ITS TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

NOME CORSO

Unità Formativa (UF) Programmazione Java

Docente: Mauro Bogliaccino

Titolo argomento: La programmazione orientata agli oggetti in Java

La programmazione orientata agli oggetti in Java

1. Ambienti di sviluppo (JDK) e primi approcci al codice
2. Le basi della programmazione object oriented: classi e oggetti
3. Variabili, attributi, metodi e costruttori
4. Identificatori, tipi di dati e array
5. Operatori e gestione del flusso di esecuzione
6. Costrutti di programmazione semplice: if, operatore ternario, while
7. Costrutti di programmazione avanzati: for, do while, for migliorato, switch
8. Classi ed oggetti
9. Classi innestate, classi anonime

1) JAVA FONDAMENTALE

- Java è un linguaggio di alto livello e orientato agli oggetti, creato dalla Sun Microsystem nel 1995.

Le motivazioni, che guidarono lo sviluppo di Java, erano quelle di creare un linguaggio semplice e familiare.

Le caratteristiche del linguaggio di programmazione Java sono:

- La tipologia di linguaggio **orientato agli oggetti** (ereditarietà, polimorfismo, ...)
- la **gestione della memoria** effettuata automaticamente dal sistema, il quale si preoccupa dell'allocazione e della successiva de-allocazione della memoria (il programmatore viene liberato dagli obblighi di gestione della memoria
- la **portabilità**, cioè la capacità di un programma di poter essere eseguito su piattaforme diverse senza dover essere e modificato e ricompilato

---

Caratteristiche di Java

- Semplice e familiare
- Orientato a oggetti
- Indipendente dalla piattaforma
- interpretato

- Sicuro
- Robusto
- Distribuito e dinamico
- Multi-thread

Semplice e familiare

- Basato su C
- Sviluppato da zero
- Estremamente semplice: senza puntatori, macro, registri
- Apprendimento rapido
- Semplificazione della programmazione
- Riduzione del numero di errori

Orientato a oggetti

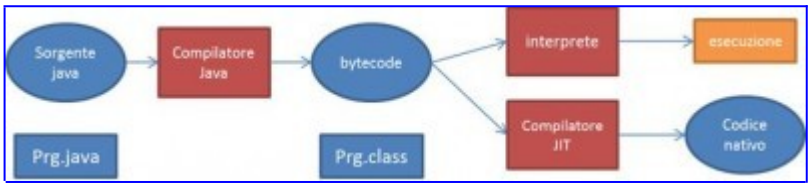
- Orientato a oggetti dalla base
- In Java tutto è un oggetto
- Incorpora le caratteristiche
- Incapsulamento
- Polimorfismo
- Ereditarietà
- Collegamento dinamico
- Non sono disponibili
- Ereditarietà multipla
- Overload degli operatori

Indipendente dalla piattaforma

- Più efficiente di altri linguaggi interpretati
- Soluzione: la macchina virtuale
- JVM (non è proprio la JVM)
- Linguaggio macchina bytecodes

Interpretato

- Il bytecode deve essere interpretato



- Vantaggi rispetto ad altri linguaggi interpretati
- Codice più compatto
- Efficiente
- Codice confidenziale (non esposto)

---

sicuro

- Supporta la sicurezza di tipo sandboxing
- Verifica del bytecode
- Altre misure di sicurezza
- Caricatore di classi
- Restrizioni nell'accesso alla rete

Robusto

- L'esecuzione nella JVM impedisce di bloccare il sistema
- L'assegnazione dei tipi è molto restrittiva
- La gestione della memoria è sempre a carico del sistema
- Il controllo del codice avviene sia a tempo di compilazione sia a tempo di esecuzione (runtime)

Distribuito e dinamico

- Disegnato per un'esecuzione remota e distribuita
- Sistema dinamico
- Classe collegata quando è richiesta

- Può essere caricata via rete
- Dinamicamente estensibile
- Disegnato per adattarsi ad ambienti in evoluzione

Multi-thread

- Soluzione semplice ed elegante per la multiprogrammazione
- Un programma può lanciare differenti processi
- Non si tratta di nuovi processi, condividono il codice e le variabili col processo principale
- Simultaneamente si possono svolgere vari compiti

2) Classi Java

Le classi estendono il concetto di "struttura" di altri linguaggi

3) Definiscono

- I dati (detti campi o attributi)
- Le azioni (metodi, comportamenti) che agiscono sui dati

4) Possono essere definite

- Dal programmatore (ex. Automobile)
- Dall'ambiente Java (ex. String, System, etc.)

5) La "gestione" di una classe avviene mediante

- Definizione della classe
- Instanziazione di Oggetti della classe

---

6) Creazione di classi in Java

- Definire i termini oggetto e classe
- Descrivere la forma nella quale possiamo creare nuove classi in Java
- mostrare come, una volta creata una classe possiamo creare oggetti di questa classe e utilizzarli

7) Struttura di una classe

```
public class HelloWorld {

    public static void main(String[] args) {
                System.out.println("Hello World");
    }

}
```

---

8) Java è un linguaggio orientato agli oggetti

- In Java quasi tutto è un oggetto
- Come definire classi e oggetti in Java?
- Classe: codice che definisce un tipo concreto di oggetto, con proprietà e comportamenti in un unico file
- Oggetto: istanza, esemplare della classe, entità che dispone di alcune proprietà e comportamenti propri, come gli oggetti della realtà
- In Java quasi tutto è un oggetto, ci sono solo due eccezioni: i tipi di dato semplici (tipi primitivi) e gli array (un oggetto trattato in modo *particolare*)
- Le classi, in quanto tipi di dato strutturati, prevedono usi e regole più complessi rispetto ai tipi semplici

---

9) Le classi in Java

- Le classi, in quanto tipi di dato strutturati, prevedono usi e regole più complessi rispetto ai tipi semplici
- Il primo passo per definire una classe in Java è creare un file che deve chiamarsi esattamente come la classe e con estensione .java
- Java permette di definire solo una classe per ogni file
- Una classe in Java è formata da:
- Attributi: (o campi/proprietà) che immagazzinano alcune informazioni sull'oggetto. Definiscono lo stato dell'oggetto
- Costruttore: metodo che si utilizza per inizializzare un oggetto
- Metodi: sono utilizzati per modificare o consultare lo stato di un oggetto. Sono equivalenti alle funzioni o procedure di altri linguaggi di programmazione

---

Incapsulamento e visibilità in Java

- Quando disegniamo un software ci sono **due aspetti** che risultano fondamentali:
- Interfaccia: definita come gli **elementi che sono visibili dall'esterno**, come il sw può essere utilizzato
- Implementazione: definita definendo alcuni attributi e scrivendo il codice dei differenti metodi per leggere e/o scrivere gli attributi

---

10) Incapsulamento

- L'incapsulamento consiste nell'**occultamento degli attributi** di un oggetto in modo che possano essere **manipolati solo attraverso metodi** appositamente implementati. p.es la proprietà `saldo` di un oggetto `conto corrente`
- Bisogna fare in modo che l'interfaccia sia più indipendente possibile dall'implementazione
- In Java l'incapsulamento è strettamente relazionato con la visibilità

---

11) Visibilità

- Per indicare la visibilità di un elemento (attribuito o metodo) possiamo farlo precedere da una delle seguenti parole riservate
- `public`: accessibile da qualsiasi classe
- `private`: accessibile solo dalla classe attuale
- `protected`: solo dalla classe attuale, le discendenti e le classi del nostro package
- Se **non indichiamo la visibilità**: sono accessibili **solo dalle classi del nostro package**

---

12) Accesso agli attributi della classe

- Gli attributi di una classe sono strettamente relazionati con la sua implementazione.
- Conviene contrassegnarli come `private` e impedirne l'accesso dall'esterno
- In futuro potremo cambiare la rappresentazione interna dell'oggetto senza alterare l'interfaccia
- Quindi non permettiamo di accedere agli attributi!
- per consultarli e modificarli aggiungiamo i metodi accessori e mutatori: `getters` e `setters`

---

13) Modifica di rappresentazione interna di una classe

- Uno dei maggiori vantaggi di occultare gli attributi è che in futuro potremo cambiarli senza la necessità di cambiare l'interfaccia
- Un linguaggio di programmazione **ORIENTATO AGLI OGGETTI** fornisce meccanismi per definire nuovi tipi di dato basati sul concetto di classe
- Una classe definisce un insieme di oggetti (conti bancari, dipendenti, automobili, rettangoli, ecc...).
- Un oggetto è una struttura dotata di proprie **variabili** (che rappresentano il suo stato) propri **metodi** (che realizzano le sue funzionalità)

Classi e documentazione

- Come la maggior parte dei linguaggi di programmazione, Java è dotato di una libreria di classi "pronte all'uso" che coprono molte esigenze
- Usare classi già definite da altri è la norma per non sprecare tempo a risolvere problemi già risolti o a reinventare la ruota (DRY)
- La libreria Java standard è accompagnata da documentazione che illustra lo scopo e l'utilizzo di ciascuna classe presente,
- Dalla versione 9 di Java la libreria è stata divisa in moduli
- Documentazione Java 8
- Documentazione Java 9
- Documentazione Java 11
- Documentazione Java 13

Definizione di una Classe

14) Definizione

```
class <nomeClasse) {
      <campi>
      <metodi>
}
```

15) Esempi

- Classe contenente dati ma non azioni

```
class DataOnly {
      boolean b;
      char c;
      int i;
```

```
      float f;
}
```

- Classe contenente dati e azioni

```
class Automobile {

    //Attributi
    String colore;
    String marca;
    boolean accesa;

    //metti i in moto
    void mettiInMoto() {
        accesa = true;
    }

    //vernicia
    void vernicia (String nuovoCol) {
        colore = nuovoCol;
    }

    // stampaStato
    void stampaStato () {
        System.out.println("Questa automobile è una "+ marca + " " +
colore);
        if (accesa)
            System.out.println("Il motore è acceso");
        else
            System.out.println("Il motore è spento");
    }
}
```

16) Dati & Metodi

- Public: visibili all'esterno della classe
- Private: visibili solo dall'interno della classe
- Protected: ...
- Nessuna specifica (amichevole): ...

La definizione di classe non rappresenta alcun oggetto.

17) Espressioni aritmetiche

```
public class Triangolo {
    public static void main ( String [] args ) {
        System.out.println (5*10/2);
```

```
    }
}
```

Il programma risolve l'espressione 5*10/2 e stampa il risultato a video

---

18) Espressioni aritmetiche e precedenza

singoli "letterali"

- Letterali interi: 3425, 12, -34, 0, -4, 34, -1234, ...
- Letterali frazionari: 3.4, 5.2, -0.1, 0.0, -12.45, 1235.3423, ...

operatori aritmetici

- moltiplicazione *
- divisione /
- modulo % (resto della divisione tra interi)
- addizione +
- sottrazione -

Le operazioni sono elencate in **ordine decrescente di priorità** ossia 3+2*5 fa 13, non 25

Le parentesi tonde cambiano l'ordine di valutazione degli operatori ossia (3+2)*5 fa 25

Inoltre, tutti gli operatori sono associativi a sinistra ossia 3+2+5 corrisponde a (3+2)+5 quindi 18/6/3 fa 1, non 9

---

19) operazione di divisione

- L'operazione di divisione / si comporta diversamente a seconda che sia applicato a letterali interi o frazionari
- 25/2 = 12 (divisione intera)
- 25%2 = 1 (resto della divisione intera)
- 25.0/2.0 = 12.5 (divisione reale)
- 25.0%2.0 = 1.0 (resto della divisione intera)
- Una operazione tra un letterale intero e un frazionario viene eseguita come tra due frazionari
- 25/2.0 = 12.5
- 1.5 + (25/2) = 13.5 (attenzione all'ordine di esecuzione delle operazioni)
- 2 + (25.0/2.0) = 14.5

20) OPERATORI ARITMETICI, RELAZIONALI, DI ASSEGNAZIONE

- Di assegnazione: = += -= *= /= &= |= ^=
- Di assegnazione/incremento: ++ -- %=

Operatore	Significato
=	assignment
+=	addition assignment
-=	subtraction assignment
*=	multiplication assignment
/=	division assignment
%=	remainder assignment

- Operatori Aritmetici: + - * / %

Operatore	Significato
+	addition
-	subtraction
*	multiplication
/	division
%	remainder
++var	preincrement
--var	predecrement
var++	postincrement
var--	postdecrement

- Relazionali: == != > < >= <=

Operatore	Significato
<	less than

Operatore	Significato
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal

---

21) Operatori per Booleani

- Bitwise (interi): & | ^ << >> ~

Operatore	Significato
&&	short circuit AND
||	short circuit OR
!	NOT
^	exclusive OR

Attenzione:

- Gli operatori logici agiscono solo su booleani
  - Un intero NON viene considerato un booleano
  - Gli operatori relazionali forniscono valori booleani

Operatori su reference

22) Per i puntatori/reference, sono definiti:

- Gli operatori relazionali == e !=
  - N.B. test sul riferimento **NON** sull'oggetto
- Le assegnazioni
- L'operatore "punto"
- NON è prevista l'aritmetica dei puntatori

Operatori matematici

23) Operazioni matematiche complesse sono permesse dalla classe Math (package java.lang)

- `Math.sin (x) calcola sin(x)`
- `Math.sqrt (x) calcola x^(1/2)`
- `Math.PI ritorna pi`
- `Math.abs (x) calcola |x|`
- `Math.exp (x) calcola e^x`
- `Math.pow (x, y) calcola x^y`

Esempio

- `z = Math.sin (x) – Math.PI / Math.sqrt(y)`

Caratteri speciali

Literal	Represents
`\n`	New line
`\t`	Horizontal tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\\`	Backslash
`\"`	Double quote
`\ddd`	Octal character
`\xdd`	Hexadecimal character
`\udddd`	Unicode character

24) LE VARIABILI

- Una variabile è un'area di memoria identificata da un nome
- Il suo scopo è di contenere un valore di un certo tipo
- Serve per memorizzare dati durante l'esecuzione di un programma

- Il nome di una variabile è un identificatore
- può essere costituito da lettere, numeri e underscore
- non deve coincidere con una parola chiave del linguaggio
- è meglio scegliere un identificatore che sia significativo per il programma

25) esempio

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

        int base , altezza ;
        int area ;

        base = 5;
        altezza = 10;
        area = base * altezza / 2;

        System.out.println ( area );
    }
}
```

Usando le variabili il programma risulta essere più chiaro:

- Si capisce meglio quali siano la base e l'altezza del triangolo
- Si capisce meglio che cosa calcola il programma

---

26) Dichiarazione

- In Java ogni variabile deve essere dichiarata prima del suo uso
- Nella dichiarazione di una variabile se ne specifica il nome e il tipo
- Nell'esempio, abbiamo dichiarato tre variabili con nomi base, altezza e area, tutte di tipo int (numeri interi)
    - int base , altezza ;
    - int area ;

ATTENZIONE! Ogni variabile deve essere dichiarata UNA SOLA VOLTA (la prima volta che compare nel programma)

```
base =5;
altezza =10;
area = base * altezza /2;
```

---

27) Assegnamento

- Si può memorizzare un valore in una variabile tramite l'operazione di assegnamento

- Il valore da assegnare a una variabile può essere un letterale o il risultato della valutazione di un'espressione
- Esempi:

```
base =5;
altezza =10;
area = base * altezza /2;
```

- I valori di base e altezza vengono letti e usati nell'espressione
- Il risultato dell'espressione viene scritto nella variabile area

---

28) Dichiarazione + Assegnamento

Prima di poter essere usata in un'espressione una variabile deve:

- essere stata dichiarata
- essere stata assegnata almeno una volta (inizializzata)
- NB: si possono combinare dichiarazione e assegnamento.

Ad esempio:

```
int base = 5;
int altezza = 10;
int area = base * altezza / 2;
```

Costanti

Nella dichiarazione delle variabili che **NON DEVONO** mai cambiare valore si può utilizzare il modificatore **final**

```
final double IVA = 0.22;
```

- Il modificatore **final** trasforma la variabile in una costante
- Il compilatore si occuperà di controllare che il valore delle costanti non venga mai modificato (ri-assegnato) dopo essere stato inizializzato.
- Aggiungere il modificatore **final** non cambia funzionamento programma, ma serve a prevenire errori di programmazione
- Si chiede al compilatore di controllare che una variabile non venga ri-assegnata per sbaglio
- Sapendo che una variabile non cambierà mai valore, il compilatore può anche eseguire delle ottimizzazioni sull'uso di tale variabile.

---

29) Input dall'utente

- Per ricevere valori in input dall'utente si può usare la classe Scanner, contenuta nel package **java.util**
- La classe Scanner deve essere richiamata usando la direttiva import prima dell'inizio del corpo della classe

30) TIPI DI DATO PRIMITIVI

- In un linguaggio ad oggetti puro, vi sono solo classi e istanze di classi:
- i dati dovrebbero essere definiti sotto forma di oggetti

Java definisce alcuni tipi primitivi

- Per efficienza Java definisce dati primitivi
- La dichiarazione di una istanza alloca spazio in memoria
- Un valore è associato direttamente alla variabile
- (e.g, i == 0)
- Ne vengono definiti dimensioni e codifica
- Rappresentazione indipendente dalla piattaforma

Tabelle riassuntive: tipi di dato

Primitive Data Types

type	bits
byte	8 bit
short	16 bit
int	32 bit
long	64 bit
float	32 bit
double	64 bit
char	16 bit
boolean	true/false

I caratteri sono considerati interi

---

31) I tipi numerici, i char

- Esempi
- `123` (int)
- `256789L` (L o l = long)
- `0567` (ottale) `0xff34` (hex)
- `123.75 0.12375e+3` (float o double)
- `'a' '%' '\n'` (char)
- `'\123'` (\ introduce codice ASCII)

32) Tipo boolean

- `true`
- `false`

Esempi

```
int i = 15;
long longValue = 10000000000001;
byte b = (byte)254;

float f = 26.012f;
double d = 123.567;
boolean isDone = true;
boolean isGood = false;
char ch = 'a';
char ch2 = ';';
```

```
public class Applicazione {

        public static void main(String[] args) {

                int mioNumero;
                mioNumero = 100;
                System.out.println(mioNumero);

                short mioShort = 851;
                System.out.println(mioShort);

                long mioLong = 34093;
                System.out.println(mioLong);

                double mioDouble = 3.14159732;
                System.out.println(mioDouble);
```

```
        float mioFloat = 324.4f;
        System.out.println(mioFloat);

        char mioChar = 'y';
        System.out.println(mioChar);

        boolean mioBoolean = true;
        System.out.println(mioBoolean);

        byte mioByte = 127;
        System.out.println(mioByte);
    }

}
```

Data Type	Bits	Minimum	Maximum
byte	8	-128	127
short	16	-32,768	32,767
int	32	-2,147,483,648	2,147,483,647
long	64	-9.22337E+18	9.22337E+18
float	32	See the docs	
double	64	See the docs	

Esempi gist

33) STRUTTURE DEL LINGUAGGIO

Selezione

if //Statements

```
if (condition) {

    //statements;

}

[optional]
else if (condition2) {

    //statements;
```

```
}


[optional]
else {

//statements;

}
```

---

switch Statements

```
switch (Expression) {

    case value1:

    //statements;

    break;

    ...

    case valuen:

    //statements;

    break;

    default:

    //statements;

}
```

---

loop Statements

```
while (condition) {

//statements;

}

do {

//statements;
```

```
} while (condition);


for (init; condition; adjustment) {

//statements;

}
```

Cicli definiti

Se il numero di iterazioni è prevedibile dal contenuto delle variabili all'inizio del ciclo.

Esempio: prima di entrare nel ciclo so già che verrà ripetuto 10 volte

```
int n=10;
for (int i=0; i<n; ++i) {
    ...
}
```

Cicli indefiniti

Se il numero di iterazioni non è noto all'inizio del ciclo.

Esempio: il numero di iterazioni dipende dai valori immessi dall'utente.

```
while(true) {
    x = Integer.parseInt(JOptionPane.showInputDialog("Immetti numero
positivo"));
    if (x > 0) break;
}
```

Cicli annidati

Se un ciclo appare nel corpo di un altro ciclo.

Esempio: stampa quadrato di asterischi di lato n

```
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) System.out.print("*");
    System.out.println();
}
```

Cicli con filtro

Vengono passati in rassegna un insieme di valori e per ognuno di essi viene fatto un test per verificare se il valore ha o meno una certa proprietà in base alla quale decideremo se prenderlo in considerazione o meno.

Esempio: stampa tutti i numeri pari fino a 100

```
for (int i=1; i<100; ++i) { // passa in rassegna tutti i numeri fra 1 e
100
    if (i % 2 == 0) // filtra quelli pari
        System.out.println(i);
}
```

Cicli con filtro e interruzione

Se il ciclo viene interrotto dopo aver filtrato un valore con una data proprietà.

Esempio: verifica se un array contiene o meno numeri negativi

```
boolean trovato = false;
for (int i=0; i<v.length; ++i) // passa in rassegna tutti gli indici
dell'array v
    if (v[i]<0) { // filtra le celle che contengono valori negativi
        trovato = true;
        break; // interrompe ciclo
    }
// qui trovato vale true se e solo se vi sono numeri negativi in v
```

Cicli con accumulatore

Vengono passati in rassegna un insieme di valori e ne viene tenuta una traccia cumulativa usando una opportuna variabile.

Esempio: somma i primi 100 numeri interi.

```
int somma = 0; // variabile accumulatore di tipo int
for (int i=1; i<100; ++i) { // passa in rassegna tutti i numeri fra 1 e
100
    somma = somma + i; // accumula i valori nella variabile accumulatore
}
```

Esempio: data una stringa s, ottieni la stringa rovesciata

```
String rovesciata = ""; // variabile accumulatore di tipo String
for (int i=0; i<s.length(); ++i) { // passa in rassegna tutti gli indici
dei caratteri di s
```

```
    rovesciata = s.substring(i, i+1) + rovesciata; // accumula i
caratteri in testa all'accumulatore
}
```

Cicli misti

Esempio di ciclo definito con filtro e accumulatore: calcola la somma dei soli valori positivi di un array

```
int somma = 0;
for (int i=0; i<v.length; ++i) // passa in rassegna tutti gli indici
dell'array v
    if (v[i]>0) // filtra le celle che contengono valori positivi
        somma = somma + v[i]; // accumula valore nella variabile
accumulatore
        ```
```

## 34) ARRAY

- Sequenze ordinate di
  - Tipi primitivi (int, float, etc.)
  - Riferimenti ad oggetti ( vedere classi! )
- Elementi dello stesso tipo
  - Indirizzati da indici
  - Raggiungibili con l'operatore di indicizzazione: le **parentesi quadre** []
  - Raggruppati sotto lo stesso nome

### 35) In Java gli array sono Oggetti

- Sono allocati nell'area di memoria riservata agli oggetti creati dinamicamente (heap)

---

### 36) Dimensione

- Può essere stabilita a run-time (quando l'oggetto viene creato)
- È fissa (non può essere modificata)
- E' nota e ricavabile per ogni array Array Mono-dimensionali (vettori)

## Dichiarazione di Array

### 37) Dichiarazione di un riferimento a un array

- `int[] voti;`
- `int voti[];`
```

38) La dichiarazione di un array non assegna alcuno spazio

```
voti == null
```

Creazione di un Array

39) L'operatore new crea un array:

- Con costante numerica

```
int[] voti;
...
voti = new int[10];
```

- Con costante simbolica

```
final int ARRAY_SIZE = 10;
int[] voti;
...
voti = new int[ARRAY_SIZE];
```

- Con valore definito a run-time

```
int[] voti;
... definizione di x (run-time) ...
voti = new int[x];
```

- L'operatore new inizializza le variabili
  - 0 - per variabili di tipo numerico (inclusi i char)
  - false - per le variabili di tipo boolean

40) Utilizzando un inizializzatore

(che permette anche di riempire l'array)

```
int[] primi = {2,3,5,7,11,13};
...
int [] pari = {0, 2, 4, 6, 8, 10,};
// La virgola finale e' facoltativa
// (elenchi lunghi)
```

- Dichiarazione e creazione possono avvenire contestualmente
- L'attributo length indica la lunghezza dell'array, cioè il numero di elementi
- Gli elementi vanno da 0 a length-1

```
for (int i=0; i<voti.length; i++)
voti[i] = i;
```

---

41) In Java viene fatto il bounds checking

- Maggior sicurezza
- Maggior lentezza di accesso

42) Il riferimento ad array

- Non è un puntatore al primo elemento
- È un puntatore all'oggetto array
- Incrementandolo non si ottiene il secondo elemento

Array di oggetti

Per gli array di oggetti (e.g., Integer) `Integer [] voti = new Integer [5];` ogni elemento e' un riferimento

43) L'inizializzazione va completata con quella dei singoli elementi

```
voti[0] = new Integer (1);
voti[1] = new Integer (2);
...
voti[4] = new Integer (5);
```

Array Multi-dimensionali (Matrici)

44) Array contenenti riferimenti ad altri array

Sintatticamente sono estensioni degli array a una dimensione

45) Sono possibili righe di lunghezza diverse

(matrice = array di array)

```
int[][] triangle = new int[3][]
```

46) Le righe non sono memorizzate in posizioni adiacenti

- Possono essere spostate facilmente

```
// Scambio di due righe
double[][] saldo = new double[5][6];
...
double[] temp = saldo[i];
saldo[i] = saldo[j];
saldo[j] = temp;
```

- L'array è una struttura dati efficiente ogni volta che il numero di elementi è noto
- Il ridimensionamento di un array in Java risulta poco efficiente
- Utilizzare altre strutture dati se il numerodi elementi contenuto non è noto

---

47) Il pacchetto java.util contiene metodi statici di utilità per gli array

- Copia di un valore in tutti gli (o alcuni) elementi di un array
  - `Arrays.fill (<array>, <value>);`
  - `Arrays.fill (<array>, <from>, <to>,<value>);`
- Copia di array
  - `System.arraycopy (<arraySrc>, <offsetSrc>,<arrayDst>, <offsetDst>, <#elements>);`
- Confronta due array
  - `Arrays.equals (<array1>, <array2>);`
- Ordina un array (di oggetti che implementino l'interfaccia Comparable)
  - `Arrays.sort (<array>);`
- Ricerca binaria (o dicotomica)
  - `Arrays.binarySearch (<array>);`

Snippets Tipi di Array

Array Monodimensionali

```
int[] list = new int[10];

list.length;

int[] list = {1, 2, 3, 4};
```

Array Multidimensionali

```
int[][] list = new int[10][10];

list.length;

list[0].length;

int[][] list = {{1, 2}, {3, 4}};
```

Array irregolari

```
int[][] m = {
    {1, 2, 3, 4},
    {1, 2, 3},
    {1, 2},
    {1}
};
```

---

esempi ed esercizi su array

48) CASTING E PROMOTION

- `( nometipo ) variabile`
- `( nometipo ) espressione`
- Trasforma il valore della variabile (espressione) in quello corrispondente in un tipo diverso
- Il cast si applica anche a `char`, visto come tipo intero positivo
- La promotion è automatica quando necessaria
  - Es. `double d = 3 + 4;`
- Il casting deve essere esplicito: il programmatore si assume la responsabilità di eventuali perdite di informazione
  - Per esempio
  - `int i = ( int ) 3.0 * ( int ) 4.5;` i assume il valore 12
  - `int j = ( int ) (3.0 * 4.5);` j assume il valore 13

---

casting dei tipi reference (oggetti)

- è permesso solo in caso di ereditarietà
- la conversione da sotto-classe a super-classe è automatica
- la conversione da super-classe a sotto-classe richiede cast esplicito

- la conversione tra riferimenti non in relazione tra loro non è permessa

esempio promotion

```
char a = 'a';
// promotion int è più grande e i valori sono compatibili
int b = a;

System.out.println(a); // a
System.out.println(b); // 97
```

---

esempi type casting

```
byte b = (byte) 261;
System.out.println(b); // 5

System.out.println(  Integer.toBynaryString(b)  ); // 101
System.out.println(  Integer.toBynaryString(261)  ); // 100000101

int a = (int) 1936.27;

System.out.println(a); // 1936
```

con il tipo boolean non si può fare il typecasting

```
int a = (int) true; // vietato - ... cannot be converted to ...
boolean falso = (boolean) 0; // vietato - ... cannot be converted to ...
```

49) METODO

- Termine caratteristico dei linguaggi OOP
- Un insieme di istruzioni con un nome
- Uno strumento per risolvere gradualmente i problemi scomponendoli in sottoproblemi
- Uno strumento per strutturare il codice
- Uno strumento per ri-utilizzare il lavoro già svolto
- Uno strumento per rendere il programma più chiaro e leggibile

1. Quando il programma da realizzare è articolato diventa conveniente identificare **sottoproblemi** che possono essere risolti individualmente
2. scrivere **sottoprogrammi** che risolvono i sottoproblemi richiamare i **sottoprogrammi** dal programma principale (main)
3. Questo approccio prende il nome di **programmazione procedurale** (o astrazione funzionale)

4. In Java i **sottoprogrammi** si realizzano tramite metodi ausiliari
5. Sinonimi usati in altri linguaggi di programmazione: funzioni, procedure e (sub)routines

---

Metodi ausiliari (static)

- metodi statici: dichiarati `static`
- richiamabili attraverso nome della classe
- p.es: `Math.sqrt()`

```
public class ProvaMetodi
{
        public static void main(String[] args) {
                stampaUno();
                stampaUno();
                stampaDue();
        }

        public static void stampaUno() {
                System.out.stampaln("Hello World");
        }

        public static void stampaDue() {
                stampaUno();
                stampaUno();
        }
}
```

Metodi non static

- I metodi non static rappresentano operazioni effettuabili su singoli oggetti
- La documentazione indica per ogni metodo il tipo ritornato e la lista degli argomenti formali che rappresentano i dati che il metodo deve ricevere in ingresso da chi lo invoca
- Per ogni argomento formale sono specificati:
  - un tipo (primitivo o reference)
  - un nome (identificatore che segue le regole di naming)

Invocazione di metodi non static

- L'invocazione di un metodo non static su un oggetto istanza della classe in cui il metodo è definito si effettua con la sintassi:

- Ogni volta che si invoca un metodo si deve specificare una lista di argomenti attuali
- Gli argomenti attuali e formali sono in corrispondenza posizionale
- Gli argomenti attuali possono essere delle variabili o delle espressioni
- Gli argomenti attuali devono rispettare il tipo attribuito agli argomenti formali
- La documentazione di ogni classe (istanziabile o no) contiene l'elenco dei metodi disponibili
- La classe **Math** non è istanziabile
- La classe **String** è "istanziabile ibrida"
- La classe **StringBuilder** è "istanziabile pura"

Esempi sui metodi

50) LA DOPPIA NATURA DELLE CLASSI

- Le classi disponibili nella libreria standard si possono distinguere in due tipologie principali:
  - Classi istanziabili
  - Classi non istanziabili
- La stessa distinzione è applicabile alle nostre classi
- La distinzione tra classi istanziabili e non istanziabili riguarda il senso logico del loro utilizzo
- Il termine "classe non istanziabile " sarà utilizzato per indicare una classe che non ha senso istanziare, date le sue caratteristiche
- Tecnicamente sarebbe possibile usare l'operatore new su classi "non istanziabili " (composte di metodi e attributi tutti static) ma non avrebbe senso pratico
- Alcune classi (p.es. quelle astratte) non permettono l'uso dell'operatore new
- La stragrande maggioranza delle classi è istanziabile ma l'esistenza di alcune classi non istanziabili è necessaria
- La classe (indispensabile) che contiene il main è normalmente non istanziabile
- Poiché i numeri non sono oggetti, i metodi numerici appartengono a classi non istanziabili

---

Classi istanziabili

- Una classe istanziabile fornisce il prototipo di una famiglia di oggetti (istanze della classe) che hanno struttura simile ma proprietà distinte a livello individuale (valori diversi degli attributi e quindi risultati diversi prodotti dai metodi)
- L'uso tipico è la costruzione di istanze (tramite new) e quindi l'invocazione di metodi su di esse
- Nel caso di una classe istanziabile attributi e metodi rappresentano proprietà possedute da tutti gli oggetti istanza della classe
- Ogni oggetto istanza di una classe ha una sua identità "contiene" individualmente gli attributi e i metodi definiti nella classe
- Ogni volta che si costruisce un'istanza con new si crea un nuovo insieme di attributi e metodi individuali

- Nel caso di una classe non istanziabile attributi e metodi sono "unici" a livello della classe (non esistono istanze diversificate)
- Una classe istanziabile rappresenta "qualcosa" che esiste in molteplici versioni individuali che hanno una struttura comune ma ciascuna con una propria identità:
- esistono molte sequenze di caratteri (la classe String è istanziabile)
- esistono molte valute (la classe Valuta è istanziabile)
- esistono molte persone (un'ipotetica classe Persona è istanziabile)

---

una classe istanziabile

- Normalmente ha costruttori
- Attributi e metodi sono tutti (o quasi) **non static**
- Quando penso all'esecuzione dei suoi metodi ho bisogno di immaginare un'istanza individuale a cui applicarli (anche senza argomenti esterni, perché usano attributi interni)
- Nel caso di classi istanziabili attributi e metodi sono definiti a livello di istanza
- Nel caso di classi non istanziabili attributi e metodi sono definiti a livello di classe

---

Classi non istanziabili

- Una classe non istanziabile contiene un insieme di metodi (ed eventualmente attributi) di natura generale non legati alle proprietà di oggetti individuali specifici
- Non ha senso la nozione di istanza della classe poiché non ci sono caratteristiche differenziabili tra oggetti distinti
- Una classe non istanziabile rappresenta "qualcosa" di concettualmente unico, che non esiste e non può esistere in versioni separate ciascuna con una propria identità:
- esiste una sola matematica (la classe Math non è istanziabile)
- esiste un solo sistema su cui un programma è eseguito (la classe System non è istanziabile)
- esiste un solo punto di inizio di un programma (le classi contenenti il main non sono istanziabili)

---

una classe non istanziabile

- Non ha costruttori
- Attributi e metodi sono tutti static
- Quando penso all'esecuzione dei suoi metodi non ho bisogno di immaginare un'istanza individuale: sono applicabili direttamente alla classe con almeno un argomento

```
Math . sqrt (2)
Math . abs ( - 3)
```

```
// In memoria ...
Math.E //2.7182
MATH.PI //3.1415
```

Classi istanziabili "ibride"

- Alcune classi istanziabili (p.e. String) della libreria standard contengono attributi o metodi static ed hanno quindi natura ibrida
- E' come se la classe avesse due sottoparti (una static e una no) ognuna delle quali segue le proprie regole
- Salvo rari casi, è sconsigliabile realizzare classi istanziabili ibride (sono accettabili attributi costanti definiti come static)

51) Instanziare una Classe: gli oggetti

Gli oggetti sono caratterizzati da

- Classe di appartenenza - tipo (ne descrive attributi e metodi)
- Stato (valore attuale degli attributi)
- Identificatore univoco (reference - handle - puntatore)

52) Per creare un oggetto occorre

- Dichiarare una istanza
- La dichiarazione non alloca spazio ma solo una riferimento (puntatore) che per default vale null
- Allocazione e inizializzazione
- Riservano lo spazio necessario creando effettivamente l'oggetto appartenente a quella classe

---

Notazioni Puntate

53) Le notazioni puntate possono essere combinate

- System.out.println("Hello world!");
- System è una classe del package java.lang
- out è una variabile di classe contenente il riferimento ad un oggetto della classe PrintStream che punta allo standard output
- println è un metodo della classe PrintStream che stampa una linea di testo

---

Operazioni su reference

54) Definiti gli operatori relazionali == e !=

- Attenzione: il test di uguaglianza viene fatto sul puntatore (reference) e NON sull'oggetto
- Stabiliscono se i reference si riferiscono allo stesso oggetto

È definita l'assegnazione

È definito l'operatore punto (notazione puntata)

NON è prevista l'aritmetica dei puntatori

Variabili di classe

- Rappresentano proprietà comuni a tutte le istanze
- Esistono anche in assenza di istanze (oggetti)
- Dichiarazione: static
- Accesso: NomeClasse.attributo

```
class Automobile {
static int numeroRuote = 4;
}
Automobile.numeroRuote;
```

Metodi di classe

55) Funzioni non associate ad alcuna istanza

- Dichiarazione: static
- Accesso: nome-classe . metodo()

```
class HelloWorld {
public static void main (String args[]) {
System.out.println("Hello World!");

//p.es  cos(x): metodo static della classe Math, ritorna un double
double y = Math.cos(x);
}
}
```

Operazioni su istanze

- Le principali operazioni che si possono effettuare sulle variabili che riferiscono istanze di una classe sono:
  - assegnamento
  - confronto
  - invocazione di metodi
- Il valore di una variabile di tipo strutturato è il riferimento ad un oggetto (istanza di una classe)
- Una stessa variabile può riferire oggetti diversi in momenti diversi a seguito di operazioni di assegnazione sul suo valore
- Se la variabile contiene il valore null non riferisce nessun oggetto in quel momento

---

Oggetti e riferimenti

- Le variabili hanno un nome, gli oggetti no
- Per utilizzare un oggetto bisogna passare attraverso una variabile che ne contiene il riferimento
- Uno stesso oggetto può essere riferito da più variabili e quindi essere raggiunto tramite nomi diversi (di variabili)
- Il rapporto variabili - oggetti riferiti è dinamico, il riferimento iniziale non necessariamente rimane legato all'oggetto per tutta la sua esistenza
- Se un oggetto non è (più) riferito da nessuna variabile diventa irraggiungibile (e quindi interviene il garbage collector)

Confronti tra variabili di tipo strutturato

- E' possibile applicare gli operatori di confronto == e != a variabili di tipo strutturato
- Se uno dei due termini del confronto è il valore null si verifica se una certa variabile riferisce un oggetto oppure no, p.e. saluto3 != null
- Se entrambi i termini del confronto sono variabili, si verifica se hanno lo stesso valore (cioè riferiscono esattamente lo stesso oggetto)

Confronto tra riferimenti vs. confronto tra oggetti

- Usare == fa il confronto tra i riferimenti non fra i valori contenuti negli oggetti (p.e. le sequenze di caratteri contenute nelle istanze di String)
- Di solito si vogliono confrontare i contenuti non i riferimenti: per questo si usa il metodo **equals**

- Il metodo booleano equals della classe String accetta come argomento il riferimento ad un altro oggetto e ritorna true se le stringhe contenute sono uguali (in modo case sensitive), false altrimenti
- Il metodo booleano equalsIgnoreCase fa lo stesso senza distinguere maiuscole/minuscole

56) Il Metodo Costruttore

Specifica le operazioni di inizializzazione (attributi, etc.) che vogliamo vengano eseguite su ogni oggetto della classe appena viene creato

Tale metodo ha

- Lo **stesso nome** della classe
- Tipo **non** specificato

Non possono esistere attributi non inizializzati

- Gli attributi vengono inizializzati comunque con valori di **default**

Se non viene dichiarato un costruttore, ne viene creato uno di default vuoto e senza parametri

Spesso si usa l'**overloading** definendo diversi costruttori

La distruzione di oggetti (garbage-collection) non è a carico del programmatore

---

57) Il costrutto new

- Crea una nuova istanza della classe specificata, allocandone la memoria
- Restituisce il riferimento all'oggetto creato
- Chiama il costruttore del nuovo oggetto

```
Automobile a = new Automobile ();
Motorcycle m = new Motorcycle ();
String s = new String ("ABC");
```

58) Per "gestire" una classe occorre

- Accedere ai metodi della classe
- Accedere agli attributi della classe

Messaggi

- L'invio di un messaggio provoca l'esecuzione del metodo

59) Inviare un messaggio ad un oggetto

- Usare la notazione "puntata" oggetto.messaggio(parametri)
- Sintassi analoga alla chiamata di funzioni in altri linguaggi
- I metodi definiscono l'implementazione delle operazioni
- I messaggi che un oggetto può accettare coincidono con i nomi dei metodi
- p.es mettiInMoto(), vernicia(), etc.
- Spesso i messaggi includono uno o più parametri
- .vernicia("Rosso")

60) Esempi

```
Automobile a = new Automobile();
a.mettiInMoto();
a.vernicia("Blu");
```

---

61) All'interno della classe

- I metodi che devono inviare messaggi allo stesso oggetto cui appartengono non devono utilizzare la notazione puntata, in quanto è sottinteso il riferimento

```
public class Libro {
    int nPagine;
    public void leggiPagina (int nPagina) {...}
    public void leggiTutto () {
        for (int i=0; i<nPagine; i++)
            leggiPagina (i);
    }
}
```

Attributi

- Stessa notazione "puntata" dei messaggi oggetto.attributo
- Il riferimento viene usato come una qualunque variabile

```
Automobile a=new Automobile();
a.colore = "Blu";
boolean x = a.accesa;
```

I metodi che fanno riferimento ad attributi dello stesso oggetto possono tralasciare il rif-oggetto

```
public class Automobile {
    String colore;
```

```
    void vernicia(){
        colore = "Verde";// colore si riferisce all'oggetto corrente
    }
}
```

- Esempio (messaggi e attributi)

```
public class Automobile {
    String colore;
    public void vernicia () {
        colore = "bianco";
    }
    public void vernicia (String nuovoCol) {
        colore = nuovoCol;
    }
}

Automobile a1, a2;
a1 = new Automobile ();
a1.vernicia ("verde");
a2 = new Automobile ();
```

---

62) Esempio (costruttori con overloading)

```
Class Finestra {
    String titolo;
    String colore;
    // Finestra senza titolo nè colore
    Finestra () {
    ...
    }
    // Finestra con titolo senza colore
    Finestra (String t) {
    ...
    titolo = t;
    }
    // Finestra con titolo e colore
    Finestra (String t, String c) {
    ...
    titolo = t; colore = c;
    }
}
```

Operatore this (Puntatore Auto-referenziante)

La parola riservata this e' utilizzata quale puntatore auto-referenziante

- this riferisce l'oggetto (e.g., classe) corrente

Utilizzato per:

- Referenziare la classe appena istanziata

```
class Automobile{
String colore;
...
...
void vernicia (String colore) {
this.colore = colore;
}
}
. . .
Automobile a2, a1 = new Automobile;
a1.vernicia("bianco"); // a1 == this
a2.vernicia("rosso");
// this == a2
```

- Evitare il conflitto tra nomi

```
class Automobile{
String colore;
...
...
void vernicia (String colore) {
this.colore = colore;
}
}
. . .
Automobile a2, a1 = new Automobile;
a1.vernicia("bianco"); // a1 == this
a2.vernicia("rosso");
// this == a2
```

63) METODI GETTER E SETTER

- Danno accesso agli attributi:

```
class NomeClasse{  private double nome1, nome2;

       public double getNome1(){
       return nome1;<
       }
```

```
public void setNome1(double nome1){
this.nome1 = nome1;
}
```

- Vantaggi: possiamo cambiare la rappresentazione interna, verificare che i valori siano corretti, modificare altri aspetti dell'oggetto

---

Visibilità ed encapsulation

Motivazione

- Modularità = diminuire le interazioni
- Information Hiding = delegare responsabilità

Supporto sintattico

- private attributo/metodo visibile solo da istanze della stessa classe
- public attributo/metodo visibile ovunque

Getters e setters

Aggiungere metodi per leggere scrivere un attributo privato

Può infrangere (indirettamente) l'encapsulation

```
String getColore() {
	return colore;
}
void setColore(String nuovoColore) {
	this.colore = nuovoColore;
}
```

Proprietà visibili

```
class Automobile {
	public String colore;
}
//creazione oggetto
Automobile a = new Automobile();
a.colore = "bianco"; // ok
```

Classe incapsulata

```
class Automobile {
	private String colore;
```

```
        public void vernicia(String colore) {
                this.colore = colore;
        }
}

//creazione oggetto
Automobile a = new Automobile();
a.colore = "bianco"; // error
a.vernicia("verde"); // ok
```

64) EREDITARIETÀ IN JAVA

Riutilizzare il codice

- Uno dei grandi vantaggi della programmazione a oggetti è la facilità nel riutilizzare il codice
- In Java si realizza attraverso l'ereditarietà
- Per esempio immaginiamo di disporre di un sistema di georeferenziazione in cui la classe principale è

`Geopunto {attributi: longitudine, latitudine}`

- Dobbiamo modificare il sistema per aggiungere un nuovo attributo, l'altitudine

a) Modifichiamo direttamente la classe Geopunto Errore

b) Creiamo una nuova classe che erediti da Geopunto:

`NuovoGeopunto {Geopunto + attributo: altitudine}`

---

65) Estendere la classe

66) Quando utilizzare l'ereditarietà

- Per modificare classi già esistenti
- In un ambiente di sviluppo reale disponiamo di centinaia di classi. Per esempio la classe Button che rappresenta un bottone
- Immagina di dover creare un nuovo tipo di bottone che riproduca un suono quando viene schiacciato
- Con l'ereditarietà risulta molto semplice
- Per non dover riprogrammare due volte lo stesso codice
- Devi creare varie classi, p.es Cliente, Venditore, Distributore
- Esiste una parte del codice comune a tutte quante, p.es Piva, indirizzo, SpedireLettera()

- Crei una classe addizionale che raccolga il codice comune, es ParteCommerciale da cui ereditino le parti comuni

67) POLIMORFISMO (OVERLOADING)

Una classe può avere più metodi con lo stesso nome

68) I metodi devono essere distinguibili in base a

- Numero dei parametri
- Tipo dei parametri

69) Il metodo da eseguire viene scelto in base a

- Numero e tipo di parametri

Il metodo da eseguire **NON** viene scelto in base al valore di ritorno

---

70) Esempio

```
class Automobile {
    String colore;
    void vernicia () {
    colore = "bianco";
    }
    void vernicia (int i) {
    switch (i) {
    case 1: colore = "nero"; break;
    ...
    }
    }
    void vernicia (String nuovoCol) {
    colore = nuovoCol;
    }
}
```

71) IL MODIFICATORE STATIC

- La parola riservata static viene usata per indicare il livello di definizione di un attributo o metodo
- Se la parola static è presente l'attributo o metodo è definito a livello di classe
- Se la parola static non è presente l'attributo o metodo è definito a livello di istanza

Static vs. non static

- Ogni attributo o metodo non static esiste concretamente in ogni istanza creata

- Esiste in “molteplici versioni” se vengono create più istanze
- Non esiste concretamente se il programma non crea almeno un’istanza (tramite operatore new)
- Ogni attributo o metodo static esiste concretamente a livello di classe
- Esiste in un’unica versione
- La sua esistenza non dipende da cosa fa il programma

Accesso a metodi e attributi static

- Essendo definiti a livello di classe, attributi e metodi static sono acceduti/invocati tramite il nome della classe:

```
NomeClasse.nomeMetodoStatic (...)
NomeClasse.nomeAttributoStatic
//P.e.
Math.sqrt (2);
String.valueOf ( Math .PI);
```

- In caso di classe istanziabile ibrida è equivalente, ma sconsigliabile, accedere tramite una qualunque istanza della classe:

```
nomeIstanza.nomeMetodoStatic (...)
nomeIstanza.nomeAttributoStatic
```

Accesso a metodi e attributi non static

- La sintassi è simile al caso precedente, ma ovviamente l’accesso/invocazione è possibile solo tramite un’istanza specifica (ed ogni accesso è diversificato):

```
nomeIstanza.nomeMetodoNonStatic (...)
nomeIstanza.nomeAttributoNonStatic
```

- Accesso ad attributi e metodi della propria classe
- All'interno del corpo di un metodo si possono riferire in modo abbreviato attributi e metodi definiti nella stessa classe
- Se nel corpo di un metodo appare il nome di un metodo o attributo static della sua classe è sottinteso che sia preceduto dal nome della classe stessa

---

Accesso ad attributi e metodi della propria classe

- Se nel corpo di un metodo (non static) appare il nome di un metodo o attributo non static della sua classe è sottinteso che sia riferito all'istanza su cui è stato invocato il metodo

Attenzione

- Nel corpo di un metodo static non si può accedere ad attributi e metodi non static della stessa classe
- Il metodo static deve poter essere invocato a livello di classe (anche in assenza di istanze) mentre attributi e metodi non static esistono solo se c'è almeno un'istanza
- Nel corpo di un metodo non static si può accedere a qualunque attributo e metodo della stessa classe (se esiste l'istanza ovviamente esiste la classe)
- Eventuali mescolanze improprie di static e non static causano errori di compilazione

72) Classi Astratte

73) Una classe astratta è una classe avente almeno un metodo astratto

```
abstract class Forma {
...
abstract void stampa();
...
}
```

Un metodo astratto è un metodo di cui non viene specificata l'implementazione

---

74) Una classe astratta

- È una classe non completamente definita
- Non può essere istanziata

Per ottenere una classe concreta (istanziabile) da una astratta occorre definire tutte le implementazioni mancanti ovvero

- Ereditare la classe facendo l'overriding di tutti i metodi astratti

75) Permette di trattare omogeneamente oggetti con caratteristiche diverse (tramite upcasting)

Man mano che si sale nella gerarchia dell'ereditarietà, le classi diventano sempre più generiche e probabilmente più astratte.

Ad un certo punto la classe superiore diventa a tal punto generica che la si può pensare come una base per le altre classi piuttosto che come una classe di cui creare un oggetto.

---

76) METODO ASTRATTO

Voglio obbligare tutte le sottoclassi di una classe A ad avere un metodo `nomeMetodo()` ma allo stesso tempo non voglio implementare tale metodo nella classe A.

SINTASSI:

```
public abstract int nomeMetodo( ) ;
```

- un metodo astratto deve essere obbligatoriamente pubblico, altrimenti non avrebbe senso
- lascio il metodo indefinito. infatti non apro il relativo blocco con le parentesi graffe ed il codice del metodo.

Le Classi Astratte, oltre ad avere Metodi Astratti, possono avere metodi ed attributi *normali*.

Se una classe è astratta, non posso istanziarla.

NON POSSO CREARE OGGETTI DI UNA CLASSE ASTRATTA

Che senso ha definire dei metodi astratti e di conseguenza rendere una classe astratta?

Definisco un metodo astratto quando voglio forzare tutte le sottoclassi ad avere un determinato metodo.

I metodi astratti funzionano come segnaposto dei metodi implementati poi nella sottoclasse.

le sottoclassi

La sottoclasse deve implementare tutti i metodi che la superclasse aveva astratti.

Se la sottoclasse a sua volta non definisce i metodi che la superclasse aveva dichiarato astratti, anche la sottoclasse deve essere dichiarata astratta.

77) INTERFACCE

78) Un'interfaccia è una classe

completamente astratta, cioè del tutto priva della parte di implementazione

- Tutti i metodi sono astratti
- Non vi sono attributi
    - È possibile definire solo "attributi" final (in pratica costanti)

- Definendo un attributo in un'interfaccia questo viene automaticamente considerato final

---

79) Un'interfaccia

- Ha tutti i vantaggi e le indicazioni d'uso delle classi astratte
- Presenta maggior flessibilità rispetto all'ereditarietà di una classe astratta

---

Si può pensare a un'interfaccia come a una classe astratta che ha tutti e soli metodi astratti (ci sono però differenze).

Un'interfaccia può essere considerata un modo per cosa dovrebbero fare le classi senza specificare come farlo.

Quindi un'interfaccia non è una classe ma un insieme di requisiti per le classi che si vogliono conformare ad essa.

Sintassi:

```
public interface NomeInterfaccia
{
  int metodo1( ... );
}
```

---

interface

Si utilizza la parola chiave interface anzichè class

I metodi sono implicitamente pubblici e astratti, non bisogna indicarlo

Se una classe decide di soddisfare i requisiti di un'interfaccia si dice che la classe implementa l'interfaccia.

---

esempio d'uso

Per indicare che una classe implementa un'interfaccia si utilizza la seguente sintassi:

```
public class NomeClasse implements NomeInterfaccia
{
  //codice relativo alla classe
}
```

Tale classe deve implementare tutti i metodi elencati nell'interfaccia.

---

80) proprietà

Le interfacce non sono classi; non si può utilizzare new per crearne oggetti.

I metodi di un'interfaccia sono automaticamente public (quindi non è necessario scriverlo)

Gli attributi di un'interfaccia sono sempre public static final (non è necessario scriverlo).

Un'interfaccia con il nome NomeInterfaccia va salvata nel file NomeInterfaccia.java (come accade per le classi).

---

81) ereditarietà multipla

Una sottoclasse può estendere solo 1 superclasse (non permettendo l'ereditarietà multipla)

Con le interfacce invece la situazione è diversa: una classe può implementare quante interfacce vuole.

---

82) variabili

Posso dichiarare variabili del tipo dell'interfaccia e, sfruttando il polimorfismo, assegnargli oggetti di classi che implementano tali interfacce.

---

esempio d'uso

```
public interface NomeInterfaccia {
int metodo1( );
}
public class NomeClasse implements NomeInterfaccia{
int metodo1( ) {
//codice del metodo
}
}
NomeInterfaccia a = new NomeClasse ( );
```

Tipi di interfacce

- Normali
- Single Abstract method - @FunctionalInterface

- Marker

Con java 1.8 le interfacce sono state modificate: è possibile implementare due tipi di metodi (!!!)

- default
- static

Lambda expressions

Con le interfacce contenenti un singolo metodo astratto, è possibile utilizzare le espressioni lambda

```
Integer raddoppiato = (o) -> o * 2 ;
```

83) ECCEZIONI

Situazioni anomale a run-time

- Java prevede un sofisticato utilizzo dei tipi (primitivi e classi) che consente di individuare molti errori al momento della compilazione del programma (prima dell'esecuzione vera e propria)
- Ciò nonostante si possono verificare varie situazioni impreviste o anomale durante l'esecuzione del programma che possono causare l'interruzione del programma stesso
- Ad esempio:
  - Tentativi di accedere a posizioni di un array che sono fuori dai limiti
  - Errori aritmetici (divisione per zero, ...)
  - Errori di formato: (errore di input dell'utente)

---

84) Le eccezioni si dividono in

Checked (o controllate) per le quali il compilatore richiede che ci sia un gestore

- **Controllate**
  - Istanze di RuntimeException o delle sue sottoclassi
  - Il compilatore si assicura esplicitamente che quando un metodo solleva una'eccezione la tratti esplicitamente
  - Questo può essere fatto mediante i costrutti try-catch o throws
  - In caso contrario segnala un errore di compilazione
  - Le eccezioni controllate vincolano il programmatore ad occuparsi della loro gestione
  - Le eccezioni controllate possono rendere troppo pesante la scrittura del codice

Unchecked (o non controllate) per le quali il gestore non è obbligatorio

- Per essere unchecked un'eccezione *deve essere una sottoclasse di* **RuntimeException**, altrimenti è checked
- **Non controllate**
  - Sono tutte le altre eccezioni, ovvero istanze di Exception ma non di RuntimeException
  - L'eccezione può non essere gestita esplicitamente dal codice
  - Viene "passata" automaticamente da metodo chiamato a metodo chiamante

---

85) Esempi tipici di eccezioni checked:

- le eccezioni che descrivono errori di input/output
  - lettura o scrittura su file,
  - comunicazione via rete, ecc...
- le eccezioni definite dal programmatore

La gerarchia delle eccezioni

La classe **Exception** descrive un'eccezione generica, situazioni anomale più specifiche sono descritte dalle sottoclassi di Exception

Le RuntimeException comprese nel pacchetto java.lang

Eccezione	Significato
ArithmeticException	Operazione matematica non valida.
ArrayIndexOutOfBoundsException	L'indice usato in un array non è valido.
ArrayStoreException	Incompatibilità di tipo durante la assegnazione di un elemento di un array.
ClassCastException	Conversione di tipo non valida.
IllegalArgumentException	Argomento di un metodo non valido.
IllegalMonitorStateException	Monitor su thread non valido.
IllegalStateException	Oggetto in uno stato che non consente l'operazione richiesta.
IllegalThreadStateException	Operazione incompatibile con lo stato attuale di un thread.
IndexOutOfBoundsException	Indice non valido.

Eccezione	Significato
NegativeArraySizeException	Array creato con dimensione negativa.
NullPointerException	Utilizzo non corretto di un valore null.
NumberFormatException	Conversione non valida di una stringa in un valore numerico.
SecurityException	Violazione delle norme di sicurezza.
StringIndexOutOfBoundsException	Indice non valido per i caratteri di una stringa.
UnsupportedOperationException	Operazione non supportata.

Gestione delle eccezioni

- In Java, le situazioni anomale che si possono verificare a run-time possono essere controllate tramite meccanismi di gestione delle eccezioni
- Esistono classi che descrivono le possibili anomalie
- Ogni volta che la Java Virtual Machine si trova in una situazione anomala;
    1. sospende il programma
    2. crea un oggetto della classe corrispondente all'anomalia che si è verificata
    3. passa il controllo a un gestore di eccezioni (implementato dal programmatore)
    4. se il programmatore non ha previsto nessun gestore, interrompe il programma e stampa il messaggio di errore

---

• Il costrutto try-catch

- Il costrutto try-catch consente di
    - **monitorare** una porzione di programma (all'interno di un metodo)
    - **specificare** cosa fare in caso si verifichi una anomalia (eccezione)

Si usa cosi:

```
// ... blocchi di codice NON monitorati ....

try {
  // ... blocchi di codice monitorati ....
  }
  catch (Exception e) {
  // ... blocchi di codice da eseguire IN CASO DI ECCEZIONE
```

```
  }

// ... altri blocchi di codice NON monitorati ....
```

Gestire le eccezioni

- Un costrutto try-catch può gestire più tipi di eccezione contemporaneamente
- I vari gestori (ognuno denotato da un catch) vengono controllati in sequenza
- Viene eseguito (solo) il primo catch che prevede un tipo di eccezione che è supertipo dell'eccezione che si è verificata
- Quindi, è meglio non mettere Exception per prima (verrebbe richiamata in tutti i casi)
- La variabile e è un oggetto che può contenere informazioni utili sull'errore che si è verificato...

```
try {
  //istruzioni da controllare
}
catch (NumberFormatException e) {
  //codice
}
catch (Exception e) {
  //codice
}
```

---

quando definire un gestore di eccezioni

- Per capire quando preoccuparsi di definire un gestore di eccezioni:
  - bisogna avere un'idea di quali sono le eccezioni più comuni e in quali casi si verificano (esperienza)
  - bisogna leggere la documentazione dei metodi di libreria che si utilizzano:
    - la documentazione della classe Scanner spiega che il metodo nextInt() può lanciare l'eccezione InputMismatchException
  - in alcuni casi le eccezioni non vanno gestite: segnalano un errore di programmazione che deve essere corretto!
    - verifica la correttezza dei cicli nel caso di scorrimento di una collezione

---

Il comando throw

- Il meccanismo delle eccezioni può anche essere usato per segnalare situazioni di errore

- Il comando throw consente di lanciare un'eccezione quando si vuole
- Si può usare la classe Exception, una sua sottoclasse già definita, o una sua sottoclasse custom
- **throw** si aspetta di essere seguito da un oggetto, che solitamente è costruito al momento (tramite new)
- Il costruttore di una eccezione prende come parametro (opzionale) una stringa di descrizione

```
throw new Exception("Operazione non consentita");
     throw new AritmeticException ();
     throw new EccezionePersonalizzata ();
```

- Il comando throw si può usare direttamente dentro un try-catch,
- l'uso più comune di throw è all'interno dei metodi
- L'utilizzo di throw dentro a un metodo consente di interrompere il metodo in caso di situazioni anomale:
  - parametri ricevuti errati
  - operazione prevista dal metodo non realizzabile
  - (esempio: prelievo dal conto corrente di una somma superiore al saldo)
- Chi invoca il metodo dovrà preoccuparsi di implementare un gestore delle eccezioni possibilmente sollevate
- Questo consente di evitare valori di ritorno dei metodi che servono solo a dire se l'operazione è andata a buon fine
- in caso di problemi si lancia l'eccezione, non si restituisce un valore particolare

---

parola chiave throws

- Un metodo che contiene dei comandi throw deve elencare le eccezioni che possono essere sollevate
- L'elenco deve essere fatto nell'intestazione, usando la parola chiave **throws**
- throws si usa nell'intestazione del metodo
- throw si usa all'interno (nel punto in cui si verifica l'errore) public void preleva(int somma)

```
throws IOException , IllegalParameterException { ... }
```

---

86) Terminologia

- **Errore**: problema con la logica applicativa, errore del programmatore (non gestibile)
- **Eccezione**: evento anomalo recuperabile

---

87) Una istruzione non terminata

- Causa la creazione di un oggetto che rappresenta quanto è successo
- Tale oggetto appartiene a una classe derivata da Throwable
- Tali oggetti sono le eccezioni

---

88) Si dice che l'eccezione

- Viene "gettata" (thrown) e
- In seguito deve essere "gestita" ovvero "catturata" (catch)

---

89) Costrutti per la gestione delle eccezioni

- try { } ... catch {}
  - "Getta" l'eccezione a livello di un blocco di istruzioni
  - "Cattura" l'eccezione effettuandone la gestione
- throws
  - "Getta" l'eccezione a livello di metodi
- throw
  - "Getta" l'eccezione a livello di codice / istruzioni

---

try ... catch

- Cattura le eccezioni generate in una regione di codice

```
try {
  // codice in cui possono verificarsi le eccezioni
...
}
catch (IOException e) {
  // codice per gestire IOException e
...
}
```

---

90) Per catturare eccezioni di classi diverse si possono usare blocchi catch multipli

```
try {
...
}
catch(MalformedURLException mue) {
```

```
// qui recupero l'errore "malformedURL"
...
}
catch(IOException e) {
// qui recupero tutti altri errori di IO
...
}
```

---

91) Costrutti try-catch possono essere annidati

(catch che include try-catch)

---

92) Il blocco "finally" esegue istruzioni al termine del blocco try-catch

- sia che si verifichino le eccezioni
- sia che NON si verifichino le eccezioni
- Anche in presenza di istruzioni `return, break e continue`

```
try {
...
}
catch (...) {
...
}
catch (...) {
...
}
...
finally {
...
}
```

---

93) Permette a un metodo di gettare

eccezioni () throws <classeEccezione 1 > [, <classeEccezione 2 > ... [,]...] { ... }

---

94) Le eccezioni gettate sono catturate

(responsabilità) dal chiamante si chiama il metodo leggi deve sapere se la lettura è andata a buon fine oppure no

```
* Con try-catch gestiamo l'eccezione a livello
del chiamato (metodo leggi)
...
byte b[] = new byte[10];
try {
System.in.read (b);
} catch (Exception e) {
...
}
...
```6* Sapere se la lettura è andata a buon fine, non
"interessa" tanto al chiamato (metodo leggi)
quanto al chiamante
static String leggi (String val) throws
IOException {
byte b[] = new byte[10];
System.in.read (b); // Senza try ... Catch
val = "";
for (int i=0; i<b.length; i++) {
val = val + (char) b[i];
}
return (val);
}
```

---

95) throw

Permette di "gettare" in modo "esplicito" una eccezione a livello di codice `throw <oggettoEccezione>`

---

96) Provoca

- L'interruzione dell'esecuzione del metodo
- L'avvio della fase di cattura dell'eccezione generata
- Dato che le eccezioni sono oggetti, chi getta l'eccezione deve creare l'oggetto eccezione (operatore new) che la descrive

```
if ( y==0 ){
throw new ArithmeticException (
"Frazione con denominatore nullo.");
}
```
```

```
z = x/y;
```

Classi di Eccezioni

- È una classe, subclass di Throwable o discendenti, definita in java.lang
- Error: hard failure
- Exception: non sistemiche
- RuntimeException: il compilatore non forza il catch
- Error
- Gli errori sono trattabili ma in genere costituiscono situazioni non recuperabili
- OutOfMemoryError
- Exception

Definizione di una eccezione

- È possibile dichiarare eccezioni proprie, se quelle fornite dal sistema (java.lang) non sono sufficienti
- Si realizza creando sottoclassi di Throwable
- Tali sottoclassi sono del tutto "assimilabili" a classi "standard", e.g., possono
  - ereditare attributi e metodi
  - ridefinire il metodo costruttore
  - definire dei metodi get/set
  - etc.

- **Esempi**

---

97) EccezioneArray

```
public class EccezioneArray {
  public static void main(String[] args) {
  int[] a = {5,3,6,5,4};
  // attenzione al <=...
  for (int i=0; i<=a.length; i++)
  System.out.println(a[i]);
  System.out.println("Ciao");
  }
}
```

---

98) EccezioneAritmetico

```
import java.util.Scanner;
public class EccezioneAritmetico {
  public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  System.out.println("Inserisci due interi");
  int x = input.nextInt();
  int y = input.nextInt();
  System.out.println(x/y);
  // che succede se y == 0??
  }
}
```

---

99) EccezioneFormato

```
import java.util.Scanner;
public class EccezioneFormato {
  public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  System.out.println("Inserisci un intero");
  int x = input.nextInt();

  // che succede se l'utente inserisce un carattere?
  System.out.println(x);
  }
}
```

---

100) Esempio gestione eccezione: EccezioneAritmetico

```
import java.util.Scanner;
public class EccezioneAritmetico {
  public static void main(String[] args) {
  Scanner input = new Scanner(System.in);

  System.out.println("Inserisci due interi");
  int x = input.nextInt();

  int y = input.nextInt();

  try { System.out.println(x/y);
  System.out.println("CIAO");

  }
```

```
  catch (ArithmeticException e) {
  // se si verifica un eccezione di tipo ArithmeticException
  // nella divisione x/y il programma salta qui (non stampa CIAO)
  System.out.println("Non faccio la divisione..."); // gestita
l'anomalia, l'esecuzione riprende...
  }
  System.out.println("Fine Programma"); }
}
```

---

101) Esempio gestione eccezione: EccezioneFormato

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class EccezioneFormato {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("Inserisci un intero");
    int x;
    boolean ok;
    do {
      ok = true;
      try {
        x = input.nextInt();
        System.out.println(x);
      }
      catch (InputMismatchException e) {
        input.nextLine(); // annulla l'input ricevuto
System.out.println("Ritenta...");
        ok = false;
      }
    } while (!ok);
  }
}
```

---

102) Esempio: controllo correttezza parametri - Rettangolo

```
public class Rettangolo {
  private base;
  private altezza;
  // ... altri metodi e costruttori

  public void setBase(int x) throws EccezioneBaseNegativa {
    if (x<0) throw new EccezioneBaseNegativa()
```

```
    else base=x;
  }
}
```

---

103) EccezioneBaseNegativa

```
public class EccezioneBaseNegativa extends Exception {
  EccezioneBaseNegativa() {
    super ();
  }

  EccezioneBaseNegativa(String msg) {
    super(msg);
    }
}
```

---

104) System.in.read

- può provocare una eccezione controllata di tipo IOException
- Occorre quindi inserirla in un blocco

```
try...catch...
byte b[] = new byte[10];
try {
System.in.read (b);
} catch (Exception e) {
...
}
```