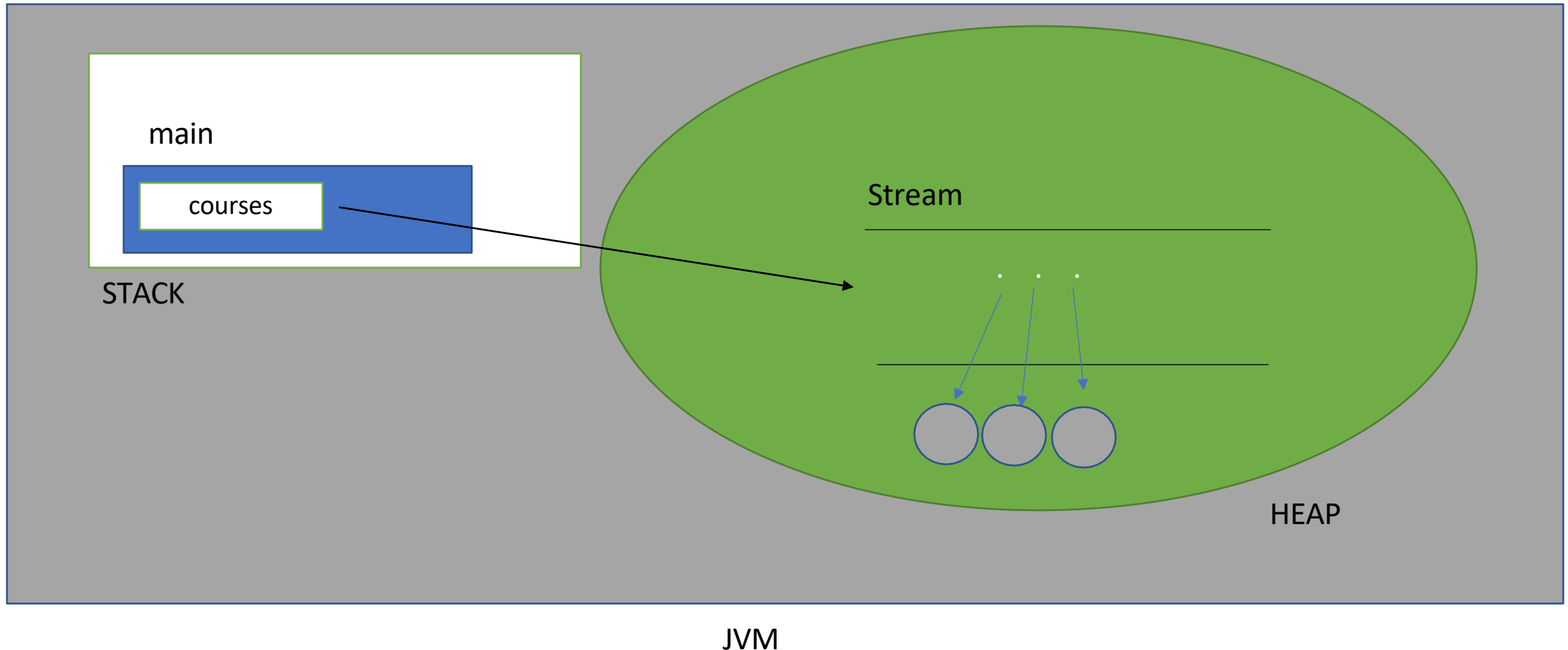
Fotografia a Runtime di un'HashMap di reference ad Oggetti di tipo Course



Stream è una Struttura Dati Dinamica introdotta da Java 8, con le seguenti caratteristiche:

- STRUTTURA DATI DINAMICA (FLUSSO DI DATI DINAMICO)
- SIA OMOGENEA CHE ETEROGENEA
- PUO' CONTENERE SOLO REFERENCE AD OGGETTI
- ORDINATA MA NON INDICIZZATA
- AMMETTE DUPLICATI

UNO STREAM E' CONVENIENTE DAL PUNTO DI VISTA COMPUTAZIONALE RISPETTO ALLA COLLECTION IN QUANTO NON VENGONO ALLOCATE LOCAZIONI DI MEMORIA. INOLTRE L'API Stream ESPONE UNA SERIE DI METODI FUNZIONALI CHIAMATI OPERATORI MOLTO EFFICACI PER EFFETTUARE OPERAZIONI DI UNA CERTA COMPLESSITA', CHE RICHIEDEREBBERO INVECE UN EFFORT APPLICATIVO ELEVATO PER LE COLLECTION.



In Java esiste una Classe speciale chiamata Enum

Una Classe Enum consente di creare un tipo al quale possono essere assegnate informazioni immutabili per creare una struttura dati fissa in termini di informazioni contenute

UN'ECCEZIONE JAVA RAPPRESENTA UNA ANOMALIA VERIFICABILE SOLO ED ESCLUSIVAMENTE A RUNTIME DALLA JVM

ESISTONO DUE TIPI DI ECCEZIONI:

1. ECCEZIONI API

2. ECCEZIONI UNCHECKED

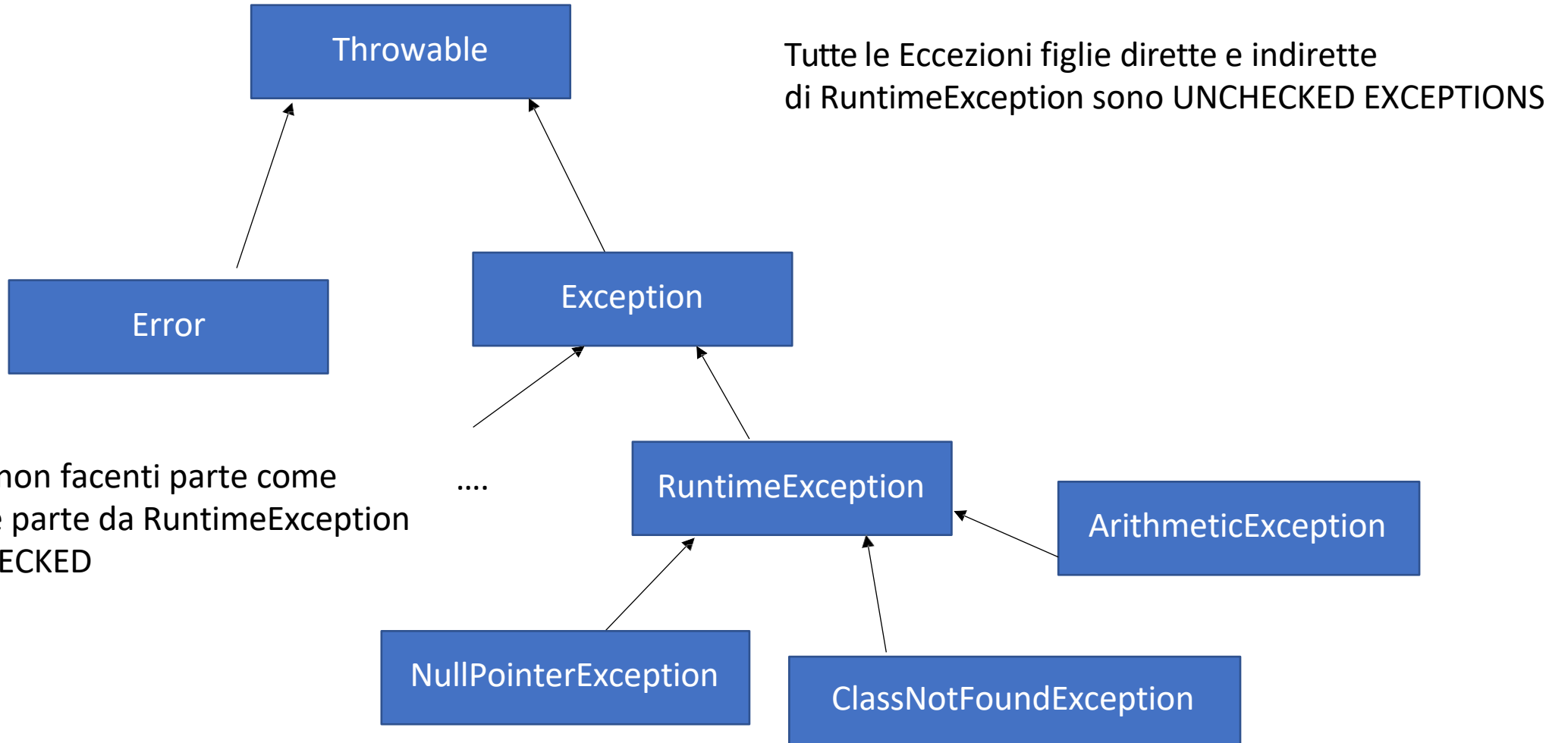
ECCEZIONI CHE VENGONO AUTONOMAMENTE RESTITUITE DALLA JVM NEL CASO IN CUI VENGANO RILEVATE, SENZA CHE LO SVILUPPATORE SCRIVA DEL CODICE PER CHIEDERE DI RESTITUIRLA

2. ECCEZIONI CHECKED

ECCEZIONE CHE DEVE ESSERE GESTITA APPLICATIVAMENTE DALLO SVILUPPATORE PER CHIEDERE ALLA JVM DI RESTITUIRLA NEL CASO DI UN ERRORE DI UN DETERMINATO TIPO

2. ECCEZIONI CUSTOM

## ESTRATTO DEL CONTESTO GERARCHICO DI EREDITARIETA' DELLE ECCEZIONI API





```
try {
```

```
    codice critico
```

```
}
```

```
catch(Exception ex){
```

```
    ex.printStackTrace();
```

```
}
```

```
finally{
```

```
    codice sempre eseguito
```

```
}
```

throw

PER GESTIRE APPLICATIVAMENTE LE ECCEZIONI CHECKED  
SI POSSONO USARE IL COSTRUTTO try-catch o la parola chiave throw

## STEP PER L'IMPLEMENTAZIONE DI UNA ECCEZIONE CUSTOM

1. CREAZIONE DI UNA CLASSE CHE ESTENDE EXCEPTION
2. UTILIZZO UNA MODALITA' PER LA GESTIONE DEL MESSAGGIO CUSTOM DELL'ECCEZIONE come ad esempio l'invocazione del supercostruttore della Classe Exception
3. IMPLEMENTAZIONE DEI METODI CHE GESTISCONO L'ECCEZIONE CUSTOM CHIEDENDO ALLA JVM DI RILEVARLA TRAMITE I COSTRUTTI MESSI A DISPOSIZIONE DAL LINGUAGGIO COME il try/catch o la parola Throw + new + costruttore della Custom Exception

Esiste un ASSIOMA Java secondo il quale un METODO invocante un METODO CHE ESEGUE IL THROW  
Di UNA ECCEZIONE DEVE LANCIARE LA STESSA ECCEZIONE INSERENDO NELLA FIRMA LA PAROLA CHIAVE  
throws

static è una parola chiave che si può associare:  
a una variabile  
a un metodo  
a una classe

In Java ogni variabile è caratterizzata da:  
un tipo ed una tipologia

Il tipo può essere:

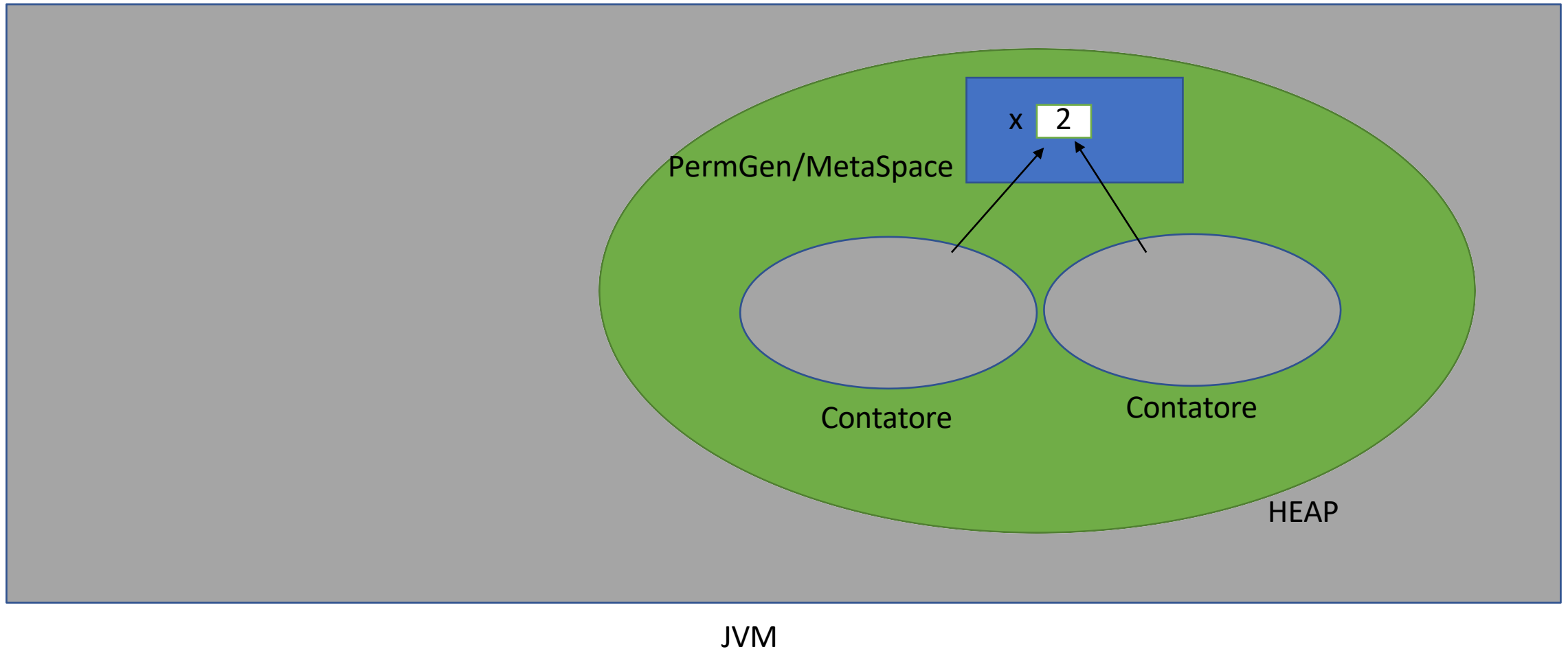
- Primitivo
- Object

La tipologia può essere:

- di istanza (variabile dichiarata esternamente a qualsiasi metodo, all'interno di un corpo di una classe, ed è una variabile di cui ogni istanza avrà una propria copia quando verrà allocata nell'Heap)
- locale (variabile dichiarata ed inizializzata all'interno di un metodo)
- di classe (variabile alla quale viene associata la parola chiave static, viene dichiarata esternamente ai metodi all'interno del corpo della classe, ed è una variabile condivisa da tutte le istanze della stessa classe)

LE VARIABILI DI CLASSE VIVONO ALL'INTERNO DI UNA PORZIONE DI MEMORIA SPECIALE DELL'HEAP:  
PermGen(pre Java 8)/Metaspace (da Java 8 in poi)

Anche i metodi static vengono allocati all'interno del PermGen/Metaspace e sono parecchio efficienti dal punto di vista computazionale in quanto vengono allocati solo alla prima invocazione, e poi deallocati alla fine dell'esecuzione dell'applicazione



UN METODO STATIC VIENE CHIAMATO ANCHE **METODO DI CLASSE** (MENTRE I METODI NON STATIC VENGONO DEFINITI **METODI DI ISTANZA**).

UN METODO DI ISTANZA COMPORTA SEMPRE DUE OPERAZIONI COMPUTAZIONALI PER OGNI INVOCAZIONE (ALLOCAZIONE E DEALLOCAZIONE).

$N \text{ INVOCAZIONI} = 2 \times N \text{ OPERAZIONI COMPUTAZIONALI}$

UN METODO DI CLASSE INVECE VIENE ALLOCATO NELLO STACK SOLO DURANTE LA PRIMA INVOCAZIONE, E VI PERMANE FINO ALLA FINE DELLA DURATA DELL'INTERA APPLICAZIONE.

$N \text{ INVOCAZIONI} = 2 \text{ OPERAZIONI COMPUTAZIONALI}$

I METODI DI CLASSE SONO ESTREMAMENTE PIU' EFFICIENTI RISPETTO A QUELLI DI ISTANZA

LA BEST PRACTICE COMPUNQUE NON E' QUELLA DI DICHIARARE TUTTI I METODI STATIC, IN QUANTO PERMANENDO MOLTI METODI ALL'INTERNO DELLO STACK ESAUTOREREBBERO TUTTO LO SPAZIO, MA QUELLA DI DICHIARARE STATIC I METODI PIU' FREQUENTEMENTE INVOCATI ALL'INTERNO DELL'APPLICAZIONE.

UNA CLASSE STATIC E' UNA CLASSE CHE CONTIENE SOLO MEMBRI STATIC

La parola chiave **final** può essere accostata a:

una variabile-----→>>>> COSTANTE (IMMUTABILE)

un metodo-----→>>>> un metodo final non può essere sovrascritto da nessuna eventuale Classe Figlia

classe-----→>>>> CLASSE CHE NON PUO' AVERE FIGLI

UNA CLASSE final non può avere figli ma può essere istanziata

Una CLASSE abstract può avere figli ma non può essere istanziata  
abstract e final non possono essere utilizzate insieme

UNA INTERFACCIA FUNZIONALE E' UNA COMPONENTE APPLICATIVA CHE DEVE ESSERE ANNOTATA CON ANNOTATION @FunctionalInterface

Una Interfaccia Funzionale ha le seguenti caratteristiche:

- DEVE CONTENERE UNO ED UN SOLO METODO ASTRATTO
- PUO' CONTENERE UNO O PIU' METODI CONCRETI (non static) E TALI METODI DEVONO AVERE NELLA FIRMA LA PAROLA CHIAVE default
- PUO' CONTENERE UNO O PIU' METODI CONCRETI static

```
@FunctionalInterface
public interface Name {

    public void method1();

    public default void method2(){
        .....
    }

    public static void method3(){
    }

}
```

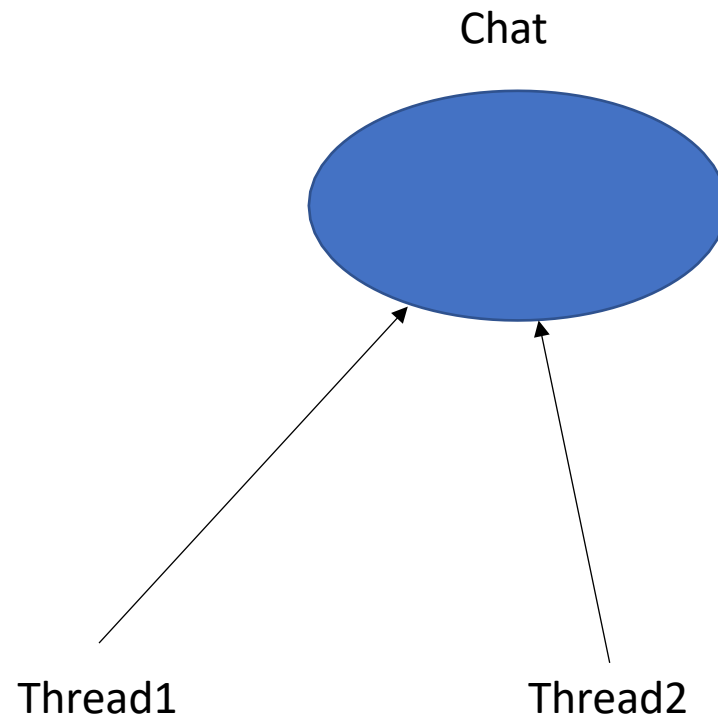


PARADIGMA FUNZIONALE

Java supporta la possibilità di implementare la PROGRAMMAZIONE CONCORRENTE

PROGRAMMAZIONE CONCORRENTE VUOL DIRE gestire più PROCESSI concorrenti che intendono accedere ad una risorsa condivisa, ovvero gestire più processi che possono richiedere anche contemporaneamente l'utilizzo di una risorsa condivisa

In generale un Processo è rappresentato da un'azione umana o applicativa



ESEMPIO DI UNA CHAT (RISORSA CONDIVISA  
DA PIU' PROCESSI/THREAD)

**JAVA SE FORNISCE UNA API DI NOME  
THREAD CHE CONSENTE DI IMPLEMENTARE  
PROCESSI CONCORRENTI**



LA JVM HA DUE PROCESSI NATIVI:

- GC (spazzatura della memoria)
- Thread Monitor

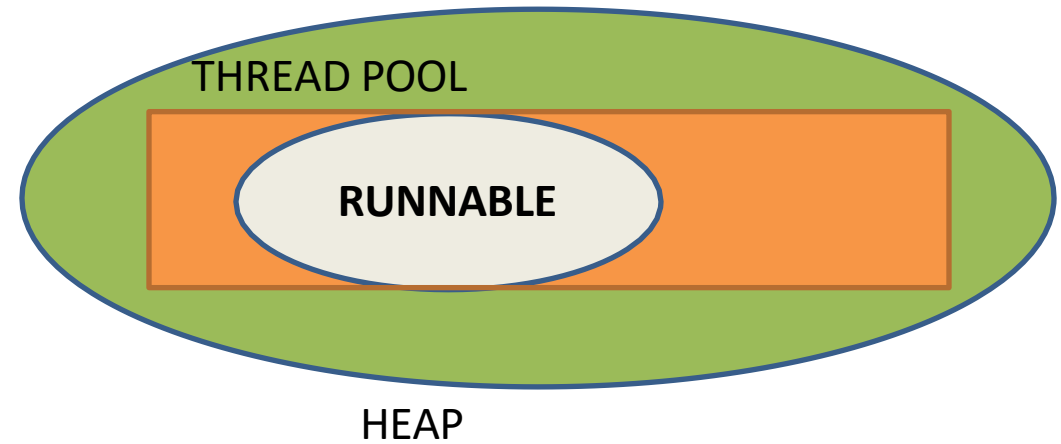
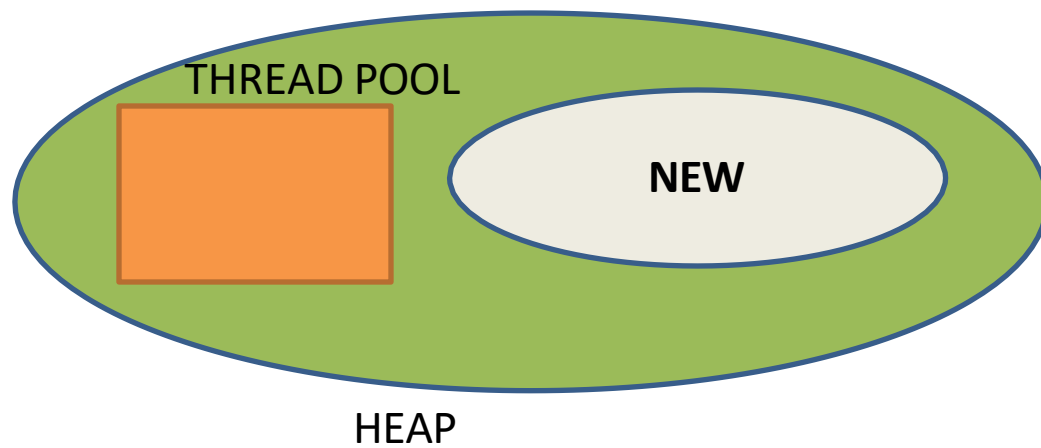
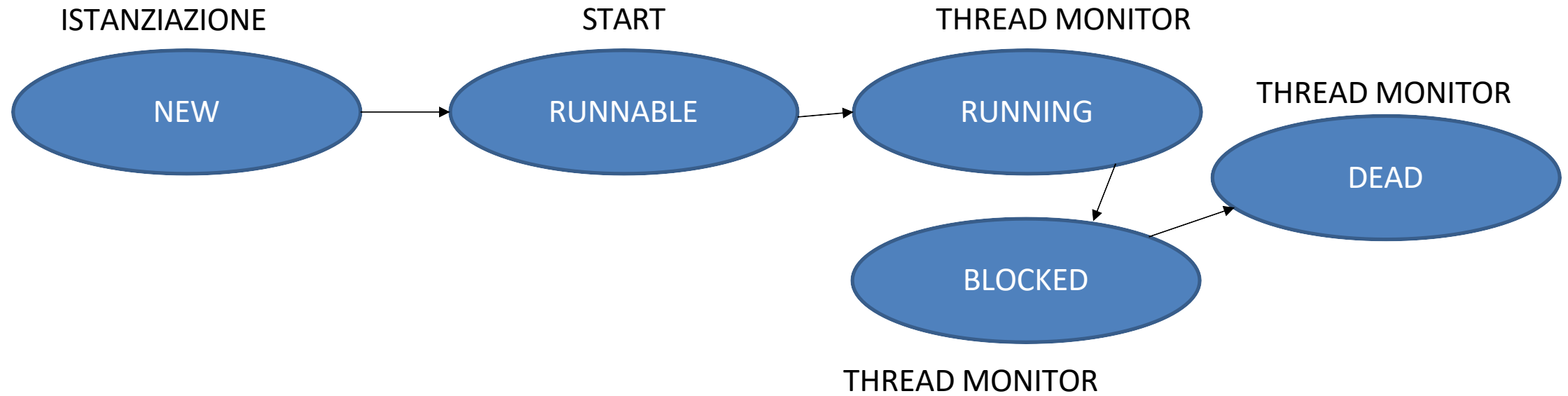
Processo nativo eseguito internamente al Thread Pool; il Thread è in grado di gestire l'esecuzione a runtime degli Oggetti Thread allocati nel Thread Pool garantendo l'accesso concorrente degli stessi ad una risorsa condivisa

Ogni Thread in ogni momento del suo ciclo di vita si trova in un determinato stato (CE NE SONO 5 POSSIBILI)

Ogni Oggetto Thread è un Oggetto Speciale, ovvero il suo CICLO di vita è diverso da quello degli altri Oggetti Java, ed è fatto da 5 stati: NEW,RUNNABLE,RUNNING,BLOCKED, DEAD

Il Thread Monitor è IN GRADO di gestire l'accesso concorrente degli Oggetti Thread ad una RISORSA condivisa solo se gli Oggetti Thread si trovano in stato RUNNABLE

## CICLO DI VITA DI UN THREAD



IN GENERALE IL THREAD MONITOR NON DA' GARANZIA DELL'ORDINE DI ESECUZIONE DI THREAD E NEMMENO DELL' ATOMICITA' DI ESECUZIONE DEL CODICE CONTENUTO NELLA RISORSA CONDIVISA (OVVERO NON C'E' GARANZIA CHE IL THREAD IN ESECUZIONE UNA VOLTA ATTIVATO ESEGUA L'INTERO METODO run)

UTILIZZARE LA PAROLA CHIAVE `synchronized` per un blocco di codice equivale a chiedere esplicitamente al Thread Monitor di blindare l'esecuzione di quel blocco per ogni singolo Thread, garantendo che ogni qual volta un Thread inizia ad eseguirlo lo eseguirà tutto nella sua interezza

```
public class ChatThread extends Thread{
    /*
     * La risorsa condivisa deve essere
     * rappresentata dall'Override del metodo
     * run firmato nell'Interfaccia Runnable
     * padre della Classe Thread
     */
    public void run() {
        // cycle interval
        try {
            synchronized(ChatThread.class) {
                Thread.sleep(5000);
                System.out.println("Thread Name " +
                    Thread.currentThread().getName());
                System.out.println("Hello " + Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

The diagram illustrates the execution of the `run()` method in the `ChatThread` class. Three threads, labeled Thread1, Thread2, and Thread3, are shown with arrows pointing to the `synchronized(ChatThread.class)` block within the `run()` method. This indicates that all three threads are attempting to execute the same synchronized block of code, which will ensure mutual exclusion.

A PARTIRE DALLA VERSIONE 5 IL LINGUAGGIO JAVA HA INTRODOTTO UN NUOVO TIPO : enum

LA PAROLA CHIAVE **enum** consente di definire un tipo che rappresenta un set di informazioni secondo uno schema fisso e riutilizzabile

Tecnicamente parlando in Java una enum è una classe come le altre ma che "implicitamente" (cioè senza che lo scriviamo noi) estende sempre la classe Enum, cosa che ha l'unico inconveniente di rendere impossibile di scrivere enum che derivino da altri tipi.

Il compilatore per ogni classe enum sintetizza per noi un metodo statico (**values**) che ritorna un array di tutti i possibili valori che potranno assumere le variabili che avranno come tipo l'enum.

il fatto che le enum siano a tutti gli effetti classi (anche se speciali) , apre la possibilità di aggiungere dentro di essi metodi e field e, e questo è più sorprendente, anche costruttori (che devono essere necessariamente private).

## Proprietà enum

- Ogni costante enum rappresenta un **oggetto** di tipo enum.
- Il tipo enum può essere passato come tipo argomento ad un metodo.
- Ogni costante enum è sempre implicitamente **public static final** . Poiché è **final** , non possiamo creare enumerazioni figlio.
- Possiamo scorrere l'Enum utilizzando valori() e loop. La funzione values() restituisce una matrice di valori Enum come costanti utilizzando le quali possiamo scorrere i valori.

## EnumMap

EnumMap di Java è un'implementazione di mappa specializzata progettata specificamente per l'uso con le chiavi enum.

EnumMap fornisce prestazioni ed efficienza della memoria migliori rispetto alle normali implementazioni HashMap o TreeMap quando si lavora con le chiavi enum.

Una EnumMap è un'implementazione della mappa ad alte prestazioni ed efficiente in termini di memoria che funziona esclusivamente con le chiavi enum.

È supportato da un array e fornisce prestazioni a tempo costante per operazioni di base come get e put, a differenza di altre implementazioni di mappe. EnumMap funziona solo con enumerazioni, il che significa che il tentativo di utilizzarlo con chiavi non enumerative comporterà un'eccezione di runtime.

Link di riferimento per esempi su EnumMap

<https://www.baeldung.com/java-enum-map>

**PATTERN = UNA O UN INSIEME DI LINEE GUIDA RILASCIATE ALLO SCOPO DI OTTIMIZZARE LE APPLICAZIONI O AI FINI DELLA RISOLUZIONE DI UN PROBLEMA RICORRENTE**

**GANGS OF FOURS DESIGN PATTERNS = PATTERN SUDDIVISI IN 3 CATEGORIE:**

**CREATIONAL**

LINEE GUIDA RELATIVAMENTE ALLA CREAZIONE DI OGGETTI

**STRUCTURAL**

LINEE GUIDA RELATIVAMENTE ALLA STRUTTURA DA CONFERIRE AD UNA APPLICAZIONE

**BEHAVIOURAL**

LINEE GUIDA RELATIVAMENTE A COME GLI OGGETTI DEVONO INTERAGIRE FRA DI LORO

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

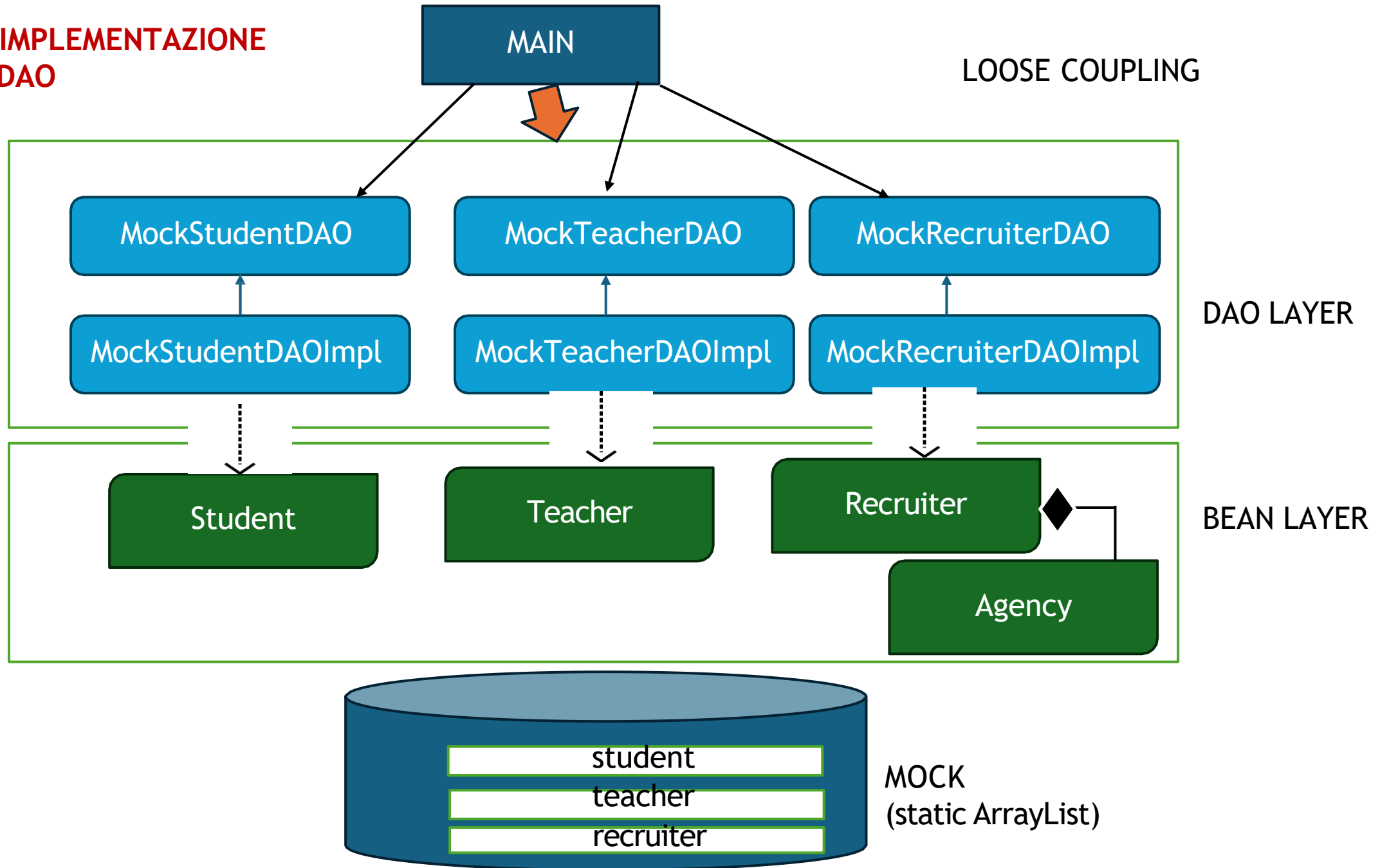


**PATTERN LOOSE COUPLING -->> NON FA PARTE DEI GANGS OF FOUR**

**PATTERN DAO -->>>NON FA PARTE DEI GANGS OF FOUR (CONCETTUALMENTE PUO' ESSERE CONSIDERATO COME STRUCTURAL MA FUORI DAI GANGS OF FOUR)**

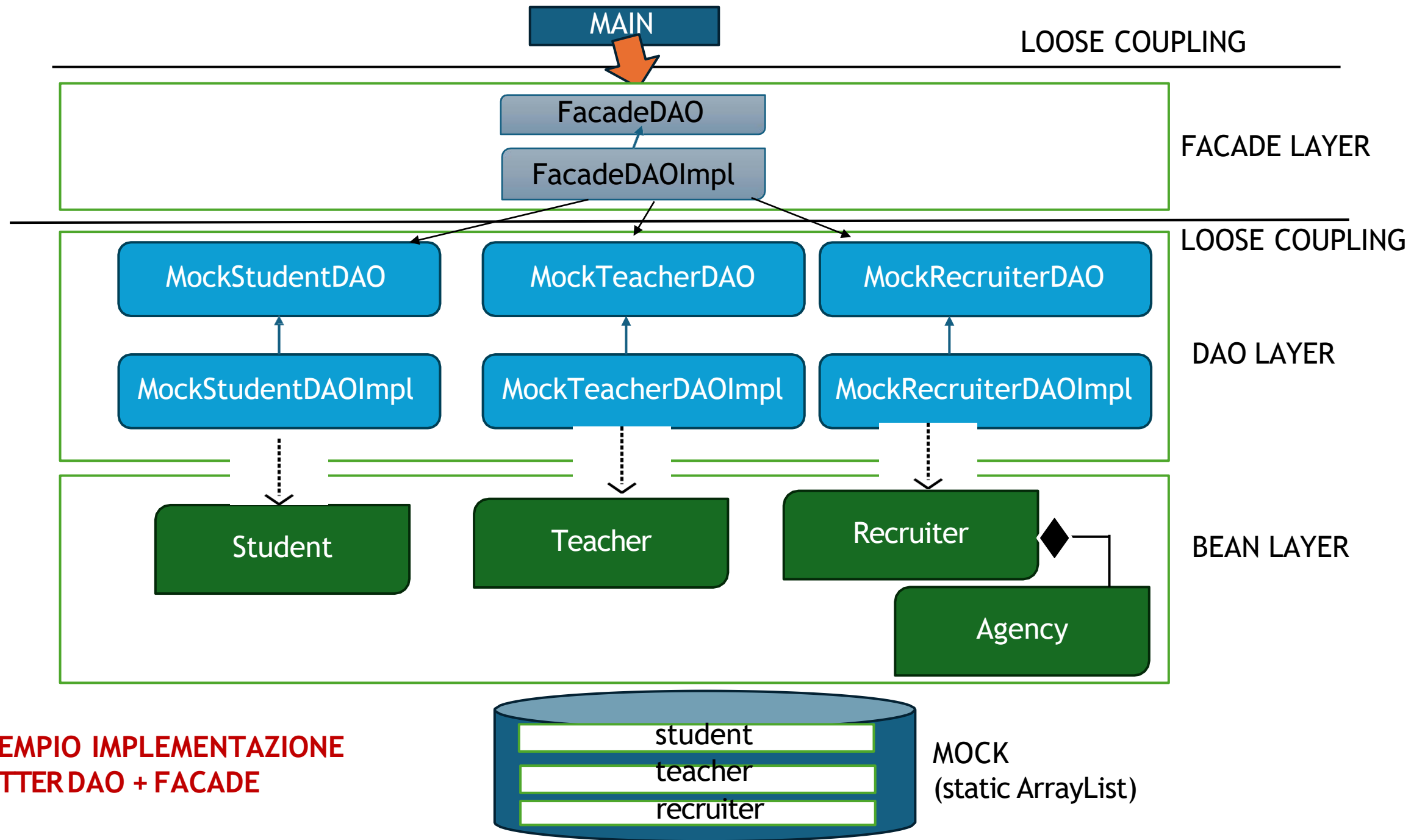
**DAO = DATA ACCESS OBJECT = PER APPLICAZIONI CHE INTERAGISCONO CON DATABASE E' PREFERIBILE SEPARARE LA LOGICA DI PERSISTENZA (LOGICA DI CRUD) IN UNO STRATO APPLICATIVO DEDICATO, COMPOSTO DA UNA INTERFACCIA E UNA CLASSE CHE ESPONGANO LE OPERAZIONI DI CRUD PER OGNI ENTITA' CONCETTUALE DEL DATABASE, IN MANIERA TALE CHE LO STRATO APPLICATIVO DI CRUD SIA AGNOSTICO RISPETTO ALLE APPLICAZIONI INVOCANTI**

## ESEMPIO IMPLEMENTAZIONE PATTERN DAO



**FACADE** = GoF (Structural) = E' POSSIBILE STRUTTURARE UNA APPLICAZIONE INVOCATA IN MANIERA TALE CHE SI ESPONGA ALLE APPLICAZIONI INVOCATE TRAMITE UN UNICO E CENTRALIZZATO PUNTO DI INGRESSO AGNOSTICO RISPETTO AL SISTEMA INVOCANTE

IMPLEMENTAZIONE PATTERN FACADE = CREARE UNA INTERFACCIA CHE ESPONGA TUTTE LE FUNZIONALITA' DEL SISTEMA E UNA CLASSE CHE INVOCI TUTTI I METODI DELLA STRUTTURA APPLICATIVA SOTTOSTANTE

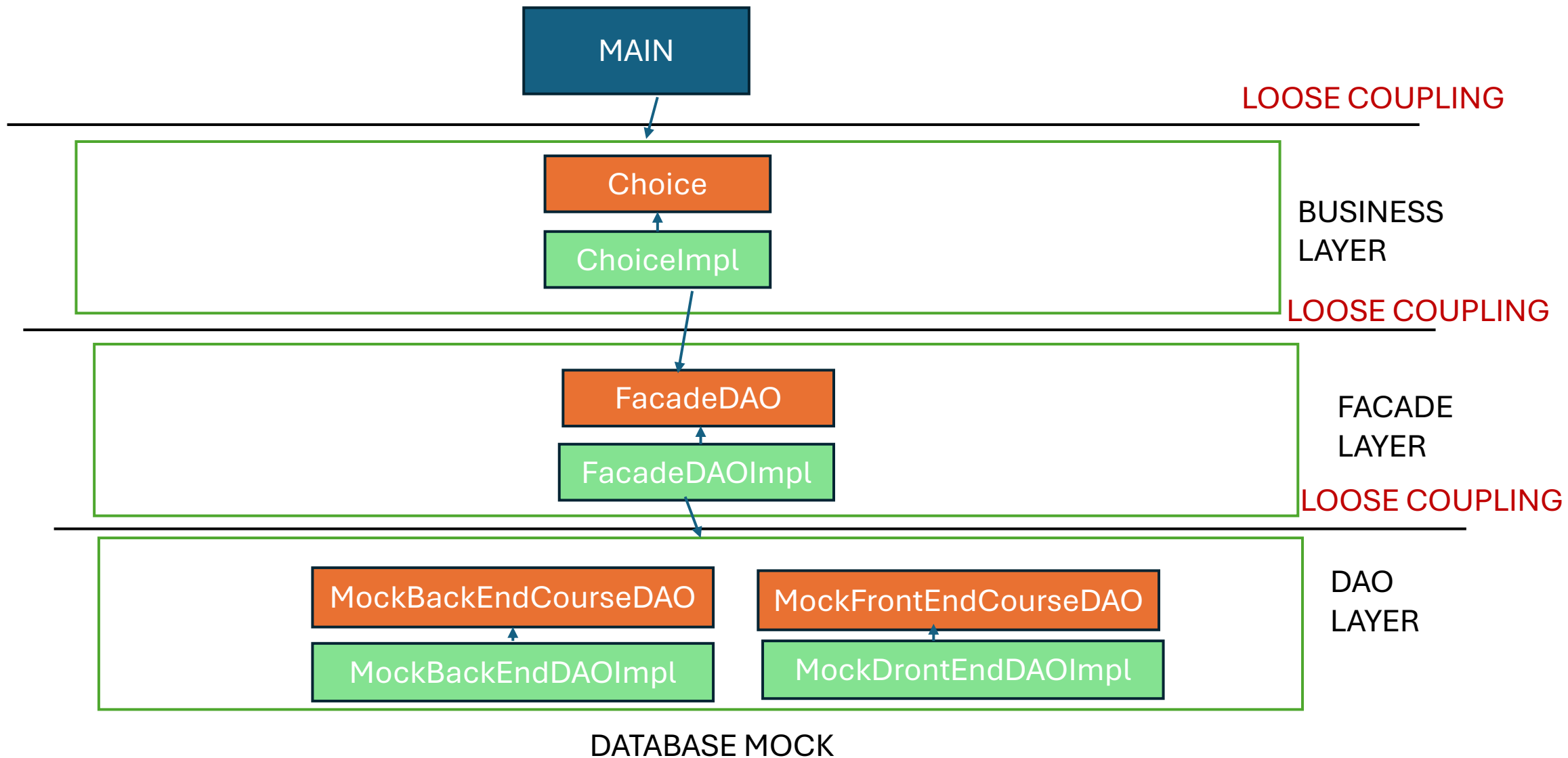


**ESEMPIO IMPLEMENTAZIONE  
PATTER DAO + FACADE**

PATTERN FACTORY = PATTERN CHE SI CONSIDERA UTILE APPLICARE IN CONTESTI RELATIVAMENTE AI QUALI NON SI VOGLIA DARE ALL'APPLICAZIONE INVOCANTE ALCUNA RESPONSABILITA' DI CREAZIONE DI OGGETTI E DI BUSINESS LOGIC, MA SI VOGLIA DARE LA POSSIBILITA' ALL'APPLICAZIONE INVOCANTE IN BASE A DIVERSI INPUT (SCELTE) DI DELEGARE L'APPLICAZIONE INVOCATA PER LA LOGICA DI CREAZIONE DEGLI OGGETTI E QUALSIASI LOGICA DI BUSINESS.

OGNI VOLTA CHE VIENE ISTANZIATA UNA CLASSE FIGLIA VIENE AUTOMATICAMENTE INVOCATO PRIMA IL COSTRUTTORE DELLA CLASSE PADRE E POI IL COSTRUTTORE DELLA CLASSE FIGLIA

ASSIOMIA : NEL CASO IN CUI LA CLASSE PADRE DICHIARI UN COSTRUTTORE CUSTOM, LA CLASSE FIGLIA E' OBBLIGATA A DICHIARARE UN COSTRUTTORE CUSTOM CHE ABBIA ALMENO LE PROPRIETA' GIA' UTILIZZATE NEL COSTRUTTORE CUSTOM DELLA CLASSE PADRE (POTENDO EVENTUALMENTE AGGIUNGERNE ALTRE) E INVOCARE ESPPLICITAMENTE TRAMITE LA PAROLA CHIAVE **super** IL COSTRUTTORE DELLA CLASSE PADRE



**ESEMPIO DI IMPLEMENTAZIONE DEI PATTERN FACTORY + FACADE + DAO**

PATTERN SINGLETON = GoF (Creational)

HA UN SENSO DI APPLICAZIONE QUANDO SI VUOLE CHE UN OGGETTO, UNA VOLTA CREATO, SIA IRRIPETIBILE, OVVERO NON SI POSSANO CREARE SUCCESSIVI OGGETTI (QUINDI UNA SOLA ISTANZA)

CREARE PIU' OGGETTI A PARTIRE DALLA STESSA CLASSE ALLOCANDO DIVERSE AREE DI MEMORIA PER OGNI OGGETTO CREATO VUOL DIRE CREARE OGGETTI IN MODALITA' **PROTOTYPE**

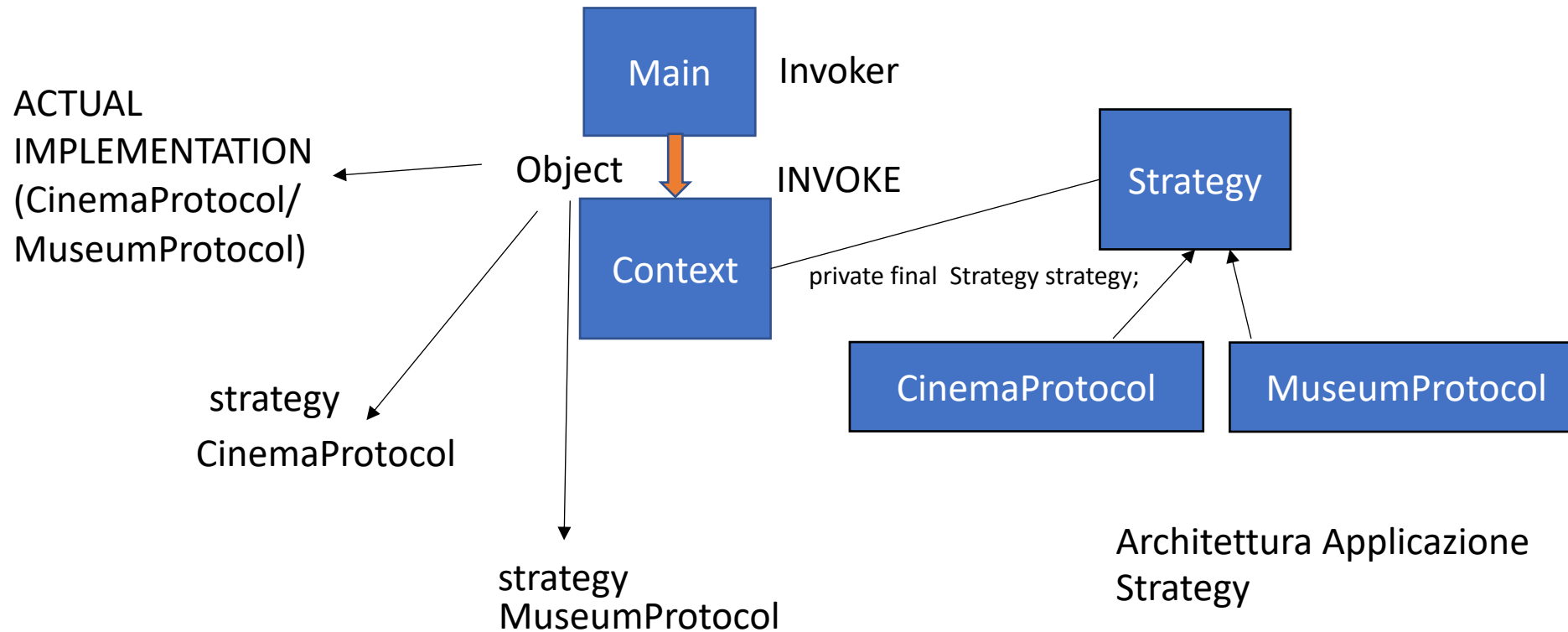


## MODALITA' DI IMPLEMENTAZIONE DEL PATTERN SINGLETON (JAVA SE):

- dichiarazione di una variabile di classe (static) dello stesso tipo della Classe rispetto alla quale si intende implementare il pattern Singleton
- implementazione del costruttore private
- implementazione di un metodo di classe (static) che ritorni una reference dello stesso tipo della Classe e costruisca una logica per istanziare una sola volta un Oggetto della classe tramite Costruttore private

## BEHAVIORAL

**PATTERN STRATEGY = PATTERN CHE SI APPLICA QUANDO IN BASE ALL'INVIO DI INFORMAZIONI DI DIVERSA TIPOLOGIA DA PARTE DELL'APPLICAZIONE INVOCANTE OCCORRE CHE L'APPLICAZIONE INVOCATA ATTUI UNA STRATEGIA DIFFERENTE DI IMPLEMENTAZIONE. IL PATTERN STRATEGY A DIFFERENZA DEL PATTERN FACTORY DELEGA ALL'APPLICAZIONE INVOCANTE LA CREAZIONE DEGLI OGGETTI CHE RAPPRESENTANO LE INFORMAZIONI IN BASE ALLE QUALI L'APPLICAZIONE INVOCATA ATTUA UNA DIVERSA STRATEGIA DI RISPOSTA ALL'APPLICAZIONE INVOCANTE.**



## CREATIONAL

**BUILDER PATTERN = PATTERN UTILE NEI CASI IN CUI SI VOGLIA SEPARARE LA CREAZIONE DI UN OGGETTO DALLA SUA RAPPRESENTAZIONE, OVVERO NEI CASI IN CUI SI ABBA L'ESIGENZA DI VALORIZZARE UN OGGETTO IN MANIERA DINAMICA EVITANDO DI UTILIZZARE PIU' COSTRUTTORI O UN SOLO COSTRUTTORE CON ELEMENTI CHE IN UN MOMENTO SPECIFICO NON RAPPRESENTANO DI FATTO LO STATO DELL'OGGETTO**

### STEP PER L'IMPLEMENTAZIONE DEL PATTERN BUILDER

1. CREAZIONE DELLA CLASSE OUTER CHE RAPPRESENTA LA CLASSE DI CUI VORREMO OTTENERE GLI OGGETTI
2. DICHIARAZIONE DI TUTTE LE VARIABILI DI ISTANZA ALL'INTERNO DELLA CLASSE OUTER, SIA QUELLE DELLA STRUTTURA FISSA SIA QUELLE DELLA STRUTTURA OPZIONALE
3. CREAZIONE DI UNA INNER CLASS STATIC
4. DICHIARAZIONE DELLE VARIABILI DI ISTANZA ALL'INTERNO DELLA INNERCLASS, SPECULARI A QUELLE DICHIARATE NELLA OUTER CLASS
5. DICHIARAZIONE DI METODI DI SET ALL'INTERNO DELLA INNER CLASS, CHE ABBIAMO COME TIPO DI RITORNO IL TIPO DELLA CLASSE INNER, RELATIVAMENTE ALLE VARIABILI DI ISTANZA OPZIONALI
6. DICHIARAZIONE DI UN COSTRUTTORE ALL'INTERNO DELLA INNERCLASS CON I PARAMETRI FISSI
7. DICHIARAZIONE DI UN COSTRUTTORE PRIVATE NELLA OUTER CLASS CHE RICEVE IN INPUT UNA REFERENCE DI TIPO INNER E ESEGUE UNA COPIA DEL CONTENUTO DELL'OGGETTO INNER ALL'INTERNO DELL'OGGETTO OUTER
8. DICHIARAZIONE DI UN METODO (preferibilmente di nome build) ALL'INTERNO DELLA INNERCLASS CHE INVOCA IL COSTRUTTORE DELLA OUTER CLASS PASSANDOGLI IN INPUT UNA REFERENCE DI TIPO INNER (this)
9. OVERRIDE DEL METODO toString ALL'INTERNO DELLA OUTER CLASS

DATABASE = INFRASTRUTTURA FISICA CHE CONSENTE DI ARCHIVIARE (STORICIZZARE) INFORMAZIONI IN MODALITA' PERMANENTE FINO AD EVENTUALE CANCELLAZIONE DELLE STESSE

ESISTONO TRE TIPOLOGIE DI DATABASE:

- DATABASE RELAZIONALE (COMPOSIZIONE A TABELLE) (mySQL, Oracle, PostgreSQL)
- DATABASE NON RELAZIONALE (COMPOSIZIONE A COLLECTION) (MongoDB, RavenDB)
- DATABASE A GRAFI (COMPOSIZIONE A NODI) Neo4J

DATABASE RELAZIONALE = DATABASE CHE CONSENTE DI ARCHIVIARE I DATI IN TABELLE **EVENTUALMENTE** RELAZIONATE TRA DI LORO

DBMS = DATABASE MANAGEMENT SYSTEM = SOFTWARE CON INTERFACCIA GRAFICA CHE CONSENTE LA GESTIONE DI UN DATABASE (CREAZIONE E MANUTENZIONE). SE NE DISTINGUONO 2 TIPI: RDBMS E NORDBMS.

RDBMS = RELATIONAL DBMS = SOFTWARE CHE CONSENTE TRAMITE INTERFACCIA GRAFICA LA GESTIONE DI UN DATABASE RELAZIONALE

NORDBMS = NO RELATIONAL DBMS = SOFTWARE CHE CONSENTE TRAMITE INTERFACCIA GRAFICA LA GESTIONE DI UN DATABASE NON RELAZIONALE

TABELLA = L'UNITA' PIU' ELEMENTARE CHE CONSENTE DI EFFETTUARE 4 OPERAZIONI (OPERAZIONI DI CRUD)

CRUD E' UN ACRONIMO CHE STA PER:

C = CREATION (INSERIMENTO DI NUOVE INFORMAZIONI)

R = READ (LETTURA DI DATI GIA' ESISTENTI)

U = UPDATE (MODIFICA DI DATI GIA' ESISTENTI)

D = DELETE (CANCELLAZIONE DI DATI GIA' ESISTENTI)

Tabella ->> composta da record (righe) e field (colonne o campi)

I record rappresentano il contenuto informativo della tabella

I campi rappresentano il tipo delle informazioni

person

id (int)	first_name (varchar)	last_name (varchar)	age (byte)
1	Mario	Rossi	24

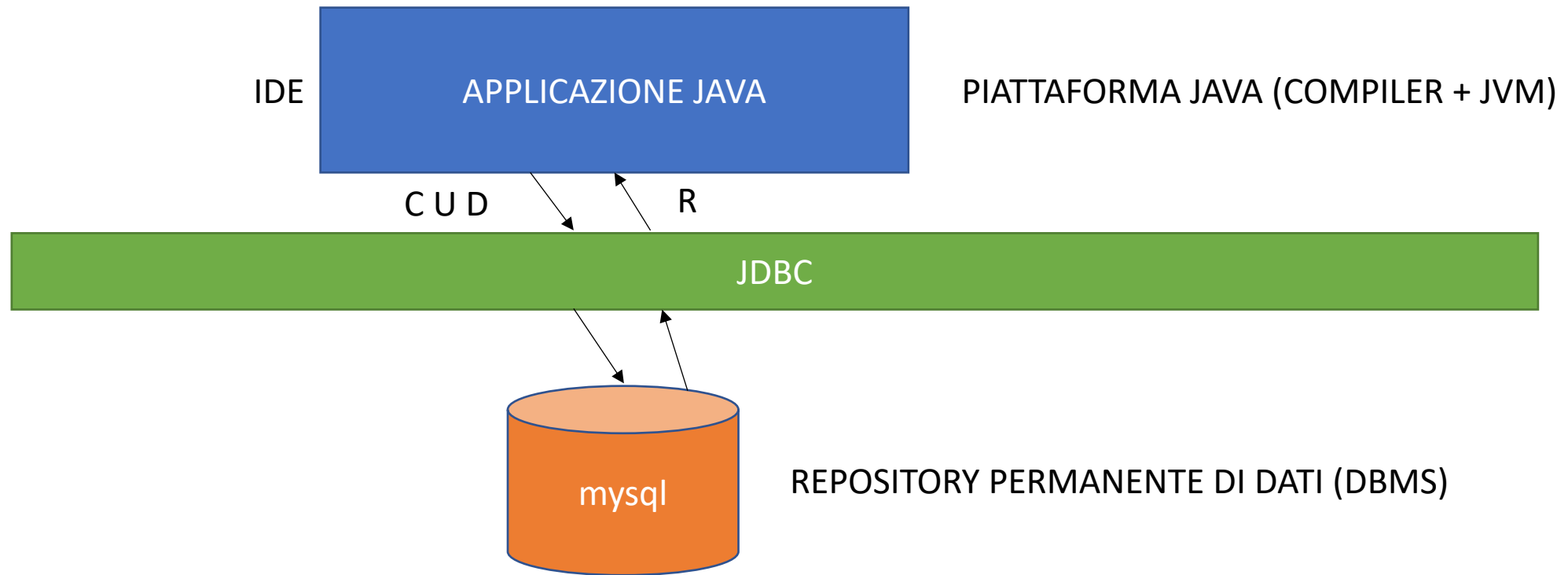
Esempio di tabella

PK = PRIMARY KEY = IL VALORE ASSUNTO DAL CAMPO DEVE ESSERE UNIVOCO, OVVERO NON E' POSSIBILE AVERE PIU' RECORD CHE HANNO LO STESSO VALORE DI PRIMARY KEY

SQL = Structured Query Language = LINGUAGGIO RICONOSCIUTO UNIVERSALMENTE DA TUTTI I DATABASE RELAZIONALI PER L'ESECUZIONE DELLE 4 OPERAZIONI DI CRUD

JDBC = JAVA DATABASE CONNECTIVITY = INSIEME DI API FORNITE DA JAVA SE CHE CONSENTONO L'IMPLEMENTAZIONE (ALL'INTERNO DI UNA QUALSIASI APPLICAZIONE JAVA SIA STAND ALONE SIA ENTERPRISE) DI TUTTE LE OPERAZIONI DI CRUD SU UN DATABASE RELAZIONALE

JDBC IMPONE L'USO DEL LINGUAGGIO NATIVO SQL



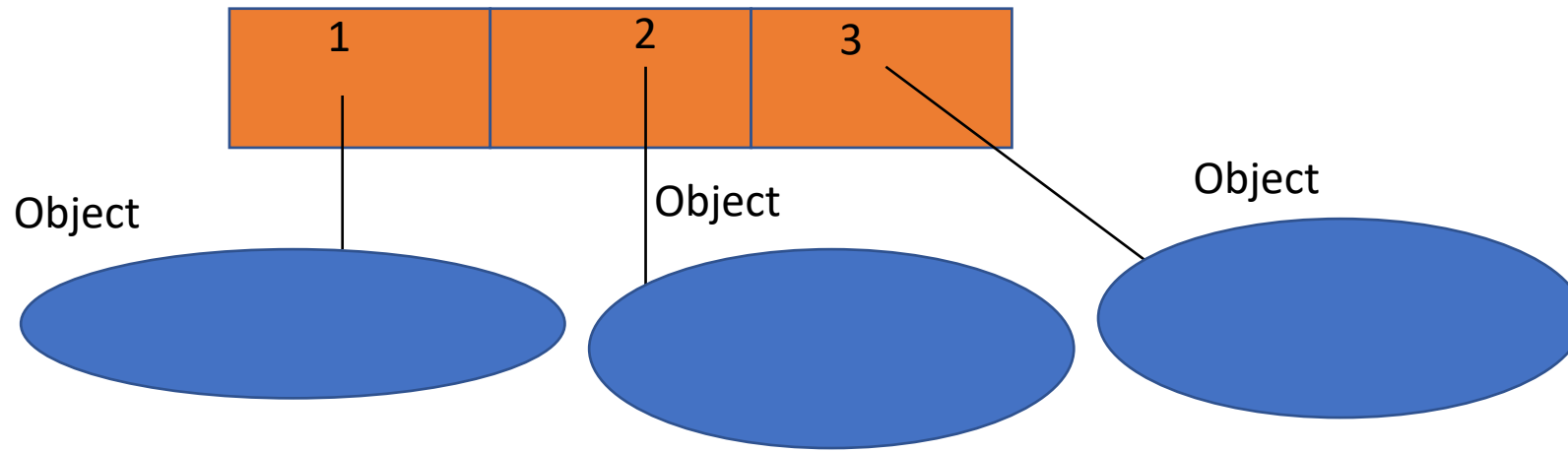
## STEP DI IMPLEMENTAZIONE DI UNA APPLICAZIONE JDBC:

1. CONNESSIONE AL database relazionale (driver) specificandone l'esatta tipologia (mySQL, Oracle..) -->> **facoltativo da JDBC 4 in poi per la Applicazioni Stand Alone, necessario invece per le Applicazioni Enterprise (richiesto dal Web Container/Application Server)**
2. INDICAZIONE DEI CORRETTI PARAMETRI DI CONNESSIONE (url, username, password)
3. OPERAZIONI DI CRUD (PreparedStatement/Statement/ResultSet)

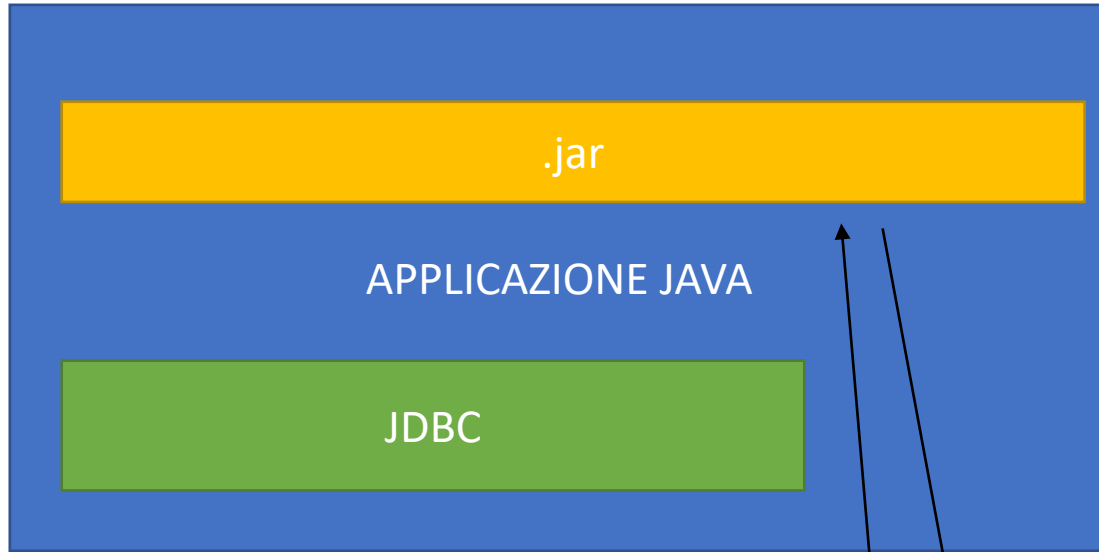
PreparedStatement è una API JDBC che assume la forma della specifica CRUD richiesta in base alla prima parola chiave dello script sql passato in input al metodo prepareStatement. E' per questo motivo che PreparedStatement viene considerata una API polimorfica nel senso che può assumere di volta in volta la forma della CRUD richiesta

### 4. CHIUSURA DELLA CONNESSIONE

ResultSet (API JDBC) è una STRUTTURA DATI DINAMICA che si autocompone a runtime strutturandosi con tante locazioni di memoria per quanti sono i record presenti sulla tabella al momento dell'operazione di lettura; ogni locazione di memoria conterrà una reference ad un Oggetto speculare al contenuto di un record della tabella, nel senso che tale Oggetto verrà valorizzato con tante variabili di istanza per quante sono le colonne della tabella estratte dall'operazione di lettura



IDE



1. Driver di connessione

```
Class.forName(dbDriver);
```

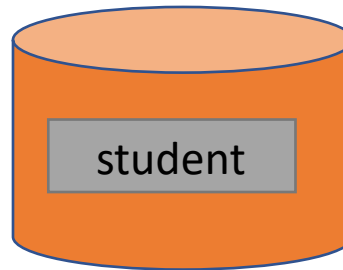
2. Parametri di connessione

```
Connection con = DriverManager.getConnection(dbUrl,dbUser,dbPass)
```

3. Operazioni di CRUD

La configurazione delle dipendenze di progetto (.jar) può essere effettuata manualmente o tramite un tool di automation (MAVEN o GRADLE)

DBMS



openjobmetis

REPOSITORY PERMANENTE DI DATI (DBMS)



Ogni vendor di database relazionale ha rilasciato uno o più file di estensione .jar che devono essere inseriti all'interno di un'applicazione Java per far sì che l'applicazione possa fisicamente connettersi al database

La configurazione di connettori all'interno di una applicazione Java può essere eseguita in due modalità differenti:

- MODALITA' MANUALE (SCARICARE FISICAMENTE I FILE .JAR DA INTERNET ALL'INTERNO DEL PROPRIO E INSERIRLI MANUALMENTE ALL'INTERNO DEL PROGETTO)
- MODALITA' AUTOMATICA (UTILIZZO DI UN PACKAGE MANAGER COME MAVEN O GRADLE)

MAVEN = BUILDING TOOL (COMPILATORE) + PACKAGE MANAGER (GESTORE DI PACCHETTI DI DIPENDENZE ALL'INTERNO DI PROGETTI JAVA)

pom.xml -->>>> MASTER FILE DI MAVEN (all'interno di questo file si può indicare il nome delle dipendenze in maniera tale che MAVEN si colleghi via web al MAVEN REPOSITORY e scarichi le dipendenze richieste inserendole automaticamente senza intervento manuale all'interno del progetto)

Ogni pom.xml deve essere identificato da due Stringhe: Group id e Artifact id

PATTERN DAO = Data Access Object = E' PREFERIBILE CONCENTRARE L'IMPLEMENTAZIONE DELLE OPERAZIONI DI CRUD ALL'INTERNO DI UNO STRATO APPLICATIVO (APPLICATION LAYER) DEDICATO RENDENDOLE AGNOSTICHE DALLE APPLICAZIONI INVOCANTI

PATTER DTO = DATA TRANSFER OBJECT = E' CONSIGLIABILE, ALL'INTERNO DI UNA APPLICAZIONE CHE DEBBA IMPLEMENTARE LE OPERAZIONI DI CRUD CREARE, PER OGNI TABELLA DEL DATABASE, UNA CORRISPONDENTE CLASSE CHE CONTENGA TANTE VARIABILI DI ISTANZA PER QUANTE SONO LE COLONNE DELLA CORRISPONDENTE TABELLA, AD ECCEZION FATTA DI EVENTUALI COLONNE AUTO INCREMENT, TANTI METODI DI SET E GET PER QUANTE SONO LE VARIABILI DI ISTANZA, ED ALMENO IL COSTRUTTORE DI DEFAULT

PATTERN VO = VALUE OBJECT = E' CONSIGLIABILE, ALL'INTERNO DI UNA APPLICAZIONE CHE DEBBA IMPLEMENTARE OPERAZIONI CRUD DI LETTURA CREARE, PER OGNI TABELLA DEL DATABASE, UNA CORRISPONDENTE CLASSE CHE CONTENGA TANTE VARIABILI DI ISTANZA PER QUANTE SONO LE COLONNE DELLA TABELLA CORRISPONDENTE, TANTI METODI DI SET E GET PER QUANTE SONO LE VARIABILI DI ISTANZA, E ALMENO IL COSTRUTTORE DI DEFAULT

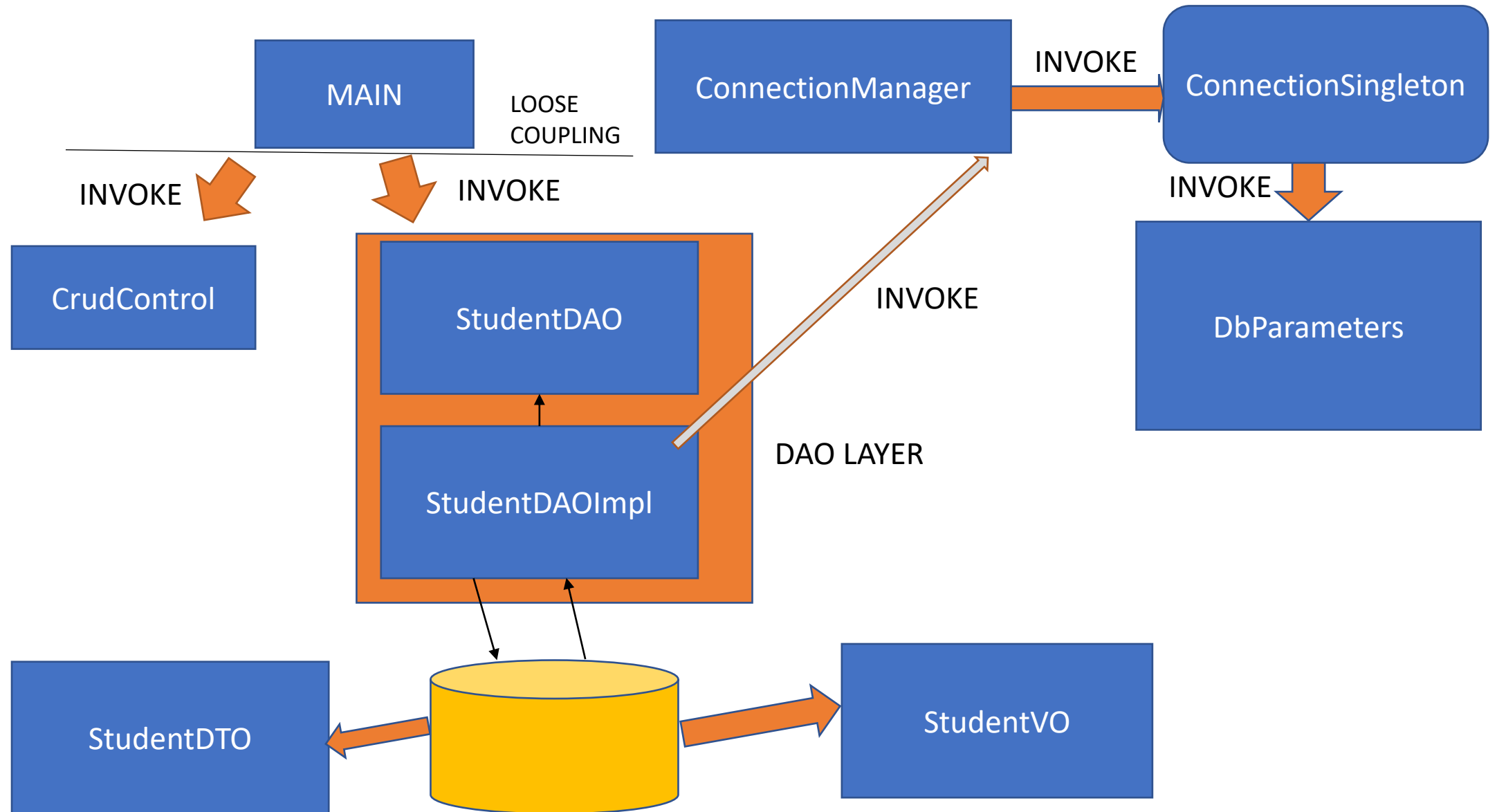
JAVA SE fornisce alcune INTERFACCE API chiamate MARKER INTERFACE

Le MARKER INTERFACE SONO INTERFACCE CHE, SE IMPLEMENTATE DA UNA CLASSE, CONFERISCONO AGLI OGGETTI DI QUELLA CLASSE UN COMPORTAMENTO «SPECIALE» A RUNTIME

LA MARKER INTERFACE Serializable, se implementata da una Classe, conferisce agli Oggetti di quella Classe la caratteristica di essere SERIALIZZABILI

Un OGGETTO SERIALIZZABILE E' UN OGGETTO CHE, NEL CASO DI Malfunzionamento DELLA JVM, VIENE DEALLOCATO DALL'HEAPE VIENE SALVATO SUL DISCO FISICO DELLA MACCHINA SU CUI E' INSTALLATA LA JVM E, AD UN EVENTUALE RIFUNZIONAMENTO DELLA JVM, VIENE RIPRISTINATO ALL'INTERNO DELL'HEAP ESATTAMENTE CON LO STESSO CONTENUTO CHE AVEVA NEL MOMENTO DEL CRASH

## ARCHITETTURA JDBC\_CRUD\_REENGINEERING



DATABASE RELAZIONALE = STORE DATI IN TABELLE EVENTUALMENTE RELAZIONATE TRA DI LORO

## RELAZIONI TRA TABELLE

ESISTONO TRE TIPOLOGIE DI RELAZIONI POSSIBILI:

- RELAZIONE OneToOne (ad un record di una tabella è associato il record di un'altra tabella)
- RELAZIONE OneToMany (ad un record di una tabella sono associati n record di un'altra tabella)
- RELAZIONE ManyToMany (a più record di una tabella sono associati più record di un'altra tabella)

Per impostare una relazione OneToMany occorre creare due tabelle di cui una rappresenta la tabella padre e una rappresenta la tabella figlia all'interno della quale è necessario assegnare ad un campo il vincolo di Foreign Key verso la Primary Key della tabella padre

La Relazione OneToMany contiene due versi :

- verso OneToMany
- verso ManyToOne

La tabella padre rappresenta il verso OneToMany

La tabella figlia rappresenta il verso ManyToOne

ESISTE LA POSSIBILITA' DI ASSEGNARE ALLA TABELLA FIGLIA UN VINCOLO DI CASCADE IN MANIERA TALE CHE AD UNA DETERMINATA OPERAZIONE DI CRUD COMPIUTA SULLA TABELLA PADRE VENGA AUTOMATICAMENTE ATTUATA LA CORRISPONDENTE AZIONE SUI RECORD ASSOCIATI DELLA TABELLA FIGLIA

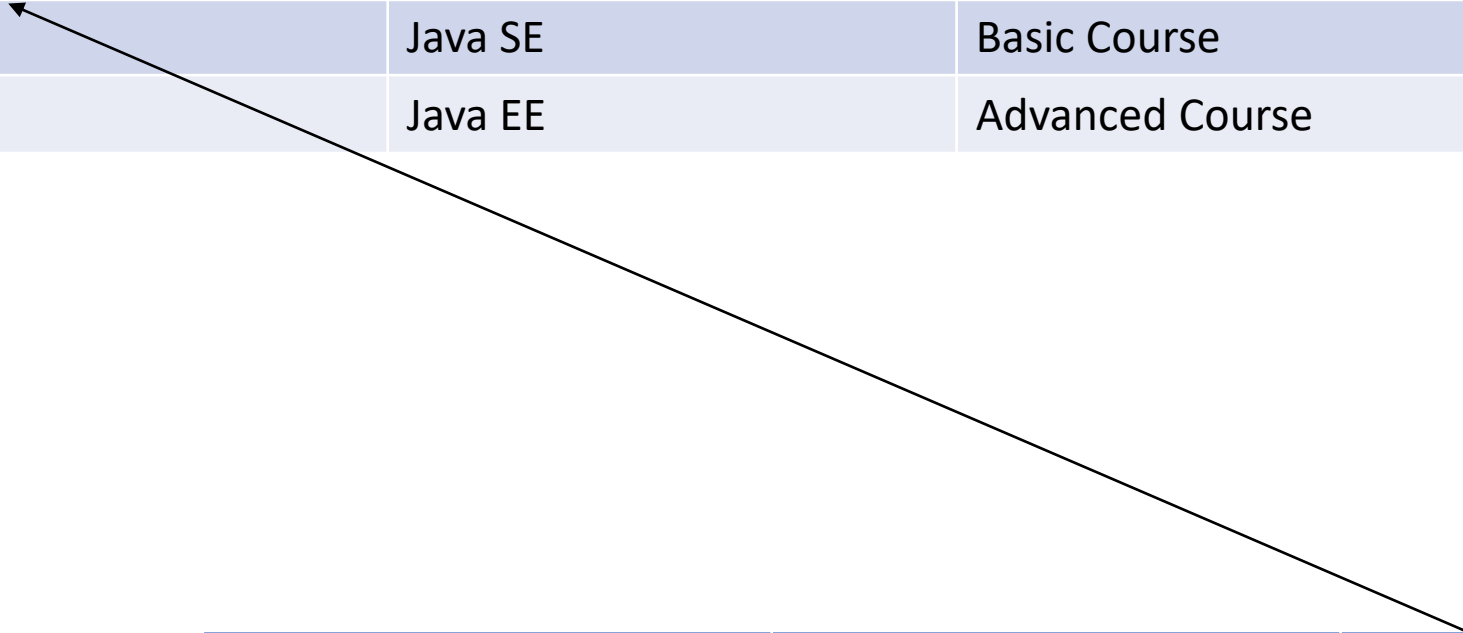
## ***ESEMPIO DI RELAZIONE ONE TO MANY***

course

id (int, PK,A_I)	title (varchar)	description (varchar)
1	Java SE	Basic Course
2	Java EE	Advanced Course

id (int, PK, A_I)	location	course_id (FK)
1	Milan	1
2	Rome	1

review



## Esempio di relazione Many To Many

customer

id (PK,AI)	name (varchar)	country (varchar)
1	Giancarlo Sesti	Italy
2	Marco Proto	USA

FK

FK

JOIN TABLE

customer_id	product_id
1	1
1	2
2	1
2	2

Una relazione Many To Many tra due tabelle si imposta creando una terza tabella (JOIN TABLE) in cui ciascuna colonna è una fk verso la pk di una delle due tabelle ed Entrambe le colonne della JOIN TABLE rappresentano una PK composta

id (PK,AI)	name (varchar)	origin (varchar)
1	pc	Japan
2	smartphone	Korea

product