Programmazione Java
Corso Pratico
4 – Dalla teoria alla pratica

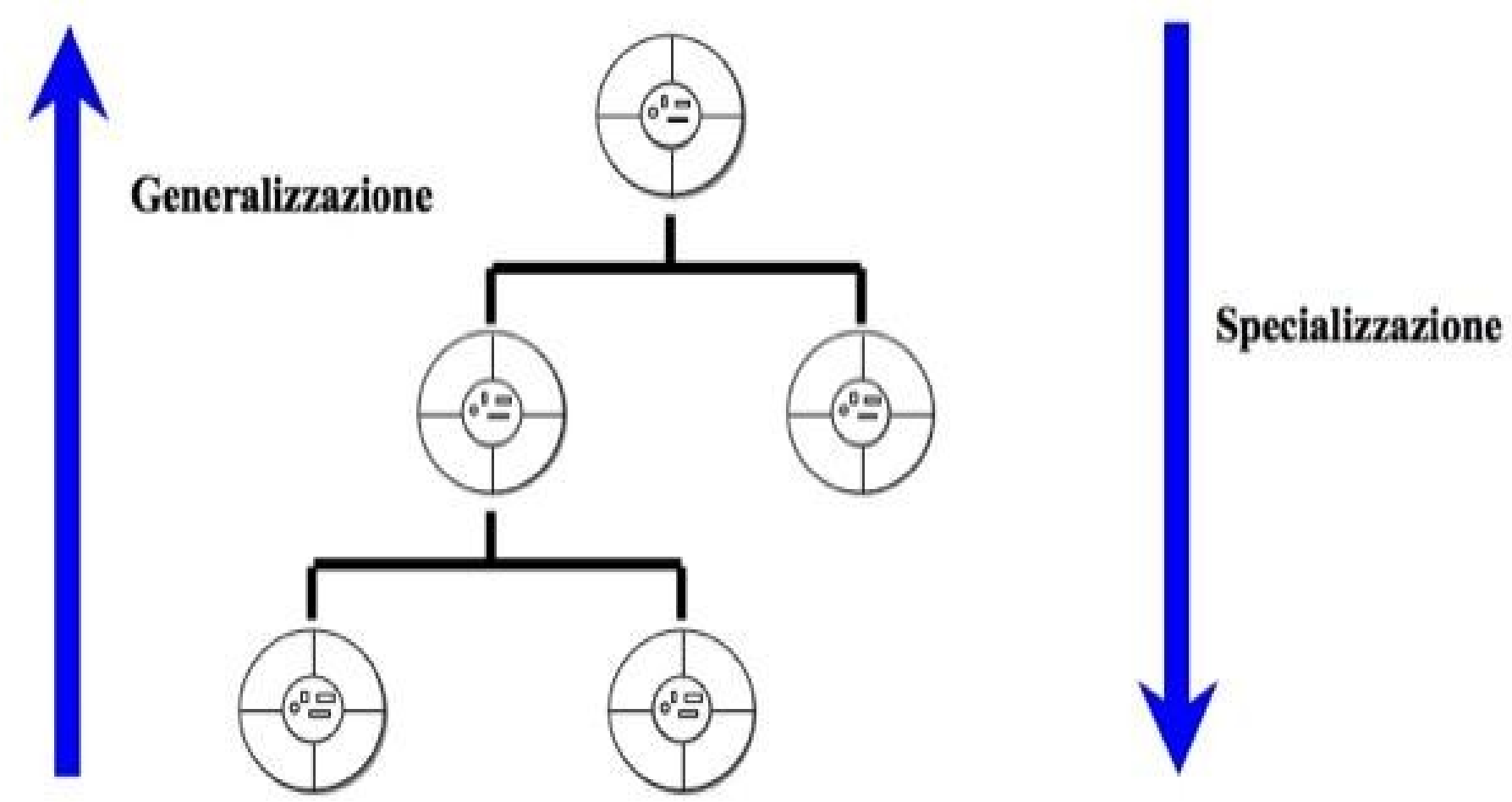
Maurizio Franco

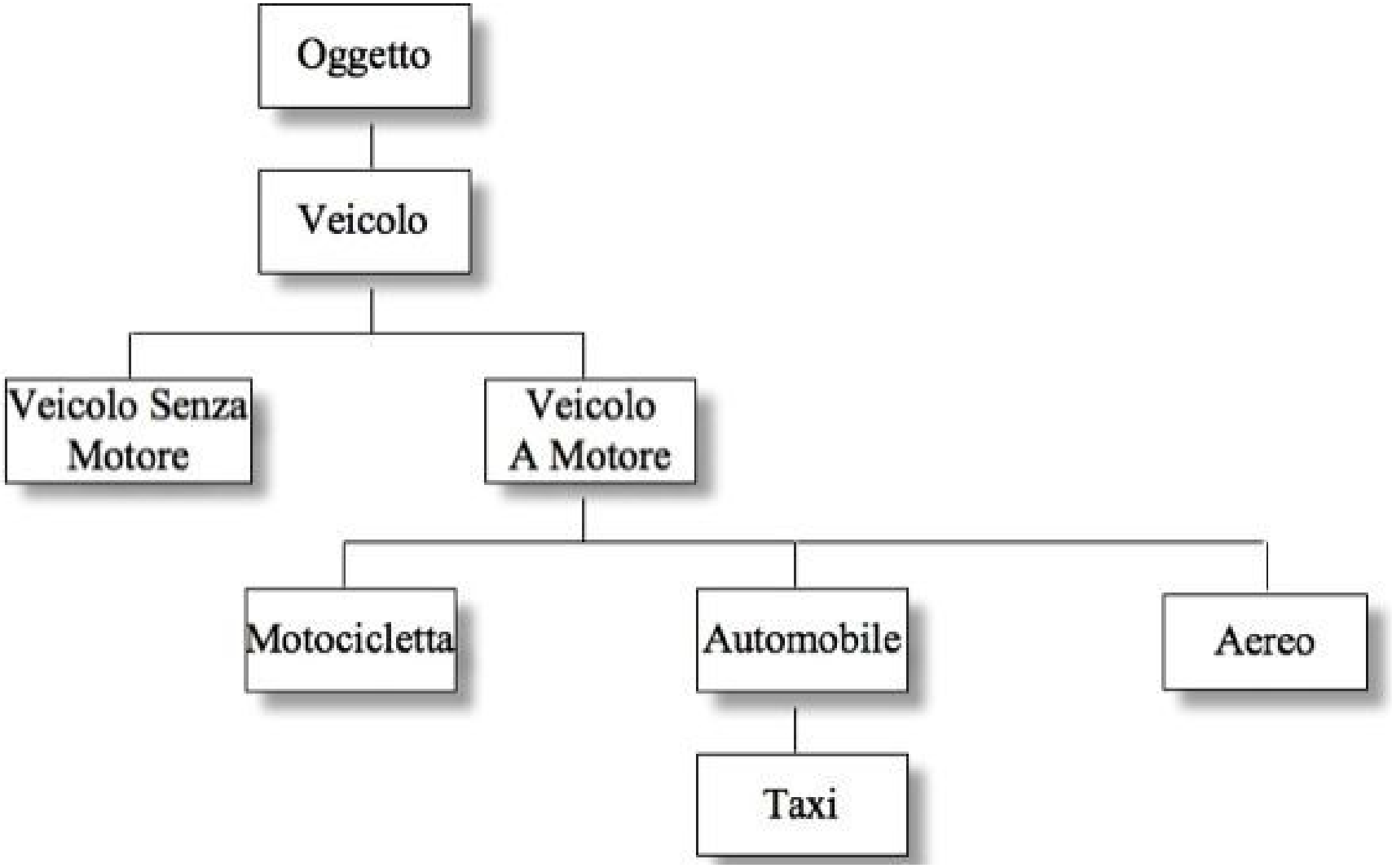
Dalla Teoria alla Pratica.

Costrutti e sintassi del linguaggio

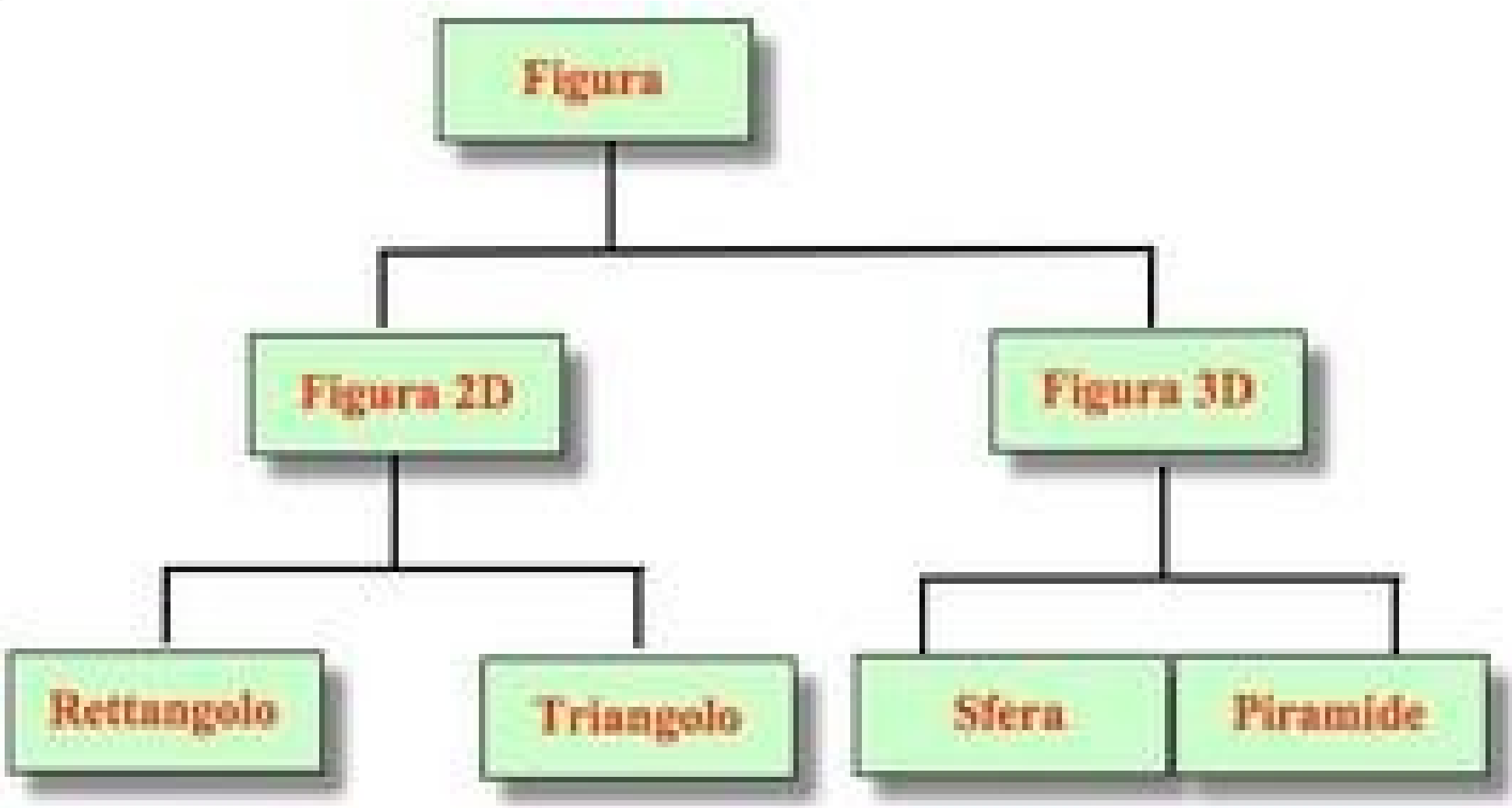
Ripercorriamo i concetti teorici visti aggiungendo un pò di pratica....

....Ereditarietà... Polimorfismo... Incapsulamento... Genericità....


Generalizzazione
Specializzazione


Oggetto
Veicolo
Veicolo Senza Motore
Veicolo A Motore
Motocicletta
Automobile
Aereo
Taxi

Dalla Teoria alla Pratica
Costrutti e Sintassi del linguaggio



Esempio 1 di Classe Taxi

```
public class Taxi {
    public String targa = “ABC123”;

   public void start () {
      System.out.println ("BROOOOMMMMM") ;
   }
}
```

Esempio 2 di Classe Taxi

```
public class Veicolo {
    public String targa = “ABC123”;

   public void start () {
      System.out.println ("BROOOOMMMMM") ;
   }
}

public class Taxi extends Veicolo {

}
```

Il principio del polimorfismo si realizza "praticamente" nel momento in cui andiamo a specializzare lo stesso metodo dichiarato o ereditato da una classe padre, differenziandolo per ciascuna classe figlia.

CENNI SU

CLASSI ASTRATTE

ED

INTERFACCE

L'Astrazione dei Dati viene applicata per decomporre sistemi software complessi in componenti più piccoli e semplici che possono essere gestiti con maggiore facilità ed efficienza. In breve, essa viene utilizzata per gestire al meglio la complessità di un programma.

Classe Astratta cos'è?

E' (soprattutto) un modello, dal quale poi estrapolare classi derivate più dettagliate e specifiche.

Classe Astratta cos'è? (pt.2)

Ma è anche una classe padre, nonchè un contenitore di metodi(definiti o anche solo dichiarati) e attributi che ereditano, ed eventualmente specializzano le classi figlie.

- Java ci consente di definire classi in cui uno o più metodi non sono implementati, ma solo dichiarati
- Questi metodi sono detti astratti e vengono marcati con la parola chiave abstract
- Non hanno un corpo tra parentesi graffe ma solo la dichiarazione terminata con ;
- Una classe che ha almeno un metodo astratto si dice classe astratta
- Una classe astratta deve essere marcata a sua volta con la parola chiave abstract
- Una classe astratta non può essere istanziata
- Una classe astratta serve come superclasse comune per un insieme di sottoclassi concrete.

```
public abstract class Veicolo {

   public String targa ;

   public void start () {
      System.out.println ("BROOOOMMMMM") ;
   }

   public abstract void stop () ;
}
```

Java, per semplicità, permette solo l'ereditarietà singola.
Semplice, ma limitativa, soprattutto nel caso in cui si devono duplicare dei rami della gerarchia.

Si arriva per questo al concetto di interfaccia.

L'interfaccia è una collezione di metodi(solamente dichiarati) che possono essere aggiunti ad una classe per fornire ulteriori funzionalità(oltre a quelli propri ed a quelli ereditati dalla superclasse)

Una classe Java, pur potendo avere una sola superclasse, può implementare un qualunque numero di interfacce.

Implementare un interfaccia vuol dire fornire le implementazioni dei metodi in essa contenuti.

```
package test;

public interface VeicoliADieselInterface {

    public final String DIESEL = "vado a diesel" ;

    public void scaldareCandele () ;

    public void fareIlPienoDiDiesel() ;

}
```

INCAPSULAMENTO

Il principio dell'incapsulamento, nella programmazione in generale, consiste nella tecnica di "nascondere"/inglobare un particolare comportamento/funzionamento di una parte del programma, in modo da prevenire, nel futuro, doverosi e sostanziosi cambiamenti(al codice), resisi necessari da malfunzionamenti o da necessari cambi di implementazione.

In particolare in Java, l'incapsulamento si realizza mediante:

- realizzazione di interfacce;

- creazione di metodi set e get per l'accesso agli attributi di classi(che diventano privati, e quindi non accessibili direttamente).

CENNO SU CLASSI INTERNE

E' possibile definire una classe anche internamente ad un'altra.

Una classe interna può essere definita in qualunque ambito; cioè come membro di un'altra classe, all'interno di un blocco di istruzioni oppure all'interno di un'espressione, in modo anonimo. Questa modifica dona a Java una maggiore struttura a blocchi.

MODIFICATORI DI ACCESSO

I modificatori di accesso permettono di definire la visibilità di una classe, di un metodo e di un attributo.

I modificatori di accesso sono:

- public
- protected
- private

MODIFICATORI DI ACCESSO pt.2

public: l'accesso ad una caratteristica public è consentito a tutte le altre classi. Può essere utilizzato sia per classi, per metodi che per attributi.

private: l'accesso è consentito solo agli elementi della classe stessa. Può essere usato solo per metodi e attributi. Gli elementi private, tranne i costruttori, vengono sempre ereditati anche se la loro esistenza è sconosciuta all'oggetto che li eredita.

protected, l'accesso è consentito a tutte le classi nello stesso pacchetto e a tutte le sottoclassi. Può essere applicato solo a metodi e attributi.

I MODIFICATORI FINAL E STATIC

Il modificatore final può essere attribuito ad una variabile/attributo, che di fatto lo farà diventare una costante.

Final può essere applicato anche ai metodi.
Un metodo definito come final (finale non modificabile) implica che se eredito la classe che contiene il metodo su questo non potrà essere eseguito l'override.

Anche una classe può essere definita come final.
Una classe definita in questo modo non potrà essere ereditata.
Ciò è logicamente comprensibile in quanto se dichiaro una classe con il modificatore final intendo una classe finale ovvero una classe completa che non necessita di specializzazioni o estensioni e dunque è più che logico che non sia possibile ereditarla.

Il modificatore static può essere applicato solo a metodi e a variabili di una classe.

L'effetto di dichiarare un metodo static è quello di rendere il metodo di fatto comune a tutte le istanze della classe.

L'effetto di dichiarare una variabile static è quello di definire una variabile globale accessibile a tutte le istanze di una classe.

Esempio di chiamata di metodo statico:

Math.max(3,4);

Esempio di chiamata di variabile statica:

```
public class Prova {
  public static int numero;
}
public class Implementazione {

  public static void main (String args[]) {
    Prova prova1 = new Prova();
    Prova prova2 = new Prova();
    prova1.numero=10;
    System.out.println("il valore numero per  l'istanza 1 è: " + prova1.numero);
    System.out.println("il valore numero per l'istanza 2 è: " + prova2.numero);
  }
}
```

Esempio di chiamata di variabile statica(pt. 2):

il valore numero per l'istanza 1 è 10;
il valore numero per l'istanza 2 è 10;

Vale a dire che la classe

```
public class Pippo {
}
```

e la classe

```
public class Pippo {
    public Pippo () {}
}
```

si equivalgono.

1) testAlbero.famiglia = “betulla” ;

2) testAlbero.potaAlbero() ;

...riprendendo il concetto di polimorfismo.....

Vediamo un esempio di applicazione del polimorfismo in java, relativamente ai metodi...

Dal punto di vista implementativo il polimorfismo per i metodi si ottiene utilizzando l'overload e l'override dei metodi stessi.

Un metodo è univocamente determinato se prendiamo in considerazione sia il suo nome che la lista di tutti i suoi parametri.
Ciò permette, di fatto, di avere all'interno di una classe più metodi che hanno lo stesso nome, ma parametri diversi.

L'overload quindi si concretizza nella scrittura di più metodi identificati dallo stesso nome che però hanno, in ingresso, parametri di tipo e numero diverso.

esempio

Il termire override indica invece la riscrittura di un certo metodo, ereditato da una classe padre.

Dunque, necessariamente, l'override implica anche l'ereditarietà.

esempio

La dichiarazione alloca uno spazio fisico della memoria a quella determinata variabile.

L'inizializzazione invece è proprio l'assegnazione del dato a quello spazio di memoria creato.

GLI OPERATORI

Gli operatori sono simboli speciali utilizzati nelle espressioni.
Abbiamo tre tipologie di operatori:

- Operatori aritmetici
- Operatori di assegnazione
- Operatori logici

OPERATORI ARITMETICI[...]

\+ somma

\- sottrazione

* moltiplicazione

/ divisione

% modulo

OPERATORI DI ASSEGNAZIONE[...]

= assegnazione

++ incremento

-- decremento

OPERATORI LOGICI [...] pt.1

== uguale a

!= diverso da

< minore di

> maggiore di

<= minore uguale di

>= maggiore uguale di

OPERATORI LOGICI [...] pt.2

& operatore AND vera se entrambe vere

&& doppio operatore AND il confronto ha termine alla prima espressione falsa che viene incontrata

| operatore OR vera se almeno una è vera

|| doppio operatore OR il confronto ha termine alla prima espressione vera che viene incontrata

COSTRUTTI DEL LINGUAGGIO

Si intende, per costrutti del linguaggio, quelle strutture sintattiche proprie del linguaggio in oggetto.

- Istruzioni condizionali
- Istruzione di ciclo incondizionato
- Istruzione di ciclo condizionato

ISTRUZIONI CONDIZIONALI

Le istruzioni condizionali consentono di decidere le operazioni da eseguire in funzione del verificarsi o meno di certe condizioni.
In Java esistono due tipi di istruzioni condizionali:

- if
- switch

L'istruzione if utilizza una espressione booleana(singola o articolata, tramite gli operatori logici) per decidere se debba essere eseguita una certa istruzione.

Esempi:

```
if (espressione) {
    istruzioni
}
```

Altri esempi:

```
if (espressione) {
    istruzioni1
} else {
    istruzioni2
}
```

Oppure

```
if (espressione1) {
    istruzioni1
} else if (espressione2) {
    istruzioni2
} else if (espressione3) {
    istruzioni3
} else
```

L'istruzione switch viene utilizzata per eseguire istruzioni in base al valore di un'espressione.

l'istruzione switch valuta l'espressione ed esegue le istruzioni elencate in corrispondenza dell'istruzione case che fornisce il valore corrispondente a quello assunto dall'espressione.

Infine è possibile specificare delle istruzioni di default da eseguire nel caso che nessuno dei case elencati corrispondano al valore assunto dall'espressione; questa opzione può essere specificata aggiungendo l'istruzione default dopo l'ultimo case.

Esempio:

```
switch(espressione) {
    case valore1 : istruzioni1;
    break;
    case valore2 : istruzioni2;
    break;
    [...]
    default : istruzioniXXX;
}
```

Esempio:

```
switch(parametroInIngresso) {
   case 1:
      System.out.println("Il mio valore è 1");
      break;
   case 2:
      System.out.println("Il mio valore è 2");
      break;
   [...]
   default :
      System.out.println("Nessuno di questi valori è valido.");
}
```

ISTRUZIONI DI CICLO NON CONDIZIONALI

Il ciclo mediante l'istruzione for permette di ripetere una serie di istruzioni per un numero prefissato di volte.

Il ciclo non condizionale si esprime così:

```
for (inizializzazione; condizione; incremento) {
    blocco di istruzioni
}
```

Esempio:

```
for (int i=0; i<10; i++) {
    System.out.println("L'indice è in posizione " + i);
}
```

ISTRUZIONI DI CICLO CONDIZIONALI

I cicli condizionali, eseguono un blocco di istruzioni finchè una data espressione/condizione è valida.
I cicli condizionali in java si realizzano mediante l'uso di due tipologie di istruzioni:

- while
- do-while

Il ciclo while esegue ripetutamente un'istruzione (o un blocco) per tutto il tempo in cui la data condizione si mantiene vera.
La condizione è un'espressione booleana

```
while (condizione) {
  blocco istruzioni;
}
```

Nel ciclo do-while la condizione viene valutata in coda al ciclo; in tal modo il corpo viene eseguito almeno una volta.

```
do {
   blocco istruzioni;
} while (condizione);
```

CONDIZIONI DI USCITA DAI CICLI

Da un ciclo si esce generalmente quando la condizione(espressione booleana) diviene falsa.
Questo è il modo normale di terminazione del ciclo.
Può essere però utile uscire dal ciclo prima che ciò accada.
In tali casi potremo adoperare le istruzioni break e continue.

break

termina immediatamente l'esecuzione del ciclo.
Nel caso di cicli annidati, si passa al ciclo più esterno.
Termina quindi solo il ciclo nel quale è direttamente contenuta l'istruzione break.

continue

interrompe l'esecuzione della ripetizione in corso, ricominciando dalla successiva (senza uscire quindi dal ciclo).

Esempio:

```
for (int i = 1; i <= 100; i++) {
        if (i%dividendo != 0) {
              continue;
        }
        System.out.println("prova");
        quanti++;
        if (quanti == limite) {
            break;
        }
}
```

CASTING

Il casting rappresenta l'operazione di conversione di tipo tra oggetti e tra tipi primitivi.

Il casting è consentito:

- tra due tipi primitivi (int e float per esempio)
- tra due oggetti legati da un legame di ereditarietà (superclasse e sottoclasse)

Esempio di conversione tra tipi primitivi:

```
int i;
long l;
float f;
double d;

i = (int) l;
f = (float) d;
```

Esempio di conversione tra oggetti:

```
Mammifero essere;
Cane fido = new Cane () ;

essere = (Mammifero) fido;
```

ECCEZIONI

Un'eccezione è un evento, inatteso, che si verifica durante l'esecuzione di un programma e ne impedisce la normale prosecuzione.

Java offre un meccanismo che aiuta il programmatore a scrivere applicazioni robuste, permettendo di definire degli exception handler, ovvero porzioni di codice che reagiscono ad uno specifico tipo di eccezione consentendo al programma di proseguire dopo averla opportunamente gestita.

Le eccezioni in Java sono oggetti appartenenti alla gerarchia delle classi: quelle predefinite dal linguaggio hanno origine dalla superclasse Throwable, mentre quelle create dal programmatore estendono la classe Exception.
Un oggetto eccezione indica il tipo di errore e contiene alcune altre informazioni, come ad es. il punto in cui è avvenuto l'errore.

Ecco come si concretizza l'exception handler in java:

```
try {
      Istruzioni1;
} catch (TipoEccezione1 varEccezioneX) {
      Istruzioni2;
} catch (TipoEccezione2 varEccezioneY) {
      Istruzioni3;
} finally {
      Istruzioni4;
}
```

Asserzioni

Un'asserzione può essere definita come una condizione che deve essere verificata affinché lo sviluppatore consideri corretta una parte di codice.

A differenza delle eccezioni, le asserzioni rappresentano uno strumento da abilitare per testare la robustezza del software, ed eventualmente disabilitare in fase di rilascio.

Questo concetto è implementato tramite la parola chiave assert

Una asserzione è una espressione booleana che deve essere "true" se e solo se il codice sta funzionando correttamente.

Se l'asserzione risulta falsa, viene segnalato l'errore e bisogna correggere il codice

```
public class EsempioAssertion {
   public void mioMetodo(int valore) {
      Object foo = null;
      // ... elaboro l'oggetto foo
      // Controllo che l'oggetto sia valorizzato (
      postcondition
      )
      assert foo != null: "Impossibile ottenere foo con valore " + valore;
   }

   public static void main(String[] args) {
      EsempioAssertion mioOggetto = new EsempioAssertion();
      mioOggetto.mioMetodo(10);
   }
}
```

Come si abilitano?

Per default le asserzioni sono disabilitate.

Bisogna quindi esplicitamente abilitarle. Come....?

>java -ea <classe da eseguire></classe>

In questo modo abilitiamo l'uso delle asserzioni nella nostra classe e in tutte le sue dipendenze, ma non nelle classi di sistema (quelle caricate direttamente dalla JVM). Per abilitare anche quelle ci tocca scrivere:

>java -esa <classe da eseguire></classe>

Collection

Una Collection - a volte chiamata contenitore(container) - è semplicemente un oggetto che raggruppa più elementi in una singola unità.

Le collezioni sono utilizzati per memorizzare, recuperare, manipolare e comunicare i dati aggregati.

Tipicamente, essi rappresentano elementi di dati che formano un gruppo naturale.

Fra le Collections distinguamo:

Interfacce: definizioni di interfacce che permettono di manipolare collezioni di oggetti indipendentemente dai dettagli implementativi

Implementazioni: classi concrete che implementano le varie interfacce

Le principali interfacce sono Collection, List, Set e Map.

Collection è l'interfaccia più generica, non definisce nè l'ordine in cui sono memorizzati gli elementi nè se ci possono essere elementi duplicati.

Un oggetto List contiene oggetti ordinati in base all'ordine di inserimento, può contenere duplicati e permette di inserire e ottenere gli elementi in base all'indice. E' la versione "evoluta" di un Array in quanto non contiene la limitazione della dimensione massima prefissata.

ArrayList è l'implementazione di List che memorizza gli elementi in un Array, si occupa della gestione della dimensione dell'Array in modo trasparente per lo sviluppatore. Ogni volta che l'Array è pieno viene invocato il metodo ensureCapacity che crea un nuovo Array di dimensione (old *3) / 2 +1 in cui vengono copiati tutti gli elementi presenti usando il metodo Arrays.copyOf.

Generics

Introdotto con la jdk 1.5, il meccanismo delle Generics ci consente di definire il tipo di parametro che vogliamo utilizzare.
La sua importanza risulta evidente nel caso di oggetti contenitore, come le Collection.

Il tutto avviene definendo il tipo di parametro dentro parentesi angolari (<>).

Questo permette di definire il parametro che il compilatore dovrà controllare.

Questo tipo di collezioni sono anche note come Type-Safe, in quanto non sarà possibile utilizzare sia in fase di inserimento che in fase di lettura tipi diversi da quelli definiti dal parametro.

```
public static void test(){
   Collection<String> lista = new ArrayList<String>();

   lista.add("Ciao ");
   lista.add("Mondo");
   //segnala errore in fase di compilazione
   lista.add(new Integer(5));

   Iterator<String> it=lista.iterator();
   while(it.hasNext()){
      //Non è necessario definire il tipo di ritorno
      //in quanto il parametro è già definito
      String tmp = it.next();
      //...
   }
}
```