Quaderno di Java

Tutte le informazioni scritte in corsivo presenti nel testo, portano ad un collegamento sull'argomento con spiegazioni più approfondite

INTRODUZIONE

Come nasce Java?

Verso l'inizio degli anni ottanta, e la fine degli anni novanta, la *Sun Micro System* diede avvio allo *Stealth Project* (Guidato dal *Green Team*, formato dai migliori programmatori di quel tempo), un progetto che potesse approfittare della "next-wave", della rivoluzione tecnologica, cioè computer sempre più potenti e meno costosi, per far si che tutti potessero utilizzarne uno, andando cosi a creare un nuovo linguaggio di alto livello più semplice e facile per i programmatori futuri. Tutto questo però doveva andare in contrasto con i linguaggi di alto livello presenti a quel tempo, tra cui:

- Il 'C'
- Il 'C++', un evoluzione del 'C' più OOP (Object Orientated Programming)
- Il 'Java-Script', essenziale per la produzione delle pagine Web

Dopo anni di lavoro, nasce una prima versione di questo nuovo linguaggio: *OAK*, ma che non ebbe grande successo. La Sun Micro System stessa, infatti, stava quasi per cancellare l'idea, finché il Green Team non decise di approfittare del web, per andare a sponsorizzare questo nuovo linguaggio, creando un'intera pagina scritta soltanto con quest'ultimo (aggiungendo applicazioni e feature). Dopo questa mossa, OAK venne sempre più utilizzato, e venne rinominato poi JAVA e aggiornato progressivamente fino ad arrivare alla versione 1.8

Come Divento famoso Java?

Anche se Java si sviluppò e diventò famoso grazie a questa sua 'propaganda', le motivazioni per cui divenne cosi utilizzato sono anche altre. Le più importanti sono, per esempio, le novità che portava rispetto ai linguaggi di alto livello a quel tempo presenti. Andiamo ad analizzarle:

- La prima, difatti, era una migliore gestione della memoria. Mentre in altri linguaggi si preferiva utilizzare un "Buco unico" per scaricare e gestire calcoli, variabili, testi… In Java, in base anche alle decisione del programmatore, quest'ultimi potevano essere tenuti, cancellati, riutilizzati dentro il programma con il passare del codice (Vedremo meglio dopo nella lezione sulle variabili)
- Un'altra cosa che venne fatta, e di cui abbiamo anche parlato prima, era una sintassi più semplice rispetto a quella che trovavamo, per esempio, nel 'C'. Tutto questo sempre per facilitare i programmatori futuri.
- L'ultima cosa, ma anche la più importante è la introduzione del *JDK*, un compilatore automatico, che permetteva a Java di essere utilizzato ovunque, e non soltanto su computer specifici.

Un altra cosa importante di Java è anche il tipo di programmazione: non procedurale, bensì OOP, dove si preferiva l'utilizzo di Classi e Metodi per semplificare il lavoro ma anche il programma stesso (evitando cosi errori di sintassi o matematici). Questo tipo di programmazione la vediamo anche in C++

Un primo programma…

Saltando la parte di installazione del JDK e di un compilatore per Java (dove le informazioni possono essere trovate a lezione 3 e 5 di questo sito: https://www.html.it/guide/guida-java/) andiamo a parlare di come potremmo introdurre un nostro primo programma. Per prima cosa andiamo a a richiamare il Package (La cartella) nel quela il nostro file si trova, andando a dividere gli spazi presenti nel nome utilizzando un punto

```
package Corso.Java;
```

In questo esempio, la nostra cartella si chiama 'Corso Java', ma noi abbiamo messo un punto per suddividere il nome. Dopo questo , andiamo invece a richiamare la classe e il metodo principale per inizializzare il programma:

```
package Corso.Java;

no usages
class Corso_Java {

    no usages
    public static void Corso_Java (String [] args) {

        System.out.print("Hello Word !");
    }
}
```

Qua dobbiamo invece ad andare a ‘sottolineare’ una serie di cose:

- Le **sottolineature in rosso** presentano degli errori (Nel nostro caso, ‘Corso.Java’ prima rappresentava soltanto un esempio di un Package, e il programma non potrà quindi richiamarlo. Le **sottolineature in giallo**, d’altra parte, rappresentano soltanto degli errori logici, che il programma rileva (nel nostro caso, sia il metodo e la classe non vengono propriamente utilizzati, presentando infatti il messaggio ‘no usages’)
- La parola **class** è la keyword necessaria per rappresentare le classi (se messo poi prima la keyword **public** il nome del file sarà quello della classe), mentre per le keyword nei metodi le spiegheremo meglio in futuro. Sia le **classi** che i **metodi**, comunque, hanno come identificatore un nome, che non deve essere obbligatoriamente uguale. Per le classi sarà poi preferibile rappresentarlo con le lettere iniziali capitalizzate e senza l’uso dello ‘snake case’ (“_”).
- Con **System.out.print(“Hello Word!”)** andiamo invece a stampare sul terminale la stringa: >>> Hello Word ! (‘Ciao Word’, non mondo. Infatti si intendeva il mio amico John Word).
- I punto e virgola presenti saranno invece necessari da mettere su ogni singola riga, mentre le aperte e chiuse graffe i metodi verranno messe per le classi, le condizione, i cicli….

Dopo questa fase introduttiva, possiamo finalmente entrare nel vivo della programmazione, andando a parlare come primo argomento delle: variabili

Cosa sono le variabili?

Le variabili, in Java, sono considerate come dei contenitori, nel quali al proprio interno possiamo inserire diversi tipi di valori; per esempio possiamo inserirci un numero intero (3), un numero reale (4, 56756), un valore booleano (true e false) ecc…
Prima di tutto, avranno bisogno di un nome come loro identificatore, che permetterà al programma di creare una ‘directory’, cioè un percorso, per ritrovare quella parte di memoria immagazzinata. Questo identificatore potrà essere il più variegato possibile (diversamente dall’identificatore delle classi). Infatti potrà essere tutto maiuscolo, tutto minuscolo, iniziare con caratteri come ‘$’ o ‘_’ (ma non con altri caratteri speciali) e chiamarsi con nomi del tipo: ‘palla’, ‘zaino’, ‘calzino’… (anche se è preferibile darli un nome che permette di ricollegarla ad una sua funzione nel programma) . Prima dell’identificatore, però, viene descritto il ‘tipo’ della variabile; che gli dirà se dovrà contenere un numero intero, uno reale, un valore booleano… Prima di continuare; ecco uno schema generico sulle variabili:

[public|protected|private] [static] [final] Tipo identificatore [= value];

Mentre le Keyword (le parole descritte nelle parentesi quadre) le vedremo meglio col proseguire della spiegazione; la prossima parte di cui andremo a parlare sarà invece il campo di azioni delle variabili. Le variabili in Java si suddividono infatti in tre diverse categorie: Variabili locali, di istanza e statiche.

Variabili Locali

il ‘campo di azione’ della variabili locali è la parte di indentazione dove queste ultime si trovano. Per capire cosa intendiamo, ecco un esempio. Se dovessi avere la mia variabile scritta dentro un metodo, in questa maniera:

```
public class Test{
        public static void main(String[] args){
                int variabile_locale = 1;
                …
        }
}
```

vorrà dire che la mia variabile sarà accessibile (e quindi modificabile) in qualsiasi momento dentro il 'public static void main(String[] args)' e non al di fuori di quest'ultimo. Il loro 'scope' (campo) di azione sarà quindi, molto semplicemente, le parentesi graffe in cui saranno contenute. Molto simili a queste sono anche le…

Variabili di istanza

L'unica differenza che hanno rispetto alle variabili locali, e che queste ultime sono contenute all'interno di una classe. Vuol dire che potranno essere richiamate con la classe all'interno dei singoli metodi presenti. Ecco un esempio per capire meglio:

```
public class Test {
        int var = 6;
        public void primoMetodo(int var) {
                int i = var;
                // in questo caso ha precedenza il parametro e
                // i assume il valore che sarà passato come parametro al metodo
                // ...
        }
        public void secondoMetodo() {
                int var = 7;
                int i = var;
                // qui ha precedenza la variabile locale al metodo, quindi
                // i ha il valore 7
                // ...
        }
        public void terzoMetodo() {
                int i = var;
                // qui semplicemente assegnamo ad i il valore della
                // variabile di istanza e i prende il valore 6
                // ...
        }
}
```

(I commenti in Java vengono rappresentati dal simbolo '//' mentre i paragrafi di commenti dal simbolo '/* */')

Il loro 'scope' quindi, sarà l'intera classe. Nel primo metodo possiamo vedere che viene richiamata come parametro, e viene assegnato il suo valore ad un'altra variabile (' i '), che sarà quindi uguale a 6. nel secondo metodo, invece, non viene richiamata come parametro, e all'interno sarà infatti presente una variabile col suo stesso nome, che essendo locale, avrà la precedenza, dando cosi alla variabile ' i ' il valore 7. Nel terzo metodo, anche se non è stata messa come parametro, le variabile di istanza assegna il valore 6 alla variabile ' i '. Per finire, abbiamo…

Variabili statiche

Queste variabili vanno a creare un vero e proprio oggetto all'interno del programma. Ciò vuol dire che quando andremo a richiamarle, lo si potrà fare praticamente ovunque, essendo queste ultime oggetti (come se "esistenti"). L'unica differenza è che presenteranno una keyword. Una delle tante che abbiamo visto prima: 'STATIC'. Per capire come funzionano, ecco un esempio:

```
public class Main{

   static int variabile_statica = 0;

   public static void main(String[] args){
      variabile_statica ++;
      System.out.print(variabile_statica);
   }
}
```

Possiamo notare che, a differenza sia di quelle di istanza e locali, qui ho semplicemente dichiarato la variabile, non ho dovuto richiamarla, e fin da subito ho potuto utilizzarla, essendo che il suo valore viene modificato ovunque. Dopo avere finito questa parte sul campo di azioni delle variabili, andiamo a descrivere le parole chiavi che ci rimangono:

- **Public:** Rende la variabile all'interno del programma visibile da tutte le classi che si trovano all'interno di quel package;
- **Protected**: Rende la variabile all'interno del programma visibile solo alle classi con lo stesso nome che si trovano all'interno di quel package;
- **Private:** Rende la variabile all'interno del programma visibile sono a quella classe che si trova all'interno di quel medesimo package;

- **Static:** Come già detto, ma ribadiamo, rende la variabile un oggetto;
- **Final:** Rende la variabile una costante, o subito alla sua dichiarazione o dopo la sua assegnazione.

Dopo aver finito questa introduzione, passiamo invece ai tipi della variabili…

Tipi delle variabili

A differenza di altri linguaggi di programmazione, come Python, nel quale l'assegnazione delle variabili è più semplice, essendo che non deve essere specificato il tipo, in Java sarà invece essenziale. Prima di tutto, come si suddividono i tipi in Java? Ci sono due categorie in cui quest'ultimi vengono classificati: **Tipi primitivi** e **Tipi Wrapper.** I primi non vengono considerati come oggetti mentre i secondi si (In Java, un oggetto, è un corpo che può subire modifiche, iterazione… Che vedremo meglio successivamente). Partiamo dai tipi più semplici… I tipi primitivi:

Tipi primitivi

I tipi primitivi in Java sono i seguenti (tutti scritti con le lettere minuscole):

- **byte:** Serve per rappresentare numeri da -127 a +127. Il loro peso è di 8 bit;
- **int:** Serve per rappresentare numeri da -2^{34} a $+2^{34}$. Il loro perso è di 32 bit;
- **long;** Serve per rappresentare numeri da -2^{64} a $+2^{64}$. Il loro peso è di 64 bit;
- **float:** Derivato da floating-point (virgola mobile), rappresenta i numeri reali con una certa precisione decimale. Il loro peso è di 32 bit;
- **double:** Come i float, rappresenta i numeri reali, ma con una maggiore precisione decimale. Il loro peso è di 64 bit;
- **char:** Serve per andare a rappresentare i caratteri ascii o i singoli caratteri (es. char = 64 / char = "A").

Vengono utilizzati la maggior parte del tempo.

Tipi wrapper

Diversamente dai tipi primitivi, i tipi wrapper vengono considerati oggetti all'interno del programma, e possono di conseguenza essere utilizzati all'interno delle JAVA COLLECION, che analizzeremo meglio dopo. Non ci sono differenze abissali rispetto ai tipi primitivi, per quanto riguarda l'occupazione di memoria e i valori da quest'ultimi contenuti, le uniche differenze sono le seguenti:

- Si presenteranno con la prima lettera maiuscola;
- il tipo int diventerà Integer, e il tipo char diventerà Character.
- Si aggiungerà un nuovo tipo: String

Quando vogliamo andare a creare un tipo wrapper, possiamo quindi contenere il tipo primitivo dentro uno di quest'ultimi, però questa pratica è considerata obsoleta. Proprio per questo motivo, ciò che andremo adesso ad analizzare sono le operazzione di Boxing, Auto-Bozing ed Unboxing, necessarie per capire al meglio il funzionamento e la dichiarazione di questo tipo di variabile appena descritto.

Boxing, Unboxing e AutoBoxing

All'interno del nostro programma possiamo quindi andare a immagazzinare i nostri tipti primitivi all'interno dei tipi wrapper, ma quali sono i metodi che possiamo utilizzare? Adesso ne analizzeremo uno per uno. Partiamo con il primo:

- Boxing: Questo metodo è ormai deprecato da Java 1.4. Il funzionamento era particolarmente semplice, dopo aver creato il tipo primitivo, successivamente si sarebbe andato a creare un nuovo "involucro". Ecco un esempio di questa sintassi:

```
char c = "c";
String s = new String(c);
```

Richiamato quindi il tipo primitivo, vado poi a immagazzinarlo all'interno del tipo wrapper. Il secondo metodo, d'altra parte, e quello tuttora utilizzato in Java 1.8:

- Auto-Boxing: Questo metodo permette di creare un tipo wrapper, indipendentemente dalla presenza oppure no di un tipo primitivo già presente. Oltre ad essere più conveniente, e anche più veloce, visto che permette di velocizzare il processo di immagazzinazione dei valori, e di ridurre lo spazio di memoria occupato all'interno del programma. Ecco un esempio di questa sintassi:

```
String s = "s";
```

- Unboxing: per finire, questo metodo permette invece di creare un tipo wrapper, avendo già creato precedentemente un tipo primitivo, senza dover utilizzare tutta la operazione dai noi vista precedentemente per il Boxing. Se dovessimo schematizzare il tutto, molto semplicemente, si potrebbe visualizzare in questo modo:

```
int x = 1;
Integer y = x;
```

Andiamo adesso ad approfondire i metodi che possiamo utilizzare con le variabili in Java, andando quindi a entrare nell'argomento del:

Casting e operatori in java

Mettiamo che, quando tu stai scrivendo il tuo codice, si presenta la casistica dove devi moltiplicare due variabili di tipo diverso. Una per esempio potrebbe essere un int, e l'altra un long. Si potrebbe pensare che il programma, nativamente, permetterà di svolgere questo tipo di operazione. Il problema è, però, che noi in Java possiamo svolgere operazioni tra tipi di variabili uguali tra di loro, di conseguenza se volessimo moltiplicare questi due tipi, dovremmo utilizzare la seguente formula:

```
int res = (int) y * x;
```

Siamo quindi andati a mettere (tra parentesi tonde) il tipo con cui volevamo che una delle due variabili fosse modificato, alterato, per permettere così la moltiplicazione. Parlando proprio di operazioni, quali sono gli operatori che possono essere utilizzati in Java? Andiamo a vederli e a suddividerli:
Gli operatori matematici, in Java, ma come in molti altri linguaggi, si suddividono in tre categorie:

MATEMATICI DI CONFRONTO BOOLEANI

Ma come funzionano ognuno di questi? Analizziamoli con calma:
- Gli operatori matematici si suddividono in: (+) per l'addizione, (-) per la divisione, (*) per la moltiplicazione, (/) per la divisione con il resto e per finire (%) per il resto della divisione. Questi sono gli unici operatori matematici supportati nativamente dal programma, essendo che altri (come l'elevamento a potenza) devono essere importati attraverso l'utilizzo di librerie interne presenti nel programma.
- Gli operatori di confronto sono formati invece dalla seguente formula:

CONFRONTO + "="

La prima parte, di conseguenza sarà formata da un tipo di carattere che determinerà il tipo di confronto svolto, mentre la seconda parte sarà semplicemente il simbolo "= ". I caratteri che possono essere utilizzati sono (=) per determinare l'uguaglianza tra due valori, (!) per determinare la diversità tra due valori, (<) per determinare se il primo valore è minore o uguale del secondo, e per finire, (>) per determinare se il primo valore è maggiore o uguale del secondo. Questi operatori di confronto, per di più, possono essere utilizzati solo in maggior parte con i tipi primitivi, essendo che i tipi wrapper richiedono formule specifiche, e per di più, simbolo > e < possono essere utilizzati da soli come operatori di confronto. Se aggiungiamo all'operatore di confronto un operatore matematico prima dell'uguale, piuttosto che un operatore di confronto, possiamo svolgere un "operazione matematica-self assignment", per esempio:

VARIABILE += 1;

Qui sto aggiungendo uno alla variabile. Delle abbreviazioni di questo metodo, però, se si deve solo aggiungere un uno o rimuoverlo dalla variabile è la seguente scrittura: x++; x--. Molto simile a --x; ++x; difatti i due metodi presentano una differenza molto importante: con i primi due, somma o sottrae uno e poi valuta il valore della variabile, mentre con gli altri due metodi fa l'opposto. Adesso che abbiamo finito di parlare di operatori matematici, passiamo poi agli operatori booleani in Java:

&& || !

Inizialmente, i simboli che vediamo possono non sembrarci "umani". Cosa stanno a significare? Cosa vogliono rappresentare? Effettivamente questi operatori vanno a rappresentare i nostri AND OR NOT all'interno del nostro programma. Ma come funzionano? AND, in primis, verrà utilizzato per collegare due condizioni all'interno del programma, svolgendo così la "moltiplicazione booleana" tra i due (ritornando vero se, e soltanto se, tutti i valori concatenati hanno come valore di ritorno il true). Per quando riguarda OR, invece, svolgerà la "somma booleana" tra le due condizioni (ritornando vero, se e soltanto se, almeno una delle due condizioni risulta vera). Per finire, il NOT avrà il compit di ritornare il contrario da quello espresso nella nostra espressione Booleana.

BITWISE E MATH.LONG
Una aggiunta che potremmo dire in capitolo sono gli operatori BITWISE. Che cosa sono e come funzionano? Quest'ultimi hanno il compito di effettuare operazioni logiche su dati rappresentati da cifre binarie. Possono per di più essere utilizzati solo con i tipi interi nel programma. Andiamo adesso a vedere ciascuno:

- (-x): provoca una inversione dei bit all'interno dell'intero, se di partenza avessimo avuto 1100, in uscita avremmo 0011;
- x & y: effettua l'operazione AND tra i bit. Ciò vuol dire una "moltiplicazione" tra quest'ultimi;
- x | y: effettua l'operazione OR tra i bit. Ciò vuol dire una "somma (a 2)" tra quest'ultimi;
- x ^ y: effettua l'operazione XOR tra i bit, Cioè vuol dire una "somma a due (XOR version) tra quest'ultimi;
- x >> y: sposta i bit a destra di posizione. Se avessi 1001, diverrebbe 0100;
- x << y: sposta i bit a sinistra di posizione, se avessi 0100, diventerebbe 1000;

ovviamente per poter utilizzare quest'ultimi, bisogna avere una conoscenza approfondita delle operazioni booleane e delle lavorazioni dei bit, qualita non richieste nel linguaggio Java.

Oltre agli operatori Bitwise, però, dovremmo fare un approfondimento anche sulla libreria “java.lang.math” Come funziona quest’ultima? In primis, ci permette di ampliare le operazioni matematiche presenti all’interno del programma. Di metodi che possiamo utilizzare ce ne sono moltissimi, di conseguenza, qua sotto, andremo ad elencare quelli più importanti che possono essere utilizzati:

- Math.abs(double/float/int/long a) = Permetterà di ritornare il valore assoluto di un numero;
- Math.ceil(double a) = Permette di ritornare il valore (double) arrotondato per difetto da una divisione;
- Math.floor(double a) = Permette di ritornare il valore (double) arrotondato per eccesso da una divisione;
- Math.max(double, float, int (a, b)) = Permette di ritornare il valore massimo tra due numeri comparati;
- Math.min(double, float, int (a, b)) = Permette di ritornare il valore minimo tra due numeri comparati;
- Math.pow(double (a, b)) = Permette di ritornare l’elevamento a potenza di a con esponente b;
- Math.sqrt(double a) = Permette di ritornare la radice quadrata di un numero (double);

Finito la discusssione su questo capitolo, possiamo finalmente passare ai blocchi condizionali e switch-case:

Condizioni o switch-case?

Mettiamo che devi fare un quiz per la tua classe, dove ad una risposta sbagliata, il punteggio venga ribassato, e ad una risposta esatta, il punteggio viene alzato. Adesso, il metodo migliore sarebbe quindi introdurre una CONDIZIONE:

“SE il giocatore risponderà correttamente, allora il punteggio verrà alzato, ALTRIMENTI il punteggio verrà ribassato”

Per far ciò, introduciamo quindi il funzionamento dei “if, else if ed else”:
- if: Viene posto come prima condizione rispetto alle altre. Ciò vuol dire che, appena creato un blocco condizionale, non potremmo trascrivere per primo un else if oppure un else. Il suo scopo sarà controllare la propria condizione in maniera costante all’interno del blocco di codice delineato. Se quest’ultimo, primo degli altri if, sarà vero, allora verrà svolto, altrimenti se si presentano altri if prima di quest’ultimo veri, all’ora questo if non verrà svolto. Non è una buona pratica, comunque (eccetto condizioni specifiche) andare a trascrivere più if nello stesso blocco di condizione.

- else if : a differenza dell’if, non controlla la condizione in maniera costante, bensì controlla la propria condizione se, e solo se, la condizione di livello superiore risulta falsa. Ciò vuol dire che se anche un if risultasse vero sopra di lui, il programma non farebbe neanche il controllo della condizione di quest’ultimo, differentemente dall’if.

- else: Infine, il compito dell’else, sarà quello di venir eseguito solo e soltanto se tutte le condizioni presenti nei vari else if o nei vari if risultano false. A differenza di quest’ultimi, di else se ne può mettere solo uno a blocco di condizione.

In Java, a differenza di Python, la condizione che dovrà successivamente essere espressa all’interno del programma verrà trascritta in parentesi tonde. Ciò vuol dire che, eccetto per l’else, tutti gli altri blocchi di codice presenteranno la condizione all’interno di queste parentesi. Ricordiamo che all’interno dei blocchi condizionali dovremmo OBBLIGATORIAMENTE andare ad utilizzare valori o condizioni dette booleani, utilizzando gli operatori di confronto e concatenandoli, se necessario, con degli operatori booleani.

Mettiamo, però, che tu debba adesso andare a fare un bel programma con un menu a discesa, grazie al quale, in base alle decisioni dell’utente, avviene un qualcosa. Ecco qui che entra in campo lo switch-case, il quale non richiede espressioni booleane al suo interno, ma bensì valori numerici, andando cosi a rendere il controllo degli input dell’utente (come nel nostro esempio) molto più semplice e veloce. Ma come funziona? Differentemente dal blocco condizionale, per poter andare a lavorare sullo switch-case, dovremmo prima inizializzare quest’ultimo con un determinato valore di controllo, in questa maniera:

switch(c){

In questa maniera, il valore c sarà un valore di controllo, il quale compito sarà quello di tener traccia dei vari “casi” che potrebbero avvenire all’interno del nostro programma. Ma che cosa sono effettivamente i casi? Quando utilizziamo lo switch case, infatti, dobbiamo determinare i “valori possibile” che la variabile c dipendente potrebbe assumere. Per farlo utilizzeremo una struttura simile alla seguente:

```
switch(c){
case 1:
   System.out.print(“Hai premuto 1”);
case 2:
   System.out.print(“Hai premuto 2”);
case 3:
   System.out.print(“Hai premuto 3”);
default:
   System.out.print(“Stai uscendo dal ciclo!”);
}
```

Possiamo quindi analizzare ciò: se il valore di c fosse uno, verrà stampato “Hai premuto 1” se fosse due, verrà stampato “Hai premuto 2”, e cosi via. Il valore di default, poi, fa si che venga stampato “Stai uscendo dal ciclo” indipendentemente dal valore effettivo di c. Ma, come perché e trascritto proprio ciclo? Effettivamente, anche se a prima vista non può sembrare, lo “switch-case” è effettivamente un ciclo. Immaginiamocelo come un saltatore olimpionico: Questo ragazzo salterà fino ai 10, 11, 12 metri e cosi via, e poi si metterà a camminare fino alla fine della pista, per arrivare al suo allenatore. Nello stesso modo, lo “switch-case”, preso un valore di riferimento, scorrerà fino alla fine, stampando anche i valori che la c non ha assunto. Se per esempio c fosse 1, allora stamperebbe anche i valori del caso due e del caso tre (il default, come dice il nome, verrà stampato indipendentemente, e se ne potrà mettere solo uno all’interno del nostro “switch-case”). Per evitare questo problema, quindi, dobbiamo inserire il blocco “break”, che ci farà uscire direttamente dall’iterazione. Ma che che keyword è quest’ultima?

BREAK AND CONTINUE

Break e continue sono due parole chiave utilizzate in informatica per determinare l’andamento di un ciclo, ciò vuol dire che andranno a modificare la iterazione presente. Ma in che modo? Analizziamoli in singolo:

- Break: Quest’ultimo farà uscire il ciclo iterativo direttamente fuori, non svolgendo ulteriori iterazioni. Il suo compito, di conseguenza, è quello di “abbattere” il ciclo iterativo in esecuzione in quel momento. Non è una buona pratica utilizzare quest’ultimo, visto che non sempre può dar l’effetto voluto nel codice, e provocarne degli errori.

- Continue: Differentemente dall’altro, il suo compito sarà quello di passare alla prossima iterazione all’interno del ciclo. Se stessimo parlando di un ciclo while, oppure “switch-case”, questa operazione non avrebbe senso, ma se parlassimo per esempio di un ciclo for, vuol dire che una determinata iterazione verrebbe aumentata/spostata di uno. Se dovessi, avere quindi un for con iteratore x, e usassi un continue, vorrebbe dire che, avvenuta un certo condizione, la x “salterebbe” una iterazione, passando da 0 a 2, per esempio, piuttosto che da 0 a 1.

Stiamo parlando però di questi “cicli iterativi”, while, for… Che cosa sono? Andiamo adesso a vederli

Cicli iterativi

Quando stiamo lavorando all’interno di un programma, potremmo aver bisogno, per esempio, di andare a controllare tutti i valori presenti all’interno di una lista, ma come possiamo farlo? Ecco qui che entrano in campo i famosi cicli iterativi. Il loro compito è quello di avere o un numero determinato, o un numero indeterminato di iterazioni, che noi vogliamo facciano ad uno scopo specifico. Andiamo a partire con a prima sotto categoria: gli while

WHILE E DO-WHILE

Questi cicli iterativi funzionano attraverso una condizione booleana che permette a quest’ultimi di “rimanere in vita”. Ma come funzionano effettivamente? I cicli while hanno il compito di svolge un numero indeterminato di iterazioni, ciò vuol dire che, se per esempio, ci trovassimo in un videogioco, noi vogliamo che un ciclo while tenga aperto il menù finché l’utente non decide di chiuderlo, oppure se stessimo su una applicazione, vogliamo che l’utente possa tenere aperta la telecamera finché non decide di chiuderla, ecc… ma come possiao dichiarare i diversi cicli while? Ecco la rappresentazione:

while(condizione booleana){

}

do{

}while(condizione booleana)

A prima vista potrebbero anche sembrarci uguali. La differenza fondamentale, però, sta nel fatto che il primo while andrà a svolgere la iterazione se e solo se la condizione di partenza sarà vera, mentre nel secondo caso, anche se la condizione risultasse falsa, almeno una iterazione verrebbe svolta.

FOR E FOR-EACH

Differentemente dai cicli appena visti, il loro compito sarà quello di svolgere un numero specifico e determinato di iterazioni. Ma come possiamo dichiarare quest'ultimi, partiamo con il FOR:

for(variabile; condizione booleana; incremento){

}

Capiamo quindi che, all'interno del for avremmo bisogno come iteratore di una variabile (o dichiarata sul momento, o già esistente), una condizione che terrà traccia del ciclo (quanto farlo smettere) e per finire dell'incremento della variabile. Praticamente identico è il for-each, la cui unica differenza è che l'iteratore sarà determinato come valore interno presente in una lista (Array), e l'incremento non sarà quindi necessario. Ecco un esempio:

for(variabile type(variabile):Array){

}

Adesso che abbiamo finito di approfondire con i cicli iterativi, parliamo finalmente dei metodi.

Metodi, procedure e funzioni

Dobbiamo fare una distinzione di quest'ultimi, però:

"con metodi, intendiamo l'insieme di procedure e e funzioni. I primi non ritorneranno nessun valore, mentre i secondi ritorneranno un valore che successivamente dovrà essere immagazzinato"

La domanda, iniziato questo capitolo, sorge spontanea: come possiamo creare una metodo in java? Ecco la sintassi di scrittura di quest'ultimi:

[public|private|protected] static final tipo identificatore(tipo paramtro 1, tipo parametro 2, …){

Anche se inizialmente può sembrare "spaventoso", comunque la loro sintassi è, in realtà, abbastanza semplice da comprendere:

- Con public, private, e protected andremo ad identificare lo stato di visualizzazione dei vari metodi all'interno del programma. Se non si ricorda come quest'ultimi funzionano, basta andare a guardare la legenda descritta all'interno delle variabili;
- con static faremo si che il metodi sia, difatti "statico", cioè esistente. Quindi quest'ultimo potra essere riconosciuto anche fuori da altre classi, essendo considerato un oggetto;
- con final, faremo si che il valore di return del metodo sarà una costante, e che di conseguenza quest'ultimo non potrà essere modificato;

- con il tipo, andiamo a rappresentare invece che tipo di return vogliamo dare: se quest'ultimo dovesse essere una funzione, all'ora vogliamo si che il tipo di return sia un interno (int), una stringa (String), e cosi via. Se volessimo, però, che non ci fosse un return, basterà scrivere void (rendendo cosi il metodo una procedura);
- L'identificatore non ha influenza sul metodo (eccetto per il metodo di "partenza" che tra poco andremo ad analizzare);
- Per finire, nei parametri andiamo ad inserire tutte le variabili su cui dobbiamo andare a lavorare. Ovviamente dovremmo anche andare a specificare il tipo di quest'ultimo.

Ecco un metodo che potrebbe di conseguenza capitarci:

```
public static final int Test(int Parametro1, int Parametro2){
        int Somma = Parametro1+Parametro2;
        return Somma;
}
```

Importantissimo, sarà pero far si che la variabile che conterrà il valore di return abbia lo stesso tipo del metodo. Ma se volessimo andare a mettere delle condizioni quanto apriamo il nostro metodo? Introduciamo quindi ila gestione degli errori

Gestione degl errori: Trow & Try-and-catch

L'obbiettivo del trow, differentemente dal "try and catch" che analizzeremo meglio in futuro, è quello di andare a creare degli errori "personali" che possono successivamente essere visualizzati all'interno del programma. Adesso, cosa servono i trow e trows? Mettiamo che all'interno di un metodo dobbiamo andare a determinare se un ragazzo ha meno di diciotto anni o più di 60. Per far ciò, possiamo quindi creare una classe detta Trows, in questa maniera:

```
trows IllegalArgumentException{
        if (age < 18){
                trow new  IllegalArgumentException("Troppo Giovane");
        }
        if (age > 60){
                trow new  IllegalArgumentException("Troppo vecchio");
        }
}
```

In questo modo, di conseguenza, stiamo andando a creare degli errori personali che il metodo potrebbe rilevare (che se rilevati, ci faranno uscire direttamente da quest'ultimo). È importante capire una cosa: all'interno del trow dobbiamo inserire un tipo di eccezione ("exception") esistente, che in questo caso è illegalArgumentException. Lo scopo di ciò è "avvertire" il programma che quel errore è presente. Possiamo anche mettere altri tipi di errore, ma solo quelli esistenti, per "avvertire" così il codice. Anche se abbiamo visto come funziona il trow, come funziona il "Try and catch"?

Molto semplicemente, il suo compito sarà quello di testare una parte di codice, e successivamente, se rilevato un errore, non uscire direttamente dal programma, ma bensì mostrare una seconda parte di codice da noi descritta (ovviamente la prima parte di codice presente non verrà "continuata"/completata. Il suo richiamo è particolarmente semplice:

```
try{

}catch(exception e){

}
```

grazie a questo metodo, di conseguenza, riduciamo i rischi di bug e crush all'interno del nostro programma (per di più, andando a scrivere exception e, piuttosto che utilizzare exception + nome di un errore, andiamo a "salvaguardarci" da tutti gli errori che si potrebbero presentare durante la RunTime. Dopo aver detto questo, continuiamo con i metodi parlando di

Overload di metodi e differenza tra metodi statici e di istanza

potrebbe capitare, durante la scrittura di un codice in Java, di dover possedere più metodi con lo stesso nome all'interno della stessa classe. Potremmo quindi pensare di dover modificare ogni singolo identificatore presente nei vari metodi, altrimenti il codice non funzionerà, vero? Per fortuna, in Java, esiste una tecnica detta Overload dei metodi. Questa tecnica consiste nel andare a richiamare i metodi all'interno del programma (modificando semplicemente il numero di parametri che posseggono) con lo stesso identificatore, non provocando nessun errore. Questa cosa sarà poi possibile, andando a richiamare il metodo nel "metodo principale" con il numero di parametri esatti. Ma se volessimo avere un metodo con "parametri infiniti" ? Ecco qui che entra in campo la tecnica detta vargars. Richiamando, all'interno di un metodo un parametro con due parentesi quadre (aperte e chiuse), avremmo una "serie infinità di parametri" che potremmo introdurre. Tutto grazie al fatto che quelle parentesi quadre aperte e chiuse stanno ad identificare un Array (il quale tra poco spiegheremo il significato), che potrà contenere un numero N di elementi. Ecco un esempio di un metodo che implementa la tecnica vargars:

public static void main(String [] args){

(Una particolarità di questo metodo è che quest'ultimo è per di più il metodo principale della classe Main, essenziale per iniziare il programma). Tralasciando i particolari, comunque, noi stiamo quindi richiamando un numero "infinito" di elementi-parametri che possano essere introdotti nel metodo. La parola finale args è solo un identificatore, ma che sta a significare ciò che abbiamo detto prima parlando di parametri infiniti: la parola Vargars. Ovviamente i parametri che verranno introdotti dovranno essere collegati al tipo dell'Array (non potrò quindi inserire dei valori interi se, all'interno del metodo, ho specificato String. Dopo aver quindi capito il significato del vargars, passiamo ai metodi statici contro i metodi di istanza.

Le differenze non sono molte, ma una tra tutte è la più significativa: Come abbiamo specificato all'inizio di questo quaderno, la keyword static stava ad identificare la creazione di un oggetto. Ciò vuol dire che, se dovessimo andare a utilizzare un metodo al di fuori di una classe, quest'ultima dovrebbe esistere. La differenza fondamentale (e più importante) è quindi la seguente: i metodi statici permettono di essere richiamati anche fuori dalla classe di appartenenza. Per di più, la differenza sarà (se richiamati fuori dalla medesima classe), che verranno identificati dalla seguente sintassi:

NomeClasse.IdentificatoreMetodo;

Questa tecnica ovviamente non verrà applicata per i metodi di istanza, i quali in primis non possono essere richiamati fuori dalla classe di appartenenza, ma che non richiedono ovviamente il nome della classe durante la loro "invocazione". Le regole di "supremazia" tra istanza e statico si applicano anche in questo caso (se non se lo si ricorda, guardare la lezione sulle variabili). Finito quindi di parlare di metodi, passiamo poi agli Array ed ArrayList

Array & ArrayList

Finalmente ci troviamo a lavorare con dei "contenitori" di variabili veri e propri, e non più soltanto con variabili singole. Ma come funzionano quest'ultimi? Per prima cosa, bisogna dire che gli Array sono nativi su Java, a differenza degli ArrayList, e proprio per questo motivo andremo ad analizzare prima loro. Abbiamo prima accennato gli Array in parte, come se potessero contenere un numero infinito di variabili al loro interno, ma questo è vero? Iniziamo quindi a descrivere l'Array:

"*L'Array è un insieme FINITO di oggetti mutabili omogenei. Si avvicinano maggiormente alle tuple di Python, con un unica differenza: la loro lunghezza è determinata al momento della dichiarazione, e non sarà possibile svolgere un ulteriore modifica verso quest'ultima"*

Una grandissima limitazione tecnica che abbiamo, quindi, è il fatto che gli Array posseggono lunghezza già determinata, e per di più, non possono contenere valori che posseggono un type diverso rispetto al type dell'Array stesso. Adesso, come possiamo però dichiarare un Array? È abbastanza semplice. Ecco la sua sintassi:

tipo [] identificatore = new tipo[length];

I requisiti dell'Array saranno quindi: Possedere un tipo predefinito (al quale i valori al suo interno collegati si dovranno adattare), un identificatore (per poter richiamare quest'ultimo) e per finire una lunghezza. Adesso. Per andare ad accedere ai singoli elementi di un Array si possono utilizzare gli stessi metodi per le liste in Python (come per esempio

il processo di indexing), ma se sappiamo già gli elementi che dovranno essere contenuti all'interno dell'Array, non saremmo obbligati andare a trascrivere Array[0] = …, ma bensì, possiamo utilizzare questa sintassi semplificata

tipo [] identificatore = {elem1, elem2 elem3, …};

In questo modo sto già assegnando i valori che successivamente vorrò andare ad utilizzare. Adesso, se volessi creare però delle matrici con gli Array (detti Array multidimensionali) dovrò andare a dichiararli in questa maniera:

tipo [][] identificatore = new tipo[length1][length2];

Per poter accedere poi agli altri elementi, dovrò specificare quindi due coordinate piuttosto che una. Il metodo visto precedentemente per assegnare direttamente i valori può ancora essere svolto. Per finire, di Array Multidimensionali non esistono solo in due dimensioni, ma anche a tre, a quattro, a cinque, e cosi via. L'unico svantaggio, è che gli elementi presenti al loro interno non possono essere cancellati, e di conseguenza dovranno essere sovrascritti. Analizziamo adesso una serie di metodi che possono essere utilizzati sugli Array:

- Array.length = Permette di determinare gli elementi contenuti all'interno di un Array;
- Arrays.sort(Array) = Permette di ordinare un Array in ordine crescente (obbligo di importare java.utils.Arrays);
- Array.indexOf(Element) = Restituisce la posizione di un elemento presente in un Array;

Come posiamo però creare metodi che introducano un Array? O che lo prendano come parametro, e cosi via? Ecco qui la sintassi utilizzata. Per prima cosa, vediamo un metodo che come parametro di return ha un Array:

public static int [] MetodoArray(){

Questo metodo, di conseguenza, deve ritornare un Array di interi, che successivamente potremmo andare ad utilizzare. Per quanto riguarda invece un Array come parametro, sarà rappresentato in questa maniera:

public static int [] MetodoArray(int [] elementi){

Vediamo quindi che, come nel primo caso, la unica cosa che andrà a cambiare sarà la sua dichiarazione nel metodo usando le parentesi quadre, come avevamo anche visto precedentemente parlando di overload di metodi. Adesso che abbiamo analizzato concretamente gli Array, vedendo alcune peculiarità di quest'ultimi, possiamo passare alla sua "controparte", gli ArrayList. Diamo una prima definizione:

"A differenza degli Array, gli ArrayList è un insieme DINAMICO di oggetti omogenei. Ciò vuol dire che non avrà una lunghezza fissa, e che potrà essere modificata nel corso del programma"

Adesso, come possiamo andare a creare un ArrayList? Per prima cosa, l'ArrayList doovrà essere importato attraverso l'utilizzo di una libreria: java.util.ArrayList. Successivamente, dovremmo andare a creare un nuovo oggetto all'interno del singolo metodo, in questa maniera:

ArrayList<tipo wrapper> identificatore = new ArrayList<tipo wrapper>();

Utilizzando questa linea di codice, di conseguenza, abbiamo finalmente creato il nostro ArrayList. Ma come possiamo andare ad introdurre degli elementi, a modificarli, e cosi via? Andiamo a vedere parlando dei metodi utilizzabili sugli ArrayList:

- ArrayList.add(index, el) = Permette di aggiungere un elemento all'ArrayList ad una posizione arbitraria;
- ArrayList.add(el) = Permette di aggiungere un elemento all'ArrayList in ultima posizione;
- ArrayList.addAll(Collection C) = Permette di aggiungere tutti gli elementi di un altra lista nell'ArrayList;
- ArrayList.addAll(index, Collection C) = Permette di aggiungere tutti gli elementi di un altra lista con indice arbitrario;
- ArrayList.clear() = Rimuove tutti gli elementi di un ArrayList;
- ArrayList.get(index) = Permette di ritornare l'elemento in una certa posizione nell'ArrayList;
- ArrayList.remove(index) = Permette di rimuovere un elemento nell'ArrayList ad una certa posizione x;
- ArrayList.size() = Ritorna la lunghezza dell'ArrayList.

Differentemente dall'Array, però, la sua dichiarazione nei metodi non può essere di cosi facile intuizione. Partiamo col vedere come possiamo richiamare quest'ultimo come parametro di return di un metodo:

public static ArrayList<Integer> MetodoArrayList(){

Dobbiamo quindi andare a specificare, all'interno degli accenti circonflessi, il tipo di return dell'ArrayList, altrimenti (come già sappiamo) il metodo non potrà funzionare mancato il valore di return. Per quanto riguarda, invece, l'introduzione di quest'ultimi come parametri del metodo, andremo semplicemente (come abbiamo fatto per gli array) a richiamare l'ArrayList più il nome. Ecco un esmpio:

public static ArrayList<Integer> MetodoArrayList(ArrayList<String> P){

In questa maniera potremmo introdurre un ArrayList con facilità nel nostro metodo. Passiamo adesso alle Stringhe

Le stringhe

Potrebbe sembrare strano, ma anche le stringhe richiedono un paragrafo a parte di approfondimento. Difatti le stringhe, come sappiamo, sono un oggetto wrapper in Java, e di conseguenza non primitivo. All'inizio era comparso quest'ultimo, dal nulla, nei nostri nuovi type. Ma da dove arriva? Difatti la stringa possiamo immaginarla come un insieme di caratteri, quali quest'ultimi "compattatti" in un Array, per andare successivamente a formare la stringa da noi utilizzata. Ma come possiamo andare a lavorare su una Stringa? Ecco i metodi più comuni utilizzati:

- String.length() = Restituisce la lunghezza della Stringa;
- String.valueOf(el) = Converte il valore di un elemento in Stringa;
- String.charAt(index) = Restituisce l'elemento di una stringa alla posizione x;
- String1.concat(String2) = String1 + String2 = Concateno due stringhe tra di loro;
- String.substring(indexStart, indexFinal) = Restituisce una sotto stringa con posizione iniziale indexStart e posizione finale IndexFinal;
- String.replace(oldPart, newPart) = Sostituisce una parte di stringa con una nuova;
- String [] split(String) = Permette di suddividere la stringa in un Array Di caratteri;
- String.trim() = Rimuove gli spazi bianchi a sinistra e a destra di una Stringa.

Adesso. Questi sono i metodi più importanti che possiamo utilizzare con una Stringa. È importante ricordarseli, dato che saranno una parte essenziale nel nostro programma per poterci lavorare. Passiamo adesso alle classi Enum:

Classi Enum

Mettiamo che all'interno del nostro programma volessimo andare a creare un calendario. Per far ciò, potremmo pensare di utilizzare un Array, oppure un ArrayList, per andare a numerare i vari giorni presenti nella settimana, facendo così che, per esempio, il lunedì sia uno, il martedì sia due, ecc… Facendo ciò, però, perdiamo uno dei vantaggi più importanti posseduti in Java: il type checking. Molto secplicemente, il type checking avrà il compito di determinare se un valore sia effettivamente contenuto oppure no all'interno di una "lista". La classe enum, di conseguenza, possiamo immaginarla come una grande lista di valori, il cui compito principale è tenere traccia delle sua variabili costanti presenti al suo interno. Adesso, come possiamo generare una classe Enum? La sua sintassi è la seguente:

[public|private|protected] enum identificatore{

Per andare poi a trascrivere gli elementi presenti, basterà andare a scrivere il valore, seguito poi da una virgola. Di questi valori, non saremo obbligati a specificare il tipo, ma soltanto come quest'ultimi devono essere rappresentati (per esempio, stringhe, interi, booleani, ecc…). Per andare ad accedere agli elementi presenti all'interno della nostra classe enum, basterà utilizzare la seguente tecnica:

IdentificatoreEnum Variabile = IdentificatoreEnum ValoreEnum;

Con questa sintassi, a prima vista complessa, potremmo pensare sia impossibile andare ad accedere ad un elemento, ma ciò che stiamo scrivendo, semplicemente, sarà: con identificatore enum rappresentiamo il nome della classe enum da noi decisa di utilizzare, la variabile sarà soltanto l'identificatore della variabile stessa che conterrà il valore (come srmpre), e con valore enum intendiamo i valori contenuti all'interno della classe. Una peculiarità della classe enum, è il vantaggio di poter collegare più valori ad un singolo elemento, semplicemente rappresentandoli in parentesi tonde con il corretto identificatore. Passiamo adesso alle librerie in Java.

Java.util.Scanner

Richiamando questo metodo, avremo la possibilità di andare a leggere i valori in input presenti all'interno del nostro programma. Per prima cosa, dovremmo andare a creare un oggetto che ci permetterà di generare il nuovo scanner di valori. La sua sintassi è la seguente:

Scanner identificatore = new Scanner(System.in);

In questo modo, di conseguenza, avremo richiamato lo scanner all'interno del metodo o della classe, che successivamente potremmo andare ad utilizzare, una particolarità di quest'ultimo, è che può essere accompagnato, come quasi tutti i richiami nelle libreria, con le solite keywords static, public protected, ecc… per far si che acquisisca le proprietà date da queste ltime (se non se lo si ricorda, guardare la lezione sulle variabili). Adesso, per poter poi leggere gli elementi in input, se riguarda in qualsiasi tipo primitivo, basterà usare la sintassi:

identificatore.nextTipo();

Sono andato a specificare la prima lettera maiuscola, visto che la prima lettera del tipo dovrà obbligatoriamente essere maiuscola. Se parlassimo dei tipi wrapper, l'unica differenza sarebbe il nome utilizzato, eccetto per le stringhe, che richiedono obbligatoriamente l'utilizzo di "nextLine()".