Il Java

di Roberta Molinari

La storia

- Progettato da James Gosling della *Sun Microsystem* nel 1995 come linguaggio ad oggetti che tenesse conto dello sviluppo delle reti e di Internet quindi i suoi programmi sono portabili su diverse piattaforme senza operazioni di porting.
- Nel 2010 la Sun è stata acquistata dalla *Oracle*

Piattaforma: HW di base + SW di base.

Portabilità: capacità di un programma di essere eseguito su piattaforme diverse senza operazione di porting (modifica e ricompilazione)

Caratteristiche del linguaggio

- Linguaggio **ad oggetti**, **general purpose**
- È un linguaggio **semi-interpretato:** Il sorgente (*.java) viene compilato in un linguaggio intermedio detto **bytecode** eseguibile su una **Java Virtual Machine** (**JVM**). Il file ottenuto (*.class) nella fase di esecuzione, è **interpretato** dal modulo di run-time **Java Runtime Environment** (**JRE**)
- **Portabile** (il file compilato è eseguibile immediatamente sulle varie JVM senza modifiche del sorgente) *"write once, run everywhere"*

Caratteristiche del

Package: insieme di librerie di classe correlate, già compilate. Sono organizzati in modo gerarchico, la root è formata dal package *java*. Il package è come una cartella e le classi come i file.

Sono i **namespace** di .net

- L'ambiente di programmazione contiene un insieme di librerie i **package** contenenti classi o metodi di varia utilità, il loro insieme forma le **API** (Application Programming Interface)
- Strumenti e librerie per il **networking**
- **Gestione automatica della memoria** da parte del modulo *garbage collector* (non esistono i tipi puntatori, si allocano e disallocano gli oggetti in modo automatico)
- **C-like** (case sensitive, ogni istruzione termina con ;)
- **Multithreading nativo** e indipendente dal SO di compilazione e esecuzione, permette di sfruttare i moderni multicore

I programmi in Java

Con Java si possono scrivere:

- **Applicazioni:** programmi veri e propri che possono essere eseguiti autonomamente
- **Applet**: programmi lato client che aumentano le funzioni del browser
- **Servlet**: programmi lato server che aumentano le funzioni del server web

È possibile scaricare la JRE all'indirizzo: http://www.java.com/it/

Per eseguire un programma Java

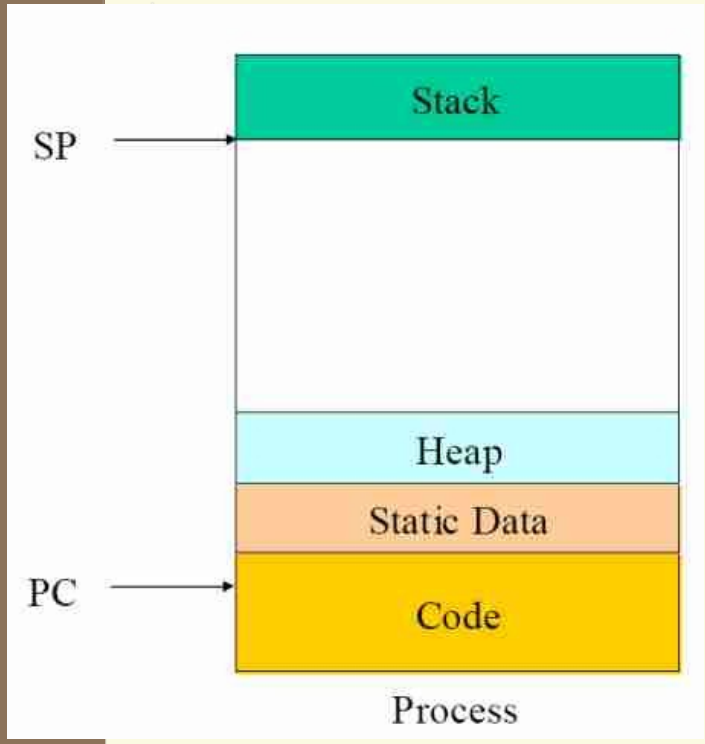
È necessario avere il file *.class* in bytecode e l'**interprete JRE** (***Java Runtime Environment***) che è un'implementazione della Macchina virtuale Java, specifica per un S.O. ed un'architettura hardware. Questo include oltre la JVM, delle librerie di base e altri componenti aggiuntivi per eseguire le applicazioni e le applet scritte in Java. Contiene il modulo **garbage collector** per gestire la memoria in quanto in Java non esistono i puntatori e l'allocazione e disallocazione avviene in automatico.

Utilizzo della memoria in C

A ogni programma in esecuzione, il SO assegna un'area di memoria riservata suddivisa in zone:

- Il **codice** e le **variabili globali** utilizzano un'area di dimensione costante, definita in fase di compilazione
- Le **variabili locali** (dette automatiche) vengono allocate nella **Stack** e liberano questo spazio all'uscita dalla funzione a cui appartengono. Lo Stack è uno spazio variabile che cresce dall'alto verso il basso (LIFO) in base alle chiamate di funzioni nel programma: contiene le varie chiamate ai metodi e le variabili e i parametri di questi ultimi, ogni chiamata ad un metodo si sovrappone all'altra creando così una Pila.

Utilizzo della memoria in C



- le variabili allocate **dinamicamente** (malloc) occuperanno l'area di **Heap.** Ne libereranno lo spazio occupato in seguito alla funzione free().
- La Heap e cresce verso la Stack, che a sua volta cresce verso la Heap, quindi lo spazio libero si può esaurire.
- La Stack è gestita automaticamente, mentre la Heap è gestita dal programmatore

Per eseguire un programma Java

La JVM è logicamente formata da:

- Un insieme di istruzioni
- Un gruppo di registri
- Un'area di memoria per l'esecuzione dei metodi (stack)
- Un'area di allocazione degli oggetti (heap) su cui opera il garbage collector per liberare lo spazio occupato da oggetti non più utilizzati (riferiti)
- Un'area di memorizzazione dei metodi (condivisa da tutti gli oggetti della stessa classe)

Per creare un programma Java

1. È necessario installare un ambiente di sviluppo come quello a linea di comando distribuito dalla Oracle il **JDK** (***Java Development Kit***). Contiene JRE e l'insieme delle librerie di classi Java (le cosidette *API-Application Programming Interface*), il compilatore Java, Web Start e i file aggiuntivi necessari per scrivere le applet e le applicazioni Java.
2. Un editor di testo o un IDE

***IDE* Integrated Development Environment:** ovvero ambiente di sviluppo integrato che generalmente contiene l'editor, il compilatore, il debbuger e altri strumenti di automazione dello sviluppo

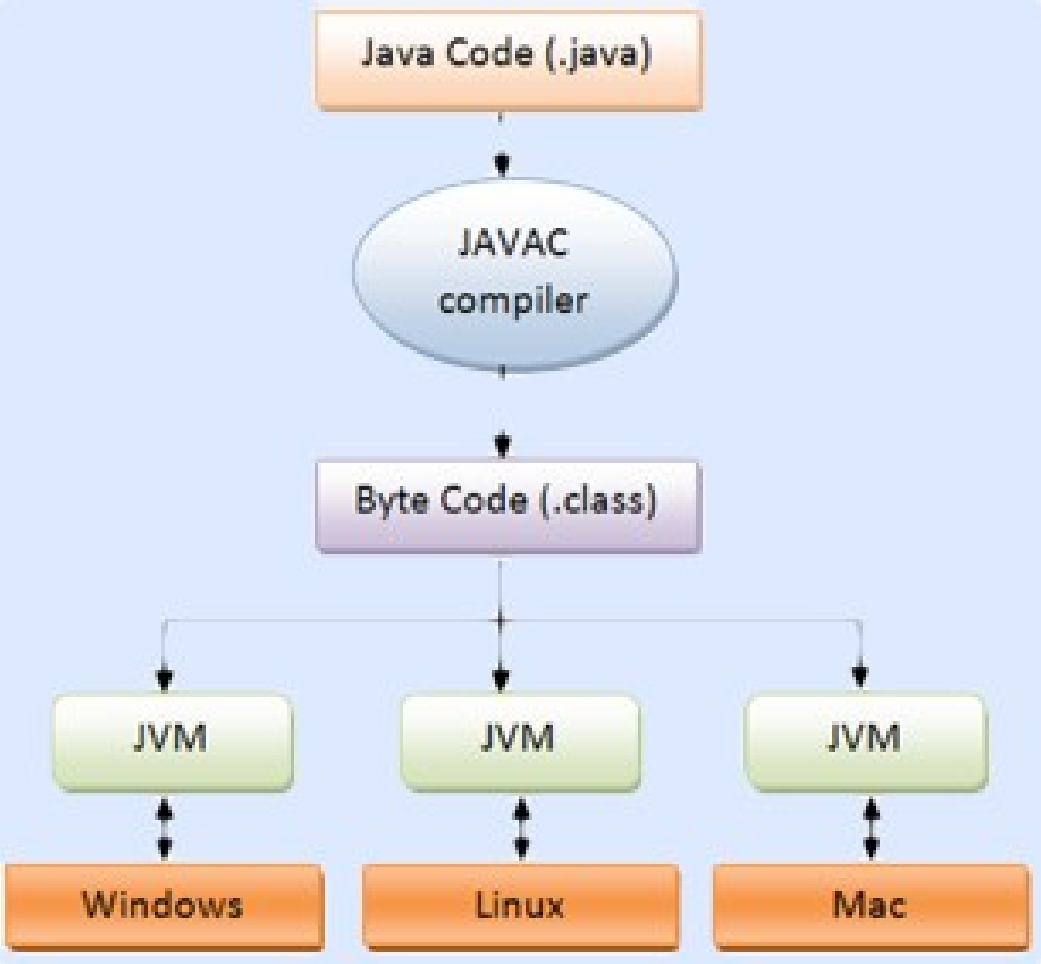
Per creare un programma Java

JVM è una macchina astratta (o virtuale). È una specificazione che fornisce un ambiente esecutivo (runtime environment) – una simulazione di un calcolatore con processore e RAM – nel quale il bytecode java gira nello pseduo-processore. Le implementazioni delle JVM fanno le seguenti operazioni: Carica codice → Verifica codice → Esegui codice

JRE È l'implementazione della JVM. Esiste fisicamente. Contiene le librerie + altri files che la VM usa durante l'esecuzione (java, javaw, rt.jar, etc).

JDK esiste fisicamente. Contiene il JRE + strumenti per lo sviluppo (javac, javaws, etc)

Per creare un programma Java



bundle: aggregazione di contenuti

Per creare un programma Java

La JDK è disponibile in diversi tipi di bundle:

- **Java SE (Standard Edition)** è orientata alle applicazioni su ambienti desktop
- **Java EE (Enterprise Edition**) offre strumenti orientati allo sviluppo di applicazioni enterprise basate sulle tecnologie Web, ha API aggiuntive per grandi quantità di dati
- **Java ME (Micro Edition)** fornisce un ambiente per le applicazioni rivolte ai sistemi embedded, per esempio televisori, dispositivi mobili o sistemi per la domotica.

IDE per Java

- **NetBeans nasce** come progetto universitario negli anni 90, acquistato da Sun, che nel 2000 lo rende un progetto open source è considerato l'IDE "ufficiale" essendo supportato da Oracle.
- **Eclipse** è il più usato in ambito Java (anche per altri linguaggi). Nasce agli inizi del 2000 da un accordo tra molte grandi società (Borland, IBM, QNX Software Systems, Red Hat, SuSE e molti altri) che fondano la *Eclipse Foundation* per promuovere questo IDE originariamente sviluppata da IBM (con un investimento in termini di tecnologia e sviluppo di 40 milioni di dollari). Integra più linguaggi, il suo plugin Android Developer Tools (ADT) ora non è più supportato da Google che ora sviluppa il progetto Android Studio basato su IntlliJ IDEA
- **IntelliJ IDEA** il più giovane tra gli IDE semplice e leggero, integra più linguaggi e Android. Idea è un prodotto commerciale di JetBrains che ha la versione **"community"** liberamente scaricabile e quella commerciale "ultimate").
- **BlueJ**: un IDE nato per scopi didattici, molto leggero e semplice, permette di creare codice dall'UML

Per la versione SE del JDK
www.oracle.com/technetwork/java/javase/downloads/

Note per installare JDK

1. Disinstallare versioni precedenti di Java JRE e riavviare (lo fa in automatico installano l'ultima versione jdk)
2. Installare il JDK adeguato al sistema in uso (32 o 64) dopo averlo scaricato dal sito facendo doppio click sul file exe (jdk-8u25-windows-x64.exe). Specificare la cartella *Programmi* (per def) *o Programmi(x86).* Si crea la cartella *JAVA*
3. Installa sia un JDK che un JRE

Per Eclipse for Java Developers
www.eclipse.org/downloads/

Note per installare Eclipse

Dopo aver installato il JDK

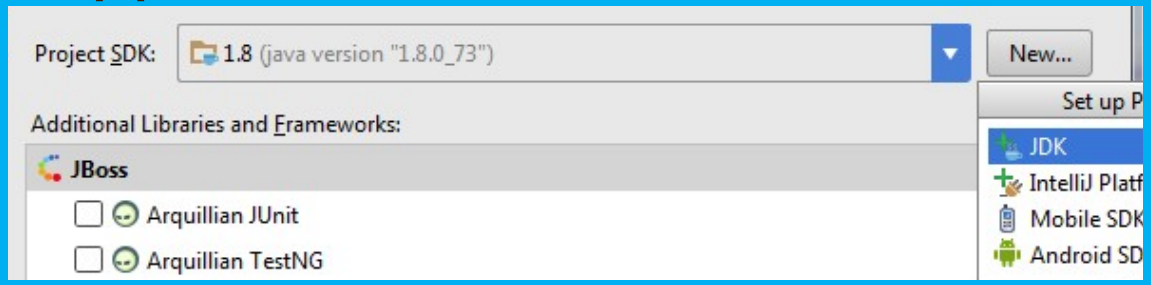
1. Scompattare il file di Eclipse (SE luna) scaricato e copiarlo manualmente in C:\
2. Se no doppio click se marte e installa tutto
3. Lanciare eclipse dalla cartella omonima creata

Si può installante JDK e Eclipse scaricandoli in una sola volta dal sito: https://ninite.com/

Per scaricare IntelliJ IDEA
https://www.jetbrains.com/idea/

Note per installare IntelliJ IDEA

- Dopo aver installato il JDK. Installare il SW IntelliJ IDEA scaricato
- Quando si crea per la prima volta un nuovo progetto chiede di selezionare la cartella con i JDK appena creata



- Si possono aprire dei progetti realizzati con altri IDE facendo *File→New→Project Existing Sources* e selezionando la cartella in cui si trova il progetto

Note per installare IntelliJ IDEA

- Per importare file già esistenti, fare un copia del file e poi incollarlo nella cartella src per aggiungerlo al progetto
- Per eseguire il programma dal menu *Run→Run* (Alt+maiusc+F10) e accettare la configurazione di default e rifacendo *Run* selezionare la classe con il metodo main che si vuole lanciare (si attiva la freccia verde in alto a dx)

Cosa fare per creare programma

Con Eclipse premere su *Run*
In InelliJ la prima volta dal menu *Run→Run* e selezionare la classe contenente il main(). Quindi è possibile usare il pulsante *Run*

1. Scrivere il sorgente contenente la classe con il metodo statico `main()` e salvarlo con nome *NomeClasse*`.java`
2. Compilarlo con il compilatore "Javac" scelto o editare il comando

   `javac` *NomeClasse*
3. Si ottiene il file *NomeClasse*`.class`
3. Richiamare l'interprete "Java" con il comando (non specificare l'estensione)

   `java` *NomeClasse* `[arg1, arg2,…]`

Cerca nella classe *NomeClasse* il metodo `main` e gli passa gli eventuali argomenti

Struttura di un programma Java

Un programma in Java è formato da un insieme di classi, per convenzione ognuna in un file *.java* con lo stesso nome della classe. Almeno la classe principale deve avere un metodo *main* da ci inizierà il programma.

Le classi sono suddivise in gruppi logicamente indipendenti detti **package**

Per garantire l'univocità dei nomi dei package generalmente si usa una gerarchia di nomi in minuscolo del tipo

```
package it.itisdelpozzo.informatica.esempio
```

Le librerie principali di Java

La riusabilità del SW viene realizzata includendo sempre nuove librerie **API** alle nuove versioni di Java. Le principali sono riunite nei seguenti package:

- **java.lang:** importato per default, contiene le classi di base
- **java.io:** per la gestione dei file e dei flussi di I/O
- **java.awt :** per la grafica
- **java.net**: per scambio di dati sulla rete
- **java.util**: per gli array dinamici, le date, la stack,…

Le librerie principali di Java
Comando import

Una classe viene caricata dalla JVM solo quando viene utilizzata (lazy load). Per trovare una classe la VM utilizza il nome del file .class associato alla classe stessa. Il path del file viene generato tramite il nome del package associato alla classe. Per fare in modo che il compilatore risolva i riferimenti ad una classe si può:

- Richiamarla con il nome esteso (*nome qualificato*)

```
java.awt.Font
```

- Importare la singola classe

```
import java.awt.Font;
```

- Importare l'intero package (ma non i sottopackage)

```
import java.awt.*;
```

Struttura minima di un programma formato da una sola classe

```
//import
[public] class Nome{
  public  static  void  main(String
  args[]) {  //parametri passati dalla riga di comando
     //variabili
     //istruzioni
  }
}
```

Public : visibile e pubblico
Static metodi di classe, vengono richiamati senza creare un oggetto della classe e si invoca riferendosi alla classe `classe.metodo`.
Possono operare solo su attributi e metodi statici

Sintassi

- È case-sensitive
- Ogni istruzione può essere scritta su più righe e termina con `;`
- Le {} individuano un blocco (classe metodo o istruzioni)
- Per identificare le stringhe si utilizza `"stringa"`, mentre per i caratteri `'a'`

Commenti

Ci sono 3 tipi:

- // commento su una riga
- /* */ commento su più righe
- /** */ commento di documentazione su più righe: viene riconosciuto dallo strumento di generazione automatica di documentazione Javadoc, che genera file HTML. Incorporato negli IDE, è anche invocabile da linea di comando con

  `javadoc` *`NomeClasse.java`*

Creare un file di documentazione JavaDoc

- Per creare in automatico la documentazione dell'applicazione, prima di ogni metodo o classe, commentare usando `/** */`
- Qualificare il testo con i tag
  - `@author`
  - `@version`
  - `@param`
  - `@return`
  - `@exception`
  - `@throw`
- Per vedere la documentazione aprire l'*index.html*
- Si può formattare il testo con comandi html

Variabili e costanti

- Gli **identificatori** sono alfanumerici, non possono iniziare con un numero, si può utilizzare _ . Per convenzione si usano nomi con l'iniziale:
  - *minuscola*: per attributi, metodi e oggetti
  - *Maiuscola*: per le classi
  - *TUTTO IN MAIUSCOLO*: costante
- Non esistono variabili globali
- Prima di essere usata una variabile deve essere inizializzata, fanno eccezione gli elementi di un array che sono inizializzati per default

Variabili e costanti

Una variabile/costante può essere dichiarata in un punto qualsiasi prima di essere usata ed avrà visibilità e "vita" solo all'interno del blocco {} in cui sono dichiarate.

```
tipo nome [= valore];
final tipo nome = valore;
```

Per dichiarare più variabili dello stesso tipo

```
tipo nome1, nome2;
```

Attenzione all'inizializzazione: il valore è applicato solo alla variabile con l'=

```
int x,y=2; //solo y vale 2
```

I tipi

Java non è un linguaggio ad oggetti puri, esistono dei tipi di dato predefiniti:

- **Primitivi**
  - Numerici (inizializzati per default a 0)
    - Interi
    - Virgola mobile
  - Carattere (inizializzati per default a \0)
  - Booleani (inizializzati per default a false)
- **Riferimento** (inizializzati per default a null)
  - Array
  - Classi tra cui *String*

OVERFLOW: quando il numero di bit necessario per rappresentare un numero è minore di quello disponibile

I tipi primitivi
Numerici interi

Tipo	Dominio non si segnala overflow	N byte
`byte`	$x \in [-128..127]$	1
`short`	$x \in [-32768..32767]$	2
`int`	$x \in [-2.147.483.648..2.147.483.647]$	4
`long`	$x \in [-2^{64}..2^{63}-1]$	8

Non esistono i tipi unsigned

In fase di run-time, Java gestisce gli insieme numerici interi in modo circolare: per un BYTE il successivo di 127 è -128

UNDERFLOW: quando il numero reale da rappresentare è < del più piccolo numero rappresentabile e quindi diventa 0

I tipi primitivi

Numerici reali

Tipo	Descrizione	N° byte
`float`	Floating point a precisione singola (7 cifre decimali)	4
`double`	Floating point a precisione doppia (16 cifre decimali)	8

Se si assegna un valore ad un float, bisogna specificare la precisione singola con **`f`** o **`F`** perché in automatico è considerato un double

```
float x=8.5; //errore "tipo incompatibile"
float x=8.5f;
```

I tipi primitivi
Operazioni sui numeri

- Operatori (in ordine di priorità):
  % (mod)
  – (cambio segno)
  * / / tra interi restituisce un intero
  + -
- Incremento e decremento
  ++, --
- Operatori composti
  +=, -=,*=,/=,%=
- Operatore ternario `cond?valVero:valfalso`
  `y=(x>0)?5:10;` //se x>0 y=5; altrimenti y=10

Il risultato di operazioni tra interi è sempre promosso a **`int`** o **`long`**

I tipi primitivi
Operazioni sui numeri

- Potenza

```
Math.pow(b,e)//b^e con b ed e double
```

- Arrotondamento

```
Math.round(reale) //restituisce un intero da un double
```

- Costanti già definite

`Math.PI` è un attributo costante = π

`Math.E` è un attributo costante = e

I tipi primitivi
Operazioni sui numeri

- Math.random (non occorre randomize)

```
Math.random()//restituisce un double [0..1)
(int)(Math.random()*numvalori)+partenza
```

- Oppure si usa un oggetto Random

```
import java.util.Random;
Random rand= new Random(); //imposta in automatico il seed della succesione pseudocasuale
int x= rand.nextInt(numValori)+partenza;
```

Esistono metodi analoghi

- nextBoolean()
- nextFloat() // Tra [0.0 e 1)
- nextDouble() // Tra [0.0 e 1)

I tipi primitivi
Carattere

Tipo	Descrizione	N° byte
`char`	Carattere UNICODE $\in$[0..65536]	2

I char assumono come valore il corrispondente codice UNICODE UTF-16, i primi 128 caratteri corrispondono a quelli ASCII.

Vengono visualizzati come caratteri, ma sono gestiti come numeri su cui è possibile fare operazioni (attenzione al cast!):

```
char c='A'; c=(char)c+32; // c='a'
```

I tipi primitivi
Carattere

Sequenze di escape utilizzate per i caratteri non rappresentabili (perché del linguaggio o speciali) sono:

- `\n /* a capo nuova riga (line feed)*/`
- `\t /* tabulazione */`
- `\0 /* fine stringa */`
- `\f /* salto pagina */`
- `\a /* beep */`
- `\r /* inizio stessa riga (carriage return)*/`
- `\" /* doppi apici */`
- `\\ /* barra \ */`
- `\' /* apice singolo */`
- `\? /* punto di domanda */`

I tipi primitivi
Booleani

Tipo	Descrizione	N bit
`boolean`	true, false	1

Possono assumere solo i valori `true` e `false`

Non si possono convertire in un dato numerico

I tipi primitivi
Casting

- Java è fortemente tipizzato. La compatibilità è fatta a livello di compilazione. Il **casting** è:
- **implicito:** avviene solo nel caso non si perdano dati (float←int, stringhe←numero), altrimenti genera un errore di compilazione

```
int a; float b;
a=b;  //segnala errore
b=a;  //ok
```

- **esplicito:** con perdita di dati (int=float)

```
int a=1,b=2;   //1/2=0 è una DIV
float c;
   c=a/b;      // c=0.0 implicito
c=(float)(a/b);   // c=0.0    c=(float)a/b;  //
c=0.5
a=(int)c;      // a=0 tronca
```

I tipi primitivi
Operatori relazionali e logici

- == /*UGUALE per i tipi primitivi. Se usato con gli oggetti significa che puntano alla stessa area di memoria
Se nelle condizioni metto = il compilatore mi segnala errore*/
- > >= < <=
- != DIVERSO
- && AND (se la prima è falsa non valuta la seconda)
- || OR (se la prima è vera non valuta la seconda)
- ! NOT
- ^ XOR
- & o | AND e OR bit a bit

I tipi riferimento
La classe String

- Per definire una stringa si deve instanziare un nuovo oggetto della classe **String** del package *java.lang*
- Non è un array di caratteri: length() è un metodo e non una proprietà. Non si può cambiare la dimensione né il contenuto se non istanziando un nuovo oggetto. Il primo carattere è in posizione 0 e l'ultimo in length()-1
- Per creare una stringa si può invocare esplicitamente il costruttore

```
String s= new String("ciao");
```

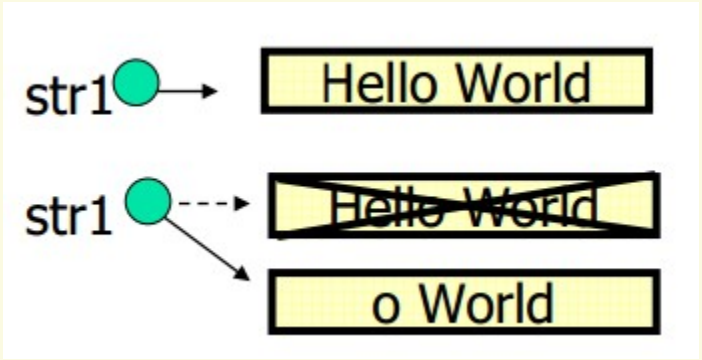
oppure implicitamente `String s="ciao";`

I tipi riferimento
String: modifica

Non si può modificare il contenuto di un oggetto stringa (Si dice che una stringa è immutabile)

Se si vuole modificare una stringa, viene creato un nuovo oggetto String, il riferimento viene aggiornato e la vecchia stringa viene eliminata dal garbage collector

```
String str1 = "Hello World"
str1 = str1.substring(4)
```



La classe StringBuffer supera questa limitazione

I tipi riferimento
String: metodi

Metodi	Caratteristiche
`char` **`charAt`**`(int pos)`	restituisce il carattere nelle posizione `pos`
`int` **`lenght`**`()`	restituisce il numero di caratteri presenti
`boolean` **`equals`**`(String s)`	Verifica se le stringhe hanno gli stessi caratteri
`String` **`substring`**`(int da[,int a])`	restituisce la sottostringa tra `da` e `a-1` inclusi
`String` **`toLowerCase`**`()` `String` **`toUpperCase`**`()`	restituisce la stringa in minusc/maiusc
`String` **`replace`**`(char oldC, char newC)`	restituisce la stringa in cui sono stati sostituiti tutte le occorenze di oldC con newC
`String` **`trim`**`()`	Restituisce una stringa senza spazi prima e dopo

- Notare che i metodi di modifica non cambiano il contenuto della stringa, ma restituiscono un nuovo oggetto String

I tipi riferimento
String: split

Il metodo `String[] split(String separatore)` permette di suddividere una stringa in sotto stringhe in base ad una stringa indicata come separatore di campi. Viene creato un vettore contenente tutte le sottostringhe

```
String str="Mario Rossi 35"
String separatore= " ";
String strVett[] = str.split(separatore);
for (int k=0,k<strVett.length; k++)
  System.out.println(str[k]);
```

Il separatore può essere una regex

I tipi riferimento
String: join

Il metodo statico

`String join(String separatore, String el,…)`

permette di ottenere una stringa data dalla concatenazione delle varie stringhe passate come parametri intercalate dal carattere separatore (non viene aggiunto al fondo).

```
String s = String.join("-","Hello","world");
// Hello-world
```

Se un elemento è null viene aggiunta la stringa "null"

I tipi riferimento
String: confronto

È possibile utilizzare == per confrontare due String, quando si assegna loro dei valori costanti o se ne fa una copia ottenendo un risultato corretto:

```
String strHello1 = "Hello";
String strHello2 = "Hello";
```

Il compilatore con valori costanti è "furbo" abbastanza per riconoscere che le due stringhe sono identiche. Quindi decide di risparmiare memoria ed utilizzare la stessa locazione di memoria. I due riferimenti strHello1 e strHello2 puntano alla stessa locazione di memoria, per cui in tal caso il confronto strHello1==strHello2 da true.

Lo stesso risultato si ottiene scrivendo:

```
String strHello2 = "Hell" + "o";
```

Oppure con

```
strHello2= strHello1;
```

Invece ne fa una copia

```
strHello2= new String(strHello1);
```

I tipi riferimento
String: confronto

L' == nel confronto fra oggetti String NON SEMPRE FUNZIONA, in particolare se un oggetto String è creato con l'uso della parola chiave `new`, o se i valori sono dati in input dall'utente, i due oggetti String non occuperanno comunque lo stesso spazio di memoria, anche se i caratteri sono gli stessi.

Pertanto conviene in generale non confrontare String con ==, utilizzarlo solo per confrontare tipi primitivi e utilizzare equals per confrontare oggetti quindi anche String.

```
String s1 = "ciao: 7";
String s2 = "ciao: " + s1.length();
System.out.println((s1 == s2));      //false
System.out.println((s1.equals(s2))); //true
```

I tipi riferimento
String: confronto

```
String str1, str2;

if (str1.compareTo(str2) < 0) {
      //str1 viene alfabeticamente prima di str2
} else if (str1.compareTo(str2)==0) {
      // str1 uguale str2
} else {
      //str1 viene alfabeticamente dopo a str2
}
```

Il confronto si basa sul valore numerico dei caratteri UNICODE

I tipi riferimento
String: confronto

Altri metodi per confrontare stringhe:

metodo	caratteristiche

`int` **`indexOf(char`** `c[,int start])`
`int` **`indexOf(String`** `s[,int start])`
`int` **`lastIndexOf(char`** `c[,int start])`
`int` **`lastIndexOf(String`** `s[,int start])`
	Cerca la prima/ultima occorrenza di c/s a partire da start se indicato. -1 se non c'è

`boolean` **`startsWith`** `(String pref [,int start])` `boolean` **`endsWith`** `(String suff)`	Verifica se la stringa inizia/finisce con pref/suff (a partire dalla posizione start se indicata in startWith)
`boolean` **`regionMatches`**`(int start, String s, int ostart, int count)`	Ricerca s a partire dalla posizione start
`boolean` **`equalsIgnoreCase`**`(String s)`	Confronta non considerando le maiuscole/ minuscole

I tipi riferimento
String: operatori e casting

- + Concatenazione
- += accodamento
- **Operatore polimorfo +:** operazioni tra numeri e stringhe generano stringhe tramite un casting automatico

```
s = "io "+10          //"io 10"
s = 3+4+"="+3+4;      //"7=34"
```

attenzione ai char `'A'+1;` dà 66 perché i caratteri sono convertiti in `int`. Funziona se si fa il casting in `char`

- **Casting →String**

```
s=String.valueOf(numero);
s=String.valueOf(carattere);
s=Character.toString(carattere);
s=""+carattere; //meno efficiente
```

- **Casting String→**

```
s = "10";
intero = Integer.parseInt(s);
reale = Float.parseFloat(s);
doppiaPrecis= Double.parseDouble(s);
```

I tipi riferimento
La classe StringBuffer

- **StringBuffer** permette di creare oggetti stringa modificabili. È contenuta nel package *java.lang.* Per istanziarne un oggetto si deve usare il costruttore esplicitamente

```
StringBuffer sB=new StringBuffer("ciao");
```

- I suoi metodi non restituiscono un nuovo oggetto come quelli di String, ma modificano l'oggetto a cui sono applicati.
- Si può convertire uno StringBuffer in String con il metodo `toString()`. Non si può fare il contrario, né fare sB="ciao"

Metodi (tutti void)	Caratteristiche
setCharAt(int pos,char c)	Assegna `c` al carattere in `pos`
setLenght(int n)	Modifica la lunghezza dello stringBuffer

insert (int pos,String s)
 	
sB=new StringBuffer("ci!");
sB.insert(2,"ao"); //"ciao!"
	Inserisce a partire dalla posizione pos la stringa `s`, mantenendo i restanti caratteri

append(String s)	Aggiunge in fondo i caratteri di `s`

I tipi riferimento
La classe GregorianCalendar

- Permette di gestire le date, nel formato "aa-mm-gg"
- La classe java.util.GregorianCalendar estende la classe astratta java.util.Calendar e sostituisce la deprecata java.util.Date.
- Il suo costruttore vuoto restituisce la data odierna
- Per ottenere informazioni sull'anno, mese, giorno e ora si utilizza il metodo get() passandogli come parametro le costanti statiche definite nella classe `Calendar: YEAR, MONTH, DATE, HOUR, MINUTE, SECOND, DAY_OF_WEEK`
- I mesi iniziano dallo 0=gennaio, può essere comodo usare le costanti pubbliche fornite dalla classe Calendar `JANUARY, FEBRUARY,ecc.`
- Esistono anche le costanti per `MONDAY,…`

I tipi riferimento
La classe GregorianCalendar

Metodi	Caratteristiche
GregorianCalendar()	Data e ora attuale
GregorianCalendar(int y,m,d)	
get(int field)	Restituisce il campo specificato dal field
getTime()	Visualizza la data come Mon Jan 12 17:32:50 CET 2015
getTimeInMillis()	Restituisce i millisecondi corrispondenti
add(int field, quanto)	d.add(GregorianCalendar.DATE,45);
Boolean **before**(date) Boolean **after**(date)	restituisce true o false a seconda se l'oggetto è < o > del parametro
int compareTo(date)	Restituisce unnumero <> o = 0 a seconda se l'oggetto è <> o = al parametro

```
GregorianCalendar oggi = new GregorianCalendar();
int anno = oggi.get(Calendar.YEAR);
```

I tipi riferimento
La classe GregorianCalendar

- Per calcolare i giorni intercorsi tra due date:

```
long milliseconds1 = data1.getTimeInMillis();
long milliseconds2 = data1.getTimeInMillis();
long diff = milliseconds2 - milliseconds1;
long diffDays = diff / (24 * 60 * 60 * 1000);
```

- Per formattare l'output si usa la classe java.text.SimpleDateFormat

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

Output

Si usano i seguenti metodi di oggetti associati allo standard output e standard error ovvero la console del video `System.out` (`out` è un attributo final static della classe `System`)

Si può usare l'operatore di concatenamento + ("ris è " + x)

```
System.out.println("stringa"); //va a capo
System.out.print ("stringa");

System.err.println("stringa"); //va a capo
System.err.print ("stringa");
```

Output formattazione

1. Usando `String.format`

```
String s = String.format("%.2f", Math.PI);
System.out.println(s);
```

2. Usando `printf`

```
System.out.printf("%.2f %3d%n",Math.PI,2);
```

3. Creando un *Formatter* e linkandolo a un *StringBuilder*. L'output formattato si aggiunge allo *StringBuilder*.

```
StringBuilder sbuf = new StringBuilder();
Formatter fmt = new Formatter(sbuf);
fmt.format("PI = %.2f%n", Math.PI);
System.out.print(sbuf.toString());
// you can continue to append data to sbuf here.
```

	Applies to	Output
%a	floating point (except BigDecimal)	Hex output of floating point number
%b	Any type	"true" if non-null, "false" if null
%c	character	Unicode character
%d	integer	Decimal Integer
%e	floating point	decimal number in scientific notation
%f	floating point	decimal number
%g	floating point	decimal number, possibly in scientific notation depending on the precision and value.
%h	any type	Hex String of value from hashCode() method.
%n	none	Platform-specific line separator.
%o	integer (incl. byte, short, int, long, bigint)	Octal number
%s	any type	String value
%t	Date/Time (incl. long, Calendar, Date and TemporalAccessor)	%t is the prefix for Date/Time conversions. More formatting flags are needed after this.
%x	integer (incl. byte, short, int, long, bigint)	Hex string.

Input Buffere

Se scrivo la dichiarazione con il completamento automatico di Ctrl+spazio, Eclipse/IntelliJ importa automaticamente il package. Se no lo sottolinea e posizionandosi con il mouse lo suggerisce.

IntelliJ lo scrive in rosso con Alt+Invio importa il package

Si usano i metodi di oggetti associati al flusso di dati da tastiera. Si deve creare un oggetto tastiera di classe BufferedReader su un flusso di input

```
InputStreamReader input= new InputStreamReader
   (System.in);
BufferedReader tastiera=new BufferedReader(input);
```

Prima si devono importare le librerie necessarie

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

oppure

```
import java.io.*
```

Quindi si usa il metodo *.readLine()* che restituisce sempre una stringa e quindi bisogna convertire

Input BufferedReader

Quando scrivi .readLine Eclipse segnala un errore, posizionandoti sopra ti chiede se vuoi gestire l'eccezione o meno ti suggerisce anche un multicatch (NumberFormatException | IOException e). Analogamente IntelliJ

```
String s; int n;
try {//operazioni da testare
   s=   tastiera.readLine();          //legge   fino
   all'invio e restituisce una stringa
   n=   Integer.valueOf(s).intValue();  //  valueOf()
   restituisce un oggetto di tipo Integer inizializzato al valore ricevuto,
   e .intValue() restituisce il suo valore. Sono metodi della classe Integer.
   Si può fare anche solo il valueOf()
   }
catch(Exception e) { //gestione eccezione generica
   System.out.println ("Errore di input");
}
```

Eccezione: situazione anomala che si verifica durante l'esecuzione di un programma

Input BufferedReader

Più brevemente

```
int n;
BufferedReader tastiera= new BufferedReader(new
   InputStreamReader(System.in));
try {
   n=Integer.parseInt(tastiera.readLine());
   //restituisce il primitivo senza creare un Integer
}
catch (Exception e) {
    System.out.println("Errore di input");
}
```

Se non inserisco un numero corretto .intValue() e .parseInt genera una eccezione *NumberFormatException*, per cui si può sostituire all'eccezione generica Exception questo tipo

Input BufferedReader

Con la gestione delle eccezioni specifiche diventa

```
int n;
BufferedReader tastiera= new BufferedReader(new
  InputStreamReader(System.in));
try {
  n=Integer.parseInt(tastiera.readLine());
}
catch (IOException e) {   //generata dalla reaLine()
   System.out.println("Errore di input");
}
catch (NumberFormatException e) {
  //generata dalla parseInt()
    System.out.println("Errore: non hai inserito
  un numero intero");
}
```

Input Scanner

Dalla versione 5 è possibile utilizzare la classe **Scanner** che consente di leggere da qualsiasi flusso di ingresso (ad es. un file), tra cui la tastiera tramite l'oggetto System.in (di tipo InputStream per la lettura di flussi di byte).Possiede diversi metodi per leggere e convertire i caratteri inseriti (si attende l'invio):

- INTERI →int **nextInt**()
- FLOAT/DOUBLE → double **nextDouble**(),float **nextFloat**() //separatore virgola
- STRINGA → String **nextLine**() //fino all'enter
- PAROLA →String **next**() // fino al primo carattere di spaziatura: spazio, fine riga, tabulazione
- BOOLEAN → boolean **nextBoolean**()

Input Scanner

```
Scanner in = new Scanner(System.in);

int number = in.nextInt();
double price = in.nextDouble();
String city = in.nextLine();
String state = in.next();
```

Se si aspetta un numero e si inserisce un formato non corretto viene sollevata un eccezione di tipo `InputMismatchException`

A differenza della gestione con `BufferedReader`, non si ha l'obbligo di gestire le eccezioni

Input Scanner

Si può gestire il corretto inserimento tramite i seguenti metodi che restituiscono true se nel flusso c'è un:

- INTERO →**hasNextInt**()
- FLOAT/DOUBLE → **hasNextDouble**(), **hasNextFloat**()
- STRINGA → **hasNextLine**() //fino all'enter
- PAROLA → **hasNext**()
- BOOLEAN → **hasNextBoolean**()

I metodi non fanno spostare il puntatore al flusso

```
Scanner sc = new Scanner(System.in);
if(sc.hasNextInt())
   int x= sc.nextInt();
```

Istruzioni alternative
if

- Si scrive

```
if (condizione){]
    //istrRamoVero;
[}][else[{]
  //istrRamoFalso;
[}]]
```

Le tonde sono obbligatorie anche se ho condizioni composte

Istruzioni alternative switch

```
switch (varIntera) {
    case val1:  //non ci devono essere {}
          //istrCaso1;
          [break;]
    case val2:
          //istrCaso2;
          [break;]
    …
    [default:
          //istrCasoElse;
          [break;]]
}
```

Si valuta varIntera e si eseguono di seguito tutte le istruzioni a partire dal case vero fino al primo break o alla }.

Se ho più valori in or

```
case val1:
case val2:
   //istruzioniPerVal1OrVal2;
   break;
```

Istruzioni iterative
do ..while e while

```
do [{]
    //isruzioni;
[}]while (cond);
```

Si eseguono le istruzioni nel nucleo almeno una volta e finchè *cond* è vera

```
while (cond) [{]
    //isruzioni;
  [}]
```

Prima si testa la cond, se è vera si esegue il nucleo. Le istruzioni nel nucleo possono non essere mai eseguite

Istruzioni iterative
for

- Si dovrebbe usare solo quando si conosce a priori quante volte si deve eseguire un ciclo

```
for ([i=iniz];[i<=fine];[i++]){
  //istruz;
}
```

- `i` può essere dichiarata nel for, così è distrutta all'uscita del ciclo (ci possono essere più inizializzazioni separate da una ,)
- Se non c'è la condizione di terminazione si considera sempre vera. Può essere qualunque condizione (una sola)
- L'istruzione di incremento può essere qualunque cosa (è eseguita per ultima prima di ritestare la condizione) (ci possono essere più incrementi separati da una ,)

Iterator: dispongono di un iteratore (java.util.Iterator). Le Collections di java (Linkedlist, ArrayList, etc.) sono tutti oggetti di tipo Iterable

Istruzioni iterati
for each

- **For each** (a partire da *Java 5*) permette di iterare velocemente su una struttura dati. Si usa su un array o su oggetti di tipo Iterable.

```
for(type element: varArray) {
      //ciclo usando element
}
```

Esempio.

In una funzione con la variabile `int[]vett` invece di

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

si può fare

```
for(int element:vett) somma+=element;
```

L' intero **element** è l'i-esimo elemento corrispondente al *for* sopra.

Istruzioni iterative for

- Equivale logicamente ad

```
i=iniz;
while (i<=fine) {
        istruz;
        i++;
}
```

Pertanto all'interno del ciclo posso utilizzare i valori `iniz`, `fine` e il valore variabile di `i`, ma non ha senso modificare nessuno di essi!!
(N.B. alla fine del ciclo `i=fine+1`)

I cicli interrotti e infiniti
break - continue

- **break**: determina l'uscita dal blocco di istruzioni in cui appare (nella switch o nei cicli)

- **continue**: può solo comparire nel corpo di un ciclo e causa l'interruzione dell'esecuzione del corpo del ciclo e passa il controllo alla condizione del ciclo (nel caso del for prima fa l'assegnamento e poi verifica la condizione).

array: insieme di elementi tutti dello stesso tipo, ciascuno individuabile tramite l'indice della posizione

I tipi riferimento
Vettori o Array

- Un array è realizzato con un puntatore all'area di memoria dove si trovano i suoi elementi
- Può essere un insieme di tipi predefiniti o di oggetti
- Il primo elemento ha indice 0
- Nelle procedure il vettore è sempre passato per riferimento `void proc(int vett[]);`
- Per usarli si deve:
  1. Dichiararlo
  2. Allocarlo
  3. Eventualmente inizializzarlo

I tipi riferimento
Vettori o Array

- Per dichiarare e allocare un vettore

```
final int N=5;
tipo nomeVet[];
nomeVet = new tipo[N];
nomeVet[0]= val_iniz;
```

oppure

```
tipo nomeVet[]= new tipo[N];
```

- In Java si tende a dichiarare i vettori con la nuova sintassi

```
tipo[] nomeVet;
```

I tipi riferimento
Vettori o Array

- Se ci si riferisce ad un indice inesistente si genera l'eccezione **ArrayIndexOutOfBoundsException**
- Ci si può riferire alla sua dimensione con l'attributo di sola lettura *.length*

I tipi riferimento
Vettori o Array

- Se non inizializzati gli elementi hanno per default il seguente valore
  - Numeri 0
  - Booleani false
  - Caratteri \0
  - Oggetti null

- Per inizializzare gli elementi al momento della dichiarazione (la dimensione viene dedotta)

```
int[] vet1={0,1,2,3};
String[] vet2={"ciao","a","tutti"};
```

I tipi riferimento
Array di array (Matrici)

- Per gestire le matrici si deve dichiarare un vettore di vettore

```
final int NR=5;
final int NC=10;
tipo[][] nomeMat;    //anche tipo nomeMat[][];
nomeMat = new tipo[NR][NC];
nomeMat[0][0]= valIniz;
```

oppure

```
tipo[][] nomeMat= new tipo[NR][NC];
```

- L'attributo **length** avrà i seguenti valori
  - `nomeMat.length`→`NR`
  - `nomeMat[0].length`→`NC`

I tipi riferimento
Array di array (Matrici)

- È possibile dichiarare e inizializzare (le dimensioni sono calcolate in automatico)

```
int[][] mat={{1,2},{3,4},{5,6}};
```

  - `mat.length`→`3`
  - `mat[0].length`→`2`

- Come parametri sono sempre per riferimento

```
void visMat(int[][] mat){
   for (int r=0;r<mat.length;r++){
      for (int c=0;c<mat[r].length;c++)
         System.out.print(mat[r][c]);
   System.out.println();
   }
}
```

Interruzione del programma

Per far terminare il programma si può utilizzare il seguente metodo statico

```
System.exit(1)
```

Il parametro 1 viene restituito al sistema operativo, qualunque valore diverso da 0 è considerato per convenzione un errore

Le eccezioni

Durante l'esecuzione di un programma si può verificare una situazione anomala. Se non si gestisce l'eccezione, il programma termina segnalando l'errore. Per evitare la terminazione anomale si usa:

```
try {
   //istruzione da controllare
  }
  catch(TipoEcc1 [|TipoEcc2] var1){
   //da fare se si verifica l'eccezione TipoEcc1 o
   TipoEcc2
  }
  [catch (TipoEcc3 var2){}]
  [finally {
   //operazioni comunque eseguite (rilascio risorse o
   chiusure file) se in un catch c'è una throw prima
   di lanciarla esegue prima la finally
  }]
```

Le eccezioni predefinite

Sono tutte sottoclassi della classe **Exception**

- **NumberFormatException**: errore di formato numerico (nell'input)
- **ArithmeticException**: segnala errori aritmetici
- **NullPointerException**: errore dovuto all'utilizzo di un riferimento che possiede il valore null (usato attributo o metodo di un oggetto non istanziato)
- **IndexOutOfBoundException**: errore nell'indice di un array
- **IOException**: generico errore di input/output.
- **FileNotFound:** errore di file inesistente
- **EOFException:** lettura oltre la fine di un file

Le eccezioni
Clausola throws

Se non si vogliono gestire le eccezioni, si deve indicare quali sono che non vengono gestite e rimandate "al chiamante" (*propagarla*) che dovrà gestirle lui o rimandarle a sua volta.

Per fare questo si deve utilizzare la clausola throws vicina al nome della classe o al metodo

```
Class miaClasse throws tipoEccezione{}
public static void main(String[] args)
  throws
  NumberFormatException, IOException {
}
```

Approfondimenti

Le enumerazioni enum

Una **enumerazione** è un elenco di costanti intere a cui, per ogni valore numerico, è stato assegnato un nome.

In Java definiscono un nuova classe/tipo (iniziale maiuscola). Se non specificato il valore, si assegna 0 alla prima costante, 1 alla seconda,…

```
visibilità enum Nome{ NOME1[(val1)],
        NOME2 [(val2)], …}[;]

Nome x=NOME1;
```

- I membri di una classe enum sono implicitamente **static**
- I valori sono implicitamente **static** e **final** (non si può più cambiarne il valore) e vanno scritti in maiuscolo

Le enumerazioni enum

Generalmente costituiscono una "classe" a sé in un file separato, ma possono essere dichiarate all'interno di una classe e in questo caso, se public, possono essere utilizzate in altre classi importando la classe che le contiene.

```
class A{
 public enum Stagione{
    PRIMAVERA, ESTATE, AUTUNNO, INVERNO};
…}

import A.Stagione;
class B{
  Stagione s=Stagione.ESTATE;
 …}
```

Le enumerazioni enum

- Sono **type-safe**, ovvero a una variabile di tipo enum non si può assegnare un valore diverso da quello in elenco. I tipi definiti in una enumerazione sono istanze di classe, non tipi interi!

```
x=1; //errore di compilazione
```

- Si possono usare negli **switch**

```
public class EnumTest {
 public enum Stagione{ //sottinteso static
  PRIMAVERA, ESTATE, AUTUNNO, INVERNO};
 public static void main(String[] args) {
     Stagione s=Stagione.ESTATE;
  switch(s){
            case PRIMAVERA:
                …
                break;
  …
```

Le enumerazioni enum

- Si possono **confrontare con l'==** perché i valori sono final

```
Stagione s1=Stagione.ESTATE;
if (s1==s) …
```

Il metodo == è sovrascritto, quindi può essere usato in maniera intercambiabile al metodo equals che resta il consigliato

- A partire dalla versione 5, in Java è stato introdotto uno speciale tipo chiamato **Enum o Enumerated Type** (vai all'approfondimento)

Gli oggetti in Java

di Roberta Molinari

Le classi
Classe con metodo main

```
//import
[public] class Nome{
   //attributi
   //costruttori
   //metodi
   public static void main(String args[]) {
   //parametri passati dalla riga di comando
      //variabili
      //istruzioni
   }
}
```

Deve esistere in ogni applicazione, il suo metodo *main* viene eseguito per primo (più classi possono avere il metodo main, il programma parte da quello della classe dichiarata principale)

Le classi

Di default, la definizione di una classe Java può essere utilizzata solo dalle classi all'interno del suo stesso package. Se **public** sarà visibile a tutte le classi dell'applicazione

```
[public] class NomeClasse{
   visibilità [final] tipo attr1[, attr2] [=valIniz1];
   visibilità {tipo|void} metodo1(parametri){
      //variabili
      //istruzioni
      [return val;] //per le "funzioni"
    }
}
```

La dichiarazione di una classe viene salvata in un file con lo stesso nome della classe `nomeClasse.java` (meglio uno per classe). Dopo la compilazione il file avrà estensione `*.class`. Se in un file `.java` non c'è una classe con lo stesso nome del file non si può nemmeno compilare.

Modificatori di visibilità

```
<modificatore>[static][final]<tipoRitornato><nomeMetodo>
([<parametri>])
        { <body> }
```

Metodi: sintassi completa

```
<modificatore>[static][final]<tipo><identificatore>[=<valore>]
[,<identificatore>[=<valore>]…]
```

Attributi: sintassi completa

<modificatore>	Descrizione
public	• accessibile ai metodi della classe in cui è usato • accessibile ai metodi di altre classi
protected	• accessibile ai metodi della classe in cui è usato • accessibile ai metodi delle sottoclassi della classe in cui è usato • accessibile ai metodi di classi dello stesso package della classe in cui è usato
	• accessibile ai metodi della classe in cui è usato • accessibile ai metodi di classi dello stesso package della classe in cui è usato
private	• accessibile ai metodi della classe in cui è usato

Classi-file

- Un'applicazione (progetto) si compone da diverse classi, ognuna si può trovare in file differenti e un file può contenere più classi, in questo caso solo quella con lo stesso nome sarà considerata public e visibile anche all'esterno, indipendentemente dalla visibilità dichiarate.
- In un'applicazione una sola classe avrà il metodo main e sarà la classe principale, avrà per default visibilità public e dovrà avere lo stesso nome del file.
- Ogni file sorgente deve essere salvato nella stessa cartella della classe principale, quindi va compilato separatamente e solo quando nessuno di questi presenta errori di compilazione si potrà invocare l'interprete sulla classe principale che inizierà l'esecuzione.

Classi-file

- Visto che il codice sorgente di una classe pubblica viene memorizzato in un file `.java` avente lo stesso nome della classe (incluse maiuscole e minuscole), può esistere solo una classe pubblica per ogni file sorgente.
- Inoltre il compilatore scrive il bytecode di ogni classe in un file `.class` avente lo stesso nome della classe (incluse maiuscole e minuscole).
- Questo per semplificare la ricerca di sorgenti e bytecode

Per esempio se abbiamo 3 classi A, B e C in un file unico, con A la sola pubblica, il codice sorgente di tutte e 3 le classi si trova nel file `A.java`

Quando `A.java` viene compilato, il compilatore crea una classe per ogni classe nel file: `A.class`, `B.class` e `C.class`

Livelli di visibilità

Visibilità modificatore di accesso	Public	Protected	Package (se non specificato)	Private
Stessa classe	Sì	Sì	Sì	Sì
Classe nello stesso package	Sì	Sì	Sì	No
Sottoclassi stesso package	Sì	Sì	Sì	No
Sottoclasse in package diverso	Sì	Sì *	No	No
Non sottoclasse in package diverso	Sì	No	No	No

* è consentito ad una istanza di una classe figlia far riferimento ai metodi e alle proprietà protected implementate nella classe padre ma non è permesso accedere ad essi attraverso una istanza della classe padre.

Gli attributi

```
{private|public|protected} [final] [static] tipo
    attr1[, attr2] [=valIniz1];
```

Per realizzare l'information hiding gli attributi dovrebbero essere *private.*

Se non specificato hanno livello di visibiltà **PACKAGE.**

Il `valIniz1` sarà il valore che ha l'attributo al momento dell'istanzazione dell'oggetto, se non gli viene assegnato un valore diverso dal costruttore

Static: attributo **statico** o **di classe**, esiste ed è visibile e utilizzabile anche se non ci sono istanze (viene allocato al primo utilizzo). Utilizzabile anche senza istanze tramite `NomeClasse.attrib`. È memorizzato in un area di memoria comune a tutti gli oggetti della classe, è condiviso da tutte le eventuali istanze

Final: attributi **costanti.** Possono essere anche pubblici

firma: nome del metodo +parametri
static: sono richiamabili senza istanze della classe tramite `NomeClasse.metod()` NON hanno accesso alle variabili di istanza e ai metodi non-static di una classe. Possono lavorare solo sui parametri o attributi statici e chiamare altri metodi statici. Usati per Utility

```
[{private|public|protected}]
[static] {tipo|void} metodo1(parametri)
   {
   //variabili locali
   [return val;]//valore restituito per le funzioni, comunque fa terminare il metodo . Anche più di uno
   }
```

- I parametri sono passati per valore (tipi primitivi e wrapper e String) per riferimento gli array e gli oggetti
- È obbligatorio solo specificare il tipo restituito e il nome.
- I metodi pubblici sono richiamabili tramite il *messaggio*

  `nomeObj.NomeM()`
- I metodi privati (visibili solo nella classe) sono richiamabili tramite `NomeM()` o `this.NomeM()`
- È possibile fare l'**overloading** dei metodi: ovvero avere 2 metodi con lo stesso nome, ma numero o tipo di parametro diversi

I metodi varargs

- Dalla versione 1.5 in Java sono disponibili i ***varargs* (variable arguments)**, un elegante metodo di avere metodi con zero o più parametri di uno specifico tipo. Vengono creati utilizzando tre punti (...) dopo il tipo di parametro. Normalmente si utilizza il foreach per recuperare i vari elementi

```
public static int somma(int... args) {
   int sum = 0;
   for (int arg : args) sum += arg;
   return sum;
} //somma(1, 2, 3) →6,  somma() →0.
```

- Se il metodo ha almeno un parametro obbligatorio deve essere messo per primo e separatamente

```
public int min(int primoArg, int... altriArgs)
```

I metodi varargs

```
public class VarArgsExample {
   int sumArrays(int[]... intArrays) {
      int sum, i, j; sum=0;
      for(i=0; i<intArrays.length; i++) {
         for(j=0; j<intArrays[i].length; j++) {
            sum += intArrays[i][j]; } }
      return(sum);
   }
   public static void main(String args[]) {
      VarArgsExample va = new VarArgsExample();
      int sum=0;
      sum = va.sumArrays(new int[]{1,2,3},
         new int[]{4,5,6}, new int[]{100,200});
    System.out.println(sum);
   }
}
```

L'inizializzatore static

È possibile fare delle operazioni iniziali agli attributi di classe usando il blocco static

```
static {
    //inizializzazione degli attributi di classe
}
```

Istanza di classe: l'oggetto

Per creare un oggetto, cioè un'istanza di una classe si deve:

1. dichiarare l'oggetto (in RAM occupa lo spazio per un puntatore, la **maniglia**)

```
NomeClasse nomeOggetto;  //=null
```

2. allocare l'oggetto (si invoca implicitamente il metodo **costruttore** della classe e in RAM si allocano tutti gli attributi e si assegna alla var oggetto il puntatore all'area. Se non c'è spazio sufficiente la JVM segnala errore)

```
nomeOggetto= new NomeClasse();
```

Si può fare tutto in un'istruzione

```
NomeClasse nomeOggetto= new NomeClasse();
```

Il costruttore dell'oggetto

- È il metodo che viene richiamato automaticamente per primo quando si fa una **new**.
- Se non viene esplicitato si crea automaticamente un costruttore di default vuoto e senza parametri
- Generalmente contiene le istruzioni per inizializzare lo **stato interno iniziale** dell'oggetto.
- Solitamente è di tipo pubblico e non può restituire dei valori perché è sottinteso che restituisce la maniglia all'oggetto istanziato
- È considerato **static** perché invocabile prima dell'istanzazione

Il costruttore dell'oggetto

- Si può fare l'overload, ma comunque dovrebbe sempre esistere uno senza parametri e uno che ha come parametro un oggetto della classe per il *clone* (per essere *Javabean*)
- Se si crea un costruttore con parametri non si può più riferirsi a quello di default senza parametri, non esiste più, bisogna eventualmente ridefinirlo (il compilatore dà un errore)

JavaBean standard per oggetti gestibili da tutti gli strumenti di sviluppo per Java, generalmente sono persistenti-serializzabili, ovvero salvati in modo permanente

Per creare il costruttore con parametri cliccare con il tasto destro sul codice, quindi in Eclipse *Source→Generate Constructor using Fields…*,in IntelliJ *Generate..Costructor* e selezionare gli attributi desiderati

Il costruttore dell'oggetto

```
public NomeClasse(parametri) {
  //variabili locali
  //istruzioni
  }
```

Agli attributi viene prima assegnato il valore indicato dall'eventuale inizializzazione e poi quello indicato nel costruttore

Una classe per essere **JavaBean** deve**:**

- avere un costruttore senza argomenti;
- le sue proprietà devono essere accessibili usando get, set, is (usato per i booleani al posto di get) e altri metodi (così detti metodi accessori) seguendo una convenzione standard per i nomi;
- la classe dovrebbe essere serializzabile (capace di salvare e ripristinare il suo stato in modo persistente su file);
- non dovrebbe contenere alcun metodo richiesto per la gestione degli eventi;

L'oggetto this

Sono implicitamente presenti in ogni classe

- **this**: fa riferimento all'oggetto medesimo. È sempre sottinteso, si usa in caso di eventuali omonimie tra attributi e nomi dei parametri.
- **this():** fa riferimento al costruttore dell'oggetto stesso (è invocato in automatico quando si crea un oggetto). Può solo essere invocato in costruttori e deve essere la loro prima istruzione

```
Public nomeClasse(par, attr)   {
 this(par);   //chiama il costruttore con un solo par
 this.attr =attr; //toglie ambiguità
}
```

this NON può essere usato in metodi stataici perché si riferisce all'istanza dell'oggetto

I metodi set e get

Se si applica *l'information hiding*, gli attributi sono privati e bisogna inizializzarli o leggerli tramite i metodi pubblici **set** e **get**.

In generale, per ogni attributo private *NomeAttr*, si definisce un metodo ***set/get****NomeAttr* per poter fare le operazioni di r/w del campo.

```
public void setNomeAttr(tipo par){
    //controlli
    this.nomeAttr=par;
}

public tipo getNomeAttr(){
    return nomeAttr;
}
```

Per creare i metodi **get** e **set** di un attributo cliccare con il tasto destro sul codice, quindi in Eclipse *Source→Generate Getters and Setters…* in IntelliJ *Generate…* *Getters and Setters* e selezionare di quali attributi creare i metodi

Wrapper

- Una classe **Wrapper** (involucro) ricopre un tipo primitivo, trasformando un tipo primitivo in un oggetto
- Ha lo stesso nome del tipo, ma con l'iniziale maiuscola

```
int→Integer
double→Double
char→Character
```

- Le istanze di oggetti wrapper assumono il valore passato al costruttore e non lo cambiano più.

```
Integer n= new Integer(12);  //n=12
```

- Mettono a disposizione molti metodi static in quanto le classi wrapper vengono viste come fornitrici di servizi.
- Hanno il metodo `valueOf(numero)` che restituisce un oggetto della classe wrapper a cui è applicato trasformando il primitivo in un oggetto (**boxing**)

```
Integer n= Integer.valueOf(123);
```

Wrapper

La classe **Integer** si può immaginare così

```
class Integer{
   private int value;
   Integer (int val){
     value=val;
   }
   Integer (String s) {
   //assegnano    a    value    il    valore
     corrispondente alla stringa
   }
   public static int intValue() {
     return value;
   }
}
```

Wrapper .tipoValue()

Hanno un metodo get invocato come `.tipoValue()` che serve per acquisire un valore incapsulato, per ottenere il primitivo dalla classe wrapper (**unboxing**).

I metodi sono:

- `int` **`intValue`**`()`
- `double` **`doubleValue`**`()`
- `boolean` **`booleanValue`**`()`
- `char` **`charValue`**`()`

Per esempio

```
int n=100;
Integer numero = new Integer(10);
n=n*numero.intValue(); //n=1000
```

Wrapper .parseTipo()

Il metodo

```
public static int parseInt(String s)
```

converte una stringa in un intero senza creare un'istanza di Integer.

Per convertire una stringa in un numero si può fare:

```
Integer in=new Integer("23");   //istanzia
int n=in.intValue();
```

oppure

```
int n=Integer.valueOf("23").intValue();
```

oppure

```
int n=Integer.parseInt("23");
```

Analogamente esistono `parseLong(s)`, `parseFloat(s)`, `parseDouble()`

Invece `parseBoolean("true")`→`true` altrimenti qualunque altra stringa fa restituire `false` (non si guarda maiusc/minusc)

Wrapper
Metodi di conversione

Per convertire un valore numerico in stringa di caratteri si usa il metodo poliforme

- `String` **`toString`** `(value)`

```
Integer x = new Integer(5);
System.out.println(x.toString());
System.out.println(Integer.toString(12));
```

Ecco degli esempi per dichiarare e costruire un oggetto wrapper da un valore di tipo primitivo si può fare:

- `Integer i=new Integer(123);`
- `Double d=new Double(2.3);`
- `Boolean b= new Boolean (true);`
- `Character c= new Character('A');`

Wrapper
Boxing, unboxing e autoboxing

- **Boxing**: incapsulare un valore primitivo in un oggetto della corrispondente classe wrapper (valueOf())
- **Unboxing**: estrarre il valore di tipo primitivo da un oggetto della corrispondente classe wrapper (intValue())

Nelle più recenti versioni di Java il passaggio da primitivo a oggetto e viceversa è fatto in automatico per cui si parlerà di **autoboxing** per la conversione automatica da dato primitivo ad oggetto wrapper e di **unboxing** per la conversione automatica inversa.

Pertanto si vengono a creare oggetti wrapper senza la invocazione esplicita dell'operatore `new`

È possibile scrivere istruzioni come:

```
Integer x=123, z;
int y=x;
z=y;
```

Wrapper
Boxing, unboxing e autoboxing

Esempi:

```
//before autoboxing
   Integer iObject = Integer.valueOf(3);
   int iPrimitive = iObject.intValue()
//after java5
   Integer iObject = 3; //autobxing
   int iPrimitive = iObject; //unboxing

public static Integer show(Integer iParam){
      System.out.println("autoboxing example -
   method invocation i: " + iParam);
      return iParam;
   }
   //autoboxing and unboxing in method invocation
   show(3); //autoboxing
   int result = show(3); //unboxing because return type
   of method is Integer
```

Wrapper
Boxing, unboxing e autoboxing

Esempi:

```
ArrayList<Integer> intList  = new ArrayList<Integer>();
  intList.add(1); //autoboxing   - primitive to   object
  intList.add(2); //autoboxing
ThreadLocal<Integer>intLocal=new ThreadLocal<Integer>();
  intLocal.set(4); //autoboxing memorizza 4 nella
  ThreadLocal
  int number = intList.get(0); // unboxing
  int local = intLocal.get(); // unboxing restituisce
  cosa è memorizzato nella ThreadLocal
```

Wrapper
Boxing, unboxing e autoboxing

Possibili problemi:

1. **Autoboxing non intenzionali**

```
Integer sum = 0;
for(int i=1000; i<5000; i++){
   sum+=i;
}
```

Visto che non esiste l'operatore + per gli Integer, si farà un unboxing di `sum` e quindi il risultato sarà riconvertito in Integer è come se ci fosse:

```
int result= sum.intValue() + i;
Integer sum = new Integer(result);
```

Quindi `sum` è creato Integer per 4000 volte inutilmente!!!

Se questo capita spesso si può potenzialmente rallentare il sistema perchè viene invocato in continuazione il garbage collector

Wrapper
Boxing, unboxing e autoboxing

2. **Problemi nei confronti con l'operatore ==**

   L'operatore == funziona "bene" solo con i tipi primitivi perché per le classi wrapper è true solo se gli oggetti puntano alla stessa area di memoria

```
int i1 = 1;
int i2 = 1;
System.out.println (i1 == i2);    // true

Integer num1 = 1; // autoboxing
int num2 = 1;
System.out.println (num1 == num2);   // true unboxing

Integer obj1 = 1; // autoboxing con Integer.valueOf()
Integer obj2 = 1; // uguali per valori [-128..127]
System.out.println(obj1 == obj2);    // true

Integer one = new Integer(1); // no autoboxing
Integer anotherOne = new Integer(1);  // no autoboxing
System.out.println(one == anotherOne); // false
```

Wrapper
Boxing, unboxing e autoboxing

3. **Problemi nei confronti tra oggetti e primitivi**
   Confrontando primitivi con oggetti si può generare un'eccezione di tipo *NullPointerException*

```
Private Integer count;     //attribute
    ….

void method(){
    //NullPointerException on unboxing
    if( count <= 0)
      System.out.println("Count is not started yet");
    }
```

Se l'attributo `count` fosse stato dichiarato `int`, sarebbe stato inizailizzato a 0 e non si genererebbe l'eccezione

Wrapper
Metodo .equals()

Anche con le classi wrapper bisogna quindi usare il metodo **equals**().

Il metodo valueOf per evitare di creare molti oggetti mantiene una cache con i primi 256 numeri interi (da -128 a 127). Per questo invocando più volte il metodo valueOf passando argomenti minori di 128 viene ritornato sempre lo stesso oggetto (e quindi anche l'operatore == funziona correttamente).

Non è prudente sfruttare questa cosa, sugli oggetti wrapper è sempre bene usare il metodo equals() oppure l'operatore == dopo aver estratto il valore primitivo corrispondente:

```
i1.equals(i2);
i1.intValue() == i2.intValue();
```

Oggetti come parametri

Java assegna ai parametri formali sempre il valore del parametro attuale. Nel caso di variabili di tipo riferimento (oggetti e array), il loro valore è un indirizzo in memoria, per cui assegnando il suo valore al parametro formale questo farà riferimento alla stessa area di memoria, quindi l'oggetto referenziato può essere modificato dal metodo.

Si dice così che gli oggetti sono passati ai metodi **per riferimento**. Ma il riferimento stesso è passato per valore, cioè viene fatta una copia del valore dell'indirizzo in memoria dell'oggetto quindi se si cambia il riferimento del parametro formale, questo non cambia nel parametro attuale.

Oggetti come parametri

```
public class A_Class{
     ...
   void A_method(B_Class bb){
        bb.j=20;
         ...
        bb= new B_Class();
        bb.j=100;
       }
    }
public class B_Class{
     int j=0;
      ...
      }
    }
```

```
{
    int j=2;
    A_Class a;
    B_Class b;

    a = new A_Class();
    b = new B_Class();
    b.j = 5;
    a.A_method(b);
    j=b.j;  //j=20
}
```

Array di oggetti

Si dichiarano come i vettori di tipi predefiniti, in più si devono allocare i singoli elementi

```
final int N=5;
Classe nomeVet[]= new Classe[N];
nomeVet[0]= new Classe(val_iniz);
    //costruttore della classe
```

Array dinamici
Vector

- È un vettore di oggetti senza dimensione predefinita.
- La classe **Vector** si trova nella `java.util` che va quindi importata
- Il primo elemento ha indice 0
- Ogni elemento non ha una posizione fissa, perché quando se ne elimina, uno il vector si compatta
- Aggiungendo degli elementi aumenta lo spazio occupato (*capacity*) e il numero di elementi presenti (*size*). Quando si eliminano degli elementi diminuirà solo *size*

Array dinamici
Vector: costruttori

- Possiede 3 costruttori

```
1. Vector v= new Vector();
   //dim=10 per default e incremento di 10
      quando viene raggiunto il limite
1. Vector v= new Vector(n);
   //dim=n incremento di n
1. Vector v= new Vector(n,k);
    //dim=n, v si incrementa di k elementi
```

- Per specificare il tipo di oggetto si usa uno qualsiasi dei costuttori con la specifica del tipo

```
Vector <Integer> v =
   new Vector <Integer>();
```

- Non si può fare di tipi primitivi

Array dinamici
Vector: metodi

- `boolean` **`add(`**`Object obj`**`)`** / `void` **`addElement(`**`Object obj`**`)`**: aggiunge l'oggetto passato come elemento in fondo al vector
- **`removeElementAt (`**`int index`**`)`**: elimina l'elemento in posizione index, ricompatta il vector. Si genera un'eccezione ArrayIndexOutOfBoundsException se index è fuori dal range dei possibili valori
- **`clear()`**: svuota il vector. La size sarà 0
- **`capacity()`**: dimensione del vector
- **`size()`**: numero di oggetti realmente presenti nel vector (può essere < capacity)

Array dinamici
Vector: metodi

- `Object` **`get(`**`int index`**`)/`** **`elementAt(`**`int index`**`)`**: restituisce l'oggetto in posizione index. Va fatto un casting. Si genera un'eccezione se index è fuori dal range dei possibili valori
- `Object` **`set (`**`int index, Object obj`**`)/`** **`setElementAt (`**`Object obj, int index`**`)`**: sostituisce l'oggetto in posizione index. **`set`** restituisce l'oggetto che era prima in quella posizione. Si genera un'eccezione se index è fuori dal range dei possibili valori
- `int` **`lastIndexOf/indexOf(`**`Object obj`**`)`**: restituisce l'indice della ultima/prima occorrenza dell'elemento obj -1 se no c'è