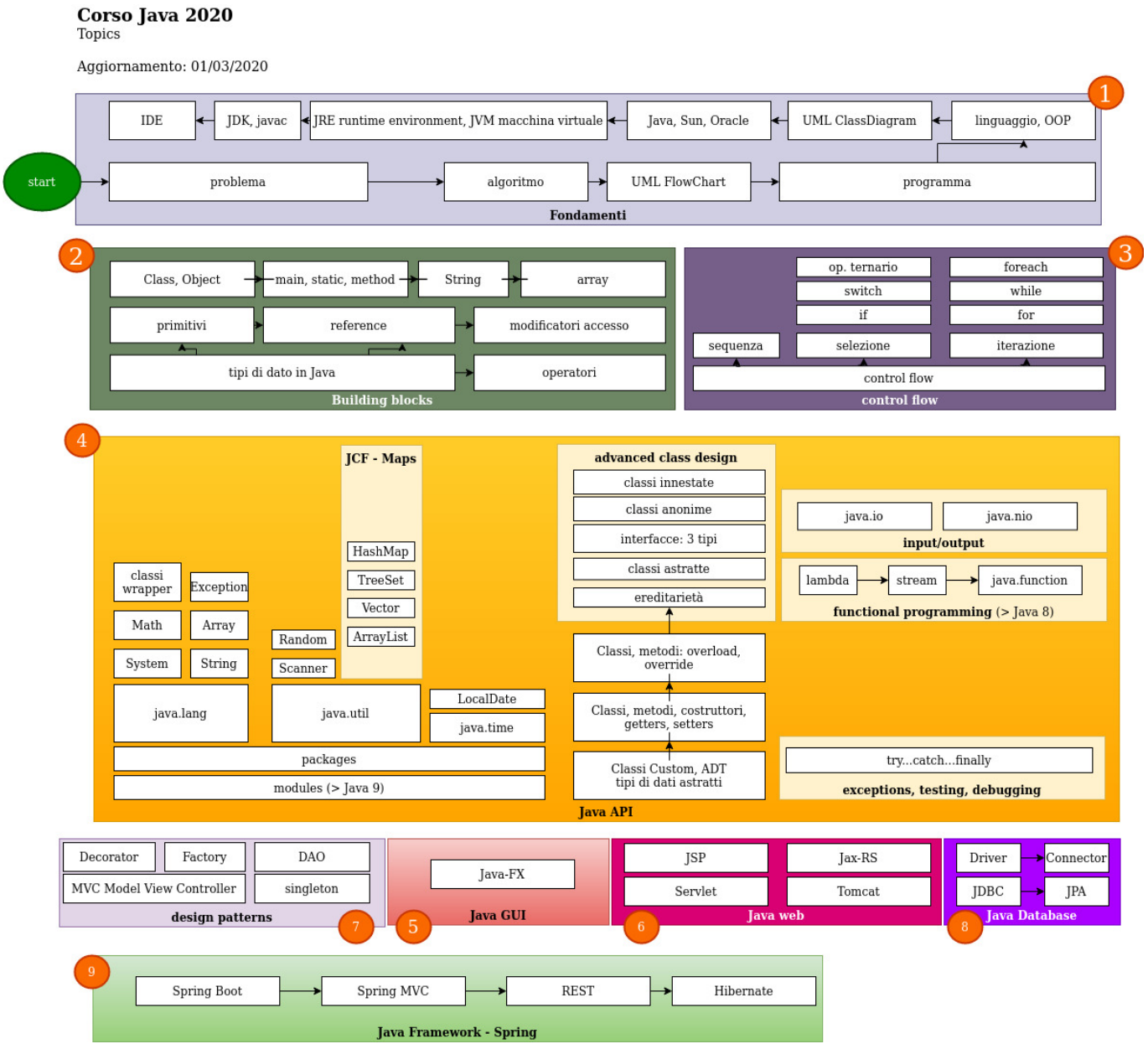
Corso Java

Appunti Java disponibili su maboglia.github.io

Mauro Bogliaccino - 2020

Corso OOP JAVA

Schema Argomenti



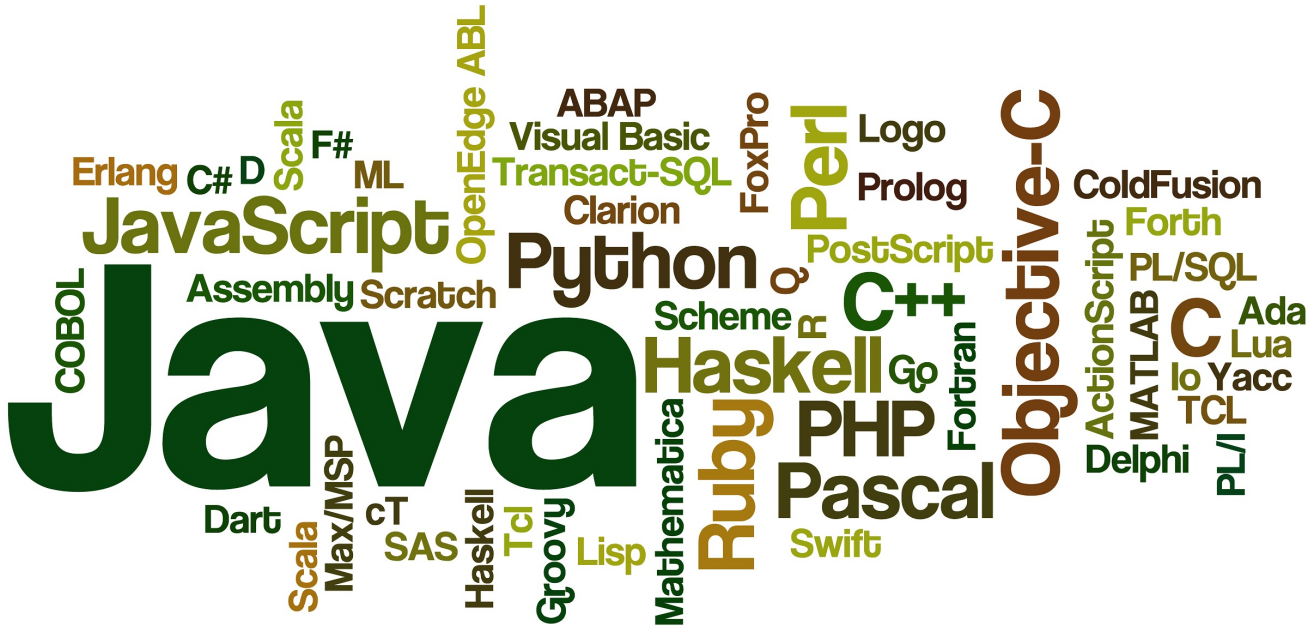
Bibliografia e risorse

- Java SE Documentation – Oracle/Sun (en)
- Java tutorials – Oracle/Sun (en)
- OCA Java Study Guide – Oracle (en)
- Programma OCA
- Programma OCP
- Programma GUI
- **Programmazione di base e avanzata con Java** – Valter Savitch (it)
- **Java 9 Guida allo sviluppo** in ambienti Windows, macOS e * GNU/Linux - Pellegrino Principe (it)
- **Manuale di Java** – Claudio de Sio – Hoepli Informatica (it)

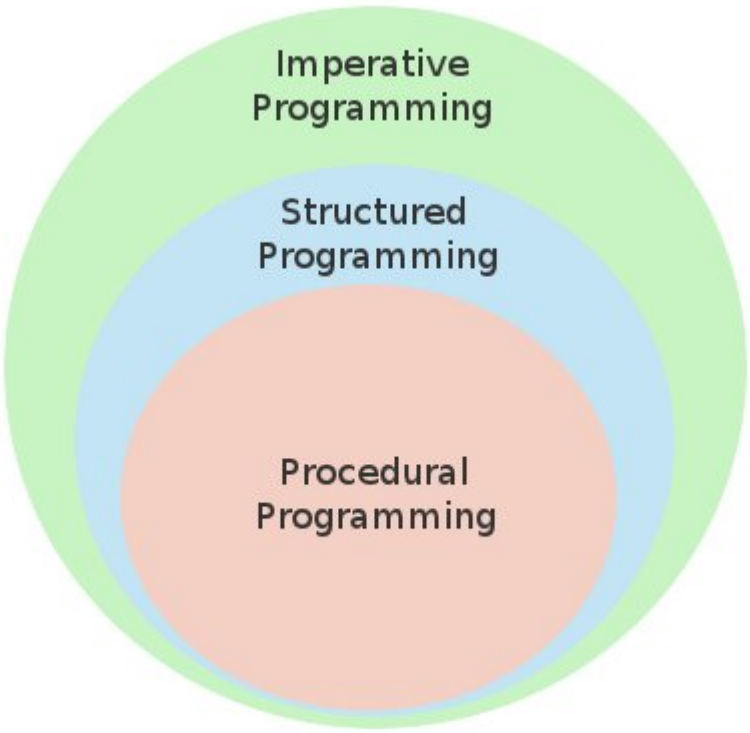
- **Concetti di informatica e fondamenti di Java** - Cay Horstmann (it)
- Guida Java html.it (it)

Paradigmi di programmazione e linguaggi

Quanti Linguaggi...



… e quanti paradigmi di programmazione



Un paradigma di programmazione è uno stile fondamentale di programmazione utile per portare ordine e criteri di lavoro più efficienti nella produzione dei programmi

Teorema di Jacopini-Bohm (1966)

Un qualsiasi algoritmo può essere espresso utilizzando esclusivamente le tre strutture di controllo:

- sequenza, selezione e iterazione.
- Teorema Jacopini Bohm

Paradigmi

- La programmazione strutturata
- La programmazione imperativa
- La programmazione procedurale
- La programmazione ad oggetti
- La programmazione funzionale
- La programmazione logica

La programmazione imperativa

- In informatica, la programmazione imperativa è un paradigma di programmazione secondo cui un programma viene inteso come un insieme di istruzioni (dette anche direttive o comandi), ciascuna delle quali può essere pensata come un "ordine" che viene impartito alla macchina virtuale del linguaggio di programmazione utilizzato.
- Da un punto di vista sintattico, i costrutti di un linguaggio imperativo sono spesso identificati da verbi all'imperativo, per esempio:
    - 1: read i
    - 2: print i
    - 3: goto 1

Questo modello di progettazione del software si basa sull'idea della sequenza ordinata di passi e sull'istruzione di assegnamento che serve per cambiare il valore delle variabili.

Il paradigma imperativo è molto vicino al modo di funzionamento dell'elaboratore: è molto efficiente per un uso generale e non è riservato a specifici problemi. (Es.Fortran, Cobol, Pascal, Basic, C)

È uno dei paradigmi di programmazione più antichi. Presenta una stretta relazione con l'architettura della macchina. Si basa sull'architettura Von Neumann. Funziona modificando lo stato del programma tramite istruzioni di assegnazione. Esegue attività passo-passo cambiando lo stato. L'attenzione principale è su come raggiungere l'obiettivo. Il paradigma consiste in diverse affermazioni e dopo l'esecuzione di tutto il risultato viene memorizzato.

Calcola la media di 5 numeri interi in linguaggio C

```
int voti[5] = { 28, 29, 30, 25, 123 } int somma = 0;
float media = 0.0;
for (int i = 0; i < 5; i++) {
    somma = somma + voti[i];
}
media = somma / 5;
```

La programmazione imperativa è suddivisa in tre grandi categorie:

- procedurale,
- OOP
- elaborazione parallela.

Argomenti correlati

- differenza tra OOP e procedurale
- Paradigmi e Linguaggi

La programmazione procedurale

In informatica la programmazione procedurale è un paradigma di programmazione che consiste nel creare dei blocchi di codice sorgente, identificati da un nome e racchiusi da dei delimitatori, che variano a seconda del linguaggio di programmazione; questi sono detti anche sottoprogrammi (in inglese subroutine) procedure o funzioni, a seconda del linguaggio e dei loro ruoli all'interno del linguaggio stesso.

Il nome deriva dal linguaggio COBOL, che è stato il primo ad utilizzare questo concetto.

I linguaggi imperativi (o procedurali) sono certamente i più diffusi, considerando anche quelli **Object Oriented**. Per esempio: C++, Java, Python, Objective C, e ovviamente i linguaggi come C, Fortran, Cobol, Pascal.

La programmazione è incentrata sui dati (variabili, strutture dati, oggetti) e sui modi per processarli (si parla di procedure, funzioni, metodi a seconda dei casi).

Si parte da un insieme di dati cui si applicano una o più **manipolazioni** da eseguire in un determinato ordine al fine di ottenere il risultato finale.

La programmazione procedurale può essere il primo paradigma di programmazione che un nuovo sviluppatore imparerà. Fondamentalmente, il codice procedurale è quello che istruisce direttamente un dispositivo su come completare un'attività in passaggi logici.

Questo paradigma utilizza un approccio lineare dall'alto verso il basso e tratta i dati e le procedure come due entità diverse. Sulla base del concetto di una chiamata di procedura, la Programmazione procedurale suddivide il programma in procedure, note anche come routine o funzioni, contenenti semplicemente una serie di passaggi da eseguire.

In poche parole, la Programmazione procedurale comporta la redazione di un elenco di istruzioni per dire al computer cosa dovrebbe fare passo dopo passo per completare l'attività a portata di mano.

Caratteristiche principali della programmazione procedurale

Le caratteristiche principali della programmazione procedurale sono riportate di seguito:

- Funzioni predefinite: una funzione predefinita è in genere un'istruzione identificata da un nome. Di solito, le funzioni predefinite sono integrate in linguaggi di programmazione di livello superiore, ma derivano dalla libreria o dal registro, piuttosto che dal programma. Un esempio di una funzione predefinita è `charAt()`, che cerca la posizione di un carattere in una stringa.
- Variabile locale: una variabile locale è una variabile dichiarata nella struttura principale di un metodo ed è limitata all'ambito locale che viene fornito. La variabile locale può essere utilizzata solo nel metodo in cui è definita e, se dovesse essere utilizzata al di fuori del metodo definito, il codice smetterà di funzionare.
- Variabile globale: una variabile globale è una variabile dichiarata al di fuori di ogni altra funzione definita nel codice. Per questo motivo, le variabili globali possono essere utilizzate in tutte le funzioni, diversamente da una variabile locale.
- Modularità: la modularità è quando due sistemi diversi hanno due compiti diversi a portata di mano ma sono raggruppati insieme per concludere prima un compito più ampio. Ogni gruppo di sistemi dovrebbe

quindi completare i propri compiti uno dopo l'altro fino al completamento di tutti i compiti.
- Passaggio parametri: il passaggio parametri è un meccanismo utilizzato per passare parametri a funzioni, subroutine o procedure. Il passaggio dei parametri può essere eseguito tramite "passa per valore", "passa per riferimento", "passa per risultato", "passa per risultato valore" e "passa per nome".

Vantaggi e svantaggi della programmazione procedurale

La programmazione procedurale include una serie di pro e contro, alcuni dei quali sono indicati di seguito.

vantaggi

- La programmazione procedurale è eccellente per la programmazione generale
- La semplicità insieme alla facilità di implementazione di compilatori e interpreti
- Una grande varietà di libri e materiale didattico online disponibile su algoritmi testati, che semplifica l'apprendimento lungo il percorso
- Il codice sorgente è portabile, pertanto può essere utilizzato anche come target per una CPU diversa
- Il codice può essere riutilizzato in diverse parti del programma, senza la necessità di copiarlo
- Attraverso la tecnica di programmazione procedurale, anche il requisito di memoria viene ridotto
- Il flusso del programma può essere monitorato facilmente

svantaggi

- Il codice del programma è più difficile da scrivere quando si utilizza la Programmazione procedurale
- Il codice procedurale spesso non è riutilizzabile, il che può comportare la necessità di ricreare il codice se necessario per l'uso in un'altra applicazione
- Difficile relazionarsi con oggetti del mondo reale
- L'importanza è data all'operazione piuttosto che ai dati, il che potrebbe comportare problemi in alcuni casi sensibili ai dati
- I dati sono esposti a tutto il programma, il che rende meno sicuro il programma

Esistono diversi tipi di paradigma di programmazione, che non sono altro che uno stile di programmazione. Il paradigma non si adatta a una lingua specifica, ma al modo in cui è scritto il programma.

Paradigmi e Linguaggi

I principi della OOP

- Definire nuovi tipi di dati.
- Incapsulare i valori e le operazioni.
- Riusare il codice esistente.
- Supportare il polimorfismo.

L’oggetto

- un oggetto è una istanza di una classe.
- un oggetto deve essere conforme alla descrizione di una classe.
- un oggetto è contraddistinto da:
    1. attributi;
    2. metodi;
    3. identità;
- un oggetto non deve mai manipolare direttamente i dati di un altro oggetto
- la classe è una entità statica cioè a tempo di compilazione;
- l’oggetto è una entità dinamica cioè a tempo di esecuzione (run time);

ADT (Abstract Data Types): creare nuovi tipi di oggetto

- i dati (o attributi)
    - contengono le informazioni di un oggetto;
- le operazioni (o metodi)
    - consentono di leggere/scrivere gli attributi di un oggetto;

La descrizione di una classe

- La classe rappresenta la descrizione di un oggetto, contiene le definizioni di proprietà e metodi dell'oggetto che rappresenta.
- La classe consente di implementare gli ADT attraverso il meccanismo di incapsulamento.
- i dati devono rimanere privati insieme all’implementazione
- solo l’interfaccia delle operazioni è resa pubblica all’esterno della classe.

Relazioni fra le classi

- uso: una classe può usare oggetti di un’altra classe;
- aggregazione: una classe può avere oggetti di un’altra classe;
- ereditarietà: una classe può estendere un’altra classe.

Uso

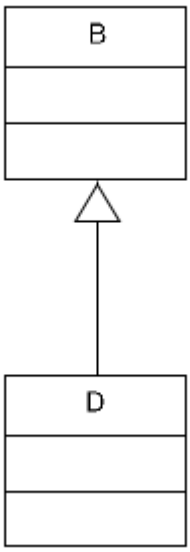
- L’uso o associazione è la relazione più semplice che intercorre fra due classi.
- Per definizione diciamo che una classe A usa una classe B se:
    - un metodo della classe A invia messaggi agli oggetti della classe B , oppure
    - un metodo della classe A crea, restituisce, riceve oggetti della classe B .

Aggregazione

- una classe A aggrega oggetti di una classe B quando la classe A contiene oggetti della classe B
- è un caso speciale della relazione di uso
- relazione has-a (ha-un)

Ereditarietà

- riuso del codice
- classe derivata o sottoclasse
- classe base o superclasse
- relazione is-a (è-un)



si ottiene il riuso del codice

Consideriamo la classe base B che ha un metodo f(...) e la classe derivata D che eredita da B. La classe D può usare il metodo f(...) in tre modi:

- lo eredita: quindi f(...) può essere usato come se fosse un metodo di D ;
- lo riscrive (override): cioè si da un nuovo significato al metodo riscrivendo la sua implementazione nella classe derivata, in modo che tale metodo esegua una azione diversa;
- lo estende: cioè richiama il metodo f(...) della classe base ed aggiunge altre operazioni.

Quando usare l'ereditarietà

- Usare l'ereditarietà solo quando il legame fra la classe base e la classe derivata è per sempre, cioè dura per tutta la vita degli oggetti, istanze della classe derivata.
- Se tale legame non è duraturo è meglio usare l'aggregazione al posto della specializzazione.

Polimorfismo

- La parola polimorfismo deriva dal greco e significa letteralmente molte forme.
- Nella OOP tale termine si riferisce ai metodi: per definizione, il polimorfismo è la capacità di un oggetto, la cui classe fa parte di una gerarchia, di chiamare la versione corretta di un metodo.
- Quindi il polimorfismo è necessario quando si ha una gerarchia di classi.

La programmazione orientata agli oggetti nei linguaggi

- OOP in Java
- OOP in PHP
- OOP in javascript
- OOP vs Procedurale

Programmazione ad oggetti vs procedurale

Programmazione imperativa

Possiamo programmare utilizzando i seguenti tipi di dati:

- Tipi di dato primitivi (int, double, char, boolean, ecc...)
- Le stringhe
- Gli array

I programmi consistono di una sequenza di comandi, strutture di controllo (cicli, scelte condizionali, ecc...) ed eventualmente metodi ausiliari che consentono di manipolare i dati per calcolare il risultato voluto.

Questo modo di programmare prende il nome di **PROGRAMMAZIONE IMPERATIVA**, imperativa in quanto basata su comandi

Programmazione imperativa

Nella programmazione imperativa:

- Un programma prevede uno stato globale costituito dai valori delle sue variabili
- I comandi del programma modificano lo stato fino a raggiungere uno stato finale (che include il risultato)

Programmazione imperativa

Ad esempio, il seguente programma (che calcola il prodotto di x e y) ha la seguente dinamica:

```
int x=10, y=3, p=0;
for (int i=0; i<y; i++)
p+=x;
```

Procedurale vs OOP

- Nella programmazione procedurale, il codice è centrale e i dati sono subordinati
- abbiamo programmi che agiscono sui dati che di solito non sono strettamente collegati
- Nella programmazione a oggetti, gli oggetti sono l'elemento centrale.
- Un oggetto consiste nei **dati** (attributi, proprietà, ...)
- e nel codice che opera su tali dati: **metodi**
- **dati e metodi** sono strettamente collegati: è il concetto di **incapsulamento**, che
- l'**incapsulamento** permette anche di nascondere l'implementazione interna, utilizzando l'oggetto attraverso l'**interfaccia** pubblica.

Per esempio,

abbiamo un numero e vogliamo raddoppiarlo.

Nel modo procedurale faremmo:

```
n = n * 2
```

Il codice moltiplica n per 2 e registra il risultato in n.

Nella programmmazione orientata agli oggetti

si invia un "messaggio" all'oggetto chiamando un metodo per raddoppiare:

```
n.double();
```

puoi creare un oggetto di tipo stringa che accetta la chiamata al metodo `double()`, ma lo implementa in maniera differente.

```
class Operazioni{


        public int double(int n){
                return n * 2;
        }

        public String double(String s){
                return s+s;
        }

}
```

Il vantaggio di questa tecnica è definito **polimorfismo**.

Se il programma richiede di replicare la procedura su un oggetto di tipo string come "bob", nel modo procedurale occorre invocare una nuova funzione con un codice e un nome differente.

OOP

Il linguaggio Java

Caratteristiche principali di Java

- 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 che si preoccupa dell'allocazione e della successiva deallocazione 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 agli 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 agli oggetti

- Orientato agli oggetti dalla base

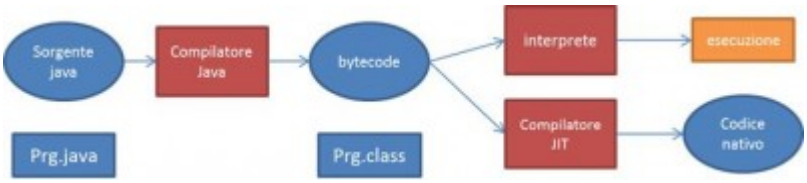
- In Java tutto è un oggetto
- Incorpora le caratteristiche
- Incapsulamento
- Polimorfismo
- Ereditarietà
- Collegamento dinamico
- Non sono disponibili
    - Ereditarietà multipla
    - Overload degli operatori

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

Versioni Java

Version	**Release date**	**End of Free Public Updates**	**Extended Support Until**
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?

Version	Release date	End of Free Public Updates	Extended Support Until
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014 January 2019 for Oracle (commercial) Indefinitely for Oracle (personal use) At least May 2026 for AdoptOpenJDK At least May 2026 for Amazon Corretto	December 2030	N/A

Versioni Java (nuovo approccio Oracle)

Version	Release date	End of Free Public Updates	Extended Support Until
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	At least October 2024 for AdoptOpenJDK At least September 2027 for Amazon Corretto	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A

Version	Release date	End of Free Public Updates	Extended Support Until
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA

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

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)

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	

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

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

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;
```

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

---

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.

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

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

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

Per i puntatori/reference, sono definiti:

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

Operatori matematici

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

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

---

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

---

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

Il controllo del flusso

Java mette a disposizione del programmatore diverse strutture sintattiche per consentire il **controllo del flusso**

Selezione, scelta condizionale

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;
}
```

Cicli definiti

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

```
for (init; condition; adjustment) {
//statements;
}
```

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.

```
while (condition) {
//statements;
```

```
}
```

```
do {

//statements;

} while (condition);
```

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
```

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

In Java gli array sono Oggetti

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

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 un riferimento a un array

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

La dichiarazione di un array non assegna alcuno spazio

`voti == null`

Creazione di un Array

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];
```

**Utilizzando un inizializzatore* (che permette anche di riempire l'array)

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

```
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;
```

In Java viene fatto il bounds checking

- Maggior sicurezza
- Maggior lentezza di accesso

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

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)

Array contenenti riferimenti ad altri array

Sintatticamente sono estensioni degli array a una dimensione

Sono possibili righe di lunghezza diverse (matrice = array di array)

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

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

---

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>);`

Esempi 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}
};
```

Stringhe e Caratteri

Caratteristiche principali

Classi disponibili

- String
  - Modella stringhe (sequenze – array di caratteri)
  - **Non modificabile** (dichiarata final)
- StringBuilder
  - Modificabile
- StringBuffer (non si usa più)
  - Modificabile
- Character
- CharacterSet

Definizione

```
String myString; myString = new String ("stringa esempio");
```

- Oppure

```
String myString = new String ("stringa esempio");
```

- Solo per il tipo String vale

```
String myString = "stringa esempio";
```

- Il carattere " (doppi apici) può essere incluso come "
- Il nome della stringa è il riferimento alla stringa stessa
- Confrontare due stringhe NON significa confrontare i riferimenti

NB: I metodi che gestiscono il tipo String NON modificano la stringa, ma ne creano una nuova

---

Concatenare stringhe

- Operatore concat
  - `myString1.concat(myString2)`
  - `String s2 = "Ciao".concat(" a tutti").concat("!");`
  - `String s2 = "Ciao".concat(" a tutti".concat("!"));`
- Utile per definire stringhe che occupano più di una riga
- Operatore + `"questa stringa" + "e` formata da tre" + "stringhe"`
- La concatenazione funziona anche con altri tipi, che vengono automaticamente convertiti in stringhe

```
System.out.println ("pi Greco = " + 3.14);
```

```
System.out.println ("x = " + x);
```

Lunghezza stringa

- int length()
  - myString.length()
  - "Ciao".length() restituisce 4
  - "".length() restituisce 0
- Se la lunghezza è N, i caratteri sono indicizzati da 0 a N-1

Carattere i-esimo

- char charAt (int)
- myString.charAt(i)
-

Confronta stringa con s

- `boolean equals(String s)` * `myString.equals("stringa")` ritorna true o false
- `boolean equalsIgnoreCase(String s)`
- `myString.equalsIgnoreCase("StRiNgA")`

Confronta con s facendone la differenza

- `int compareTo(String str)`
- `myString.compareTo("stringa")` ritorna un valore >=< 0

Trasforma int in String

- `String valueOf(int)`
- Disponibile per tutti tipi primitivi

Restituisce indice prima occorrenza di c

- `int indexOf(char c)`
- `int indexOf(char c, int fromCtrN)`

Altri metodi

- `String toUpperCase(String str)`
- `String toLowerCase(String str)`
- `String substring(int startIndex, int endIndex)`
- `String substring(int startIndex)`

Esempio

```
String s1, s2;
s1 = new String("Prima stringa");
s2 = new String("Prima stringa");
System.out.println(s1);
/// Prima stringa
System.out.println("Lunghezza di s1 = " +
s1.length());
// 26
if (s1.equals(s2)) ...
// true
if (s1 == s2) ...
// false
String s3 = s3.substring (2, 6);
// s3 == "ima s"
```

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 ...
```

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"

Metodi predicativi

Un metodo che restituisce un tipo primitivo `boolean` si definisce **predicativo** e può essere utilizzato direttamente in una condizione. In inglese sono spesso introdotti da `is` oppure `has`: isMale(), hasNext().

Esempi sui metodi

Classi Java

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

Definiscono

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

Possono essere definite

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

La "gestione" di una classe avviene mediante

- Definizione della classe
- Instanziazione di Oggetti della classe

Struttura di una classe

```
public class HelloWorld {

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

}
```

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

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

---

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

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**

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`

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

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)

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)

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

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

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

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

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

---

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");
```

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

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")

Esempi

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

All'interno della classe

- I metodi che devono inviare messaggi allo stesso oggetto cui appartengono
- non devono obbligatoriamente utilizzare la notazione puntata: 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 nuovoColore){
        colore = nuovoColore;
    }
}

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

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
- 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
```

```
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
```

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)

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

---

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.
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 { ... }
```

Terminologia

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

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

Si dice che l'eccezione

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

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
...
}
```

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
...
}
```

Costrutti try-catch possono essere annidati

(catch che include try-catch)

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 {
...
}
```

Permette a un metodo di gettare

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

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);
}
```

---

throw

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

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

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");
  }
}
```

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??
```

```
    }
}
```

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);
  }
}
```

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"); }
}
```

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);
  }
}
```

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;
  }
}
```

EccezioneBaseNegativa

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

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

```
    }
}
```

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) {
...
}
```

Java Packages

Package java.lang

- Il package java.lang è il package più importante dell'API di Java, in quanto contiene moltissime classi e interfacce fondamentali per la programmazione Java, tanto che questo package viene importato in automatico in tutti i programmi.

Astrazioni di classe, oggetto, sistema, ...

- Object
- System
- Package
- Class
- ClassLoader
- ClassValue

Classi wrapper e gestione tipi

- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Short
- Void
- Enum

Stringhe

- String
- StringBuffer
- StringBuilder

Matematica

- Math
- StrictMath
- Number

Altre funzionalità

- Compiler
- Process
- Runtime
- SecurityManager
- StackTraceElement
- Thread

- Throwable

Classe Runtime

- Questa classe astrae il concetto di runtime (esecuzione) del programma. Non ha costruttori pubblici e una sua istanza si ottiene chiamando il metodo factory getRuntime().
- Caratteristica interessante di questa classe è permette di eseguire comandi del sistema operativo direttamente da Java, come ad esempio exec (di cui esistono più versioni).
- Bisogna tener conto che l'uso della classe Runtime potrebbe compromettere la portabilità delle applicazioni, infatti questa classe dipende fortemente dal sistema operativo.

I membri della classe System.

Costanti pubbliche statiche:

- java.io.PrintStream **err**
- java.io.InputStream **in**
- java.io.PrintStream **out**

Metodi pubblici statici:

- void **arraycopy**(Object src, int srcPos, Object dest, int destPos, int length)
- long **currentTimeMillis**()
- void **exit**(int status)
- void **gc**()
- java.util.Properties **getProperties**()
- String **getProperty**(String key)
- String **getProperty**(String key, String default)
- SecurityManager **getSecurityManager**()
- void **runFinalization**()
- void **setErr**(java.io.PrintStream err)
- void **setIn**(java.io.InputStream in)
- void **setOut**(java.io.PrintStream out)
- void **setProperties**(java.util.Properties properties)
- String **setProperty**(String key, String value)
- void **setSecurityManager**(SecurityManager s)

Classe System

La classe System ha il compito di interfacciare il programma Java con il sistema operativo sul quale sussiste la virtual machine.

Tutto ciò che esiste nella classe System è dichiarato statico.

VARIABILI

Iniziamo subito esaminando tre attributi statici, che rappresentano i flussi (stream) di informazioni scambiati con la console (standard input, standard output, standard error):

```
static PrintStream out

static PrintStream err

static InputStream in
```

- Ciascuno di questi tre attributi è un oggetto e sfrutta i metodi della classe relativa.
- l'oggetto out è di tipo PrintStream e viene usato per indicare l'output di default del sistema.

- l'oggetto err, anch'esso di tipo PrintStream, che viene usato per segnalare gli errori che avvengono durante l'esecuzione del programma.
- l'oggetto in è di tipo InputStream: serve per ricevere il flusso di informazioni dallo standard input. p.es la tastiera.
- E' possibile modificare il puntamento di queste tre variabili verso altre fonti di input o di output usando i metodi statici setOut(), setErr() e setIn().

METODI Principali

- il metodo arraycopy() permette di copiare il contenuto di un array in un altro.
- il metodo exit(int code) che consente di bloccare istantaneamente l'esecuzione del programma.

```
if (continua == false) {
    System.err.println("Si è verificato un problema!");
    System.exit(0);
}
```

Altri metodi interessanti:

```
    setProperty(String key, String value) e
    getProperty(String key) che servono rispettivamente ad impostare le
proprietà del sistema e a recuperare informazioni sulle proprietà del
sistema.
```

Gli oggetti di tipo Properties sono specializzazioni di tabelle hash di Java, semplici coppie chiave-valore.

Per esempio:

```
System.out.print("Versione Java Runtime Environment (JRE): ");
System.out.println(System.getProperty("java.version"));
*
System.out.print("Java è installato su: ");
System.out.println(System.getProperty("java.home"));
```

impostare una nuova proprietà mediante il codice:

```
System.setProperty("User.lastName", "Verdi");
```

Le proprietà automaticamente disponibili nell'ambiente Java.

Chiave	Valore
java.version	La versione di Java in uso.
java.vendor	Il produttore della versione di Java in uso.
java.vendor.url	L'URL del produttore della versione di Java in uso.
java.home	La directory di installazione di Java.
java.vm.specification.version	La versione delle specifiche della macchina virtuale in uso.
java.vm.specification.vendor	Il produttore delle specifiche della macchina virtuale in uso.
java.vm.specification.name	Il nome delle specifiche della macchina virtuale in uso.
java.vm.version	La versione della macchina virtuale in uso.
java.vm.vendor	Il produttore della macchina virtuale in uso.
java.vm.name	Il nome della macchina virtuale in uso.
java.specification.version	La versione delle specifiche di Java in uso.
java.specification.vendor	Il produttore delle specifiche di Java in uso.
java.specification.name	Il nome delle specifiche di Java in uso.
java.class.version	La versione delle classi di Java.
java.class.path	Il percorso delle classi di Java.
java.library.path	Il percorso delle librerie di Java.
java.io.tmpdir	Il percorso della directory dei file temporanei.
java.ext.dirs	I percorsi delle directory che contengono le estensioni di Java.
os.name	Il nome del sistema operativo in uso.
os.arch	L'architettura del sistema operativo in uso.
os.version	La versione del sistema operativo in uso.
file.separator	La sequenza per la separazione degli elementi dei percorsi nel sistema in uso.
path.separator	La sequenza per la separazione dei percorsi nel sistema in uso.
line.separator	La sequenza impiegata dal sistema in uso per esprimere il ritorno a capo.
user.name	Il nome dell'utente che sta usando l'applicazione.
user.home	La home directory dell'utente che sta usando l'applicazione.
user.dir	L'attuale cartella di lavoro dell'utente che sta usando l'applicazione.

I membri della classe Object.

Costruttori pubblici: Object()

Metodi protetti: Object **clone**() void **finalize**()

Metodi pubblici:

- boolean **equals**(Object obj)
- final Class **getClass**()
- int **hashCode**()
- final void **notify**()
- final void **notifyAll**()
- String **toString**()
- final void **wait**()
- final void **wait**(int millis)
- final void **wait**(int millis, int nanos)

In Java everything is object!

I membri della classe Math.

Questa classe serve per fare calcoli matematici e ha due attributi:

```
static double E ; //E di Eulero
static double PI; //Pi greca
```

metodi disponibili per le principali funzioni matematiche:

- valore assoluto,
- tangente,
- logaritmo,
- potenza,
- massimo,
- minimo,
- seno,
- coseno,
- esponenziale,
- radice quadrata
- arrotondamento classico, per eccesso e per difetto
- generazione di numeri casuali

Costanti pubbliche statiche:

- double E
- double PI

Metodi pubblici statici:

- double **abs**(double a)
- float **abs**(float a)
- int **abs**(int a)
- long **abs**(long a)
- double **acos**(double a)
- double **asin**(double a)
- double **atan**(double a)
- double **atan2**(double y, double x)
- double **ceil**(double a)
- double **cos**(double a)
- double **exp**(double a)
- double **floor**(double a)
- double **log**(double a)
- double **max**(double a, double b)
- float **max**(float a, float b)
- int **max**(int a, int b)

- long **max**(long a, long b)
- double **min**(double a, double b)
- float **min**(float a, float b)
- int **min**(int a, int b)
- long **min**(long a, long b)
- double **pow**(double a, double b)
- double **random**()
- double **rint**(double a)
- long **round**(double a)
- int **round**(float a)
- double **sin**(double a)
- double **sqrt**(double a)
- double **tan**(double a)
- double **toDegrees**(double angrad)
- double **toRadians**(double angdeg)

Classi wrapper:

- Nella versione 1.5 di Java è stata introdotta una funzionalità davvero comoda che viene chiamata boxing (inscatolamento) che permette ai valori dei tipi primitivi di essere convertiti in oggetti, e viceversa. In particolare:
- l'autoboxing è un casting automatico che permette ai valori dei tipi primitivi di essere convertiti in oggetti
- l'unboxing effettua il casting inverso
- Gli oggetti che rappresentano i tipi primitivi sono detti classi wrapper (in italiano "involucro") che sono classi che fanno da contenitore a un tipo di dato primitivo, astraendo proprio il concetto di tipo. **N.B. Tutte le classi wrapper sono classi final per cui non possono essere estese.**

Classi "Wrapper" per tipi Primitivi

- Se si vogliono trattare anche i dati primitivi come oggetti si possono utilizzare le classi "wrapper"
- Un oggetto di una classe "wrapper" incorpora un dato primitivo e fornisce metodi per operare su di esso
- Le classi wrapper dei tipi primitivi sono definite nel package java.lang

Corrispondenza

Tipo Primitivo	ClasseWrapper
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
void	Void

- In Java, ogni tipo primitivo possiede una corrispondente classe wrapper: Byte, Short, Integer, Long, Float, Double, Boolean, Character.
- Ciascuna di queste classi permette di manipolare i valori di tipo primitivo come se fossero valori di oggetti.
- Spesso abbiamo a che fare con tipi primitivi (int, double, boolean, …) che sono tipi semplici e, quindi, non possiedono metodi.
- I wrapper, invece, essendo degli oggetti, sono dotati di metodi ed attributi.

- Prima dell'introduzione dell'autoboxing, programmando in Java ci si poteva trovare nella necessità di convertire un tipo primitivo nella sua corrispondente classe wrapper.

```
Integer x = new Integer(10);
Double y = new Double(5.5f);
Boolean z = Boolean.parseBoolean("true");
```

Occorreva prima creare un nuovo oggetto di una classe wrapper.

Le stesse operazioni precedenti possono essere ora eseguite mediante il seguente codice.

```
Integer x = 10;
Double y = 5.5f;
Boolean z = true;
```

L'autoboxing permette al developer di non preoccuparsi delle operazioni di conversione. Da notare che è grazie all'autoboxing che possiamo inserire i tipi primitivi all'interno delle Collection in maniera totalmente trasparente e senza preoccuparci di convertire i dati.

Classi wrapper	tipo primitivo
Boolean	è la classe wrapper per il tipo boolean.
Byte	è la classe wrapper per il tipo byte.
Character	è la classe wrapper per il tipo char.
Double	è la classe wrapper per il tipo double.
Float	è la classe wrapper per il tipo float.
Integer	è la classe wrapper per il tipo int.
Long	è la classe wrapper per il tipo long.
Short	è la classe wrapper per il tipo short.

Metodi statici della famiglia parse:

- `Byte.parseByte(String s)`
- `Double.parseDouble(String s)`
- `Float.parseFloat(String s)`
- `Integer.parseInt(String s)`
- `Long.parseLong(String s)`
- `Short.parseShort(String s)`

Classe Character

La classe involucro Character si trova in java.lang

Include metodi statici per verificare le proprietà di un carattere

Esempio

String s = "Stringa Generica." char c = s.charAt (0); ... Character.isUpperCase(c) ... // true ... Character.isDigit(c) ... // false ...

Predicati vari

- boolean isLetter (char c)
- boolean isDigit (char c)
- boolean isLetterOrDigit (char c)
- boolean isSpaceChar (char c)
- boolean isLowerCase (char c)
- boolean isUpperCase (char c)

Trasformazioni varie

- char toUpperCase (char c)
- char toLowerCase (char c)

String e StringBuilder

- le classi String e StringBuilder del package java.lang
- La classe String ha lo scopo di rappresentare stringhe (sequenze) di caratteri che non devono essere modificate dopo essere state costruite (oggetti immutabili)
- La classe StringBuilder ha lo scopo di rappresentare stringhe (sequenze) di caratteri che possono essere modificate dopo essere state costruite

Definizione di variabili

- tipo nome; oppure
- tipo nome1,..., nomeN ;
- String nome;
- StringBuilder risultato;
- Dopo la definizione esiste solo il riferimento, non un oggetto di tipo nome, null, String !

null

- Il valore speciale null è il valore iniziale di default per qualunque variabile di tipo strutturato.
- indica che il riferimento è nullo e non c'e' nessun oggetto riferito
- nome non è un oggetto di tipo String è solo un riferimento utilizzabile per accedere ad un oggetto String

Operatore new

- L'operatore new NomeClasse crea un nuovo oggetto con le proprietà definite in NomeClasse (istanza dellaclasse) e ritorna il riferimento ad esso
- L'operatore new dà luogo all'invocazione di un metodo costruttore passandogli gli argomenti necessari
- Il costruttore invocato deve essere di una classe uguale o "compatibile" con la definizione della variabile
- Ogni classe può avere più costruttori che si differenziano per la lista degli argomenti

Costruttori

- La scelta del costruttore da invocare avviene tramite gli argomenti attuali che vengono passati
- New di un oggetto String

```
String saluto;
saluto = new String("Ciao ciao");
```

- L'operatore new può essere usato al momento della definizione

```
String saluto = new String("Ciao ciao");
```

- Solo per la classe String , in quanto di uso molto comune, Java offre la forma compatta

```
String s = "Ciao ciao";
```

Una particolarità di String

- Usare esplicitamente new oppure la forma abbreviata per inizializzare un oggetto String non è esattamente la stessa cosa
- Se si usa esplicitamente new, la Java Virtual Machine crea oggetti distinti anche se di contenuto uguale
- Se non si usa esplicitamente new, la Java Virtual Machine evita di creare oggetti distinti ma dal contenuto uguale

I più importanti metodi di cui sono dotati gli oggetti di tipo String.

Tipo restituito	**Metodi e parametri**	**Descrizione**
int	charAt(int i)	Restituisce il carattere alla posizione i.
boolean	endsWith(String s)	Restituisce true se l'oggetto di invocazione termina con la sottostringa s.
boolean	equals(String s)	Restituisce true quando l'oggetto di invocazione e s rappresentano la medesima sequenza.
int	indexOf(char c)	Restituisce la prima posizione del carattere c, oppure -1 nel caso tale carattere non faccia parte della stringa.
int	indexOf(char c, int i)	Come il precedente, con la differenza che la ricerca del carattere c comincia dalla posizione i.
int	indexOf(String s)	Restituisce la prima posizione della sottostringa s, oppure -1 nel caso tale sottostringa non compaia nell'oggetto di invocazione.
int	indexOf(String s, int i)	Come il precedente, con la differenza che la ricerca della sottostringa s prende piede dalla posizione i.
int	length()	Restituisce la lunghezza della stringa.
String	replace(char c1, char c2)	Restituisce una nuova stringa, ottenuta dall'oggetto di invocazione sostituendo il carattere c2 ad ogni occorrenza del carattere c1.
boolean	starsWith(String s)	Restituisce true se l'oggetto di invocazione inizia con la sottostringa s.
String	toLowerCase()	Restituisce una nuova stringa, ottenuta traslando verso il minuscolo ogni carattere dell'oggetto di invocazione.

Tipo restituito	Metodi e parametri	Descrizione
String	toUpperCase()	Restituisce una nuova stringa, ottenuta traslando verso il maiuscolo ogni carattere dell'oggetto di invocazione.
String	trim()	Restituisce una nuova stringa, ottenuta dall'oggetto di invocazione eliminando gli spazi che precedono il primo carattere significativo e quelli che seguono l'ultimo. In pratica, " ciao ".trim() restituisce "ciao".

- Le stringhe in Java sono oggetti.
- La particolarità di questa classe è quella di essere l'unica classe che è possibile istanziare come se fosse un tipo di dato primitivo.

```
int compareTo(String other)
```

Esegue una comparazione lessicale. Ritorna un intero:

- < 0 se la stringa corrente è minore della stringa other
- = 0 se le due stringhe sono identiche
- > 0 se la stringa corrente è maggiore di other

```
int indexOf(int ch)
```

Restituisce l'indice del carattere specificato

```
int lastIndexOf(int ch)
```

E' come indexOf() ma viene restituito l'indice dell'ultima occorrenza trovata

```
int length()
```

Restituisce il numero di caratteri di cui è costituita la stringa corrente

```
String replace(char oldChar, char newChar)
```

Restituisce una nuova stringa, dove tutte le occorrenze di oldChar sono rimpiazzate con newChar

```
String substring(int startIndex)
```

Restituisce una sottostringa della stringa corrente, composta dai caratteri che partono dall'indice startIndex alla fine

```
String substring(int startIndex, int number)
```

Restituisce una sottostringa della stringa corrente, composta dal numero number di caratteri che partono dall'indice startIndex

```
String toLowerCase()
```

Restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri minuscoli

```
String toUpperCase()
```

Restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri maiuscoli

Package java.util

Il package java.util contiene una serie di classi utili come il framework "Collections" per gestire collezioni eterogenee di ogni tipo, il modello a eventi, classi per la gestione facilitata delle date e degli orari, classi per la gestione dell'internazionalizzazione e tante altre utilità come un separatore di stringhe (StringTokenizer), un generatore di numeri casuali ecc.

StringTokenizer

La classe StringTokenizer permette l'estrazione di sottostringhe

- StringTokenizer (String str, String delim)
- Costruisce un estrattore di token per la stringa str
- delim e' il delimitatore ricercato tra i token estratti
- La classe StringTokenizer mette quindi a disposizione metodi per la gestione dei token
    - public boolean hasMoreTokens()
    - public String nextToken()

Esempio

// il numero di token e' noto: nome, eta', reddito String str; StringTokenizer st = new StringTokenizer(str," "); // Anche: StringTokenizer st = // new StringTokenizer (str); while (st.hasMoreTokens()){ String token = st.nextToken(); ... Integer.parseInt(token) ... // int eta = Integer.parseInt (st.nextToken ()); // double reddito = Double.parseDouble // (st.nextToken ()); }

Classe StringTokenizer

Spesso risulta necessario manipolare dei token di testo. Una semplice classe che permette di separare i contenuti di una stringa in più parti, chiamate token, è la classe **StringTokenizer**.

Questa classe si utilizza solitamente per estrarre le parole di una stringa. L'utilizzo di base è estremamente semplice, occorrono: una stringa da "navigare", cioè da cui estrarre i token un delimitatore, che serve per identificare i token Un token è, quindi, la sequenza massima di caratteri consecutivi che non sono delimitatori.

CREARE OGGETTO STRINGTOKENIZER

Occorre creare in prima istanza l'oggetto StringTokenizer, usando il costruttore dell'omonima classe.

- Il costruttore può accettare da 1 a 3 parametri: la stringa da cui estrarre i token il delimitatore, che può essere: esplicito [st2 – st3] di default " \t\n\r\f" (notare che il primo delimitatore e uno spazio) [st1] un booleano che, se settato a true, considera token anche gli stessi delimitatori

```
StringTokenizer st1 = new StringTokenizer("Stringa da dividere");
StringTokenizer st2 = new StringTokenizer("Stringa sezionata", ";");
StringTokenizer st3 = new StringTokenizer("Ciao Mamma", "a", true);
```

Per scandire l'intero testo si può usare un ciclo while con all'interno l'invocazione del metodo hasMoreTokens() che ritorna true se sono presenti altri token, altrimenti false. Per stampare il token appena recuperato si può invocare il metodo nextToken() sull'oggetto StringTokenizer.

```
StringTokenizer st = new StringTokenizer("Stringa da dividere");
while (st.hasMoreTokens()) {
    // Due metodi per fare la stessa cosa
System.out.println(st.nextToken());
System.out.println(st.nextElement().toString());
}
```

Costruttori pubblici:

Costruttore	**definizione**
StringTokenizer(String str)	Costruisce uno StringTokenizer per la stringa str, che come delimitatori usa i caratteri "\t\n\r\f".
StringTokenizer(String str, String delim)	Costruisce uno StringTokenizer per la stringa str, che come delimitatori usa i caratteri contenuti nella stringa delim.
StringTokenizer(String str, String delim, boolean returnDelims)	Costruisce uno StringTokenizer per la stringa str, che come delimitatori usa i caratteri contenuti nella stringa delim. Se returnDelims ﷺ true, i caratteri divisori verranno restituiti come token.

Metodi pubblici:

Metodo	Definizione
int countTokens()	Restituisce il numero dei token elaborati.
boolean hasMoreElements()	Come il successivo hasMoreTokens().
boolean hasMoreTokens()	Restituisce true se ci sono ancora dei token da considerare.
Object nextElement()	Restituisce il token successivo, sotto forma di Object.
String nextToken()	Restituisce il token successivo, sotto forma di String.
String nextToken(String delim)	Imposta una nuova serie di caratteri delimitatori, quindi restituisce il token successivo.

Classe StringBuffer

Un oggetto String

- NON è modificabile
- Una volta creato non possiamo aggiungere, eliminare, modificare caratteri (i metodi visti creano nuove stringhe)
- Tale restrizione è dovuta a ragioni di efficienza

Le considerazione precedenti non sono vere per la classe StringBuffer

Esempio

`StringBuffer myStringBuffer = new stringBuffer ("stringa modificabile");

myStringBuffer.setCharAt (8, 'M'); // Trasforma in "stringa Modificabile"`

Si usa raramente

Un oggetto StringBuffer non può essere utilizzato per operazioni di I/O

```
System.out.println (myStringBuffer.toString());
```

Metodi:

- Aggiunta di caratteri `myStringBuffer.append ("aggiunta");`
- insert
- delete
- reverse

I più importanti metodi di cui sono dotati gli oggetti di tipo StringBuffer.

Tipo restituito	Metodi e parametri	Descrizione

Tipo restituito	Metodi e parametri	Descrizione
StringBuffer	append(boolean b)	Aggiunge il valore b in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(char c)	Aggiunge il carattere c in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(char[] c)	Aggiunge i caratteri contenuti nell'array in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(double d)	Aggiunge il valore di d in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(float f)	Aggiunge il valore di f in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(int i)	Aggiunge il valore di i in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(long l)	Aggiunge il valore di l in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(Object obj)	Aggiunge il valore di obj.String() in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(String s)	Aggiunge s in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	append(StringBuffer s)	Aggiunge s in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
char	chatAt(int i)	Restituisce il carattere alla posizione i.
StringBuffer	delete(int start, int end)	Rimuove tutti i caratteri dall'indice start (incluso) all'indice end (escluso). Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.

Tipo restituito	Metodi e parametri	Descrizione
StringBuffer	deleteCharAt(int i)	Rimuove il carattere alla posizione i. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
int	indexOf(String s)	Restituisce la prima posizione della sottostringa s, oppure -1 nel caso tale sottostringa non compaia nell'oggetto di invocazione.
int	indexOfString(String s, int i)	Come il precedente, con la differenza che la ricerca della sottostringa s prende prende piede dalla posizione i.
StringBuffer	insert(int offset, boolean b)	Aggiunge il valore di b alla stringa, inserendolo alla posizione offset. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, char c)	Aggiunge il carattere c alla stringa, inserendolo alla posizione offset. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, char[] c)	Aggiunge i caratteri contenuti nell'array alla stringa, inserendoli alla posizione offset. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, double d)	Aggiunge il valore di d alla stringa, inserendolo alla posizione offset. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, float f)	Aggiunge il valore di f alla stringa, inserendolo alla posizione offset. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, int i)	Aggiunge il valore di i alla stringa, inserendolo alla posizione offset. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, long l)	Aggiunge il valore di l alla stringa, inserendolo alla posizione offset. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, Object obj)	Aggiunge il valore di obj.toString() in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
StringBuffer	insert(int offset, String s)	Aggiunge s in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.

Tipo restituito	Metodi e parametri	Descrizione
StringBuffer	insert(int offset, StringBuffer s)	Aggiunge s in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso StringBuffer.
int	length()	Restituisce la dimensione della stringa.
StringBuffer	setCharAt(int i, char c)	Cambia in c il carattere alla posizione i.
void String toString()	Restituisce un oggetto String con il medesimo contenuto dello StringBuffer di invocazione.	

Date e orari

Le date sono degli oggetti molto complesse da gestire: assumono forme diverse a seconda del luogo geografico in cui ci troviamo. La manipolazione delle date e delle ore è una delle attività ricorrenti di un programmatore.

Se lavoriamo con Java 7, a classe principale per gestire date e orari è Calendar (che ha sostituito la classe Date, dichiarata **deprecata__).

Altre classi utili sono **GregorianCalendar**, **SimpleTimeZone** e **SimpleTimeZone**. Inoltre sono disponibili le più moderne LocalDate e LocalDateTime. Oltre a queste classi, è molto probabile che serva utilizzarne altre quali: **DateFormat** e **SimpleDateFormat**, che permettono la trasformazione da stringa a data e viceversa.

Novità in Java 8: LocalDate, LocalTime, LocalDateTime

```
        LocalDate oggi = LocalDate.now(); // Data di oggi
        System.out.println("oggi: " + oggi);

        //settare una data precisa (mese 1-based)
        LocalDate mauroBirthday = LocalDate.of(1969, 7, 28);
        //Per il mese possiamo usare le costanti di Month
        mauroBirthday = LocalDate.of(1969, Month.JULY, 28);

        System.out.println("mauroBirthday: " + mauroBirthday);

        LocalDate inizioCorsi = LocalDate.of(2017, Month.OCTOBER, 12);
        LocalDate natale = LocalDate.of(2017, Month.DECEMBER, 25);

        System.out.println("Fino a  natale: " + inizioCorsi.until(natale));
        System.out.println("Fino a  natale: " + inizioCorsi.until(natale,
ChronoUnit.DAYS));

        LocalDate festaLiberazione = LocalDate.of(2017, Month.APRIL, 25);
        LocalDate natale = LocalDate.of(2017, Month.DECEMBER, 25);

        System.out
              .println("Fino a  natale: " + festaLiberazione.until(natale));
        System.out.println("Fino a  natale: "
              + festaLiberazione.until(natale, ChronoUnit.DAYS));

        System.out.println(oggi.plusMonths(1));
        System.out.println(oggi.minusMonths(1));

        DayOfWeek inizioMillenio = LocalDate.of(2000, 1, 1).getDayOfWeek();
        System.out.println("inizioMillenio: " + inizioMillenio);
        System.out.println(inizioMillenio.getValue());
        System.out.println(DayOfWeek.SATURDAY.plus(3));
```

```
        LocalDateTime ldt = LocalDateTime.now();
        System.out.println(ldt);

        LocalDate ld = LocalDate.of(2009, 1, 28);
        System.out.println(ld);

        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("M/d/yyyy");
        System.out.println(dtf.format(ld));
```

Convertire LocalDate a java.sql.Date

```
import java.sql.Date;
//...
LocalDate locald = LocalDate.of(1969, 07, 28);
Date date = Date.valueOf(locald);
r.setDateOfBirth(date);
```

---

l'operazione contraria è

```
Date date = r.getDate();
LocalDate localD = date.toLocalDate();
```

r è il record e .getDate() il metodo per farsi ritornare la data. Se avessi un campo dataNascita ilmetodo dovrebbe chiamarsi getDateNascita().

Usa le classi del package java.time invece di java.util.Date e java.sql.Date con JDBC 4.2 o superiore.

Esempio con PreparedStatement

```
myPreparedStatement.setObject(
    … ,                                         // qui passa il numero
ordinale dell'argomento.
    myJavaUtilDate.toInstant()                  // Converti da
`java.util.Date` nel più moderno `java.time.Instant` (UTC).
        .atZone( ZoneId.of( "Europe/Rome" ) )   // Sette un time zone
particolare, per determinare la data. Instanziando un `ZonedDateTime`.
        .toLocalDate()                          // Estrai la data di tipo
`java.time.LocalDate` dall'oggetto.
)
```

---

esempi

```
LocalDate todayLocalDate = LocalDate.now( ZoneId.of( "Europe/Paris" ) );
// Usare "continent/region" come region name; non usare quelli codificati
da 3 lettere.

LocalDate localDate = ResultSet.getObject( 1 , LocalDate.class );

//la questione è irrelevante in JDBC 4.2 o successive.
```

Converti a java.sql.Date

```
java.sql.Date sqlDate = java.sql.Date.valueOf( todayLocalDate );

//viceversa

LocalDate localDate = sqlDate.toLocalDate();
```

Nuovi metodi di java.util.Date

- java.util.Date.from(Instant)
- java.util.Date::toInstant.

```
Instant instant = myUtilDate.toInstant();

ZoneId zoneId = ZoneId.of ( "America/Montreal" );
ZonedDateTime zdt = ZonedDateTime.ofInstant ( instant , zoneId );
LocalDate localDate = zdt.toLocalDate();
```

```
public class MainClass {

  public static void main(String[] args) {
    java.util.Date utilDate = new java.util.Date();
    java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());
    System.out.println("utilDate:" + utilDate);
    System.out.println("sqlDate:" + sqlDate);

  }

}

//http://www.java2s.com/Tutorial/Java/0040__Data-
Type/ConvertfromajavautilDateObjecttoajavasqlDateObject.htm
```

GreorianCalendar

La classe **GregorianCalendar** è molto semplice da utilizzare.

Sono disponibili diversi costruttori. Il costruttore senza parametri inizializza l'oggetto con la data e l'ora attuale. Con il metodo get(), ereditato da Calendar, è possibile ricevere tutte le informazioni disponibili per l'oggetto di tipo data.

Primo esempio

stampiamo semplicemente la data odierna con l'orario attuale.

```
GregorianCalendar calendario = new GregorianCalendar();
int anno = calendario.get(Calendar.YEAR);
int mese = calendario.get(Calendar.MONTH) + 1;
int giorno = calendario.get(Calendar.DATE);
int ore = calendario.get(Calendar.HOUR);
int min = calendario.get(Calendar.MINUTE);
int sec = calendario.get(Calendar.SECOND);

System.out.println(giorno + "/" + mese + "/" + anno);
System.out.println(ore + ":" + minuti + ":" + secondi);
```

Formattare la data: SimpleDateFormat

la classe SimpleDateFormat che permette di trattare le date nel formato più adatto alla nostra esigenza.

Secondo esempio

come stampare la data odierna usando **SimpleDateFormat** per formattare l'output.

```
GregorianCalendar calendario = new GregorianCalendar();
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yy - HH:mm:ss");
System.out.println(sdf.format(calendario.getTime()));
```

```
- Il costruttore della classe SimpleDateFormat prende in ingresso una
stringa che rappresenta il formato della data che vogliamo stampare.
- Il metodo getTime() della classe GregorianCalendar restituisce un'istanza
di Date.
- Il metodo format() della classe SimpleDateFormat, che restituisce in
ingresso una Date, restituisce una stringa che corrisponde al formato che
```

```
abbiamo impostato.
- E' possibile sfruttare la classe SimpleDateFormat anche per ottenere
un'istanza della classe Calendar.
```

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yy - HH:mm:ss");
String miaData = "15/04/1988";
GregorianCalendar calendario = new GregorianCalendar();
try {
    calendario.setTime(sdf.parse(miaData));
} catch(ParseException exc) {
    exc.printStackTrace();
}
```

```
Il metodo __parse()__ della classe SimpleDateFormat riceve in ingresso una
stringa e restituisce un oggetto Date.
Il metodo __setTime()__ della classe GregorianCalendar ci permette di
impostare la data.
```

Bisogna utilizzare un blocco try-catch perché potrebbe essere sollevata una **ParseException**, nel caso in cui una stringa passata al metodo parse(), non rappresenti una data convertibile.

Terzo esempio

come convertire una data dal formato americano in quello italiano utilizzando le tecniche analizzate in precedenza.

```
SimpleDateFormat formatIT = new SimpleDateFormat("dd/MM/yyyy");
SimpleDateFormat formatUS = new SimpleDateFormat("yyyy/MM/dd");

Date dataIT;
try {
    dataIT = formatUS.parse("2017/12/25");
    String dataUS = formatIT.format(dataIT);
    System.out.println(dataUS);
} catch(ParseException exc) {
    exc.printStackTrace();
}
```

Volendo confrontare due date possiamo utilizzare i metodi - **after()** - **before()** - **equals()** presenti nella classe Date.

```
GregorianCalendar c1 =
new gregorianCalendar(2013, GregorianCalendar.FEBRUARY, 05);
```

```
GregorianCalendar c2 =
new gregorianCalendar(2013, GregorianCalendar.FEBRUARY, 05);

Date data1 = c1.getTime();
Date data2 = c2.getTime();
```

Il metodo:

```
data1.after(data2)  restituirà false
data1.equals(data2) restituirà false
data1.before(data2) restituirà true
```

Java Legacy

I membri resi disponibili dalla classe Date.

Costruttori pubblici:

..	..
Date()	Costruisce un oggetto Date che incapsula la data e l'ora correnti.
Date(long t)	Costruisce un oggetto Date che incapsula la data e l'ora espressi dall'argomento t. L'argomento è un long che riporta i millisecondi trascorsi dal 1è Gennaio 1970 sino alla data rappresentata.

Metodi pubblici:

..	..
boolean after(Date d)	Restituisce true se la data di invocazione è successiva alla data d.
boolean before(Date d)	Restituisce true se la data di invocazione è precedente alla data d.
Object clone()	Clona l'oggetto. Date implementa l'interfaccia Cloneable.
int compareTo(Date d)	Compara la data di invocazione con d. Restituisce 0 se le due date sono uguali, un valore negativo se la data di invocazione precede la data d o un valore positivo se la data di invocazione è successiva alla data d.
int compareTo(Object)	Se l'argomento fornito è una istanza di Date, agisce come compareTo(Date d). In caso contrario, propaga una ClassCastException.
boolean equals(Object o)	Restituisce true se l'argomento è una data equivalente a quella di invocazione.

..	..
long getTime()	Restituisce la data dell'oggetto sotto forma di valore long, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata.
int hashCode()	Calcola un codice hash per l'oggetto.
void setTime(long t)	Imposta la data rappresentata con un argomento di tipo long, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata.
String toString()	Fornisce una rappresentazione in stringa della data.

Classe GregorianCalendar.

Costruttori pubblici:

GregorianCalendar Costruisce un GregorianCalendar che rappresenta la data e l'ora correnti.

..	..
GregorianCalendar(int year, int month, int date)	Costruisce un GregorianCalendar che rappresenta la data espressa mediante gli argomenti forniti. GregorianCalendar(int year, int month, int date,int hour, int minute)

Metodi pubblici:

..	..
boolean after(Object o)	Restituisce true se la data rappresentata È successiva alla data espressa dall'oggetto o, che deve essere istanza di Calendar.
boolean before(Object o)	Restituisce true se la data rappresentata È precedente alla data espressa dall'oggetto o, che deve essere istanza di Calendar.
Object clone()	Clona l'oggetto. GregorianCalendar implementa l'interfaccia Cloneable.
booleans equals(Object o)	Restituisce true se l'argomento rappresenta una data equivalente a quella rappresentata dall'oggetto di invocazione.
int get(int field)	Recupera il valore di uno dei campi della data rappresentata. L'argomento specifica il campo di interesse.
Date getTime()	Restituisce un oggetto Date che rappresenta la data incapsulata dall'oggetto di invocazione.
long getTimeInMillis()	Restituisce la data rappresentata dall'oggetto sotto forma di valore long, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata.
int hashCode()	Calcola un codice hash per l'oggetto.
void set(int field, int value)	Imposta su value il valore del campo rappresentato dall'intero field.

..	..
void setTime(Date d)	Imposta la data rappresentata prelevando il suo valore all'argomento d.
void SetTimeInMillis(long l)	Imposta la data rappresentata mediante un argomento di tipo long, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata.
String toString()	Fornisce una rappresentazione in stringa della data. rappresentata.

Costanti statiche:

..	..
AM_PM	Il campo che informa se l'ora espressa è prima o dopo mezzogiorno.
DAY_OF_MONTH	Il campo che riporta il giorno del mese.
DAY_OF_WEEK	Il campo che riporta il giorno della settimana.
DAY_OF_YEAR	Il campo che riporta il giorno dell'anno.
HOUR	Il campo che riporta l'ora del mattino o del pomeriggio, a seconda del contenuto del campo AM_PM
HOUR_OF_DAY	Il campo che riporta l'ora del giorno, in un intervallo tra 0 e 23.
MILLISECOND	Il campo che riporta i millisecondi.
MINUTE	Il campo che riporta i minuti.
MONTH	Il campo che riporta il mese.
SECOND	Il campo che riporta i secondi.
WEEK_OF_MONTH	Il campo che riporta la settimana del mese.
WEEK_OF_YEAR	Il campo che riporta la settimana dell'anno.
YEAR	Il campo che riporta l'anno.